Table of Contents

Introduction ................................................. 1

Acknowledgments ............................................ 3

1 Notational Conventions ................................. 5
   1.1 Keyboard Characters .................................. 5
   1.2 Mode Spec .............................................. 5
   1.3 Development and Runtime Systems .................. 5
   1.4 Function Prototypes .................................. 5
   1.5 ISO Compliance ....................................... 6

2 Glossary ................................................... 7

3 How to Run Prolog .................................... 21
   3.1 Getting Started ....................................... 21
   3.2 Reading in Programs ................................. 22
   3.3 Inserting Clauses at the Terminal ................. 22
   3.4 Queries and Directives ............................... 23
      3.4.1 Queries ........................................... 23
      3.4.2 Directives ....................................... 24
   3.5 Syntax Errors ......................................... 25
   3.6 Undefined Predicates ................................. 26
   3.7 Program Execution And Interruption ............... 26
   3.8 Exiting From The Top Level ......................... 27
   3.9 Nested Executions—Break ......................... 27
   3.10 Saving and Restoring Program States ............ 28
   3.11 SICStus Prolog IDE ................................. 29
   3.12 Emacs Interface .................................... 32
      3.12.1 Installation .................................... 32
         3.12.1.1 Quick-Start .................................. 32
         3.12.1.2 Customizing Emacs ....................... 33
         3.12.1.3 Enabling Emacs Support for SICStus .... 33
         3.12.1.4 Enabling Emacs Support for SICStus Documentation . 34
      3.12.2 Basic Configuration ............................. 34
      3.12.3 Usage .......................................... 34
      3.12.4 Mode Line ....................................... 37
      3.12.5 Configuration .................................. 38
      3.12.6 Tips ............................................. 41
         3.12.6.1 Font-locking .................................. 41
         3.12.6.2 Auto-fill Mode .............................. 41
         3.12.6.3 Speed ....................................... 41
         3.12.6.4 Changing Colors ............................ 42
4 The Prolog Language ........................................ 43
  4.1 Syntax ................................................................ 43
    4.1.1 Overview .................................................... 43
    4.1.2 Terms .......................................................... 43
      4.1.2.1 Overview ................................................. 43
      4.1.2.2 Integers .................................................. 43
      4.1.2.3 Floating-point Numbers ............................... 44
      4.1.2.4 Atoms .................................................... 44
      4.1.2.5 Variables ............................................... 44
      4.1.2.6 Foreign Terms ......................................... 45
    4.1.3 Compound Terms .......................................... 45
      4.1.3.1 Lists .................................................... 46
      4.1.3.2 Strings As Lists ...................................... 46
    4.1.4 Character Escaping ....................................... 47
    4.1.5 Operators and their Built-in Predicates ............. 47
      4.1.5.1 Overview ................................................ 47
      4.1.5.2 Manipulating and Inspecting Operators .......... 50
      4.1.5.3 Syntax Restrictions .................................. 50
      4.1.5.4 Built-in Operators .................................... 51
    4.1.6 Commenting ................................................. 52
      4.1.6.1 Overview ................................................ 52
      4.1.6.2 Notation ................................................ 53
      4.1.6.3 Syntax of Sentences as Terms ..................... 53
      4.1.6.4 Syntax of Terms as Tokens ......................... 55
      4.1.6.5 Syntax of Tokens as Character Strings .......... 56
      4.1.6.6 Escape Sequences .................................... 60
      4.1.6.7 Notes ................................................... 60
    4.1.7 Formal Syntax ............................................. 52
      4.1.7.1 Overview ................................................ 52
      4.1.7.2 Notation ................................................ 53
      4.1.7.3 Syntax of Sentences as Terms ..................... 53
      4.1.7.4 Syntax of Terms as Tokens ......................... 55
      4.1.7.5 Syntax of Tokens as Character Strings .......... 56
      4.1.7.6 Escape Sequences .................................... 60
      4.1.7.7 Notes ................................................... 60
    4.1.8 Summary of Predicates .................................. 61
  4.2 Semantics ..................................................... 61
    4.2.1 Programs .................................................. 61
    4.2.2 Types of Predicates Supplied with SICStus Prolog .... 63
      4.2.2.1 Hook Predicates ...................................... 63
    4.2.3 Control Structures ...................................... 63
      4.2.3.1 The Cut ............................................... 64
      4.2.3.2 Disjunction .......................................... 66
      4.2.3.3 If-Then-Else ......................................... 67
      4.2.3.4 Negation as Failure .................................. 68
      4.2.3.5 Do Loops since release 4.1 ........................ 68
      4.2.3.6 Other Control Structures ........................... 74
    4.2.4 Declarative and Procedural Semantics ............... 74
    4.2.5 Meta-Calls ................................................ 77
    4.2.6 Exceptions Related to Procedure Calls ............. 77
    4.2.7 Occurs Check ............................................. 78
    4.2.8 Summary of Control Predicates ....................... 78
  4.3 Loading Programs .......................................... 79
    4.3.1 Overview ................................................ 79
    4.3.2 The Load Predicates .................................... 80
4.12.5.1 A Note on Efficient Use of retract/1 .......................... 180
4.12.6 Accessing Clauses .................................................. 181
4.12.7 Modification of Running Code: Examples ....................... 181
  4.12.7.1 Example: assertz .............................................. 181
  4.12.7.2 Example: retract .............................................. 182
  4.12.7.3 Example: abolish .............................................. 183
4.12.8 The Internal Database ............................................. 183
4.12.9 Blackboard Primitives ............................................ 184
4.12.10 Summary of Predicates .......................................... 185
4.13 Sets and Bags: Collecting Solutions to a Goal .................... 186
  4.13.1 Introduction ..................................................... 186
  4.13.2 Collecting a Sorted List .................................... 187
    4.13.2.1 Existential Quantifier .................................. 188
  4.13.3 Collecting a Bag of Solutions .................................. 188
    4.13.3.1 Collecting All Instances ................................. 188
  4.13.4 Predicate Summary ............................................... 189
4.14 Grammar Rules ....................................................... 189
  4.14.1 Definite Clause Grammars ...................................... 189
  4.14.2 How to Use the Grammar Rule Facility .......................... 190
  4.14.3 An Example ..................................................... 191
  4.14.4 Semantics of Grammar Rules .................................... 192
  4.14.5 Summary of Predicates .......................................... 196
4.15 Errors and Exceptions ............................................... 197
  4.15.1 Overview ....................................................... 197
  4.15.2 Throwing Exceptions ........................................... 197
  4.15.3 Handling Exceptions ............................................ 198
    4.15.3.1 Protecting a Particular Goal .......................... 198
    4.15.3.2 Handling Unknown Predicates ........................... 199
  4.15.4 Error Classes .................................................. 200
    4.15.4.1 Instantiation Errors ..................................... 203
    4.15.4.2 Uninstantiation Errors .................................. 203
    4.15.4.3 Type Errors ............................................... 204
    4.15.4.4 Domain Errors ............................................. 204
    4.15.4.5 Evaluation Errors ......................................... 205
    4.15.4.6 Representation Errors .................................... 205
    4.15.4.7 Existence Errors .......................................... 206
    4.15.4.8 Permission Errors ........................................ 206
    4.15.4.9 Context Errors ............................................ 207
    4.15.4.10 Consistency Errors ...................................... 207
    4.15.4.11 Syntax Errors ............................................ 208
    4.15.4.12 Resource Errors ........................................... 208
    4.15.4.13 System Errors ............................................ 209
  4.15.5 An Example .................................................... 209
  4.15.6 Legacy Predicates ............................................... 210
  4.15.7 Interrupting Execution ....................................... 211
  4.15.8 Summary of Predicates ........................................ 211
4.16 Messages and Queries ................................................ 212
  4.16.1 Message Processing ............................................. 212
### 4.16 Phases of Message Processing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.16.1.1 Phases of Message Processing</td>
<td>213</td>
</tr>
<tr>
<td>4.16.1.2 Message Generation Phase</td>
<td>214</td>
</tr>
<tr>
<td>4.16.1.3 Message Printing Phase</td>
<td>215</td>
</tr>
<tr>
<td>4.16.2 Message Handling Predicates</td>
<td>215</td>
</tr>
<tr>
<td>4.16.3 Query Processing</td>
<td>216</td>
</tr>
<tr>
<td>4.16.3.1 Query Classes</td>
<td>216</td>
</tr>
<tr>
<td>4.16.3.2 Phases of Query Processing</td>
<td>217</td>
</tr>
<tr>
<td>4.16.3.3 Hooks in Query Processing</td>
<td>220</td>
</tr>
<tr>
<td>4.16.3.4 Default Input Methods</td>
<td>221</td>
</tr>
<tr>
<td>4.16.3.5 Default Map Methods</td>
<td>221</td>
</tr>
<tr>
<td>4.16.3.6 Default Query Classes</td>
<td>221</td>
</tr>
<tr>
<td>4.16.4 Query Handling Predicates</td>
<td>222</td>
</tr>
<tr>
<td>4.16.5 Predicate Summary</td>
<td>223</td>
</tr>
<tr>
<td>4.17 Other Topics</td>
<td>224</td>
</tr>
<tr>
<td>4.17.1 System Properties and Environment Variables</td>
<td>224</td>
</tr>
<tr>
<td>4.17.1.1 System Properties Set by SICStus Prolog</td>
<td>225</td>
</tr>
<tr>
<td>4.17.1.2 System Properties Affecting Initialization</td>
<td>226</td>
</tr>
<tr>
<td>4.17.1.3 Other System Properties</td>
<td>228</td>
</tr>
</tbody>
</table>

### 5 Debugging

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 The Procedure Box Control Flow Model</td>
<td>231</td>
</tr>
<tr>
<td>5.2 Basic Debugging Predicates</td>
<td>233</td>
</tr>
<tr>
<td>5.3 Plain Spypoints</td>
<td>235</td>
</tr>
<tr>
<td>5.4 Format of Debugging Messages</td>
<td>236</td>
</tr>
<tr>
<td>5.5 Commands Available during Debugging</td>
<td>237</td>
</tr>
<tr>
<td>5.6 Advanced Debugging — an Introduction</td>
<td>243</td>
</tr>
<tr>
<td>5.6.1 Creating Breakpoints</td>
<td>243</td>
</tr>
<tr>
<td>5.6.2 Processing Breakpoints</td>
<td>244</td>
</tr>
<tr>
<td>5.6.3 Breakpoint Tests</td>
<td>245</td>
</tr>
<tr>
<td>5.6.4 Specific and Generic Breakpoints</td>
<td>251</td>
</tr>
<tr>
<td>5.6.5 Breakpoint Actions</td>
<td>252</td>
</tr>
<tr>
<td>5.6.6 Advice points</td>
<td>258</td>
</tr>
<tr>
<td>5.6.7 Built-in Predicates for Breakpoint Handling</td>
<td>260</td>
</tr>
<tr>
<td>5.6.8 Accessing Past Debugger States</td>
<td>262</td>
</tr>
<tr>
<td>5.6.9 Storing User Information in the Backtrace</td>
<td>265</td>
</tr>
<tr>
<td>5.6.10 Hooks Related to Breakpoints</td>
<td>267</td>
</tr>
<tr>
<td>5.6.11 Programming Breakpoints</td>
<td>268</td>
</tr>
<tr>
<td>5.7 Breakpoint Handling Predicates</td>
<td>272</td>
</tr>
<tr>
<td>5.8 The Processing of Breakpoints</td>
<td>275</td>
</tr>
<tr>
<td>5.9 Breakpoint Conditions</td>
<td>277</td>
</tr>
<tr>
<td>5.9.1 Tests Related to the Current Goal</td>
<td>277</td>
</tr>
<tr>
<td>5.9.2 Tests Related to Source Information</td>
<td>278</td>
</tr>
<tr>
<td>5.9.3 Tests Related to the Current Port</td>
<td>279</td>
</tr>
<tr>
<td>5.9.4 Tests Related to the Break Level</td>
<td>280</td>
</tr>
<tr>
<td>5.9.5 Other Conditions</td>
<td>281</td>
</tr>
<tr>
<td>5.9.6 Conditions Usable in the Action Part</td>
<td>281</td>
</tr>
<tr>
<td>5.9.7 Options for Focusing on a Past State</td>
<td>282</td>
</tr>
<tr>
<td>5.9.8 Condition Macros</td>
<td>282</td>
</tr>
</tbody>
</table>
5.9.9 The Action Variables ........................................ 282
5.10 Consulting during Debugging ................................. 285
5.11 Catching Exceptions ........................................ 285
5.12 Predicate Summary .......................................... 285

6 Mixing C/C++ and Prolog ................................. 289

6.1 Notes .................................................................. 290
6.2 Calling C from Prolog .......................................... 291
6.2.1 Foreign Resources .......................................... 291
6.2.2 Conversion Declarations .................................. 292
6.2.3 Conversions between Prolog Arguments and C Types .. 293
6.2.4 Interface Predicates ........................................ 295
6.2.5 The Foreign Resource Linker ............................. 296
6.2.5.1 Customizing splfr ...................................... 296
6.2.5.2 Creating Linked Foreign Resources Manually under UNIX .. 296
6.2.6 Init and Deinit Functions ................................. 297
6.2.7 Creating the Linked Foreign Resource ............... 298
6.2.8 Foreign Code Examples .................................. 298
6.3 Calling C++ from Prolog ...................................... 301
6.4 Support Functions ............................................ 301
6.4.1 Creating and Manipulating SP_term_refs ............. 301
6.4.2 Atoms in C .................................................. 302
6.4.3 Creating Prolog Terms .................................... 303
6.4.4 Accessing Prolog Terms .................................. 304
6.4.5 Testing Prolog Terms ..................................... 304
6.4.6 Unifying and Comparing Terms ......................... 305
6.4.7 Operating System Services ............................... 305
6.4.7.1 Memory Management ................................. 305
6.4.7.2 File System ............................................ 306
6.4.7.3 Threads ................................................. 306
6.5 Calling Prolog from C ......................................... 306
6.5.1 Finding One Solution of a Call ......................... 307
6.5.2 Finding Multiple Solutions of a Call .................. 307
6.5.3 Backtracking Loops ....................................... 309
6.5.4 Calling Prolog Asynchronously ......................... 309
6.5.4.1 Signal Handling ..................................... 310
6.5.5 Exception Handling in C ................................. 310
6.5.6 Reading a goal from a string ............................ 311
6.6 SICStus Streams ............................................. 311
6.6.1 Prolog Streams ............................................ 311
6.6.2 Defining a New Stream ................................... 313
6.6.2.1 Low Level I/O Functions ............................. 317
6.6.3 Hookable Standard Streams ............................. 317
6.6.3.1 Writing User-stream Hooks .......................... 318
6.6.3.2 Writing User-stream Post-hooks ................. 318
6.7 Stand-Alone Executables ..................................... 318
6.7.1 Runtime Systems ................................................. 319
6.7.2 Runtime Systems on Target Machines ....................... 319
  6.7.2.1 Runtime Systems on UNIX Target Machines ......... 320
  6.7.2.2 Runtime Systems on Windows Target Machines .... 321
6.7.3 The Application Builder ....................................... 323
  6.7.3.1 Customizing `spld' ..................................... 323
  6.7.3.2 All-in-one Executables ................................ 324
  6.7.3.3 Setting up the C compiler on Windows .............. 327
  6.7.3.4 Extended Runtime Systems ............................ 328
  6.7.3.5 Examples .............................................. 328
6.7.4 User-defined Main Programs .................................. 330
  6.7.4.1 Initializing the Prolog Engine ....................... 330
  6.7.4.2 Loading Prolog Code ................................ 331
6.7.5 Generic Runtime Systems .................................... 331
6.8 Mixing C and Prolog Examples ................................... 331
  6.8.1 Train Example (connections) .............................. 331
  6.8.2 Building for a Target Machine ............................ 335
  6.8.3 Exceptions from C ....................................... 340
  6.8.4 Stream Example .......................................... 342
6.9 Debugging Runtime Systems ..................................... 342
  6.9.1 Locating the License Information ....................... 343
  6.9.2 Customizing the Debugged Runtime System ............ 343
  6.9.3 Examples of Debugging Runtime Systems ............... 344
7 Interfacing .NET and Java ...................................... 347
8 Multiple SICStus Runtimes in a Process .......................... 349
  8.1 Multiple SICStus Runtimes in Java .......................... 349
  8.2 Multiple SICStus Runtimes in C ............................. 349
    8.2.1 Using a Single SICStus Runtime ...................... 349
    8.2.2 Using More than One SICStus Runtime ............... 350
  8.3 Foreign Resources and Multiple SICStus Runtimes ......... 351
    8.3.1 Foreign Resources Supporting Only One SICStus Runtime ... 351
    8.3.2 Foreign Resources Supporting Multiple SICStus Runtimes ... 352
      8.3.2.1 Full Support for Multiple SICStus Runtimes .... 352
  8.4 Multiple Runtimes and Threads ................................ 354
9 Writing Efficient Programs .................................... 355
  9.1 Overview ....................................................... 355
  9.2 Execution Profiling ......................................... 355
  9.3 Coverage Analysis .......................................... 356
  9.4 The Cut ...................................................... 357
    9.4.1 Overview ............................................... 357
    9.4.2 Making Predicates Determinate ...................... 357
    9.4.3 Placement of Cuts ..................................... 359
    9.4.4 Terminating a Backtracking Loop .................... 359
  9.5 Indexing ..................................................... 360
10 The Prolog Library ........................................ 381

10.1 An Aggregation Operator for Data-Base-Style
    Queries—library(aggregate) .................................. 385
10.2 Association Lists—library(assoc) .......................... 389
10.3 Attributed Variables—library(atts) ......................... 391
10.4 AVL Trees—library(avl) ..................................... 398
10.5 Bags, or Multisets—library(bags) .......................... 401
10.6 External Storage of Terms (Berkeley DB)—library(bdb) .. 404
10.6.1 Basics ...................................................... 405
10.6.2 Current Limitations ....................................... 405
10.6.3 Berkeley DB .............................................. 405
10.6.4 The DB-Spec—Informal Description ...................... 406
10.6.5 Predicates ................................................ 406
   10.6.5.1 Conventions ........................................ 406
   10.6.5.2 The Environment .................................... 407
   10.6.5.3 Memory Leaks ...................................... 407
   10.6.5.4 The Predicates .................................... 407
10.6.6 An Example Session ...................................... 411
10.6.7 The DB-Spec ............................................ 411
10.6.8 Exporting and importing a database ...................... 412
10.11.9 Implementation Architecture ........................................ 523
10.11.9.1 Fragments and Bits ............................................. 523
10.11.9.2 Bugs .......................................................... 523
10.12 I/O on Lists of Character Codes—library(codesio) .......... 524
10.13 I/O on Comma-Separated Values (CSV) Files
and Strings—library(csv) .................................................. 525
10.14 COM Client—library(comclient) .................................. 527
10.14.1 Preliminaries .................................................... 527
10.14.2 Terminology ...................................................... 527
10.14.3 Predicate Reference ............................................. 528
10.14.4 Examples .......................................................... 530
10.15 Finite Domain Constraint Debugger—library(fdbg) ....... 532
10.15.1 Introduction ...................................................... 532
10.15.2 Concepts .......................................................... 532
10.15.2.1 Events ......................................................... 532
10.15.2.2 Labeling Levels ............................................. 532
10.15.2.3 Visualizers ..................................................... 533
10.15.2.4 Names of Terms ............................................. 533
10.15.2.5 Selectors ....................................................... 534
10.15.2.6 Name Auto-Generation ..................................... 534
10.15.2.7 Legend ........................................................ 534
10.15.2.8 The fdbg_output Stream .................................. 535
10.15.3 Basics ............................................................ 535
10.15.3.1 FDBG Options .............................................. 535
10.15.3.2 Naming Terms ................................................ 536
10.15.3.3 Built-In Visualizers ......................................... 537
10.15.3.4 New Debugger Commands ................................ 538
10.15.3.5 Annotating Programs ....................................... 539
10.15.3.6 An Example Session ......................................... 539
10.15.4 Advanced Usage ................................................ 543
10.15.4.1 Customizing Output ........................................ 543
10.15.4.2 Writing Visualizers ........................................ 544
10.15.4.3 Writing Legend Printers ................................... 545
10.15.4.4 Showing Selected Constraints (simple version) .... 546
10.15.4.5 Showing Selected Constraints (advanced version) .. 547
10.15.4.6 Debugging Global Constraints ............................. 551
10.16 Accessing Files And Directories—library(file_systems) .. 556
10.17 The Gauge Profiling Tool—library(gauge) .................... 561
10.18 Heap Operations—library(heaps) .............................. 563
10.19 Declaring determinacy attributes—library(is_directives) .. 565
10.19.1 Introduction ...................................................... 565
10.19.2 Available Determinacy Annotations ........................ 566
10.19.3 Syntax of Determinacy Declarations ......................... 566
10.19.3.1 Specifying Instantiation Patterns ....................... 567
10.19.3.2 Declaring Meta Predicate Determinacy ................. 568
10.19.4 Using Determinacy Declarations ............................... 570
10.19.5 Accessing Determinacy Declarations at Runtime ........... 570
10.20 Jasper Interface—library(jasper) ............................... 572
10.45.5.3 Prolog And Tcl Interact through Prolog Event Queue ................. 850
10.45.5.4 The Whole 8-Queens Example ........................................ 852
10.45.6 Quick Reference ............................................................. 858
  10.45.6.1 Command Format Summary .......................................... 858
  10.45.6.2 Predicates for Prolog to Interact with Tcl Interpreters ........... 860
  10.45.6.3 Predicates for Prolog to Interact with Tcl Interpreters with Tk Extensions ....................................................... 860
  10.45.6.4 Commands for Tcl Interpreters to Interact with The Prolog System ................................................................. 861
10.45.7 Resources ................................................................. 861
  10.45.7.1 Web Sites ................................................................ 862
  10.45.7.2 Manual Pages ............................................................. 862
10.46 Term Utilities—library(terms) ............................................. 862
10.47 Meta-Call with Limit on Execution Time—library(timeout) ............. 867
10.48 Dense Arrays as Binary Trees—library(trees) ............................... 868
10.49 Type Checking—library(types) ............................................... 869
10.50 Unweighted Graph Operations—library(ugraphs) ............................ 871
10.51 An Inverse of numvar/3—library(varnumbers) ............................... 874
10.52 Weighted Graph Operations—library(wgraphs) ................................ 875
10.53 Parsing and Generating XML—library(xml) ................................... 878
10.54 Zinc Interface—library(zinc) .................................................. 879
  10.54.1 Prerequisites ................................................................ 880
  10.54.2 FlatZinc ........................................................................ 880
    10.54.2.1 Exported Predicates ................................................ 881
  10.54.3 MiniZinc ........................................................................ 889
    10.54.3.1 Exported Predicates ................................................ 889
  10.54.4 Error Messages ............................................................... 895
  10.54.5 Limitations ..................................................................... 896

11 Prolog Reference Pages ......................................................... 899
11.1 Reading the Reference Pages .................................................. 899
  11.1.1 Overview ......................................................................... 899
  11.1.2 Mode Annotations ............................................................. 899
  11.1.3 Predicate Annotation ......................................................... 900
  11.1.4 Argument Types ............................................................... 901
    11.1.4.1 Simple Types ............................................................ 902
    11.1.4.2 Extended Types ........................................................ 902
  11.1.5 Exceptions ....................................................................... 903
  11.1.6 Other Fields ..................................................................... 903
11.2 Topical List of Prolog Built-Ins ............................................... 903
  11.2.1 All Solutions ................................................................. 903
  11.2.2 Arithmetic ....................................................................... 903
  11.2.3 Character I/O ................................................................. 904
  11.2.4 Control ........................................................................... 905
  11.2.5 Database ......................................................................... 906
  11.2.6 Debugging ................................................................. 908
11.3 Built-In Predicates

11.3.1 abolish/[1,2] ISO
11.3.2 abort/0
11.3.3 absolute_file_name/[2,3] hookable
11.3.4 acyclic_term/1 ISO
11.3.5 add_breakpoint/2 development
11.3.6 ,/2 ISO
11.3.7 append/3
11.3.8 arg/3 ISO
11.3.9 ask_query/4 hookable
11.3.10 assert/[1,2] ISO
11.3.11 asserta/[1,2] ISO
11.3.12 assertz/[1,2] ISO
11.3.13 at_end_of_line/[0,1]
11.3.14 at_end_of_stream/[0,1] ISO
11.3.15 atom/1 ISO
11.3.16 atom_chars/2 ISO
11.3.17 atom_codes/2 ISO
11.3.18 atom_concat/3 ISO
11.3.19 atom_length/2 ISO
11.3.20 atomic/1 ISO
11.3.21 bagof/3 ISO
11.3.22 bb_delete/2
11.3.23 bb_get/2
11.3.24 bb_put/2
11.3.25 bb_update/3
11.3.26 block/1 declaration
11.3.27 break/0 development
11.3.28 breakpoint_expansion/2 hook, development
11.3.29 byte_count/2 hook, development
11.3.30 call/[1,2,...,255] ISO
call_cleanup/2 ........................................ 969
call_residue_vars/2 .................................. 970
callable/1 ISO ......................................... 971
catch/3 ISO ............................................. 972
char_code/2 ISO ....................................... 973
char_conversion/2 ISO ................................ 974
character_count/2 .................................... 975
clause/[2,3] ISO ..................................... 976
close/[1,2] ISO ........................................ 978
compare/3 ISO ......................................... 980
copy_term/[2,3] ISO ................................... 984
current_atom/1 .......................................... 988
current_breakpoint/5 development .................. 989
current_char_conversion/2 ISO ....................... 995
current_input/1 ISO .................................... 991
current_key/2 .......................................... 992
current_module/[1,2] ................................... 993
current_op/3 ISO ...................................... 995
current_output/1 ISO ................................... 996
current_predicate/[1,2] ISO ........................... 997
current_prolog_flag/2 ISO ............................. 999
current_stream/3 ....................................... 1000
/db_reference/1 ....................................... 1001
dbg/0 development ..................................... 1003
debug/0 development .................................. 1005
discontiguous/1 declaration, ISO ................... 1008
display/1 .............................................. 1009
do/2 ...................................................... 1010
dynamic/1 declaration, ISO ............................ 1012
debugger_command_hook/2 hook, development ....... 1004
debugging/0 development ................................ 1005
dif/2 ..................................................... 1006
disable_breakpoints/1 development .................. 1007
disable/0 development .................................. 1009
do/2 ...................................................... 1010
equal/2 ISO ............................................. 1015
erase/1 ................................................... 1016
error_exception/1 hook, development ............... 1017
execution_state/[1,2] development .................. 1018
expand_term/2 hookable ................................ 1020
fail/0 ISO ............................................... 1021
false/0 ISO ............................................. 1022
11.3.79  file_search_path/2  hook  .................  1023
11.3.80  findall/[3,4]  ISO  .........................  1025
11.3.81  float/1  ISO  ............................  1028
11.3.82  flush_output/[0,1]  ISO  ..................  1029
11.3.83  foreign/[2,3]  hook  .....................  1030
11.3.84  foreign_resource/2  hook  ...............  1031
11.3.85  format/[2,3]  ................................  1032
11.3.86  freeze/2  ................................  1038
11.3.87  frozen/2  ................................  1039
11.3.88  functor/3  ISO  ............................  1040
11.3.89  garbage_collect/0  .........................  1042
11.3.90  garbage_collect_atoms/0  ..................  1043
11.3.91  generate_message/3  hook  ...............  1044
11.3.92  generate_message_hook/3  hook  ...........  1046
11.3.93  get_byte/[1,2]  ISO  .......................  1048
11.3.94  get_char/[1,2]  ISO  .......................  1049
11.3.95  get_code/[1,2]  ISO  .......................  1050
11.3.96  get.mutable/2  ............................  1051
11.3.97  goal_expansion/5  hook  ...................  1052
11.3.98  goal_source_info/3  .........................  1054
11.3.99  >/2  ISO  ..................................  1055
11.3.100  ground/1  ISO  ............................  1056
11.3.101  halt/[0,1]  ISO  ...........................  1057
11.3.102  if/3  ....................................  1058
11.3.103  ->/2  ISO  ................................  1059
11.3.104  include/1  declaration, ISO  ............  1060
11.3.105  initialization/1  declaration, ISO  ......  1061
11.3.106  instance/2  ................................  1062
11.3.107  integer/1  ISO  ...........................  1063
11.3.108  is/2  ISO  ................................  1064
11.3.109  keysort/2  ISO  ...........................  1066
11.3.110  leash/1  development  .....................  1067
11.3.111  length/2  ................................  1069
11.3.112  </2  ISO  ................................  1071
11.3.113  library_directory/1  hook  ...............  1072
11.3.114  line_count/2  .............................  1073
11.3.115  line_position/2  ..........................  1074
11.3.116  listing/[0,1]  ............................  1075
11.3.117  load_files/[1,2]  .........................  1076
11.3.118  load_foreign_resource/1  hookable  ......  1079
11.3.119  member/2  ................................  1081
11.3.120  memberchk/2  ..............................  1082
11.3.121  message_hook/3  hook  ....................  1083
11.3.122  meta_predicate/1  declaration  ..........  1084
11.3.123  mode/1  declaration  .......................  1086
11.3.124  module/[2,3]  declaration  ...............  1087
11.3.125  multfile/1  declaration, ISO  ..........  1089
11.3.126  mutable/1  ...............................  1091
11.3.223  subsumes_term/2  ISO .......................... 1213
11.3.224  tell/1 ........................................... 1215
11.3.225  telling/1 ......................................... 1216
11.3.226  ==/2  ISO ........................................ 1218
11.3.227  term_expansion/6 hook .......................... 1219
11.3.228  @>/2 ISO ......................................... 1221
11.3.229  @>/2 ISO ......................................... 1222
11.3.230  \=/2 ISO ......................................... 1223
11.3.231  @=</2 ISO ....................................... 1224
11.3.232  @=>/2 ISO ....................................... 1225
11.3.233  ?=/2 ............................................... 1226
11.3.234  term_variables/2 ISO ............................ 1227
11.3.235  throw/1 ISO ...................................... 1228
11.3.236  told/0 ............................................ 1229
11.3.237  trace/0 development ............................ 1230
11.3.238  trimcore/0 ...................................... 1231
11.3.239  true/0 ISO ....................................... 1232
11.3.240  =/2 ISO ......................................... 1233
11.3.241  unify_with_occurs_check/2 ISO .................. 1234
11.3.242  =../2 ISO ....................................... 1235
11.3.243  unknown/2 development ........................ 1237
11.3.244  unknown_predicate_handler/3 hook ............ 1238
11.3.245  unload_foreign_resource/1 hookable .......... 1239
11.3.246  update_mutable/2 ................................ 1240
11.3.247  use_module/[1,2,3] .............................. 1241
11.3.248  var/1 ISO ....................................... 1244
11.3.249  volatile/1 declaration .......................... 1245
11.3.250  when/2 ........................................... 1246
11.3.251  write/[1,2] ISO .................................. 1247
11.3.252  write_canonical/[1,2] ISO ...................... 1248
11.3.253  write_term/[2,3] hookable, ISO ................ 1249
11.3.254  writeq/[1,2] ISO .................................. 1252
11.3.255  zip/0 development .............................. 1253

12  C Reference Pages ............................................. 1255
12.1  Return Values and Errors .............................. 1255
12.2  Topical List of C Functions ............................ 1255
  12.2.1  C Errors ........................................... 1255
  12.2.2  I/O ................................................. 1255
  12.2.3  Exceptions ........................................ 1256
  12.2.4  Files and Streams ................................ 1256
  12.2.5  Foreign Interface ................................ 1256
  12.2.6  Initialization .................................... 1259
  12.2.7  Memory Management ................................ 1259
  12.2.8  Signal Handling ................................... 1260
  12.2.9  Terms in C ........................................ 1260
  12.2.10 Type Tests ......................................... 1260
12.3  API Functions ............................................. 1261
12.3.1 SP_atom_from_string() .............................. 1262
12.3.2 SP_atom_length() ................................. 1263
12.3.3 SPcalloc() ..................................... 1264
12.3.4 SP_close_query() ................................. 1265
12.3.5 SP_compare() .................................. 1266
12.3.6 SP_cons_functor() ............................... 1267
12.3.7 SP_cons_functor_array() ......................... 1268
12.3.8 SP_cons_list() ................................ 1269
12.3.9 SP_create_stream() .............................. 1270
12.3.10 SP_cut_query() ................................ 1272
12.3.11 SP_define_c_predicate() ......................... 1273
12.3.12 SP_deinitialize() ............................... 1275
12.3.13 SP_error_message() ............................. 1276
12.3.14 SP_event() ..................................... 1277
12.3.15 SP_exception_term() ............................ 1280
12.3.16 SP_expand_file_name() ......................... 1281
12.3.17 SP_fail() ....................................... 1283
12.3.18 SP_fclose() .................................... 1284
12.3.19 SP_flush_output() ................................ 1286
12.3.20 SP_fopen() ...................................... 1288
12.3.21 SP_foreign_stash() macro ....................... 1290
12.3.22 SPfprintf() macro ............................. 1291
12.3.23 SP_free() ....................................... 1292
12.3.24 SP_get_address() ................................ 1293
12.3.25 SP_get_arg() .................................... 1294
12.3.26 SP_get_atom() .................................. 1295
12.3.27 SP_get_byte() .................................. 1296
12.3.28 SP_get_code() .................................. 1297
12.3.29 SP_get_current_dir() ............................ 1298
12.3.30 SP_get_dispatch() ................................ 1299
12.3.31 SP_get_float() .................................. 1300
12.3.32 SP_get_functor() ................................ 1301
12.3.33 SP_get_integer() ................................ 1302
12.3.34 SP_get_integer_bytes() ......................... 1303
12.3.35 SP_get_list() ................................... 1305
12.3.36 SP_get_list_codes() ............................. 1306
12.3.37 SP_get_list_n_bytes() ......................... 1307
12.3.38 SP_get_list_n_codes() ......................... 1308
12.3.39 SP_get_number_codes() ......................... 1309
12.3.40 SP_get_stream_counts() ......................... 1310
12.3.41 SP_get_stream_user_data() ..................... 1312
12.3.42 SP_get_string() ................................ 1314
12.3.43 SP_getenv() ..................................... 1315
12.3.44 SP_initialize() macro .......................... 1316
12.3.45 SP_is_atom() .................................... 1318
12.3.46 SP_is_atomic() .................................. 1319
12.3.47 SP_is_compound() ............................... 1320
12.3.48 SP_is_float() .................................... 1321
<table>
<thead>
<tr>
<th>Page</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3.49</td>
<td>SP_is_integer()</td>
</tr>
<tr>
<td>12.3.50</td>
<td>SP_is_list()</td>
</tr>
<tr>
<td>12.3.51</td>
<td>SP_is_number()</td>
</tr>
<tr>
<td>12.3.52</td>
<td>SP_is_variable()</td>
</tr>
<tr>
<td>12.3.53</td>
<td>SP_load()</td>
</tr>
<tr>
<td>12.3.54</td>
<td>SP_load_sicstus_run_time()</td>
</tr>
<tr>
<td>12.3.55</td>
<td>SP_malloc()</td>
</tr>
<tr>
<td>12.3.56</td>
<td>SP_mutex_lock()</td>
</tr>
<tr>
<td>12.3.57</td>
<td>SP_mutex_unlock()</td>
</tr>
<tr>
<td>12.3.58</td>
<td>SP_new_term_ref()</td>
</tr>
<tr>
<td>12.3.59</td>
<td>SP_next_solution()</td>
</tr>
<tr>
<td>12.3.60</td>
<td>SP_next_stream()</td>
</tr>
<tr>
<td>12.3.61</td>
<td>SP_open_query()</td>
</tr>
<tr>
<td>12.3.62</td>
<td>SP_pred()</td>
</tr>
<tr>
<td>12.3.63</td>
<td>SP_predicate()</td>
</tr>
<tr>
<td>12.3.64</td>
<td>SP_printf()</td>
</tr>
<tr>
<td>12.3.65</td>
<td>SP_put_address()</td>
</tr>
<tr>
<td>12.3.66</td>
<td>SP_put_atom()</td>
</tr>
<tr>
<td>12.3.67</td>
<td>SP_put_byte()</td>
</tr>
<tr>
<td>12.3.68</td>
<td>SP_put_bytes()</td>
</tr>
<tr>
<td>12.3.69</td>
<td>SP_put_code()</td>
</tr>
<tr>
<td>12.3.70</td>
<td>SP_put_codes()</td>
</tr>
<tr>
<td>12.3.71</td>
<td>SP_put_encoded_string()</td>
</tr>
<tr>
<td>12.3.72</td>
<td>SP_put_float()</td>
</tr>
<tr>
<td>12.3.73</td>
<td>SP_put_functor()</td>
</tr>
<tr>
<td>12.3.74</td>
<td>SP_put_integer()</td>
</tr>
<tr>
<td>12.3.75</td>
<td>SP_put_integer_bytes()</td>
</tr>
<tr>
<td>12.3.76</td>
<td>SP_put_list()</td>
</tr>
<tr>
<td>12.3.77</td>
<td>SP_put_list_codes()</td>
</tr>
<tr>
<td>12.3.78</td>
<td>SP_put_list_n_bytes()</td>
</tr>
<tr>
<td>12.3.79</td>
<td>SP_put_list_n_codes()</td>
</tr>
<tr>
<td>12.3.80</td>
<td>SP_put_number_codes()</td>
</tr>
<tr>
<td>12.3.81</td>
<td>SP_put_string()</td>
</tr>
<tr>
<td>12.3.82</td>
<td>SP_put_term()</td>
</tr>
<tr>
<td>12.3.83</td>
<td>SP_put_variable()</td>
</tr>
<tr>
<td>12.3.84</td>
<td>SP_query()</td>
</tr>
<tr>
<td>12.3.85</td>
<td>SP_query_cut_fail()</td>
</tr>
<tr>
<td>12.3.86</td>
<td>SP_raise_exception()</td>
</tr>
<tr>
<td>12.3.87</td>
<td>SP_read_from_string()</td>
</tr>
<tr>
<td>12.3.88</td>
<td>SP_realloc()</td>
</tr>
<tr>
<td>12.3.89</td>
<td>SP_register_atom()</td>
</tr>
<tr>
<td>12.3.90</td>
<td>SP_restore()</td>
</tr>
<tr>
<td>12.3.91</td>
<td>SP_set_argv()</td>
</tr>
<tr>
<td>12.3.92</td>
<td>SP_set_current_dir()</td>
</tr>
<tr>
<td>12.3.93</td>
<td>SP_set_user_stream_hook()</td>
</tr>
<tr>
<td>12.3.94</td>
<td>SP_set_user_stream_post_hook()</td>
</tr>
<tr>
<td>12.3.95</td>
<td>SP_signal()</td>
</tr>
<tr>
<td>12.3.96</td>
<td>SP_strdup()</td>
</tr>
</tbody>
</table>
12.3.97  SP_string_from_atom() .................................................. 1373
12.3.98  SP_term_type() ............................................................. 1374
12.3.99  SP_unget_byte() ........................................................... 1375
12.3.100 SP_unget_code() ............................................................ 1376
12.3.101 SP_unify() ................................................................. 1377
12.3.102 SP_unregister_atom() .................................................... 1378
12.3.103 SU_initialize() hook .................................................... 1379
12.3.104 user_close() ................................................................. 1380
12.3.105 user_flush_output() ....................................................... 1382
12.3.106 user_read() ................................................................. 1384
12.3.107 user_write() ............................................................... 1386

13  Command Reference Pages .................. 1389

13.1  sicstus — SICStus Prolog Development System ................. 1390
13.2  mzn-sicstus — Shortcut for MiniZinc with SICStus back-end .. 1392
13.3  spfz — FlatZinc Interpreter ............................................. 1393
13.4  spdet — Determinacy Checker .......................................... 1394
13.5  spld — SICStus Prolog Application Builder ......................... 1395
13.6  splfr — SICStus Prolog Foreign Resource Linker ................. 1402
13.7  splm — SICStus Prolog License Manager ............................ 1406
13.8  spxref — Cross Referencer ............................................. 1407

References ................................................................. 1409

Predicate Index ......................................................... 1413

Keystroke Index ....................................................... 1427

Book Index .............................................................. 1429
Introduction

Prolog is a simple but powerful programming language developed at the University of Mar-
seille [Roussel 75], as a practical tool for programming in logic [Kowalski 74]. From a user’s
point of view the major attraction of the language is ease of programming. Clear, readable,
concise programs can be written quickly with few errors.

For an introduction to programming in Prolog, readers are recommended to consult [Sterling
& Shapiro 86]. However, for the benefit of those who do not have access to a copy of this
book, and for those who have some prior knowledge of logic programming, we include a
summary of the language. For a more general introduction to the field of Logic Programming
see [Kowalski 79]. See Chapter 4 [Prolog Intro], page 43.

This manual describes a Prolog system developed at the Swedish Institute of Computer
Science. Parts of the system were developed by the project “Industrialization of SICStus
Prolog” in collaboration with Ericsson Telecom AB, NobelTech Systems AB, Infologics AB
and Televerket. The system consists of a \textit{WAM} emulator written in C, a library and runtime
system written in C and Prolog and an interpreter and a compiler written in Prolog. The
Prolog engine is a Warren Abstract Machine (WAM) emulator [Warren 83]. Two modes
of compilation are available: in-core i.e. incremental, and file-to-file. When compiled, a
predicate will run about 8 times faster and use memory more economically. Implementation
details can be found in [Carlsson 90] and in several technical reports available from SICS.

SICStus Prolog follows the mainstream Prolog tradition in terms of syntax and built-in
predicates. As of release 4, SICStus Prolog is fully compliant with the International Stan-
dard ISO/IEC 13211-1 (PROLOG: Part 1—General Core) and the subsequent Technical
Corrigenda 1, 2 and 3.
Acknowledgments

The following people have contributed to the development of SICStus Prolog:


The Industrialization of SICStus Prolog (1988-1991) was funded by

Ericsson Telecom AB, NobelTech Systems AB, Infologics AB, and Televerket, under the National Swedish Information Technology Program IT4.

The development of release 3 (1991-1995) was funded in part by

Ellemtel Utvecklings AB

This manual is based in part on DECsystem-10 Prolog User’s Manual by


See Section 10.8 [lib-chr], page 413, for acknowledgments relevant to the CHR constraint solver.

See Section 10.11 [lib-clpqr], page 499, for acknowledgments relevant to the clp(Q,R) constraint solver.

UNIX is a registered trademark of The Open Group. Windows is a registered trademark of Microsoft Corp.
1 Notational Conventions

1.1 Keyboard Characters

When referring to keyboard characters, printing characters are written thus: a, while control characters are written like this: \^A. Thus \^C is the character you get by holding down the CTL key while you type c. Finally, the special control characters carriage-return, line-feed and space are often abbreviated to RET, LFD and SPC respectively.

Throughout, we will assume that \^D is the EOF character (it is usually \^Z under Windows) and that \^C is the interrupt character. In most contexts, the term end_of_file terminated by a full stop (.) can be typed instead of the EOF character.

1.2 Mode Spec

When describing a predicate, we present its usage with a mode spec, which has the form name(arg, . . ., arg), where each arg denotes how that argument is used by the predicate, and has one of the following forms:

:ArgName The argument is used as a term denoting a goal or a clause or a predicate name, or that otherwise needs special handling of module prefixes. It is is subject to module name expansion (see Section 4.11.15 [ref-mod-mne], page 171).

+ArgName The argument is an input argument. Usually, but not always, this implies that the argument should be instantiated.

-ArgName The argument is an output argument. Usually, but not always, this implies that the argument should be uninstantiated.

?ArgName The argument may be used for both input and output.

Please note: The reference pages for built-in predicate use slightly different mode specs.

1.3 Development and Runtime Systems

The full Prolog system with top level, compiler, debugger etc. is known as the development system.

It is possible to link user-written C code with a subset of SICStus Prolog to create runtime systems. When introducing a built-in predicate, any limitations on its use in runtime systems will be mentioned.

1.4 Function Prototypes

Whenever this manual documents a C function as part of SICStus Prolog’s foreign language interface, the function prototype will be displayed in ANSI C syntax.
1.5 ISO Compliance

SICStus Prolog is fully compliant with the International Standard ISO/IEC 13211-1 (PROLOG: Part 1—General Core) as augmented by Technical Corrigenda 1, 2 and 3.

To aid programmers who wish to write standard compliant programs, built-in predicates and arithmetic functors that are part of the ISO Prolog Standard are annotated with $/ISO/$ in this manual.
2 Glossary

abolish  To abolish a predicate is to retract all the predicate’s clauses and to remove all information about it from the Prolog system, to make it as if that predicate had never existed.

advice point  A special case of breakpoint, the advice breakpoint. It is distinguished from spypoints in that it is intended for non-interactive debugging, such as checking of program invariants, collecting information, profiling, etc.

alphanumeric  An alphanumeric character is any of the lowercase characters from ‘a’ to ‘z’, the uppercase characters from ‘A’ to ‘Z’, the numerals from ‘0’ to ‘9’, or underscore (‘_’).

ancestors  An ancestor of a goal is any goal that the system is trying to solve when it calls that goal. The most distant ancestor is the goal that was typed at the top-level prompt.

anonymous variable  An anonymous variable is one that has no unique name, and whose value is therefore inaccessible. An anonymous variable is denoted by an underscore (‘_’).

argument  See predicate, structure, and arity.

arity  The arity of a structure is its number of arguments. For example, the structure `customer(jones,85)` has an arity of 2.

atom  A character sequence used to uniquely denote some entity in the problem domain. A number is not an atom. Examples of legal atoms are: `hello *= #\$%' 'New York' 'don't'`. See Section 4.1.2.4 [ref-syn-trm-ato], page 44. Atoms are recognized by the built-in predicate `atom/1`. Each Prolog atom is represented internally by a unique integer, represented in C as an `SP_atom`.

atomic term  Synonym for constant.

backtrace  A collection of information on the control flow of the program, gathered by the debugger. Also the display of this information produced by the debugger. The backtrace includes data on goals that were called but not exited and also on goals that exited nondeterminately.

backtracking  The process of reviewing the goals that have been satisfied and attempting to resatisfy these goals by finding alternative solutions.

binding  The process of assigning a value to a variable; used in unification.

blocked goal  A goal that is suspended because it is not instantiated enough.
**body**

The body of a clause consists of the part of a Prolog clause following the ‘:-’ symbol.

**breakpoint**

A description of certain invocations in the program where the user wants the debugger to stop, or to perform some other actions. A breakpoint is specific if it applies to the calls of a specific predicate, possibly under some conditions; otherwise, it is generic. Depending on the intended usage, breakpoints can be classified as debugger breakpoints, also known as spypoints, or advice breakpoints, also called advice points; see Section 5.6 [Advanced Debugging], page 243.

**breakpoint specification**

A term describing a breakpoint. Composed of a test part, specifying the conditions under which the breakpoint should be applied, and an action part, specifying the effects of the breakpoint on the execution.

**byte list**

A byte list is a list of bytes, i.e. integers in \([0, \ldots, 255]\).

**buffer**

A temporary workspace in Emacs that contains a file being edited.

**built-in predicate**

A predicate that comes with the system and that does not have to be explicitly loaded before it is used.

**callable term**

A callable term is either a compound term or an atom. Callable terms are recognized by the built-in predicate `callable/1`.

**char list**

A char list is a list of chars, i.e. atoms made up of a single character.

**character code**

An integer that is the numeric representation of a character in the character code set.

**character code set**

A subset of the set \([0, \ldots, 2^{31}-1]\) that can be handled in input/output. SICStus Prolog fixes the character code set to a superset of Unicode, which includes the ASCII code set, i.e. codes 0..127, and these codes are interpreted as ASCII characters.

**character-conversion mapping**

SICStus Prolog maintains a character-conversion mapping, which is used while reading terms and programs. Initially, the mapping prescribes no character conversions. It can be modified by the built-in predicate `char_conversion(In, Out)`, following which `In` will be converted to `Out`. Character conversion can be switched off by the `char_conversion` Prolog flag. Please note: the mapping is *global*, as opposed to being local to the current module, Prolog text, or otherwise.

**character-type mapping**

A function mapping each element of the character code set to one of the character categories (whitespace, letter, symbol-char, etc.), required for parsing tokens.
choicepoints
A memory block representing outstanding choices for some goals or disjunctions.

clause
A fact or a rule. A rule comprises a head and a body. A fact consists of a head only, and is equivalent to a rule with the body true.

code list
A code list is a list of character codes.

conditional compilation
Conditionally including or excluding parts of a file at compile time.

compactcode
Virtual code representation of compiled code. A reasonable compromise between performance and space requirement. A valid value for the compiling Prolog flag.

compile
To load a program (or a portion thereof) into Prolog through the compiler. Compiled code runs more quickly than interpreted code, and provides better precision for execution profiling and coverage analysis. On the other hand, you cannot debug compiled code in as much detail as interpreted code.

compound term
A compound term is a term that is an atom together with one or more arguments. For example, in the term father(X), father is the name, and X is the first and only argument. The argument to a compound term can be another compound term, as in father(father(X)). Compound terms are recognized by the built-in predicate compound/1.

conjunction
A series of goals connected by the connective “and” (that is, a series of goals whose principal operator is ‘,’).

console-based executable
An executable that inherits the standard streams from the process that invoked it, e.g. a UNIX shell or a DOS-prompt.

constant
An integer (for example: 1, 20, -10), a floating-point number (for example: 12.35), or an atom. Constants are recognized by the built-in predicate atomic/1.

consult
To load a program (or a portion thereof) into Prolog through the interpreter. Interpreted code runs more slowly than compiled code, and does not provide as good precision for execution profiling and coverage analysis. On the other hand, you can debug interpreted code in more detail than compiled code.

control structure
A built-in predicate that is “part of the language” in the sense that it is treated specially in certain language features. The set of such control structures and language features is enumerated in Section 4.2.3 [ref-sem-ctr], page 63.

creep
What the debugger does in trace mode, also known as single-stepping. It goes to the next port of a procedure box and prints the goal, then prompts you for input. See Section 5.2 [Basic Debug], page 233.
**cursor**  The point on the screen at which typed characters appear. This is usually highlighted by a line or rectangle the size of one space, which may or may not blink.

**cut**  Written as !. A built-in predicate that succeeds when encountered; if backtracking should later return to the cut, then the goal that matched the head of the clause containing the cut fails immediately.

**database**  The Prolog database comprises all of the clauses that have been loaded or asserted into the Prolog system or that have been asserted, except those clauses that have been retracted or abolished.

**db reference**  A compound term denoting a unique reference to a dynamic clause. Recognized by the built-in predicate `db_reference/1`.

**debug**  A mode of program execution in which the debugger stops to print the current goal only at predicates that have spypoints set on them (see leap).

**debugcode**  Interpreted representation of compiled code. A valid value for the `compiling` Prolog flag.

**declaration**  A declaration looks like a directive, but is not executed but rather conveys information about predicates about to be loaded.

**deinit function**  A function in a foreign resource that is called prior to unloading the resource.

**determinate**  A predicate is determinate if it can supply only one answer.

**development system**  A stand-alone executable with the full programming environment, including top level, compiler, debugger etc. The default sicstus executable is a development system; new development systems containing prelinked foreign resources can also be created.

**directive**  A directive is a goal preceded by the prefix operator ‘:-’, whose intuitive meaning is “execute this as a query, but do not print out any variable bindings.”

**disjunction**  A series of goals connected by the connective “or” (that is, a series of goals whose principal operator is ‘;’).

**do loop**  A control structure of the form *(Iterators do Body)*. It expresses a simple iteration. See Section 4.2.3.5 [ref-sem-ctr-dol], page 68.

**dynamic predicate**  A predicate that can be modified while a program is running. The semantics of such updates is described in Section 4.12.1 [ref-mdb-bas], page 176. A predicate must explicitly be declared to be dynamic or it must be added to the database via one of the assertion predicates.
**encoded string**
A sequence of bytes representing a sequence of possibly wide character codes, using the UTF-8 encoding.

**escape sequence**
A sequence of characters beginning with ‘\’ inside certain syntactic tokens (see Section 4.1.7.6 [ref-syn-syn-esc], page 60).

**export**
A module exports a predicate so that other modules can import it.

**extended runtime system**
A stand-alone executable. In addition to the normal set of built-in runtime system predicates, extended runtime systems include the compiler. Extended runtime systems require the extended runtime library, available from SICS as an add-on product.

**fact**
A clause with no conditions—that is, with an empty body. A fact is a statement that a relationship exists between its arguments. Some examples, with possible interpretations, are:

- `king(louis, france). % Louis was king of France.
- `have_beaks(birds). % Birds have beaks.
- `employee(nancy, data_processing, 55000). % Nancy is an employee in the data processing department.

**file specification**
An atom or a compound term denoting the name of a file. The rules for mapping such terms to absolute file names are described in Section 4.5 [ref-fdi], page 95.

**floundered query**
A query where all unsolved goals are blocked.

**foreign predicate**
A predicate that is defined in a language other than Prolog, and explicitly bound to Prolog predicates by the Foreign Language Interface.

**foreign resource**
A named set of foreign predicates.

**functor**
The functor of a compound term is its name and arity. For example, the compound term `foo(a,b)` is said to have “the functor `foo` of arity two”, which is generally written `foo/2`.

The functor of a constant is the term itself paired with zero. For example, the constant `nl` is said to have “the functor `nl` of arity zero”, which is generally written `nl/0`.

**garbage collection**
The freeing up of space for computation by making the space occupied by terms that are no longer available for use by the Prolog system.

**generalized predicate specification**
A generalized predicate specification corresponds to the argument type `pred_spec_tree` (see Section 11.1.4.2 [mpg-ref-aty-ety], page 902) and is a term
of one of the following forms. It is always interpreted wrt. a given module context:

- **Name**
  all predicates called *Name* no matter what arity, where *Name* is an atom for a specific name or a variable for all names, or

- **Name/Arity**
  the predicate of that name and arity, or

- **Module:Spec**
  specifying a particular module *Module* instead of the default module, where *Module* is an atom for a specific module or a variable for all modules, or

- **[Spec, . . . , Spec]**
  the set of all predicates covered by the Specs.

**glue code**
Interface code between the Prolog engine and foreign predicates. Automatically generated by the foreign language interface as part of building a linked foreign resource.

**goal**
A simple goal is a predicate call. When called, it will either succeed or fail.
A compound goal is a formula consisting of simple goals connected by connectives such as “and” (’,’) or “or” (’;’).
A goal typed at the top level is called a query.

**ground**
A term is ground when it is free of (unbound) variables. Ground terms are recognized by the built-in predicate *ground/1*.

**guarded clause**
A clause of the form

\[
\text{Head} :- \text{Goals}, !, \text{Goals}.
\]

**head**
The head of a clause is the single goal, which will be satisfied if the conditions in the body (if any) are true; the part of a rule before the ’:-’ symbol. The head of a list is the first element of the list.

**hook predicate**
A hook predicate is a predicate that somehow alters or customizes the behavior of a hookable predicate. Typically, it is undefined initially, belongs to the *user* module, and if defined by the user, it is best defined as multifile, so that new clauses can be added by different software modules. **Please note:** any exception thrown by a hook predicate is caught locally. Then it is either printed as an error message, or merely ignored.

**hookable predicate**
A hookable predicate is a built-in predicate whose behavior is somehow altered or customized by a hook predicate.

**interactive stream**
A stream with the *interactive* stream property. Certain behavior of interactive streams are optimized for the case where a human is at the other end of the stream.
import

Exported predicates in a module can be imported by other modules. Once a predicate has been imported by a module, it can be called, or exported, as if it were defined in that module.

There are two kinds of importation: predicate importation, in which only specified predicates are imported from a module; and module importation, in which all the predicates exported by a module are imported.

indexing

The process of filtering a set of potentially matching clauses of a predicate given a goal.

For both interpreted and compiled code, indexing is done on the principal functor of the first argument. Additionally, for interpreted code only, principal functor filtering is done on each argument, but the filtering done for the first argument is more efficient.

init function

A function in a foreign resource that is called upon loading the resource.

initialization

An initialization is a goal that is executed when the file in which the initialization is declared is loaded. An initialization is declared as a directive :- initialization Goal. They are executed in input order.

instantiation

A variable is instantiated if it is bound to a non-variable term; that is, to an atomic term or a compound term.

interpret

Load a program or set of clauses into Prolog through the interpreter (also known as consulting). Interpreted code runs more slowly than compiled code, does not provide as good precision for execution profiling and coverage analysis. On the other hand, more extensive facilities are available for debugging interpreted code.

invocation box

Same as procedure box.

iterator

A compound term expressing how a do loop should be iterated. See Section 4.2.3.5 [ref-sem-ctr-dol], page 68.

large integer

An integer that is not a small integer.

layout term

In the context of handling line number information for source code, a source term Source gets associated to a layout term Layout, which is one of the following:

- [], if no line number information is available for Source.
- If Source is a simple term, then Layout is the number of the line where Source occurs.
- If Source is a compound term, then Layout is a list whose head is the number of the line where the first token of Source occurs, and whose remaining elements are the layouts of the arguments of Source.
leap  What the debugger does in debug mode. The debugger shows only the ports of predicates that have spypoints on them. It then normally prompts you for input, at which time you may leap again to the next spypoint (see trace).

leashing  Determines how frequently the debugger will stop and prompt you for input when you are tracing. A port at which the debugger stops is called a “leashed port”.

linked foreign resource  A foreign resource that is ready to be installed in an atomic operation, normally represented as a shared object or DLL.

list  A partial list is either a variable, or a compound term whose principal functor is the list constructor (’./2) and whose second argument is a partial list. Often it is implied that the partial list is not a variable.

A proper list is either the empty list, i.e. the atom [], or a compound term whose principal functor is the list constructor (’./2) and whose second argument is a proper list.

A partial list or a proper list that is a compound term is said to be non-empty. In many cases list is used to denote both the case of a proper list and the case of a, most often non-variable, partial list.

A cyclic list is a compound term whose principal functor is the list constructor (’./2) and whose second argument is a cyclic list, e.g. what could be constructed using L=’.(a,L), or L=[a,b|L]. Passing a cyclic list as an argument to a predicate that expects a partial or proper list should be avoided as not all predicates are prepared to handle such input.

A list is often written using list syntax, e.g. using [a,b] to denote the (proper) list ’.(a,’.(b,[])), or using [a,b|End] to denote the (partial) list ’.(a,’.(b,End)).

load  To load a Prolog clause or set of clauses, in source or binary form, from a file or set of files.

meta-call  The process of interpreting a callable term as a goal. This is done e.g. by the built-in predicate call/1.

meta-logical predicate  A predicate that performs operations that require reasoning about the current instantiation of terms or decomposing terms into their constituents. Such operations cannot be expressed using predicate definitions with a finite number of clauses.

meta-predicate  A meta-predicate is one that calls one or more of its arguments; more generally, any predicate that needs to assume some module in order to operate is called a meta-predicate. Some arguments of a meta-predicate are subject to module name expansion.

module  Every predicate belongs to a module. The name of a module is an atom. Some predicates in a module are exported. The default module is user.
**module name expansion**

The process by which certain arguments of meta-predicates get prefixed by the source module. See Section 4.11.15 [ref-mod-mne], page 171.

**module file**

A module file is a file that is headed with a module declaration of the form:

```prolog
:- module(ModuleName, ExportedPredList).
```

which must appear as the first term in the file. When a module file or its corresponding object file is loaded, all predicates defined in the module are removed, and all predicate imported into the module are unimported.

**multifile predicate**

A predicate whose definition is to be spread over more than one file. Such a predicate must be preceded by an explicit multifile declaration in all files containing clauses for it.

**mutable term**

A special form of compound term subject to destructive assignment. See Section 4.8.9 [ref-lte-mut], page 131. Mutable terms are recognized by the built-in predicate `mutable/1`.

**name clash**

A name clash occurs when a module attempts to define or import a predicate that it has already defined or imported.

**nondeterminate**

A predicate is determinate if it can supply more than one answer.

**occurs check**

A test to ensure that binding a variable does not bind it to a term where that variable occurs.

**one-char atom**

An atom that consists of a single character.

**operator**

A notational convenience that allows you to express any compound term in a different format. For example, if `likes` in

```prolog
| ?- likes(sue, cider).
```

is declared an infix operator, then the query above could be written:

```prolog
| ?- sue likes cider.
```

An operator does not have to be associated with a predicate. However, certain built-in predicates are declared as operators. For example,

```prolog
| ?- =..(X, Y).
```

can be written as

```prolog
| ?- X =.. Y.
```

because `=..` has been declared an infix operator.

Those predicates that correspond to built-in operators are written using infix notation in the list of built-in predicates at the beginning of the part that contains the reference pages.
Some built-in operators do **not** correspond to built-in predicates; for example, arithmetic operators. See Section 4.1.5.4 [ref-syn-ops-bop], page 51, for a list of built-in operators.

### pair
A compound term $K-V$. Pairs are used by the built-in predicate `keysort/2` and by many library modules.

### parent
The parent of the current goal is a goal that, in its attempt to obtain a successful solution to itself, is calling the current goal.

### port
One of the seven key points of interest in the execution of a Prolog predicate. See Section 5.1 [Procedure Box], page 231, for a definition.

### prelinked foreign resource
A linked foreign resource that is linked into a stand-alone executable as part of building the executable.

### precedence
A number associated with each Prolog operator, which is used to disambiguate the structure of the term represented by an expression containing a number of operators. Operators of lower precedence are applied before those of higher precedence; the operator with the highest precedence is considered the principal functor of the expression. To disambiguate operators of the same precedence, the associativity type is also necessary. See Section 4.1.5 [ref-syn-ops], page 47.

### predicate
A functor that specifies some relationship existing in the problem domain. For example, `< /2` is a built-in predicate specifying the relationship of one number being less than another. In contrast, the functor `+ /2` is not (normally used as) a predicate.

A predicate is either built-in or is implemented by a procedure.

### predicate specification
A compound term `name/arity` or `module:name/arity` denoting a predicate.

### procedure
A set of clauses in which the head of each clause has the same predicate. For instance, a group of clauses of the following form:

```
connects(san_francisco, oakland, bart_train).
connects(san_francisco, fremont, bart_train).
connects(concord, daly_city, bart_train).
```

is identified as belonging to the predicate `connects/3`.

### procedure box
A way of visualizing the execution of a Prolog procedure. A procedure box is entered and exited via ports.

### profiling
The process of gathering execution statistics of the program, essentially counting the number of times selected program points have been reached.

### program
A set of procedures designed to perform a given task.

### PO file
A PO (Prolog object) file contains a binary representation of a set of modules, predicates, clauses and directives. They are portable between different platforms, except between 32-bit and 64-bit platforms. They are created by `save_files/2`, `save_modules/2`, and `save_predicates/2`. 
query
A query is a question put by the user to the Prolog system. A query is written as a goal followed by a full stop in response to the top-level prompt. For example,

```
| ?- father(edward, ralph).
```

refers to the predicate father/2. If a query has no variables in it, then the system will respond either 'yes' or 'no'. If a query contains variables, then the system will try to find values of those variables for which the query is true. For example,

```
| ?- father(edward, X).
    X = ralph
```

After the system has found one answer, the user can direct the system to look for additional answers to the query by typing ;.

recursion
The process in which a running predicate calls itself, presumably with different arguments and for the purpose of solving some subset of the original problem.

region
The text between the cursor and a previously set mark in an Emacs buffer.

rule
A clause with one or more conditions. For a rule to be true, all of its conditions must also be true. For example,

```
has_stiff_neck(ralph) :-
    hacker(ralph).
```

This rule states that if the individual ralph is a hacker, then he must also have a stiff neck. The constant ralph is replaced in

```
has_stiff_neck(X) :-
    hacker(X).
```

by the variable X. X unifies with anything, so this rule can be used to prove that any hacker has a stiff neck.

runtime kernel
A shared object or DLL containing the SICStus virtual machine and other runtime support for stand-alone executables.

runtime system
A stand-alone executable with a restricted set of built-in predicates and no top level. Stand-alone applications containing debugged Prolog code and destined for end-users are typically packaged as runtime systems.

saved state
A snapshot of the state of Prolog saved in a file by save_program/[1,2].

semantics
The relation between the set of Prolog symbols and their combinations (as Prolog terms and clauses), and their meanings. Compare syntax.

sentence
A clause or directive.

side effect
A predicate that produces a side effect is one that has any effect on the “outside world” (the user’s terminal, a file, etc.), or that changes the Prolog database.

simple term
A simple term is a constant or a variable. Simple terms are recognized by the built-in predicate simple/1.
skeletal goal

A compound term name(arg, ..., arg) or module:name(arg, ..., arg) denoting a predicate.

small integer

An integer in the range \([-2^{28}, 2^{28}-1]\) on 32-bit platforms, or \([-2^{60}, 2^{60}-1]\) on 64-bit platforms. The start and end of this range is available as the value of the Prolog flags \texttt{min\_tagged\_integer} and \texttt{max\_tagged\_integer}, respectively.

source code

The human-readable, as opposed to the machine-executable, representation of a program.

source module

The module that is the context of a file being loaded. For module files, the source module is named in the file's module declaration. For other files, the source module is inherited from the context.

\texttt{SP\_atom}

\textit{since release 4.3}

A C type for the internal representation of Prolog atoms. Used in SICStus API functions.

\texttt{SP\_integer}

\textit{since release 4.3}

A C type denoting an integer that is large enough to hold a pointer. Used in SICStus API functions.

\texttt{SP\_term\_ref}

A C type denoting a “handle” object providing an interface from C to Prolog terms. Used in SICStus API functions.

ssypoint

A special case of breakpoint, the debugger breakpoint, intended for interactive debugging. Its simplest form, the plain ssypoint instructs the debugger to stop at all ports of all invocations of a specified predicate. Conditional ssypoints apply to a single predicate, but are more selective: the user can supply applicability tests and prescribe the actions to be carried out by the debugger. A generic ssypoint is like a conditional ssypoint, but not restricted to a single predicate. See Section 5.6 [Advanced Debugging], page 243.

stand-alone executable

A binary program that can be invoked from the operating system, containing the SICStus runtime kernel. A stand-alone executable is a development system (e.g. the default sicstus executable), or a runtime system. Both kinds are created by the application builder. A stand-alone executable does not itself contain any Prolog code; all Prolog code must be loaded upon startup.

static predicate

A predicate that can be modified only by being reloaded or by being abolished. See dynamic predicate.

steadfast

A predicate is steadfast if it refuses to give the wrong answer even when the query has an unexpected form, typically with values supplied for arguments intended as output.
stream  An input/output channel. See Section 4.6 [ref-iou], page 102.

stream alias  
A name assigned to a stream at the time of opening, which can be referred to in I/O predicates. Must be an atom. There are also three predefined aliases for the standard streams: user_input, user_output and user_error. Although not a stream alias proper, the atom user also stands for the standard input or output stream, depending on context.

stream object  
A term denoting an open Prolog stream. See Section 4.6 [ref-iou], page 102.

stream position  
A term representing the current position of a stream. This position is determined by the current byte, character and line counts and line position. Standard term comparison on stream position terms works as expected. When SP1 and SP2 refer to positions in the same stream, SP1@<SP2 if and only if SP1 is before SP2 in the stream. You should not otherwise rely on their internal representation.

stream property  
A term representing the property of an open Prolog stream. The possible forms of this term are defined in Section 4.6.7.8 [ref-iou-sfh-bos], page 113.

string  
A special syntactic notation, which, by default, denotes a code list, e.g.: "SICStus"

By setting the Prolog flag double_quotes, the meaning of strings can be changed. With an appropriate setting, a string can be made to denote a char list, or an atom. Strings are not a separate data type.

subterm selector  
A list of argument positions selecting a subterm within a term (i.e. the subterm can be reached from the term by successively selecting the argument positions listed in the selector). Example: within the term q, (r, s; t) the subterm s is selected by the selector [2, 1, 2].

syntax  
The part of Prolog grammar dealing with the way in which symbols are put together to form legal Prolog terms. Compare semantics.

system property  
SICStus Prolog stores some information in named variables called system properties. System properties are used as of release 4.1, where previous releases used environment variables.

The default value for a system property is taken from the corresponding environment variable. Any exceptions to this rule is explicitly mentioned in the documentation. See Section 4.17.1 [System Properties and Environment Variables], page 224, for more information.

term  
A basic data object in Prolog. A term can be a constant, a variable, or a compound term.

trace  
A mode of program execution in which the debugger creeps to the next port and prints the goal.
type-in module
The module that is the context of queries.

unblocked goal
A goal that is not blocked any more.

unbound
A variable is unbound if it has not yet been instantiated.

unification
The process of matching a goal with the head of a clause during the evaluation of a query, or of matching arbitrary terms with one another during program execution.

The rules governing the unification of terms are:
- Two constants unify with one another if they are identical.
- A variable unifies with a constant or a compound term. As a result of the unification, the variable is instantiated to the constant or compound term.
- A variable unifies with another variable. As a result of the unification, they become the same variable.
- A compound term unifies with another compound term if they have the same functor and if all of the arguments can be unified.

unit clause
See fact.

variable
A logical variable is a name that stands for objects that may or may not be determined at a specific point in a Prolog program. When the object for which the variable stands is determined in the Prolog program, the variable becomes instantiated. A logical variable may be unified with a constant, a compound term, or another variable. Variables become uninstantiated when the predicate they occur in backtracks past the point at which they were instantiated.

Variables may be written as any sequence of alphanumeric characters starting with either a capital letter or ‘_’; e.g.:

X Y Z Name Position _c _305 One_stop

See Section 4.1.2.5 [ref-syn-trm-var], page 44.

volatile
Predicate property. The clauses of a volatile predicate are not saved in saved states.

windowed executable
An executable that pops up its own window when run, and that directs the standard streams to that window.

zip
Same as debug mode, except no debugging information is collected while zipping.


3 How to Run Prolog

SICStus Prolog offers the user an interactive programming environment with tools for incrementally building programs, debugging programs by following their executions, and modifying parts of programs without having to start again from scratch.

The text of a Prolog program is normally created in a file or a number of files using one of the standard text editors. The Prolog interpreter can then be instructed to read in programs from these files; this is called consulting the file. Alternatively, the Prolog compiler can be used for compiling the file.

3.1 Getting Started

Under UNIX, SICStus Prolog can be started from one of the shells. On other platforms, it is normally started by clicking on an icon. However, it is often convenient to run SICStus Prolog under GNU Emacs or the SPIDER IDE (see Section 3.11 [SPIDER], page 29), instead. A GNU Emacs interface for SICStus Prolog is described later (see Section 3.12 [Emacs Interface], page 32). From a UNIX or Windows shell, SICStus Prolog can be started by invoking the sicstus command-line tool, using the full path to sicstus unless its location has been added to the shell’s path.

Under UNIX, a saved state file can be executed directly by typing:

% file argument...

This is equivalent to:

% sicstus -r file [-- argument...]

Please note: saved states do not store the complete path of the binary sicstus. Instead, they call the main executable using the version specific name sicstus-4.7.0, which is assumed to be found in the shell’s path. If there are several versions of SICStus installed, then it is up to the user to make sure that the correct start-script is found.

Notice that the flags are not available when executing saved states—all the command-line arguments are treated as Prolog arguments.

The development system checks that a valid SICStus license exists and, unless the --nologo option was used, responds with a message of identification and the prompt ‘| ?- ’ as soon as it is ready to accept input, thus:

SICStus 4.7.0 ...
Licensed to SICS
| ?- 

At this point the top level is expecting input of a query. You cannot type in clauses or directives immediately (see Section 3.3 [Inserting Clauses], page 22). While typing in a
query, the prompt (on following lines) becomes ‘|’. That is, the ‘| ?- ’ appears only for the first line of the query, and subsequent lines are indented.

### 3.2 Reading in Programs

A program is made up of a sequence of clauses and directives. The clauses of a predicate do not have to be immediately consecutive, but remember that their relative order may be important (see Section 4.2 [ref-sem], page 61).

To input a program from a file `file`, issue a query of the form:

```prolog
| ?- consult(file).
```

This instructs the interpreter to read in (consult) the program. Note that it may be necessary to enclose the filename `file` in single quotes to make it a legal Prolog atom; e.g.:

```prolog
| ?- consult('myfile.pl').
```

```prolog
| ?- consult('/usr/prolog/somefile').
```

The specified file is then read in. Clauses in the file are stored so that they can later be interpreted, while any directives are obeyed as they are encountered. When the end of the file is found, the system displays on the standard error stream the time spent. This indicates the completion of the query.

Predicates that expect the name of a Prolog source file, or more generally a file specification, use the facilities described in Section 4.5 [ref-fdi], page 95, to resolve the file name. File extensions are optional. There is also support for libraries.

This query can also take any list of filenames, such as:

```prolog
| ?- consult([myprog,extras_tests]).
```

In this case all three files would be consulted. The clauses for all the predicates in the consulted files will replace any existing clauses for those predicates, i.e. any such previously existing clauses in the database will be deleted.

### 3.3 Inserting Clauses at the Terminal

Clauses may also be typed in directly at the terminal, although this is only recommended if the clauses will not be needed permanently, and are few in number. To enter clauses at the terminal, you must give the special query:

```prolog
| ?- consult(user).
```

and the new prompt ‘| ’ shows that the system is now in a state where it expects input of clauses or directives. To return to top level, type “D. The system responds thus:

```
% consulted user in module user, 20 msec 200 bytes
```
3.4 Queries and Directives

Queries and directives are ways of directing the system to execute some goal or goals.

In the following, suppose that list membership has been defined by loading the following clauses from a file:

\[
\begin{align*}
\text{memb}(X, [X|_]). \\
\text{memb}(X, [_|L]) & :\! - \! \text{memb}(X, L).
\end{align*}
\]

(Notice the use of anonymous variables written ‘_’.)

3.4.1 Queries

The full syntax of a query is ‘?-’ followed by a sequence of goals. The top level expects queries. This is signaled by the initial prompt ‘| ?-’. Thus a query at top-level looks like:

\[
| ?- \text{memb}(b, [a,b,c]).
\]

Remember that Prolog terms must terminate with a full stop (‘.’, possibly followed by whitespace), and that therefore Prolog will not execute anything until you have typed the full stop (and then RET) at the end of the query.

If the goal(s) specified in a query can be satisfied, and if no variables not beginning with ‘_’ were bound, as in this example, then the system answers:

\[
\text{yes}
\]

and execution of the query terminates.

If some query variables not beginning with ‘_’ were bound, then the final value of each variable is displayed. Thus the query:

\[
| ?- \text{memb}(X, [a,b,c]).
\]

would be answered by:

\[
X = a ?
\]

At this point, the development system accepts one-letter commands corresponding to certain actions. To execute an action simply type the corresponding character (lower or upper case) followed by RET. The available commands in development systems are:

RET

y "accepts" the solution; the query is terminated and the development system responds with ‘yes’.

; “rejects” the solution; the development system backtracks (see Section 4.2 [ref-sem], page 61) looking for alternative solutions. If no further solutions can be found, then it outputs ‘no’.

b invokes a recursive top level.
In the top level, a global \texttt{printdepth} is in effect for limiting the subterm nesting level when printing bindings. The limit is initially 10. This command, without arguments, resets the printdepth to 10. With an argument of \texttt{n}, the printdepth is set to \texttt{n}, treating \texttt{0} as infinity. This command works by changing the value of the \texttt{toplevel_print_options} Prolog flag.

A local \texttt{subterm selector}, initially \texttt{[]}, is maintained. The subterm selector provides a way of zooming in to some subterm of each binding. For example, the subterm selector \texttt{[2,3]} causes the 3rd subterm of the 2nd subterm of each binding to be selected.

This command, without arguments, resets the subterm selector to \texttt{[]}. With an argument of \texttt{0}, the last element of the subterm selector is removed. With an argument of \texttt{n} (\texttt{n > 0}), \texttt{n} is added to the end of the subterm selector. With multiple arguments separated by whitespace, the arguments are applied from left to right.

\texttt{h} \hfill lists available commands.

While the variable bindings are displayed, all variables occurring in the values are replaced by friendlier variable names. Such names come out as a sequence of letters and digits preceded by ‘\_’. The outcome of some queries is shown below.

\begin{verbatim}
| ?- memb(X, [tom, dick, harry]).
  
  X = tom ;
  X = dick ;
  X = harry ;
  
  no
  | ?- memb(X, [a, b, f(Y, c)]), memb(X, [f(b, Z), d]).
  
  X = f(b, c),
  Y = b,
  Z = c
  
  | ?- memb(X, [f(_), g]).

  X = f(_, A)
\end{verbatim}

Directives are like queries except that:

1. Variable bindings are not displayed if and when the directive succeeds.
2. You are not given the chance to backtrack through other solutions.

\subsection*{3.4.2 Directives}

Directives start with the symbol ‘\texttt{:-}’. Any required output must be programmed explicitly; e.g. the directive:
:- memb(3, [1,2,3]), write(ok).

asks the system to check whether 3 belongs to the list [1,2,3]. Execution of a directive terminates when all the goals in the directive have been successfully executed. Other alternative solutions are not sought. If no solution can be found, then the system prints:

* Goal - goal failed

as a warning.

The principal use for directives (as opposed to queries) is to allow files to contain directives that call various predicates, but for which you do not want to have the answers printed out. In such cases you only want to call the predicates for their effect, i.e. you do not want terminal interaction in the middle of consulting the file. A useful example would be the use of a directive in a file that loads a whole list of other files, e.g.:

:- ensure_loaded(['bits', 'bobs', 'main', 'tests', 'data', 'junk']).

If a directive like this were contained in the file myprog, then typing the following at top level would be a quick way of reading in your entire program:

| ?- [myprog].

When simply interacting with the top level, this distinction between queries and directives is not normally very important. At top level you should just type queries normally. In a file, queries are in fact treated as directives, i.e. if you wish to execute some goals, then the directive in the file must be preceded by `:-' or `?-'; otherwise, it would be treated as a clause.

### 3.5 Syntax Errors

Syntax errors are detected during reading\(^1\). Each clause, directive or, in general, any term read in by the built-in predicate `read/1` that fails to comply with syntax requirements is displayed on the standard error stream as soon as it is read, along with its position in the input stream and a mark indicating the point in the string of symbols where the parser has failed to continue analysis, e.g.:

| memb(X, X$L).
| ! Syntax error
| ! , or ) expected in arguments
| ! in line 5
| ! memb ( X , X
| ! <<here>>
| ! $ L ).

if `$` has not been declared as an infix operator.

---

\(^1\) The SICStus Prolog IDE (see Section 3.11 [SPIDER], page 29) will show syntax errors and many other programming errors directly in the editor, while the code is written, without loading the code.
Note that any comments in the faulty line are not displayed with the error message. If you are in doubt about which clause was wrong, then you can use `consult/1`\(^2\) to load the code and then use the `listing/1` predicate to list all the clauses that were successfully read in, e.g.:

```
| ?- listing(memb/2).
```

**Please note:** The built-in predicates `read/[1,2]` normally raise an exception on syntax errors (see Section 4.15 [ref-ere], page 197). The behavior is controlled by the Prolog flag `syntax_errors`.

### 3.6 Undefined Predicates

There is a difference between predicates that have no definition and predicates that have no clauses. The latter case is meaningful e.g. for dynamic predicates (see Section 4.3.4 [ref-lod-dcl], page 83) that clauses are being added to or removed from. There are good reasons for treating calls to undefined predicates as errors, as such calls easily arise from typing errors.

The system can optionally catch calls to predicates that have no definition. First, the user defined predicate `user:unknown_predicate_handler/3` (see Section 4.15 [ref-ere], page 197) is called. If undefined or if the call fails, then the action is governed by the state of the `unknown` Prolog flag; see Section 4.9.4 [ref-lps-flg], page 136. Calls to predicates that have no clauses are not caught. See Section 11.3.244 [mpg-ref-unknown_predicate_handler], page 1238. Two development system predicates are handy in this context:

```
| ?- unknown(X,error).
% Undefined predicates will raise an exception (error)
X = error
```

This sets the flag and prints a message about the new value.

```
| ?- debugging.
The debugger is switched off
Using leashing stopping at [call, exit, redo, fail, exception] ports
Undefined predicates will raise an exception (error)
There are no breakpoints
```

This prints a message about the current value, as well as information about the state of the debugger.

### 3.7 Program Execution And Interruption

Execution of a program is started by giving the system a query that contains a call to one of the program’s predicates.

---

\(^2\) By default, e.g. when using `[myprog]`, code is *compiled*. Compiled code can not be listed with `listing/1`. Using `consult([myprog])` ensures that the code is *interpreted*, making it available for listing with `listing/1`.\n
Only when execution of one query is complete does the system become ready for another 
query. However, one may interrupt the normal execution of a query by typing \textasciitilde \textasciicircum C. This 
\textasciitilde C interruption has the effect of suspending the execution, and the following message is 
displayed:

\textbf{Prolog interruption (h or ? for help) ?}

At this point, the development system accepts one-letter commands corresponding to certain 
actions. To execute an action simply type the corresponding character (lower or upper case) 
followed by RET. The available commands in development systems are:

- \texttt{a} aborts the current computation.
- \texttt{c} continues the execution.
- \texttt{e} exits from SICStus Prolog, closing all files.
- \texttt{h} ? lists available commands.
- \texttt{b} invokes a recursive top level.
- \texttt{d} \texttt{z} \texttt{t} switch on the debugger. See Chapter 5 [Debug Intro], page 231.

If the standard input stream is not connected to the terminal, e.g. by redirecting standard 
input to a file or a pipe, then the above \textasciitilde C interrupt options are not available. Instead, 
typing \textasciitilde C causes SICStus Prolog to exit, and no terminal prompts are printed.

\subsection*{3.8 Exiting From The Top Level}

To exit from the top level and return to the shell, either type \textasciitilde D at the top level, or call the 
built-in predicate \texttt{halt/0}, or use the \texttt{e} (exit) command following a \textasciitilde C interruption.

\subsection*{3.9 Nested Executions—Break}

The Prolog system provides a way to suspend the execution of your program and to enter 
a new incarnation of the top level where you can issue queries to solve goals etc. This is 
achieved by issuing the query (see Section 3.7 [Execution], page 26):

\begin{verbatim}
| ?- break.
\end{verbatim}

This invokes a recursive top level, indicated by the message:

\begin{verbatim}
% Break level 1
\end{verbatim}

You can now type queries just as if you were at top level.

If another call of \texttt{break/0} is encountered, then it moves up to level 2, and so on. To close 
the break and resume the execution that was suspended, type \textasciitilde D. The debugger state and
current input and output streams will be restored, and execution will be resumed at the
predicate call where it had been suspended after printing the message:

% End break

3.10 Saving and Restoring Program States

Once a program has been read, the system will have available all the information necessary
for its execution. This information is called a program state.

The saved state of a program may be saved on disk for future execution. To save a program
into a file File, type the following query. On UNIX platforms, the file becomes executable:

| ?- save_program(File).

You can also specify a goal to be run when a saved program is restored. This is done by:

| ?- save_program(File, start).

where start/0 is the predicate to be called.

Once a program has been saved into a file File, the following query will restore the system
to the saved state:

| ?- restore(File).

If a saved state has been moved or copied to another machine, or if it is a symbolic link,
then the path names of foreign resources and other files needed upon restore are typically
different at restore time from their save time values. To solve this problem, certain atoms
will be renamed during restore as follows:

- Atoms that had $SP\_PATH/library (the name of the directory containing the Prolog
  Library) as prefix at save time will have that prefix replaced by the corresponding
  restore time value.
- Atoms that had the name of the directory containing File as prefix at save time will
  have that prefix replaced by the corresponding restore time value.

The purpose of this procedure is to be able to build and deploy an application consisting of
a saved state and other files as a directory tree with the saved state at the root: as long as
the other files maintain their relative position in the deployed copy, they can still be found
upon restore. See Section 6.8.2 [Building for a Target Machine], page 335, for an example.

Please note: When creating a saved state with \texttt{save\_program/[1,2]}, the names
and paths of foreign resources, are included in the saved state. After restoring
a saved state, this information is used to reload the foreign resources again.

The state of the foreign resource in terms of global C variables and allocated
memory is thus not preserved. Foreign resources may define init and deinit
functions to take special action upon loading and unloading; see Section 6.2.6
[Init and Deinit Functions], page 297.
As of release 3.8, partial saved states corresponding to a set of source files, modules, and predicates can be created by the built-in predicates `save_files/2`, `save_modules/2`, and `save_predicates/2` respectively. These predicates create files in a binary format, by default with the suffix `.po` (for Prolog object), which can be loaded by `load_files/[1,2]`. In fact, PO files use exactly the same binary format as saved states, and are subject to the same above-mentioned atom renaming rules. For example, to compile a program split into several source files into a single PO file, type:

```
| ?- compile(Files), save_files(Files, Object).
```

For each filename given, the first goal will try to locate a source file and compile it into memory. The second goal will save the program just compiled into a PO file whose default suffix is `.po`. Thus the PO file will contain a partial memory image.

Please note: PO files can be created with any suffix, but cannot be loaded unless the suffix is `.po`!

### 3.11 SICStus Prolog IDE

SICStus Prolog IDE, also known as SPIDER, is an Eclipse-based development environment for SICStus with many powerful features.

SPIDER was added in release 4.1 and is described on its own site, [https://sicstus.sics.se/spider/](https://sicstus.sics.se/spider/).

Some of the features of SPIDER are:

- **Semantic Highlighting**
  
  Code is highlighted based on semantic properties such as singleton variables, ...

- **On-the-fly warnings**
  
  The editor flags things like calls to undefined predicates, incorrect use of directives, missing declarations, ...

- **Pop-up documentation**
  
  Predicate documentation is parsed on-the-fly and shown when the mouse is hovering over a call. This works for both built-in and user-defined predicates.

- **Open Definition**
  
  Clicking on a called predicate can bring up its source code.

- **Call Hierarchy**
  
  Show callers and other references to a predicate or file.

- **Profiling**
  
  Show profiling data

- **Source Code Coverage**
  
  Show source code coverage, both as margin annotations and in various tabular forms.

- **Call Hierarchy**
  
  Show callers and other references to a predicate or file.
File outline
The predicates in a file are shown in an outline. They can be alphabetically sorted and non-exported predicates can be hidden from the outline.

Variable Bindings in Debugger
The debugger shows the names and values of variables.

Debugger Backtrace
Backtrace is shown and there are buttons for common debugger actions (Step Over, Step Out, Redo, . . .).

Source Code Debugging
Source-linked debugging. Works also for code that has no recorded source info, like the SICStus library.

Prolog Toplevel
The ordinary toplevel is still available, including the traditional debugger interface.

Attach to embedded code
SPIDER can attach to a SICStus runtime embedded in some other program.

Works With Existing Code
No need to reorganize your code, SPIDER can work with your existing folder structure.

Powered by Eclipse
Eclipse provides many features for free, like support for other programming languages, revision control, and much more.

Some of these features can be seen in the following screen shot.
Chapter 3: How to Run Prolog
Some Features of SPIDER

3.12 Emacs Interface

This section explains how to use the GNU Emacs interface for SICStus Prolog, and how to customize your GNU Emacs environment for it. Note that the SPIDER IDE (see Section 3.11 [SPIDER], page 29) has many more features than the GNU Emacs interface.

Emacs is a powerful programmable editor especially suitable for program development. It is available for free for many platforms, including various UNIX dialects, Windows and OS X. For information specific to GNU Emacs, see http://www.gnu.org. For information on running Emacs under Windows, see the ‘GNU Emacs FAQ For MS Windows’ at http://www.gnu.org/software/emacs/windows/ntemacs.html.

The advantages of using SICStus in the Emacs environment are source-linked debugging, auto indentation, syntax highlighting, help on predefined predicates (requires the SICStus info files to be installed), loading code from inside Emacs, auto-fill mode, and more.

The Emacs interface is not part of SICStus Prolog proper, but is included in the distribution for convenience. It was written by Emil Åström and Milan Zamazal, based on an earlier version of the mode written by Masanobu Umeda. Contributions have also been made by Johan Andersson, Peter Olin, Mats Carlsson, Johan Bevemyr, Stefan Andersson, Per Danielsson, Per Mildner, Henrik Båkman, and Tamás Rozmán. Some ideas and also a few lines of code have been borrowed (with permission) from Oz.el, by Ralf Scheidhauer and Michael Mehl, the Emacs major mode for the Oz programming language.

3.12.1 Installation


3.12.1.1 Quick-Start

Assuming the Emacs interface for SICStus Prolog has been installed in the default location, inserting a single line in your ~/.emacs will make Emacs use the SICStus Prolog mode automatically when editing files with a ‘.pro’ or ‘.pl’ extension. It will also ensure Emacs can find the SICStus executables and on-line documentation, etc.


Under UNIX, assuming SICStus 4.7.0 was installed in /usr/local/sicstus4.7.0/, add the following line:

    (load "/usr/local/sicstus4.7.0/lib/sicstus-4.7.0/emacs/sicstus_emacs_init")

Under Windows, assuming SICStus 4.7.0 was installed in C:\Program Files\SICStus Prolog VC15 4.7.0\, add the following line:
(load "C:/Program Files/SICStus Prolog VC15 4.7.0/emacs/sicstus_emacs_init")

No other configuration should be needed to get started. If you want to customize things, then look in the sicstus_emacs_init.el file and the rest of this section.

### 3.12.1.2 Customizing Emacs

Version 20 of GNU Emacs and XEmacs introduced a new method for editing and storing user settings. This feature is available from the menu bar as ‘Customize’ and particular Emacs variables can be customized with `M-x customize-variable`. Using ‘Customize’ is the preferred way to modify the settings for Emacs and the appropriate customize commands will be indicated below, sometimes together with the old method of directly setting Emacs variables.

### 3.12.1.3 Enabling Emacs Support for SICStus

This section is for reference only, it can safely be skipped; it will let you understand the setup that is performed by the sicstus_emacs_init.el file.

Assuming the Emacs interface for SICStus Prolog has been installed in the default location, inserting the following lines in your `~/.emacs` will make Emacs use this mode automatically when editing files with a `.pro` or `.pl` extension:

```lisp
(setq load-path
  (cons (expand-file-name "/usr/local/sicstus4.7.0/lib/sicstus-4.7.0/emacs")
    load-path))
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
(setq prolog-use-sicstus-sd t)
(setq auto-mode-alist (append '((("\.pro$" . prolog-mode)
    ("\.pl$" . prolog-mode))
  auto-mode-alist))
```

where the path in the first line is the file system path to prolog.el (the generic Prolog mode) and sicstus-support.el (SICStus specific code). For example, `~/emacs` means that the file is in the user’s home directory, in directory emacs. Windows paths can be written like C:/Program Files/SICStus Prolog VC15 4.7.0/emacs.

The last line above makes sure that files ending with `.pro` or `.pl` are assumed to be Prolog files and not Perl, which is the default Emacs setting for `.pl`. If this is undesirable, then remove that line. It is then necessary for the user to manually switch to Prolog mode by typing `M-x prolog-mode` after opening a Prolog file; for an alternative approach, see Section 3.12.4 [Mode Line], page 37.

If the shell command sicstus is not available in the default path, then it is necessary to set the value of the environment variable EPROLOG to a shell command to invoke SICStus Prolog. This is an example for C Shell:

```
% setenv EPROLOG /usr/local/sicstus4.7.0/bin/sicstus
```
3.12.1.4 Enabling Emacs Support for SICStus Documentation

If you follow the steps in Section Quick Start above, then you can skip this section.

It is possible to look up the documentation for any built-in or library predicate from within Emacs (using C-c ? or the menu). For this to work Emacs must be told about the location of the ‘info’-files that make up the documentation.

The default location for the ‘info’-files are <prefix>/lib/sicstus-4.7.0/doc/info/ on UNIX platforms and C:/Program Files/SICStus Prolog VC15 4.7.0/doc/info/ under Windows.

Add the following to your ~/.emacs file, assuming INFO is the path to the info files, e.g. C:/Program Files/SICStus Prolog VC15 4.7.0/doc/info/

```
(setq Info-default-directory-list
     (append Info-default-directory-list '("INFO")))
```

for GNU Emacs, or

```
(setq Info-directory-list
     (append Info-directory-list '("INFO")))
```

for XEmacs. You can also use M-x customize-group RET info RET if your Emacs is recent enough. You may have to quit and restart Emacs for these changes to take effect.

3.12.2 Basic Configuration

If the following lines are not present in ~/.emacs, then we suggest they are added, so that the font-lock mode (syntax coloring support) is enabled for all major modes in Emacs that support it.

```
(global-font-lock-mode t) ; GNU Emacs
(setq font-lock-auto-fontify t) ; XEmacs
(setq font-lock-maximum-decoration t)
```

These settings and more are also available through M-x customize-group RET font-lock.

If one wants to add font-locking only to the Prolog mode, then the two lines above could be replaced by:

```
(add-hook 'prolog-mode-hook 'turn-on-font-lock)
```

Similarly, to turn it off only for Prolog mode use:

```
(add-hook 'prolog-mode-hook 'turn-off-font-lock)
```

3.12.3 Usage

A Prolog process can be started by choosing Run Prolog from the Prolog menu, by typing C-c RET, or by typing M-x run-prolog. It is however not strictly necessary to start a Prolog process manually since it is automatically done when consulting or compiling, if needed.
The process can be restarted (i.e. the old one is killed and a new one is created) by typing `C-u C-c RET`, in this case Emacs will also prompt for a Lisp list of extra parameters to pass on the command line.

Programs are run and debugged in the normal way, with terminal I/O via the `*prolog*` buffer. The most common debugging predicates are available from the menu or via key-bindings.

A particularly useful feature under the Emacs interface is source-linked debugging. This is enabled or disabled using the Prolog/Source-linked debugging menu entry. It can also be enabled by setting the Emacs variable `prolog-use-sicstus-sd` to `t` in `~/emacs`. Both these methods set the Prolog flag `source_info` to `emacs`. Its value should be `emacs` while loading the code to be debugged and while debugging. If so, then the debugger will display the source code location of the current goal when it prompts for a debugger command, by highlighting the current line. If `source_info` was `off` when the code was loaded, or if it was asserted or loaded from `user`, then the current goal will still be shown but out of context.

Note that if the code has been modified since it was last loaded, then Prolog’s line number information may be invalid. If this happens, then just reload the relevant buffer.

Another useful feature which is available for code loaded with `source_info` switched on is that the debugger can show the variable bindings for the current goal, its ancestors, and the clauses they occur in. The bindings are shown in a separate `*Prolog Bindings*` buffer. This is enabled by the `C-c C-g` command and disabled by the `C-u C-c C-g` command.

Yet another feature which is available for compiled code loaded with `source_info` switched on is code coverage highlighting (see Section 9.3 [Coverage Analysis], page 356). Highlighting of the current buffer is refreshed by the `C-c C-o` command and cleared by the `C-u C-c C-o` command.

Consultation and compilation is either done via the menu or with the following key-bindings:

- `C-c C-f` Consult file.
- `C-c C-b` Consult buffer.
- `C-c C-r` Consult region.
- `C-c C-p` Consult predicate.
- `C-c C-c f` Compile file.
- `C-c C-c b` Compile buffer.
- `C-c C-c r` Compile region.
- `C-c C-c p` Compile predicate.

The boundaries used when consulting and compiling predicates are the first and last clauses of the predicate the cursor is currently in.

Other useful key-bindings are:
M-a  Go to beginning of clause. Go to the previous clause if already at the beginning.
M-e  Go to end of clause. Go to the next clause if already at the end.
C-M-c  Mark clause.
C-M-a  Go to beginning of predicate.
C-M-e  Go to end of predicate.
C-M-h  Mark predicate.
M-{  Go to the previous paragraph (i.e. empty line).
M-}  Go to the next paragraph (i.e. empty line).
M-h  Mark paragraph.
C-M-n  Go to matching right parenthesis.
C-M-p  Go to matching left parenthesis.
M-;  Creates a comment at comment-column. This comment will always stay at this position when the line is indented, regardless of changes in the text earlier on the line, provided that prolog-align-comments-flag is set to t.

C-c C-t
C-u C-c C-t
Enable and disable creeping, respectively.

C-c C-d
C-u C-c C-d
Enable and disable leaping, respectively.

C-c C-z
C-u C-c C-z
Enable and disable zipping, respectively.

C-c C-g  since release 4.2
C-u C-c C-g  since release 4.2
Enable and disable bindings window, respectively. When enabled, SICStus will endeavor to show the variable bindings of the clause containing the current goal. C-c C-g splits the *Prolog* window vertically and inserts the *Prolog Bindings* window, which shows the bindings and is updated upon every debugger command. C-u C-c C-g deletes the *Prolog Bindings* window.

C-c <  since release 4.3.2
Set the print depth for the bindings window as well as for the top level. Prompts for an integer value. Equivalent to the < top-level command; see Section 3.4.1 [Queries], page 23.

C-c C-o  since release 4.2
C-u C-c C-o  since release 4.2
Refresh and clear coverage highlighting for the current buffer, respectively. Lines containing coverage sites (see Section 9.3 [Coverage Analysis], page 356) will be highlighted in face pltrace-face-reached-det (defaults to green) if
they were hit at least once and made no nondet calls with the execution profiler switched on; in face `pltrace-face-reached-nondet' (defaults to yellow) if they were hit at least once and made one or more nondet calls with the execution profiler switched on; otherwise, they will be highlighted in face `pltrace-face-reached-not' (defaults to red). Lines not containing coverage sites are not highlighted.

\texttt{C-x SPC} \texttt{C-u C-x SPC}

Set and remove a line breakpoint. This uses the advanced debugger features introduced in release 3.8; see Section 5.6 [Advanced Debugging], page 243.

\texttt{C-c C-s} \hspace{1cm} Insert the PredSpec of the current predicate into the code.

\texttt{C-c C-n} \hspace{1cm} Insert the template of the current predicate (name, parentheses, commas) into the code.

\texttt{M-RET} \hspace{1cm} \texttt{since release 4.2}

Insert a line break followed by the template of the current predicate into the code. This can be useful when writing recursive predicates or predicates with several clauses. See also the \texttt{prolog-electric-dot-flag} variable below.

\texttt{C-c C-v a} \hspace{1cm} Convert all variables in a region to anonymous variables. See also the \texttt{prolog-electric-underscore-flag} Emacs variable.

\texttt{C-c ?} \hspace{1cm} Help on predicate. This requires the SICStus info files to be installed. If the SICStus info files are installed in a nonstandard way, then you may have to change the Emacs variable \texttt{prolog-info-predicate-index}.

\texttt{C-c RET} \hspace{1cm} \texttt{since release 4.2}
\texttt{C-u C-c RET} \hspace{1cm} \texttt{since release 4.2}

Run Prolog. With the second variant, Emacs will prompt for a Lisp list of extra parameters to pass on the command line.

\texttt{C-c C-c} \hspace{1cm} \texttt{since release 4.2}

Interrupt Prolog. The same as typing `\texttt{^C}' in a shell.

\texttt{C-c C-\textbackslash} \hspace{1cm} \texttt{since release 4.2}

Kill Prolog. Immediately kills the process.

### 3.12.4 Mode Line

If working with an application split into several modules, then it is often useful to let files begin with a “mode line”:

```
%% %%% -*- Mode: Prolog; Module: ModuleName; -*-
```

The Emacs interface will look for the mode line and notify the SICStus Prolog module system that code fragments being incrementally reconsulted or recompiled should be imported into the module \texttt{ModuleName}. If the mode line is missing, then the code fragment will be imported into the type-in module. An additional benefit of the mode line is that it tells Emacs that the file contains Prolog code, regardless of the setting of the Emacs variable \texttt{auto-mode-alist}. A mode line can be inserted by choosing \texttt{Insert/Module modeline} in the Prolog menu.
3.12.5 Configuration

The behavior of the Emacs interface can be controlled by a set of user-configurable settings. Some of these can be changed on the fly, while some require Emacs to be restarted. To set a variable on the fly, type \texttt{M-x set-variable RET VariableName RET Value RET}. Note that variable names can be completed by typing a few characters and then pressing \texttt{TAB}.

To set a variable so that the setting is used every time Emacs is started, add lines of the following format to \\texttt{~/.emacs}:

\[
\texttt{(setq VariableName Value)}
\]

Note that the Emacs interface is presently not using the ‘Customize’ functionality to edit the settings.

The available settings are:

\textbf{prolog-system}

The Prolog system to use. Defaults to \texttt{sicstus}, which will be assumed for the rest of this chapter. See the on-line documentation for the meaning of other settings. For other settings of \texttt{prolog-system} the variables below named \texttt{sicstus-something} will not be used, in some cases corresponding functionality is available through variables named \texttt{prolog-something}.

\textbf{sicstus-version}

The version of SICStus that is used. Defaults to \texttt{(4 . 2)}. Note that the spaces are significant!

\textbf{prolog-use-sicstus-sd}

Set to \texttt{t} (the default) to enable the source-linked debugging extensions by default. The debugging can be enabled via the Prolog menu even if this variable is \texttt{nil}. Note that the source-linked debugging only works if \texttt{sicstus-version} is set correctly.

\textbf{prolog-indent-width}

How many positions to indent the body of a clause. Defaults to \texttt{tab-width}, normally 8.

\textbf{prolog-paren-indent}

The number of positions to indent code inside grouping parentheses. Defaults to 4, which gives the following indentation.

\[
p :-
\quad ( \quad q1
\quad ; \quad q2,
\quad q3
\quad ).
\]

Note that the spaces between the parentheses and the code are automatically inserted when \texttt{TAB} is pressed at those positions.
prolog-align-comments-flag
Set to nil to prevent single \%-comments from being automatically aligned. Defaults to t.
Note that comments with one \% are indented to comment-column, comments with two \% to the code level, and that comments with three \% are never changed when indenting.

prolog-indent-mline-comments-flag
Set to nil to prevent indentation of text inside /* ... */ comments. Defaults to t.

sicstus-keywords
This is a list with keywords that are highlighted in a special color when used as directives (i.e. as :- keyword). Defaults to
'("block" "discontiguous" "dynamic" "initialization"
"meta_predicate" "mode" "module" "multifile" "public" "volatile"
"det" "nondet" ; for spdet
)

prolog-electric-newline-flag
Set to nil to prevent Emacs from automatically indenting the next line when pressing RET. Defaults to t.

prolog-hungry-delete-key-flag
Set to t to enable deletion of all whitespace before the cursor when pressing DEL (unless inside a comment, string, or quoted atom). Defaults to nil.

prolog-electric-dot-flag
Set to t to enable the electric dot function. If enabled, then pressing . at the end of a non-empty line inserts a dot and a newline. When pressed at the beginning of a line, a new head of the last predicate is inserted. When pressed at the end of a line with only whitespace, a recursive call to the current predicate is inserted. The function respects the arity of the predicate and inserts parentheses and the correct number of commas for separation of the arguments. Defaults to nil.

prolog-electric-underscore-flag
Set to t to enable the electric underscore function. When enabled, pressing underscore (_) when the cursor is on a variable, replaces the variable with the anonymous variable. Defaults to nil.

prolog-use-prolog-tokenizer-flag
Set to nil to use built-in functions of Emacs for parsing the source code when indenting. This is faster than the default but does not handle some of the syntax peculiarities of Prolog. Defaults to t.

prolog-parse-mode
What position the parsing is done from when indenting code. Two possible settings: 'beg-of-line and 'beg-of-clause. The first is faster but may result in erroneous indentation in /* ... */ comments. The default is 'beg-of-line.
prolog-imenu-flag
Set to t to enable a new Predicate menu that contains all predicates of the current file. Choosing an entry in the menu moves the cursor to the start of that predicate. Defaults to nil.

prolog-info-predicate-index
The info node for the SICStus predicate index. This is important if the online help function is to be used (by pressing C-c ?, or choosing the Prolog/Help on predicate menu entry). The default setting is "(sicstus)Predicate Index".

prolog-underscore-wordchar-flag
Set to nil to not make underscore (_) a word-constituent character. Defaults to t.

Font-locking uses a number of “faces”, which can be customized with regular Emacs commands, for instance M-x describe-face RET FaceName RET. The following faces are relevant:

highlight
Source code highlight at debug ports.

pltrace-face-reached-det
Highlight for a line of code reached by coverage analysis with no nondet calls made from that line of code.

pltrace-face-reached-nondet
Highlight for a line of code reached by coverage analysis with one or more nondet calls made from that line of code.

pltrace-face-reached-not
Highlight for a line of code not reached by coverage analysis.

prolog-warning-face
Face used in warning messages.

prolog-informational-face
Face used in informational messages.

prolog-exception-face
Face used in the first line of an error exception message, as well as to highlight Exception port displays.

prolog-error-face
Face used in other lines of exception messages.

prolog-call-face
Face used to highlight Call port displays.

prolog-exit-face
Face used to highlight Exit port displays.

prolog-redo-face
Face used to highlight Redo port displays.
prolog-fail-face  
    Face used to highlight Fail port displays.

prolog-builtin-face  
    Face used to highlight keywords used in directives (see sicstus-keywords).

3.12.6 Tips
Some general tips and tricks for using the SICStus mode and Emacs in general are given here. Some of the methods may not work in all versions of Emacs.

3.12.6.1 Font-locking
When editing large files, it might happen that font-locking is not done because the file is too large. Typing \texttt{M-x lazy-lock-mode}, which is much faster, results in only the visible parts of the buffer being highlighted; see its Emacs on-line documentation for details.

If the font-locking seems to be incorrect, then choose Fontify Buffer from the Prolog menu.

3.12.6.2 Auto-fill Mode
Auto-fill mode is enabled by typing \texttt{M-x auto-fill-mode}. This enables automatic line breaking with some features. For example, the following multiline comment was created by typing \texttt{M-;} followed by the text. The second line was indented and a \texttt{"\%\"} was added automatically.

\begin{verbatim}
dynamics([]). % A list of pit furnace
% dynamic instances
\end{verbatim}

3.12.6.3 Speed
There are several things to do if the speed of the Emacs environment is a problem:

- First of all, make sure that \texttt{prolog.el} and \texttt{sicstus-support.el} are compiled, i.e. that there is a \texttt{prolog.elc} and a \texttt{sicstus-support.elc} file at the same location as the original files. To do the compilation, start Emacs and type \texttt{M-x byte-compile-file RET path RET}, where \texttt{path} is the path to the \texttt{'*.el'} file. Do Not be alarmed if there are a few warning messages as this is normal. If all went well, then there should now be a compiled file, which is used the next time Emacs is started.

- The next thing to try is changing the setting of \texttt{prolog-use-prolog-tokenizer-flag} to \texttt{nil}. This means that Emacs uses built-in functions for some of the source code parsing, thus speeding up indentation. The problem is that it does not handle all peculiarities of the Prolog syntax, so this is a trade-off between correctness and speed.

- The setting of the \texttt{prolog-parse-mode} variable also affects the speed, \texttt{'beg-of-line} being faster than \texttt{'beg-of-clause}.

- Font locking may be slow. You can turn it off using customization, available through \texttt{M-x customize-group RET font-lock RET}. An alternative is to enable one of the lazy font locking modes. You can also turn it off completely; see Section 3.12.2 [Basic Configuration], page 34.
3.12.6.4 Changing Colors

The Prolog mode uses the default Emacs colors for font-locking as far as possible. The only custom settings are in the Prolog process buffer. The default settings of the colors may not agree with your preferences, so here is how to change them.

If your Emacs supports it, then use ‘Customize’. `M-x customize-group RET font-lock RET` will show the ‘Customize’ settings for font locking and also contains pointers to the ‘Customize’ group for the font lock (type)faces. The rest of this section outlines the more involved methods needed in older versions of Emacs.

First of all, list all available faces (a face is a combined setting of foreground and background colors, font, boldness, etc.) by typing `M-x list-faces-display`.

There are several functions that change the appearance of a face, the ones you will most likely need are:

- `set-face-foreground`
- `set-face-background`
- `set-face-underline-p`
- `make-face-bold`
- `make-face-bold-italic`
- `make-face-italic`
- `make-face-unbold`
- `make-face-unitalic`

These can be tested interactively by typing `M-x function-name`. You will then be asked for the name of the face to change and a value. If the buffers are not updated according to the new settings, then refontify the buffer using the `Fontify Buffer` menu entry in the `Prolog` menu.

Colors are specified by a name or by RGB values. Available color names can be listed with `M-x list-colors-display`.

To store the settings of the faces, a few lines must be added to `~/.emacs`. For example:

```lisp
;; Customize font-lock faces
(add-hook 'font-lock-mode-hook
  '(lambda ()
     (set-face-foreground font-lock-variable-name-face "#00a000")
     (make-face-bold font-lock-keyword-face)
     (set-face-foreground font-lock-reference-face "Blue")
  )))
```
4 The Prolog Language

This chapter describes the syntax and semantics of the Prolog language, and introduces the central built-in predicates and other important language constructs. In many cases, an entry in a list of built-in predicates, will be annotated with keywords. These annotations are defined in Section 11.1.3 [mpg-ref-cat], page 900.

4.1 Syntax
4.1.1 Overview
This section describes the syntax of SICStus Prolog.

4.1.2 Terms
4.1.2.1 Overview
The data objects of the language are called terms. A term is either a constant, a variable, or a compound term.

A constant is either a number (integer or floating-point) or an atom. Constants are definite elementary objects, and correspond to proper nouns in natural language.

Variables and compound terms are described in Section 4.1.2.5 [ref-syn-trm-var], page 44, and Section 4.1.3 [ref-syn-cpt], page 45, respectively.

Foreign data types are discussed in the context of library(structs); see Section 10.43 [lib-structs], page 759.

4.1.2.2 Integers
The printed form of an integer consists of a sequence of digits optionally preceded by a minus sign (‘-’). These are normally interpreted as base 10 integers. It is also possible to enter integers in base 2 (binary), 8 (octal), and 16 (hexadecimal); this is done by preceding the digit string by the string ‘0b’, ‘0o’, or ‘0x’ respectively. The characters A-F or a-f stand for digits greater than 9. For example, the following tokens all represent the integer fifteen:

\[
15 \quad 0b1111 \quad 0o17 \quad 0xf
\]

Note that

\[
+525
\]

is not a valid integer.

There is also a special notation for character constants. E.g.:

\[
0'A \quad 0'\x41\quad 0'\101\n\]

are all equivalent to 65 (the character code for ‘A’). ‘0’ followed by any character except ‘\’ (backslash) is thus read as an integer. If ‘0’ is followed by ‘\’, then the ‘\’ denotes the start of an escape sequence with special meaning (see Section 4.1.7.6 [ref-syn-syn-esc], page 60).
### 4.1.2.3 Floating-point Numbers

A floating-point number (float) consists of a sequence of digits with an embedded decimal point, optionally preceded by a minus sign (-), and optionally followed by an exponent consisting of upper- or lowercase ‘E’ and a signed base 10 integer. Examples of floats are:

```
1.0  -23.45  187.6E12  -0.0234e15  12.0E-2
```

Note that there must be at least one digit before, and one digit after, the decimal point.

### 4.1.2.4 Atoms

An atom is identified by its name, which is a sequence characters, and can be written in any of the following forms:

- Any sequence of alphanumeric characters (including ‘_’), starting with a lowercase letter. Note that an atom may not begin with an underscore. The characters that are allowed to occur in such an unquoted atom are restricted to a subset of Unicode; see Section 4.1.7.5 [ref-syn-syn-tok], page 56.

- Any sequence from the following set of characters (except ‘/*’, which begins a comment):
  
  + - * / \ ^ < > = \ . ? @ # $ &

- Any sequence of characters delimited by single quotes. Backslashes in the sequence denote escape sequences (see Section 4.1.7.6 [ref-syn-syn-esc], page 60), and if the single quote character is included in the sequence, then it must be escaped, e.g. 'can\'t'. The characters that are allowed to occur in such a quoted atom are restricted to a subset of Unicode; see Section 4.1.7.5 [ref-syn-syn-tok], page 56.

- Any of:
  
  ! ; [] {} 

Note that the bracket pairs are special: ‘[]’ and ‘{}’ are atoms but ‘[’, ‘]’, ‘{’, and ‘}’ are not. The form \[X\] is a special notation for lists (see Section 4.1.3.1 [ref-syn-cpt-lis], page 46) as an alternative to .(X,[]) and the form {X} is allowed as an alternative to {[X]}.

Examples of atoms are:

```
a  void  =  :=  'Anything in quotes'  []
```

**Please note:** It is recommended that you do not invent atoms beginning with the character ‘$', since it is possible that such names may conflict with the names of atoms having special significance for certain built-in predicates.

### 4.1.2.5 Variables

Variables may be written as any sequence of alphanumeric characters (including ‘_’) beginning with either a capital letter or ‘_’. For example:

```
X  Value  A  A1  _3  _RESULT
```
If a variable is referred to only once in a clause, then it does not need to be named and may be written as an anonymous variable, represented by the underline character ‘\_' by itself. Any number of anonymous variables may appear in a clause; they are read as distinct variables. Anonymous variables are not special at run time.

### 4.1.2.6 Foreign Terms

Pointers to C data structures can be handled using the Structs package.

### 4.1.3 Compound Terms

The structured data objects of Prolog are compound terms. A compound term comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is characterized by its name, which is an atom, and its arity or number of arguments. For example, the compound term whose principal functor is ‘point’ of arity 3, and which has arguments X, Y, and Z, is written

\[
\text{point}(X, Y, Z)
\]

When we need to refer explicitly to a functor we will normally denote it by the form \text{Name/Arity}. Thus, the functor ‘point’ of arity 3 is denoted

\[
\text{point}/3
\]

Note that a functor of arity 0 is represented as an atom.

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the (compound) term

\[
\text{s(np(john), vp(v(likes), np(mary)))}
\]

would be pictured as the following tree:

```
       s
       /\n      np  vp
     / \  /  \\
  john v  np
    |    |    |
   likes mary
```

The principal functor of this term is \text{s/2}. Its arguments are also compound terms. In illustration, the principal functor of the first argument is \text{np/1}.

Sometimes it is convenient to write certain functors as operators; binary functors (that is, functors of two arguments) may be declared as infix operators, and unary functors (that is, functors of one argument) may be declared as either prefix or postfix operators. Thus it is possible to write

\[
X+Y \quad P;Q \quad X<Y \quad +X \quad P;
\]
as optional alternatives to

\[ + (X, Y) ; (P, Q) \quad < (X, Y) \quad + (X) ; (P) \]

The use of operators is described fully in Section 4.1.5 [ref-syn-ops], page 47.

### 4.1.3.1 Lists

Lists form an important class of data structures in Prolog. They are essentially the same as the lists of Lisp: a list is either the atom [], representing the empty list, or else a compound term with functor . and two arguments, which are the head and tail of the list respectively, where the tail of a list is another list. Thus a list of the first three natural numbers is the structure

```
/ \  / \  / \
1  . 2  . 3  []
```

which could be written using the standard syntax, as (A) but which is normally written in a special list notation, as (B). Two examples of this list notation, as used when the tail of a list is a variable, are (C), which represent the structure in (D).

- \( .(1, .(2, .(3, []))) \)  
- \([1, 2, 3]\)  
- \([X|L] \quad [a, b|L]\)

Note that the notation \([X|L]\) does not add any new power to the language; it simply improves readability. These examples could be written equally well as (E).

- \( .(X, L) \quad .(a, .(b, L)) \)

### 4.1.3.2 Strings As Lists

For convenience, a further notational variant is allowed for lists of integers that correspond to character codes. Lists written in this notation are called strings. E.g.:

"SICStus"

which, by default, denotes exactly the same list as
The Prolog flag double_quotes can be used to change the way strings are interpreted. The default value of the flag is codes, which implies the above interpretation. If the flag is set to chars, then a string is transformed to a list of character atoms. E.g. with this setting the above string represents the list:

['S','I','C','S',t,u,s]

Finally if double_quotes has the value atom, then the string is made equivalent to the atom formed from its characters: the above sample string is then the same as the atom 'SICStus'.

Please note: Most code assumes that the Prolog flag double_quotes has its default value (codes). Changing this flag is not recommended.

Backslashes in the sequence denote escape sequences (see Section 4.1.7.6 [ref-syn-syn-esc], page 60). As for quoted atoms, if a double quote character is included in the sequence, then it must be escaped, e.g. "can\"t".

The built-in predicates that print terms (see Section 4.6.4 [ref-iou-tou], page 104) do not use string syntax even if they could.

The characters that are allowed to occur within double quotes are restricted to a subset of Unicode; see Section 4.1.7.5 [ref-syn-syn-tok], page 56.

4.1.4 Character Escaping

The character escaping facility is prescribed by the ISO Prolog standard, and allows escape sequences to occur within strings and quoted atoms, so that programmers can put non-printable characters in atoms and strings and still be able to see what they are doing.

Strings or quoted atoms containing escape sequences can occur in terms obtained by read/[1,2], compile/1, and so on. The ‘0’ notation for the integer code of a character is also affected by character escaping.

The only characters that can occur in a string or quoted atom are the printable characters and SPC. All other whitespace characters must be expressed with escape sequences (see Section 4.1.7.6 [ref-syn-syn-esc], page 60).

4.1.5 Operators and their Built-in Predicates

4.1.5.1 Overview

Operators in Prolog are simply a notational convenience. For example, ‘+’ is an infix operator, so

\[ 2 + 1 \]

is an alternative way of writing the term +(2, 1). That is, \( 2 + 1 \) represents the data structure
and not the number 3. (The addition would only be performed if the structure were passed as an argument to an appropriate procedure, such as \texttt{is/2}; see Section 4.7.2 [ref-ari-ae], page 119.)

Prolog syntax allows operators of three kinds: \textit{infix}, \textit{prefix}, and \textit{postfix}. An infix operator appears between its two arguments, while a prefix operator precedes its single argument and a postfix operator follows its single argument.

Each operator has a \textit{precedence}, which is a number from 1 to 1200. The precedence is used to disambiguate expressions in which the structure of the term denoted is not made explicit through the use of parentheses. The general rule is that the operator with the \textit{highest} precedence is the principal functor. Thus if `+` has a higher precedence than `/`, then

\[ a+b/c = a+(b/c) \]

are equivalent, and denote the term \(+\langle a,/(b,c)\rangle\). Note that the infix form of the term \(/\langle+(a,b),c\rangle\) must be written with explicit parentheses:

\[ (a+b)/c \]

If there are two operators in the expression having the same highest precedence, then the ambiguity must be resolved from the \textit{types} of the operators. The possible types for an infix operator are

- \texttt{xfx}
- \texttt{xfy}
- \texttt{yfx}

Operators of type `\texttt{xfx}` are not associative: it is required that both of the arguments of the operator be subexpressions of \textit{lower} precedence than the operator itself; that is, the principal functor of each subexpression must be of lower precedence, unless the subexpression is written in parentheses (which gives it zero precedence).

Operators of type `\texttt{xfy}` are right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the \textit{same} precedence as the main operator. Left-associative operators (type `\texttt{yfx}`) are the other way around.

An atom named \texttt{Name} is declared as an operator of type \texttt{Type} and precedence \texttt{Precedence} by the command

\[ :-op(Precedence, \texttt{Type}, \texttt{Name}). \]

An operator declaration can be canceled by redeclaring the \texttt{Name} with the same \texttt{Type}, but \texttt{Precedence} 0.
The argument Name can also be a list of names of operators of the same type and precedence.

It is possible to have more than one operator of the same name, so long as they are of different kinds: infix, prefix, or postfix. Note that the ISO Prolog standard contains the restriction that there should be no infix and postfix operators with the same name, however, SICStus Prolog lifts this restriction.

An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to operators that are provided as standard, except for the ',' operator. Declarations for all these built-in operators can be found in Section 4.1.5.4 [ref-syn-ops-bop], page 51.

For example, the built-in operators '+’ and ‘-’ are as if they had been declared by (A) so that (B) is valid syntax, and means (C) or pictorially (D).

\[\text{:-op(500, yfx, [+,-]).} \quad \text{(A)}\]

\[a-b+c \quad \text{(B)}\]

\[(a-b)+c \quad \text{(C)}\]

\[\begin{array}{c}
+ \\
/ \\
\_ \\
\text{- c} \\
/ \\
\text{\_ b} \\
\end{array} \quad \text{(D)}\]

The list functor ./2 is not a standard operator, but we could declare it to be (E) and then (F) would represent the structure (G).

\[\text{:-op(600, xfy, .).} \quad \text{(E)}\]

\[a.b.c \quad \text{(F)}\]

\[\begin{array}{c}
. \\
/ \\
\_ a \\
/ \\
\text{\_ b c} \\
\end{array} \quad \text{(G)}\]

Contrasting this with the diagram above for a-b+c shows the difference between ‘yfx’ operators where the tree grows to the left, and ‘xfy’ operators where it grows to the right. The tree cannot grow at all for ‘xfx’ operators; it is simply illegal to combine ‘xfx’ operators having equal precedences in this way.

The possible types for a prefix operator are:

- fx
- fy
and for a postfix operator they are:

- \( xf \)
- \( yf \)

The meaning of the types should be clear by analogy with those for infix operators. As an example, if \( \text{not} \) were declared as a prefix operator of type \( fy \), then

\[
\text{not not } P
\]

would be a permissible way to write \( \text{not}(\text{not}(P)) \). If the type were \( fx \), then the preceding expression would not be legal, although

\[
\text{not } P
\]

would still be a permissible form for \( \text{not}(P) \).

If these precedence and associativity rules seem rather complex, then remember that you can always use parentheses when in any doubt.

### 4.1.5.2 Manipulating and Inspecting Operators

To add or remove an operator, use \( \text{op}(\text{Precedence}, \text{Type}, \text{Name}) \). \( \text{op}/3 \) declares the atom \( \text{Name} \) to be an operator of the stated \( \text{Type} \) and \( \text{Precedence} \). If \( \text{Precedence} \) is 0, then the operator properties of \( \text{Name} \) (if any) are canceled. **Please note**: operators are *global*, as opposed to being local to the current module, Prolog text, or otherwise. See Section 11.3.147 [mpg-ref-op], page 1114.

To examine the set of operators currently in force, use \( \text{current_op}(\text{Precedence}, \text{Type}, \text{Name}) \). See Section 11.3.53 [mpg-ref-current_op], page 995.

### 4.1.5.3 Syntax Restrictions

Note carefully the following syntax restrictions, which serve to remove potential ambiguities associated with prefix operators.

1. The arguments of a compound term written in standard syntax must be expressions of precedence *less than* 1000. Thus it is necessary to write the expression \( P:-Q \) in parentheses

\[
\text{assert}((P:-Q))
\]

because the precedence of the infix operator ‘:-’, and hence of the expression \( P:-Q \), is 1200. Enclosing the expression in parentheses reduces its precedence to 0.

2. Similarly, the elements of a list written in standard syntax must be expressions of precedence *less than* 1000. Thus it is necessary to write the expression \( P->Q \) in parentheses

\[
[(P->Q)]
\]

because the precedence of the infix operator ‘->’, and hence of the expression \( P->Q \), is 1050. Enclosing the expression in parentheses reduces its precedence to 0.

3. In a term written in standard syntax, the principal functor and its following ‘(’ must *not* be separated by any intervening spaces, newlines, or other characters. Thus
point (X,Y,Z)
is invalid syntax.

4. If the argument of a prefix operator starts with a ‘(’, then this ‘(’ must be separated from the operator by at least one space or other whitespace character. Thus

\[-(p;q),r.

(where ‘:-‘ is the prefix operator) is invalid syntax. The system would try to interpret it as the structure:

```
,  
/ \   
:-   r
|  
;   
/ \   
p q
```

That is, it would take ‘:-‘ to be a functor of arity 1. However, since the arguments of a functor are required to be expressions of precedence less than 1000, this interpretation would fail as soon as the ‘;‘ (precedence 1100) were encountered.

In contrast, the term:

\[- (p;q),r.

is valid syntax and represents the following structure:

```
:-  
|  
,   
/ \   
; r   
/ \   
p q
```

4.1.5.4 Built-in Operators
The above operators are as in the ISO Prolog standard, except the following, which are not present in ISO Prolog at all:

4.1.6 Commenting
Comments have no effect on the execution of a program, but they are very useful for making programs more comprehensible. Two forms of comments are allowed:

1. The character ‘%’ followed by any sequence of characters up to the end of the line.
2. The symbol ‘/∗’ followed by any sequence of characters (including newlines) up to the symbol ‘*/’.

4.1.7 Formal Syntax
4.1.7.1 Overview
A Prolog program consists of a sequence of sentences. Each sentence is a Prolog term. How sentences are interpreted as terms is defined in Section 4.1.7.3 [ref-syn-syn-sen], page 53, below. Note that a term representing a sentence may be written in any of its equivalent syntactic forms. For example, the functor :-/2 could be written in standard functional notation instead of as the usual infix operator.
Terms are written as sequences of tokens. Tokens are sequences of characters, which are treated as separate symbols. Tokens include the symbols for variables, constants, and functors, as well as punctuation characters such as parentheses and commas.

The interpretation of sequences of tokens as terms is defined in Section 4.1.7.4 [ref-syn-syn-trm], page 55. Each list of tokens that is read in (for interpretation as a term or sentence) must be terminated by a full stop (a period followed by a whitespace character such as newline or space) token. Two tokens must be separated by a space if they could otherwise be interpreted as a single token. Both spaces and comments are ignored when interpreting the token list as a term. A comment may appear at any point in a token list (separated from other tokens by spaces where necessary).

The interpretation of sequences of characters as tokens is defined in Section 4.1.7.5 [ref-syn-syn-tok], page 56. The next section describes the notation used in the formal definition of Prolog syntax.

4.1.7.2 Notation

- Syntactic categories (or nonterminals) are printed in italics, for example query. Depending on the section, a category may represent a class of either terms, token lists, or character strings.
- A syntactic rule takes the general form
  \[ C ::= F1 | F2 | F3 \]
  which states that an entity of category \( C \) may take any of the alternative forms \( F1 \), \( F2 \), or \( F3 \).
- Certain definitions and restrictions are given in ordinary English, enclosed in braces (\( \{ \} \)).
- A category written as ‘\( C... \)’ denotes a sequence of one or more \( C \)s.
- A category written as ‘\( ?C \)’ denotes an optional \( C \). Therefore ‘\( ?C... \)’ denotes a sequence of zero or more \( C \)s.
- A few syntactic categories have names with arguments, and rules in which they appear may contain meta-variables in the form of italicized capital letters. The meaning of such rules should be clear from analogy with the definite clause grammars described in Section 4.14 [ref-gru], page 189.
- In Section 4.1.7.4 [ref-syn-syn-trm], page 55, particular tokens of the category Name (a name beginning with a capital letter) are written as quoted atoms, while tokens that are individual punctuation characters are written literally.

4.1.7.3 Syntax of Sentences as Terms

\[
\text{sentence} ::= \text{module} : \text{sentence} \\
| \text{list} \{ \text{where list is a list of sentence} \}
\]
| clause  |
| directive  |
| query  |
| grammar-rule  |

clause ::= rule | unit-clause
rule ::= head :- body
unit-clause ::= head  
            { where head is not otherwise a sentence }
directive ::= :- body
query ::= ?- body
head ::= module : head
       { where goal is not a variable }
body ::= module : body
       | body -> body disj body
       | body -> body
       | ^ body
       | body disj body
       | body , body
       | once(body)
       | do(iter,body)
       | if(body,body,body)
       | term ^ body
       | goal
goal ::= term  
       { where term is not otherwise a body }
gr-rule ::= gr-head --> gr-body
gr-head ::= module : gr-head
          | gr-head , terminals
          { where non-terminal is not a variable }
gr-body ::= module : gr-body
          | gr-body -> gr-body disj gr-body
          | gr-body -> gr-body
          | ^ gr-body
          | gr-body disj gr-body
          | gr-body , gr-body
          | once(gr-body)
          | do(iter,gr-body)
          | if(gr-body,gr-body,gr-body)
          | term ^ gr-body
          | non-terminal
          | terminals
          | gr-condition
non-terminal ::= term  \{ where term is not otherwise a gr-body \}

terminals ::= list | string

gr-condition ::= ! | \{body\}

module ::= atom

disj ::= ; | |
\{ read as ; unless | is declared infix \}

iter ::= iter, iter
| fromto(term,term,term,term)
| foreach(term,term)
| foreacharg(term,term)
| foreacharg(term,term,term)
| count(term,term,term)
| for(term,term,term)
| param(term)

4.1.7.4 Syntax of Terms as Tokens

term-read-in ::= subterm(1200) full stop  \{ where M is less than or equal to N \}

subterm(N) ::= term(M)
\{ except in the case of a number if subterm starts with a ‘,’ op must be followed by whitespace-text \}

subterm(N) ::= op(N,fx) subterm(N-1)
\{ if subterm starts with a ‘,’ op must be followed by whitespace-text \}

| op(N,fy) subterm(N)

| subterm(N-1) op(N,xfx)

subterm(N-1)
| subterm(N-1) op(N,xfy) subterm(N)
| subterm(N) op(N,yfx)

subterm(N-1)
| subterm(N-1) op(N,xf)

subterm(N) op(N,yf)

| ( subterm(1200) )  \{ term with functor ;/2 unless | is declared infix \}

| \{ subterm(1200) \}
| list
| string
| constant
\[
\begin{align*}
\text{op}(N,T) & \quad ::= \text{name} \quad \{ \text{where name has been declared as an operator of type } T \text{ and precedence } N \} \\
\text{arguments} & \quad ::= \text{subterm}(999) \\
& \quad \mid \text{subterm}(999) \; , \; \text{arguments} \\
\text{list} & \quad ::= \emptyset \\
& \quad \mid [ \; \text{listexpr} \; ] \\
\text{listexpr} & \quad ::= \text{subterm}(999) \\
& \quad \mid \text{subterm}(999) \; , \; \text{listexpr} \\
& \quad \mid \text{subterm}(999) \; | \; \text{subterm}(999) \\
\text{constant} & \quad ::= \text{atom} \mid \text{number} \\
\text{number} & \quad ::= \text{unsigned-number} \\
& \quad \mid \text{sign} \; \text{unsigned-number} \\
\text{unsigned-number} & \quad ::= \text{natural-number} \mid \text{unsigned-float} \\
\text{atom} & \quad ::= \text{name} \\
\text{functor} & \quad ::= \text{name}
\end{align*}
\]

### 4.1.7.5 Syntax of Tokens as Character Strings

SICStus Prolog supports wide characters (up to 31 bits wide), interpreted as a superset of Unicode.

Each character in the code set has to be classified as belonging to one of the character categories, such as small-letter, digit, etc. This classification is called the character-type mapping, and it is used for defining the syntax of tokens.

Only character codes 0..255, i.e. the ISO-8859-1 (Latin 1) subset of Unicode, can be part of unquoted tokens\(^1\), unless the Prolog flag \texttt{legacy\_char\_classification} is set; see Section 4.9.4 [ref-lps-flg], page 136. This restriction may be lifted in the future.

For quoted tokens, i.e. quoted atoms and strings, almost any sequence of code points assigned to non-private abstract characters in Unicode 5.0 is allowed. The disallowed characters are those in the whitespace-char category except that space (character code 32) is allowed despite it being a whitespace-char.

An additional restriction is that the sequence of characters that makes up a quoted token must be in Normal Form C (NFC) \url{http://www.unicode.org/reports/tr15/}. This is currently not enforced. A future release may enforce this restriction or perform this normalization automatically.

NFC is the normalization form used on the web (\url{http://www.w3.org/TR/charmmod/}) and what most software can be expected to produce by default. Any sequence consisting of only characters from Latin 1 is already in NFC.

\(^1\) Characters outside this range can still be included in quoted atoms and strings by using escape sequences (see Section 4.1.7.6 [ref-syn-syn-esc], page 60).
When the Prolog flag `legacy_char_classification` is set, characters in the whitespace-char category are still treated as whitespace but other character codes outside the range 0..255, assigned to non-private abstract characters in Unicode 5.0, are treated as lower case. Such characters can therefore appear as themselves, without using escape sequences, both in quoted and unquoted tokens.

Note: Any output produced by `write_term/2` with the option `quoted(true)` will be in NFC. This includes output from `writeq/[1,2]` and `write_canonical/[1,2].`

### whitespace-char

These are character codes 0..32, 127..160, 8206..8207, and 8232..8233. This includes ASCII characters such as `TAB`, `LFD`, and `SPC`, as well as all characters with Unicode property “Pattern_Whitespace” including the Unicode-specific `LINE_SEPARATOR` (8232).

### small-letter

These are character codes 97..122, i.e. the letters ‘a’ through ‘z’, as well as the non-ASCII character codes 170, 186, 223..246, and 248..255.

If the Prolog flag `legacy_char_classification` (see Section 4.9.4 [ref-lps-flg], page 136) is set, then the `small-letter` set will also include almost every code point above 255 assigned to non-private abstract characters in Unicode 5.0.

### capital-letter

These are character codes 65..90, i.e. the letters ‘A’ through ‘Z’, as well as the non-ASCII character codes 192..214, and 216..222.

### digit

These are character codes 48..57, i.e. the digits ‘0’ through ‘9’.

### symbol-char

These are character codes 35, 36, 38, 42, 43, 45..47, 58, 60..64, 92, 94, and 126, i.e. the characters:

```
+ - * / \ ^ < > = : . ? @ # $ &
```

In addition, the non-ASCII character codes 161..169, 171..185, 187..191, 215, and 247 belong to this character type.

### solo-char

These are character codes 33 and 59 i.e. the characters ‘!’ and ‘;’.

### punctuation-char

These are character codes 37, 40, 41, 44, 91, 93, and 123..125, i.e. the characters:

```
% ( ) , [ ] { | }
```

### quote-char

These are character codes 34 and 39 i.e. the characters ‘”’ and ‘‘’.

### underline

This is character code 95 i.e. the character ‘_’.

Other characters are unclassified and may only appear in comments and to some extent, as discussed above, in quoted atoms and strings.

---

2 In release 3 and 4.0.0 the lower case characters 170 and 186 were incorrectly classified as symbol-char. This was corrected in release 4.0.1.
token ::= name  
| natural-number  
| unsigned-float  
| variable  
| string  
| punctuation-char  
| whitespace-text  
| full stop  

name ::= quoted-name  
| word  
| symbol  
| solo-char  
| [ ?whitespace-text ]  
| { ?whitespace-text }  

word ::= small-letter ?alpha...  

symbol ::= symbol-char...  

natural-number ::= digit...  
| base-prefix alpha...  

| 0 ' char-item  

{ except in the case of a full stop or where the first 2 chars are ‘/*’ }  

{ where each alpha must be digits of the base indicated by base-prefix, treating a,b,... and A,B,... as 10,11,... }  

unsigned-float ::= simple-float  
| simple-float exp exponent  

simple-float ::= digit... digit...  

exp ::= e | E  

exponent ::= digit... | sign digit...  

sign ::= - | +  

variable ::= underline ?alpha...  
| capital-letter ?alpha...  

string ::= " ?string-item... "  

string-item ::= quoted-char  
| ""  

| \ escape-sequence  

quoted-atom ::= ' ?quoted-item... '  

quoted-item ::= quoted-char  
| ' '  

| \ escape-sequence  

whitespace-text ::= whitespace-text-item...  

whitespace-text-item ::= whitespace-char | comment  

comment ::= */ ?char... */  

{ where ?char... must not contain ‘*/’ }
Chapter 4: The Prolog Language

| % ?char... LFD { where ?char... must not contain LFD } |
| full stop ::= . { the following token, if any, must be whitespace-text } |

char ::= whitespace-char

printing-char ::= alpha

| symbol-char |
| solo-char |
| punctuation-char |
| quote-char |

alpha ::= capital-letter | small-letter | digit | underline

escape-sequence ::= b { backspace, character code 8 }

| t { horizontal tab, character code 9 }
| n { newline, character code 10 }
| v { vertical tab, character code 11 }
| f { form feed, character code 12 }
| r { carriage return, character code 13 }
| e { escape, character code 27 }
| d { delete, character code 127 }
| a { alarm, character code 7 }

| other-escape-sequence |

quoted-name ::= quoted-atom

base-prefix ::= 0b { indicates base 2 }

| 0o { indicates base 8 }
| 0x { indicates base 16 }

char-item ::= quoted-item

other-escape-sequence ::= x alpha...\ { treating a,b,... and A,B,... as 10,11,... in the range [0..15], hex character code }

| digit...\ { in the range [0..7], octal character code }
| LFD { ignored }
| \ { stands for itself }
| ' { stands for itself }
| " { stands for itself }
| ' { stands for itself }

quoted-char ::= SPC

| printing-char |
4.1.7.6 Escape Sequences

A backslash occurring inside integers in ‘0’ notation or inside quoted atoms or strings has special meaning, and indicates the start of an escape sequence. The following escape sequences exist:

\b backspace (character code 8)
\t horizontal tab (character code 9)
\n newline (character code 10)
\v vertical tab (character code 11)
\f form feed (character code 12)
\r carriage return (character code 13)
\e escape (character code 27)
\d delete (character code 127)
\a alarm (character code 7)
\xhex-digit...
the character code represented by the hexadecimal digits
\octal-digit...
the character code represented by the octal digits.
\LFD A backslash followed by a single newline character is ignored. The purpose of this is to allow a string or quoted-name to be spread over multiple lines.
\, \', \", \'
Stand for the character following the ‘\’.  

4.1.7.7 Notes

1. The expression of precedence 1000 (i.e. belonging to syntactic category term(1000)), which is written
   \m X, Y \n denotes the term ‘, ’(X, Y) in standard syntax.
2. The parenthesized expression (belonging to syntactic category term(0))
   (X)
   denotes simply the term X.
3. The curly-bracketed expression (belonging to syntactic category term(0))
   {X}
   denotes the term {}(X) in standard syntax.
4. Note that, for example, −3 denotes a number whereas −(3) denotes a compound term that has − /1 as its principal functor.
5. The character ‘”’ within a string must be written duplicated: ‘""’. Similarly for the character ‘’’ within a quoted atom.
6. Backslashes in strings, quoted atoms, and integers written in ‘0’ notation denote escape sequences.

7. A name token declared to be a prefix operator will be treated as an atom only if no term-read-in can be read by treating it as a prefix operator.

8. A name token declared to be both an infix and a postfix operator will be treated as a postfix operator only if no term-read-in can be read by treating it as an infix operator.

9. The whitespace following the full stop is not considered part of the full stop, and so it remains in the input stream.

4.1.8 Summary of Predicates
Detailed information is found in the reference pages for the following:

\[
\text{current_op}(P, T, A) \quad IS0
\]
\[
\text{atom } A \text{ is an operator of type } T \text{ with precedence } P
\]

\[
\text{op}(P, T, A) \quad IS0
\]
\[
\text{make atom } A \text{ an operator of type } T \text{ with precedence } P
\]

4.2 Semantics
This section gives an informal description of the semantics of SICStus Prolog.

4.2.1 Programs
A fundamental unit of a logic program is the goal or procedure call for example:

\[
gives(tom, apple, teacher)
\]

\[
reverse([1,2,3], L)
\]

\[
X < Y
\]

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The principal functor of a goal is called a predicate. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic program consists simply of a sequence of statements called sentences, which are analogous to sentences in natural language.

A sentence comprises a head and a body. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (it may be empty). If the head is not empty, then the sentence is called a clause.

If the body of a clause is empty, then the clause is called a unit clause, and is written in the form (A) where \( P \) is the head goal. We interpret this declaratively as (B) and procedurally as (C).
If the body of a clause is non-empty, then the clause is called a *non-unit clause*, and is written in the form (D) where \( P \) is the head goal and \( Q, R, \) and \( S \) are the goals that make up the body. We can read such a clause either declaratively as (E) or procedurally as (F).

\[
P :\!-\! Q, R, S. \quad \text{(D)}
\]

“\( P \) is true if \( Q \) and \( R \) and \( S \) are true.” \hspace{1cm} \text{(E)}

“To satisfy goal \( P \), satisfy goals \( Q, R, \) and \( S \).” \hspace{1cm} \text{(F)}

A sentence with an empty head is called a *directive*, of which the most important kind is called a *query* and is written in the form (G). Such a query is read declaratively as (H), and procedurally as (I).

\[
?- P, Q. \quad \text{(G)}
\]

“Are \( P \) and \( Q \) true?” \hspace{1cm} \text{(H)}

“Satisfy goals \( P \) and \( Q \).” \hspace{1cm} \text{(I)}

Sentences generally contain variables. A variable should be thought of as standing for some definite but unidentified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writable storage location as in most programming languages; rather it is a local name for some data object, like the variable of pure Lisp. Note that variables in different sentences are completely independent, even if they have the same name—the *lexical scope* of a variable is limited to a single sentence.

To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

\[
\text{employed}(X) :\!-\! \text{employs}(Y, X).
\]

“Any \( X \) is employed if any \( Y \) employs \( X \).”

“To find whether a person \( X \) is employed, find whether any \( Y \) employs \( X \).”

\[
\text{derivative}(X, X, 1).
\]

“For any \( X \), the derivative of \( X \) with respect to \( X \) is 1.”
“The goal of finding a derivative for the expression X with respect to X itself is satisfied by the result 1.”

?- ungulate(X), aquatic(X).
   “Is it true, for any X, that X is an ungulate and X is aquatic?”
   “Find an X that is both an ungulate and aquatic.”

In any program, the procedure for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a predicate concatenate of three arguments might well consist of the two clauses shown in (J) where concatenate(L1, L2, L3) means “the list L1 concatenated with the list L2 is the list L3”.

\[
\begin{align*}
\text{concatenate}([], L, L) & . \\
\text{concatenate}([X|L1], L2, [X|L3]) & :- \\
& \text{concatenate}(L1, L2, L3). 
\end{align*}
\]

(J)

(K)

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form Name/Arity is used, for example concatenate/3.

### 4.2.2 Types of Predicates Supplied with SICStus Prolog

Certain predicates are predefined by the Prolog system. Most of these cannot be changed or retracted. Such predicates are called built-in predicates.

Other predicates can be modified or totally redefined. These are the hook predicates used e.g. in term expansion, the foreign language interface, file name resolution, printing, message handling, and query handling.

#### 4.2.2.1 Hook Predicates

Hook predicates are called by the system. They enable you to modify SICStus Prolog’s behavior. Most of them are undefined by default. The idea of a hook predicate is that its clauses are independent of each other, and it makes sense to spread their definitions over several files (which may be written by different people). In other words, a hook predicate is typically declared to be multifile (see Section 4.3.4.1 [Multifile Declarations], page 83). Often, an application needs to combine the functionality of several software modules, among which some define clauses for such hook predicates. By simply declaring every hook predicate as multifile, the functionality of the clauses for the hook predicate is automatically combined. If this is not done, then the last software module to define clauses for a particular hook predicate will effectively supersede any clauses defined for the same hook predicate in a previous module. Most hook predicates must be defined in the user module, and usually only their first solution is relevant.

### 4.2.3 Control Structures

As we have seen, the goals in the body of a sentence are linked by the operator ‘,’ which can be interpreted as conjunction (and). The Prolog language provides a number of other operators, known as control structures, for building complex goals. Apart from being built-in predicates, these control structures play a special role in certain language features, namely
Grammar Rules (see Section 4.14 [ref-gru], page 189), and when code is loaded or asserted in the context of modules (see Section 4.11 [ref-mod], page 161). The set of control structures is described in this section, and consists of:

\[ P, Q \] ISO
prove \( P \) and \( Q \)

\[ P; Q \] ISO
prove \( P \) or \( Q \)

\[ M: P \] ISO
call \( P \) in module \( M \)

\[ P->:Q;:R \] ISO
if \( P \) succeeds, then prove \( Q \); if not, then prove \( R \)

\[ P->:Q \] ISO
if \( P \) succeeds, then prove \( Q \); if not, then fail

\[ ! \] ISO
cut any choices taken in the current procedure

\[ \! + :P \] ISO
goal \( P \) is not provable

\[ ?X ^ :P \] ISO
there exists an \( X \) such that \( P \) is provable (used in setof/3 and bagof/3)

\[ +\text{Iterators do :Body} \] ISO
executes \( \text{Body} \) iteratively according to \( \text{Iterators} \)

\[ \text{if}(:P,:Q,:R) \] ISO
for each solution of \( P \) succeeds, prove \( Q \); if none, then prove \( R \)

\[ \text{once}(:P) \] ISO
Find the first solution, if any, of goal \( P \).

4.2.3.1 The Cut
Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the cut, written ‘!’. It is inserted in the program just like a goal, but is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned.

The effect of the cut is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, then the effect is to fail the parent goal, i.e. the goal that matched the head of the clause containing the cut, and caused the clause to be activated. In other words, the cut operation commits the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded. The goals thus rendered determinate are the parent goal itself, any goals occurring before the cut in the clause containing the cut, and any subgoals that were executed during the execution of those preceding goals.

For example, the procedure
can be used to test whether a given term is in a list:

```
?- member(b, [a,b,c]).
```

returns the answer ‘yes’. The procedure can also be used to extract elements from a list, as in

```
?- member(X, [d,e,f]).
```

With backtracking this will successively return each element of the list. Now suppose that the first clause had been written instead:

```
member(X, [X|L]) :- !.
```

In this case, the second call above would extract only the first element of the list (‘d’). On backtracking, the cut would immediately fail the entire procedure.

Another example:

```
x :- p, !, q.
x :- r.
```

This is analogous to “if p then q else r” in an Algol-like language.

Note that a cut discards all the alternatives subsequent to the parent goal, even when the cut appears within a disjunction. This means that the normal method for eliminating a disjunction—by defining an extra predicate—cannot be applied to a disjunction containing a cut.

A proper use of the cut is usually a major difficulty for new Prolog programmers. The usual mistakes are to over-use cut, and to let cuts destroy the logic. A cut that does not destroy the logic is called a green cut; a cut that does is called a red cut. We would like to advise all users to follow these general rules. Also see Chapter 9 [Writing Efficient Programs], page 355.

- Write each clause as a self-contained logic rule, which just defines the truth of goals that match its head. Then add cuts to remove any fruitless alternative computation paths that may tie up memory.
- Cuts are hardly ever needed in the last clause of a predicate.
- Use cuts sparingly, and only at proper places. A cut should be placed at the exact point that it is known that the current choice is the correct one; no sooner, no later, usually placed right after the head, sometimes preceded by simple tests.
- Make cuts as local in their effect as possible. If a predicate is intended to be determinate, then define it as such; do not rely on its callers to prevent unintended backtracking.
- Binding output arguments before a cut is a common source of programming errors. If
a predicate is not steadfast, then it is usually for this reason.

To illustrate the last issue, suppose that you want to write a predicate \texttt{max/3} that finds the greater of two numbers. The pure version is:

\begin{verbatim}
max(X, Y, X) :- X \geq Y.
max(X, Y, Y) :- X < Y.
\end{verbatim}

Now since the two conditions are mutually exclusive, we can add a green cut to the first clause:

\begin{verbatim}
max(X, Y, X) :- X \geq Y, !.
max(X, Y, Y) :- X < Y.
\end{verbatim}

Furthermore, if the \( X \geq Y \) test fails, then we know that \( X < Y \) must be true, and therefore it is tempting to turn the green cut into a red one and drop the \( X < Y \) test:

\begin{verbatim}
max(X, Y, X) :- X \geq Y, !.
max(X, Y, Y).
\end{verbatim}

Unfortunately, this version of \texttt{max/3} can give wrong answers, for example:

\begin{verbatim}
| ?- max(10, 0, 0). yes
\end{verbatim}

The reason is that the query does not match the head of the first clause, and so we never executed the \( X \geq Y \) test. When we dropped the \( X < Y \) test, we made the mistake of assuming that the head of the first clause would match any query. This is an example of a predicate that is \textit{not steadfast}. A steadfast version is:

\begin{verbatim}
max(X, Y, Z) :- X \geq Y, !, Z = X.
max(X, Y, Y).
\end{verbatim}

\subsection{Disjunction}

It is sometimes convenient to use an additional operator ‘;’, standing for disjunction (or). (The precedence of ‘;’ is such that it dominates ‘,’ but is dominated by ‘:-’.) An example is the clause (A), which can be read as (B).

\begin{verbatim}
grandfather(X, Z) :-
    ( mother(X, Y) ; father(X, Y) ),
    father(Y, Z).
\end{verbatim}

\begin{verbatim}
(A)
\end{verbatim}

“For any \( X, Y, \) and \( Z, \)
\( X \) has \( Z \) as a grandfather if
either the mother of \( X \) is \( Y \)
or the father of \( X \) is \( Y, \)
and the father of Y is Z.”  \( \text{(B)} \)

Such uses of disjunction can usually be eliminated by defining an extra predicate. For instance, \( \text{(A)} \) is equivalent to \( \text{(C)} \)

\[
\begin{align*}
\text{grandfather}(X, Z) & : - \text{parent}(X, Y), \text{father}(Y, Z). \\
\text{parent}(X, Y) & : - \text{mother}(X, Y). \\
\text{parent}(X, Y) & : \text{father}(X, Y). \quad \text{(C)}
\end{align*}
\]

For historical reasons, the token ‘|’, when used outside a list, is actually an alias for ‘;’. The aliasing is performed when terms are read in. Since release 4.3, however, ‘|’ can be defined as a proper infix operator, which then disables the aliasing. So the use of ‘|’ instead of ‘;’ for disjunction is not recommended in new code.

4.2.3.3 If-Then-Else

As an alternative to the use of cuts, and as an extension to the disjunction syntax, Prolog provides the construct:

\[
( \text{If} \rightarrow \text{Then} ; \text{Else} )
\]

This is the same as the if-then-else construct in other programming languages. Procedurally, it calls the \text{If} goal, committing to it if it succeeds, then calling the \text{Then} goal, otherwise calling the \text{Else} goal. \text{Then} and \text{Else}, but not \text{If}, can produce more solutions on backtracking.

Cuts inside of \text{If} do not make much sense and are not recommended. If you do use them, then their scope is limited to \text{If} itself.

The if-then-else construct is often used in a multiple-branch version:

\[
( \begin{align*}
\text{If}_1 & \rightarrow \text{Then}_1 \\
\text{If}_2 & \rightarrow \text{Then}_2 \\
\ldots
\end{align*} ; /* otherwise \rightarrow */ \\
\text{WhenAllElseFails}
\)
\]

In contexts other than as the first argument of ;/2, the following two goals are equivalent:

\[
(\text{If} \rightarrow \text{Then})
\]

\[
(\text{If} \rightarrow \text{Then} ; \text{fail})
\]

That is, the ‘\rightarrow’ operator has nothing to do with, and should not be confused with, logical implication.

\text{once}/1 is a control construct that provides a “local cut”. That is, the following three goals are equivalent:
once(If)

(If -> true)

(If -> true ; fail)

Finally, there is another version of if-then-else of the form:

if(If,Then,Else)

which differs from (If -> Then ; Else) in that if/3 explores all solutions to If. This feature is also known as a “soft cut”. There is a small time penalty for this—if If is known to have only one solution of interest, the form (If -> Then ; Else) should be preferred.

4.2.3.4 Negation as Failure

The following construct provides a kind of pseudo-negation meaning “P is not provable”. This is not real negation (“P is false”). The following two goals are equivalent:

\+ P

(P -> fail ; true)

4.2.3.5 Do Loops

Proposed in [Schimpf 2002], the control structure

(Iterators do Body)

often eliminates the need to write an auxiliary predicate to perform some simple iteration. Semantically a do loop can be viewed as a shorthand for a goal:

PreCallGoals, aux(CallArgs).

where aux is a new, unique predicate symbol, CallArgs is its initial arguments, and PreCallGoals is a sequence of goals to be executed before calling aux, which is of the following form:

aux(BaseArgs) :- !.

aux(HeadArgs) :- PreBodyGoals, Body, aux(RecArgs).

where BaseArgs, HeadArgs and RecArgs are sequence of arguments and PreBodyGoals is a sequence of goals. Thus the semantics of a do loop is precisely defined by a set of rewrite rules from Iterators to those sequences of arguments and goals. Those rules are given in tabular form at the end of this section.

The ‘do’ operator is an infix operator of the same priority as ‘;’. It is recommended to always enclose a do loop in parentheses. Iterators is a comma-separated sequence of iterators, and Goal is any goal.

Before giving the full list of available iterators, we now show some simple examples.
The iterator `foreach(Var,List)` provides iteration over a list:

```
| ?- (foreach(X,[1,2,3]) do write(X), nl).
  1
  2
  3
yes
```

The same iterator can be used to construct a list:

```
| ?- (foreach(X,[1,2,3]), foreach(Y,List) do Y is X+3).
List = [4, 5, 6]
```

The iterator `fromto(First,In,Out,Last)` can be used to express an accumulator with initial value `First`, final value `Last`, with `In` and `Out` being local variables in `Body`:

```
| ?- (foreach(X,[1,2,3]), fromto(0,In,Out,Sum) do Out is In+X).
  Sum = 6
```

The iterator `for(Var,Min,Max)` will iterate `Body` with `Var` ranging over integers `Min` thru `Max`, which can be expressions:

```
| ?- (for(I,1,5), foreach(I,List) do true).
List = [1,2,3,4,5]
```

The iterator `count(Var,Min,Max)` will iterate `Body` with `Var` ranging over ascending integers from `Min`, unifying `Max` with the final value. Its main use is to count the number of iterations:

```
| ?- (foreach(X,[a,b,c,d,e]), count(I,1,N), foreach(I-X,Pairs) do true).
  N = 5,
  Pairs = [1-a,2-b,3-c,4-d,5-e]
```

The iterator `foreacharg(Var,Struct)` provides iteration over the arguments of a structure. The variant `foreacharg(Var,Struct,I)` also exists, with `I` ranging over the argument number, 1-based:

```
| ?- (foreacharg(A,f(1,2,3)), foreach(A,List) do true).
List = [1,2,3]
| ?- (foreacharg(A,f(a,b,c,d,e),I), foreach(I-A,List) do true).
List = [1-a,2-b,3-c,4-d,5-e]
```

Do loops have special variable scoping rules, which sometimes contradict the default rule that the scope of a variable is the clause in which it occurs: the scope of variables occurring in `Body` as well as variables quantified by iterators is one loop iteration. The exact scope of variables is given in the table below. To override the scoping rule, i.e. to enable a variable to be passed to all loop iterations, use the `param(Var)` declaration:
An omitted param(Var) iterator is often spotted by the compiler, which issues a warning. Suppose that we want to define a predicate that removes all occurrences of the element Kill from the list List giving Residue. A do loop formulation is given below, along with a buggy version where param(Kill) is missing:

\[
\text{delete1}(\text{List}, \text{Kill}, \text{Residue}) \leftarrow \% \text{ correct }
(\text{foreach}(X, \text{List}),
\text{fromto}(\text{Residue}, S0, S, []),
\text{param}(\text{Kill})
\text{do } (X = \text{Kill} \rightarrow S0 = S ; S0 = [X|S])
).
\]

\[
\text{delete2}(\text{List}, \text{Kill}, \text{Residue}) \leftarrow \% \text{ wrong }
(\text{foreach}(X, \text{List}),
\text{fromto}(\text{Residue}, S0, S, [])
\text{do } (X = \text{Kill} \rightarrow S0 = S ; S0 = [X|S])
).
\]

The compiler warns about the missing param(Kill), and for a good reason: the first version works as intended, but the second does not:

\[
\text{?- delete1([1,2,3,4,5], 3, R).}
\]
\[
R = [1,2,4,5]
\]

\[
\text{?- delete2([1,2,3,4,5], 3, R).}
\]
\[
R = []
\]

In some cases it is harmless and even useful to use the same local variable name in two do-loops. The compiler tries to detect this and will suppress warnings in these cases:
% do_scope.pl

sums_and_squares(List, Doubles, Squares) :-
  % X is used as a local variable in both do-loops. This is
  % harmless and will give no warning.
  ( foreach(X, List),
    foreach(Double, Doubles)
    do
      Double is X+X
  ),
  ( foreach(X, List),
    foreach(Square, Squares)
    do
      Square is X*X
  ).

Please note: In the context of multiple iterators, for the loop to terminate, all termination
conditions must hold simultaneously. For example:

```
| ?- (for(I,1,2), for(J,1,3) do writeq(I-J), nl).
1-1
2-2
3-3
4-4
...
```

will not terminate, because the two termination condition never hold simultaneously.

Finally, do loops can be used as a control structure in grammar rules as well. A do loop in
a grammar rule context will generate (or parse) the concatenation of the lists of symbols
generated (or parsed) by each loop iteration. For example, suppose that you are representing
three-dimensional points as lists \([x, y, z]\). Suppose that you need to generate a list of all
such points for x between 1 and Length, y between 1 and Width, and z between 1 and Height. A generator of such lists can be written as a grammar rule with nested do loops as follows.
| ?- compile(user).
| points3d(Length, Width, Height) -->
|   (   for(X,1,Length),
|       param(Width,Height)
|   do (   for(Y,1,Width),
|       param(X,Height)
|   do (   for(Z,1,Height),
|       param(X,Y)
|   do [[X,Y,Z]]
|   )
| )
| )
| ?- ^D
% compiled user in module user, 0 msec 1024 bytes
| ?- phrase(points3d(3,2,4), S).
S = [[[1,1,1],[1,1,2],[1,1,3],[1,1,4],
    [1,2,1],[1,2,2],[1,2,3],[1,2,4],
    [2,1,1],[2,1,2],[2,1,3],[2,1,4],
    [2,2,1],[2,2,2],[2,2,3],[2,2,4],
    [3,1,1],[3,1,2],[3,1,3],[3,1,4],
    [3,2,1],[3,2,2],[3,2,3],[3,2,4]]

We now summarize the available iterators. In this table, the phrase “var is a local variable” means that var should occur in Goal and is a brand new variable in each iteration. All other variables have global scope, i.e. the scope is the clause containing the do loop.

**fromto**(First, In, Out, Last)

In the first iteration, In = First. In the n:th iteration, In is the value that Out had at the end of the (n-1):th iteration. In and Out are local variables. The termination condition is Out = Last.

**foreach**(X, List)

Iterate with X ranging over all elements of List. X is a local variable. Can also be used for constructing a list. The termination condition is Tail = [], where Tail is the suffix of List that follows the elements that have been iterated over.

**foreacharg**(X, Struct)

with X ranging over all arguments of Struct. X is a local variable. Cannot be used for constructing a term. So the termination condition is true iff all arguments have been iterated over.

**foreacharg**(X, Struct, Idx)

Iterate with X ranging over all arguments of Struct and Idx ranging over the argument number, 1-based. X and Idx are local variables. Cannot be used for constructing a term. So the termination condition is true iff all arguments have been iterated over.

**for**(I, MinExpr, MaxExpr)

This is used when the number of iterations is known. Let Min take the value integer(MinExpr), let Max take the value integer(MaxExpr), and let Past
take the value \( \max(Min, Max+1) \). Iterate with \( I \) ranging over integers from \( Min \) to \( \max(Min, Max) \) inclusive. \( I \) is a local variable. The termination condition is \( I = Past \).

\( \text{count}(I, MinExpr, Max) \)
This is normally used for counting the number of iterations. Let \( Min \) take the value \( \text{integer}(MinExpr) \). Iterate with \( I \) ranging over integers from \( Min \). \( I \) is a local variable. The termination condition is \( I = Max \), i.e. \( Max \) can be and typically is a variable.

\( \text{param}(Var) \)
For declaring variables global, i.e. shared with the context. \( Var \) can be a single variable, or a list of them. The termination condition is true. **Please note:** By default, variables have local scope.

**IterSpec1, IterSpec2**
The iterators are iterated synchronously; that is, they all take their first value for the first iteration, their second value for the second iteration, etc. The order in which they are written does not matter. The set of local variables is the union of those of the iterators. The termination condition is the conjunction of those of the iterators.

Finally, we present the set of rewrite rules for the conceptual \( aux \) predicate that was introduced above. The rules define the translation from the iterators to the previously introduced \( PreCallGoals, CallArgs, BaseArgs, HeadArgs, PreBodyGoals, \) and \( RecArgs \). This defines the precise semantics of any do loop:

<table>
<thead>
<tr>
<th>iterator</th>
<th>PreCallGoals</th>
<th>PreBodyGoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromto(F,I0,I1,T)</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>foreach(X,L)</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>foreacharg(A,S)</td>
<td>functor(S,_,N), N is N+1</td>
<td>I1 is I0+1, arg(I0,S,A)</td>
</tr>
<tr>
<td>foreacharg(A,S,I1)</td>
<td>functor(S,_,N), N is N+1</td>
<td>I1 is I0+1, arg(I0,S,A)</td>
</tr>
<tr>
<td>count(I,FE,T)</td>
<td>F is integer(FE)-1</td>
<td>I is I0+1</td>
</tr>
<tr>
<td>for(I,FE,TE)</td>
<td>F is integer(FE),</td>
<td>I1 is I+1</td>
</tr>
<tr>
<td></td>
<td>S is max(F,integer(TE)+1)</td>
<td></td>
</tr>
<tr>
<td>param(P)</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
4.2.3.6 Other Control Structures

The “all solution” predicates recognize the following construct as meaning “there exists an \( X \) such that \( P \) is true”, and treats it as equivalent to \( P \). The use of this explicit existential quantifier outside the \texttt{setof/3} and \texttt{bagof/3} constructs is superfluous and discouraged. Thus, the following two goals are equivalent:

\[
X.\neg P
\]
\[
P
\]

The following construct is meaningful in the context of modules (see Section 4.11 \[ref-mod\], page 161), meaning “\( P \) is true in the context of the \( M \) module”:

\[
M.\neg P
\]

4.2.4 Declarative and Procedural Semantics

The semantics of definite clauses should be fairly clear from the informal interpretations already given. However, it is useful to have a precise definition. The \textit{declarative semantics} of definite clauses tells us which goals can be considered true according to a given program, and is defined recursively as follows:

A goal is \textit{true} if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true, where an \textit{instance} of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

For example, if a program contains the procedure for \texttt{concatenate/3}, declared in Section 4.2.1 \[ref-sem-pro\], page 61, then the declarative semantics tells us that (A) is true, because this goal is the head of a certain instance of the second clause (K) for \texttt{concatenate/3}, namely (B), and we know that the only goal in the body of this clause instance is true, because it is an instance of the unit clause that is the first clause for \texttt{concatenate/3}.

\[
\text{concatenate([a], [b], [a,b])}
\]
\[
\text{concatenate([a], [b], [a,b]):-}
\]
\[
\text{concatenate([], [b], [b]).}
\]
Note that the declarative semantics makes no reference to the sequencing of goals within
the body of a clause, nor to the sequencing of clauses within a program. This sequencing
information is, however, very relevant for the procedural semantics that Prolog gives to defi-
nite clauses. The procedural semantics defines exactly how the Prolog system will execute a
goal, and the sequencing information is the means by which the Prolog programmer directs
the system to execute his program in a sensible way. The effect of executing a goal is to
e numerate, one by one, its true instances. Here is an informal definition of the procedural
semantics.

To *execute* a goal, the system searches forwards from the beginning of the
program for the first clause whose head *matches or unifies* with the goal. The
unification process [Robinson 65] finds the most general common instance of the
two terms, which is unique if it exists. If a match is found, then the matching
clause instance is then *activated* by executing in turn, from left to right, each of
the goals (if any) in its body. If at any time the system fails to find a match for
a goal, then it *backtracks*; that is, it rejects the most recently activated clause,
undoing any substitutions made by the match with the head of the clause. Next
it reconsider s the original goal that activated the rejected clause, and tries to
find a subsequent clause that also matches the goal.

For example, if we execute the goal expressed by the query (A), then we find that it matches
the head of the second clause for `concatenate/3`, with \( X \) instantiated to \([a|X1]\). The new
variable \( X1 \) is constrained by the new goal produced, which is the recursive procedure call
(B) and this goal matches the second clause, instantiating \( X1 \) to \([b|X2]\), and yielding the
new goal (C).

\[
\text{(A)} \quad \text{| ?- concatenate}(X, Y, [a,b]).
\]

\[
\text{concatenate}(X1, Y, [b]) \quad \text{(B)}
\]

\[
\text{concatenate}(X2, Y, []) \quad \text{(C)}
\]

Now this goal will only match the first clause, instantiating both \( X2 \) and \( Y \) to \([\]. Since
there are no further goals to be executed, we have a solution

\[
X = [a,b] \\
Y = []
\]

That is, the following is a true instance of the original goal:

\[
\text{concatenate}([a,b], [], [a,b])
\]

If this solution is rejected, then backtracking will generate the further solutions

\[
X = [a] \\
Y = [b]
\]

\[
X = [] \\
Y = [a,b]
\]
in that order, by re-matching goals already solved once using the first clause of \texttt{concatenate/3}, against the second clause.

Thus, in the procedural semantics, the set of clauses

\[
H := B_1, \ldots, B_m. \\
H' := B'_1, \ldots, B'_m. \\
\ldots
\]

are regarded as a \textit{procedure definition} for some predicate \(H\), and in a query

\[
?- G_1, \ldots, G_n.
\]

each \(G_i\) is regarded as a \textit{procedure call}. To execute a query, the system selects by its \textit{computation rule} a goal, \(G_j\) say, and searches by its \textit{search rule} a clause whose head matches \(G_j\). Matching is done by the \textit{unification} algorithm (see [Robinson 65]), which computes the most general unifier, \(mgu\), of \(G_j\) and \(H\). The \(mgu\) is unique if it exists. If a match is found, then the current query is \textit{reduced} to a new query

\[
?- (G_1, \ldots, G_{j-1}, B_1, \ldots, B_m, G_{j+1}, \ldots, G_n)_{mgu}.
\]

and a new cycle is started. The execution terminates when the empty query has been produced.

If there is no matching head for a goal, then the execution backtracks to the most recent successful match in an attempt to find an alternative match. If such a match is found, then an alternative new query is produced, and a new cycle is started.

In SICStus Prolog, as in other Prolog systems, the search rule is simple: “search forward from the beginning of the program”.

The computation rule in traditional Prolog systems is also simple: “pick the leftmost goal of the current query”. However, SICStus Prolog and other modern implementations have a somewhat more complex computation rule “pick the leftmost unblocked goal of the current query”.

A goal can be blocked on one or more uninstantiated variables, and a variable may block several goals. Thus binding a variable can cause blocked goals to become unblocked, and backtracking can cause currently unblocked goals to become blocked again. Moreover, if the current query is

\[
?- G_1, \ldots, G_{j-1}, G_j, G_{j+1}, \ldots, G_n.
\]

where \(G_j\) is the first unblocked goal, and matching \(G_j\) against a clause head causes several blocked goals in \(G_1, \ldots, G_{j-1}\) to become unblocked, then these goals may become reordered.

Another consequence is that a query may be derived consisting entirely of blocked goals. Such a query is said to have \textit{floundered}. The top level checks for this condition. If detected, then the outstanding blocked subgoals are printed on the standard error stream along
with the answer substitution, to notify the user that the answer (s)he has got is really a speculative one, since it is only valid if the blocked goals can be satisfied.

A goal is blocked if certain arguments are uninstantiated and its predicate definition is annotated with a matching block declaration (see Section 4.3.4.5 [Block Declarations], page 84). Goals of certain built-in predicates may also be blocked if their arguments are not sufficiently instantiated.

When this mechanism is used, the control structure resembles that of coroutines, suspending and resuming different threads of control. When a computation has left blocked goals behind, the situation is analogous to spawning a new suspended thread.

When a blocked goal becomes unblocked, the situation is analogous to temporarily suspending the current thread and resuming the thread to which the blocked goal belongs. Please note that a stretch of unifications is treated as atomic with respect to unblocking goals. Also, if such a stretch is followed by an explicit failure (\( \text{fail/0, false/0} \)), any unblocked goal is not run.

### 4.2.5 Meta-Calls

If \( X \) is instantiated to a term that would be acceptable as the body of a clause, then the goal \( \text{call}(X) \) is executed exactly as if that term appeared textually in place of the \( \text{call}(X) \), except the scope of any cut occurring in \( X \) is limited to the execution of \( X \). That is, the cut does not “propagate” into the clause in which \( \text{call}(X) \) occurs. If \( X \) is not properly instantiated, then an error exception is raised as described in Section 4.2.6 [ref-sem-exc], page 77.

In release 4, \( \text{call/1} \) has been generalized to \( \text{call/N} \) of any arity \( N \) between 1 and 255: the first argument is treated as template, which should be augmented by the remaining arguments, giving the goal to call. For example, the goal \( \text{call}(p(X),Y,Z) \) is equivalent to the goal \( p(X,Y,Z) \). Note in particular that the first argument does not need to be an atom.

### 4.2.6 Exceptions Related to Procedure Calls

All predicates that take a call argument will raise the following exceptions:

- **instantiation_error**
  
  Module prefix or goal uninstantiated.

- **type_error**
  
  Goal not a callable.

- **existence_error**
  
  Procedure does not exist.

- **context_error**
  
  Declaration or clause construct called as procedure.

The reference page for such predicates will simply refer to these as “Call errors” and will go on to detail other exceptions that may be raised for a particular predicate.
4.2.7 Occurs Check

Prolog’s unification does not have an *occurs check*; that is, when unifying a variable against a term, the system does not check to see if the variable occurs in the term. When the variable occurs in the term, unification should fail, but the absence of the check means that the unification succeeds, producing a cyclic term. Operations such as trying to print a cyclic term will cause a loop.

The absence of the occurs check is not a bug or a design oversight, but a conscious design decision. The reason for this decision is that unification of a variable and a term with the occurs check is at best linear in the sum of the term, whereas such unification without the occurs check runs in constant time. For any programming language to be practical, basic operations should take constant time. Unification against a variable may be thought of as the basic operation of Prolog, and this can take constant time only if the occurs check is omitted. Thus the absence of an occurs check is essential to Prolog’s practicality as a programming language. The inconvenience caused by this restriction is, in practice, very slight.

SICStus Prolog unifies, compares (see Section 4.8.8 [ref-lte-cte], page 130), asserts, and copies cyclic terms without looping. The `write_term/[2,3]` built-in predicate can optionally handle cyclic terms. Unification with occurs check is available as a built-in predicate; see Section 4.8.1.2 [ref-lte-met-usu], page 126. The `acyclic_term/1` built-in predicate can test whether a term is acyclic; `subsumes_term/2` can test whether a term is subsumed by another one (see Section 4.8 [ref-lte], page 126). Additional predicates for subsumption and testing (a)cyclicity are available in a library package; see Section 10.46 [lib-terms], page 862. Other predicates usually do not handle cyclic terms well.

### 4.2.8 Summary of Control Predicates

\[ :P, :Q \]
prove \( P \) and \( Q \)

\[ :P; :Q \]
prove \( P \) or \( Q \)

\[ +M: :P \]
call \( P \) in module \( M \)

\[ :P->:Q; :R \]
if \( P \) succeeds, prove \( Q \); if not, prove \( R \)

\[ :P->:Q \]
if \( P \) succeeds, prove \( Q \); if not, fail

\[ ! \]
cut any choices taken in the current procedure

\[ \setminus + :P \]
goal \( P \) is not provable

\[ ?X \setminus ^ :P \]
there exists an \( X \) such that \( P \) is provable (used in `setof/3` and `bagof/3`)
Chapter 4: The Prolog Language

4.3 Loading Programs

4.3.1 Overview

There are two ways of loading programs into Prolog—loading source files and loading precompiled PO files. Source files can be compiled into virtual machine code, as well as consulted for interpretation. Dynamic predicates are always stored in interpreted form, however.
Virtual machine code runs about 8 times faster than interpreted code, and requires less runtime storage. Compiled code is fully debuggable, except certain constructs compile inline and cannot be traced. Compiled code also provides better precision for execution profiling and coverage analysis.

Since release 4.3, on 32 and 64 bit x86 platforms running Windows, OS X, and Linux, SICStus Prolog has the ability to compile predicates from virtual machine code to native code. This process, known as Just In Time (JIT) compilation, is controlled by a couple of system properties (see Section 4.17.1 [System Properties and Environment Variables], page 224), but is otherwise automatic. JIT compilation is seamless wrt. debugging, profiling, coverage analysis, etc. JIT compiled code runs up to 4 times faster than virtual machine code, but takes more space.

The virtual machine compiler operates in two different modes, controlled by the compiling Prolog flag. The possible values of the flag are:

- **compactcode**: Compilation produces byte-coded abstract instructions. The default.
- **debugcode**: Compilation produces interpreted code, i.e. compiling is replaced by consulting.

This section contains references to the use of the module system. These can be ignored if the module system is not being used (see Section 4.11 [ref-mod], page 161, for information on the module system).

### 4.3.2 The Load Predicates

Loading a program is accomplished by one of these predicates

```prolog
[] 
[:File|:Files] 
load_files(:Files) 
load_files(:Files, +Options) 
    loads source or PO file(s), whichever is the more recent, according to Options. 
compile(:Files) 
    loads source file(s) into virtual machine code. 
consult(:Files) 
reconsult(:Files) 
    loads source file(s) into interpreted representation. 
ensure_loaded(:Files) 
    loads source or PO file(s), whichever is the more recent, unless the file has already been loaded and it has not been modified since it was loaded. 
use_module(:Files) 
use_module(:File,+I) 
use_module(?M,:File,+I) 
    loads module files, see Section 4.11.3 [ref-mod-def], page 162, for information about module files. 
```
The following notes apply to all the Load Predicates:

1. The *File* argument must be one of the following:
   - a file specification, that is the name of a file containing Prolog code; a ‘.pro’, ‘.pl’ or a ‘.po’ suffix to a filename may be omitted (see Section 4.5.1 [ref-fdi-fsp], page 95)
   - the atom *user*

2. The *Files* argument can be a *File*, as above, or a list of such elements.

3. These predicates resolve file specifications in the same way as *absolute_file_name/2*. For information on file names refer to Section 4.5 [ref-fdi], page 95.

4. The above predicates raise an exception if any of the files named in *Files* does not exist, unless the fileerrors flag is set to off.

   Errors detected during compilation, such as an attempt to redefine a built-in predicate, also cause exceptions to be raised. However, these exceptions are caught by the compiler, an appropriate error message is printed, and compilation continues.

5. There are a number of *style warnings* that may appear when a file is compiled. These are designed to aid in catching simple errors in your programs and are initially on, but can be turned off if desired by setting the appropriate flags, which are:

   **single_var_warnings**

   If on, then warnings are printed when a sentence (see Section 4.1.7.3 [ref-syn-syn-sen], page 53) containing variables not beginning with ‘_’ occurring once only is compiled or consulted.

   The Prolog flag legacy_char_classification (see Section 4.9.4 [ref-lps-flg], page 136) expands the set of variable names for which warnings are printed. When legacy_char_classification is in effect warnings are printed also for variables that occur only once and whose name begin with ‘_’ followed by a character that is not an uppercase Latin 1 character.

   **redefine_warnings**

   This flag can take more values; see Section 4.9.4 [ref-lps-flg], page 136. If on, then the user is asked what to do when:
   - a module or predicate is being redefined from a different file than its previous definition.
   - a predicate is being imported whilst it was locally defined already.
   - a predicate is being redefined locally whilst it was imported already.
   - a predicate is being imported whilst it was imported from another module already.

   **discontiguous_warnings**

   If on, then warnings are printed when clauses are not together in source files, and the relevant predicate has not been declared discontiguous.

6. By default, all clauses for a predicate are required to come from just one file. A predicate must be declared multifile if its clauses are to be spread across several different files. See the reference page for multifile/1.
7. If a file being loaded is not a module file, then all the predicates defined in the file are loaded into the source module. The form `load_files(Module:Files)` can be used to load the file into the specified module. See Section 4.11.3 [ref-mod-def], page 162, for information about module files. If a file being loaded is a module file, then it is first loaded in the normal way, the source module imports all the public predicates of the module file except for `use_module/[1,2,3]` and `load_files/[1,2]` if you specify an import list.

8. If there are any directives in the file being loaded, that is, any terms with principal functor `:-/1` or `?-/1`, then these are executed as they are encountered. Only the first solution of directives is produced, and variable bindings are not displayed. Directives that fail or raise exceptions give rise to warning or error messages, but do not terminate the load. However, these warning or error messages can be intercepted by the hook `user:portray_message/2`, which can call `abort/0` to terminate the load, if that is the desired behavior.

9. A common type of directive to have in a file is one that loads another file, such as `:- [otherfile].`

   In this case, if `otherfile` is a relative filename, then it is resolved with respect to the directory containing the file that is being loaded, not the current working directory of the Prolog system.

   Any legal Prolog goal may be included as a directive. There is no difference between a `:-/1` and a `?-/1` goal in a file being compiled.

10. If `Files` is the atom `user`, or `Files` is a list, and during loading of the list `user` is encountered, then procedures are to be typed directly into Prolog from `user_input`, e.g. the terminal. A special prompt, `| `, is displayed at the beginning of every new clause entered from the terminal. Continuation lines of clauses typed at the terminal are preceded by a prompt of five spaces. When all clauses have been typed in, the last should be followed by end-of-file, or the atom `end_of_file` followed by a full stop.

11. During loading of source code, all terms being read in are subject to term expansion. Grammar rules is a special, built-in case of this mechanism. By defining the hook predicates `user:term_expansion/6` and `goal_expansion/5`, you can specify any desired transformation to be done as clauses are loaded.

12. The current load context (module, file, stream, directory) can be queried using `prolog_load_context/2`.

13. Predicates loading source code are affected by the character-conversion mapping, cf. `char_conversion/2`.

### 4.3.3 Redefining Procedures during Program Execution

You can redefine procedures during the execution of the program, which can be very useful while debugging. The normal way to do this is to use the ‘break’ option of the debugger to enter a break state (see `break/0`, Section 4.15.7 [ref-ere-int], page 211), and then load an altered version of some procedures. If you do this, then it is advisable, after redefining the procedures and exiting from the break state, to wind the computation back to the first call to any of the procedures you are changing; you can do this by using the ‘retry’ option with an argument that is the invocation number of that call. If you do not wind the computation back like this, then:
• if you are in the middle of executing a procedure that you redefine, then you will find that the old definition of the procedure continues to be used until it exits or fails;
• if you should backtrack into a procedure you have just redefined, then alternative clauses in the old definition will still be used.

See Section 11.3.27 [mpg-ref-break], page 965.

4.3.4 Declarations and Initializations

When a program is to be loaded, it is sometimes necessary to tell the system to treat some of the predicates specially. This information is supplied by including declarations about such predicates in the source file, preceding any clauses for the predicates that they concern. A declaration is written just as a directive is, beginning with ‘:-‘. A declaration is effective from its occurrence through the end of file.

Although declarations that affect more than one predicate may be collapsed into a single declaration, the recommended style is to write the declarations for a predicate immediately before its first clause.

Operator declarations are not declarations proper, but rather directives that modify the global table of syntax operators. Operator declarations are executed as they are encountered while loading programs.

The rest of this section details the available forms of predicate declarations.

4.3.4.1 Multifile Declarations

A declaration

:- multifile :PredSpec, ..., :PredSpec. ISO

where each PredSpec is a predicate specification, causes the specified predicates to become multifile. This means that if more clauses are subsequently loaded from other files for the same predicate, then the new clauses will not replace the old ones, but will be added at the end instead.

An example where multifile declarations are particularly useful is in defining hook predicates. A hook predicate is a user-defined predicate that somehow alters or customizes the behavior of SICStus Prolog. A number of such hook predicates are described in this manual. See also Section 4.2.2.1 [ref-sem-typ-hok], page 63.

Multifile declarations must precede any other declarations for the same predicate(s)!

If a file containing clauses for a multifile predicate is reloaded, then the old clauses from the same file are removed. The new clauses are added at the end. Please note: if the file being reloaded is a module file, however, then all predicates belonging to the module are abolished, including any multifile predicate.

If a multifile predicate is loaded from a file with no multifile declaration for it, then the predicate is redefined as if it were an ordinary predicate (i.e. the user is asked for confirmation).
If a multifile predicate is declared dynamic in one file, then it must also be done so in the other files from which it is loaded. See Section 11.3.125 [mpg-ref-multifile], page 1089.

4.3.4.2 Dynamic Declarations
A declaration

\[ \text{:- dynamic :PredSpec, ..., :PredSpec.} \quad \text{ISO} \]

where each \text{PredSpec} is a predicate specification, causes the specified predicates to become dynamic, which means that other predicates may inspect and modify them, adding or deleting individual clauses. Dynamic predicates are always stored in interpreted form even if a compilation is in progress. This declaration is meaningful even if the file contains no clauses for a specified predicate—the effect is then to define a dynamic predicate with no clauses.

The semantics of dynamic code is described in Section 4.12.1 [ref-mdb-bas], page 176. See Section 11.3.68 [mpg-ref-dynamic], page 1012.

4.3.4.3 Volatile Declarations
A declaration

\[ \text{:- volatile :PredSpec, ..., :PredSpec.} \]

where each \text{PredSpec} is a predicate specification, causes the specified predicates to become \text{volatile}.

A predicate should be declared as volatile if it refers to data that cannot or should not be saved in a saved state. In most cases a volatile predicate will be dynamic, and it will be used to keep facts about streams or memory references. When a program state is saved at runtime, the clauses of all volatile predicates will be left unsaved. The predicate definitions will be saved though, which means that the predicates will keep all its properties such as \text{volatile}, \text{dynamic} or \text{multifile} when the saved state is restored. See Section 11.3.249 [mpg-ref-volatile], page 1245.

4.3.4.4 Discontiguous Declarations
By default, the development system issues warnings if it encounters clauses that are not together for some predicate. A declaration:

\[ \text{:- discontiguous :PredSpec, ..., :PredSpec.} \quad \text{ISO} \]

disables such warnings for the predicates specified by each \text{PredSpec}. The warnings can also be disabled globally by setting the \text{discontiguous_warnings} flag to \text{off}. See Section 11.3.65 [mpg-ref-discontiguous], page 1008.

4.3.4.5 Block Declarations
The declaration

\[ \text{:- block :BlockSpec, ..., :BlockSpec.} \]
where each *BlockSpec* is a skeletal goal, specifies conditions for blocking goals of the predicate referred to by the skeletal goal (*f/3* say). The arguments of the skeletal goal can be:

'-' see below

'?'

'anything else'

ignored

When a goal for *f/3* is to be executed, the mode specs are interpreted as conditions for blocking the goal, and if at least one condition evaluates to *true*, the goal is blocked.

A block condition evaluates to *true* if and only if all arguments specified as '‐' are uninstantiated, in which case the goal is blocked until at least one of those variables is instantiated. If several conditions evaluate to *true*, then the implementation picks one of them and blocks the goal accordingly.

The recommended style is to write the block declarations in front of the source code of the predicate they refer to. Indeed, they are part of the source code of the predicate, and must precede the first clause. For example, with the definition:

```
:- block merge(-,?,‐), merge(?‐,‐,‐).
```

```
merge([], Y, Y).
merge(X, [], X).
merge([H|X], [E|Y], [H|Z]) :- H @< E, merge(X, [E|Y], Z).
merge([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).
```

calls to *merge/3* having uninstantiated arguments in the first *and* third position or in the second *and* third position will suspend.

The behavior of blocking goals for a given predicate on uninstantiated arguments cannot be switched off, except by abolishing or redefining the predicate. See Section 11.3.26 [mpg-ref-block], page 963.

### 4.3.4.6 Meta-Predicate Declarations

To ensure correct semantics in the context of multiple modules, some predicates are subject to module name expansion. Clauses or directives containing goals for such predicates need to have certain arguments annotated by a module prefix. A declaration:

```
```

where each *MetaPredSpec* is a skeletal goal, informs the compiler which predicates and which of its arguments should be subject to such annotations. See Section 4.11.16 [ref-mod-met], page 171, and Section 4.11.15 [ref-mod-mne], page 171, for details.

### 4.3.4.7 Module Declarations

One of the following declarations:
:- module(+ModuleName, +ExportList).

:- module(+ModuleName, +ExportList, +Options).

where ExportList is a list of predicate specifications, declares that the forthcoming predicates should go into the module named ModuleName and that the predicates listed should be exported. See Section 4.11 [ref-mod], page 161, for details. See Section 11.3.122 [mpg-ref-meta_predicate], page 1084.

4.3.4.8 Public Declarations

The only effect of a declaration


where each PredSpec is a predicate specification, is to give the SICStus cross-referencer (see Section 9.12 [The Cross-Referencer], page 379) a starting point for tracing reachable code. In some Prologs, this declaration is necessary for making compiled predicates visible. In SICStus Prolog, predicate visibility is handled by the module system. See Section 4.11 [ref-mod], page 161.

4.3.4.9 Mode Declarations

A declaration

    :- mode :ModeSpec, ..., :ModeSpec.

where each ModeSpec is a skeletal goal, has no effect whatsoever, but is accepted for compatibility reasons. Such declarations may be used as a commenting device, as they express the programmer’s intention of data flow in predicates.

4.3.4.10 is/2 Declarations

A declaration

    :- SPEC is ANNOTATION.

is treated as a piece of documentation about the predicate specified by SPEC. It has no effect on normal execution, but the information is recorded and can be accessed by e.g. analysis and debugging tools. For details, see Section 10.19 [lib-is_directives], page 565.

4.3.4.11 Include Declarations

A directive

    :- include(+Files).

where Files is a file name or a list of file names, instructs the processor to literally embed the Prolog clauses and directives in Files into the file being loaded. This means that the effect of the include directive is as if the include directive itself were being replaced by the text in the Files. Including some files is thus different from loading them in several respects:
• The embedding file counts as the source file of the predicates loaded, e.g. with respect to the built-in predicate `source_file/2`; see Section 4.9.3 [ref-lps-apf], page 136.

• Some clauses of a predicate can come from the embedding file, and some from included files.

• When including a file twice, all the clauses in it will be entered twice into the program (although this is not very meaningful).

• The virtual clauses `beginning_of_file` and `end_of_file` are seen by term expansions for source files, but not for included files.

SICStus Prolog uses the included file name (as opposed to the embedding file name) only in source-linked debugging and error reporting. See Section 11.3.104 [mpg-ref-include], page 1060.

4.3.4.12 Initializations
A directive

    :- initialization :Goal.  ISO

in a file appends `Goal` to the list of goals that shall be executed after that file has been loaded.

`initialization/1` is actually callable at any point during loading of a file. Initializations are saved by `save_modules/2` and `save_program/[1,2]`, and so are executed after loading or restoring such files too, in input order.

`Goal` is associated with the file loaded, and with a module, if applicable. When a file, or module, is going to be reloaded, all goals earlier installed by that file, or in that module, are removed first. See Section 11.3.105 [mpg-ref-initialization], page 1061.

4.3.5 Term and Goal Expansion
During loading of source code, all terms being read in are subject to term expansion. Grammar rules is a special, built-in case of this mechanism. By defining the hook predicates `user:term_expansion/6` and `goal_expansion/5`, you can specify any desired transformation to be done as clauses are loaded.

Term expansions are added by defining clauses for the following hook predicate. Such clauses should follow the pattern:

    :- multifile user:term_expansion/6.
    user:term_expansion(Term1, Layout1, Ids, Term2, Layout2, [token|Ids]) :- ... 
    nonmember(token, Ids),
    token_expansion(Term1, Layout1, Term2, Layout2), !.

where `token_expansion/4` should be a predicate defining how to transform a given `Term1` into `Term2`. The hook is called for every `Term1` read, including at end of file, represented as the term `end_of_file`. If it succeeds, then `Term2` is used for further processing; otherwise,
the default grammar rule expansion is attempted. It is often useful to let a term expand to a list of directives and clauses, which will then be processed sequentially.

A key idea here is *Ids*, which is used to look up what expansions have already been applied. The argument is supposed to be a list of tokens, each token uniquely identifying an expansion. The tokens are arbitrary atoms, and are simply added to the input list, before expansions recursively are applied. This token list is used to avoid cyclic expansions.

The other arguments are for supporting source-linked debugging; see the reference page for details. See Section 11.3.227 [mpg-ref-term_expansion], page 1219.

**Please note:** term expansions are global, i.e. they affect all code that are compiled or consulted. In particular a term expansion is not affected by module imports. Care should be taken so that a term expansion does not unintentionally affect some unrelated source code. `goal_expansion/5` provides a more robust, and module aware, way to transform individual goals.

Goal expansions are added by defining the hook predicate:

```
M:goal_expansion(Goal1, Layout1, Module, Goal2, Layout2) :- ...
```

which should define how to transform a given `Goal1` into `Goal2`. Expansions are per module and should be defined in the module `M` in which `Goal1` is locally defined. It is called for every goal occurring in a clause being loaded, asserted, or meta-called. If it succeeds, then `Goal2` is used for further processing, and may be arbitrarily complex.

**Please note:** In general, the goal expansion can happen both at compile time and at runtime (and sometimes both, even for the same goal). For this reason the code that implements goal expansion should be present both at compile time and at runtime.

The other arguments are for supporting source-linked debugging and passing the source module; see the reference page for details.

To invoke term expansion from a program, use:

```
?- expand_term(Term1, Term2).
```

which transforms `Term1` into `Term2` using the built-in (for grammar rules) as well as user-defined term expansion rules. See Section 11.3.97 [mpg-ref-goal_expansion], page 1052.

### 4.3.6 Conditional Compilation

A pair of directives

```
:- if(:Goal).
...  
:- endif.
```

will evaluate `Goal` and, if the goal succeeds, then the sentences between the `if/1` directive and the matching `endif/0` directive will be processed as usual.
If the evaluation of *Goal* does not succeed, i.e. fails or raises an exception, then the sentences between the *if/1* directive and the *endif/0* directive are completely ignored, except that any conditional directives must be properly nested. In particular, term expansion will not be performed on such ignored sentences and the goals of any nested conditional directives will not be evaluated.

The full form of conditional compilation directives include optional *else/0* and *elif/1* and are used as follows:

```prolog
:- if(:Goal1).
... :- else.
... :- endif.
```

If the goal of the *if/1* directive succeeds, then the sentences up to the matching *else/0* directive are processed as usual. Otherwise, if the goal fails or raises an exception, then the sentences between the *else/0* directive and the matching *endif/0* directive are processed as usual.

Finally, *elif/1* is available as a shorthand for nested uses of *if/1* and *else/0*:

```prolog
:- if(:Goal1).
... :- elif(:Goal2).
... :- elif(:Goal3).
... :- else.
... :- endif.
```

will evaluate the goals in turn, until one of them succeeds in which case the following sentences will be processed as usual up to the corresponding *else/0, endif/0* or *elif/1*.

A valid sequence of conditional compilation directives must contain exactly one *if/1* directive followed by zero or more *elif/1* directives followed by at most one *else/0* directive followed by exactly one *endif/0* directive. Valid sequences of conditional directives can be nested.

All directives that make up a sequence of conditional compilation directives must be in the same file. For instance, you cannot have a *if/1* directive in one file and then have the corresponding *endif/0* directive in a file included with an *include/1* directive. Nested conditional compilation sequences can of course be located in included files.

Conditional compilation directives are handled very early in the processing of an input file. In particular, term expansion hooks will never see *if/1, else/0, elif/1* or *endif/0* directives. Also, neither of *if/1, else/0, elif/1* or *endif/0* are defined as predicates.
If evaluation of a goal for if/1 directive or an elif/1 directive raises an exception, then an error message will be written and the goal will be treated as if it failed.

### 4.3.6.1 Conditional Compilation Examples

Conditional compilation is useful for writing portable Prolog code since it makes it possible to adapt to peculiarities of various implementations. The Prolog flag `dialect`, used by several Prolog implementations, is especially useful here.

```prolog
:- if(current_prolog_flag(dialect, sicstus).
%% We are being compiled in SICStus
%% Only SICStus has this library
:- use_module(library(process), [process_create/2]).

:- elif(current_prolog_flag(dialect, othervendor)).
%% We are being compiled in Other Vendor, we need to provide our own
%% compatibility layer
:- use_module(...).
process_create(A,B) :- ...

:- else.
%% We are being compiled in some unknown Prolog, give up.
process_create(_,_) :- throw(not_implemented).

:- endif.
```

Another possible usage is for disabling, perhaps costly, debugging code when building an optimized version of the code.

```prolog
%% Only need environ/2 at compile-time for conditional compilation
:- load_files(library(system), [when(compile_time), imports([environ/2])]).

:- if(\+ environ(optimize, true)).

%% This clause does some expensive sanity checks. Disabled when building
%% an optimized version.
foo(X) :-
  \+ valid_x(X),
  throw(invalid_x(X)).

:- endif.

%% This clause is always present.
foo(X) :-
  do_x_things(X).
```
Invoking the SICStus development system with an option `-Doptimize=true`, to set the system property `optimize`, and then compiling the above code will ensure that the first, sanity checking, clause is not part of the `foo/1` predicate. Invoking the development system without such an option will ensure that the sanity checking clause is part of the `foo/1` predicate.

### 4.3.7 Predicate List

Detailed information is found in the reference pages for the following:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>same as `load_files([F</td>
</tr>
<tr>
<td>[:F</td>
<td>+Fs]</td>
</tr>
<tr>
<td>block :P</td>
<td>predicates specified by <code>P</code> should block until sufficiently instantiated</td>
</tr>
<tr>
<td>compile(:F)</td>
<td>load compiled clauses from files <code>F</code></td>
</tr>
<tr>
<td>consult(:F)</td>
<td>load interpreted clauses from files <code>F</code></td>
</tr>
<tr>
<td>reconsult(:F)</td>
<td>load interpreted clauses from files <code>F</code></td>
</tr>
<tr>
<td>expand_term(+T,-X)</td>
<td>term <code>T</code> expands to term <code>X</code> using <code>user:term_expansion/6</code> or grammar rule expansion</td>
</tr>
<tr>
<td>goal_expansion(+Term1, +Layout1, +Module, -Term2, -Layout2)</td>
<td>hook Defines transformations on goals while clauses are being compiled or asserted, and during meta-calls.</td>
</tr>
<tr>
<td>discontiguous :P</td>
<td>clauses of predicates <code>P</code> do not have to appear contiguously</td>
</tr>
<tr>
<td>dynamic :P</td>
<td>predicates specified by <code>P</code> are dynamic</td>
</tr>
<tr>
<td>elsif(:Goal)</td>
<td>provides an alternative branch in a sequence of conditional compilation directives.</td>
</tr>
<tr>
<td>else</td>
<td>provides an alternative branch in a sequence of conditional compilation directives.</td>
</tr>
<tr>
<td>endif</td>
<td>terminates a sequence of conditional compilation directives.</td>
</tr>
<tr>
<td>ensure_loaded(:F)</td>
<td>ISO load <code>F</code> if not already loaded</td>
</tr>
<tr>
<td>if(:Goal)</td>
<td>declares a sequence of conditional compilation directives for conditionally including parts of a source file.</td>
</tr>
</tbody>
</table>
include(+F) 
declaration, ISO

include the source file(s) F verbatim

initialization :G 
declaration, ISO

declares G to be run when program is started

load_files(:F) 
load_files(:F,+O) 
load files according to options O

meta_predicate :P 
declaration

declares predicates P that are dependent on the module from which they are called

mode :P 
NO-OP: document calling modes for predicates specified by P

module(+M,+L) 
declaration

module(+M,+L,+O) 
declaration

module(\textit{M},+L,+O) 
declaration

module \textit{M} exports predicates in \textit{L}, options \textit{O

multifile :P 
declaration, ISO

the clauses for \textit{P} are in more than one file

public :P 
declaration

NO-OP: declare predicates specified by \textit{P} public

restore(+F) 
restore the state saved in file \textit{F}

user:term_expansion(+Term1, +Layout1, +Tokens1, -Term2, -Layout2, -Tokens2) 
hook

Overrides or complements the standard transformations to be done by expand_term/2.

use_module(:F) 
use_module(:F,+I) 
import the procedure(s) \textit{I} from the module file \textit{F}

use_module(?\textit{M},:F,+I) 
import \textit{I} from module \textit{M}, loading module file \textit{F} if necessary

volatile :P 
declaration

predicates specified by \textit{P} are not to be included in saves

4.4 Saving and Loading the Prolog Database

4.4.1 Overview of PO Files

A PO file (Prolog object file) contains a binary representation of a set of modules, predicates, clauses and directives. They are portable between different platforms, except between 32-bit and 64-bit platforms.

PO files are created by \texttt{save_files/2}, \texttt{save_modules/2}, and \texttt{save_predicates/2}, which all save a selected set of code and data from the running application. They can be loaded by the predicates described in Section 4.3 [ref-lod], page 79.
PO files provide tremendous flexibility that can be used for many purposes, for example:

- precompiling Prolog libraries for fast loading;
- packaging Prolog code for distribution;
- generating precompiled databases of application data;
- selectively loading particular application databases (and rule bases);
- saving Prolog data across application runs;
- building and saving new application databases from within applications;

The facilities for saving and loading PO files are more than just a convenience when developing programs; they are also a powerful tool that can be used as part of the application itself.

### 4.4.2 Saved States

Saved states are just a special case of PO files. The `save_program/1,2` predicate will save the execution state in a file. The state consists of all predicates and modules except built-in predicates and clauses of volatile predicates, the current operator declarations, the current character-conversion mapping, the values of all writable Prolog flags except those marked as `volatile` in Section 4.9.4 [ref-lps-flg], page 136, any blackboard data (see Section 4.12.9 [ref-mdb-bbd], page 184), database data (see Section 4.12.1 [ref-mdb-bas], page 176), and as of release 4.2, information for source-linked debugging. See Section 11.3.198 [mpg-ref-save_program], page 1180.

A saved state, can be restored using the `restore/1` predicate from within Prolog:

```
| ?- restore(File).
```

which will replace the current program state by the one in `File`. See Section 11.3.192 [mpg-ref-restore], page 1173.

A saved state can also be given as an option to the `sicstus` command:

```
% sicstus -r File
```

which will start execution by restoring `File`.

The location (i.e. directory) of the saved state, when it is created with `save_program/1,2` and loaded with `restore/1`, is treated in a special way in order to make it possible locate files when the saved state has been moved. See Section 3.10 [Saving], page 28, for more information.

The `save_program/2` predicate can be used to specify an initial goal that will be run when the saved state is restored. For example:

```
| ?- save_program(saved_state, initial_goal([a,b,c])).
```

When `saved_state` is loaded `initial_goal/1` will be called. This allows saved states to be generated that will immediately start running the user’s program when they are restored. In
addition to this `save_program/2` facility, see also the `initialization/1` facility to declare goal to be executed upon loading (see Section 4.3.4.12 [Initializations], page 87).

4.4.3 Selective Saving and Loading of PO Files

The `save_program/[1,2]` and `restore/1` predicates discussed in the previous section are used for saving and restoring the entire Prolog database. To save selected parts of a Prolog database, the predicates `save_files/2`, `save_modules/2`, and `save_predicates/2` are used.

To save everything that was loaded from the files `src1.pl` and `src2.pl into file1.po` (extensions optional), as well as from any file included by them, you would use:

```
| ?- save_files([src1,src2],file1).
```

Any module declarations, predicates, multifile clauses, or directives encountered in those files will be saved. Source file information as provided by `source_file/[1,2]` for the relevant predicates and modules is also saved.

To save the modules `user` and `special` into `file2.po` you would use:

```
| ?- save_modules([user,special],file2).
```

The module declarations, predicates and initializations belonging to those modules will be saved. Source file information and embedded directives (except initializations) are *not* saved.

To just save certain predicates into `file3.po` you would use:

```
| ?- save_predicates([person/2,dept/4],file3).
```

This will only save the predicates specified. When the PO file is loaded the predicates will be loaded into the same module they were in originally.

Any PO file, however generated, can be loaded into Prolog with `load_files/[1,2]`:

```
| ?- load_files(file1).
```

or, equivalently:

```
| ?- [file1].
```

The information from each PO file loaded is incrementally added to the database. This means that definitions from later loads may replace definitions from previous loads.

The predicates `load_files/[1,2]` are used for compiling and loading source files as well as PO files. If `file1.po` and `file1.pl` both exist (and `file1` does not), then `load_files(file1)` will load the source (`.pl`) or the PO, whichever is the most recent.

Refer to Section 4.3 [ref-lod], page 79, for more information on loading programs, and also to the reference page for `load_files/[1,2]`. 
4.4.4 Predicate List

Detailed information is found in the reference pages for the following:

- **initialization :G**  
  declaration, ISO  
  declares G to be run when program is started

- **load_files(:F)**  
  load files according to options O

- **load_files(:F,+:O)**  
  load files according to options O

- **user:runtime_entry(+S)**  
  hook  
  entry point for a runtime system

- **save_files(+L,+F)**  
  saves the modules, predicates, clauses and directives in the given files L into file F

- **save_modules(+L,+F)**  
  save the modules specified in L into file F

- **save_predicates(:L,+F)**  
  save the predicates specified in L into file F

- **save_program(+F)**  
  save all Prolog data into file F with startup goal G

- **volatile :P**  
  declaration  
  declares predicates specified by P to not be included in saves.

4.5 Files and Directories

4.5.1 The File Search Path Mechanism

As a convenience for the developer and as a means for extended portability of the final application, SICStus Prolog provides a flexible mechanism to localize the definitions of the system dependent parts of the file and directory structure a program relies on, in such a way that the application can be moved to a different directory hierarchy or to a completely new file system, with a minimum of effort.

This mechanism, which can be seen as a generalization of the `user:library_directory/1` scheme available in previous releases, presents two main features:

1. An easy way to create aliases for frequently used directories, thus localizing to one single place in the program the physical directory name, which typically depends on the file system and directory structure.

2. A possibility to associate more than one directory specification with each alias, thus giving the developer full freedom in sub-dividing libraries, and other collections of programs, as it best suits the structure of the external file system, without making the process of accessing files in the libraries any more complicated. In this case, the alias can be said to represent a file search path, not only a single directory.
The directory aliasing mechanism, together with the additional file search capabilities of `absolute_file_name/3`, can effectively serve as an intermediate layer between the external world and a portable program. For instance, the developer can hide the directory representation by defining directory aliases, and he can automatically get a proper file extension added, dependent on the type of file he wants to access, by using the appropriate options to `absolute_file_name/3`.

A number of directory aliases and file search paths, are predefined in the SICStus Prolog system. The most important of those is the `library` file search path, giving the user instant access to the SICStus library, consisting of several sub-directories and extensive supported programs and tools.

Specifying a library file, using the alias, is possible simply by replacing the explicit file (and directory) specification with the following term:

```
library(file)
```

The name of the file search path, in this case `library`, is the main functor of the term, and indicates that `file` is to be found in one of the library directories.

The association between the alias `library` (the name of the search path) and the library directories (the definitions of the search path), is extended by Prolog facts, `user:library_directory/1`, which are searched in sequence to locate the file. Each of these facts specifies a directory where to search for `file`, whenever a file specification of the form `library(file)` is encountered.

The library mechanism discussed above, which can be extended with new directories associated with the alias `library`, has become subsumed by a more general aliasing mechanism, in which arbitrary names can be used as aliases for directories. The general mechanism also gives the possibility of defining path aliases in terms of already defined aliases.

In addition to `library`, the following aliases are predefined in SICStus Prolog: `runtime`, `system`, `application`, `temp`, and `path`. The interpretation of the predefined aliases are explained below.

### 4.5.1.1 Defining File Search Paths

The information about which directories to search when an alias is encountered is extended by clauses for the hook predicate `user:file_search_path/2`, of the following form:

```
user:file_search_path(PathAlias, DirectorySpec).
```

- **PathAlias** must be an atom. It can be used as an alias for `DirectorySpec`.
- **DirectorySpec**
  - Can either be an atom, spelling out the name of a directory, or a compound term using other path aliases to define the location of the directory.

The directory path may be absolute, as in (A) or relative as in (B), which defines a path relative to the current working directory.
Then, files may be referred to by using file specifications of the form similar to `library(file)`. For example, (C), names the file `/usr/jackson/.login`, while (D) specifies the path `etc/demo/my_demo` relative to the current working directory.

```
user:file_search_path(home, '/usr/jackson').  (A)
user:file_search_path(demo, 'etc/demo').   (B)
home('.login')         (C)
demo(my_demo)          (D)
```

As mentioned above, it is also possible to have multiple definitions for the same alias. If clauses (E) and (F) define the `home` alias, then to locate the file specified by (G) each `home` directory is searched in sequence for the file `.login`. If `/usr/jackson/.login` exists, then it is used. Otherwise, `/u/jackson/.login` is used if it exists.

```
user:file_search_path(home, '/usr/jackson').  (E)
user:file_search_path(home, '/u/jackson').   (F)
home('.login')       (G)
```

The directory specification may also be a term of arity 1, in which case it specifies that the argument of the term is relative to the `user:file_search_path/2` defined by its functor. For example, (H) defines a directory relative to the directory given by the `home` alias. Therefore, the alias `sp_directory` represents the search path `/usr/jackson/prolog/sp` followed by `/u/jackson/prolog/sp`. Then, the file specification (I) refers to the file (J), if it exists. Otherwise, it refers to the file (K), if it exists.

```
user:file_search_path(sp_directory, home('prolog/sp')).  (H)
sp_directory(test)         (I)
/usr/jackson/prolog/sp/test  (J)
/u/jackson/prolog/sp/test    (K)
```

Aliases such as `home` or `sp_directory` are useful because even if the `home` directory changes, or the `sp_directory` is moved to a different location, only the appropriate `user:file_search_path/2` facts need to be changed. Programs relying on these paths are not affected by the change of directories because they make use of file specifications of the form `home(file)` and `sp_directory(file)`.

All built-in predicates that take file specification arguments allow these specifications to include path aliases defined by `user:file_search_path/2` facts. The main predicate for expanding file specifications is `absolute_file_name/[2,3]`. See Section 11.3.79 [mpg-ref-file_search_path], page 1023.
Please note: The user:file_search_path/2 database may contain directories that do not exist or are syntactically invalid (as far as the operating system is concerned). If an invalid directory is part of the database, then the system will fail to find any files in it, and the directory will effectively be ignored.

4.5.1.2 Frequently Used File Specifications

Frequently used user:file_search_path/2 facts are best defined using the initialization file "/.sicstusrc" or "/sicstus.ini", which is consulted at startup time by the Development System. Therefore, with reference to the examples from Section 4.5.1.1 [ref-fdi-fsp-def], page 96, clauses like the one following should be placed in the initialization file so that they are automatically available to user programs after startup:

```prolog
:- multifile user:file_search_path/2.
user:file_search_path(home, '/usr/jackson').
user:file_search_path(sp_directory, home('prolog/sp')).
user:file_search_path(demo, 'etc/demo').
```

4.5.1.3 Predefined File Search Paths

user:file_search_path/2 is undefined at startup, but all callers first try a number of default file search paths, almost as if user:file_search_path/2 had the following initial clauses. Therefore, to expand file search paths, you should not call user:file_search_path/2 directly, but instead call absolute_file_name/[2,3]. See Section 4.9.4 [ref-lps-flg], page 136, for more info on the Prolog flag host_type.

The system properties SP_APP_DIR and SP_RT_DIR expand respectively to the absolute path of the directory that contains the executable and the directory that contains the SICStus runtime. The system property SP_TEMP_DIR expands to a directory suitable for storing temporary files, it is particularly useful with the open/4 option if_exists(generate_unique_name).

```prolog
%%% file_search_path/2 (virtual) initial clauses
file_search_path(library, '$SP_LIBRARY_DIR').
file_search_path(library, Path) :-
    library_directory(Path).
file_search_path(system, Platform) :-
    prolog_flag(host_type, Platform).
file_search_path(application, '$SP_APP_DIR').
file_search_path(runtime, '$SP_RT_DIR').
file_search_path(temp, '$SP_TEMP_DIR').
file_search_path(path, Path) :-
    %% enumerate all directories in $PATH
    ...
```

The only default expansion for the library file search path is the value of the system property SP_LIBRARY_DIR. However, you can add expansions for library by adding clauses to user:library_directory/1 (which is initially undefined). This feature is mainly for
compatibility with earlier releases. It is better to add your own names for file search paths
directly to \texttt{user:file_search_path/2} and not extend the file search path \texttt{library} at all.

\begin{verbatim}
:- multifile user:library_directory/1.
  user:library_directory('/home/joe/myprologcode/').
  user:library_directory('/home/jane/project/code').
\end{verbatim}

4.5.2 Syntactic Rewriting

A file specification must be an atom or a compound term with arity 1. Such compound
terms are transformed to atoms as described in Section 4.5.1 [ref-fdi-fsp], page 95. Let \textit{FileSpec} be the given or transformed atomic file specification.

A file specification \textit{FileSpec} is subject to \textit{syntactic rewriting}. Depending on the opera-
tion, the resulting absolute filename is subject to further processing. Syntactic rewriting is
performed wrt. a context directory \textit{Context} (an absolute path), in the following steps:

- Under Windows, all ‘\textbackslash’ characters are converted to ‘/’. This replacement is also per-
  formed, as needed, during all subsequent steps.
- A ‘$PROP’ in the beginning of \textit{FileSpec}, followed by ‘/’ or the end of the path, is
  replaced by the absolute path of the value of the system property \textit{PROP}. This is
  especially useful when the system property has no explicit value and thus takes its
  value from the environment variable with the same name. If \texttt{var} does not exist or its
  value is empty, then a permission error is raised.
- A relative path that does not begin with ‘/’ is made absolute by prepending \textit{Context}
  followed by a ‘/’. Note that, under UNIX, all paths that begin with ‘/’ are absolute.
  Under Windows only, a relative path that begins with a ‘/’ is made absolute by prepend-
  ing the root (see below) of \textit{Context}.
- A ‘~user’ in the beginning of \textit{FileSpec}, followed by ‘/’ or the end of the path, is
  replaced by the absolute path of the home directory of \texttt{user}. If the home directory of \texttt{user}
  cannot be determined, then a permission error is raised.
  Under Windows this has not been implemented, instead a permission error is raised.
  If the home directory of \texttt{user} is a relative path, then it is made absolute using \textit{Context}
  if needed.
- A ‘~’ in the beginning of \textit{FileSpec}, followed by ‘/’ or the end of the path, is replaced
  by the absolute path of the home directory of the current user. If the home directory
  of the current user cannot be determined, then a permission error is raised.
  The home directory of the current user is a relative path it is made absolute using \textit{Context}
  if needed.
- If \textit{FileSpec} is a relative file name, then \textit{Context} is prepended to it.
- The \texttt{root} of the file name is determined. Under UNIX this is simply the initial ‘/’, if
  any. Under Windows there are several variants of roots, as follows.
  - \texttt{driveletter:/} where \texttt{driveletter} is a single upper or lower case character in the
    range ‘a’ to ‘z’. For example, ‘C:/’.
- //?/driveletter:/ This is transformed to driveletter:/.
- //host/share/ (a ‘UNC’ path, also known as a network path) where host and share are non-empty and do not contain /.
- //?/UNC/host/share/ This is transformed to //host/share/

If no root can be determined, then a permission error is raised.

A path is absolute if and only if it is begins with a root, as above.

- The following steps are repeatedly applied to the last ‘/’ of the root and the characters that follow it repeatedly until no change occurs.
  1. Repeated occurrences of / are replaced by a single /.
  2. ‘/.’, followed by ‘/’ or the end of the path, is replaced by ‘/’.
  3. /parent/..., followed by ‘/’ or the end of the path, is replaced by ‘/’.

If the path still contains /., followed by ‘/’ or the end of the path, then a permission error is raised.

- Any trailing ‘/’ is deleted unless it is part of the root.

- Finally, under Windows, the case-normalized path is obtained as follows: All Latin 1 characters (i.e. character codes in [0..255]) are converted to lower case. All other characters are converted to upper case.

File systems under Windows are generally case insensitive. This step ensures that two file names that differ only in case, and therefore would reference the same file in the file system, will case-normalize to identical atoms.

Since release 4.3, open/[3,4], and other build-in predicates that create files and directories, creates files using the file name argument as obtained from syntactic rewriting but before applying case-normalization. This means that open('HelloWorld.txt', write, S), file_property(S, file_name(Name). will create a file that has the mixed-case name HelloWorld.txt in the file system but Name will end in 'helloworld.txt', i.e. the stream property will reflect the case-normalized path.

The fact that open/[3,4] et al. preserves case when creating files seldom matters, except for aesthetics, since any Windows application that tries to open a file named HelloWorld.txt will also find helloworld.txt.

The following UNIX examples assumes that Context is ‘/usr’; that the environment variables VAR1, VAR2, VAR3 have the values ‘/opt/bin’, ‘foo’ and ‘~/.temp’ respectively and that the home directory of the current user, ‘joe’, is ‘/home/joe’.

/foo/bar
⇒ /foo/bar

/foo//bar../blip///
⇒ /foo/blip

/foo//../bar//../blip
⇒ error
The following Windows examples assume that Context is ‘C:/Source/proj1’; that the
environment variables VAR1, VAR2, VAR3 have the values ‘\server\docs\brian’, ‘foo’ and
‘~/temp’ respectively and that the home directory of the current user is ‘C:/home’.

\texttt{/foo/bar} \\
\rightarrow \texttt{c:/foo/bar}

\texttt{foo// ../../blip} \\
\rightarrow \texttt{c:/source/blip}

\$\texttt{VAR1/../local/} \\
\rightarrow \texttt{server/docs/local}

\$\texttt{VAR2/misc/.} \\
\rightarrow \texttt{c:/source/proj1/foo/misc}

\$\texttt{VAR3/misc/.} \\
\rightarrow \texttt{c:/home/temp/misc}

\~\texttt{joe/../jenny/bin.} \\
\rightarrow \texttt{error}

\textbf{4.5.3 List of Predicates}

Detailed information is found in the reference pages for the following:

\begin{verbatim}
absolute_file_name(+R,-A) hookable
\end{verbatim}

expand relative filename \texttt{R} to absolute file name \texttt{A} using options specified in \texttt{O}

\begin{verbatim}
user:file_search_path(+F,-D) hook
\end{verbatim}

directory \texttt{D} is included in file search path \texttt{F}
4.6 Input and Output

4.6.1 Introduction
Prolog provides two classes of predicates for input and output: those that handle individual bytes or characters, and those that handle complete Prolog terms.

Input and output happen with respect to streams. Therefore, this section discusses predicates that handle files and streams in addition to those that handle input and output of bytes, characters and terms.

4.6.2 About Streams
A Prolog stream can refer to a file or to the user’s terminal. Each stream is used either for input or for output, but typically not for both. A stream is either text, for character and term I/O, or binary, for byte I/O. At any one time there is a current input stream and a current output stream.

Input and output predicates fall into two categories:
1. those that use the current input or output stream;
2. those that take an explicit stream argument;

Initially, the current input and output streams both refer to the user’s terminal. Each input and output built-in predicate refers implicitly or explicitly to a stream. The predicates that perform byte, character and term I/O operations come in pairs such that (A) refers to the current stream, and (B) specifies a stream.

    predicate_name/n  \hspace{1cm} (A)
    predicate_name/n+1 \hspace{1cm} (B)

4.6.2.1 Programming Note
Deciding which version to use involves a trade-off between speed and readability of code: in general, version (B), which specifies a stream, runs slower than (A). So it may be desirable to write code that changes the current stream and uses version (A). However, the use of (B) avoids the use of global variables and results in more readable programs.

4.6.2.2 Stream Categories
SICStus Prolog streams are divided into two categories, those opened by see/1 or tell/1 and those opened by open/[3,4]. A stream in the former group is referred to by its file specification, while a stream in the latter case is referred to by its stream object (see the figure “Categorization of Stream Handling Predicates”). For further information about file specifications, see Section 4.5 [ref-fdi], page 95. Stream objects are discussed in Section 4.6.7.1

---

3 At the C level, you can define more general streams, e.g. referring to pipes or to encrypted files.
Chapter 4: The Prolog Language

Each operating system permits a different number of streams to be open.

4.6.3 Term Input

Term input operations include:

- reading a term and
- changing the prompt that appears while reading.

4.6.3.1 Reading Terms: The "Read" Predicates

The “Read” predicates are

- \texttt{read(-Term)}
- \texttt{read(+Stream, -Term)}
- \texttt{read_term(-Term, +Options)}
- \texttt{read_term(+Stream, -Term, +Options)}

\texttt{read_term/[2,3]} offers many options to return extra information about the term. See Section 11.3.185 [mpg-ref-read_term], page 1163.

When Prolog reads a term from the current input stream the following conditions must hold:

- The term must be followed by a full stop. See Section 4.1.7.1 [ref-syn-syn-ove], page 52. The full stop is removed from the input stream but is not a part of the term that is read.
- \texttt{read/[1,2]} does not terminate until the full stop is encountered. Thus, if you type at top level
  \[
  \text{\| \ ?- \ read(X)}
  \]
  then you will keep getting prompts (first ‘|: ’, and five spaces thereafter) every time you type RET, but nothing else will happen, whatever you type, until you type a full stop.
- The term is read with respect to current operator declarations. See Section 4.1.5 [ref-syn-ops], page 47, for a discussion of operators.
- When a syntax error is encountered, an error message is printed and then the “read” predicate tries again, starting immediately after the full stop that terminated the erroneous term. That is, it does not fail on a syntax error, but perseveres until it eventually manages to read a term. This behavior can be changed with \texttt{prolog_flag/3} or using \texttt{read_term/[2,3]}. See Section 11.3.167 [mpg-ref-prolog_flag], page 1143.
- If the end of the current input stream has been reached, then \texttt{read(X)} will cause \texttt{X} to be unified with the atom \texttt{end_of_file}. 

4.6.3.2 Changing the Prompt
To query or change the sequence of characters (prompt) that indicates that the system is waiting for user input, call prompt/2.

This predicate affects only the prompt given when a user’s program is trying to read from the terminal (for example, by calling read/1 or get_code/1). Note also that the prompt is reset to the default ‘1: ’ on return to the top level. See Section 11.3.169 [mpg-ref-prompt], page 1146.

4.6.4 Term Output
Term output operations include:

• writing to a stream (various “write” Predicates)
• displaying, usually on the user’s terminal (display/1)
• changing the effects of print/[1,2] (user:portray/1)
• writing a clause as listing/[0,1] does, except original variable names are not retained (portray_clause/[1,2])

4.6.4.1 Writing Terms: the “Write” Predicates

• write(+Stream, +Term)
• write(+Term)
• writeq(+Stream, +Term)
• writeq(+Term)
• write_canonical(+Term)
• write_canonical(+Stream, +Term)
• write_term(+Stream, +Term, +Options)
• write_term(+Term, +Options)

write_term/[2,3] is a generalization of the others and provides a number of options. See Section 11.3.253 [mpg-ref-write_term], page 1249.

4.6.4.2 Common Characteristics
The output of the “write” predicates is not terminated by a full stop; therefore, if you want the term to be acceptable as input to read/[1,2], then you must send the terminating full stop to the output stream yourself. For example,

| ?- write(a), write(’ .’), nl. |

Note that, in general, you need to prefix the full stop with a layout character, like space, to ensure that it can not “glue” with characters in the term.

If Term is uninstantiated, then it is written as an anonymous variable (an underscore followed by a non-negative integer).
Please note: The “name” used when writing a variable may differ between separate calls to a ‘Write“ predicate. If this is a concern, then you can use either of the following methods to ensure that the variable is always written in same way.

- Avoid the problem altogether by writing the entire term with a single call to a “write“ predicate. Multiple occurrence of the same variable within the written term will be written in the same way.
- Use the variable_names/1 write_term option to explicitly name the variable. This option was added in release 4.3.
- Use numbervars/3 to bind the variables in the written term to (ground) ’$VAR'(N) terms and use the numbervars(true) write_term option. Note that this may not work with attributed variables, like those used by library(clpfd).

write_canonical/[1,2] is provided so that Term, if written to a file, can be read back by read/[1,2] regardless whether there are special characters in Term or prevailing operator declarations.

4.6.4.3 Distinctions Among the “Write” Predicates

- For write and writeq, the term is written with respect to current operator declarations (See Section 4.1.5 [ref-syn-ops], page 47, for a discussion of operators).

write_canonical(Term) writes Term to the current or specified output stream in standard syntax (see Section 4.1 [ref-syn], page 43, on Prolog syntax), and quotes atoms and functors to make them acceptable as input to read/[1,2]. That is, operator declarations are not used and compound terms are always written in the form:

\[
\text{name(arg1, ..., argn)}
\]

and the special list syntax, e.g. [a,b,c], or braces syntax, e.g. {a,b,c} are not used. Calling write_canonical/1 is a good way of finding out how Prolog parses a term with several operators.

- Atoms output by write/[1,2] cannot in general be read back using read/[1,2]. For example,

\[
| \Leftarrow \text{write('a b').}
\]

\[a\ b\]

For this reason write/[1,2] is only useful as a way to treat atoms as strings of characters. It is rarely, if ever, useful to use write/[1,2] with other kinds of terms, i.e. variables, numbers or compound terms.

If you want to be sure that the atom can be read back by read/[1,2], then you should use writeq/[1,2], or write_canonical/[1,2], which put quotes around atoms when necessary, or use write_term/[2,3] with the quoted option set to true. Note also that the printing of quoted atoms is sensitive to character escaping (see Section 4.1.4 [ref-syn-ces], page 47).

- write/[1,2] and writeq/[1,2] use the write option numbervars(true), so treat terms of the form ’$VAR'(N) specially: they write ‘A’ if N=0, ‘B’ if N=1, ..., ‘Z’ if N=25, ‘A1’ if N=26, etc. Terms of this form are generated by numbervars/3 (see Section 4.8.6 [ref-lte-any], page 129).
\[ \text{?- writeq(a('VAR'(0),'VAR'(1))).} \]

\[ \text{write canonical/1 does not treat terms of the form 'VAR'(N) specially.} \]

### 4.6.4.4 Displaying Terms

Like \texttt{write canonical/[1,2]}, \texttt{display/1} ignores operator declarations and shows all compound terms in standard prefix form. For example, the command

\[ \text{?- display(a+b).} \]

produces the following:

\[ +(a,b) \]

Unlike \texttt{write canonical/[1,2]}, \texttt{display/1} does not put quotes around atoms and functors, even when needed for reading the term back in, so \texttt{write canonical/[1,2]} is often preferable. See Section 11.3.66 [mpg-ref-display], page 1009.

### 4.6.4.5 Using the Portray Hook

By default, the effect of \texttt{print/[1,2]} is the same as that of \texttt{write/[1,2]}, but you can change its effect by providing clauses for the hook predicate \texttt{user:portray/1}.

If \texttt{X} is a variable, then it is printed using \texttt{write(X)}. Otherwise the user-definable procedure \texttt{user:portray(X)} is called. If this succeeds, then it is assumed that \texttt{X} has been printed and \texttt{print/[1,2]} exits (succeeds).

If the call to \texttt{user:portray/1} fails, and if \texttt{X} is a compound term, then \texttt{write/[1,2]} is used to write the principal functor of \texttt{X} and \texttt{print/[1,2]} is called recursively on its arguments. If \texttt{X} is atomic, then it is written using \texttt{write/[1,2]}.

When \texttt{print/[1,2]} has to print a list, say \texttt{[X1,X2,...,Xn]}, it passes the whole list to \texttt{user:portray/1}. As usual, if \texttt{user:portray/1} succeeds, then it is assumed to have printed the entire list, and \texttt{print/[1,2]} does nothing further with this term. Otherwise \texttt{print/[1,2]} writes the list using bracket notation, calling \texttt{print/[1,2]} on each element of the list in turn.

Since \texttt{[X1,X2,...,Xn]} is simply a different way of writing \texttt{.(X1,[X2,...,Xn])}, one might expect \texttt{print/[1,2]} to be called recursively on the two arguments \texttt{X1} and \texttt{[X2,...,Xn]}, giving \texttt{user:portray/1} a second chance at \texttt{[X2,...,Xn]}. This does not happen; lists are a special case in which \texttt{print/[1,2]} is called separately for each of \texttt{X1,X2,...Xn}.

### 4.6.4.6 Portraying a Clause

If you want to print a clause, then \texttt{portray_clause/[1,2]} is almost certainly the command you want. None of the other term output commands puts a full stop after the written term. If you are writing a file of facts to be loaded by \texttt{compile/1}, then use \texttt{portray_clause/[1,2]}, which attempts to ensure that the clauses it writes out can be read in again as clauses.

The output format used by \texttt{portray_clause/[1,2]} and \texttt{listing/[0,1]} has been carefully designed to be clear. We recommend that you use a similar style. In particular, never
put a semicolon (disjunction symbol) at the end of a line in Prolog. See Section 11.3.157 [mpg-ref-portray_clause], page 1129.

4.6.5 Byte and Character Input

4.6.5.1 Overview

The operations in this category are:

- reading (“get” predicates),
- peeking (“peek” predicates),
- skipping (“skip” predicates),
- checking for end of line or end of file (“at_end” predicates).

4.6.5.2 Reading Bytes and Characters

- `get_byte([Stream,] N)` unifies `N` with the next consumed byte from the current or given input stream, which must be binary.
- `get_code([Stream,] N)` unifies `N` with the next consumed character code from the current or given input stream, which must be text.
- `get_char([Stream,] A)` unifies `A` with the next consumed character atom from the current or given input stream, which must be text.

4.6.5.3 Peeking

Peeking at the next character without consuming it is useful when the interpretation of “this character” depends on what the next one is.

- `peek_byte([Stream,] N)` unifies `N` with the next unconsumed byte from the current or given input stream, which must be binary.
- `peek_code([Stream,] N)` unifies `N` with the next unconsumed character code from the current or given input stream, which must be text.
- `peek_char([Stream,] A)` unifies `A` with the next unconsumed character atom from the current or given input stream, which must be text.

4.6.5.4 Skipping

There are two ways of skipping over characters in the current or given input stream: skip to a given character, or skip to the end of a line.

- `skip_byte([Stream,] N)` skips over bytes through the first occurrence of `N` from the current or given input stream, which must be binary.
- `skip_code([Stream,] N)` skips over character codes through the first occurrence of `N` from the current or given input stream, which must be text.
- `skip_char([Stream,] A)` skips over character atoms through the first occurrence of `A` from the current or given input stream, which must be text.
- `skip_line` or `skip_line(Stream)` skips to the end of line of the current or given input stream. Use of this predicate helps portability of code since it avoids dependence on any particular character code(s) being returned at the end of a line.
4.6.5.5 Finding the End of Line and End of File

To test whether the end of a line on the end of the file has been reached on the current or given input stream, use at_end_of_line/[0,1] or at_end_of_stream/[0,1].

Note that these predicates never block waiting for input. This means that they may fail even if the stream or line is in fact at its end. An alternative that will never guess wrong is to use peek_code/[1,2] or peek_byte/[1,2].

4.6.6 Byte and Character Output

The byte and character output operations are:

- writing (putting) bytes and characters
- creating newlines and tabs
- flushing buffers
- formatting output.

4.6.6.1 Writing Bytes and Characters

- put_byte([Stream,] N) writes the byte N to the current or given output stream, which must be binary.
- put_code([Stream,] N) writes the character code N to the current or given output stream, which must be text.
- put_char([Stream,] A) writes the character atom A to the current or given output stream, which must be text.

The byte or character is not necessarily printed immediately; they may be flushed if the buffer is full. See Section 4.6.7.10 [ref-iou-sfh-flu], page 114.

4.6.6.2 New Line

nl or nl(Stream) terminates the record on the current or given output stream. A linefeed character is printed.

4.6.6.3 Formatted Output

format([Stream,] Control, Arguments) interprets the Arguments according to the Control string and prints the result on the current or given output stream. Alternatively, an output stream can be specified in an initial argument. This predicate is used to produce formatted output, like the following example.

| ?- toc(1.5). |
|-------------|-----------------|-----------------|-----------------|-----------------|
| Table of Contents | i |
| *********************** NICE TABLE *********************** |
| * | * |
| * Right aligned | Centered | Left aligned | * |
| * | 123 | 45 | 678 | * |
| * | 1 | 2345 | 6789 | * |
| *********************** ***************************************** |
Chapter 4: The Prolog Language

For details, including the code to produce this example, see the example program in the reference page for `format/[2,3]`. See Section 11.3.85 [mpg-ref-format], page 1032.

### 4.6.7 Stream and File Handling

The operations implemented are opening, closing, querying status, flushing, error handling, setting.

The predicates in the “see” and “tell” families are supplied for compatibility with other Prologs. They take either file specifications or stream objects as arguments (see Section 11.1 [mpg-ref], page 899) and they specify an alternative, less powerful, mechanism for dealing with files and streams than the similar predicates (`open/[3,4]`, etc.), which take stream objects (see the figure “Categorization of Stream Handling Predicates”).

#### 4.6.7.1 Stream Objects

Each input and output stream is represented by a unique Prolog term, a stream object. In general, this term is of the form

*user*  
Stands for the standard input or output stream, depending on context.

*$stream'(X)  
A stream connected to some file. X is an integer.

*Atom*  
A stream alias. Aliases can be associated with streams using the `alias(Atom)` option of `open/4`. There are also three predefined aliases:

*user_input*  
An alias initially referring to the UNIX `stdin` stream. The alias can be changed with `prolog_flag/3` and accessed by the C variable `SP_stdin`.

*user_output*  
An alias initially referring to the UNIX `stdout` stream. The alias can be changed with `prolog_flag/3` and accessed by the C variable `SP_stdout`.

*user_error*  
An alias initially referring to the UNIX `stderr` stream. The alias can be changed with `prolog_flag/3` and accessed by the C variable `SP_stderr`. This stream is used by the Prolog top level and debugger, and for all unsolicited messages by built-in predicates.

Stream objects are created by the predicate `open/[3,4]` Section 4.6.7.4 [ref-iou-sfh-opn], page 110, and passed as arguments to those predicates that need them. Representation for stream objects to be used in C code is different. Use `stream_code/2` to convert from one to the other when appropriate. See Section 11.3.218 [mpg-ref-stream_code], page 1205.

#### 4.6.7.2 Exceptions Related to Streams

All predicates that take a stream argument will raise the following exceptions:
4.6.7.3 Suppressing Error Messages

If the fileerrors flag is set to off, then the built-in predicates that open files simply fail, instead of raising an exception if the specified file cannot be opened.

4.6.7.4 Opening a Stream

Before I/O operations can take place on a stream, the stream must be opened, and it must be set to be current input or current output. As illustrated in the figure “Categorization of Stream Handling Predicates”, the operations of opening and setting are separate with respect to the stream predicates, and combined in the File Specification Predicates.

- **open(File, Mode, Stream)** attempts to open the file File in the mode specified (read, write or append). If the open/3 request is successful, then a stream object, which can be subsequently used for input or output to the given file, is unified with Stream.
  
The read mode is used for input. The write and append modes are used for output. The write option causes a new file to be created for output. If the file already exists, then it is set to empty and its previous contents are lost. The append option opens an already-existing file and adds output to the end of it. The append option will create the file if it does not already exist.

  Options can be specified by calling open/4. See Section 11.3.148 [mpg-ref-open], page 1115.

- **set_input(Stream)** makes Stream the current input stream. Subsequent input predicates such as read/1 and get_code/1 will henceforth use this stream.
- **set_output(Stream)** makes Stream the current output stream. Subsequent output predicates such as write/1 and put_code/1 will henceforth use this stream.

Opening a stream and making it current are combined in *see* and *tell*:
• **see(S)** makes file S the current input stream. If S is an atom, then it is taken to be a file specification, and
  
  – if there is an open input stream associated with the filename, and that stream was opened by see/1, then it is made the current input stream;
  
  – Otherwise, the specified file is opened for input and made the current input stream. If it is not possible to open the file, and the fileerrors flag is on (as it is by default), then see/1 raises an error exception. Otherwise, see/1 merely fails.

See Section 11.3.199 [mpg-ref-see], page 1182.

• **tell(S)** makes S the current output stream.
  
  – If there is an open output stream currently associated with the filename, and that stream was opened by tell/1, then it is made the current output stream;
  
  – Otherwise, the specified file is opened for output and made the current output stream. If the file does not exist, then it is created. If it is not possible to open the file (because of protections, for example), and the fileerrors flag is on (which it is by default), then tell/1 raises an error exception. Otherwise, tell/1 merely fails.

See Section 11.3.224 [mpg-ref-tell], page 1215.

It is important to remember to close streams when you have finished with them. Use seen/0 or close/1 for input files, and told/0 or close/1 for output files.

• **open_null_stream(Stream)** opens a text output stream that is not connected to any file and unifies its stream object with Stream. Characters or terms that are sent to this stream are thrown away. This predicate is useful because various pieces of local state are kept for null streams: the predicates character_count/2, line_count/2 and line_position/2 can be used on these streams (see Section 4.6.8 [ref-iot-sos], page 114).

### 4.6.7.5 Text Stream Encodings

SICStus Prolog supports character codes up to 31 bits wide where the codes are interpreted as for Unicode for the common subset.

When a character code (a “code point” in Unicode terminology) is read or written to a stream, it must be encoded into a byte sequence. The method by which each character code is encoded to or decoded from a byte sequence is called “character encoding”.

The following character encodings are currently supported by SICStus Prolog:

#### ANSI_X3.4-1968
The 7-bit subset of Unicode, commonly referred to as ASCII.

#### ISO-8859-1
The 8-bit subset of Unicode, commonly referred to as Latin 1.

#### ISO-8859-2
A variant of ISO-8859-1, commonly referred to as Latin 2.
ISO-8859-15
   A variant of ISO-8859-1, commonly referred to as Latin 9.

windows 1252
   The Microsoft Windows code page 1252.

UTF-8
UTF-16
UTF-16LE
UTF-16BE
UTF-32
UTF-32LE
UTF-32BE

   The suffixes LE and BE denote respectively little endian and big endian.
   These encodings can be auto-detected if a Unicode signature is present in a
   file opened for read. A Unicode signature is also known as a Byte order mark
   (BOM).

In addition, it is possible to use all alternative names defined by the IANA registry http://
www.iana.org/assignments/character-sets

All encodings in the table above, except the UTF-XXX encodings, supports the
reposition(true) option to open/4 (see Section 11.3.148 [mpg-ref-open], page 1115).

The encoding to use can be specified when using open/4 and similar predicates using the
option encoding/1. When opening a file for input, the encoding can often be determined
automatically. The default is ISO-8859-1 if no encoding is specified and no encoding can
be detected from the file contents.

The encoding used by a text stream can be queried using stream_property/2.

See Section 11.3.148 [mpg-ref-open], page 1115, for details on how character encoding is
auto-detected when opening text files.

4.6.7.6 Finding the Current Input Stream

- current_input(Stream) unifies Stream with the current input stream.
- If the current input stream is user_input, then seeing(S) unifies S with user. Otherwise,
  if the current input stream was opened by see(F), then seeing(S) unifies S with F. Otherwise,
  if the current input stream was opened by open/[3,4], then seeing(S) unifies S with the corresponding stream object.

seeing/1 can be used to verify that a section of code leaves the current input stream
unchanged as follows:

/* nonvar(FileNameOrStream), */
see(FileNameOrStream),
...
seeing(FileNameOrStream)

WARNING: The sequence
seeing(File),
...
set_input(File),
will signal an error if the current input stream was opened by see/1. The only sequences that are guaranteed to succeed are
seeing(FileOrStream),
...
see(FileOrStream)
and
current_input(Stream),
...
set_input(Stream)

4.6.7.7 Finding the Current Output Stream

- current_output(Stream) unifies Stream with the current output stream.
- If the current output stream is user_output, then telling(S) unifies S with user. Otherwise, if the current output stream was opened by tell(F), then telling(S) unifies S with F. Otherwise, if the current output stream was opened by open/[3,4], then telling(S) unifies S with the corresponding stream object.

telling/1 can be used to verify that a section of code leaves the current output stream unchanged as follows:
/* nonvar(FileNameOrStream), */
tell(FileNameOrStream),
...
telling(FileNameOrStream)
WARNING: The sequence
telling(File),
...
set_output(File),
will signal an error if the current output stream was opened by tell/1. The only sequences that are guaranteed to succeed are
telling(FileOrStream),
...
tell(FileOrStream)
and
current_output(Stream),
...
set_output(Stream)

4.6.7.8 Finding Out About Open Streams

current_stream(File, Mode, Stream) succeeds if Stream is a stream that is currently open on file File in mode Mode, where Mode is either read, write, or append. None of the arguments need be initially instantiated. This predicate is nondeterminate and can be
used to backtrack through all open streams. current_stream/3 ignores certain predefined streams, including the initial values of the special streams for the standard input, output, and error channels. See Section 11.3.57 [mpg-ref-current_stream], page 1000.

stream_property(Stream, Property) succeeds if Stream is a currently open stream with property Property. Predefined streams, like the three standard channels, are not ignored. See Section 11.3.221 [mpg-ref-stream_property], page 1208.

4.6.7.9 Closing a Stream
- close(X) closes the stream corresponding to X, where X should be a stream object created by open/[3,4], or a file specification passed to see/1 or tell/1. In the example:
  ```prolog
  see(foo),
  ...
  close(foo)
  ```
  foo will be closed. However, in the example:
  ```prolog
  open(foo, read, S),
  ...
  close(foo)
  ```
  an exception will be raised and foo will not be closed. See Section 11.3.39 [mpg-ref-close], page 978.
- told/0 closes the current output stream. The current output stream is then set to be user_output.
- seen/0 closes the current input stream. The current input stream is then set to be user_input.

4.6.7.10 Flushing Output
Output to a stream is not necessarily sent immediately; it is buffered. The predicate flush_output/1 flushes the output buffer for the specified stream and thus ensures that everything that has been written to the stream is actually sent at that point.

- flush_output(Stream) sends all data in the output buffer to stream Stream.

See Section 11.3.82 [mpg-ref-flush_output], page 1029.

4.6.8 Reading the State of Opened Streams
Byte, character, line count and line position for a specified stream are obtained as follows:

- byte_count(Stream, N) unifies N with the total number of bytes either read or written on the open binary stream Stream.
- character_count(Stream, N) unifies N with the total number of characters either read or written on the open text stream Stream.
- line_count(Stream, N) unifies N with the total number of lines either read or written on the open text stream Stream. A freshly opened text stream has a line count of 0, i.e. this predicate counts the number of newlines seen.
• line_position(Stream, N) unifies N with the total number of characters either read or written on the current line of the open text stream Stream. A fresh line has a line position of 0, i.e. this predicate counts the length of the current line.

4.6.8.1 Stream Position Information for Terminal I/O
Input from Prolog streams that have opened the user’s terminal for reading is echoed back as output to the same terminal. This is interleaved with output from other Prolog streams that have opened the user’s terminal for writing. Therefore, all streams connected to the user’s terminal share the same set of position counts and thus return the same values for each of the predicates character_count/2, line_count/2 and line_position/2.

4.6.9 Random Access to Files
There are two methods of finding and setting the stream position, stream positioning and seeking. The current position of the read/write pointer in a specified stream can be obtained by using stream_position/2 or stream_property/2. It may be changed by using set_stream_position/2. Alternatively, seek/4 may be used.

Seeking is more general, and stream positioning is more portable. The differences between them are:

• stream_position/2 is similar to seek/4 with Offset = 0, and Method = current.
• Where set_stream_position/2 asks for stream position objects, seek/4 uses integer expressions to represent the position or offset. Stream position objects are obtained by calling stream_position/2, and are discussed in the reference page.

4.6.10 Summary of Predicates and Functions
Reference pages for the following provide further detail on the material in this section.

at_end_of_line
at_end_of_line(+S)  ISO
  testing whether at end of line on input stream S

at_end_of_stream
at_end_of_stream(+S)  ISO
  testing whether end of file is reached for the input stream S

flush_output
flush_output(+S)  ISO
  flush the output buffer for stream S

get_byte(-C)  ISO
get_byte(+S,-C)  ISO
  C is the next byte on binary input stream S

get_char(-C)  ISO
get_char(+S,-C)  ISO
  C is the next character atom on text input stream S
get_code(-C)
get_code(+S,-C)  
    C is the next character code on text input stream S

nl
nl(+S)  
    send a newline to stream S

peek_byte(+C)
peek_byte(+S,+C)  
    looks ahead for next input byte on the binary input stream S

peek_char(+C)
peek_char(+S,+C)  
    looks ahead for next input character atom on the text input stream S

peek_code(+C)
peek_code(+S,+C)  
    looks ahead for next input character code on the text input stream S

put_byte(+C)
put_byte(+S,+C)  
    write byte C to binary stream S

put_char(+C)
put_char(+S,+C)  
    write character atom C to text stream S

put_code(+C)
put_code(+S,+C)  
    write character code C to text stream S

skip_byte(+C)
skip_byte(+S,+C)  
    skip input on binary stream S until after byte C

skip_char(+C)
skip_char(+S,+C)  
    skip input on text stream S until after char C

skip_code(+C)
skip_code(+S,+C)  
    skip input on text stream S until after code C

skip_line
skip_line(+S)  
    skip the rest input characters of the current line on the input stream S

byte_count(+S,-N)  
    N is the number of bytes read/written on binary stream S

character_count(+S,-N)  
    N is the number of characters read/written on text stream S
close(+F)

ISO

close(+F,+O)

ISO

close file or stream F with options O

current_input(~S)

ISO

S is the current input stream

current_output(~S)

ISO

S is the current output stream

current_stream(?F,?M,?S)

ISO

S is a stream open on file F in mode M

line_count(+S,-N)

ISO

N is the number of lines read/written on text stream S

line_position(+S,-N)

ISO

N is the number of characters read/written on the current line of text stream S

open(+F,+M,-S)

ISO

open(+F,+M,-S,+O)

ISO

file F is opened in mode M, options O, returning stream S

open_null_stream(+S)

ISO

new output to text stream S goes nowhere

prompt(-O,+N)

ISO

queries or changes the prompt string of the current input stream

see(+F)

ISO

make file F the current input stream

seeing(-N)

ISO

the current input stream is named N

seek(+S,+O,+M,+N)

ISO

seek to an arbitrary byte position on the stream S

seen

ISO

close the current input stream

set_input(+S)

ISO

select S as the current input stream

set_output(+S)

ISO

select S as the current output stream

set_stream_position(+S,+P)

ISO

P is the new position of stream S

stream_code(?S,?C)

ISO

Converts between Prolog and C representations of a stream

stream_position(+S,-P)

ISO

P is the current position of stream S

stream_position_data(?Field,?Position,?Data)

ISO

The Field field of the stream position term Position is Data.
stream_property(\(\text{Stream}, \text{Property}\)) \quad \text{ISO}
Stream Stream has property Property.
tell(+\(F\)) make file F the current output stream
telling(-\(N\)) to file N
told close the current output stream

char_conversion(+\(\text{InChar}, \text{OutChar}\)) \quad \text{ISO}
The mapping of \(\text{InChar}\) to \(\text{OutChar}\) is added to the character-conversion mapping.
current_char_conversion(?\(\text{InChar}, \text{OutChar}\)) \quad \text{ISO}
\(\text{InChar}\) is mapped to \(\text{OutChar}\) in the current character-conversion mapping.
current_op(?\(\text{P, T, A}\)) \quad \text{ISO}
atom A is an operator of type T with precedence P
display(+\(T\)) write term \(T\) to the user output stream in functional notation

format(+\(C, :A\))
format(+\(S, +C, :A\))
write arguments A on stream S according to control string C

op(+\(\text{P, T, A}\)) \quad \text{ISO}
make atom A an operator of type T with precedence P

user:portray(+\(T\)) \quad \text{hook}
tell print/[1,2] and write_term/[2,3] what to do

portray_clause(+\(C\))
portray_clause(+\(S, +C\))
write clause C to the stream S

print(+\(T\)) \quad \text{hookable}
print(+\(S, +T\)) \quad \text{hookable}
display the term \(T\) on stream S using user:portray/1 or write/2

read(-\(T\)) \quad \text{ISO}
read(+\(S, -T\)) \quad \text{ISO}
read term \(T\) from stream S

read_term(-\(T, +O\)) \quad \text{ISO}
read_term(+\(S, -T, +O\)) \quad \text{ISO}
read \(T\) from stream S according to options \(O\)

write(+\(T\)) \quad \text{ISO}
write(+\(S, +T\)) \quad \text{ISO}
write term \(T\) on stream S

write_canonical(+\(T\)) \quad \text{ISO}
write_canonical(+\(S, +T\)) \quad \text{ISO}
write term \(T\) on stream S so that it can be read back by read/[1,2]
4.7 Arithmetic

4.7.1 Overview

In Prolog, arithmetic is performed by certain built-in predicates, which take arithmetic expressions as their arguments and evaluate them. Arithmetic expressions can evaluate to integers or floating-point numbers (floats).

The range of integers is \([-2^{2147483616}, 2^{2147483616})\]. Thus for all practical purposes, the range of integers can be considered infinite.

The range of floats is the one provided by the C double type, typically \([4.9e-324, 1.8e+308]\) (plus or minus). In case of overflow or division by zero, an evaluation error exception will be raised. Floats are represented by 64 bits and they conform to the IEEE 754 standard.

The arithmetic operations of evaluation and comparison are implemented in the predicates described in Section 4.7.2 [ref-ari-eae], page 119, and Section 4.7.4 [ref-ari-acm], page 120. All of them take arguments of the type Expr, which is described in detail in Section 4.7.5 [ref-ari-aex], page 120.

4.7.2 Evaluating Arithmetic Expressions

The most common way to do arithmetic calculations in Prolog is to use the built-in predicate is/2.

\[-Term is +Expr\]

Term is the value of arithmetic expression Expr.

Expr must not contain any uninstantiated variables. Do Not confuse is/2 with =/2.

4.7.3 Exceptions Related to Arithmetic

All predicates that evaluate arithmetic expressions will raise the following exceptions:

instantiation_error

Nonground expression given.

type_error

Float given where integer required, or integer given where float is required, or term given as expression with a principal functor that is not a defined function.

evaluation_error

Function undefined for the given argument. For example, attempt to divide by zero.
representation_error

Integer value too large to be represented.

The reference page for such predicates will simply refer to these as “Arithmetic errors” and will go on to detail other exceptions that may be raised for a particular predicate.

4.7.4 Arithmetic Comparison

Each of the following predicates evaluates each of its arguments as an arithmetic expression, then compares the results. If one argument evaluates to an integer and the other to a float, then the integer is coerced to a float before the comparison is made.

Note that two floating-point numbers are equal if and only if they have the same bit pattern. Because of rounding error, it is not normally useful to compare two floats for equality.

\[ \text{Expr1} =:= \text{Expr2} \]

succeeds if the results of evaluating terms \( \text{Expr1} \) and \( \text{Expr2} \) as arithmetic expressions are equal

\[ \text{Expr1} =\neq \text{Expr2} \]

succeeds if the results of evaluating terms \( \text{Expr1} \) and \( \text{Expr2} \) as arithmetic expressions are not equal

\[ \text{Expr1} < \text{Expr2} \]

succeeds if the result of evaluating \( \text{Expr1} \) as an arithmetic expression is less than the result of evaluating \( \text{Expr2} \) as an arithmetic expression.

\[ \text{Expr1} > \text{Expr2} \]

succeeds if the result of evaluating \( \text{Expr1} \) as an arithmetic expression \( \text{Expr1} \) is greater than the result of evaluating \( \text{Expr2} \) as an arithmetic expression.

\[ \text{Expr1} =< \text{Expr2} \]

succeeds if the result of evaluating \( \text{Expr1} \) as an arithmetic expression is not greater than the result of evaluating \( \text{Expr2} \) as an arithmetic expression.

\[ \text{Expr1} >= \text{Expr2} \]

succeeds if the result of evaluating \( \text{Expr1} \) as an arithmetic expression is not less than the result of evaluating \( \text{Expr2} \) as an arithmetic expression.

4.7.5 Arithmetic Expressions

Arithmetic evaluation and testing is performed by predicates that take arithmetic expressions as arguments. An arithmetic expression is a term built from numbers, variables, and functors that represent arithmetic functions. These expressions are evaluated to yield an arithmetic result, which may be either an integer or a float; the type is determined by the rules described below.

At the time of evaluation, each variable in an arithmetic expression must be bound to a number or another arithmetic expression. If the expression is not sufficiently bound or if
it is bound to terms of the wrong type, then Prolog raises exceptions of the appropriate type (see Section 4.15.3 [ref-ere-hex], page 198). Some arithmetic operations can also detect overflows. They also raise exceptions, e.g. division by zero results in an evaluation error being raised.

Only certain functors are permitted in arithmetic expressions. These are listed below, together with a description of their arithmetic meanings. For the rest of the section, X and Y are considered to be arithmetic expressions. Unless stated otherwise, the arguments of an expression may be any numbers and its value is a float if any of its arguments is a float; otherwise, the value is an integer. Any implicit coercions are performed with the integer/1 and float/1 functions. All trigonometric and transcendental functions take float arguments and deliver float values. The trigonometric functions take arguments or deliver values in radians.

Integers can for all practical purposes be considered to be of infinite size. Negative integers can be considered to be infinitely sign extended.

The arithmetic functors are annotated with ISO, with the same meaning as for the built-in predicates; see Section 1.5 [ISO Compliance], page 6.

\[
\begin{align*}
+X & \quad \text{The value is } X. \\
-X & \quad \text{The value is the negative of } X. \\
X+Y & \quad \text{The value is the sum of } X \text{ and } Y. \\
X-Y & \quad \text{The value is the difference between } X \text{ and } Y. \\
X\times Y & \quad \text{The value is the product of } X \text{ and } Y. \\
X/Y & \quad \text{The value is the float quotient of } X \text{ and } Y. \\
X\div Y & \quad \text{The value is the integer quotient of } X \text{ and } Y, \text{ truncated towards zero. } X \text{ and } Y \text{ have to be integers.} \\
X \text{ rem } Y & \quad \text{The value is the integer remainder after truncated division of } X \text{ by } Y, \text{ i.e. } X-Y\times(X\div Y). \text{ The sign of a nonzero remainder will thus be the same as that of the dividend. } X \text{ and } Y \text{ have to be integers.}
\end{align*}
\]
$X \mod Y$

The value is the *integer* remainder after floored division of $X$ by $Y$, i.e. $X - Y \times (X \div Y)$. The sign of a nonzero remainder will thus be the same as that of the divisor. $X$ and $Y$ have to be integers.

**integer($X$)**

The value is the closest integer between $X$ and 0, if $X$ is a float; otherwise, $X$ itself.

**float_integer_part($X$)**

The same as float(integer($X$)). $X$ has to be a float.

**float_fractional_part($X$)**

The value is the fractional part of $X$, i.e. $X - \text{float_integer_part}(X)$.

$X$ has to be a float.

**float($X$)**

The value is the float equivalent of $X$, if $X$ is an integer; otherwise, $X$ itself.

$X \land Y$

The value is the bitwise conjunction of the integers $X$ and $Y$. $X$ and $Y$ have to be integers, treating negative integers as infinitely sign extended.

$X \lor Y$

The value is the bitwise disjunction of the integers $X$ and $Y$. $X$ and $Y$ have to be integers, treating negative integers as infinitely sign extended.

**xor($X,Y$)**

The value is the bitwise exclusive or of the integers $X$ and $Y$. $X$ and $Y$ have to be integers, treating negative integers as infinitely sign extended.

$X \bmod Y$

The same as xor($X,Y$).

**(X)**

The value is the bitwise negation of the integer $X$. $X$ has to be an integer, treating negative integers as infinitely sign extended.

**X\ll Y**

The value is the integer $X$ shifted arithmetically left by $Y$ places. i.e. filling with a copy of the sign bit. $X$ and $Y$ have to be integers, and $Y$ can be negative, in which case the shift is right.

**X\gg Y**

The value is the integer $X$ shifted arithmetically right by $Y$ places, i.e. filling with a copy of the sign bit. $X$ and $Y$ have to be integers, and $Y$ can be negative, in which case the shift is left.

$\lfloor X \rfloor$

A list of just one number $X$ evaluates to $X$. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its character code; e.g. "$A$" behaves within arithmetic expressions as the integer 65.
abs(X)

The value is the absolute value of X.

sign(X)

The value is the sign of X, i.e. -1, if X is negative, 0, if X is zero, and 1, if X is positive, coerced into the same type as X (i.e. the result is an integer, if and only if X is an integer).

gcd(X, Y)

The value is the greatest common divisor of the two integers X and Y. X and Y have to be integers.

min(X, Y)

The value is the lesser value of X and Y.

max(X, Y)

The value is the greater value of X and Y.

msb(X)

The value is the position of the most significant nonzero bit of the integer X, counting bit positions from zero. It is equivalent to, but more efficient than, integer(log(2, X)). X must be greater than zero, and X has to be an integer.

round(X)

The value is the closest integer to X. If X is exactly half-way between two integers, then it is rounded up (i.e. the value is the least integer greater than X).

truncate(X)

The value is the closest integer between X and 0.

floor(X)

The value is the greatest integer less or equal to X.

ceiling(X)

The value is the least integer greater or equal to X.

sin(X)

The value is the sine of X.

cos(X)

The value is the cosine of X.

tan(X)

The value is the tangent of X.

cot(X)

The value is the cotangent of X.

sinh(X)

The value is the hyperbolic sine of X.

cosh(X)

The value is the hyperbolic cosine of X.
\texttt{tanh(X)}

The value is the hyperbolic tangent of \(X\).

\texttt{coth(X)}

The value is the hyperbolic cotangent of \(X\).

\texttt{asin(X)}

The value is the arc sine of \(X\).

\texttt{acos(X)}

The value is the arc cosine of \(X\).

\texttt{atan(X)}

The value is the arc tangent of \(X\).

\texttt{atan2(X,Y)}

The value is the four-quadrant arc tangent of \(X\) and \(Y\).

\texttt{acot(X)}

The value is the arc cotangent of \(X\).

\texttt{acot2(X,Y)}

The value is the four-quadrant arc cotangent of \(X\) and \(Y\).

\texttt{asinh(X)}

The value is the hyperbolic arc sine of \(X\).

\texttt{acosh(X)}

The value is the hyperbolic arc cosine of \(X\).

\texttt{atanh(X)}

The value is the hyperbolic arc tangent of \(X\).

\texttt{acoth(X)}

The value is the hyperbolic arc cotangent of \(X\).

\texttt{sqrt(X)}

The value is the square root of \(X\).

\texttt{log(X)}

The value is the natural logarithm of \(X\).

\texttt{log(Base,X)}

The value is the logarithm of \(X\) in the base \(Base\).

\texttt{exp(X)}

The value is the natural exponent of \(X\).

\texttt{X ** Y}

The value is \(X\) raised to the power of \(Y\), represented as a float. In particular, the value of \(0.0 ** 0.0\) is \(1.0\).

\texttt{exp(X,Y)}

The same as \(X ** Y\).
\( X \cdot Y \)

The value is \( X \) raised to the power of \( Y \), represented as a float if any of \( X \) and \( Y \) is a float; otherwise, as an integer. In particular, the value of \( 0^0 \) is 1.

\( \pi \)

The value is approximately 3.14159.

The following operation is included in order to allow integer arithmetic on character codes.

\[ [X] \]

Evaluates to \( X \) for numeric \( X \). This is relevant because character strings in Prolog are lists of character codes, that is, integers. Thus, for those integers that correspond to character codes, the user can write a string of one character in place of that integer in an arithmetic expression. For example, the expression (A) is equivalent to (B), which in turn becomes (C) in which case \( X \) is unified with 2:

\[
\begin{align*}
X & \text{ is } "c" - "a" & \text{(A)} \\
X & \text{ is } [99] - [97] & \text{(B)} \\
X & \text{ is } 99 - 97 & \text{(C)}
\end{align*}
\]

A cleaner way to do the same thing is

\[
X \text{ is } 0\'c - 0\'a
\]

4.7.6 Predicate Summary

\(-Y = +X\)

\( Y \) is the value of arithmetic expression \( X \)

\(+X =: = +Y\)

the results of evaluating terms \( X \) and \( Y \) as arithmetic expressions are equal.

\(+X =\not=: = +Y\)

the results of evaluating terms \( X \) and \( Y \) as arithmetic expressions are not equal.

\(+X < +Y\)

the result of evaluating \( X \) as an arithmetic expression is less than the result of evaluating \( Y \) as an arithmetic expression.

\(+X >= +Y\)

the result of evaluating \( X \) as an arithmetic expression is not less than the result of evaluating \( Y \) as an arithmetic expression.

\(+X > +Y\)

the result of evaluating \( X \) as an arithmetic expression \( X \) is greater than the result of evaluating \( Y \) as an arithmetic expression.

\(+X =< +Y\)

the result of evaluating \( X \) as an arithmetic expression is not greater than the result of evaluating \( Y \) as an arithmetic expression.
4.8 Looking at Terms

4.8.1 Meta-logical Predicates

Meta-logical predicates are those predicates that allow you to examine the current instantiation state of a simple or compound term, or the components of a compound term. This section describes the meta-logical predicates as well as others that deal with terms as such.

4.8.1.1 Type Checking

The following predicates take a term as their argument. They are provided to check the type of that term. The reference pages for these predicates include examples of their use.

atom(+T)  
   term T is an atom

atomic(+T)  
   term T is an atom or a number

callable(+T)  
   \( T \) is an atom or a compound term

compound(+T)  
   \( T \) is a compound term

db_reference(+X)  
   \( X \) is a db_reference  
   since release 4.1

float(+N)  
   \( N \) is a floating-point number

ground(+T)  
   term T is a nonvar, and all substructures are nonvar

integer(+T)  
   term T is an integer

mutable(+X)  
   \( X \) is a mutable term

nonvar(+T)  
   term T is one of atom, number, compound (that is, T is instantiated)

number(+N)  
   \( N \) is an integer or a float

simple(+T)  
   \( T \) is not a compound term; it is either atomic or a var

var(+T)  
   term T is a variable (that is, T is uninstantiated)

4.8.1.2 Unification

The following predicates are related to unification. Unless mentioned otherwise, unification is performed without occurs check (see Section 4.2.7 [ref-sem-occ], page 78).
To unify two terms, simply use:

?- X = Y.

Please note:

- Do Not confuse this predicate with =:=/2 (arithmetic comparison) or ==/2 (term identity).
- =/2 binds free variables in X and Y in order to make them identical.

To unify two terms with occurs check, use:

?- unify_with_occurs_check(X,Y).

To check whether two terms do not unify, use the following, which is equivalent to \+ (X=Y):

?- X \= Y.

To check whether two terms are either strictly identical or do not unify, use the following. This construct is useful in the context of when/2:

?- ?=(X,Y).

To constrain two terms to not unify, use the following. It blocks until ?=(X,Y) holds:

?- dif(X,Y).

The goal:

?- subsumes_term(General,Specific).

is true when Specific is an instance of General. It does not bind any variables.

4.8.2 Analyzing and Constructing Terms

The built-in predicate functor/3:

- decomposes a given term into its name and arity, or
- given a name and arity, constructs the corresponding compound term creating new uninstantiated variables for its arguments.

The built-in predicate arg/3 unifies a term with a specified argument of another term.

The built-in predicate Term =.. List unifies List with a list whose head is the atom corresponding to the principal functor of Term and whose tail is a list of the arguments of Term.

The built-in predicate acyclic_term/2(Term) succeeds if and only if Term is finite (acyclic).
The built-in predicate \texttt{term_variables(Term,Variables)} unifies Variables with the set of variables that occur in Term, in first occurrence order.

### 4.8.3 Analyzing and Constructing Lists

To combine two lists to form a third list, use \texttt{append(Prefix, Suffix, List)}.

To analyze a list into its component lists in various ways, use \texttt{append/3} with \texttt{List} instantiated to a proper list. The reference page for \texttt{append/3} includes examples of its usage, including backtracking.

To check the length of a list call \texttt{length(List, Length)}.

To produce a list of a certain length, use \texttt{length/2} with \texttt{Length} instantiated and \texttt{List} partly uninstantiated.

To check if a term is the element of a list, use \texttt{memberchk(Element, List)}.

To enumerate the elements of a list via backtracking, use \texttt{member(Element, List)}.

To check that a term is NOT the element of a list, use \texttt{nonmember(Element, List)}, which is equivalent to \texttt{\+	ext{member(Element, List)}}.

### 4.8.4 Converting between Constants and Text

Three predicates convert between constants and lists of character codes: \texttt{atom_codes/2}, \texttt{number_codes/2}, and \texttt{name/2}. Two predicates convert between constants and lists of character atoms: \texttt{atom_chars/2}, \texttt{number_chars/2}.

\texttt{atom_codes(Atom, Codes)} is a relation between an atom Atom and a list Codes consisting of the character codes comprising the printed representation of Atom. Initially, either Atom must be instantiated to an atom, or Codes must be instantiated to a proper code list.

\texttt{number_codes(Number, Codes)} is a relation between a number Number and a list Codes consisting of the character codes comprising the printed representation of Number. Initially, either Number must be instantiated to a number, or Codes must be instantiated to a proper code list.

Similarly, \texttt{atom_chars(Atom, Chars)} and \texttt{number_chars(Atom, Chars)} are relations between a constant and a list consisting of the character atoms comprising the printed representation of the constant.

\texttt{name/2} converts between a constant and a code list. Given a code list, \texttt{name/2} will convert it to a number if it can, otherwise to an atom. This means that there are atoms that can be constructed by \texttt{atom_codes/2} but not by \texttt{name/2}. \texttt{name/2} is retained for backwards compatibility with other Prologs. New programs should use \texttt{atom_codes/2} or \texttt{number_codes/2} as appropriate.

\texttt{char_code/2} converts between a character atom and a character code.
4.8.5 Atom Operations

To compute Length, the number of characters of the atom Atom, use:

?- atom_length(Atom,Length).

To concatenate Atom1 with Atom2 giving Atom12, use the following. The predicate can also be used to split a given Atom12 into two unknown parts:

?- atom_concat(Atom1,Atom2,Atom12).

To extract a sub-atom SubAtom from Atom, such that the number of characters preceding SubAtom is Before, the number of characters after SubAtom is After, and the length of SubAtom is Length, use the following. Only Atom needs to be instantiated:

?- sub_atom(Atom,Before,Length,After,SubAtom).

4.8.6 Assigning Names to Variables

Each variable in a term is instantiated to a term of the form 'VAR'(N), where N is an integer, by the predicate numbervars/3. The “write” predicates (write/[1,2], writeq/[1,2], and write_term/[2,3] with the numbervars(true) option) transform these terms into variable names starting with upper case letters.

4.8.7 Copying Terms

The meta-logical predicate copy_term/2 makes a copy of a term in which all variables have been replaced by brand new variables, and all mutables by brand new mutables. This is precisely the effect that would have been obtained from the definition:

\[
\text{copy_term(Term, Copy) :-}
\text{recorda(copy, copy(Term), DBref),}
\text{instance(DBref, copy(Temp))),}
\text{erase(DBref),}
\text{Copy = Temp.}
\]

although the built-in predicate copy_term/2 is more efficient.

When you call clause/[2,3] or instance/2, you get a new copy of the term stored in the database, in precisely the same sense that copy_term/2 gives you a new copy. One of the uses of copy_term/2 is in writing interpreters for logic-based languages; with copy_term/2 available you can keep “clauses” in a Prolog data structure and pass this structure as an argument without having to store the “clauses” in the Prolog database. This is useful if the set of “clauses” in your interpreted language is changing with time, or if you want to use clever indexing methods.

A naive way to attempt to find out whether one term is a copy of another is shown in this example:
identical_but_for_variables(X, Y) :-
  \+ \+ (numbervars(X, 0, N), numbervars(Y, 0, N), X = Y).

This solution is sometimes sufficient, but will not work if the two terms have any variables in common. If you want the test to succeed even when the two terms do have some variables in common, then you need to copy one of them; for example,

identical_but_for_variables(X, Y) :-
  \+ \+ (copy_term(X, Z), numbervars(Z, 0, N), numbervars(Y, 0, N), Z = Y).

Please note: If the term being copied contains attributed variables (see Section 10.3 [lib-atts], page 391) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 74), then those attributes are not retained in the copy. To retain the attributes, you can use:

copy_term(Term, Copy, Body)

which in addition to copying the term unifies Body with a goal such that executing Body will reinstate the attributes in the Copy. Copy as well as Body contain brand new (unattributed) variables only.

4.8.8 Comparing Terms

4.8.8.1 Introduction

The predicates described in this section are used to compare and order terms, rather than to evaluate or process them. For example, these predicates can be used to compare variables; however, they never instantiate those variables. These predicates should not be confused with the arithmetic comparison predicates (see Section 4.7.4 [ref-ari-acm], page 120) or with unification.

4.8.8.2 Standard Order of Terms

These predicates use a standard total order when comparing terms. The standard total order is:

- Variables, by age (oldest first—the order is not related to the names of variables).
- Floats, in numeric order (e.g. -1.0 is put before 1.0).
- Integers, in numeric order (e.g. -1 is put before 1).
- Atoms, in alphabetical (i.e. character code) order.
- Compound terms, ordered first by arity, then by the name of the principal functor,
then by age for mutables and by the arguments in left-to-right order for other terms. Recall that lists are equivalent to compound terms with principal functor ./2.

For example, here is a list of terms in standard order:

\[
[ X, -1.0, -9, 1, \text{fie, foe, } X = Y, \text{foe}(0,2), \text{fie}(1,1,1) ]
\]

**Please note**: the standard order is only well-defined for finite (acyclic) terms. There are infinite (cyclic) terms for which no order relation holds. Furthermore, blocking goals (see Section 4.2.4 [ref-sem-sec], page 74) on variables or modifying their attributes (see Section 10.3 [lib-atts], page 391) does not preserve their order.

The predicates for comparison of terms are described below.

\[+T1 == +T2\]

\[T1\] and \[T2\] are literally identical (in particular, variables in equivalent positions in the two terms must be identical).

\[+T1 \not== +T2\]

\[T1\] and \[T2\] are not literally identical.

\[+T1 @< +T2\]

\[T1\] is before term \[T2\] in the standard order.

\[+T1 @> +T2\]

\[T1\] is after term \[T2\]

\[+T1 @=< +T2\]

\[T1\] is not after term \[T2\]

\[+T1 @>= +T2\]

\[T1\] is not before term \[T2\]

\[\text{compare}(-Op, +T1, +T2)\]

the result of comparing terms \[T1\] and \[T2\] is \[Op\], where the possible values for \[Op\] are:

\[=\]
if \[T1\] is identical to \[T2\],

\[<\]
if \[T1\] is before \[T2\] in the standard order,

\[>\]
if \[T1\] is after \[T2\] in the standard order.

### 4.8.8.3 Sorting Terms

Two predicates, \texttt{sort/2} and \texttt{keysort/2} sort lists into the standard order. \texttt{keysort/2} takes a list consisting of key-value pairs and sorts according to the key.

Further sorting predicates are available in \texttt{library(samsort)}.

### 4.8.9 Mutable Terms

One of the tenets of logic programming is that terms are immutable objects of the Herbrand universe, and the only sense in which they can be modified is by means of instantiating
non-ground parts. There are, however, algorithms where destructive assignment is essential for performance. Although alien to the ideals of logic programming, this feature can be defended on practical grounds.

SICStus Prolog provides an abstract datatype and four operations for efficient backtrackable destructive assignment. In other words, any destructive assignments are transparently undone on backtracking. Modifications that are intended to survive backtracking must be done by asserting or retracting dynamic program clauses instead. Unlike previous releases of SICStus Prolog, destructive assignment of arbitrary terms is not allowed.

A mutable term is represented as a compound term with a reserved functor: '$_mutable'(Value,Timestamp) where Value is the current value and Timestamp is reserved for bookkeeping purposes [Aggoun & Beldiceanu 90].

Any copy of a mutable term created by copy_term/[2,3], assert, retract, a database predicate, or an all solutions predicate, is an independent copy of the original mutable term. Any destructive assignment done to one of the copies will not affect the other copy.

The following operations are provided:

create_mutable(+Datum,-mutable)
  Datum.

get_mutable(-Datum,+mutable)
  The current value of the mutable term Datum is Datum.

update_mutable(+Datum,+mutable)
  Updates the current value of the mutable term Datum to become Datum.

mutable(+mutable)
  X is currently instantiated to a mutable term.

Please note: the effect of unifying two mutables is undefined.

4.8.10 Summary of Predicates

atom(+T)  ISO
  term T is an atom
atomic(+T)  ISO
  term T is an atom or a number
callable(+T)  ISO
  T is an atom or a compound term
compound(+T)  ISO
  T is a compound term

db_reference(+X)  since release 4.1
  X is a db Reference

float(+N)  ISO
  N is a floating-point number
ground(+T)  
- term T is a nonvar, and all substructures are nonvar

integer(+T)  
- term T is an integer

mutable(+X)  
- X is a mutable term

nonvar(+T)  
- term T is one of atom, number, compound (that is, T is instantiated)

number(+N)  
- N is an integer or a float

simple(+T)  
- T is not a compound term; it is either atomic or a var

var(+T)  
- term T is a variable (that is, T is uninstantiated)

compare(-C,+X,+Y)  
- C is the result of comparing terms X and Y
  
- +X == +Y  
  - terms X and Y are strictly identical

- +X \== +Y  
  - terms X and Y are not strictly identical

- +X @< +Y  
  - term X precedes term Y in standard order for terms

- +X @>= +Y  
  - term X follows or is identical to term Y in standard order for terms

- +X @> +Y  
  - term X follows term Y in standard order for terms

- +X @=< +Y  
  - term X precedes or is identical to term Y in standard order for terms

- ?T =.. ?L  
  - the functor and arguments of term T comprise the list L.

- ?X = ?Y  
  - terms X and Y are unified.

- +X \== +Y  
  - terms X and Y no not unify.

- ?=(+X,+Y)  
  - X and Y are either strictly identical or do not unify.

acyclic_term(+T)  
- term T is a finite (acyclic) term.
arg(+N,+T,-A)  
the Nth argument of term T is A.

atom_chars(?A,?L)  
A is the atom containing the character atoms in list L.

atom_codes(?A,?L)  
A is the atom containing the characters in code list L.

atom_concat(?Atom1,?Atom2,?Atom12)  
Atom Atom1 concatenated with Atom2 gives Atom12.

atom_length(+Atom,-Length)  
Length is the number of characters of the atom Atom.

char_code(?Char,?Code)  
Code is the character code of the one-char atom Char.

copy_term(+T,-C)  
C is a copy of T in which all variables have been replaced by new variables.

copy_term(+T,-C,-G)  
C is a copy of T in which all variables have been replaced by new variables, and G is a goal for reinstating any attributes in C.

create_mutable(+Datum,-Mutable)  
Mutable is a new mutable term with current value Datum.

dif(+X,+Y)  
X and Y are constrained to be different.

frozen(+Term,-Goal)  
Goal is the conjunction of all goals blocked on some variable in Term.

functor(?T,?F,?N)  
the principal functor of term T has name F and arity N.

get_mutable(-Datum,+Mutable)  
The current value of the mutable term Mutable is Datum.

name(?A,?L)  
the code list of atom or number A is L.

number_chars(?N,?L)  
N is the numeric representation of list of character atoms L.

number_codes(?N,?L)  
N is the numeric representation of code list L.

numbervars(+T,+M,-N)  
number the variables in term T from M to N-1.

sub_atom(+Atom,+Before,?Length,+After,+SubAtom)  
The characters of SubAtom form a sublist of the characters of Atom, such that the number of characters preceding SubAtom is Before, the number of characters after SubAtom is After, and the length of SubAtom is Length.
subsumes_term(General, Specific)  
since release 4.3, ISO

Specific is an instance of General.

term_variables(+Term, -Variables)  
since release 4.3, ISO

Variables is the set of variables that occur in Term, in first occurrence order.

unify_with_occurs_check(?X, ?Y)  
ISO

True if X and Y unify to a finite (acyclic) term.

?T =.. ?L  
ISO

the functor and arguments of term T comprise the list L

append(?A, ?B, ?C)  
ISO

the list C is the concatenation of lists A and B

keysort(+L, -S)  
ISO

the list L sorted by key yields S

length(?L, ?N)  
ISO

the length of list L is N

member(?X, ?L)  
ISO

X is a member of L

memberchk(+X, +L)  
ISO

X is a member of L

nonmember(+X, +L)  
ISO

X is not a member of L

sort(+L, -S)  
ISO

sorting the list L into order yields S

4.9 Looking at the Program State

4.9.1 Overview

Various aspects of the program state can be inspected: the clauses of all or selected dynamic procedures, currently available atoms, user defined predicates, source files of predicates and clauses, predicate properties and the current load context can all be accessed by calling the predicates listed in Section 4.9.1 [ref-lps-ove], page 135. Furthermore, the values of Prolog flags can be inspected and, where it makes sense, changed.

4.9.2 Associating Predicates with their Properties

The following properties are associated with predicates either implicitly or by declaration:

built_in  The predicate is built-in.

compiled  The predicate is in virtual code representation.

interpreted  The predicate is in interpreted representation.

fd_constraint  The predicate is a so-called FD predicate; see Section 10.10.10 [Defining Primitive Constraints], page 485.
**dynamic** The predicate was declared dynamic.

**volatile** The predicate was declared volatile.

**multifile**

The predicate was declared multifile.

**block(SkeletalGoal)**

The predicate has block declarations.

**meta_predicate(SkeletalGoal)**

The predicate is a meta-predicate.

As of release 4.2, the *SkeletalGoal* will contain the specifications used in the original meta-predicate declaration.

**exported** The predicate was exported from a module.

**imported_from(Module)**

The predicate was imported from the module *Module*.

Every predicate has exactly one of the properties [built_in, compiled, interpreted, fd_constraint], at most one of the properties [exported, imported_from(Module)], zero or more block(SkeletalGoal) properties, and at most one of the remaining properties.

To query these associations, use predicate_property/2. The reference page contains several examples. See Section 11.3.159 [mpg-ref-predicate_property], page 1132.

### 4.9.3 Associating Predicates with Files

Information about loaded files and the predicates and clauses in them is returned by source_file/[1,2]. source_file/1 can be used to identify an absolute filename as loaded, or to backtrack through all loaded files. To find out the correlation between loaded files and predicates, call source_file/2. See Section 11.3.215 [mpg-ref-source_file], page 1201.

### 4.9.4 Prolog Flags

Certain aspects of the state of the program are accessible as values of the global Prolog flags. Some of these flags are read-only and correspond to implementation defined properties and exist to aid portability. Others can be set and impact the behavior of certain built-in predicates.

The flags are accessed by the built-in predicates prolog_flag/[2,3], current_prolog_flag/2, and set_prolog_flag/2.

**Please note:** Prolog flags are *global*, as opposed to being local to the current module, Prolog text, or otherwise.

The possible Prolog flag names and values are listed below. Flags annotated *ISO* are prescribed by the ISO standard. Flags annotated *volatile* are not saved by save_program/[1,2]. Flags annotated *read-only* are read-only:
agc_margin

An integer Margin. The atoms will be garbage collected when Margin new atoms have been created since the last atom garbage collection. Initially 10000.

argv volatile

The value is a list of atoms of the program arguments supplied when the current SICStus Prolog process was started. For example, if SICStus Prolog were invoked with:

% sicstus -- hello world 2001

then the value will be [hello, world, '2001'].

Setting the value can be useful when writing test cases for code that expects to be run with command line parameters.

bounded read-only, volatile, ISO

One of the flags defining the integer type. For SICStus, its value is false, indicating that the domain of integers is practically unbounded.

char_conversion volatile, ISO

If this flag is on, then unquoted characters in terms and programs read in will be converted, as specified by previous invocations of char_conversion/2. If the flag is off, then no conversion will take place. The default value is on.

compiling

Governs the mode in which compile/1 operate (see Section 4.3 [ref-lod], page 79).

compactcode

Compilation produces byte-coded abstract instructions (the default).

debugcode

Compiling is replaced by consulting.

debugging volatile

Corresponds to the predicates debug/0, nodebug/0, trace/0, notrace/0, zip/0, nozip/0. The flag describes the mode the debugger is in, or is required to be switched to:

trace Trace mode (the debugger is creeping).
debug Debug mode (the debugger is leaping).
zip Zip mode (the debugger is zipping).
off The debugger is switched off (the default).

debug volatile, ISO

The flag debug, prescribed by the ISO Prolog standard, is a simplified form of the debugging flag:

off ISO The debugger is switched off (the default).
on ISO The debugger is switched on (to trace mode, if previously switched off).
profiling  
This flag describes the mode the execution profiler (see Section 9.2 [Execution Profiling], page 355) is in, or is required to be switched to:

off  The profiler is switched off (the default).
on  The profiler is switched on.

(The flags profiling, debugging and debug have no effect in runtime systems.)

double_quotes  
Governs the interpretation of double quoted strings (see Section 4.1.3.2 [ref-syn-cpt-sli], page 46):

codes  Code list comprising the string. The default.  
chars  Char list comprising the string.
atom  The atom composed of the same characters as the string.

quoted_charset  
This flag is relevant when quoted(true) holds when writing terms. Its value should be one of the atoms:

portable  Atoms and functors are written using character codes less than 128 only, i.e. using the 7-bit subset of the ISO-8859-1 (Latin 1) character set (see Section 4.1.7.5 [ref-syn-syn-tok], page 56).

prolog  Atoms and functors are written using a character set that can be read back by read/[1,2]. This is a subset of Unicode that includes all of ISO-8859-1 (Latin 1) as well as some additional characters. This character set may grow but not shrink in subsequent releases. This ensures that future releases can always read a term written by an older release.

Note that the character set supported by the stream is not taken into account. You can use portable instead of prolog if the stream does not support Unicode.

debugger_print_options  
The value is a list of options for write_term/3 (see Section 4.6.4.1 [ref-iou-tou-wrt], page 104), to be used in the debugger’s messages. The initial value is [quoted(true), numbervars(true), portrayed(true), max_depth(10)].

dialect  
The value of this flag is sicstus. It is useful for distinguishing between Prolog implementations.

Also see the Prolog flag version_data, below.

discontiguous_warnings  
Enable or disable warning messages when clauses are not together in source files. Initially on in development systems, off in runtime systems.
fileerrors
- **on** or **off**. Enables or disables raising of file error exceptions. Initially **on** (enabled).

gc
- **on** or **off**. Enables or disables garbage collection of the global stack. Initially **on** (enabled).

gc_margin
- **Margin**: At least Margin kilobytes of free global stack space are guaranteed to exist after a garbage collection. Also, no garbage collection is attempted if the global stack has grown less than Margin kilobytes since the last garbage collection. Initially 1000.

gc_trace
- Governs global stack garbage collection trace messages.
  - **verbose** Turn on verbose tracing of garbage collection.
  - **terse** Turn on terse tracing of garbage collection.
  - **off** Turn off tracing of garbage collection (the default).

host_type
- The value is an atom identifying the platform on which SICStus was compiled, such as `'x86-linux-glibc2.1'` or `'sparc-solaris-5.7'`.

informational
- **volatile**
- **on** or **off**. Enables or disables the printing of informational messages. Initially **on** (printing enabled) in development systems, unless the `--noinfo` command line option was used; **off** (printing disabled) in runtime systems.

integer_rounding_function
- **read-only,** **volatile,** **ISO**
- One of the flags defining the integer type. In SICStus Prolog its value is **toward_zero**, indicating that the integer division (`//`) and integer remainder (`rem/2`) arithmetic functions use rounding toward zero; see Section 4.7 [ref-ari], page 119.

legacy_char_classification
- **since release 4.0.3,** **volatile**
- **on** or **off**. When enabled, most legal Unicode codepoints above 255 are treated as lowercase characters when reading Prolog terms. This improves compatibility with earlier versions of SICStus Prolog and makes it possible to use full Unicode, e.g., Chinese characters, in unquoted atoms as well as variable names, Section 4.1.7.5 [ref-syn-syn-tok], page 56. Initially **off** (disabled).

Setting this flag affects the `read_term/[2,3]` option `singletons/1`; see Section 11.3.185 [mpg-ref-read_term], page 1163. It also affects the style warning for singleton variables; see the description of the `single_var_warnings` in Section 4.3.2 [ref-lod-lod], page 80.

max_arity
- **read-only,** **volatile,** **ISO**
- Specifies the maximum arity allowed for a compound term. In SICStus Prolog this is 255.

max_integer
- **read-only,** **volatile,** **ISO**
- Specifies the largest possible integer value. As in SICStus Prolog the range of integers in not bounded, `prolog_flag/[2,3]` and `current_prolog_flag/2` will fail when accessing this flag.
max_tagged_integer  
**since release 4.1, read-only, volatile**  
The largest small integer, i.e. integers larger than this are less efficient to manipulate and are not available in `library(clpfd)`.

min_integer  
**read-only, volatile, ISO**  
Specifies the smallest possible integer value. As in SICStus Prolog the range of integers in not bounded, `prolog_flag/[2,3]` and `current_prolog_flag/2` will fail, when accessing this flag.

min_tagged_integer  
**since release 4.1, read-only, volatile**  
The smallest small integer, i.e. integers smaller than this are less efficient to manipulate and are not available in `library(clpfd)`.

os_data  
**since release 4.1, read-only, volatile**  
The value is a term `os(Family,Name,Extra)` describing the operating system on which this SICStus process is running, i.e. it is the runtime version of the `platform_data` flag, below.

*Family* has the same value and meaning as for the `platform_data` flag, below. On UNIX-like systems the *Name* is the lower case value `sysname` returned from `uname(3)` at runtime, i.e. the same as from the command `uname -s`. On all supported versions of Microsoft Windows this is `win32nt`.

*Extra* is a list of extra information. Entries may be added to this list without prior notice.

Currently, at least up to release 4.2.3, the *Family* and *Name* for the `platform_data` and `os_data` flags happens to be the same but this may change in the unlikely case that the operating system starts to return something new. For this reason it is probably better to use `platform_data` than `os_data` in most cases.

The *Extra* value for `os_data` may differ from its `platform_data` counterpart in order to accurately describe the running operating system.

platform_data  
**since release 4.1, read-only, volatile**  
The value is a term `platform(Family,Name,Extra)` describing the operating system platform for which this version of SICStus was built.

*Family* describes the family or class of operating system. Currently documented values are `unix`, for UNIX-like systems like Linux, OS X, Solaris and Android; and `windows` for all supported versions of Microsoft Windows. You should not assume that these are the only two possibilities.

*Name* describes the name of the operating system. On UNIX-like systems this correspond to the (lower case) output from `uname -s`. Currently documented values are `linux`, `darwin`, `sunos`, `android` and `win32nt`.

Note that this implies that some operating systems may have unexpected names. In particular the name for Apple OS X is `darwin`, for Oracle Solaris it is `sunos` and for 64-bit versions of SICStus on Microsoft Windows it is `win32nt`.

*Extra* is bound to a list of extra information. Entries may be added to this list without prior notice.
**redefine_warnings**

Enable or disable warning messages when:

- a module or predicate is being redefined from a different file than its previous definition. Such warnings are currently not issued when a `.po` file is being loaded.
- a predicate is being imported while it was locally defined already.
- a predicate is being redefined locally while it was imported already.
- a predicate is being imported while it was imported from another module already.

The possible values are:

- **on** The default in development systems. The user is queried about what to do in each case.
- **off** The default in runtime systems, but note that this flag is not volatile. Redefinitions are performed silently, as if the user had accepted them.
- **reject** since release 4.0.3 Redefinitions are refused silently, as if the user had rejected them.
- **proceed** since release 4.0.3 Redefinitions are performed, and warnings are issued.
- **suppress** since release 4.0.3 Redefinitions are refused, and warnings are issued.

**single_var_warnings** volatile

**on** or **off**. Enable or disable warning messages when a sentence (see Section 4.1.7.3 [ref-syn-syn-sen], page 53) containing variables not beginning with `_` occurring once only is compiled or consulted. Initially **on** in development systems, **off** in runtime systems.

**source_info** volatile

**emacs** or **on** or **off**. If not **off** while source code is being loaded, then information about line numbers and file names are stored with the loaded code. If the value is **on** while debugging, then this information is used to print the source code location while prompting for a debugger command. If the value is **on** while printing an uncaught error exception message, then the information is used to print the source code location of the culprit goal or one of its ancestors, as far as it can be determined. If the value is **emacs** in any of these cases, then the appropriate line of code is instead highlighted, and no extra text is printed. The value is **off** initially, and that is its only available value in runtime systems.

**syntax_errors**

Controls what action is taken upon syntax errors in `read/[1,2].`

- **dec10** The syntax error is reported and the read is repeated.
- **error** An exception is raised. See Section 4.15 [ref-ere], page 197. (the default).
fail The syntax error is reported and the read fails.
quiet The read quietly fails.

**system_type** read-only, volatile
The value is development in development systems and runtime in runtime systems.

**title** The window title. The default value is the same as the boot message ‘SICStus 4.7.0 ... Licensed to SICS’. It is currently only used as the window title on the Windows platform.

toplevel_print_options
The value is a list of options for write_term/3 (see Section 4.6.4.1 [ref-iou-tou-wrt], page 104), to be used when the top level displays variable bindings and answer constraints. It is also used when messages are displayed. The initial value is [quoted(true), numbervars(true), portrayed(true), max_depth(10)].

typein_module
Permitted values are atoms. Controls the current type-in module (see Section 4.11.8 [ref-mod-tyi], page 166). Corresponds to the predicate set_module/1.

**unknown** ISO
The system can optionally catch calls to predicates that have no definition. First, the user defined predicate user:unknown_predicate_handler/3 (see Section 4.15 [ref-ere], page 197) is called. If undefined or if the call fails, then the action is governed by the state of the this flag, which can be:

**trace** Causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity. Not available in runtime systems.

**error** ISO
Causes calls to such predicates to raise an exception (the default). See Section 4.15 [ref-ere], page 197.

**warning** ISO
Causes calls to such predicates to display a warning message and then fail.

**fail** ISO Causes calls to such predicates to fail.

**user_input** volatile
Permitted values are any stream opened for reading. Controls which stream is referenced by user_input and SP_stdin. It is initially set to a stream connected to UNIX stdin.

**user_output** volatile
Permitted values are any stream opened for writing. Controls which stream is referenced by user_output and SP_stdout. It is initially set to a stream connected to UNIX stdout.
user_error volatile
Permitted values are any stream opened for writing. Controls which stream is referenced by user_error and SP_stderr. It is initially set to a stream connected to UNIX stderr.

version read-only, volatile
The value is an atom containing the banner text displayed on startup, such as 'SICStus 4.1.0 (i386-darwin-9.8.0): Wed Oct 14 14:43:58 CEST 2009'. Also see the Prolog flag version_data, below.

version_data since release 4.1, read-only, volatile
The value is a term sicstus(Major,Minor,Revision,Beta,Extra) with integer major, minor, revision, and beta version. Extra is bound to a list of extra information. Entries may be added to this list without prior notice.
Also see the Prolog flag dialect, above.

You can use prolog_flag/2 to enumerate all the FlagNames that the system currently understands, together with their current values. Use prolog_flag/2 to make queries, prolog_flag/3 to make changes.

4.9.5 Load Context
When a Prolog source file is being read in, some aspects of the load context can be accessed by the built-in predicate prolog_load_context/2, which accesses the value of a given key. The available keys are:

source The absolute path name of the file being loaded. During loading of a PO file, the corresponding source file name is returned.

file Outside included files (see Section 4.3.4.11 [Include Declarations], page 86) this is the same as the source key. In included files this is the absolute path name of the file being included.

directory The absolute path name of the directory of the file being loaded. In included files this is the directory of the file being included.

module The source module (see Section 4.11.15 [ref-mod-mne], page 171). This is useful for example if you are defining clauses for user:term_expansion/6 and need to access the source module at compile time.

stream The stream being loaded. This key is not available during loading of a PO file.

term_position A term representing the stream position of the last clause read. This key is not available during loading of a PO file.

4.9.6 Predicate Summary

current_atom(?A)
backtrack through all atoms
current_module(?M)
    M is the name of a current module

current_module(?M, ?F)
    F is the name of the file in which M’s module declaration appears

current_predicate(:A/?N)
    ISO
    A is the name of a predicate with most general goal P and arity N

current_predicate(?A, :P)
    ISO
    A is the name of a predicate with most general goal P and arity N

current_prolog_flag(?F, ?V)
    ISO
    V is the current value of Prolog flag F

listing
    list all dynamic procedures in the type-in module

listing(:P)
    list the dynamic procedure(s) specified by P

predicate_property(:P, ?Prop)
    Prop is a property of the loaded predicate P

prolog_flag(?F, ?V)
    V is the current value of Prolog flag F

prolog_flag(+F, =O,+N)
    O is the old value of Prolog flag F; N is the new value

prolog_load_context(?K, ?V)
    find out the context of the current load

set_module(+M)
    make M the type-in module

set_prolog_flag(+F, +N)
    ISO
    N is the new value of Prolog flag F

source_file(?F)
    F is a source file that has been loaded into the database

source_file(:P, ?F)
    P is a predicate defined in the loaded file F

unknown(-O,+N)
    development
    Changes action on undefined predicates from O to N.

4.10 Memory Use and Garbage Collection

4.10.1 Overview

SICStus Prolog uses five data areas: program space, global stack, local stack, choice stack, and trail stack. Each of these areas is automatically expanded if it overflows.

The local stack contains all the control information and variable bindings needed in a Prolog execution. Space on the local stack is reclaimed on determinate success of predicates and by tail recursion optimization, as well as on backtracking.
The choice stack contains data representing outstanding choices for some goals or disjunctions. Space on the choice stack is reclaimed on backtracking.

The global stack contains all the data structures constructed in an execution of the program. This area grows with forward execution and shrinks on backtracking.

The trail stack contains references to all the variables that need to be reset when backtracking occurs. This area grows with forward execution and shrinks on backtracking.

The program space contains compiled and interpreted code, recorded terms, and atoms. The space occupied by compiled code, interpreted code, and recorded terms is recovered when it is no longer needed; the space occupied by atoms that are no longer in use can be recovered by atom garbage collection described in Section 4.10.7 [ref-mgc-ago], page 156.

These fluctuations in memory usage of the above areas can be monitored by statistics/[0,2].

SICStus Prolog uses the global stack to construct compound terms, including lists. Global Stack space is used as Prolog execution moves forward. When Prolog backtracks, it automatically reclaims space on the global stack. However, if a program uses a large amount of space before failure and backtracking occur, then this type of reclamation may be inadequate.

Without garbage collection, the Prolog system must attempt to expand the global stack whenever a global stack overflow occurs. To do this, it first requests additional space from the operating system. If no more space is available, then the Prolog system attempts to allocate unused space from the other Prolog data areas. If additional space cannot be found, then a resource error is raised.

Global stack expansion and abnormal termination of execution due to lack of stack space can occur even if there are structures on the global stack that are no longer accessible to the computation (these structures are what is meant by “garbage”). The proportion of garbage to non-garbage terms varies during execution and with the Prolog code being executed. The global stack may contain no garbage at all, or may be nearly all garbage.

The garbage collector periodically reclaims inaccessible global stack space, reducing the need for global stack expansion and lessening the likelihood of running out of global stack. When the garbage collector is enabled (as it is by default), the system makes fewer requests to the operating system for additional space. The fact that less space is required from the operating system can produce a substantial savings in the time taken to run a program, because paging overhead can be much less.

For example, without garbage collection, compiling a file containing the sequence

\[
p(_) :- p([a]).
\]

causes the global stack to expand until the Prolog process eventually runs out of space. With garbage collection enabled, the above sequence continues indefinitely. The list built
on the global stack by each recursive call is inaccessible to future calls (since p/1 ignores its argument) and can be reclaimed by the garbage collector.

Garbage collection does not guarantee freedom from out-of-space errors, however. Compiling a file containing the sequence

\[
\text{p(X) :- p([X]).} \\
\text{:- p(a).}
\]

expands the global stack until the Prolog process eventually runs out of space. This happens in spite of the garbage collector, because all the terms built on the global stack are accessible to future computation and cannot be reclaimed.

4.10.1.1 Reclaiming Space

\texttt{trimcore/0} reclaims space in all of Prolog’s data areas. At any given time, each data area contains some free space. For example, the local stack space contains the local stack and some free space for that stack to grow into. The data area is automatically expanded when it runs out of free space, and it remains expanded until \texttt{trimcore/0} is called, even though the stack may have shrunk considerably in the meantime. The effect of \texttt{trimcore/0} is to reduce the free space in all the data areas as much as possible, and to endeavor to give the space no longer needed back to the operating system.

The system property \texttt{PROLOGKEEPSIZE} can be used to define a lower bound on the amount of memory to be retained. Also, the system property \texttt{PROLOGINITSIZE} can be used to request that an initial amount of memory be allocated. This initially allocated memory will not be touched by \texttt{trimcore/0}.

When trimming a given stacks, \texttt{trimcore/0} will retain at least the amount of space initially allocated for that stack.

\texttt{trimcore/0} is called each time Prolog returns to the top level or the top of a break level, except it does not trim the stacks then. See Section 11.3.238 [mpg-ref-trimcore], page 1231.

4.10.1.2 Displaying Statistics

Statistics relating to memory usage, run time, and garbage collection, including information about which areas of memory have overflowed and how much time has been spent expanding them, can be displayed by calling \texttt{statistics/0}.

The output from \texttt{statistics/0} looks like this:
memory (total) 3334072 bytes
  global stack 1507184 bytes: 2516 in use, 1504668 free
  local stack 49296 bytes: 276 in use, 49020 free
  trail stack 34758 bytes: 248 in use, 34510 free
  choice stack 34874 bytes: 364 in use, 34510 free
  program space 1707960 bytes: 1263872 in use, 444088 free

program space breakdown:
  compiled code 575096 bytes
  atom 166528 bytes
  predicate 157248 bytes
  try_node 144288 bytes
  sw_on_key 105216 bytes
  incore_info 51096 bytes
  atom table 36864 bytes
  interpreted code 13336 bytes
  atom buffer 2560 bytes
  SP_malloc 2288 bytes
  FLI stack 2048 bytes
  miscellaneous 1640 bytes
  BDD hash table 1560 bytes
  source info (B-tree) 1024 bytes
  numstack 1024 bytes
  int_info 880 bytes
  file table 400 bytes
  source info (itable) 328 bytes
  module 320 bytes
  source info (lheap) 80 bytes
  foreign resource 32 bytes
  all solutions 16 bytes

4323 atoms (151927 bytes) in use, 1044252 free
No memory resource errors

0.020 sec. for 7 global, 20 local, and 0 choice stack overflows
0.060 sec. for 15 garbage collections which collected 5461007 bytes
0.000 sec. for 0 atom garbage collections which collected 0 atoms (0 bytes)
0.000 sec. for 4 defragmentations
0.000 sec. for 7 dead clause reclamations
0.000 sec. for 0 dead predicate reclamations
39.410 sec. runtime

========
39.490 sec. total runtime
109.200 sec. elapsed time

Note the use of indentation to indicate sub-areas. That is, memory contains the program space and the four stacks: global, local, choice, and trail.
The memory (total) figure shown as “in use” is the sum of the spaces for the program space and stacks. The “free” figures for the stacks are for free space within those areas. However, this free space is considered used as far as the memory (total) area is concerned, because it has been allocated to the stacks. The program space is not considered to have its own free space. It always allocates new space from the general memory (total) free area.

If a memory resource error has occurred previously in the execution, then the memory area for which memory could not be allocated is displayed.

Individual statistics can be obtained by `statistics/2`, which accepts a keyword and returns a list of statistics related to that keyword.

The keys and values for `statistics(Keyword, Value)` are summarized below. The keywords `core` and `heap` are included to retain compatibility with other Prologs. Times are given in milliseconds and sizes are given in bytes.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime</td>
<td>[since start of Prolog, since previous statistics]</td>
</tr>
<tr>
<td></td>
<td>These refer to CPU time used while executing, excluding time spent in memory management tasks or in system calls. The second element is the time since the latest call to <code>statistics/2</code> with this key or to <code>statistics/0</code>.</td>
</tr>
<tr>
<td>total_runtime</td>
<td>[since start of Prolog, since previous statistics]</td>
</tr>
<tr>
<td></td>
<td>These refer to total CPU time used while executing, including memory management tasks such as garbage collection but excluding system calls. The second element is the time since the latest call to <code>statistics/2</code> with this key or to <code>statistics/0</code>.</td>
</tr>
<tr>
<td>walltime</td>
<td>[since start of Prolog, since previous statistics]</td>
</tr>
<tr>
<td></td>
<td>These refer to absolute time elapsed. The second element is the time since the latest call to <code>statistics/2</code> with this key or to <code>statistics/0</code>.</td>
</tr>
<tr>
<td>global_stack</td>
<td>[size used, free]</td>
</tr>
<tr>
<td></td>
<td>This refers to the global stack, where compound terms are stored. The values are gathered before the list holding the answers is allocated. Formed from basic values below.</td>
</tr>
<tr>
<td>local_stack</td>
<td>[size used, free]</td>
</tr>
<tr>
<td></td>
<td>This refers to the local stack, where recursive predicate environments are stored. Formed from basic values below.</td>
</tr>
<tr>
<td>trail</td>
<td>[size used, free]</td>
</tr>
<tr>
<td></td>
<td>This refers to the trail stack, where conditional variable bindings are recorded. Formed from basic values below.</td>
</tr>
<tr>
<td>choice</td>
<td>[size used, free]</td>
</tr>
<tr>
<td></td>
<td>This refers to the choice stack, where partial states are stored for backtracking purposes. Formed from basic values below.</td>
</tr>
</tbody>
</table>
memory

core [size used,0]
These refer to the amount of memory actually allocated by the Prolog engine. The zero is there for compatibility with other Prolog implementations. Formed from basic values below.

program heap [size used, size free]
These refer to the amount of memory allocated for the database, symbol tables, and the like. Formed from basic values below.

garbage_collection [no. of GCs, bytes freed, time spent]
Formed from basic values below.

stack_shifts [no. of global shifts, no. of local/choice shifts, time spent]
Formed from basic values below.

atoms [no. of atoms, bytes used, atoms free]
The number of atoms free is the number of atoms allocated (the first element in the list) subtracted from the maximum number of atoms, i.e. 262143 (33554431) on 32-bit (64-bit) architectures. Note that atom garbage collection may be able to reclaim some of the allocated atoms. Formed from basic values below.

atom_garbage_collection [no. of AGCs, bytes freed, time spent]
Formed from basic values below.

defragmentation [no. of defragmentations, time spent]
Formed from basic values below.

memory_used since release 4.1
bytes used

memory_free since release 4.1
bytes free

global_stack_used since release 4.1
bytes used

global_stack_free since release 4.1
bytes free

local_stack_used since release 4.1
bytes used
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Since Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>local_stack_free</td>
<td>bytes free</td>
<td>4.1</td>
</tr>
<tr>
<td>trail_used</td>
<td>bytes used</td>
<td>4.1</td>
</tr>
<tr>
<td>trail_free</td>
<td>bytes free</td>
<td>4.1</td>
</tr>
<tr>
<td>choice_used</td>
<td>bytes used</td>
<td>4.1</td>
</tr>
<tr>
<td>choice_free</td>
<td>bytes free</td>
<td>4.1</td>
</tr>
<tr>
<td>atoms_used</td>
<td>bytes used</td>
<td>4.1</td>
</tr>
<tr>
<td>atoms_nbused</td>
<td>atoms used</td>
<td>4.1</td>
</tr>
<tr>
<td>atoms_nbfree</td>
<td>atoms free</td>
<td>4.1</td>
</tr>
<tr>
<td>ss_global</td>
<td>number of global stack shifts</td>
<td>4.1</td>
</tr>
<tr>
<td>ss_local</td>
<td>number of local stack shifts</td>
<td>4.1</td>
</tr>
<tr>
<td>ss_choice</td>
<td>number of choice stack shifts</td>
<td>4.1</td>
</tr>
<tr>
<td>ss_time</td>
<td>time spent stack shifting</td>
<td>4.1</td>
</tr>
<tr>
<td>gc_count</td>
<td>number of garbage collections</td>
<td>4.1</td>
</tr>
<tr>
<td>gc_freed</td>
<td>number of bytes freed</td>
<td>4.1</td>
</tr>
</tbody>
</table>
gc_time  
    time spent collecting garbage  since release 4.1

agc_count  
    number of atom garbage collections  since release 4.1

agc_nbfreed  
    number of garbage collected atoms  since release 4.1

agc_freed  
    number of bytes freed by atom garbage collected  since release 4.1

agc_time  
    time spent garbage collected atoms  since release 4.1

defrag_count  
    number of memory defragmentations  since release 4.1

deffrag_time  
    time spent defragmenting memory  since release 4.1

dpgc_count  
    number of dead predicate reclaimations  since release 4.1

dpgc_time  
    time spent reclaiming dead predicates  since release 4.1

dcgc_count  
    number of dead clause reclaimations  since release 4.1

dcgc_time  
    time spent reclaiming dead clauses  since release 4.1

memory_culprit  
    memory bucket in which latest memory resource error occurred  since release 4.1

memory_buckets  
    list of bucket-size pair  since release 4.1
    where size is the amount of memory in use for memory bucket bucket.

jit_count  
    number of JIT-compiled predicates  since release 4.3
    This is zero when JIT compilation is not available.
jit_time

*time spent JIT-compiling predicates*

This is zero when JIT compilation is not available.

To see an example of the use of each of these keywords, type

```prolog
| ?- statistics(K, L).
```

and then repeatedly type `;` to backtrack through all the possible keywords. As an additional example, to report information on the runtime of a predicate `p/0`, add the following to your program:

```prolog
:- statistics(runtime, [T0|_]),
p,
statistics(runtime, [T1|_]),
T is T1 - T0,
format('p/0 took ~3d sec.~n', [T]).
```

See Section 11.3.217 [mpg-ref-statistics], page 1204.

### 4.10.2 Garbage Collection and Programming Style

The availability of garbage collection can lead to a more natural programming style. Without garbage collection, a procedure that generates global stack garbage may have to be executed in a failure-driven loop. Failure-driven loops minimize global stack usage from iteration to iteration of a loop via SICStus Prolog’s automatic recovery of global stack space on failure. For instance, in the following procedure `echo/0` echoes Prolog terms until it reads an end-of-file character. It uses a failure-driven loop to recover inaccessible global stack space.

```prolog
echo :- repeat,
    read(Term),
echo_term(Term),
!.
```

```prolog
echo_term(Term) :-
    Term == end_of_file.
echo_term(Term) :-
    writeq(Term), nl,
    fail.
```

Any global stack garbage generated by `read/1` or `write/1` is automatically reclaimed by the failure of each iteration.

Although failure-driven loops are an accepted Prolog idiom, they are not particularly easy to read or understand. So we might choose to write a clearer version of `echo/0` using recursion instead, as in

```prolog
echo :- repeat,
    read(Term),
echo_term(Term),
!.
```

```prolog
echo_term(Term) :-
    Term == end_of_file.
echo_term(Term) :-
    writeq(Term), nl,
    fail.
```
Without garbage collection the more natural recursive loop accumulates global stack garbage that cannot be reclaimed automatically. While it is unlikely that this trivial example will run out of global stack space, larger and more practical applications may be unable to use the clearer recursive style without garbage collection. With garbage collection, all inaccessible global stack space will be reclaimed by the garbage collector.

Using recursion rather than failure-driven loops can improve programming style further. We might want to write a predicate that reads terms and collects them in a list. This is naturally done in a recursive loop by accumulating results in a list that is passed from iteration to iteration. For instance,

\[
\text{collect(List)} : - \\
\text{read(Term),} \\
\text{collect\_term(Term, List).}
\]

\[
\text{collect\_term(Term, [\]) : -} \\
\text{Term == end\_of\_file,} \\
\text{!.} \\
\text{collect\_term(Term, [Term|List0]) : -} \\
\text{collect(List0).}
\]

For more complex applications this sort of construction might prove unusable without garbage collection. Instead, we may be forced to use a failure-driven loop with side effects to store partial results, as in the following much less readable version of \text{collect/1}:
The variable bindings made in one iteration of a failure-driven loop are unbound on failure of the iteration. Thus partial results cannot simply be stored in a data structure that is passed along to the next iteration. We must instead resort to storing partial results via side effects (here, `assertz/1`) and collect (and clean up) partial results in a separate pass. The second example is much less clear to most people than the first. It is also much less efficient than the first. However, if there were no garbage collector, then larger examples of the second type might be able to run where those of the first type would run out of memory.

### 4.10.3 Enabling and Disabling the Garbage Collector

The user has the option of executing programs with or without garbage collection. Procedures that do not use a large amount of global stack space before backtracking may not be affected when garbage collection is enabled. Procedures that do use a large amount of global stack space may execute more slowly due to the time spent garbage collecting, but will be more likely to run to completion. On the other hand, such programs may run faster when the garbage collector is enabled because the virtual memory is not expanded to the extent that “thrashing” occurs. The `gc` Prolog flag can be set to `on` or `off`. To monitor garbage collections in verbose mode, set the `gc_trace` flag to `verbose`. By default, garbage collection is enabled.

### 4.10.4 Monitoring Garbage Collections

By default, the user is given no indication that the garbage collector is operating. If no program ever runs out of space and no program using a lot of global stack space requires an inordinate amount of processing time, then such information is unlikely to be needed.

However, if a program thought to be using much global stack space runs out of space or runs inordinately slowly, then the user may want to determine whether more or less
frequent garbage collections are necessary. Information obtained from the garbage collector by turning on the gc_trace Prolog flag can be helpful in this determination.

4.10.5 Interaction of Garbage Collection and Global Stack Expansion

For most programs, the default settings for the garbage collection parameters should suffice. For programs that have high global stack requirements, the default parameters may result in a higher ratio of garbage collection time to run time. These programs should be given more space in which to run.

The gc_margin is a non-negative integer specifying the desired margin in kilobytes. For example, the default value of 1000 means that the global stack will not be expanded if garbage collection can reclaim at least one megabyte. The advantage of this criterion is that it takes into account both the user’s estimate of the global stack usage and the effectiveness of garbage collecting.

1. Setting the gc_margin higher than the default will cause fewer global stack expansions and garbage collections. However, it will use more space, and garbage collections will be more time-consuming when they do occur.
   Setting the margin too large will cause the global stack to expand so that if it does overflow, then the resulting garbage collection will significantly disrupt normal processing. This will be especially so if much of the global stack is accessible to future computation.

2. Setting the gc_margin lower than the default will use less space, and garbage collections will be less time-consuming. However, it will cause more global stack expansions and garbage collections.
   Setting the margin too small will cause many garbage collections in a small amount of time, so that the ratio of garbage-collecting time to computation time will be abnormally high.

3. Setting the margin correctly will cause the global stack to expand to a size where expansions and garbage collections are infrequent and garbage collections are not too time-consuming, if they occur at all.

The correct value for the gc_margin is dependent upon many factors. Here is a non-prioritized list of some of them:

- The amount of memory available to the Prolog process
- The maximum memory limit imposed on the Prolog process
- The program’s rate of global stack garbage generation
- The program’s rate of global stack non-garbage generation
- The program’s backtracking behavior
- The amount of time needed to collect the generated garbage
- The growth rate of the other Prolog stacks

The algorithm used when the global stack overflows is as follows:
if gc is on and
the global stack has grown at least gc_margin kilobytes
since the last garbage collection then
garbage collect the global stack
if less than gc_margin kilobytes are reclaimed then
    try to expand the global stack
endif
else
    try to expand the global stack
endif

The user can use the gc_margin option of prolog_flag/3 to reset the gc_margin (see Section 4.9.1 [ref-lps-ove], page 135). If a garbage collection reclaims at least the gc_margin kilobytes of global stack space, then the global stack is not expanded after garbage collection completes. Otherwise, the global stack is expanded after garbage collection. This expansion provides space for the future global stack usage that will presumably occur. In addition, no garbage collection occurs if the global stack has grown less than gc_margin kilobytes since the last garbage collection.

4.10.6 Invoking the Garbage Collector Directly

Normally, the garbage collector is invoked only when some Prolog data area overflows, so the time of its invocation is not predictable. In some applications it may be desirable to invoke the garbage collector at regular intervals (when there is known to be a significant amount of garbage on the global stack) so that the time spent garbage collecting is more evenly distributed in the processing time. For instance, it may prove desirable to invoke the garbage collector after each iteration of a question-and-answer loop that is not failure-driven.

In rare cases the default garbage collection parameters result in excessive garbage collecting costs or global stack expansion, and the user cannot tune the gc_margin parameter adequately. Explicitly invoking the garbage collector using the built-in predicate garbage_collect/0 can be useful in these circumstances.

See Section 11.3.89 [mpg-ref-garbage_collect], page 1042.

4.10.7 Atom Garbage Collection

By default, atoms created during the execution of a program remain permanently in the system until Prolog exits. For the majority of applications this behavior is not a problem and can be ignored. However, for two classes of application this can present problems. Firstly the internal architecture of SICStus Prolog limits the number of atoms that be can created to 1,048,575 on 32-bit machines, and this can be a problem for database applications that read large numbers of atoms from a database. Secondly, the space occupied by atoms can become significant and dominate memory usage, which can be a problem for processes designed to run perpetually.

These problems can be overcome by using atom garbage collection to reclaim atoms that are no longer accessible to the executing program.
Atoms can be created in many ways: when an appropriate token is read with \texttt{read_term/3}, when source or PO files are loaded, when \texttt{atom_codes/2} is called with a character list, or when \texttt{SP_atom_from_string()} is called in C code. In any of these contexts an atom is only created if it does not already exist; all atoms for a given string are given the same identification number, which is different from the atom of any other string. Thus, atom recognition and comparison can be done quickly, without having to look at strings. An occurrence of an atom is always of a fixed, small size, so where a given atom is likely to be used in several places simultaneously the use of atoms can also be more compact than the use of strings.

A Prolog functor is implemented like an atom, but also has an associated arity. For the purposes of atom garbage collection, a functor is considered to be an occurrence of the atom of that same name.

Atom garbage collection is similar to global stack garbage collection, invoked automatically as well as through a call to the built-in predicate \texttt{garbage_collect_atoms/0}. The atom garbage collector scans Prolog’s data areas looking for atoms that are currently in use and then throws away all unused atoms, reclaiming their space.

Atom garbage collection can turn an application that continually grows and eventually either runs into the atom number limit or runs out of space into one that can run perpetually. It can also make feasible applications that load and manipulate huge quantities of atom-rich data that would otherwise become full of useless atoms.

\textbf{4.10.7.1 The Atom Garbage Collector User Interface}

Because the creation of atoms does not follow any other system behaviors like memory growth or global stack garbage collection, SICStus has chosen to keep the invocation of atom garbage collection independent of any other operation and to keep the invocation of atom garbage collection explicit rather than making it automatic. It is often preferable for the programmer to control when it will occur in case preparations need to be made for it.

Atom garbage collection is invoked automatically when the number of new atoms created since the last atom garbage collection reaches the value of the \texttt{agc_margin} flag.

Atom garbage collection can be invoked explicitly by calling \texttt{garbage_collect_atoms/0}. The predicate normally succeeds silently. The user may determine whether to invoke atom garbage collection at a given point based on information returned from a call to \texttt{statistics/2} with the keyword \texttt{atoms}. That call returns a list of the form

\[
\text{\{number of atoms, atom space in use, atom space free\}}
\]

For example,

\[
\text{\{number of atoms, atom space in use, atom space free\}}
\]

For example,

\[
\text{\{number of atoms, atom space in use, atom space free\}}
\]

For example,
One would typically choose to call `garbage_collect_atoms/0` prior to each iteration of an iterative application, when either the number of atoms or the atom space in use passes some threshold, e.g.

```prolog
<driver loop> :-
  ... 
  repeat,
    maybe_atom_gc,
    <do next iteration>
  ... 
  fail.
<driver loop>.
```

where

```prolog
maybe_atom_gc :-
  statistics(atoms, [_,Inuse,_]),
  atom_gc_space_threshold(Space),
  ( Inuse > Space -> garbage_collect_atoms ; true ).
```

% Atom GC if there are more than 100000 bytes of atoms:
atom_gc_space_threshold(100000).

More sophisticated approaches might use both atom number, atom space and `agc_margin` thresholds, or could adjust a threshold if atom garbage collection did not free an adequate number of atoms.

To be most effective, atom garbage collection should be called when as few as possible atoms are actually in use. In the above example, for instance, it makes the most sense to do atom garbage collection at the beginning of each iteration rather than at the end, as at the beginning of the iteration the previous failure may just have freed large amounts of atom-rich global and local stack. Similarly, it is better to invoke atom garbage collection after abolishing or retracting a large database than to do so before. See Section 11.3.90 [mpg-ref-garbage_collect_atoms], page 1043.

### 4.10.7.2 Protecting Atoms in Foreign Memory

SICStus Prolog’s foreign language interface allows atoms to be passed to foreign functions. When calling foreign functions from Prolog, atoms are passed via the `+atom` argument type in the predicate specifications of `foreign/[2,3]` facts. The strings of atoms can be passed to foreign functions via the `+string` argument type. In the latter case a pointer to the Prolog symbol table’s copy of the string for an atom is what is passed. When calling Prolog from C, atoms are passed back from C to Prolog using the `-atom` and `-string` argument types in `extern/1` declarations. Atoms can also be created in foreign code via functions like `SP_atom_from_string()`.

Prolog does not keep track of atoms (or strings of atoms) stored in foreign memory. As such, it cannot guarantee that those atoms will be retained by atom garbage collection. Therefore SICStus Prolog provides functions to `register` atoms (or their strings) with the atom garbage
collector. Registered atoms will not be reclaimed by the atom garbage collector. Atoms can be registered while it is undesirable for them to be reclaimed, and then unregistered when they are no longer needed.

Of course, the majority of atoms passed as atoms or strings to foreign functions do not need to be registered. Only those that will be stored across foreign function calls (in global variables) or across nested calls to Prolog are at risk. An extra margin of control is given by the fact the programmer always invokes atom garbage collection explicitly, and can ensure that this is only done in contexts that are “safe” for the individual application.

To register or unregister an atom, one of the following functions is used:

```c
int SP_register_atom(atom)
SP_atom atom;

int SP_unregister_atom(atom)
SP_atom atom;
```

These functions return either SP_ERROR or a non-negative integer. The return values are discussed further in Section 4.10.7.4 [ref-mgc-ago-are], page 160.

As noted above, when an atom is passed as a string (+string) to a foreign function, the string the foreign function receives is the one in Prolog’s symbol table. When atom garbage collection reclaimed the atom for that string, the space for the string will also be reclaimed.

Thus, if the string is to be stored across foreign calls, then either a copy of the string or else the atom (+atom) should be passed into the foreign function so that it can be registered and SP_string_from_atom() can be used to access the string from the atom.

Keep in mind that the registration of atoms only pertains to those passed to foreign functions or created in foreign code. Atoms in Prolog’s data areas are maintained automatically. Note also that even though an atom may be unregistered in foreign code, atom garbage collection still may not reclaim it as it may be referenced from Prolog’s data areas. But if an atom is registered in foreign code, then it will be preserved regardless of its presence in Prolog’s data areas.

The following example illustrates the use of these functions. In this example the current value of an object (which is an atom) is being stored in a C global variable. There are two C functions that can be called from Prolog, one to update the current value and one to access the value.

#include <sicstus/sicstus.h>

SP_atom current_object = NULL;

update_object(SP_atom newvalue)
{
    /* if current_object contains an atom, unregister it */
    if (current_object)
        (void) SP_unregister_atom(current_object);

    /* register new value */
    (void) SP_register_atom(newvalue);
    current_object = newvalue;
}

SP_atom get_object(void)
{
    return current_object;
}

4.10.7.3 Permanent Atoms

Atom garbage collection scans all Prolog's dynamic data areas when looking for atoms that are in use. Scanning finds atoms in the Prolog stacks and in all compiled and interpreted code that has been dynamically loaded into Prolog via consult/1, use_module/1, assert/2, etc. However, there are certain potential sources of atoms in the Prolog image from which atoms cannot be reclaimed. Atoms for Prolog code that has been statically linked with either the Prolog Development Environment or the Runtime Environment have been placed in the text space, making them (and the code that contains them) effectively permanent. Although such code can be abolished, its space can never be reclaimed.

These atoms are internally flagged as permanent by the system and are always retained by atom garbage collection. An atom that has become permanent cannot be made non-permanent, so can never be reclaimed.

4.10.7.4 Details of Atom Registration

The functions that register and unregister atoms are in fact using reference counting to keep track of atoms that have been registered. As a result, it is safe to combine your code with libraries and code others have written. If the other code has been careful to register and unregister its atoms as appropriate, then atoms will not be reclaimed until everyone has unregistered them.

Of course, it is possible when writing code that needs to register atoms that errors could occur. Atoms that are registered too many times simply will not be garbage collected until they are fully unregistered. However, atoms that are not registered when they should be may be reclaimed on atom garbage collection. One normally does not need to think about the reference counting going on in SP_register_atom() and SP_unregister_atom(), but some understanding of its details could prove helpful when debugging.
To help you diagnose problems with registering and unregistering atoms, \texttt{SP\_register\_atom()} and \texttt{SP\_unregister\_atom()} both normally return the current reference count for the atom. If an error occurs, e.g. a nonexistent atom is registered or unregistered, then \texttt{SP\_ERROR} is returned.

An unregistered atom has a reference count of 0. Unregistering an atom that is unregistered is a no-op; in this case, \texttt{SP\_unregister\_atom()} returns 0. A permanent atom has a reference count of 256. In addition, if an atom is simultaneously registered 256 times, then it becomes permanent. (An atom with 256 distinct references is an unlikely candidate for reclamation!) Registering or unregistering an atom that is permanent is also a no-op; \texttt{SP\_register\_atom()} and \texttt{SP\_unregister\_atom()} return 256.

### 4.10.8 Summary of Predicates

- \texttt{garbage\_collect} force an immediate garbage collection
- \texttt{garbage\_collect\_atoms} garbage collect atom space
- \texttt{statistics} display various execution statistics
- \texttt{statistics(?K,?V)} the execution statistic with key \texttt{K} has value \texttt{V}
- \texttt{trimcore} reduce free stack space to a minimum

### 4.11 Modules

#### 4.11.1 Overview

The module system lets the user divide large Prolog programs into \textit{modules}, or rather smaller sub-programs, and define the interfaces between those modules. Each module has its own name space; that is, a predicate defined in one module is distinct from any predicates with the same name and arity that may be defined in other modules. The module system encourages a group of programmers to define the dependence each has on others’ work before any code is written, and subsequently allows all to work on their own parts independently. It also helps to make library predicates behave as extensions of the existing set of built-in predicates.

The SICStus Prolog library uses the module system and can therefore serve as an extended example of the concepts presented in the following text. The design of the module system is such that loading library files and calling library predicates can be performed without knowledge of the module system.

Some points to note about the module system are that:

- It is based on predicate modularity rather than on data modularity; that is, atoms and functors are global.
- It is flat rather than hierarchical; any module may refer to any other module by its name—there is no need to specify a path of modules.
• It is not strict; modularity rules can be explicitly overridden. This is primarily for flexibility during debugging.
• It is efficient; calls to predicates across module boundaries incur little or no overhead.

4.11.2 Basic Concepts

Each predicate in a program is identified by its module, as well as by its name and arity.

A module defines a set of predicates, among which some have the property of being public. Public predicates are predicates that can be imported by other modules, which means that they can then be called from within those modules. Predicates that are not public are private to the module in which they are defined; that is, they cannot be called from outside that module (except by explicitly overriding the modularity rules as described in Section 4.11.6 [ref-mod-vis], page 164).

There are two kinds of importation:

1. A module $M_1$ may import a specified set of predicates from another module $M_2$. All the specified predicates should be public in $M_2$.
2. A module $M_1$ may import all the public predicates of another module $M_2$.

Built-in predicates do not need to be imported; they are automatically available from within any module.

There is a special module called user, which is used by default when predicates are being defined and no other module has been specified.

The other predefined module is the prolog module where all the built-in predicates reside. The exported built-in predicates are automatically imported into each new module as it is created.

If you are using a program written by someone else, then you need not be concerned as to whether or not that program has been made into a module. The act of loading a module from a file using compile/1, or ensure_loaded/1 (see Section 4.3 [ref-lod], page 79) will automatically import all the public predicates in that module. Thus the command

```prolog
:- ensure_loaded(library(lists)).
```

will load the list-processing predicates from the library and make them available.

4.11.3 Defining a Module

The normal way to define a module is by creating a module file for it and loading it into the Prolog system. A module file is a Prolog file that begins with a module declaration.

A module declaration has one of the forms:

```prolog
:- module(+ModuleName, +PublicPredList).
:- module(+ModuleName, +PublicPredList, +Options).
```
Such a declaration must appear as the first term in a file, and declares that file to be a module file. The predicates in the file will become part of the module ModuleName, and the predicates specified in PublicPredList are those that can be imported by other modules; that is, the public predicates of this module.

Options is an optional argument, and should be a list. The only available option is hidden(Boolean), where Boolean is false (the default) or true. In the latter case, tracing of the predicates of the module is disabled (although spypoints can be set), and no source information is generated at compile time.

Instead of creating and loading a module file, it is also possible to define a module dynamically by, for example, asserting clauses into a specified module. A module created in this way has no public predicates; all its predicates are private. This means that they cannot be called from outside that module except by explicitly overriding the modularity rules as described in Section 4.11.6 [ref-mod-vis], page 164. Dynamic creation of modules is described in more detail in Section 4.11.9 [ref-mod-dmo], page 167.

4.11.4 Converting Non-module Files into Module Files

The Prolog cross-referencer can automatically generate module/2 declarations from its cross-reference information. This is useful if you want to take a set of files making up a program and make each of those files into a module file. For more information, see Section 9.12 [The Cross-Referencer], page 379.

Alternatively, if you have a complete Prolog program consisting of a set of source files {file1, file2, ...}, and you wish to encapsulate it in a single module mod, then this can be done by creating a “driver” file of the following form:

```prolog
:- module(mod, [ ... ]).
:- ensure_loaded(file1).
:- ensure_loaded(file2).
```

When a module is created in this way, none of the files in the program {file1, file2, ...} have to be changed.

4.11.5 Loading a Module

To gain access to the public predicates of a module file, load it as you would any other file—using compile/1, or ensure_loaded/1 as appropriate. For example, if your code contains a directive such as

```prolog
:- ensure_loaded(File).
```
then this directive will load the appropriate file File whether or not File is a module file. The only difference is that if File is a module file, then any private predicates that it defines will not be visible to your program.

The load predicates are adequate for use at Prolog’s top level, or when the file being loaded is a utility such as a library file. When you are writing modules of your own, use_module/[1,2,3] is the most useful.

The following predicates are used to load modules:

use_module(F)  
import the module file(s) F, loading them if necessary; same as ensure_loaded(F) if all files in F are module files

use_module(:F,+I)  
import the procedure(s) I from the module file F, loading module file F if necessary

use_module(?M,:F,+I)  
import I from module M, loading module file F if necessary

Before a module file is loaded, the associated module is reinitialized: any predicates previously imported into or defined in that module are forgotten by the module.

If a module of the same name with a different PublicPredList or different meta-predicate list has previously been loaded from a different module file, then a warning is printed and you are given the option of abandoning the load. Only one of these two modules can exist in the system at one time.

Normally, a module file can be reloaded after editing with no need to reload any other modules. However, when a module file is reloaded after its PublicPredList has been changed, any modules that import predicates from it may have become inconsistent. This is because a module is associated with a predicate at compile time, rather than run time. Thus, other modules may refer to predicates in a module file that are no longer public. In the case of module importation (where all, rather than specific, public predicates of a module are imported), it is possible that some predicates in the importing module should now refer to a newly-public predicate but do not. SICStus Prolog tries to detect such inconsistencies, and issues a warning when it does detect one. Similarly, if a meta-predicate declaration of an exported predicate changes, then modules that have already imported that predicate become inconsistent, because module name expansion requirements have changed. The current release of SICStus Prolog is unable to detect such inconsistencies.

Modules may be saved to a PO file by calling save_modules(Modules,File) (see Section 4.4 [ref-sls], page 92).

**4.11.6 Visibility Rules**

By default, predicates defined in one module cannot be called from another module. This section enumerates the exceptions to this—the ways in which a predicate can be visible to modules other than the one in which it is defined.
1. The built-in predicates can be called from any module.

2. Any predicate that is named in the PublicPredList of a module, and that is imported by some other module $M$, can be called from within $M$.

3. Module Prefixing: Any predicate, whether public or not, can be called from any other module if its module is explicitly given as a prefix to the goal, attached with the :/2 operator. The module prefix overrides the default module. For example,

   ```prolog
   :- mod:foo(X,Y).
   ```

   always calls $\text{foo}/2$ in module $\text{mod}$. This is effectively a loophole in the module system, which allows you to override the normal module visibility rules. It is intended primarily to facilitate program development and debugging, and it should not be used extensively since it subverts the original purposes of using the module system.

   Note that a predicate called in this way does not necessarily have to be defined in the specified module. It may be imported into it. It can even be a built-in predicate, and this is sometimes useful—see Section 4.11.7 [ref-mod-som], page 165, for an example.

4.11.7 The Source Module

For any given procedure call, or goal, the source module is the module in which the corresponding predicate must be visible. That is, unless the predicate is built-in, it must be defined in, or imported into, the source module.

For goals typed at the top level, the source module is the type-in module, which is user by default—see Section 4.11.8 [ref-mod-tyi], page 166. For goals appearing in a file, whether in a directive or in the body of a clause, the source module is the one into which that file has been loaded.

There are a number of built-in predicates that take predicate specifications, clauses, or goals as arguments. Each of these types of argument must be understood with reference to some module. For example, $\text{assert}/1$ takes a clause as its argument, and it must decide into which module that clause should be asserted. The default assumption is that it asserts the clause into the source module. Another example is $\text{call}/1$. The goal (A) calls the predicate $\text{foo}/1$ in the source module; this ensures that in the compound goal (B) both occurrences of $\text{foo}/1$ refer to the same predicate.

   ```prolog
   call(foo(X))  \textbf{(A)}
   call(foo(X)), foo(Y)  \textbf{(B)}
   ```

All predicates that refer to the source module allow you to override it by explicitly naming some other module to be used instead. This is done by prefixing the relevant argument of the predicate with the module to be used followed by a ‘:’ operator. For example (C), asserts $f(x)$ in module $m$.

   ```prolog
   | ?- assert(m:f(x)).  \textbf{(C)}
   ```

Note that if you call a goal in a specified module, overriding the normal visibility rules (see Section 4.11.6 [ref-mod-vis], page 164), then the source module for that goal is the one you specify, not the module in which this call occurs. For example (D), has exactly the same
effect as \((C) - f(x)\) is asserted in module \(m\). In other words, prefixing a goal with a module duplicates the effect of calling that goal from that module.

\[
| ?- m:assert(f(x)).
\]

Another built-in predicate that refers to the source module is \texttt{compile/1}. In this case, the argument is a file, or list of files, rather than a predicate specification, clause, or goal. However, in the case where a file is not a module file, \texttt{compile/1} must decide into which module to compile its clauses, and it chooses the source module by default. This means that you can compile a file \texttt{File} into a specific module \(M\) using

\[
| ?- compile(M:File).
\]

Thus if \texttt{File} is a module file, then this command would cause its public predicates to be imported into module \(M\). If \texttt{File} is a non-module file, then it is loaded into module \(M\).

For a list of the built-in predicates that depend on the source module, see Section 4.11.15 \[ref-mod-mne\], page 171. In some cases, user-defined predicates may also require the concept of a source module. This is discussed in Section 4.11.16 \[ref-mod-met\], page 171.

### 4.11.8 The Type-in Module

The type-in module is the module that is taken as the source module for goals typed in by the user. The name of the default type-in module is \texttt{user}. That is, the predicates that are available to be called directly by the user are those that are visible in the module \texttt{user}.

When debugging, it is often useful to call, directly from the top level, predicates that are private to a module, or predicates that are public but that are not imported into \texttt{user}. This can be done by prefixing each goal with the module name, as described in Section 4.11.6 \[ref-mod-vis\], page 164; but rather than doing this extensively, it may be more convenient to make this module the type-in module.

The type-in module can be changed using the built-in predicate \texttt{set_module/1}; for example,

\[
| ?- set_module(mod).
\]

This command will cause subsequent goals typed at the top level to be executed with \texttt{mod} as their source module.

The name of the type-in module is always displayed, except when it is \texttt{user}. If you are running Prolog under the editor interface, then the type-in module is displayed in the status line of the Prolog window. If you are running Prolog without the editor interface, then the type-in module is displayed before each top-level prompt.

For example, if you are running Prolog without the editor:

\[
| ?- set_module(foo).
\]

yes
[foo]
| ?-
It should be noted that it is unlikely to be useful to change the type-in module via a directive
embedded in a file to be loaded, because this will have no effect on the load—it will only
change the type-in module for commands subsequently entered by the user.

4.11.9 Creating a Module Dynamically

There are several ways in which you can create a module without loading a module file
for it. One way to do this is by asserting clauses into a specified module. For example,
the command (A) will create the dynamic predicate \( f/1 \) and the module \( m \) if they did not
previously exist.

\[
| \text{?- } \text{assert}(m:f(x)). \quad \text{(A)}
\]

Another way to create a module dynamically is to compile a non-module file into a specified
module. For example (B), will compile the clauses in \( \text{File} \) into the module \( M \).

\[
| \text{?- } \text{compile}(M:\text{File}). \quad \text{(B)}
\]

The same effect can be achieved by (temporarily) changing the type-in module to \( M \) (see
Section 4.11.8 [ref-mod-tyi], page 166) and then calling \( \text{compile(File)} \), or executing the
command in module \( M \) as in (C).

\[
| \text{?- } M:\text{compile(File)}. \quad \text{(C)}
\]

4.11.10 Module Prefixes on Clauses

Every clause in a Prolog file has a source module implicitly associated with it. If the file is
a module file, then the module named in the module declaration at the top of the file is the
source module for all the clauses. If the file is not a module file, then the relevant module
is the source module for the command that caused this file to be loaded.

The source module of a predicate decides in which module it is defined (the module of the
head), and in which module the goals in the body are going to be called (the module of
the body). It is possible to override the implicit source module, both for head and body, of
clauses and directives, by using prefixes. For example, consider the module file:

\[
:- \text{module(a, [])}.
\]

\[
:- \text{dynamic m:a/1}.
\]

\[
\begin{align*}
b(1).
\text{m:c([])}.
\text{m:d([H|T]) :- } & \text{q(H), r(T).} \\
\text{m:(e(X) :- } & \text{s(X), t(X)).} \\
\text{f(X) :- } & \text{m:(u(X), v(X)).}
\end{align*}
\]

In the previous example, the following modules apply:

1. \( a/1 \) is declared dynamic in the module \( m \).
2. \( b/1 \) is defined in module \( a \) (the module of the file).
3. \( c/1 \) is defined in module \( m \).
4. d/1 is defined in module m, but q/1 and r/1 are called in module a (and must therefore be defined in module a).
5. e/1 is defined in module m, and s/1 and t/1 are called in module m.
6. f/1 is defined in module a, but u/1 and v/1 are called in module m.

Module prefixing is especially useful when the module prefix is user. There are several predicates that have to be defined in module user but that you may want to define (or extend) in a program that is otherwise entirely defined in some other module or modules; see Section 11.2.12 [mpg-top-hok], page 912.

Note that if clauses for one of these predicates are to be spread across multiple files, then it will be necessary to declare that predicate to be multifile by putting a multifile declaration in each of the files.

4.11.10.1 Current Modules
A loaded, or dynamically created, module becomes current as soon as it is encountered, and a module can never lose the property of being current. The set of current modules can be obtained with current_module/1, see Section 4.11.13 [ref-mod-ilm], page 169.

4.11.11 Debugging Code in a Module
Having loaded a module to be debugged, you can trace through its execution in the normal way. When the debugger stops at a port, the procedure being debugged is displayed with its module name as a prefix unless the module is user.

The predicate spy/1 depends on the source module. It can be useful to override this during debugging. For example,

```
| ?- spy mod1:f/3.
```

puts a spypoint on f/3 in module mod1.

It can also be useful to call directly a predicate that is private to its module in order to test that it is doing the right thing. This can be done by prefixing the goal with its module; for example,

```
| ?- mod1:f(a,b,X).
```

4.11.12 Name Clashes
A name clash can arise if:

1. a module tries to import a predicate from some other module m1 and it has already imported a predicate with the same name and arity from a module m2;
2. a module tries to import a predicate from some other module m1 and it already contains a definition of a predicate with the same name and arity; or
3. a module tries to define a predicate with the same name and arity as one that it has imported.
Whenever a name clash arises, a message is displayed beginning with the words ‘NAME CLASH’. The user is asked to choose from one of several options; for example,

```
NAME CLASH: f/3 is already imported into module user
from module m1;
do you want to override this definition with
the one in m2? (y,n,p,s,a or ?)
```

The meanings of the five recognized replies are as follows:

- **y**: forget the previous definition of f/3 from m1 and use the new definition of f/3 from m2 instead.
- **n**: retain the previous definition of f/3 from m1 and ignore the new definition of f/3 from m2.
- **p**: (for proceed) means forget the previous definition of f/3 and of all subsequent predicate definitions in m1 that clash during the current load of m2. Instead, use the new definitions in m2. When the p option is chosen, predicates being loaded from m1 into m2 will cause no ‘NAME CLASH’ messages for the remainder of the load, though clashes with predicates from other modules will still generate such messages.
- **s**: (for suppress) means forget the new definition of f/3 and of all subsequent predicate definitions in m1 that clash during the current load of m2. Instead, use the old definitions in m2. When the s option is chosen, predicates being loaded from m1 into m2 will cause no ‘NAME CLASH’ messages for the remainder of the load, though clashes with predicates from other modules will still generate such messages.
- **?**: gives brief help information.

### 4.11.13 Obtaining Information about Loaded Modules

The built-in predicate `current_module/2` can be used to find all the currently loaded module, and where they were loaded from. See Section 11.3.52 [mpg-ref-current_module], page 993.

```
current_module(\?M)
    \M is the name of a current module

current_module(\?M, \?F)
    \F is the name of the file in which \M’s module declaration appears. Not all modules have a corresponding file.
```

### 4.11.13.1 Predicates Defined in a Module

The built-in predicate `current_predicate/2` can be used to find the predicates that are defined in a particular module.

To backtrack through all of the predicates defined in module m, use

```
| ?- current_predicate(_, m:Goal).
```
To backtrack through all predicates defined in any module, use

```
| ?- current_predicate(_, M:Goal).
```

This succeeds once for every predicate in your program. See Section 11.3.55 [mpg-ref-current_predicate], page 997.

### 4.11.13.2 Predicates Visible in a Module

The built-in predicate `predicate_property/2` can be used to find the properties of any predicate that is visible to a particular module.

To backtrack through all of the predicates imported by module `m`, use

```
| ?- predicate_property(m:Goal, imported_from(_)).
```

To backtrack through all of the predicates imported by module `m1` from module `m2`, use

```
| ?- predicate_property(m1:Goal, imported_from(m2)).
```

For example, you can load the `between` module from the library and then remind yourself of what predicates it defines like this:

```
| ?- compile(library(between)).
% ... loading messages ...

yes
| ?- predicate_property(P, imported_from(between)).
P = numlist(_A,_B) ? ;
P = numlist(_A,_B,_C,_D,_E) ? ;
  .
  .
  .
```

This tells you what predicates are imported into the type-in module from `basics`.

You can also find all imports into all modules using

```
| ?- predicate_property(M1:G, imported_from(M2)).
```

To backtrack through all of the defined predicates exported by module `m`, use

```
| ?- predicate_property(m:Goal, exported).
```

See Section 11.3.159 [mpg-ref-predicate_property], page 1132.

### 4.11.14 Importing Dynamic Predicates

Imported dynamic predicates may be asserted and retracted. For example, suppose the following file is loaded via `use_module/1`:
Then \( f/1 \) can be manipulated as if it were defined in the current module. For example,

\[
?- \text{clause}(f(X), \text{true}).
\]

\[
X = 0
\]

The built-in predicate \texttt{listing/[0,1]} distinguishes predicates that are imported into the current source module by prefixing each clause with the module name. Thus,

\[
?- \text{listing}(f).
\]

\[
m1:f(0).
\]

However, \texttt{listing/[0,1]} does not prefix clauses with their module if they are defined in the source module itself. Note that

\[
?- \text{listing}.
\]

can be used to see all the dynamic predicates defined in or imported into the current type-in module. And

\[
?- \text{listing}(m1:__).
\]

can be used to see all such predicates that are defined in or imported into module \texttt{m1}. See Section 11.3.116 \cite{mpg-ref-listing}, page 1075.

### 4.11.15 Module Name Expansion

The concept of a source module is explained in Section 4.11.7 \cite{ref-mod-som}, page 165. For any goal, the applicable source module is determined when the goal is compiled rather than when it is executed.

A procedure that needs to refer to the source module has arguments designated for module name expansion. These arguments are expanded when code is consulted, compiled or asserted by the transformation \( X \rightarrow M:X \) where \( M \) is the name of the source module. For example, the goal \texttt{call(X)} is expanded into \texttt{call(M:X)} and the goal \texttt{clause(Head, Body)} is expanded into \texttt{clause(M:Head, Body)}.

Module name expansion is avoided if the argument to be expanded is already a \texttt{:/2} term. In this case it is unnecessary since the module to be used has already been supplied by the programmer.

### 4.11.16 The \texttt{meta_predicate} Declaration

Sometimes a user-defined predicate will require module name expansion (see Section 4.11.15 \cite{ref-mod-nme}, page 171). This can be specified by providing a \texttt{meta_predicate} declaration for that procedure.
Module name expansion is needed whenever the argument of a predicate has some module-dependent meaning. For example, if this argument is a goal that is to be called, then it will be necessary to know in which module to call it—or, if the argument is a clause to be asserted, in which module it should go.

Consider, for example, a sort routine to which the name of the comparison predicate is passed as an argument. In this example, the comparison predicate should be called, with two arguments like the built-in $\texttt{==/2}$, with respect to the module containing the call to the sort routine. Suppose that the sort routine is

\[
\texttt{mysort(CompareProc, InputList, OutputList)}
\]

An appropriate \texttt{meta_predicate} declaration for this is

\[
:- \texttt{meta_predicate mysort(2, +, -)}.
\]

The significant argument in the \texttt{mysort/3} term is the ‘2’, which indicates that module name expansion is required for this argument and that two additional arguments will be added when this argument is invoked as a goal. This means that whenever a goal \texttt{mysort(A, B, C)} appears in a clause, it will be transformed at load time into \texttt{mysort(M:A, B, C)}, where \texttt{M} is the source module. There are some exceptions to this compile-time transformation rule; the goal is not transformed if either of the following applies:

1. \texttt{A} is of the form \texttt{Module:Goal}.
2. \texttt{A} is a variable and the same variable appears in the head of the clause in a module-name-expansion position.

The reason for (2) is that otherwise module name expansion could build larger and larger structures of the form \texttt{Mn: . . . :M2:M1:Goal}. For example, consider the following program fragment adapted from the library (see \texttt{library(samsort)} for the full program):

\[
:- \texttt{module(samsort, [samsort/3])}.
\]

\[
:- \texttt{meta_predicate}
\]

\[
\begin{align*}
\texttt{samsort}(2, +, ?), \\
\texttt{sam_sort}(+, 2, +, +, ?).
\end{align*}
\]

\[
\texttt{samsort(_, [], [])} :- !.
\]

\[
\texttt{samsort(Order, List, Sorted)} :-
\]

\[
\begin{align*}
\texttt{sam_sort(List, Order, [], 0, Sorted)}.
\end{align*}
\]

Normally, the \texttt{sam_sort/5} goal in this example would have the module name of its second argument expanded thus:

\[
\texttt{sam_sort(List, samsort:Order, [], 0, Sorted)}
\]
because of the `meta_predicate` declaration. However, in this situation the appropriate source module will have already been attached to `Order` because it is the first argument of `samsort/3`, which also has a `meta_predicate` declaration. Therefore it is not useful to attach the module name (`samsort`) to `Order` in the call of `sam_sort/5`.

The argument of a `meta_predicate` declaration can be a term, or a sequence of terms separated by commas. Each argument of each of these terms must be one of the following:

- 
  - `:` requires module name expansion
  - If the argument will be treated as a goal, then it is better to explicitly indicate this using an integer; see the next item.

  `nsuppressed`
  
  a non-negative integer.
  
  This is a special case of `:` which means that the argument can be made into a goal by adding `nsuppressed` additional arguments. E.g., if the argument will be passed to `call/1`, then 0 (zero) should be used.

  An integer is treated the same as `:` above by the SICStus runtime. Other tools, such as the cross referencer (see Section 9.12 [The Cross-Referencer], page 379) and the SICStus Prolog IDE (see Section 3.11 [SPIDER], page 29), will use this information to better follow predicate references in analyzed source code.

  If the number of extra arguments is unknown or varies, then the generic `:` is always safe to use, but will give less accurate results from source analysis tools.

- `*`
- `+`
- `-`
- `?` ignored

The reason for `+`, `-` and `?` is simply so that the information contained in a DEC-10 Prolog-style “mode” declaration may be represented in the `meta_predicate` declaration if you wish. There are many examples of `meta_predicate` declarations in the library.

Prior to release 4.1, only `:` (colon) was used and the integer form was undocumented (but supported, e.g. by the cross referencer).

### 4.11.17 Semantics of Module Name Expansion

Although module name expansion is performed when code is consulted, compiled or asserted, it is perhaps best explained in terms of an interpreter, especially the issue of how deeply clauses are expanded. The semantics of `call/1`, taking `meta_predicate` declarations into account, is shown as if defined by the interpreter shown below. The interpreter’s case analysis is as follows:

- `control constructs`
  
  (Including cuts and module prefixes). The interpreter implements the semantics of the construct, expanding its argument.
callable terms with functor N/A

First, we look for a meta_predicate declaration for N/A. If one exists, then the relevant arguments are expanded. Otherwise, the goal is left unexpanded. Then, if N/A is a built-in predicate, then it is called. Otherwise, a clause with head functor N/A is looked up using the imaginary predicate :-/2, unified against, and its body is interpreted.

non-callable terms

Raise error exception.

Throughout the interpretation, we must keep track of the module context. The interpreter is as follows, slightly simplified. ++>/2 is not a predicate:
call(M:Body) :-
    icall(Body, M).

icall(Var, M) :- \+callable(Var), !,
    must_be(Var, callable, call(M:Var), 1).
icall(M:Body, _) :- !,
    icall(Body, M).
icall(_, _) :- !,
    \% cut relevant choicepoints.
icall((A, B), M) :- !,
    icall(A, M),
    icall(B, M).
icall((A -> B), M) :- !,
    ( icall(A, M) ->
    icall(B, M)
    ).
icall((A -> B ; C), M) :- !,
    ( icall(A, M) ->
    icall(B, M)
    ; icall(C, M)
    ).
icall((A ; B), M) :- !,
    ( icall(A, M)
    ; icall(B, M)
    ).
icall(+A), M) :- !,
    ( icall(A, M) ->
    fail
    ; true
    ).
icall(-A, M) :- !,
    icall(A, M).
icall(do(Iter,Body), M) :- !,
    ( Iter,
    param(M)
    do icall(Body, M)
    ).
icall(if(A,B,C), M) :- !,
    if(icall(A, M),
    icall(B, M),
    icall(C, M)).
icall(once(A), M) :- !,
    ( icall(A, M) -> true
    ).
icall(Goal, M) :-
    ( predicate_property(M:Goal, meta_predicate(Meta)) ->
    functor(Goal, Name, Arity),
    functor(AGoal, Name, Arity),
    ( foreacharg(Spec,Meta),
    foreacharg(Arg,Goal),
    foreacharg(Ann,AGoal),
    param(M)
    do ( Spec==(:) -> Ann = M:Arg
    )
    ).
4.11.18 Predicate Summary

current_module(?M)
M is the name of a current module

current_module(?M, ?F)
F is the name of the file in which M’s module declaration appears

meta_predicate :P declaration
declares predicates P that are dependent on the module from which they are called

module(+M,+L) declaration
module(+M,+L,+O) declaration
declaration that module M exports predicates in L, options O

save_modules(+L,+F)
save the modules specified in L into file F

set_module(+M)
make M the type-in module

use_module(:F)
import the module file(s) F, loading them if necessary

use_module(:F,+I)
import the procedure(s) I from the module file F

use_module(?M, :F,+I)
import I from module M, loading module file F if necessary

4.12 Modification of the Database

4.12.1 Introduction

The family of assertion and retraction predicates described below enables you to modify a Prolog program by adding or deleting clauses while it is running. These predicates should not be overused. Often people who are experienced with other programming languages have a tendency to think in terms of global data structures, as opposed to data structures that are passed as procedure arguments, and hence they make too much use of assertion and retraction. This leads to less readable and less efficient programs.

An interesting question in Prolog is what happens if a procedure modifies itself, by asserting or retracting a clause, and then fails. On backtracking, does the current execution of the procedure use new clauses that are added to the bottom of the procedure?

**Historical note:** In some non-ISO-conforming implementations of Prolog, changes to the Prolog database become globally visible upon the success of the built-in predicate modifying the database. An unsettling consequence is that the definition of a procedure can change while it is being run. This can lead to code that is difficult to understand. Furthermore, the memory performance of the interpreter implementing these semantics is poor. Worse yet, the semantics rendered ineffective the added determinacy detection available through indexing.
SICStus Prolog implements the “logical” view in updating dynamic predicates, conforming to the ISO standard. This means that the definition of a dynamic procedure that is visible to a call is effectively frozen when the call is made. A procedure always contains, as far as a call to it is concerned, exactly the clauses it contained when the call was made.

A useful way to think of this is to consider that a call to a dynamic procedure makes a virtual copy of the procedure and then runs the copy rather than the original procedure. Any changes to the procedure made by the call are immediately reflected in the Prolog database, but not in the copy of the procedure being run. Thus, changes to a running procedure will not be visible on backtracking. A subsequent call, however, makes and runs a copy of the modified Prolog database. Any changes to the procedure that were made by an earlier call will now be visible to the new call.

In addition to being more intuitive and easy to understand, the new semantics allow interpreted code to execute with the same determinacy detection (and excellent memory performance) as static compiled code (see Section 9.5 [Indexing], page 360, for more information on determinacy detection).

### 4.12.2 Dynamic and Static Procedures

All Prolog procedures are classified as being either static or dynamic procedures. Static procedures can be changed only by completely redefining them using the Load Predicates (see Section 4.3 [ref-lod], page 79). Dynamic procedures can be modified by adding or deleting individual clauses using the assert and retract procedures.

If a procedure is defined by loading source code, then it is static by default. If you need to be able to add, delete, or inspect the individual clauses of such a procedure, then you must make the procedure dynamic.

There are two ways to make a procedure dynamic:

- If the procedure is defined by loading source code, then it must be declared to be dynamic before it is defined.
- If the procedure is to be created by assertions only, then the first assert operation on the procedure automatically makes it dynamic.

A procedure is declared dynamic by preceding its definition with a declaration of the form:

```prolog
:- dynamic :Pred
```

where `Pred` must be a procedure specification of the form `Name/Arity`, or a sequence of such specifications, separated by commas. For example,

```prolog
:- dynamic exchange_rate/3, spouse_of/2, gravitational_constant/1.
```

where ‘dynamic’ is a built-in prefix operator. If `Pred` is not of the specified form, then an exception is raised, and the declaration is ignored.
Note that the symbol ‘:- ’ preceding the word ‘dynamic’ is essential. If this symbol is omitted, then a permission error is raised because it appears that you are trying to define a clause for the built-in predicate `dynamic/1`. Although `dynamic/1` is a built-in predicate, it may only be used in declarations.

When a dynamic declaration is encountered in a file being loaded, it is considered to be a part of the redefinition of the procedures specified in its argument. Thus, if you load a file containing only

```
:- dynamic hello/0
```

then the effect will be to remove any previous definition of `hello/0` from the database, and to make the procedure dynamic. You cannot make a procedure dynamic retroactively. If you wish to make an already-existing procedure dynamic, then it must be redefined.

It is often useful to have a dynamic declaration for a procedure even if it is to be created only by assertions. This helps another person to understand your program, since it emphasizes the fact that there are no pre-existing clauses for this procedure, and it also avoids the possibility of Prolog stopping to tell you there are no clauses for this procedure if you should happen to call it before any clauses have been asserted. This is because unknown procedure catching (see Section 3.6 [Undefined Predicates], page 26) does not apply to dynamic procedures; it is presumed that a call to a dynamic procedure should simply fail if there are no clauses for it.

If a program needs to make an undefined procedure dynamic, then this can be achieved by calling `clause/2` on that procedure. The call will fail because the procedure has no clauses, but as a side effect it will make the procedure dynamic and thus prevent unknown procedure catching on that procedure. See the Reference page for details of `clause/2`.

Although you can simultaneously declare several procedures to be dynamic, as shown above, it is recommended that you use a separate dynamic declaration for each procedure placed immediately before the clauses for that procedure. In this way when you reload the procedure using the editor interface, you will be reminded to include its dynamic declaration.

Dynamic procedures are implemented by interpretation, even if they are included in a file that is compiled. This means that they are executed more slowly than if they were static, and also that they can be printed using `listing/0`. Dynamic procedures, as well as static procedures, are indexed on their first argument; see Section 9.5 [Indexing], page 360.

### 4.12.3 Database References

A database reference is a term that uniquely identifies a clause or recorded term (see Section 4.12.8 [ref-mdb-idb], page 183) in the database. Database references are provided only to increase efficiency in programs that access the database in complex ways. Use of a database reference to a clause can save repeated searches using `clause/2`. However, it does not normally pay to access a clause via a database reference when access via first argument indexing is possible.
4.12.4 Adding Clauses to the Database

The assertion predicates are used to add clauses to the database in various ways. The relative position of the asserted clause with respect to other clauses for the same predicate is determined by the choice among \texttt{assert/1}, \texttt{asserta/1}, and \texttt{assertz/1}. A database reference that uniquely identifies the clause being asserted is established by providing an optional second argument to any of the assertion predicates.

\texttt{assert(C)}
\begin{itemize}
  \item clause \textit{C} is asserted in an arbitrary position in its predicate
\end{itemize}

\texttt{assert(C,-R)}
\begin{itemize}
  \item as \texttt{assert/1}; reference \textit{R} is returned
\end{itemize}

\texttt{asserta(C)}
\begin{itemize}
  \item clause \textit{C} is asserted before existing clauses
\end{itemize}

\texttt{asserta(C,-R)}
\begin{itemize}
  \item as \texttt{asserta/1}; reference \textit{R} is returned
\end{itemize}

\texttt{assertz(C)}
\begin{itemize}
  \item clause \textit{C} is asserted after existing clauses
\end{itemize}

\texttt{assertz(C,-R)}
\begin{itemize}
  \item as \texttt{assertz/1}; reference \textit{R} is returned
\end{itemize}

\textbf{Please note:} If the term being asserted contains attributed variables (see Section 10.3 \cite{lib-atts}, page 391) or suspended goals (see Section 4.2.4 \cite{ref-sem-sec}, page 74), then those attributes are not stored in the database. To retain the attributes, you can use \texttt{copy_term/3} (see Section 4.8.7 \cite{ref-lte-cpt}, page 129).

4.12.5 Removing Clauses from the Database

This section briefly describes the predicates used to remove the clauses and/or properties of a predicate from the system.

\textbf{Please note:} Removing all of a predicate’s clauses by \texttt{retract/1} and/or \texttt{erase/1} (see Section 4.12.5.1 \cite{ref-mdb-rcd-efu}, page 180) does not remove the predicate’s properties (and hence its definition) from the system. The only way to completely remove a predicate’s clauses and/or properties is to use \texttt{abolish/[1,2]}.

\texttt{retract(C)}
\begin{itemize}
  \item erase the first dynamic clause that matches \textit{C}
\end{itemize}

\texttt{retractall(H)}
\begin{itemize}
  \item erase every clause whose head matches \textit{H}
\end{itemize}

\texttt{abolish(F)}
\begin{itemize}
  \item abolish the predicate(s) specified by \textit{F}
\end{itemize}

\texttt{abolish(F,+O)}
\begin{itemize}
  \item abolish the predicate(s) specified by \textit{F} with options \textit{O}
\end{itemize}
erase(+R)

erase the clause or recorded term (see Section 4.12.8 [ref-mdb-idb], page 183) with reference R

4.12.5.1 A Note on Efficient Use of retract/1

WARNING: retract/1 is a nondeterminate procedure. Thus, we can use

| ?- retract((foo(X) :- Body)), fail.

to retract all clauses for foo/1. A nondeterminate procedure in SICStus Prolog uses a choicepoint, a data structure kept on an internal stack, to implement backtracking. This applies to user-defined procedures as well as to built-in and library procedures. In a simple model, a choicepoint is created for each call to a nondeterminate procedure, and is deleted on determinate success or failure of that call, when backtracking is no longer possible. In fact, SICStus Prolog improves upon this simple model by recognizing certain contexts in which choicepoints can be avoided, or are no longer needed.

The Prolog cut (‘!’) works by removing choicepoints, disabling the potential backtracking they represented. A choicepoint can thus be viewed as an “outstanding call”, and a cut as deleting outstanding calls.

To avoid leaving inconsistencies between the Prolog database and outstanding calls, a retracted clause is reclaimed only when the system determines that there are no choicepoints on the stack that could allow backtracking to the clause. Thus, the existence of a single choicepoint on the stack can disable reclamation of retracted clauses for the procedure whose call created the choicepoint. Space is recovered only when the choicepoint is deleted.

Often retract/1 is used determinately; for example, to retract a single clause, as in

| ?- <do some stuff>
  | retract(Clause),
  | <do more stuff without backtracking>.

No backtracking by retract/1 is intended. Nonetheless, if Clause may match more than one clause in its procedure, then a choicepoint will be created by retract/1. While executing “<do more stuff without backtracking>”, that choicepoint will remain on the stack, making it impossible to reclaim the retracted Clause. Such choicepoints can also disable tail recursion optimization. If not cut away, then the choicepoint can also lead to runaway retraction on the unexpected failure of a subsequent goal. This can be avoided by simply cutting away the choicepoint with an explicit cut or a local cut (‘->’). Thus, in the previous example, it is preferable to write either

| ?- <do some stuff>
  | retract(Clause),
  | !,
  | <do more stuff without backtracking>.

or
\begin{verbatim}
| ?- <do some stuff>
  ( retract(Clause) -> true ),
  <do more stuff without backtracking>.
\end{verbatim}

This will reduce stack size and allow the earliest possible reclamation of retracted clauses.

### 4.12.6 Accessing Clauses

**Goal Succeeds If:**

- `clause(:P, ?Q)`
  There is a clause for a dynamic predicate with head `P` and body `Q`.
- `clause(:P, ?Q, ?R)`
  There is a clause for a dynamic predicate with head `P`, body `Q`, and reference `R`.
- `instance(+R, -T)`
  `T` is an instance of the clause or term referenced by `R`.

### 4.12.7 Modification of Running Code: Examples

The following examples show what happens when a procedure is modified while it is running. This can happen in two ways:

1. The procedure calls some other procedure that modifies it.
2. The procedure succeeds nondeterminately, and a subsequent goal makes the modification.

In either case, the question arises as to whether the modifications take effect upon backtracking into the modified procedure. In SICStus Prolog the answer is that they do not. As explained in the overview to this section (see Section 4.12.1 [ref-mdb-bas], page 176), modifications to a procedure affect only calls to that procedure that occur after the modification.

#### 4.12.7.1 Example: `assertz`

Consider the procedure `foo/0` defined by

\begin{verbatim}
:- dynamic foo/0.
foo :- assertz(foo), fail.
\end{verbatim}

Each call to `foo/0` asserts a new last clause for `foo/0`. After the Nth call to `foo/0` there will be N+1 clauses for `foo/0`. When `foo/0` is first called, a virtual copy of the procedure is made, effectively freezing the definition of `foo/0` for that call. At the time of the call, `foo/0` has exactly one clause. Thus, when `fail/0` forces backtracking, the call to `foo/0` simply fails: it finds no alternatives. For example,
Even though the virtual copy of foo/0 being run by the first call is not changed by the assertion, the Prolog database is. Thus, when a second call to foo/0 is made, the virtual copy for that call contains two clauses. The first clause fails, but on backtracking the second clause is found and the call succeeds.

### 4.12.7.2 Example: retract

```prolog
| ?- assert(p(1)), assert(p(2)), assert(p(3)).
yes
| ?- p(N), write(N), nl, retract(p(2)), retract(p(3)), fail.
  1
  2
  3
no
| ?- p(N), write(N), fail.
  1
no
| ?-
```

At the first call to p/1, the procedure has three clauses. These remain visible throughout execution of the call to p/1. Thus, when backtracking is forced by fail/0, N is bound to 2 and written. The retraction is again attempted, causing backtracking into p/1. N is bound to 3 and written out. The call to retract/1 fails. There are no more clauses in p/1, so the query finally fails. A subsequent call to p/1, made after the retractions, sees only one clause.
4.12.7.3 Example: abolish

```prolog
| ?- compile(user).
| :- dynamic q/1.
| q(1).
| q(2).
| q(3).
| ^D
% user compiled in modules user, 0.117 sec 260 bytes
```

```
yes
| ?- q(N), write(N), nl, abolish(q/1), fail.
1
2
3
```

```prolog
no
| ?-
```

Procedures that are abolished while they have outstanding calls do not become invisible to those calls. Subsequent calls however, will find the procedure undefined.

4.12.8 The Internal Database

The following predicates are provided solely for compatibility with other Prolog systems. Their semantics can be understood by imagining that they are defined by the following clauses:

```prolog
recorda(Key, Term, Ref) :-
  functor(Key, Name, Arity),
  functor(F, Name, Arity),
  asserta('$_recorded'(F,Term), Ref).
recordz(Key, Term, Ref) :-
  functor(Key, Name, Arity),
  functor(F, Name, Arity),
  assertz('$_recorded'(F,Term), Ref).
recorded(Key, Term, Ref) :-
  functor(Key, Name, Arity),
  functor(F, Name, Arity),
  clause('$_recorded'(F,Term), _, Ref).
```

The reason for the calls to `functor/3` in the above definition is that only the principal functor of the key is significant. If `Key` is a compound term, then its arguments are ignored.

Please note: Equivalent functionality and performance, with reduced memory costs, can usually be had through normal dynamic procedures and indexing (see Section 4.12.1 [ref-mdb-bas], page 176, and Section 9.5 [Indexing], page 360).
recorda(Key, Term, Ref) records the Term in the internal database as the first item for the key Key; a database reference to the newly-recorded term is returned in Ref.

recordz(Key, Term, Ref) is like recorda/3 except that it records the term as the last item in the internal database.

recorded(Key, Term, Ref) searches the internal database for a term recorded under the key Key that unifies with Term, and whose database reference unifies with Ref.

current_key(KeyName, KeyTerm) succeeds when KeyName is the atom or integer that is the name of KeyTerm. KeyTerm is an integer, atom, or compound term that is the key for a currently recorded term.

4.12.9 Blackboard Primitives

The predicates described in this section store arbitrary terms in a per-module repository known as the “blackboard”. The main purpose of the blackboard was initially to provide a means for communication between branches executing in parallel, but the blackboard works equally well during sequential execution. The blackboard implements a mapping from keys to values. Keys are restricted to being atoms or small integers, whereas values are arbitrary terms. In contrast to the predicates described in the previous sections, a given key can map to at most a single term.

Each Prolog module maintains its own blackboard, so as to avoid name clashes if different modules happen to use the same keys. The “key” arguments of these predicates are subject to module name expansion, so the module name does not have to be explicitly given unless multiple Prolog modules are supposed to share a single blackboard.

The predicates below implement atomic blackboard actions.

bb_put(:Key, +Term)

A copy of Term is stored under Key. See Section 11.3.24 [mpg-ref-bb_put], page 961.

bb_get(:Key, ?Term)

If a term is currently stored under Key, then a copy of it is unified with Term. Otherwise, bb_get/2 silently fails. See Section 11.3.23 [mpg-ref-bb_get], page 960.

bb_delete(:Key, ?Term)

If a term is currently stored under Key, then the term is deleted, and a copy of it is unified with Term. Otherwise, bb_delete/2 silently fails. See Section 11.3.22 [mpg-ref-bb_delete], page 959.

bb_update(:Key, ?OldTerm, ?NewTerm)

If a term is currently stored under Key and unifies with OldTerm, then the term is replaced by a copy of NewTerm. Otherwise, bb_update/3 silently fails. This predicate provides an atomic swap operation. See Section 11.3.25 [mpg-ref-bb_update], page 962.
Please note: If the term being stored contains attributed variables (see Section 10.3 [lib-atts], page 391) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 74), then those attributes are not stored. To retain the attributes, you can use copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 129).

The following example illustrates how these primitives may be used to implement a “maxof” predicate that finds the maximum value computed by some nondeterminate goal. We use a single key max. We assume that Goal does not produce any “false” solutions that would be eliminated by cuts in a sequential execution. Thus, Goal may need to include redundant checks to ensure that its solutions are valid, as discussed above.

```
maxof(Value, Goal, _) :-
    bb_put(max, -1), % initialize max-so-far
    call(Goal),
    update_max(Value),
    fail.
maxof(_, _, Max) :-
    bb_delete(max, Max),
    Max > 1.
update_max(New) :-
    bb_get(max, Old),
    compare(C, Old, New),
    update_max(C, Old, New).
update_max(<, Old, New) :- bb_update(max, Old, New).
update_max(=, _, _).
update_max(>, _, _).
```

4.12.10 Summary of Predicates

abolish(:F) ISO
abolish the predicate(s) specified by F
abolish(:F,+O)
abolish the predicate(s) specified by F with options O
assert(:C)
assert(:C,-R)
clause C is asserted; reference R is returned
asserta(:C) ISO
asserta(:C,-R)
clause C is asserted before existing clauses; reference R is returned

---

4 This is not necessarily a good example of using the blackboard. For instance, the implementation is not reentrant, e.g. it will not work if the Goal itself uses maxof/3. A reentrant implementation would need to ensure that multiple nested calls to maxof/3 do not interfere with each other.
assertz(:C)
assertz(:C, -R)
clause C is asserted after existing clauses; reference R is returned
bb_delete(:Key, -Term)
Delete from the blackboard Term stored under Key.
bb_get(:Key, -Term)
Get from the blackboard Term stored under Key.
bb_put(:Key, +Term)
Store Term under Key on the blackboard.
bb_update(:Key, -OldTerm, +NewTerm)
Replace OldTerm by NewTerm under Key on the blackboard.
clause(:P, ?Q)
clause(:P, ?Q, ?R)
there is a clause for a dynamic predicate with head P, body Q, and reference R
current_key(?N, ?K)
N is the name and K is the key of a recorded term
dynamic :P
predicates specified by P are dynamic
erase(+R)
erase the clause or record with reference R
instance(+R, -T)
T is an instance of the clause or term referenced by R
recorda(+K, +T, -R)
make term T the first record under key K; reference R is returned
recorded(?K, ?T, ?R)
term T is recorded under key K with reference R
recordz(+K, +T, -R)
make term T the last record under key K; reference R is returned
retract(:C)
erase the first dynamic clause that matches C
retractall(:H)
erase every clause whose head matches H

4.13 Sets and Bags: Collecting Solutions to a Goal
4.13.1 Introduction
When there are many solutions to a goal, and a list of all those solutions is desired, one means of collecting them is to write a procedure that repeatedly backtracks into that goal to get another solution. In order to collect all the solutions together, it is necessary to use the database (via assertion) to hold the solutions as they are generated, because backtracking
to redo the goal would undo any list construction that had been done after satisfying the goal.

The writing of such a backtracking loop can be avoided by the use of one of the built-in predicates setof/3, bagof/3 and findall/[3,4], which are described below. These provide a nice logical abstraction, whereas with a user-written backtracking loop the need for explicit side effects (assertions) destroys the declarative interpretation of the code. The built-in predicates are also more efficient than those a user could write.

Please note: If the solutions being collected contain attributed variables (see Section 10.3 [lib-atts], page 391) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 74), then those attributes are not retained in the list of solutions. To retain the attributes, you can use copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 129).

4.13.2 Collecting a Sorted List

setof(Template, Generator, Set) returns the set Set of all instances of Template such that Generator is provable, where that set is non-empty. The term Generator specifies a goal to be called as if by call/1. Set is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 4.8.8 [ref-lte-cte], page 130).

Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case Set will only provide an imperfect representation of what is in reality an infinite set.

If Generator is instantiated, but contains uninstantiated variables that do not also appear in Template, then setof/3 can succeed nondeterminately, generating alternative values for Set corresponding to different instantiations of the free variables of Generator. (It is to allow for such usage that Set is constrained to be non-empty.) For example, if your program contained the clauses

likes(tom, beer).
likes(dick, beer).
likes(harry, beer).
likes(bill, cider).
likes(jan, cider).
likes(tom, cider).

then the call

| ?- setof(X, likes(X,Y), S).

might produce two alternative solutions via backtracking:

Y = beer,
S = [dick,harry,tom] ;
Y = cider,
S = [bill,jan,tom] ;
no
The call

| ?- setof((Y,S), setof(X,likes(X,Y),S), SS).

would then produce

SS = [(beer,[dick,harry,tom]),(cider,[bill,jan,tom])];
no

See Section 11.3.208 [mpg-ref-setof], page 1193.

4.13.2.1 Existential Quantifier

\( X \ ^\exists \ P \) is recognized as meaning “there exists an \( X \) such that \( P \) is true”, and is treated as equivalent to simply calling \( P \). The use of the explicit existential quantifier outside \texttt{setof/3} and \texttt{bagof/3} is superfluous.

Variables occurring in \textit{Generator} will not be treated as free if they are explicitly bound within \textit{Generator} by an existential quantifier. An existential quantification is written:

\( Y^\exists Q \)

meaning “there exists a \( Y \) such that \( Q \) is true”, where \( Y \) is some Prolog variable. For example:

| ?- setof(X, Y^\exists likes(X,Y), S).

would produce the single result

S = [bill,dick,harry,jan,tom] ;
no

in contrast to the earlier example.

Furthermore, it is possible to existentially quantify a term, where all the variables in that term are taken to be existentially quantified in the goal. E.g.

\( A=\text{term}(X,Y), \text{setof}(Z, A^\exists \text{foo}(X,Y,Z), L) \).

will treat \( X \) and \( Y \) as if they are existentially quantified.

4.13.3 Collecting a Bag of Solutions

\texttt{bagof/3} is exactly the same as \texttt{setof/3} except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. This relaxation saves time and space in execution. See Section 11.3.21 [mpg-ref-bagof], page 958.

4.13.3.1 Collecting All Instances

\texttt{findall/3} is a special case of \texttt{bagof/3}, where all free variables in the generator are taken to be existentially quantified. Thus the use of the operator \( ^\exists \) is avoided. Because \texttt{findall/3} avoids the relatively expensive variable analysis done by \texttt{bagof/3}, using \texttt{findall/3} where appropriate rather than \texttt{bagof/3} can be considerably more efficient.
findall/4 is a variant of findall/3 with an extra argument to which the list of solutions is appended. This can reduce the amount of append operations in the program. See Section 11.3.80 [mpg-ref-findall], page 1025.

### 4.13.4 Predicate Summary

\[ ?X \neq :P \]  
there exists an \( X \) such that \( P \) is provable (used in setof/3 and bagof/3)

\[ \text{bagof}(?X,:P,-B) \]  
\( B \) is the bag of instances of \( X \) such that \( P \) is provable  
ISO

\[ \text{findall}(?T,:G,-L) \]  
\( L \) is the list of all solutions \( T \) for the goal \( G \), concatenated with \( R \) or with the empty list  
ISO

\[ \text{findall}(?T,:G,?-L,?-R) \]  
ISO

\[ \text{setof}(?X,:P,-S) \]  
\( S \) is the set of instances of \( X \) such that \( P \) is provable  
ISO

### 4.14 Grammar Rules

This section describes SICStus Prolog’s grammar rules, and the translation of these rules into Prolog clauses. At the end of the section is a list of grammar-related built-in predicates.

#### 4.14.1 Definite Clause Grammars

Prolog’s grammar rules provide a convenient notation for expressing definite clause grammars, which are useful for the analysis of both artificial and natural languages.

The usual way one attempts to make precise the definition of a language, whether it is a natural language or a programming language, is through a collection of rules called a “grammar”. The rules of a grammar define which strings of words or symbols are valid sentences of the language. In addition, the grammar generally analyzes the sentence into a structure that makes its meaning more explicit.

A fundamental class of grammar is the context-free grammar (CFG), familiar to the computing community in the notation of “BNF” (Backus-Naur form). In CFGs, the words, or basic symbols, of the language are identified by “terminal symbols”, while categories of phrases of the language are identified by non-terminal symbols. Each rule of a CFG expresses a possible form for a non-terminal, as a sequence of terminals and non-terminals. The analysis of a string according to a CFG is a parse tree, showing the constituent phrases of the string and their hierarchical relationships.

Context-free grammars (CFGs) consist of a series of rules of the form:

\[ \text{nt} \rightarrow \text{body}. \]

where \( \text{nt} \) is a non-terminal symbol and \( \text{body} \) is a sequence of one or more items separated by commas. Each item is either a non-terminal symbol or a sequence of terminal symbols. The meaning of the rule is that \( \text{body} \) is a possible form for a phrase of type \( \text{nt} \). A non-terminal symbol is written as a Prolog atom, while a sequence of terminals is written as a Prolog list, whereas a terminal may be any Prolog term.
Definite clause grammars (DCGs) are a generalization of context-free grammars and rules corresponding to DCGs are referred to as “Grammar Rules”. A grammar rule in Prolog takes the general form

\[
\text{head} \to \text{body}.
\]

meaning “a possible form for head is body”. Both body and head are sequences of one or more items linked by the standard Prolog conjunction operator ‘,’ (comma).

Definite clause grammars extend context-free grammars in the following ways:

- A non-terminal symbol may be any callable Prolog term.
- A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list ‘[]’. If the terminal symbols are character codes, then such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list (‘[]’ or ‘"’).
- Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. These extra conditions allow the explicit use of procedure calls in the body of a rule to restrict the constituents accepted. Such procedure calls are written enclosed in curly brackets (‘{’ and ‘}’).
- The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
- Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator ‘;’ (semicolon) as in Prolog.
- The cut symbol ‘!’ may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in curly brackets. The same is true for the control constructs. However, all other built-in predicates not enclosed in curly brackets will be treated as non-terminal symbols. The precise meaning of this rule is clarified in Section 4.14.4 [ref-gru-tra], page 192.
- The extra arguments of non-terminals provide the means of building structure (such as parse trees) in grammar rules. As non-terminals are “expanded” by matching against grammar rules, structures are progressively built up in the course of the unification process.
- The extra arguments of non-terminals can also provide a general treatment of context dependency by carrying test and contextual information.

### 4.14.2 How to Use the Grammar Rule Facility

Following is a summary of the steps that enable you to construct and utilize definite clause grammars:

**STEPS:**

1. Write a grammar, using \( \to \)/2 to formulate rules.
2. Compile the file containing the grammar rules. The Load Predicates automatically translate the grammar rules into Prolog clauses.
Chapter 4: The Prolog Language

191

3. Use phrase/[2,3] to parse or generate strings.
OPTIONAL STEPS:
1. Modify the way in which Prolog translates your grammar rules by defining clauses for
user:term_expansion/6; see Section 4.3.5 [ref-lod-exp], page 87.
2. In debugging or in using the grammar facility for more obscure purposes it may be
useful to understand more about expand_term/2.

4.14.3 An Example
As an example, here is a simple grammar that parses an arithmetic expression (made up of
digits and operators) and computes its value. Create a file containing the following rules:
grammar.pl
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).
term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).
number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
In the last rule, C is the character code of a decimal digit.
This grammar can now be used to parse and evaluate an expression by means of the built-in
predicates phrase/[2,3]. See Section 11.3.155 [mpg-ref-phrase], page 1126. For example,


\[ \begin{align*}
\text{| \ ?- } & \ [\text{grammar}]. \\
\text{| \ ?- } & \ \text{phrase}(\text{expr}(Z), "-2+3*5+1"). \\
Z &= 14 \\
\text{| \ ?- } & \ \text{phrase}(\text{expr}(Z), "-2+3*5", \text{Rest}). \\
Z &= 13, \\
\text{Rest} &= []; \\
Z &= 1, \\
\text{Rest} &= "*5"; \\
Z &= -2, \\
\text{Rest} &= "+3*5";
\end{align*} \]

no

4.14.4 Semantics of Grammar Rules

Grammar rules are best explained in terms of an interpreter. The semantics of \texttt{phrase/3} is shown as if defined by the interpreter shown below. The interpreter’s case analysis is as follows:

\textit{control constructs}

(\textit{Including cuts and module prefixes}). The interpreter implements the semantics of the construct, descending into its argument. Note that other built-in predicates are \textit{not} treated this way.

\textit{lists}

Treated as terminal symbols.

\textit{curly brackets}

Treated as procedure calls.

\textit{callable terms with functor N/A}

A grammar rule with head functor \texttt{N/A} is looked up using the imaginary predicate \texttt{-->/2}, unified against, and its body is interpreted. If none exists, then this is treated as a procedure call to a predicate \texttt{N/A+2}.

\textit{non-callable terms}

Raise error exception.

The following points are worth noting:

- The code below defines what constructs of and to what depth grammar rule bodies are interpreted, as opposed to being treated as non-terminals.
- Throughout the interpretation, we must keep track of the module context.
- The head non-terminal of a grammar rule is optionally followed by a sequence of terminals. This feature is not supported by the interpreter, but is supported in the actual implementation.
• As a general rule, the last argument is unified after any side effects, including cuts. This is in line with the rule that output arguments should not be unified before a cut (see Section 9.1 [Eff Overview], page 355). In other words, grammar rules are steadfast.

• The last clause gives a clue to how grammar rules are actually implemented, i.e. by compile-time transformation to ordinary Prolog clauses. A grammar rule with head functor N/A is transformed to a Prolog clause with head functor N/A+2, the extra arguments being S0 and S. -->/2 is not a predicate.

The interpreter is as follows, slightly simplified:
phrase(M:Body, S0, S) :-
    phrase(Body, M, S0, S).

phrase(Var, M, S0, S) :- \+callable(Var), !,
    must_be(Var, callable, phrase(M:Var,S0,S), 1).
phrase(M:Body, _, S0, S) :- !,
    phrase(Body, M, S0, S).
phrase(!, _, S0, S) :- !,
    cut relevant choicepoints,
    S0 = S. % unification AFTER action
phrase((A, B), M, S0, S) :- !,
    phrase(A, M, S0, S1),
    phrase(B, M, S1, S).
phrase((A -> B), M, S0, S) :- !,
    ( phrase(A, M, S0, S1) ->
      phrase(B, M, S1, S)
    ).
phrase((A -> B ; C), M, S0, S) :- !,
    ( phrase(A, M, S0, S1) ->
      phrase(B, M, S1, S)
    ; phrase(C, M, S0, S) )
phrase((A ; B), M, S0, S) :- !,
    ( phrase(A, M, S0, S)
    ; phrase(B, M, S0, S) )
phrase(\+(A), M, S0, S) :- !,
    ( phrase(A, M, S0, _) ->
      fail
    ; S0 = S )
phrase(_^A, M, S0, S) :- !,
    phrase(A, M, S0, S).
phrase(do(Iter,Body), M, S0, S) :- !,
    ( Iter,
      fromto(S0,S1,S2,S)
      do phrase(Body, M, S1, S2) )
phrase(if(A,B,C), M, S0, S) :- !,
    if(phrase(A, M, S0, S1),
      phrase(B, M, S1, S),
      phrase(C, M, S0, S))
phrase(once(A), M, S0, S) :- !,
    ( phrase(A, M, S0, S1) ->
      S1 = S % unification AFTER call
    )
phrase([], _, S0, S) :- !,
    S0 = S.
phrase([H|T], M, S0, S) :- !,
    S0 = [H|S1],
    phrase(T, M, S1, S).
phrase({G}, M, S0, S) :- !,
    call(M:G), % Please note: transparent to cuts
    S0 = S.
As mentioned above, grammar rules are merely a convenient abbreviation for ordinary Prolog clauses. Each grammar rule is translated into a Prolog clause as it is compiled. This translation is exemplified below.

The procedural interpretation of a grammar rule is that it takes an input list of symbols or character codes, analyzes some initial portion of that list, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output lists are not written explicitly in a grammar rule, but are added when the rule is translated into an ordinary Prolog clause. The translations shown differ from the output of \texttt{listing/[0,1]} in that internal translations such as variable renaming are not represented. This is done in the interests of clarity. For example, a rule such as (A) will be depicted as translating into (B) rather than (C).

\[
p(X) --> q(X).
\]

\[
p(X, S0, S) :-
q(X, S0, S).
\]

\[
p(A, B, C) :-
q(A, B, C).
\]

If there is more than one non-terminal on the right-hand side, as in (D), then the corresponding input and output arguments are identified, translating into (E):

\[
p(X, Y) --> q(X), r(X, Y), s(Y).
\]

\[
p(X, Y, S0, S) :-
q(X, S0, S1),
r(X, Y, S1, S2),
s(Y, S2, S).
\]

Terminals are translated using the built-in predicate \texttt{=/2}. For instance, (F) is translated into (G):

\[
p(X) --> [\text{go, to}], q(X), [\text{stop}].
\]

\[
p(X, S0, S) :-
S0 = [\text{go, to}|S1],
q(X, S1, S2),
S2 = [\text{stop}|S].
\]

Extra conditions expressed as explicit procedure calls, enclosed in curly braces, naturally translate into themselves. For example (H) translates to (I):

\[
p(X) --> [X], \{\text{integer}(X), X > 0\}, q(X).
\]
\[ p(X, S_0, S) :\]
\[
\begin{align*}
S_0 &= [X|S_1], \\
\text{integer}(X), \\
X &> 0, \\
q(X, S_1, S).
\end{align*}
\]

Terminals on the left-hand side of a rule, enclosed in square brackets, also translate into a unification. For example, (J) becomes (K):

\[
is(N), [\text{not}] \rightarrow [\text{aint}]. \quad (J)
\]
\[
is(N, S_0, S) :\]
\[
\begin{align*}
S_0 &= [\text{aint}|S_1], \\
S &= [\text{not}|S_1].
\end{align*}
\]

Disjunction and other control constructs have a fairly obvious translation. For example, (L), a rule that equates phrases like “(sent) a letter to him” and “(sent) him a letter”, translates to (M):

\[
\text{args}(X, Y) \rightarrow 
\left( 
\begin{array}{ll}
\text{indir}(X), & [\text{to}], \text{indir}(Y) \\
\text{indir}(Y), & \text{dir}(X)
\end{array}
\right). \quad (L)
\]
\[
\text{args}(X, Y, S_0, S) :\]
\[
\begin{align*}
\text{dir}(X, S_0, S_1), \\
S_1 &= [\text{to}|S_2], \\
\text{indir}(Y, S_2, S) \\
\text{indir}(Y, S_0, S_1), \\
\text{dir}(X, S_1, S)
\end{align*}
\]

In order to look at these translations, declare the grammar rules dynamic and use \texttt{listing/[0,1]}. However, bear in mind that a grammar rule with head functor N/A is transformed to a Prolog clause with head functor N/A+2. For example, the following declaration for grammar rule (L) would enable you to list its translation, (M):

\[
:- \text{dynamic args/4}.
\]

### 4.14.5 Summary of Predicates

**Head --> Body**

A possible form for \textit{Head} is \textit{Body}

- \texttt{expand_term(+T,-X)} \\
  term \(T\) expands to term \(X\) using \texttt{user:term_expansion/6} or grammar rule expansion

- \texttt{phrase(:P, -L)}
- \texttt{phrase(:P, ?L, ?R)}
  \(R\) or the empty list is what remains of list \(L\) after phrase \(P\) has been found
user:term_expansion(+Term1, +Layout1, +Terms1, -Term2, -Layout2, -Tokens2)

hook

Overides or complements the standard transformations to be done by expand_term/2.

4.15 Errors and Exceptions

4.15.1 Overview

Whenever the Prolog system encounters a situation where it cannot continue execution, it throws an exception. For example, if a built-in predicate detects an argument of the wrong type, then it throws a type_error exception. The manual page description of each built-in predicate lists the kinds of exceptions that can be thrown by that built-in predicate.

The default effect of throwing an exception is to terminate the current computation and then print an error message. After the error message, you are back at Prolog's top level. For example, if the goal

\[ X \text{ is a/2} \]

is executed somewhere in a program, then you get

\[
\text{! Type error in argument 2 of (is)/2}
\]

\[
\text{! expected evaluable, but found a/0}
\]

\[
\text{! goal: _255 is a/2}
\]

\[
\text{| ?-}
\]

Particular things to notice in this message are:

'!' This character indicates that this is an error message rather than a warning or informational message.

'Type Error' This is the error class. Exceptions thrown by the Prolog system are called errors. Every error is categorized into one of a small number of classes. The classes are listed in Section 4.15.4 [ref-ere-err], page 200.

'goal:' The goal that caused the exception to be thrown.

Built-in predicates check their arguments, but predicates exported by library modules generally do not, although some do check their arguments to a lesser or greater extent.

4.15.2 Throwing Exceptions

You can throw exceptions from your own code using:
throw(+ExceptionTerm)  ISO

The argument to this predicate is the exception term, an arbitrary non-variable term. See Section 11.3.235 [mpg-ref-throw], page 1228.

**Please note:** If the exception term contains attributed variables (see Section 10.3 [lib-atts], page 391) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 74), then those attributes do not become part of the exception. To retain the attributes, you can use `copy_term/3` (see Section 4.8.7 [ref-lte-cpt], page 129).

### 4.15.3 Handling Exceptions

It is possible to protect a part of a program against abrupt termination in the event of an exception. There are several ways to do this:

- Trap exceptions to a particular goal by calling `catch/3` as described in Section 4.15.3.1 [ref-ere-hex-pgo], page 198.
- Handle undefined predicates or subsets of them through the hook predicate `user:unknown_predicate_handler/3`; see Section 4.15.3.2 [ref-ere-hex-hup], page 199.
- Trap exceptions matching `Exception` to the debugger by defining the following hook predicate:

  ```prolog
  user:error_exception(+Exception) hook, development
  ```

  See Section 11.3.73 [mpg-ref-error_exception], page 1017.
- Control syntax errors with the `syntax_errors` Prolog flag or with the same option to `read_term/[2,3]`; see Section 4.15.4.11 [ref-ere-err-syn], page 208.
- Control existence and permission errors in the context of opening files with the `fileerrors` Prolog flag or with the same option to `absolute_file_name/3`; see Section 4.15.4.7 [ref-ere-err-exi], page 206, and Section 4.15.4.8 [ref-ere-err-per], page 206.

#### 4.15.3.1 Protecting a Particular Goal

The built-in predicate `catch/3` enables you to handle exceptions to a specific goal:

```prolog
catch(:ProtectedGoal, ?ExceptionTerm, :Handler) ISO
```

`ProtectedGoal` is executed. If all goes well, then it will behave just as if you had written `call(ProtectedGoal)` instead. If an exception is thrown while `ProtectedGoal` is running, then Prolog will abandon `ProtectedGoal` entirely. Any bindings made by `ProtectedGoal` will be undone, just as if it had failed. If the exception occurred in the scope of a `call_cleanup(Goal, Cleanup)`, then `Cleanup` will be called. Side effects, such as asserts and retracts, are not undone, just as they are not undone when a goal fails. After undoing the bindings, Prolog tries to unify the exception term thrown with the `ExceptionTerm` argument. If this unification succeeds, then `Handler` will be executed as if you had written:

```
ExceptionTerm=<the actual exception term>,
Handler
```
If this unification fails, then Prolog will keep searching up the ancestor list looking for another exception handler. If during this search it reaches a recursive call to Prolog from C, then the recursive calls returns with an uncaught exception. If it reaches the top level (or a break level), then an appropriate error message is printed (using print_message/2).

ProtectedGoal need not be determinate. That is, backtracking into ProtectedGoal is possible, and the exception handler becomes reactivated in this case. However, if ProtectedGoal is determinate, then the call to catch/3 is also determinate.

The ProtectedGoal is logically inside the catch/3 goal, but the Handler is not. If an exception is thrown inside the Handler, then this catch/3 goal will not be reactivated. If you want an exception handler that protects itself, then you have to program it, perhaps like this:

```prolog
recursive_catch_handler(Err, Goal, Handler) :-
    catch(Goal, Err,
        recursive_catch_handler(Err, Handler, Handler)).
```

See Section 11.3.34 [mpg-ref-catch], page 972.

Certain built-in and library predicates rely on the exception mechanism, so it is usually a bad idea to let Pattern be a variable, matching any exception. If it must be a variable, then the Handler should examine the exception and pass it on if it is not relevant to the current invocation.

### 4.15.3.2 Handling Unknown Predicates

Users can write a handler for the specific exception occurring when an undefined predicate is called by defining clauses for the hook predicate user:unknown_predicate_handler/3. This can be thought of as a “global” exception handler for this particular exception, because unlike catch/3, its effect is not limited to a particular goal. Furthermore, the exception is handled at the point where the undefined predicate is called.

The handler can be written to apply to all unknown predicates, or to a class of them. The reference page contains an example of constraining the handler to certain predicates.

If call(Module:Goal) is the trapped call to the undefined predicate, then the hook is called as:

```
user:unknown_predicate_handler(+Goal, +Module, -NewGoal) hook
```

If this succeeds, then Prolog replaces the call to the undefined predicate with the call to Module:NewGoal. Otherwise, the action taken is governed by the unknown Prolog flag (see Section 4.9.4 [ref-lps-flg], page 136), the allowed values of which are:

- **trace** Causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity. Not available in runtime systems.
- **error** Causes calls to such predicates to raise an exception (the default).
warning
   Causes calls to such predicates to display a warning message and then fail.
fail
   Causes calls to such predicates to fail.

Finally, this flag can be accessed by the built-in predicate:

unknown(?OldValue, ?NewValue)  

This unifies OldValue with the current value, sets the flag to NewValue, and prints a message about the new value. See Section 11.3.244 [mpg-ref-unknown_predicate_handler], page 1238.

4.15.4 Error Classes

Exceptions thrown by the Prolog system are called errors.

Error terms have the form:

   error(ISO_Error, SICStus_Error)

where the principal functor of ISO_Error (resp. SICStus_Error) indicates the error class (see Section 4.15.4 [ref-ere-err], page 200). The classification always coincides.

Please note: Do Not throw error terms except when you re-throw a previously caught error term. They correspond to the exceptions thrown by the built-in predicates. Throwing such forged error terms can lead to unexpected results.

See Section 10.49 [lib-types], page 869, for an alternative interface to throwing error exceptions, which tries to include line number information for source-linked debugging.

Error messages like the one shown earlier are printed using the built-in predicate print_message/2. One of the arguments to print_message/2 is the exception term. print_message/2 can be customized, as described in Section 4.16 [ref-msg], page 212.

The set of error classes used by the system has been kept small:

Instantiation Error  
   An input argument is insufficiently instantiated.

Uninstantiation Error  
   An input argument is too instantiated.

Type Error  
   An input argument is of the wrong type.

Domain Error  
   An input argument is illegal but of the right type.

Evaluation Error  
   An incorrect arithmetic expression was evaluated.

Representation Error  
   A computed value cannot be represented.
Existence Error
Something does not exist.

Permission Error
Specified operation is not permitted.

Context Error
Specified operation is not permitted in this context.

Consistency Error
Two otherwise correct values are inconsistent with each other.

Syntax Error
Error in reading a term.

Resource Error
Some resource limit has been exceeded.

System Error
An error detected by the operating system.

The format of the exception thrown by the built-in predicates is:

\[ \text{error}(\text{ISO\_Error}, \text{SICStus\_Error}) \]

where \text{ISO\_Error} is the error term prescribed by the ISO Prolog standard, while \text{SICStus\_Error} is the part defined by the standard to be implementation defined. This so called SICStus error term has the same principal functor as \text{ISO\_Error} but more arguments containing additional information, such as the goal and the argument number causing the error. Arguments are numbered from 1 upwards. An argument number given as zero means that an unspecific argument caused the error.

The list below itemizes the error terms, showing the \text{ISO\_Error} and \text{SICStus\_Error} form of each one, in that order. The SICStus and ISO error terms always belong to the same error class, but note that the Context and Consistency error classes are extensions to the ISO Prolog standard.

The goal part of the error term may optionally have the form $@($Callable,PC$)$ where PC is an internal encoding of the line of code containing the culprit goal or one of its ancestors. To decompose an annotated goal \text{AGoal} into a \text{Goal} proper and a \text{SourceInfo} descriptor term, indicating the source position of the goal, use:

?– goal_source_info(AGoal, Goal, SourceInfo).

The reference page gives details about the \text{SourceInfo} format. See Section 11.3.98 [mpg-ref-goal_source_info], page 1054.

\text{instantiation\_error}
\text{instantiation\_error}(\text{Goal}, \text{ArgNo})

\text{Goal} was called with insufficiently instantiated arguments.
uninstantiation_error(Culprit)  

ISO

uninstantiation_error(Goal, ArgNo, Culprit)  

Goal was called with too instantiated arguments, expecting Culprit to be uninstantiated.

type_error(TypeName, Culprit)  

ISO

type_error(Goal, ArgNo, TypeName, Culprit)  

Goal was called with the wrong type of argument(s). TypeName is the expected type and Culprit what was actually found.

domain_error(Domain, Culprit)  

ISO

domain_error(Goal, ArgNo, Domain, Culprit)  

Goal was called with argument(s) of the right type but with illegal value(s). Domain is the expected domain and Culprit what was actually found.

existence_error(ObjectType, Culprit)  

ISO

existence_error(Goal, ArgNo, ObjectType, Culprit, Reserved)  

Something does not exist as indicated by the arguments. See Section 4.15.4.7 [ref-ere-err-exi], page 206, for ways of controlling this behavior.

permission_error(Operation, ObjectType, Culprit)  

ISO

permission_error(Goal, Operation, ObjectType, Culprit, Reserved)  

The Operation is not permitted on Culprit of the ObjectType. See Section 4.15.4.8 [ref-ere-err-per], page 206, for ways of controlling this behavior.

context_error(ContextType, CommandType)  

context_error(Goal, ContextType, CommandType)  

The CommandType is not permitted in ContextType.

syntax_error(Message)  

ISO

syntax_error(Goal, Position, Message, Tokens, AfterError)  

A syntax error was found when reading a term with read/[1,2] or assembling a number from its characters with number_chars/2 or number_codes/2. See Section 4.15.4.11 [ref-ere-err-syn], page 208, for ways of controlling this behavior.

evaluation_error(ErrorType, Culprit)  

ISO

evaluation_error(Goal, ArgNo, ErrorType, Culprit)  

An incorrect arithmetic expression was evaluated.

representation_error(ErrorType)  

ISO

representation_error(Goal, ArgNo, ErrorType)  

A representation error occurs when the program tries to compute some well-defined value that cannot be represented, such as a compound term with arity > 255.

consistency_error(Culprit1, Culprit2, Message)  

ISO

consistency_error(Goal, Culprit1, Culprit2, Message)  

A consistency error occurs when two otherwise valid values or operations have been specified that are inconsistent with each other.
**resource_error( ResourceType)**  
ISO

**resource_error(Goal, ResourceType)**  
A resource error occurs when SICStus Prolog has insufficient resources to complete execution. The only value for `ResourceType` that is currently in use is memory.

**system_error**  
ISO

**system_error(Message)**  
An error occurred while dealing with the operating system.

Most exception terms include a copy of the `Goal` that threw the exception.

In general, built-in predicates that cause side effects, such as the opening of a stream or asserting a clause into the Prolog database, attempt to do all error checking before the side effect is performed. Unless otherwise indicated in the documentation for a particular predicate or error class, it should be assumed that goals that throw exceptions have not performed any side effect.

### 4.15.4.1 Instantiation Errors

An instantiation error occurs when a predicate or command is called with one of its input arguments insufficiently instantiated.

The `SICStus_Error` term associated with an instantiation error is

\[
\text{instantiation_error}(\text{Goal}, \text{ArgNo})
\]

where `ArgNo` is a non-negative integer indicating which argument caused the problem. `ArgNo`=0 means that the problem could not be localized to a single argument.

Note that the `ArgNo`th argument of `Goal` might well be a non-variable: the error is in that argument. For example, the goal

\[
X \text{ is } Y+1
\]

where `Y` is uninstantiated throws the exception

\[
\text{error}(\text{instantiation_error}, \\
\text{instantiation_error}(_A \text{ is } _B+1,2))
\]

because the second argument to `is/2` contains a variable.

### 4.15.4.2 Uninstantiation Errors

An uninstantiation error occurs when a predicate or command is called with one of its input arguments instantiated when an unbound variable was expected.

The `SICStus_Error` term associated with an instantiation error is

\[
\text{uninstantiation_error}(\text{Goal}, \text{ArgNo}, \text{Culprit})
\]

For example, the goal
open(f, write, bar)

throws the exception

\[
\text{error}(<\text{uninstantiateation\_error(bar)}, \text{uninstantiateation\_error(open(f,write,bar),3,bar)})
\]
because the third argument was not a variable.

### 4.15.4.3 Type Errors

A type error occurs when an input argument is of the wrong type. In general, a type is taken to be a class of terms for which there exists a unary type test predicate. Some types are built-in, such as \text{atom/1} and \text{integer/1}.

The type of a term is the sort of thing you can tell just by looking at it, without checking to see how \textit{big} it is. So “integer” is a type, but “non-negative integer” is not, and “atom” is a type, but “atom with 5 letters in its name” and “atom starting with ‘x’” are not.

The point of a type error is that you have \textit{obviously} passed the wrong sort of argument to a command; perhaps you have switched two arguments, or perhaps you have called the wrong predicate, but it is not a subtle matter of being off by one.

Most built-in predicates check all their input arguments for type errors.

The \textit{SICStus\_Error} term associated with a type error is

\[
\text{type\_error}(\text{Goal}, \text{ArgNo}, \text{TypeName}, \text{Culprit})
\]

\text{ArgNo} \quad \text{Culprit} occurs somewhere in the \text{ArgNo}th argument of \text{Goal}.

\text{TypeName} \quad \text{says what sort of term was expected; it should be the name of a unary predicate that is true of whatever terms would not provoke a type error.}

\text{Culprit} \quad \text{is the actual term being complained about: \text{TypeName}(\text{Culprit}) should be false.}

For example:

\[
\text{?- catch}((\text{write}(3),1), \text{Error}, \text{true}).
\]

\[
\text{Error} = \text{error(type\_error(callable,(write(3),1)), type\_error(user:(write(3),1),0,callable,(write(3),1)))}
\]

### 4.15.4.4 Domain Errors

A domain error occurs when an input argument is of the right type but there is something wrong with its value. For example, the second argument to \text{open/[3,4]} is supposed to be an atom that represents a valid mode for opening a file, such as \text{read} or \text{write}. If a number or a compound term is given instead, then that is a type error. If an atom is given that is not a valid mode, then that is a domain error.
The main reason that we distinguish between type errors and domain errors is that they usually represent different sorts of mistakes in your program. A type error usually indicates that you have passed the wrong argument to a command, whereas a domain error usually indicates that you passed the argument you meant to check, but you hadn’t checked it enough.

The SICStus_Error term associated with a domain error is

\[
\text{domain_error}(\text{Goal}, \text{ArgNo}, \text{DomainName}, \text{Culprit})
\]

The arguments correspond to those of the SICStus_Error term for a type error, except that DomainName is not in general the name of a unary predicate: it needn’t even be an atom. For example, if some command requires an argument to be an integer in the range 1..99, then it might use \text{between}(1,99) as the DomainName. With respect to the date_plus example under \textbf{Type Errors}, if the month had been given as 13, then it would have passed the type test but would throw a domain error.

For example, the goal

\[
\text{open(somefile,rread,S)}
\]

throws the exception

\[
\text{error(domain_error(io_mode,rread),}
\]
\[
\text{domain_error(open(somefile,rread,_A),2,io_mode,rread))}
\]

The Message argument is used to provide extra information about the problem.

\textbf{4.15.4.5 Evaluation Errors}

An evaluation error occurs when an incorrect arithmetic expression was evaluated. Floating-point overflow is another evaluation error. The SICStus_Error term associated with an evaluation error is

\[
\text{evaluation_error}(\text{Goal}, \text{ArgNo}, \text{TypeName}, \text{Culprit})
\]

This has the same arguments as a type error.

\textbf{4.15.4.6 Representation Errors}

A representation error occurs when your program calls for the computation of some well-defined value that cannot be represented.

Most representation errors are some sort of overflow. For example, creating a compound term with arity greater than 255 results in a representation error.

The SICStus_Error term for a representation error is

\[
\text{representation_error}(\text{Goal}, \text{ArgNo}, \text{Message})
\]

ArgNo identifies the argument of the goal that cannot be constructed.
Message further classifies the problem. A message of 0 or "" provides no further information.

4.15.4.7 Existence Errors

An existence error occurs when a predicate attempts to access something that does not exist. For example, trying to compile a file that does not exist, erasing a database reference that has already been erased.

The SICStus_Error term associated with an existence error is

\[
\text{existence_error} (\text{Goal, ArgNo, ObjectType, Culprit, Message})
\]

- **ArgNo** index of argument of Goal where Culprit appears
- **ObjectType** expected type of non-existent object
- **Culprit** name for the non-existent object
- **Message** the constant 0 or '', or some additional information provided by the operating system or other support system indicating why Culprit is thought not to exist.

For example, ‘see('../brother/niece')’ might throw the exception

\[
\text{error} (\text{existence_error (source_sink, '../brother/niece')},
\quad \text{existence_error (see ('../brother/niece'), 1, file, '../brother/niece', 0)})
\]

An existence error does not necessarily cause an exception to be thrown. For I/O predicates, the behavior can be controlled with the fileerrors Prolog flag (see Section 4.9.4 [ref-lps-flg], page 136) or with the fileerrors/1 alias file_errors/1 option to absolute_file_name/3. The following values are possible:

- **on** (fileerrors flag value)
  - **error** (absolute_file_name/3 fileerrors value)
    - Throw an exception if a given file cannot be opened. The default.
- **off** (fileerrors flag value)
  - **fail** (absolute_file_name/3 fileerrors value)
    - Merely fail if a given file cannot be opened.

4.15.4.8 Permission Errors

A permission error occurs when an operation is attempted that is among the kinds of operation that the system is in general capable of performing, and among the kinds that you are in general allowed to request, but this particular time it is not permitted. Usually, the reason for a permission error is that the owner of one of the objects has requested that the object be protected.

For example, an attempt to assert or retract clauses for a predicate that has not been declared :-dynamic is rejected with a permission error.

File system protection is another major source of such errors.
The SICStus_Error term associated with a permission error is

\[ \text{permission_error}(\text{Goal, Operation, ObjectType, Culprit, Message}) \]

*Operation* operation attempted; *Operation* exists but is not permitted with *Culprit*.

*ObjectType* Culprit’s type.

*Culprit* name of protected object.

*Message* provides such operating-system-specific additional information as may be available. A message of 0 or ‘’ provides no further information.

A permission error does not necessarily cause an exception to be thrown. For I/O predicates, the behavior can be controlled with the fileerrors Prolog flag (see Section 4.9.4 [ref-lps-flg], page 136) or with the fileerrors/1 alias file_errors/1 option to absolute_file_name/3, exactly as for existence errors.

### 4.15.4.9 Context Errors

A context error occurs when a goal or declaration appears in the wrong place. There may or may not be anything wrong with the goal or declaration as such; the point is that it is out of place. Calling multifile/1 as a goal is a context error, as is having :-module/2 anywhere but as the first term in a source file. This error class is an extension to the ISO Prolog standard.

The SICStus_Error term associated with a context error is

\[ \text{context_error}(\text{Goal, ContextType, CommandType}) \]

*ContextType* the context in which the command was attempted.

*CommandType* the type of command that was attempted.

### 4.15.4.10 Consistency Errors

A consistency error occurs when two otherwise valid values or operations have been specified that are inconsistent with each other. For example, if two modules each import the same predicate from the other, then that is a consistency error. This error class is an extension to the ISO Prolog standard.

The SICStus_Error term associated with a consistency error is

\[ \text{consistency_error}(\text{Goal, Culprit1, Culprit2, Message}) \]

*Culprit1* One of the conflicting values/operations.

*Culprit2* The other conflicting value/operation.

*Message* Additional information, or 0, or ‘’.
4.15.4.11 Syntax Errors

A syntax error occurs when data are read from some external source but have an improper format or cannot be processed for some other reason. This category mainly applies to \texttt{read/1} and its variants.

The SICStus \texttt{Error} term associated with a syntax error is

\begin{verbatim}
syntax_error(\texttt{Goal, Position, Message, Left, Right})
\end{verbatim}

where \texttt{Goal} is the goal in question, \texttt{Position} identifies the position in the stream where reading started, and \texttt{Message} describes the error. \texttt{Left} and \texttt{right} are lists of tokens before and after the error, respectively.

Note that the \texttt{Position} is where reading started, not where the error is.

\texttt{read/1} does two things. First, it reads a sequence of characters from the current input stream up to and including a clause terminator, or the end of file marker, whichever comes first. Then it attempts to parse the sequence of characters as a Prolog term. If the parse is unsuccessful, then a syntax error occurs. Thus, in the case of syntax errors, \texttt{read/1} disobeys the normal rule that predicates should detect and report errors before they perform any side effects, because the side effect of reading the characters has been done.

A syntax error does not necessarily cause an exception to be thrown. For I/O predicates, but not for \texttt{number_chars/2} and \texttt{number_codes/2}, The behavior can be controlled via the \texttt{syntax_errors} Prolog flag (see Section 4.9.4 [ref-lps-flg], page 136), or via the \texttt{syntax_errors/1} option to \texttt{read_term/[2,3]}. The following values are possible:

- \texttt{quiet}: When a syntax error is detected, nothing is printed, and \texttt{read/1} just quietly fails.
- \texttt{dec10}: This provides compatibility with other Prologs: when a syntax error is detected, a syntax error message is issued with \texttt{print_message/2}, and the \texttt{read} is repeated. This is the default.
- \texttt{fail}: This provides compatibility with other Prologs. When a syntax error is detected, a syntax error message is printed on \texttt{user_error}, and the \texttt{read} then fails.
- \texttt{error}: When a syntax error is detected, an exception is thrown.

4.15.4.12 Resource Errors

A resource error occurs when some resource runs out. For example, you can run out of virtual memory, or you can exceed the operating system limit on the number of simultaneously open files.

Often a resource error arises because of a programming mistake: for example, you may exceed the maximum number of open files because your program does not close files when it has finished with them. Or, you may run out of virtual memory because you have a non-terminating recursion in your program.
The SICStus_Error term for a resource error is

\[
\text{resource\_error}(\text{Goal}, \text{Resource})
\]

**Goal** A copy of the goal, or 0 if no goal was responsible; for example there is no particular goal to blame if you run out of virtual memory.

**Resource** identifies the resource that was exhausted. The only value currently in use is memory.

### 4.15.4.13 System Errors

System errors are problems that the operating system notices (or causes). Note that many of the exception indications returned by the operating system (such as “file does not exist”) are mapped to Prolog exceptions; it is only really unexpected things that show up as system errors.

The SICStus_Error term for a system error is

\[
\text{system\_error} (\text{Message})
\]

where Message is not further specified.

### 4.15.5 An Example

Suppose you want a routine that is to prompt for a file name and open the file if it can; otherwise it is to prompt the user for a replacement name. If the user enters an empty name, then it is to fail. Otherwise, it is to keep asking the user for a name until something works, and then it is to return the stream that was opened. There is no need to return the file name that was finally used. We can get it from the stream. Code:
retry_open_output(Stream) :-
    ask_query(filename, format('Type name of file to open
',[]), -, FileName),
    FileName \== ' ',
    catch(open(FileName, write, Stream),
    Error,
        ( Error = error(_,Excp),
          file_error(Excp)
        -> print_message(warning, Excp),
          retry_open_output(Stream)
        ; throw(Error) )).

file_error(existence_error(open(_,_,_), 1, _, _, _)).
file_error(permission_error(open(_,_,_), _, _, _, _)).

:- multifile 'SU_messages':query_class/5.
'SU_messages':query_class(filename, '> ', line, atom_codes, help_query) :- !.

:- multifile 'SU_messages':query_map/4.
'SU_messages':query_map(atom_codes, Codes, success, Atom) :- !,
    (Codes==end_of_file -> Atom = '' ; atom_codes(Atom, Codes)).

Sample session:

| ?- retry_open_output(S).
Type name of file to open
> nodir/nofile
* Existence error in argument 1 of open/3
* file '/tmp/nodir/nofile' does not exist
* goal: open('nodir/nofile',write,_701)
Type name of file to open
> newfile
S = '$stream'(3491752)

What this example does not catch is as interesting as what it does. All errors except existence and permission errors are re-thrown, as they represent errors in the program. The example also shows that you generally do not want to catch all exceptions that a particular goal might throw.

4.15.6 Legacy Predicates

Exception handling for Prolog was originally introduced in Quintus Prolog, and later inherited by SICStus Prolog, with an API that predated the ISO standard. This API is still supported but should be regarded as legacy, and consists of the two predicates raise_exception/1 and on_exception/3:
on_exception(?Template, :ProtectedGoal, :Handler)
   Equivalent to catch(:ProtectedGoal, ?Template, :Handler). Any exception term matching Template is caught and handled. See Section 11.3.145 [mpg-ref-on_exception], page 1112.

raise_exception(+ExceptionTerm)
   If ExceptionTerm matches one of the SICStus error terms listed in Section 4.15.4 [ref-ere-err], page 200, then the corresponding error term error(ISO_Error, SICStus_Error) is constructed and thrown. Otherwise, ExceptionTerm is thrown as is.

Prior to release 4.3, throw/1 and raise_exception/3 used to be equivalent and throw their argument as is, whereas catch/3 and on_exception/3 both used to attempt to recognize and expand SICStus error terms into error/2 terms. Unless a forged SICStus error term is thrown by throw/1, the net behavior is unchanged. See Section 11.3.182 [mpg-ref-raise_exception], page 1159.

4.15.7 Interrupting Execution
There exist more drastic means of interrupting the normal control flow. To invoke a recursive top level, use:

?- break.

See Section 11.3.27 [mpg-ref-break], page 965.

To exit from Prolog, use:

?- halt.

To exit from Prolog with return code Code, use:

?- halt(Code).

See Section 11.3.101 [mpg-ref-halt], page 1057.

To abort the execution of the current query and return to the top level, use:

?- abort.

See Section 11.3.2 [mpg-ref-abort], page 926.

Please note: halt/[0,1] and abort/0 are implemented by throwing a reserved exception, which has a handler at the top level of development systems and executables built with the spld tool. Thus they give the opportunity for cleanup goals (see call_cleanup/2) to run.

4.15.8 Summary of Predicates
abort        abort execution of the program; return to current break level
break        start a new break level to interpret commands from the user
catch(:P, ?E, :H)
    specify a handler \( H \) for any exception \( E \) arising in the execution of the goal \( P \)

user:error_exception(+Exception)
    Exception is an exception that traps to the debugger if it is switched on.

goal_source_info(+AGoal, -Goal, -SourceInfo)
    Decomposes the annotated goal \( AGoal \) into a Goal proper and the SourceInfo descriptor term, indicating the source position of the goal.

halt
halt(C)
    exit from Prolog with exit code \( C \)

on_exception(?E, :P, :H)
    specify a handler \( H \) for any exception \( E \) arising in the execution of the goal \( P \)

raise_exception(+E)
    raise exception \( E \)

throw(+E)
    raise exception \( E \)

unknown(?OldValue, ?NewValue)
    access the unknown Prolog flag and print a message

user:unknown_predicate_handler(+Goal, +Module, -NewGoal)
    tell Prolog to call \( Module:NewGoal \) if \( Module:Goal \) is undefined

4.16 Messages and Queries

This section describes the two main aspects of user interaction, displaying messages and querying the user. We will deal with these two issues in turn.

4.16.1 Message Processing

Every message issued by the Prolog system is displayed using a single predicate:

\[
\text{print_message(Severi ty, Message)}
\]

*Message* is a term that encodes the message to be printed. The format of message terms is subject to change, but can be inspected in the file *library('SU_messages')* of the SICStus Prolog distribution.

The atom *Severity* specifies the type (or importance) of the message. The following table lists the severities known to the SICStus Prolog system, together with the line prefixes used in displaying messages of the given severity:

<table>
<thead>
<tr>
<th>Severity</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>'!'</td>
</tr>
<tr>
<td>warning</td>
<td>'*'</td>
</tr>
<tr>
<td>informational</td>
<td>'%'</td>
</tr>
<tr>
<td>help</td>
<td>''</td>
</tr>
<tr>
<td>query</td>
<td>''</td>
</tr>
</tbody>
</table>

For error messages

For warning messages

For informational messages

For help messages

For query texts (see Section 4.16.3 [Query Processing], page 216)
silent '' a special kind of message, which normally does not produce any output, but can be intercepted by hooks

print_message/2 is a built-in predicate, so that users can invoke it to have their own messages processed in the same way as the system messages.

The processing and printing of the messages is highly customizable. For example, this allows the user to change the language of the messages, or to make them appear in dialog windows rather than on the terminal.

### 4.16.1.1 Phases of Message Processing

Messages are processed in two major phases. The user can influence the behavior of each phase using appropriate hooks, described later.

The first phase is called the *message generation phase*: it determines the text of the message from the input (the abstract message term). No printing is done here. In this phase the user can change the phrasing or the language of the messages.

The result of the first phase is created in the form of a *format-command list*. This is a list whose elements are *format-commands*, or the atom nl denoting the end of a line. A format-command describes a piece of text not extending over a line boundary and it can be one of the following:

- **FormatString-Args**
  - `format(FormatString, Args)`
  - This indicates that the message text should appear as if printed by `format(FormatString, Args).

- **write_term(Term, Options)**
  - This indicates that the message text should appear as if printed by `write_term(Term, Options).

- **write_term(Term)**
  - Equivalent to `write_term(Term, Options)` where Options is the actual value of the Prolog flag toplevel_print_options.

As an example, let us see what happens in case of the toplevel call `_ :=:= 3`. An instantiation error is raised by the Prolog system, which is caught, and the abstract message term `error(instantiation_error,instantiation_error(_:=:=3,1))` is generated—the first argument is the goal, and the second argument is the position of the uninstantiated variable within the goal. In the first phase of message processing this is converted to the following format-command list:

```
["Instantiation error"-[],' in argument ^d of ^q'[1,=:= /2],nl,
'goal: '[-[],write_term(_:=:=3),nl]
```

A minor transformation, so-called *line splitting* is performed on the message text before it is handed over to the second phase. The format-command list is broken up along the nl
atoms into a list of lines, where each line is a list of format-commands. We will use the
term format-command lines to refer to the result of this transformation.

In the example above, the result of this conversion is the following:

```
[,['Instantiation error','-[]',' in argument 'd of 'q'-[1,=:= /2]],
 [,'goal: '[-[],write_term(_=:=3)]]
```

The above format-command lines term is the input of the second phase of message processing.

The second phase is called the *message printing phase*, this is where the message is actually
displayed. The severity of the message is used here to prefix each line of the message with
some characters indicating the type of the message, as listed above.

The user can change the exact method of printing (e.g. redirection of messages to a stream,
a window, or using different prefixes, etc.) through appropriate hooks.

In our example the following lines are printed by the second phase of processing:

```
! Instantiation error in argument 1 of =:= /2
! goal: _=:=3
```

The user can override the default message processing mechanism in the following two ways:

- A global method is to define the hook predicate `portray_message/2`, which is the first
  thing called by message processing. If this hook exists and succeeds, then it overrides
  all other processing—nothing further is done by `print_message/2`.
- If a finer method of influencing the behavior of message processing is needed, then there
  are several further hooks provided, which affect only one phase of the process. These
  are described in the following paragraphs.

### 4.16.1.2 Message Generation Phase

The default message generation predicates are located in the `library('SU_messages')`
file, in the 'SU_messages' module, together with other message and query related predicates.
This is advantageous when these predicates have to be changed as a whole (for example
when translating all messages to another language), because this can be done simply by
replacing the file `library('SU_messages')` by a new one.

In the message generation phase three alternative methods are tried:

- First the hook predicate `generate_message_hook/3` is executed, if it succeeds, then it
  is assumed to deliver the output of this phase.
- Next the default message generation is invoked via `SU_messages:generate_message/3`.
- In the case that neither of the above methods succeed, a built-in fall-back message
  generation method is used.
The hook predicate `generate_message_hook/3` can be used to override the default behavior, or to handle new messages defined by the programmer that do not fit the default message generation schemes. The latter can also be achieved by adding new clauses to the hook predicate `SU_messages':generate_message/3.

If both the hook and the default method refuses to handle the message, then the following simple format-command list is generated from the abstract message term `Message`:

```['~q'-[Message],nl]```

This will result in displaying the abstract message term itself, as if printed by `writeq/1`.

For messages of the severity `silent` the message generation phase is skipped, and the `[ ]` format-command list is returned as the output.

### 4.16.1.3 Message Printing Phase

By default this phase is handled by the built-in predicate `print_message_lines/3`. Each line of the message is prefixed with a string depending on the severity, and is printed to `user_error`. The query severity is special—no newline is printed after the last line of the message.

This behavior can be overridden by defining the hook predicate `message_hook/3`, which is called with the severity of the message, the abstract message term and its translation to format-command lines. It can be used to make smaller changes, for example by calling `print_message_lines/3` with a stream argument other than `user_error`, or to implement a totally different display method such as using dialog windows for messages.

For messages of the severity `silent` the message printing phase consists of calling the hook predicate `message_hook/3` only. Even if the hook fails, no printing is done.

### 4.16.2 Message Handling Predicates

```print_message(+Severity, +Message) hookable
Portrays or else writes Message of a given Severity on the standard error stream.
See Section 11.3.162 [mpg-ref-print_message], page 1137.
```

```portray_message(+Severity, +Message) hook
user:portray_message(+Severity, +Message)
Tells print_message/2 what to do.
```

```generate_message_hook(+Message, -L0, -L) hook
user:generate_message_hook(+Message, -L0, -L)
A way for the user to override the call to 'SU_messages':generate_message/3
in the message generation phase in print_message/2.
```

```'SU_messages':generate_message(+Message, -L0, -L) hook
Predefined message generation rules.
```
message_hook(+Severity, +Message, +Lines) hook
user:message_hook(+Severity, +Message, +Lines)
    Overrides the call to print_message_lines/3 in print_message/2. A way for the user to intercept the abstract message term Message of type Severity, whose translation is Lines, before it is actually printed.

print_message_lines(+Stream, +Severity, +Lines)
    Print the Lines to Stream, preceding each line with a prefix defined by Severity.

goal_source_info(+AGoal, -Goal, -SourceInfo)
    Decomposes the annotated goal AGoal into a Goal proper and the SourceInfo descriptor term, indicating the source position of the goal.

4.16.3 Query Processing
All user input in the Prolog system is handled by a single predicate:

\[
\text{ask_query}(\text{QueryClass}, \text{Query}, \text{Help}, \text{Answer})
\]

QueryClass, described below, specifies the form of the query interaction. Query is an abstract message term specifying the query text, Help is an abstract message term used as a help message in certain cases, and Answer is the (abstract) result of the query.

ask_query/4 is a built-in predicate, so that users can invoke it to have their own queries processed in the same way as the system queries.

The processing of queries is highly customizable. For example, this allows changing the language of the input expected from the user, or to make queries appear in dialog windows rather than on the terminal.

4.16.3.1 Query Classes
Queries posed by the system can be classified according to the kind of input they expect, the way the input is processed, etc. Queries of the same kind form a query class.

For example, queries requiring a yes/no answer form a query class with the following characteristics:

- the text ‘ (y or n) ’ is used as the prompt;
- a single line of text is input;
- if the first non-whitespace character of the input is y or n (possibly in capitals), then the query returns the atom yes or no, respectively, as the abstract answer;
- otherwise a help message is displayed and the query is repeated.

There are built-in query classes for reading in yes/no answers, toplevel queries, debugger commands, etc.

A query class is characterized by a ground Prolog term, which is supplied as the first argument to the query processing predicate ask_query/4. The characteristics of a query class are normally described by the hook predicate.
Chapter 4: The Prolog Language

'SU_messages':query_class(QueryClass, Prompt, InputMethod, MapMethod, FailureMode).

The arguments of the query_class predicate have the following meaning:

**Prompt** an atom to be used for prompting the user.

**InputMethod** a non-variable term, which specifies how to obtain input from the user.

For example, a built-in input method is described by the atom line. This requests that a line is input from the user, and the code list is returned. Another built-in input method is term(Options); here, a Prolog term is read and returned.

The input obtained using InputMethod is called raw input, as it may undergo further processing.

In addition to the built-in input methods, the user can define his/her own extensions.

**MapMethod** a non-variable term, which specifies how to process the raw input to get the abstract answer to the query.

For example, the built-in map method char([yes="yY", no="nN"])] expects a code list as raw input, and gives the answer term yes or no depending on the first non-whitespace character of the input. For another example, the built-in map method = requests that the raw input itself be returned as the answer term—this is often used in conjunction with the input method term(Options).

In addition to the built-in map methods the user can define his/her own extensions.

**FailureMode** This is used only when the mapping of raw input fails, and the query must be repeated. This happens for example if the user typed a character other than y or n in case of the yes_or_no query class. FailureMode determines what to print before re-querying the user. Possible values are:

- help_query print a help message, then print the text of the query again
- help only print the help message
- query only print the text of the query
- none do not print anything

### 4.16.3.2 Phases of Query Processing

Query processing is done in several phases, described below. We will illustrate what is done in each phase through a simple example: the question put to the user when the solution to the toplevel query 'X is 1+1' is displayed, requesting a decision whether to find alternative answers or not:
218 SICStus Prolog

| ?- X is 1+1.

X = 2 ? no
Please enter ";" for more choices; otherwise, <return>
? ;

We focus on the query ‘X = 2 ? ’ in the above script.

The example query belongs to the class `next_solution`, its text is described by the message term `solutions([binding("X",2)])`, and its help text by the message term `bindings_help`. Accordingly, such a query is executed by calling:

```
ask_query(next_solution, /* QueryClass */
solutions([binding("X",2)]), /* Query */
bindings_help, /* Help */
Answer)
```

In general, execution of `ask_query(QueryClass, Query, Help, Answer)` consists of the following phases:

**Preparation phase**

The abstract message terms `Query` (for the text of the query) and `Help` (for the help message) are converted to format-command lines via the message generation and line splitting phases (see Section 4.16.1 [Message Processing], page 212). Let us call the results of the two conversions `QueryLines` and `HelpLines`, respectively. The text of the query, `QueryLines` is printed immediately (via the message printing phase, using `query` severity). `HelpLines` may be printed later, and `QueryLines` printed again, in case of invalid user input.

The characteristics of `QueryClass` (described in the previous subsubsection) are retrieved to control the exact behavior of the further phases.

In our example, the following parameters are sent in the preparation phase:

- `QueryLines` = `[[],['^s = '-'["X"],write_term(2)]]`
- `HelpLines` = `[['Please enter ";" for more choices; otherwise, <return>'-[]]]`
- `Prompt` = `' ? '
- `InputMethod` = `line`
- `MapMethod` = `char([yes=";", no=[0\n]])`
- `FailureMode` = `help`

`QueryLines` is displayed immediately, printing:

```
X = 2
```

(Note that the first element of `QueryLines` is `[]`, therefore the output is preceded by a newline. Also note that no newline is printed at the end of the last line, because the `query` severity is used.)

The subsequent phases will be called repeatedly until the mapping phase succeeds in generating an answer.
**Input phase**

By default, the input phase is implemented by the hook predicate

```prolog
'SU_messages':query_input(InputMethod, Prompt, RawInput).
```

This phase uses the `Prompt` and `InputMethod` characteristics of the query class. `InputMethod` specifies the method of obtaining input from the user. This method is executed, and the result (`RawInput`) is passed on to the next phase.

The use of `Prompt` may depend on `InputMethod`. For example, the built-in input method `line` prints the prompt unconditionally, while the input method `term(_)` passes `Prompt` to `prompt/2`.

In the example, first the `?` prompt is displayed. Next, because `InputMethod` is `line`, a line of input is read, and the code list is returned in `RawInput`. Supposing that the user typed `noRET`, `RawInput` becomes "no" = [32,110,111].

**Mapping phase**

By default, the mapping phase is implemented by the hook predicate

```prolog
'SU_messages':query_map(MapMethod, RawInput, Result, Answer).
```

This phase uses the `MapMethod` parameter to control the method of converting the raw input to the abstract answer.

In some cases `RawInput` is returned as it is, but otherwise it has to be processed (parsed) to generate the answer.

The conversion process may have two outcomes indicated in the `Result` returned:

- success, in which case the query processing is completed with the `Answer` term returned;
- failure, the query has to be repeated.

In the latter case a message describing the cause of failure may be returned, to be printed before the query is repeated.

In our example, the map method is `char([yes-";", no-0'\n]])`. The mapping phase fails for the `RawInput` passed on by the previous phase of the example, as the first non-whitespace character is `n`, which does not match any of the given characters.

**Query restart phase**

This phase is executed only if the mapping phase returned with failure.

First, if a message was returned by the mapping, then it is printed. Subsequently, if requested by the `FailureMode` parameter, then the help message `HelpLines` and/or the text of the query `QueryLines` is printed.

The query is then repeated—the input and mapping phase will be called again to try to get a valid answer.

In the above example, the user typed an invalid character, so the mapping failed. The `char(_)` mapping does not return any message in case of failure.
The FailureMode of the query class is help, so the help message HelpLines is printed, but the query is not repeated:

Please enter ;" for more choices; otherwise, <return>

Having completed the query restart phase, the example script continues by re-entering the input phase: the prompt ‘?’ is printed, another line is read, and is processed by the mapping phase. If the user types the character ; this time, then the mapping phase returns successfully and gives the abstract answer term yes.

4.16.3.3 Hooks in Query Processing

As explained above, the major parts of query processing are implemented in the 'SU_messages' module in the file library('SU_messages') through the following hook predicates:

- 'SU_messages':query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)
- 'SU_messages':query_input(+InputMethod, +Prompt, -RawInput)
- 'SU_messages':query_map(+MapMethod, +RawInput, -Result, -Answer)

This is to enable the user to change the language used, the processing done, etc., simply by changing or replacing the library('SU_messages') file.

To give more control to the user and to make the system more robust (for example if the 'SU_messages' module is corrupt) the so-called four step procedure is used in the above three cases—obtaining the query class parameters, performing the query input and performing the mapping. The four steps of this procedure, described below, are tried in the given order until the first one that succeeds. Note that if an exception is raised within the first three steps, then a warning is printed and the step is considered to have failed.

- First, a hook predicate is tried. The name of the hook is derived from the name of the appropriate predicate by appending '_hook' to it, e.g. user:query_class_hook/5 in case of the query class. If this hook predicate exists and succeeds, then it is assumed to have done all necessary processing, and the following steps are skipped.
- Second, the predicate in the 'SU_messages' module is called (this is the default case, these are the predicates listed above). Normally this should succeed, unless the module is corrupt, or an unknown query-class/input-method/map-method is encountered. These predicates are hook, so new classes and methods can be added easily by the user.
- Third, as a fall-back, a built-in minimal version of the predicates in the original 'SU_messages' is called. This is necessary because the library('SU_messages') file is modifiable by the user, therefore vital parts of the Prolog system (e.g. the toplevel query) could be damaged.
- If all the above steps fail, then nothing more can be done, and an exception is raised.
4.16.3.4 Default Input Methods

The following InputMethod types are implemented by the default 'SU_messages':query_input(InputMethod, Prompt, RawInput) (and these are the input methods known to the third, fall-back step):

- **line**
  - The Prompt is printed, a line of input is read using read_line/2 and the code list is returned as RawInput.

- **term(Options)**
  - Prompt is set to be the prompt (cf. prompt/2), and a Prolog term is read by read_term/2 using the given Options, and is returned as RawInput.

- **FinalTerm**
  - A Prolog term is read as above, and is unified with Term. FinalTerm is returned as RawInput. For example, the T=Vs^term(T,[variable_names(Vs)]) input method will return the term read, paired with the list of variable names.

4.16.3.5 Default Map Methods

The following MapMethod types are known to 'SU_messages':query_map(MapMethod, RawInput, Result, Answer) and to the built-in fall-back mapping:

- **char(Pairs)**
  - In this map method RawInput is assumed to be a code list.
  - Pairs is a list of Name-Abbreviations pairs, where Name is a ground term, and Abbreviations is a code list. The first non-whitespace character of RawInput is used for finding the corresponding name as the answer, by looking it up in the abbreviation lists. If the character is found, then Result is success, and Answer is set to the Name found; otherwise, Result is failure.
  - =
    - No conversion is done, Answer is equal to RawInput and Result is success.

- **debugger**
  - This map method is used when reading a single line debugger command. It parses the debugger command and returns the corresponding abstract command term. If the parse is unsuccessful, then the answer unknown(Line,Warning) is returned. This is to allow the user to extend the debugger command language via debugger_command_hook/2, see Section 5.5 [Debug Commands], page 237. The details of this mapping can be obtained from the library('SU_messages') file.
  - Note that the fall-back version of this mapping is simplified, it only accepts parameterless debugger commands.

4.16.3.6 Default Query Classes

Most of the default query classes are designed to support some specific interaction with the user within the Prolog development environment. The full list of query classes can be inspected in the file library('SU_messages'). Here, we only describe the two classes defined by 'SU_messages':query_class/5 that may be of general use:

- **QueryClass**
  - yes_or_no
  - yes_no_proceed
Prompt ' (y or n) ' ' (y, n, p, s, a, or ?) '
InputMethod line line
MapMethod char([yes="yY", no="nN", proceed="pP", suppress="sS", abort="aA"])
FailureMode help_query help_query

4.16.4 Query Handling Predicates

ask_query(+QueryClass, +Query, +Help, -Answer) hookable
  Prints the question Query, then reads and processes user input according to
  QueryClass, and returns the result of the processing, the abstract answer term
  Answer. The Help message is printed in case of invalid input. See Section 11.3.9
  [mpg-ref-ask_query], page 941.

query_hook(+QueryClass, +Query, +QueryLines, +Help, +HelpLines, -Answer) hook
  user:query_hook(+QueryClass, +Query, +QueryLines, +Help, +HelpLines, -Answer)
  Called by ask_query/4 before processing the query. If this predicate succeeds,
  then it is assumed that the query has been processed and nothing further is
  done.

query_class_hook(+QueryClass, -Prompt, -InputMethod, -MapMethod,
  -FailureMode) hook
  user:query_class_hook(+QueryClass, -Prompt, -InputMethod, -MapMethod,
  -FailureMode)
  Provides the user with a method of overriding the call to 'SU_
  messages':query_class/5 in the preparation phase of query processing. This
  way the default query class characteristics can be changed.

'SU_messages':query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod,
  -FailureMode) hook
  Predefined query class characteristics table.

'SU_messages':query_abbreviation(+QueryClass, -Prompt, -Pairs) hook
  Predefined query abbreviation table.

query_input_hook(+InputMethod, +Prompt, -RawInput) hook
  user:query_input_hook(+InputMethod, +Prompt, -RawInput)
  Provides the user with a method of overriding the call to 'SU_
  messages':query_input/3 in the input phase of query processing. This way
  the implementation of the default input methods can be changed.

'SU_messages':query_input(+InputMethod, +Prompt, -RawInput) hook
  Predefined query input methods.

query_map_hook(+MapMethod, +RawInput, -Result, -Answer) hook
  user:query_map_hook(+MapMethod, +RawInput, -Result, -Answer)
  Provides the user with a method of overriding the call to 'SU_
  messages':query_map/4 in the mapping phase of query processing. This way
  the implementation of the default map methods can be changed.

'SU_messages':query_map(+MapMethod, +RawInput, -Result, -Answer) hook
  Predefined query map methods.
4.16.5 Predicate Summary

ask_query(+QueryClass, +Query, +Help, -Answer)  
  Prints the question Query, then reads and processes user input according to  
  QueryClass, and returns the result of the processing, the abstract answer term  
  Answer. The Help message is printed in case of invalid input.

user:message_hook(+M, +S, +L)  
  Intercepts the printing of a message

'SU_messages':generate_message(+M, ?SO, ?S)  
  Determines the mapping from a message term into a sequence of lines of text  
  to be printed

user:generate_message_hook(+M, ?SO, ?S)  
  Intercepts message before it is given to 'SU_messages':generate_message/3

goal_source_info(+AGoal, -Goal, -SourceInfo)  
  Decomposes the annotated goal AGoal into a Goal proper and the SourceInfo  
  descriptor term, indicating the source position of the goal.

user:portray_message(+Severity, +Message)  
  Tells print_message/2 what to do.

print_message(+S, +M)  
  Hook: print a message M of severity S

print_message_lines(+S, +P, +L)  
  print the message lines L to stream S with prefix P

'SU_messages':query_abbreviation(+T, -P)  
  Specifies one letter abbreviations for responses to queries from the Prolog system

user:query_hook(+QueryClass, +Query, +QueryLines, +Help, +HelpLines, -Answer)  
  Called by ask_query/4 before processing the query. If this predicate succeeds,  
  then it is assumed that the query has been processed and nothing further is  
  done.

'SU_messages':query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod,  
  -FailureMode)  
  Access the parameters of a given QueryClass.

user:query_class_hook(+QueryClass, -Prompt, -InputMethod, -MapMethod,  
  -FailureMode)  
  Provides the user with a method of overriding the call to 'SU_  
  messages':query_class/5 in the preparation phase of query processing. This  
  way the default query class characteristics can be changed.

'SU_messages':query_input(+InputMethod, +Prompt, -RawInput)  
  Implements the input phase of query processing.

user:query_input_hook(+InputMethod, +Prompt, -RawInput)  
  Provides the user with a method of overriding the call to 'SU_  
  messages':query_input/3 in the input phase of query processing. This way  
  the implementation of the default input methods can be changed.
4.17 Other Topics

This section describes topics that do not fit elsewhere.

4.17.1 System Properties and Environment Variables

SICStus Prolog stores some information in named variables called system properties. System properties are used since release 4.1, whereas previous releases used environment variables.

The default value when reading a system property is taken from the corresponding environment variable. This makes system properties largely backward compatible with how environment variables were used in previous releases. Any exceptions to this rule are explicitly mentioned in the documentation.

You can obtain the value of system properties and environment variables using `system:environ/[2,3]` (see Section 10.44 [lib-system], page 768) and `SP_getenv()`.

Some system properties affect the SICStus Prolog initialization process and must therefore be set before SICStus Prolog has been initialized. There are three ways to affect the initial values of system properties:

1. Set the corresponding environment variable.
   
   System properties get their default value from the environment so this is often a convenient method. It was the only method available prior to release 4.1.

2. Pass the `-Dvar=value` option to the sicstus command line tool. See Section 13.1 [too-sicstus], page 1390.

3. Pass an option block to `SP_initialize()` if you initialize the SICStus runtime from C. See Section 6.7.4.1 [Initializing the Prolog Engine], page 330.

Looking up system properties follows the platform convention for environment variables. This means that the lookup is case sensitive on UNIX-like platforms and case insensitive on Windows.

On UNIX-like systems, the environment is assumed to use the UTF-8 character encoding; on Windows, the native Unicode encoding is used.

SICStus reads and copies the process environment during initialization, e.g. in `SP_initialize()`. Any subsequent changes to the process environment will not be detected by SICStus. Note that, at least on UNIX-like systems, changing the process environment, e.g. using `setenv()`, has undefined behavior when the process has multiple threads, which is the case for any process running SICStus.
While copying the environment, each entry in the environment is normalized as follows:

- If it does not contain an equal sign, then the entry is ignored.
- On Windows only, if it starts with an equal sign but has no other equal signs, then the entry is ignored.
- If the entry consists of valid UTF-8, then it is kept as is. This is always true on Windows where a Unicode encoding is used internally by the operating system.
- If the entry does not consist of valid UTF-8, then it is treated as Latin-1 and converted to UTF-8. This cannot happen on Windows.
- On Windows only, if the entry starts with an equal sign, then the equal sign is treated as part of the variable name.

In particular, on UNIX-like systems, this means that the environment should preferably be in UTF-8.

### 4.17.1.1 System Properties Set by SICStus Prolog

The following system properties are set automatically on startup.

**SP_APP_DIR**

The absolute path to the directory that contains the executable. Also available as the application file search path.

**SP_APP_PATH**

The absolute path to the executable. Unlike **SP_APP_DIR**, this system property may not be available under all circumstances.

**SP_RT_DIR**

The full path to the directory that contains the SICStus runtime. If the application was linked statically to the SICStus runtime, then **SP_RT_DIR** is the same as **SP_APP_DIR**. Also available as the runtime file search path.

**SP_RT_PATH**

The absolute path to the SICStus runtime. Unlike **SP_RT_DIR**, this system property may not be available under all circumstances, e.g. if the runtime is not a shared library.

**SP_LIBRARY_DIR**

The absolute path to the directory that contains the SICStus library files. Also available as the initial value of the library file search path.

**SP_TEMP_DIR**

A directory suitable for storing temporary files. It is particularly useful with the open/4 option `if_exists(generate_unique_name)`. Also available as the temp file search path.

**SP_STARTUP_DIR**

During initialization the **SP_STARTUP_DIR** system property will be set to the working directory used by SICStus.

Note that this system property can also be set prior to initialization, in order to tell SICStus which working directory to use. See below.
4.17.1.2 System Properties Affecting Initialization

The following system properties can be set before starting SICStus Prolog.

Some of these override the default sizes of certain areas. For variables ending with ‘SIZE’, the size is in bytes, but may be followed by ‘K’, ‘M’, or ‘G’ meaning 2**10, 2**20 and 2**30 respectively.

Boolean values true and false are represented by ‘yes’ and ‘no’, respectively.

See Section 4.10 [ref-mgc], page 144, for more information about the properties that affect memory management.

SP_PATH  Can be used to specify the location of the Runtime Library. In most cases there is no need to use it, but see Section 6.1 [CPL Notes], page 290.

SP_STARTUP_DIR  The value of this system property, if set, is used as the initial working directory. Note that this system property is also set automatically during initialization; see above.

This value of this system property is not read from the corresponding environment variable.

SP_ALLOW_CHDIR  If this system property is set to ‘no’, then SICStus will not change the process’s working directory when the SICStus working directory changes. This is useful when embedding SICStus and would probably be the better default behavior except for backwards compatibility.

GLOBALSTKSIZE  Controls the initial size of the global stack. Please note: The global stack will not be subsequently trimmed to a size smaller than this initial size.

LOCALSTKSIZE  Controls the initial size of the local stack. Please note: The local stack will not be subsequently trimmed to a size smaller than this initial size.

CHOICESTKSIZE  Controls the initial size of the choicepoint stack. Please note: The choicepoint stack will not be subsequently trimmed to a size smaller than this initial size.

TRAILSTKSIZE  Controls the initial size of the trail stack. Please note: The trail stack will not be subsequently trimmed to a size smaller than this initial size.

GROWTHFACTOR  since release 4.0.8

Meaningful values are between 10 and 100; the default is 62. Controls the rate at which the Prolog stacks grow when they are expanded. These stacks are stored in two data areas: one holding the global and local stacks; another one holding the choicepoint and trail stacks. In addition, both data areas hold some memory reserved for the garbage collector.
The sizes of the two data areas are constrained to take certain discrete values only. The initial size as well as the size after expansion is constrained to be \( w^\ast ((1 + g)^\ast n) \) kilobytes, rounded up to an integral number of words, where \( w \) is the word length in bits, \( g \) is \texttt{GROWTHFACTOR}/100, and \( n \) is an integer.

**PROLOGINITSIZE**

Controls the size of Prolog's initial memory allocation. **Please note:** This initially allocated memory will be kept by the Prolog process until \texttt{SP_deinitialize()} is called or the process exits.

**PROLOGMAXSIZE**

Defines an upper bound on the amount of memory that Prolog will use. If not set, then Prolog will try to use the available address space. Thus if Prolog needs to allocate memory beyond this bound, then a memory resource error will be raised.

**PROLOGINCSIZE**

Controls the amount of memory Prolog asks the operating system for in any given memory expansion.

**PROLOGKEEPSIZE**

Defines a lower bound on the amount of memory retained by \texttt{trimcore/0}. By default, Prolog gets memory from the O/S as the user program executes, whereas \texttt{trimcore/0} endeavors to return free memory back to the O/S. If the programmer knows that their program, once it has grown to a certain size, then is likely to need as much memory for future computations, they can advise Prolog not to return all the free memory back to the operating system by setting this variable. \texttt{trimcore/0} only endeavors to return memory that is allocated above and beyond \texttt{PROLOGKEEPSIZE}; the rest will be kept. **Please note:** The initially allocated memory will be kept by the Prolog process forever, so it is not meaningful to set \texttt{PROLOGKEEPSIZE} smaller than \texttt{PROLOGINITSIZE}.

**SP_ULIMIT_DATA_SEGMENT_SIZE**

Sets the maximum size of the data segment of the Prolog process. The value can be \texttt{unlimited} or a numeric value as described in the first paragraph in this section. A numeric value of zero (0) is equivalent to \texttt{unlimited}. Not used under Windows.

**SP_USE_MALLOC**

If \texttt{yes}, then \texttt{malloc()} et al. will be used for memory management instead of the default memory allocator. This is sometimes useful, e.g$: with debugging tools like \texttt{valgrind}.

**Please note:** Enabling \texttt{malloc()} allocation is not compatible with JIT compilation.

**SP_JIT**

Affects whether the JIT (Just In Time) compiler should be used to compile Prolog code into native (machine) code. One of:

\texttt{yes} 

JIT compilation is enabled and happens automatically. This is the default on platforms that support JIT compilation.
JIT compilation is enabled but does not happen automatically. Currently, there is no documented way to JIT compile predicates manually.

disabled: JIT compilation is disabled completely. Please report if you encounter any reason to disable the JIT compiler.

JIT compilation may need to be disabled on certain security-hardened operating systems, e.g. because they do not permit memory to be both writable and executable.

This system property is ignored on platforms that do not support the JIT compiler.

\textbf{SP\_JIT\_COUNTER\_LIMIT} \textit{since release 4.3}

Determines how many times a predicate can be called before it is JIT compiled. The default is 0.

The heuristics used in order to decide when, and whether, a predicate should be JIT compiled, is subject to change without notice. In particular, this system property may be treated differently in some future release.

\textbf{SP\_JIT\_CLAUSE\_LIMIT} \textit{since release 4.3}

Sets an upper bound on the number of clauses of a predicate for JIT compilation to be attempted. The default is 1024.

\textbf{SP\_SPTI\_PATH} \textit{since release 4.3}

Specify a plugin that will be told when predicates are JIT compiled. The details of writing or using such plugins are currently not documented, and subject to change without notice.

There are two predefined plugins,

\textbf{verbose}: Write verbose information when a predicate is JIT compiled. This can be useful when troubleshooting problems with JIT compilation, e.g. if some predicate takes too long to JIT-compile.

This plugin can be activated by passing \texttt{-DSP\_SPTI\_PATH=verbose} to \texttt{sicstus}.

\textbf{perf}: Tell the Linux perf profiler about the location and name of the JIT compiled predicates. This makes it possible to use \texttt{perf record} etc. for getting accurate and low-overhead profiling info about JIT compiled code.

This plugin can be activated either by passing \texttt{-DSP\_SPTI\_PATH=perf} to \texttt{sicstus}, or, once SICStus has started, with the goal \texttt{use\_module(library(perf)).}

The \texttt{perf} integration is only available on Linux.

\textbf{4.17.1.3 Other System Properties}

In addition some system properties are read during normal execution. In this case the system property is typically not meant to be explicitly set, instead the value is intended to
be taken from the corresponding environment variable. Examples of such system properties include `PATH` and `HOME`. 
5 Debugging

This chapter describes the debugging facilities that are available in development systems. The purpose of these facilities is to provide information concerning the control flow of your program.

The main features of the debugging package are as follows:

- The Procedure Box model of Prolog execution, which provides a simple way of visualizing control flow, especially during backtracking. Control flow is viewed at the predicate level, rather than at the level of individual clauses.
- The ability to exhaustively trace your program or to selectively set spypoints. Spypoints allow you to nominate interesting predicates at which, for example, the program is to pause so that you can interact.
- The ability to set advice points. An advice point allows you to carry out some actions at certain points of execution, independently of the tracing activity. Advice points can be used, e.g. for checking certain program invariants (cf. the assert facility of the C programming language), or for gathering profiling or branch coverage information. Spypoints and advice points are collectively called breakpoints.
- The wide choice of control and information options available during debugging.

The Procedure Box model of execution is also called the Byrd Box model after its inventor, Lawrence Byrd.

Much of the information in this chapter is also in Chapter eight of [Clocksin & Mellish 81], which is recommended as an introduction.

Unless otherwise stated, the debugger prints goals using write_term/3 with the value of the Prolog flag debugger_print_options.

The debugger is not available in runtime systems and the predicates defined in this chapter are undefined; see Section 6.7.1 [Runtime Systems], page 319.

5.1 The Procedure Box Control Flow Model

During debugging, the debugger prints out a sequence of goals in various states of instantiation in order to show the state the program has reached in its execution. However, in order to understand what is occurring it is necessary to understand when and why the debugger prints out goals. As in other programming languages, key points of interest are predicate entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly starts backtracking. The Procedure Box model of Prolog execution views program control flow in terms of movement about the program text. This model provides a basis for the debugging mechanism in development systems, and enables the user to view the behavior of the program in a consistent way.

Let us look at an example Prolog predicate:
The first clause states that $Y$ is a descendant of $X$ if $Y$ is an offspring of $X$, and the second clause states that $Z$ is a descendant of $X$ if $Y$ is an offspring of $X$ and if $Z$ is a descendant of $Y$. In the diagram a box has been drawn around the whole predicate and labeled arrows indicate the control flow in and out of this box. There are five such arrows, which we shall look at in turn.

**Call** This arrow represents initial invocation of the predicate. When a goal of the form $\text{descendant}(X,Y)$ is required to be satisfied, control passes through the Call port of the descendant box with the intention of matching a component clause and then satisfying the subgoals in the body of that clause. Note that this is independent of whether such a match is possible; i.e. first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for descendant when meeting a call to descendant in some other part of the code.

**Exit** This arrow represents a successful return from the predicate. This occurs when the initial goal has been unified with one of the component clauses and the subgoals have been satisfied. Control now passes out of the Exit port of the descendant box. Textually we stop following the code for descendant and go back to the place we came from.

**Redo** This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes through the Redo port of the descendant box. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary.

**Fail** This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing. Control now passes out of the Fail port of the descendant box and the system continues to backtrack. Textually we move back to the code that called this predicate and keep moving backwards up the code looking for choicepoints.
Exception  This arrow represents an exception that was raised in the initial goal, either by a call to throw/1 or raise_exception/1 or by an error in a built-in predicate. See Section 4.15 [ref-ere], page 197. Control now passes out of the Exception port of the descendant box and the system continues to pass the exception to outer levels. Textually we move back to the code that called this predicate and keep moving backwards up the code looking for a call to catch/3 or on_exception/3.

In terms of this model, the information we get about the procedure box is only the control flow through these five ports. This means that at this level we are not concerned with identifying the matching clause, and how any subgoals are satisfied, but rather we only wish to know the initial goal and the final outcome. However, it can be seen that whenever we are trying to satisfy subgoals, what we are actually doing is passing through the ports of their respective boxes. If we were to follow this, then we would have complete information about the control flow inside the procedure box.

Note that the box we have drawn round the predicate should really be seen as an invocation box. That is, there will be a different box for each different invocation of the predicate. Obviously, with something like a recursive predicate, there will be many different Calls and Exits in the control flow, but these will be for different invocations. Since this might get confusing each invocation box is given a unique integer identifier.

In addition to the five basic ports discussed above, there are two more ports for invocations involving a blocked goal:

Block  This port is passed through when a goal is blocked.

Unblock  This port is passed through when a previously blocked goal is unblocked.

5.2 Basic Debugging Predicates

Development systems provide a range of built-in predicates for control of the debugging facilities. The most basic predicates are as follows:

**debug**  
Switches the debugger on, and ensures that the next time control reaches a spypoint, it will be activated. In basic usage this means that a message will be produced and you will be prompted for a command. In order for the full range of control flow information to be available it is necessary to have the debugger on from the start. When it is off the system does not remember invocations that are being executed. (This is because it is expensive and not required for normal running of programs.) You can switch Debug Mode on in the middle of execution, either from within your program or after a ^C (see trace/0 below), but information prior to this will be unavailable. See Section 11.3.60 [mpg-ref-debug], page 1003.
zip

Same as debug/0, except no debugging information is being collected, and so is almost as fast as running with the debugger switched off. See Section 11.3.255 [mpg-ref-zip], page 1253.

trace

Switches the debugger on, and ensures that the next time control enters an invocation box, a message will be produced and you will be prompted for a command. The effect of trace/0 can also be achieved by typing t after a ^C interruption of a program.

At this point you have a number of options. See Section 5.5 [Debug Commands], page 237. In particular, you can just type RET to creep (or single-step) into your program. If you continue to creep through your program, then you will see every entry and exit to/from every invocation box, including compiled code, except for code belonging to hidden modules (see Section 4.11 [ref-mod], page 161). You will notice that the debugger stops at all ports. However, if this is not what you want, then the next predicate gives full control over the ports at which you are prompted. See Section 11.3.237 [mpg-ref-trace], page 1230.

leash(+Mode)

Leashing Mode is set to Mode. Leashing Mode determines the ports of invocation boxes at which you are to be prompted when you creep through your program. At unleashed ports a tracing message is still output, but program execution does not stop to allow user interaction. Note that leash/1 does not apply to spypoints, the leashing mode of these can be set using the advanced debugger features; see Section 5.6 [Advanced Debugging], page 243. Block and Unblock ports cannot be leashed. Mode can be a subset of the following, specified as a list of the following:

call Prompt on Call.
exit Prompt on Exit.
redo Prompt on Redo.
fail Prompt on Fail.
exception Prompt on Exception.

The following shorthands are also allowed:

leash(all)
Same as leash([exception,call,exit,redo,fail]).

leash(half)
Same as leash([exception,call,redo]).

leash(loose)
Same as leash([exception,call]).

leash(tight)
Same as leash([exception,call,redo,fail]).
Chapter 5: Debugging

leash(off)

Same as leash([]).

The initial value of Leashing Mode is \([\text{call, exit, redo, fail, exception}]\) (full leashing). See Section 11.3.110 [mpg-ref-leash], page 1067.

nodebug
development
notrace
development
nozip
development

Switches the debugger off. Any spypoints set will be kept but will never be activated.

debugging
development

Prints information about the current debugging state. This will show:
1. Whether undefined predicates are being trapped.
2. What breakpoints have been set (see below).
3. What mode of leashing is in force (see above).

See Section 11.3.62 [mpg-ref-debugging], page 1005.

5.3 Plain Spypoints

For programs of any size, it is clearly impractical to creep through the entire program. Spypoints make it possible to stop the program whenever it gets to a particular predicate of interest. Once there, one can set further spypoints in order to catch the control flow a bit further on, or one can start creeping.

In this section we discuss the simplest form of spypoints, the plain spypoints. The more advanced forms, the conditional and generic spypoints will be discussed later; see Section 5.6 [Advanced Debugging], page 243.

Setting a plain spypoint on a predicate indicates that you wish to see all control flow through the various ports of its invocation boxes, except during skips. When control passes through any port of an invocation box with a spypoint set on it, a message is output and the user is asked to interact. Note that the current mode of leashing does not affect plain spypoints: user interaction is requested on every port.

Spypoints are set and removed by the following built-in predicates. The first two are also standard operators:

spy :Spec
development

Sets plain spypoints on all the predicates given by the generalized predicate specification Spec.

Examples:

| ?- spy [user:p, m:q/2, m:q/3].
| ?- spy m:[p/1, q/1].

If you set some spypoints when the debugger is switched off, then it will be automatically switched on, entering zip mode. See Section 11.3.216 [mpg-ref-spy], page 1203.
nospy :Spec

Similar to spy Spec except that all the predicates given by Spec will have all previously set spypoints removed from them (including conditional spypoints; see Section 5.6.1 [Creating Breakpoints], page 243). See Section 11.3.132 [mpg-ref-nospy], page 1098.
	nospyall

Removes all the spypoints that have been set, including the conditional and generic ones. See Section 11.3.133 [mpg-ref-nospyall], page 1099.

The commands available when you arrive at a spypoint are described later. See Section 5.5 [Debug Commands], page 237.

5.4 Format of Debugging Messages

We shall now look at the exact format of the message output by the system at a port. All trace messages are output to the standard error stream, using the print_message/2 predicate; see Section 4.16 [ref-msg], page 212. This allows you to trace programs while they are performing file I/O. The basic format is as follows:

    N S 23 F6 Call: T foo(hello, there, _123) ?

N is only used at Exit ports and indicates whether the invocation could backtrack and find alternative solutions. Unintended nondeterminacy is a source of inefficiency, and this annotation can help spot such efficiency bugs. It is printed as ‘?’, indicating that foo/3 could backtrack and find alternative solutions, or ‘ ’ otherwise.

S is a spypoint indicator. If there is a plain spypoint on foo/3, then it is printed as ‘+’. In case of conditional and generic spypoints it takes the form ‘*’ and ‘#’, respectively. Finally, it is printed as ‘ ’, if there is no spypoint on the predicate being traced.

The first number is the unique invocation identifier. It is increasing regardless of whether or not debugging messages are output for the invocations (provided that the debugger is switched on). This number can be used to cross correlate the trace messages for the various ports, since it is unique for every invocation. It will also give an indication of the number of procedure calls made since the start of the execution. The invocation counter starts again for every fresh execution of a command, and it is also reset when retries (see later) are performed.

Just before the second number is an optional frame marker, printed as ‘@’ if present. This marks the location of the current frame, the meaning of which is explained in the next section.

The second number is the current depth; i.e. the number of direct ancestors this goal has, for which a procedure box has been built by the debugger.

The next word specifies the particular port (Call, Exit, Redo, Fail, or Exception).

T is a subterm trace. This is used in conjunction with the ‘^’ command (set subterm), described below. If a subterm has been selected, then T is printed as the sequence of
commands used to select the subterm. Normally, however, $T$ is printed as ' ', indicating that no subterm has been selected.

The goal is then printed so that you can inspect its current instantiation state.

The final ‘?’ is the prompt indicating that you should type in one of the commands allowed (see Section 5.5 [Debug Commands], page 237). If this particular port is unleashed, then you will not get this prompt since you have specified that you do not wish to interact at this point.

At Exception ports, the trace message is preceded by a message about the pending exception, formatted as if it would arrive uncaught at the top level.

Note that calls that are compiled inline are not traced.

Block and Unblock ports are exceptions to the above debugger message format. A message

\[ S \text{- - Block: } p(_133) \]

indicates that the debugger has encountered a blocked goal, i.e. one which is temporarily suspended due to insufficiently instantiated arguments (see Section 4.2.4 [ref-sem-sec], page 74). By default, no interaction takes place at this point, and the debugger simply proceeds to the next goal in the execution stream. The suspended goal will be eligible for execution once the blocking condition ceases to exist, at which time a message

\[ S \text{- - Unblock: } p(_133) \]

is printed. Although Block and Unblock ports are unleashed by default in trace mode, you can make the debugger interact at these ports by using conditional spypoints.

5.5 Commands Available during Debugging

This section describes the particular commands that are available when the system prompts you after printing out a debugging message. All the commands are one or two letter mnemonics, among which some can be optionally followed by an argument. They are read from the standard input stream with any blanks being completely ignored up to the end of the line (RET).

While you are typing commands at a given port, the debugger maintains a notion of current frame of the ancestor stack. The “current goal”, referred to by many commands, is the goal of the current frame. The current frame is initially at the bottom of the ancestor stack, but can be moved by certain commands. If the current frame is above the bottom of the stack, then the port indicator, displayed in front of the current goal, is replaced by the word Ancestor.

The only command that you really have to remember is ‘h’ (followed by RET). This provides help in the form of the following list of available commands.
creep causes the debugger to single-step to the very next port and print a message. Then if the port is leashed (see Section 5.2 [Basic Debug], page 233), then the user is prompted for further interaction. Otherwise, it continues creeping. If leashing is off, then creep is the same as leap (see below) except that a complete trace is printed on the standard error stream.

leap causes the debugger to resume running your program, only stopping when a spypoint is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing. All you need to do is to set spypoints on an evenly spread set of pertinent predicates, and then follow the control flow through these by leaping from one to the other. Debugging information is collected while leaping, so when a spypoint is reached, it is possible to inspect the ancestor goals, or creep into them upon entry to Redo ports.

zip is like leap, except no debugging information is being collected while zipping, resulting in significant savings in memory and execution time.

skip is only valid for Call and Redo ports. It skips over the entire execution of the predicate. That is, you will not see anything until control comes back to this predicate (at either the Exit port or the Fail port). Skip is particularly
useful while creeping since it guarantees that control will be returned after the (possibly complex) execution within the box. If you skip, then no message at all will appear until control returns. This includes calls to predicates with spy-points set; they will be masked out during the skip. No debugging information is being collected while skipping.

If you supply an integer argument, then this should denote an invocation number of an ancestral goal. The system tries to get you to the Exit or Fail port of the invocation box you have specified.

out is a shorthand for skipping to the Exit or Fail port of the immediate ancestor goal. If you supply an integer argument n, then it denotes skipping to the Exit or Fail port of the nth ancestor goal.

quasi-skip is like a combination of zip and skip: execution stops when either control comes back to this predicate, or a spypoint is reached. No debugging information is being collected while quasi-skipping.

An integer argument can be supplied as for skip.

retry can be used at any port (although at the Call port it has no effect). It transfers control back to the Call port of the box. This allows you to restart an invocation when, for example, you find yourself leaving with some weird result. The state of execution is exactly the same as when you originally called, (unless you use side effects in your program; i.e. asserts etc. will not be undone). When a retry is performed the invocation counter is reset so that counting will continue from the current invocation number regardless of what happened before the retry. This is in accord with the fact that you have, in executional terms, returned to the state before anything else was called.

If you supply an integer argument, then it should denote an invocation number of an ancestral goal. The system tries to get you to the Call port of the box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts (!) in your program. In this case the system fails to the latest surviving Call port before the correct one.

fail can be used at any of the four ports (although at the Fail port it has no effect). It transfers control to the Fail port of the box, forcing the invocation to fail prematurely.

If you supply an integer after the command, then it is taken as specifying an invocation number and the system tries to get you to the Fail port of the invocation box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts (!) in your program. In this case the system fails to the latest surviving Fail port before the correct one.

jump to port transfers control back to the prescribed port <p>. Here, <p> is one of: ‘c’, ‘e’, ‘r’, ‘f’, standing for Call, Exit, Redo and Fail ports. Takes an optional integer argument, an invocation number.
Jumping to a Call port is the same as retrying it, i.e. ‘jc’ is the same as the ‘r’ debugger command; and similarly ‘jf’ is the same as ‘f’.

The ‘je’ jump to Exit port command transfers control back to the Exit port of the box. It can be used at a Redo or an Exit port (although at the latter it has no effect). This allows you to restart a computation following an Exit port, which you first leapt over, but because of its unexpected failure you arrived at the Redo port. If you supply an integer argument, then it should denote an exact invocation number of an exited invocation present in the backtrace, and then the system will get you to the specified Exit port. The debugger requires here an exact invocation number so that it does not jump too far back in the execution (if an Exit port is not present in the backtrace, it may be a better choice to jump to the preceding Call port, rather than to continue looking for another Exit port).

The ‘jr’ jump to Redo port command transfers control back to the Redo port of the box. It can be used at an Exit or a Redo port (although at the latter it has no effect). This allows you to force the goal in question to try to deliver another solution. If you supply an integer argument, then it should denote an exact invocation number of an exited invocation present in the backtrace, and then the system will get you to the specified Redo port.

d display goal displays the current goal using display/1. See Write (below).

p print goal displays the current goal using print/1. An argument will override the default printdepth, treating 0 as infinity.

w write goal displays the current goal using writeq/1.

g print ancestor goals provides you with a list of ancestors to the current goal, i.e. all goals that are hierarchically above the current goal in the calling sequence. You can always be sure of jumping to the Call or Fail port of any goal in the ancestor list (by using retry etc). If you supply an integer n, then only that number of ancestors will be printed. That is to say, the last n ancestors will be printed counting back from the current goal. Each entry is displayed just as they would be in a trace message, except the current frame is indicated by a @ in front of the invocation number.

t print backtrace is the same as the above, but also shows any goals that have exited nondeterminately and their ancestors. This information shows where there are outstanding choices that the program could backtrack to. If you supply an integer n, then only that number of goals will be printed.

Ancestors to the current goal are annotated with the ‘Call:’ port, as they have not yet exited, whereas goals that have exited are annotated with the ‘Exit:’ port. You can always be sure of jumping to the Exit or Redo port of any goal shown to be exited in the backtrace listing.

The backtrace is a tree rather than a stack: to find the parent of a given goal with depth indicator d, look for the closest goal above it with depth indicator d-1.
frame up: moves the frame up one step. If you supply an argument, then it should denote an invocation number of an existing goal.

frame down: moves the frame down one step. If you supply an argument, then it should denote an invocation number of an existing goal.

print variable bindings endeavors to print the variable bindings of the clause containing the current goal. This is available for both compiled and interpreted code, if the source code was originally loaded with the source_info Prolog flag switched on. The coverage is usually better for compiled code. If you supply an argument, then it should denote an invocation number of an existing goal. Just like the top level, the debugger displays variable bindings as well as any goals that are blocked on a variable found among those bindings, and prompts for the same one-letter commands as the top level does; see Section 3.4.1 [Queries], page 23. To return to the debugger, simply type RET.

print blocked goals prints a list of the goals that are currently blocked in the current debugging session together with the variable that each such goal is blocked on (see Section 4.2.4 [ref-sem-sec], page 74). The goals are enumerated from 1 and up. If you supply an integer n, then only that goal will be printed. Each entry is preceded by the goal number followed by the variable name.

nodebug switches the debugger off. Note that this is the correct way to switch debugging off at a trace point. You cannot use the @ or b commands because they always return to the debugger.

debugging outputs information concerning the status of the debugging package. See the built-in predicate debugging/0.

spy this sets a plain spypoint on the current goal.

spy this conditionally sets a conditional spypoint on the current goal. Prompts for the Conditions, and calls the

\[
\text{spy(Func, Conditions)}
\]
goal, where(Func is the predicate specification of the current invocation. For spy/2, see Section 5.7 [Breakpoint Predicates], page 272.

nospy this removes all spypoints applicable to the current goal. Equivalent to nospy Func, where Func is the predicate specification of the current invocation.

remove this removes the spypoint that caused the debugger to interact at the current port. With an argument n, it removes the breakpoint with identifier n. Equivalent to remove_breakpoints(BID), where BID is the current breakpoint identifier, or the supplied argument (see Section 5.7 [Breakpoint Predicates], page 272).

disable this disables the spypoint that caused the debugger to interact at the current port. With an argument n, it disables the breakpoint with identifier n. Equivalent to disable_breakpoints(BID), where BID is the current
breakpoint identifier, or the supplied argument (see Section 5.7 [Breakpoint Predicates], page 272).

**E** enable this enables all specific spypoints for the predicate at the current port. With an argument n, it enables the breakpoint with identifier n. Equivalent to enable_breakpoints(BID), where BID is the breakpoint identifiers for the current predicate, or the supplied argument (see Section 5.7 [Breakpoint Predicates], page 272).

. find this outputs information about where the predicate being called is defined.

**a** abort causes an abort of the current execution. All the execution states built so far are destroyed and you are put right back at the top level. (This is the same as the built-in predicate abort/0.)

**b** break calls the built-in predicate break/0, thus putting you at a recursive top level with the execution so far sitting underneath you. When you end the break (^D) you will be reprompted at the port at which you broke. The new execution is completely separate from the suspended one; the invocation numbers will start again from 1 during the break. The debugger is temporarily switched off as you call the break and will be re-switched on when you finish the break and go back to the old execution. However, any changes to the leashing or to spypoints will remain in effect.

**@** command gives you the ability to call arbitrary Prolog goals. It is effectively a one-off break (see above). The initial message ‘l :- ’ will be output on the standard error stream, and a command is then read from the standard input stream and executed as if you were at top level. If the term read is of form Pattern ^ Body, then Pattern is unified with the current goal and Body is executed. Please note:

1. If Body is compound, then it should be parenthesized.
2. If the current goal has a module qualifier, then Pattern should not include the module qualifier.

**u** unify is available at the Call port and gives you the option of providing a solution to the goal from the standard input stream rather than executing the goal. This is convenient e.g. for providing a “stub” for a predicate that has not yet been written. A prompt will be output on the standard error stream, and the solution is then read from the standard input stream and unified with the goal. If the term read in is of the form Head :- Body, then Head will be unified with the current goal, and Body will be executed in its place.

**e** raise exception is available at all ports. A prompt will be output on the standard error stream, and an exception term is then read from the standard input stream and raised in the program being debugged.

**<** This command, without arguments, resets the printdepth to 10. With an argument of n, the printdepth is set to n, treating 0 as infinity. This command works by changing the value of the debugger_print_options Prolog flag.
While at a particular port, a current subterm of the current goal is maintained. It is the current subterm that is displayed, printed, or written when prompting for a debugger command. Used in combination with the printdepth, this provides a means for navigating in the current goal for focusing on the part of interest. The current subterm is set to the current goal when arriving at a new port. This command, without arguments, resets the current subterm to the current goal. With an argument of $n (> 0)$, the current subterm is replaced by its $n$:th subterm. With an argument of 0, the current subterm is replaced by its parent term. With multiple arguments separated by whitespace, the arguments are applied from left to right.

?- help displays the table of commands given above.

The user can define new debugger commands or modify the behavior of the above ones using the user:debugger_command_hook/2 hook predicate, see Section 5.7 [Breakpoint Predicates], page 272.

5.6 Advanced Debugging — an Introduction

This section gives an overview of the advanced debugger features. These center around the notion of breakpoint. Breakpoints can be classified as either spypoints (a generalization of the plain spypoint introduced earlier) or advice points (e.g. for checking program invariants independently from tracing). The first five subsections will deal with spypoints only. Nevertheless we will use the term breakpoint, whenever a statement is made that applies to both spypoints and advice points.

Section 5.8 [Breakpoint Processing], page 275, describes the breakpoint processing mechanism in full detail. Reference style details of built-in predicates dealing with breakpoints are given in Section 5.7 [Breakpoint Predicates], page 272, and in Section 5.9 [Breakpoint Conditions], page 277.

5.6.1 Creating Breakpoints

Breakpoints can be created using the add_breakpoint/2 built-in predicate. Its first argument should contain the description of the breakpoint, the so called breakpoint specification. It will return the breakpoint identifier (BID) of the created breakpoint in its second argument. For example:

?- add_breakpoint(pred(foo/2), BID).
% Plain spypoint for user:foo/2 added, BID=1
BID = 1

Here, we have a simple breakpoint specification, prescribing that the debugger should stop at all ports of all invocations of the predicate foo/2. Thus the above goal actually creates a plain spypoint, exactly as ?- spy foo/2. does.

A slightly more complicated example follows:
This breakpoint will be activated only for those calls of `foo/2` that occur in line 123 of the Prolog program file `/myhome/bar.pl`. Because of the additional condition, this is called a conditional spypoint.

The breakpoint identifier (BID) returned by `add_breakpoint/2` is an integer, assigned in increasing order, i.e. more recent breakpoints receive higher identifier values. When looking for applicable breakpoints, the debugger tries the breakpoints in descending order of BIDs, i.e. the most recent applicable breakpoint is used. Breakpoint identifiers can be used for referring to breakpoints to be deleted, disabled or enabled (see later).

Generally, the breakpoint specification is a pair Tests-Actions. Here, the Tests part describes the conditions under which the breakpoint should be activated, while the Actions part contains instructions on what should be done at activation. The test part is built from tests, while the action part from actions and tests. Test, actions and composite constructs built from these are generally referred to as breakpoint conditions, or simply conditions.

The action part can be omitted, and then the breakpoint specification consists of tests only. For spypoints, the default action part is `[show(print),command(ask)]`. This instructs the debugger to print the goal in question and then ask the user what to do next, exactly as described in Section 5.4 [Debug Format], page 236. To illustrate other possibilities let us explain the effect of the `[show(display),command(proceed)]` action part: this will use `display/1` for presenting the goal (just as the 'd' debugger command does, see Section 5.5 [Debug Commands], page 237), and will then proceed with execution without stopping (i.e. the spypoint is unleashed).

### 5.6.2 Processing Breakpoints

We first give a somewhat simplified sketch of how the debugger treats the breakpoints. This description will be refined in the sequel.

The debugger allows us to prescribe some activities to be performed at certain points of execution, namely at the ports of procedure boxes. In principle, the debugger is entered at each port of each procedure invocation. It then considers the current breakpoints one by one, most recent first. The first breakpoint for which the evaluation of the test part succeeds is then activated, and the execution continues according to its action part. The activated breakpoint “hides” the remaining (older) ones, i.e. those are not tried here. If none of the current breakpoints is activated, then the debugger behaves according to the actual debugging mode (trace, debug or zip).

Both the test and the action part can be simple or composite. Evaluating a simple test amounts to checking whether it holds in the current state of execution, e.g. `pred(foo/2)` holds if the debugger is at a port of predicate `foo/2`.

Composite conditions can be built from simple ones by forming lists, or using the ``, `;`, `->` and `\+` operators, with the usual meaning of conjunction, disjunction, if-then-else and negation. A list of conditions is equivalent to a conjunction of the same conditions.
For example, the condition \([\text{pred}(\text{foo}/2), \neg\text{port}(\text{fail})]\) will hold for all ports of \text{foo}/2, except for the Fail port.

### 5.6.3 Breakpoint Tests

This section gives a tour of the most important simple breakpoint tests. In all examples here the action part will be empty. Note that the examples are independent, so if you want to try out these, then you should get rid of the old breakpoints (e.g. using \(?-\text{nospyall}.\)) before you enter a new one.

The \texttt{goal(...)} test is a generalization of the \texttt{pred(...)} test, as it allows us to check the arguments of the invocation. For example:

\[
\text{\texttt{?- add_breakpoint(goal(foo(1,_)), \_).}}
\]

\%
Conditional spynpoint for user:foo/2 added, BID=1

The \texttt{goal(G)} breakpoint test specifies that the breakpoint should be applied only if the current goal is an instance of \(G\), i.e. \(G\) and the current goal can be unified without substituting any variables in the latter. This unification is then carried out. The \texttt{goal(G)} condition is thus equivalent to the \texttt{subsumes(G, CurrentGoal)} test (\texttt{subsumes/2} is defined in \texttt{library(terms)}, see Section 10.46 [lib-terms], page 862).

In the above example the debugger will stop if \texttt{foo/2} is called with \(1\) as its first argument, but not if the first argument is, say, \(2\), nor if it is a variable.

You can use non-anonymous variables in the \texttt{goal} test, and then put further constraints on these variables using the \texttt{true} condition:

\[
\text{\texttt{?- add_breakpoint([goal(foo(X,_)),true(X>1)], \_).}}
\]

\%
Conditional spynpoint for user:foo/2 added, BID=1

Here the first test, \texttt{goal}, specifies that we are only interested in invocations of \texttt{foo/2}, and names the first argument of the goal as \(X\). The second, the \texttt{true/1} test, specifies a further condition stated as a Prolog goal: \(X\) is greater than \(1\) (we assume here that the argument is numeric). Thus this breakpoint will be applicable if and only if the first argument of \texttt{foo/2} is \texttt{greater} than \(1\). Generally, an arbitrary Prolog goal can be placed inside the \texttt{true} test: the test will succeed if and only if the goal completes successfully.

Any variable instantiations in the test part will be undone before executing the action part, as the evaluation of the test part is enclosed in a double negation (\(+ \neg (+ \ldots)\)). This ensures that the test part has no effect on the variables of the current goal.

Both the \texttt{pred} and the \texttt{goal} tests may include a module name. In fact, the first argument of \texttt{add_breakpoint} is module name expanded, and the (explicit or implicit) module name of this argument is then inherited by default by the \texttt{pred}, \texttt{goal}, and \texttt{true} tests. Notice the module qualification inserted in front of the breakpoint specification of the last example, as shown in the output of the \texttt{debugging/0} built-in predicate:
| ?- debugging.
(....)
Breakpoints:
  1 * user:foo/2 if user:[goal(foo(A,B)),true(A>1)]

As no explicit module qualifications were given in the tests, this breakpoint specification is transformed to the following form:

[goal(user:foo(A,B)),true(user:(A>1))]

For exported predicates, a pred or goal test will be found applicable for all invocations of the predicate, irrespective of the module the call occurs in. When you add the breakpoint you can use the defining or an importing module name, but this information is not remembered: the module name is “normalized”, i.e. it is changed to the defining module. The example below shows this: although the spypoint is placed on user:append, the message and the breakpoint list both mention lists:append.

| ?- use_module(library(lists)).
(....)
% module lists imported into user
(....)
| ?- spy user:append.
% Plain spypoint for lists:append/3 added, BID=1
| ?- debugging.
(....)
Breakpoints:
  1 + lists:append/3

Note that the debugger does not stop inside a library predicate when doing an exhaustive trace. This is because the library modules are declared hidden (see Section 4.11 [ref-mod], page 161), and no trace is produced for calls inside hidden modules that invoke predicates defined in hidden modules. However, a spypoint is always shown in the trace, even if it occurs in a hidden module:

  + 1 1 Call: append([1,2],[3,4],_531) ? RET
  + 2 2 Call: lists:append([2],[3,4],_1182) ? RET
  + 3 3 Call: lists:append([],3,4],_1670) ? RET
  + 3 3 Exit: lists:append([],3,4],[3,4]) ? RET
(....)

You can narrow a breakpoint to calls from within a particular module by using the module test, e.g.
| ?- add_breakpoint([pred(append/3),module(user)], _).
% The debugger will first zip -- showing spypoints (zip)
% Conditional spypoint for lists:append/3 added, BID=1
% zip
| ?- append([1,2], [3,4], L).
  * 1 1 Call: append([1,2],[3,4],_531) ? RET
  * 1 1 Exit: append([1,2],[3,4],[1,2,3,4]) ? RET
L = [1,2,3,4]

With this spypoint, the debugger will only stop at the invocations of append/3 from the user module.

Note that calling module information is not kept by the compiler for the built-in predicates, therefore the module test will always unify its argument with prolog in case of compiled calls to built-in predicates.

There are two further interesting breakpoint tests related to invocations: inv(Inv) and depth(Depth). These unify their arguments with the invocation number and the depth, respectively (the two numbers shown at the beginning of each trace message). Such tests are most often used in more complex breakpoints, but there may be some simple cases when they are useful.

Assume you put a plain spypoint on foo/2, and start leaping through your program. After some time, you notice some inconsistency at an Exit port, but you cannot go back to the Call port for retrying this invocation, because of side effects. So you would like to restart the whole top-level goal and get back to the Call port of the suspicious goal as fast as possible. Here is what you can do:

| ?- spy foo/2.
% Plain spypoint for user:foo/2 added, BID=1
| ?- debug, foo(23, X).
% The debugger will first leap -- showing spypoints (debug)
  + 1 1 Call: foo(23,_414) ? l
  (...) 
  + 81 17 Call: foo(7,_9151) ? l
  + 86 18 Call: foo(6,_9651) ? l
  + 86 18 Exit: foo(6,8) ? -
% Plain spypoint for user:foo/2, BID=1, removed (last)
  86 18 Exit: foo(6,8) ? *
Placing spypoint on user:foo/2 with conditions: inv(86).
% Conditional spypoint for user:foo/2 added, BID=1
  * 86 18 Exit: foo(6,8) ? a
% Execution aborted
% source_info
| ?- debug, foo(23, X).
% The debugger will first leap -- showing spypoints (debug)
  * 86 18 Call: foo(6,_2480) ? RET
When you reach the Exit port of the suspicious invocation (number 86), you remove the plain spypoint (via the \(-\) debugger command), and add a conditional one using the ‘\(\ast\)’ debugger command. This automatically includes \texttt{pred(foo/2)} among the conditions and displays the prompt ‘\texttt{Placing spypoint ... with conditions:}’, requesting further ones. You enter here the \texttt{inv} test with the invocation number in question, resulting in a breakpoint with the \([\texttt{pred(foo/2)}, \texttt{inv}(86)]\) conditions. If you restart the original top-level goal in debug mode, then the debugger immediately positions you at the invocation with the specified number.

Note that when the debugger executes a \texttt{skip} or a \texttt{zip} command, no procedure boxes are built. Consequently, the invocation and depth counters are not incremented. If \texttt{skip} and/or \texttt{zip} commands were used during the first execution, then the suspicious invocation gets an invocation number higher than 86 in the second run. Therefore it is better to supply the \texttt{inv(I)}, \texttt{true(I>=86)} condition to the ‘\(\ast\)’ debugger command, which will bring you to the first call of \texttt{foo/2} at, or after invocation number 86 (which still might not be the suspicious invocation).

In the examples, the \texttt{inv} test was used both with a numeric and a variable argument (\texttt{inv}(86) and \texttt{inv}(I)). This is possible because the debugger \textit{unifies} the given feature with the argument of the test. This holds for most tests, we will mention the exceptions.

Another similar example: if you suspect that a given predicate goes into an infinite recursion, and would like the execution to stop when entering this predicate somewhere inside the recursion, then you can do the following:

\[
\begin{verbatim}
| ?- add_breakpoint([pred(foo/2),depth(_D),true(_D>=100)], _).
% Conditional spypoint for user:foo/2 added, BID=1
% zip,source_info
| ?- debug, foo(200, X).
% The debugger will first leap -- showing spypoints (debug)
  * 496 100 Call: foo(101,_12156) ?
\end{verbatim}
\]

The above breakpoint specification will cause the debugger to stop at the first invocation of \texttt{foo/2} at depth 100 or greater. Note again that debug mode has to be entered for this to work (in zip mode no debugging information is kept, so the depth does not change).

We now continue with tests that restrict the breakpoint to an invocation at a specific place in the code.

Assume file \texttt{/home/bob/myprog.pl} contains the following Prolog program:

\[
\begin{verbatim}
% /home/bob/myprog.pl
\end{verbatim}
\]
Chapter 5: Debugging

p(X, U) :- % line 1
   q(X, Y), % line 2
   q(Y, Z), % line 3
   ( \+ q(Z, _) % line 4
     -> q(Z+1, U) % line 5
     ; q(Z+2, U) % line 6
   ). % ...

q(X, Y) :-
   X < 10, !, Y is X+1. % line 10
q(X, Y) :-
   Y is X+2. % line 12

If you are interested only in the last invocation of \( q/2 \) within \( p/2 \), then you can use the following breakpoint:

\[
| ?- \text{add_breakpoint([pred(q/2), line('home/bob/myprog.pl',6)], _).} \\
% Conditional spypoint for user:q/2 added, BID=1
\]

Generally, the test \text{line(File,Line)} holds if the current invocation was in line number Line of a file whose absolute name is File. This test (as well as the \text{line/1} and \text{file/1} tests; see below) require the presence of source information: the file in question had to be consulted or compiled with the \text{source_info} Prolog flag switched on (i.e. set to \text{on} or \text{emacs}).

If e.g. \( q/2 \) is called only from a single file, then the file name need not be mentioned and a \text{line/1} test suffices: \text{line(6)}. On the other hand, if we are interested in all invocations of a predicate within a file, then we can omit the line number and use the \text{file(File)} test.

For Prolog programs that are interpreted (consulted or asserted), further positioning information can be obtained, even in the absence of source information. The test \text{parent_pred(Pred)} unifies the module name expanded Pred with a predicate specification (of form \text{Module:PredName/Arity}) identifying the predicate in which the current invocation resides. The test \text{parent_pred(Pred,N)} will additionally unify \( N \) with the serial number of the clause containing the current goal.

For example, assuming the above \text{myprog.pl} file is consulted, the breakpoint below will cause the execution to stop when the call of \text{is/2} in the second clause of \( q/2 \) is reached:

\[
| ?- \text{add_breakpoint([pred(is/2), parent_pred(q/2,2)], _).} \\
% Conditional spypoint for prolog:is/2 added, BID=1
* Predicate prolog:is/2 compiled inline, breakable only in interpreted code \\
% zip,source_info \\
| ?- p(20, X).
in scope of a goal at line 12 in /home/bob/myprog.pl \\
* 1 1 Call: _579 is 20+2 ?
\]
Notice the warning issued by `add_breakpoint/2`: there are some built-in predicates (e.g. arithmetic, `functor/3`, `arg/3`, etc.), for which the compiler generates specific inline translation, rather than the generic predicate invocation code. Therefore compiled calls to such predicates are not visible to the debugger.

More exact positioning information can be obtained for interpreted programs by using the `parent_clause(Cl,Sel,I)` test. This unifies `Cl` with the clause containing the current invocation, while `Sel` and `I` both identify the current invocation within the body of this clause. `Sel` is unified with a subterm selector, while `I` with the serial number of the call. This test has the variants `parent_clause/[1,2]`, in which only the `Cl` argument, or the `Cl,Sel` arguments are present.

As an example, two further alternatives of putting a breakpoint on the last call of `q/2` within `myprog.pl` (line 6) are shown below, together with a listing showing the selectors and call serial numbers for the body of `p/2`:

```
| ?- add_breakpoint([pred(q/2),parent_clause((p(_,_):-_),[2,2,2])],_).
| ?- add_breakpoint([pred(q/2),parent_clause((p(_,_):-_),_,5)],_).
```

```
p(X, U) :-                       % line % call no. % subterm selector
    q(X, Y),               % 2  1 [1]
    q(Y, Z),               % 3  2 [2,1]
    ( \+ q(Z, _)             % 4  3 [2,2,1,1,1]
        -> q(Z+1, U)         % 5  4 [2,2,1,2]
    ;   q(Z+2, U)            % 6  5 [2,2,2]
 ).                                 % 7
```

Here, the first argument of the `parent_clause` test ensures that the current invocation is in (the only clause of) `p/2`. If `p/2` had more clauses, then we would have to use an additional test, say `parent_pred(user:p/2,1)`, and then the first argument of `parent_clause` could be an anonymous variable.

In the examples so far the breakpoint tests referred only to the goal in question. Therefore, the breakpoint was found applicable at all ports of the procedure box of the predicate. We can distinguish between ports using the `port` breakpoint test:

```
| ?- add_breakpoint([pred(foo/2),port(call)], _).
```

With this breakpoint, the debugger will stop at the Call port of `foo/2`, but not at other ports. Note that the `port(call)` test can be simplified to `call` — `add_breakpoint/2` will recognize this as a port name, and treat it as if it were enclosed in a `port/1` functor.

Here are two equivalent formulations for a breakpoint that will cause the debugger to stop only at the Call and Exit ports of `foo/2`:
in both cases we have to use disjunction. In the first example we have a disjunctive breakpoint condition of the two simple tests \texttt{port(call)} and \texttt{port(exit)} (with the \texttt{port} functor omitted). In the second case the disjunction is inside the Prolog test within the \texttt{true} test.

Notice that the two examples refer to the Exit port differently. When you use \texttt{port(P)}, where \texttt{P} is a variable, then, at an exit port, \texttt{P} will be unified with either \texttt{exit(nondet)} or \texttt{exit(det)}, depending on the determinacy of the exited predicate. However, for convenience, the test \texttt{port(exit)} will also succeed at Exit ports. So in the first example above, \texttt{exit} can be replaced by \texttt{exit(_)}, but the \texttt{exit(_)} in the second cannot be replaced by \texttt{exit}.

Finally, there is a subtle point to note with respect to activating the debugger at non Call ports. Let us look at the following breakpoint:

\begin{verbatim}
| ?- add_breakpoint([pred(foo/2),fail], _).
| ?- add_breakpoint([pred(foo/2),port(P),true((P=call;P=exit(_)))], _).
\end{verbatim}

The intention here is to have the debugger stop at only the Fail port of \texttt{foo/2}. This is very useful if \texttt{foo/2} is not supposed to fail, but we suspect that it does. The above breakpoint will behave as expected when the debugger is leaping, but not while zipping. This is because for the debugger to be able to stop at a non Call port, a procedure box has to be built at the Call port of the given invocation. However, no debugging information is collected in zip mode by default, i.e. procedure boxes are not built. Later we will show how to achieve the required effect, even in zip mode.

### 5.6.4 Specific and Generic Breakpoints

In all the examples so far a breakpoint was put on a specific predicate, described by a \texttt{goal} or \texttt{pred} test. Such breakpoints are called \textit{specific}, as opposed to \textit{generic} ones.

Generic breakpoints are the ones that do not specify a concrete predicate. This can happen when the breakpoint specification does not contain \texttt{goal} or \texttt{pred} tests at all, or their argument is not sufficiently instantiated. Here are some examples of generic breakpoints:

\begin{verbatim}
| ?- add_breakpoint(line('/home/bob/myprog.pl',6), _).
% Generic spypoint added, BID=1
| ?- add_breakpoint(pred(foo/_), _).
% Generic spypoint added, BID=2
| ?- add_breakpoint([goal(G),true((arg(1,G,X),X=bar))], _).
% Generic spypoint added, BID=3
\end{verbatim}

The first breakpoint will stop at all calls in line 6 of the given file, the second at all calls of a predicate \texttt{foo}, irrespective of the number of arguments, while the third one will stop at any predicate with \texttt{bar} as its first argument. However, there is an additional implicit condition: the module name expansion inserts the type-in module as the default module name in the
**goal** and **pred** conditions. Consequently, the second and third breakpoint applies only to predicates in the type-in module (user by default). If you would like the breakpoint to cover all modules, then you have to include an anonymous module prefix in the argument of the **goal** or **pred** test:

```
| ?- add_breakpoint(pred(_:foo/_), _).
% Generic spypoint added, BID=1
% zip
| ?- add_breakpoint([goal(_:G),true((arg(1,G,X),X==bar))], _).
% Generic spypoint added, BID=2
```

Generic breakpoints are very powerful, but there is a price to pay: the zip mode is slowed down considerably.

As said earlier, in principle the debugger is entered at each port of each procedure invocation. As an optimization, the debugger can request the underlying Prolog engine to run at full speed and invoke the debugger only when one of the specified predicates is called. This optimization is used in zip mode, provided there are no generic breakpoints. In the presence of generic breakpoints, however, the debugger has to be entered at each call, to check their applicability. Consequently, with generic breakpoints, zip mode execution will not give much speed-up over debug mode, although its space requirements will still be much lower.

It is therefore advisable to give preference to specific breakpoints over generic ones, whenever possible. For example, if your program includes predicates **foo/2** and **foo/3**, then it is much better to create two specific breakpoints, rather than a single generic one with conditions `[pred(foo/_),...]`.

**spy/2** is a built-in predicate that will create specific breakpoints only. Its first argument is a generalized predicate specification, much like in **spy/1**, and the second argument is a breakpoint specification. **spy/2** will expand the first argument to one or more predicate specifications, and for each of these will create a breakpoint, with a **pred** condition added to the **test** part of the supplied breakpoint specifications. For example, in the presence of predicates **foo/2** and **foo/3**

```
| ?- spy(foo/_, file(...))
```

is equivalent to:

```
| ?- add_breakpoint([pred(foo/2),file(...)], _),
    add_breakpoint([pred(foo/3),file(...)], _).
```

Note that with **spy/[1,2]** it is not possible to put a breakpoint on a (yet) undefined predicate. On the other hand, **add_breakpoint/2** is perfectly capable of creating such breakpoints, but warns about them.

### 5.6.5 Breakpoint Actions

The action part of a breakpoint specification supplies information to the debugger as to what should be done when the breakpoint is activated. This is achieved by setting the
three so called **debugger action variables**. These are listed below, together with their most important values.

- **The show variable** prescribes how the debugged goal should be displayed:
  
  - `print` write the goal according to the `debugger_print_options` Prolog flag.
  - `silent` do not display the goal.

- **The command variable** prescribes what the debugger should do:
  
  - `ask` ask the user.
  - `proceed` continue the execution without stopping, creating a procedure box for the current goal at the Call port,
  - `flit` continue the execution without stopping, without creating a procedure box for the current goal at the Call port.

- **The mode variable** prescribes in what mode the debugger should continue the execution:
  
  - `trace` creeping.
  - `debug` leaping.
  - `zip` zipping.
  - `off` without debugging.

For example, the breakpoint below specifies that whenever the Exit port of `foo/2` is reached, no trace message should be output, no interaction should take place and the debugger should be switched off.

```
| ?- add_breakpoint([pred(foo/2),port(exit)]-
                   [show(silent),command(proceed),mode(off)], _).
```

Here, the action part consists of three actions, setting the three action variables. This breakpoint specification can be simplified by omitting the wrappers around the variable values, as the sets of possible values of the variables are all disjoint. If we use `spy/2`, then the `pred` wrapper goes away, too, resulting in a much more concise, equivalent formulation of the above breakpoint:

```
| ?- spy(foo/2,exit-[silent,proceed,off]).
```

Let us now revisit the process of breakpoint selection. When the debugger arrives at a port it first initializes the action variables according to the current debugging and leashing modes, as shown below:
It then considers each breakpoint, most recent first, until it finds a breakpoint whose test part succeeds. If such a breakpoint is found, then its action part is evaluated, normally changing the action variable settings. A failure of the action part is ignored, in the sense that the breakpoint is still treated as the selected one. However, as a side effect, a procedure box will always be built in such cases. More precisely, the failure of the action part causes the `flit` command value to be changed to `proceed`, all other command values being left unchanged. This is to facilitate the creation of breakpoints that stop at non-Call ports (see below for an example).

If no applicable breakpoint is found, then the action variables remain unchanged.

The debugger then executes the actions specified by the action variables. This process, referred to as the *action execution*, means the following:

- The current debugging mode is set to the value of the `mode` action variable.
- A trace message is displayed according to the `show` variable.
- The program continues according to the `command` variable.

Specifically, if `command` is `ask`, then the user is prompted for a debugger command, which in turn is converted to new assignments to the action variables. The debugger will then repeat the action execution process, described above. For example, the ‘c’ (creep) interactive command is converted to `[silent,proceed,trace]`, the ‘d’ (display) command to `[display,ask]` (when command is `ask`, the mode is irrelevant), etc.

The default values of the action variables correspond to the standard debugger behavior described in Section 5.2 [Basic Debug], page 233. For example, when an unleashed port is reached in trace mode, a trace message is printed and the execution proceeds in trace mode, without stopping. In zip mode, no trace message is shown, and execution continues in zip mode, without building procedure boxes at Call ports.

Note that a spypoint action part that is empty ([] or not present) is actually treated as `[print,ask]`. Again, this is the standard behavior of spypoints, as described in Section 5.2 [Basic Debug], page 233.

If an action part is nonempty, but it does not set the action variables, then the only effect it will have is to hide the remaining older spypoints, as the debugger will behave in the
standard way, according to the debugging mode. Still, such breakpoints may be useful if they have side effects, for example:

```
| ?- spy(foo/2, [-parent_pred(P),
goal(G),
    true(format('~q called from:~w~n',[G,P]))]).
% The debugger will first zip -- showing spypoints (zip)
% Conditional spypoint for user:foo/2 added, BID=1
true
% zip
| ?- foo(3,X).
foo(2,701) called from:bar/3
foo(1,1108) called from:bar/3
foo(0,1109) called from:bar/3
foo(1,702) called from:bar/3
X = 2 ? ;
no
```

This spypoint produces some output at ports of `foo/2`, but otherwise will not influence the debugger. Notice that a breakpoint specification with an empty test part can be written to Actions.

Let us look at some simple examples of what other effects can be achieved by appropriate action variable settings:

```
| ?- spy(foo/2, [-print,proceed]).
```

This is an example of an unleashed spypoint: it will print a trace message passing each port of `foo/2`, but will not stop there. Note that because of the `proceed` command a procedure box will be built, even in zip mode, and so the debugger will be activated at non-Call ports of `foo/2`

The next example is a variant of the above:

```
| ?- spy(foo/2, [-print,flit]).
```

This will print a trace message at the Call port of `foo/2` and will then continue the execution in the current debugging mode, without building a procedure box for this call. This means that the debugger will not be able to notice any other ports of `foo/2`.

Now let us address the task of stopping at a specific non-Call port of a predicate. For this to work in zip mode, one has to ensure that a procedure box is built at the Call port. In the following example, the first spypoint causes a box to be built for each call of `foo/2`, while the second one makes the debugger stop when the Fail port of `foo/2` is reached.

```
| ?- spy(foo/2, call-proceed), spy(foo/2, fail).
% Conditional spypoint for user:foo/2 added, BID=1
% Conditional spypoint for user:foo/2 added, BID=2
```
You can achieve the same effect with a single spypoint, by putting the `fail` condition (which is a shortcut for `port(fail)`) in the `action` part, rather than in the `test` part.

```
| ?- spy(foo/2, -[fail,print,ask]).
```

Here, when the execution reaches the Call port of `foo/2`, the test part (which contains the `pred(foo/2)` condition only) succeeds, so the breakpoint is found applicable. However, the action part fails at the Call port. This has a side effect in zip mode, as the default `flit` command value is changed to `proceed`. In other modes the action variables are unaffected. The net result is that a procedure box is always built for `foo/2`, which means that the debugger will actually reach the Fail port of this predicate. When this happens, the action part succeeds, and executing the actions `print,ask` will cause the debugger to stop.

Note that we have to explicitly mention the `print,ask` actions here, because the action part is otherwise nonempty (contains the `fail` condition). It is only the empty or missing action part, which is replaced by the default `[print,ask]`. If you want to include a condition in the action part, then you have to explicitly mention all action variable settings you need.

To make this simpler, the debugger handles breakpoint condition macros, which expand to other conditions. For example `leash` is a macro that expands to `[print,ask]`. Consequently, the last example can be simplified to:

```
| ?- spy(foo/2, -[fail,leash]).
```

Similarly, the macro `unleash` expands to `[print,proceed]`, while `hide` to `[silent,proceed].

We now briefly describe further possible settings to the action variables.

The `mode` variable can be assigned the values `skip(Inv)` and `qskip(Inv)`, meaning skipping and quasi-skipping until a port is reached whose invocation number is less or equal to `Inv`. When the debugger arrives at this port it sets the `mode` variable to `trace`.

It may be surprising that `skip(...)` is a mode, rather than a command. This is because commands are executed and immediately forgotten, but skipping has a lasting effect: the program is to be run with no debugging until a specific point, without creating new procedure boxes, and ignoring the existing ones in the meantime.

Here is an example using the `skip` mode:

```
| ?- spy(foo/2, call-[print,proceed,inv(Inv),skip(Inv)]).
```

This breakpoint will be found applicable at Call ports of `foo/2`. It will print a trace message there and will skip over to the Exit or Fail port without stopping. Notice that the number of the current invocation is obtained in the action part, using the `inv` condition with a variable argument. A variant of this example follows:
This spypoint makes `foo/2` invisible in the output of the debugger: at all ports we silently proceed (i.e. display nothing and do not stop). Furthermore, at the Call port we perform a skip, so neither `foo/2` itself, nor any predicate called within it will be shown by the debugger.

Notice the use of the `true/0` test in the above conditional! This is a breakpoint test that always succeeds. The debugger also recognizes `false` as a test that always fails. Note that while `false` and `fail` are synonyms as built-in predicates, they are completely different as breakpoint conditions: the latter is a shortcut for `port(fail)`.

The `show` variable has four additional value patterns. Setting it to `display`, `write`, or `write_term(Options)` will result in the debugged goal `G` being shown using `display(G)`, `writeq(G)`, or `write_term(G, Options)`, respectively. The fourth pattern, `Method-Sel`, can be used for replacing the goal in the trace message by one of its subterms, the one pointed to by the selector `Sel`.

For example, the following spypoint instructs the debugger to stop at each port of `foo/2`, and to only display the first argument of `foo/2` in the trace message, instead of the complete goal.

```prolog
?- spy(foo/2, -[print-[1],ask]).
% Conditional spypoint for user:foo/2 added, BID=1
?- foo(5,X).
* 1 1 Call: ^1 5 ?
```

The `command` variable has several further value patterns. The variable can be set to `proceed(OldGoal,NewGoal)`. At a Call port this instructs the debugger to first build a procedure box for the current goal, then to unify it with `OldGoal` and finally execute `NewGoal` in its place (cf. the `’u’` (unify) interactive debugger command). At non-Call ports this command first goes back to the Call port (cf. the `’r’` (retry) command), and then does the above activities.

A variant of the `proceed/2` command is `flit(OldGoal,NewGoal)`. This has the same effect, except for not building a procedure box for `OldGoal`.

We now just briefly list further command values (for the details, see Section 5.9.9 [Action Variables], page 282). Setting `command` to `raise(E)` will raise an exception `E`, `abort` will abort the execution. The values `retry(Inv)`, `reexit(Inv)`, `redo(Inv)`, `fail(Inv)` will cause the debugger to go back to an earlier Call, Exit, Redo, or Fail port with invocation number `Inv` (cf. the `’j’` (jump) interactive debugger command).

Sometimes it may be useful to access the value of an action variable. This can be done with the `get` condition: e.g. `get(mode(M))` will unify `M` with the current execution mode. The
get(...) wrapper can be omitted in the test part, but not in the action part (since there a mode(M) action will set, rather than read, the mode action variable). For example:

```
| ?- spy(foo/2, mode(trace)-show(print-[1])).
```

This spypoint will be found applicable only in trace mode (and will cause the first argument of foo/2 to appear in the trace message). (The mode and show wrappers can also be omitted in the above example, they are used only to help with interpreting the breakpoint specification.)

### 5.6.6 Advice points

As mentioned earlier, there are two kinds of breakpoints: spypoints and advice points. The main purpose of spypoints is to support interactive debugging. In contrast with this, advice points can help you to perform non-interactive debugging activities. For example, the following advice point will check a program invariant: whether the condition Y-X<3 always holds at exit from foo(X,Y).

```
| ?- add_breakpoint([pred(foo/2),advice]-[exit,goal(foo(X,Y)),\+true(Y-X<3),trace], _).
```

The test part of the above breakpoint contains a pred test, and the advice condition, making it an advice point. (You can also include the debugger condition in spypoint specs, although this is the default interpretation.)

The action part starts with the exit port condition. Because of this the rest of the action part is evaluated only at Exit ports. By placing the port condition in the action part, we ensure the creation of a procedure box at the Call port, as explained earlier.

Next, we get hold of the goal arguments using the goal condition, and use the \+true(Y-X<3) test to check if the invariant is violated. If this happens, then the last condition sets the mode action variable to trace, switching on the interactive debugger.

Following the add_breakpoint/2 call the above example shows two top-level calls to foo/2. The invariant holds within the first goal, but is violated within the second. Notice that the advice mechanism works with the interactive debugger switched off.

You can ask the question, why do we need advice points? The same task could be implemented using a spypoint. For example:
The main reason to have a separate advice mechanism is to be able to perform checks independently of the interactive debugging. With the second solution, if you happen to start some interactive debugging, then you cannot be sure that the invariant is always checked. For example, no spypoints will be activated during a skip. In contrast with this, the advice mechanism is watching the program execution all the time, independently of the debugging mode.

Advice points are handled in very much the same way as spypoints are. When arriving at a port, advice point selection takes place first, followed by spypoint selection. This can be viewed as the debugger making two passes over the current breakpoints, considering advice points only in the first pass, and spypoints only in the second.

In both passes the debugger tries to find a breakpoint that can be activated, checking the test and action parts, as described earlier. However, there are some differences between the two passes:

- Advice processing is performed if there are any (non-disabled) advice points. Spypoint processing is only done if the debugger is switched on, and is not doing a skip.
- For advice points, the action variables are initialized as follows: `mode` is set to current debugging mode, `command = proceed`, `show = silent`. Note that this is done independently of the debugging mode (in contrast with the spypoint search initialization).
- The default action part for advice points is `[]`. This means that if no action part is given, then the only effect of the advice point will be to build a procedure box (because of the `command = proceed` initialization).
- If no advice point was found applicable, then `command` is set to `flit`.

Having performed advice processing, the debugger inspects the `command` variable. The command values different from `proceed` and `flit` are called `divertive`, as they alter the normal flow of control (e.g. `proceed(., ., ., .)`, or involve user interaction (`ask`). If the `command` value is divertive, then the prescribed action is performed immediately, without executing the spypoint selection process. Otherwise, if `command = proceed`, then it is noted that the advice part requests the building of a procedure box. Next, the second, spypoint processing pass is carried out, and possible user interaction takes place, as described earlier. A procedure box is built if either the advice point or the spypoint search requests this.
Let us conclude this section by another example, a generic advice point for collecting branch coverage information:

```
| ?- add_breakpoint(
  (advice,call) -
    ( line(F,L) -> true(assert(line_reached(F,L))), flit
      ; flit
    ), _).
% Generic advice point added, BID=1
% advice,source_info
| ?- foo(4,X).
X = 3 ? ;
no
% advice,source_info
| ?- setof(X, line_reached(F,X), S).
F = '/home/bob/myprog.pl',
S = [31,33,34,35,36]
```

This advice point will be applicable at every Call port. It will then assert a fact with the file name and the line number if source information is available. Finally, it will set the command variable to flit on both branches of execution. This is to communicate the fact that the advice point does not request the building of a procedure box.

It is important to note that this recording of the line numbers reached is performed independently of the interactive debugging.

In this example we used the ','/2 operator, rather than list notation, for describing the conjunction of conditions, as this seems to better fit the if-then-else expression used in the action part. We could have still used lists in the tests part, and in the “then” part of the actions. Note that if we omit the “else” branch, then the action part will fail if no source information is available for the given call. This will cause a procedure box to be built, which is an unnecessary overhead. An alternative solution, using the line/2 test twice, is the following:

```
| ?- add_breakpoint([advice,call,line(_,_)]-
  [line(F,L),true(assert(line_reached(F,L))),flit], _).
```

Further examples of advice points are available in library(debugger_examples).

### 5.6.7 Built-in Predicates for Breakpoint Handling

This section introduces built-in predicates for evaluating breakpoint conditions, and for retrieving, deleting, disabling and enabling breakpoints.

The breakpoint specification of the last advice point example was quite complex. And, to be practical, it should be improved to assert only line numbers not recorded so far. For this you will write a Prolog predicate for the conditional assertion of file/line information, assert_line_reached(File,Line), and use it instead of the assert(line_reached(F,L)) condition.
Because of the complexity of the breakpoint specification, it looks like a good idea to move the if-then-else condition into Prolog code. This requires that we test the line(F,L) condition from Prolog. The built-in predicate execution_state/1 serves for this purpose. It takes a simple or a composite breakpoint condition as its argument and evaluates it, as if in the test part of a breakpoint specification. The predicate will succeed if and only if the breakpoint condition evaluates successfully. Thus execution_state/1 allows you to access debugging information from within Prolog code. For example, you can write a Prolog predicate, assert_line_reached/0, which queries the debugger for the current line information and then processes the line number:

```
assert_line_reached :-
    ( execution_state(line(F,L)) -> assert_line_reached(F,L).
    ;    true
).
```

| ?- add_breakpoint([advice,call]-
| [true(assert_line_reached),flit], _).

Arbitrary tests can be used in execution_state/1, if it is called from within a true condition. It can also be called from outside the debugger, but then only a subset of conditions is available. Furthermore, the built-in predicate execution_state/2 allows accessing information from past debugger states (see Section 5.6.8 [Accessing Past Debugger States], page 262). See Section 11.3.74 [mpg-ref-execution_state], page 1018.

The built-in predicates remove.breakpoints(BIDs), disable.breakpoints(BIDs) and enable.breakpoints(BIDs) serve for removing, disabling and enabling the given breakpoints. Here BIDs can be a single breakpoint identifier, a list of these, or one of the atoms all, advice, debugger.

We now show an application of remove.breakpoints/1 for implementing one-off breakpoints, i.e. breakpoints that are removed when first activated.

For this we need to get hold of the currently selected breakpoint identifier. The bid(BID) condition serves for this purpose: it unifies its argument with the identifier of the breakpoint being processed. The following is an example of a one-off breakpoint.

```
| ?- spy(foo/2, -[bid(BID),true(remove.breakpoints(BID)),leash]).
% Conditional spypoint for user:foo/2 added, BID=1
% zip
| ?- foo(2, X).
% Conditional spypoint for user:foo/2, BID=1, removed (last)
  1 1 Call: foo(2,_402) ? z
X = 1
```

The action part of the above breakpoint calls the bid test to obtain the breakpoint identifier. It then uses this number as the argument to the built-in predicate remove.breakpoints/1, which removes the activated breakpoint. See Section 11.3.190 [mpg-ref-remove.breakpoints], page 1170.
The built-in predicate `current_breakpoint(Spec, BID, Status, Kind, Type)` enumerates all breakpoints present in the debugger. For example, if we call `current_breakpoint/5` before the invocation of `foo/2` in the last example, then we get this:

```prolog
| ?- current_breakpoint(Spec, BID, Status, Kind, Type).
  Spec = [pred(user:foo/2)]-
    [bid(_A),true(remove_breakpoints(_A)),leash],
  BID = 1,
  Status = on,
  Kind = conditional(user:foo/2),
  Type = debugger
```

Here `Spec` is the breakpoint specification of the breakpoint with identifier `BID`. `Status` is `on` for enabled breakpoints and `off` for disabled ones. `Kind` is one of `plain(MFunc)`, `conditional(MFunc)` or `generic`, where `MFunc` is the module qualified functor of the specific breakpoint. Finally `Type` is the breakpoint type: `debugger` or `advice`.

The `Spec` returned by `current_breakpoint/5` is exactly the same as the one given in `add_breakpoint/2`. If the breakpoint was created by `spy/2`, then the test part is extended by a `pred` condition, as exemplified above. Earlier we described some preprocessing steps that the specification goes through, such as moving the module qualification of the specification to certain conditions. These transformations are performed on the copy of the breakpoint used for testing. Independently of this, the debugger also stores the original breakpoint, which is returned by `current_breakpoint/5`. See Section 11.3.48 [mpg-ref-current_breakpoint], page 989.

### 5.6.8 Accessing Past Debugger States

In this section we introduce the built-in predicates for accessing past debugger states, and the breakpoint conditions related to these.

The debugger collects control flow information about the goals being executed, more precisely about those goals, for which a procedure box is built. This collection of information, the backtrace, includes the invocations that were called but not exited yet, as well as those that exited nondeterminately. For each invocation, the main data items present in the backtrace are the following: the goal, the module, the invocation number, the depth and the source information, if any.

Furthermore, as you can enter a new break level from within the debugger, there can be multiple backtraces, one for each active break level.

You can access all the information collected by the debugger using the built-in predicate `execution_state(Focus, Tests)`. Here `Focus` is a ground term specifying which break level and which invocation to access. It can be one of the following:

- `break_level(BL)` selects the current invocation within the break level `BL`.
- `inv(Inv)` selects the invocation number `Inv` within the current break level.
- A list containing the above two elements, selects the invocation with number `Inv` within break level `BL`. 

Note that the top level counts as break level 0, while the invocations are numbered from 1 upwards.

The second argument of `execution_state/2`, `Tests`, is a simple or composite breakpoint condition. Most simple tests can appear inside `Tests`, with the exception of the `port`, `bid`, `advice`, `debugger`, and `get` tests. These tests will be interpreted in the context of the specified past debugger state. Specifically, if a `true/1` condition is used, then any `execution_state/1` queries appearing in it will be evaluated in the past context.

To illustrate the use of `execution_state/2`, we now define a predicate `last_call_arg(ArgNo, Arg)`, which is to be called from within a break, and which will look at the last debugged goal of the previous break level, and return in `Arg` the `ArgNo`th argument of this goal.

```
last_call_arg(ArgNo, Arg) :-
    execution_state(break_level(BL1)),
    BL is BL1-1,
    execution_state(break_level(BL), goal(Goal)),
    arg(ArgNo, Goal, Arg).
```

We see two occurrences of the term `break_level(...)` in the above example. Although these look very similar, they have different roles. The first one, in `execution_state/1`, is a breakpoint test, which unifies the current break level with its argument. Here it is used to obtain the current break level and store it in `BL1`. The second use of `break_level(...)`, in the first argument of `execution_state/2`, is a focus condition, whose argument has to be instantiated, and which prescribes the break level to focus on. Here we use it to obtain the goal of the current invocation of the previous break level.

Note that the goal retrieved from the backtrace is always in its latest instantiation state. For example, it is not possible to get hold of the goal instantiation at the Call port, if the invocation in question is at the Exit port.

Here is an example run, showing how `last_call_arg/2` can be used:

```
5 2 Call: _937 is 13+8 ? b
  % Break level 1
  % 1
  | ?- last_call_arg(2, A).
  A = 13+8
```

There are some further breakpoint tests that are primarily used in looking at past execution states.

The test `max_inv(MaxInv)` returns the maximal invocation number within the current (or selected) break level. The test `exited(Boolean)` unifies `Boolean` with `true` if the invocation has exited, and with `false` otherwise.

The following example predicate lists those goals in the backtrace, together with their invocation numbers, that have exited. These are the invocations that are listed by the `t`
interactive debugger command (print backtrace), but not by the g command (print ancestor goals). Note that the predicate \texttt{between}(N, M, I) enumerates all integers such that $N \leq I \leq M$.

\begin{verbatim}
exited_goals :-
  execution_state(max_inv(Max)),
  between(1, Max, Inv),
  execution_state(inv(Inv), [exited(true),goal(G)]),
  format("-t\"d-6| ~p\n", [Inv,G]),
  fail.
exited_goals.
(....)
\end{verbatim}

Note that similar output can be obtained by entering a new break level and calling \texttt{exited_goals} from within an execution\_state/2:

\begin{verbatim}
% 1
| ?- execution_state(break_level(0), true(exited_goals)).
\end{verbatim}

The remaining two breakpoint tests allow you to find parent and ancestor invocations in the backtrace. The parent\_inv(Inv) test unifies Inv with the invocation number of the youngest ancestor present in the backtrace, called debugger-parent for short. The test ancestor(AncGoal, Inv) looks for the youngest ancestor in the backtrace that is an instance of AncGoal. It then unifies the ancestor goal with AncGoal and its invocation number with Inv.

Assume you would like to stop at all invocations of \texttt{foo/2} that are somewhere within \texttt{bar/1}, possibly deeply nested. The following two breakpoints achieve this effect:

\begin{verbatim}
| ?- spy(bar/1, advice), spy(foo/2, ancestor(bar(\_),\_)).
% Plain advice point for user:bar/1 added, BID=3
% Conditional spypoint for user:foo/2 added, BID=4
\end{verbatim}
We added an advice point for \texttt{bar/1} to ensure that all calls to it will have procedure boxes built, and so become part of the backtrace. Advice points are a better choice than spypoints for this purpose, as with \texttt{?- spy(bar/1, -proceed)} the debugger will not stop at the call port of \texttt{bar/1} in trace mode. Note that it is perfectly all right to create an advice point using \texttt{spy/2}, although this is a bit of terminological inconsistency.

See Section 11.3.74 [mpg-ref-execution_state], page 1018. Further examples of accessing past debugger states can be found in \texttt{library(debugger_examples)}.

### 5.6.9 Storing User Information in the Backtrace

The debugger allows the user to store some private information in the backtrace. It allocates a Prolog variable in each break level and in each invocation. The breakpoint test \texttt{private(Priv)} unifies \texttt{Priv} with the private information associated with the break level, while the test \texttt{goal_private(GPriv)} unifies \texttt{GPriv} with the Prolog variable stored in the invocation.

Both variables are initially unbound, and behave as if they were passed around the program being debugged in additional arguments. This implies that any variable assignments done within these variables are undone on backtracking.

In practice, the \texttt{private} condition gives you access to a Prolog variable shared by all invocations of a break level. This makes it possible to remember a term and look at it later, in a possibly more instantiated form, as shown by the following example.

```prolog
memory(Term) :-
    execution_state(private(P)),
    memberchk(myterm(Term), P).
```

| ?- trace, append([1,2,3,4], [5,6], L). |
| 1 1 Call: append([1,2,3,4],[5,6],_514) ? @ |
| 1 1 Call: append([1,2,3,4],[5,6],_514) ? c |
| 2 2 Call: append([2,3,4],[5,6],_2064) ? c |
| 3 3 Call: append([3,4],[5,6],_2422) ? c |
| 4 4 Call: append([4],[5,6],_2780) ? @ |
| :- memory(L), write(L), nl. |
| [1,2,3|_2780] |
| 4 4 Call: append([4],[5,6],_2780) ? |
```

The predicate \texttt{memory/1} receives the term to be remembered in its argument. It gets hold of the private field associated with the break level in variable \texttt{P}, and calls \texttt{memberchk/2} (see Section 10.24 [lib-lists], page 614), with the term to be remembered, wrapped in \texttt{myterm}, as the list element, and the private field, as the list. Thus the latter, initially unbound variable, is used as an open-ended list. For example, when \texttt{memory/1} is called for the first time, the private field gets instantiated to \texttt{[myterm(Term)]}. If later you call \texttt{memory/1} with an uninstantiated argument, then it will retrieve the term remembered earlier and unify it with the argument.
The above trace excerpt shows how this utility predicate can be used to remember an interesting Prolog term. Within invocation number 1 we call `memory/1` with the third, output argument of `append/3`, using the `@` command (see Section 5.5 [Debug Commands], page 237). A few tracing steps later, we retrieve the term remembered and print it, showing its current instantiation. Being able to access the instantiation status of some terms of interest can be very useful in debugging. In `library(debugger_examples)` we describe new debugger commands for naming Prolog variables and providing name-based access to these variables, based on the above technique.

We could have avoided the use of `memberchk/2` in the example by simply storing the term to be remembered in the private field itself (`memory(Term) :- execution_state(private(Term)).`). But this would have made the private field unusable for other purposes. For example, the finite domain constraint debugger (see Section 10.15 [lib-fdbg], page 532) would stop working, as it relies on the private fields.

There is only a single private variable of both kinds within the given scope. Therefore the convention of using an open ended list for storing information in private fields, as shown in the above example, is very much recommended. The different users of the private field are distinguished by the wrapper they use (e.g. `myterm/1` above, `fdbg/1` for the constraint debugger, etc.). Future releases may enforce this convention by providing appropriate breakpoint tests.

We now present an example of using the goal private field. Earlier we have shown a spypoint definition that made a predicate invisible in the sense that its ports are silently passed through and it is automatically skipped over. However, with that earlier solution, execution always continues in trace mode after skipping. We now improve the spypoint definition: the mode in which the Call port was reached is remembered in the goal private field, and the mode action variable is reset to this value at the Exit port.

```prolog
mode_memory(Mode) :-
    execution_state(goal_private(GP)),
    memberchk(mymode(Mode), GP).

| ?- spy(foo/2, -[silent,proceed,
true(mode_memory(MM)),
(   call -> get(mode(MM)), inv(Inv), skip(Inv)
;   exit -> mode(MM)
;   true
)].
```

Here, we first define an auxiliary predicate `mode_memory/1`, which uses the open list convention for storing information in the goal private field, applying the `mymode/1` wrapper. We then create a spypoint for `foo/2`, whose action part first sets the `print` and `command` action variables. Next, the `mode_memory/1` predicate is called, unifying the mode memory with the `MM` variable. We then branch in the action part: at Call ports the uninstantiated `MM` is unified with the current mode, and a `skip` command is issued. At Exit ports `MM` holds the mode saved at the Call port, so the `mode(MM)` action re-activates this mode. At all other ports we just silently proceed without changing the debugger mode.
5.6.10 Hooks Related to Breakpoints

There are two hooks related to breakpoints.

The hook `breakpoint_expansion(Macro, Body)` makes it possible for the user to extend the set of allowed conditions. This hook is called, at breakpoint addition time, with each simple test or action within the breakpoint specification, as the `Macro` argument. If the hook succeeds, then the term returned in the `Body` argument is substituted for the original test or action. Note that `Body` cannot span both the test and the action part, i.e. it cannot contain the `- /2` operator. The whole `Body` will be interpreted either as a test or as an action, depending on the context of the original condition. See Section 11.3.28 [mpg-ref-breakpoint_expansion], page 966.

We now give a few examples for breakpoint macros. The last example defines a condition making a predicate invisible, a reformulation of the last example of the previous subsection.

```prolog
?- multifile user:breakpoint_expansion/2.
user:breakpoint_expansion(
    skip, [inv(I), skip(I)]).

user:breakpoint_expansion(
    gpriv(Value),
    [goal_private(GP), true(memberchk(Value, GP))]).

user:breakpoint_expansion(
    invisible,
    [silent, proceed,
     ( call -> get(mode(M)), gpriv(mymode(M)), skip
     ;   exit -> gpriv(mymode(MM)), mode(MM)
     ;   true
     )]).

| ?- spy(foo/2, -invisible).
```

We first define the `skip` macro, instructing the debugger to skip the current invocation. This macro is only meaningful in the action part.

The second clause defines the `gpriv/2` macro, a generalization of the earlier `mode_memory/1` predicate. For example, `gpriv(mymode(M))` expands to `goal_private(GP), true(memberchk(mymode(M), GP)).` This embodies the convention of using open-ended lists for the goal private field.

Finally, the last clause implements the action macro `invisible/0`, which makes the predicate in question disappear from the trace. The last line shows how this macro can be used to make `foo/2` invisible.

Below is an alternative implementation of the same macro. Here we use a Prolog predicate that returns the list of action variable settings to be applied at the given port. Notice that a variable can be used as a breakpoint condition, as long as this variable gets instantiated.
to a (simple or composite) breakpoint condition by the time it is reached in the process of breakpoint evaluation.

\[
\text{user:breakpoint_expansion(invisible,} \\
\quad \text{[true(invisible(Settings)),Settings]).}
\]

\[
\text{invisible([proceed,silent,NewMode]) :-} \\
\quad \text{execution_state([mode(M),port(P),inv(Inv),goal_private(GP)]),} \\
\quad \text{memberchk(mymode(MM), GP),} \\
\quad \begin{cases} 
\text{P == call -> MM = M, NewMode = skip(Inv)} \\
\text{P = exit(_) -> NewMode = MM} \\
\text{NewMode = M}
\end{cases}
\]

The second hook related to breakpoints is \text{debugger_command_hook(DCommand, Actions)}. This hook serves for customizing the behavior of the interactive debugger, i.e. for introducing new interactive debugger commands. The hook is called for each debugger command read in by the debugger. \text{DCommand} contains the abstract format of the debugger command read in, as returned by the query facility (see Section 4.16.3 [Query Processing], page 216). If the hook succeeds, then it should return in \text{Actions} an action part to be evaluated as the result of the command.

If you want to redefine an existing debugger command, then you should study \text{library('SU_messages')} to learn the abstract format of this command, as returned by the query facility. If you want to add a new command, then it suffices to know that unrecognized debugger commands are returned as \text{unknown(Line,Warning)}. Here, \text{Line} is the code list typed in, with any leading whitespace removed, and \text{Warning} is a warning message.

The following example defines the ‘S’ interactive debugger command to behave as skip at Call and Redo ports, and as creep otherwise:

\[
\text{:- multifile user:debugger_command_hook/2.} \\
\text{user:debugger_command_hook(unknown([0'S|\_],_), Actions) :-} \\
\quad \text{execution_state([port(P),inv(I)]),} \\
\quad \text{Actions = [Mode,proceed,silent],} \\
\quad \begin{cases} 
\text{P = call -> Mode = skip(I)} \\
\text{P = redo -> Mode = skip(I)} \\
\text{Mode = trace}
\end{cases}
\]

Note that the \text{silent} action is needed above; otherwise, the trace message will be printed a second time, before continuing the execution.

See Section 11.3.61 [mpg-ref-debugger_command_hook], page 1004. \text{library(debugger_examples)} contains some of the above hooks, as well as several others.

\textbf{5.6.11 Programming Breakpoints}

We will show two examples using the advanced features of the debugger.
The first example defines a `hide_exit(Pred)` predicate, which will hide the Exit port for `Pred` (i.e. it will silently proceed), provided the current goal was already ground at the Call port, and nothing was traced inside the given invocation. The `hide_exit(Pred)` goal creates two spypoints for predicate `Pred`:

```prolog
:- meta_predicate hide_exit(:).
hide_exit(Pred) :-
    add_breakpoint([pred(Pred),call]-true(save_groundness), _),
    add_breakpoint([pred(Pred),exit,true(hide_exit)]-hide, _).
```

The first spypoint is applicable at the Call port, and it calls `save_groundness` to check if the given invocation was ground, and if so, then it stores a term `hide_exit(ground)` in the `goal_private` attribute of the invocation.

```prolog
save_groundness :-
    execution_state([goal(_:G),goal_private(Priv)]),
    ground(G), !, memberchk(hide_exit(ground), Priv).
save_groundness.
```

The second spypoint created by `hide_exit/1` is applicable at the Exit port and it checks whether the `hide_exit/0` condition is true. If so, then it issues a `hide` action, which is a breakpoint macro expanding to `[silent,proceed]`.

```prolog
hide_exit :-
    execution_state([inv(I),max_inv(I),goal_private(Priv)]),
    memberchk(hide_exit(Ground), Priv), Ground == ground.
```

Here, `hide_exit` encapsulates the tests that the invocation number be the same as the last invocation number used (`max_inv`), and that the `goal_private` attribute of the invocation be identical to `ground`. The first test ensures that nothing was traced inside the current invocation.

If we load the above code, as well as the small example below, then the following interaction, discussed below, can take place. Note that the `hide_exit` predicate is called with the `_:-_` argument, resulting in generic spypoints being created.
?- consult(user).
? cnt(0) :- !.
? cnt(N) :-
    N > 0, N1 is N-1, cnt(N1).
?-
% consulted user in module user, 0 msec 424 bytes

?- hide_exit(_:_), trace, cnt(1).
% The debugger will first zip -- showing spypoints (zip)
% Generic spypoint added, BID=1
% Generic spypoint added, BID=2
% The debugger will first creep -- showing everything (trace)
#  1  1 Call: cnt(1) ? c
#  2  2 Call: 1>0 ? c
#  3  2 Call: _2019 is 1-1 ? c
  3  2 Exit: 0 is 1-1 ? c
#  4  2 Call: cnt(0) ? c
  1  1 Exit: cnt(1) ? c
%
% trace
?-

Invocation 1 is ground, its Exit port is not hidden, because further goals were traced inside it. On the other hand, Exit ports of ground invocations 2 and 4 are hidden.

Our second example defines a predicate call_backtrace(Goal, BTrace), which will execute Goal and build a backtrace showing the successful invocations executed during the solution of Goal.

The advantages of such a special backtrace over the one incorporated in the debugger are the following:

- it has much lower space consumption;
- the user can control what is put on and removed from the backtrace (e.g. in this example all goals are kept, even the ones that exited determinately);
- the interactive debugger can be switched on and off without affecting the “private” backtrace being built.

The call_backtrace/2 predicate is based on the advice facility. It uses the variable accessible via the private(_) condition to store a mutable (see Section 4.8.9 [ref-lte-mut], page 131) holding the backtrace. Outside the call_backtrace predicate the mutable will have the value off.

The example is a module file, so that internal invocations can be identified by the module name. We load the lists library, because memberchk/2 will be used in the handling of the private field.
:- module(backtrace, [call_backtrace/2]).
:- use_module(library(lists)).

:- meta_predicate call_backtrace(0, ?).
call_backtrace(Goal, BTrace) :-
    Spec = [advice,call]
        -[true((goal(M:G),store_goal(M,G))),flit],
        ( current_breakpoint(Spec, _, on, _, _) -> B = [];
            add_breakpoint(Spec, B)
        ),
        call_cleanup(call_backtrace1(Goal, BTrace),
            remove_breakpoints(B)).

call_backtrace(Goal, BTrace) is a meta-predicate, which first sets up an appropriate advice point for building the backtrace. The advice point will be activated at each Call port and will call the store_goal/2 predicate with arguments containing the module and the goal in question. Note that the advice point will not build a procedure box (cf. the flit command in the action part).

The advice point will be added just once: any further (recursive) calls to call_backtrace/2 will notice the existence of the breakpoint and will skip the add_breakpoint/2 call.

Having ensured the appropriate advice point exists, call_backtrace/2 calls call_backtrace1/2 with a cleanup operation that removes the breakpoint added, if any.

:- meta_predicate call_backtrace1(0, ?).
call_backtrace1(Goal, BTrace) :-
    execution_state(private(Priv)),
    memberchk(backtrace_mutable(Mut), Priv),
    ( mutable(Mut) -> get_mutable(Old, Mut),
        update_mutable([], Mut)
    ;
        create_mutable([], Mut), Old = off
    ),
    call(Goal),
    get_mutable(BTrace, Mut), update_mutable(Old, Mut).

The predicate call_backtrace1/2 retrieves the private field of the execution state and uses it to store a mutable, wrapped in backtrace_mutable. When first called within a top level, the mutable is created with the value []. In later calls, the mutable is re-initialized to []. Having set up the mutable, Goal is called. In the course of the execution of the Goal, the debugger will accumulate the backtrace in the mutable. Finally, the mutable is read, its value is returned in BTrace, and it is restored to its old value (or off).
store_goal(M, G) :-
    M \== backtrace,
    G \= call(_),
    execution_state(private(Priv)),
    memberchk(backtrace_mutable(Mut), Priv),
    mutable(Mut),
    get_mutable(BTrace, Mut),
    BTrace \== off, !, 
    update_mutable([M:G|BTrace], Mut).
store_goal(_, _).

store_goal/2 is the predicate called by the advice point, with the module and the goal as arguments. We first ensure that calls from within the backtrace module and those of call/1 get ignored. Next, the module qualified goal term is prepended to the mutable value retrieved from the private field, provided the mutable exists and its value is not off.

Below is an example run, using a small program:

| ?- consult(user). |
| cnt(N):- N =< 0, !. |
| cnt(N) :- |
| N > 0, N1 is N-1, cnt(N1). |
| ^D |
% consulted user in module user, 0 msec 424 bytes |

| ?- call_backtrace(cnt(1), B). |
% Generic advice point added, BID=1 |
% Generic advice point, BID=1, removed (last) |
| B = [user:(0=<0),user:cnt(0),user:(0 is 1-1),user:(1>0),user:cnt(1)] |
| ?- |

Note that the backtrace produced by call_backtrace/2 can not contain any information regarding failed branches. For example, the very first invocation within the above execution, 1 =< 0, is first put on the backtrace at its Call port, but this is immediately undone because the goal fails. If you would like to build a backtrace that preserves failed branches, then you have to use side effects, e.g. dynamic predicates.

Further examples of complex breakpoint handling are contained in library(debugger_examples).

This concludes the tutorial introduction of the advanced debugger features.

### 5.7 Breakpoint Handling Predicates

This section describes the advanced built-in predicates for creating and removing breakpoints.
add_breakpoint(:Spec, ?BID)  
   Adds a breakpoint with a spec Spec; the breakpoint identifier assigned is unified with BID. Spec is one of the following:

Tests-Actions
Tests      standing for Tests-[]
-Actions   standing for []-Actions

Here, both Tests and Actions are either a simple Condition, see Section 5.9 [Breakpoint Conditions], page 277, or a composite Condition. Conditions can be composed by forming lists, or by using the ‘,’ ‘;’, ‘->’, and ‘\+’ operators, with the usual meaning of conjunction, disjunction, if-then-else, and negation, respectively. A list of conditions is equivalent to a conjunction of the same conditions ([A\|B] is treated as (A,B)).

The add_breakpoint/2 predicate performs some transformations and checks before adding the breakpoint. All condition macros invoked are expanded into their bodies, and this process is repeated for the newly introduced bodies. The goal and pred conditions are then extracted from the outermost conjunctions of the test part and moved to the beginning of the conjunction. If these are inconsistent, then a consistency error is signaled. Module name expansion is performed for certain tests, as described below.

Both the original and the transformed breakpoint specification is recorded by the debugger. The original is returned in current_breakpoint/5, while the transformed specification is used in determining the applicability of breakpoints.

There can only be a single plain spypoint for each predicate. If a plain spypoint is added, and there is already a plain spypoint for the given predicate, then:
   a. the old spypoint is deleted and a new added as the most recent breakpoint, if this change affects the breakpoint selection mechanism.
   b. otherwise, the old spypoint is kept and enabled if needed.

See Section 11.3.5 [mpg-ref-add_breakpoint], page 935.

spy(:PredSpec, :Spec)  
   Adds a conditional spypoint with a breakpoint specification formed by adding pred(Pred) to the test part of Spec, for each predicate Pred designated by the generalized predicate specification PredSpec. See Section 11.3.216 [mpg-ref-spy], page 1203.

current_breakpoint(:Spec, ?BID, ?Status, ?Kind, ?Type)  
   There is a breakpoint with breakpoint specification Spec; identifier BID, status Status, kind Kind, and type Type. Status is one of on or off, referring to enabled and disabled breakpoints. Kind is one of plain(MFunc), conditional(MFunc) or generic, where MFunc is the module qualified functor of the specific breakpoint. Type is the breakpoint type: debugger or advice.

current_breakpoint/5 enumerates all breakpoints on backtracking.

The Spec as returned by current_breakpoint/5 is exactly the same as supplied at the creation of the breakpoint. See Section 11.3.48 [mpg-ref-current_breakpoint], page 989.
remove_breakpoints(+BIDs)  
disable_breakpoints(+BIDs)  
enable_breakpoints(+BIDs)  

Removes, disables or enables the breakpoints with identifiers specified by \textit{BID}s. \textit{BID}s can be a number, a list of numbers or one of the atoms: \texttt{all}, \texttt{debugger}, \texttt{advice}. The atoms specify all breakpoints, debugger type breakpoints and advice type breakpoints, respectively.

\begin{verbatim}
execution_state(:Tests)  
Tests are satisfied in the current state of the execution. Arbitrary tests can be used in this predicate, if it is called from inside the debugger, i.e. from within a \texttt{true} condition. Otherwise only those tests can be used, which query the data stored in the backtrace. An exception is raised if the latter condition is violated, i.e. a non-backtraced test (see Section 5.9 [Breakpoint Conditions], page 277) occurs in a call of \texttt{execution_state/1} from outside the debugger.
\end{verbatim}

\begin{verbatim}
execution_state(+FocusConditions, :Tests)  
Tests are satisfied in the state of the execution pointed to by \textit{FocusConditions} (see Section 5.9.7 [Past States], page 282). An exception is raised if there is a non-backtraced test among \textit{Tests}.
\end{verbatim}

Note that the predicate arguments holding a breakpoint specification (\textit{Spec} or \textit{Tests} above) are subject to module name expansion. The first argument within simple tests \texttt{goal(_)}, \texttt{pred(_)}, \texttt{parent_pred(_)}, \texttt{parent_pred(_,_)}, \texttt{ancestor(_,_)}, and \texttt{true(_)} will inherit the module name from the (module name expanded) breakpoint specification/tests predicate argument, if there is no explicit module qualification within the simple test. Within the \texttt{proceed(Old,New)} and \texttt{flit(Old,New)} command value settings, \texttt{Old} will get the module name from the \texttt{goal} or \texttt{pred} condition by default, while \texttt{New} from the whole breakpoint specification argument. See Section 11.3.74 [mpg-ref-execution_state], page 1018.

The following hook predicate can be used to customize the behavior of the interactive debugger.

\begin{verbatim}
debugger_command_hook(+DCommand, ?Actions)  
user:debugger_command_hook(+DCommand, ?Actions)  
This predicate is called for each debugger command that SICStus Prolog reads. The first argument is the abstract format of the debugger command \texttt{DCommand}, as returned by the query facility (see Section 4.16.3 [Query Processing], page 216). If it succeeds, then \texttt{Actions} is taken as the list of actions (see Section 5.9.6 [Action Conditions], page 281) to be done for the given debugger command. If it fails, then the debugger command is interpreted in the standard way.

Note that if a line typed in response to the debugger prompt cannot be parsed as a debugger command, then \texttt{debugger_command_hook/2} is called with the term \texttt{unknown(Line,Warning)}. Here, \texttt{Line} is the code list typed in, with any leading whitespace removed, and \texttt{Warning} is a warning message. This allows the user to define new debugger commands, see Section 5.6.10 [Hooks
Related to Breakpoints], page 267, for an example. See Section 11.3.61 [mpg-ref-debugger_command_hook], page 1004.

5.8 The Processing of Breakpoints

This section describes in detail how the debugger handles the breakpoints. For the purpose of this section disabled breakpoints are not taken into account: whenever we refer to the existence of some breakpoint(s), we always mean the existence of enabled breakpoint(s).

The Prolog engine can be in one of the following three states with respect to the debugger:

- **no debugging**
  - if there are no advice points and the debugger is either switched off, or doing a skip;

- **full debugging**
  - if the debugger is in trace or debug mode (creeping or leaping), or there are any generic breakpoints;

- **selective debugging**
  - in all other cases.

In the **selective debugging** state only those predicate invocations are examined, for which there exists a specific breakpoint. In the **full debugging** state all invocations are examined, except those calling a predicate of a hidden module (but even these will be examined, if there is a specific breakpoint for them). In the **no debugging** state the debugger is not entered at predicate invocations.

Now we describe what the debugger does when examining an invocation of a predicate, i.e. executing its Call port. The debugger activities can be divided into three stages: advice point processing, spypoint processing and interaction with the user. The last stage may be repeated several times before program execution continues.

The first two stages are similar, as they both search for an applicable breakpoint (spypoint or advice point). This common breakpoint search is carried out as follows. The debugger considers all breakpoints of the given type, most recent first. For each breakpoint, the test part of the specification is evaluated, until one successful is found. Any variable bindings created in this successful evaluation are then discarded (this is implemented by enclosing it in double negation). The first breakpoint, for which the evaluation of the test part succeeds is selected. If such a breakpoint can be found, then the breakpoint search is said to have completed successfully, otherwise it is said to have failed.

If a breakpoint has been selected, then its action part is evaluated, normally setting some debugger action variables. If the action part fails, then as a side effect, it is ensured that a procedure box will be built. This is achieved by changing the value of the command action variable from flit to proceed.

Having described the common breakpoint search, let us look at the details of the first stage, advice point processing. This stage is executed only if there are any advice points set. First, the debugger action variables are initialized: mode is set to the current debugger
mode, command to proceed and show to silent. Next, advice point search takes place. If this fails, then command is set to flit, otherwise its value is unchanged.

After completing the advice point search the command variable is examined. If its value is divertive, i.e. different from proceed and flit, then the spypoint search stage is omitted, and the debugger continues with the third stage. Otherwise, it is noted that the advice point processing has requested the building of a procedure box (i.e. command = proceed), and the debugger continues with the second stage.

The second stage is spypoint processing. This stage is skipped if the debugger is switched off or doing a skip (mode is off or skip(_)). First the show and command variables are re-assigned, based on the hiddenness of the predicate being invoked, the debugger mode, and the leashing status of the port. If the predicate is both defined in, and called from a hidden module, then their values will be silent and flit. An example of this is when a built-in predicate is called from a hidden module, e.g. from a library. Otherwise, in trace mode, their values are print and ask for leashed ports, and print and proceed for unleashed ports. In debug mode, the variables are set to silent and proceed, while in zip mode to silent and flit (Section 5.6.5 [Breakpoint Actions], page 252, contains a tabulated listing of these initialization values).

Having initialized the debugger action variables, the spypoint search phase is performed. If an empty action part has been selected in a successful search, then show and command are set to print and ask. The failure of the search is ignored.

The third stage is the interactive part. First, the goal in question is displayed according to the value of show. Next, the value of command is checked: if it is other than ask, then the interactive stage ends. Otherwise, (it is ask), the variable show is re-initialized to print, or to print-Sel, if its value was of form Method-Sel. Next, the debugger prompts the user for a command, which is interpreted either in the standard way, or through user:debugger_command_hook/2. In both cases the debugger action variables are modified as requested, and the interactive part is repeated.

After the debugger went through all the three stages, it decides whether to build a procedure box. This will happen if either the advice point stage or the other two stages require it. The latter is decided by checking the command variable: if that is flit or flit(Old,New), then no procedure box is required by the spypoint part. If the advice point does require the building of a procedure box, then the above command values are replaced by proceed and proceed(Old,New), respectively.

At the end of the process the value of mode will be the new debugging mode, and command will determine what the debugger will do; see Section 5.9.9 [Action Variables], page 282.

A similar three-stage process is carried out when the debugger arrives at a non-Call port of a predicate. The only difference is that the building of a procedure box is not considered (flit is equivalent to proceed), and the hiddenness of the predicate is not taken into account.
Chapter 5: Debugging

While the Prolog system is executing the above three-stage process for any of the ports, it is said to be inside the debugger. This is relevant, because some of the conditions can only be evaluated in this context.

5.9 Breakpoint Conditions

This section describes the format of simple breakpoint conditions. We first list the tests that can be used to inquire the state of execution. We then proceed to describe the conditions usable in the action part and the options for focusing on past execution states. Finally, we describe condition macros and the format of the values of the debugger action variables.

We distinguish between two kinds of tests, based on whether they refer to information stored in the backtrace or not. The latter category, the non-backtraced tests, contains the conditions related to the current port (port, bid, mode, show, command, get) and the breakpoint type selection conditions (advice and debug). All remaining tests refer to information stored in the backtrace.

Non-backtraced tests will raise an exception, if they appear in calls to execution_state/1 from outside the debugger, or in queries about past execution state, in execution_state/2.

Backtraced tests are allowed both inside and outside the debugger. However such tests can fail if the given query is not meaningful in the given context, e.g. if execution_state(goal(G)) is queried before any breakpoints were encountered.

Note that if a test is used in the second argument of execution_state/2, then the term current, in the following descriptions, should be interpreted as referring to the execution state focused on (described by the first argument of execution_state/2).

5.9.1 Tests Related to the Current Goal

The following tests give access to basic information about the current invocation.

inv(Inv) The invocation number of the current goal is Inv. Invocation numbers start from 1.

depth(Depth) The current execution depth is Depth.

goal(MGoal) The current goal is an instance of the module name expanded MGoal template. The current goal and MGoal are unified. This condition is equivalent to subsumes(MGoal,CurrentGoal) (subsumes/2 is defined in library(terms), see Section 10.46 [lib-terms], page 862).

pred(MFunc) The module name expanded MFunc template matches (see notes below) the functor (M:F/N) of the current goal. The unification required for matching is carried out.

module(Module) The current goal is invoked from module Module. For compiled calls to built-in predicates Module will always be prolog.
goal_private(GoalPriv)

The private information associated with the current goal is GoalPriv. This is initialized to an unbound variable at the Call port. It is strongly recommended that GoalPriv be used as an open ended list, see Section 5.6.9 [Storing User Information in the Backtrace], page 265.

exited(Boolean)

Boolean is true if the current invocation has exited, and false otherwise. This condition is mainly used for looking at past execution states.

parent_inv(Inv)

The invocation number of the debugger-parent (see notes below) of the current goal is Inv.

ancestor(AncGoal, Inv)

The youngest debugger-ancestor of the current goal, which is an instance of the module name expanded AncGoal template, is at invocation number Inv. The unification required for matching is carried out.

Notes:

The debugger-parent of a goal is the youngest ancestor of the goal present on the backtrace. This will differ from the ordinary parent if not all goals are traced, e.g. if the goal in question is reached in zip mode. A debugger-ancestor of a goal is any of its ancestors on the backtrace.

In the goal and ancestor tests above, there is a given module qualified goal template, say ModT:GoalT, and it is matched against a concrete goal term Mod:Goal in the execution state. This matching is carried out as follows:

a. It is checked that Goal is an instance of GoalT.
b. Goal and GoalT are unified.
c. It is checked that Mod and ModT are either unifiable (and are unified), or name such modules in which Goal has the same meaning, i.e. either one of Mod:Goal and ModT:Goal is an exported variant of the other, or both are imported from the same module.

Similar matching rules apply for predicate functors, in the pred condition. In this test the argument holds a module qualified functor template, say ModT:Name/Arity, and this is matched against a concrete goal term Mod:Goal in the execution state.

a. It is checked that the functor of Goal unifies with Name/Arity, and this unification is carried out.
b. It is checked that Mod and ModT are either unifiable (and are unified), or name such modules in which Goal has the same meaning.

5.9.2 Tests Related to Source Information

These tests provide access to source related information. The file and line tests will fail if no source information is present. The parent_clause and parent_pred tests are available for interpreted code only, they will fail in compiled code.
file(File)
   The current goal is invoked from a file whose absolute name is File.

line(File,Line)
   The current goal is invoked from line Line, from within a file whose absolute
   name is File.

line(Line)
   The current goal is invoked from line Line.

parent_clause(Cl)
   The current goal is invoked from clause Cl.

parent_clause(Cl,Sel)
   The current goal is invoked from clause Cl and within its body it is pointed to
   by the subterm selector Sel.

parent_clause(Cl,Sel,I)
   The current goal is invoked from clause Cl, it is pointed to by the subterm
   selector Sel within its body, and it is the Ith goal within it. The goals in the
   body are counted following their textual occurrence.

parent_pred(Pred)
   The current goal is invoked from predicate Pred.

parent_pred(Pred,N)
   The current goal is invoked from predicate Pred, clause number N.

The parent_pred tests match their first argument against the functor of the parent predi-
cate in the same way as the pred test does; see the notes in the previous section (Section 5.9.1
[Goal Tests], page 277).

5.9.3 Tests Related to the Current Port

These tests can only be used inside the debugger and only when focused on the current
invocation. If they appear in execution_state/2 or in execution_state/1 called from
outside the debugger, then an exception will be raised.

The notion of port in breakpoint handling is more general than outlined earlier in Section 5.1
[Procedure Box], page 231. Here, the following terms are used to describe a port:

   call, exit(nondet), exit(det), redo, fail,
   exception(Exception), block, unblock

Furthermore, the atoms exit and exception can be used in the port condition (see below),
to denote either of the two exit ports and an arbitrary exception port, respectively.

port(Port)
   The current execution port matches Port in the following sense: either Port
   and the current port unify, or Port is the functor of the current port (e.g.
   port(exit) holds for both exit(det) and exit(nondet) ports).

   As explained earlier, the port condition for a non Call port is best placed in
   the action part. This is because the failure of the action part will cause the
debugger to pass through the Call port silently, and to build a procedure box, even in zip mode. The following idiom is suggested for creating breakpoints at non Call ports:

\[ \text{add_breakpoint}((\text{Tests} - [\text{port}(\text{Port}), \text{Actions}]), \text{BID}). \]

\text{bid}(\text{BID}) The breakpoint being examined has a breakpoint identifier \text{BID}. (\text{BID} = \text{off} if no breakpoint was selected.)

\text{mode}(\text{Mode}) \quad \text{Mode} \text{ is the value of the mode variable, which normally reflects the current debugger mode.}

\text{command}(\text{Command}) \quad \text{Command} \text{ is the value of the command variable, which is the command to be executed by default, if the breakpoint is selected.}

\text{show}(\text{Show}) \quad \text{Show} \text{ is the value of the show variable, i.e. the default show method (the method for displaying the goal in the trace message).}

The last three of the above tests access the debugger action variables. These breakpoint conditions have a different meaning in the action part. For example, the condition \text{mode}('trace'), if it occurs in the tests, checks if the current debugger mode is \text{trace}. On the other hand, if the same term occurs within the action part, then it sets the debugger mode to \text{trace}.

To support the querying of the action variables in the action part, the following breakpoint condition is provided:

\text{get}(\text{ActVar}) \quad \text{Equivalent to ActVar, where this is an action variable test, i.e. one of the terms mode(\text{Mode}), command(\text{Command}), show(\text{Show}). It has this meaning in the action part as well.}

For the \text{port}, \text{mode}, \text{command} and \text{show} conditions, the condition can be replaced by its argument, if that is not a variable. For example the condition \text{call} can be used instead of \text{port}(\text{call}). Conditions matching the terms listed above as valid port values will be converted to a \text{port} condition. Similarly, any valid value for the three debugger action variables is converted to an appropriate condition. These valid values are described in Section 5.9.9 [Action Variables], page 282.

### 5.9.4 Tests Related to the Break Level

These tests can be used both inside and outside the condition evaluation process, and also can be used in queries about past break levels.

\text{break_level}(N) \quad \text{We are at (or focused on) break level N (N = 0 for the outermost break level).}
The last invocation number used within the current break level is \texttt{MaxInv}. Note that this invocation number may not be present in the backtrace (because the corresponding call exited determinately).

The private information associated with the break level is \texttt{Priv}. Similarly to \texttt{goal_private/1}, this condition refers initially to an unbound variable and can be used to store an arbitrary Prolog term. However, it is strongly recommended that \texttt{Priv} be used as an open ended list, see Section 5.6.9 [Storing User Information in the Backtrace], page 265.

### 5.9.5 Other Conditions

The following conditions are for prescribing or checking the breakpoint type. They cause an exception if used outside the debugger or in \texttt{execution_state/2}.

- **advice**: The breakpoint in question is of advice type.
- **debugger**: The breakpoint in question is of debugger type.

The following construct converts an arbitrary Prolog goal into a condition.

- **true(Cond)**
  
  The Prolog goal \texttt{Cond} is true, i.e. \texttt{once(Cond)} is executed and the condition is satisfied if and only if this completes successfully. If an exception is raised during execution, then an error message is printed and the condition fails.
  
  The substitutions done on executing \texttt{Cond} are carried out. \texttt{Cond} is subject to module name expansion. If used in the test part of spypoint conditions, then the goal should not have any side effects, as the test part may be evaluated several times.

The following conditions represent the Boolean constants.

- **true**
- **false**

#### 5.9.6 Conditions Usable in the Action Part

The meaning of the following conditions, if they appear in the action part, is different from their meaning in the test part.

- **mode(\textit{Mode])**
  
  Set the debugger mode to \textit{Mode}.

- **command(\textit{Command])**
  
  Set the command to be executed to \textit{Command}.

- **show(\textit{Show})**
  
  Set the show method to \textit{Show}.
The values admissible for Mode, Command and Show are described in Section 5.9.9 [Action Variables], page 282.

Furthermore, any other condition can be used in the action part, except for the ones specifying the breakpoint type (advice and debugger). Specifically, the get condition can be used to access the value of an action variable.

5.9.7 Options for Focusing on a Past State

The following ground terms can be used in the first argument of execution_state/2 (see Section 5.7 [Breakpoint Predicates], page 272). Alternatively, a list containing such terms can be used. If a given condition occurs multiple times, then only the last one is considered. The order of conditions within the list does not matter.

- break_level(\textit{BL})
  Focus on the current invocation of break level $\textit{BL}$. $\textit{BL}$ is the break level number, the top level being break_level(0). For past break levels, the current invocation is the one from which the next break level was entered.

- inv(Inv)
  Focus on the invocation number $\textit{Inv}$ of the currently focused break level.

5.9.8 Condition Macros

There are a few condition macros expanding to a list of other conditions:

- unleash
  Expands to [show(print), command(proceed)]

- hide
  Expands to [show(silent), command(proceed)]

- leash
  Expands to [show(print), command(ask)]

The user can also define condition macros using the hook predicate below.

- \texttt{breakpoint}\_\texttt{expansion(}+\textit{Macro}, -\textit{Body})  
  \texttt{hook, development}

\texttt{user:breakpoint}\_\texttt{expansion(}+\textit{Macro}, -\textit{Body})

  This predicate is called with each (non-composite) breakpoint test or action, as its first argument. If it succeeds, then the term returned in the second argument (\textit{Body}) is substituted for the original condition. The expansion is done at the time the breakpoint is added.

  Note that \textit{Body} can be composite, but it cannot be of form \textit{Tests-Actions}. This means that the whole \textit{Body} will be interpreted as being in either the test or the action part, depending on the context.

  The built-in breakpoint conditions cannot be redefined using this predicate. See Section 11.3.28 [mpg-ref-breakpoint\_expansion], page 966.

5.9.9 The Action Variables

In this section we list the possible values of the debugger action variables, and their meaning.

Note that the Prolog terms, supplied as values, are copied when a variable is set. This is relevant primarily in the case of the proceed/2 and flit/2 values.
Values allowed in the `show` condition:

- **print**: Write using options stored in the `debugger_print_options` Prolog flag.
- **silent**: Display nothing.
- **display**: Write using `display`.
- **write**: Write using `writeq`.
- **write_term(Options)**: Write using options `Options`.

**Method-Sel**

Display only the subterm selected by `Sel`, using `Method`. Here, `Method` is one of the methods above, and `Sel` is a subterm selector.

Values allowed in the `command` condition:

- **ask**: Ask the user what to do next.
- **proceed**: Continue the execution without interacting with the user (cf. unleashing).
- **flit**: Continue the execution without building a procedure box for the current goal (and consequently not encountering any other ports for this invocation). Only meaningful at Call ports, at other ports it is equivalent to `proceed`.

**proceed(Goal,New)**

Unless at call port, first go back to the call port (retry the current invocation; see the `retry(Inv)` command value below). Next, unify the current goal with `Goal` and execute the goal `New` in its place. Create (or keep) a procedure box for the current goal.

This construct is used by the ‘u’ (unify) interactive debugger command.

Both the `Goal` and `New` arguments are module name expanded when the breakpoint is added: the module of `Goal` defaults to the module of the current goal, while that of `New` to the module name of the breakpoint specification. If the command value is created during run time, then the module name of both arguments defaults to the module of the current goal.

The term `proceed(Goal,New)` will be copied when the `command` action variable is set. Therefore breakpoint specs of form

\[
\text{Tests} - [\text{goal}(\text{foo}(X)), \ldots, \text{proceed}(\_, \text{bar}(X))]\]

should be avoided, and

\[
\text{Tests} - [\text{goal}(\text{foo}(X)), \ldots, \text{proceed}(\text{foo}(Y), \text{bar}(Y))]
\]

should be used instead. The first variant will not work as expected if `X` is non-ground, as the variables in the `bar/1` call will be detached from the original ones in `foo/1`. Even if `X` is ground, the first variant may be much less efficient, as it will copy the possibly huge term `X`.

**flit(Goal,New)**

Same as `proceed(Goal,New)`, but do not create (or discard) a procedure box for the current goal. (Consequently no other ports will be encountered for this invocation.)
Notes for `proceed/2`, on module name expansion and copying, also apply to `flit/2`.

`raise(E)` Raise the exception `E`.

`abort` Abort the execution.

`retry(Inv)` Retry the most recent goal in the backtrace with an invocation number less or equal to `Inv` (go back to the Call port of the goal). This is used by the interactive debugger command `r`, retry; see Section 5.5 [Debug Commands], page 237.

`reexit(Inv)` Re-exit the invocation with number `Inv` (go back to the Exit port of the goal). `Inv` must be an exact reference to an exited invocation present in the backtrace (exited nondeterminately, or currently being exited). This is used by the interactive debugger command `je`, jump to Exit port; see Section 5.5 [Debug Commands], page 237.

`redo(Inv)` Redo the invocation with number `Inv` (go back to the Redo port of the goal). `Inv` must be an exact reference to an exited invocation present in the backtrace. This is used by the interactive debugger command `jr`, jump to Redo port; see Section 5.5 [Debug Commands], page 237.

`fail(Inv)` Fail the most recent goal in the backtrace with an invocation number less or equal to `Inv` (transfer control back to the Fail port of the goal). This is used by the interactive debugger command `f`, fail; see Section 5.5 [Debug Commands], page 237.

Values allowed in the `mode` condition:

`qskip(Inv)` Quasi-skip until the first port with invocation number less or equal to `Inv` is reached. Having reached that point, `mode` is set to `trace`. Valid only if `Inv ≥ 1` and furthermore `Inv ≤ CurrInv` for entry ports (Call, Redo), and `Inv < CurrInv` for all other ports, where `CurrInv` is the invocation number of the current port.

`skip(Inv)` Skip until the first port with invocation number less or equal to `Inv` is reached, and set `mode` to `trace` there. `Inv` should obey the same rules as for `qskip`.

`trace` Creep.

`debug` Leap.

`zip` Zip.

`off` Continue without debugging.
5.10 Consulting during Debugging

It is possible, and sometimes useful, to consult a file whilst in the middle of program execution. Predicates that have been successfully executed and are subsequently redefined by a consult and are later reactivated by backtracking, will not notice the change of their definitions. In other words, it is as if every predicate, when called, creates a copy of its definition for backtracking purposes.

5.11 Catching Exceptions

Usually, exceptions that occur during debugging sessions are displayed only in trace mode and for invocation boxes for predicates with spypoints on them, and not during skips. However, it is sometimes useful to make exceptions trap to the debugger at the earliest opportunity instead. The hook predicate `user:error_exception/1` provides such a possibility:

```
error_exception(+Exception) hook
user:error_exception(+Exception)
```

This predicate is called at all Exception ports. If it succeeds, then the debugger enters trace mode and prints an exception port message. Otherwise, the debugger mode is unchanged and a message is printed only in trace mode or if a spypoint is reached, and not during skips. See Section 11.3.73 [mpg-ref-error_exception], page 1017.

Note that this hook takes effect when the debugger arrives at an Exception port. For this to happen, procedure boxes have to be built, e.g. by running (the relevant parts of) the program in debug mode.

A useful definition that ensures that all standard error exceptions causes the debugger to enter trace mode, is as follows:

```
:- multifile user:error_exception/1.
user:error_exception(error(_,_)).
```

(this example would not have worked prior to release 4.0.5).

5.12 Predicate Summary

```
add_breakpoint(+Conditions, -BID) development
  Creates a breakpoint with Conditions and with identifier BID.

user:breakpoint_expansion(+Macro, -Body) hook,development
  defines debugger condition macros

coverage_data(?Data) since release 4.2,development
  Data is the coverage data accumulated so far

current_breakpoint(?Conditions, ?BID, ?Status, ?Kind, ?Type) development
  There is a breakpoint with conditions Conditions, identifier BID, enabledness Status, kind Kind, and type Type.
```
debug switch on debugging

user:debugger_command_hook(+DCommand,-Actions) hook, development
  Allows the interactive debugger to be extended with user-defined commands.

debugging development display debugging status information

disable_breakpoints(+BIDs) development
  Disables the breakpoints specified by BIDs.

enable_breakpoints(+BIDs) development
  Enables the breakpoints specified by BIDs.

user:error_exception(+Exception) hook
  Exception is an exception that traps to the debugger if it is switched on.

execution_state(+Tests) development
  Tests are satisfied in the current state of the execution.

execution_state(+FocusConditions, +Tests) development
  Tests are satisfied in the state of the execution pointed to by FocusConditions.

leash(+M) development
  set the debugger’s leashing mode to M

nodebug development switch off debugging

nospy(:P) development
  remove spypoints from the procedure(s) specified by P

nospyall development
  remove all spypoints

notrace development
  switch off debugging (same as nodebug/0)

nozip development
  switch off debugging (same as nodebug/0)

print_coverage(+) development since release 4.2
print_coverage(?Data) since release 4.2, development
  The coverage data Data is displayed in a hierarchical format. Data defaults to the coverage data accumulated so far.

print_profile(+) development since release 4.2
print_profile(?Data) since release 4.2, development
  The profiling data Data is displayed in a format similar to gprof(1). Data defaults to the profiling data accumulated so far.

profile_data(?Data) since release 4.2, development
  Data is the profiling data accumulated so far

profile_reset development since release 4.2, development
  All profiling data is reset.
remove_breakpoints(+BIDs)
   Removes the breakpoints specified by BIDs.
spy(:P)
spy(:P,:C)
   set spypoints on the procedure(s) specified by P with conditions C
trace
   switch on debugging and start tracing immediately
unknown(-O,+N)
   Changes action on undefined predicates from O to N.
user:unknown_predicate_handler(+G,+M,-N)
   handle for unknown predicates.
zip
   switch on debugging in zip mode
6 Mixing C/C++ and Prolog

SICStus Prolog provides several ways of interfacing with code written in other languages:

- A uni-directional, loosely coupled interface from client components written in any language to a Prolog server. This is library(jsonrpc) (see Section 10.22 [lib-jsonrpc], page 604). It uses JSON in its process communication.
- A uni-directional, loosely coupled interface from client components written in Java or .NET to a Prolog server. This is library(prologbeans) (see Section 10.35 [lib-prologbeans], page 731). It uses a custom protocol in its process communication.
- A bi-directional, tightly coupled interface between program components written in Java and Prolog. This is library(jasper) (see Section 10.20 [lib-jasper], page 572).
- A bi-directional, tightly coupled interface for program components written in C/C++ and Prolog. This is the topic of the current chapter.

The C side of the interface defines a number of functions and macros for various operations. On the Prolog side, you have to supply declarations specifying the names and argument/value types of C functions being called as predicates. These declarations are used by the predicate load_foreign_resource/1, which performs the actual binding of functions to predicates. They are also needed when the functions are unloaded, for example when SICStus is halted.

In most cases, the argument/value type declarations suffice for making the necessary conversions of data automatically as they are passed between C and Prolog. However, it is possible to declare the type of an argument to be a Prolog term, in which case the receiving function will see it as a “handle” object, called an SP_term_ref, for which access functions are provided.

The C support routines are available in a development system as well as in runtime systems. The support routines include:

- Static and dynamic linking of C code into the Prolog environment.
- Automatic conversion between Prolog terms and C data with foreign/[2,3] declarations.
- Functions for accessing and creating Prolog terms, and for creating and manipulating SP_term.refs.
- The Prolog system may call C predicates, which may call Prolog back without limits on recursion. Predicates that call C may be defined dynamically from C.
- Support for creating stand-alone executables.
- Support for creating user defined Prolog streams.
- Functions to read and write on Prolog streams from C.
- Functions to install interrupt handlers that can safely call Prolog.
- Functions for manipulating mutual exclusion locks.
- User hooks that can be used to perform user defined actions e.g. for customizing the memory management bottom layer.
In addition to the interface described in this chapter, `library(structs)` and `library(objects)` (see Section 10.43 [lib-structs], page 759, and Section 10.29 [lib-objects], page 639) allow Prolog to hold pointers to C data structures and arrays and access and store into fields in those data structures in a very efficient way, allowing the programmer to stay completely inside Prolog.

6.1 Notes

The `SP_PATH` variable

It is normally not necessary, nor desirable, to set this system property (or environment variable), but its value will be used, as a fall-back, at runtime if it cannot be determined automatically during initialization of a runtime or development system. In this chapter, `SP_PATH` is used as a shorthand, as follows.

On Windows, `SP_PATH` is a shorthand for the SICStus Prolog installation directory, whose default location for SICStus 4.7.0 is `C:\Program Files\SICStus Prolog VC15 4.7.0`.

On UNIX, the default installation directory for SICStus 4.7.0 is `/usr/local/sicstus4.7.0` and `SP_PATH` is a shorthand for the subdirectory `lib/sicstus-4.7.0` of the installation directory, e.g.: `/usr/local/sicstus4.7.0/lib/sicstus-4.7.0/`.

See Section 4.17.1 [System Properties and Environment Variables], page 224, for more information.

Definitions and declarations

Type definitions and function declarations for the interface are found in the header file `<sicstus/sicstus.h>`.

Error Codes

The value of many support functions is a return code, namely: `SP_SUCCESS` for success, `SP_FAILURE` for failure, `SP_ERROR` if an error condition occurred, or if an uncaught exception was raised during a call from C to Prolog. If the value is `SP_ERROR`, then the macro `SP_errno` will return a value describing the error condition:

```c
int SP_errno
```

The function `SP_error_message()` returns a pointer to the diagnostic message corresponding to a specified error number.

Wide Characters

The foreign interface supports wide characters. Whenever a sequence of possibly wide character codes is to be passed to or from a C function it is encoded as a sequence of bytes, using the UTF-8 encoding. Unless noted otherwise the encoded form is terminated by a NUL byte. This sequence of bytes will be called an `encoded string`, representing the given sequence of character codes. Note that it is a property of the UTF-8 encoding that it does not change ASCII character code sequences.

If a foreign function is specified to return an encoded string, then an exception will be raised if, on return to Prolog, the actual string
is malformed (is not a valid sequence of UTF-8 encoded characters). The exception raised is `error(representation_error(misEncodedString),representation_error(...,...,misEncodedString))`.

### 6.2 Calling C from Prolog

Functions written in the C language may be called from Prolog using an interface in which automatic type conversions between Prolog terms and common C types are declared as Prolog facts. Calling without type conversion can also be specified, in which case the arguments and values are passed as `SP_term_refs`. This interface is partly modeled after Quintus Prolog.

The functions installed using this foreign language interface may invoke Prolog code and use the support functions described in the other sections of this chapter.

Functions, or their equivalent, in any other language having C compatible calling conventions may also be interfaced using this interface. When referring to C functions in the following, we also include such other language functions. Note however that a C compiler is needed since a small amount of glue code (in C) must be generated for interfacing purposes.

As an alternative to this interface, `SP_define_c_predicate()` defines a Prolog predicate such that when the Prolog predicate is called it will call a C function with a term corresponding to the Prolog goal. For details, see Section 12.3.11 [cpg-ref-SP_define_c.predicate], page 1273.

### 6.2.1 Foreign Resources

A foreign resource is a set of C functions, defined in one or more files, installed as an atomic operation. The name of a foreign resource, the resource name, is an atom, which should uniquely identify the resource. Thus, two foreign resources with the same name cannot be installed at the same time, even if they correspond to different files.

The resource name of a foreign resource is derived from its file name by deleting any leading path and the suffix. Therefore the resource name is not the same as the absolute file name. For example, the resource name of both `~john/foo/bar.so` and `~ringo/blip/bar.so` is `bar`. If `load_foreign_resource('~john/foo/bar')` has been done, then `~john/foo/bar.so` will be unloaded if either `load_foreign_resource('~john/foo/bar')` or `load_foreign_resource('~ringo/blip/bar')` is subsequently called.

It is recommended that a resource name be all lowercase, starting with ‘a’ to ‘z’ followed by a sequence consisting of ‘a’ to ‘z’, underscore (‘_’), and digits. The resource name is used to construct the file name containing the foreign resource.

For each foreign resource, a `foreign_resource/2` fact is used to declare the interfaced functions. For each of these functions, a `foreign/[2,3]` fact is used to specify conversions between predicate arguments and C-types. These conversion declarations are used for creating the necessary interface between Prolog and C.
The functions making up the foreign resource, the automatically generated glue code, and any libraries, are compiled and linked, using the splfr tool (see Section 6.2.5 [The Foreign Resource Linker], page 296), to form a linked foreign resource. A linked foreign resource can be either static or dynamic. A static resource is simply a relocatable object file containing the foreign code. A dynamic resource is a shared library (`.so' under most UNIX dialects, `.dll' under Windows), which is loaded into the Prolog executable at runtime.

Foreign resources can be linked into the Prolog executable either when the executable is built (prelinked), or at runtime. Prelinking can only be done using static resources. Runtime-linking can only be done using dynamic resources. Dynamic resources can also be unlinked.

In all cases, the declared predicates are installed by the built-in predicate load_foreign_resource/1. If the resource was prelinked, then only the predicate names are bound; otherwise, runtime-linking is attempted (using dlopen(), LoadLibrary(), or similar).

### 6.2.2 Conversion Declarations

Conversion declaration predicates:

```prolog
foregn_resource(+ResourceName,+Functions) hook
```

Specifies that a set of foreign functions, to be called from Prolog, are to be found in the resource named by ResourceName. Functions is a list of functions exported by the resource. Only functions that are to be called from Prolog and optionally one init function and one deinit function should be listed. The init and deinit functions are specified as init(Function) and deinit(Function) respectively (see Section 6.2.6 [Init and Deinit Functions], page 297). This predicate should be defined entirely in terms of facts (unit clauses) and will be called in the relevant module, i.e. not necessarily in the user module. For example:

```prolog
foreign_resource('terminal', [scroll,pos_cursor,ask]).
```

specifies that functions scroll(), pos_cursor() and ask() are to be found in the resource terminal. See Section 11.3.84 [mpg-ref-foreign_resource], page 1031.

```prolog
foreign(+CFunctionName, +Predicate)
foreign(+CFunctionName, +Language, +Predicate)
```

Specify the Prolog interface to a C function. Language is at present constrained to the atom c, so there is no advantage in using foreign/3 over foreign/2. CFunctionName is the name of a C function. Predicate specifies the name of the Prolog predicate that will be used to call CFunction(). Predicate also specifies how the predicate arguments are to be translated to and from the corresponding C arguments. These predicates should be defined entirely in terms of facts (unit clauses) and will be called in the relevant module, i.e. not necessarily in the user module. For example:

```prolog
foreign(pos_cursor, c, move_cursor(+integer, +integer)).
```

The above example says that the C function pos_cursor() has two integer value arguments and that we will use the predicate move_cursor/2 to call this
function. A goal `move_cursor(5, 23)` would translate into the C call `pos_cursor(5, 23);`.

The third argument of the predicate `foreign/3` specifies how to translate between Prolog arguments and C arguments. A call to a foreign predicate will throw an Instantiation Error if an input argument is uninstantiated, a Type Error if an input argument has the wrong type, or a Domain Error if an input argument is in the wrong domain. The call will fail upon return from the function if the output arguments do not unify with the actual arguments.

The available conversions are listed in the next subsection. See Section 11.3.83 [mpg-ref-foreign], page 1030.

### 6.2.3 Conversions between Prolog Arguments and C Types

The following table lists the possible values for the arguments in the predicate specification of `foreign/[2,3]`. The value declares which conversion between corresponding Prolog argument and C type will take place.

**Prolog: +integer**

C: `SP_integer`

The argument should be a number. It is converted to a C `SP_integer` and passed to the C function. If the number does not fit in a `SP_integer`, an exception is thrown.

**Prolog: +float**

C: `double`

The argument should be a number. It is converted to a C `double` and passed to the C function. If the number is a large integer that does not fit in a `double`, then an exception is thrown.

**Prolog: +atom**

C: `SP_atom`

The argument should be an atom. Its canonical representation is passed to the C function.

**Prolog: +codes**

C: `char const *`

The argument should be a code list. The C function will be passed the address of an array with the encoded string representation of these characters. The array is subject to reuse by other support functions, so if the value is going to be used on a more than temporary basis, then it must be moved elsewhere.

**Prolog: +string**

C: `char const *`

The argument should be an atom. The C function will be passed the address of an encoded string representing the characters of the atom. **Please note:** The C function must not overwrite the string.

**Prolog: +address**

C: `void *`

The value passed will be a `void *` pointer.
Prolog: +address(TypeName)
C: TypeName *
   The value passed will be a TypeName * pointer.

Prolog: +term
C: SP_term_ref
   The argument could be any term. The value passed will be the internal representation of the term.

Prolog: -integer
C: SP_integer *
   The C function is passed a reference to an uninitialized SP_integer. The value returned will be converted to a Prolog integer.

Prolog: -float
C: double *
   The C function is passed a reference to an uninitialized double. The value returned will be converted to a Prolog float. If the value returned is not finite, i.e. it is infinite or NaN, then an exception is thrown.

Prolog: -atom
C: SP_atom *
   The C function is passed a reference to an uninitialized SP_atom. The value returned should be the canonical representation of a Prolog atom.

Prolog: -codes
C: char const **
   The C function is passed the address of an uninitialized char *. The returned encoded string will be converted to a Prolog code list.

Prolog: -string
C: char const **
   The C function is passed the address of an uninitialized char *. The returned encoded string will be converted to a Prolog atom. Prolog will copy the string to a safe place, so the memory occupied by the returned string may be reused during subsequent calls to foreign code.

Prolog: -address
C: void **
   The C function is passed the address of an uninitialized void *.

Prolog: -address(TypeName)
C: TypeName **
   The C function is passed the address of an uninitialized TypeName *.

Prolog: -term
C: SP_term_ref
   The C function is passed a new SP_term_ref, and is expected to set its value to a suitable Prolog term. Prolog will try to unify the value with the actual argument.
Chapter 6: Mixing C/C++ and Prolog

Prolog: [-integer]
C: SP_integer F()
   The C function should return an SP_integer. The value returned will be
   converted to a Prolog integer.

Prolog: [-float]
C: double F()
   The C function should return a double. The value returned will be converted
to a Prolog float. If the value returned is not finite, i.e. it is infinite or NaN,
then an exception is thrown.

Prolog: [-atom]
C: SP_atom F()
   The C function should return an SP_atom. The value returned must be the
   canonical representation of a Prolog atom.

Prolog: [-codes]
C: char const *F()
   The C function should return a char *. The returned encoded string will be
   converted to a Prolog code list.

Prolog: [-string]
C: char const *F()
   The C function should return a char *. The returned encoded string will be
   converted to a Prolog atom. Prolog will copy the string to a safe place, so the
   memory occupied by the returned string may be reused during subsequent calls
to foreign code.

Prolog: [-address]
C: void *F()
   The C function should return a void *, which will be converted to a Prolog
   integer.

Prolog: [-address(TypeName)]
C: TypeName *F()
   The C function should return a TypeName *.

Prolog: [-term]
C: SP_term_ref F()
   The C function should return an SP_term_ref. Prolog will try to unify its value
   with the actual argument.

6.2.4 Interface Predicates

load_foreign_resource(:Resource)
   Unless a foreign resource with the same name as Resource has been statically
linked, the linked foreign resource specified by Resource is linked into the Prolog
load image. In both cases, the predicates defined by Resource are installed, and
any init function is called. Dynamic linking is not possible if the foreign resource
was linked using the --static option.
If a resource with the same name has been previously loaded, then it will be unloaded, as if `unload_foreign_resource(Resource)` were called, before `Resource` is loaded.

An example of usage of `load_foreign_resource/1` can be found in its reference page, Section 11.3.118 [mpg-ref-load_foreign_resource], page 1079.

```
unload_foreign_resource(:ResourceName)
```

Any deinit function associated with `ResourceName`, a resource name, is called, and the predicates defined by `ResourceName` are uninstalled. If `ResourceName` has been dynamically linked, then it is unlinked from the Prolog load image.

If no resource named `ResourceName` is currently loaded, then an existence error is raised.

For backward compatibility, `ResourceName` can also be of the same type as the argument to `load_foreign_resource/1`. In that case the resource name will be derived from the absolute file name in the same manner as for `load_foreign_resource/1`. Also for backward compatibility, `unload_foreign_resource/1` is a meta-predicate, but the module is ignored. See Section 11.3.245 [mpg-ref-unload_foreign_resource], page 1239.

Please note: all foreign resources are unloaded before Prolog exits.
This implies that the C library function `atexit(func)` cannot be used if `func` is defined in a dynamically linked foreign resource.

### 6.2.5 The Foreign Resource Linker

The foreign resource linker, `splfr`, is used for creating foreign resources (see Section 6.2.1 [Foreign Resources], page 291). `splfr` reads terms from a Prolog file extracting any `foreign_resource/2` fact with first argument matching the resource name and all foreign/[2,3] facts. Based on this information, it generates the necessary glue code, including a header file that the user code should include, and combines it with any additional C or object files provided by the user into a linked foreign resource. The output file name will be the resource name with a suitable extension.

Note that no pathnames passed to `splfr` should contain spaces. Under Windows, this can be avoided by using the short version of pathnames as necessary.

See Section 13.6 [too-splfr], page 1402, for detailed information about `splfr` options etc..

#### 6.2.5.1 Customizing splfr.

The `splfr` tool reads a configuration file at start-up that contains default values for many configurable parameters. It is sometimes useful to modify these in order to adapt to local variations.

Both `splfr` and `spld` use the same configuration file and use the same options for changing the default parameters. See Section 6.7.3.1 [Customizing spld], page 323, for details.

#### 6.2.5.2 Creating Linked Foreign Resources Manually under UNIX

The only supported method for building foreign resources is by compiling and linking them with `splfr`. However, this is sometimes inconvenient, for instance when writing a Makefile
for use with `make`. To figure out what needs to be done to build a foreign resource, you should build it once with `splfr --verbose --keep ...`, note what compiler and linker flags are used, and save away any generated files. You can then mimic the build commands used by `splfr` in your `Makefile`. You should repeat this process each time you upgrade SICStus Prolog.

6.2.5.3 Windows-specific `splfr` issues

`splfr` needs to be able to invoke the C compiler from the command line. On Windows, this will only work if the command line environment has been properly set up. See Section 6.7.3.3 [Setting up the C compiler on Windows], page 327, for Windows-specific information about getting the C compiler to work.

6.2.6 Init and Deinit Functions

An init function and/or a deinit function can be declared by `foreign_resource/2`. If this is the case, then these functions should have the prototype:

```c
void FunctionName (int when)
```

The init function is called by `load_foreign_resource/1` after the resource has been loaded and the interfaced predicates have been installed. If the init function fails (using `SP_fail()`) or raises an exception (using `SP_raise_exception()`), then the failure or exception is propagated by `load_foreign_resource/1` and the foreign resource is unloaded (without calling any deinit function). However, using `SP_fail()` is not recommended, and operations that may require `SP_raise_exception()` are probably better done in an init function that is called explicitly after the foreign resource has been loaded.

The deinit function is called by `unload_foreign_resource/1` before the interfaced predicates have been uninstalled and the resource has been unloaded. If the deinit function fails or raises an exception, then the failure or exception is propagated by `unload_foreign_resource/1`, but the foreign resource is still unloaded. However, neither `SP_fail()` nor `SP_raise_exception()` should be called in a deinit function. Complex deinitialization should be done in an explicitly called deinit function instead.

The init and deinit functions may use the C-interface to call Prolog etc.

Foreign resources are unloaded when the saved state is restored; see Section 3.10 [Saving], page 28. Foreign resources are also unloaded when exiting Prolog execution. The parameter `when` reflects the context of the (un)load_foreign_resource/1 and is set as follows for init functions:

- `SP_WHEN_EXPLICIT`
  Explicit call to `load_foreign_resource/1`.

- `SP_WHEN_RESTORE`
  Resource is reloaded after restore.

For deinit functions:
SP_WHEN_EXPLICIT

Explicit call to unload_foreign_resource/1 or a call to load_foreign_resource/1 with the name of an already loaded resource.

SP_WHEN_EXIT

Resource is unloaded before exiting Prolog.

6.2.7 Creating the Linked Foreign Resource

Suppose we have a Prolog source file ex.pl containing:

```prolog
% ex.pl

foreign(f1, p1(+integer,[-integer])).
foreign(f2, p2(+integer,[-integer])).
foreign_resource(ex, [f1,f2]).
:- load_foreign_resource(ex).
```

and a C source file ex.c with definitions of the functions f1 and f2, both returning SP_integer and having an SP_integer as the only parameter. The conversion declarations in ex.pl state that these functions form the foreign resource ex. Normally, the C source file should contain the following two line near the beginning (modulo the resource name):

```c
#include <sicstus/sicstus.h>
/* ex_glue.h is generated by splfr from the foreign/[2,3] facts.
   Always include the glue header in your foreign resource code.
*/
#include "ex_glue.h"
```

To create the linked foreign resource, simply type (to the Shell):

```
% splfr ex.pl ex.c
```

The linked foreign resource ex.so (file suffix .so is system dependent) has been created. It will be dynamically linked by the directive :- load_foreign_resource(ex). when the file ex.pl is loaded. For a full example, see Section 6.2.8 [Foreign Code Examples], page 298.

Dynamic linking of foreign resources can also be used by runtime systems.

6.2.8 Foreign Code Examples

Given: a Prolog file ex.pl and a C file ex.c shown below.
ex.pl

foreign_resource(ex, [c1, c2, c11, c21, c3, c4, c5, c6]).

foreign(c1, c, c1(+integer, [-integer])).
foreign(c2, c, c2(-integer)).
foreign(c11, c, c11(+atom, [-atom])).
foreign(c21, c, c21(+atom, -atom)).
foreign(c3, c, c3(+float, [-float])).
foreign(c4, c, c4(-float)).
foreign(c5, c, c5(+string, [-string])).
foreign(c6, c, c6(-string)).

:- load_foreign_resource(ex).
/* ex.glue.h is generated by splfr from the foreign/[2,3] facts.
   Always include the glue header in your foreign resource code. */
#include "ex.glue.h"

/* c1(+integer, [\-integer]) */
SP_integer c1(SP_integer a)
{
   return(a+9);
}

/* c2(-integer) */
void c2(SP_integer *a)
{
   *a = 99;
}

/* c11(+atom, [-atom]) */
SP_atom c11(SP_atom a)
{
   return(a);
}

/* c21(+atom, -atom) */
void c21(SP_atom a, SP_atom *b)
{
   *b = a;
}

/* c3(+float, [-float]) */
double c3(double a)
{
   return(a+9.0);
}

/* c4(-float) */
void c4(double *a)
{
   *a = 9.9;
}

/* c5(string, [-string]) */
char const * c5(char const * a)
{
   return(a);
}

/* c6(-string) */
void c6(char const * *a)
{
   *a = "99";
}
Chapter 6: Mixing C/C++ and Prolog

Dialog at the command level:

```
% splfr ex.pl ex.c
% sicstus -l ex
% compiling /home/matsc/sicstus4/ex.pl...
% loading foreign resource /home/matsc/sicstus4/ex.so in module user
% compiled /home/matsc/sicstus4/ex.pl in module user, 0 msec 3184 bytes
SICStus 4.7.0 ...
Licensed to SICS
| ?- c1(1,X1), c2(X2), c11(foo,X11), c21(foo,X21),
  c3(1.5,X3), c4(X4), c5(foo,X5), c6(X6).
X1 = 10,
X2 = 99,
X3 = 10.5,
X4 = 9.9,
X5 = foo,
X6 = '99',
X11 = foo,
X21 = foo ? RET
yes
```

### 6.3 Calling C++ from Prolog

Functions in C++ files that should be called from Prolog must use C linkage, e.g.

```c
extern "C" {
    void myfun(SP_integer i)
    {...};
};
```

On Windows, C++ is a first class citizen and no special steps are needed in order to mix C++ and C code.

On other platforms, to build a dynamically linked foreign resource with C++ code, you may have to explicitly include certain libraries and you may need to use an executable compiled and linked with a C++ compiler. The details are platform and C++ compiler dependent and outside the scope of this manual.

### 6.4 Support Functions

The support functions include functions to manipulate SP_term_refs, functions to convert data between the basic C types and Prolog terms, functions to test whether a term can be converted to a specific C type, and functions to unify or compare two terms.

#### 6.4.1 Creating and Manipulating SP_term_refs

Normally, C functions only have indirect access to Prolog terms via SP_term_refs. C functions may receive arguments as unconverted Prolog terms, in which case the actual argu-


ments received will have the type `SP_term_ref`. Also, a C function may return an unconverted Prolog term, in which case it must create an `SP_term_ref`. Finally, any temporary Prolog terms created by C code must be handled as `SP_term_refs`.

`SP_term_refs` are motivated by the fact that SICStus Prolog's memory manager must have a means of reaching all live Prolog terms for memory management purposes, including such terms that are being manipulated by the user's C code. Previous releases provided direct access to Prolog terms and the ability to tell the memory manager that a given memory address points to a Prolog term, but this approach was too low level and highly error-prone. The current design is modeled after and largely compatible with Quintus Prolog release 3.

`SP_term_refs` are created dynamically. At any given time, an `SP_term_ref` has a value (a Prolog term, initially []). This value can be examined, accessed, and updated by the support functions described in this section.

A new `SP_term_ref` is created by calling `SP_new_term_ref()`.

An `SP_term_ref` can be assigned the value of another `SP_term_ref` by calling `SP_put_term()`.

It is important to understand the rules governing the scope of `SP_term_refs`, and the terms they hold, in conjunction with calls from Prolog to C and vice versa. This is explained in Section 6.5.2 [Finding Multiple Solutions of a Call], page 307.

### 6.4.2 Atoms in C

Each Prolog atom is represented internally by a unique integer, its canonical representation, with the corresponding C type `SP_atom`. This mapping between atoms and integers depends on the execution history. Certain functions require this representation as opposed to an `SP_term_ref`. It can be obtained by a special argument type declaration when calling C from Prolog, by calling `SP_get_atom()`, or by looking up an encoded string `s` in the Prolog symbol table by calling `SP_atom_from_string(s)` which returns the atom, or zero if the given string is malformed (is not a valid sequence of UTF-8 encoded characters).

The encoded string containing the characters of a Prolog atom `a` can be obtained by calling `SP_string_from_atom()`.

The length of the encoded string representing a Prolog atom `a` can be obtained by calling `SP_atom_length()`.

Prolog atoms, and the space occupied by their print names, are subject to garbage collection when the number of atoms has reached a certain threshold, under the control of the `agc_margin` Prolog flag, or when the atom garbage collector is called explicitly. The atom garbage collector will find all references to atoms from the Prolog specific memory areas, including `SP_term_refs` and arguments passed from Prolog to foreign language functions. However, atoms created by `SP_atom_from_string()` and merely stored in a local variable are endangered by garbage collection. The functions `SP_register_atom()` and `SP_unregister_atom()` make it possible to protect an atom while it is in use. The operations are implemented using reference counters to support multiple, independent use of the same atom in different foreign resources.
6.4.3 Creating Prolog Terms

The following functions create a term and store it as the value of an SP_term_ref, which must exist prior to the call. They return zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise, assigning to \( t \) the converted value.

- **SP_put_variable()**
  Creates a variable.

- **SP_put_integer()**
  Creates an integer.

- **SP_put_float()**
  Creates a float.

- **SP_put_atom()**
  Creates an atom.

- **SP_put_string()**
  Creates an atom.

- **SP_put_address()**
  Creates an integer representing a pointer.

- **SP_put_list_codes()**
  Creates a char list.

- **SP_put_list_n_codes()**
  Creates a char list.

- **SP_put_list_n_bytes()**
  Creates a byte list.

- **SP_put_integer_bytes()**
  Creates an arbitrarily sized integer.

- **SP_put_number_codes()**
  Creates a char list denoting a number.

- **SP_put_functor()**
  Creates a compound term.

- **SP_put_list()**
  Creates a list.

- **SP_cons_functor()**
  Creates a compound term with arguments filled in.

- **SP_cons_list()**
  Creates a list with arguments filled in.

- **SP_read_from_string()** (C function)
  Reads a term from its textual representation, replacing variables by specified terms.
6.4.4 Accessing Prolog Terms

The following functions will take an SP_term_ref and convert it to C data. They return zero if the conversion fails, and a nonzero value otherwise, and store the C data in output arguments, except the last two, which merely decompose compound terms.

`SP_get_integer()`  
Accesses an integer.

`SP_get_float()`  
Accesses a float.

`SP_get_atom()`  
Accesses an atom.

`SP_get_string()`  
Accesses an atom.

`SP_get_address()`  
Accesses an integer representing a pointer.

`SP_get_list_codes()`  
Accesses a code list.

`SP_get_list_n_codes()`  
Accesses a code list.

`SP_get_list_n_bytes()`  
Accesses a byte list.

`SP_get_number_codes()`  
Accesses a code list denoting a number.

`SP_get_integer_bytes()`  
Accesses an arbitrarily sized integer.

`SP_get_functor()`  
Accesses a compound term.

`SP_get_list()`  
Accesses a list.

`SP_get_arg()`  
Accesses an argument of a compound term.

6.4.5 Testing Prolog Terms

There is one general function for type testing of Prolog terms as well as a set of specialized, more efficient, functions—one for each term type:

`SP_term_type()`  
Accesses term type.

`SP_is_variable()`  
Checks whether term is a variable.
SP\_is\_integer()
Checks whether term is an integer.

SP\_is\_float()
Checks whether term is a float.

SP\_is\_atom()
Checks whether term is an atom.

SP\_is\_compound()
Checks whether term is compound.

SP\_is\_list()
Checks whether term is a list cell.

SP\_is\_atomic()
Checks whether term is atomic.

SP\_is\_number()
Checks whether term is a number.

### 6.4.6 Unifying and Comparing Terms
The two functions are:

SP\_unify()
Unify terms.

SP\_compare()
Compare terms.

### 6.4.7 Operating System Services
#### 6.4.7.1 Memory Management
The standard C library memory allocation functions (malloc, calloc, realloc, and free) are available in foreign code, but cannot reuse any free memory that SICStus Prolog's memory manager may have available, and so may contribute to memory fragmentation.

The following functions provide the same services via SICStus Prolog's memory manager.

SP\_malloc()
Allocates a piece of memory.

SP\_calloc()
Allocates memory for an array of elements, and clears the allocated memory.

SP\_realloc()
Changes the size of an allocated piece of memory.

SP\_free()
Deallocates a piece of memory.

SP\_strdup()
Makes a copy of a string in allocated memory.
6.4.7.2 File System

SICStus Prolog caches the name of the current working directory. To take advantage of the cache and to keep it consistent, foreign code should call the following interface functions instead of calling chdir() and getcwd() directly:

- **SP_set_current_dir()**: Obtains the absolute name of the current working directory.
- **SP_get_current_dir()**: Sets the current working directory.

6.4.7.3 Threads

When running more that one SICStus runtime in the same process it is often necessary to protect data with mutual exclusion locks. The following functions implement recursive mutual exclusion locks, which only need static initialization.

- **SP_mutex_lock()**: Locks the mutex.
- **SP_mutex_unlock()**: Unlocks the mutex.

A (recursive) mutual exclusion lock is declared as type `SP_mutex`. It should be initialized to (the static initializer) `SP_MUTEX_INITIALIZER` before use.

Note that the SICStus runtime is not thread safe in general.

A dynamic foreign resource that is used by multiple SICStus runtimes in the same process may need to maintain a global state that is kept separate for each SICStus runtime. Each SICStus runtime maintains a location (containing a `void*`) for each foreign resource. By calling `SP_foreign_stash()`, a foreign resource can then access this location to store any data that is specific to the calling SICStus runtime.

6.5 Calling Prolog from C

In development and runtime systems alike, Prolog and C code may call each other to arbitrary depths.

Before calling a predicate from C you must look up the predicate definition by module, name, and arity. The function `SP_predicate()` will return a pointer to this definition or return NULL if the predicate is not visible in the module. This definition can be used in more than one call to the same predicate.

The function `SP_pred()` may be used as an alternative to the above. The only difference is that the name and module arguments are passed as Prolog atoms rather than strings, and the module argument is mandatory. This saves the cost of looking up the two arguments in the Prolog symbol table. This cost dominates the cost of the operation.
6.5.1 Finding One Solution of a Call

The easiest way to call a predicate if you are only interested in the first solution is to call the function `SP_query()`. It will create a goal from the predicate definition and the arguments, call it, and commit to the first solution found, if any.

If you are only interested in the side effects of a predicate, then you can call `SP_query_cut_fail()`. It will try to prove the predicate, cut away the rest of the solutions, and finally fail. This will reclaim any memory used after the call, and throw away any solution found.

6.5.2 Finding Multiple Solutions of a Call

If you are interested in more than one solution, then a more complicated scheme is used. You find the predicate definition as above, but you do not call the predicate directly.

1. Set up a call with `SP_open_query()`
2. Call `SP_next_solution()` to find a solution. Call this predicate again to find more solutions if there are any.
3. Terminate the call with `SP_close_query()` or `SP_cut_query()`

The function `SP_open_query()` will return an identifier of type `SP_qid` that you use in successive calls. Note that if a new query is opened while another is already open, then the new query must be terminated before exploring the solutions of the old one. That is, queries must be strictly nested.

The function `SP_next_solution()` will cause the Prolog engine to backtrack over any current solution of an open query and look for a new one.

A query must be terminated in either of two ways. The function `SP_cut_query()` will discard the choices created since the corresponding `SP_open_query()`, like the goal `!`. The current solution is retained in the arguments until backtracking into any enclosing query.

Alternatively, the function `SP_close_query()` will discard the choices created since the corresponding `SP_open_query()`, and then backtrack into the query, throwing away any current solution, like the goal `!, fail`.

A simple way to call arbitrary Prolog code, whether for one solution or for multiple solutions, is to use `SP_read_from_string()` (see Section 6.4.3 [Creating Prolog Terms], page 303) to create an argument to `call/1`. It is a good idea to always explicitly specify the module context when using `call/1` or other meta-predicates from C.

It is important to understand the rules governing the scope of `SP_termrefs`, and the terms they hold, in conjunction with calls from Prolog to C and vice versa. `SP_termrefs` are internally stored on a stack, which is manipulated by the various API functions as follows:

`SP_new_term_ref()`

The new `SP_term_ref` is pushed onto the stack.
**calling C from Prolog**

- **SP_query()**
  - The top of the stack is saved on call and restored upon return.

- **SP_query_cut_fail()**
  - The top of the stack is saved on call and restored upon return.

- **SP_open_query()**
  - The top of the stack is saved in the new query.

- **SP_close_query()**

- **SP_cut_query()**

- **SP_next_solution()**
  - The top of the stack is restored from the query argument.

Among other things, this means that an SP_term_ref cannot be saved across multiple calls from Prolog to C. Thus it makes no sense to declare an SP_term_ref as a static C variable.

Prolog terms are also generally stored on a stack, which keeps growing until the execution backtracks, either spontaneously or by calling **SP_close_query()** or **SP_next_solution()**. It is an abuse of the **SP_open_query()** API to assign a term to an SP_term_ref, and then backtrack over the term while the SP_term_ref is still live. Such abuse results in a dangling pointer that can potentially crash SICStus Prolog. The API typically follows the pattern:

```c
...  
SP_pred_ref pred = SP_predicate(...);
SP_term_ref ref1 = SP_new_term_ref();
SP_qid goal = SP_open_query(pred,ref1,...);
/*
 * PART A: perform some initializations, and
 * loop through all solutions.
 */
while (SP_next_solution(goal)==SP_SUCCESS) {
  /*
   * PART B: perform some action on the current solution.
  */
}
SP_close_query(goal);
...  
```

In order to avoid dangling pointer hazards, we recommend some simple coding rules:

**PART A**  
In this part of the code, do not call **SP_new_term_ref()** or the functions in Section 6.4.3 [Creating Prolog Terms], page 303, at all.

**PART B**  
In this part of the code, do not call **SP_new_term_ref()** except to initialize any SP_term_refs declared locally to Part B. Do Not call the functions in Section 6.4.3 [Creating Prolog Terms], page 303, except to set SP_term_refs declared locally to Part B.
6.5.3 Backtracking Loops

If you want to call Prolog multiple times in a loop for side effect, for example over the elements of a list, then some care is required in order not to cause a memory leak by creating more and more SP_term.refs. The recommended coding scheme is to use a backtracking loop (see Section 9.4.4 [Terminating a Backtracking Loop], page 359). For example, suppose that you want the C equivalent of the following code:

```c
process_list(L) :-
    member(X, L),
    once(process(X)),
    fail.
process_list(_).
process(X) :- ...
```

That can be encoded as follows, where refL is the SP_term_ref that holds L:

```c
...  
SP_qid goal;
SP_pred_ref member2 = SP_predicate("member", 2, "user");
SP_pred_ref process1 = SP_predicate("process", 1, "user");
SP_term_ref refX = SP_new_term_ref();
SP_put_variable(refX);
goal = SP_open_query(member2, refX, refL);
while (SP_next_solution(goal)==SP_SUCCESS)
    SP_query_cut_fail(process1, refX);
SP_close_query(goal);
...  
```

This programming style is particularly relevant in a stand-alone executable, where the top level iterates over some transactions to be processed.

6.5.4 Calling Prolog Asynchronously

If you wish to call Prolog back from a signal handler or a thread other than the thread that called SP_initialize(), that is, the main thread, then you cannot use SP_query() etc. directly. The call to Prolog has to be delayed until such time that the Prolog execution can accept an interrupt and the call has to be performed from the main thread (the Prolog execution thread). The function SP_event() serves this purpose, and installs the function func to be called from Prolog (in the main thread) when the execution can accept a callback.

A queue of functions, with corresponding arguments, is maintained; that is, if several calls to SP_event() occur before Prolog can accept an interrupt, then the functions are queued and executed in turn at the next possible opportunity. A func installed with SP_event() will not be called until SICStus is actually running. One way of ensuring that all pending functions installed with SP_event() are run is to call, from the main thread, some dummy goal, such as, SP_query_cut_fail(SP_predicate("true",0,"user")).
While `SP_event()` is safe to call from any thread, it is not safe to call from arbitrary signal handlers. If you want to call `SP_event()` when a signal is delivered, then you need to install your signal handler with `SP_signal()` (see below).

Note that `SP_event()` is one of the very few functions in the SICStus API that can safely be called from another thread than the main thread.

### 6.5.4.1 Signal Handling

As noted above it is not possible to call e.g. `SP_query()` or even `SP_event()` from an arbitrary signal handler. That is, from signal handlers installed with `signal` or `sigaction`. Instead you need to install the signal handler using `SP_signal()`.

When the OS delivers a signal `sig` for which `SP_signal(sig,func,user_data)` has been called SICStus will not call `func` immediately. Instead the call to `func` will be delayed until it is safe for Prolog to do so, in much the same way that functions installed by `SP_event()` are handled (this is an incompatible change as of release 3.9).

Since the signal handling function `func` will not be called immediately upon delivery of the signal to the process it only makes sense to use `SP_signal()` to handle certain asynchronous signals such as `SIGINT`, `SIGUSR1`, `SIGUSR2`. Other asynchronous signals handled specially by the OS, such as `SIGCHLD` are not suitable for handling via `SP_signal()`. Note that the development system installs a handler for ‘`SIGINT`’, and, under Windows, ‘`SIGBREAK`’, to catch keyboard interrupts. As of release 4.4, `library(timeout)` no longer uses any signals.

When `func` is called, it cannot call any SICStus API functions except `SP_event()`. Note that `func` will be called in the main thread.

### 6.5.5 Exception Handling in C

When an exception has been raised, the functions `SP_query()`, `SP_query_cut_fail()` and `SP_next_solution()` return `SP_ERROR`. To access the exception term (the argument of the call to `raise_exception/1`), which is asserted when the exception is raised, the function `SP_exception_term()` is used. As a side effect, the exception term is retracted, so if your code wants to pass the exception term back to Prolog, then use `SP_raise_exception()`.

To raise an exception from a C function called from Prolog, just call `SP_raise_exception()`. Upon return, Prolog will detect that an exception has been raised, any value returned from the function will be ignored, and the exception will be passed back to Prolog. Please note: this should only be called right before returning to Prolog.

To propagate failure to Prolog, call `SP_fail()`. Upon return, Prolog will backtrack. Please note: this should only be called right before returning to Prolog.

Prolog error handling is mostly done by raising and catching exceptions. However, some faults are of a nature such that when they occur, the internal program state may be corrupted, and it is not safe to merely raise an exception. In runtime systems, the C macro `SP_on_fault()` provides an environment for handling faults.
The function `SP_raise_fault()` can be used to raise a fault with an encoded string explaining the reason.

### 6.5.6 Reading a goal from a string

A simple way to call arbitrary Prolog code is to use `SP_read_from_string()` (see Section 6.4.3 [Creating Prolog Terms], page 303) to create an argument to `call/1`. It is a good idea to always explicitly specify the module context when using `call/1` or other meta-predicates from C.

This example calls a compound goal (without error checking):

```c
SP_pred_ref call_pred = SP_predicate("call", 1, "prolog");
SP_term_ref x = SP_new_term_ref();
SP_term_ref goal = SP_new_term_ref();
SP_term_ref vals[] = {x, 0 /* zero termination */};
SP_integer len;

// The X_= is a trick to ensure that X is the first variable
// in the depth-first order and thus corresponds to vals[0] (x).
// There are no entries in vals for _,L1,L2.
SP_put_variable(x);
SP_read_from_string(goal, 
    "user:(X=_, length([0,1,2],L1), length([3,4],L2), X is L1+L2).", vals);

SP_query(call_pred, goal);
SP_get_integer(x, &len);
// here len is 5 */
```

### 6.6 SICStus Streams

With the SICStus Prolog C interface, the user can define his/her own streams as well as from C read or write on the predefined streams. The stream interface provides:

- C functions to perform I/O on Prolog streams. This way you can use the same stream from Prolog and C code.
- User defined streams. You can define your own Prolog streams in C.
- Bidirectional streams. A SICStus stream supports reading or writing or both.
- Hookable standard input/output/error streams.

#### 6.6.1 Prolog Streams

From the Prolog level there is a unique number that identifies a stream. This identifier can be converted from/to a Prolog stream:
stream_code(?Stream, ?StreamCode)

StreamCode is the C stream identifier (an integer) corresponding to the Prolog stream Stream. This predicate is only useful when streams are passed between Prolog and C. See Section 11.3.218 [mpg-ref-stream_code], page 1205.

The StreamCode is a Prolog integer representing an SP_stream * pointer.

To read or write on a Prolog stream from C, the following functions and macros can be used:

SP_get_byte()
Read one byte from a binary stream.

SP_get_code()
Read one character code from a text stream.

SP_put_byte()
Write one byte to a binary stream.

SP_put_code()
Write one character code to a text stream.

SP_put_bytes()
Write multiple bytes to a binary stream.

SP_put_codes()
Write multiple character codes to a text stream.

SP_put_encoded_string()
Write a NUL terminated encoded string to a text stream.

SP_printf()

SP_fprintf()
Perform formatted output.

SP_flush_output()
Flush buffered data of an output stream.

SP_fclose()
Close a stream.

The following predefined streams are accessible from C:

SP_stdin Standard input. Refers to the same stream as user_input in Prolog. Which stream is referenced by user_input is controlled by the Prolog flag user_input.

SP_stdout Standard output. Refers to the same stream as user_output in Prolog. Which stream is referenced by user_output is controlled by the Prolog flag user_output.

SP_stderr Standard error. Refers to the same stream as user_error in Prolog. Which stream is referenced by user_error is controlled by the flag user_error.
SP_curin  Current input. It is initially set equal to SP_stdin. It can be changed with the predicates see/1 and set_input/1.

SP_curout  Current output. It is initially set equal to SP_stdout. It can be changed with the predicates tell/1 and set_output/1.

Note that these variables are read only.

6.6.2 Defining a New Stream

The following steps are required to define a new stream in C:

- Define low level functions (byte or character reading, writing etc).
- Initialize and open your stream.
- Allocate memory needed for your particular stream.
- Initialize and install a Prolog stream with SP_create_stream().

The following sample makes it possible to create read-only binary streams that use the C FILE* API.
```c
#include <sicstus/sicstus.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

struct stdio_t_stream {
    FILE *f;
};
typedef struct stdio_t_stream stdio_t_stream;

static spio_t_error_code SPCDECL stdio_read(void *user_data,
    void *buf,
    size_t *pbuf_size,
    spio_t_bits read_options)
{
    spio_t_error_code ecode = SPIO_E_ERROR;
    stdio_t_stream *s;
    size_t res;

    if (read_options & SPIO_DEVICE_READ_OPTION_NONBLOCKING) {
        ecode = SPIO_E_NOT_SUPPORTED;
        goto barf;
    }

    s = (stdio_t_stream *)user_data;

    res = fread(buf, 1, *pbuf_size, s->f);
    if (res == 0) {
        /* error */
        if (feof(s->f)) {
            ecode = SPIO_E_END_OF_FILE;
        } else {
            /* some other error */
            ecode = SPIO_E_OS_ERROR;
        }
        goto barf;
    }
    *pbuf_size = res; /* number of bytes read */

    return SPIO_S_NOERR;

barf:
    return ecode;
}
static spio_t_error_code SPCDECL stdio_close(void **puser_data, spio_t_bits close_options) {
    spio_t_stream *s;

    s = (stdio_t_stream *)puser_data;
    /* we can ignore SPIODEVICE_CLOSEOPTION_FORCE */

    if (close_options & SPIO_DEVICE_CLOSE_OPTION_READ) {
        *puser_data = NULL;  /* tell caller we are gone */
        if (fclose(s->f) != 0) {
            ;  /* ignore errors */
        }
    }
    return SPIO_S_NOERR;
}
/* Identify our streams with (an arbitrary) pointer that is unique to us */
#define STDIO_STREAM_CLASS ((void*)&stdio_open_c)

int stdio_open_c(char const *path,
        char const *direction,
        SP_stream **pstream)
{
    spio_t_error_code ecode = SPIO_E_ERROR;
    stdio_t_stream *s = NULL;
    SP_stream *stream = NULL;

    if (strcmp(direction, "read") != 0) goto not_supported;

    /* read */
    s = (stdio_t_stream*)SP_malloc(sizeof *s);
    if (s == NULL) goto out_of_memory;

    /* open binary */
    s->f = fopen(path, "rb");
    if (s->f == NULL) {
        ecode = SPIO_E_OPEN_ERROR;
        goto barf;
    }
    ecode = SP_create_stream((void*)s,
            STDIO_STREAM_CLASS,
            stdio_read,
            NULL, /* write */ NULL, /* flush_output */ NULL, /* seek */
            stdio_close,
            NULL, /* interrupt */ NULL, /* ioctl */ NULL, /* args */
            SP_CREATE_STREAM_OPTION_BINARY,
            &stream);
    if (SPIO_FAILED(ecode)) goto barf;

    *pstream = stream;
    return 0; /* success */

    barf:
    if (s != NULL) {
        if (s->f != NULL) fclose(s->f);
        SP_free(s);
    }
    return ecode;

    out_of_memory:
    ecode = SPIO_E_OUT_OF_MEMORY;
    goto barf;

    not_supported:
    ecode = SPIO_E_NOT_IMPLEMENTED;
    goto barf;
}
Calling `stdio_open_c("foo", "read", &stream)` will open the file `foo` as binary stream that can be read by all SICStus stream operations.

There are several stream implementations in the SICStus Prolog library that can serve as sample, e.g. `library(codesio)` and `library(tcltk)`.

See Section 12.3.9 [cpg-ref-SP_create_stream], page 1270, for details.

### 6.6.2.1 Low Level I/O Functions

For each new stream the appropriate low level I/O functions have to be defined. Error handling, prompt handling and character counting is handled in a layer above these functions. They all operate on a user defined private data structure specified when the stream is created.

- `user_read()`
  Should fill a buffer with data available from the stream. See Section 12.3.106 [cpg-ref-user_read], page 1384.

- `user_write()`
  Should write data from a buffer to the stream. See Section 12.3.107 [cpg-ref-user_write], page 1386.

- `user_flush_output()`
  Should flush the (output) stream.

- `user_close()`
  Should close the stream in the specified directions. Note that bi-directional streams can be closed one direction at a time.

**Please note:** A foreign resource that defines user defined streams must ensure that all its streams are closed when the foreign resource is unloaded. Failure to do this will lead to crashes when SICStus tries to close the stream using a `user_close` method that is no longer present.

The easiest way to ensure that all user defined streams of a particular class is closed is to use `SP_fclose` with the `SP_FCLOSE_OPTION_USER_STREAMS`. Another way is to use `SP_next_stream` and `SP_get_stream_user_data` to find all your streams and close them one by one. See Section 12.3.18 [cpg-ref-SP_fclose], page 1284, Section 12.3.60 [cpg-ref-SP_next_stream], page 1333, and Section 12.3.41 [cpg-ref-SP_get_stream_user_data], page 1312.

### 6.6.3 Hookable Standard Streams

The standard I/O streams (input, output, and error) are hookable, i.e. the streams can be redefined by the user by calling `SP_set_user_stream_hook()` and/or `SP_set_user_stream_post_hook()`. These hook functions must be called before `SP_initialize()` (see Section 6.7.4.1 [Initializing the Prolog Engine], page 330). In custom built systems, they may be called in the hook function `SU_initialize()`. See Section 6.7.3 [The Application Builder], page 323.
6.6.3.1 Writing User-stream Hooks

The user-stream hook is, if defined, called during `SP_initialize()`. It has the following prototype:

```c
SP_stream *user_stream_hook(void *user_data, int which)
```

If the hook is not defined, then SICStus will attempt to open the standard TTY/console versions of these streams. If they are unavailable (such as for non-console executables under Windows), then the result is undefined.

It is called once for each stream. The `which` argument indicates which stream it is called for. The value of `which` is one of the following:

- `SP_STREAMHOOK_STDIN` Create stream for standard input.
- `SP_STREAMHOOK_STDOUT` Create stream for standard output.
- `SP_STREAMHOOK_STDERR` Create stream for standard error.

The set of possible values for `which` may be expanded in the future.

The hook should return a standard SICStus I/O stream, as described in Section 6.6.2 [Defining a New Stream], page 313.

See Section 12.3.93 [cpg-ref-SP_set_user_stream_hook], page 1368, for details.

6.6.3.2 Writing User-stream Post-hooks

If defined, then the user-stream post-hook is called after all the streams have been defined, once for each of the standard streams. It has a slightly different prototype:

```c
void user_stream_post_hook(void *user_data, int which, SP_stream *str)
```

where `str` is a pointer to the corresponding `SP_stream` structure. There are no requirements as to what this hook must do; the default behavior is to do nothing at all.

The post-hook is intended to be used to do things that may require that all streams have been created.

See Section 12.3.94 [cpg-ref-SP_set_user_stream_post_hook], page 1369, for details.

6.7 Stand-Alone Executables

So far, we have only discussed foreign code as pieces of code loaded into a Prolog executable. This is often not the desired situation. Instead, one often wants to create stand-alone executables, i.e. an application where Prolog is used as a component, accessed through the API described in the previous sections.
6.7.1 Runtime Systems

Stand-alone applications containing debugged Prolog code and destined for end-users are typically packaged as runtime systems. No SICStus license is needed by a runtime system. A runtime system has the following limitations:

- No top level. The executable will restore a saved state and/or load code, and call `user:runtime_entry(start)`. Alternatively, you may supply a main program and explicitly initialize the Prolog engine with `SP_initialize()`. `break/0` and `require/1` are unavailable.
- No debugger. `debugging`, `debug` and `debugger_print_options` have no effect. Predicates annotated as `[development]` in the reference pages are unavailable.
- Except in extended runtime systems: no compiler; compiling is replaced by consulting. Extended runtime systems do provide the compiler.
- The `discontiguous_warnings`, `single_var_warnings`, `redefine_warnings`, and `informational` Prolog flags are off by default, suppressing warnings about clauses not being together, singleton variables, queries and warnings about name clashes and redefinitions, and informational messages. Note that they can be switched on though, to enable such warnings, queries and messages.
- No profiler or coverage analysis. The predicates `profile_reset/0`, `profile_data/1`, `print_profile/[0,1]`, `coverage_data/1`, and `print_coverage/[0,1]` are unavailable. The Prolog flag `profiling` is unavailable.
- No signal handling except as installed by `SP_signal()`.

It is possible to tell a runtime system to start a development system instead, for debugging purposes. See Section 6.9 [Debugging Runtime Systems], page 342, for details.

6.7.2 Runtime Systems on Target Machines

When a runtime system is delivered to the end user, chances are that the user does not have an existing SICStus installation. To deliver such an executable, you need:

- the executable
  This is your executable program, usually created by `spld` (see Section 6.7.3 [The Application Builder], page 323).

- the runtime kernel
  This is a shared object or a DLL, usually `libsprt4-7-0.so` under UNIX, or `sprt4-7-0.dll` under Windows.

- the (extended) runtime library
  The saved state `sprt.sav` contains the built-in predicates written in Prolog. It is restored into the program at runtime by the function `SP_initialize()`. Extended runtime systems restore `spre.sav` instead, which requires a license, available from SICS as an add-on product. See also Section “Managing Extended Runtime License Information” in SICStus Prolog Release Notes.
your Prolog code
As a saved state, ‘.po’ files, or source code (‘.pl’ files). They must be explicitly loaded by the program at runtime (see Section 6.7.4.2 [Loading Prolog Code], page 331).

your linked foreign resources
Any dynamically linked foreign resources, including any linked foreign resources for library modules located in $SP\_PATH/library.

The following two sections describe how to package the above components for UNIX and Windows target machines, i.e. machines that do not have SICStus Prolog installed, respectively. It is also possible to package all the above components into a single executable file, an all-in-one executable. See Section 6.7.3.2 [All-in-one Executables], page 324.

6.7.2.1 Runtime Systems on UNIX Target Machines
In order to build a runtime system for distribution on a target machine, the option --moveable must be passed to spld. This option prevents spld from hardcoding any (absolute) paths into the executable. As of release 4.2, --moveable is the default on most platforms, including Linux, Mac OS X and Solaris.

Next, in order for SICStus to be able to locate all relevant files, the following directory structure should be used.

    myapp.exe
    sp-4.7.0/
    +---- libsprt4-7-0.so
    +---- sicstus-4.7.0/
        +---- bin/
        |   +---- sprt.sav
        +---- library/
            +---- <files from $SP\_PATH/library>

If support for multiple SICStus instances is needed, then the runtimes named e.g. libsprt4-7-0_instance_01_.so need to be available as well, in the same place as libsprt4-7-0.so.

If SICStus Prolog is installed on the target machine, then a symbolic link named sp-4.7.0 can be used, in which case it should point to the directory of the SICStus installation that contains the libsprt4-7-0.so (or equivalent).

If the runtime system needs to be debugged, then the above file system layout should be complemented as follows: The file spds.sav from the development system should be copied and placed in the same folder as sprt.sav and the license information must be made available. See Section 6.9 [Debugging Runtime Systems], page 342, for details.

myapp.exe is typically created by a call to spld:

    % spld --main=user --moveable [...] -o ./myapp.exe
On most platforms, the above directory layout will enable the executable to find the SICStus runtime (e.g., `libsprt4-7-0.so`) as well as the boot file `sprt.sav` (or `spre.sav`). In addition, application specific files, e.g. a `.sav` file, can be found using the automatically set system properties `SP_APP_DIR` or `SP_RT_DIR`. On some platforms a wrapper script, generated by `spld`, is needed to ensure that the files are found.

Unless the `--static` option is passed to `spld`, it might also be necessary to set `LD_LIBRARY_PATH` (or equivalent) to `/home/joe/lib` (in the example above) in order for the dynamic linker to find `libsprt4-7-0.so`. If the `--static` option is used, then this is not necessary. Setting `LD_LIBRARY_PATH` is not recommended unless it is really needed.

When a runtime system is redistributed to third parties, only the following files may be included in the distribution. All filenames are relative to `<prefix>/lib/sicstus-4.7.0`:

```
../*.{a,so,sl,dylib}
bin/sprt.sav
bin/spre.sav
bin/jasper.jar
bin/prologbeans.jar
bin/sprt (generic runtime system)
bin/sprti (generic runtime system)
library/*/.{tcl,po,pl}
   Except license.pl!
library/*/.*.{s.o,so,sl,dylib}
library/*/.*.{po,pl}
sp_platform
   (Located with InstallSICStus)
```

Please note: you cannot redistribute `spds.sav` or `license.pl`.

### 6.7.2.2 Runtime Systems on Windows Target Machines

In order to locate all relevant files, the following directory structure should be used:

```
myapp.exe
sprt4-7-0.dll
sp-4.7.0\
   \--- bin\
   |   \--- sprt.sav
   \--- library\
      \--- <files from %SP_PATH%\library>
```

If support for multiple SICStus instances is needed, then the runtimes named e.g. `sprt4-7-0_instance_01..dll` need to be available as well, in the same place as `sprt4-7-0.dll`.

If the runtime system needs to be debugged, then the above file system layout should be complemented as follows: The file `spds.sav` from the development system should be copied
and placed in the same folder as `sprt.sav` and the license information must be made available. See Section 6.9 [Debugging Runtime Systems], page 342, for details.

`myapp.exe` is typically created by a call to `spld`:

```bash
% spld --main=user [...] -o ./myapp.exe
```

If the directory containing `sprt4-7-0.dll` contains a directory called `sp-4.7.0`, then SICStus assumes that it is part of a runtime system as described in the picture below. The (extended) runtime library, `sprt.sav (spre.sav)`, is then looked up in the directory (`sp-4.7.0/bin`), as in the picture. Furthermore, the initial `library_directory/1` fact will be set to the same directory with `sp-4.7.0/library` appended.

The directory structure under `library/` should look like in a regularly installed SICStus, including the platform-specific subdirectory (`x86-win32-nt-4` in this case). If your application needs to use `library(timeout)1` and `library(random)`, then your directory structure may look like:

```bash
myapp.exe
sprt4-7-0.dll
sp-4.7.0/
+--- bin/
  |   +--- sprt.sav
+--- library/
  +--- random.po
  +--- timeout.po
  +--- x86-win32-nt-4 
    +--- random.dll
```

The `sp*` files can also be put somewhere else in order to be shared by several applications provided the `sprt4-7-0.dll` can be located by the DLL search.

Naming the files with version number enables applications using different SICStus versions to install the `sp*` files in the same directory.

When a runtime system is redistributed to third parties, only the following files may be included in the distribution. All filenames are relative to `%SP_PATH%`:

```bash
bin\sprt.sav
bin\spre.sav
bin\jasper.jar
bin\prologbeans.jar
bin\sprt.exe
  (generic runtime system)
bin\sprti.exe
  (generic runtime system)
```

---

1 Prior to release 4.4, `library(timeout)` also used a foreign resource. This is no longer the case.
bin/sprtw.exe
   (generic runtime system)
bin\*.dll
bin\*.po
library\*.{tcl,po,pl,bas}
   Except license.pl!
library\*\*.dll
library\*\*.{po,pl}

Please note: you cannot redistribute spds.sav or license.pl.

6.7.3 The Application Builder
The application builder, spld, is used for creating stand-alone executables. spld takes the
files specified on the command line and combines them into an executable file, much like
the UNIX ld or the Windows link commands.

Note that no pathnames passed to spld should contain spaces. Under Windows, this can
be avoided by using the short version of pathnames as necessary.

See Section 13.5 [too-spld], page 1395, for detailed information about spld options etc.

6.7.3.1 Customizing spld
The spld tool reads a configuration file at start-up that contains default values for many
configurable parameters. It is sometimes useful to modify these in order to adapt to local
variations.

The following methods can be used also with the splfr command, Section 6.2.5 [The Foreign
Resource Linker], page 296.

There are two methods

• Override some parameters with --conf VAR=VALUE.
   This is useful when only a few parameters need to be changed, e.g. the C compiler.
   You can override multiple parameters by specified --conf more than once.
   For instance, to use a non-default C compiler you can pass --conf
   CC=/home/joe/bin/mycc.
   The option --conf was introduced in release 4.0.3.

• Use a modified configuration file with --config=File.
   It may sometimes be convenient to use a separate, possibly modified, configuration file.
   This should seldom be needed, use --conf instead.
   To use a modified configuration file, follow these instructions:
   1. Locate the configuration file spconfig-version. It should be located in the same
directory as spld.
   2. Make a copy for spconfig-version; let us call it hacked_spld.config. Do Not
      edit the original file.
3. The configuration file contains lines on the form `CFLAGS=-g -O2`. Edit these according to your needs. Do Not add or remove any options.

4. You may now use the modified `spconfig-version` together with `spld` like this:
   ```
   % spld [...] --config=/path/to/hacked_spld.config
   ```

5. Replace `/path/to` with the actual path to the hacked configuration file.

### 6.7.3.2 All-in-one Executables

It is possible to embed saved states into an executable (or shared object). Together with static linking, this gives an all-in-one executable, an executable (or shared object) that does not depend on external SICStus files.

In the simplest case, creating an all-in-one executable `main.exe` from a saved state `main.sav` can be done with a command like:

```
% spld --output=main.exe --static main.sav
```

This will automatically embed the saved state, any foreign resources needed by the saved state as well the SICStus runtime and its runtime saved state.

Creating a shared object is similar, e.g.

```
% spld --output=main.dll --shared --static main.sav
```

but the following examples cover the more common case of creating an ordinary executable.

The keys to this feature are:

- Static linking. By linking an application with a static version of the SICStus runtime, you avoid any dependency on e.g. `sprt4-7-0.dll` (Windows) or `libsprt4-7-0.so` (UNIX).

  If the application needs foreign resources (predicates written in C code), as used for example by `library(random)` and `library(clpfd)`, then these foreign resources can be linked statically with the application as well.

  The remaining component is the Prolog code itself; see the next item.

- Data Resources (in-memory files). It is possible to link an application with data resources that can be read directly from memory. In particular, saved states can be embedded in an application and used when restoring the saved state of the application.

An application needs two saved states:

1. The SICStus runtime system (`sprt.sav`).
   
   This is added automatically when `spld` is invoked with the `--static` (or `-S`) option unless the `spld-option --no-embed-rt-sav` is specified. It can also be added explicitly with the option `--embed-rt-sav`.

2. The user written code of the application as well as any SICStus libraries.

   This saved state is typically created by loading all application code using `compile/1` and then creating the saved state with `save_program/2`. 
Data resources are added by specifying their internal name and the path to a file as part of the comma separated list of resources passed with the `spld` option `--resources`. Each data resource is specified as `file=name` where `file` is the path to the file containing the data (it must exist during the call to `spld`) and `name` is the name used to access the content of `file` during runtime. A typical choice of `name` would be the base name, i.e. without directories, of `file`, preceded by a slash (`/`). `name` should begin with a slash (`/`) and look like an ordinary lowercase file path made up of `'/'-separated, non-empty, names consisting of 'a' to 'z', underscore ('_'), period ('.'), and digits.

Typically, you would use `spld --main=restore`, which will automatically restore the first `.sav` argument. To manually restore an embedded saved state you should use the syntax `URL:x-sicstus-resource:name`, e.g. `SP_restore("URL:x-sicstus-resource:/main.sav")`.

An example will make this clearer. Suppose we create a runtime system that consists of a single file `main.pl` that looks like:

```prolog
% main.pl
:- use_module(library(random)).
:- use_module(library(clpfd)).

% This will be called when the application starts:
user:runtime_entry(start) :-
   %% You may consider putting some other code here...
   write('hello world'),nl,
   write('Getting a random value:'),nl,
   random(1,10,R), % from random
   write(R),nl,
   ( all_different([3,9]) -> % from clpfd
     write('3 != 9'),nl
   ; otherwise ->
     write('3 = 9!?'),nl
   ).
```

Then create the saved state `main.sav`, which will contain the compiled code of `main.pl` as well as the Prolog code of `library(random)` and `library(clpfd)` and other Prolog libraries needed by `library(clpfd)`: 
Finally, tell `spld` to build an executable statically linked with the SICStus runtime and the foreign resources needed by `library(random)` and `library(clpfd)`. Also, embed the Prolog runtime saved state and the application specific saved state just created.

As noted above, it is possible to build the all-in-one executable with the command line:

```
% spld --output=main.exe --static main.sav
```

but for completeness the example below uses all options as if no options were added automatically.

The example is using Cygwin bash (http://www.cygwin.com) under Windows but would look much the same on other platforms. The command should be given on a single line; it is broken up here for better layout:

```
% spld
   --output=main.exe
   --static
   --embed-rt-sav
   --main=restore
   --resources=main.sav=/main.sav,clpfd,random
```

The arguments in the example are as follows:

```
--output=main.exe
  This tells `spld` where to put the resulting executable.

--static
  Link statically with the SICStus runtime and foreign resources (clpfd and random, in this case).

--embed-rt-sav
  This option embeds the SICStus runtime `.sav` file (sprt.sav). This option is not needed since it is added automatically by `--static`.

--main=restore
  Start the application by restoring the saved state and calling `user:runtime_entry(start)`. This is not strictly needed in the above example since it is the
default if any file with extension '.sav' or a data resource with a name where the extension is '.sav' is specified.

```
--resources=...
```

This is followed by comma-separated resource specifications:

```
main.sav=/main.sav
```

This tells spld to make the content (at the time spld is invoked) of the file main.sav available at runtime in a data resource named /main.sav. That is, the data resource name corresponding to "URL:x-sicstus-resource:/main.sav".

Alternatively, spld can create a default data resource specification when passed a '.sav' file argument and the option --embed-sav-file (which is the default with --static).

```
clpfd
random
```

These tell spld to link with the foreign resources (that is, C-code) associated with library(clpfd) and library(random). Since --static was specified the static versions of these foreign resources will be used.

Alternatively, spld can extract the information about the required foreign resources from the saved state (main.sav). This feature is enabled by adding the option --resources-from-sav (which is the default with --static). Using --resources-from-sav instead of an explicit list of foreign resources is preferred since it is hard to know what foreign resources are used by the SICStus libraries.

Since both --embed-sav-file and --resources-from-sav are the default when --static is used the example can be built simply by doing:

```
% spld --output=main.exe --static main.sav
```

Finally, we may run this executable on any machine, even if SICStus is not installed:

```
bash-2.04$ ./main.exe
hello world
Getting a random value:
4
3 != 9
bash-2.04$
```

### 6.7.3.3 Setting up the C compiler on Windows

spld (and splfr) are command line tools and need to have access to a working C compiler and linker. This is typically not a problem on UNIX-like systems but on Windows there are some special steps needed in order to set up the environment so that the C compiler can be used.

The easiest way to get a command prompt where the C compiler works is to open the ‘Visual Studio 2005 Command Prompt’ from the Start menu. On Windows Vista this is located under ‘All Programs/Microsoft Visual Studio 2005/Visual Studio Tools/’. This opens
up a command prompt where `cl.exe` (the C compiler) can be found via the `PATH` environment variable.

An alternative is to run the Visual Studio set up script from the command prompt, something like:

```
C:\>"C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat" x86
```

This is in fact what the 'Visual Studio 2005 Command Prompt' shortcut does.

Similar steps will work for other versions of Visual Studio. Note that there are different versions of SICStus Prolog for different versions of Visual Studio. This is necessary since each version of Visual Studio comes with its own version of the C library.

Once the environment is set up for using the C compiler you should be able to use the `spld` (and `splfr`) tools without problem.

### 6.7.3.4 Extended Runtime Systems

An extended runtime system is a variant of a runtime system with additional capabilities, including the presence of the Prolog compiler. Extended runtime systems are created with `spld` in a way similar to how ordinary runtime systems are created. An extended runtime system requires a license; see Section “Managing Extended Runtime License Information” in *SICStus Prolog Release Notes* for details about managing such license information.

### 6.7.3.5 Examples

1. The character-based SICStus development system executable (`sicstus`) can be created using:

   ```bash
   % spld --main=prolog -o sicstus
   ```

   This will create a development system that is dynamically linked and has no prelinked foreign resources.

2. ```bash
   % spld --static -D --resources=random -o main
   ```

   This will create a statically linked executable called `main` that has the resource `random` prelinked (statically).

3. An all-in-one executable with a home-built foreign resource.

   This example is similar to the example in Section 6.7.3.2 [All-in-one Executables], page 324, with the addition of a foreign resource of our own.
:- use_module(library(random)).
:- use_module(library(clpfd)).

% This will be called when the application starts:
user:runtime_entry(start) :-
   %%% You may consider putting some other code here...
   write('hello world'),nl,
   write('Getting a random value:'),nl,
   random(1, 10, R), % from random
   write(R),nl,
   ( all_different([3,9]) -> % from clpfd
      write('3 != 9'),nl
   ; otherwise ->
      write('3 = 9!?'),nl
   ),
   '$pint'(4711). % from our own foreign resource 'bar'

foreign(print_int, '$pint(+integer)').
foreign_resource(bar, [print_int]).
:- load_foreign_resource(bar).

/* bar.c */
#include <sicstus/sicstus.h>
#include <stdio.h>
/* bar_glue.h is generated by splfr from the foreign/[2,3] facts.
   Always include the glue header in your foreign resource code. */
#include "bar_glue.h"

extern void print_int(SP_integer a);

void print_int(SP_integer a) 
{
   /* Note the use of SPRIdINTEGER to get a format specifier corresponding
       to the SP_integer type. For most platforms this corresponds
       to "ld" and long, respectively. */
   printf("a=%" SPRIdINTEGER "\n", (SP_integer)a);
}

To create the saved state bar.sav we will compile the file bar.pl and save it with
save_program('bar.sav'). When compiling the file the directive :- load_foreign_ 
resource(bar). is called so a dynamic foreign resource must be present.
Thus, first we build a dynamic foreign resource.

% splfr bar.c bar.pl
Then, we create the saved state.

% sicstus --goal "compile(bar), save_program('bar.sav'), halt."
We also need a static foreign resource to embed in our all-in-one executable.

% splfr --static bar.c bar.pl

Finally, we build the all-in-one executable with spld. We do not need to list the foreign resources needed. spld will extract their names from the .sav file. Adding the --verbose option will make spld output lots of progress information, among which are the names of the foreign resources that are needed. Look for “Found resource name” in the output.

% spld --verbose --static --main=restore --respath=. --resources=bar.sav=/mystuff/bar.sav --output=bar

In this case four foreign resource names are extracted from the .sav file: bar, clpfd, and random. The source file bar.pl loads the foreign resource named bar. It also uses the library(random) module, which loads the foreign resource named random, and the library(clpfd) module, which loads the foreign resource named clpfd.

By not listing foreign resources when running spld, we avoid the risk of omitting a required resource.

### 6.7.4 User-defined Main Programs

Runtime systems may or may not have an automatically generated main program. This is controlled by the --main option to spld. If --main=user is given, then a function user_main() must be supplied:

```c
int user_main(int argc, char *argv[])
```

user_main() is responsible for initializing the Prolog engine, loading code, and issuing any Prolog queries. An alternative is to use --main=none and write your own main() function.

#### 6.7.4.1 Initializing the Prolog Engine

The Prolog Engine is initialized by calling SP_initialize(). This must be done before any interface functions are called, except those marked ‘preinit’ in this manual.

The function will allocate data areas used by Prolog and load the Runtime Library.

It will also initialize command line arguments so that they can be accessed by the argv Prolog flag but it may be preferable to use SP_set_argv() for this.

To unload the SICStus emulator, SP_deinitalize() can be called.

You may also call SP_force_interactive() before calling SP_initialize(). This will force the I/O built-in predicates to treat the standard streams as an interactive, even if they do not appear to be connected to a terminal or console. Same as the -i option in development systems (see Section 3.1 [Start], page 21).

The SICStus Prolog memory manager has a two-layer structure. The top layer has roughly the same functionality as the standard UNIX functions malloc and free, whereas the bottom layer is an interface to the operating system. It is the bottom layer that can be customized by setting these hooks.
6.7.4.2 Loading Prolog Code

You can load your Prolog code with the call SP_load(). This is the C equivalent of the Prolog predicate load_files/1.

Alternatively, you can restore a saved state with the call SP_restore(), which is the C equivalent of the Prolog predicate restore/1.

6.7.5 Generic Runtime Systems

There are three ready-made runtime systems provided with the distributions, sprt.exe, sprti.exe, and (only on Windows) sprtw.exe. These have been created using spld:

$ spld --main=restore 'SP_APP_DIR/main.sav' -o sprt.exe
$ spld --main=restore 'SP_APP_DIR/main.sav' -i -o sprti.exe
$ spld --main=restore 'SP_APP_DIR/main.sav' --window -o sprtw.exe

These are provided for users who do not have a C-compiler available. Each program launches a runtime system by restoring the saved state main.sav (located in the same folder as the program).

The saved state is created by save_program/[1,2]. If it was created by save_program/2, then the given startup goal is run. Then, user:runtime_entry(start) is run. The program exits with 0 upon normal termination and with 1 on failure or exception.

The program sprti.exe assumes that the standard streams are connected to a terminal, even if they do not seem to be (useful under Emacs, for example). On Windows only, sprtw.exe is a windowed executable, corresponding to spwin.exe.

6.8 Mixing C and Prolog Examples

6.8.1 Train Example (connections)

This is an example of how to create a runtime system. The Prolog program train.pl will display a route from one train station to another. The C program train.c calls the Prolog code and writes out all the routes found between two stations:
connected(From, From, [From], _):- !.
connected(From, To, [From| Way], Been):-
    (  no_stop(From, Through) ;
        no_stop(Through, From)
    ),
    not_been_before(Been, Through),
    connected(Through, To, Way, Been).

no_stop('Stockholm', 'Katrineholm').
no_stop('Stockholm', 'Vasteras').
no_stop('Katrineholm', 'Hallsberg').
no_stop('Katrineholm', 'Linkoping').
no_stop('Hallsberg', 'Kumla').
no_stop('Hallsberg', 'Goteborg').
no_stop('Orebro', 'Vasteras').
no_stop('Orebro', 'Kumla').

not_been_before(Way, _) :- var(Way),!.
not_been_before([Been| Way], Am) :-
    Been \= Am,
    not_been_before(Way, Am).
/ train.c */

#include <stdio.h>
#include <stdlib.h>
#include <sicstus/sicstus.h>

static void write_path(SP_term_ref path)
{
    char const *text = NULL;
    SP_term_ref
    tail = SP_new_term_ref(),
    via = SP_new_term_ref();

    SP_put_term(tail,path);

    while (SP_get_list(tail,via,tail)) {
        if (text)
            printf(" -> ");

        SP_get_string(via, &text);
        printf("%s",text);
    }
    printf("\n");
}

int user_main(int argc, char **argv)
{
    int rval;
    SP_pred_ref pred;
    SP_qid goal;
    SP_term_ref from, to, path;

    /* Initialize Prolog engine. The third arg to SP_initialize is
     * an option block and can be NULL, for default options. */
    if (SP_FAILURE == SP_initialize(argc, argv, NULL)) {
        fprintf(stderr, "SP_initialize failed: %s\n",
                SP_error_message(SP_errno));
        exit(1);
    }
    rval = SP_restore("train.sav");

    if (rval == SP_ERROR || rval == SP_FAILURE) {
        fprintf(stderr, "Could not restore \"train.sav\".\n");
        exit(1);
    }
}
/* train.c */

/* Look up connected/4. */
if (!(pred = SP_predicate("connected",4,"user"))) {
    fprintf(stderr, "Could not find connected/4.\n\n"); exit(1);
}

/* Create the three arguments to connected/4. */
SP_put_string(from = SP_new_term_ref(), "Stockholm");
SP_put_string(to = SP_new_term_ref(), "Orebro");
SP_put_variable(path = SP_new_term_ref());

/* Open the query. In a development system, the query would look like:
 * 
 * | ?- connected('Stockholm','Orebro',X).
 * */
if (!(goal = SP_open_query(pred,from,to,path,path))) {
    fprintf(stderr, "Failed to open query.\n\n"); exit(1);
}

/* 
 * Loop through all the solutions. 
 */
while (SP_next_solution(goal)==SP_SUCCESS) {
    printf("Path: ");
    write_path(path);
}

SP_close_query(goal);

exit(0);

Create the saved state containing the Prolog code:

% sicutus
SICStus 4.7.0 ...
Licensed to SICS
| ?- compile(train),save_program('train.sav').
% compiling [...]/train.pl...
% compiled [...]/train.pl in module user, 10 msec 2848 bytes
% [...]/train.sav created in 0 msec

| ?- halt.

Create the executable using the application builder:
And finally, run the executable:

```
% ./train
Path: Stockholm -> Katrineholm -> Hallsberg -> Kumla -> Orebro
Path: Stockholm -> Vasteras -> Orebro
```

6.8.2 Building for a Target Machine

The following example shows how to build an application with a dynamically loaded foreign resource in such a way that it can be deployed into an arbitrary folder on a target system that does not have SICStus installed. The example is run on Linux but it would be very similar on other platforms.

The example consists of three source files, one top-level file (`main.pl`) which in turn loads a module file (`b.pl`). The latter also loads a foreign resource (`b.c`).

The initial directory structure, and the contents of the source files can be seen from the following transcript:
The following transcript shows how the foreign resource and the SICStus runtime executable is built:
$ cd build/myfiles/
$ splfr b.pl b.c
$ cd ..
$ sicstus --nologo
% optional step for embedding source info in saved state.
| ?- set_prolog_flag(source_info, on).
yes
% source_info
| ?- compile('myfiles/main.pl').
% compiling .../build/myfiles/main.pl...
% module main imported into user
% compiling .../build/myfiles/b.pl...
% module b imported into main
% loading foreign resource .../build/myfiles/b.so in module b
% compiled .../build/myfiles/b.pl in module b, 0 msec 3104 bytes
% compiled .../build/myfiles/main.pl in module main, 0 msec 5344 bytes
yes
% source_info
| ?- save_program('main.sav').
% .../build/main.sav created in 20 msec
yes
% source_info
| ?- halt.
$ spld '$SP_APP_DIR/main.sav' -o main.exe
Created "main.exe"

(instead of creating main.exe you could use the generic runtime system sprt.exe provided as part of the installation (see Section 6.7.5 [Generic Runtime Systems], page 331)).

Please note: it is important that main.sav be saved to a folder that is the “root” of the folder tree. The folder in which the saved state is created (.../build/ above) is treated specially by save_program/[1,2] and by restore/1. This special handling ensures that myfiles/b.so will be found relative to the location of main.sav when main.sav is restored on the target system. See Section 3.10 [Saving], page 28, for details.

Next, the necessary runtime files must be copied from the SICStus installation:

$ mkdir -p sp-4.7.0/sicstus-4.7.0/bin
$ cp /usr/local/sicstus4.7.0/lib/libsprt4-7-0.so sp-4.7.0/
$ cp /usr/local/sicstus4.7.0/lib/sicstus-4.7.0/bin/sprt.sav \  
sp-4.7.0/sicstus-4.7.0/bin/sprt.sav

The resulting folder contents can be seen by running the find command:
$ find . -print
.
./sp-4.7.0
./sp-4.7.0/libsprt4-7-0.so
./sp-4.7.0/sicstus-4.7.0
./sp-4.7.0/sicstus-4.7.0/bin
./sp-4.7.0/sicstus-4.7.0/bin/sprt.sav
./myfiles
./myfiles/b.so
./myfiles/main.pl
./myfiles/b.pl
./myfiles/b.c
./main.sav
./main.exe

It is possible to run the program from its current location:

$ ./main.exe
Hello World!

The folder `build/myfiles/` contains some files that do not need to be present on the target machine, i.e. the source files. The following transcript shows how a new folder, `target/`, is created that contains only the files that need to be present on the target system.

```
$ cd ..
$ mkdir target
$ mkdir target/myfiles
$ cp build/main.sav target
$ cp build/main.exe target
$ cp build/myfiles/b.so target/myfiles/
$ cp -R build/sp-4.7.0 target
$ find target/ -print
  target/
  target/myfiles
  target/myfiles/b.so
  target/main.sav
  target/main.exe
  target/sp-4.7.0
  target/sp-4.7.0/sicstus-4.7.0
  target/sp-4.7.0/sicstus-4.7.0/bin
  target/sp-4.7.0/sicstus-4.7.0/bin/sprt.sav
  target/sp-4.7.0/libsprt4-7-0.so
$ target/main.exe
Hello World!
```

Note that `target/myfiles/b.so` was found since its location relative the directory containing the saved state (`main.sav`) is the same on the target system as on the build system.
The folder target/ can now be moved to some other system and target/main.exe will not depend on any files of the build machine.

As a final example, the following transcripts show how the runtime system can be debugged on the build machine. It is possible to do this on the target system as well, if the necessary files are made available. See Section 6.9 [Debugging Runtime Systems], page 342, for more information.

First, the development system files and the license file must be made available:

```
$ mkdir sp-4.7.0/sicstus-4.7.0/library
$ cp /usr/local/sicstus4.7.0/lib/sicstus-4.7.0/library/SU_messages.po \
    sp-4.7.0/sicstus-4.7.0/library/
$ cp /usr/local/sicstus4.7.0/lib/sicstus-4.7.0/bin/spds.sav \
    sp-4.7.0/sicstus-4.7.0/bin/
$ cp /usr/local/sicstus4.7.0/lib/sicstus-4.7.0/library/license.pl \
    sp-4.7.0/sicstus-4.7.0/library/
```

As before, the resulting folder contents can be seen by running the `find` command:

```
$ find . -print
.
./sp-4.7.0
./sp-4.7.0/libsprt4-7-0.so
./sp-4.7.0/sicstus-4.7.0
./sp-4.7.0/sicstus-4.7.0/library
./sp-4.7.0/sicstus-4.7.0/library/SU_messages.po
./sp-4.7.0/sicstus-4.7.0/library/license.pl
./sp-4.7.0/sicstus-4.7.0/bin
./sp-4.7.0/sicstus-4.7.0/bin/spds.sav
./sp-4.7.0/sicstus-4.7.0/bin/sprt.sav
./myfiles
./myfiles/b.so
./myfiles/main.pl
./myfiles/b.pl
./myfiles/b.c
./main.sav
./main.exe
$```

To tell the runtime system to start a development system, you can set the `SP_USE_DEVSYS` environment variable as shown below. You could also set `SP.Attach.SPIDER` and debug in the SICStus IDE (see Section 6.9 [Debugging Runtime Systems], page 342).
Please note: source info is available, since we used `set_prolog_flag(source_info, on)` before we compiled `main.pl` and created the saved state `main.sav`.

6.8.3 Exceptions from C

Consider, for example, a function returning the square root of its argument after checking that the argument is valid. If the argument is invalid, then the function should raise an exception instead.
/* math.c */

#include <math.h>
#include <stdio.h>
#include <sicstus/sicstus.h>

/* math_glue.h is generated by splfr from the foreign/[2,3] facts.
   Always include the glue header in your foreign resource code. */
#include "math_glue.h"

extern double sqrt_check(double d);

double sqrt_check(double d)
{
  if (d < 0.0) { /* build a domain_error/4 exception term */
    SP_term_ref culprit=SP_new_term_ref();
    SP_term_ref argno=SP_new_term_ref();
    SP_term_ref expdomain=SP_new_term_ref();
    SP_term_ref t1=SP_new_term_ref();

    SP_put_float(culprit, d);
    SP_put_integer(argno, 1);
    SP_put_string(expdomain, ">=0.0");
    SP_cons_functor(t1, SP_atom_from_string("sqrt"), 1, culprit);
    SP_cons_functor(t1, SP_atom_from_string("domain_error"), 4,
                    t1, argno, expdomain, culprit);
    SP_raise_exception(t1); /* raise the exception */
    return 0.0;
  }
  return sqrt(d);
}

The Prolog interface to this function is defined in a file math.pl. The function uses the sqrt() library function, and so the math library -lm has to be included:

% math.pl

foreign_resource(math, [sqrt_check]).

foreign(sqrt_check, c, sqrt(+float, [-float])).

:- load_foreign_resource(math).

A linked foreign resource is created:

% splfr math.pl math.c -lm

A simple session using this function could be:
$ sicstus
SICStus 4.7.0 ...
Licensed to SICS
| ?- [math].
% compiling .../math.pl...
% loading foreign resource .../math.so in module user
% compiled .../math.pl in module user, 0 msec 2400 bytes
yes
| ?- sqrt(5.0,X).
X = 2.23606797749979 ?
yes
| ?- sqrt(a,X).
! Type error in argument 1 of user:sqrt/2
! expected a number, but found a
! goal: user:sqrt(a,_110)
| ?- sqrt(-5,X).
! Domain error in argument 1 of user:sqrt/1
! expected '>=0.0', but found -5.0
! goal: sqrt(-5.0)

6.8.4 Stream Example
See Section 6.6.2 [Defining a New Stream], page 313, for a simple example of defining a stream that reads from a C FILE stream.

For a more realistic example, library(codesio) implements a stream that can return a list of all characters written to it. The source code for this library is located in library/codesio.pl and library/codesio.c and can serve as a useful sample for user defined streams both for input and output. That code also illustrates other important features of user defined streams, for instance ensuring that all the streams have been closed when the foreign resource is unloaded.

6.9 Debugging Runtime Systems
A runtime system does not contain the Prolog debugger by default. This makes it hard to troubleshoot problems that only occur when the code is embedded in some other application.

As of release 4.2, it is possible to tell a runtime system to start the full development system instead. This way, the Prolog debugger, compiler etc. can be used to debug the application, either from the command line or by attaching to the SICStus Prolog IDE (SPIDER). In the simplest case, this feature is enabled by setting the system property (or environment variable) SP_USE_DEVSYS to yes before starting the runtime system. This will cause the runtime system to become a development system and it will start the debugger, as if by a call to trace/0. See Section 6.8.2 [Building for a Target Machine], page 335, for a complete example.

For best effect, you should ensure that any compiled prolog code (’.sav’ and ‘.po’ files) has been compiled with the Prolog flag source_info enabled, i.e. with set_prolog_flag(source_info, on).
When the runtime system is started as a development system in this way, it needs to be able to find the file that makes up an ordinary development system, i.e. spds.sav; see Section 6.7.2.1 [Runtime Systems on UNIX Target Machines], page 320, and Section 6.7.2.2 [Runtime Systems on Windows Target Machines], page 321, above. It also needs to find the license information for the development system; see Section 6.9.1 [Locating the License Information], page 343, below.

6.9.1 Locating the License Information

The license information for debugged runtime systems can be provided in several ways. Most of them can also be used as alternative ways for providing the license information to extended runtime systems (see Section “Managing Extended Runtime License Information” in SICStus Prolog Release Notes).

On Windows only, if you have installed the SICStus Prolog development system on the machine where the runtime system is to be debugged, then the license information will be found in the Windows registry and no extra steps need to be performed. This method cannot be used for providing an extended runtime system license since the license information for the full development system is not the same as for an extended runtime system.

If you have the license in a file license.pl, i.e. you are using a UNIX platform or have manually created a license.pl file on Windows, then you can make this file available to the debugged runtime system in one of two ways:

- Set the system property or environment variable SP_LICENSE_FILE to the absolute path of the license.pl file of a SICStus Prolog installation, or
- Copy the license.pl file into the appropriate location relative to the runtime system executable, i.e. to sp-4.7.0/sicstus-4.7.0/library/license.pl on UNIX or sp-4.7.0\library\license.pl on Windows.

Please note: you cannot redistribute license.pl.

The final alternative, available on all platforms, is to set the following system properties or environment variables

SP_LICENSE_SITE
Set to the site name part of your license.

SP_LICENSE_CODE
Set to the code part of your license, e.g. something like a111-b222-c333-d444-e444.

SP_LICENSE_EXPIRATION
Set to the expiration part of your license, e.g. permanent if you have a permanent (non-evaluation) license.

6.9.2 Customizing the Debugged Runtime System

It is possible to fine-tune the behavior of the debugged runtime system in various ways, both at compile time (setting C preprocessor symbols and passing system properties to SP_initialize()) and at runtime (passing system properties as environment variables).
The system properties and environment variables that affect the debugged runtime system are:

**SP_USE_DEVSYS**
- if set to **yes**, then the runtime system will try to start a development system, as described above.

**SP_ATTACH_SPIDER**
- if set to **yes**, then this has the same effect as **SP_USE_DEVSYS=yes** and in addition tries to attach to the SICStus Prolog IDE (SPIDER). You have to tell SPIDER to ‘Attach to Prolog Process’, i.e. listen for an incoming connection. This command is available from the SICStus top-level view menu in SPIDER.

**SP_DEVSYS_NO_TRACE**
- if set to **yes**, then this will prevent the runtime system from calling `trace/0` at initialization. This is useful if you prefer to manually enable the debugger later from your C or Prolog code.

**SP_ALLOW_DEVSYS**
- if set to **no**, then this will prevent the runtime system from starting as a development system. This may be useful in order to prevent inheriting **SP_USE_DEVSYS** or **SP_ATTACH_SPIDER** from the environment. The same effect can be obtained by passing the option **--no-allow-devsys** to `spld` when building the runtime system.

**SP_LICENSE_FILE**
**SP_LICENSE_SITE**
**SP_LICENSE_CODE**
**SP_LICENSE_EXPIRATION**

These are described in Section 6.9.1 [Locating the License Information], page 343, above.

If your C code calls **SP_initialize()**, then you can pass these system properties in the call to **SP_initialize()** (see Section 12.3.44 [SP.initialize], page 1316). You can also pass these options to **SP_initialize()** by setting the SPLD_DSP C macro. See the definition of **SP_initialize()** in the header file `sictus/sicstus.h` for details.

### 6.9.3 Examples of Debugging Runtime Systems

The following examples show how to start Prolog debugging when SICStus is run from within Java via Jasper. The examples assume that the SICStus files are part of a development system installation.

The first example initializes the SICStus system property **SP_USE_DEVSYS** by setting the environment variable with the same name. This method of passing SICStus system properties also works well when SICStus is embedded in some other, non-Java, program.
$ SP=/usr/local/sicstus4.7.0
$ SP_USE_DEVSYS=yes
$ export SP_USE_DEVSYS
$ java -jar "$SP/lib/sicstus-4.7.0/bin/jasper.jar"

Trying to load SICStus.
% The debugger will first creep -- showing everything (trace)
  1 1 Call: write('If you see this message, you have successfully') ? RET
If you see this message, then you have successfully
  1 1 Exit: write('If you see this message, you have successfully') ? n

initialized the SICStus Prolog engine.
$ unset SP_USE_DEVSYS

The second example initializes the SICStus system property SP_USE_DEVSYS by setting
the Java system property se.sics.sicstus.property.SP_USE_DEVSYS. This method of
passing SICStus system properties is specific to Jasper.

$ SP=':/usr/local/sicstus4.7.0'
$ java -Dse.sics.sicstus.property.SP_USE_DEVSYS=yes \ 
   -jar "$SP/lib/sicstus-4.7.0/bin/jasper.jar"

Trying to load SICStus.
% The debugger will first creep -- showing everything (trace)
  1 1 Call: write('If you see this message, you have successfully') ? n
If you see this message, then you have successfully
initialized the SICStus Prolog engine.
$
7 Interfacing .NET and Java

SICStus Prolog supports two different ways of interfacing a Prolog program with a Java client, and one for interfacing with a .NET client.

SICStus Prolog provides a uniform way of interfacing to Java and .NET clients via the PrologBeans (see Section 10.35 [lib-prologbeans], page 731) interface. This is a loosely coupled interface, which means that the client code runs in a different process from the Prolog code. In fact, the client program and the Prolog program can run on separate machines, since the communication is done via TCP/IP sockets. This design has the following advantages over a tightly coupled interface, where they run in the same process:

- There is no competition for memory or other process-wide resources between the virtual machines of the client (.NET or JVM) and of Prolog.
- Distribution over a network is trivial when using PrologBeans. The application is distributable from the beginning.
- PrologBeans has support for user session handling both at the Java level (with support for HTTP sessions and JNDI lookup) and at the Prolog level. This makes it easy to integrate Prolog applications into applications based on Java servers.

The main limitation of the design is that callbacks from Prolog to the client is not provided for.

PrologBeans is the recommended package unless you have special needs and are interfacing with Java in which case you may consider using Jasper.

For interfacing to Java clients SICStus Prolog also provides Jasper (see Section 10.20 [lib-jasper], page 572), a “tightly coupled” interface. This means that everything runs in the same process (the necessary code is loaded at run time via dynamic linking).

Advantages of Jasper:

- Jasper is bi-directional. Callbacks are possible (limited in levels only by memory), and queries can backtrack.
8 Multiple SICStus Runtimes in a Process

It is possible to have more than one SICStus runtime in a single process. These are completely independent (except that they dynamically load the same foreign resources; see Section 8.3 [Foreign Resources and Multiple SICStus Runtimes], page 351).

Even though the SICStus runtime can only be run in a single thread, it is now possible to start several SICStus runtimes, optionally each in its own thread.

SICStus runtimes are rather heavy weight and you should not expect to be able to run more than a handful.

8.1 Multiple SICStus Runtimes in Java

In Java, you can now create more than one `se.sics.jasper.SICStus` object. Each will correspond to a completely independent copy of the SICStus runtime. Note that a SICStus runtime is not deallocated when the corresponding SICStus object is no longer used. Thus, the best way to use multiple SICStus objects is to create them early and then re-use them as needed.

It is probably useful to create each in its own separate thread. One reason would be to gain speed on a multi-processor machine.

8.2 Multiple SICStus Runtimes in C

Unless otherwise noted, this section documents the behavior when using dynamic linking to access a SICStus runtime.

The key implementation feature that makes it possible to use multiple runtimes is that all calls from C to the SICStus API (`SP_query()`, etc.) go through a dispatch vector. Two runtimes can be loaded at the same time since their APIs are accessed through different dispatch vectors.

By default, there will be a single dispatch vector, referenced from a global variable (`sp_GlobalSICStus`). A SICStus API functions, such as `SP_query()`, is then defined as a macro that expands to something similar to `sp_GlobalSICStus->SP_query_pointer`. The name of the global dispatch vector is subject to change without notice; it should not be referenced directly. If you need to access the dispatch vector, then use the C macro `SICStusDISPATCHVAR` instead; see below.

8.2.1 Using a Single SICStus Runtime

When building an application with `spld`, by default only one SICStus runtime can be loaded in the process. This is similar to what was the case before release 3.9. For most applications built with `spld`, the changes necessary to support multiple SICStus runtimes should be invisible, and old code should only need to be rebuilt with `spld`.

In order to maintain backward compatibility, the global dispatch vector is automatically set up implicitly by `SP_initialize()` and explicitly by `SP_setup_dispatch()`. Other SICStus API functions will not set up the dispatch vector, and will therefore lead to memory
access errors if called before SP_initialize(). Currently, hook functions such as SP_set_user_stream_hook() also set up the dispatch vector to allow them to be called before SP_initialize(). However, only SP_initialize() and SP_setup_dispatch() are guaranteed to set up the dispatch vector. The hook installation functions may change to use a different mechanism in the future. The SICStus API functions that perform automatic setup of the dispatch vector are marked with SPEXPFLAG_PREINIT in sicstus.h.

8.2.2 Using More than One SICStus Runtime

Using more than one SICStus runtime in a process is only supported when the dynamic library version of the SICStus runtime is used (e.g. sprt4-7-0.dll, libsprt4-7-0.so).

An application that wants to use more than one SICStus runtime needs to be built using the --multi-sp-aware option to spld. C-code compiled by spld --multi-sp-aware will have the C preprocessor macro MULTI_SP_AWARE defined and non-zero.

Unlike the single runtime case described above, an application built with --multi-sp-aware will not have a global variable that holds the dispatch vector. Instead, your code will have to take steps to ensure that the appropriate dispatch vector is used when switching between SICStus runtimes.

There are several steps needed to access a SICStus runtime from an application built with --multi-sp-aware.

1. You must obtain the dispatch vector of the initial SICStus runtime using SP_get_dispatch(). Note that this function is special in that it is not accessed through the dispatch vector; instead, it is exported in the ordinary manner from the SICStus runtime dynamic library (sprt4-7-0.dll under Windows and, typically, libsprt4-7-0.so under UNIX).

2. You must ensure that SICStusDISPATCHVAR expands to something that references the dispatch vector obtained in step 1.

The C preprocessor macro SICStusDISPATCHVAR should expand to a SICSTUS_API_STRUCT_TYPE *, that is, a pointer to the dispatch vector that should be used. When --multi-sp-aware is not used SICStusDISPATCHVAR expands to sp_GlobalSICStus as described above. When using --multi-sp-aware it is probably best to let SICStusDISPATCHVAR expand to a local variable.

3. Once you have access to the SICStus API of the initial SICStus runtime you can call the SICStus API function SP_load_sicstus_run_time() to load additional runtimes.

```c
SICSTUS_API_STRUCT_TYPE *SP_get_dispatch(void *reserved);
```

SP_get_dispatch() returns the dispatch vector of the SICStus runtime. The argument reserved should be NULL. This function can be called from any thread.

```c
typedef SICSTUS_API_STRUCT_TYPE *SP_get_dispatch_type(void *);
```

```c
int SP_load_sicstus_run_time(SP_get_dispatch_type **ppfunc, void *reserved);
```
**SP_load_sicstus_run_time()** loads a new SICStus runtime. If a new runtime could be loaded, then a positive value is returned and the address of the **SP_get_dispatch()** function of the newly loaded SICStus runtime is stored at the address **ppfunc**. The second argument, **phandle**, is reserved and should be **NULL**.

As a special case, if **SP_load_sicstus_run_time()** is called from a SICStus runtime that has not been initialized (with **SP_initialize()**) and that has not previously been loaded as the result of calling **SP_load_sicstus_run_time()**, then no new runtime is loaded. Instead, the **SP_get_dispatch()** of the runtime itself is returned. In particular, the first time **SP_load_sicstus_run_time()** is called on the initial SICStus runtime, and if this happens before the initial SICStus runtime is initialized, then no new runtime is loaded.

Calling **SP_load_sicstus_run_time()** from a particular runtime can be done from any thread.

An application that links statically with the SICStus runtime should not call **SP_load_sicstus_run_time()**.

You should not use prelinked foreign resources when using multiple SICStus runtimes in the same process.

For an example of loading and using multiple SICStus runtimes, see **library/jasper/spnative.c** that implements this functionality for the Java interface Jasper.

### 8.3 Foreign Resources and Multiple SICStus Runtimes

Foreign resources access the SICStus C API in the same way as an embedding application, that is, through a dispatch vector. As for applications, the default and backward compatible mode is to only support a single SICStus runtime. An alternative mode makes it possible for a foreign resource to be shared between several SICStus runtimes in the same process.

Unless otherwise noted, this section documents the behavior when using dynamically linked foreign resources. That is, shared objects (e.g.: .so-files) under UNIX, dynamic libraries (DLLs) under Windows.

#### 8.3.1 Foreign Resources Supporting Only One SICStus Runtime

A process will only contain one instance of the code and data of a (dynamic) foreign resource even if the foreign resource is loaded and used from more than one SICStus runtime.

This presents a problem in the likely event that the foreign resource maintains some state, e.g. global variables, between invocations of functions in the foreign resource. The global state will probably need to be separate between SICStus runtimes. Requiring a foreign resource to maintain its global state on a per SICStus runtime basis would be an incompatible change. Instead, by default, only the first SICStus runtime that loads a foreign resource will be allowed to use it. If a subsequent SICStus runtime (in the same process) tries to load the foreign resource, then an error will be reported to the second SICStus runtime.
When `splfr` builds a foreign resource, it will also generate glue code. When the foreign resource is loaded, the glue code will set up a global variable pointing to the dispatch vector used in the foreign resource to access the SICStus API. This is similar to how an embedding application accesses the SICStus API.

The glue code will also detect if a subsequent SICStus runtime in the same process tries to initialize the foreign resource. In this case, an error will be reported.

This means that pre 3.9 foreign code should only need to be rebuilt with `splfr` to work with the latest version of SICStus. However, a recommended change is that all C files of a foreign resource include the header file generated by `splfr`. Inclusion of this generated header file may become mandatory in a future release. See Section 6.2.5 [The Foreign Resource Linker], page 296.

### 8.3.2 Foreign Resources Supporting Multiple SICStus Runtimes

A foreign resource that wants to be shared between several SICStus runtimes must somehow know which SICStus runtime is calling it so that it can make callbacks using the SICStus API into the right SICStus runtime. In addition, the foreign resource may have global variables that should have different values depending on which SICStus runtime is calling the foreign resource.

A header file is generated by `splfr` when it builds a foreign resource (before any C code is compiled). This header file provides prototypes for any `foreign`-declared function, but it also provides other things needed for multiple SICStus runtime support. This header file must be included by any C file that contains code that either calls any SICStus API function or that contains any of the functions called by SICStus. See Section 6.2.5 [The Foreign Resource Linker], page 296.

#### 8.3.2.1 Full Support for Multiple SICStus Runtimes

To fully support multiple SICStus runtimes, a foreign resource should be built with `splfr --multi-sp-aware`.

C code compiled by `splfr --multi-sp-aware` will have the C preprocessor macro `MULTI_SP_AWARE` defined to a non-zero value.

Full support for multiple SICStus runtimes means that more than one SICStus runtime can execute code in the foreign resource at the same time. This rules out the option to use any global variables for information that should be specific to each SICStus runtime. In particular, the SICStus dispatch vector cannot be stored in a global variable. Instead, the SICStus dispatch vector is passed as an extra first argument to each foreign function.

To ensure some degree of link time type checking, the name of each foreign function will be changed (using `#define` in the generated header file).

The extra argument is used in the same way as when using multiple SICStus runtimes from an embedding application. It must be passed on to any function that needs access to the SICStus API.
To simplify the handling of this extra argument, several macros are defined so that the same foreign resource code can be compiled both with and without support for multiple SICStus runtimes:

- `SPAPI_ARG0`
- `SPAPI_ARG`
- `SPAPI_ARG_PROTO_DECL0`
- `SPAPI_ARG_PROTO_DECL`

Their use is easiest to explain with an example. Suppose the original foreign code looked like:

```c
static int f1(void)
{
    some SICStus API calls
}

static int f2(SP_term_ref t, int x)
{
    some SICStus API calls
}

/* :- foreign(foreign_fun, c, foreign_pred(+integer)). */
void foreign_fun(SP_integer x)
{
    ... some SICStus API calls ...
    f1();
    ...
    f2(SP_new_term_ref(), 42);
    ...
}
```

Assuming no global variables are used, the following change will ensure that the SICStus API dispatch vector is passed around to all functions:
static int f1(SPAPI_ARG_PROTO_DECL0) // _DECL<ZERO> for no-arg functions
{
    some SICStus API calls
}

static int f2(SPAPI_ARG_PROTO_DECL SP_term_ref t, int x) // Note: no comma
{
    some SICStus API calls
}

/* :- foreign(foreign_fun, c, foreign_pred([-integer])). */
void foreign_fun(SPAPI_ARG_PROTO_DECL SP_integer x) // Note: no comma
{
    ... some SICStus API calls ...
    f1(SPAPI_ARG0); // ARG<ZERO> for no-arg functions
    ...
    f2(SPAPI_ARG SP_new_term_ref(), 42); // Note: no comma
    ...
}

If MULTI_SP_AWARE is not defined, i.e. --multi-sp-aware is not specified to splfr, then all these macros expand to nothing, except SPAPI_ARG_PROTO_DECL0, which will expand to void.

You can use SP_foreign_stash() to get access to a location, initially set to NULL, where the foreign resource can store a void*. Typically this would be a pointer to a C struct that holds all information that need to be stored in global variables. This struct can be allocated and initialized by the foreign resource init function. It should be deallocated by the foreign resource deinit function. See Section 6.4.7.3 [OS Threads], page 306, for details.

Most foreign resources that come with SICStus fully support multiple SICStus runtimes. For a particularly simple example, see the code for library(random). For an example that hides the passing of the extra argument by using the C preprocessor, see the files in library/clpfd/.

8.4 Multiple Runtimes and Threads

Perhaps the primary reason to use more than one SICStus runtime in a process is to have each runtime running in a separate thread. To this end, a few mutual exclusion primitives are available. See Section 6.4.7 [Operating System Services], page 305, for details on mutual exclusion locks.

Please note: the SICStus runtime is not thread safe in general. See Section 6.5.4 [Calling Prolog Asynchronously], page 309, for ways to safely interact with a running SICStus from arbitrary threads.
9 Writing Efficient Programs

9.1 Overview

This chapter gives a number of tips on how to organize your programs for increased efficiency. A lot of clarity and efficiency is gained by sticking to a few basic rules. This list is necessarily very incomplete. The reader is referred to textbooks such as [O’Keefe 90] for a thorough exposition of the elements of Prolog programming style and techniques.

- Do Not write code in the first place if there is a library predicate that will do the job.
- Write clauses representing base case before clauses representing recursive cases.
- Input arguments before output arguments in clause heads and goals.
- Use pure data structures instead of database changes.
- Use cuts sparingly, and only at proper places (see Section 4.2.3.1 [ref-sem-ctr-cut], page 64). A cut should be placed at the exact point that it is known that the current choice is the correct one: no sooner, no later.
- Make cuts as local in their effect as possible. If a predicate is intended to be determinate, then define it as such; do not rely on its callers to prevent unintended backtracking.
- Binding output arguments before a cut is a common source of programming errors. If a predicate is not steadfast, then it is usually for this reason.
- Replace cuts by if-then-else constructs if the test is simple enough (see Section 9.10 [Conditionals and Disjunction], page 373).
- Use disjunctions sparingly, always put parentheses around them, never put parentheses around the individual disjuncts, never put the ‘;’ at the end of a line.
- Write the clauses of a predicate so that they discriminate on the principal functor of the first argument (see below). For maximum efficiency, avoid “defaulty” programming (“catch-all” clauses).
- Do Not use lists ([...]), “round lists” ((...)), or braces ({...}) to represent compound terms, or “tuples”, of some fixed arity. The name of a compound term comes for free.
- Before trying to optimize your program for speed, use execution profiling to get an idea of where most of the time is being spent, and, more importantly, why.

9.2 Execution Profiling

Execution profiling is a common aid for improving software performance. As of release 4.2, execution profiling is available for compiled as well as interpreted code. Execution profiling requires no recompilation with instrumentation. Execution profiling is either globally on or globally off for all compiled code. This is reflected by the profiling Prolog flag. When the flag is on, the information gathered depends on the execution mode:

compiled code

Execution profiling counts the number of calls per caller-callee pair, the number of instructions executed, and the number of choicepoint accesses per predicate.
Calls that succeed nondeterminately are detected. Compiled codes runs 2-10 times slower with execution profiling than without.

interpreted code

Execution profiling counts the number of calls per caller-callee pair if the source_info Prolog flag was on when the code was loaded; otherwise, the number of calls per predicate. Calls that succeed nondeterminately are detected.

A typical query pattern is:

| ?- [Load some code.]  
| ?- prolog_flag(profiling,_,on).  
| ?- [Run some queries.]  
| ?- prolog_flag(profiling,_,off).  
| ?- print_profile.

The predicate profile_data/1 makes the accumulated data available as a Prolog term. The predicate print_profile/0 prints the execution profile in a format similar to gprof(1). It can also be given an argument which should be of the same type as the output of profile_data/1. The predicate profile_reset/0 clears all profiling data. For the details, see the respective reference page. See also the Gauge graphical user interface for inspecting execution profiles (see Section 10.17 [lib-gauge], page 561) and the SICStus Prolog IDE (see Section 3.11 [SPIDER], page 29) which both can visualize the profiling information.

9.3 Coverage Analysis

Coverage analysis is the gathering of information about which points in the code, or coverage sites, were executed, and how many times, during a particular run of the program. It is available as of release 4.2, for compiled as well as interpreted code, provided that such code was loaded with the source_info Prolog flag switched on. In fact, it uses the same underlying support as execution profiling: while the program is running with execution profiling switched on, the data accumulated can be used for both purposes. Roughly, coverage sites correspond to points in the code at which the debugger would stop in trace mode, plus one site at entry to every clause. A typical query pattern is:
Chapter 9: Writing Efficient Programs

| ?- [Load some code.]
| ?- prolog_flag(profiling,_,on).
| ?- [Run some queries.]
| ?- prolog_flag(profiling,_,off).
| ?- print_coverage.

The predicate coverage_data/1 makes the accumulated data available as a Prolog term. The predicate print_coverage/0 prints the execution coverage in a hierarchical format. It can also be given an argument which should be of the same type as the output of coverage_data/1. The collected coverage information can be presented by the SICStus Prolog IDE, SPIDER (see Section 3.11 [SPIDER], page 29). The Emacs interface also has commands for code coverage highlighting of source code buffers (see Section 3.12.3 [Usage], page 34). For the details, see the respective reference page.

profile_reset

Resets all profiling and coverage data. See Section 11.3.166 [mpg-ref-profile_reset], page 1142.

coverage_data(-Data)

Data is the coverage data accumulated so far. See Section 11.3.45 [mpg-ref-coverage_data], page 986.

print_coverage

print_coverage(+Data)

The coverage data Data is displayed in a hierarchical format. Data defaults to the profiling data accumulated so far. See Section 11.3.161 [mpg-ref-print_coverage], page 1135.

9.4 The Cut

9.4.1 Overview

One of the more difficult things to master when learning Prolog is the proper use of the cut. Often, when beginners find unexpected backtracking occurring in their programs, they try to prevent it by inserting cuts in a rather random fashion. This makes the programs harder to understand and sometimes stops them from working.

During program development, each predicate in a program should be considered independently to determine whether or not it should be able to succeed more than once. In most applications, many predicates should at most succeed only once; that is, they should be determinate. Having decided that a predicate should be determinate, it should be verified that, in fact, it is. The debugger can help in verifying that a predicate is determinate (see Section 9.7 [The Determinacy Checker], page 363).

9.4.2 Making Predicates Determinate

Consider the following predicate, which calculates the factorial of a number:
fac(0, 1).
fac(N, X) :-
    N1 is N - 1,
    fac(N1, Y),
    X is N * Y.

The factorial of 5 can be found by typing:

| ?- fac(5, X).

X = 120

However, backtracking into the above predicate by typing a semicolon at this point, causes an infinite loop because the system starts attempting to satisfy the goals fac(-1, X), fac(-2, X), etc. The problem is that there are two clauses that match the goal fac(0, F), but the effect of the second clause on backtracking has not been taken into account. There are at least three possible ways of fixing this:

1. Efficient solution: rewrite the first clause as
   fac(0, 1) :- !.
   Adding the cut essentially makes the first solution the only one for the factorial of 0 and hence solves the immediate problem. This solution is space-efficient because as soon as Prolog encounters the cut, it knows that the predicate is determinate. Thus, when it tries the second clause, it can throw away the information it would otherwise need in order to backtrack to this point. Unfortunately, if this solution is implemented, then typing ‘fac(-1, X)’ still generates an infinite search.

2. Robust solution: rewrite the second clause as
   fac(N, X) :-
       N > 0,
       N1 is N - 1,
       fac(N1, Y),
       X is N * Y.
   This also solves the problem, but it is a more robust solution because this way it is impossible to get into an infinite loop.
   This solution makes the predicate logically determinate—there is only one possible clause for any input—but the Prolog system is unable to detect this and must waste space for backtracking information. The space-efficiency point is more important than it may at first seem; if fac/2 is called from another determinate predicate, and if the cut is omitted, then Prolog cannot detect the fact that fac/2 is determinate. Therefore, it will not be able to detect the fact that the calling predicate is determinate, and space will be wasted for the calling predicate as well as for fac/2 itself. This argument applies again if the calling predicate is itself called by a determinate predicate, and so on, so that the cost of an omitted cut can be very high in certain circumstances.

3. Preferred solution: rewrite the entire predicate as the single clause
fac(N, X) :-
  ( N > 0 ->
    N1 is N - 1,
    fac(N1, Y),
    X is N * Y
  ; N =:= 0 ->
    X = 1
).

This solution is as robust as solution 2, and more efficient than solution 1, since it exploits conditionals with arithmetic tests (see Section 9.10 [Conditionals and Disjunction], page 373, for more information on optimization using conditionals).

### 9.4.3 Placement of Cuts

Programs can often be made more readable by the placing of cuts as early as possible in clauses. For example, consider the predicate \( p/0 \) defined by

\[
p :- a, b, !, c, d.
p :- e, f.
\]

Suppose that \( b/0 \) is a test that determines which clause of \( p/0 \) applies; \( a/0 \) may or may not be a test, but \( c/0 \) and \( d/0 \) are not supposed to fail under any circumstances. A cut is most appropriately placed after the call to \( b/0 \). If in fact \( a/0 \) is the test and \( b/0 \) is not supposed to fail, then it would be much clearer to move the cut before the call to \( b/0 \).

A tool to aid in determinacy checking is included in the distribution. It is described in depth in Section 9.7 [The Determinacy Checker], page 363.

### 9.4.4 Terminating a Backtracking Loop

Cut is also commonly used in conjunction with the generate-and-test programming paradigm. For example, consider the predicate \( \text{find_solution}/1 \) defined by

\[
\text{find_solution}(X) :-
  \text{candidate_solution}(X),
  \text{test_solution}(X),
  !.
\]

where \( \text{candidate_solution}/1 \) generates possible answers on backtracking. The intent is to stop generating candidates as soon as one is found that satisfies \( \text{test_solution}/1 \). If the cut were omitted, then a future failure could cause backtracking into this clause and restart the generation of candidate solutions. A similar example is shown below:
process_file(F) :-
    see(F),
    repeat,
    read(X),
    process_and_fail(X),
    !,
    seen.

process_and_fail(end_of_file) :- !.
process_and_fail(X) :-
    process(X),
    fail.

The cut in process_file/1 is another example of terminating a generate-and-test loop. In general, a cut should always be placed after a repeat/0 so that the backtracking loop is clearly terminated. If the cut were omitted in this case, then on later backtracking Prolog might try to read another term after the end of the file had been reached.

The cut in process_and_fail/1 might be considered unnecessary because, assuming the call shown is the only call to it, the cut in process_file/1 ensures that backtracking into process_and_fail/1 can never happen. While this is true, it is also a good safeguard to include a cut in process_and_fail/1 because someone may unwittingly change process_file/1 in the future.

9.5 Indexing
9.5.1 Overview
In SICStus Prolog, predicates are indexed on their first arguments. This means that when a predicate is called with an instantiated first argument, a hash table is used to gain fast access to only those clauses having a first argument with the same primary functor as the one in the predicate call. If the first argument is atomic, then only clauses with a matching first argument are accessed. Indexes are maintained automatically by the built-in predicates manipulating the Prolog database (for example, assert/1, retract/1, and compile/1).

Keeping this feature in mind when writing programs can help speed their execution. Some hints for program structuring that will best use the indexing facility are given below. Note that interpreted, e.g. dynamic, predicates as well as compiled predicates are indexed. The programming hints given in this section apply equally to compiled and to interpreted code.

9.5.2 Data Tables
The major advantage of indexing is that it provides fast access to tables of data. For example, a table of employee records might be represented as shown below in order to gain fast access to the records by employee name:
% employee(LastName, FirstNames, Department, Salary, DateOfBirth)

employee('Smith', ['John'], sales, 20000, 1-1-59).
employee('Jones', ['Mary'], engineering, 30000, 5-28-56).
...

If fast access to the data via department is also desired, then the data can be organized little differently. The employee records can be indexed by some unique identifier, such as employee number, and additional tables can be created to facilitate access to this table, as shown in the example below. For example,

% employee(Id, LastName, FirstNames, Department, Salary, DateOfBirth)

employee(1000000, 'Smith', ['John'], sales, 20000, 1-1-59).
employee(1000020, 'Jones', ['Mary'], engineering, 30000, 5-28-56).
...

% employee_name(LastName, EmpId)

employee_name('Smith', 1000000).
employee_name('Jones', 1000020).
...

% department_member(Department, EmpId)

department_member(sales, 1000000).
department_member(engineering, 1000020).
...

Indexing would now allow fast access to the records of every employee named Smith, and these could then be backtracked through looking for John Smith. For example:

| ?- employee_name('Smith', Id),
  employee(Id, 'Smith', ['John'], Dept, Sal, DoB).

Similarly, all the members of the engineering department born since 1965 could be efficiently found like this:

| ?- department_member(engineering, Id),
  employee(Id, LN, FN, engineering, _, M-D-Y),
  Y > 65.

9.5.3 Determinacy Detection

The other advantage of indexing is that it often makes possible early detection of determinacy, even if cuts are not included in the program. For example, consider the following simple predicate, which joins two lists together:
concat([], L, L).
concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).

If this predicate is called with an instantiated first argument, then the first argument indexing of SICStus Prolog will recognize that the call is determinate—only one of the two clauses for concat/3 can possibly apply. Thus, the Prolog system knows it does not have to store backtracking information for the call. This significantly reduces memory use and execution time.

Determinacy detection can also reduce the number of cuts in predicates. In the above example, if there was no indexing, then a cut would not strictly be needed in the first clause as long as the predicate was always to be called with the first argument instantiated. If the first clause matched, then the second clause could not possibly match; discovery of this fact, however, would be postponed until backtracking. The programmer might thus be tempted to use a cut in the first clause to signal determinacy and recover space for backtracking information as early as possible.

With indexing, if the example predicate is always called with its first argument instantiated, then backtracking information is never stored. This gives substantial performance improvements over using a cut rather than indexing to force determinacy. At the same time greater flexibility is maintained: the predicate can now be used in a nondeterminate fashion as well, as in

\[
| ?- \text{concat}(L1, L2, [a,b,c,d]).
\]

which will generate on backtracking all the possible partitions of the list \([a,b,c,d]\) on backtracking. If a cut had been used in the first clause, then this would not work.

For interpreted code, but not for compiled code, a filtering similar to indexing is done for all argument positions. The primary benefit of this filtering is that it makes it possible to detect determinacy in more cases. This filtering is currently not using hashing techniques, so it is not as performant as the first argument indexing.

We may improve indexing and other filtering techniques in future releases, which may decrease the number of choicepoints created.

### 9.6 Last Clause Determinacy Detection

Even if the determinacy detection made possible by indexing is unavailable to a predicate call, SICStus Prolog still can detect determinacy before determinate exit from the predicate. Space for backtracking information can thus be recovered as early as possible, reducing memory requirements and increasing performance. For instance, the predicate member/2 (found in the SICStus Prolog library) could be defined by:

\[
\begin{align*}
\text{member}(\text{Element}, \ [\text{Element}|\_]). \\
\text{member}(\text{Element}, \ [\_|\text{Rest}]) :&- \\
&\quad \text{member}(\text{Element}, \ \text{Rest}).
\end{align*}
\]
member/2 might be called with an instantiated first argument in order to check for membership of the argument in a list, which is passed as a second argument, as in

\[
| ?- \text{member}(4, [1,2,3,4]).
\]

The first arguments of both clauses of member/2 are variables, so first argument indexing cannot be used. However, determinacy can still be detected before determinate exit from the predicate. This is because on entry to the last clause of a nondeterminate predicate, a call becomes effectively determinate; it can tell that it has no more clauses to backtrack to. Thus, backtracking information is no longer needed, and its space can be reclaimed. In the example, each time a call fails to match the first clause and backtracks to the second (last) clause, backtracking information for the call is automatically deleted.

Because of last clause determinacy detection, a cut is never needed as the first subgoal in the last clause of a predicate. Backtracking information will have been deleted before a cut in the last clause is executed, so the cut will have no effect except to waste time.

Note that last clause determinacy detection is exploited by dynamic code as well as static code in SICStus Prolog.

### 9.7 The Determinacy Checker

**Please note:** the Determinacy Checker tool is mostly superseded by the analysis performed by the SICStus Prolog IDE, SPIDER (see Section 3.11 [SPIDER], page 29). SPIDER will analyze the source code fully automatically and will annotate the edited source code to highlight unwanted nondeterminism. The analysis performed by SPIDER is more precise than the analysis implemented by the determinism checker described below.

The determinacy checker can help you spot unwanted nondeterminacy in your programs. This tool examines your program source code and points out places where nondeterminacy may arise. It is not in general possible to find exactly which parts of a program will be nondeterminate without actually running the program, best with the execution profiler, which endeavors to find exactly those parts. However, this tool can find most unwanted nondeterminacy. Unintended nondeterminacy should be eradicated because:

1. it may give you wrong answers on backtracking
2. it may cause a lot of memory to be wasted

#### 9.7.1 Using the Determinacy Checker

There are two different ways to use the determinacy checker, either as a stand-alone tool, or during compilation. You may use it whichever way fits best with the way you work. Either way, it will discover the same nondeterminacy in your program.

The stand-alone determinacy checker is called `spdet`, and is run from the shell prompt, specifying the names of the Prolog source files you wish to check.

The determinacy checker can also be integrated into the compilation process, so that you receive warnings about unwanted nondeterminacy along with warnings about singleton variables or discontiguous clauses. To make this happen, simply insert the line
Once this line is added, every time that file is loaded, it will be checked for unwanted nondeterminacy.

9.7.2 Declaring Nondeterminacy

Some predicates are intended to be nondeterminate. By declaring intended nondeterminacy, you avoid warnings about predicates you intend to be nondeterminate. Equally importantly, you also inform the determinacy checker about nondeterminate predicates. It uses this information to identify unwanted nondeterminacy.

Nondeterminacy is declared by putting a declaration of the form

```prolog
:- name/arity is nondet.
```

using an \texttt{is/2}-declaration (see Section 10.19 \texttt{lib-is directives}, page 565), or the legacy form

```prolog
:- nondet name/arity.
```

in your source file. This is similar to a \texttt{dynamic} or \texttt{discontiguous} declaration. You may have multiple \texttt{is} or \texttt{nondet} declarations, and a single declaration may mention several predicates, separating them by commas.

Similarly, a predicate \texttt{P/N} may be classified as nondeterminate by the checker, whereas in reality it is determinate. This may happen e.g. if \texttt{P/N} calls a dynamic predicate that in reality never has more than one clause. To prevent false alarms arising from this, you can inform the checker about determinate predicates by declarations of the form:

```prolog
:- name/arity is det.
```

using an \texttt{is/2}-declaration (see Section 10.19 \texttt{lib-is directives}, page 565), or the legacy form

```prolog
:- det name/arity.
```

If you wish to include the legacy \texttt{det} and \texttt{nondet} declarations in your file and you plan to use the stand-alone determinacy checker, then you must include the line

```prolog
:- load_files(library(nondetdecl),
               [when(compile_time), if(changed)]).
```

near the top of each file that contains such declarations. If you instead use the recommended \texttt{is/2}-declarations, or the integrated determinacy checker, then you do not need (and should not have) this line.

9.7.3 Checker Output

The output of the determinacy checker is quite simple. For each clause containing unexpected nondeterminacy, a single line is printed showing the module, name, arity, and clause number (counting from 1). The form of the information is:
* Non-determinate: module:name/arity (clause number)

A second line for each nondeterminate clause indicates the cause of the nondeterminacy. The recognized causes are:

- The clause contains a disjunction that is not forced to be determinate with a cut or by ending the clause with a call to fail/0 or raise_exception/1.
- The clause calls a nondeterminate predicate. In this case the predicate is named.
- There is a later clause for the same predicate whose first argument has the same principal functor (or one of the two clauses has a variable for the first argument), and this clause does not contain a cut or end with a call to fail/0 or raise_exception/1. In this case, the clause number of the other clause is mentioned.
- If the predicate is multifile, then clause indexing is not considered sufficient to ensure determinacy. This is because other clauses may be added to the predicate in other files, so the determinacy checker cannot be sure it has seen all the clauses for the predicate. It is good practice to include a cut (or fail) in every clause of a multifile predicate.

The determinacy checker also occasionally prints warnings when declarations are made too late in the file or not at all. For example, if you include a dynamic, nondet, or discontiguous declaration for a predicate after some clauses for that predicate, or if you put a dynamic or nondet declaration for a predicate after a clause that includes a call to that predicate, then the determinacy checker may have missed some nondeterminacy in your program. The checker also detects undeclared discontiguous predicates, which may also have undetected nondeterminacy. Finally, the checker looks for goals in your program that indicate that predicates are dynamic; if no dynamic declaration for those predicates exists, then you will be warned.

These warnings take the following form:

```
! warning: predicate module:name/arity is property.
! Some nondeterminacy may have been missed.
! Add (or move) the directive
! :- property module:name/arity.
! near the top of this file.
```

**9.7.4 Example**

Here is an example file:

```
:- load_files(library(detcheck),
            [when(compile_time), if(changed)]).

parent(abe, rob).
parent(abe, sam).
parent(betty, rob).
parent(betty, sam).

is_parent(Parent) :- parent(Parent, _).
```
The determinacy checker notices that the first arguments of clauses 1 and 2 have the same principal functor, and similarly for clauses 3 and 4. It reports:

* Non-determinate: user:parent/2 (clause 1)
  * Indexing cannot distinguish this from clause 2.
* Non-determinate: user:parent/2 (clause 3)
  * Indexing cannot distinguish this from clause 4.

In fact, `parent/2` should be nondeterminate, so we should add the declaration

```prolog
:- parent/2 is nondet.
```

before the clauses for `parent/2`. If run again after modifying file, then the determinacy checker prints:

* Non-determinate: user:is_parent/1 (clause 1)
  * This clause calls user:parent/2, which may be nondeterminate.

It no longer complains about `parent/2` being nondeterminate, since this is declared. But now it notices that because `parent/2` is nondeterminate, then so is `is_parent/1`.

### 9.7.5 Options

When run from the command line, the determinacy checker has a few options to control its workings.

The `-r` option specifies that the checker should recursively check files in such a way that it finds nondeterminacy caused by calls to other nondeterminate predicates, whether they are declared so or not. Also, predicates that appear to be determinate will be treated as such, whether declared `nondet` or not. This option is quite useful when first running the checker on a file, as it will find all predicates that should be either made determinate or declared `nondet` at once. Without this option, each time a `nondet` declaration is added, the checker may find previously unnoticed nondeterminacy.

For example, if the original example above, without any `nondet` declarations, were checked with the `-r` option, then the output would be:

* Non-determinate: user:parent/2 (clause 1)
  * Indexing cannot distinguish this from clause 2.
* Non-determinate: user:parent/2 (clause 3)
  * Indexing cannot distinguish this from clause 4.
* Non-determinate: user:is_parent/1 (clause 1)
  * Calls nondet predicate user:parent/2.

The `-d` option causes the tool to print out the needed `nondet` declarations. These can be readily pasted into the source files. Note that it only prints the `nondet` declarations that are not already present in the files. However, these declarations should not be pasted into your code without each one first being checked to see if the reported nondeterminacy is intended.
The -D option is like -d, except that it prints out all nondet declarations that should appear, whether they are already in the file or not. This is useful if you prefer to replace all old nondet declarations with new ones.

Your code will probably rely on operator declarations and possibly term expansion. The determinacy checker handles this in the following way: you must supply an initialization file, using the -i ifile option. spdet will execute any operator declaration it encounters.

9.7.6 What is Detected

As mentioned earlier, it is not in general possible to find exactly which places in a program will lead to nondeterminacy. The determinacy checker gives predicates the benefit of the doubt: when it is possible that a predicate will be determinate, it will not be reported. The checker will only report places in your program that will be nondeterminate regardless of which arguments are bound. Despite this, the checker catches most unwanted nondeterminacy in practice.

The determinacy checker looks for the following sources of nondeterminacy:

- Multiple clauses that cannot be distinguished by the principal functor of the first arguments, and are not made determinate with an explicit cut, fail/0, false/0, or raise_exception/1. First argument indexing is not considered for multifile predicates, because another file may have a clause for this predicate with the same principal functor of its first argument.
- A clause with a disjunction not forced to be determinate by a cut, fail/0, false/0, or raise_exception/1 in each arm of the disjunction but the last, or where the whole disjunction is followed by a cut, fail/0, false/0, or raise_exception/1.
- A clause that calls something known to be nondeterminate, other than when it is followed by a cut, fail/0, false/0, or raise_exception/1, or where it appears in the condition of an if-then-else construct. Known nondeterminate predicates include hooks and those declared nondeterminate or dynamic (since they can be modified, dynamic predicates are assumed to be nondeterminate), plus the following built-in predicates:
  - absolute_file_name/3, when the options list contains solutions(all).
  - atom_concat/3, when the first two arguments are variables not appearing earlier in the clause (including the clause head).
  - bagof/3, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
  - clause/[2,3].
  - current_op/3, when any argument contains any variables not appearing earlier in the clause (including the clause head).
  - current_key/2, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
  - current_predicate/2, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
  - length/2, when both arguments are variables not appearing earlier in the clause (including the clause head).
- `predicate_property/2`, when either argument contains any variables not appearing earlier in the clause (including the clause head).
- `recorded/3`.
- `repeat/0`.
- `retract/1`.
- `setof/3`, when the second argument contains any variables not appearing earlier in the clause (including the clause head).
- `source_file/[1,2]` when the last argument contains any variables not appearing earlier in the clause (including the clause head).
- `sub_atom/5`, when at least two of the second, fourth and fifth arguments are variables not appearing earlier in the clause (including the clause head).

### 9.8 Last Call Optimization

Another important efficiency feature of SICStus Prolog is last call optimization. This is a space optimization technique, which applies when a predicate is determinate at the point where it is about to call the last goal in the body of a clause. For example,

```prolog
% for(Int, Lower, Upper)
% Lower and Upper should be integers such that Lower =< Upper.
% Int should be uninstantiated; it will be bound successively on
% backtracking to Lower, Lower+1, ... Upper.

for(Int, Int, _Upper).
for(Int, Lower, Upper) :-
  Lower < Upper,
  Next is Lower + 1,
  for(Int, Next, Upper).
```

This predicate is determinate at the point where the recursive call is about to be made, since this is the last clause and the preceding goals `<>/2` and `is/2` are determinate. Thus last call optimization can be applied; effectively, the stack space being used for the current predicate call is reclaimed before the recursive call is made. This means that this predicate uses only a constant amount of space, no matter how deep the recursion.

#### 9.8.1 Accumulating Parameters

To take best advantage of this feature, make sure that goals in recursive predicates are determinate, and whenever possible put the recursive call at the end of the predicate.

This is not always possible, but often can be done through the use of accumulating parameters. An accumulating parameter is an added argument to a predicate that builds up the result as computation proceeds. For example, in our factorial example, the last goal in the body of the recursive case is `is/2`, not the recursive call to `fac/2`. 
fac(N, X) :-
    (   N > 0 ->
        N1 is N - 1,
        fac(N1, Y),
        X is N * Y
    ;   N =:= 0 ->
        X = 1
    ).

This can be corrected by adding another argument to `fac/2` to accumulate the factorial.

fac(N, X) :- fac(N, 1, X).

% fac(+N, +M, -X)
% X is M * the factorial of N.

fac(N, M, X) :-
    (   N > 0 ->
        N1 is N - 1,
        M1 is N * M,
        fac(N1, M1, X)
    ;   N =:= 0 ->
        X = M
    ).

Here, we do the multiplication before calling `fac/3` recursively. Note that we supply the base case, 1, at the start of the computation, and that we are multiplying by decreasing numbers. In the earlier version, `fac/2`, we multiply after the recursive call, and so we multiply by increasing numbers. Effectively, the new version builds the result backwards. This is correct because multiplication is associative.

### 9.8.2 Accumulating Lists

This technique becomes much more important when extended to lists, as in this case it can save much building of unneeded lists through unnecessary calls to `append` sublists together. For example, the naive way to reverse a list is:

```prolog
nreverse([], []).
nreverse([H|T], L) :-
    nreverse(T, L1),
    append(L1, [H], L).
```

This is very wasteful, since each call to `append/3` copies the initial part of the list, and adds one element to it. Fortunately, this can be very easily rewritten to use an accumulating parameter:
reverse(L1, L2) :- reverse(L1, [], L2).

% reverse(+X, +Y, -Z)
% Z is X reversed, followed by Y
reverse([], Z, Z).
reverse([H|T], L0, L) :-
    reverse(T, [H|L0], L).

This version of reverse is many times faster than the naive version, and uses much less memory. The key to understanding the behavior of this predicate is the observation made earlier: using an accumulating parameter, we build the result backwards.

Do Not let this confuse you. Building a list forward is easy. For example, a predicate returning a list L of consecutive numbers from 1 to N could be written in two different ways: counting up and collecting the resulting list forward, or counting down and accumulating the result backward.

iota1(N, L) :- iota1(1, N, L).
iota1(N, Max, L) :-
    ( N > Max ->
      L = []
    ;   N1 is N+1,
      L = [N|L1],
      iota1(N1, Max, L1)
    ).

or,

iota2(N, L) :- iota2(N, [], L).
iota2(N, L0, L) :-
    ( N <= 0 ->
      L = L0
    ;   N1 is N-1,
      iota2(N1, [N|L0], L)
    ).

Both versions generate the same results, and neither waste any space. The second version is slightly faster. Choose whichever approach you prefer.

9.9 Building and Dismantling Terms

The built-in predicate (=..)/2 is a clear way of building terms and taking them apart. However, it is almost never the most efficient way. functor/3 and arg/3 are generally much more efficient, though less direct. The best blend of efficiency and clarity is to write a clearly-named predicate that implements the desired operation and to use functor/3 and arg/3 in that predicate.
Here is an actual example. The task is to reimplement the built-in predicate (==)/2. The first variant uses (=..)/2 (this symbol is pronounced “univ” for historical reasons). Some Prolog textbooks recommend code similar to this.

```prolog
ident_univ(X, Y) :-
  var(X), !, var(Y), !,
  samevar(X, Y). % If X is a variable, so must Y be, and they must be the same.
ident_univ(X, Y) :- % If X is not a variable, nonvar(Y),
  X =.. [F|L], % neither may Y be;
  Y =.. [F|M], % they must have the same function symbol F
  ident_list(L, M). % and identical arguments

ident_list([], []).
ident_list([H1|T1], [H2|T2]) :-
  ident_univ(H1, H2),
  ident_list(T1, T2).

samevar(29, Y) :- % If binding X to 29 leaves Y unbound,
  var(Y), !, fail. % they were not the same variable.
samevar(_, _). % Otherwise they were.
```

This code performs the function intended; however, every time it touches a non-variable term of arity \( N \), it constructs a list with \( N+1 \) elements, and if the two terms are identical, then these lists are reclaimed only when backtracked over or garbage collected.

Better code uses `functor/3` and `arg/3`.
ident_farg(X, Y) :-
    ( var(X) -> % If X is a variable,
      var(Y), % so must Y be, and
      samevar(X, Y) % they must be the same;
    ;
      nonvar(Y), % otherwise Y must be nonvar
      functor(X, F, N), % The principal functors of X
      functor(Y, F, N), % and Y must be identical,
      ident_farg(N, X, Y) % including the last N args.
    ).

ident_farg(0, _, _) :- !.
ident_farg(N, X, Y) :- % The last N arguments are
    arg(N, X, Xn), % identical
    arg(N, Y, Yn), % if the Nth arguments
    ident_farg(Xn, Yn), % are identical,
    M is N-1, % and the last N-1 arguments
    ident_farg(M, X, Y). % are also identical.

This approach to walking through terms using functor/3 and arg/3 avoids the construction of useless lists.

The pattern shown in the example, in which a predicate of arity K calls an auxiliary predicate of the same name of arity K+1 (the additional argument denoting the number of items remaining to process), is very common. It is not necessary to use the same name for this auxiliary predicate, but this convention is generally less prone to confusion.

In order to simply find out the principal function symbol of a term, use

| ?- the_term_is(Term),
|   functor(Term, FunctionSymbol, _).

The use of (=..)/2, as in

| ?- the_term_is(Term),
|   Term =.. [FunctionSymbol|_].

is wasteful, and should generally be avoided. The same remark applies if the arity of a term is desired.

 (=..)/2 should not be used to locate a particular argument of some term. For example, instead of

    Term =.. [\_F,\_,\_ArgTwo|\_]

you should write

    arg(2, Term, ArgTwo)
It is generally easier to get the explicit number “2” right than to write the correct number of anonymous variables in the call to (=..)/2. Other people reading the program will find the call to arg/3 a much clearer expression of the program’s intent. The program will also be more efficient. Even if several arguments of a term must be located, it is clearer and more efficient to write

\[
\begin{align*}
\text{arg}(1, \text{Term}, \text{First}), \\
\text{arg}(3, \text{Term}, \text{Third}), \\
\text{arg}(4, \text{Term}, \text{Fourth})
\end{align*}
\]

than to write

\[
\text{Term} = .. \left[_, \text{First}, _, \text{Third}, \text{Fourth} | _\right]
\]

Finally, (=..)/2 should not be used when the functor of the term to be operated on is known (that is, when both the function symbol and the arity are known). For example, to make a new term with the same function symbol and first arguments as another term, but one additional argument, the obvious solution might seem to be to write something like the following:

\[
\begin{align*}
\text{add_date}(\text{OldItem}, \text{Date}, \text{NewItem}) : - \\
\text{OldItem} = .. \left[\text{item}, \text{Type}, \text{Ship}, \text{Serial}\right], \\
\text{NewItem} = .. \left[\text{item}, \text{Type}, \text{Ship}, \text{Serial}, \text{Date}\right].
\end{align*}
\]

However, this could be expressed more clearly and more efficiently as

\[
\begin{align*}
\text{add_date}(\text{OldItem}, \text{Date}, \text{NewItem}) : - \\
\text{OldItem} = \text{item}(\text{Type}, \text{Ship}, \text{Serial}), \\
\text{NewItem} = \text{item}(\text{Type}, \text{Ship}, \text{Serial}, \text{Date}).
\end{align*}
\]

or even

\[
\text{add_date}(\text{item}(\text{Type}, \text{Ship}, \text{Serial}), \\
\text{Date}, \\
\text{item}(\text{Type}, \text{Ship}, \text{Serial}, \text{Date}) \\
).
\]

9.10 Conditionals and Disjunction

There is an efficiency advantage in using conditionals whose test part consists only of arithmetic comparisons or type tests. Consider the following alternative definitions of the predicate type_of_character/2. In the first definition, four clauses are used to group characters on the basis of arithmetic comparisons.
type_of_character(Ch, Type) :-
    Ch >= "a", Ch =< "z",
    !,
    Type = lowercase.

type_of_character(Ch, Type) :-
    Ch >= "A", Ch =< "Z",
    !,
    Type = uppercase.

type_of_character(Ch, Type) :-
    Ch >= "0", Ch =< "9",
    !,
    Type = digit.

type_of_character(_, Type) :-
    Type = other.

In the second definition, a single clause with a conditional is used. The compiler generates equivalent, optimized code for both versions.

    type_of_character(Ch, Type) :-
        ( Ch >= "a", Ch =< "z" ->
            Type = lowercase
        ; Ch >= "A", Ch =< "Z" ->
            Type = uppercase
        ; Ch >= "0", Ch =< "9" ->
            Type = digit
        ; otherwise ->
            Type = other
    ).

Following is a list of built-in predicates that are compiled efficiently in conditionals:

- atom/1
- atomic/1
- callable/1
- compound/1
- db_reference/1
- float/1
- ground/1
- integer/1
- nonvar/1
- mutable/1
- number/1
- simple/1
- var/1
- </2
This optimization is actually somewhat more general than what is described above. A sequence of guarded clauses:

\[
\begin{align*}
\text{Head}_1 & :\!\!- \text{Guard}_1, \!, \text{Body}_1. \\
\text{...} \\
\text{Head}_m & :\!\!- \text{Guard}_m, \!, \text{Body}_m. \\
\text{Head}_n & :\!\!- \text{Body}_m.
\end{align*}
\]

is eligible for the same optimization, provided that the arguments of the clause heads are all unique variables and that the “guards” are simple tests as listed above.

### 9.11 Programming Examples

The rest of this chapter contains a number of simple examples of Prolog programming, illustrating some of the techniques described above.

#### 9.11.1 Simple List Processing

The goal \texttt{concatenate(L1,L2,L3)} is true if list \( L3 \) consists of the elements of list \( L1 \) concatenated with the elements of list \( L2 \). The goal \texttt{member(X,L)} is true if \( X \) is one of the elements of list \( L \). The goal \texttt{reverse(L1,L2)} is true if list \( L2 \) consists of the elements of list \( L1 \) in reverse order.

\[
\begin{align*}
\text{concatenate}([], L, L). \\
\text{concatenate}([X|L1], L2, [X|L3]) & :\!\!- \text{concatenate}(L1, L2, L3). \\
\text{member}(X, [X|_]). \\
\text{member}(X, [_|L]) & :\!\!- \text{member}(X, L). \\
\text{reverse}(L, L1) & :\!\!- \text{reverse_concatenate}(L, [], L1). \\
\text{reverse_concatenate}([], L, L). \\
\text{reverse_concatenate}([X|L1], L2, L3) & :\!\!- \text{reverse_concatenate}(L1, [X|L2], L3).
\end{align*}
\]
9.11.2 Family Example (descendants)

The goal \texttt{descendant(X,Y)} is true if \( Y \) is a descendant of \( X \).

\[
\text{descendant}(X, Y) : \text{offspring}(X, Y).
\]
\[
\text{descendant}(X, Z) : \text{offspring}(X, Y), \text{descendant}(Y, Z).
\]

\[
\text{offspring(abraham, ishmael)}.
\]
\[
\text{offspring(abraham, isaac)}.
\]
\[
\text{offspring(isaac, esau)}.
\]
\[
\text{offspring(isaac, jacob)}.
\]

If for example the query

\[
| \ ?- \text{descendant(abraham, X)}.
\]

is executed, then Prolog’s backtracking results in different descendants of Abraham being returned as successive instances of the variable \( X \), i.e.

\[
X = \text{ismael}
\]
\[
X = \text{isaac}
\]
\[
X = \text{esau}
\]
\[
X = \text{jacob}
\]

9.11.3 Association List Primitives

These predicates implement “association list” primitives. They use a binary tree representation. Thus the time complexity for these predicates is \( O(lg \, N) \), where \( N \) is the number of keys. These predicates also illustrate the use of \texttt{compare/3} for case analysis.

The goal \texttt{get_assoc(Key, Assoc, Value)} is true when \( Key \) is identical to one of the keys in \( Assoc \), and \( Value \) unifies with the associated value.

\[
\text{get_assoc(Key, t(K,V,L,R), Val)} :\text{-}
\]
\[
\text{compare(Rel, Key, K),}
\]
\[
\text{get_assoc(Rel, Key, V, L, R, Val)}.
\]

\[
\text{get_assoc(=, _, Val, _, _, Val)}.
\]
\[
\text{get_assoc(<, Key, _, Tree, _, Val)} :\text{-}
\]
\[
\text{get_assoc(Key, Tree, Val)}.
\]
\[
\text{get_assoc(>, Key, _, _, Tree, Val)} :\text{-}
\]
\[
\text{get_assoc(Key, Tree, Val)}.
\]

9.11.4 Differentiation

The goal \texttt{d(E1, X, E2)} is true if expression \( E2 \) is a possible form for the derivative of expression \( E1 \) with respect to \( X \).
\[ \begin{align*}
    d(X, X, D) & : - \text{atomic}(X), !, D = 1. \\
    d(C, X, D) & : - \text{atomic}(C), !, D = 0. \\
    d(U+V, X, DU+DV) & : - d(U, X, DU), d(V, X, DV). \\
    d(U-V, X, DU-DV) & : - d(U, X, DU), d(V, X, DV). \\
    d(U*V, X, DU*V+U*DV) & : - d(U, X, DU), d(V, X, DV). \\
    d(U**N, X, N*U**N1*DU) & : \text{integer}(N), N1 \text{ is } N-1, d(U, X, DU). \\
    d(-U, X, -DU) & : - d(U, X, DU). 
\end{align*} \]

### 9.11.5 Use of Meta-Logical Predicates

This example illustrates the use of the meta-logical predicates \texttt{var/1}, \texttt{arg/3}, and \texttt{functor/3}. The goal \texttt{variables(Term, L, [])} instantiates variable \( L \) to a list of all the variable occurrences in \( \text{Term} \). E.g.:

\[
\begin{align*}
    | ?- \text{variables}(d(U*V, X, DU*V+U*DV), L, []). \\
    \text{variables}(X, [X|L0], L) & : \text{var}(X), !, L = L0. \\
    \text{variables}(T, L0, L) & : \\
    \% & \text{nonvar}(T), \\
    \text{functor}(T, \_, A), \\
    \text{variables}(0, A, T, L0, L). \\
    \text{variables}(A, A, \_, L0, L) & : !, L = L0. \\
    \text{variables}(A0, A, T, L0, L) & : \\
    \% & A0 < A, \\
    A1 & \text{is } A0 + 1, \\
    \text{arg}(A1, T, X), \\
    \text{variables}(X, L0, L1), \\
    \text{variables}(A1, A, T, L1, L). 
\end{align*} \]

### 9.11.6 Prolog in Prolog

This example shows how simple it is to write a Prolog interpreter in Prolog, and illustrates the use of a variable goal. In this mini-interpreter, goals and clauses are represented as ordinary Prolog data structures (i.e. terms). Terms representing clauses are specified using the predicate \texttt{my_clause/1}, e.g.:

\[
\text{my_clause}( (\text{grandparent}(X, Z) : - \text{parent}(X, Y), \text{parent}(Y, Z)) ).
\]

A unit clause will be represented by a term such as

\[
\text{my_clause}( (\text{parent}(\text{john}, \text{mary}) : - \text{true}) ).
\]

The mini-interpreter consists of three clauses:

\[
\begin{align*}
    \text{execute}(P, Q) & : - !, \text{execute}(P), \text{execute}(Q). \\
    \text{execute}(P) & : - \text{predicate_property}(P, \text{built_in}), !, P. \\
    \text{execute}(P) & : - \text{my_clause}((P : - Q)), \text{execute}(Q).
\end{align*} \]
The second clause enables the mini-interpreter to cope with calls to ordinary Prolog predicates, e.g., built-in predicates. The mini-interpreter needs to be extended to cope with the other control structures, i.e., \( T \), \( (P;Q) \), \( (P\rightarrow Q) \), \( (P\rightarrow Q;R) \), \( (\neg P) \), and \( \text{if}(P,Q,R) \).

### 9.11.7 Translating English Sentences into Logic Formulae

The following example of a definite clause grammar defines in a formal way the traditional mapping of simple English sentences into formulae of classical logic. By way of illustration, if the sentence

Every man that lives loves a woman.

is parsed as a sentence by the call

\[
\text{\texttt{| ?- phrase(sentence(P), [every,man,that,lives,loves,a,woman]).}}
\]

then \( P \) will get instantiated to

\[
\text{\texttt{all(X):(man(X)&lives(X) => exists(Y):(woman(Y)&loves(X,Y)))}}
\]

where \( :, \& \) and \( => \) are infix operators defined by

\[
\text{\texttt{:- op(900, xfx, =>).}} \quad \text{\texttt{:- op(800, xfy, \&).}} \quad \text{\texttt{:- op(550, xfy, :). /* predefined */}}
\]

The grammar follows:

\[
\text{sentence(P) } \rightarrow \text{ noun_phrase(X, P1, P), verb_phrase(X, P1).}
\]

\[
\text{noun_phrase(X, P1, P) } \rightarrow \text{ determiner(X, P2, P1, P), noun(X, P3), rel_clause(X, P3, P2).}
\]

\[
\text{noun_phrase(X, P, P) } \rightarrow \text{ name(X).}
\]

\[
\text{verb_phrase(X, P) } \rightarrow \text{ trans_verb(X, Y, P1), noun_phrase(Y, P1, P).}
\]

\[
\text{verb_phrase(X, P) } \rightarrow \text{ intrans_verb(X, P).}
\]

\[
\text{rel_clause(X, P1, P1&P2) } \rightarrow \text{ [that], verb_phrase(X, P2).}
\]

\[
\text{rel_clause(_, P, P) } \rightarrow \text{ []}. \quad \text{determiner(X, P1, P2, all(X):(P1=>P2)) } \rightarrow \text{ [every].}
\]

\[
\text{determiner(X, P1, P2, exists(X):(P1&P2)) } \rightarrow \text{ [a].}
\]

\[
\text{noun(X, man(X)) } \rightarrow \text{ [man].}
\]

\[
\text{noun(X, woman(X)) } \rightarrow \text{ [woman].}
\]

\[
\text{name(john) } \rightarrow \text{ [john].}
\]

\[
\text{trans_verb(X, Y, loves(X,Y)) } \rightarrow \text{ [loves].}
\]

\[
\text{intrans_verb(X, lives(X)) } \rightarrow \text{ [lives].}
\]
9.12 The Cross-Referencer

9.12.1 Introduction

Please note: the Cross-References tool is mostly superseeded by the SICStus Prolog IDE, SPIDER (see Section 3.11 [SPIDER], page 29). SPIDER will analyze the source code fully automatically and will annotate the edited source code to highlight unused and undefined predicates. The cross-reference analysis performed by SPIDER is more precise than the analysis implemented by the cross-referencer described below.

The main purpose of the cross-referencer, spxref, is to find undefined predicates and unreachable code. To this end, it begins by looking for initializations, hooks and public directives to start tracing the reachable code from. If an entire application is being checked, then it also traces from user:runtime_entry/1. If individual module files are being checked, then it also traces from their export lists.

A second function of spxref is to aid in the formation of module statements. spxref can list all of the required module/2 and use_module/2 statements by file.

The cross-referencer is called spxref, and is run from the shell prompt, specifying the names of the Prolog source files you wish to check.

9.12.2 Practice and Experience

Your code will probably rely on operator declarations and possibly term expansion. The cross-referencer handles this in the following way: you must supply an initialization file, using the -i ifile option. spxref will execute any operator declaration it encounters.

Supply meta-predicate declarations for your meta-predicates. Otherwise, the cross-referencer will not follow the meta-predicates’ arguments. Be sure the cross-referencer encounters the meta-predicate declarations before it encounters calls to the declared predicates.

The cross-referencer traces from initializations, hooks, predicates declared public, and optionally from user:runtime_entry/1 and module declarations. The way it handles meta-predicates requires that your application load its module files before its non-module files.

This cross-referencer was written in order to tear out the copious dead code from the application that the author became responsible for. If you are doing such a thing, then the cross-referencer is an invaluable tool. Be sure to save the output from the first run that you get from the cross referencer: this is very useful resource to help you find things that you’ve accidentally ripped out and that you really needed after all.

There are situations where the cross-referencer does not follow certain predicates. This can happen if the predicate name is constructed on the fly, or if it is retrieved from the database. In this case, add public declarations for these. Alternatively, you could create term expansions that are peculiar to the cross-referencer.
10 The Prolog Library

The Prolog library comprises a number of packages that are thought to be useful in a number of applications. Note that the predicates in the Prolog library are not built-in predicates. One has to explicitly load each package to get access to its predicates.

As opposed to built-in predicates, predicates exported by library modules generally do not check their arguments, although some do to a lesser or greater extent. Input arguments that are lists are usually supposed to be proper lists, i.e., not end with an unbound variable. Input arguments that are trees are usually not supposed to have uninstantiated leaves, and so on.

To load a library package Package, you will normally enter a query:

```
| ?- use_module(library(Package)).
```

A library package normally consists of one or more hidden (see Section 4.11 [ref-mod], page 161) modules. There are packages for many different purposes, such as providing an abstract data type, database access, constraint solvers, I/O, interoperability with other software, file and operating system access, object-oriented programming, a unit-test framework, random numbers, and statistics.

Among the first category, several packages provide array-like functionality, because arrays are not part of the Prolog language. If you are about to choose an array representation, the following comparison of their key properties may be helpful. Prolog lists are included in the comparison. In the table, read cost is the cost of looking up an element given a key; update cost is the cost of replacing an element given a key, creating a new array-like term; sparse means that the set of keys does not have to be known up front:

<table>
<thead>
<tr>
<th>representation</th>
<th>read cost</th>
<th>update cost</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prolog list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>no</td>
</tr>
<tr>
<td>library(assoc)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>yes</td>
</tr>
<tr>
<td>library(avl)</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>yes</td>
</tr>
<tr>
<td>library(logarr)</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>yes</td>
</tr>
<tr>
<td>library(mutarray)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>no</td>
</tr>
<tr>
<td>library(trees)</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>no</td>
</tr>
</tbody>
</table>

Please note: library(mutarray) updates run in $O(1)$ time for all indices in the best case, and in $O(n)$ time otherwise.

Following is the list of provided packages, in alphabetic order:

aggregate (see Section 10.1 [lib-aggregate], page 385) provides an aggregation operator for data-base-style queries.
assoc (see Section 10.2 [lib-assoc], page 389)
uses unbalanced binary trees trees to implement dictionaries, i.e. extendible mappings from terms to terms. Can be used as sparse arrays with linear read and update cost.

atts (see Section 10.3 [lib-atts], page 391)
provides a means of associating with variables arbitrary attributes, i.e. named properties that can be used as storage locations as well as hooks into Prolog’s unification.

avl (see Section 10.4 [lib-avl], page 398)
uses AVL trees to implement dictionaries, i.e. extendible finite mappings from terms to terms. Can be used as sparse arrays with logarithmic read and update cost.

bags (see Section 10.5 [lib-bags], page 401)
defines operations on bags, or multisets

bdb (see Section 10.6 [lib-bdb], page 404)
provides an interface to Berkeley DB, for storage and retrieval of terms on disk files with user-defined multiple indexing.

between (see Section 10.7 [lib-between], page 412)
provides some means of generating integers.

chr (see Section 10.8 [lib-chr], page 413)
provides Constraint Handling Rules

clpb (see Section 10.9 [lib-clpb], page 422)
since release 4.0.7, unsupported
provides constraint solving over Booleans

clpfd (see Section 10.10 [lib-clpfd], page 426)
provides constraint solving over Finite (Integer) Domains

clpq (see Section 10.11 [lib-clpq], page 499)
unsupported
provides constraint solving over Q (Rationals) or R (Reals)

clpqr (see Section 10.11 [lib-clpq], page 499)
unsupported
provides constraint solving over Q (Rationals) or R (Reals)

codesio (see Section 10.12 [lib-codesio], page 524)
defines I/O predicates that read from, or write to, a code list.

csv (see Section 10.13 [lib-csv], page 525)
defines I/O predicates that read from, or write to, comma-separated values (CSV) files and strings.

comclient (see Section 10.14 [lib-comclient], page 527)
An interface to Microsoft COM automaton objects.

fdbg (see Section 10.15 [lib-fdbg], page 532)
provides a debugger for finite domain constraint programs

file_systems (see Section 10.16 [lib-file_systems], page 556)
accesses files and directories.

gauge (see Section 10.17 [lib-gauge], page 561)
A profiling tool for Prolog programs with a graphical interface based on tcltk.
heaps (see Section 10.18 [lib-heaps], page 563)
  implements binary heaps, the main application of which are priority queues.

is_directives (see Section 10.19 [lib-is_directives], page 565)
  Access information about predicates that have been declared with is/2 directives.

jasper (see Section 10.20 [lib-jasper], page 572)
  Access Prolog from Java.

json (see Section 10.21 [lib-json], page 601)
  defines I/O predicates that read and write using the JSON serialization format.

jsonrpc (see Section 10.22 [lib-jsonrpc], page 604)
  since release 4.7.0
  is a collection of examples for using JSON to communicate between SICStus and other programming languages.

linda/client (see Section 10.23 [lib-linda], page 609)
  linda/server (see Section 10.23 [lib-linda], page 609)
  provides an implementation of the Linda concept for process communication.

lists (see Section 10.24 [lib-lists], page 614)
  provides basic operations on lists.

lmdb (see Section 10.25 [lib-lmdb], page 629)
  provides an interface to LMDB, for storage and retrieval of terms on disk files with user-defined multiple indexing.

logarr (see Section 10.26 [lib-logarr], page 636)
  provides a binary tree implementation of dense arrays with logarithmic read and update cost.

mutarray (see Section 10.27 [lib-mutarray], page 637)
  since release 4.7
  provides an implementation of mutable, dense arrays with constant read cost and update cost for all indices in the best case.

mutdict (see Section 10.28 [lib-mutdict], page 638)
  since release 4.7
  provides an implementation of mutable, extendible dictionaries, i.e. mutable extendible finite mappings from terms to terms, with constant lookup and update cost for all keys in the best case.

objects (see Section 10.29 [lib-objects], page 639)
  provides a package for object-oriented programming, and can be regarded as a high-level alternative to library(structs).

odbc (see Section 10.30 [lib-odbc], page 702)
  since release 4.1
  provides an interface to an ODBC database driver.

ordsets (see Section 10.31 [lib-ordsets], page 712)
  defines operations on sets represented as lists with the elements ordered in Prolog standard order.

pillow (see Section 10.32 [lib-pillow], page 714)
  The PiLLoW Web Programming Library,
plunit (see Section 10.33 [lib-plunit], page 714) since release 4.1.3
A Prolog unit-test framework.

process (see Section 10.34 [lib-process], page 722)
provides process creation primitives.

prologbeans (see Section 10.35 [lib-prologbeans], page 731)
Access Prolog from Java and .NET.

queues (see Section 10.36 [lib-queues], page 745)
defines operations on queues (FIFO stores of information).

random (see Section 10.37 [lib-random], page 748)
provides a random number generator.

rem (see Section 10.38 [lib-rem], page 749)
provides Rem’s algorithm for maintaining equivalence classes.

samsort (see Section 10.39 [lib-samsort], page 750)
provides generic stable sorting and merging.

sets (see Section 10.40 [lib-sets], page 750)
defines operations on sets represented as lists with the elements unordered.

sockets (see Section 10.41 [lib-sockets], page 754)
provides an interface to sockets.

statistics (see Section 10.42 [lib-statistics], page 757) since release 4.3.4
provides commonly used sample and population statistics functions.

structs (see Section 10.43 [lib-structs], page 759)
provides access to C data structures, and can be regarded as a low-level alternative to library(objects).

system (see Section 10.44 [lib-system], page 768)
provides access to operating system services.

tcltk (see Section 10.45 [lib-tcltk], page 769)
An interface to the Tcl/Tk language and toolkit.

terms (see Section 10.46 [lib-terms], page 862)
provides a number of operations on terms.

timeout (see Section 10.47 [lib-timeout], page 867)
Meta-call with limit on execution time.

trees (see Section 10.48 [lib-trees], page 868)
provides a binary tree implementation of dense arrays with logarithmic read and update cost.

types (see Section 10.49 [lib-types], page 869)
Provides type checking.

ugraphs (see Section 10.50 [lib-ugraphs], page 871)
Provides an implementation of directed and undirected graphs with unlabeled edges.
**varnumbers** (see Section 10.51 [lib-varnumbers], page 874)
An inverse of **numbervars/3**.

**wgraphs** (see Section 10.52 [lib-wgraphs], page 875)
provides an implementation of directed and undirected graphs where each edge has an integral weight.

**xml** (see Section 10.53 [lib-xml], page 878)
provides an XML parser.

**zinc** (see Section 10.54 [lib-zinc], page 879)
since release 4.0.5
provides an interpreter for FlatZinc programs

For the purpose of migrating code from release 3, the following deprecated library modules are also provided. For documentation, please see the release 3 documentation for the corresponding library module with the trailing ‘3’ removed from its name:

arrays3
assoc3
lists3
queues3
random3
system3

### 10.1 An Aggregation Operator for Data-Base-Style Queries—library(aggregate)

Data base query languages usually provide so-called "aggregation" operations. Given a relation, aggregation specifies

- a column of the relation
- an operation, one of \{sum,max,min,ave,var\} or more

One might, for example, ask

```
PRINT DEPT, SUM(AREA) WHERE OFFICE(_ID, DEPT, AREA, _OCCUPANT)
```

and get a table of \langle Department, TotalArea \rangle pairs. The Prolog equivalent of this might be

```
department_area(Dept, TotalArea) :-
    aggregate(sum(Area),
              I^O^office(I, Dept, Area, O), TotalArea).
```

where Area is the column and **sum(_)** is the aggregation operator. We can also ask who has the smallest office in each department:
smallest_office(Dept, Occupant) :-
    aggregate(min(Area),
        I^office(I,Dept,Area,O), MinArea),
    office(_, Dept, MinArea, Occupant).

This module provides an aggregation operator in Prolog:

    aggregate(Template, Generator, Results)

where:

- **Template** is operator(expression) or constructor(arg,...,arg)
- each arg is operator(expression)
- **operator** is sum | min | max {for now}
- **expression** is an arithmetic expression

**Results** is unified with a form of the same structure as **Template**.

Things like mean and standard deviation can be calculated from sums, e.g. to find the average population of countries (defined as "if you sampled people at random, what would be the mean size of their answers to the question 'what is the population of your country?'?"), we could do

    |- aggregate(x(sum(Pop),sum(Pop*Pop)),
                Country^population(Country,Pop),
                x(People,PeopleTimesPops)),
           AveragePop is PeopleTimesPops/People.

Note that according to this definition, **aggregate/3** FAILS if there are no solutions. For **max(_)**, **min(_)**, and many other operations (such as **mean(_)**) this is the only sensible definition (which is why **bagof/3** works that way). Even if **bagof/3** yielded an empty list, **aggregate/3** would still fail.

Concerning the minimum and maximum, it is convenient at times to know Which term had the minimum or maximum value. So we write

    min(Expression, Term)
    max(Expression, Term)

and in the constructed term we will have

    min(MinimumValue, TermForThatValue)
    max(MaximumValue, TermForThatValue)

So another way of asking who has the smallest office is

    smallest_office(Dept, Occupant) :-
        aggregate(min(Area,0),
            I^office(I,Dept,Area,O), min(_,Occupant)).
Consider queries like

\[ \text{aggregate}(\text{sum}(\text{Pay}), \text{Person} \cdot \text{pay}(\text{Person}, \text{Pay}), \text{TotalPay}) \]

where for some reason \( \text{pay}/2 \) might have multiple solutions. (For example, someone might be listed in two departments.) We need a way of saying "treat identical instances of the Template as a single instance, UNLESS they correspond to different instances of a Discriminator." That is what

\[ \text{aggregate}(\text{Template}, \text{Discriminator}, \text{Generator}, \text{Results}) \]

does.

Operations available:

- `count`: \( \text{sum}(1) \)
- `sum(E)`: sum of values of \( E \)
- `min(E)`: minimum of values of \( E \)
- `min(E, X)`: \( \text{min}(E) \) with corresponding instance of \( X \)
- `max(E)`: maximum of values of \( E \)
- `max(E, X)`: \( \text{max}(E) \) with corresponding instance of \( X \)
- `set(X)`: ordered set of instances of \( X \)
- `bag(X)`: list of instances of \( X \) in generated order.

\[ \text{bagof}(X, G, B) :- \text{aggregate}(\text{bag}(X), G, L). \]
\[ \text{setof}(X, G, B) :- \text{aggregate}(\text{set}(X), X, G, L). \]

Exported predicates:

- `forall(:Generator, :Goal)`
  succeeds when \( \text{Goal} \) is provable for each true instance of \( \text{Generator} \). Note that there is a sort of double negation going on in here (it is in effect a nested pair of failure-driven loops), so it will never bind any of the variables which occur in it.

- `foreach(:Generator, :Goal)`
  for each proof of \( \text{Generator} \) in turn, we make a copy of \( \text{Goal} \) with the appropriate substitution, then we execute these copies in sequence. For example, `foreach(between(1,3,I), p(I))` is equivalent to \( p(1), p(2), p(3) \).

Note that this is not the same as `forall/2`. For example, `forall(between(1,3,I), p(I))` is equivalent to \( \vdash \vdash p(1), \vdash \vdash p(2), \vdash \vdash p(3) \).

The trick in `foreach/2` is to ensure that the variables of \( \text{Goal} \) which do not occur in \( \text{Generator} \) are restored properly. (If there are no such variables, you might as well use `forall/2`.)
Like `forall/2`, this predicate does a failure-driven loop over the `Generator`. Unlike `forall/2`, the `Goals` are executed as an ordinary conjunction, and may succeed in more than one way.

`aggregate(+Template, +Discriminator, :Generator, -Result)` is a generalization of `setof/3` which lets you compute sums, minima, maxima, and so on.

`aggregate(+Template, :Generator, -Result)` is a generalization of `findall/3` which lets you compute sums, minima, maxima, and so on.

`aggregate_all(+Template, +Discriminator, :Generator, -Result)` is like `aggregate/4` except that it will find at most one solution, and does not bind free variables in the `Generator`.

`aggregate_all(+Template, :Generator, -Result)` is like `aggregate/3` except that it will find at most one solution, and does not bind free variables in the `Generator`.

`free_variables(:Goal, +Bound, +Vars0, -Vars)` binds `Vars` to the union of `Vars0` with the set of `free` variables in `Goal`, that is the set of variables which are captured neither by `Bound` nor by any internal quantifiers or templates in `Goal`. We have to watch out for `setof/3` and `bagof/3` themselves, for the explicit existential quantifier `Vars^Goal`, and for things like `\+(_)` which might look as though they bind variables but can’t.

`term_variables(+Term, +Vars0, -Vars)` binds `Vars` to a union of `Vars0` and the variables which occur in `Term`. This doesn’t take quantifiers into account at all.

New code should consider the built in `term_variables/2` which is likely to be faster, and works for cyclic terms.

Could be defined as:

```prolog
term_variables(Term, Vars0, Vars) :-
  nonvar(Term), !,
  ( foreacharg(Arg,Term),
    fromto(Vars0,S0,S,Vars)
  do  term_variables(Arg, S0, S)
  ).

term_variables(Term, Vars0, Vars) :-
  ( foreach(X,Vars0),
    param(Term)
  do  X\==Term
  ), !,
  Vars = [Term|Vars0].

term_variables(_, Vars, Vars).
```
10.2 Association Lists—library(assoc)

This library provides a binary tree implementation of "association lists". The binary tree is not kept balanced, as opposed to library(avl), which provides similar functionality based on balanced AVL trees.

Exported predicates:

empty_assoc(?Assoc)
  is true when Assoc is an empty assoc.

assoc_to_list(+Assoc, -List)
  assumes that Assoc is a proper "assoc" tree, and is true when List is a list of Key-Value pairs in ascending order with no duplicate Keys specifying the same finite function as Assoc. Use this to convert an assoc to a list.

gen_assoc(?Key, +Assoc, ?Value)
  assumes that Assoc is a proper "assoc" tree, and is true when Key is associated with Value in Assoc. Use this to enumerate Keys and Values in the Assoc, or to find Keys associated with a particular Value. If you want to look up a particular Key, you should use get_assoc/3. Note that this predicate is not determinate. If you want to maintain a finite bijection, it is better to maintain two assocs than to drive one both ways. The Keys and Values are enumerated in ascending order of Keys.

get_assoc(+Key, +Assoc, -Value)
  assumes that Assoc is a proper "assoc" tree. It is true when Key is identical to (==) one of the keys in Assoc, and Value unifies with the associated value. Note that since we use the term ordering to identify keys, we obtain logarithmic access, at the price that it is not enough for the Key to unify with a key in Assoc, it must be identical. This predicate is determinate. The argument order follows the pattern established by the built-in predicate arg/3 (called the arg/3, or selector, rule):

  predicate(indices, structure, element).

The analogy with arg(N, Term, Element) is that

  Key:N :: Assoc:Term :: Value:Element.

get_next_assoc(+Key, +Assoc, -Knext, -Vnext)
  is true when Knext is the smallest key in Assoc such that Knext@>Key, and Vnext is the value associated with Knext. If there is no such Knext, get_next_assoc/4 naturally fails. It assumes that Assoc is a proper assoc. Key should normally be ground. Note that there is no need for Key to be in the association at all. You can use this predicate in combination with min_assoc/3 to traverse an association tree; but if there are N pairs in the tree the cost will be O(N lg N). If you want to traverse all the pairs, calling assoc_to_list/2 and walking down the list will take O(N) time.

get_prev_assoc(+Key, +Assoc, -Kprev, -Vprev)
  is true when Kprev is the largest key in Assoc such that Kprev@<Key, and Vprev is the value associated with Kprev. You can use this predicate in com-
Combination with `max_assoc/3` to traverse an assoc. See the notes on `get_next_assoc/4`.

`is_assoc(+Thing)`

is true when `Thing` is a (proper) association tree. If you use the routines in this file, you have no way of constructing a tree with an unbound tip, and the heading of this file explicitly warns against using variables as keys, so such structures are NOT recognized as being association trees. Note that the code relies on variables (to be precise, the first anonymous variable in `is_assoc/1`) being `<@ than any non-variable.

`list_to_assoc(+List, -Assoc)`

is true when `List` is a proper list of `Key-Val` pairs (in any order) and `Assoc` is an association tree specifying the same finite function from `Keys` to `Values`. Note that the list should not contain any duplicate keys. In this release, `list_to_assoc/2` doesn’t check for duplicate keys, but the association tree which gets built won’t work.

`ord_list_to_assoc(+List, -Assoc)`

is a version of `list_to_assoc/2` which trusts you to have sorted the list already. If you pair up an ordered set with suitable values, calling this instead will save the sort.

`map_assoc(:Pred, +Assoc)`

is true when `Assoc` is a proper association tree, and for each `Key-Val` pair in `Assoc`, the proposition `Pred(Val)` is true. `Pred` must be a closure, and `Assoc` should be proper. There should be a version of this predicate which passes `Key` to `Pred` as well as `Val`, but there isn’t.

`map_assoc(:Pred, ?OldAssoc, ?NewAssoc)`

is true when `OldAssoc` and `NewAssoc` are association trees of the same shape (at least one of them should be provided as a proper assoc, or `map_assoc/3` may not terminate), and for each `Key`, if `Key` is associated with `Old` in `OldAssoc` and with `New` in `NewAssoc`, the proposition `Pred(Old, New)` is true. Normally we assume that `Pred` is a function from `Old` to `New`, but the code does not require that. There should be a version of this predicate which passes `Key` to `Pred` as well as `Old` and `New`, but there isn’t. If you’d have a use for it, please tell us.

`max_assoc(+Assoc, -Key, -Val)`

is true when `Key` is the largest `Key` in `Assoc`, and `Val` is the associated value. It assumes that `Assoc` is a proper assoc. This predicate is determinate. If `Assoc` is empty, it just fails quietly; an empty set can have no largest element!

`min_assoc(+Assoc, -Key, -Val)`

is true when `Key` is the smallest `Key` in `Assoc`, and `Val` is the associated value. It assumes that `Assoc` is a proper assoc. This predicate is determinate. If `Assoc` is empty, it just fails quietly; an empty set can have no smallest element!

`portray_assoc(+Assoc)`

writes an association tree to the current output stream in a pretty form so that you can easily see what it is. Note that an association tree written out this way
can NOT be read back in. For that, use writeq/1. The point of this predicate is to get association trees displayed nicely by print/1.

```prolog
put_assoc(+Key, +OldAssoc, +Val, -NewAssoc)
```

is true when OldAssoc and NewAssoc define the same finite function, except that NewAssoc associates Val with Key. OldAssoc need not have associated any value at all with Key.

### 10.3 Attributed Variables—library(atts)

This package implements attributed variables. It provides a means of associating with variables arbitrary attributes, i.e. named properties that can be used as storage locations as well as to extend the default unification algorithm when such variables are unified with other terms or with each other. This facility was primarily designed as a clean interface between Prolog and constraint solvers, but has a number of other uses as well. The basic idea is due to Christian Holzbaur and he was actively involved in the final design. For background material, see the dissertation [Holzbaur 90].

The package provides a means to declare and access named attributes of variables. The attributes are compound terms whose arguments are the actual attribute values. The attribute names are private to the module in which they are defined. They are defined with a declaration

```prolog
:- attribute AttributeSpec, ..., AttributeSpec.
```

where each AttributeSpec has the form (Name/Arity). There must be at most one such declaration in a module Module. At most 255 modules can declare attributes at the same time.

Having declared some attribute names, these attributes can now be added, updated and deleted from unbound variables. For each declared attribute name, any variable can have at most one such attribute (initially it has none).

The declaration causes the following two access predicates to become defined by means of the goal_expansion/5 mechanism. They take a variable and an AccessSpec as arguments where an AccessSpec is either +(Attribute), -(Attribute), or a list of such. The ‘+’ prefix may be dropped for convenience. Attribute must be nonvariable at compile time. The meaning of the ‘+’/‘-’ prefix is documented below:

```prolog
Module:get_atts(-Var, ?AccessSpec)
```

Gets the attributes of Var, which must be a variable, according to AccessSpec. If AccessSpec was unbound at compile time, it will be bound to a list of all present attributes of Var, otherwise the elements of AccessSpec have the following meaning:

```prolog
+(Attribute)
```

The corresponding actual attribute must be present and is unified with Attribute. The ‘+’ prefix may be dropped for convenience.
-(Attribute)
The corresponding actual attribute must be absent. The arguments of Attribute are ignored, only the name and arity are relevant.

Module: put_atts(-Var, +AccessSpec)
Sets the attributes of Var, which must be a variable, according to AccessSpec. The effects of put_atts/2 are undone on backtracking.

+(Attribute)
The corresponding actual attribute is set to Attribute. If the actual attribute was already present, it is simply replaced. The '+' prefix may be dropped for convenience.

-(Attribute)
The corresponding actual attribute is removed. If the actual attribute was already absent, nothing happens.

A module that contains an attribute declaration has an opportunity to extend the default unification algorithm by defining the following predicate:

Module: verify_attributes(-Var, +Value, -Goals) hook
This predicate is called whenever a variable Var that might have attributes in Module is about to be bound to Value (it might have none). The unification resumes after the call to verify_attributes/3. Value is a nonvariable, or another attributed variable. Var might have no attributes present in Module; the unification extension mechanism is not sophisticated enough to filter out exactly the variables that are relevant for Module.

verify_attributes/3 is called before Var has actually been bound to Value. If it fails, the unification is deemed to have failed. It may succeed nondeterminately, in which case the unification might backtrack to give another answer. It is expected to return, in Goals, a list of goals to be called after Var has been bound to Value. Finally, after calling Goals, goals blocked on Var may have become unblocked, in which case they are called.

verify_attributes/3 may invoke arbitrary Prolog goals, but Var should not be bound by it. Binding Var will result in undefined behavior.

If Value is a nonvariable, verify_attributes/3 will typically inspect the attributes of Var and check that they are compatible with Value and fail otherwise. If Value is another attributed variable, verify_attributes/3 will typically copy the attributes of Var over to Value, or merge them with Value’s, in preparation for Var to be bound to Value. In either case, verify_attributes/3 may determine Var’s current attributes by calling get_atts(Var, List) with an unbound List.

In the case when a single unification binds multiple attributed variables, first all such bindings are undone, then the following actions are carried out for each relevant variable:

1. For each relevant module M, M:verify_attributes/3 is called, collecting a list of returned Goals.
2. The variable binding is *redone*.
3. Any *Goals* are called.
4. Any goals blocked on the variable, that has now become unblocked, are called.

An important use for attributed variables is in implementing coroutining facilities as an alternative or complement to the built-in coroutining mechanisms. In this context it might be useful to be able to interpret some of the attributes of a variable as a goal that is blocked on that variable. Certain built-in predicates (*frozen/2, copy_term/3*) and the Prolog top level need to access blocked goals, and so need a means of getting the goal interpretation of attributed variables by calling:

```
Module:attribute_goal(-Var, -Goal)  
```

This predicate is called in each module that contains an attribute declaration, when an interpretation of the attributes as a goal is needed, in particular in *frozen/2, copy_term/3* and the Prolog top level. It should unify *Goal* with the interpretation, or merely fail if no such interpretation is available.

An important use for attributed variables is to provide an interface to constraint solvers. An important function for a constraint solver in the constraint logic programming paradigm is to be able to perform projection of the residual constraints onto the variables that occurred in the top-level query. A module that contains an attribute declaration has an opportunity to perform such projection of its residual constraints by defining the following predicate:

```
Module:project_attributes(+QueryVars, +AttrVars)  
```

This predicate is called by the Prolog top level in each module that contains an attribute declaration. *QueryVars* is the list of variables occurring in the query, or in terms bound to such variables, and *AttrVars* is a list of possibly attributed variables created during the execution of the query. The two lists of variables may or may not be disjoint.

If the attributes on *AttrVars* can be interpreted as constraints, this predicate will typically “project” those constraints onto the relevant *QueryVars*. Ideally, the residual constraints will be expressed entirely in terms of the *QueryVars*, treating all other variables as existentially quantified. Operationally, *project_attributes/2* must remove all attributes from *AttrVars*, and add transformed attributes representing the projected constraints to some of the *QueryVars*.

Projection has the following effect on the Prolog top level. When the top-level query has succeeded, *project_attributes/2* is called first. The top level then prints the answer substitution and residual constraints. While doing so, it searches for attributed variables created during the execution of the query. For each such variable, it calls *attribute_goal/2* to get a printable representation of the constraint encoded by the attribute. Thus, *project_attributes/2* is a mechanism for controlling how the residual constraints should be displayed at top level.

The exact definition of *project_attributes/2* is constraint system dependent, but see Section 10.10.7 [Answer Constraints], page 477, and see Section 10.11.5
[CLPQR Projection], page 513, for details about projection in CLPFD and CLP(Q,R) respectively.

In the following example we sketch the implementation of a finite domain “solver”. Note that an industrial strength solver would have to provide a wider range of functionality and that it quite likely would utilize a more efficient representation for the domains proper. The module exports a single predicate `domain(-Var, ?Domain)`, which associates `Domain` (a list of terms) with `Var`. A variable can be queried for its domain by leaving `Domain` unbound.

We do not present here a definition for `project_attributes/2`. Projecting finite domain constraints happens to be difficult.
:- module(domain, [domain/2]).
:- use_module(library(atts)).
:- use_module(library(ordsets), [ord_intersection/3, ord_intersect/2, list_to_ord_set/2]).

:- attribute dom/1.

verify_attributes(Var, Other, Goals) :-
    get_atts(Var, dom(Da)), !, % are we involved?
    ( var(Other) -> % must be attributed then
        ( get_atts(Other, dom(Db)) -> % has a domain?
            ord_intersection(Da, Db, Dc),
            Dc = [El|Els], % at least one element
            ( Els = [] -> % exactly one element
                Goals = [Other=El] % implied binding
            ; Goals = [],
                put_atts(Other, dom(Dc)) % rescue intersection
            )
        ; Goals = [],
            put_atts(Other, dom(Da)) % rescue the domain
        )
    ; Goals = [],
        ord_intersect([Other], Da) % value in domain?
    ).

verify_attributes(_, _, []). % unification triggered

attribute_goal(Var, domain(Var,Dom)) :- % interpretation as goal
    get_atts(Var, dom(Dom)).

domain(X, Dom) :-
    var(Dom), !,
    get_atts(X, dom(Dom)).

domain(X, List) :-
    list_to_ord_set(List, Set),
    Set = [El|Els], % at least one element
    ( Els = [] -> % exactly one element
        X = El % implied binding
    ; put_atts(Fresh, dom(Set)),
        X = Fresh % may call
    ).
Note that the “implied binding” Other=El was deferred until after the completion of verify_attribute/3. Otherwise, there might be a danger of recursively invoke verify_attribute/3, which might bind Var, which is not allowed inside the scope of verify_attribute/3. Deferring unifications into the third argument of verify_attribute/3 effectively serializes the calls to verify_attribute/3.

Assuming that the code resides in the file domain.pl, we can load it via:

```prolog
| ?- use_module(domain).
```

Let’s test it:

```prolog
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]).

domain(X,[1,5,6,7]),
domain(Y,[3,4,5,6]),
domain(Z,[1,6,7,8])
```

```prolog
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
X=Y.

Y = X,
domain(X,[5,6]),
domain(Z,[1,6,7,8])
```

```prolog
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
X=Y, Y=Z.

X = 6,
Y = 6,
Z = 6
```

To demonstrate the use of the Goals argument of verify_attributes/3, we give an implementation of freeze/2. We have to name it myfreeze/2 in order to avoid a name clash with the built-in predicate of the same name.
% myfreeze.pl

:- module(myfreeze, [myfreeze/2]).

:- use_module(library(atts)).

:- meta_predicate myfreeze(*, 0).

:- attribute frozen/1.

verify_attributes(Var, Other, Goals) :-
   get_atts(Var, frozen(Fa)), !, % are we involved?
   ( var(Other) -> % must be attributed then
     ( get_atts(Other, frozen(Fb)) % has a pending goal?
     ->
       put_atts(Other, frozen((Fa,Fb))) % rescue conjunction
     ;
       put_atts(Other, frozen(Fa)) % rescue the pending goal
     ),
     Goals = []
   ;
   Goals = [Fa] % wake our frozen goal
   ).

verify_attributes(_, _, []).

attribute_goal(Var, myfreeze(Var,Goal)) :- % interpretation as goal
   get_atts(Var, frozen(Goal)).

myfreeze(X, Goal) :-
   put_atts(Fresh, frozen(Goal)),
   Fresh = X.

Assuming that this code lives in file myfreeze.pl, we would use it via:

| ?- use_module(myfreeze). |
| ?- myfreeze(X,print(bound(x,X))), X=2. |

bound(x,2) % side-effect
X = 2 % bindings

The two solvers even work together:

| ?- myfreeze(X,print(bound(x,X))), domain(X,[1,2,3]),
  domain(Y,[2,10]), X=Y. |

bound(x,2) % side-effect
X = 2, % bindings
Y = 2

The two example solvers interact via bindings to shared attributed variables only. More complicated interactions are likely to be found in more sophisticated solvers. The cor-
responding \texttt{verify_attributes/3} predicates would typically refer to the attributes from other known solvers/modules via the module prefix in \texttt{Module:get_atts/2}.

\section*{10.4 AVL Trees—library(avl)}

This library module provides an AVL tree implementation of "association lists". The binary tree is kept balanced, as opposed to \texttt{library(assoc)}, which provides similar functionality based on binary trees that are not kept balanced.

Exported predicates:

\begin{verbatim}
empty_avl(?AVL) is true when AVL is an empty AVL tree.
avl_to_list(+AVL, -List) assumes that AVL is a proper AVL tree, and is true when List is a list of Key-Value pairs in ascending order with no duplicate keys specifying the same finite function as AVL. Use this to convert an AVL to an ordered list.
is_avl(+AVL) is true when AVL is a (proper) AVL tree. It checks both the order condition (that the keys are in ascending order as you go from left to right) and the height balance condition. This code relies on variables (to be precise, the first anonymous variable in \texttt{is_avl/1}) being \texttt{@<} than any non-variable. in strict point of fact you can construct an AVL tree with variables as keys, but \texttt{is_avl/1} doesn't believe it, and it is not good taste to do so.
avl_domain(+AVL, -Domain) unifies Domain with the ordered set representation of the domain of the AVL tree (the keys of it). As the keys are in ascending order with no duplicates, we just read them off like \texttt{avl_to_list/2}.
avl_range(+AVL, -Range) unifies Range with the ordered set representation of the range of the AVL tree (the values associated with its keys, not the keys themselves). Note that the cardinality (length) of the domain and the range are seldom equal, except of course for trees representing invertible maps.
avl_min(+AVL, -Key) is true when Key is the smallest key in AVL.
avl_min(+AVL, -Key, -Val) is true when Key is the smallest key in AVL and Val is its value.
avl_max(+AVL, -Key) is true when Key is the greatest key in AVL.
avl_max(+AVL, -Key, -Val) is true when Key is the greatest key in AVL and Val is its value.
avl_height(+AVL, -Height) is true when Height is the height of the given AVL tree, that is, the longest path in the tree has Height 'node's on it.
\end{verbatim}
avl_size(+AVL, -Size)
   is true when Size is the size of the AVL tree, the number of 'node's in it.

portray_avl(+AVL)
   writes an AVL tree to the current output stream in a pretty form so that you
can easily see what it is. Note that an AVL tree written out this way can NOT
be read back in; for that use writeq/1. The point of this predicate is to get
AVL trees displayed nicely by print/1.

avl_member(?Key, +AVL)
   is true when Key is one of the keys in the given AVL. This predicate should be
used to enumerate the keys, not to look for a particular key (use avl_fetch/2
or avl_fetch/3 for that). The Keys are enumerated in ascending order.

avl_member(?Key, +AVL, ?Val)
   is true when Key is one of the keys in the given AVL and Val is the value the
AVL associates with that Key. This predicate should be used to enumerate
the keys and their values, not to look up the value of a known key (use avl_fetch/3)
for that. The Keys are enumerated in ascending order.

avl_fetch(+Key, +AVL)
   is true when the (given) Key is one of the keys in the (given) AVL. Use this
to test whether a known Key occurs in AVL and you don't want to know the
value associated with it.

avl_fetch(+Key, +AVL, -Val)
   is true when the (given) Key is one of the keys in the (given) AVL and the value
associated with it therein is Val. It should be used to look up known keys, not
to enumerate keys (use either avl_member/2 or avl_member/3 for that).

avl_next(+Key, +AVL, -Knext)
   is true when Knext is the next key after Key in AVL; that is, Knext is the
smallest key in AVL such that Knext @> Key.

avl_next(+Key, +AVL, -Knext, -Vnext)
   is true when Knext is the next key after Key in AVL and Vnext is the value
associated with Knext in AVL. That is, Knext is the smallest key in AVL such
that Knext @> Key, and avl_fetch(Knext, AVL, Vnext).

avl_prev(+Key, +AVL, -Kprev)
   is true when Kprev is the key previous to Key in AVL; that is, Kprev is the
greatest key in AVL such that Kprev @< Key.

avl_prev(+Key, +AVL, -Kprev, -Vprev)
   is true when Kprev is the key previous to Key in AVL and Vprev is the value
associated with Kprev in AVL. That is, Kprev is the greatest key in AVL such
that Kprev @< Key, and avl_fetch(Kprev, AVL, Vprev).

avl_change(+Key, ?AVL1, ?Val1, ?AVL2, ?Val2)
   is true when AVL1 and AVL2 are avl trees of exactly the same shape, Key
is a key of both of them, Val1 is the value associated with Key in AVL1 and
Val2 is the value associated with it in AVL2, and when AVL1 and AVL2 are
identical except perhaps for the value they assign to Key. Use this to change
the value associated with a Key which is already present, not to insert a new
Key (it won’t).

ord_list_to_avl(+List, -AVL)
is given a list of Key-Val pairs where the Keys are already in standard order
with no duplicates (this is not checked) and returns an AVL representing the
same associations. This takes \(O(N)\) time, unlike list_to_avl/2 which takes
\(O(N \lg N)\).

list_to_avl(+Pairs, -AVL)
is given a proper list of Key-Val pairs where the Keys are in no particular
order (but are sufficiently instantiated to be told apart) and returns an AVL
representing the same associations. This works by starting with an empty tree
and inserting the elements of the list into it. This takes \(O(N \lg N)\) time. Since
it is possible to read off a sorted list in \(O(N)\) time from the result, \(O(N \lg N)\)
is as good as can possibly be done. If the same Key appears more than once in
the input, the last value associated with it will be used. Could be defined as:

list_to_avl(Pairs, AVL) :-
( foreach(K-V,Pairs),
  fromto(empty,AVL0,AVL1,AVL)
do  avl_store(K, AVL0, V, AVL1) )
).

avl_store(+Key, +OldAVL, +Val, ?NewAVL)
is true when OldAVL and NewAVL define the same finite function except that
NewAVL associates Val with Key. OldAVL need not have associated any value
at all with Key. When it didn’t, you can read this as "insert (Key->Val) into
OldAVL giving NewAVL".

avl_incr(+Key, +OldAVL, +Incr, +NewAVL)
if Key is not present in OldAVL, adds Key->Incr. if Key->N is present in
OldAVl, changes it to Key->N+Incr.

avl_delete(+Key, +OldAVL, -Val, -NewAVL)
is true when OldAVL and NewAVL define the same finite function except that
OldAVL associates Key with Val and NewAVL doesn’t associate Key with any
value.

avl_del_min(+OldAVL, -Key, -Val, -NewAVL)
is true when OldAVL and NewAVL define the same finite function except that
OldAVL associates Key with Val and NewAVL doesn’t associate Key with any
value and Key precedes all other keys in OldAVL.

avl_del_max(+OldAVL, -Key, -Val, -NewAVL)
is true when OldAVL and NewAVL define the same finite function except that
OldAVL associates Key with Val and NewAVL doesn’t associate Key with any
value and Key is preceded by all other keys in OldAVL.

avl_map(:Pred, +AVL)
is true when AVL is an association tree, and for each Key, if Key is associated
with Value in AVL, Pred(Value) is true.
avl_map(_Pred, +OldAVL, -NewAVL)
is true when OldAVL and NewAVL are association trees of the same shape, and for each Key, if Key is associated with Old in OldAVL and with New in NewAVL, Pred(Old, New) is true.

10.5 Bags, or Multisets—library(bags)

This library module provides operations on bags. Bags are also known as multisets. A bag B is a function from a set dom(B) to the non-negative integers. For the purposes of this module, a bag is constructed from two functions:

\[
\text{bag} \quad \text{creates an empty bag}
\]

\[
\text{bag}(E, M, B) \quad \text{extends the bag } B \text{ with a new element } E \text{ which occurs with multiplicity } M, \text{ and which precedes all elements of } B \text{ in Prolog’s order.}
\]

A bag is represented by a Prolog term mirroring its construction. There is one snag with this: what are we to make of

\[
\text{bag}(f(a,Y), 1, \text{bag}(f(X,b), 1, \text{bag})).
\]

As a term it has two distinct elements, but \( f(a, b) \) will be reported as occurring in it twice. But according to the definition above,

\[
\text{bag}(f(a,b), 1, \text{bag}(f(a,b), 1, \text{bag}))
\]

is not the representation of any bag, that bag is represented by

\[
\text{bag}(f(a,b), 2, \text{bag})
\]

alone. We are apparently stuck with a scheme which is only guaranteed to work for "sufficiently instantiated" terms, but then, that’s true of a lot of Prolog code.

The reason for insisting on the order is to make union and intersection linear in the sizes of their arguments. library(ordsets) does the same for ordinary sets.

Exported predicates:

\[
\text{is_bag(+Bag)}
\]
recognizes proper well-formed bags. You can pass variables to is_bag/1, and it will reject them.

\[
\text{portray_bag(+Bag)}
\]
writes a bag to the current output stream in a pretty form so that you can easily see what it is. Note that a bag written out this way can not be read back in. For that, use write_canonical/1. The point of this predicate is to have bags displayed nicely by print/1 and the debugger. This will print things which are not fully instantiated, which is mainly of interest for debugging this module.
checkbag(:Pred, +Bag)

is true when Bag is a Bag\{E1:M1, ..., En:Mn\} with elements Ei of multiplicity Mi, and Pred(Ei, Mi) is true for each i.

mapbag(:Pred, +Bag)

is true when Bag is a Bag\{E1:M1, ..., En:Mn\} with elements Ei of multiplicity Mi, and Pred(Ei) is true for each element Ei. The multiplicities are ignored: if you don’t want this, use checkbag/2.

mapbag(:Pred, +OldBag, -NewBag)

is true when OldBag is a Bag\{E1:M1, ..., En:Mn\} and NewBag is a Bag\{F1:M'1, ..., Fn:M'n\} and the elements of OldBag and NewBag are related by Pred(Ei, Fj). What happens is that the elements of OldBag are mapped, and then the result is converted to a bag, so there is no positional correspondence between Ei and Fj. Even when Pred is bidirectional, mapbag/3 is not. OldBag should satisfy is_bag/1 before mapbag/3 is called.

somebag(:Pred, +Bag)

is true when Bag is a Bag\{E1:M1, ..., En:Mn\} with elements Ei of multiplicity Mi and Pred(Ei, Mi) is true of some element Ei and its multiplicity. There is no version which ignores the Mi.

somechkbag(:Pred, +Bag)

is like somebag(Pred, Bag), but commits to the first solution it finds. For example, if p(X,X,-), somechk(p(X), Bag) would be an analogue of memberchk/2 for bags.

bag_to_list(+Bag, -List)

converts a Bag\{E1:M1, ..., En:Mn\} to a list where each element appears as many times as its multiplicity requires. For example, Bag\{a:1, b:3, c:2\} would be converted to [a,b,b,b,c,c].

bag_to_ord_set(+Bag, -Ordset)

converts a Bag\{E1:M1, ..., En:Mn\} to a list where each element appears once without its multiplicity. The result is always an ordered (representation of a) set, suitable for processing by library(ordsets). See also bag_to_list/2.

bag_to_ord_set(+Bag, +Threshold, -Ordset)

given a Bag\{E1:M1, ..., En:Mn\} returns a list in standard order of the set of elements \{Ei | Mi > Threshold\}. The result is an Ordset.

list_to_bag(+List, -Bag)

converts a proper list List to a Bag representing the same multi-set. Each element of the List appears once in the Bag together with the number of times it appears in the List.

bag_to_set(+Bag, -Set)

converts a Bag\{E1:M1, ..., En:Mn\} to a list which represents the Set \{E1, ..., En\}. The order of elements in the result is not defined: for a version where the order is defined use bag_to_ord_set/2.
bag_to_set(+Bag, +Threshold, -Set)
given a Bag\{E1:M1, ..., En:Mn\} returns a list which represents the Set of elements \{Ei \mid Mi >= Threshold\}. Because the Bag is sorted, the result is necessarily an ordered set.

empty_bag(?Bag)
is true when Bag is the representation of an empty bag. It can be used both to make and to recognize empty bags.

member(?Element, ?Multiplicity, +Bag)
is true when Element appears in the multi-set represented by Bag with the indicated Multiplicity. Bag should be instantiated, but Element and Multiplicity may severally be given or solved for.

memberchk(+Element, ?Multiplicity, +Bag)
is true when Element appears in the multi-set represented by Bag, with the indicated Multiplicity. It should only be used to check whether a given element occurs in the Bag, or whether there is an element with the given Multiplicity. Note that guessing the multiplicity and getting it wrong may force the wrong choice of clause, but the result will be correct if is_bag(Bag).

bag_max(+Bag, -CommonestElement)
unifies CommonestElement with the element of Bag which occurs most often, picking the leftmost element if several have this multiplicity. To find the multiplicity as well, use bag_max/3. bag_max/2 and bag_min/2 break ties the same way.

bag_min(+Bag, -RarestElement)
unifies RarestElement with the element of Bag which occurs least often, picking the leftmost element if several have this multiplicity. To find the multiplicity as well, use bag_min/3. bag_max/2 and bag_min/2 break ties the same way, so

\[
\text{bag_max(Bag, Elt), bag_min(Bag, Elt)}
\]
is true only when all the elements of Bag have the same multiplicity.

bag_max(+Bag, -CommonestElement, -Multiplicity)
unifies CommonestElement with the element of Bag which occurs most often, and Multiplicity with the multiplicity of that element. If there are several elements with the same greatest multiplicity, the left-most is returned. bag_max/3 breaks ties the same way.

bag_min(+Bag, -RarestElement)
unifies RarestElement with the element of Bag which occurs least often, and Multiplicity with the multiplicity of that element. If there are several elements with the same least multiplicity, the left-most is returned. bag_max/3 breaks ties the same way, so

\[
\text{bag_max(Bag, Elt, Mult), bag_min(Bag, Elt, Mult)}
\]
is true only when all the elements of Bag have multiplicity Mult.
length(+Bag, -BagCardinality, -SetCardinality)
unifies BagCardinality with the total cardinality of the multi-set Bag (the sum
of the multiplicities of its elements) and SetCardinality with the number of
distinct elements.

make_sub_bag(+Bag, -SubBag)
enumerates the sub-bags of Bag, unifying SubBag with each of them in turn.
The order in which the sub-bags are generated is such that if SB2 is a sub-bag
of SB1 which is a sub-bag of Bag, SB1 is generated before SB2. In particular,
Bag is enumerated first and bag last.

test_sub_bag(+Bag, +SubBag)
is true when SubBag is (already) a sub-bag of Bag. That is, each element of
SubBag must occur in Bag with at least the same multiplicity. If you know
SubBag, you should use this to test, not make_sub_bag/2.

bag_union(+Bag1, +Bag2, -Union)
unifies Union with the multi-set union of bags Bag1 and Bag2.

bag_union(+ListOfBags, -Union)
is true when ListOfBags is given as a proper list of bags and Union is their
multi-set union. Letting K be the length of ListOfBags, and N the sum of the
sizes of its elements, the cost is $O(N \log K)$.

bag_intersection(+Bag1, +Bag2, -Intersection)
unifies Intersection with the multi-set intersection of bags Bag1 and Bag2.

bag_intersection(+ListOfBags, -Intersection)
is true when ListOfBags is given as a non-empty proper list of Bags and Intersection is their intersection. The intersection of an empty list of Bags would be the universe with infinite multiplicities!

bag_intersect(+Bag1, +Bag2)
is true when the multi-sets Bag1 and Bag2 have at least one element in common.

bag_add_element(+Bag1, +Element, +Multiplicity, -Bag2)
computes Bag2 = Bag1 U [Element:Multiplicity]. Multiplicity must be an
integer.

bag_del_element(+Bag1, +Element, +Multiplicity, -Bag2)
computes Bag2 = Bag1 \ [Element:Multiplicity]. Multiplicity must be an
integer.

bag_subtract(+Bag1, +Bag2, -Difference)
is true when Difference is the multiset difference of Bag1 and Bag2.

10.6 External Storage of Terms (Berkeley DB)—
library(bdb)

Please note: library(bdb) has been deprecated. See library(lmdb) for a replacement with similar feature set.
This library module handles storage and retrieval of terms on files. By using indexing, the store/retrieve operations are efficient also for large data sets. The package is an interface to the Berkeley DB toolset.

10.6.1 Basics
The idea is to get a behavior similar to \texttt{assert/1}, \texttt{retract/1} and \texttt{clause/2}, but the terms are stored on files instead of in primary memory.

The differences compared with the Prolog database are:

- A \texttt{database} must be opened before any access and closed after the last access. (There are special predicates for this: \texttt{db_open/[4,5]} and \texttt{db_close/1}.)
- The functors and the indexing specifications of the terms to be stored have to be given when the database is created. (see Section 10.6.7 [The DB-Spec], page 411).
- The indexing is specified when the database is created. It is possible to index on other parts of the term than just the functor and first argument.
- Changes affect the database immediately.
- The database will store variables with attributes or with blocked goals as ordinary variables.

Some commercial databases can’t store non-ground terms or more than one instance of a term. This library module can however store terms of either kind.

10.6.2 Current Limitations

- The terms are not necessarily fetched in the same order as they were stored.
- If the process dies during an update operation (\texttt{db_store/3}, \texttt{db_erase/[2,3]}), the database can be inconsistent.
- Databases can only be shared between processes running on the machine where the environment is created (see Section 10.6.5 [Predicates], page 406). The database itself can be on a different machine.
- The number of terms ever inserted in a database cannot exceed $2^{32}-1$.
- Duplicate keys are not handled efficiently by Berkeley DB. This limitation is supposed to get lifted in the future. Duplicate keys can result from indexing on non-key attribute sets, inserting terms with variables on indexing positions, or simply from storing the same term more than once.

10.6.3 Berkeley DB
This library module is an interface to the Berkeley DB toolset to support persistent storage of Prolog terms. Some of the notions of Berkeley DB are directly inherited, e.g. the environment.

The interface uses the Concurrent Access Methods product of Berkeley DB. This means that multiple processes can open the same database, but transactions and disaster recovery are not supported.
The environment and the database files are ordinary Berkeley DB entities which means that
the standard support utilities (e.g. db_stat) will work.

10.6.4 The DB-Spec—Informal Description

The db-spec defines which functors are allowed and which parts of a term are used for
indexing in a database. The syntax of a db-spec is a skeletal goal with no module. The
db-spec is a list of atoms and compound terms where the arguments are either + or −. A
term can be inserted in the database if there is a spec in the spec list with the same functor.

Multilevel indexing is not supported, terms have to be “flattened”.

Every spec with the functor of the indexed term specifies an indexing. Every argument
where there is a + in the spec is indexed on.

The idea of the db-spec is illustrated with a few examples. (A section further down explains
the db-spec in a more formal way).

Given a spec of \([f(+,-), (+,-), g, f(-,+)]\) the indexing works as follows. (The parts
with indexing are underlined.)

<table>
<thead>
<tr>
<th>Term</th>
<th>Store</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>g(x,y)</td>
<td>domain error</td>
<td>domain error</td>
</tr>
<tr>
<td>f(A,B)</td>
<td>f(A,B)</td>
<td>instantiation error</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>f(a,b)</td>
<td>f(a,b)</td>
<td>f(a,b)</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>[a,b]</td>
<td>.(a,.(b,[]))</td>
<td>.(a,.(b,[]))</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>

The specification \([f(+,-), f(-,+)]\) is different from \([f(+,+)]\). The first specifies that two
indices are to be made whereas the second specifies that only one index is to be made on
both arguments of the term.

10.6.5 Predicates
10.6.5.1 Conventions

The following conventions are used in the predicate descriptions below.

- **Mode** is either update or read or enumerate. In mode read no updates can be made. Mode enumerate is like mode read, but indexing cannot be used, i.e. you can only sequentially enumerate the items in the database. In mode enumerate only the file storing the terms along with their references is used.

- **EnvRef** is a reference to an open database environment. The environment is returned when it is opened. The reference becomes invalid after the environment has been closed.

- **DBRef** is a reference to an open database. The reference is returned when the database is opened. The reference becomes invalid after the database has been closed.
• **TermRef** is a reference to a term in a given database. The reference is returned when a term is stored. The reference stays valid even after the database has been closed and hence can be stored permanently as part of another term. However, if such references are stored in the database, automatic compression of the database (using `db_compress/[2,3]`) is not possible, in that case the user has to write her own compressing predicate.

• **SpecList** is a description of the indexing scheme; see Section 10.6.7 [The DB-Spec], page 411.

• **Term** is any Prolog term.

• **Iterator** is a non-backtrackable mutable object. It can be used to iterate through a set of terms stored in a database. The iterators are unidirectional.

### 10.6.5.2 The Environment

To enable sharing of databases between process, programs have to create environments and the databases should be opened in these environments. A database can be shared between processes that open it in the same environment. An environment physically consists of a directory containing the files needed to enable sharing databases between processes. The directory of the environment has to be located in a local file system.

Databases can be opened outside any environment (see `db_open/4`), but in that case a process writing the database must ensure exclusive access or the behavior of the predicates is undefined.

### 10.6.5.3 Memory Leaks

In order to avoid memory leaks, environments, databases and iterators should always be closed explicitly. Consider using `call_cleanup/2` to automate the closing/deallocation of these objects. You can always use `db_current_env/1`, `db_current/5` and `db_current_iterator/3` to enumerate the currently living objects.

**Please note:** a database must not be closed while there are outstanding choices for some `db_fetch/3` goal that refers to that database. Outstanding choices can be removed with a cut (!).

### 10.6.5.4 The Predicates

```prolog
db_open_env(+EnvName, -EnvRef)
db_open_env(+EnvName, +CacheSize, -EnvRef)
```

Opens an environment with the name `EnvName`. A directory with this name is created for the environment if necessary.

By using `db_open_env/3` one can specify the size of the cache: `CacheSize` is the (integer) size of the cache in kilobytes. The size of the cache cannot be less than 20 kilobytes. `db_open_env/2` will create a cache of the system’s default size.

The size of the cache is determined when the environment is created and cannot be changed by future openings.

A process cannot open the same environment more than once.
db_close_env(+EnvRef)
   Closes an environment. All databases opened in the environment will be closed as well.

db_current_env(?EnvName, ?EnvRef)
   Unifies the arguments with the open environments. This predicate can be used for enumerating all currently open environments through backtracking.

db_open(+DBName, +Mode, ?SpecList, -DBRef)
   db_open(+DBName, +Mode, ?SpecList, +Options, -DBRef)
   Opens a database with the name DBName. The database physically consists of a directory with the same name, containing the files that make up the database. If the directory does not exist, it is created. In that case Mode must be update and the db-spec SpecList must be ground. If an existing database is opened and Mode is read or update, SpecList is unified with the db-spec given when the database was created. If the unification fails an error is raised. DBRef is unified with a reference to the opened database.
   If Mode is enumerate then the indexing specification is not read, and SpecList is left unbound.
   Options provides a way to specify an environment in which to open the database, or a cache size. Options should be a list of terms of the following form:
      environment(EnvRef)
         The database will be opened in this environment.
      cache_size(CacheSize)
         This is the (integer) size of the cache in kilobytes. The size of the cache cannot be less than 20 kilobytes. If CacheSize is given as the atom default, a default cache size will be used. If CacheSize is given as the atom off or the atom none, all modified records will be flushed to disk after each operation.
   To avoid inconsistency, if multiple processes open the same database, then all of them should do that with Mode set to read or enumerate. (This is not enforced by the system.)

db_close(+DBRef)
   Closes the database referenced by DBRef. Any iterators opened in the database will be deallocated.

db_current(?DBName, ?Mode, ?SpecList, ?EnvRef, ?DBRef)
   Unifies the arguments with the open databases. This predicate can be used to enumerate all currently open databases through backtracking. If the database was opened without an environment, then EnvRef will be unified with the atom none.

db_store(+DBRef, +Term, -TermRef)
   Stores Term in the database DBRef. TermRef is unified with a corresponding term reference. The functor of Term must match the functor of a spec in the db-spec associated with DBRef.
db_fetch(+DBRef, ?Term, ?TermRef)
  Unifies Term with a term from the database DBRef. At the same time, TermRef is unified with a corresponding term reference. Backtracking over the predicate unifies with all terms matching Term.
  If TermRef is not instantiated then both the functor and the instantiatedness of Term must match a spec in the db-spec associated with DBRef.
  If TermRef is instantiated, the referenced term is read and unified with Term.
  If you simply want to find all matching terms, it is more efficient to use db_findall/5 or dbEnumerate/3.

db_erase(+DBRef, +TermRef)
db_erase(+DBRef, +TermRef, +Term)
  Deletes the term from the database DBRef that is referenced by TermRef.
  In the case of db_erase/2 the term associated with TermRef has to be looked up. db_erase/3 assumes that the term Term is identical with the term associated with TermRef (modulo variable renaming). If this is not the case, the behavior is undefined.

db_enumerate(+DBRef, ?Term, ?TermRef)
  Unifies Term with a term from the database DBRef. At the same time, TermRef is unified with a corresponding term reference. Backtracking over the predicate unifies with all terms matching Term.
  Implemented by linear search—the db-spec associated with DBRef is ignored. It is not useful to call this predicate with TermRef instantiated.

db_findall(+DBRef, +Template, +Term, :Goal, -Bag)
  Unifies Bag with the list of instances of Template in all proofs of Goal found when Term is unified with a matching term from the database DBRef. Both the functor and the instantiatedness of Term must match a spec in the db-spec associated with DBRef. Conceptually, this predicate is equivalent to findall(Template, (db_fetch(DBRef, Term, _), Goal), Bag).

db_compress(+DBRef, +DBName)
db_compress(+DBRef, +DBName, +SpecList)
  Copies the database given by DBRef to a new database named by DBName. The new database will be a compressed version of the first one in the sense that it will not have “holes” resulting from deletion of terms. Deleted term references will also be reused, which implies that references that refer to terms in the old database will be invalid in the new one.
  db_compress/2 looks for a database with the db-spec of the original one. db_compress/3 stores the terms found in the original database with the indexing specification SpecList. db_compress/2 cannot be used if the database DBRef was opened in mode enumerate.
  If the database DBName already exists then the terms of DBRef will be appended to it. Of course DBName must have an indexing specification, which enables the terms in DBRef to be inserted into it.
  In the case of db_compress/3 if the database DBName does not exist, then SpecList must be a valid indexing specification.
db_sync(+DBRef)
    Flushes any cached information from the database referenced by DBRef to stable storage.

db_make_iterator(+DBRef, -Iterator)

db_make_iterator(+DBRef, +Term, -Iterator)
    Creates a new iterator and unifies it with Iterator. Iterators created with db_make_iterator/2 iterate through the whole database. Iterators created with db_make_iterator/3 iterate through the terms that would be found by db_fetch(DBRef, Term, _).
    Every iterator created by db_make_iterator/[2,3] must be destroyed with db_iterator_done/1.

db_iterator_next(+Iterator, -Term, -TermRef)
    Iterator advances to the next term, Term and TermRef is unified with the term and its reference pointed to by Iterator. If there is no next term, the predicate fails.

db_iterator_done(+Iterator)
    Deallocates Iterator, which must not be in use anymore.

db_current_iterator(?DBRef, ?Term, ?Iterator)
    Unifies the variables with the respective properties of the living iterators. This predicate can be used to enumerate all currently alive iterators through backtracking. If Iterator was made with db_make_iterator/2 then Term will be left unbound.

db_export(+DBName, +ExportFile)

db_export(+DBName, +Options, +ExportFile)
    Exports the database with the name DBName to the text file ExportFile. ExportFile can be imported by db_import/[2,3].
    Options should be an options list of the form acceptable by db_open/[4,5].

db_import(+DBName, +ImportFile)

db_import(+DBName, +Options, +ImportFile)
    Imports the text file ImportFile into the database with the name DBName.
    If ImportFile is imported into an existing database, the SpecList found in the ImportFile will be unified with the SpecList in the database.
    Options should be an options list of the form acceptable by db_open/[4,5].
10.6.6 An Example Session

| ?- db_open(tempdb, update, [a(+,-)], DBRef), assert(tempdb(DBRef)).
| DBRef = '$db'(1077241400)

| ?- tempdb(DBRef), db_store(DBRef, a(b,1), _).
| DBRef = '$db'(1077241400)

| ?- tempdb(DBRef), db_store(DBRef, a(c,2), _).
| DBRef = '$db'(1077241400)

| ?- tempdb(DBRef), db_fetch(DBRef, a(b,X), _).
| X = 1,
| DBRef = '$db'(1077241400) ? ;
| no

| ?- tempdb(DBRef), dbEnumerate(DBRef, X, _).
| X = a(b,1),
| DBRef = '$db'(1077241400) ? ;
| X = a(c,2),
| DBRef = '$db'(1077241400) ? ;
| no

| ?- dbCurrent(DBName, Mode, Spec, EnvRef, DBRef).
| Mode = update,
| Spec = [a(+,-)],
| DBRef = '$db'(1077241400),
| DBName = tempdb,
| EnvRef = none ? ;
| no

| ?- tempdb(DBRef), db_close(DBRef).
| DBRef = '$db'(1077241400)

10.6.7 The DB-Spec

A db-spec has the form of a speclist:

speclist = [spec1, ..., specM]
spec      = functor(argspec1, ..., argspecN)
argspec   = + | -

where functor is a Prolog atom. The case $N = 0$ is allowed.

A spec $F(argspec1, ..., argspecN)$ is applicable to any nonvar term with principal functor $F/N$. 


When storing a term $T$ we generate a hash code for every applicable spec in the db-spec, and a reference to $T$ is stored with each of them. (More precisely with each element of the set of generated hash codes). If $T$ contains nonvar elements on each $+$ position in the spec, then the hash code depends on each of these elements. If $T$ does contain some variables on $+$ position, then the hash code depends only on the functor of $T$.

When fetching a term $Q$ we look for an applicable spec for which there are no variables in $Q$ on positions marked $+$. If no applicable spec can be found a domain error is raised. If no spec can be found where on each $+$ position a nonvar term occurs in $Q$ an instantiation error is raised. Otherwise, we choose the spec with the most $+$ positions in it breaking ties by choosing the leftmost one.

The terms that contain nonvar terms on every $+$ position will be looked up using indexing based on the principal functor of the term and the principal functor of terms on $+$ positions. The other (more general) terms will be looked up using an indexing based on the principal functor of the term only.

As can be seen, storing and fetching terms with variables on $+$ positions are not vigorously supported operations.

**10.6.8 Exporting and importing a database**

Since the database format of a Berkeley DB may change from version to version it may become necessary to migrate a database when upgrading. To this purpose there are two predicates available: `db_export/[2,3]` and `db_import/[2,3]` (see Section 10.6.5.4 [The Predicates], page 407).

The export/import feature was introduced in SICStus 3.12.0, but in that version you have to use `bdb:export/[2,3]` and `bdb:import/[2,3]`. Neither is exported from the bdb module, but can be used with module prefixing.

Since the bdb interface prior to SICStus 4 uses a custom hash function, the standard Berkeley DB migration tools will not work when migrating a database from SICStus 3 to SICStus 4.

**10.7 Generating Integers—library(between)**

This library module provides some means of generating integers. Exported predicates:

`between(+Lower, +Upper, -Number)`

is true when $Lower$, $Upper$, and $Number$ are integers, and $Lower < Number <= Upper$. If $Lower$ and $Upper$ are given, $Number$ can be tested or enumerated. If either $Lower$ or $Upper$ is absent, there is not enough information to find it, and an error will be reported.

`gen_nat(?N)`

is true when $N$ is a natural number. If $N$ is a variable, it will enumerate the natural numbers 0,1,2,... and of course not terminate. It is not meant to be applied to anything but integers and variables.
gen_int(?I)

is true when I is an integer. If I is a variable, it will enumerate the integers in the order 0, 1, -1, 2, -2, 3, -3, &c. Of course this sequence has no end. It is not meant to be applied to anything but integers and variables.

repeat(+N)

(where N is a non-negative integer) succeeds exactly N times. You can only understand it procedurally, and really it is only included for compatibility with some other Prologs.

numlist(?Upper, ?List)

is true when List is the list of integers [1, ..., Upper]. For example, numlist(3,L) binds L = [1,2,3].

numlist(?Lower, ?Upper, ?List)

is true when List is [Lower, ..., Upper], Lower and Upper integers. For example, numlist(1,3,L) binds L = [1,2,3].


is true when List is the list of integers [Lower, Lower+Step, ..., Upper] and of length Length. For example, numlist(L,2,U,S,[1,X,Y,Z]) binds L=1, S=4, U=7, X=3, U=5, Z=7.

10.8 Constraint Handling Rules—library(chr)

This section is written by Tom Schrijvers, K.U. Leuven, and adjustments by Jan Wiele-maker.

The CHR system of SICStus Prolog is the K.U.Leuven CHR system. The runtime environment is written by Christian Holzbaur and Tom Schrijvers while the compiler is written by Tom Schrijvers. Both are integrated with SICStus Prolog and licensed under compatible conditions with permission from the authors.

The main reference for the CHR system is [Schrijvers & Demoen 04].

10.8.1 Introduction

Constraint Handling Rules (CHR) is a committed-choice rule-based language embedded in Prolog. It is designed for writing constraint solvers and is particularly useful for providing application-specific constraints. It has been used in many kinds of applications, like scheduling, model checking, abduction, type checking among many others.

CHR has previously been implemented in other Prolog systems (SICStus, Eclipse, Yap), Haskell and Java. This CHR system is based on the compilation scheme and runtime environment of CHR in SICStus.

In this documentation we restrict ourselves to giving a short overview of CHR in general and mainly focus on elements specific to this implementation. For a more thorough review of CHR we refer the reader to [Fruehwirth 98].

In Section 10.8.2 [CHR Syntax and Semantics], page 414, we present the syntax of CHR in Prolog and explain informally its operational semantics. Next, Section 10.8.3 [CHR in
Prolog Programs], page 416, deals with practical issues of writing and compiling Prolog programs containing CHR. Section 10.8.4 [CHR Debugging], page 418, explains the currently primitive CHR debugging facilities. Section 10.8.4.3 [CHR Debugging Predicates], page 420, provides a few useful predicates to inspect the constraint store and Section 10.8.5 [CHR Examples], page 421, illustrates CHR with two example programs. Finally, Section 10.8.6 [CHR Guidelines], page 421, concludes with a few practical guidelines for using CHR.

10.8.2 Syntax and Semantics

10.8.2.1 Syntax

The syntax of CHR rules is the following:

\[
\begin{align*}
\text{rules} & ::= \text{rule rules} \\
\text{rules} & ::= \text{empty} \\
\text{rule} & ::= \text{name actual_rule pragma} \\
\text{name} & ::= \text{atom} @ \\
\text{name} & ::= \text{empty} \\
\text{actual_rule} & ::= \text{simplication_rule} \\
\text{actual_rule} & ::= \text{propagation_rule} \\
\text{actual_rule} & ::= \text{simpagation_rule} \\
\text{simplication_rule} & ::= \text{head} <=> \text{guard body} \\
\text{propagation_rule} & ::= \text{head} => \text{guard body} \\
\text{simpagation_rule} & ::= \text{head} \\text{head} <=> \text{guard body} \\
\text{head} & ::= \text{constraints} \\
\text{constraints} & ::= \text{constraint constraint_id} \\
\text{constraints} & ::= \text{constraint constraint_id}, \text{constraints} \\
\text{constraint} & ::= \text{compound_term} \\
\text{constraint_id} & ::= \text{empty} \\
\text{constraint_id} & ::= \text{# variable} \\
\text{guard} & ::= \text{empty} \\
\text{guard} & ::= \text{goal disj} \\
\text{body} & ::= \text{goal} \\
\text{pragma} & ::= \text{empty} \\
\text{pragma} & ::= \text{pragma actual_pragmas} \\
\text{actual_pragmas} & ::= \text{actual pragma, actual_pragmas} \\
\text{actual pragma} & ::= \text{passive(variable)} \\
\text{disj} & ::= ; | \{ \text{read as ; unless | is declared infix } \}
\end{align*}
\]

Note that the guard of a rule may not contain any goal that binds a variable in the head of the rule with a non-variable or with another variable in the head of the rule. It may however bind variables that do not appear in the head of the rule, e.g. an auxiliary variable introduced in the guard.

Note also that, unless | has been declared as an operator, | and ; are indistinguishable as infix operators—both are read as ; (see Section 4.1.7.3 [ref-syn-syn-sen], page 53). So if e.g. a simplification rule is given as:
\[ \text{head} \leftrightarrow (P ; Q) \]

then CHR will break the ambiguity by treating \( P \) as the guard and \( Q \) as the body, which is probably not what you want. To get the intended interpretation, you must supply a dummy guard ‘\texttt{true} |’:

\[ \text{head} \leftrightarrow \texttt{true} \mid (P ; Q) \]

**Please note:** the above is true as long as you do not declare \( | \) as an infix operator, which is possible since release 4.3 for ISO compliance. Declaring \( | \) as an infix operator will confuse CHR.

### 10.8.2.2 Semantics

In this subsubsection the operational semantics of CHR in Prolog are presented informally. They do not differ essentially from other CHR systems.

When a constraint is called, it is considered an active constraint and the system will try to apply the rules to it. Rules are tried and executed sequentially in the order they are written.

A rule is conceptually tried for an active constraint in the following way. The active constraint is matched with a constraint in the head of the rule. If more constraints appear in the head, then they are looked for among the suspended constraints, which are called passive constraints in this context. If the necessary passive constraints can be found and all match with the head of the rule and the guard of the rule succeeds, then the rule is committed and the body of the rule executed. If not all the necessary passive constraint can be found, then the matching fails or the guard fails, the body is not executed and the process of trying and executing simply continues with the following rules. If for a rule, there are multiple constraints in the head, then the active constraint will try the rule sequentially multiple times, each time trying to match with another constraint.

This process ends either when the active constraint disappears, i.e. it is removed by some rule, or after the last rule has been processed. In the latter case the active constraint becomes suspended.

A suspended constraint is eligible as a passive constraint for an active constraint. The other way it may interact again with the rules, is when a variable appearing in the constraint becomes bound to either a non-variable or another variable involved in one or more constraints. In that case the constraint is triggered, i.e. it becomes an active constraint and all the rules are tried.

**Rule Types.** There are three different kinds of rules, each with their specific semantics:

- **simplification**
  The simplification rule removes the constraints in its head and calls its body.

- **propagation**
  The propagation rule calls its body exactly once for the constraints in its head.
simpagation

The simpagation rule removes the constraints in its head after the \ and then calls its body. It is an optimization of simplification rules of the form:

\( constraints_1, constraints_2 \iff constraints_1, body \)

namely, in the simpagation form:

\( constraints_1 \\backslash constraints_2 \iff body \)

the constraints_1 constraints are not called in the body.

Rule Names. Naming a rule is optional and has no semantical meaning. It only functions as documentation for the programmer.

Pragmas. The semantics of the pragmas are:

\texttt{passive(Identifier)}

The constraint in the head of a rule \texttt{Identifier} can only match a passive constraint in that rule.

Additional pragmas may be released in the future.

Options.

It is possible to specify options that apply to all the CHR rules in the module. Options are specified with the \texttt{chr_option/2} declaration:

\begin{verbatim}
:- chr_option(Option,Value).
\end{verbatim}

and may appear in the file anywhere after the first constraints declaration.

Available options are:

\texttt{check_guard_bindings}

This option controls whether guards should be checked for (illegal) variable bindings or not. Possible values for this option are \texttt{on}, to enable the checks, and \texttt{off}, to disable the checks. If this option is on, then any guard fails when it binds a variable that appears in the head of the rule. When the option is off, the behavior of a binding in the guard is undefined.

\texttt{optimize}

This option controls the degree of optimization. Possible values are \texttt{full}, to enable all available optimizations, and \texttt{off} (the default), to disable all optimizations. If optimization is enabled, then debugging must be disabled.

\texttt{debug}

This options enables or disables the possibility to debug the CHR code. Possible values are \texttt{on} (the default) and \texttt{off}. See Section 10.8.4 [CHR Debugging], page 418, for more details on debugging.

10.8.3 CHR in Prolog Programs

10.8.3.1 Embedding in Prolog Programs

The CHR constraints defined in a \texttt{.pl} file are associated with a module. The default module is \texttt{user}. One should never load different \texttt{.pl} files with the same CHR module name.
10.8.3.2 Constraint Declaration

Every constraint used in CHR rules has to be declared with a `chr_constraint/1` declaration by the constraint specifier. For convenience multiple constraints may be declared at once with the same `chr_constraint/1` declaration followed by a comma-separated list of constraint specifiers.

A constraint specifier is, in its compact form, \( F/A \) where \( F \) and \( A \) are respectively the functor name and arity of the constraint, e.g.

\[
\begin{align*}
\text{:- chr_constraint foo/1.} \\
\text{:- chr_constraint bar/2, baz/3.}
\end{align*}
\]

In its extended form, a constraint specifier is \( c(A_1, \ldots, A_n) \) where \( c \) is the constraint’s functor, \( n \) its arity and the \( A_i \) are argument specifiers. An argument specifier is a mode, optionally followed by a type. E.g.

\[
\begin{align*}
\text{:- chr_constraint get_value(+,?).} \\
\text{:- chr_constraint domain(?int,+list(int)),} \\
\quad \text{alldifferent(?list(int)).}
\end{align*}
\]

A mode is one of the following:

- The corresponding argument of every occurrence of the constraint is always unbound.
- The corresponding argument of every occurrence of the constraint is always ground.
- The corresponding argument of every occurrence of the constraint can have any instantiation, which may change over time. This is the default value.

A type can be a user-defined type or one of the built-in types. A type comprises a (possibly infinite) set of values. The type declaration for a constraint argument means that for every instance of that constraint the corresponding argument is only ever bound to values in that set. It does not state that the argument necessarily has to be bound to a value.

The built-in types are:

- `int` The corresponding argument of every occurrence of the constraint is an integer.
- `float` ... a floating point number.
- `number` ... a number.
- `natural` ... a positive integer.
- `any` The corresponding argument of every occurrence of the constraint can have any type. This is the default value.

User-defined types are algebraic data types, similar to those in Haskell or the discriminated unions in Mercury. An algebraic data type is defined using
:- chr_type type ---> body.

If the type term is a functor of arity zero (i.e. one having zero arguments), then it names a monomorphic type. Otherwise, it names a polymorphic type; the arguments of the functor must be distinct type variables. The body term is defined as a sequence of constructor definitions separated by semi-colons.

Each constructor definition must be a functor whose arguments (if any) are types. Discriminated union definitions must be transparent: all type variables occurring in the body must also occur in the type.

Here are some examples of algebraic data type definitions:

```prolog
:- chr_type color ---> red ; blue ; yellow ; green.
:- chr_type tree ---> empty ; leaf(int) ; branch(tree, tree).
:- chr_type list(T) ---> [] ; [T | list(T)].
:- chr_type pair(T1, T2) ---> (T1 - T2).
```

Each algebraic data type definition introduces a distinct type. Two algebraic data types that have the same bodies are considered to be distinct types (name equivalence).

Constructors may be overloaded among different types: there may be any number of constructors with a given name and arity, so long as they all have different types.

Aliases can be defined using ‘==’. For example, if your program uses lists of lists of integers, then you can define an alias as follows:

```prolog
:- chr_type lli == list(list(int)).
```

### 10.8.3.3 Compilation

The Prolog CHR compiler exploits `user:term_expansion/6` rules to translate the constraint handling rules to plain Prolog. These rules are loaded from `library(chr)`. They are activated after finding a declaration of the format:

```prolog
:- chr_constraint ...
```

It is advised to define CHR rules in a module file, where the module declaration is immediately followed by loading `library(chr)` as exemplified below:

```prolog
:- module(zebra, [ zebra/0 ]).
:- use_module(library(chr)).
:- chr_constraint ...
```

### 10.8.4 Debugging

The CHR debugging facilities are currently rather limited. Only tracing is currently available. To use the CHR debugging facilities for a CHR file it must be compiled for debugging. Generating debug info is controlled by the CHR option `debug`, whose default is derived from the CHR flag `generate_debug_info`. 

```prolog
:- chr_constraint ...
```
10.8.4.1 Ports

For CHR constraints the four standard ports are defined:

- **call**: A new constraint is called and becomes active.
- **exit**: An active constraint exits: it has either been inserted in the store after trying all rules or has been removed from the constraint store.
- **fail**: An active constraint fails.
- **redo**: An active constraint starts looking for an alternative solution.

In addition to the above ports, CHR constraints have five additional ports:

- **wake**: A suspended constraint is woken and becomes active.
- **insert**: An active constraint has tried all rules and is suspended in the constraint store.
- **remove**: An active or passive constraint is removed from the constraint store.
- **try**: An active constraint tries a rule with possibly some passive constraints. The try port is entered just before committing to the rule.
- **apply**: An active constraint commits to a rule with possibly some passive constraints. The apply port is entered just after committing to the rule.

10.8.4.2 Tracing

Tracing is enabled with the `chr_trace/0` predicate and disabled with the `chr_notrace/0` predicate.

When enabled, the tracer will step through the **call**, **exit**, **fail**, **wake** and **apply** ports, accepting debug commands, and simply write out the other ports.

The following debug commands are currently supported:

CHR debug options:

```
<cr>  creep  c  creep
s      skip
a      ancestors
n      nodebug
b      break
f      fail
?      help  h help
```

Their meaning is:

- **creep**: Step to the next port.
- **skip**: Skip to exit port of this call or wake port.
ancestors
Print list of ancestor call and wake ports.
nodebug
Disable the tracer.
break
Enter a recursive Prolog top level. See break/0.
abort
Exit to the top level. See abort/0.
fail
Insert failure in execution.
help
Print the above available debug options.

10.8.4.3 Debugging Predicates
The chr module exports several predicates that allow inspecting and printing the content of the constraint store.

chr_trace/0
Activate the CHR tracer. By default the CHR tracer is activated and deacti-
vated automatically by the Prolog predicates trace/0 and notrace/0.

chr_notrace/0
De-activate the CHR tracer. By default the CHR tracer is activated and deac-
tivated automatically by the Prolog predicates trace/0 and notrace/0.

chr_leash(+Spec)
Define the set of CHR ports on which the CHR tracer asks for user intervention (i.e. stops). Spec is either a list of ports as defined in Section 10.8.4.1 [CHR Ports], page 419, or a predefined alias. Defined aliases are: full to stop at all ports, none or off to never stop, and default to stop at the call, exit, fail, wake and apply ports. See also leash/1.

chr_flag(+FlagName, ?OldValue, ?NewValue)
OldValue is the value of the CHR flag FlagName, and the new value of FlagName is set to NewValue. The valid CHR flag are the following:

toplevel_show_store
If on (the default), then the Prolog top level displays the constraint store at the end of each query. If off, then the top level does not display this.

generate_debug_info
Provides the default if the debug option is not given. The valid values are true and false (the default).

optimize
Provides the default if the optimize option is not given. The valid values are full and off (the default).

chr_show_store(+Mod)
Prints all suspended constraints of module Mod to the current output stream.

find_chr_constraint(-Constraint)
Unifies Constraint with a constraint in the store.
10.8.5 Examples

Here are two example constraint solvers written in CHR.

1. The program below defines a solver with one constraint, \texttt{leq/2}, which is a less-than-or-equal constraint, also known as a partial order constraint.

   ```prolog
   :- module(leq, [leq/2]).
   :- use_module(library(chr)).
   
   :- chr_constraint leq/2.
   reflexivity leq(X,X) <=> true.
   antisymmetry leq(X,Y), leq(Y,X) <=> X = Y.
   idempotence leq(X,Y) \ leq(X,Y) <=> true.
   transitivity leq(X,Y), leq(Y,Z) ==> leq(X,Z).
   
   When the above program is loaded, you can call the \texttt{leq/2} constraint in a query, e.g.:
   
   ```prolog
   | ?- leq(X,Y), leq(Y,Z).
   leq(X,Y), leq(X,Z), leq(Y,Z) ?
   ```

2. The program below implements a simple finite domain constraint solver.

   ```prolog
   :- module(dom, [dom/2]).
   :- use_module(library(chr)).
   :- use_module(library(sets), [intersection/3]).
   
   :- chr_constraint dom(?int,+list(int)).
   :- chr_type list(T) ---> [] ; [T|list(T)].
   
   dom(X,[]) <=> fail.
   dom(X,[Y]) <=> X = Y.
   dom(X,L) <=> nonvar(X) | memberchk(X,L).
   dom(X,L1), dom(X,L2) <=> intersection(L1,L2,L3), dom(X,L3).
   
   When the above program is loaded, you can call the \texttt{dom/2} constraint in a query, e.g.:
   
   ```prolog
   | ?- dom(A,[1,2,3]), dom(A,[3,4,5]).
   A = 3
   ```

Finally, Martin Keser's WebCHR package at \url{http://chr.informatik.uni-ulm.de/~webchr/} contains more than 40 example programs for SICStus 4, complete with documentation and example queries.

10.8.6 Guidelines

In this subsection we cover several guidelines on how to use CHR to write constraint solvers and how to do so efficiently.

Check guard bindings yourself.

It is considered bad practice to write guards that bind variables of the head and to rely on the system to detect this at runtime. It is inefficient and obscures the working of the program.
Set semantics.
The CHR system allows the presence of identical constraints, i.e. multiple con-
straints with the same functor, arity and arguments. For most constraint
solvers, this is not desirable: it affects efficiency and possibly termination.
Hence appropriate simpagation rules should be added of the form:

\[ \text{constraint} \setminus \text{constraint} \leftrightarrow \text{true}. \]

Multi-headed rules.
Multi-headed rules are executed more efficiently when the constraints share one
or more variables.

Mode and type declarations.
Provide mode and type declarations to get more efficient program execution.

Compile once, run many times.
Does consulting your CHR program take a long time? Probably it takes the
CHR compiler a long time to compile the CHR rules into Prolog code. When
you disable optimizations the CHR compiler will be a lot quicker, but you may
lose performance.

10.9 Constraint Logic Programming over Booleans—
library(clpb)

10.9.1 Introduction
The clp(B) system provided by this library module is an instance of the general Constraint
Logic Programming scheme introduced in [Jaffar & Michaylov 87]. It is a solver for con-
straints over the Boolean domain, i.e. the values 0 and 1. The library module is a direct
port from SICStus Prolog 3. It is not supported by SICS in any way.

The Boolean domain is particularly useful for modeling digital circuits, and the constraint
solver can be used for verification, design, optimization etc. of such circuits.

To load the solver, enter the query:

\[
\text{| ?- use_module(library(clpb)).}.
\]

The solver contains predicates for checking the consistency and entailment of a constraint
wrt. previous constraints, and for computing particular solutions to the set of previous
constraints.

The underlying representation of Boolean functions is based on Boolean Decision Diagrams
[Bryant 86]. This representation is very efficient, and allows many combinatorial problems
to be solved with good performance.

Boolean expressions are composed from the following operands: the constants 0 and 1
(FALSE and TRUE), logical variables, and symbolic constants, and from the following con-
nectives. \( P \) and \( Q \) are Boolean expressions, \( X \) is a logical variable, \( Is \) is a list of integers or
integer ranges, and \( Es \) is a list of Boolean expressions:

\[ \sim P \quad \text{True if } P \text{ is false.} \]
Chapter 10: The Prolog Library

\[
P \ast Q \quad \text{True if } P \text{ and } Q \text{ are both true.}
\]
\[
P + Q \quad \text{True if at least one of } P \text{ and } Q \text{ is true.}
\]
\[
P \# Q \quad \text{True if exactly one of } P \text{ and } Q \text{ is true.}
\]
\[
X \sim P \quad \text{True if there exists an } X \text{ such that } P \text{ is true. Same as } P[X/0] + P[X/1].
\]
\[
P =:= Q \quad \text{Same as } \neg P \# Q.
\]
\[
P =\not= Q \quad \text{Same as } P \# Q.
\]
\[
P =< Q \quad \text{Same as } \neg P + Q.
\]
\[
P >= Q \quad \text{Same as } P + \neg Q.
\]
\[
P < Q \quad \text{Same as } \neg P * Q.
\]
\[
P > Q \quad \text{Same as } P * \neg Q.
\]
\[
card(Is, Es) \quad \text{True if the number of true expressions in } Es \text{ is a member of the set denoted by } Is.
\]

Symbolic constants (Prolog atoms) denote parametric values and can be viewed as all-quantified variables whose quantifiers are placed outside the entire expression. They are useful for forcing certain variables of an equation to be treated as input parameters.

### 10.9.2 Solver Interface

The following predicates are defined:

**sat(+Expression)**

*Expression* is a Boolean expression. This checks the consistency of the expression wrt. the accumulated constraints, and, if the check succeeds, tells the constraint that the expression be true.

If a variable \(X\), occurring in the expression, is subsequently unified with some term \(T\), then this is treated as a shorthand for the constraint

\[
| \dashv \text{sat}(X=:=T).
\]

**taut(+Expression, ?Truth)**

*Expression* is a Boolean expression. This asks whether the expression is now entailed by the accumulated constraints (\(Truth=1\)), or whether its negation is entailed by the accumulated constraints (\(Truth=0\)). Otherwise, it fails.

**labeling(+Variables)**

*Variables* is a list of variables. The variables are instantiated to a list of 0s and 1s, in a way that satisfies any accumulated constraints. Enumerates all solutions by backtracking, but creates choicepoints only if necessary.

### 10.9.3 Examples
10.9.3.1 Example 1

\[ \text{sat}(X + Y) \]

\[ \text{sat}(X = \_A*Y#Y) \]

illustrates three facts. First, any accumulated constraints affecting the top-level variables are displayed floundered goals, since the query is not true for all \( X \) and \( Y \). Secondly, accumulated constraints are displayed as \( \text{sat}(V = :=\text{Expr}) \) or \( \text{sat}(V = \_\text{Expr}) \) where \( V \) is a variable and \( \text{Expr} \) is a “polynomial”, i.e. an exclusive or of conjunctions of variables and constants. Thirdly, \( \_A \) had to be introduced as an artificial variable, since \( Y \) cannot be expressed as a function of \( X \). That is, \( X + Y \) is true iff there exists an \( \_A \) such that \( X = \_A*Y#Y \). Let’s check it!

\[ \text{taut}(\_A ^ (X = \_A*Y#Y) = := X + Y, T). \]

\( T = 1 \)

verifies the above answer. Notice that the formula in this query is a tautology, and so it is entailed by an empty set of constraints.

10.9.3.2 Example 2

\[ \text{taut}(A = < \_A ^ C, T). \]

no

\[ \text{taut}(A = < B), \text{sat}(B = < C), \text{taut}(A = < C, T). \]

\( T = 1, \)
\( \text{sat}(A = :=_A*B*C), \)
\( \text{sat}(B = :=_B*C) \)

\[ \text{taut}(a, T). \]

\( T = 0 \)

\[ \text{taut}(\neg a, T). \]

\( T = 0 \)

illustrates the entailment predicate. In the first query, the expression “\( A \) implies \( C \)” is neither known to be true nor false, so the query fails. In the second query, the system is told that “\( A \) implies \( B \)” and “\( B \) implies \( C \)”, so “\( A \) implies \( C \)” is entailed. The expressions in the third and fourth queries are to be read “for each \( a \), \( a \) is true” and “for each \( a \), \( a \) is false”, respectively, and so \( T = 0 \) in both cases since both are unsatisfiable. This illustrates the fact that the implicit universal quantifiers introduced by symbolic constants are placed in front of the entire expression.
10.9.3.3 Example 3

| ?- [user].
| adder(X, Y, Sum, Cin, Cout) :-
|   sat(Sum =:= card([1,3],[X,Y,Cin])),
|   sat(Cout =:= card([2-3],[X,Y,Cin])).
| ^
| % consulted user in module user, 0 msec 424 bytes

| ?- adder(x, y, Sum, cin, Cout).
| sat(Sum=:=cin#x#y),
| sat(Cout=:=x*cin#x*y#y*cin)

| ?- adder(x, y, Sum, 0, Cout).
| sat(Sum=:=x#y),
| sat(Cout=:=x*y)

| ?- adder(X, Y, 0, Cin, 1), labeling([X,Y,Cin]).

Cin = 0,
X = 1,
Y = 1 ? ;

Cin = 1,
X = 0,
Y = 1 ? ;

Cin = 1,
X = 1,
Y = 0 ? ;

illustrates the use of cardinality constraints and models a one-bit adder circuit. The first query illustrates how representing the input signals by symbolic constants forces the output signals to be displayed as functions of the inputs and not vice versa. The second query computes the simplified functions obtained by setting carry-in to 0. The third query asks for particular input values satisfying sum and carry-out being 0 and 1, respectively.

10.9.3.4 Example 4

The predicate fault/3 below describes a 1-bit adder consisting of five gates, with at most one faulty gate. If one of the variables \( F_i \) is equal to 1, then the corresponding gate is faulty, and its output signal is undefined (i.e. the constraint representing the gate is relaxed).

Assuming that we have found some incorrect output from a circuit, we are interesting in finding the faulty gate. Two instances of incorrect output are listed in fault_ex/2:
fault([F1,F2,F3,F4,F5], [X,Y,Cin], [Sum,Cout]) :-
    sat(
        card([0-1],[F1,F2,F3,F4,F5]) * 
        (F1 + (U1 =:= X * Cin)) * 
        (F2 + (U2 =:= Y * U3)) * 
        (F3 + (Cout =:= U1 + U2)) * 
        (F4 + (U3 =:= X # Cin)) * 
        (F5 + (Sum =:= Y # U3))
   ).

fault_ex(1, Faults) :- fault(Faults, [1,1,0], [1,0]).
fault_ex(2, Faults) :- fault(Faults, [1,0,1], [0,0]).

To find the faulty gates, we run the query

| ?- fault_ex(I,L), labeling(L).

I = 1,
L = [0,0,0,1,0] ;

I = 2,
L = [1,0,0,0,0] ;

I = 2,
L = [0,0,1,0,0] ;

no

Thus for input data [1,1,0], gate 4 must be faulty. For input data [1,0,1], either gate 1 or gate 3 must be faulty.

To get a symbolic representation of the outputs in terms of the input, we run the query

| ?- fault([0,0,0,0,0], [x,y,cin], [Sum,Cout]).

sat(Cout=:=x*cin#x*y#y*cin),
sat(Sum=:=cin#x#y)

which shows that the sum and carry out signals indeed compute the intended functions if no gate is faulty.

10.10 Constraint Logic Programming over Finite Domains—
library(clpfd)

10.10.1 Introduction

The clp(fd) solver described in this chapter is an instance of the general Constraint Logic Programming scheme introduced in [Jaffar & Michaylov 87]. This constraint domain is particularly useful for modeling discrete optimization and verification problems such as
scheduling, planning, packing, timetabling etc. The treatise [Van Hentenryck 89] is an excellent exposition of the theoretical and practical framework behind constraint solving in finite domains, and summarizes the work up to 1989.

This solver has the following highlights:

- A rich set of global constraints with state-of-the-art propagators.
- Two classes of propagators are handled internally: indexicals and global propagators.
- Propagators of both classes can be user-defined, by means of programming interfaces.
- The constraints described in this chapter are automatically translated to sets of propagators.
- The truth value of a primitive constraint can be reflected into a 0/1-variable, i.e. a variable with domain 0..1 (reification).

This library fully supports multiple SICStus runtimes in a process.

The rest of this chapter is organized as follows: How to load the solver and how to write simple programs is explained in Section 10.10.3 [CLPFD Interface], page 428. A description of all constraints that the solver provides is contained in Section 10.10.4 [Available Constraints], page 433. The predicates for searching for solution are documented in Section 10.10.5 [Enumeration Predicates], page 470. The predicates for getting execution statistics are documented in Section 10.10.6 [Statistics Predicates], page 477. A few notes on debugging are given in Section 10.10.8 [CLPFD Debugging], page 478. A few example programs are given in Section 10.10.12 [CLPFD Example Programs], page 493. Finally, Section 10.10.13 [Syntax Summary], page 496, contains syntax rules for all expressions.

The following sections discuss advanced features and are probably only relevant to experienced users: How to control the amount of information presented in answers to queries is explained in Section 10.10.7 [Answer Constraints], page 477. How to add new global constraints via a programming interface is described in Section 10.10.9 [Defining Global Constraints], page 478. How to define new primitive constraints with indexicals is described in Section 10.10.10 [Defining Primitive Constraints], page 485. The fine points of coexisting with attributes and blocked goals are described in Section 10.10.11 [CLPFD Coexisting], page 493.

10.10.1.1 Referencing this Software

When referring to this implementation of clp(FD) in publications, please use the following reference:


10.10.1.2 Acknowledgments

The first version of this solver was written as part of Key Hyckenberg’s MSc thesis in 1995, with contributions from Greger Ottosson at the Computing Science Department, Uppsala...
University. The code was later rewritten by Mats Carlsson with contributions by Nicolas Beldiceanu. Péter Szeredi contributed material for this manual chapter.

The development of this software was supported by the Swedish National Board for Technical and Industrial Development (NUTEK) under the auspices of Advanced Software Technology (ASTEC) Center of Competence at Uppsala University.

We include a collection of examples, among which some have been distributed with the INRIA implementation of clp(FD) [Diaz & Codognet 93].

10.10.2 Caveats and Limitations

Following are some general statements about the constraints and libraries of this library module.

Domain Variables

Only small integers (see Chapter 2 [Glossary], page 7) and domain variables are allowed as arguments to finite domain constraints. Whenever a domain variable is required in the argument of a constraint, a small integer can be given instead. The conversion from unbound variable to domain variable is automatic.

Aliasing

In case of variable aliasing, i.e. if a variable occurs more than once in a global constraint that is being posted, or due to a subsequent variable-variable unification, then any guarantee to maintain a particular level of consistency no longer holds, and idempotency is almost always lost.

Termination

Of course, all constraints and predicates terminate. However, due to the combinatorial nature of constraint solving, and to the fact that constraint solving is based on filtering domains, which can be huge, pathological cases where termination takes extremely long time are easily constructed. After about 15,000 years on a 64-bit machine, the following query terminates with a representation error, when the lower bound of \( X \) exceeds the small integer range:

\[
| ?- X #> abs(X).
[^... ... ...]
! Representation error in user: 't=<u+c'/3
! CLPFD integer overflow
! goal: 't=<u+c'(_245,_247,-1)
\]

Anyway, if you find non-pathological cases that take longer than reasonable time to terminate, then please write to sicstus-support@ri.se.

Error Checking

Contrary to most library modules, CLPFD constraints and predicates check their arguments to almost the same extent as built-in predicates. If you find a case where reasonable error checking is missing, then please write to sicstus-support@ri.se.

10.10.3 Solver Interface

The solver contains predicates for checking the consistency and entailment of finite domain constraints, as well as solving for solution values for your problem variables.
In the context of this constraint solver, a *finite domain* is a subset of small integers, and a *finite domain constraint* denotes a relation over a tuple of small integers (see Chapter 2 [Glossary], page 7). Hence, only small integers and unbound variables are allowed in finite domain constraints.

A finite domain is denoted symbolically by a *ConstantRange* (see Section 10.10.13.1 [Syntax of Indexicals], page 496), a special case of which is an interval, written as one of the expressions $A..B$, $A..\text{sup}$, $\text{inf}..B$, or $\text{inf}..\text{sup}$. Here, $A$ and $B$ should be small integers, \text{inf} denotes minus infinity, and \text{sup} denotes plus infinity. **Please note:** inf and sup do not denote integers, they only denote the absence of a lower resp. upper bound. Such ConstantRange terms occur in certain contexts, the most common of which is the unary constraint of the form:

```
| ?- X in 1..5.
X in 1..5
```

which constrains X to be in the given interval. Note that variables do not have to be “declared” in this way before they are used in constraints. If an unconstrained variable occurs in a constraint, then it will be treated as having the domain $\text{inf}..\text{sup}$.

All domain variables, i.e. variables that occur as arguments to finite domain constraints get associated with a finite domain, either explicitly declared by the program, or implicitly imposed by the constraint solver. Temporarily, the domain of a variable may actually be infinite, if it does not have a finite lower or upper bound. If during the computation a variable receives a new lower or upper bound that cannot be represented as a small integer, then an overflow condition is issued. This is expressed as silent failure or as a representation error, subject to the overflow option of \texttt{fd_flag/3}.

The set of current domains of all domain variables is called the domain store. Domain store $S$ is an extension of domain store $T$ if each domain in $S$ is a subset of the corresponding domain in $T$. If some domain is empty, then the store is contradictory and execution backtracks; otherwise, it is consistent.

At the end of a successful computation, all domains have usually become singletons, i.e. the domain variables have become assigned. The domains do not become singletons automatically. Usually, it takes some amount of search to find an assignment that satisfies all constraints. It is the programmer’s responsibility to do so. If some domain variables are left unassigned in a computation, then the garbage collector will preserve all constraint data that is attached to them.

**Please note:** if a term containing domain variables is written, copied, asserted, gathered as a solution to \texttt{findall/3} and friends, or raised as an exception, then those domain variables will be replaced by brand new variables in the copy. To retain the domains and any attached constraints, you can use \texttt{copy_term/3} with \texttt{clpfd:full_answer} asserted (see Section 4.8.7 [ref-lte-cpt], page 129, and Section 10.10.7 [Answer Constraints], page 477). **API change wrt. release 3.**
Every finite domain constraint is implemented by a propagator, or a set of such. Some constraints have alternative propagators with differing properties. All propagators act as coroutines performing incremental constraint solving, removing values from domains, and/or entailment checking. They wake up by changes in the domains of its arguments. A propagator \( P \) can be seen as a function on constraint store \( S \): \( P(S) \) denotes the extension of \( S \) resulting from applying \( P \) on \( S \).

Propagators come in two kinds: indexicals, stateless reactive functional rules implemented by a stack machine and running, and global propagators, usually stateful, implemented in C or Prolog, and using algorithms from many fields of computer science. At the heart of the constraint solver is a scheduler for propagators, where indexicals have priority over global propagators.

Certain properties of propagators are desirable:

**Correct** A correct propagator never removes values that are consistent wrt. its constraint. This property is mandatory.

**Checking** A checking propagator accepts all ground assignments that satisfies the given constraint, and rejects all ground assignments that violate it. This property is also mandatory.

**Contracting** A contracting propagator never adds any value to any domain. This property is also mandatory.

**Monotone** A propagator \( P \) is monotone if, for all domain stores \( S \) and \( T \), \( S \) is an extension of \( T \) implies that \( P(S) \) is an extension of \( P(T) \). This property is not mandatory but helps understanding and debugging.

**Idempotent** A propagator \( P \) is idempotent if, for all domain stores \( S \), \( P(S) \) equals \( P(P(S)) \).

**Domain-Consistent** A domain-consistent propagator removes all inconsistent values. This property is not mandatory and only a few propagators have it. The reason is that the complexity of maintaining domain consistency is often prohibitively high.

**Bounds-Consistent** A bounds-consistent propagator adjusts all inconsistent upper and lower domain bounds. This property is not mandatory, and is implied by domain consistency. This property is more widespread and usually less costly to maintain than domain consistency, but far from all propagators have it.

10.10.3.1 Posting Constraints

A constraint is called as any other Prolog predicate. When called, the constraint is posted to the store. For example:
| ?- X in 1..5, Y in 2..8, X+Y #= T.
X in 1..5,
Y in 2..8,
T in 3..13
| ?- X in 1..5, T in 3..13, X+Y #= T.
X in 1..5,
T in 3..13,
Y in -2..12

Note that the answer constraint shows the domains of nonground query variables, but does not show any constraints that may be attached to them.

Normally, after posting a constraint, propagation to fixpoint is performed, which can be an overkill. The following provides a means of posting a set of constraints in one batch, suspending all propagation until the whole set has been posted. Suspending propagation can significantly reduce posting overhead.

```prolog
fd_batch(+Constraints)  \textit{since release 4.2.1}
```
where \textit{Constraints} should be a list of constraints, user-defined or exported by \texttt{library(clpfd)}. General Prolog goals among the constraints will have undefined behavior.

10.10.3.2 Forgetting Constraints

Normally, once a constraint has been posted, it remains active until its entailment has been detected, or it has been backtracked over. In special circumstances, it may be desirable to completely forget any constraints attached to a given variable, as if they never had been posted. The following predicate can be used for that purpose:

```prolog
fd_purge(+Var)  \textit{since release 4.6}
```
Given a domain variable \textit{Var}, any constraints and domain information attached to \textit{Var} are erased. \textit{Var} ceases to be a domain variable.

10.10.3.3 Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs) are a major class of problems for which this solver is ideally suited. In a CSP, the goal is to pick values from predefined domains for certain variables so that the given constraints on the variables are all satisfied.

As a simple CSP example, let us consider the Send More Money puzzle. In this problem, the variables are the letters S, E, N, D, M, O, R, and Y. Each letter represents a digit between 0 and 9. The problem is to assign a value to each digit, such that SEND + MORE equals MONEY.

A program that solves the puzzle is given below. The program contains the typical three steps of a clp(FD) program:

1. declare the domains of the variables
2. post the problem constraints
3. look for a feasible solution via backtrack search, or look for an optimal solution via branch-and-bound search

Sometimes, an extra step precedes the search for a solution: the posting of surrogate constraints to break symmetries or to otherwise help prune the search space. No surrogate constraints are used in this example.

The domains of this puzzle are stated via the domain/3 goal and by requiring that S and M be greater than zero. The two problem constraint of this puzzle are the equation (sum/8) and the constraint that all letters take distinct values (all_different/1). Finally, the backtrack search is performed by labeling/2. Different search strategies can be encoded in the Type parameter. In the example query, the default search strategy is used (select the leftmost variable, try values in ascending order).

```prolog
:- use_module(library(clpfd)).

mm([S,E,N,D,M,O,R,Y], Type) :-
   domain([S,E,N,D,M,O,R,Y], 0, 9), % step 1
   S#>0, M#>0,
   all_different([S,E,N,D,M,O,R,Y]), % step 2
   sum(S,E,N,D,M,O,R,Y),
   labeling(Type, [S,E,N,D,M,O,R,Y]). % step 3

sum(S, E, N, D, M, O, R, Y) :-
   1000*S + 100*E + 10*N + D +
   1000*M + 100*O + 10*R + E
   #= 10000*M + 1000*O + 100*N + 10*E + Y.

?- mm([S,E,N,D,M,O,R,Y], []).
D = 7,
E = 5,
M = 1,
N = 6,
O = 0,
R = 8,
S = 9,
Y = 2
```

10.10.3.4 Reified Constraints

Instead of merely posting constraints it is often useful to reflect its truth value into a 0/1-variable $B$, so that:

- the constraint is posted if $B$ is set to 1
- the negation of the constraint is posted if $B$ is set to 0
- $B$ is set to 1 if the constraint becomes entailed
- $B$ is set to 0 if the constraint becomes disentailed
This mechanism is known as *reification*. Several frequently used operations can be defined in terms of reified constraints. A reified constraint is written:

\[
\text{?- Constraint } \#<=> B.
\]

where *Constraint* is reifiable. As an example of a constraint that uses reification, consider `exactly(X, L, N)`, defined to be true if `X` occurs exactly `N` times in the list `L`. It can be defined thus:

\[
\text{exactly(_, [], 0).}
\]

\[
\text{exactly}(X, [Y|L], N) :-}
\]

\[
\quad X \#= Y \#<=> B, \]

\[
\quad N \#= M+B, \]

\[
\quad \text{exactly}(X, L, M).
\]

Finally, reified constraints can be used as terms inside arithmetic expressions. The value of the term is 1 if the constraint is true, and 0 otherwise. For example:

\[
\text{?- X \#= 10, B \#= (X\#>=2) + (X\#>=4) + (X\#>=8).}
\]

\[
B = 3, \quad X = 10
\]

10.10.4 Available Constraints

This section describes constraints that can be used with this solver, organized into classes. Unless documented otherwise, constraints are not reifiable and do not guarantee any particular level of consistency. Whenever a domain variable is required in the argument of a constraint, a small integer can be given instead.

10.10.4.1 Arithmetic Constraints

\[
?\text{Expr RelOp ?Expr}
\]

defines an arithmetic constraint. The syntax for *Expr* and *RelOp* is defined by a grammar (see Section 10.10.13.2 [Syntax of Arithmetic Expressions], page 498). Note that the expressions are not restricted to being linear. Constraints over nonlinear expressions, however, will usually yield less constraint propagation than constraints over linear expressions.

Arithmetic constraints can be reified as e.g.:

\[
\text{?- X in 1..2, Y in 3..5, X\#=<Y \#<=> B.}
\]

\[
B = 1, \quad X \text{ in 1..2,} \quad Y \text{ in 3..5}
\]

Linear arithmetic constraints, except equalities, maintain bounds consistency. Their reified versions detect bounds entailment and disentailment.

The following constraints are among the library constraints that general arithmetic constraints compile to. They express a relation between a sum or a scalar product and a value, using a dedicated algorithm, which avoids creating any temporary variables holding
intermediate values. If you are computing a sum or a scalar product, then it can be much more efficient to compute lists of coefficients and variables and post a single sum or scalar product constraint than to post a sequence of elementary constraints.

\[
\text{sum}(+Xs, +RelOp, ?Value)
\]

where \(Xs\) is a list of domain variables, \(RelOp\) is a relational symbol as above, and \(Value\) is an integer or a domain variable. True if \(\text{sum}(Xs) RelOp Value\). Corresponds roughly to \text{sumlist}/2\) in \text{library(lists)}.

\[
\text{scalar_product}(+Coeffs, +Xs, +RelOp, ?Value)
\]

where \(Coeffs\) is a list of length \(n\) of integers, \(Xs\) is a list of length \(n\) of integers or domain variables, \(RelOp\) is a relational symbol as above, and \(Value\) is a domain variable. True if \(
\text{sum}(\text{Coeffs} \times Xs) RelOp Value\).

\(Options\) is a list that may include the following options:

\[
\text{among}(Least, Most, Range) \quad \text{since release 4.3.1}
\]

If given, then \(Least\) and \(Most\) should be integers and \(Range\) should be a \text{ConstantRange} (see Section 10.10.13.1 [Syntax of Indexicals], page 496). This option imposes the additional constraint on \(Xs\) that at least \(Least\) and at most \(Most\) elements belong to \(Range\). This side constraint invokes the algorithm of [Razakarison, Carlsson, Beldiceanu & Simonis 13].

\[
\text{consistency}(Cons)
\]

This can be used to control the level of consistency used by the constraint. The value is one of the following:

- \text{domain}\) The constraint maintains domain consistency. \textbf{Please note}: This option is only meaningful if \(RelOp\) is \=#, and requires that any domain variables have finite bounds.

- \text{bounds}\) The constraint tries to maintain bounds consistency (the default).

\[
\text{scalar_product_reif}(+Coeffs, +Xs, +RelOp, ?Value, ?Reif) \quad \text{since release 4.5}
\]

is the reified version of \text{scalar_product}/[4,5], i.e., \(Reif\) is 1 if \text{scalar_product}/[4,5] with the same argument holds; otherwise, \(Reif\) is 0.

The following constraints constrain a variable to be the minimum (maximum) value of a given list.

\[
\text{minimum}(?Value, +Xs)
\]

where \(Xs\) is a list of domain variables, and \(Value\) is a domain variable. True if \(Value\) is the minimum of \(Xs\). Corresponds to \text{min_member}/2\) in \text{library(lists)} and to \text{minimum} in MiniZinc.
maximum(?Value, +Xs)
where Xs is a list of domain variables, and Value is a domain variable. True if Value is the maximum of Xs. Corresponds to max_member/2 in library(lists) and to maximum in MiniZinc.

The following constraints constrain a variable to be index of the minimum (maximum) value of a given list. They maintain domain consistency.

minimum_arg(+Xs, ?Index) since release 4.6
where Xs is a list of domain variables, and Index is a domain variable. True if Index is the index of the minimum value of Xs. If that value occurs more than once, Index is the index of the first occurrence. Corresponds to arg_min and minimum_arg in MiniZinc.

maximum_arg(+Xs, ?Index) since release 4.6
where Xs is a list of domain variables, and Index is a domain variable. True if Index is the index of the maximum value of Xs. If that value occurs more than once, Index is the index of the first occurrence. Corresponds to arg_max and maximum_arg in MiniZinc.

10.10.4.2 Membership Constraints

domain(+Variables, +Min, +Max)
where Variables is a list of domain variables, Min is an integer or the atom inf (minus infinity), and Max is an integer or the atom sup (plus infinity). True if the variables all are elements of the range Min..Max.

?X in +Range reifiable
where X is a domain variable and Range is a ConstantRange (see Section 10.10.13.1 [Syntax of Indexicals], page 496). True if X is an element of the range.

?X in_set +FDSet reifiable
where X is a domain variable and FDSet is an FD set (see Section 10.10.9.3 [FD Set Operations], page 481). True if X is an element of the FD set.

in/2 and in_set/2 constraints maintain domain consistency and their reified versions detect domain entailment and disentailment.

10.10.4.3 Propositional Constraints

Propositional combinators can be used to combine reifiable constraints into propositional formulae over such constraints. Such formulae are goal expanded by the system into sequences of reified constraints and arithmetic constraints. For example,

\[ X \neq 4 \lor Y \neq 6 \]

expresses the disjunction of two equality constraints.
The leaves of propositional formulae can be reifiable constraints, the constants 0 and 1, or 0/1-variables. New primitive, reifiable constraints can be defined with indexicals as described in Section 10.10.10 [Defining Primitive Constraints], page 485.

The propositional combinators maintain domain consistency and their reified versions detect domain entailment and disentailment. The following propositional combinators are available:

\[
\neg : Q \quad \text{reifiable}
\]

True if the constraint \( Q \) is false.

\[
: P \land : Q \quad \text{reifiable}
\]

True if the constraints \( P \) and \( Q \) are both true.

\[
: P \lor : Q \quad \text{reifiable}
\]

True if exactly one of the constraints \( P \) and \( Q \) is true.

\[
: P \Rightarrow : Q \quad \text{reifiable}
\]

True if at least one of the constraints \( P \) and \( Q \) is true.

\[
: P \Rightarrow : Q \quad \text{reifiable}
\]

True if the constraint \( Q \) is true or the constraint \( P \) is false.

\[
: P \iff : Q \quad \text{reifiable}
\]

True if the constraints \( P \) and \( Q \) are both true or both false.

Note that the reification scheme introduced in Section 10.10.3.4 [Reified Constraints], page 432, is a special case of a propositional constraint.

10.10.4.4 Arithmetic-Logical Constraints

\texttt{smt(:ConstraintBody)} \quad \textit{since release 4.2, deprecated}

The arithmetic, membership, and propositional constraints described earlier are transformed at compile time to conjunctions of library constraints. Although linear in the size of the source code, the expansion of a propositional formula over reifiable constraints to library goals can have time and memory overheads, and propagates disjunctions very weakly. Temporary variables holding intermediate values may have to be introduced, and the grain size of the constraint solver invocations can be rather small. As an alternative to the default propagation of such constraint formulas, this constraint is a front-end to the \texttt{case/[3,4]} propagator, which treats such a formula globally.

Although often convenient, this constraint is deprecated, because it cannot guarantee better performance than a decomposition, nor any particular level of consistency.

\textit{ConstraintBody} should be of one of the following forms, or a propositional combination of such forms. See Section 10.10.13.1 [Syntax of Indexicals], page 496, for the exact definition:

- \texttt{var in ConstantRange}
- \texttt{element(var,CList,var)}
• table([VList],CTable)
• LinExpr RelOp LinExpr
• var { X stands for X#=1 }

count(+Val,+List,+RelOp,?Count)  

since release 4.0.5, deprecated

where Val is an integer, List is a list of domain variables, Count a domain variable, and RelOp is a relational symbol as in Section 10.10.4.1 [Arithmetic Constraints], page 433. True if N is the number of elements of List that are equal to Val and N RelOp Count. Implemented by decomposition into one sum/3 constraint, one arithmetic comparison, and several reified equalities.

Corresponds to count_* and exactly in MiniZinc.

count/4 maintains domain consistency, but in practice, the following constraint is a better alternative.

global_cardinality(+Xs,+Vals)
global_cardinality(+Xs,+Vals,+Options)

where Xs = [X1,…,Xd] is a list of domain variables, and Vals = [K1-V1,…,Kn-Vn] is a list of pairs where each key Ki is a unique integer and Vi is a domain variable. True if every element of Xs is equal to some key and for each pair Ki-Vi, exactly Vi elements of Xs are equal to Ki.

If either Xs or Vals is ground, and in many other special cases, then global_cardinality/[2,3] maintains domain consistency, but generally, bounds consistency cannot be guaranteed. A domain-consistency algorithm [Regin 96] is used, roughly linear in the total size of the domains.

Corresponds to global_cardinality* and distribute in MiniZinc.

Options is a list of zero or more of the following:

consistency(Cons)
Which filtering algorithm to use. One of the following:

domain  The constraint will use the algorithm mentioned above. Implies on(dom). The default.

bounds  The constraint will use the algorithm mentioned above. Implies on(minmax).

value  The constraint will use a simple algorithm, which prevents too few or too many of the Xs from taking values among the Vals. Implies on(val).

on(On)  How eagerly to wake up the constraint. One of the following:

dom  to wake up when the domain of a variable is changed (the default);

minmax  to wake up when a bound of a variable is changed;

val  to wake up when a variable is fixed.

cost(Cost,Matrix)
Overrides any consistency/1 option value. A cost is associated with the constraint and reflected into the domain variable Cost.
Matrix should be a \( d \times n \) matrix of integers, represented as a list of \( d \) lists, each of length \( n \). Assume that each \( X_i \) equals \( K(p_i) \). The cost of the constraint is then \( \text{Matrix}[1,p1]+\ldots+\text{Matrix}[d,pd] \).

With this option, a domain-consistency algorithm [Regin 99] is used, the complexity of which is roughly \( O(d(m + n \log n)) \) where \( m \) is the total size of the domains.

\[
\text{all_equal}(\text{+Variables}) \quad \text{reifiable since release 4.7}
\]

where \( \text{Variables} \) is a list of domain variables. They are constrained to take the same value.

Corresponds to \text{all_equal} in MiniZinc.

\[
\text{all_equal\_reif}(\text{+Variables}, \ ?\text{Reif})) \quad \text{since release 4.7}
\]

is the reified version of \text{all_equal}/1, i.e., \( \text{Reif} \) is 1 if \text{all_equal}/1 with the same argument holds; otherwise, \( \text{Reif} \) is 0.

\[
\text{all_different}(\text{+Variables})
\]

\[
\text{all_different}(\text{+Variables}, \ ?\text{Options})
\]

\[
\text{all_distinct}(\text{+Variables})
\]

\[
\text{all_distinct}(\text{+Variables}, \ ?\text{Options})
\]

where \( \text{Variables} \) is a list of domain variables. Each variable is constrained to take a value that is unique among the variables. Declaratively, this is equivalent to an inequality constraint for each pair of variables.

Corresponds to \text{all_different} in MiniZinc.

\( \text{Options} \) is a list of zero or more of the following:

\[
L \#= R \quad \text{since release 4.3}
\]

where \( R \) should be an integer, and \( L \) an expressions of one of the following forms, where \( X_1, \ldots, X_j \) occur among \( \text{Variables} \):

\[
X_1 + \ldots + X_j
\]

\[
X_1 \times X_1 + \ldots + X_j \times X_j
\]

\[
X_1 \times \ldots \times X_j
\]

The given equation is a side constraint for the constraint to hold. A special bounds-consistency algorithm is used, which considers the main constraint and the side constraints globally. This option is only valid if the domains of \( X_1, \ldots, X_j \) consist of integers strictly greater than zero.

\[
\text{consistency(Cons)}
\]

Which algorithm to use, one of the following:

\[
\text{domain} \quad \text{The default for all_distinct/[1,2] and assignment/[2,3]. A domain-consistency algorithm [Regin 94] is used, roughly linear in the total size of the domains. Implies on(dom).}
\]

\[
\text{bounds} \quad \text{A bounds-consistency algorithm [Lopez-Ortiz 03] is used, which runs in } O(n \log n) \text{ time for } n \text{ variables. Implies on(minmax).}
\]
value  The default for all_different/[1,2]. An algorithm achieving exactly the same pruning as a set of pairwise inequality constraints is used, roughly linear in the number of variables. Implies on(val).

on(On)  How eagerly to wake up the constraint. One of the following:

  dom  (the default for all_distinct/[1,2] and assignment/[2,3]), to wake up when the domain of a variable is changed;

  min  to wake up when the lower bound of a domain is changed;

  max  to wake up when the upper bound of a domain is changed;

  minmax  to wake up when some bound of a domain is changed;

  val  (the default for all_different/[1,2]), to wake up when a variable is fixed.

all_different_except_0(+Variables)  since release 4.6
all_distinct_except_0(+Variables)  since release 4.6
where Variables is a list of domain variables with finite bounds. The variables are constrained to be all different, except those elements that are assigned the value 0.
In terms of consistency, all_different_except_0/1 corresponds to all_different/1, and all_distinct_except_0/1 to all_distinct/1.
Corresponds to alldifferent_except_0 in MiniZinc.

nvalue(?N, +Variables)
where Variables is a list of domain variables with finite bounds, and N is a domain variable. True if N is the number of distinct values taken by Variables. Approximates bounds consistency in N and domain consistency in Variables. Can be thought of as a relaxed version of all_distinct/2.
Corresponds to nvalue in MiniZinc.

The following is a constraint over two lists of length n of variables. Each variable is constrained to take a value in 1..n that is unique for its list. Furthermore, the lists are dual in a sense described below.

assignment(+Xs, +Ys)
assignment(+Xs, +Ys, +Options)
where Xs = [X1,...,Xn] and Ys = [Y1,...,Yn] are lists of domain variables. True if all Xi, Yi are in 1..n and Xi=j iff Yj=i.
Corresponds to inverse in MiniZinc.

The following is a constraint over a list of length n of variables, which all are constrained to 1..n.
symmetric_all_different(+Xs)  since release 4.6
symmetric_all_distinct(+Xs)  since release 4.6

True if the list is all different, and that all Xi in 1..n, and Xi=j iff Xj=i.
Corresponds to symmetric_all_different in MiniZinc. symmetric_all_distinct/1 maintains stronger consistency.

Options is a list of zero or more of the following, where Boolean must be true or false (false is the default):

on(On)    Same meaning as for all_different/2.

consistency(Cons)
    Same meaning as for all_different/2.

circuit(Boolean)
    If true, then circuit(Xs,Ys) must hold for the constraint to be true.

subcircuit(Boolean)  since release 4.6
    If true, then subcircuit(Xs,Ys) must hold for the constraint to be true.

cost(Cost,Matrix)
    A cost is associated with the constraint and reflected into the domain variable Cost. Matrix should be an n*n matrix of integers, represented as a list of lists. The cost of the constraint is Matrix[1,X1]+...+Matrix[n,Xn].
    With this option, a domain-consistency algorithm [Sellmann 02] is used, the complexity of which is roughly O(n(m + n log n)) where m is the total size of the domains.

The following constraint captures the relation between a list of values, a list of the values in ascending order, and their positions in the original list:

sorting(+Xs,+Ps,+Ys)
    where Xs = [X1,..,Xn], Ps = [P1,..,Pn], and Ys = [Y1,..,Yn] are lists of domain variables. The constraint holds if the following are true:
    • Ys is in ascending order.
    • Ps is a permutation of 1..n.
    • for all i in 1..n : Xi = Y(Pi)
    In practice, the underlying algorithm [Mehlhorn 00] is likely to achieve bounds consistency, and is guaranteed to do so if Ps is ground or completely free.
    Corresponds to sort in MiniZinc.

The following constraint is a generalization of sorting/3, namely:

• It sorts not domain variables, but tuples of them.
• The tuples are split into a key prefix and a value suffix. They are sorted wrt. the key part only.
• The sort is stable: if two tuples have identical keys, then their relative order is preserved in the output.

\[ \text{keysorting}(+Xs,+Ys) \quad \text{since release 4.3.1} \]
\[
\text{keysorting}(+Xs,+Ys,+Options)
\]

where \( Xs \) and \( Ys \) are lists of tuples of domain variables. Both lists should be of the same length \( n \), and all tuples should have the same length \( m \). The constraint holds if the following are true:

- \( Ps \) is a permutation of \( 1..n \).
- for all \( i \) in \( 1..n, j \) in \( 1..m : Ys[i,j] = Xs[Ps[i],j] \).
- \([\{Ys[i,1],...,Ys[i,k],Ps[i]\} | i \in 1..n] \) is in lex ascending order, where \( k \) equals \( Keys \) as defined below in the options.

The filtering algorithm is based on [Mehlhorn 00] and endeavors to achieve bounds consistency, but does not guarantee it.

Corresponds to Prolog’s \texttt{keysort/2}. In particular, the sort is stable by definition. Corresponds to \texttt{arg_sort} in MiniZinc.

\( Options \) is a list of zero or more of the following:

\[ \text{keys}(Keys) \]
where \( Keys \) should be a positive integer, denoting the length of the key prefix. The default is 1.

\[ \text{permutation}(Ps) \]
where \( Ps \) should be a list of length \( n \) of domain variables. Its meaning is described above.

The following constraints express the fact that several vectors of domain variables are in ascending lexicographic order:

\[ \text{lex\_chain}(+Vectors) \]
\[ \text{lex\_chain}(+Vectors,+Options) \]

where \( Vectors \) is a list of vectors (lists) of domain variables with finite bounds. The constraint holds if \( Vectors \) are in ascending lexicographic order.

Corresponds to \texttt{*lex2, lex\_greater*, lex\_less*} in MiniZinc.

\( Options \) is a list of zero or more of the following:

\[ \text{op}(Op) \]
If \( Op \) is the atom \#=< (the default), then the constraints holds if \( Vectors \) are in non-descending lexicographic order. If \( Op \) is the atom \#<, then the constraints holds if \( Vectors \) are in strictly ascending lexicographic order.

\[ \text{increasing} \]
This option imposes the additional constraint that each vector in \( Vectors \) be sorted in strictly ascending order.

\[ \text{among}(Least,Most,Values) \]
If given, then \( Least \) and \( Most \) should be integers such that \( 0 \leq Least \leq Most \) and \( Values \) should be a list of distinct integers. This
option imposes the additional constraint on each vector in Vectors that at least Least and at most Most elements belong to Values.

\[ \text{global}(\text{Boolean}) \text{ since release 4.2.1} \]

if true, then a more expensive algorithm \[\text{Carlsson & Beldiceanu 02}\], which guaranteed domain consistency unless the increasing/0 or among/3 options are given, will be used.

In the following constraints, a literal is either a term \(X\) or a term \(\#X\), where \(X\) is a 0/1 variable. They maintain domain consistency:

\[ \text{bool\_and}(\text{+Lits}, \text{+Lit}) \text{ since release 4.3} \]

where \(Lits\) is a list of literals \([L0, \ldots, Lj]\) and \(Lit\) is a literal. True if \(Lit\) equals the Boolean conjunction of \(Lits\), and usually more efficient than the equivalent \(L0#\ldots#\ldj\ #<=> Lit\).

\[ \text{bool\_or}(\text{+Lits}, \text{+Lit}) \text{ since release 4.3} \]

where \(Lits\) is a list of literals \([L0, \ldots, Lj]\) and \(Lit\) is a literal. True if \(Lit\) equals the Boolean disjunction of \(Lits\), and usually more efficient than the equivalent \(L0#\ldots#\ldj\ #<=> Lit\).

\[ \text{bool\_xor}(\text{+Lits}, \text{+Lit}) \text{ since release 4.3} \]

where \(Lits\) is a list of literals \([L0, \ldots, Lj]\) and \(Lit\) is a literal. True if \(Lit\) equals the Boolean exclusive or of \(Lits\), and usually more efficient than the equivalent \(L0#\ldots#\ldj\ #<=> Lit\).

\[ \text{bool\_channel}(\text{+Lits}, ?Y, \text{+Relop}, \text{+Offset}) \text{ since release 4.3} \]

where \(Lits\) is a list of literals \([L0, \ldots, Lj]\), \(Y\) is a domain variable, \(RelOp\) is an arithmetic comparison as in Section 10.10.13.2 [Syntax of Arithmetic Expressions], page 498, and \(Offset\) is an integer. Expresses the constraint \(Li #<=> (Y \ RelOp i+\text{Offset})\) for \(i\) in \(0..j\). Usually more efficient than a bunch of reified comparisons between a given variable and a sequence of integers.

## 10.10.4.5 Extensional Constraints

\[ \text{element}(?X, \text{+List}, ?Y) \text{ reifiable} \]

where \(X\) and \(Y\) are domain variables and \(List\) is a list of domain variables. True if the \(X\):th element of \(List\) equals \(Y\). Operationally, the domains of \(X\) and \(Y\) are constrained so that for every element in the domain of \(X\), there is a compatible element in the domain of \(Y\), and vice versa. Maintains domain consistency in \(X\) and bounds consistency in \(List\) and \(Y\). Corresponds to \text{nth1/3} in \text{library(lists)} and to \text{element} in MiniZinc.

\[ \text{element}(\text{+List}, ?Y) \text{ since release 4.7.0} \]

where \(Y\) is a domain variable and \(List\) is a list of domain variables. True if some element of \(List\) equals \(Y\). Maintains domain consistency in \(X\) and bounds consistency in \(List\) and \(Y\). Corresponds to the built-in \text{member/2} and to \text{member} in MiniZinc.

\[ \text{relation}(?X, \text{+MapList}, ?Y) \text{ since release 4.0.5, deprecated} \]

where \(X\) and \(Y\) are domain variables and \(MapList\) is a list of \text{integer–ConstantRange} pairs, where the integer keys occur uniquely (see
Section 10.10.13.1 [Syntax of Indexicals], page 496). True if MapList contains a pair X-R and Y is in the range denoted by R. Maintains domain consistency. An arbitrary binary constraint can be defined with relation/3. relation/3 is implemented by straightforward transformation to the following, more general constraint, with which arbitrary relations can be defined compactly:

\[
table(+Tuples,+Extension) \quad \text{reifiable}
\]

\[
table(+Tuples,+Extension,+Options)
\]

Defines an n-ary constraint by extension. Extension should be a list of lists of integers, each of length n. Tuples should be a list of lists of domain variables, each also of length n. The constraint holds if every Tuple in Tuples occurs in the Extension. The constraint will maintain domain consistency.

Corresponds to table in MiniZinc.

For convenience, Extension may contain ConstantRange (see Section 10.10.13.1 [Syntax of Indexicals], page 496) expressions in addition to integers.

Options is a list of zero or more of the following:

order(Order) since release 4.1

Determines the variable order used internally. The following values are valid:

leftmost The order is the one given in the arguments (the default).

id3 Each tuple, and the columns of the extension, is permuted according to the heuristic that more discriminating columns should precede less discriminating ones.

method(Method) since release 4.1

Controls the choice of internal data structure and algorithm. The following values are valid:

default since release 4.4

SICStus choice handles propagation as it sees fit (the default).

noaux SICStus handles propagation with the case/[3,4] constraint, see below, converting the extension to a DAG.

aux SICStus handles propagation with the case/[3,4] constraint, see below. Before converting the extension to a DAG, an auxiliary, first variable is introduced, denoting extension row number.

table/[2,3] can be implemented in terms of the following, more general constraint, which allows a compact representation of arbitrary relations:

\[
case(+Template, +Tuples, +Dag)
\]

\[
case(+Template, +Tuples, +Dag, +Options)
\]

This constraint encodes an n-ary constraint, defined by extension and/or linear inequalities. It uses a DAG-shaped data structure where nodes corresponds to
variables and every arc is labeled by an admissible interval for the node above it and optionally by linear inequalities. The variable order is fixed: every path from the root node to a leaf node should visit every variable exactly once, in the order in which they occur lexically in Template. The constraint is true for a single ground tuple if there is a path from the root node to a leaf node such that (a) each tuple element is contained in the corresponding Min..Max interval on the path, and (b) any encountered linear inequalities along the path are true. The case constraint is true for a set of ground tuples if it is true for each tuple of the set. The details are given below.

Template is a nonground Prolog term, each variable of which should occur exactly once. Its variables are merely place-holders; they should not occur outside the constraint nor inside Tuples. Tuples is a list of terms of the same shape as Template. They should not share any variables with Template. Dag is a list of nodes of the form node(ID, X, Children), where X is a template variable, and ID should be an integer, uniquely identifying each node. The first node in the list is the root node. Nodes are either internal nodes or leaf nodes. For an internal node, Children is a list of terms (Min..Max)-ID2 or (Min..Max)-SideConstraints-ID2, where ID2 is the ID of a child node, Min is an integer or the atom inf (minus infinity), and Max is an integer or the atom sup (plus infinity). For a leaf node, Children is a list of terms (Min..Max) or (Min..Max)-SideConstraints. SideConstraints is a list of side constraints of the form scalar_product(Coeffs, Xs, #=<, Bound), where Coeffs is a list of length k of integers, Xs is a list of length k of template variables, and Bound is an integer. Variables in Tuples for which their template variable counterparts are constrained by side constraints, must have bounded domains. In the absence of side constraint, the constraint maintains domain consistency. The use of side constraint is deprecated, because the level of consistency is no better than the alternative, namely to use separate, reified arithmetic constraints and possibly auxiliary variables.

Options is a list of zero or more of the following:

scalar_product(Coeffs, Xs, #=<, Bound) since release 4.2, deprecated

A side constraint located at the root of the DAG.

For example, recall that element(X, L, Y) wakes up when the domain of X or the lower or upper bound of Y has changed, performs full pruning of X, but only prunes the bounds of Y. The following two constraints:

element(X, [1,1,1,1,2,2,2,2], Y),
element(X, [10,10,20,20,10,10,30,30], Z)

can be replaced by the following single constraint, which is equivalent declaratively, but which maintains domain consistency:
elts(X, Y, Z) :-
    case(f(A,B,C), [f(X,Y,Z]),
        [node(0, A, [(1..2)-1, (3..4)-2, (5..6)-3, (7..8)-4]),
        node(1, B, [(1..1)-5]),
        node(2, B, [(1..1)-6]),
        node(3, B, [(2..2)-5]),
        node(4, B, [(2..2)-7]),
        node(5, C, [(10..10)]),
        node(6, C, [(20..20)]),
        node(7, C, [(30..30)])].

The DAG of the previous example has the following shape:

DAG corresponding to elts/3.

A couple of sample queries:
446 SICStus Prolog

| ?- elts(X, Y, Z).
X in 1..8,
Y in 1..2,
Z in \{10\}\/{20}\/{30}\n
| ?- elts(X, Y, Z), Z \\= 15.
X in(3..4)\/{7..8},
Y in 1..2,
Z in \{20\}\/{30}\n
| ?- elts(X, Y, Z), Y = 1.
Y = 1,
X in 1..4,
Z in \{10\}\/{20}\n
As an example with side constraints, consider assigning tasks to machines with given unavailability periods. In this case, one can use a calendar constraint [CHIP 03, Beldiceanu, Carlsson & Rampon 05] to link the real origins of the tasks (taking the unavailability periods into account) with virtual origins of the tasks (not taking the unavailability periods into account). One can then state machine resource constraints using the virtual origins, and temporal constraints between the tasks using the real origins.

Assume, for example, three machines with unavailability periods given by the following table:

```
+---+---+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
+---+---+---+---+---+---+---+---+
| 1 | 2 |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| 2 |   |   | 3 | 4 | 5 |   |   |   |
+---+---+---+---+---+---+---+---+
| 3 |   |   |   |   |   | 4 | 5 |   |
+---+---+---+---+---+---+---+---+
```

Unavailability periods for three machines.

Machine 1 is not available during (real) time periods 1–2 and 6–6, machine 2 is not available during (real) time periods 3–4 and 7–7, and machine 3 is always available.

The following can then be used to express a calendar constraint for a given task scheduled on machine \( M \) in 1..3, with virtual origin \( V \) in 1..8, and real origin \( R \) in 1..8:
calendar(M, V, R) :-
    M in 1..3,
    V in 1..8,
    R in 1..8,
    smt((M#=1 #\ V in 1..3 #\ R#=V+2) #\/
         (M#=1 #\ V in 4..5 #\ R#=V+3) #\/
         (M#=2 #\ V in 1..2 #\ R#=V) #\/
         (M#=2 #\ V in 3..4 #\ R#=V+2) #\/
         (M#=2 #\ V in 5..5 #\ R#=V+3) #\/
         (M#=3 #\ R#=V)).

or equivalently as:

calendar(M, V, R) :-
    case(f(A,B,C),
        [f(M,V,R),
         node(0, A, [(1..1)-1, (2..2)-2, (3..3)-3]),
         node(1, B, [(1..3)-[scalar_product([1,-1],[B,C],#=<,-2),
                       scalar_product([1,-1],[C,B],#=<, 2)]-4,
                       (4..5)-[scalar_product([1,-1],[B,C],#=<,-3),
                       scalar_product([1,-1],[C,B],#=<, 3)]-4]),
         node(2, B, [(1..2)-[scalar_product([1,-1],[B,C],#=<, 0),
                       scalar_product([1,-1],[C,B],#=<, 0)]-4,
                       (3..4)-[scalar_product([1,-1],[B,C],#=<,-2),
                       scalar_product([1,-1],[C,B],#=<, 2)]-4,
                       (5..5)-[scalar_product([1,-1],[B,C],#=<,-3),
                       scalar_product([1,-1],[C,B],#=<, 3)]-4]),
         node(3, B, [(1..8)-[scalar_product([1,-1],[B,C],#=<, 0),
                       scalar_product([1,-1],[C,B],#=<, 0)]-4]),
         node(4, C, [(1..8)]))].

Note that equality must be modeled as the conjunction of inequalities, as only constraints of the form scalar_product(+Coeffs, +Xs, #=<, +Bound) are allowed as side constraints.

The DAG of the calendar constraint has the following shape:
A couple of sample queries:

\[
\begin{align*}
&\text{?- M in 1..3, V in 1..8, R in 1..8,} \\
&M \text{ in 1..3,} \\
&V \text{ in 1..8,} \\
&R \text{ in 1..8} \\
&\text{calendar(M, V, R).}
\end{align*}
\]

\[
\begin{align*}
&M = 1, \\
&V \text{ in 1..5,} \\
&R \text{ in 1..8}
\end{align*}
\]

\[
\begin{align*}
&\text{?- M in 1..3, V in 1..8, R in 1..8,} \\
&M \text{ in 1..3,} \\
&V \text{ in 1..8,} \\
&R \text{ in 1..8} \\
&\text{calendar(M, V, R), M #= 1.}
\end{align*}
\]

\[
\begin{align*}
&M = 2, \\
&V = 5, \\
&R = 8
\end{align*}
\]

10.10.4.6 Graph Constraints

The following constraint can be thought of as constraining $n$ nodes in a graph to form a Hamiltonian (sub-)circuit. The nodes are numbered from 1 to $n$. A full circuit visits each node exactly once and returns to the origin. A subcircuit visits a subset of the nodes exactly once and returns to the origin.
circuit(+Succ)
circuit(+Succ, +Pred)
where Succ is a list of length \( n \) of domain variables. The value of the \( i \):th element of Succ (Pred) is the successor (predecessor) of node \( i \) in the graph. True if the nodes form exactly one Hamiltonian circuit.
Corresponds to circuit in MiniZinc.

subcircuit(+Succ) since release 4.6
subcircuit(+Succ, +Pred) since release 4.6
where Succ is a list of length \( n \) of domain variables. If the value of the \( i \):th element of Succ (Pred) is \( i \), then that corresponds to a node not in the graph. Otherwise, the value is the successor (predecessor) of node \( i \) in the graph. True if the nodes that are included form at most one Hamiltonian subcircuit.
Corresponds to subcircuit in MiniZinc.

10.10.4.7 Scheduling Constraints
The following constraint can be thought of as constraining \( n \) tasks so that the total resource consumption does not exceed a given limit at any time. API change wrt. release 3:

cumulative(+Tasks)
cumulative(+Tasks,+Options)
A task is represented by a term task\((O_i,D_i,E_i,H_i,T_i)\) where \( O_i \) is the start time, \( D_i \) the non-negative duration, \( E_i \) the end time, \( H_i \) the non-negative resource consumption, and \( T_i \) the task identifier. All fields are domain variables with bounded domains.
Let \( n \) be the number of tasks and \( L \) the global resource limit (by default 1, but see below), and:
\[
H_{ij} = H_i, \text{ if } O_i \leq j < O_i + D_i \\
H_{ij} = 0 \text{ otherwise}
\]
The constraint holds if:
1. For every task \( i \), \( O_i + D_i = E_i \), and
2. For all instants \( j \), \( H_1j + \ldots + H_nj \leq L \).
Corresponds to cumulative in MiniZinc. If all durations are 1, then it corresponds to bin_packing in MiniZinc.
Options is a list of zero or more of the following, where Boolean must be true or false (false is the default).

limit\((L)\) See above.

precedences\((Ps)\)
\( Ps \) encodes a set of precedence constraints to apply to the tasks. \( Ps \) should be a list of terms of the form:
\[
T_i - T_j \#= D_{ij}
\]
where \( T_i \) and \( T_j \) should be task identifiers, and \( D_{ij} \) should be a a domain variable, denoting:
\[
O_i - O_j = D_{ij}
\]
global(Boolean)
   if true, then a more expensive algorithm will be used in order to achieve tighter pruning of the bounds of the parameters.

This constraint is due to Aggoun and Beldiceanu [Aggoun & Beldiceanu 93].

The following constraint can be thought of as constraining \( n \) tasks to be placed in time and on \( m \) machines. Each machine has a resource limit, which is interpreted as a lower or upper bound on the total amount of resource used on that machine at any point in time that intersects with some task.

\[
\text{cumulatives(+Tasks,+Machines)}
\]

\[
\text{cumulatives(+Tasks,+Machines,+Options)}
\]

A task is represented by a term \( \text{task}(O_i,D_i,E_i,H_i,M_i) \) where \( O_i \) is the start time, \( D_i \) the non-negative duration, \( E_i \) the end time, \( H_i \) the resource consumption (if positive) or production (if negative), and \( M_i \) a machine identifier. All fields are domain variables with bounded domains.

A machine is represented by a term \( \text{machine}(M_j,L_j) \) where \( M_j \) is the identifier, an integer; and \( L_j \) is the resource bound of the machine, which must be a domain variable with bounded domains.

Let there be \( n \) tasks and:

\[
H_{ijm} = H_i \text{ if } M_i=m \text{ and } O_i \leq j < O_i+D_i
\]

\[
H_{ijm} = 0 \text{ otherwise}
\]

If the resource bound is \textit{lower} (the default), then the constraint holds if:

1. For every task \( i \), \( Si+Di=Ei \), and
2. For all machines \( m \) and instants \( j \) such that there exists a task \( i \) where \( M_i=m \) and \( O_i \leq j < O_i+D_i \), \( H_{1jm}+\ldots+H_{njm} \geq L_m \).

If the resource bound is \textit{upper}, then the constraint holds if:

1. For every task \( i \), \( Si+Di=Ei \), and
2. For all machines \( m \) and instants \( j \), \( H_{1jm}+\ldots+H_{njm} \leq L_m \).

\textit{Options} is a list of zero or more of the following, where \textit{Boolean} must be \text{true} or \text{false} (\text{false} is the default):

\textbf{bound(\textit{B})} If \textit{lower} (the default), then each resource limit is treated as a lower bound. If \textit{upper}, then each resource limit is treated as an upper bound.

\textbf{prune(\textit{P})} If \textit{all} (the default), then the constraint will try to prune as many variables as possible. If \textit{next}, then only variables that occur in the first nonground task term (wrt. the order given when the constraint was posted) can be pruned.

\textbf{generalization(\textit{Boolean})}
   If \text{true}, then extra reasoning based on assumptions on machine assignment will be done to infer more.
task_intervals(Boolean)
If true, then extra global reasoning will be performed in an attempt to infer more.

The following constraint is a generalization of cumulative/[1,2] in the following sense:

- The new constraint deals with the consumption of multiple resources simultaneously, not just a single resource. For the constraint to succeed, none of the resources can exceed its limit.
- Resources can be of two kinds:
  
  cumulative
  This is the kind of resource that cumulative/[1,2] deals with: at no point in time can the total resource use exceed the limit.

  colored
  For this kind of resource, each task specifies not a resource use, but a color, encoded as an integer. At no point in time can the total number of distinct colors in use exceed the limit. The color code 0 is treated specially: it denotes that the task does not have any color.

On the other hand, the new constraint has the limitation that all fields and parameters except start and end times must be given as integers:

multi_cumulative(+Tasks,+Capacities)
   since release 4.3.1
multi_cumulative(+Tasks,+Capacities,+Options)
   since release 4.3.1

A task is represented by a term task(Oi,Di,Ei,Hsi,Ti) where Oi is the start time, Di the non-negative duration, Ei the end time, Hsi the list of non-negative resource uses or colors, and Ti the task identifier. The start and end times should be domain variables with bounded domains. The other fields should be integers.

The capacities should be a list of terms of the following form, where Limit should be a non-negative integer. Capacities and all the Hsi should be of the same length:

cumulative(Limit)
   denotes a cumulative resource.

colored(Limit)
    denotes a colored resource.

Options is a list of zero or more of the following:

greedy(Flag)
   If given, then Flag is a domain variable in 0..1. If Flag equals 1, either initially or by binding Flag during search, then the constraint switches behavior into greedy assignment mode. The greedy assignment will either succeed and assign all start and end times to values that satisfy the constraint, or merely fail. Flag is never bound by the constraint; its sole function is to control the behavior of the constraint.
precedences(Ps)

Ps encodes a set of precedence constraints to apply to the tasks. Ps should be a list of pairs Ti-Tj where Ti and Tj should be task identifiers, denoting that task Ti must complete before task Tj can start.

This constraint is due to [Letort, Beldiceanu & Carlsson 14].

10.10.4.8 Placement Constraints

disjoint1(+Lines)

disjoint1(+Lines,+Options)

constrains a set of lines to be non-overlapping. This constraint is best replaced by diffn/[1,2] in new code.

Lines is a list of terms F(Sj,Dj) or F(Sj,Dj,Tj), Sj and Dj are domain variables with finite bounds denoting the origin and length of line j respectively, F is any functor, and the optional Tj is an atomic term denoting the type of the line. Tj defaults to 0 (zero).

Options is a list of zero or more of the following, where Boolean must be true or false (false is the default):

global(Boolean)

if true, then a redundant algorithm using global reasoning is used to achieve more complete pruning.

wrap(Min,Max)

If used, then the space in which the lines are placed should be thought of as a circle where positions Min and Max coincide, where Min and Max should be integers. That is, the space wraps around. Furthermore, this option forces the domains of the origin variables to be inside [Min,Max-1].

margin(T1,T2,D)

This option imposes a minimal distance D between the end point of any line of type T1 and the origin of any line of type T2. D should be a positive integer or sup. If sup is used, then all lines of type T2 must be placed before any line of type T1.

This option interacts with the wrap/2 option in the sense that distances are counted with possible wrap-around, and the distance between any end point and origin is always finite.

disjoint2(+Rectangles)

disjoint2(+Rectangles,+Options)

constrains a set of rectangles to be non-overlapping. This constraint is best replaced by diffn/[1,2] in new code.

Rectangles is a list of terms F(Xj,Lj,Yj,Hj) or F(Xj,Lj,Yj,Hj,Tj), Xj and Lj are domain variables with finite bounds denoting the origin and size of rectangle j in the X dimension, Yj and Hj are the values for the Y dimension, F is any functor, and the optional Tj is an atomic term denoting the type of the rectangle. Tj defaults to 0 (zero).
Options is a list of zero or more of the following, where Boolean must be true or false (false is the default):

**global(Boolean)**

Disabled.

**wrap(Min1,Max1,Min2,Max2)**

Min1 and Max1 should be either integers or the atoms inf and sup respectively. If they are integers, then the space in which the rectangles are placed should be thought of as a cylinder wrapping around the X dimension where positions Min1 and Max1 coincide. Furthermore, this option forces the domains of the Xj variables to be inside [Min1,Max1-1].

Min2 and Max2 should be either integers or the atoms inf and sup respectively. If they are integers, then the space in which the rectangles are placed should be thought of as a cylinder wrapping around the Y dimension where positions Min2 and Max2 coincide. Furthermore, this option forces the domains of the Yj variables to be inside [Min2,Max2-1].

If all four are integers, then the space is a toroid wrapping around both dimensions.

**margin(T1,T2,D1,D2)**

This option imposes minimal distances D1 in the X dimension and D2 in the Y dimension between the end point of any rectangle of type T1 and the origin of any rectangle of type T2. D1 and D2 should be positive integers or sup. If sup is used, then all rectangles of type T2 must be placed before any rectangle of type T1 in the relevant dimension.

This option interacts with the wrap/4 option in the sense that distances are counted with possible wrap-around, and the distance between any end point and origin is always finite.

**synchronization(Boolean)**

Disabled.

**diffn(+Boxes)**

**since release 4.6**

**diffn(+Boxes,+Options)**

constrains a set of multidimensional boxes to be non-overlapping.

A box is represented by a term [Facet,Facet,...]. A facet is a term of the form Origin-Length, where the domain variables Origin and Length are the coordinate and length of the box in the given dimension. All boxes should have the same dimensionality (length of the box term).

Options is a list of zero or more of the following, where Boolean must be true or false:

**strict(Boolean)**

If false (the default), then the constraint is true iff, for all pairs of boxes i, j, there exists a dimension with respective facets Oi-Li and Oj-Lj and their overlap is zero, i.e.:
If true, then the constraint is true iff, for all pairs of boxes \(i, j\), there exists a dimension with respective facets \(O_i-L_i\) and \(O_j-L_j\) and one precedes the other, i.e.:

\[
O_i+L_i \leq O_j \text{ or } O_j+L_j \leq O_i \text{ or } L_i = 0 \text{ or } L_j = 0
\]

Corresponds to \texttt{diffn*} and \texttt{disjunctive*} in MiniZinc.

\texttt{bin	packing(+Items,+Bins)} \hspace{1cm} \text{since release 4.4}

constrains the placement of items of given size in bins of given capacity, so that
the total load of any bin matches its capacity.

\textit{Items} is a list of terms of the form \texttt{item(Bin,Size)} where \textit{Bin} is a domain variable denoting the bin where the item should be placed, and \textit{Size} is an integer \(\geq 0\) denoting its size.

\textit{Bins} is a list of terms of the form \texttt{bin(ID,Cap)} where \textit{ID} is an integer identifying the bin, and \textit{Cap} is a domain variable denoting its capacity. The \textit{ID} values should be all different.

The constraint holds if every \textit{Bin} equals one of the \textit{ID} values, and for every bin \texttt{bin(ID,Cap)}, the total size of the items assigned to it equals \textit{Cap}.

Corresponds to \texttt{bin_packing*} in MiniZinc.

\texttt{geost(+Objects,+Shapes)} \hspace{1cm} \text{since release 4.1}

\texttt{geost(+Objects,+Shapes,+Options)} \hspace{1cm} \text{since release 4.1}

\texttt{geost(+Objects,+Shapes,+Options,+Rules)} \hspace{1cm} \text{since release 4.1}

constrains the location in space of non-overlapping multi-dimensional \textit{Objects}, each of which taking a shape among a set of \textit{Shapes}.

Each shape is defined as a finite set of \textit{shifted boxes}, where each shifted box is described by a box in a \textit{k}-dimensional space at the given offset with the given sizes. A shifted box is described by a ground term \texttt{sbox(Sid,Offset,Size)} where \textit{Sid}, an integer, is the shape id; \textit{Offset}, a list of \textit{k} integers, denotes the offset of the shifted box from the origin of the object; and \textit{Size}, a list of \textit{k} integers greater than zero, denotes the size of the shifted box. Then, a \textit{shape} is a collection of shifted boxes all sharing the same shape id. The shifted boxes associated with a given shape must not overlap. \textit{Shapes} is thus the list of such \texttt{sbox/3} terms.

Each object is described by a term \texttt{object(Oid,Sid,Origin} where \textit{Oid}, an integer, is the unique object id; \textit{Sid}, a domain variable, is the shape id; and \textit{Origin}, a list of domain variables, is the origin coordinate of the object. If \textit{Sid} is nonground, then the object is said to be \textit{polymorphic}. The possible values for \textit{Sid} are the shape ids that occur in \textit{Shapes}. \textit{Objects} is thus the list of such \texttt{object/3} terms.

If given, then \textit{Options} is a list of zero or more of the following, where \textit{Boolean} must be \texttt{true} or \texttt{false} (\texttt{false} is the default):
lex(ListOfOid)
where ListOfOid should be a list of distinct object ids, denotes that the
origin vectors of the objects according to ListOfOid should be
in ascending lexicographic order. Multiple lex/1 options can be
given, but should mention disjoint sets of objects.

cumulative(Boolean)
If true, then redundant reasoning methods are enabled, based on
projecting the objects onto each dimension.

disjunctive(Boolean)
If true, then cliques of objects are detected that clash in one di-
mension and so must be separated in the other dimension. This
method only applies in the 2D case.

longest_hole(Value,Maxbacks)
This method only applies in the 2D case and in the absence of poly-
morphic objects. Value can be all, true or false. If true, then
the filtering algorithm computes and uses information about holes
that can be tolerated without necessarily failing the constraint. If
all, then more precise information is computed. If false, then no
such information is computed. Maxbacks should be an integer >=
-1 and gives a bound on the effort spent tightening the longest hole
information. Experiments suggest that 1000 may be a reasonable
compromise value.

parconflict(Boolean)
If true, then redundant reasoning methods are enabled, based on
computing the number of items that can be put in parallel in the
different dimensions.

visavis_init(Boolean)
If true, then a redundant method is enabled that statically detects
placements that would cause too large holes. This method can be
quite effective.

visavis_floating(Boolean)
Disabled, because it has not been shown to pay off experimentally
except in rare cases.

visavis(Boolean)
Disabled, because it has not been shown to pay off experimentally.

corners(Boolean)
Disabled, because it has not been shown to pay off experimentally.

task_intervals(Boolean)
Disabled, because it has not been shown to pay off experimentally.

dynamic_programming(Boolean)
If true, then a redundant reasoning method is enabled that solves a
2D knapsack problem for every two adjacent columns of the projec-
tion of the objects onto each dimension. This method has pseudo-polynomial complexity but can be quite powerful.

**polymorphism(Boolean)**

Disabled, because it has not been shown to pay off experimentally.

**pallet_loading(Boolean)**

If true, and if all objects consist of a single shifted box of the same shape, modulo rotations, then a redundant method is enabled that recognizes necessary conditions for this special case.

**overlap(Boolean)**

If true, then the constraint that objects be non-overlapping is lifted. This option is useful mainly in conjunction with the *Rules* argument, in case the placement of objects should be restricted by the *Rules* only.

**volume(Total)**

If given, then *Total* is constrained to be the total volume of *Objects*.

**bounding_box(Lower,Upper)**

*Lower*=[*L1,...,Lk*] and *Upper*=[*U1,...,Uk*] should be lists of domain variables. The following conditions are imposed:

- For every point *P* = [*P1,...,Pk*] occupied by an object, *L1* <= *P1* <= *U1*, ..., *Lk* <= *Pk* < *Uk*.
- For every *j* in 1..*k*, there exists a point *P* = [*P1,...,Pj,...,Pk*] occupied by an object such that *Pj*=*Lj*.
- For every *j* in 1..*k*, there exists a point *P* = [*P1,...,Pj,...,Pk*] occupied by an object such that *Pj*=*Uj-1*.

**fixall(Flag,Patterns)**

If given, then *Flag* is a domain variable in 0..1. If *Flag* equals 1, then either initially or by binding *Flag* during search, the constraint switches behavior into greedy assignment mode. The greedy assignment will either succeed and assign all shape ids and origin coordinates to values that satisfy the constraint, or merely fail. *Flag* is never bound by the constraint; its sole function is to control the behavior of the constraint.

Greedy assignment is done one object at a time, in the order of *Objects*. The assignment per object is controlled by *Patterns*, which should be a list of one or more pattern terms of the form *object(_,SidSpec,OriginSpec)*, where *SidSpec* is a term *min(I)* or *max(I)*, *OriginSpec* is a list of *k* such terms, and *I* is a unique integer between 1 and *k*+1.

The meaning of the pattern is as follows. The variable in the position of *min(1)* or *max(1)* is fixed first; the variable in the position of *min(2)* or *max(2)* is fixed second; and so on. *min(I)* means trying values in ascending order; *max(I)* means descending order.

If *Patterns* contains *m* pattern, then object 1 is fixed according to pattern 1, . . . , object *m* is fixed according to pattern *m*, object *m+1*
is fixed according to pattern 1, and so on. For example, suppose that the following option is given:

\[
\text{fixall}(F, \text{[object(\_, \text{min}(1), [\text{min}(3), \text{max}(2)]), object(\_, \text{max}(1), [\text{min}(2), \text{max}(3)])])}
\]

Then, if the program binds \( F \) to 1, then the constraint enters greedy assignment mode and endeavors to fix all objects as follows.

- For object 1, 3, ..., (a) the shape is fixed to the smallest possible value, (b) the \( Y \) coordinate is fixed to the largest possible value, (c) the \( X \) coordinate is fixed to the smallest possible value.
- For object 2, 4, ..., (a) the shape is fixed to the largest possible value, (b) the \( X \) coordinate is fixed to the smallest possible value, (c) the \( Y \) coordinate is fixed to the largest possible value.

If given, then Rules is a list of zero or more terms of the form shown below, and impose extra constraints on the placement of the objects. For the time being, the details are described in [Carlsson, Beldiceanu & Martin 08]. Please note: the rules require that all shapes of a polymorphic objects consist of the same number of shifted boxes. For example, \( \text{Shapes} = \text{[sbox}(1, [0,0], [3,1]), \text{sbox}(1, [0,1], [2,4]), \text{sbox}(2, [0,0], [3,1])]) \) will not work.

```
sentence ::= macro | fol
macro ::= head --> body
head ::= term { to be substituted by a body }
body ::= term { to substitute for a head }
fol ::= #\ fol { negation } | fol #\ fol { conjunction } | fol #\ fol { disjunction } | fol # => fol { implication } | fol # <= > fol { equivalence } | exists(var, collection, fol) quantification } | forall(var, collection, fol) quantification } | card(var, collection, integer, integer, fol) | true | false | expr relop expr { rational arithmetic } | head { macro application }
expr ::= expr + expr | - expr | expr - expr | min(expr, expr)
```
| max(expr, expr) |
| expr * ground-expr |
| groundexpr * expr |
| expr / ground-expr |
| attref |
| integer |
| fold(var, collection, fop, expr, expr) |
| variable \{ quantified variable \} |
| head \{ macro application \} |

groundexpr ::= expr \{ where expr is ground \}
attref ::= entity ^ attr
attr ::= term \{ attribute name \}
| variable \{ quantified variable \} |
relop ::= #< | #= | #> |
| \#\= | \#=< | \#>= |
fop ::= + | min | max
collection ::= list
| objects(list) \{ list of oids \} |
| sboxes(list) \{ list of sids \} |

Corresponds to geost* in MiniZinc.
The following example shows geost/2 modeling three non-overlapping objects. The first object has four possible shapes, and the other two have two possible shapes each.
\[ \text{domain([X1,X2,X3,Y1,Y2,Y3], 1, 4)}, \]
\[ \text{S1 in 1..4,} \]
\[ \text{S2 in 5..6,} \]
\[ \text{S3 in 7..8,} \]
\[ \text{geost([object(1,S1,[X1,Y1]),} \]
\[ \text{object(2,S2,[X2,Y2]),} \]
\[ \text{object(3,S3,[X3,Y3])],} \]
\[ \text{[sbox(1,[0,0],[2,1]),} \]
\[ \text{sbox(1,[0,1],[1,2]),} \]
\[ \text{sbox(1,[1,2],[3,1]),} \]
\[ \text{sbox(2,[0,0],[3,1]),} \]
\[ \text{sbox(2,[0,1],[1,3]),} \]
\[ \text{sbox(2,[2,1],[1,1]),} \]
\[ \text{sbox(3,[0,0],[2,1]),} \]
\[ \text{sbox(3,[1,1],[1,2]),} \]
\[ \text{sbox(3,[-2,2],[3,1]),} \]
\[ \text{sbox(4,[0,0],[3,1]),} \]
\[ \text{sbox(4,[0,1],[1,1]),} \]
\[ \text{sbox(4,[2,1],[1,3]),} \]
\[ \text{sbox(5,[0,0],[2,1]),} \]
\[ \text{sbox(5,[1,1],[1,1]),} \]
\[ \text{sbox(5,[0,2],[2,1]),} \]
\[ \text{sbox(6,[0,0],[3,1]),} \]
\[ \text{sbox(6,[0,1],[1,1]),} \]
\[ \text{sbox(6,[2,1],[1,1]),} \]
\[ \text{sbox(7,[0,0],[3,2]),} \]
\[ \text{sbox(8,[0,0],[2,3])}).} \]

The shapes are illustrated in the following picture:

geost/2: three objects and eight shapes
A ground solution is shown in the following picture:

The following example shows how to encode in Rules "objects with oid 1, 2 and 3 must all be at least 2 units apart from objects with oid 4, 5 and 6".

\[
\begin{align*}
& \text{(origin}(O1,S1,D) \rightarrow 01^x(D)+S1^t(D)), \\
& \text{(end}(O1,S1,D) \rightarrow 01^x(D)+S1^t(D)+S1^l(D)), \\
& \text{(tooclose}(O1,O2,S1,S2,D) \rightarrow \\
& \quad \text{end}(O1,S1,D)+2 \#> \text{origin}(O2,S2,D) \#/\\
& \quad \text{end}(O2,S2,D)+2 \#> \text{origin}(O1,S1,D)), \\
& \text{(apart}(O1,O2) \rightarrow \\
& \quad \text{forall}(S1,\text{sboxes}([01^\text{sid}]), \\
& \quad \quad \text{forall}(S2,\text{sboxes}([02^\text{sid}]), \\
& \quad \quad \quad \#\ \text{tooclose}(O1,O2,S1,S2,1) \#/\\
& \quad \quad \quad \#\ \text{tooclose}(O1,O2,S1,S2,2))))), \\
& \text{(forall}(O1,\text{objects}([1,2,3]), \\
& \quad \text{forall}(O2,\text{objects}([4,5,6]), \text{apart}(O1,O2)))).
\end{align*}
\]

The following example shows how to encode in Rules "objects 3 and 7 model rooms that must be adjacent and have a common border at least 1 unit long".
[ (origin(O1,S1,D) ---\> O1^x(D)+S1^t(D)),
(end(O1,S1,D) ---\> O1^x(D)+S1^t(D)+S1^l(D)),

(overlap(O1,S1,O2,S2,D) ---\>
  end(O1,S1,D) #> origin(O2,S2,D) #/\
  end(O2,S2,D) #> origin(O1,S1,D)),

(abut(O1,O2) ---\>
  forall(S1,sboxes([O1^sid]),
    forall(S2,sboxes([O2^sid]),
      ((origin(O1,S1,1) #= end(O2,S2,1) #/\
        origin(O2,S2,1) #= end(O1,S1,1)) #/\
       overlap(O1,S1,O2,S2,2)) #/\
      ((origin(O1,S1,2) #= end(O2,S2,2) #/\
        origin(O2,S2,2) #= end(O1,S1,2)) #/\
       overlap(O1,S1,O2,S2,1))))),

  (forall(O1,objects([3]),
    forall(O2,objects([7]), abut(O1,O2))))].

10.10.4.9 Sequence Constraints

The following constraint provides a general way of defining any constraint involving sequences whose checker, i.e. a procedure that classifies ground instances as solutions or non-solutions, can be expressed by a finite automaton, deterministic or nondeterministic, extended with counter operations on its arcs. The point is that it is very much easier to come up with such a checker than to come up with a filtering algorithm for the constraint of interest. In the absence of counters, it maintains domain consistency.

Corresponds to regular\* in MiniZinc, but see also regular/2 below.

automaton\(\text{Signature, SourcesSinks, Arcs}\) since release 4.1
automaton\(\text{Sequence, Template, Signature, SourcesSinks, Arcs, Counters, Initial, Final}\)
automaton\(\text{Sequence, Template, Signature, SourcesSinks, Arcs, Counters, Initial, Final, Options}\) since release 4.1

The arguments are described below in terms of their abstract syntax:

**Sequence** The sequence of terms of interest; abstract grammar category `sequence`.

**Template** A template for an item of the sequence; abstract grammar category `template`. Only relevant if some state transition involving counter arithmetic mentions a variable occurring in `Template`, in which case the corresponding term in a sequence element will be accessed.

**Signature** The signature of `Sequence`; abstract grammar category `signature`. The automaton is not driven by `Sequence` itself, but by `Signature`, which ranges over some alphabet, implicitly defined by the values
used by Arcs. In addition to `automaton/[8,9]`, you must call a constraint that maps `Sequence` to `Signature`.

**SourcesSinks**

The source and sink nodes of the automaton; abstract grammar category `sourcesinks`.

**Arcs**

The arcs (transitions) of the automaton; abstract grammar category `arcs`. Any transition not mentioned is assumed to go to an implicit failure node. An arc optionally contains expressions for updated counter values; by default, the counters remain unchanged. Conditional updates can be specified.

**Counters**

A list of variables, local to the constraint; abstract grammar category `counters`.

**Initial**

A list of initial values, usually instantiated; abstract grammar category `initial`.

**Final**

A list of final values, usually uninstantiated; abstract grammar category `final`.

**Options**

Abstract grammar category `options`; a list of zero or more of the following terms. All but the last option are implemented by adding auxiliary counters to the automaton including the necessary updates in the arcs:

- `valueprec(First,Later,N)` \textit{since release 4.1.3}
  
  $N$ is unified with $n$, computed such that: if the value `Later` occurs in the `Signature`, then `First` occurs $n$ times before the first occurrence of `Later`, otherwise $n=0$.

- `anystretchocc(N)` \textit{since release 4.1.3}
  
  $N$ is unified with the number of (nonempty) stretches of any single value in the `Signature`.

- `stretchocc(ValuePat,N)` \textit{since release 4.1.3}
  
  $N$ is unified with the number of stretches of values matching `ValuePat` (abstract grammar category `valuepat`) in the `Signature`.

- `stretchoccmod(ValuePat,Mod,N)` \textit{since release 4.1.3}
  
  $N$ is unified with the number (modulo `Mod`) of stretches of values matching `ValuePat` (abstract grammar category `valuepat`) the `Signature`.

- `stretchmaxlen(ValuePat,N)` \textit{since release 4.1.3}
  
  $N$ is unified with $n$, computed such that: if values matching `ValuePat` (abstract grammar category `valuepat`) occur the `Signature`, then $n$ is the length of the longest such stretch, otherwise $n=0$. 
stretchminlen(ValuePat,N) \hspace{1em} \textit{since release 4.1.3}

$N$ is unified with $n$, computed such that: if values matching \texttt{ValuePat} (abstract grammar category \texttt{valuepat}) occur the \textit{Signature}, then $n$ is the length of the shortest such stretch, otherwise $n$ is a large integer.

\[ \text{wordocc}(\texttt{WordPat},N) \hspace{1em} \textit{since release 4.1.3} \]

$N$ is unified with the number of words matching \texttt{WordPat} (abstract grammar category \texttt{wordpat}) in the \textit{Signature}.

\[ \text{wordoccm}(\texttt{WordPat},M,N) \hspace{1em} \textit{since release 4.1.3} \]

$N$ is unified with the number (modulo $M$) of words matching \texttt{WordPat} (abstract grammar category \texttt{wordpat}) in the \textit{Signature}.

\[ \text{wordprefix}(\texttt{WordPat},Z0) \hspace{1em} \textit{since release 4.1.3} \]

If the prefix of the \textit{Signature} matches \texttt{WordPat} (abstract grammar category \texttt{wordpat}), then $Z0$ is unified with 1, otherwise with 0.

\[ \text{wordsuffix}(\texttt{WordPat},Z0) \hspace{1em} \textit{since release 4.1.3} \]

If the suffix of the \textit{Signature} matches \texttt{WordPat} (abstract grammar category \texttt{wordpat}), then $Z0$ is unified with 1, otherwise with 0.

\[ \text{state}(\texttt{Map},\texttt{StateSequence}) \hspace{1em} \textit{since release 4.1} \]

For a signature of length $k$, the constraint is implemented by decomposition into $k$ smaller constraints mapping an old state to a new state. The states are represented as domain variables. \texttt{StateSequence} forms the list of these $k+1$ domain variables, starting with the initial state and ending with the final state. \texttt{Map} gives the interpretation of their values: it is a list of pairs \texttt{Node-Value} such that if the $n$th state variable $S_n$ equals $Value$, then the automaton is in state $Node$ having read $n$ symbols.

\[ \text{counterseq}(\texttt{CounterSequence}) \hspace{1em} \textit{since release 4.2.1} \]

Similarly to the list of states, \texttt{CounterSequence} forms the list of the $k+1$ instances of \texttt{Counters}, beginning with \texttt{Initial} and ending with \texttt{Final}.

Abstract syntax:

\[
\begin{align*}
\text{sequence} & ::= \text{list of template} & \{ \text{all of which of the same shape} \} \\
\text{template} & ::= \text{term} & \{ \text{most general shape of the sequence} \} \\
& & \{ \text{its variables should be local to the constraint} \}
\end{align*}
\]

\[
\begin{align*}
\text{signature} & ::= \text{list of variable}
\end{align*}
\]
sourcessinks ::= list of nodespec
nodespec ::= source(node)  {an initial state}
| sink(node)        {an accept state}
node ::= term
arcs ::= list of arc
arc ::= arc(node, integer, node)  {from node, integer, to node}
| arc(node, integer, node, exprs)   {exprs correspond to new counter values}
| arc(node, integer, node, conditional)
conditional ::= (cond -> exprs)  {of same length as counters}
| (conditional ; conditional)
exprs ::= list of Expr
| {Expr as defined in Section 10.10.13.2 [Syntax of Arithmetic Expressions], page 498.}
| {over counters, template and constants}
| {variables occurring in counters correspond to old counter values}
| {variables occurring in template refer to the current element of sequence}
cond ::= constraint
| {over counters, template and constants}
| {must be reifiable or true}
counters ::= list of variable
| {should be local to the constraint}
initial ::= list of dvar
| {of same length as counters}
final ::= list of dvar
| {of same length as counters}
option ::= state(list of term, list of dvar)
| valueprec(integer, integer, dvar)
| anystretchocc(dvar)
| stretchocc(valuepat, dvar)
| stretchoccmod(valuepat, dvar, integer)
| stretchmaxlen(valuepat, dvar)
| stretchminlen(valuepat, dvar)
| wordocc(wordpat, dvar)
| wordoccmod(wordpat, dvar, integer)
| wordprefix(wordpat, dvar) |
| wordsuffix(wordpat, dvar) |

\[
\begin{align*}
\text{valuepat} &::= \text{integer} \\
&\quad| \text{list of integer} \quad\{\text{alternatives}\} \\
&\quad| \text{valuepat/valuepat} \quad\{\text{alternatives}\} \\
\text{wordpat} &::= \text{list of valuepat} \\
\text{dvar} &::= \text{variable or integer}
\end{align*}
\]

If no counters are used, then the arguments Counters, Initial and Final should be \([\,]\). The arguments Template and Sequence are only relevant if some Expr mentions a variable in Template, in which case the corresponding position in Sequence will be used at that point.

The constraint holds for a ground instance Sequence if:

- Signature is the signature corresponding to Sequence.
- The finite automaton encoded by SourcesSinks and Arcs stops in an accept state.
- Any counter arithmetic on the transitions map their Initial values to the Final values.
- Any extra constraint imposed by Options are true.

Here is an example. Suppose that you want to define the predicate inflexion\((N,L,Opt)\) which should hold if \(L\) is a list of domain variables, and \(N\) is the number of times that the sequence order switches between strictly increasing and strictly decreasing. For example, the sequence \([1,1,4,8,8,2,7,1]\) switches order three times.

Such a constraint is conveniently expressed by a finite automaton over the alphabet \([<,=,\,>]\) denoting the order between consecutive list elements. A counter is incremented when the order switches, and is mapped to the first argument of the constraint. The automaton could look as follows:
Automaton for `inflexion/3`
The following piece of code encodes this using `automaton/9`. The auxiliary predicate `inflexion_signature/2` maps the sequence to a signature where the consecutive element order is encoded over the alphabet \([0,1,2]\). We use one counter with initial value 0 and final value \(N\) (an argument of `inflexion/3`). Two transitions increment the counter. All states are accept states.
inflexion(N, Vars, Opt) :-
inflexion_signature(Vars, Sign),
automaton(Sign, _, Sign, [source(s),sink(i),sink(j),sink(s)],
    [arc(s,1,s  ),
     arc(s,2,i  ),
     arc(s,0,j  ),
     arc(i,1,i  ),
     arc(i,2,i  ),
     arc(i,0,j,[C+1]),
     arc(j,1,j  ),
     arc(j,0,j  ),
     arc(j,2,i,[C+1])],
    [C],[0],[N],Opt).

inflexion_signature([], []).
inflexion_signature([_], []) :- !.
inflexion_signature([X,Y|Ys], [S|Ss]) :-
    S in 0..2,
    X #> Y #<=> S #= 0,
    X #= Y #<=> S #= 1,
    X #< Y #<=> S #= 2,
    inflexion_signature([Y|Ys], Ss).

Some queries:
/* count the #inflections of a ground string */
| ?- inflexion(N, [1,1,4,8,8,2,7,1], []).
N = 3 ? RET
yes
/* find strings with two inflections */
| ?- length(L,4), domain(L,0,1),
   inflexion(2,L,[]), labeling([],L).
L = [0,1,0,1] ;
L = [1,0,1,0] ;
no
/* find strings that are strictly increasing, strictly decreasing or all equal */
| ?- length(L,4), domain(L,0,3),
  inflexion(I,L,[anystretchocc(1)]), labeling([],L).
I = 0,
L = [0,0,0,0] ? ;
I = 0,
L = [0,1,2,3] ? ;
I = 0,
L = [1,1,1,1] ? ;
I = 0,
L = [2,2,2,2] ? ;
I = 0,
L = [3,2,1,0] ? ;
I = 0,
L = [3,3,3,3] ? ;
no

/* find strings that contain an increase followed by a decrease */
| ?- length(L,4), domain(L,0,1),
  inflexion(I,L,[wordocc([2,0],1)]), labeling([],L).
I = 1,
L = [0,0,1,0] ? ;
I = 1,
L = [0,1,0,0] ? ;
I = 2,
L = [0,1,0,1] ? ;
I = 2,
L = [1,0,1,0] ? ;
no

This constraint uses techniques from [Beldiceanu, Carlsson & Petit 04] and [Beldiceanu, Carlsson, Flener & Pearson 10].

Following is an interface to automaton/3, which instead of taking a finite automaton argument takes a regular expression argument. Regular expressions are more concise and declarative, whereas finite automata are more procedural. Corresponds to regular in MiniZinc.

`regular(Signature, RegularExpression) since release 4.7.0`

where Signature should be a list of domain variables, and RegularExpression should be a ground Prolog term as described below. The constraint holds if Signature forms a sequence of values that is recognized by the regular expression. Maintains domain consistency.

A regular expression has one of the following forms, where all subterms stand for regular expressions. The expression $L(R)$ denotes the language (set of sequences) that regular expression $R$ recognizes.
recognizes the sequence \([I]\), where \(I\) is an integer.

\(*(R)\) recognizes \(L(R)\) repeated zero or more times.

\(+ (R)\) recognizes \(L(R)\) repeated one or more times.

\(? (R)\) recognizes \(L(R)\) repeated zero or one times. In other words, it recognizes \(L(R)\) as well as the empty sequence.

\([R1,R2,\ldots]\) recognizes the concatenation of \(L(R1), L(R2), \ldots\). In particular, \([]\) recognizes the empty sequence.

\(\{R1,R2,\ldots\}\) recognizes the union of \(L(R1), L(R2), \ldots\). In particular, \(\{\}\) does not recognize anything, i.e., \(L(\{\})\) is the empty set.

\(R1 + R2\) recognizes the concatenation of \(L(R1)\) and \(L(R2)\). Same as \([R1,R2]\).

\(R1 \setminus R2\) recognizes the union of \(L(R1)\) and \(L(R2)\). Same as \(\{R1,R2\}\).

\(R1 \cap R2\) recognizes the intersection of \(L(R1)\) and \(L(R2)\).

\(R1 \setminus R2\) recognizes the set difference of \(L(R1)\) and \(L(R2)\).

For example, suppose that we want to recognize sequences that consist of a stretch of 1s and 2s, followed by a stretch of 0s, followed by a stretch of 1s and 2s:

\[
| \text{?- length(L, 4), regular(L, [+({1,2}), +0), +({1,2})]),}
\text{labeling([], L).}
\]

The following constraints are useful for removing value symmetries.
value_precede_chain(+Values,+Vars) since release 4.5
value_precede_chain(+Values,+Vars,+Options) since release 4.5
holds if for all adjacent pairs v,w in Values, either w does not occur in Vars, or
v occurs earlier than w in Vars.

seq_precede_chain(+Vars) since release 4.6
seq_precede_chain(+Vars,+Options) since release 4.6
The same as the above, for Values = [1,2,...].

Values should be a list of integers, and Vars should be a list of domain variables,
with no restriction on their domains.

Correspond to value_precede_chain and seq_precede_chain in MiniZinc.

Options is a list that may include the following option:

global(Boolean)
If false (the default), then the constraint is implemented by de-
composition to automaton/3. If true, then a custom propagator
is used. Both methods maintain domain consistency, but their rel-
ative performance may vary from case to case.

10.10.4.10 User-Defined Constraints
New, primitive constraints can be added defined by the user on two different levels. On a
higher level, constraints can be defined using the global constraint programming interface;
see Section 10.10.9 [Defining Global Constraints], page 478. Such constraints can embody
specialized algorithms and use the full power of Prolog. They cannot be reified.

On a lower level, new primitive constraints can be defined with indexicals. In this case, they
take part in the basic constraint solving algorithm and express custom designed rules for
special cases of the overall local propagation scheme. Such constraints are called FD predi-
cates; see Section 10.10.10 [Defining Primitive Constraints], page 485. They can optionally
be reified.

10.10.5 Enumeration Predicates
As is usually the case with finite domain constraint solvers, this solver is not complete.
That is, it does not ensure that the set of posted constraints is satisfiable. One must resort
to search (enumeration) to check satisfiability and get particular solutions.

The following predicates provide several variants of search:

indomain(?X)
where X is a domain variable with a bounded domain. Assigns, in increasing
order via backtracking, a feasible value to X.

labeling(:Options, +Variables)
where Variables is a list of domain variables and Options is a list of search
options. The domain variables must all have bounded domains. True if an
assignment of the variables can be found, which satisfies the posted constraints.
first_bound(+BB0, -BB)
later_bound(+BB0, -BB)

Provides an auxiliary service for the value(Enum) option (see below).

minimize(:Goal, ?X)
minimize(:Goal, ?X,+Options) since release 4.3
maximize(:Goal, ?X)
maximize(:Goal, ?X,+Options) since release 4.3

Uses a restart algorithm to find an assignment that minimizes (maximizes) the domain variable X. Goal should be a Prolog goal that constrains X to become assigned, and could be a labeling/2 goal. The algorithm calls Goal repeatedly with a progressively tighter upper (lower) bound on X until a proof of optimality is obtained.

Whether to enumerate every solution that improves the objective function, or only the optimal one after optimality has been proved, is controlled by Options. If given, then it should be a list containing a single atomic value, one of:

best since release 4.3
Return the optimal solution after proving its optimality. This is the default.

all since release 4.3
Enumerate all improving solutions, on backtracking seek the next improving solution. Merely fail after proving optimality.

The Options argument of labeling/2 controls the order in which variables are selected for assignment (variable choice heuristic), the way in which choices are made for the selected variable (value choice heuristic), whether the problem is a satisfaction one or an optimization one, and whether all solutions or only the optimal one should be returned. The options are divided into five groups. One option may be selected per group. Also, the number of assumptions (choices) made during the search can be counted. Finally, limits on the execution time and discrepancy of the search can be imposed:

The following options control the order in which the next variable is selected for assignment. In all cases, ties are broken by selecting the leftmost variable.

leftmost
input_order

The leftmost variable is selected. This is the default.

min
smallest
The variable with the smallest lower bound is selected.

max
largest
The variable with the greatest upper bound is selected.

ff
first_fail
The first-fail principle is used: the variable with the smallest domain is selected.
anti_first_fail \textit{since release 4.3}

The variable with the largest domain is selected.

occurrence \textit{since release 4.3}

The variable that has the most constraints suspended on it is selected. \textbf{Please note}: the mapping from source-code constraints to propagators is subject to change from release to release, and after entailment, it is undefined whether a propagator is counted. Consequently, the behavior of \texttt{ffc} may change from release to release.

\texttt{ffc}

\texttt{most_constrained}

The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by selecting the variable that has the most propagators suspended on it. \textbf{Please note}: the same caveat applies as for \texttt{occurrence}.

max_regret \textit{since release 4.3}

The variable with the largest difference between its first two domain elements is selected.

impact \textit{since release 4.7.0}

The variable that has been involved in the most failures is selected.

dom_w_deg \textit{since release 4.7.0}

The variable is selected that maximizes the quantity failure count divided by domain size.

\texttt{variable}(Sel)

\textit{Sel} is a predicate to select the next variable. Given \texttt{Vars}, the variables that remain to label, it will be called as \texttt{Sel(Vars,Selected,Rest)}.

\textit{Sel} is expected to succeed determinately, unifying \texttt{Selected} and \texttt{Rest} with the selected variable and the remaining list, respectively.

\textit{Sel} should be a callable term, optionally with a module prefix, and the arguments \texttt{Vars,Selected,Rest} will be appended to it. For example, if \texttt{Sel} is \texttt{mod:sel(Param)}, then it will be called as \texttt{mod:sel(Param,Vars,Selected,Rest)}.

The following options control the way in which choices are made for the selected variable \texttt{X}:

\texttt{step}

Makes a binary choice between \texttt{X = B} and \texttt{X \neq B}, where \texttt{B} is the lower or upper bound of \texttt{X}. This is the default.

\texttt{enum}

Makes a multiple choice for \texttt{X} corresponding to the values in its domain.

\texttt{bisect}

Makes a binary choice between \texttt{X =< M} and \texttt{X > M}, where \texttt{M} is the middle of the domain of \texttt{X}, i.e. the mean of \texttt{min(X)} and \texttt{max(X)} rounded down to the nearest integer. This strategy is also known as domain splitting.
median  

Since release 4.3

Makes a binary choice between $X \# M$ and $X \# M$, where $M$ is the median of the domain of $X$. If the domain has an even number of elements, then the smaller middle value is used.

middle  

Since release 4.3

Makes a binary choice between $X \# M$ and $X \# M$, where $M$ is the middle of the domain of $X$, i.e. the mean of $\min(X)$ and $\max(X)$ rounded down to the nearest integer.

value(Enum)  

$Enum$ is a predicate that should prune the domain of $X$, possibly but not necessarily to a singleton. It will be called as $Enum(X, Rest, BB0, BB)$ where $Rest$ is the list of variables that need labeling except $X$, and $BB0$ and $BB$ are parameters described below.

$Enum$ is expected to succeed nondeterminately, pruning the domain of $X$, and to backtrack one or more times, providing alternative prunings. To ensure that branch-and-bound search works correctly, it must call the auxiliary predicate $first\_bound(BB0, BB)$ in its first solution. Similarly, it must call the auxiliary predicate $later\_bound(BB0, BB)$ in any alternative solution.

$Enum$ should be a callable term, optionally with a module prefix, and the arguments $X, Rest, BB0, BB$ will be appended to it. For example, if $Enum$ is $mod:enum(Param)$, then it will be called as $mod:enum(Param, X, Rest, BB0, BB)$.

The following options control the order in which the choices are made for the selected variable $X$. Not useful with the value(Enum) option:

up  
The domain is explored in ascending order. This is the default.

down  
The domain is explored in descending order.

The following options tell the solver whether the given problem is a satisfaction problem or an optimization problem. In a satisfaction problem, we wish to find values for the domain variables, but we do not care about which values. In an optimization problem, we wish to find values that minimize or maximize some objective function reflected in a domain variable:

satisfy  

Since release 4.3

We have a satisfaction problem. Its solutions are enumerated by backtracking. This is the default.

minimize($X$)  

We have an optimization problem, seeking an assignment that minimizes (maximizes) the domain variable $X$. The labeling should constrain $X$ to become assigned for all assignments of $Variables$. It is useful to combine these option with the time_out/2, best, and all options (see below). If these options occur more than once, then the last occurrence overrides previous ones.
The following options are only meaningful for optimization problems. They tell the solver whether to enumerate every solution that improves the objective function, or only the optimal one after optimality has been proved:

**Best** since release 4.3

Return the optimal solution after proving its optimality. This is the default.

**All** since release 4.3

Enumerate all improving solutions, on backtracking seek the next improving solution. Merely fail after proving optimality.

The following options are only meaningful for optimization problems. They tell the solver what search scheme to use, but have no effect on the semantics or on the meaning of other options:

**Bab** since release 4.3

Use a branch-and-bound scheme, which incrementally tightens the bound on the objective as more and more solutions are found. This is the default, and is usually the more efficient scheme.

**Restart** since release 4.3

Use a scheme that restarts the search with a tighter bound on the objective each time a solution is found.

The following option counts the number of assumptions (choices) made during the search:

**Assumptions(K)**

When a solution is found, $K$ is unified with the number of choices made.

Finally, limits on the discrepancy of the search and the execution time can be imposed:

**Discrepancy(D)**

On the path leading to the solution there are at most $D$ choicepoints in which a non-leftmost branch was taken.

**Time_out(Time,Flag)**

See Section 10.47 [lib-timeout], page 867. $Time$ should be an integer number of milliseconds. If the search is exhausted within this time and no solution is found, then the search merely fails, as usual. Otherwise, $Flag$ is bound to a value reflecting the outcome:

**Optimality** since release 4.4

If **Best** was selected in an optimization problem, then the search space was exhausted, having found the optimal solution. The variables are bound to the corresponding values. If **Best** was not selected, this flag value is not used.
success  since release 4.4
If best was selected in an optimization problem, then
the search timed out before the search space was ex-
hausted, having found at least one solution. If best was
not selected, then a solution was simply found before
the time limit. In any case, the variables are bound to
the values corresponding to the latest solution found.

time_out  since release 4.4
If best was selected in an optimization problem, then
the search timed out before any solution was found. If
best was not selected, then the search timed out while
searching for the next solution. The variables are left
unbound.

For example, to enumerate solutions using a static variable ordering, use:

\[
\text{\texttt{| ?- constraints(Variables),}} \\
\text{\texttt{labeling([], Variables).}} \\
\text{\texttt{\%same as [leftmost,step,up,satisfy]}}
\]

To minimize a cost function using branch-and-bound search, computing the best solution
only, with a dynamic variable ordering using the first-fail principle, and domain splitting
exploring the upper part of domains first, use:

\[
\text{\texttt{| ?- constraints(Variables, Cost),}} \\
\text{\texttt{labeling([ff,bisect,down,minimize(Cost)], Variables).}}
\]

To give a time budget and collect the solutions of a satisfiability problem up to the time
limit, use:

\[
\text{\texttt{| ?- constraints(Variables),}} \\
\text{\texttt{findall(Variables, labeling([time_out(Budget,success)|Options]), Solutions).}}
\]

where Flag=success will hold if all solutions were found, and Flag=time_out will hold if
the time expired.

Note that, when used for optimization, labeling/2 has a limitation compared to
minimize/[2,3] and maximize/[2,3]: the variable and value choice heuristics specified by
labeling/2 must apply to the whole set of variables, with no provision for different heuris-
tics for different subsets. As of release 4.3, this limitation has been lifted by the following
predicate:

solve(:Options, :Searches)  since release 4.3
where Options is a list of options of the same shape as taken by labeling/2,
and Searches is a list of labeling/2 and indomain/1 goals, or a single such
goal. The domain variables of Searches must all have bounded domains. True
if the conjunction of Searches is true.
The main purpose of this predicate is for optimization, allowing to use different heuristics in the different Searches. For satisfiability problems, a simple sequence of \texttt{labeling/2} and \texttt{indomain/1} goals does the trick.

The treatment of the \textit{Options}, as well as the suboption lists given in the \texttt{labeling/2} goals of \textit{Searches}, is a bit special. Some options are global for the whole search, and are ignored if they occur in the suboption lists. Others are local to the given \texttt{labeling/2} goal, but provides a default value for the whole search if it occurs in \textit{Options}. The following table defines the role of each option as \texttt{global} or \texttt{local}:

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>global</td>
</tr>
<tr>
<td>anti_first_fail</td>
<td>local</td>
</tr>
<tr>
<td>assumptions/1</td>
<td>global</td>
</tr>
<tr>
<td>bab</td>
<td>global</td>
</tr>
<tr>
<td>best</td>
<td>global</td>
</tr>
<tr>
<td>bisect</td>
<td>local</td>
</tr>
<tr>
<td>discrepancy/1</td>
<td>local</td>
</tr>
<tr>
<td>dom_w_deg</td>
<td>local</td>
</tr>
<tr>
<td>down</td>
<td>local</td>
</tr>
<tr>
<td>enum</td>
<td>local</td>
</tr>
<tr>
<td>ffc</td>
<td>local</td>
</tr>
<tr>
<td>ff</td>
<td>local</td>
</tr>
<tr>
<td>first_fail</td>
<td>local</td>
</tr>
<tr>
<td>impact</td>
<td>local</td>
</tr>
<tr>
<td>input_order</td>
<td>local</td>
</tr>
<tr>
<td>largest</td>
<td>local</td>
</tr>
<tr>
<td>leftmost</td>
<td>local</td>
</tr>
<tr>
<td>maximize/1</td>
<td>global</td>
</tr>
<tr>
<td>max</td>
<td>local</td>
</tr>
<tr>
<td>max_regret</td>
<td>local</td>
</tr>
<tr>
<td>median</td>
<td>local</td>
</tr>
<tr>
<td>middle</td>
<td>local</td>
</tr>
<tr>
<td>minimize/1</td>
<td>global</td>
</tr>
<tr>
<td>min</td>
<td>local</td>
</tr>
<tr>
<td>most_constrained</td>
<td>local</td>
</tr>
<tr>
<td>occurrence</td>
<td>local</td>
</tr>
<tr>
<td>restart</td>
<td>global</td>
</tr>
<tr>
<td>satisfy</td>
<td>global</td>
</tr>
<tr>
<td>smallest</td>
<td>local</td>
</tr>
<tr>
<td>step</td>
<td>local</td>
</tr>
<tr>
<td>time_out/2</td>
<td>global</td>
</tr>
<tr>
<td>up</td>
<td>local</td>
</tr>
<tr>
<td>value/1</td>
<td>local</td>
</tr>
<tr>
<td>variable/1</td>
<td>local</td>
</tr>
</tbody>
</table>

For example, suppose that you want to minimize a cost function using branch-and-bound search, enumerating every improving solution, using left-to-right search on some variables.
followed by first-fail domain splitting search on some other variables. This can be expressed as:

\[
\begin{align*}
| \ ?- & \ constraints([X1,X2,X3,Y1,Y2,Y3], Cost), \\
& solve([minimize(Cost),all], \\
& [labeling([leftmost],[X1,X2,X3]), \\
& labeling([ff,bisect],[Y1,Y2,Y3])).
\end{align*}
\]

### 10.10.6 Statistics Predicates

The following predicates can be used to access execution statistics.

**fd_statistics**

\[
\text{fd_statistics}(?Key, ?Value)
\]

This allows a program to access execution statistics specific to this solver. General statistics about CPU time and memory consumption etc. is available from the built-in predicate \text{statistics/2}.

Without arguments, displays on the standard error stream a summary of the following statistics, and zeroes all counters. With arguments, for each of the possible keys \text{Key}, Value is unified with the current value of a counter, which is simultaneously zeroed. The following counters are maintained:

- **resumptions**
  The number of times a constraint was resumed.

- **entailments**
  The number of times a (dis)entailment was detected by a constraint.

- **prunings**
  The number of times a domain was pruned.

- **backtracks**
  The number of times a contradiction was found by a domain being wiped out, or by a global constraint signaling failure. Other causes of backtracking, such as failed Prolog tests, are not covered by this counter.

- **constraints**
  The number of propagators created.

### 10.10.7 Answer Constraints

By default, the answer constraint only shows the projection of the store onto the variables that occur in the query, but not any constraints that may be attached to these variables, nor any domains or constraints attached to other variables. This is a conscious decision, as no efficient algorithm for projecting answer constraints onto the query variables is known for this constraint system.

It is possible, however, to get a complete answer constraint including all variables that took part in the computation and their domains and attached constraints. This is done by asserting a clause for the following predicate:
If false (the default), then the answer constraint, as well as constraints projected by `copy_term/3`, `clpfd:project_attributes/2`, `clpfd:attribute_goal/2` and their callers, only contain the domains of the query variables. If true, then those constraints contain the domains and any attached constraints of all variables. Initially defined as a dynamic, volatile predicate with no clauses.

10.10.8 Debugging
Code using `library(clpfd)` can be debugged with the usual debugger, but it does not capture all relevant aspects of constraint execution: the propagation cascade and domain changes are not visible. To capture such aspects, a separate, dedicated debugger is available; see Section 10.15 [lib-fdbg], page 532.

The `v` command (print variable bindings) of the usual debugger can be handy. It will endeavor to print the variable bindings of the clause containing the current goal, as well as any goals that are blocked on a variable found among those bindings. In particular, it will show the current domains of such variables. See Section 5.5 [Debug Commands], page 237.

10.10.9 Defining Global Constraints
10.10.9.1 The Global Constraint Programming Interface
This section describes a programming interface by means of which new constraints can be written. The interface consists of a set of predicates provided by this library module. Constraints defined in this way can take arbitrary arguments and may use any constraint solving algorithm, provided it makes sense. Reification cannot be expressed in this interface; instead, reification may be achieved by explicitly passing a 0/1-variable to the constraint in question.

Global constraints have state, which may be updated each time the constraint is resumed. The state information may be used e.g. in incremental constraint solving.

The following two predicates are the principal entrypoints for defining and posting new global constraints:

```prolog
clpfd:dispatch_global(+Constraint, +State0, -State, -Actions) hook
```

Tells the solver how to solve constraints of the form `Constraint`. Defined as a multifile predicate.

When defining a new constraint, a clause of this predicate must be added. Its body defines a constraint solving method and should always succeed determinately. When a global constraint is called or resumed, the solver will call this predicate to deal with the constraint.

**Please note:** the constraint is identified by its principal functor; there is no provision for having two constraints with the same name in different modules. It is good practice to include a cut in every clause of `clpfd:dispatch_global/4`.

**Please note:** During propagation, if the domain of a variable becomes reduced to a single value, then the variable will eventually
be bound to that value, but it is undefined exactly when that happens. Therefore, clauses of \texttt{clpfd:dispatch\_global/4} should not use \texttt{nonvar/1} or \texttt{integer/1} to check if a variable is fixed. Use e.g. \texttt{fd\_min/1} and \texttt{fd\_max/1} instead.

\textit{State0} and \textit{State} are the old and new state respectively.

The constraint solving method must not invoke the constraint solver recursively e.g. by binding variables or posting new constraints; instead, \textit{Actions} should be unified with a list of requests to the solver. Each request should be of the following form:

\begin{description}
\item[\texttt{exit}] The constraint has become entailed, and ceases to exist.
\item[\texttt{fail}] The constraint has become disentailed, causing the solver to backtrack.
\item[\texttt{X = V}] The solver binds \texttt{X} to \texttt{V}.
\item[\texttt{X in R}] The solver constrains \texttt{X} to be a member of the \textit{ConstantRange} \textit{R} (see Section 10.10.13.1 [Syntax of Indexicals], page 496).
\item[\texttt{X in\_set S}] The solver constrains \texttt{X} to be a member of the FD set \textit{S} (see Section 10.10.9.3 [FD Set Operations], page 481).
\item[\texttt{call(Goal)}] The solver calls the goal or constraint \textit{Goal}, which should be module prefixed unless it is a built-in predicate or an exported predicate of the \texttt{clpfd} module. \textit{Goal} is executed as any Prolog goal, but in a context where some constraints may already be enqueued for execution, in which case those constraints will run after the completion of the call request.
\end{description}

\texttt{fd\_global(Constraint, +State, +Susp)}
\texttt{fd\_global(Constraint, +State, +Susp, +Options)}

where \textit{Constraint} is a constraint goal, \textit{State} is its initial state, and \textit{Susp} is a term encoding how the constraint should wake up in response to domain changes. This predicate posts the constraint. \textit{Susp} is a list of \texttt{F(Var)} terms where \texttt{Var} is a variable to suspend on and \texttt{F} is a functor encoding when to wake up:

\begin{description}
\item[\texttt{dom(X)}] wake up when the domain of \texttt{X} has changed
\item[\texttt{min(X)}] wake up when the lower bound of \texttt{X} has changed
\item[\texttt{max(X)}] wake up when the upper bound of \texttt{X} has changed
\item[\texttt{minmax(X)}] wake up when the lower or upper bound of \texttt{X} has changed
\item[\texttt{val(X)}] wake up when the lower and upper bounds of \texttt{X} have coincided
\end{description}

\textit{Options} is a list of zero or more of the following:

\begin{description}
\item[\texttt{source(Term)}] By default, the symbolic form computed by \texttt{copy\_term/3}, and shown in the answer constraint if \texttt{clpfd:full\_answer} holds, equals
Constraint, module name expanded. With this option, the symbolic form will instead be Term. In particular, if Term equals true, then the constraint will not appear in the Body argument of copy_term/3. This can be useful if you are posting some redundant (implied) constraint.

idempotent(Boolean)
If true (the default), then the constraint solving method is assumed to be idempotent. That is, in the scope of clpfd:dispatch_global/4, the solver will not check for the resumption conditions for the given constraint, while performing its Actions. If false, then an action may well cause the solver to resume the constraint that produced the action.

If a variable occurs more than once in a global constraint that is being posted, or due to a variable-variable unification, then the solver will no longer trust the constraint solving method to be idempotent.

For an example of usage, see Section 10.10.9.4 [Global Constraint Example], page 483.

The following predicate controls operational aspects of the constraint solver:

fd_flag(+FlagName, ?OldValue, ?NewValue)
OldValue is the value of the FD flag FlagName, and the new value of FlagName is set to NewValue. The possible FD flag names and values are:

overflow Determines the behavior on integer overflow conditions. Possible values:
  error Raises a representation error (the default).
  fail Silently fails.

degug Controls the visibility of constraint propagation. Possible values are on and off (the default). For internal use by library(fdlib).

10.10.9.2 Reflection Predicates
The constraint solving method needs access to information about the current domains of variables. This is provided by the following predicates, all of which which are constant time operations, except fd_degree/2.

fd_var(?X)
Checks that X is currently an unbound variable that is known to the CLPFD solver.

fd_min(?X, ?Min)
where X is a domain variable. Min is unified with the smallest value in the current domain of X, i.e. an integer or the atom inf denoting minus infinity.

fd_max(?X, ?Max)
where X is a domain variable. Max is unified with the upper bound of the current domain of X, i.e. an integer or the atom sup denoting plus infinity.
Chapter 10: The Prolog Library

fd_size(?X, ?Size)
where X is a domain variable. Size is unified with the size of the current domain of X, if the domain is bounded, or the atom sup otherwise.

fd_set(?X, ?Set)
where X is a domain variable. Set is unified with an FD set denoting the internal representation of the current domain of X; see below.

fd_dom(?X, ?Range)
where X is a domain variable. Range is unified with a ConstantRange (see Section 10.10.13.1 [Syntax of Indexicals], page 496) denoting the current domain of X.

fd_degree(?X, ?Degree)
where X is a domain variable. Degree is unified with the number of constraints that are attached to X.

Please note: this number may include some constraints that have been detected as entailed. Also, Degree is not the number of neighbors of X in the constraint network.

fdfailures(?X, ?Failures)
since release 4.7.0
where X is a domain variable. Failures is unified with the number of times that X has been involved in a failure, as far as the solver can detect it. This number affects the variable selection methods impact and dom_w_deg. Please note: the number is not reset on backtracking, and so it is a counter for the whole search.

fd_set_failures(?X, +Failures)
since release 4.7.0
where X is a domain variable. The failure count of X is set to Failures, which should be a small integer. This can be useful for repeated searches or similar circumstances.

The following predicates can be used for computing the set of variables that are (transitively) connected via constraints to some given variable(s).

fd_neighbors(+Var, -Neighbors)
Given a domain variable Var, Neighbors is the set of other variables, domain or otherwise, that occur with Var in some constraint.

fd_closure(+Vars, -Closure)
Given a list Vars of domain variables, Closure is the set of variables (including Vars) that can be transitively reached via constraints. Thus, fd_closure/2 is the transitive closure of fd_neighbors/2.

10.10.9.3 FD Set Operations
The domains of variables are internally represented compactly as FD set terms. The details of this representation are subject to change and should not be relied on. Therefore, a number of operations on FD sets are provided, as such terms play an important role in the interface. Under no circumstances should you try to construct FD sets with other operations such as unification. The following operations are the primitive ones:
is_fdset(+Set)
   Set is a valid FD set.

empty_fdset(?Set)
   Set is the empty FD set.

fdset_parts(?Set, ?Min, ?Max, ?Rest)
   Set is an FD set, which is a union of the non-empty interval [Min,Max] and the
   FD set Rest, and all elements of Rest are greater than Max+1. Min and Max
   are both integers or the atoms inf and sup, denoting minus and plus infinity,
   respectively. Either Set or all the other arguments must be ground.

The following operations can all be defined in terms of the primitive ones, but in most cases,
a more efficient implementation is used:

empty_interval(+Min, +Max)
   [Min,Max] is an empty interval.

fdset_interval(?Set, ?Min, ?Max)
   Set is an FD set, which is the non-empty interval [Min,Max].

fdset_singleton(?Set, ?Elt)
   Set is an FD set containing Elt only. At least one of the arguments must be
   ground.

fdset_min(+Set, -Min)
   Min is the lower bound of Set.

fdset_max(+Set, -Min)
   Max is the upper bound of Set. This operation is linear in the number of
   intervals of Set.

fdset_size(+Set, -Size)
   Size is the cardinality of Set, represented as sup if Set is infinite. This operation
   is linear in the number of intervals of Set.

list_to_fdset(+List, -Set)
   Set is the FD set containing the elements of List. Slightly more efficient if List
   is ordered.

fdset_to_list(+Set, -List)
   List is an ordered list of the elements of Set, which must be finite.

range_to_fdset(+Range, -Set)
   Set is the FD set containing the elements of the ConstantRange (see
   Section 10.10.13.1 [Syntax of Indexicals], page 496) Range.

fdset_to_range(+Set, -Range)
   Range is a constant interval, a singleton constant set, or a union of such, de-
   noting the same set as Set.

fdset_add_element(+Set1, +Elt -Set2)
   Set2 is Set1 with Elt inserted in it.
fdset_del_element(+Set1, +Elt, -Set2)
  Set2 is like Set1 but with Elt removed.

fdset_disjoint(+Set1, +Set2)
The two FD sets have no elements in common.

fdset_intersect(+Set1, +Set2)
The two FD sets have at least one element in common.

fdset_intersection(+Set1, +Set2, -Intersection)
  Intersection is the intersection between Set1 and Set2.

fdset_intersection(+Sets, -Intersection)
  Intersection is the intersection of all the sets in Sets.

fdset_member(?Elt, +Set)
  is true when Elt is a member of Set. If Elt is unbound, then Set must be finite.

fdset_eq(+Set1, +Set2)
  is true when the two arguments represent the same set i.e. they are identical.

fdset_subset(+Set1, +Set2)
  Every element of Set1 appears in Set2.

fdset_subtract(+Set1, +Set2, -Difference)
  Difference contains all and only the elements of Set1 that are not also in Set2.

fdset_union(+Set1, +Set2, -Union)
  Union is the union of Set1 and Set2.

fdset_union(+Sets, -Union)
  Union is the union of all the sets in Sets.

fdset_complement(+Set, -Complement)
  Complement is the complement of Set wrt. inf..sup.

10.10.9.4 Global Constraint Example
The following example defines a new global constraint exactly(X,L,N), which is true if
X occurs exactly N times in the list L of domain variables. N must be an integer when
the constraint is posted. A version without this restriction and defined in terms of reified
equalities was presented earlier; see Section 10.10.3.4 [Reified Constraints], page 432.

This example illustrates the use of state information. The state has two components: the
list of variables that could still be X, and the number of variables still required to be X.

The constraint is defined to wake up on any domain change.
% exactly.pl

/*
   An implementation of exactly(I, X[1]...X[m], N):

   Necessary condition: 0 <= N <= m.
   Rewrite rules:

   [1] |= X[i]=I  \implies exactly(I, X[1]...X[i-1],X[i+1]...X[m], N-1):
   [2] |= X[i]!=I \implies exactly(I, X[1]...X[i-1],X[i+1]...X[m], N):
   [3] |= N=0     \implies X[1]=I ... X[m]=I
   [4] |= N=m     \implies X[1]!=I ... X[m]! =I

*/
:- use_module(library(clpfd)).

% the entrypoint
exactly(I, Xs, N) :-
   dom_suspensions(Xs, Susp),
   fd_global(exactly(I,Xs,N), state(Xs,N), Susp).

dom_suspensions([], []).
dom_suspensions([X|Xs], [dom(X)|Susp]) :-
   dom_suspensions(Xs, Susp).

% the solver method
:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_), state(Xs0,NO), state(Xs,N), Actions) :-
   exactly_solver(I, Xs0, Xs, NO, N, Actions).

exactly_solver(I, Xs0, Xs, NO, N, Actions) :-
   ex_filter(Xs0, Xs, NO, N, I),
   length(Xs, M),
   ( N=:=0  \rightarrow  Actions = [exit|Ps], ex_neq(Xs, I, Ps)
   ; N=:=M  \rightarrow  Actions = [exit|Ps], ex_eq(Xs, I, Ps)
   ; N>0, N<M  \rightarrow  Actions = []
   ; Actions = [fail]
   ).
Chapter 10: The Prolog Library

% rules [1,2]: filter the X's, decrementing N
ex_filter([], [], N, N, _).

ex_filter([X|Xs], Ys, L, N, I) :- X==I, !,
    M is L-1,
    ex_filter(Xs, Ys, M, N, I).

ex_filter([X|Xs], Ys0, L, N, I) :-
    fd_set(X, Set),
    fdset_member(I, Set), !,
    Ys0 = [X|Ys],
    ex_filter(Xs, Ys, L, N, I).

ex_filter([_|Xs], Ys, L, N, I) :-
    ex_filter(Xs, Ys, L, N, I).

% rule [3]: all must be neq I
ex_neq(Xs, I, Ps) :-
    fdset_singleton(Set0, I),
    fdset_complement(Set0, Set),
    eq_all(Xs, Set, Ps).

% rule [4]: all must be eq I
ex_eq(Xs, I, Ps) :-
    fdset_singleton(Set, I),
    eq_all(Xs, Set, Ps).

eq_all([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :-
    eq_all(Xs, Set, Ps).

end_of_file.

% sample queries:

| ?- exactly(5,[A,B,C],1), A=5.  
A = 5,  
B in(inf..4)\/(6..sup), C in(inf..4)\/(6..sup)  
| ?- exactly(5,[A,B,C],1), A in 1..2, B in 3..4.  
C = 5,  
A in 1..2,  
B in 3..4

10.10.10 Defining Primitive Constraints

Indexicals are the principal means of defining constraints, but it is usually not necessary to resort to this level of programming—most commonly used constraints are available in a library and/or via macro-expansion. The key feature about indexicals is that they give
the programmer precise control over aspects of the operational semantics of the constraints. Trade-offs can be made between the computational cost of the constraints and their pruning power. The indexical language provides many degrees of freedom for the user to select the level of consistency to be maintained depending on application-specific needs.

10.10.10.1 Definitions

For constraint store S, variable X, and finite domain R:

- D(X,S) denotes the domain of X in S.
- (X in R)(S) denotes the extension of S where D(X,S) has been intersected with R.

The following definitions, adapted from [Van Hentenryck et al. 95], define important notions of consistency and entailment of constraints wrt. stores.

A ground constraint is true if it holds and false otherwise.

A constraint C is domain-consistent wrt. S iff, for each variable Xi and value Vi in D(Xi,S), there exist values Vj in D(Xj,S), 1 <= j <= n \(\land i \neq j\), such that \(C(V_1, \ldots, V_n)\) is true.

A constraint C is domain-entailed by S iff, for all values Vj in D(Xj,S), 1 <= j <= n, \(C(V_1, \ldots, V_n)\) is true.

Let \(D'(X,S)\) denote the interval \([\min(D(X,S)), \max(D(X,S)))]\).

A constraint C is bounds-consistent wrt. S iff, for each variable Xi, there exist values Vj and Wj in \(D'(Xj,S)\), 1 <= j <= n \(\land i \neq j\), such that \(C(V_1, \ldots, \min(D(Xi,S)), \ldots, V_n)\) and \(C(W_1, \ldots, \max(D(Xi,S)), \ldots, W_n)\) are both true.

A constraint C is bounds-entailed by S iff, for all values Vj in \(D'(Xj,S)\), 1 <= j <= n, \(C(V_1, \ldots, V_n)\) is true.

Finally, a constraint is domain-disentailed (bounds-disentailed) by S iff its negation is domain-entailed (bounds-entailed) by S.

10.10.10.2 Pitfalls of Interval Reasoning

In most circumstances, arithmetic constraints maintain bounds consistency and detect bounds entailment and disentailment. There are cases where a constraint maintaining bounds consistency may detect a contradiction when the constraint is not yet bounds-disentailed, as the following example illustrates. Note that \(X \neq Y\) maintains domain consistency if both arguments are constants or variables:

```
| ?- X+Y =:= Z, X=1, Z=6, Y in 1..10, Y \neq 5.
  no
| ?- X+Y =:= Z \leftrightarrow B, X=1, Z=6, Y in 1..10, Y \neq 5.
  X = 1,
  Z = 6,
  Y in(1..4)\setminus(6..10),
  B in 0..1
```
Since 1+5≠6 holds, X+Y ≠ Z is not bounds-disentailed, although any attempt to make it bounds-consistent wrt. the store results in a contradictory store.

10.10.10.3 Indexicals

An indexical is a reactive functional rule of the form X in R, where R is a finite domain valued range expression (see below). See Section 10.10.13.1 [Syntax of Indexicals], page 496, for a grammar defining indexicals and range expressions.

Indexicals can play one of two roles: propagating indexicals are used for constraint solving, and checking indexicals are used for entailment checking. Let S(R) denote the value of R in S. When a propagating indexical fires, the current store S is extended to (X in S(R))(S). When a checking indexical fires, it checks if D(X,S) is contained in S(R), in which case the constraint corresponding to the indexical is detected as entailed.

10.10.10.4 Range Expressions

A range expression has one of the following forms, where Ri denote range expressions, Ti denote integer valued term expressions, S(Ti) denotes the integer value of Ti in S, X denotes a variable, I denotes an integer, and S denotes the current store.

dom(X) evaluates to D(X,S)
{Ti,..,Tn} evaluates to \{S(T1),...S(Tn)\}. Any Ti containing a variable that is not “quantified” by unionof/3 will cause the indexical to suspend until this variable has been assigned.
T1..T2 evaluates to the interval between S(T1) and S(T2).
R1\R2 evaluates to the intersection of S(R1) and S(R2)
R1\/R2 evaluates to the union of S(R1) and S(R2)
\R2 evaluates to the complement of S(R2)
R1+R2 evaluates to S(R2) or S(T2) added pointwise to S(R1)
-R2 evaluates to S(R2) negated pointwise
R1-R2 evaluates to S(R2) or S(T2) subtracted pointwise from S(R1) or S(T1)
R1 mod R2 evaluates to the pointwise floored modulo of S(R1) and S(R2) or S(T2)
R1 rem R2 evaluates to the pointwise truncated remainder of S(R1) and S(R2) or S(T2)
R1 ? R2 evaluates to S(R2) if S(R1) is a non-empty set; otherwise, evaluates to the empty set. This expression is commonly used in the context (R1 ? (inf..sup) \( \text{\textbackslash} \) R3), which evaluates to S(R3) if S(R1) is an empty set; otherwise, evaluates to inf..sup. As an optimization, R3 is not evaluated while the value of R1 is a non-empty set.
unionof(X,R1,R2)
evaluates to the union of \( S(E_1), \ldots, S(E_N) \), where each \( E_i \) has been formed by substituting \( K \) for \( X \) in \( R_2 \), where \( K \) is the \( i \)-th element of \( S(R_1) \). See Section 10.10.12.2 [N Queens], page 494, for an example of usage.

Please note: if \( S(R_1) \) is infinite, then the evaluation of the indexical will be abandoned, and the indexical will simply suspend.

switch(T,MapList)
evaluates to \( S(E) \) if \( S(T_1) \) equals \( K \) and \( MapList \) contains a pair \( K-E \). Otherwise, evaluates to the empty set. If \( T \) contains a variable that is not “quantified” by \( \text{unionof/3} \), then the indexical will suspend until this variable has been assigned.

10.10.10.5 Term Expressions
A term expression has one of the following forms, where \( T_1 \) and \( T_2 \) denote term expressions, \( X \) denotes a variable, \( I \) denotes an integer, and \( S \) denotes the current store.

\[
\begin{align*}
\text{min}(X) & \quad \text{evaluates to the minimum of } D(X,S) \\
\text{max}(X) & \quad \text{evaluates to the maximum of } D(X,S) \\
\text{card}(X) & \quad \text{evaluates to the size of } D(X,S) \\
X & \quad \text{evaluates to the integer value of } X. \text{ The indexical will suspend until } X \text{ is assigned.}
\end{align*}
\]

\( I \) an integer

\( \text{inf} \) minus infinity

\( \text{sup} \) plus infinity

\( -T_1 \) evaluates to \( S(T_1) \) negated

\( T_1+T_2 \) evaluates to the sum of \( S(T_1) \) and \( S(T_2) \)

\( T_1-T_2 \) evaluates to the difference of \( S(T_1) \) and \( S(T_2) \)

\( T_1*T_2 \) evaluates to the product of \( S(T_1) \) and \( S(T_2) \), where \( S(T_2) \) must not be negative

\( T_1/T_2 \) evaluates to the floored quotient of \( S(T_1) \) and \( S(T_2) \), where \( S(T_2) \) must be positive

\( T_1<T_2 \) evaluates to the ceilinged quotient of \( S(T_1) \) and \( S(T_2) \), where \( S(T_2) \) must be positive

\( T_1 \bmod T_2 \) evaluates to the floored remainder of \( S(T_1) \) and \( S(T_2) \)

\( T_1 \bmod T_2 \) evaluates to the truncated remainder of \( S(T_1) \) and \( S(T_2) \)

10.10.10.6 Monotonicity of Ranges
A range \( R \) is monotone in \( S \) iff the value of \( R \) in \( S' \) is contained in the value of \( R \) in \( S \), for every extension \( S' \) of \( S \). A range \( R \) is anti-monotone in \( S \) iff the value of \( R \) in \( S \) is contained in the value of \( R \) in \( S' \), for every extension \( S' \) of \( S \). By abuse of notation, we will say that \( X \in R \) is (anti-)monotone iff \( R \) is (anti-)monotone.
The consistency or entailment of a constraint $C$ expressed as indexicals $X \in R$ in a store $S$ is checked by considering the relationship between $D(X,S)$ and $S(R)$, together with the (anti-)monotonicity of $R$ in $S$. The details are given in Section 10.10.10.8 [Execution of Propagating Indexicals], page 491, and Section 10.10.10.9 [Execution of Checking Indexicals], page 492.

The solver checks (anti-)monotonicity by requiring that certain variables occurring in the indexical be ground. This sufficient condition can sometimes be false for an (anti-)monotone indexical, but such situations are rare in practice.

### 10.10.10.7 FD Predicates

The following example defines the constraint $X + Y = T$ as an FD predicate in terms of three indexicals. Each indexical is a rule responsible for removing values detected as incompatible from one particular constraint argument. Indexicals are not Prolog goals; thus, the example does not express a conjunction. However, an indexical may make the store contradictory, in which case backtracking is triggered:

```
plus(X,Y,T) +:
    X in min(T) - max(Y) .. max(T) - min(Y),
    Y in min(T) - max(X) .. max(T) - min(X),
    T in min(X) + min(Y) .. max(X) + max(Y).
```

The above definition contains a single clause used for constraint solving. The first indexical wakes up whenever the bounds of $S(T)$ or $S(Y)$ are updated, and removes from $D(X,S)$ any values that are not compatible with the new bounds of $T$ and $Y$. Note that in the event of “holes” in the domains of $T$ or $Y$, $D(X,S)$ may contain some values that are incompatible with $X + Y = T$ but go undetected. Like most built-in arithmetic constraints, the above definition maintains bounds consistency, which is significantly cheaper to maintain than domain consistency, and suffices in most cases. The constraint could for example be used as follows:

```
| ?- X in 1..5, Y in 2..8, plus(X,Y,T).
    X in 1..5,
    Y in 2..8,
    T in 3..13
```

Thus, when an FD predicate is called, the ‘+;’ clause is activated.

The definition of a user constraint has to specify the variables involved and the finite domains with which their domains should be intersected when the propagator is run. Therefore the FD predicate with $n$ arguments consists of $n$ indexicals, each specifying a left hand side variable and a right hand side expression that evaluates to a finite domain, which is a function of the expression and of the constraint store. For example, the third indexical in the above FD predicate evaluates to the finite domain $3..13$ for $T$ if $D(X,S) = 1..5$ and $D(Y,S) = 2..8$. As the domain of some variables gets smaller, the indexical may further narrow the domain of other variables. Therefore such an indexical (called a propagating indexical) acts as a coroutine reacting to the changes in the store by enforcing further changes in the store.
In general there are three stages in the lifetime of a propagating indexical. When it is posted it may not be evaluated immediately (e.g. has to wait until some variables are ground before being able to modify the store). Until the preconditions for the evaluation are satisfied, the coroutine is blocked. When the indexical becomes unblocked, it computes a finite domain for intersecting with the domain of its left hand side. The coroutine then waits until some change occurs in a domain of a variable occurring in its right hand side. Eventually, the computation reaches a point when the indexical is entailed by the store, i.e. no changes in its right hand side can prune its left hand side any longer, and the coroutine can cease to exist.

Note that FD predicates must be correct and checking (see Section 10.10.3 [CLPFD Interface], page 428).

There can be several alternative definitions for the same user constraint with different strengths in propagation. For example, the definition of plusd below encodes the same \(X+Y=T\) constraint as the plus predicate above, but maintaining domain consistency:

\[
\text{plusd}(X,Y,T) +: \\
\quad X \text{ in } \text{dom}(T) - \text{dom}(Y), \\
\quad Y \text{ in } \text{dom}(T) - \text{dom}(X), \\
\quad T \text{ in } \text{dom}(X) + \text{dom}(Y).
\]

\[
| \text{?- } X \text{ in } \{1\} \setminus \{3\}, \ Y \text{ in } \{10\} \setminus \{20\}, \ \text{plusd}(X, Y, T). \\
X \text{ in } \{1\} \setminus \{3\}, \\
Y \text{ in } \{10\} \setminus \{20\}, \\
T \text{ in } \{11\} \setminus \{13\} \setminus \{21\} \setminus \{23\}
\]

This costs more in terms of execution time, but gives more precise results. For singleton domains plus and plusd behave in the same way.

In our design, general indexicals can only appear in the context of FD predicate definitions. The rationale for this restriction is the need for general indexicals to be able to suspend and resume, and this ability is only provided by the FD predicate mechanism.

If the program merely posts a constraint, then it suffices for the definition to contain a single clause for solving the constraint. If a constraint is reified or occurs in a propositional formula, then the definition must contain four clauses for solving and checking entailment of the constraint and its negation. The role of each clause is reflected in the “neck” operator. The following table summarizes the different forms of indexical clauses corresponding to a constraint \(C\). In all cases, Head should be a compound term with all arguments being distinct variables:

**Head +:** Indexicals.

The body consists of propagating indexicals for solving \(C\). The body can in fact be of a more general form—see Section 10.10.10.10 [Compiled Indexicals], page 493.

**Head -:** Indexicals.

The body consists of propagating indexicals for solving the negation of \(C\).
Head +? Indexical.
The body consists of a single checking indexical for testing entailment of \( C \).

Head -? Indexical.
The body consists of a single checking indexical for testing entailment of the negation of \( C \).

When a constraint is reified as in Constraint \( #\leftrightarrow B \), the solver spawns two coroutines corresponding to detecting entailment and disentailment. Eventually, one of them will succeed in this and consequently will bind \( B \) to 0 or 1. A third coroutine is spawned, waiting for \( B \) to become assigned, at which time the constraint (or its negation) is posted. In the mean time, the constraint may have been detected as (dis)entailed, in which case the third coroutine is dismissed.

As an example of a constraint with all methods defined, consider the following library constraint defining a disequation between two domain variables:

\[
\text{'x\not= y'(X,Y) +:}
\quad X \text{ in } \emptyset\{Y\},
\quad Y \text{ in } \emptyset\{X\}.
\]

\[
\text{'x\not= y'(X,Y) -:}
\quad X \text{ in } \text{dom}(Y),
\quad Y \text{ in } \text{dom}(X).
\]

\[
\text{'x\not= y'(X,Y) +?}
\quad X \text{ in } \text{\textbackslash dom}(Y).
\]

\[
\text{'x\not= y'(X,Y) -?}
\quad X \text{ in } \{Y\}.
\]

The following sections provide more precise coding rules and operational details for indexicals. \( X \text{ in } R \) denotes an indexical corresponding to a constraint \( C \). \( S \) denotes the current store.

10.10.10.8 Execution of Propagating Indexicals

Consider the definition of a constraint \( C \) containing a propagating indexical \( X \text{ in } R \). Let \( TV(X,C,S) \) denote the set of values for \( X \) that can make \( C \) true in some ground extension of the store \( S \). Then the indexical should obey the following coding rules:

- all arguments of \( C \) except \( X \) should occur in \( R \)
- if \( R \) is ground in \( S \), \( S(R) = TV(X,C,S) \)

If the coding rules are observed, then \( S(R) \) can be proven to contain \( TV(X,C,S) \) for all stores in which \( R \) is monotone. Hence it is natural for the implementation to wait until \( R \) becomes monotone before admitting the propagating indexical for execution. The execution of \( X \text{ in } R \) thus involves the following:

- If \( D(X,S) \) is disjoint from \( S(R) \), then a contradiction is detected.
- If \( D(X,S) \) is contained in \( S(R) \), then \( D(X,S) \) does not contain any values known to be
incompatible with \( C \), and the indexical suspends, unless \( R \) is ground in \( S \), in which case \( C \) is detected as entailed.

- Otherwise, \( D(X,S) \) contains some values that are known to be incompatible with \( C \). Hence, \( S \) is extended to \((X in S(R))(S)\) \( (X \text{ is pruned}) \), and the indexical suspends, unless \( R \) is ground in \( S \), in which case \( C \) is detected as entailed.

A propagating indexical is scheduled for execution as follows:

- it is evaluated initially as soon as it has become monotone
- it is re-evaluated when one of the following conditions occurs:
  1. the domain of a variable \( Y \) that occurs as \( \text{dom}(Y) \) or \( \text{card}(Y) \) in \( R \) has been updated
  2. the lower bound of a variable \( Y \) that occurs as \( \text{min}(Y) \) in \( R \) has been updated
  3. the upper bound of a variable \( Y \) that occurs as \( \text{max}(Y) \) in \( R \) has been updated

### 10.10.10.9 Execution of Checking Indexicals

Consider the definition of a constraint \( C \) containing a checking indexical \( X \text{ in } R \). Let \( FV(X,C,S) \) denote the set of values for \( X \) that can make \( C \) false in some ground extension of the store \( S \). Then the indexical should obey the following coding rules:

- all arguments of \( C \) except \( X \) should occur in \( R \)
- if \( R \) is ground in \( S \), \( S(R) = TV(X,C,S) \)

If the coding rules are observed, then \( S(R) \) can be proven to exclude \( FV(X,C,S) \) for all stores in which \( R \) is anti-monotone. Hence it is natural for the implementation to wait until \( R \) becomes anti-monotone before admitting the checking indexical for execution. The execution of \( X \text{ in } R \) thus involves the following:

- If \( D(X,S) \) is contained in \( S(R) \), then none of the possible values for \( X \) can make \( C \) false, and so \( C \) is detected as entailed.
- Otherwise, if \( D(X,S) \) is disjoint from \( S(R) \) and \( R \) is ground in \( S \), then all possible values for \( X \) will make \( C \) false, and so \( C \) is detected as disentailed.
- Otherwise, \( D(X,S) \) contains some values that could make \( C \) true and some that could make \( C \) false, and the indexical suspends.

A checking indexical is scheduled for execution as follows:

- it is evaluated initially as soon as it has become anti-monotone
- it is re-evaluated when one of the following conditions occurs:
  1. the domain of \( X \) has been pruned, or \( X \) has been assigned
  2. the domain of a variable \( Y \) that occurs as \( \text{dom}(Y) \) or \( \text{card}(Y) \) in \( R \) has been pruned
  3. the lower bound of a variable \( Y \) that occurs as \( \text{min}(Y) \) in \( R \) has been increased
  4. the upper bound of a variable \( Y \) that occurs as \( \text{max}(Y) \) in \( R \) has been decreased
10.10.10 Compiled Indexicals

The arithmetic, membership, and propositional constraints described earlier are transformed at compile time to conjunctions of library constraints. Although linear in the size of the source code, the expansion of a constraint to library goals can have time and memory overheads. Temporary variables holding intermediate values may have to be introduced, and the grain size of the constraint solver invocations can be rather small. Therefore, an automatic translation by compilation to indexicals is also provided for a selected set of constraints. The syntax for this construction is:

$$\text{Head} \ + \ : \ \text{ConstraintBody}$$

since release 4.1.3

Head should be a compound term with all arguments being distinct variables. ConstraintBody should be a constraint amenable to compilation to indexicals, and should not contain any variable not mentioned in Head. This clause defines the constraint Head to hold iff ConstraintBody is true.

Roughly, a constraint amenable to such compilation is of one of the following forms, or is a propositional combination of such forms. See Section 10.10.13.1 [Syntax of Indexicals], page 496, for the exact definition:

- var in ConstantRange
- \text{element}(\text{var}, \text{CList}, \text{var})
- \text{table}(\text{VList}, \text{CTable})
- \text{LinExpr \ RelOp \ LinExpr}
- \text{var \ \{ \ X \ stands \ for \ X#=1 \ \}}

10.10.11 Coexisting with Attributes and Blocked Goals

Domain variables may have attributes from other modules, as well as blocked goals, attached to them. However, the CLPFD propagation phase runs to completion before invoking handlers for such attributes and resuming such blocked goals. This could mean in particular that upon completion of the propagation phase, attribute handlers and blocked goals for multiple variables are ready to execute. For details, see the verify_attributes/3 hook at Section 10.3 [lib-atts], page 391.

Please note: fd_purge(X) (see Section 10.10.3.2 [Forgetting Constraints], page 431) does not affect attributes of other modules, or blocked goals.

Please note: the garbage collector will preserve all variables with attached attributes or blocked goals.

10.10.12 Example Programs

This section contains a few example programs. The first two programs are included in a benchmark suite that comes with the distribution. The benchmark suite is run by typing:

```
| ?- compile(library('clpfd/examples/bench')).
| ?- make.
```
10.10.12.1 Send More Money

Let us return briefly to the Send More Money problem (see Section 10.10.3.3 [Constraint Satisfaction Problems], page 431). Its \texttt{sum/8} predicate will expand to a \texttt{scalar_product/4} constraint. An indexical version is defined simply by changing the neck symbol of \texttt{sum/8} from ‘:-’ to ‘+:’, thus turning it into an FD predicate:

\[
\text{sum}(S, E, N, D, M, O, R, Y) +:
\begin{align*}
& 1000\times S + 100\times E + 10\times N + D \\
& + 1000\times M + 100\times O + 10\times R + E \\
& #\times 10000\times M + 1000\times O + 100\times N + 10\times E + Y.
\end{align*}
\]

10.10.12.2 N Queens

The problem is to place \(N\) queens on an \(N\times N\) chess board so that no queen is threatened by another queen.

The variables of this problem are the \(N\) queens. Each queen has a designated row. The problem is to select a column for it.

The main constraint of this problem is that no queen threaten another. This is encoded by the \texttt{no_threat/3} constraint and holds between all pairs \((X,Y)\) of queens. It could be defined as:

\[
\text{no_threat}(X, Y, I) :-
\begin{align*}
& X \neq Y, \\
& X+I \neq Y, \\
& X-I \neq Y.
\end{align*}
\]

However, this formulation introduces new temporary domain variables and creates twelve fine-grained indexicals. Worse, the disequalities only maintain bounds consistency, and so may miss some opportunities for pruning elements in the middle of domains.

A better idea is to formulate \texttt{no_threat/3} as an FD predicate with two indexicals, as shown in the program below. This constraint will not fire until one of the queens has been assigned (the corresponding indexical does not become monotone until then). Hence, the constraint is still not as strong as it could be.

For example, if the domain of one queen is 2..3, then it will threaten any queen placed in column 2 or 3 on an adjacent row, no matter which of the two open positions is chosen for the first queen. The commented out formulation of the constraint captures this reasoning, and illustrates the use of the \texttt{unionof/3} operator. This stronger version of the constraint indeed gives less backtracking, but is computationally more expensive and does not pay off in terms of execution time, except possibly for very large chess boards.

It is clear that \texttt{no_threat/3} cannot detect any incompatible values for a queen with domain of size greater than three. This observation is exploited in the third version of the constraint.

The first-fail principle is appropriate in the enumeration part of this problem.
Chapter 10: The Prolog Library

:- use_module(library(clpfd)).

queens(N, L, LabelingType) :-
    length(L, N),
    domain(L, 1, N),
    constrain_all(L),
    labeling(LabelingType, L).

constrain_all([]).
constrain_all([X|Xs]) :-
    constrain_between(X, Xs, 1),
    constrain_all(Xs).

constrain_between(_X, [], _N).
constrain_between(X, [Y|Ys], N) :-
    no_threat(X, Y, N),
    N1 is N+1,
    constrain_between(X, Ys, N1).

% version 1: weak but efficient
no_threat(X, Y, I) +:
    X in \{{Y} \lor {Y+I} \lor {Y-I}},
    Y in \{{X} \lor {X+I} \lor {X-I}}.

/*
% version 2: strong but very inefficient version
no_threat(X, Y, I) +:
    X in unionof(B,dom(Y),\{{B} \lor {B+I} \lor {B-I}}),
    Y in unionof(B,dom(X),\{{B} \lor {B+I} \lor {B-I}}).

% version 3: strong but somewhat inefficient version
no_threat(X, Y, I) +:
    X in (4..card(Y)) ? (inf..sup) \/
        unionof(B,dom(Y),\{{B} \lor {B+I} \lor {B-I}}),
    Y in (4..card(X)) ? (inf..sup) \/
        unionof(B,dom(X),\{{B} \lor {B+I} \lor {B-I}}).
*/

?- queens(8, L, [ff]).
L = [1,5,8,6,3,7,2,4]

10.10.12.3 Cumulative Scheduling
This example is a very small scheduling problem. We consider seven tasks where each task
has a fixed duration and a fixed amount of used resource:
The goal is to find a schedule that minimizes the completion time for the schedule while not exceeding the capacity 13 of the resource. The resource constraint is succinctly captured by a cumulative/2 constraint. Branch-and-bound search is used to find the minimal completion time.

This example was adapted from [Beldiceanu & Contejean 94].

```prolog
:- use_module(library(clpfd)).

schedule(Ss, End) :-
    Ss = [S1,S2,S3,S4,S5,S6,S7],
    Es = [E1,E2,E3,E4,E5,E6,E7],
    Tasks = [task(S1,16,E1, 2,0),
             task(S2, 6,E2, 9,0),
             task(S3,13,E3, 3,0),
             task(S4, 7,E4, 7,0),
             task(S5, 5,E5,10,0),
             task(S6,18,E6, 1,0),
             task(S7, 4,E7,11,0)],
    domain(Ss, 1, 30),
    domain(Es, 1, 50),
    domain([End], 1, 50),
    maximum(End, Es),
    cumulative(Tasks, [limit(13)]),
    append(Ss, [End], Vars),
    labeling([minimize(End)], Vars). % label End last

%%% End of file

| ?- schedule(Ss, End).
Ss = [1,17,10,10,5,5,1],
End = 23
```

10.10.13 Syntax Summary

10.10.13.1 Syntax of Indexicals

<table>
<thead>
<tr>
<th>Constant</th>
<th>::= integer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\(\text{FixTerm} \quad ::= \quad \text{Constant} \\
\quad \quad | \quad \text{var} \quad \quad \{ \text{suspend until assigned} \} \\
\text{Term} \quad ::= \quad \text{min}(\text{var}) \quad \quad \{ \text{min. of domain of X} \} \\
\quad \quad | \quad \text{max}(\text{var}) \quad \quad \{ \text{max. of domain of X} \} \\
\quad \quad | \quad \text{card}(\text{var}) \quad \quad \{ \text{size of domain of X} \} \\
\quad \quad | \quad \text{FixTerm} \\
\quad \quad | \quad -\text{FixTerm} \\
\quad \quad | \quad \text{Term} + \text{FixTerm} \\
\quad \quad | \quad \text{Term} - \text{FixTerm} \\
\quad \quad | \quad \text{Term} \ast \text{FixTerm} \\
\quad \quad | \quad \text{Term} \div\text{FixTerm} \quad \quad \{ \text{ceilinged division} \} \\
\quad \quad | \quad \text{Term} \%\text{FixTerm} \quad \quad \{ \text{floored division} \} \\
\quad \quad | \quad \text{Term} \mod \text{FixTerm} \quad \quad \{ \text{floored remainder} \} \\
\quad \quad | \quad \text{Term} \rem \text{FixTerm} \quad \quad \{ \text{truncated remainder} \} \\
\text{TermSet} \quad ::= \quad \{\text{FixTerm}, \ldots, \text{FixTerm}\} \\
\text{Range} \quad ::= \quad \text{TermSet} \\
\quad \quad | \quad \text{dom}(\text{var}) \quad \quad \{ \text{domain of X} \} \\
\quad \quad | \quad \text{Term} .. \text{Term} \quad \quad \{ \text{interval} \} \\
\quad \quad | \quad \text{Range} \\text{\&\&} \text{Range} \quad \quad \{ \text{intersection} \} \\
\quad \quad | \quad \text{Range} \text{\|} \text{Range} \quad \quad \{ \text{union} \} \\
\quad \quad | \quad \text{\textbackslash} \text{Range} \quad \quad \{ \text{complement} \} \\
\quad \quad | \quad -\text{Range} \quad \quad \{ \text{pointwise negation} \} \\
\quad \quad | \quad \text{Range} + \text{Range} \quad \quad \{ \text{pointwise addition} \} \\
\quad \quad | \quad \text{Range} - \text{Range} \quad \quad \{ \text{pointwise subtraction} \} \\
\quad \quad | \quad \text{Range} \mod \text{Range} \quad \quad \{ \text{pointwise modulo} \} \\
\quad \quad | \quad \text{Range} \rem \text{Range} \quad \quad \{ \text{pointwise remainder} \} \\
\quad \quad | \quad \text{Range} + \text{FixTerm} \quad \quad \{ \text{pointwise addition} \} \\
\quad \quad | \quad \text{Range} - \text{FixTerm} \quad \quad \{ \text{pointwise subtraction} \} \\
\quad \quad | \quad \text{FixTerm} - \text{Range} \quad \quad \{ \text{pointwise subtraction} \} \\
\quad \quad | \quad \text{Range} \mod \text{FixTerm} \quad \quad \{ \text{pointwise floored remainder} \} \\
\quad \quad | \quad \text{Range} \rem \text{FixTerm} \quad \quad \{ \text{pointwise truncated remainder} \} \\
\quad \quad | \quad \text{Range} \text{?} \text{Range} \\
\quad \quad | \quad \text{unionof}(\text{var}, \text{Range}, \text{Range}) \\
\quad \quad | \quad \text{switch}(\text{FixTerm}, \text{MapList}) \\
\text{ConstantSet} \quad ::= \quad \{\text{integer}, \ldots, \text{integer}\} \\
\text{ConstantRange} \quad ::= \quad \text{ConstantSet} \\
\quad \quad | \quad \text{Constant} .. \text{Constant} \\
\quad \quad | \quad \text{ConstantRange} \text{\&\&} \text{ConstantRange} \\
\quad \quad | \quad \text{ConstantRange} \text{\|} \text{ConstantRange} \\
\quad \quad | \quad \text{\textbackslash} \text{ConstantRange} \\
\text{MapList} \quad ::= \quad [] \\
\quad \quad | \quad [\text{integer}-\text{ConstantRange}|\text{MapList}] \\
\text{CTable} \quad ::= \quad []
| [CRow|CTable]

CRow ::= []
| [integer|CRow]
| [ConstantRange|CRow]

CList ::= []
| [integer|CList]

VList ::= []
| [var|VList]

Indexical ::= var in Range

Indexicals ::= Indexical
| Indexical , Indexicals

ConstraintBody ::= var { X stands for X#=1 }
| true
| false
| 1
| 0
| var in ConstantRange
| element(var,CList,var)
| table([VList],CTable)
| LinExpr RelOp LinExpr
| #\ ConstraintBody
| ConstraintBody #/\ ConstraintBody
| ConstraintBody #/ ConstraintBody
| ConstraintBody #=>
| ConstraintBody #\ ConstraintBody
| ConstraintBody #<=>

IxConstraintBody ::= Indexicals
| ConstraintBody

Head ::= term
{ a compound term with unique variable args }

TellPos ::= Head +: IxConstraintBody
TellNeg ::= Head -: Indexicals
AskPos ::= Head +? Indexical
AskNeg ::= Head -? Indexical

ConstraintDef ::= TellPos
| TellNeg
| AskPos
| AskNeg

10.10.13.2 Syntax of Arithmetic Expressions

Please note: that the Prolog arithmetic operators / and // do not mean the same thing.
\[ N ::= \text{integer} \]
\[ \text{LinExpr} ::= N \]
\[ \text{LinExpr} \]
\[ \text{var} \]
\[ N \times \text{var} \]
\[ N \times N \]
\[ - \text{LinExpr} \]
\[ \text{LinExpr} + \text{LinExpr} \]
\[ \text{LinExpr} - \text{LinExpr} \]
\[ \text{ConstraintBody} \{ \text{if true then 1 else 0} \} \]
\[ \text{Expr} ::= \text{LinExpr} \]
\[ - \text{Expr} \]
\[ \text{Expr} + \text{Expr} \]
\[ \text{Expr} - \text{Expr} \]
\[ \text{Expr} \times \text{Expr} \]
\[ \text{Expr} / \text{Expr} \{ \text{truncated division} \} \]
\[ \text{Expr} \div \text{Expr} \{ \text{truncated division}\} \quad \text{since release 4.3} \]
\[ \text{Expr} \text{rem} \text{Expr} \{ \text{truncated remainder} \} \]
\[ \text{Expr} \text{mod} \text{Expr} \{ \text{floored remainder} \} \]
\[ \text{Expr} \text{min} (\text{Expr}, \text{Expr}) \]
\[ \text{Expr} \text{max} (\text{Expr}, \text{Expr}) \]
\[ \text{Expr} \text{abs} (\text{Expr}) \]
\[ \text{RelOp} ::= \#= \mid \#	ext{\textbackslash=} \mid \#< \mid \#<= \mid \#> \mid \#>= \]

**10.10.13.3 Operator Declarations**

\[ :- \text{op}(1200, \text{xfx}, [+:,-:,+?,-?]). \]
\[ :- \text{op}(760, \text{yfx}, \#=&). \]
\[ :- \text{op}(750, \text{xfy}, \#=&). \]
\[ :- \text{op}(750, \text{yfx}, \#<&). \]
\[ :- \text{op}(740, \text{yfx}, \#\text{\textbackslash}). \]
\[ :- \text{op}(730, \text{yfx}, \#\text{\textbackslash}). \]
\[ :- \text{op}(720, \text{yfx}, \#/\text{\textbackslash}). \]
\[ :- \text{op}(710, \text{fy}, \#\text{\textbackslash}). \]
\[ :- \text{op}(700, \text{xfx}, \text{in}, \text{in_set}). \]
\[ :- \text{op}(700, \text{xfx}, [#=,\#\text{\textbackslash}=,\#<,\#<=,\#>,\#>=]). \]
\[ :- \text{op}(550, \text{xfy}, \ldots). \]
\[ :- \text{op}(500, \text{fy}, \text{\textbackslash}). \]
\[ :- \text{op}(490, \text{yfx}, ?). \]
\[ :- \text{op}(400, \text{yfx}, [>/,<]). \]

**10.11 Constraint Logic Programming over Rationals or Reals—library([clpq,clpr])**
10.11.1 Introduction

The clp(Q,R) system described in this chapter is an instance of the general Constraint Logic Programming scheme introduced by [Jaffar & Michaylov 87]. It is a third-party product, bundled with SICStus Prolog as two library packages. It is not supported by SICS in any way.

The implementation is at least as complete as other existing clp(R) implementations: It solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

10.11.1.1 Referencing this Software

When referring to this implementation of clp(Q,R) in publications, you should use the following reference:


10.11.1.2 Acknowledgments

The development of this software was supported by the Austrian *Fonds zur Foerderung der Wissenschaftlichen Forschung* under grant P9426-PHY. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science and Research.

We include a collection of examples that has been distributed with the Monash University version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

10.11.2 Solver Interface

Until rational numbers become first class citizens in SICStus Prolog, rational arithmetics has to be emulated. Because of the emulation it is too expensive to support arithmetics with automatic coercion between all sorts of numbers, like you find it in CommonLisp, for example.

You must choose whether you want to operate in the field of Q (Rationals) or R (Reals):

| ?- use_module(library(clpq)). |

or

| ?- use_module(library(clpr)). |

You can also load both modules, but the exported predicates listed below will name clash (see Section 4.11.12 [ref-mod-ncl], page 168). You can avoid the interactive resolution dialog if the importation is skipped, e.g. via: `use_module(library(clpq),[]),use_module(library(clpr),[])`. 

10.11.2.1 Notational Conventions

Throughout this chapter, the prompts \texttt{clp(q)?-} and \texttt{clp(r)?-} are used to differentiate between \texttt{clp(Q)} and \texttt{clp(R)} in exemplary interactions.

In general there are many ways to express the same linear relationship. This degree of freedom is manifest in the fact that the printed manual and an actual interaction with the current version of \texttt{clp(Q,R)} may show syntactically different answer constraints, despite the fact the same semantic relationship is being expressed. There are means to control the presentation; see Section 10.11.5.1 [CLPQR Variable Ordering], page 514. The approximative nature of floating point numbers may also produce numerical differences between the text in this manual and the actual results of \texttt{clp(R)}, for a given edition of the software.

10.11.2.2 Solver Predicates

The solver interface for both Q and R consists of the following predicates, which are exported from module(linear).

\begin{itemize}
  \item \texttt{Constraint} is a term accepted by the grammar below. The corresponding constraint is added to the current constraint store and checked for satisfiability. Use the module prefix to distinguish the solvers if both \texttt{clp(Q)} and \texttt{clp(R)} were loaded.
  \item \texttt{Aq = 5, Ar = 5.0, Bq = 5, Br = 5.0}
\end{itemize}

Although \texttt{clp(Q)} and \texttt{clp(R)} are independent modules, you are asking for trouble if you (accidentally) share variables between them:

\begin{verbatim}
| ?- clpr:{A+B=10}, clpq:{A=B}.
  ! Type error in argument 2 of clpq:=/2
  ! a rational number expected, but 5.0 found
  ! goal: _118=5.0
\end{verbatim}

This is because both solvers eventually compute values for the variables and Reals are incompatible with Rationals.

Here is the constraint grammar:

\begin{verbatim}
Constraint ::= C
  | C \& C \{ conjunction \}
C ::= Expr \== Expr \{ equation \}
  | Expr \= Expr \{ equation \}
  | Expr < Expr \{ strict inequation \}
  | Expr > Expr \{ strict inequation \}
  | Expr <= Expr \{ nonstrict inequation \}
  | Expr >= Expr \{ nonstrict inequation \}
  | Expr \== Expr \{ disequation \}
\end{verbatim}
Expr ::= variable \{ Prolog variable \}
| number \{ floating point or integer \}
| + Expr \{ unary plus \}
| - Expr \{ unary minus \}
| Expr + Expr \{ addition \}
| Expr - Expr \{ subtraction \}
| Expr * Expr \{ multiplication \}
| Expr / Expr \{ division \}
| abs(Expr) \{ absolute value \}
| sin(Expr) \{ trigonometric sine \}
| cos(Expr) \{ trigonometric cosine \}
| tan(Expr) \{ trigonometric tangent \}
| pow(Expr,Expr) \{ raise to the power \}
| exp(Expr,Expr) \{ raise to the power \}
| min(Expr,Expr) \{ minimum of the two arguments \}
| max(Expr,Expr) \{ maximum of the two arguments \}
| #(Const) \{ symbolic numerical constants \}

 Conjunctive constraints \{ C, C \} have been made part of the syntax to control the granularity of constraint submission, which will be exploited by future versions of this software. Symbolic numerical constants are provided for compatibility only; see Section 10.11.7 [CLPQR Monash Examples], page 520.

entailed(+Constraint)
Succeeds iff the linear Constraint is entailed by the current constraint store. This predicate does not change the state of the constraint store.

clp(q) ?- \{ A =< 4 \}, entailed(A=\textbar=5).

\{A=\textbar=4\}

clp(q) ?- \{ A =< 4 \}, entailed(A=\textbar=3).

no

inf(+Expr, -Inf)
inf(+Expr, -Inf, +Vector, -Vertex)
Computes the infimum of the linear expression Expr and unifies it with Inf. If given, then Vector should be a list of variables relevant to Expr, and Vertex will be unified a list of the same length as Vector containing the values for Vector, such that the infimum is produced when assigned. Failure indicates unboundedness.

sup(+Expr, -Sup)
sup(+Expr, -Sup, +Vector, -Vertex)
Computes the supremum of the linear expression Expr and unifies it with Sup. If given, then Vector should be a list of variables relevant to Expr, and Vertex will be unified a list of the same length as Vector containing the values for
Vector, such that the supremum is produced when assigned. Failure indicates unboundedness.

```prolog
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11, 
           X+3*Y =< 15, Z = 30*X+50*Y 
       }, sup(Z, Sup, [X,Y], Vertex).
Sup = 310, 
Vertex = [7,2],
{Z=30*X+50*Y},
{X+1/2*Y=<8},
{X+3*Y=<15},
{X+2*Y=<11}
```

**minimize(+Expr)**
Computes the infimum of the linear expression Expr and equates it with the expression, i.e. as if defined as:

```prolog
minimize(Expr) :- inf(Expr, Expr).
```

**maximize(+Expr)**
Computes the supremum of the linear expression Expr and equates it with the expression.

```prolog
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11, 
           X+3*Y =< 15, Z = 30*X+50*Y 
       }, maximize(Z).
X = 7, 
Y = 2, 
Z = 310
```

**bb_inf(+Ints, +Expr, -Inf)**
Computes the infimum of the linear expression Expr under the additional constraint that all of variables in the list Ints assume integral values at the infimum. This allows for the solution of mixed integer linear optimization problems; see Section 10.11.8 [CLPQR MIP], page 521.

```prolog
clp(q) ?- {X =/> Y+Z, Y > 1, Z > 1}, bb_inf([Y,Z],X,Inf).
Inf = 4, 
{Y>1},
{Z>1},
{X-Y-Z>=0}
```

**bb_inf(+Ints, +Expr, -Inf, -Vertex, +Eps)**
Computes the infimum of the linear expression Expr under the additional constraint that all of variables in the list Ints assume integral values at the infimum. Eps is a positive number between 0 and 0.5 that specifies how close a number X must be to the next integer to be considered integral: \( \text{abs}(\text{round}(X) - X) < Eps \).
The predicate bb_inf/3 uses \( Eps = 0.001 \). With clp(Q), \( Eps = 0 \) makes sense. Vertex is a list of the same length as Ints and contains the (integral) values
for \textit{Ints}, such that the infimum is produced when assigned. Note that this will only generate one particular solution, which is different from the situation with \texttt{minimize/1}, where the general solution is exhibited.

\texttt{bb\_inf/5} works properly for non-strict inequalities only! Disequations ($\neq$) and higher dimensional strict inequalities ($>$, $<$) are beyond its scope. Strict bounds on the decision variables are honored however:

\begin{verbatim}
clp(q) ?- \{X \geq Y+Z, Y > 1, Z > 1\}, bb\_inf([Y,Z],X,Inf,Vertex,0).
\end{verbatim}

\begin{verbatim}
Inf = 4,
Vertex = [2,2],
\{Y>1\},
\{Z>1\},
\{X-Y-Z>=0\}
\end{verbatim}

The limitation(s) can be addressed by:
- transforming the original problem statement so that only non-strict inequalities remain; for example, $\{X + Y > 0\}$ becomes $\{X + Y \geq 1\}$ for integral $X$ and $Y$;
- contemplating the use of \texttt{clp(FD)}.

\texttt{ordering(+Spec)}

Provides a means to control one aspect of the presentation of the answer constraints; see Section 10.11.5.1 \cite{CLPQR Variable Ordering}, page 514.

\texttt{dump(+Target, -NewVars, -CodedAnswer)}

Reflects the constraints on the target variables into a term, where \texttt{Target} and \texttt{NewVars} are lists of variables of equal length and \texttt{CodedAnswer} is the term representation of the projection of constraints onto the target variables where the target variables are replaced by the corresponding variables from \texttt{NewVars} (see Section 10.11.5.2 \cite{CLPQR Turning Answers into Terms}, page 515).

\begin{verbatim}
clp(q) ?- \{A+B =< 10, A=4\},
dump([[A,B]],Vs,Cs),
dump([B],Bp,Cb).
\end{verbatim}

\begin{verbatim}
Cb = \{_A=<6],
Bp = \{_A],
Cs = \{_B>=4,_C+_B=<10],
Vs = \{_C,_B],
\{A=4\},
\{A+B=<=10\}
\end{verbatim}

The current version of \texttt{dump/3} is incomplete with respect to nonlinear constraints. It only reports nonlinear constraints that are connected to the target variables. The following example has no solution. From the top level’s report we have a chance to deduce this fact, but \texttt{dump/3} currently has no means to collect global constraints . . .
q(X) :-
    \{X>=10\},
    \{\sin(Z)>3\}.

\text{clp}(r) \leftarrow q(X), \text{dump}([X], V, C).

C = \[_A>=10.0\],
V = \[_A\],
\text{clpr}\{3.0-\sin(_B)<0.0\},
\{X>=10.0\}

\text{projecting_assert/1}(\text{Clause})

If you use the database, then the clauses you assert might have constraints
associated with their variables. Use this predicate instead of \text{assert/1}
in order to ensure that only the relevant and projected constraints get stored in the
database. It will transform the clause into one with plain variables and extra
body goals that set up the relevant constraint when called.

10.11.2.3 Unification

Equality constraints are added to the store implicitly each time variables that have been
mentioned in explicit constraints are bound—either to another such variable or to a number.

\text{clp}(r) \leftarrow \{2*A+3*B=C/2\}, C=10.0, A=B.

A = 1.0,
B = 1.0,
C = 10.0

Is equivalent modulo rounding errors to

\text{clp}(r) \leftarrow \{2*A+3*B=C/2, C=10, A=B\}.

A = 1.0,
B = 0.9999999999999999,
C = 10.0

The shortcut bypassing the use of \{}/1 is allowed and makes sense because the interpretation
of this equality in Prolog and \text{clp}(R) coincides. In general, equations involving interpreted
functors, +/-2 in this case, must be fed to the solver explicitly:

\text{clp}(r) \leftarrow X=3.0+1.0, X=4.0.

\text{no}

Moreover, variables known by \text{clp}(R) may be bound directly to floats only. Likewise, vari-
ables known by \text{clp}(Q) may be bound directly to rational numbers only; see Section 10.11.9.1
[CLPQR Fragments and Bits], page 523. Failing to do so is rewarded with an exception:
This is because 10.0 is not a rational constant. To make clp(Q) happy you have to say:

```
clp(q) ?- {2*A+3*B=C/2}, C=rat(10,1), A=B.
```

A = 1,  
B = 1,  
C = 10

If you use `}/1`, then you do not have to worry about such details.

### 10.11.2.4 Feedback and Bindings

What was covered so far was how the user populates the constraint store. The other direction of the information flow consists of the success and failure of the above predicates and the binding of variables to numerical values. Example:

```
clp(r) ?- {A-B+C=10, C=5+5}.
```

{A = B},  
C = 10.0

The linear constraints imply C=10.0 and the solver consequently exports this binding to the Prolog world. The fact that A=B is deduced and represented by the solver but not exported as a binding. More about answer presentation in Section 10.11.5 [CLPQR Projection], page 513.

### 10.11.3 Linearity and Nonlinear Residues

The clp(Q,R) system is restricted to deal with linear constraints because the decision algorithms for general nonlinear constraints are prohibitively expensive to run. If you need this functionality badly, then you should look into symbolic algebra packages. Although the clp(Q,R) system cannot solve nonlinear constraints, it will collect them faithfully in the hope that through the addition of further (linear) constraints they might get simple enough to solve eventually. If an answer contains nonlinear constraints, then you have to be aware of the fact that success is qualified modulo the existence of a solution to the system of residual (nonlinear) constraints:

```
clp(r) ?- {sin(X) = cos(X)}.
```

```
clpr:{sin(X)-cos(X)=0.0}
```

There are indeed infinitely many solutions to this constraint (X = 0.785398 + n*Pi), but clp(Q,R) has no direct means to find and represent them.
The system goes through some lengths to recognize linear expressions as such. The method is based on a normal form for multivariate polynomials. In addition, some simple isolation axioms, that can be used in equality constraints, have been added. The current major limitation of the method is that full polynomial division has not been implemented. Examples:

This is an example where the isolation axioms are sufficient to determine the value of $X$.

\[
\text{clp(r) } \begin{array}{l}
\text{?- } \{\sin(\cos(X)) = 1/2}\}. \\
X = 1.0197267436954502
\end{array}
\]

If we change the equation into an inequation, then clp(Q,R) gives up:

\[
\text{clp(r) } \begin{array}{l}
\text{?- } \{\sin(\cos(X)) < 1/2\}. \\
\text{clpr:}\{\sin(\cos(X))-0.5<0.0\}
\end{array}
\]

The following is easy again:

\[
\text{clp(r) } \begin{array}{l}
\text{?- } \{\sin(X+2)/\sin(4+X) = Y\}. \\
Y = 1.0
\end{array}
\]

And so is this:

\[
\text{clp(r) } \begin{array}{l}
\text{?- } \{(X+Y)*(Y+X)/X = Y*Y/X+99\}. \\
\{Y=49.5-0.5*X\}
\end{array}
\]

An ancient symbol manipulation benchmark consists in rising the expression $X+Y+Z+1$ to the 15th power:

\[
\text{clp(q) } \begin{array}{l}
\text{?- } \{\exp(X+Y+Z+1,15)=0\}. \\
\text{clpq:}\{Z^{15}+Z^{14}+15+Z^{13}+105+Z^{12}+455+Z^{11}+1365+Z^{10}+3003+\ldots \}
\end{array}
\]

Computing its roots is another story.

### 10.11.3.1 How Nonlinear Residues Are Made to Disappear

Binding variables that appear in nonlinear residues will reduce the complexity of the nonlinear expressions and eventually results in linear expressions:

\[
\text{clp(q) } \begin{array}{l}
\text{?- } \{\exp(X+Y+1,2) = 3*X*X+Y*Y\}. \\
\text{clpq:}\{Y^2-X^2+Y*X+X*2+1=0\}
\end{array}
\]

Equating $X$ and $Y$ collapses the expression completely and even determines the values of the two variables:
\[
\text{clp(q) ?- } \{ \text{exp} (X+Y+1,2) = 3 \times X \times X \times Y \}, \ X=Y. \\
X = -1/4, \\
Y = -1/4
\]

### 10.11.3.2 Isolation Axioms

These axioms are used to rewrite equations such that the variable to be solved for is moved to the left-hand side and the result of the evaluation of the right-hand side can be assigned to the variable. This allows, for example, to use the exponentiation operator for the computation of roots and logarithms; see below.

- \( A = B \times C \)  
  Residuates unless \( B \) or \( C \) is ground or \( A \) and \( B \) or \( C \) are ground.
- \( A = B / C \)  
  Residuates unless \( C \) is ground or \( A \) and \( B \) are ground.
- \( X = \min(Y,Z) \)  
  Residuates unless \( Y \) and \( Z \) are ground.
- \( X = \max(Y,Z) \)  
  Residuates unless \( Y \) and \( Z \) are ground.
- \( X = \text{abs}(Y) \)  
  Residuates unless \( Y \) is ground.
- \( X = \text{pow}(Y,Z), X = \exp(Y,Z) \)  
  Residuates unless any pair of two of the three variables is ground. Example:

  \[
  \text{clp(r) ?- } \{ 12=\text{pow}(2,X) \}.
  \]
  \[
  X = 3.5849625007211565
  \]

  \[
  \text{clp(r) ?- } \{ 12=\text{pow}(X,3.585) \}.
  \]
  \[
  X = 1.9999854993443926
  \]

  \[
  \text{clp(r) ?- } \{ X=\text{pow}(2,3.585) \}.
  \]
  \[
  X = 12.000311914286545
  \]

- \( X = \text{sin}(Y) \)  
  Residuates unless \( X \) or \( Y \) is ground. Example:

  \[
  \text{clp(r) ?- } \{ 1/2 = \text{sin}(X) \}.
  \]
  \[
  X = 0.5235987755982989
  \]

- \( X = \text{cos}(Y) \)  
  Residuates unless \( X \) or \( Y \) is ground.

- \( X = \tan(Y) \)  
  Residuates unless \( X \) or \( Y \) is ground.
10.11.4 Numerical Precision and Rationals

The fact that you can switch between clp(R) and clp(Q) should solve most of your numerical problems regarding precision. Within clp(Q), floating point constants will be coerced into rational numbers automatically. Transcendental functions will be approximated with rationals. The precision of the approximation is limited by the floating point precision. These two provisions allow you to switch between clp(R) and clp(Q) without having to change your programs.

What is to be kept in mind however is the fact that it may take quite big rationals to accommodate the required precision. High levels of precision are for example required if your linear program is ill-conditioned, i.e. in a full rank system the determinant of the coefficient matrix is close to zero. Another situation that may call for elevated levels of precision is when a linear optimization problem requires exceedingly many pivot steps before the optimum is reached.

If your application approximates irrational numbers, then you may be out of space particularly soon. The following program implements $N$ steps of Newton’s approximation for the square root function at point 2.

```
root(N, R) :-
    root(N, 1, R).

root(0, S, R) :- !, S=R.
root(N, S, R) :-
    N1 is N-1,
    { S1 = S/2 + 1/S },
    root(N1, S1, R).
```

It is known that this approximation converges quadratically, which means that the number of correct digits in the decimal expansion roughly doubles with each iteration. Therefore the numerator and denominator of the rational approximation have to grow likewise:
clp(q) :- [library('clpqr/examples/root')].
clp(q) :- root(3,R),print_decimal(R,70).
1.4142156862 7450980392 1568627450 9803921568 6274509803 9215686274
5098039215
R = 577/408

clp(q) :- root(4,R),print_decimal(R,70).
1.4142135623 7468991062 6295578890 1349101165 5962211574 4044584905
0192000543
R = 665857/470832

clp(q) :- root(5,R),print_decimal(R,70).
1.4142135623 7309504880 1689623502 5302436149 8192577619 7428498289
4986231958
R = 886731088897/627013566048

clp(q) :- root(6,R),print_decimal(R,70).
1.4142135623 7309504880 1688724209 6980785696 7187537723 4001561013
1331132652
R = 157258404803291863353217/1111984844349868137938112

clp(q) :- root(7,R),print_decimal(R,70).
1.4142135623 7309504880 1688724209 6980785696 7187537694 8073176679
7379907324
R = 4946041176255201878775086487573351061418968498177 /
34973792557579411720208518352070562919437964212608

Iterating for 8 steps produces no further change in the first 70 decimal digits of \( \sqrt{2} \).
After 15 steps the approximating rational number has a numerator and a denominator with
12543 digits each, and the next step runs out of memory.

Another irrational number that is easily computed is \( e \). The following program implements
an alternating series for \( 1/e \), where the absolute value of last term is an upper bound on
the error.
The computation of the rational number $E$ that approximates $e$ up to at least 1000 digits in its decimal expansion requires the evaluation of 450 terms of the series, i.e. 450 calls of `inv_e_series/5`. 

```prolog
% library('clpqr/examples/root')

e(N, E) :-
{ Err =:= exp(10, -(N+2)), Half =:= 1/2 },
inv_e_series(Half, Half, 3, Err, Inv_E),
{ E =:= 1/Inv_E }.

inv_e_series(Term, S0, _, Err, Sum) :-
{ abs(Term) =< Err }, !,
S0 = Sum.
inv_e_series(Term, S0, N, Err, Sum) :-
N1 is N+1,
{ Term1 =:= -Term/N, S1 =:= Term1+S0 },
inv_e_series(Term1, S1, N1, Err, Sum).
```

The computation of the rational number $E$ that approximates $e$ up to at least 1000 digits in its decimal expansion requires the evaluation of 450 terms of the series, i.e. 450 calls of `inv_e_series/5`. 

```prolog
% library('clpqr/examples/root')

e(N, E) :-
{ Err =:= exp(10, -(N+2)), Half =:= 1/2 },
inv_e_series(Half, Half, 3, Err, Inv_E),
{ E =:= 1/Inv_E }.

inv_e_series(Term, S0, _, Err, Sum) :-
{ abs(Term) =< Err }, !,
S0 = Sum.
inv_e_series(Term, S0, N, Err, Sum) :-
N1 is N+1,
{ Term1 =:= -Term/N, S1 =:= Term1+S0 },
inv_e_series(Term1, S1, N1, Err, Sum).
```
The decimal expansion itself looks like this:

E = 7149056228932760213666809592072842334297044221392610955845565494
3708750229467761730471738895197792271346693089326102132000338192
01318741878339854202268880420167840319199699494193852403223700
5853832741544191628747052136402176949163825453655900589161585723
4023097141760500482999192928304537235539145644588174733401360176
99539737065372741332836147409027715611159913069917833820285608440
31049668999965192863763465641896902707669908288742841392304807
9484725489080048360397601997717860246956202054402476586051379
3538290451208322129898069978107971226873160872046731879753034549
3103492167474809196348846916421782850086985668680640425192038155
4902863985313494692116272928265440876581064873866786120098602898
879913009887737209736065934827751120659213470528793143805903554
7928682131082164366007016698761961066948371407368962539467994627
13748582491107957693859503460699474018604042511710158848000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
/  
2629990810403002651059591555503002285441272170673105334466808931
6863103901346024240326549035084528682487048064823380723787110941
680923518735631878097230279657025110292855200370855693314795678
197839076430949584066374774037984151830363616625889963910391440709
08873457973034709592078833168383469733939777836341195624313553
883564822353659840936818391050630360633749349381528275392050975
7271468992840907541350345459011192466892177866882264242860412188
0652112744642450404625763019639086944558929249788045597537323892
1643188991444945360726899532023542969572584363761073528841147012
263418045643940558077377849081469299651735955229262198396182
183930043528583109973872348193806830382584040536394640895148751
076625673874702989490963078526010172128570461618889741995949666
6303289703193938019763434974240815397920213059790971915067856758
67144299229125566251274533670906339651002168190007668069645309
36605909327986773674772648678738515702777431353845466199680991
733618734211521654777491166010820059
10.11.5 Projection and Redundancy Elimination

Once a derivation succeeds, the Prolog system presents the bindings for the variables in the query. In a CLP system, the set of answer constraints is presented in analogy. A complication in the CLP context are variables and associated constraints that were not mentioned in the query. A motivating example is the familiar mortgage relation:

```
% library('clpqr/examples/mg')

mg(P, T, I, B, MP):-
{ T = 1,
  B + MP = P * (1 + I)
}.
mg(P, T, I, B, MP):-
{ T > 1,
  P1 = P * (1 + I) - MP,
  T1 = T - 1
},
mg(P1, T1, I, B, MP).
```

A sample query yields:

```
clp(r) ?- [library('clpqr/examples/mg')].
clp(r) ?- mg(P,12,0.01,B,MP).

{B=1.1268250301319698*P-12.682503013196973*MP}
```
Without projection of the answer constraints onto the query variables we would observe the following interaction:

\[
\text{clp}(r) \ ? - \ mg(P,12,0.01,B,Mp).
\]

\[
\{B=12.682503013196973\cdot_A-11.682503013196971\cdot P\}, \\
\{Mp=-(_A)+1.01\cdot P\}, \\
\{A=2.01\cdot A-1.01\cdot P\}, \\
\{C=3.0301\cdot A-2.0301\cdot P\}, \\
\{D=4.060401000000001\cdot_A-3.0604009999999997\cdot P\}, \\
\{E=5.101005010000001\cdot_A-4.10100501\cdot P\}, \\
\{F=6.152015060100001\cdot_A-5.152015060099999\cdot P\}, \\
\{G=7.213535210701001\cdot_A-6.213535210700999\cdot P\}, \\
\{H=8.285670562808011\cdot_A-7.285670562808009\cdot P\}, \\
\{I=9.368527268436091\cdot_A-8.36852726843609\cdot P\}, \\
\{J=10.462212541120453\cdot_A-9.46221254112045\cdot P\}, \\
\{K=11.566834666531657\cdot_A-10.566834666531655\cdot P\}
\]

The variables \_A . . . \_K are not part of the query, they originate from the mortgage program proper. Although the latter answer is equivalent to the former in terms of linear algebra, most users would prefer the former.

### 10.11.5.1 Variable Ordering

In general, there are many ways to express the same linear relationship between variables. \text{clp}(Q,R) does not care to distinguish between them, but the user might. The predicate \text{ordering}(+Spec) gives you some control over the variable ordering. Suppose that instead of \( B \), you want \( Mp \) to be the defined variable:

\[
\text{clp}(r) \ ? - \ mg(P,12,0.01,B,Mp).
\]

\[
\{B=1.1268250301319698\cdot P-12.682503013196973\cdot Mp\}
\]

This is achieved with:

\[
\text{clp}(r) \ ? - \ mg(P,12,0.01,B,Mp), \ ordering([Mp]).
\]

\[
\{Mp=-0.0788487886783417\cdot B+0.08884878867834171\cdot P\}
\]

One could go one step further and require \( P \) to appear before (to the left of) \( B \) in an addition:

\[
\text{clp}(r) \ ? - \ mg(P,12,0.01,B,Mp), \ ordering([Mp,P]).
\]

\[
\{Mp=0.0888487886783417\cdot P-0.07884878867834171\cdot B\}
\]

\text{Spec} in \text{ordering}(+Spec) is either a list of variables with the intended ordering, or of the form \( A<B \). The latter form means that \( A \) goes to the left of \( B \). In fact, \text{ordering}([A,B,C,D]) is shorthand for:
ordering(A < B), ordering(A < C), ordering(A < D),
ordering(B < C), ordering(B < D),
ordering(C < D)

The ordering specification only affects the final presentation of the constraints. For all
other operations of clp(Q,R), the ordering is immaterial. Note that ordering/1 acts like
a constraint: you can put it anywhere in the computation, and you can submit multiple
specifications.

\[
\text{clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp).} \\
\{B= -12.682503013196973*Mp+1.1268250301319698*P} \\
\text{clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp), ordering(P < Mp).} \\
\{P=0.8874492252651537*B+11.255077473484631*Mp}\]

10.11.5.2 Turning Answers into Terms

In meta-programming applications one needs to get a grip on the results computed by the
clp(Q,R) solver. You can use the predicate dump/3 for that purpose:

\[
\text{clp(r) ?- \{2*A+B+C=10,C-D=E,A<10\}, dump([A,B,C,D,E],[a,b,c,d,e],Constraints).} \\
\text{Constraints = \{e<10.0,a=10.0-c-d-2.0*e,b=c+d\},} \\
\{C=10.0-2.0*A-B\}, \\
\{E=10.0-2.0*A-B-D\}, \\
\{A<10.0}\]

10.11.5.3 Projecting Inequalities

As soon as linear inequations are involved, projection gets more demanding complexity wise.
The current clp(Q,R) version uses a Fourier-Motzkin algorithm for the projection of linear
inequalities. The choice of a suitable algorithm is somewhat dependent on the number of
variables to be eliminated, the total number of variables, and other factors. It is quite easy
to produce problems of moderate size where the elimination step takes some time. For
example, when the dimension of the projection is 1, you might be better off computing the
supremum and the infimum of the remaining variable instead of eliminating \(n-1\) variables
via implicit projection.

In order to make answers as concise as possible, redundant constraints are removed by the
system as well. In the following set of inequalities, half of them are redundant.
% library('clpqr/examples/eliminat')

define example(2, [X0,X1,X2,X3,X4]) :-
{+87*X0 +52*X1 +27*X2 -54*X3 +56*X4 =< -93,
 +33*X0 -10*X1 +61*X2 -28*X3 -29*X4 =< 63,
 -68*X0 +8*X1 +35*X2 +68*X3 +35*X4 =< -85,
 +90*X0 +60*X1 -76*X2 -53*X3 +24*X4 =< -68,
 -95*X0 -10*X1 +64*X2 +76*X3 -24*X4 =< 33,
 +43*X0 -22*X1 +67*X2 -68*X3 -92*X4 =< -97,
 +39*X0 +7*X1 +62*X2 +54*X3 -26*X4 =< -27,
 +48*X0 -13*X1 +7*X2 -61*X3 -59*X4 =< -2,
 +49*X0 -23*X1 -31*X2 -76*X3 +27*X4 =< 3,
 -50*X0 +58*X1 -1*X2 +57*X3 +20*X4 =< 6,
 -13*X0 -63*X1 +81*X2 -3*X3 +70*X4 =< 64,
 +20*X0 +67*X1 -23*X2 -41*X3 -66*X4 =< 52,
 -81*X0 -44*X1 +19*X2 -22*X3 -73*X4 =< -17,
 -43*X0 -9*X1 +14*X2 +27*X3 +40*X4 =< 39,
 +16*X0 +83*X1 +89*X2 +25*X3 +55*X4 =< 36,
 +2*X0 +40*X1 +65*X2 +59*X3 -32*X4 =< 13,
 -65*X0 -11*X1 +10*X2 -13*X3 +91*X4 =< 49,
 +93*X0 -73*X1 +91*X2 -1*X3 +23*X4 =< -87
}.

Consequently, the answer consists of the system of nine non-redundant inequalities only:

clp(q) ?- [library('clpqr/examples/eliminat')].
clp(q) ?- example(2, [X0,X1,X2,X3,X4]).

\{X0-2/17*X1-35/68*X2-X3-35/68*X4>=5/4, \
 \{X0-73/93*X1+91/93*X2-1/93*X3+23/93*X4=<-29/31}, \
 \{X0-29/25*X1+1/50*X2-57/50*X3-2/5*X4>=-3/25}, \
 \{X0+7/39*X1-62/39*X2+18/13*X3-2/3*X4=<-9/13}, \
 \{X0+2/19*X1-64/95*X2+4/5*X3+24/95*X4<=-33/95}, \
 \{X0+2/3*X1-38/45*X2-53/90*X3+4/15*X4=<-34/45}, \
 \{X0-23/49*X1-31/49*X2-76/49*X3+27/49*X4=<3/49}, \
 \{X0+44/81*X1-19/81*X2+22/81*X3+73/81*X4>=17/81}, \
 \{X0+9/43*X1-14/43*X2-27/43*X3-40/43*X4>=-39/43\}

The projection (the shadow) of this polyhedral set into the $X0,X1$ space can be computed via the implicit elimination of non-query variables:

clp(q) ?- example(2, [X0,X1|_]).

\{X0+2619277/17854273*X1=-851123/17854273}, \
\{X0+6429953/16575801*X1=-12749681/16575801}, \
\{X0+19130/1213083*X1=795400/404361}, \
\{X0-1251619/3956679*X1=21101146/3956679}, \
\{X0+601502/4257189*X1=220850/473021\}
Projection is quite a powerful concept that leads to surprisingly terse executable specifications of nontrivial problems like the computation of the convex hull from a set of points in an n-dimensional space: Given the program

```prolog
% library('clpqr/examples/elimination')
conv_hull(Points, Xs) :-
  lin_comb(Points, Lambdas, Zero, Xs),
  zero(Zero),
  polytope(Lambdas).

polytope(Xs) :-
  positive_sum(Xs, 1).

positive_sum([], Z) :- {Z=0}.
positive_sum([X|Xs], SumX) :-
  { X >= 0, SumX = X+Sum },
  positive_sum(Xs, Sum).

zero([]).
zero([Z|Zs]) :- {Z=0}, zero(Zs).

lin_comb([], [], S1, S1).
lin_comb([Ps|Rest], [K|Ks], S1, S3) :-
  lin_comb_r(Ps, K, S1, S2),
  lin_comb(Rest, Ks, S2, S3).

lin_comb_r([], _, [], []).
lin_comb_r([P|Ps], K, [S|Ss], [Kps|Ss1]) :-
  { Kps = K*P+S },
  lin_comb_r(Ps, K, Ss, Ss1).
```

we can post the following query:

```prolog
clp(q) ?- conv_hull([ [1,1], [2,0], [3,0], [1,2], [2,2] ], [X,Y]).
```

{Y=<2},
{X+1/2*Y=<3},
{X>=1},
{Y>=0},
{X+Y>=2}

This answer is easily verified graphically:
The convex hull program directly corresponds to the mathematical definition of the convex hull. What does the trick in operational terms is the implicit elimination of the *Lambdas* from the program formulation. Please note that this program does not limit the number of points or the dimension of the space they are from. Please note further that quantifier elimination is a computationally expensive operation and therefore this program is only useful as a benchmark for the projector and not so for the intended purpose.

### 10.11.6 Why Disequations

A beautiful example of disequations at work is due to [Colmerauer 90]. It addresses the task of tiling a rectangle with squares of all-different, a priori unknown sizes. Here is a translation of the original Prolog-III program to clp(Q,R):
Chapter 10: The Prolog Library

% library('clpqr/examples/squares')

filled_rectangle(A, C) :-
    { A >= 1 },
    distinct_squares(C),
    filled_zone([-1,A,1], _, C, []).

distinct_squares([]).
distinct_squares([B|C]) :-
    { B > 0 },
    outof(C, B),
    distinct_squares(C).

outof([], _).
outof([B1|C], B) :-
    { B =\= B1 }, % *** note disequation ***
    outof(C, B).

filled_zone([V|L], [W|L], C0, C0) :-
    { V=W, V >= 0 }.
filled_zone([V|L], L3, [B|C], C2) :-
    { V < 0 },
    placed_square(B, L, L1),
    filled_zone(L1, L2, C, C1),
    { Vb=V+B },
    filled_zone([Vb,B|L2], L3, C1, C2).

placed_square(B, [H,H0,H1|L], L1) :-
    { B > H, H0=0, H2=H+H1 },
    placed_square(B, [H2|L], L1).
placed_square(B, [B,V|L], [X|L]) :-
    { X=V-B }.
placed_square(B, [H|L], [X,Y|L]) :-
    { B < H, X= -B, Y=H-B }.

There are no tilings with less than nine squares except the trivial one where the rectangle equals the only square. There are eight solutions for nine squares. Six further solutions are rotations of the first two.
clp(q) ?- [library('clpqr/examples/squares')].
clp(q) ?- filled_rectangle(A, Squares).

A = 1, 
Squares = [1] ? ;

A = 33/32,

A = 69/61,

Depending on your hardware, the above query may take a few minutes. Supplying the knowledge about the minimal number of squares beforehand cuts the computation time by a factor of roughly four:

clp(q) ?- length(Squares, 9), filled_rectangle(A, Squares).

A = 33/32,

A = 69/61,

10.11.7 Monash Examples

This collection of examples has been distributed with the Monash University Version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

Assuming you are using clp(R):

clp(r) ?- [library('clpqr/examples/monash/rkf45')].

clp(r) ?- go.
Point 0.00000 : 0.75000 0.00000
Point 0.50000 : 0.61969 0.47793
Point 1.00000 : 0.29417 0.81233
Point 1.50000 : -0.10556 0.95809
Point 2.00000 : -0.49076 0.93977
Point 2.50000 : -0.81440 0.79929
Point 3.00000 : -1.05440 0.57522

Iteration finished
------------------
439 derivative evaluations
10.11.8 A Mixed Integer Linear Optimization Example

The predicates `bb_inf/[3,5]` implement a simple Branch and Bound search algorithm for Mixed Integer Linear (MIP) Optimization examples. Serious MIP is not trivial. The implementation `library('clpqr/bb.pl')` is to be understood as a starting point for more ambitious users who need control over branching, or who want to add cutting planes, for example.

Anyway, here is a small problem from miplib, a collection of MIP models, housed at Rice University:

<table>
<thead>
<tr>
<th>NAME</th>
<th>flugpl</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWS</td>
<td>18</td>
</tr>
<tr>
<td>COLUMNS</td>
<td>18</td>
</tr>
<tr>
<td>INTEGER</td>
<td>11</td>
</tr>
<tr>
<td>NONZERO</td>
<td>46</td>
</tr>
<tr>
<td>BEST SOLN</td>
<td>1201500 (opt)</td>
</tr>
<tr>
<td>LP SOLN</td>
<td>1167185.73</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Harvey M. Wagner</td>
</tr>
<tr>
<td></td>
<td>John W. Gregory (Cray Research)</td>
</tr>
<tr>
<td></td>
<td>E. Andrew Boyd (Rice University)</td>
</tr>
<tr>
<td>APPLICATION</td>
<td>airline model</td>
</tr>
<tr>
<td>COMMENTS</td>
<td>no integer variables are binary</td>
</tr>
</tbody>
</table>
example(flugpl, Obj, Vs, Ints, []) :-
    Vs = [ Anm1, Anm2, Anm3, Anm4, Anm5, Anm6, Stm1, Stm2, Stm3, Stm4, Stm5, Stm6, UE1, UE2, UE3, UE4, UE5, UE6],
    Ints = [Stm6, Stm5, Stm4, Stm3, Stm2, Anm6, Anm5, Anm4, Anm3, Anm2, Anm1],
    Obj = 2700*Stm1 + 1500*Anm1 + 30*UE1
       + 2700*Stm2 + 1500*Anm2 + 30*UE2
       + 2700*Stm3 + 1500*Anm3 + 30*UE3
       + 2700*Stm4 + 1500*Anm4 + 30*UE4
       + 2700*Stm5 + 1500*Anm5 + 30*UE5
       + 2700*Stm6 + 1500*Anm6 + 30*UE6,
    allpos(Vs),
    { Stm1 = 60, 0.9*Stm1 +1*Anm1 -1*Stm2 = 0, 0.9*Stm2 +1*Anm2 -1*Stm3 = 0, 0.9*Stm3 +1*Anm3 -1*Stm4 = 0, 0.9*Stm4 +1*Anm4 -1*Stm5 = 0, 0.9*Stm5 +1*Anm5 -1*Stm6 = 0, 150*Stm1 -100*Anm1 +1*UE1 >= 8000, 150*Stm2 -100*Anm2 +1*UE2 >= 9000, 150*Stm3 -100*Anm3 +1*UE3 >= 8000, 150*Stm4 -100*Anm4 +1*UE4 >= 10000, 150*Stm5 -100*Anm5 +1*UE5 >= 9000, 150*Stm6 -100*Anm6 +1*UE6 >= 12000, -20*Stm1 +1*UE1 =< 0, -20*Stm2 +1*UE2 =< 0, -20*Stm3 +1*UE3 =< 0, -20*Stm4 +1*UE4 =< 0, -20*Stm5 +1*UE5 =< 0, -20*Stm6 +1*UE6 =< 0, Anm1 =< 18, 57 =< Stm2, Stm2 =< 75, Anm2 =< 18, 57 =< Stm3, Stm3 =< 75, Anm3 =< 18, 57 =< Stm4, Stm4 =< 75, Anm4 =< 18, 57 =< Stm5, Stm5 =< 75, Anm5 =< 18, 57 =< Stm6, Stm6 =< 75, Anm6 =< 18 }.

allpos([ ]).
allpos([X|Xs]) :- {X >= 0}, allpos(Xs).

We can first check whether the relaxed problem has indeed the quoted infimum:

    clp(r) ?- example(flugpl, Obj, _, _, _), inf(Obj, Inf).

    Inf = 1167185.7255923203

Computing the infimum under the additional constraints that Stm6, Stm5, Stm4, Stm3, Stm2, Anm6, Anm5, Anm4, Anm3, Anm2, Anm1 assume integer values at the infimum is computationally harder, but the query does not change much:
clp(r) ?- example(flugpl, Obj, _, Ints, _), 
    bb_inf(Ints, Obj, Inf, Vertex, 0.001).

Inf = 1201500.0000000005,
Vertex = [75.0,70.0,70.0,60.0,60.0,0.0,12.0,7.0,16.0,6.0,6.0]

10.11.9 Implementation Architecture
The system consists roughly of the following components:

- A polynomial normal form expression simplification mechanism.
- A solver for linear equations [Holzbaur 92a].
- A simplex algorithm to decide linear inequalities [Holzbaur 94].

10.11.9.1 Fragments and Bits

Rationals. The internal data structure for rational numbers is \texttt{rat(Num,Den)}. \texttt{Den} is always positive, i.e. the sign of the rational number is the sign of \texttt{Num}. Further, \texttt{Num} and \texttt{Den} are relative prime. Note that integer \texttt{N} looks like \texttt{rat(N,1)} in this representation. You can control printing of terms with \texttt{user:portray/1}.

Partial Evaluation, Compilation. Once one has a working solver, it is obvious and attractive to run the constraints in a clause definition at read time or compile time and proceed with the answer constraints in place of the original constraints. This gets you constant folding and in fact the full algebraic power of the solver applied to the avoidance of computations at run time. The mechanism to realize this idea is to use \texttt{dump/3} for the expansion of \{\}/1, via the goal and term expansion hook predicates.

Asserting with Constraints. If you use the database, then the clauses you assert might have constraints associated with their variables. You should use \texttt{projecting_assert/1} instead of \texttt{assert/1} in order to ensure that only the relevant and projected constraints get stored in the database.

```
| ?- \{A+B=<33\}, projecting_assert(test(A,B)).
{A+B=<33}
```

```
| ?- listing(test).
\texttt{test(A, B) :-}
    \{A+B=<\texttt{rat(33,1)}\} 
```

```
| ?- test(A,B).
\{A+B=<33\}
```

10.11.9.2 Bugs
- The fuzzy comparison of floats is the source for all sorts of weirdness. If a result in R surprises you, then try to run the program in Q before you send me a bug report.
• The projector for floundered nonlinear relations keeps too many variables. Its output is rather unreadable.
• Disequations are not projected properly.
• This list is probably incomplete.

10.12 I/O on Lists of Character Codes—library(codesio)

This package defines I/O predicates that read from, or write to, a code list. There are also predicates to open a stream referring to a code list. The stream may be used with general Stream I/O predicates.

Exported predicates:

format_to_codes(+Format, :Arguments, -Codes)
format_to_codes(+Format, :Arguments, ?S0, ?S)

Prints Arguments into a code list using format/3. Codes is unified with the list, alternatively S0 and S are unified with the list and its end, respectively.

write_to_codes(+Term, -Codes)
write_to_codes(+Term, ?S0, ?S)

A specialized format_to_codes/[3,4]. Writes Term into a code list using write/2. Codes is unified with the list. Alternatively, S0 and S are unified with the list and its end, respectively.

write_term_to_codes(+Term, -Codes, +Options)
write_term_to_codes(+Term, ?S0, ?S, +Options)

A specialized format_to_codes/[3,4]. Writes Term into a code list using write_term/3 and Options. Codes is unified with the list. Alternatively, S0 and S are unified with the list and its end, respectively.

read_from_codes(+Codes, -Term)

Reads Term from Codes using read/2. The Codes must, as usual, be terminated by a full stop, i.e. a ‘.’, possibly followed by layout-text.

read_term_from_codes(+Codes, -Term, +Options)

Reads Term from Codes using read_term/3 and Options. The Codes must, as usual, be terminated by a full stop, i.e. a ‘.’, possibly followed by layout-text.

open_codes_stream(+Codes, -Stream)

Stream is opened as an input stream to an existing code list. The stream may be read with the Stream I/O predicates and must be closed using close/1. The list is copied to an internal buffer when the stream is opened and must therefore be a ground code list at that point.

with_output_to_codes(:Goal, -Codes)
with_output_to_codes(:Goal, ?S0, ?S)
with_output_to_codes(:Goal, -Stream, ?S0, ?S)

Goal is called with the current_output stream set to a new stream. This stream writes to an internal buffer, which is, after the successful execution of Goal, converted to a list of character codes. Codes is unified with the list,
alternatively $S_0$ and $S$ are unified with the list and its end, respectively. with_output_to_codes/4 also passes the stream in the Stream argument. It can be used only by Goal for writing.

**10.13 I/O on Comma-Separated Values (CSV) Files and Strings**—library(csv)

This library module provides some utilities for Comma-Separated Values (CSV) files and strings. In this context, a file is a sequence of records, and a record is a sequence of fields. In a CSV file, fields are separated by commas, and each record is terminated by RET.

This module does not report any syntax errors. In the event of prematurely terminated input file, the current field and record will be terminated silently.

Then a CSV record is read, it will yield a list of fields of the following form:

- **integer(Number,Codes)**
  - Stands for the integer Number, where number_codes(Number,Codes) holds, and Codes is the list of character codes actually read.

- **float(Number,Codes)**
  - Stands for the float Number, where number_codes(Number,Codes) holds, and Codes is the list of character codes actually read.

- **string(Codes)**
  - Stands for the text string (list of character codes) Codes, and number_codes(Number,Codes) does not hold.

When a CSV records is written, the Codes argument of the above terms is used, but the following fields are also allowed:

- **integer(Number)**
  - Stands for the integer Number.

- **float(Number)**
  - Stands for the float Number.

- **atom(Atom)**
  - Stands for the atom Atom.

Adapted to the conventions of this manual, RFC 4180 specifies the following. Where this module relaxes the requirements, that is explicitly mentioned:

1. Each record is located on a separate line, delimited by a line break. For example:
   - aaa,bbb,ccc RET
   - zzz,yyy,xxx RET

2. The last record in the file may or may not have an ending line break. For example:
   - aaa,bbb,ccc RET
   - zzz,yyy,xxx
3. There may be an optional header line appearing as the first line of the file with the same format as normal record lines. This header will contain names corresponding to the fields in the file and should contain the same number of fields as the records in the rest of the file. For example:

   field_name,field_name,field_name RET
   aaa,bbb,ccc RET
   zzz,yyy,xxx RET

   This module does not attempt to detect a header line nor treat it in any special way.

4. Within the header and each record, there may be one or more fields, separated by commas. Each record should contain the same number of fields throughout the file. Spaces are considered part of a field and should not be ignored. The last field in the record must not be followed by a comma, so if the record ends with a comma, the last field is treated as empty. For example, the following is treated as four fields:

   aaa,bbb,ccc,

   This module does not require or check that each record contains the same number of fields.

5. Each field may or may not be enclosed in double quotes. If fields contain line breaks (RET), double quotes or commas, then they should be enclosed in double quotes, otherwise the double quotes may be omitted. For example:

   "aaa","bbb","ccc" RET
   "aaa","b RET
   bb","ccc" RET
   zzz,yyy,xxx

   If an unenclosed field is immediately followed by a ",", (or vice versa), then this module treats that as a new enclosed (or unenclosed) field to be read and appended to the field read so far.

6. If double quotes are used to enclose fields, then a double quote appearing inside a field must be escaped by preceding it with another double quote. For example:

   "aaa","b""bb","ccc"

Exported predicates:

read_record(-Record)
read_record(+Stream, -Record)
   Reads a single record from the stream Stream, which defaults to the current input stream, and unifies it with Record. On end of file, Record is unified with end_of_file.
read_records(-Records)
read_records(+Stream, -Records)
   Reads records from the stream Stream, which defaults to the current input stream, up to the end of the stream, and unifies them with Records.
read_record_from_codes(-Record, +Codes)
read_record_from_codes(-Record, +Codes, -Suffix)
   Reads a record from the code list Codes. In the arity 2 variant, there must be no trailing character codes after the record. In the arity 3 variant, any
trailing character codes are unified with Suffix, which can be used for reading subsequent records.

```
write_record(+Record)
write_record(+Stream, +Record)
```

Writes a single record to the stream Stream, which defaults to the current output stream.

```
write_records(+Records)
write_records(+Stream, +Records)
```

Writes records to the stream Stream, which defaults to the current output stream.

```
write_record_to_codes(+Record, -Codes)
```

Writes a single record to the code list Codes, without the terminating RET.

### 10.14 COM Client—library(comclient)

This library provides rudimentary access to COM automation objects. As an example it is possible to manipulate Microsoft Office applications and Internet Explorer. It is not possible, at present, to build COM objects using this library.

Feedback is very welcome. Please contact SICStus support (sicstus-support@ri.se) if you have suggestions for how this library could be improved.

#### 10.14.1 Preliminaries

In most contexts both atoms and code lists are treated as strings. With the wide character support available in release 3.8 and later, is should now be possible to pass UNICODE atoms and strings successfully to the COM interface.

#### 10.14.2 Terminology

- **ProgID** A human readable name for an object class, typically as an atom, e.g. 'Excel.Application'.
- **CLSID** (Class Identifier) A globally unique identifier of a class, typically as an atom, e.g. '{00024500-0000-0000-C000-000000000046}'. Where it makes sense a ProgID can be used instead of the corresponding CLSID.
- **IID** (Interface Identifier) A globally unique identifier of an interface. Currently only the 'IDispatch' interface is used so you do not have to care about this.
- **IName** (Interface Name) The human readable name of an interface, e.g. 'IDispatch'. Where it makes sense an IName can be used instead of the corresponding IID.
- **Object** A COM-object (or rather a pointer to an interface).
- **ComValue** A value that can be passed from COM to SICStus Prolog. Currently numeric types, booleans (treated as 1 for true, 0 for false), strings, and COM objects.
**ComInArg**

A value that can be passed as an input argument to COM, currently one of:

- **atom**: Passed as a string (BSTR)
- **numeric**: Passed as the corresponding number
- **list**: A code list is treated as a string.
- **COM object**: A compound term referring to a COM object.
- **compound**: Other compound terms are presently illegal but will be used to extend the permitted types.

**SimpleCallSpec**

Denotes a single method and its arguments. As an example, to call the method named `foo` with the arguments 42 and the string "bar" the `SimpleCallSpec` would be the compound term `foo(42, 'bar')` or, as an alternative, `foo(42, "bar")`.

The arguments of the compound term are treated as follows:

- **ComInArg**: See above
- **variable**: The argument is assumed to be output. The variable is bound to the resulting value when the method returns.
- **mutable**: The argument is assumed to be input/output. The value of the mutable is passed to the method and when the method returns the mutable is updated with the corresponding return value.

**CallSpec**

Either a `SimpleCallSpec` or a list of `CallSpecs`. If it is a list then all but the last `SimpleCallSpec` are assumed to denote method calls that return a COM-object.

So for instance the VB statement `app.workbooks.add` can be expressed either as:

```prolog
comclient_invoke_method_proc(App, [workbooks, add])
```

or as

```prolog
comclient_invoke_method_fun(App, workbooks, WorkBooks),
comclient_invoke_method_proc(WorkBooks, add),
comclient_release(WorkBooks)
```

### 10.14.3 Predicate Reference

**comclient_garbage_collect**

Release Objects that are no longer reachable from SICStus Prolog. To achieve this the predicate `comclient_garbage_collect/0` performs an atom garbage collection, i.e. `garbage_collect_atoms/0`, so it should be used sparingly.

**comclient_is_object(+Object)**

Succeeds if `Object" looks like" an object. It does not check that the object is (still) reachable from SICStus Prolog, see `comclient_valid_object/1`. Currently an object looks like `$comclient_object'(stuff) where `stuff` is some prolog term. Do not rely on this representation!
comclient_valid_object(+Object)
Succeeds if Object is an object that is still available to SICStus Prolog.

comclient_equal(+Object1, +Object2)
Succeeds if Object1 and Object2 are the same object. (It succeeds if their 'IUnknown' interfaces are identical)

comclient_clsid_from_progid(+ProgID, -CLSID).
Obtain the CLSID corresponding to a particular ProgID. Uses the Win32 routine CLSIDFromProgID. You rarely need this since you can use the ProgID directly in most cases.

comclient_progid_from_clsid(+CLSID, -ProgID).
Obtain the ProgID corresponding to a particular CLSID. Uses the Win32 routine ProgIDFromCLSID. Rarely needed. The ProgID returned will typically have the version suffix appended.
Example, to determine what version of Excel.Application is installed:
\[
\text{?- comclient_clsid_from_progid('Excel.Application, CLSID),}
\text{comclient_progid_from_clsid(CLSID, ProgID).}
\]
\[
\text{CLSID = '{00024500-0000-0000-C000-000000000046}',}
\text{ProgID = 'Excel.Application.8'}
\]

comclient_iid_from_name(+IName, -IID)
Look in the registry for the IID corresponding to a particular Interface. Currently of little use.
\[
\text{?- comclient_iid_from_name('IDispatch', IID).}
\text{IID = '{00020400-0000-0000-C000-000000000046}'}
\]

comclient_name_from_iid(+IID, -IName)
Look in the registry for the name corresponding to a particular IID. Currently of little use.

comclient_create_instance(+ID, -Object)
Create an instance of the Class identified by the CLSID or ProgID ID.
\[
\text{comclient_create_instance('Excel.Application', App)}
\]
Corresponds to CoCreateInstance.

comclient_get_active_object(+ID, -Object)
Retrieves a running object of the Class identified by the CLSID or ProgID ID.
\[
\text{comclient_get_active_object('Excel.Application', App)}
\]
An exception is thrown if there is no suitable running object. Corresponds to GetActiveObject.

comclient_invoke_method_fun(+Object, +CallSpec, -ComValue)
Call a method that returns a value. Also use this to get the value of properties.

comclient_invoke_method_proc(+Object, +CallSpec)
Call a method that does not return a value.

comclient_invoke_put(+Object, +CallSpec, +ComInArg)
Set the property denoted by CallSpec to ComValue. Example: comclient_invoke_put(App, visible, 1)
comclient_release(+Object)
Release the object and free the data structures used by SICStus Prolog to keep track of this object. After releasing an object the term denoting the object can no longer be used to access the object (any attempt to do so will raise an exception).

Please note: The same COM-object can be represented by different prolog terms. A COM object is not released from SICStus Prolog until all such representations have been released, either explicitly by calling comclient_release/1 or by calling comclient_garbage_collect/0.

You cannot use Obj1 == Obj2 to determine whether two COM-objects are identical. Instead use comclient_equal/2.

comclient_is_exception(+ExceptionTerm)
Succeeds if ExceptionTerm is an exception raised by the comclient module.

catch(<some code>),
  Exception,
  ( comclient_is_exception(E) ->
     handle_com_related_errors(E)
  ; otherwise -> % Pass other exceptions upwards
     throw(E)
  ))

comclient_exception_code(+ExceptionTerm, -ErrorCode)
comclient_exception_culprit(+ExceptionTerm, -Culprit)
comclient_exception_description(+ExceptionTerm, -Description)
Access the various parts of a comclient exception. The ErrorCode is the HRESULT causing the exception. Culprit is a term corresponding to the call that gave an exception. Description, if available, is either a term 'EXCEPINFO'(...) corresponding to an EXCEPINFO structure or 'ARGERR'(MethodName, ArgNumber).

The EXCEPINFO has six arguments corresponding to, and in the same order as, the arguments of the EXCEPINFO struct.

10.14.4 Examples
The following example launches Microsoft Excel, adds a new worksheet, fill in some fields and finally clears the worksheet and quits Excel
:- use_module(library(comclient)).
:- use_module(library(lists)).

test :-
   test('Excel.Application').

test(ProgID) :-
   comclient_create_instance(ProgID, App),
   % Visual Basic: app.visible = 1
   comclient_invoke_put(App, visible, 1),
   % Visual Basic: app.workbooks.add
   comclient_invoke_method_proc(App, [workbooks, add]),
   % Visual Basic: with app.activesheet
   comclient_invoke_method_fun(App, activesheet, ActiveSheet),

   Rows = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
   Cols = Rows,
   % Visual Basic: .cells i,j . value = i+j/100
   ( member(I, Rows),
     member(J, Cols),
     ValIJ is I+J/100,
     comclient_invoke_put(ActiveSheet, [cells(I,J),value], ValIJ),
     fail ; true ),
   ( member(I, Rows),
     member(J, Cols),
     % retrieve cell values
     comclient_invoke_method_fun(ActiveSheet, [cells(I,J), value],CellValue),
     format(user_error, '~nCell(~w,~w) = ~w', [I,J,CellValue]),
     fail ; true ),

   Range = 'A1:O15',
   format(user_error, '~nPress return to clear range (~w)', [Range]),
   flush_output(user_error),
   get_code(_),

   % Visual Basic: .range A1:O15 .Clear
   comclient_invoke_method_proc(ActiveSheet, [range(Range),clear]),

   % Avoid Excel query "do you want to save..."
   % Visual Basic: app.activeworkbook.saved = 1
   comclient_invoke_put(App, [activeworkbook,saved], 1),
   format(user_error, '~nPress return to quit \"~w\"', [ProgID]),
   flush_output(user_error),
   get_code(_),
10.15 Finite Domain Constraint Debugger—library(fdbg)

10.15.1 Introduction

FDBG is a CLP(FD) debugger for SICStus Prolog. Its main purpose is to enable the CLP programmer to trace the changes of domains of variables. See [Hanak et al. 04].

FDBG defines the following prefix operator:

\[ \text{:- op(400, fy, #).} \]

The presence of FDBG affects the translation and execution, but not the semantics, of subsequently loaded arithmetic constraints.

10.15.2 Concepts

In this section, several concepts and terms are defined. These terms will later be heavily used in the documentation; therefore, it is important that you understand them well.

10.15.2.1 Events

An FDBG event can (currently) belong to one of the two following major classes:

*constraint event*

A constraint is woken.

*labeling event*

Three events belong to this class, namely:

- the labeling of an FD variable is started
- an FD variable gets constrained
- the labeling of an FD variable fails, i.e. all elements of its domain have been tried and caused failure

These events are intercepted by the FDBG core. When any of them occurs, the appropriate visualizer (see Section 10.15.2.3 [FDBG Visualizers], page 533) gets called with a representation of the event (a Prolog term) as extra arguments.

10.15.2.2 Labeling Levels

In this subsection we give three definitions regarding the labeling procedure.

*labeling session*

This term denotes the whole labeling procedure that starts with the call of `labeling/2` or an equivalent predicate and finishes by exiting this predicate. Normally, there is at most one labeling session per run.

*labeling attempt*

One choicepoint of a labeling session. Exactly one variable is associated with a labeling attempt, although this is not necessarily true vice versa. For example in `enum` mode labeling, a single labeling attempt tries every possible value, but in `step` mode labeling, several binary choicepoints are created.
labeling step
The event of somehow constraining the domain of a variable. This usually means either setting the variable to a specific value or limiting it with a lower or an upper bound.

As you can see there is a hierarchical relation among these definitions: a labeling session consists of several labeling attempts, which, in turn, might consist of several labeling steps.

A labeling event, on the other hand, can either be a labeling step, or the start of a labeling attempt, or the failure of the same. See Section 10.15.2.1 [FDBG Events], page 532.

10.15.2.3 Visualizers
A visualizer is a Prolog predicate reacting to FDBG events (see Section 10.15.2.1 [FDBG Events], page 532). It is called directly by the FDBG core when any FDBG event occurs. It is called visualizer, because usually it should present the events to the user, but in general it can do any kind of processing, like checking invariants, etc.

For all major event classes, a different visualizer type is used. The set of visualizers you would like to use for a session is specified in the option list of fdbg_on/1 (see Section 10.15.3.1 [FDBG Options], page 535), when FDBG is switched on.

A specific visualizer can have several arguments, some are supplied by the FDBG core, the rest (if any) should be specified when FDBG is switched on. Note that the obligatory arguments will be appended to the end of the user defined argument list.

The set of built-in visualizers installed by default (see Section 10.15.3.1 [FDBG Options], page 535) is the following:

- for constraint awakenings: fdbg_show
- for labeling events: fdbg_label_show

For details on built-in visualizers, see Section 10.15.3.3 [FDBG Built-In Visualizers], page 537.

10.15.2.4 Names of Terms
FDBG provides a service to assign names to Prolog terms for later reference. A name is an atom and it is usually associated with a compound term containing constraint variables, or with a single variable. In the former case, each variable appearing in the compound term is also assigned a name automatically by FDBG. This auto-assigned name is derived from the name of the term; see Section 10.15.2.6 [FDBG Name Auto-Generation], page 534.

Perhaps the most useful utilization of names is annotation, another service of FDBG. Here, each variable appearing in a Prolog term is replaced with a compound term describing it (i.e. containing its name, the variable itself, and some data regarding its domain). During annotation, unnamed constraint variables are also given a unique “anonymous” name automatically, these names begin with a ‘fdvar’ prefix. See Section 10.15.4.2 [FDBG Writing Visualizers], page 544.
The names will be used by the built-in visualizers when referring to constraint variables, and they can also be used to retrieve the terms assigned to them in user defined visualizers. See Section 10.15.2.3 [FDBG Visualizers], page 533.

### 10.15.2.5 Selectors

A selector is a Prolog term denoting a (path to a) subterm of a given term \( T \). Let \( \text{subterm}(T,S) \) denote the subterm of \( T \) wrt. a selector \( S \), and let \( N \) denote an integer. A selector then takes one of the following forms:

\[
S \quad \text{subterm}(T,S) \\
[] \quad T \\
[\ldots, N] \quad \text{Nth argument of the compound term subterm}(T,[\ldots]) \\
[\ldots, #N] \quad \text{Nth element of the list subterm}(T,[\ldots])
\]

### 10.15.2.6 Name Auto-Generation

There are two cases when a name is automatically generated.

1. When a name is assigned to a compound term by the user, each variable appearing in it is assigned a so called derived name, which is created by appending a variant of the selector of the variable to the original name. For example, the call:

\[
\text{fdbg_assign_name}(\text{bar}(A, [B, C], \text{foobar}(D, E)), \text{foo})
\]

will create the following name/term entries:

<table>
<thead>
<tr>
<th>Name</th>
<th>Term/Variable</th>
<th>Selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>bar(A, [B, C], foobar(D, E))</td>
<td>[]</td>
</tr>
<tr>
<td>foo_1</td>
<td>A</td>
<td>[1]</td>
</tr>
<tr>
<td>foo_2_1</td>
<td>B</td>
<td>[2,#1]</td>
</tr>
<tr>
<td>foo_2_2</td>
<td>C</td>
<td>[2,#2]</td>
</tr>
<tr>
<td>foo_3_1</td>
<td>D</td>
<td>[3,1]</td>
</tr>
<tr>
<td>foo_3_2</td>
<td>E</td>
<td>[3,2]</td>
</tr>
</tbody>
</table>

See Section 10.15.3.2 [FDBG Naming Terms], page 536.

2. If, during the annotation of a term (see Section 10.15.3.5 [FDBG Annotation], page 539) an unnamed constraint variable is found, then it is assigned a unique “anonymous” name. This name consists of the prefix ‘fdvar’, an underscore character, and an integer. The integer is automatically incremented when necessary.

### 10.15.2.7 Legend

The legend is a list of variables and their domains, usually appearing after a description of the current constraint. This is necessary because the usual visual representation of a constraint contains only the names of the variables in it (see Section 10.15.3.5 [FDBG Annotation], page 539), and does not show anything about their domain. The legend links these names to the corresponding domains. The legend also shows the changes of the domains made by the constraint. Finally, the legend may contain some conclusions regarding the behavior of the constraint, like failure or side effects.
The format of the legend is somewhat customizable by defining a hook function; see Section 10.15.4.1 [FDBG Customizing Output], page 543. The default format of the legend is the following:

\[
\begin{align*}
\text{list}_2 &= 0..3 \\
\text{list}_3 &= 0..3 \\
\text{list}_4 &= 0..3 \\
\text{fdvar}_2 &= 0..3 \rightarrow 1..3
\end{align*}
\]

Here, we see four variables, with initial domains 0..3, but the domain of the (previously unnamed) variable \text{fdvar}_2 is narrowed by the constraint (not shown here) to 1..3.

A legend is automatically printed by the built-in visualizer \textit{fdbg\_show}, but it can be easily printed from user defined visualizers too.

10.15.2.8 The \textit{fdbg\_output} Stream

The \textit{fdbg\_output} is a stream alias created when FDBG is switched on and removed when it is switched off. All built-in visualizers write to this stream, and the user defined visualizers should do the same.

10.15.3 Basics

Here, we describe the set of FDBG services and commands necessary to do a simple debugging session. No major modification of your CLP(FD) program is necessary to use FDBG this way. Debugging more complicated programs, on the other hand, might also require user written extensions to FDBG, since the wallpaper trace produced by the built-in visualizer \textit{fdbg\_show} could be too detailed and therefore hard to analyze. See Section 10.15.4 [FDBG Advanced Usage], page 543.

10.15.3.1 FDBG Options

FDBG is switched on and off with the predicates:

\begin{verbatim}
fdbg_on
fdbg_on(:Options)
\end{verbatim}

Turns on FDBG by putting advice points on several predicates of the CLP(FD) module. \textit{Options} is a list of options; see Section 10.15.3.1 [FDBG Options], page 535. The empty list is the default value.

\begin{verbatim}
fdbg_on/[0,1]
\end{verbatim}
can be called safely several times consecutively; only the first call will have an effect.

\begin{verbatim}
fdbg_off
\end{verbatim}

Turns the debugger off by removing the previously installed advice points.

\begin{verbatim}
fdbg_on/1
\end{verbatim}
accepts the following options:

\begin{verbatim}
file(Filename, Mode)
\end{verbatim}

Tells FDBG to attach the stream alias \textit{fdbg\_output} to the file called \textit{Filename} opened in mode \textit{Mode}. \textit{Mode} can either be \textit{write} or \textit{append}. The file specified is opened on a call to \textit{fdbg\_on/1} and is closed on a call to \textit{fdbg\_off/0}. 

socket(Host, Port)
Tells FDBG to attach the stream alias fdbg_output to the socket connected to Host on port Port. The specified socket is created on a call to fdbg_on/1 and is closed on a call to fdbg_off/0.

stream(Stream)
Tells FDBG to attach the stream alias fdbg_output to the stream Stream. The specified stream remains open after calling fdbg_off/0. If none of the above three options is used, then the stream alias fdbg_output is attached to the current output stream.

custom_hook(Goal)
Tells FDBG to extend Goal with two (additional) arguments and call it on the exit port of the constraint dispatcher.

no_custom_hook
Tells FDBG not to use any custom hook. If none of the above two options is used, then the default is custom_hook(fdbg:fdbg_show).

labeling_hook(Goal)
Tells FDBG to extend Goal with three (additional) arguments and call it on any of the three labeling events.

no_labeling_hook
Tells FDBG not to use any labeling hook. If none of the above two options is used, then the default is labeling_hook(fdbg:fdbg_label_show).

For both custom_hook and labeling_hook, Goal should be a visualizer, either built-in (see Section 10.15.3.3 [FDBG Built-In Visualizers], page 537) or user defined. More of these two options may appear in the option list, in which case they will be called in their order of occurrence.

See Section 10.15.4.2 [FDBG Writing Visualizers], page 544, for more details on these two options.

10.15.3.2 Naming Terms
Naming is a procedure of associating names with terms and variables; see Section 10.15.2.4 [FDBG Names of Terms], page 533. Three predicates are provided to assign and retrieve names, these are the following:

fdbg_assign_name(+Term, ?Name)
Assigns the atom Name to Term, and a derived name to each variable appearing in Term. If Name is a variable, then use a default (generated) name, and return it in Name. See Section 10.15.2.6 [FDBG Name Auto-Generation], page 534.
fdbg_current_name(?Term, ?Name)
  Retrieves Term associated with Name, or enumerates all term-name pairs.

fdbg_get_name(+Term, -Name)
  Returns the name associated to Term in Name, if it exists. Otherwise, silently fails.

10.15.3.3 Built-In Visualizers

The default visualizers are generic predicates to display FDBG events (see Section 10.15.2.1 [FDBG Events], page 532) in a well readable form. These visualizers naturally do not exploit any problem specific information—to have more “fancy” output, you have to write your own visualizers; see Section 10.15.4.2 [FDBG Writing Visualizers], page 544. To use these visualizers, pass them in the appropriate argument to fdbg_on/1; see Section 10.15.3.1 [FDBG Options], page 535, or call them directly from user defined visualizers.

fdbg_show(+Constraint, +Actions)
  This visualizer produces a trace output of all woken constraints, in which a line showing the constraint is followed by a legend (see Section 10.15.2.7 [FDBG Legend], page 534) of all the variables appearing in it, and finally an empty line to separate events from each other. The usual output will look like this:

  <fdvar_1>=0
  fdvar_1 = inf..sup -> {0}
  Constraint exited.

  Here, we can see an arithmetical constraint being woken. It narrows ‘fdvar_1’ to a domain consisting of the singleton value 0, and since this is the narrowest domain possible, the constraint does not have anything more to do: it exits.

  Note that when you pass fdbg_show as an option, you should omit the two arguments, like in:

  fdbg_on([... constraint_hook(fdbg_show), ...]).

fdbg_label_show(+Event, +LabelID, +Variable)
  This visualizer produces a wallpaper trace output of all labeling events. It is best used together with fdbg_show/2. Each labeling event produces a single line of output, some of them are followed by an empty line, some others are always followed by another labeling action and therefore the empty line is omitted.

  Here is a sample output of fdbg_label_show/3:

  Labeling [9, <list_1>]: starting in range 0..3.
  Labeling [9, <list_1>]: step: <list_1> = 0

  What we see here is the following:

  • The prefix ‘Labeling’ identifies the event.
  • The number in the brackets (9) is a unique identification number belonging to a labeling attempt. Only one labeling step with this number can be in effect at a time. This number in fact is the invocation number of the predicate doing the labeling for that variable.
  • The name in the brackets (<list_1>) identifies the variable currently being labeled. Note that several identification numbers might belong to the same variable, depending on the mode of labeling.
• The text after the colon specifies the actual labeling event. This string can be:
  
  − “starting in range Range.” meaning the starting of a labeling attempt
    in range Range
  − “Mode: Narrowing.” meaning a labeling step in mode Mode. Narrowing
    is the actual narrowing done in the labeling step. Mode is one of the following:
    
    step      meaning step mode labeling
    indomain_up  meaning enum mode labeling or a direct call to indomain/1
    indomain_down meaning enum,down mode labeling
    bisect     meaning bisect mode labeling
    dual       when the domain contains exactly two values and the labeling
                attempt is nothing more than a selection between them
  − “failed.” meaning the labeling attempt failed.

Note that when you pass fdbg_label_show/3 as an option, you should omit
the three arguments, like in

fdbg_on([..., labeling_hook(fdbg_label_show), ...]).

10.15.3.4 New Debugger Commands

The Prolog debugger is extended by FDBG. The & debugger is modified, and two new
commands are added:

&

& N  This debugger command is extended so that the annotated form of domain
variables is also printed when listing the variables with blocked goals.

A

A Selector

  Annotates and prints the current goal and a legend of the variables appearing
  in it. If a selector is specified, then the subterm specified by it is assumed to
  be an action list, and is taken into account when displaying the legend. For
  example:

  23  2 Exit: clpfd:dispatch_global_fast(no_threat(2,_1001,1),0,0,

  clpfd:dispatch_global_fast(no_threat(2,<board_2>,1),0,0,
                   [exit,<board_2> in_set[[3|3]]])

  board_2 = 1..4 -> {3}

  Constraint exited.
Name=Selector
Assigns the atom Name to the variable specified by the Selector. For example:

7 15 Call: bar(4, [_101, _102, _103]) ? W foo=[2, #2]
This would assign the name foo to _102, being the second element of the second argument of the current goal.

10.15.3.5 Annotating Programs
In order to use FDBG efficiently, you have to make some changes to your CLP(FD) program. Fortunately the calls you have to add are not numerous, and when FDBG is turned off they do not decrease efficiency significantly or modify the behavior of your program. On the other hand, they are necessary to make FDBG output easier to understand.

Assign names to the more important and more frequently occurring variables by inserting fdbg_assign_name/2 calls at the beginning of your program. It is advisable to assign names to variables in larger batches (i.e. as lists or compound terms) with a single call.

Use predefined labeling predicates if possible. If you define your own labeling predicates and you want to use them even in the debugging session, then you should follow these guidelines:

1. Add a call to clpfd:fdbg_start_labeling(+Var) at the beginning of the predicate doing a labeling attempt, and pass the currently labeled variable as an argument to the call.
2. Call clpfd:fdbg_labeling_step(+Var, +Step) before each labeling step. Step should be a compound term describing the labeling step, this will be
   a. printed “as is” by the built-in visualizer as the mode of the labeling step (see Section 10.15.3.3 [FDBG Built-In Visualizers], page 537)—you can use portray/1 to determine how it should be printed;
   b. passed as step(Step) to the user defined labeling visualizers in their Event argument; see Section 10.15.4.2 [FDBG Writing Visualizers], page 544.

This way FDBG can inform you about the labeling events created by your labeling predicates exactly like it would do in the case of internal labeling. If you ignore these rules, then FDBG will not be able to distinguish labeling events from other FDBG events any more.

10.15.3.6 An Example Session
The problem of magic sequences is a well known constraint problem. A magic sequence is a list, where the i-th item of the list is equal to the number of occurrences of the number i in the list, starting from zero. For example, the following is a magic sequence:

[1, 2, 1, 0]
The CLP(FD) solution can be found in library('clpfd/examples/magicseq'), which provides a couple of different solutions, one of which uses the global_cardinality/2 constraint. We'll use this solution to demonstrate a simple session with FDBG.

First, the debugger is imported into the user module:
| ?- use_module(library(fdbg)).  
% loading /home/matsc/sicstus3/Utils/x86-linux-glibc2.2/lib/sicstus-3.9.1/library/fdbg.po...  
% module fdbg imported into user  

[...]  

% loaded /home/matsc/sicstus3/Utils/x86-linux-glibc2.2/lib/sicstus-3.9.1/library/fdbg.po in module fdbg, 220 msec 453936 bytes  

| ?- use_module(library(clpfd)).  
[...]  

Then, the magic sequence solver is loaded:  

| ?- [library('clpfd/examples/magicseq')].  
% consulting /home/matsc/sicstus3/Utils/x86-linux-glibc2.2/lib/sicstus-3.9.1/library/clpfd/examples/magicseq.pl...  
% module magic imported into user  
% module clpfd imported into magic  
% consulted /home/matsc/sicstus3/Utils/x86-linux-glibc2.2/lib/sicstus-3.9.1/library/clpfd/examples/magicseq.pl in module magic, 30 msec 9440 bytes  

Now we turn on the debugger, telling it to save the trace in fdbg.log.  

| ?- fdbg_on([file('fdbg.log',write)]).  
% The clp(fd) debugger is switched on  

To produce a well readable trace output, a name has to be assigned to the list representing the magic sequence. To avoid any modifications to the source code, the name is assigned by a separate call before calling the magic sequence finder predicate:  

| ?- length(L,4), fdbg_assign_name(L,list), magic_demo(4,L).  
L = [1,2,1,0] ?;  
L = [2,0,2,0] ?;  

no  

Please note: the call to length/2 is necessary; otherwise, L would be a single variable instead of a list of four variables when the name is assigned.  

Finally we turn the debugger off:  

| ?- fdbg_off.  
% The clp(fd) debugger is switched off
The output of the sample run can be found in \texttt{fdbg.log}. Here, we show selected parts of the trace. In each block, the woken constraint appears on the first line, followed by the corresponding legend.

In the first selected block, \texttt{scalar_product/4} removes infeasible domain values from \texttt{list_4}, adjusting its upper bound. The legend shows the domains before and after pruning. Note also that the constraint is rewritten to a more readable form:

\[
\text{<list_2> + 2*<list_3> + 3*<list_4> # = <list_1> + <list_2> + <list_3> + <list_4>}
\]

\[
\text{list_1} = 0..3
\]
\[
\text{list_2} = 0..3
\]
\[
\text{list_3} = 0..3
\]
\[
\text{list_4} = 0..3 \rightarrow 0..1
\]

The following block shows the initial labeling events, trying the value 0 for \texttt{list_1}:

\begin{verbatim}
Labeling [9, <list_1>]: starting in range 0..3.
Labeling [9, <list_1>]: indomain_up: <list_1> = 0
\end{verbatim}

This soon leads to a dead end:

\begin{verbatim}
<list_1> = 0
list_1 = 0..3 \rightarrow \{0\}
Constraint exited.
\end{verbatim}

\begin{verbatim}
<list_2> + 2*<list_3> + 3*<list_4> # = <list_1> + <list_2> + <list_3> + <list_4>

list_2 = 0..3
list_3 = 0..3 \rightarrow \{0\}
list_4 = 0..1 \rightarrow \{0\}
Constraint exited.
\end{verbatim}

\begin{verbatim}
<list_2> + <list_3> + <list_4> # = 4
list_2 = 0..3
list_3 = \{0\}
list_4 = \{0\}
Constraint failed.
\end{verbatim}

We backtrack on \texttt{list_1}, trying instead the value 1. This leads to the following propagation steps and to the first solution. In these propagation steps, the constraint exits, which means that it holds no matter what value any remaining variable takes (like \texttt{list_2} in the second step):
Labeling [9, <list_1>]: indomain_up: <list_1> = 1

<list_1>=1
  list_1 = 0..3 -> {1}
  Constraint exited.

<list_2>+2*<list_3>+3*<list_4>#=1+<list_2>+<list_3>+<list_4>
  list_2 = 0..3
  list_3 = 0..3 -> {1}
  list_4 = 0..1 -> {0}
  Constraint exited.

1+<list_2>+<list_3>+<list_4>#=4
  list_2 = 0..3 -> {2}
  list_3 = {1}
  list_4 = {0}
  Constraint exited.

global_cardinality([1,<list_2>,<list_3>,<list_4>],[0-1,1-<list_2>,2-<list_3>,3-<list_4>]
  list_2 = {2}
  list_3 = {1}
  list_4 = {0}
  Constraint exited.

Then, we backtrack again on list_1, which leads to the second solution after a chain of propagation steps:

Labeling [9, <list_1>]: indomain_up: <list_1> = 2

[...]

global_cardinality([2,<list_2>,<list_3>,<list_4>],[0-2,1-<list_2>,2-<list_3>,3-<list_4>]
  list_2 = {0}
  list_3 = {2}
  list_4 = {0}
  Constraint exited.

Then we backtrack on list_1 yet another time, but no more solutions are found:
Labeling [9, <list_1>]: indomain_up: <list_1> = 3

[...]  
<list_2>+2*<list_3>+3*<list_4>#=3+<list_2>+<list_3>+<list_4>
   list_2 = {0}
   list_3 = {1}
   list_4 = {0}
Constraint failed.

Labeling [9, <list_1>]: failed.

10.15.4 Advanced Usage
Sometimes the output of the built-in visualizer is inadequate. There might be cases when only minor changes are necessary to produce a more readable output; in other cases, the trace output should be completely reorganized. FDBG provides two ways of changing the appearance of the output by defining hook predicates. In this section, these predicates will be described in detail.

10.15.4.1 Customizing Output
The printing of variable names is customized by defining the following hook predicate.

\[\text{fdbg:fdvar_portray(Name, Var, FDSet)}\]

This hook predicate is called whenever an annotated constraint variable (see Section 10.15.3.5 [FDBG Annotation], page 539) is printed. Name is the assigned name of the variable Var, whose domain will be FDSet as soon as the narrowings of the current constraint take effect. The current domain is not passed to the hook, but it can be easily determined with a call to \text{fd_set/2}. (Although these two sets may be the same if the constraint did not narrow it.)

If \text{fdbg:fdvar_portray/3} is undefined or fails, then the default representation is printed, which is \text{Name} between angle brackets.

The printing of legend lines is customized by defining the following hook predicate.

\[\text{fdbg:legend_portray(Name, Var, FDSet)}\]

This hook is called for each line of the legend by the built-in legend printer. The arguments are the same as in the case of \text{fdbg:fdvar_portray/3}. Note that a prefix of four spaces and a closing newline character is always printed by FDBG.

If \text{fdbg:fdvar_portray/3} is undefined or fails, then the default representation is printed, which is

\text{Name} = \text{RangeNow} [ \rightarrow \text{RangeAfter} ]

The arrow and \text{RangeAfter} are only printed if the constraint narrowed the domain of \text{Var}.
The following example will print a list of all possible values instead of the range for each variable in the legend:

```prolog
:- multifile fdbg:legend_portray/3.
fdbg:legend_portray(Name, Var, Set) :-
    fd_set(Var, Set0),
    fdset_to_list(Set0, L0),
    fdset_to_list(Set, L),
    ( L0 == L -> format('p = [p] = [p]', [Name, L])
    ; format('p = [p -> [p]', [Name, L0, L])
).
```

### 10.15.4.2 Writing Visualizers

For more complicated problems you might want to change the output more drastically. In this case you have to write and use your own visualizers, which could naturally be problem specific, not like `fdbg_show/2` and `fdbg_label_show/3`. As we described earlier, currently there are two types of visualizers:

**constraint visualizer**

```prolog
MyGlobalVisualizer([+Arg1, +Arg2, ...] +Constraint, +Actions)
```

This visualizer is passed in the `constraint_hook` option. It must take at least two arguments, the last two of which being:

- **Constraint**
  - the constraint that was handled by the dispatcher
- **Actions**
  - the action list returned by the dispatcher

Other arguments can be used for any purpose, for example to select the verbosity level of the visualizer. This way you do not have to modify your code if you would like to see less or more information. Note however, that the two obligatory arguments must appear at the end of the argument list.

When passing as an option to `fdbg_on/1`, only the optional arguments have to be specified; the two mandatory arguments should be omitted. See Section 10.15.4.6 [FDBG Debugging Global Constraints], page 551, for an example.

**labeling visualizer**

```prolog
MyLabelingVisualizer([+Arg1, +Arg2, ...] +Event, +ID, +Var)
```

This visualizer is passed in the `labeling_hook` option. It must have at least three arguments, the last three of which being:

- **Event**
  - a term representing the labeling event, can be one of the following:
    - start labeling has just started for a variable
    - fail labeling has just failed for a variable
The step(Step) variable has been constrained in a labeling step described by the compound term Step, which is either created by library(clpfd)'s labeling predicates (in this case, simply print it—FDBG will know how to handle it) or by you; see Section 10.15.3.5 [FDBG Annotation], page 539.

ID identifies the labeling session, i.e. binds step and fail events to the corresponding start event

Var the variable being the subject of labeling

The failure of a visualizer is ignored and multiple choices are cut by FDBG. Exceptions, on the other hand, are not caught.

FDBG provides several predicates to ease the work of the visualizer writers. These predicates are the following:

* fdbg_annotate(+Term0, -Term, -Variables)
  Replaces each constraint variable in Term0 by a compound term describing it and returns the result in Term. Also, collects these compound terms into the list Variables. These compound terms have the following form:

  fdvar(Name, Var, FDSet)

  Name is the name of the variable (auto-generated, if necessary; see Section 10.15.2.6 [FDBG Name Auto-Generation], page 534)

  Var is the variable itself

  FDS is the domain of the variable after narrowing with Actions, if specified; otherwise, it is the current domain of the variable

* fdbg_legend(+Vars)
  Prints a legend of Vars, which is a list of fdvar/3 compound terms returned by fdbg_annotate/[3,4].

* fdbg_legend(+Vars, +Actions)
  Prints a legend of Vars followed by some conclusions regarding the constraint (exiting, failing, etc.) based on Actions.

10.15.4.3 Writing Legend Printers

When you write your own visualizers, you might not be satisfied with the default format of the legend. Therefore you might want to write your own legend printer, replacing fdbg_annotate/[1,2]. This should be quite straightforward based on the variable list returned by fdbg_annotate/[3,4]. Processing the rest of the action list and writing conclusions about the constraint behavior is not that easy though. To help your work, FDBG provides a predicate to transform the raw action list to a more readable form:

* fdbg_transform_actions(+Actions, +Vars, -TransformedActions)
  This will do the following transformations to Actions, returning the result in TransformedActions:
1. remove all actions concerning variables in Vars (the list returned by \texttt{fdbg_annotate/[3,4]});
2. remove multiple \texttt{exit} and/or \texttt{fail} commands;
3. remove all ground actions, translating those that will cause failure into \texttt{fail(Action)};
4. substitute all other narrowings with an \texttt{fdvar/3} compound term per variable.

The transformed action list may contain the following terms:

\begin{itemize}
\item \texttt{exit} \hspace{2em} \text{the constraint exits}
\item \texttt{fail} \hspace{2em} \text{the constraint fails due to a \texttt{fail} action}
\item \texttt{fail(Action)} \hspace{2em} \text{the constraint fails because of \texttt{Action}}
\item \texttt{call(Goal)} \hspace{2em} \text{\texttt{Actions} originally contained this action. FDBG cannot do anything with that but to inform the user about it.}
\item \texttt{fdvar(Name, Var, FDSet)} \hspace{2em} \text{\texttt{Actions} also narrowed some variables that did not appear in the Vars list, this is one of them. The meaning of the arguments is the usual, described in Section 10.15.4.2 \textit{[FDBG Writing Visualizers]}, page 544. This should normally not happen.}
\end{itemize}

\textit{AnythingElse} \hspace{2em} \text{\texttt{Actions} contained unrecognized actions too, these are copied unmodified. This should not happen!}

10.15.4.4 Showing Selected Constraints (simple version)

Sometimes the programmer is not interested in every constraint, only some selected ones. Such a filter can be easily implemented with a user-defined visualizer. Suppose that you are interested in the constraints \texttt{all_different/[1,2]} and \texttt{all_distinct/[1,2]} only:

\begin{verbatim}
%% spec_filter(+Constraint, +Actions): Call fdbg_show for all constraints
%% for which interesting_event(Constraint) succeeds.
%%
%% Use this filter by giving the constraint_hook(spec_filter) option to
%% fdbg_on.
spec_filter(Constraint, Actions) :-
    interesting_event(Constraint),
    fdbg_show(Constraint, Actions).

interesting_event(all_different(_)).
interesting_event(all_different(_,_)).
interesting_event(all_distinct(_)).
interesting_event(all_distinct(_,_)).
\end{verbatim}
Here is a session using the visualizer. Note that the initialization part (domain/3 events), are filtered out, leaving only the all_distinct/[1,2] constraints:

```
| ?- [library('clpfd/examples/sudoku')].
[...]
| ?- fdbg_on([constraint_hook(spec_filter)]).
% The clp(fd) debugger is switched on
% advice
| ?- suudoku([], 1, domain).
all_distinct([1,<fdvar_1>,<fdvar_2>,8,<fdvar_3>,
        4,<fdvar_4>,<fdvar_5>,<fdvar_6>],[consistency(domain)])
    fdvar_1 = 1..9 -> (2..3)
    fdvar_2 = 1..9 -> (2..3)
    fdvar_3 = 1..9 -> (2..3)
    fdvar_4 = 1..9 -> (2..3)
    fdvar_5 = 1..9 -> (2..3)
    fdvar_6 = 1..9 -> (2..3)
[...]
all_distinct([7,6,2,5,8,4,1,3,9],[consistency(domain)])
Constraint exited.
1 5 6 8 9 4 3 2 7
9 2 8 7 3 1 4 5 6
4 7 3 2 6 5 9 1 8
3 6 2 4 1 7 8 9 5
7 8 9 3 5 2 6 4 1
5 1 4 9 8 6 2 7 3
8 3 1 5 4 9 7 6 2
6 9 7 1 2 3 5 8 4
2 4 5 6 7 8 1 3 9
yes
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off
```

Note that failure of spec_filter/2 does not cause any unwanted output.

**10.15.4.5 Showing Selected Constraints (advanced version)**

Suppose that you want to give the constraints that you are interested in as an argument to the visualizer, instead of defining them in a table. The following visualizer implements this.
%% filter_events(+CtrSpecs, +Constraint, +Actions): This predicate will
%% only show constraint events if they match an element in the list CtrSpecs,
%% or if CtrSpecs is wrapped in -/1, all the non-matching events will
%% be shown.
%% CtrSpecs can contain the following types of elements:
%% ctn_name               - matches all constraints of the given name
%% ctn_name/arity         - matches constraints with the given name and arity
%% ctn_name(...args...)  - matches constraints unifyable with the given term
%%
%% For the selected events fdbg_show(Constraint, Actions) is called.
%% This visualizer can be specified when turning fdbg on, e.g.:
%% fdbg_on([constraint_hook(filter_events([count/4]))]), or
%% fdbg_on([constraint_hook(filter_events([-[in_set]]))]).

filter_events(CtrSpecs, Constraint, Actions) :-
    filter_events(CtrSpecs, fdbg_show, Constraint, Actions).

filter_events(+CtrSpecs, +Visualizer, +Constraint, +Actions): Same as
the above predicate, but the extra argument Visualizer specifies the
predicate to be called for the selected events (in the same form as
in the constraint_hook option, i.e. without the last two arguments). E.g.
filter_events(-CtrSpecs, Visualizer, Constraint, Actions) :- !,
  \+ show_constraint(CtrSpecs, Constraint),
  call(Visualizer, Constraint, Actions).

filter_events(CtrSpecs, Visualizer, Constraint, Actions) :-
  show_constraint(CtrSpecs, Constraint),
  call(Visualizer, Constraint, Actions).

show_constraint([C|_], Constraint) :-
    matches(C, Constraint), !.

show_constraint([_|Cs], Constraint) :-
    show_constraint(Cs, Constraint).

matches(Name/Arity, Constraint) :- !,
    functor(Constraint, Name, Arity).

matches(Name, Constraint) :-
    atom(Name), !,
    functor(Constraint, Name, _).

matches(C, Constraint) :-
    C = Constraint.

Here is a session using the visualizer, filtering out everything but all_distinct/2 constraints:
| ?- [library('clpfd/examples/suudoku')].
| [...]  
| ?- fdbg_on([constraint_hook(filter_events([all_distinct/2]))]).  
% The clp(fd) debugger is switched on  
% advice  
| ?- suudoku([], 1, domain).
all_distinct([1,<fdvar_1>,<fdvar_2>,8,<fdvar_3>,
  4,<fdvar_4>,<fdvar_5>,<fdvar_6>],[consistency(domain)])
  fdvar_1 = 1..9 -> (2..3)/(5..7)/\{9}
  fdvar_2 = 1..9 -> (2..3)/(5..7)/\{9}
  fdvar_3 = 1..9 -> (2..3)/(5..7)/\{9}
  fdvar_4 = 1..9 -> (2..3)/(5..7)/\{9}
  fdvar_5 = 1..9 -> (2..3)/(5..7)/\{9}
  fdvar_6 = 1..9 -> (2..3)/(5..7)/\{9}

[...]

all_distinct([7,6,2,5,8,4,1,3,9],[consistency(domain)])
  Constraint exited.
  1 5 6 8 9 4 3 2 7
  9 2 8 7 3 1 4 5 6
  4 7 3 2 6 5 9 1 8
  3 6 2 4 1 7 8 9 5
  7 8 9 3 5 2 6 4 1
  5 1 4 9 8 6 2 7 3
  8 3 1 5 4 9 7 6 2
  6 9 7 1 2 3 5 8 4
  2 4 5 6 7 8 1 3 9
  yes
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off

In the next session, all constraints named all_distinct are ignored, irrespective of arity. Also, we explicitly specified the visualizer to be called for the events that are kept (here, we have written the default, fdbg_show, so the actual behavior is not changed).

In the last session, we specify a list of constraints to ignore, using a pattern to select the appropriate constraints. Since all constraints in the example match one of the items in the given list, no events are printed.
| ?- [library('clpfd/examples/sudoku')].
| [...]
| ?- fdbg_on([constraint_hook(filter_events(-
[domain(_,1,9),all_distinct]))]).
% The clp(fd) debugger is switched on
% advice
| ?- suudoku([], 1, domain).
1 5 6 8 9 4 3 2 7
9 2 8 7 3 1 4 5 6
4 7 3 2 6 5 9 1 8
3 6 2 4 1 7 8 9 5
7 8 9 3 5 2 6 4 1
5 1 4 9 8 6 2 7 3
8 3 1 5 4 9 7 6 2
6 9 7 1 2 3 5 8 4
2 4 5 6 7 8 1 3 9
yes
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off

10.15.4.6 Debugging Global Constraints

Missing pruning and excessive pruning are the two major classes of bugs in the implementation of global constraints. Since CLP(FD) is an incomplete constraint solver, missing pruning is mainly an efficiency concern (but *ground* instances for which the constraint does not hold should be rejected). Excessive pruning, however, means that some valid combinations of values are pruned away, leading to missing solutions. The following exported predicate helps spotting excessive pruning in user-defined global constraints:

\[
\text{fdbg\_guard}(:\text{Goal}, +\text{Constraint}, +\text{Actions})
\]

A constraint visualizer that does no output, but notifies the user by calling \text{Goal} if a solution is lost through domain narrowings. Naturally you have to inform \text{fdbg\_guard/3} about the solution in question—stating which variables should have which values. To use \text{fdbg\_guard/3}, first:

1. Set it up as a visualizer by calling:

\[
\text{fdbg\_on}([..., \text{constraint\_hook}(\text{fdbg\_guard}(\text{Goal})), ...])
\]

As usual, the two other arguments will be supplied by the FDBG core when calling \text{fdbg\_guard/3}.

2. At the beginning of your program, form a pair of lists \text{Xs-Vs} where \text{Xs} is the list of variables and \text{Vs} is the list of values in question. This pair should then be assigned the name \text{fdbg\_guard} using:

\[
| ?- \text{fdbg\_assign\_name}(:\text{Xs-Vs, fdbg\_guard}).
\]

When these steps have been taken, \text{fdbg\_guard/3} will watch the domain changes of \text{Xs} done by each constraint \text{C}. Whenever \text{Vs} is in the domains of \text{Xs} at entry to \text{C}, but not at exit from \text{C}, \text{Goal} is called with three more arguments:
Variable List

a list of Variable-Value terms for which Value was removed from
the domain of Variable

Constraint

the constraint that was handled by the dispatcher

Actions

the action list returned by the dispatcher

We will now show an example using fdbg_guard/3. First, we will need a few extra lines of code:

%%% print_and_trace(MissValues, Constraint, Actions): To be used as a Goal for
%%% fdbg_guard to call when the given solution was removed from the domains
%%% of the variables.
%%% MissValues is a list of Var-Value pairs, where Value is the value that
%%% should appear in the domain of Var, but has been removed. Constraint is
%%% the current constraint and Actions is the list of actions returned by it.
%%% This predicate prints MissValues in a textual form, then shows the current
%%% (culprit) constraint (as done by fdbg_show/2), then turns on the Prolog
%%% tracer.
print_and_trace(MissValues, Constraint, Actions) :-
    print(fdbg_output, '\nFDBG Guard:\n'),
    display_missing_values(MissValues),
    print(fdbg_output, '\nCulprit constraint:\n\n'),
    fdbg_show(Constraint, Actions),
    trace.

display_missing_values([]).
display_missing_values([Var-Val|MissVals]) :-
    fdbg_annotate(Var, AVar, _),
    format(fdbg_output, ' ~d was removed from ~p~n', [Val, AVar]),
    display_missing_values(MissVals).

Suppose that we have written the following N Queens program, using a global constraint no_threat/3 with a bug in it:
:- use_module(library(fdbg)).
:- use_module(library(clpfd)).

queens(L, N) :-
    length(L, N),
    domain(L, 1, N),
    constrain_all(L),
    labeling([enum], L).

constrain_all([]).
constrain_all([X|Xs]) :-
    constrain_between(X, Xs, 1),
    constrain_all(Xs).

constrain_between(_, [], _N).
constrain_between(X, [Y|Ys], N) :-
    no_threat(X, Y, N),
    N1 is N+1,
    constrain_between(X, Ys, N1).

no_threat(X, Y, I) :-
    fd_global(no_threat(X, Y, I), 0, [val(X), val(Y)]).

:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(no_threat(X, Y, I), S, S, Actions) :-
    ground(X), !,
    remove_threat(Y, X, I, NewYSet),
    Actions = [exit, Y in_set NewYSet].
clpfd:dispatch_global(no_threat(X, Y, I), S, S, Actions) :-
    ground(Y), !,
    remove_threat(X, Y, I, NewXSet),
    Actions = [exit, X in_set NewXSet].
clpfd:dispatch_global(no_threat(_, _, _), S, S, []).

remove_threat(X, V, I, Set) :-
    Vp is V+I+1, % Bug introduced here
%    Vp is V+I, % Good code
    Vn is V-I,
    fd_set(X, Set0),
    list_to_fdset([Vn, V, Vp], VSet),
    fdset_subtract(Set0, VSet, Set).

missing(L, Tuple) :-
    length(Tuple, N),
    length(L, N),
    domain(L, 1, N),
    lex_chain([[2, 4, 1, 3], L]),
    fdbg_assign_name(L-Tuple, fdbg_guard),
    fdbg_assign_name(L, board),
    fdbg_on([constraint_hook(fdbg_guard(print_and_trace))]),
    queens(L, N).
We will now use `print_and_trace/3` as an argument to the `fdbg_guard` visualizer to handle the case when a solution has been removed by a constraint. The bug shown above causes three invalid solutions to be found instead of the two correct solutions `[2,4,1,3]` and `[3,1,4,2]`. The constraint:

```
lex_chain([[2,4,1,3],L]),
```

constraints the search to solutions lexicographically greater than or equal to the first correct solution, and FDBG is told to watch for its disappearance. At some point, the buggy constraint removes it, and `fdbg_guard/3` calls the given predicate. This prints the cause of waking (the value that should not have been removed by the constraint), prints the constraint itself, then switches the Prolog debugger to trace mode. At this point, we type ‘A’ (see Section 10.15.3.4 [FDBG Debugger Commands], page 538) to print the annotated form of the goal containing the culprit constraint. Finally, we type ‘A [2,4]’ to print the same information, but taking into account the action list, which is the 4th argument of the 2nd argument of the module prefixed goal. For clarity, the labeling events were not turned off in the session below.

This example shows how FDBG can be used to narrow down what causes invalid pruning.
| ?- missing(L, [2,4,1,3]).
% The clp(fd) debugger is switched on
Labeling [8, <board_1>]: starting in range 2..4.
Labeling [8, <board_1>]: indomain_up: <board_1> = 2

FDBG Guard:
  4 was removed from <board_2>

Culprit constraint:

user:no_threat(2,<board_2>,1)
  board_2 = 1..4 -> {3}
Constraint exited.

% The debugger will first creep -- showing everything (trace)
11  2 Exit:
  clpfd:dispatch_global_fast(no_threat(2,_1511,1),0,0,
  [exit,_1511 in_set[[3|3]]],
  global('mutable'(0,0),no_threat(2,_1511,1),'mutable'(11,596),
  _10779,user:no_threat(2,_1511,1))) ? A

clpfd:dispatch_global_fast(no_threat(2,<board_2>,1),0,0,
  [exit,<board_2> in_set[[3|3]]],global('mutable'(0,0),no_threat(2,<board_2>,1),
  $mutable(11,596),<fdvar_1>,user:no_threat(2,<board_2>,1)))
  board_2 = 1..4
  fdvar_1 = inf..sup

11  2 Exit:
  clpfd:dispatch_global_fast(no_threat(2,_1511,1),0,0,
  [exit,_1511 in_set[[3|3]]],
  global('mutable'(0,0),no_threat(2,_1511,1),'mutable'(11,596),
  _23859,user:no_threat(2,_1511,1))) ? A [2,4]

clpfd:dispatch_global_fast(no_threat(2,<board_2>,1),0,0,
  [exit,<board_2> in_set[[3|3]]],global('mutable'(0,0),no_threat(2,<board_2>,1),
  $mutable(11,596),<fdvar_1>,user:no_threat(2,<board_2>,1)))
  board_2 = 1..4 -> {3}
  fdvar_1 = inf..sup
Constraint exited.

11  2 Exit:
  clpfd:dispatch_global_fast(no_threat(2,_1511,1),0,0,
  [exit,_1511 in_set[[3|3]]],
  global('mutable'(0,0),no_threat(2,_1511,1),'mutable'(11,596),
  _23859,user:no_threat(2,_1511,1))) ? a

% Execution aborted
% advice,source_info
| ?- fdbg_off.
% The clp(fd) debugger is switched off
10.16 Accessing Files And Directories—library(file_systems)

This module provides operations on files and directories, such as renaming, deleting, opening, checking permissions, accessing members of.

The following principles have been observed:

- An absolute distinction is drawn between files and directories. The set of operations one can usefully perform on a directory is different from the set one can perform on a file: for example, having write permission to a directory allows the user to create new files in it, not to rewrite the entire directory! If any routine in this package tells you that a “file” exists, you can be sure that it means a file and not a directory (and vice versa for “directory” exists).
- The directory scanning routines do not actually open the files they find. Thus finer discriminations, such as that between source and object code, are not made.
- All paths are expanded as if by absolute_file_name/3.
- Every predicate acts like a genuine logical relation insofar as it possibly can.
- If anything goes wrong, the predicates raise an error exception. Any time that a predicate fails quietly, it should mean “this question is meaningful, but the answer is no”.
- The directory scanning routines insist that the directory argument name a searchable directory.
- On Unix-like systems, symbolic links are followed by default and symbolic links that can not be followed are treated as non-existing. This means file_exists/1 will fail if passed such a “broken” link and that neither file_members_of_directory/1 nor directory_members_of_directory/1 et al. will return such a link.

On Windows, symbolic links (and other reparse points) are not followed when enumerating directory contents with file_members_of_directory/1 nor directory_members_of_directory/1 et al. and are not returned for these predicates.

The behavior for symbolic links (and reparse points) may change on all platforms in the future to ensure a well defined and consistent behavior on all platforms.

To see all members of a directory you can use absolute_file_name/3 with a glob('*') option.

The “property” routines use the same simplistic access control model as that used by the absolute_file_name/3 access/1-option. See Section 11.3.3 [mpg-ref-absolute_file_name], page 927, for details.

Exported predicates:

rename_file(+OldName, +NewName)

OldName must identify an existing file, which will be renamed to NewName. The details of just when this can be done are operating-system dependent. Typically it is only possible to rename within the same file system.
rename_directory(+OldName, +NewName)

*OldName* must identify an existing directory, which will be renamed to *NewName*. The details of just when this can be done are operating-system dependent. Typically it is only possible to rename empty directories within the same file system.

delete_file(+OldName)

*OldName* must identify an existing file, which will be deleted.

delete_directory(+Directory)
delete_directory(+Directory, +Options)

*Directory* must identify an existing directory, which will be deleted, if empty. *Options* should be a list of at most one term of the form:

if_nonempty(Value)

Defines what to do if the directory is nonempty. One of:

ignore The predicate simply succeeds, deleting nothing.
fail The predicate simply fails, deleting nothing.
error The predicate raises a permission error.
delete The predicate recursively deletes the directory and its contents.

directory_exists(+Directory)
directory_exists(+Directory, +Mode)

is true when *Directory* is an existing directory that is accessible according to *Mode*. *Mode* defaults to exist.

This is more or less equivalent to absolute_file_name(File, _, [file_type(directory),access([exist|Mode]),file_errors(fail)]).

make_directory(+Directory)

*Directory* is expanded, as if by absolute_file_name/3, and the resulting directory is created.

file_exists(+File)
file_exists(+File, +Mode)

is true when *File* is an existing file that is accessible according to *Mode*. *Mode* defaults to exist.

This is more or less equivalent to absolute_file_name(File, _, [access([exist|Mode]),file_errors(fail)]).

file_must_exist(+File)
file_must_exist(+File, +Mode)

is like file_exists(File[, Mode]) except that if the file is not accessible it reports an error.

This is more or less equivalent to absolute_file_name(File, _, [access([exist|Mode]),file_errors(error)]).

directory_must_exist(+File)
directory_must_exist(+File, +Mode)

is like file_must_exists(File[, Mode]), but for directories.
This is more or less equivalent to `absolute_file_name(File, _, [file_type(directory), access([exists|Mode]), file_errors(error)])`.

close_all_streams

closes all the streams (other than the standard streams) which are currently open. The time to call this is after an `abort/0`. Note that `current_stream/3` does not notice the standard streams.

directory_member_of_directory(?BaseName, ?FullName)

is true when `BaseName` is the name of a subdirectory of the current directory (other than '.' or '..') and `FullName` is its absolute name.

This uses `absolute_file_name/3` with the `glob/1` option.

directory_member_of_directory(+Directory, ?BaseName, ?FullName)

is true when `Directory` is a name (not necessarily the absolute name) of a directory, `BaseName` is the name of a subdirectory of that directory (other than '.' or '..') and `FullName` is its absolute name.

This uses `absolute_file_name/3` with the `glob/1` option.

directory_member_of_directory(+Directory, +Pattern, ?BaseName, ?FullName)

is true when `Directory` is a name (not necessarily the absolute name) of a directory, `BaseName` is the name of a directory of that directory (other than '.' or '..') which matches the given `Pattern`, and `FullName` is the absolute name of the subdirectory.

This uses `absolute_file_name/3` with a `glob(Pattern)` option.

directory_members_of_directory(-Set)

is true when `Set` is a set of `BaseName-FullName` pairs being the relative and absolute names of subdirectories of the current directory.

This uses `absolute_file_name/3` with the `glob/1` option.

directory_members_of_directory(+Directory, -Set)

is true when `Set` is a set of `BaseName-FullName` pairs being the relative and absolute names of subdirectories of the given `Directory`. `Directory` need not be absolute; the `FullNames` will be regardless.

This uses `absolute_file_name/3` with the `glob/1` option.

directory_members_of_directory(+Directory, +Pattern, -Set)

is true when `Set` is a set of `BaseName-FullName` pairs being the relative and absolute names of subdirectories of the given `Directory`, such that each `BaseName` matches the given `Pattern`.

This uses `absolute_file_name/3` with a `glob(Pattern)` option.

file_member_of_directory(?BaseName, ?FullName)

is true when `BaseName` is the name of a file in the current directory and `FullName` is its absolute name.

This uses `absolute_file_name/3` with the `glob/1` option.

file_member_of_directory(+Directory, ?BaseName, ?FullName)

is true when `Directory` is a name (not necessarily the absolute name) of a directory, `BaseName` is the name of a file in that directory, and `FullName` is its absolute name.
This uses `absolute_file_name/3` with the `glob/1` option.

`file_member_of_directory(+Directory, +Pattern, ?BaseName, ?FullName)`

is true when `Directory` is a name (not necessarily the absolute name) of a directory, `BaseName` is the name of a file in that directory which matches the given `Pattern`, and `FullName` is its absolute name.

This uses `absolute_file_name/3` with a `glob(Pattern)` option.

`file_members_of_directory(-Set)`

is true when `Set` is a set of `BaseName-FullName` pairs being the relative and absolute names of the files in the current directory.

This uses `absolute_file_name/3` with the `glob/1` option.

`file_members_of_directory(+Directory, -Set)`

is true when `Set` is a set of `BaseName-FullName` pairs being the relative and absolute names of the files in the given `Directory`. `Directory` need not be absolute; the `FullNames` will be regardless.

This uses `absolute_file_name/3` with the `glob/1` option.

`file_members_of_directory(+Directory, +Pattern, -Set)`

is true when `Set` is a set of `BaseName-FullName` pairs being the relative and absolute names of subdirectories of the given `Directory`, such that each `BaseName` matches the given `Pattern`.

This uses `absolute_file_name/3` with a `glob(Pattern)` option.

`directory_property(+Directory, ?Property)`

is true when `Directory` is a name of a directory, and `Property` is a boolean property which that directory possesses, e.g.

```
directory_property(., searchable).
```

The current set of file and directory properties include:

- `readable`
- `writable`
- `executable`
- `searchable`

Tries to determine whether the process has permission to read, write, execute (only for files) or search (only for directories) the file.

- `size_in_bytes`
  The size, in bytes, of the file. Not available for directories.

- `create_timestamp`
- `modify_timestamp`
- `access_timestamp`

The time of creation, last modification or last access expressed as a timestamp. A `timestamp` is an integer expressing the time interval, in seconds, since the “Epoch”. The `Epoch` is the time zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time (UTC).
The timestamp is what should be used when comparing information between files since it is independent of locale issues like time zone and daylight savings time etc.

```
create_localtime
modify_localtime
access_localtime
```

The same as the corresponding ..._timestamp values passed through system:datetime/2, i.e. expressed as local time and split up in the components year, month, day, hour, minute, seconds.

```
set_user_id
set_group_id
save_text
```

True if the set-uid, set-group-id, save-text bits, respectively, are set for the file. Always false on Windows.

```
who_can_read
who_can_write
who_can_execute
who_can_search
```

A list containing the subset of [user, group, other] for the process classes that can, respectively, read, write, execute (only for files) or search (only for directories.

```
owner_user_id
owner_group_id
```

The id of the owner and group of the file. The id is an integer on UNIX and an atom (expressed as a string security identifier) on Windows.

```
owner_user_name
owner_group_group
```

The atom containing the name of the files owner and group respectively. On Windows a name like 'DOMAIN\NAME' will be used.

If for some reason the name cannot be found it will fall back to using the same value as owner_user_id and owner_group_id.

Other properties may be added in the future. You can backtrack through the available properties by calling file_property/3 or directory_property/3 with an uninstantiated Property argument.

```
directory_property(+Directory, ?Property, ?Value)
```

is true when Directory is a name of a directory, Property is a property of directories, and Value is Directory’s Property Value. See directory_property/2, above, for a list of properties.

```
file_property(+File, ?Property)
```

is true when File is a name of a file, and Property is a boolean property which that file possesses, e.g.

```
file_property('foo.txt', readable).
```
See `directory_property/2`, above, for a list of properties.

```prolog
file_property(+File, ?Property, ?Value)
```

is true when `File` is a name of a file, `Property` is a property of files, and `Value` is `File`'s `Property` Value. See `directory_property/2`, above, for a list of properties.

```prolog
current_directory(-Directory)
current_directory(-Directory, +NewDirectory)
```

`Directory` is unified with the current working directory and the working directory is set to `NewDirectory`.

### 10.17 The Gauge Profiling Tool—library(gauge)

The Gauge library package is a graphical interface to the SICStus built-in predicates `profile_data/1` and `profile_reset/0`. See Section 9.2 [Execution Profiling], page 355, for more information about execution profiling. The interface is based on Tcl/Tk (see Section 10.45 [lib-tcltk], page 769).

The SICStus IDE (see Section 3.11 [SPIDER], page 29) can also show profiling information. This makes the Gauge library largely obsolescent.

```prolog
view
```

*since release 4.2*

Creates a graphical user interface for viewing the profile data accumulated so far. When the display first comes up it is blank except for the control panel. A screen shot is shown below.
The menus and buttons on the control panel are used as follows:

**Specification**

- **Calls**: The number of times a predicate was called.

**Instructions**

The number of abstract instructions executed, plus two times the number of choice point accesses.

**Choicepoints**

Number of choicepoints accessed (saved or restored).

**Sort Order**

Selects the sort order of the histogram. One of:
Alphabetic  Sort the bars in alphabetic order.

Descending values  Sort the bars by descending values.

Ascending values  Sort the bars by ascending values.

Top 40  Show just the 40 highest values in descending order.

Scale  Controls the scaling of the bars. One of:

    Linear  Display values with a linear scale.

    Logarithmic  Display values with a logarithmic scale.

Show  Controls whether to show bars with zero counts. One of:

    All  Show all values in the histogram.

    No zero values  Show only non-zero values.

Font  The font used in the histogram chart.

Calculate  Calculates the values according to the current settings. The values are displayed in a histogram.

Reset  The execution counters of the selected predicates and clauses are reset.

Print  A choice of printing the histogram on a Postscript printer, or to a file.

Help  Shows a help text.

Quit  Quits Gauge and closes its windows.

By clicking on the bars of the histogram, the figures are displayed in the Value Info window.

10.18 Heap Operations—library(heaps)

A heap is a labeled binary tree where the key of each node is less than or equal to the keys of its sons. The point of a heap is that we can keep on adding new elements to the heap and we can keep on taking out the minimum element. If there are \( N \) elements total, the total time is \( O(N \lg N) \). If you know all the elements in advance, you are better off doing a merge-sort, but this file is for when you want to do say a best-first search, and have no idea when you start how many elements there will be, let alone what they are.

A heap is represented as a triple \( \text{heap}(N, \text{Free}, \text{Tree}) \) where \( N \) is the number of elements in the tree, \( \text{Free} \) is a list of integers which specifies unused positions in the tree, and \( \text{Tree} \) is a tree made of:

\( \text{heap} \)  terms for empty subtrees and

\( \text{heap}(\text{Key}, \text{Datum}, \text{Lson}, \text{Rson}) \)  terms for the rest
The nodes of the tree are notionally numbered like this:

```
     1
    / \  
   2   3
  /   / \
 4   6   5
 /   /   / \
8   12  10  14
```

The idea is that if the maximum number of elements that have been in the heap so far is \( M \), and the tree currently has \( K \) elements, the tree is some subtree of the tree of this form having exactly \( M \) elements, and the Free list is a list of \( M-K \) integers saying which of the positions in the \( M \)-element tree are currently unoccupied. This free list is needed to ensure that the cost of passing \( N \) elements through the heap is \( O(N \lg M) \) instead of \( O(N \lg N) \). For \( M \) say 100 and \( N \) say \( 10^4 \) this means a factor of two. The cost of the free list is slight.

The storage cost of a heap in a copying Prolog is \( 2K+3M \) words. Exported predicates:

```
add_to_heap(+OldHeap, +Key, +Datum, -NewHeap)
add_to_heap/4 (heaps)

inserts the new Key-Datum pair into the heap. The insertion is not stable, that is, if you insert several pairs with the same Key it is not defined which of them will come out first, and it is possible for any of them to come out first depending on the history of the heap.
```

```
delete_from_heap(+OldHeap, +Key, -Datum, -NewHeap)
delete_from_heap/4 (heaps)

deletes a single Key-Datum pair from the OldHeap producing a NewHeap. This is useful if you want to e.g. change the priority of Datum.
```

```
get_from_heap(+OldHeap, -Key, -Datum, -NewHeap)
get_from_heap/4 (heaps)

returns the Key-Datum pair in OldHeap with the smallest Key, and also a NewHeap which is the OldHeap with that pair deleted.
```

```
heap_size(+Heap, -Size)
heap_size/2 (heaps)

reports the number of elements currently in the heap.
```

```
heap_to_list(+Heap, -List)
heap_to_list/2 (heaps)

returns the current set of Key-Datum pairs in the Heap as a List, sorted into ascending order of Keys.
```

```
list_to_heap(+List, -Heap)
list_to_heap/2 (heaps)

takes a proper list of Key-Datum pairs (such as keysort/2 could be used to sort) and forms them into a heap.
```

```
empty_heap(?Heap)
empty_heap/1 (heaps)

is true when Heap represents an empty heap. There is only one way it can be true.

is_heap(+Heap)
is_heap/1 (heaps)
is true when Heap is a well formed heap. For this to be true, the size must be right and the tree must satisfy the heap condition.

min_of_heap(+Heap, -Key, -Datum)
min_of_heap/3 (heaps)
returns the Key-Datum pair at the top of the heap (which is of course the pair with the smallest Key), but does not remove it from the heap. It fails if the heap is empty.

min_of_heap(+Heap, -Key1, -Datum1, -Key2, -Datum2)
min_of_heap/5 (heaps)
returns the smallest (Key1) and second smallest (Key2) pairs in the heap, without deleting them. It fails if the heap does not have at least two elements.

portray_heap(+Heap)
portray_heap/1 (heaps)
writes a heap to the current output stream in a pretty format so that you can easily see what it is. Note that a heap written out this way can not be read back in. The point of this predicate is that you can add a clause

portray(X) :- is_heap(X), !, portray_heap(X).

10.19 Declaring determinacy attributes—library(is_directives)

This library module gives access to the information declared by is/2 directives. The is/2 declarations can be used for declaring arbitrary predicate attributes, but the main application is for declaring determinacy information, and this is what is described here.

10.19.1 Introduction

Determinacy, i.e. whether a call to a predicate can produce more than one solution on backtracking, is an important property for understanding the predicate and the code that uses it. For this reason it is desirable to describe the behavior in the documentation that accompanies a predicate.

The determinacy properties of a predicate can also be used by various tools, and for this reason it is possible to declare the determinacy of a predicate in a way that can be automatically processed by such tools. One example is the determinacy analyzer in the SPIDER IDE (see Section 3.11 [SPIDER], page 29), which uses the declared determinacy of a called predicate to improve the precision of the analysis of the caller in the common case that the caller does not have any determinacy declaration of its own.

The SICStus libraries contain determinacy declarations for most exported predicates. These can serve as useful examples.

Determinacy declarations are hints, i.e. they are meant to convey the expected behavior. Due to the dynamic nature of Prolog, almost any call can fail, throw an exception, or succeed more than once for reasons not directly related to the called predicate. Examples
where these things can happen are timeouts (library(timeout)) and functionality that can cause goals to run when a variable is bound, e.g. \texttt{freeze/2}. For this reason, anything that uses determinacy declarations must be prepared to handle any run-time behavior, not just the behavior specified by the declarations.

10.19.2 Available Determinacy Annotations

The following determinacy attributes are available:

- **det** A call will always succeed exactly once. I.e. the number of solutions is expected to be one. This is probably the most common behavior.

- **semidet** A call will fail, or succeed exactly once. I.e. the number of solutions is expected to be zero or one. This can be used for for describing tests, e.g. \texttt{X>Y}.

- **multi** A call will succeed more than once. I.e. the number of solutions is expected to be one or more. This is uncommon, but could be used to describe the builtin \texttt{repeat/0}.

- **nondet** A call may fail or succeed any number of times. I.e. the number of solutions is expected to be zero or more. This is the most general determinacy information and is what must be assumed unless no other information is available.

- **failing** A call will fail. I.e. the number of solutions is expected to be zero. This is uncommon, but could be used to describe the builtin \texttt{false/0}.

- **throwing** A call is expected to throw an exception. This is similar to \texttt{failing} in that the number of solutions is expected to be zero but differs in situations where failure is treated specially, like if-then-else. This is uncommon, but could be used to describe the builtin \texttt{throw/1} and some of the error reporting predicates in library(types).

10.19.3 Syntax of Determinacy Declarations

Determinacy is declared using directives that use \texttt{is/2}. Note that in this usage, \texttt{is/2} has nothing to do with the arithmetic predicate of the same name.

The format of a determinacy directive is:

\[
:- \text{SPEC is } \text{ANNOTATION}.
\]

where \text{ANNOTATION} is one of the atoms used for determinacy annotation. The \text{SPEC} describes the predicate and can be either a predicate specification (like those used by the \texttt{abolish/1} predicate) or a skeletal goal (like those used by the \texttt{meta_predicate/1} directive).

Example:
\% foo/2 is expected to always succeed, once, for any argument.
:- foo/2 is det.

foo(X, Y) :-
    Y = hello(X).

\% bar/2, when called with a non-variable first argument,
\% is expected to succeed at most once.
:- bar(+, ?) is semidet.

\% bar/2, when called with a variable first argument,
\% is expected to succeed any number of times.
:- bar(-, ?) is nondet.

bar('a', lowercase).
bar('A', uppercase).
...
bar('z', lowercase).
bar('Z', uppercase).

The SPEC tells which predicate is being annotated and, if it is a skeletal goal, it can restrict
the declaration to only apply for the specified instantiation of the arguments. It can have
one of the following forms:

Module:Name/Arity
Name/Arity

Module and Name should be atoms, Arity should be a non-negative integer.
This predicate specification denotes the predicate Name with arity Arity in
module Module, where Module defaults to the source module.

Module:Name(ARG1, ARG2, ..., ARGN)
Name(ARG1, ARG2, ..., ARGN)

This skeletal goal denotes the predicate Name with arity N in the module
Module, where Module defaults to the source module. The argument positions
is typically used to indicate an instantiation pattern.

10.19.3.1 Specifying Instantiation Patterns

Many predicates have different determinacy depending on how their arguments are instanti-
ated. This can be indicated using the skeleton goal form of specification, with each argument
of the skeleton goal being one of the following:

+ Indicates that the argument is instantiated, i.e. it is not a variable when the
  predicate is called.

- Indicates that the argument is uninstantiated, i.e. it is a variable when the
  predicate is called.
?  * Indicates that the argument can be anything, i.e. the declared determinism is not affected by the instantiation of this argument

if the argument is a compound term with one argument (like + hello, or - Bar), only the functor is used when interpreting the argument. This makes it possible to use variables as descriptive names for the arguments, e.g.

    :- parent_of(+Parent, -Child) is nondet.

Not only variables can be used as descriptions in this way, any term is accepted.

When declaring determinism, the skeleton argument only specifies whether an argument is a variable or not. This is different from whether the argument should be considered input or output (var(X) has only an input argument, but will often be called with a variable as input argument).

### 10.19.3.2 Declaring Meta Predicate Determinacy

The declarations above is sufficient for most predicates. However, they do not suffice for predicates whose determinism depends on an argument goal, like lists:maplist/2.

For such predicates, i.e. meta predicates that take a single closure (goal) argument, it is possible to specify different determinacy for each of the possible determinacies of the closure argument, as in the following example:
% The last argument of dolist/3 is a closure with two
% suppressed arguments that will be supplied using call/3.
:- meta_predicate dolist(*, *, 2).

% This is the expected mode, a determinate producer. In this
% case dolist/3 will also succeed exactly once.
:- dolist(+, -, det) is det.

% If the closure can fail, then dolist/2 can also fail.
:- dolist(+, -, semidet) is semidet.

% If the closure succeeds more than once, then so will dolist/2.
:- dolist(+, -, multi) is multi.

% If the closure always fails (a strange usage, indeed) then dolist/3
% can nevertheless succeed, when the input is an empty list.
:- dolist(+, -, failing) is semidet.

% In general, dolist/3 will be nondeterminate if the closure is
% nondeterminate.
% This declaration may seem redundant, but it may not be for some tools.
:- dolist(+, -, nondet) is nondet.

% dolist/3 calls the argument closure, which expects two extra
% arguments, on each pair of corresponding list elements
dolist([], [], _G_2). % The closure is ignored here
dolist([X|Xs], [Y|Ys], G_2) :-
    call(G_2, X, Y),
    dolist(Xs, Ys, G_2).

:- square/2 is det. % always expected to succeed (once)
square(X, XX) :-
    XX is X*X.

% Example use:
% | ?- square_list([1,2,3], Squares).
% Squares = [1,4,9] ?
%
% It can be inferred that this is expected to succeed exactly once.
square_list(Numbers, Squares) :-
    % Calls square(X,Y) on each X in Numbers and Y in Squares.
    % Since square/2 is 'det', the call do dolist/3 will
    % be considered 'det' as well.
    dolist(Numbers, Squares, square).
10.19.4 Using Determinacy Declarations

Since determinacy declarations by necessity are only hints, it is often better to focus on the expected behavior rather than the exact behavior when declaring determinism for a predicate.

Also, only very simple instantiation patterns can be specified, so it may be useful to pretend they indicate more than they actually do.

As an example, consider how to declare the determinacy of the builtin length(List, Length):

1. It will succeed at most once if the second argument is instantiated (It will succeed once if the first argument can be unified with a list of the specified length, otherwise it will fail), regardless of the instantiation of the first argument. This corresponds to the attribute semidet.
2. It will succeed exactly once if the first argument is a proper list and the second argument is a variable. This corresponds to the attribute det.
3. It will succeed more than once if the first argument is a partial list (i.e. is a variable or has a variable tail). This corresponds to the attribute multi.
4. Finally, it will throw an exception for some invalid inputs, but this is not specified with determinacy declarations (... is throwing is only meant for predicates expected to always throw an exception).

However, it is not possible to specify “proper list”, “partial list” and non-list as instantiation patterns. On the other hand, it would be unfortunate to not be able to say anything about the determinacy of length/2.

In such cases it may make sense to pretend that ‘+’, the non-variable argument instantiation, means “a properly/fully instantiated input”, i.e. a “proper list” in the case of length/2. Similarly, it may make sense to pretend that ‘?’, any instantiation, means “a partially instantiated input”, i.e. a “partial list” in the case of length/2.

So, for the builtin length/2 it would make sense to specify the following determinacy declarations:

    :- length(*, +) is semidet.  % this is precise
    :- length(+, -) is det.    % pretend '+' means proper list
    :- length(?,-) is multi.   % pretend '?' means partial list

It is up to the documentation accompanying the predicate, and any tools that use these declarations, to handle this appropriately.

10.19.5 Accessing Determinacy Declarations at Runtime

The determinacy declarations are saved when code is compiled or consulted and can be accessed when the code has been loaded. This could be used by documentation generators, smart debuggers, and many other purposes, not all of which documented. New uses may
be added without notice, so you should ignore any recorded \texttt{is/2} directive that you do not understand.

The loaded \texttt{is/2} directives can be accessed using the following, non-determinate, predicate:

\begin{verbatim}
current_is_directive(Skel, M, Annotation, Spec, Directive, Context).
\end{verbatim}

- \textit{Skel} This is a compound term with the same name as the predicate and one anonymous variable in each argument position.

- \textit{M} This is the module of the predicate specification. Typically the same as the source module.

- \textit{Annotation} This is the second argument of the \texttt{is/2} directive.

- \textit{Spec} This is the first argument of the \texttt{is/2} directive, with module prefixes peeled off.

- \textit{Directive} This is the entire \texttt{is/2} directive (without the surrounding :- ...).

- \textit{Context} this is the source module.

Consider the following code:

\begin{verbatim}
:- module(example, [p1/3]).

:- p1/3 is det.
:- user:bar(+, +) is semidet.
\end{verbatim}

This corresponds to the following two clauses of \texttt{current_is_directive/6}:

\begin{verbatim}
current_is_directive(p1(_,_,_), example, det, p1/3, (p1/3 is det), example).
current_is_directive(bar(_,_), user, semidet, bar(+,+), (user:bar(+,+) is semidet), example).
\end{verbatim}

Exported predicates:

\begin{verbatim}
current_is_directive(Skel, M, Annotation, Spec, Directive, Context).
\end{verbatim}

Low-level access to the information recorded by \texttt{is/2} directives. See the library documentation for details.
current_is_directive(:MSkel, Annotation, Spec).
    Like current_is_directive(Skel, M, Annotation, Spec, _, _) where M and Skel are the parts of the meta argument MSkel.

10.20 Jasper Interface—library(jasper)

10.20.1 Jasper Overview

Jasper is a bi-directional interface between Java and SICStus. The Java side of the interface consists of a Java package (se.sics.jasper) containing classes representing the SICStus runtime system (SICStus, SPTerm, etc). The Prolog part is designed as a library module (library(jasper)).

The library module library(jasper) (see Section 10.20.8 [The Jasper Library], page 587) provides functionality for controlling the loading and unloading the JVM (Java Virtual Machine), method call functionality (jasper_call/4), and predicates for managing object references.

Jasper can be used in two modes, depending on which system acts as Parent Application. If Java is the parent application, then the SICStus runtime kernel will be loaded into the JVM using the System.loadLibrary() method (this is done indirectly when instantiating a SICStus object). In this mode, SICStus is loaded as a runtime system (see Section 6.7.1 [Runtime Systems], page 319).

As of release 3.9, it is possible to use Jasper in multi threaded mode. This means that several Java threads can call SICStus runtime via a server thread. The communication between the client threads and the server thread is hidden from the programmer, and the API is based on Java Interfaces, which are implemented both by the multi thread capable classes and the pre-3.9 classes, which are restricted to single threaded mode. The decision whether to run in single thread mode or in multi threaded mode can thus be left until runtime.

If SICStus is the parent application, then Java will be loaded as a foreign resource using the query use_module(library(jasper)). The Java engine is initialized using jasper_initialize/[1,2].

- Some of the information in this chapter is a recapitulation of the information in Chapter 6 [Mixing C and Prolog], page 289. The intention is that this chapter should be possible to read fairly independently.
- Before proceeding, please read Section “Jasper Notes” in SICStus Prolog Release Notes. It contains important information about requirements, availability, installation tips, limitations, and how to access other (online) Jasper/Java resources.

10.20.2 Getting Started

See Section “Getting Started” in SICStus Prolog Release Notes for a detailed description of how to get started using the interface. It addresses issues such as finding SICStus from within Java and vice versa, setting the classpath correctly, etc. If you have trouble in getting started with Jasper, then read that chapter before contacting SICStus Support.
10.20.3 Calling Prolog from Java

Calling Prolog from Java is done by using the Java package `se.sics.jasper`. This package contains a set of Java classes, which can be used to create and manipulate terms, ask queries and request one or more solutions. The functionality provided by this set of classes is basically the same as the functionality provided by the C-Prolog interface (see Chapter 6 [Mixing C and Prolog], page 289).

It is possible to debug the Prolog code using the Prolog debugger, either from the command line or from the SICStus Prolog IDE (SPIDER), see Section 6.9.3 [Examples of Debugging Runtime Systems], page 344.

The usage is easiest described by some examples.

10.20.3.1 Single Threaded Example

The following is a Java version of the `train` example. See Section 6.8.1 [Train Example], page 331, for information about how the `train.sav` file is created.

This code demonstrates the use of Jasper in single threaded mode. In this mode only one thread can access the SICStus runtime via a SICStus object.
import se.sics.jasper.SICStus;
import se.sics.jasper.Query;
import java.util.HashMap;

public class Simple
{
    public static void main(String argv[]) {

        SICStus sp;
        Query query;
        HashMap WayMap = new HashMap();

        try {
            sp = new SICStus(argv,null);
            sp.restore("train.sav");

            query = sp.openPrologQuery("connected('¨Orebro', 'Stockholm',
                Way, Way).",
                WayMap);

            try {
                while (query.nextSolution()) {
                    System.out.println(WayMap);
                }
            } finally {
                query.close();
            }
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}

It is assumed that the reader has read the section on Section 10.20.2 [Getting Started], page 572, which describes how to get the basics up and running.

This is how the example works:

1. Before any predicates can be called, the SICStus runtime system must be initialized. This is done by instantiating the SICStus class. Each SICStus object correspond to one independent copy of the SICStus runtime system (a rather heavy-weight entity). In this example, we have specified null as the second argument to SICStus. This instructs SICStus to search for sprt.sav using its own internal methods.

2. Queries are made through method query. The arguments to this method are a string
specifying a Prolog goal, and a Map, which will contain a mapping of variable names to bindings. This method is for finding a single solution. Note that the string is read by the Prolog reader, so it must conform to the syntax rules for Prolog, including the terminating period. There are two more methods for making queries: queryCutFail, for side effects only, and openQuery to produce several solutions through backtracking.

3. The next step is to load the Prolog code. This is done by the method restore. Corresponds to SP_restore() in the C-interface. See Section 6.7.4.2 [Loading Prolog Code], page 331. Note that this method must be called before any other SICStus method is called. See the HTML Jasper documentation for details.

4. The openQuery method returns a reference to a query, an object implementing the Query interface. To obtain solutions, the method nextSolution is called with no arguments. nextSolution returns true as long as there are more solutions, and the example above will print the value of the Map WayMap until there are no more solutions. Note that the query must be closed, even if nextSolution has indicated that there are no more solutions.

10.20.3.2 Multi Threaded Example
Following is a Java version of the train example.

This is a multi threaded version of the train example. In this mode several threads can access the SICStus runtime via a Prolog interface. The static method Jasper.newProlog() returns an object that implements a Prolog interface. A thread can make queries by calling the query-methods of the Prolog object. The calls will be sent to a separate server thread that does the actual call to SICStus runtime.
import se.sics.jasper.Jasper;
import se.sics.jasper.Query;
import se.sics.jasper.Prolog;
import java.util.HashMap;

public class MultiSimple
{
    class Client extends Thread
    {
        Prolog jp;
        String qs;

        Client(Prolog p, String queryString)
        {
            jp = p;
            qs = queryString;
        }
        public void run()
        {
            HashMap WayMap = new HashMap();
            try {
                synchronized(jp) {
                    Query query = jp.openPrologQuery(qs, WayMap);
                    try {
                        while (query.nextSolution()) {
                            System.out.println(WayMap);
                        }
                    } finally {
                        query.close();
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
{  
  try {
    Prolog jp = Jasper.newProlog(argv,null,"train.sav");

    Client c1 =
        new Client(jp,"connected('¨Orebro', 'Hallsberg',
            Way1, Way1.");
    c1.start();
    // The prolog variable names are different from above
    // so we can tell which query gives what solution.
    Client c2 =
        new Client(jp,"connected('Stockholm', 'Hallsberg',
            Way2, Way2.");
    c2.start();
  }  
  catch ( Exception e ) {
    e.printStackTrace();
  }
}

public static void main(String argv[])
{
    new MultiSimple(argv);
}

1. The Prolog object jp is the interface to SICStus. It implements the methods of interface Prolog, making it possible to write quite similar code for single threaded and multi threaded usage of Jasper. The static method Jasper.newProlog() returns such an object.

2. In this example, the Prolog code is loaded by the server thread just after creating the SICStus object (which is invisible to the user). The third argument to the method Jasper.newProlog is the .sav file to restore. Two threads are then started, which will make different queries with the connected predicate.

3. openPrologQuery is not recommended in multi threaded mode, but if you must use it from more than one Java thread, then you should enclose the call to openPrologQuery and the closing of the query in a single synchronized block, synchronizing on the Prolog object. See Section 10.20.6 [SPTerm and Memory], page 584, for details on one of the reasons why this is necessary.

10.20.3.3 Another Multi Threaded Example (Prolog Top Level)

This is another multi threaded version of the train example (see Section 6.8.1 [Train Example], page 331).

In this example, Prolog is the top level and Java is invoked via library(jasper).
import se.sics.jasper.Jasper;
import se.sics.jasper.Query;
import se.sics.jasper.Prolog;
import se.sics.jasper.SICStus;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.ListIterator;

public class MultiSimple2
{
    class Client extends Thread
    {
        Prolog jp;
        SICStus sp;
        String qs;

        Client(Prolog p, SICStus s, String queryString)
        {
            jp = p;
            sp = s;
            qs = queryString;
        }
        public void run()
        {
            HashMap WayMap = new HashMap();
            try {
                synchronized(jp) {
                    Query query = jp.openPrologQuery(qs, WayMap);
                    try {
                        while (query.nextSolution()) {
                            System.out.println(WayMap);
                        }
                    } finally {
                        query.close();
                    }
                }
            } catch ( Exception e ) {
                e.printStackTrace();
            }
        }
    }
    public class MultiSimple2
    {
        class Client extends Thread
        {
            Prolog jp;
            SICStus sp;
            String qs;

            Client(Prolog p, SICStus s, String queryString)
            {
                jp = p;
                sp = s;
                qs = queryString;
            }
            public void run()
            {
                HashMap WayMap = new HashMap();
                try {
                    synchronized(jp) {
                        Query query = jp.openPrologQuery(qs, WayMap);
                        try {
                            while (query.nextSolution()) {
                                System.out.println(WayMap);
                            }
                        } finally {
                            query.close();
                        }
                    }
                } catch ( Exception e ) {
                    e.printStackTrace();
                }
            }
        }
    }
class Watcher extends Thread
{
    SICStus mySp;
    ArrayList threadList = new ArrayList(2);

    public boolean add(Client cl)
    {
        return threadList.add((Object)cl);
    }

    boolean at_least_one_is_alive(ArrayList tl)
    {
        ListIterator li = tl.listIterator();
        boolean f = false;
        while (li.hasNext()) {
            boolean alive = ((Client)(li.next())).isAlive();
            f = f || alive;
        }
        return f;
    }

    public void run()
    {
        while (at_least_one_is_alive(threadList)) {
            try {
                this.sleep(1000);
            } catch (InterruptedException ie) {
                System.err.println("Watcher interrupted.");
            }
        }
        mySp.stopServer();
    }

    Watcher(SICStus sp)
    {
        mySp = sp;
    }
}
public void CallBack()
{
    try {
        SICStus sp = SICStus.getCaller(); // get the SICStus object
        sp.load("train.ql");
        Prolog jp = sp.newProlog(); // Create a new Prolog Interface
        Client c1 =
            new Client(jp, sp,
                      "connected('Örebro', 'Hallsberg', Way1, Way1)." );
        c1.start();
        // The prolog variable names in the Map are different from above so
        // we can tell which query gives what solution.
        Client c2 =
            new Client(jp, sp,
                      "connected('Stockholm', 'Hallsberg', Way2, Way2)." );
        c2.start();
        Watcher w = new Watcher(sp);
        w.add(c1);
        w.add(c2);
        w.start();
        sp.startServer(); // And finally start the server. This
        // method call does not return until
        // some other thread calls sp.stopServer().
    }
    catch ( Exception e ) {
        e.printStackTrace();
    }
}

% multisimple2.pl
:- use_module(library(jasper)).
main:-
    jasper_initialize(JVM),
    jasper_new_object(JVM,
                      'MultiSimple2',
                      init,
                      init,
                      Obj),
    jasper_call(JVM,
                method('', 'CallBack', [instance]),
                'CallBack'(+object(''),
                           'CallBack'(Obj)).

1. This example is similar to the previous multi threaded example See Section 10.20.3.2
2. Since a SICStus object already exists when the java method CallBack is called, we cannot use Jasper.newProlog to obtain a Prolog interface. Instead we can use the SICStus method getCaller to get a handle on the SICStus object.

3. In this example we cannot use the restore method to load the Prolog saved state, since it unloads all foreign resources. This includes library(jasper) from which the call to Java was made. Instead the method SICStus.load can be used to load a compiled Prolog file. See the HTML Jasper documentation for details on this method. See Section 4.3.2 [ref-lod-lod], page 80, for how to create a ‘.ql’ file.

4. The rest of the example is similar to the previous multi threaded example with the addition of a watcher class, which is used to monitor the client threads. This is necessary if the method startServer is to return. See the HTML Jasper documentation on the methods SICStus.startServer and SICStus.stopServer.

10.20.4 Jasper Package Class Reference

Detailed documentation of the classes in the jasper package can be found in the HTML documentation installed with SICStus and also on the SICStus documentation page (https://sicstus.sics.se/documentation.html).

Please note: None of the se.sics.jasper methods are thread safe, unless explicitly mentioned, they can only be called from the thread that created the SICStus object. (This is different from how se.sics.jasper worked in release 3.8.)

As of release 3.9, Jasper supports multi threaded mode. Several Java threads can access SICStus runtime through a server thread that does the actual calls.

The API is defined by three interfaces: Prolog, Query and Term. The methods of these interfaces are implemented by inner classes of the Jasper server. Instances of these inner classes are returned by methods of the class Jasper and can then be used from multiple threads by the Java programmer.

In multi threaded mode the Java programmer obtains an object implementing the interface Prolog. That interface has methods similar to the methods of the SICStus class described below. Interface Query and interface Term have the same relations to class SPQuery and class SPTerm, respectively. In addition the SICStus class, the SPQuery class and the SPTerm class all implement the above interfaces. The methods of the interfaces are preferred over the old methods.

See the HTML documentation for details on the methods of the interfaces.

boolean query (String module, String name, SPTerm args[]) [Method on SICStus]

Call name with args (a vector of SPTerm objects). Like once(Module:Name(Args...)).

Returns true if the call succeeded, false if the call failed, i.e. there were no solutions.

Introduced in release 3.8.5.

boolean query (String goal, Map varMap) [Method on SICStus]

Call a goal specified as a string.

goal The textual representation of the goal to execute, with terminating period.

varMap A map from variable names to SPTerm objects. Used both for passing variable bindings into the goal and to obtain the bindings produced by the goal. May be null.

On success, the values of variables with names that do not start with underscore (‘_’) will be added to the map.

Returns true if the call succeeded, false if the call failed, i.e. there were no solutions.

HashMap varMap = new HashMap();

varMap.put("File", new SPTerm(sp, "datafile.txt"));

if (sp.query("see(File),do_something(Result),seen.", varMap)) {
  System.out.println("Result==" + ((SPTerm)varMap.get("Result")).toString());
} else {
  System.out.println("Failed");
}

Introduced in release 3.8.5.

boolean query (SPPredicate pred, SPTerm args[]) [Method on SICStus]

Obsolescent version of SICStus::query() above.

boolean queryCutFail (String module, String name, SPTerm args[]) [Method on SICStus]

Call name with args for side effect only.

As SICStus.query() it only finds the first solution, and then it cuts away all other solutions and fails.

It corresponds roughly to the following Prolog code:

( \+ call(Module:Name(Args...)) -> fail; true )
Introduced in release 3.8.5.

```java
boolean queryCutFail (String goal, Map varMap)  
Call a goal specified as a string, for side effect only. The map is only used for passing variable bindings into the goal. See query for details
```

Introduced in release 3.8.5.

```java
boolean queryCutFail (SPPredicate pred, SPTerm args[])
Obsolescent version of queryCutFail above.
```

```java
SPQuery openQuery (String module, String name, SPTerm args[])
Sets up a query (an object of class SPQuery), which can later be asked to produce solutions. You must close an opened query when no more solutions are required; see below.

It corresponds roughly to the following Prolog code:

```prolog
(true % just calling openQuery does not call the predicate ; % failing (nextSolution) will backtrack for more solutions call(Module:Name(Args...)))
```

Introduced in release 3.8.5.

```java
boolean openQuery (String goal, Map varMap)
Sets up a query specified as a string. See openQuery and query for details.
```

Introduced in release 3.8.5.

```java
SPQuery openQuery (SPPredicate pred, SPTerm args[])
Obsolescent version of openQuery above.
```

The following methods are used to obtain solutions from an opened query and to tell the SICStus runtime system that no more answers are required.

```java
boolean nextSolution ()
Obtain the next solution. Returns true on success and false if there were no more solutions. When you are no longer interested in any more solutions, you should call SPQuery.close or SPQuery.cut to close the query.

Returns true if a new solution was produced, false if there were no more solutions. This corresponds roughly to fail/0. See Section 10.20.6 [SPTerm and Memory], page 584, for additional details.
close ()  
[Method on SPQuery]
Cut and fail away any previous solution to the query. After closing a query object, you should not use it anymore. This corresponds roughly to !, fail. See Section 10.20.6 [SPTerm and Memory], page 584, for additional details.

cut ()  
[Method on SPQuery]
Cut, but do not fail away, any previous solution. After closing a query object with cut, you should not use it anymore. This corresponds roughly to !. See Section 10.20.6 [SPTerm and Memory], page 584, for additional details.

10.20.5 Java Exception Handling

Exceptions are handled seamlessly between Java and Prolog. This means that exceptions can be thrown in Prolog and caught in Java and the other way around. For example, if a predicate called from Java throws an exception with throw/1 and the predicate itself does not catch the exception, then the Java method that performed the query, queryCutFail() for example, will throw an exception (of class SPException) containing the exception term. Symmetrically, a Java exception thrown (and not caught) in a method called from Prolog will cause the corresponding predicate (simple/2 in the example above) to throw an exception consisting of the exception object (in the internal Prolog representation of a Java object). See Section 10.20.8.5 [Handling Java Exceptions], page 598, for examples of catching Java exceptions in Prolog.

10.20.6 SPTerm and Memory

Java and Prolog have quite different memory management policies. In Java, memory is reclaimed when the garbage collector can determine that no code will ever use the object occupying the memory. In Prolog, the garbage collector additionally reclaims memory upon failure (such as the failure implied in the use of SPQuery.close() and SPQuery::nextSolution()). This mismatch in the notion of memory lifetime can occasionally cause problems.

10.20.6.1 Lifetime of SPTerms and Prolog Memory

There are three separate memory areas involved when manipulating Prolog terms from Java using SPTerm objects. These areas have largely independent life times.

1. The SPTerm object itself.
2. Creating SPTerm object also tells Prolog to allocate an SP_term_ref. SP_term_refs have a life-time that is independent of the lifetime of the corresponding SPTerm object.
3. Any Prolog terms allocated on the global stack. An SPTerm refer to a Prolog term indirectly via a SP_term_ref.

A SP_term_ref ref (created as a side effect of creating a SPTerm object) will be reclaimed if either:

- Java returns to Prolog. This may never happen, especially if Java is the top-level application.
• Assume there exists a still open query \( q \) that was opened before the SP_term_ref \( ref \) was created. The SP_term_ref \( ref \) will be reclaimed if the query \( q \) is closed (using \( q\).close() or \( q\).cut()) or if \( q\).nextSolution() is called.

An SPTerm object will be invalidated (and eventually reclaimed by the garbage collector) if the corresponding SP_term_ref \( ref \) is reclaimed as above. If passed an invalidated SP_term_ref \( ref \), then most methods will throw an \texttt{IllegalTermException} exception.

A Prolog term (allocated on the global stack) will be deallocated when:

• Assume there exists a still open query \( q \) that was opened before the term was created. The memory of the term will be reclaimed if the query \( q \) is closed using \( q\).close() or if \( q\).nextSolution() is called. The memory is not reclaimed if the query is closed with \( q\).cut().

Please note: it is possible to get a SPTerm object and its SP_term_ref to refer to deallocated Prolog terms, in effect creating “dangling” pointers in cases where the SPTerm \( sp \) would ordinarily still be valid. This will be detected and invalidate the SPTerm:

```java
{ 
    SPTerm old = new SPTerm(sp);
    SPQuery q;

    q = sp.openQuery(....);
    ...
    old.consFunctor(...); // allocate a Prolog term newer than q
    ...
    q.nextSolution(); // or q.close()
    // error:
    // The SP_term_ref in q refers to an invalid part of the global stack
    // the SPTerm old will be invalidated by q.nextSolution()
}
```

10.20.6.2 Preventing SPTerm Memory Leaks

Some uses of SPTerm will leak memory on the Prolog side. This happens if a new SPTerm object is allocate, but Java neither returns to Prolog nor backtracks (using the method close, cut or nextSolution) into a query opened before the allocation of the SPTerm object.

As of release 3.8.5, it is possible to explicitly delete a SPTerm object using the SPTerm.delete() method. The delete() method invalidates the SPTerm object and makes the associated SP_term_ref available for reuse.

Another way to ensure that all SP_term.refs are deallocated is to open a dummy query only for this purpose. The following code demonstrates this:
// Always synchronize over creation and closing of SPQuery objects
synchronized (sp) {
    // Create a dummy query that invokes true/0
    SPQuery context = sp.openQuery("user","true",new SPTerm[]{});
    // All SP_term_refs created after this point will be reclaimed by
    // Prolog when doing context.close() (or context.cut())
    try { // ensure context is always closed
        SPTerm tmp = new SPTerm(sp); // created after context
        int i = 0;

        while (i++ < 5) {
            // reused instead of doing tmp = new SPTerm(sp,"...");
           ,tmp.putString("Iteration #" + i + "\n");
            // e.g. user:write('Iteration #1\n')
            sp.queryCutFail("user", "write", new SPTerm[]{tmp});
        }
    }
    finally {
        // This will invalidate tmp and make Prolog
        // reclaim the corresponding SP_term_ref
        context.close(); // or context.cut() to retain variable bindings.
    }
}

10.20.7 Java Threads

None of the pre-3.9 methods in se.sics.jasper are thread safe. They can only be called from the thread that created the SICStus object. (This is different from how se.sics.jasper used to work in release 3.8.)

As of 3.9 there are two ways to set up for calls to SICStus from multiple threads.

One way is to use the static method newProlog in the class Jasper:

Prolog newProlog (String argv[], String bootPath) [Method on Jasper]
    Creates a Prolog interface object. Starts a server thread, which will serve that
    Prolog. The server thread takes care of all interaction with the Prolog runtime,
    making sure that all calls to the Prolog runtime will be done from one and the same
    thread.

See the HTML documentation on the interface Prolog for details on what methods are available for a client thread.

Another way is to create a SICStus object and use the following methods:
Prolog newProlog () [Method on SICStus]
   Returns the Prolog interface for this SICStus object. Creates a server and a client (Prolog) interface for this SICStus object. The server may be started by calling startServer().

startServer () [Method on SICStus]
   Start serving requests from a Prolog client. This method does not return until another thread calls stopServer(). Before calling this method you should call newProlog() and hand the result over to another Thread.

stopServer () [Method on SICStus]
   Stops the server. Calling this method causes the Thread running in the startServer() method to return.

As with the first method, the interface Prolog defines the methods available for the client threads.

10.20.8 The Jasper Library

The Jasper library module is the Prolog interface to the Java VM. It corresponds to the se.sics.jasper package in Java. It is loaded by executing the query:

   | ?- use_module(library(jasper)).

The Jasper library fully supports multiple SICStus runtimes in a process.

Jasper cannot be used when the SICStus runtime is statically linked to the executable, such as when using spld --static.

The following functionality is provided:

- Initializing the Java VM using the JNI Invocation API (jasper_initialize/[1,2], jasper_deinitialize/1).
- Creating and deleting Java objects directly from Prolog (jasper_new_object/5).
- Method calls (jasper_call/4).
- Global and local (object) reference management (jasper_create_global_ref/3, jasper_delete_global_ref/2, jasper_delete_local_ref/2). Global references are used to prevent the JVM from garbage collecting a Java object referenced from Prolog.
- There is also a subdirectory containing example programs (library('jasper/examples')).

10.20.8.1 Jasper Method Call Example

We begin with a small example.
import se.sics.jasper.*;

public class Simple {
    private String instanceDatum = "this is instance data";

    static int simpleMethod(int value) {
        return value*42;
    }

    public String getInstanceData(String arg) {
        return instanceDatum + arg;
    }
}

Compile Simple.java (UNIX):

    % javac -deprecation \ 
     -classpath <installdir>/lib/sicstus-4.7.0/bin/jasper.jar Simple.java

Under Windows this may look like (the command should go on a single line):

    C:\> c:\jdk1.2.2\bin\javac -deprecation \n    -classpath "D:\Program Files\SICStus Prolog VC15 4.7.0\bin\jasper.jar" Simple.java

The option ‘-deprecation’ is always a good idea, it makes javac warn if your code use deprecated methods.
:- use_module(library(jasper)).
main :-
  %% Replace '/my/java/dir' below with the path containing
  %% 'Simple.class', e.g. to look in the current directory use
  %% classpath(['/']).
  %% You can also use the CLASSPATH environment variable and call
  %% jasper_initialize(JVM)
  %% Under Windows it may look like classpath(['C:/MyTest'])
  jasper_initialize([classpath(['/my/java/dir']),JVM]),
  format('Calling a static method...~n',[]),
  jasper_call(JVM,
    method('Simple','simpleMethod',[static]), % Which method
    simple_method(+integer,[integer]), % Types of arguments
    simple_method(42,X)), % The arguments.
  format('simpleMethod(~w) = ~w~n',[42,X]),

  format('Creating an object...~n',[]),
  jasper_new_object(JVM, 'Simple', init, init, Object),
  format('Calling an instance method on ~w...~n',[Object]),
  jasper_call(JVM,
    method('Simple','getInstanceData',[instance]),
    getInstance_data(+object('Simple'), +string,[string]),
    getInstance_data(Object, 'foobar', X1)),
  format('getInstanceData(~w) = ~w~n',['foobar',X1]).

Then, run SICStus:

% echo "[simple],main." | sicstus
SICStus 4.7.0 ...
Licensed to SICS
% consulting /home1/lojo/simple.pl...
[...]
% consulted /home1/lojo/simple.pl in module user, 100 msec 26644 bytes
Calling a static method...
simpleMethod(42) = 1764
Creating an object...
Calling and instance method on $java_object(135057576)...
getInstanceData(foobar) = this is instance datafoobar

This example performed three things.

- The static method simpleMethod was called with argument '42', and returned the square of '42', '1764'.
• An object of class Simple was created.
• The method getInstanceData was executed on the object just created. The method took an atom as an argument and appended the atom to a string stored as a field in the object, yielding "this is instance datafoobar".

10.20.8.2 Jasper Library Predicates

jasper_initialize(-JVM)
jasper_initialize(+Options, -JVM)

Loads and initializes the Java VM. JVM is a reference to the Java VM. Options is a list of options. The options can be of the following types:

classpath(<classpath>)
If <classpath> is an atom, then it will be added (unmodified) to the Java VM’s classpath. If <classpath> is a list, then each element will be expanded using absolute_file_name/2 and concatenated using the Java VM’s path separator. Example:
classpath([library('jasper/examples'),'$HOME/joe'])
In addition to the classpaths specified here, Jasper will automatically add jasper.jar to the classpath together with the contents of the CLASSPATH environment variable.

if_exists(option)
This option determines what happens if a JVM has already been initialized, either through a previous call to jasper_initialize or because Prolog have been called from Java. If a JVM already exists, then the other options are ignored.
ok The default. Argument JVM is bound to the existing JVM.
fail The call to jasper_initialize/2 fails.
error The call to jasper_initialize/2 throws an exception (java_exception(some text)).

if_not_exists(option)
This option determines what happens if a JVM has not already been initialized.
ok The default. The remaining options are used to initialize the JVM.
fail The call to jasper_initialize/2 fails.
error The call to jasper_initialize/2 throws an exception (java_exception(some text)).

As an example, to access the currently running JVM and to give an error if there is no running JVM use jasper_initialize([if_exists(ok),if_not_exists(error)], JVM).
Option  The option is an atom that will be passed directly to the Java VM as an option. This enables the user to send additional options to the Java VM. Example:

\[ jasper_initialize(['-Dkenny.is.dead=42'], JVM), \]

In addition to the options specified by the user, Jasper adds a couple of options on its own in order for Java to find the Jasper classes and the Jasper native library.

There is currently no support for creating multiple JVMs (few JDKs, if any, supports this).

\[ jasper_deinitialize(+JVM) \]

De-initialize Java. Do Not call this, current versions of the JVM does not support deinitialization.

\[ jasper_call(+JVM,+Method,+TypeInfo,+Args) \]

Calls a Java static or instance method.

\[ JVM \]
A reference to the Java VM, as obtained by \[ jasper_initialize/[1,2] \].

\[ Method \]
A term of the form \[ method(ClassName, MethodName, Flags) \] that identifies the method to call.

\[ ClassName \]
This is the Fully Qualified Classname of the class (for example, \[ java/lang/String \]) of the object or where to look for the static method. Note that you need to surround the atom with single quotes since it contains / characters. The class is ignored when calling instance methods but should still be an atom, e.g. 'Java'.

\[ Name \]
This is the name of the method, as an atom.

\[ Flags \]
This is the singleton list \[ [instance] \] for instance methods and \[ [static] \] for static methods.

\[TypeInfo\]
Information about the argument types and the argument conversion that should be applied. See Section 10.20.8.3 [Conversion between Prolog Arguments and Java Types], page 593, for more information on specifying argument types.

Note that for an instance method the first argument must be an object reference (specified with \[ +object(Class) \]). In this case the class is ignored but should still be an atom, e.g. 'Java'.

\[ Args \]
A term with one position for each argument to the method. For an instance method the first argument is the instance.

\[ jasper_new_object(+JVM,+ClassName,+TypeInfo,+Args,-Object) \]

Creates a new Java object.
See `jasper_call/4` above for an explanation of the arguments JVM, ClassName, TypeInfo and Args.

**ClassName**
An an atom containing the *fully qualified classname*

**TypeInfo**
TypeInfo has the same format as for a static void method.

**Args**
A term with one position for each argument to the constructor.

**Object**
This argument is bound to a (local) reference to the created object. See Section 10.20.8.4 [Global vs. Local References], page 597.

As an example, the following code creates a `java/lang/Integer` object initialized from a string of digits. It then calls the instance method `doubleValue` to obtain the floating point representation of the Integer.

```prolog
?- Chars = "4711",
   %% get existing JVM
   jasper_initialize([if_not_exists(error)], JVM),
   jasper_new_object(JVM, 'java/lang/Integer',
     init(+chars), init(Chars), S),
   jasper_call(JVM,
     method('java/lang/Integer', doubleValue, [instance]),
     to_double(+object('java/lang/Integer'), [-double]),
     to_double(S,X)).
S = '$java_object'(135875344),
X = 4711.0, % note that this is now a floating point number
JVM = '$jvm'(1076414148),
Chars = [52,55,49,49] % a.k.a. "4711"
```

`jasper_create_global_ref(+JVM,+Ref,-GlobalRef)`
Creates a global reference (GlobalRef) for a (non-null) Java object (Ref). See Section 10.20.8.4 [Global vs. Local References], page 597.

`jasper_delete_global_ref(+JVM,+GlobalRef)`
Destroys a global reference. See Section 10.20.8.4 [Global vs. Local References], page 597.

`jasper_create_local_ref(+JVM,+Ref,-LocalRef)`
Creates a local reference (LocalRef) for a (non-null) Java object (Ref). See Section 10.20.8.4 [Global vs. Local References], page 597. Rarely needed.

`jasper_delete_local_ref(+JVM,+GlobalRef)`
Destroys a local reference. See Section 10.20.8.4 [Global vs. Local References], page 597.

`jasper_is_jvm(+JVM)`
Succeeds if JVM is a reference to a Java Virtual Machine.
jasper_is_object(+Object)
jasper_is_object(+JVM,+Object)

Succeeds if Object is a reference to a Java object. The representation of Java object will change so use jasper_is_object/1 to recognize objects instead of relying on the internal representation. Currently the JVM argument is ignored. If, and when, multiple JVMs becomes a possibility jasper_is_object/2 will verify that Object is an object in a particular JVM.

jasper_is_same_object(+JVM,+Object1,+Object2)

Succeeds if Object1 and Object2 refers to the same Java object (or both are null object references). The same object may be represented by two different terms in Prolog so ==/2 can not be used to reliably detect if two object references refer to the same object.

jasper_is_instance_of(+JVM,+Object,+ClassName)

Succeeds if Object is an instance of class ClassName; fails otherwise. ClassName is a fully qualified classname; see jasper_call/4.

jasper_object_class_name(+JVM,+Object,-ClassName)

Returns the fully qualified name of the class of +Object as an atom.

jasper_null(+JVM,-NullRef)

Create a null object reference.

jasper_is_null(+JVM,+Ref)

Succeeds if Ref is a null object reference, fails otherwise, e.g. if Ref is not an object reference.

10.20.8.3 Conversion between Prolog Arguments and Java Types

The following table lists the possible values of arguments of the argument type specification to jasper_call/4 and jasper_new_object/5 (see Section 10.20.8.2 [Jasper Library Predicates], page 590). The value specifies which conversion between corresponding Prolog argument and Java type will take place.

There is currently no mechanism for specifying Java arrays in this way.

In the following the package prefix (java/lang or se/sics/jasper) has been left out for brevity.

For several of the numerical types there is the possibility that the target type cannot accurately represent the source type, e.g. when converting from a Prolog integer to a Java byte. The behavior in such cases is unspecified.

Prolog: +integer
Java: int

The argument should be a number. It is converted to a Java int, a 32 bit signed integer.

Prolog: +byte
Java: byte

The argument should be a number. It is converted to a Java byte.
Prolog: +short
Java: short
The argument should be a number. It is converted to a Java short, a 16 bit signed integer.

Prolog: +long
Java: long
The argument should be a number. It is converted to a Java long, a 64-bit signed integer.
In releases prior to 3.9.1, the value was truncated to 32 bits when passed between Java and Prolog. This is no longer the case.

Prolog: +float
Java: float
The argument should be a number. It is converted to a Java float.

Prolog: +double
Java: double
The argument should be a number. It is converted to a Java double.

Prolog: +term
Java: SPTerm
The argument can be any term. It is passed to Java as an object of the class SPTerm.

Prolog: +object(Class)
Java: Class
The argument should be the Prolog representation of a Java object of class Class. Unless it is the first argument in a non-static method (in which case it is treated as the object on which the method should be invoked), it is passed to the Java method as an object of class Class.

Prolog: +atom
Java: SPCanonicalAtom
The argument should be an atom. The Java method will be passed an object of class SPCanonicalAtom. Often +string, see below, is more useful.

Prolog: +boolean
Java: boolean
The argument should be an atom in \{true,false\}. The Java method will receive a boolean.

Prolog: +chars
Java: String
The argument should be a code list. The Java method will receive an object of class String.

Prolog: +codes
Java: String
+codes is an alias for +chars.
The argument should be an atom. The Java method will receive an object of class `String`.

**Note.** When using `+chars`, `+codes` or `+string` the automatic type conversion mechanism will also create a type signature of the form `java/lang/String`. If you want to call a method that accepts a String object as a parameter, but has different signature, then the method lookup will fail. A workaround is to explicitly create a String object and then call the method. For example:

```prolog
:- use_module(library(jasper)).

main :-
    jasper_initialize([], JVM),
    jasper_new_object(JVM, 'java/lang/String', init(+chars), init("hamburger"), H),
    Str = "urge",
    jasper_new_object(JVM, 'java/lang/String', init(+chars), init(Str), S),
    jasper_call(JVM, method('', contains, [instance]),
                contains(+object(''), +object('java/lang/CharSequence'), [-boolean]),
                contains(H, S, B)),
    format('Contains? ~a~n', [B]).
```

The Java method will receive an object of class `SPTerm`, which should be set to an atom (e.g. using `SPTerm.putString`). The argument will be bound to the value of the atom when the method returns. Often `-term`, see below, is more useful.

The Java method will receive an (empty) object of type `StringBuffer`, which can be modified. The argument will be bound to a code list of the `StringBuffer` object.

`-codes` is an alias for `-chars`. 
Prolog: -string
Java: StringBuffer
The Java method will receive an object of type StringBuffer, which can be modified. The argument will be bound to an atom converted from the StringBuffer object.

Prolog: -term
Java: SPTerm
The Java method will receive an object of class SPTerm, which can be set to a term (e.g. using SPTerm.consFunctor). The argument will be bound to the term when the method returns.

Prolog: [-integer]
Java: int M()
The Java method should return an int. The value will be converted to a Prolog integer.

Prolog: [-byte]
Java: byte M()
The Java method should return a byte. The value will be converted to a Prolog integer.

Prolog: [-short]
Java: short M()
The Java method should return a short. The value will be converted to a Prolog integer.

Prolog: [-long]
Java: long M()
The Java method should return a long, a 64 bit signed integer. The value will be converted to a Prolog integer.

Prolog: [-float]
Java: float M()
The Java method should return a float. The value will be converted to a Prolog float.

Prolog: [-double]
Java: double M()
The Java method should return a double. The value will be converted to a Prolog float.

Prolog: [-term]
Java: SPTerm M()
The Java method should return an object of class SPTerm, which will be converted to a Prolog term.

Prolog: [-object(Class)]
Java: SPTerm M()
The Java method should return an object of class Class, which will be converted to the internal Prolog representation of the Java object.
Chapter 10: The Prolog Library

Prolog: [-atom]  
Java: SPTerm M()  
    The Java method should return an object of class SPCanonicalAtom, which will be converted to a Prolog atom. Often [-term], see above, is more useful.

Prolog: [-boolean]  
Java: boolean M()  
    The Java should return a boolean. The value will be converted to a Prolog atom in \{true,false\}.

Prolog: [-chars]  
Java: String M()  
    The Java method should return an object of class String, which will be converted to a code list.

Prolog: [-codes]  
Java: String M()  
    [-codes] is an alias for [-chars].

Prolog: [-string]  
Java: String M()  
    The Java method should return an object of class String, which will be converted to an atom.

10.20.8.4 Global vs. Local References

It is important to understand the rules determining the life-span of Java object references. These are similar in spirit to the SP_term_refs of the C-Prolog interface, but since they are used to handle Java objects instead of Prolog terms they work a little differently.

Java object references (currently represented in Prolog as '${java_object}/1 terms) exist in two flavors: local and global. Their validity are governed by the following rules.

1. A local reference is valid until Prolog returns to Java or the reference is deleted with jasper_delete_local_ref/2. It is only valid in the (native) thread in which it was created. As a rule of thumb a local reference can be used safely as long as it is not saved away using assert/3 or similar.

Since local references are never reclaimed until Prolog returns to Java (which may never happen) you should typically call jasper-delete-local-ref/2 when your code is done with an object.

2. A global reference is valid until explicitly freed. It can be used from any native thread.

3. All objects returned by Java methods are converted to local references.

4. Java exceptions not caught by Java are thrown as Prolog exceptions consisting of a global reference to the exception object, see Section 10.20.8.5 [Handling Java Exceptions], page 598.

Local references can be converted into global references (jasper_create_global_ref/3). When the global reference is no longer needed, it should be deleted using jasper_delete_global_ref/2.
For a more in-depth discussion of global and local references, consult the JNI Documentation (https://docs.oracle.com/en/java/javase/11/docs/specs/jni/index.html).

Using a local (or global) reference that has been deleted (either explicitly or by returning to Java) is illegal and will generally lead to crashes. This is a limitation of the Java Native Interface used to implement the low level interface to Java.

### 10.20.8.5 Handling Java Exceptions

If a Java method throws an exception, e.g. by using `throw new Exception("...")` and the exception is not caught by Java, then it is passed on as a Prolog exception. The thrown term is a *global* reference to the Exception object. The following example code illustrates how to handle Java exceptions in Prolog:

```prolog
exception_example(JVM, ...) :-
  catch(
    %\% Call Java method that may raise an exception
    jasper_call(JVM, ...),
    Excp,
    (    
      ( is_java_exception(JVM, Excp)
        -> print_exception_info(JVM, Excp)
        ; throw(Excp) % pass non-Java exceptions to caller
      )
    )
  ).

is_java_exception(_JVM, Thing) :- var(Thing), !, fail.
is_java_exception(_JVM, Thing) :-
  Thing = java_exception(_), % misc error in Java/Prolog glue
  !.
is_java_exception(JVM, Thing) :-
  jasper_is_object(JVM, Thing),
  jasper_is_instance_of(JVM, Thing, 'java/lang/Throwable')
```


print_exception_info(_, java_exception(Message)) :- !,
    format(user_error, "NJasper exception: "w~n", [Message]).
print_exception_info(JVM, Excp) :-
    /*
    // Approximate Java code
    {
    String messageChars = excp.getMessage();
    }
    */
    jasper_call(JVM,
        method('java/lang/Throwable', 'getMessage', [instance]),
        get_message(+object('java/lang/Throwable'), [-chars]),
        get_message(Excp, MessageChars)),
    /* // Approximate Java code
    {
    StringWriter stringWriter = new StringWriter();
    PrintWriter printWriter = new PrintWriter(stringWriter);
    excp.printStackTrace(printWriter);
    printWriter.close();
    stackTraceChars = StringWriter.toString();
    }
    */
    jasper_new_object(JVM, 'java/io/StringWriter',
        init, init, StringWriter),
    jasper_new_object(JVM, 'java/io/PrintWriter',
        init(+object('java/io/Writer')),
        init(StringWriter), PrintWriter),
    jasper_call(JVM,
        method('java/lang/Throwable', 'printStackTrace', [instance]),
        print_stack_trace(+object('java/lang/Throwable'),
            +object('java/io/PrintWriter'),
        print_stack_trace(Excp, PrintWriter)),
    jasper_call(JVM,
        method('java/io/PrintWriter','close',[instance]),
        close(+object('java/io/PrintWriter')),
        close(PrintWriter)),
    jasper_call(JVM,
        method('java/io/StringWriter','toString',[instance]),
        to_string(+object('java/io/StringWriter'),[-chars]),
        to_string(StringWriter, StackTraceChars)),
    jasper_delete_local_ref(JVM, PrintWriter),
    jasper_delete_local_ref(JVM, StringWriter),
    %% ! exceptions are thrown as global references
    jasper_delete_global_ref(JVM, Excp),
    format(user_error, "NJava Exception: "s
    StackTrace: "s~n",
    [MessageChars, StackTraceChars]).
10.20.8.6 Deprecated Jasper API

The information in this section is only of interest to those that need to read or modify code that used library(jasper) before release 3.8.5.

A different way of doing method call and creating objects was used in versions of library(jasper) predating release 3.8.5. Use of these facilities are strongly discouraged although they are still available in the interest of backward compatibility.

The old method call predicates are jasper_call_static/6 and jasper_call_instance/6 as well as the old way of calling jasper_new_object/5.

10.20.8.7 Deprecated Argument Conversions

The pre release 3.8.5 method call predicates in this library use a specific form of argument lists containing conversion information so the predicates know how to convert the input arguments from Prolog datatypes to Java datatypes. This is similar to the (new) mechanism described in Section 10.20.8.3 [Conversion between Prolog Arguments and Java Types], page 593. The argument lists are standard Prolog lists containing terms on the following form:

\[
\text{jboolean}(X) \\
X \text{ is the atom true or false, representing a Java boolean primitive type.}
\]

\[
\text{jbyte}(X) \\
X \text{ is an integer, which is converted to a Java byte.}
\]

\[
\text{jchar}(X) \\
X \text{ is an integer, which is converted to a Java char.}
\]

\[
\text{jdouble}(X) \\
X \text{ is a float, which is converted to a Java double.}
\]

\[
\text{jfloat}(X) \\
X \text{ is a float, which is converted to a Java float.}
\]

\[
\text{jint}(X) \\
X \text{ is an integer, which is converted to a Java int.}
\]

\[
\text{jlong}(X) \\
X \text{ is an integer, which is converted to a Java long.}
\]

\[
\text{jshort}(X) \\
X \text{ is an integer, which is converted to a Java short.}
\]

\[
\text{jobject}(X) \\
X \text{ is a reference to a Java object, as returned by jasper_new_object/5 (see Section 10.20.8.2 [Jasper Library Predicates], page 590).}
\]

\[
\text{jstring}(X) \\
X \text{ is an atom, which is converted to a Java String.}
\]

If the Prolog term does not fit in the corresponding Java data type (jbyte(4711), for example), then the result is undefined.

10.20.8.8 Deprecated Jasper Predicates

\[
\text{jasper_new_object(+JVM,+Class,+TypeSig,+Args,-Object)} \\
\text{obsolescent}
\]

Creates a new Java object.
JVM A reference to the Java VM, as obtained by jasper_initialize/[1,2].

Class An an atom containing the fully qualified classname (i.e. package name separated with ‘/’, followed by the class name), for example java/lang/String, se/sics/jasper/SICStus.

TypeSig The type signature of the class constructor. A type signature is a string that uniquely defines a method within a class. For a definition of type signatures, see the JNI Documentation (http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html).

Args A list of argument specifiers. See Section 10.20.8.7 [Deprecated Argument Conversions], page 600.

Object A term on the form ‘$java_object’(X), where X is a Java object reference. This is the Prolog handle to the Java object. See Section 10.20.8.4 [Global vs. Local References], page 597.

10.21 JSON format serialization—library(json)

This library module provides some utilities for reading and writing structured data using the JSON (https://json.org/) (JavaScript Object Notation) serialization format. The library module is part of SICStus Prolog since release 4.5.0.

JSON is a light-weight, language independent, data-interchange format with good support in many environments. As such, it is a convenient format when transferring data between Prolog and other programming languages. The format is specified in ECMA-404 and in RFC 8259.

The Prolog representation of JSON values is as follows:

Number A JSON number is represented as the corresponding Prolog number; as a float when the JSON number has an exponent or a fractional part, otherwise as an integer.

String A JSON string is represented as the corresponding Prolog atom (escaped surrogate pairs are combined into the corresponding Unicode code point).

Array A JSON array is represented as a list of the corresponding Prolog terms.

Object A JSON object is a sequence of name:value pairs, where each name is a JSON string and each value is an arbitrary JSON value. It is represented as a term json(Members) where Members is a list of Name=Value pairs, where Name is a representation of the JSON string name and Value is a representation of the JSON value.

null true false These special JSON literals are, by default, translated to the Prolog terms @(null), @(true), and @(false), respectively.
Examples:

123 ⇒ 123

12.3 ⇒ 12.3

12E3 ⇒ 12.0E3

"foo" ⇒ 'foo'

null ⇒ @(null)

[['a', 2, "bar"] ⇒ ['a', 2, 'bar']

{"age": 42,
  "name": {"first":"Kim", "last":"Jones"},
  "children": ["Lisa","Jim","John"]
}

⇒

json([ 'age'=42,
      'name'=json([ 'first'='Kim', 'last'='Jones' ]),
      'children'=['Lisa','Jim','John']])

It is possible to specify other Prolog representations of a JSON value using the option argument. See below for details.

10.21.1 Options

The following options are used. They are valid for all predicates that takes options, unless stated otherwise.

compact(Boolean)
Valid values for Boolean are true and false (default).
If false (default), JSON values are written with extra whitespace and end-of-line characters to make it easier for humans to read. The details of the non-compact format is subject to change without notice.
If true, JSON values are written with a minimum of whitespace. Since this implies that no end-of-line characters will be written, it makes it possible to read the resulting JSON as a single line.
Only valid for predicates that write.

ascii(Boolean)
Valid values for Boolean are true (default) and false.
If true (default), JSON values are written using only 7-bit ASCII characters, which makes the format less sensitive to stream encodings.
If false, JSON values are written using full Unicode. In this case any streams should use UTF-8 encoding.
null(X)  
true(X)  
false(X)  The specified term X, which may be a variable, is used for representing the correspond-
ing JSON literal.

array_tag(Tag)  
The Tag must be an atom.  
A JSON array is represented as the compound term Tag(Elements), where Elements is a list of the represen-
tations of the array elements. This may be useful if you need to be able to distinguish between an empty JSON array ([]), and a JSON string "[]", since these have the same Prolog representation (the atom []) in the default representation.  
If this option is not specified (the default), then JSON arrays are represented as a list of the representations of the array elements.

object_tag(Tag)  
The Tag must be an atom. Tag defaults to 'json'.  
A JSON object is represented as the compound term Tag(Members), where Members is a list of Name=Value pairs, where Name is a representation of the JSON string name and Value is a representation of the JSON value.

width(Width)  This option is present for compatibility with other systems.  
If Width is 0 (zero), it is treated as a synonym for compact(true). Otherwise, the option is currently ignored.  
Only valid for predicates that write.

value_string_as(Value)  
step(Value)  
tab(Value)  
serialize_unknown(Value)  These options are present for compatibility with other systems. They are currently ignored.  
Only valid for predicates that write.

10.21.2 Exported Predicates  
The Options argument is described in the module documentation.

json_read(+Stream, -Term)  
json_read(+Stream, -Term, +Options)  
Reads a single JSON value from the text stream Stream and unifies it with Term.

json_write(+Stream, +Term)  
json_write(+Stream, +Term, +Options)  
Write the JSON value Term to the text stream Stream.
is_json_term(+Term)
is_json_term(+Term, +Options)
  True if the Term is a valid representation of a JSON value.

json_to_codes(+Term, -JSONCodes)
json_to_codes(+Term, -JSONCodes, +Options)
  Writes Term as JSON and unifies JSONCodes with the list of resulting character codes.

json_to_atom(+Term, -JSONAtom)
json_to_atom(+Term, -JSONAtom, +Options)
  Writes Term as JSON and unifies JSONAtom with an atom consisting of the resulting character codes.

json_from_codes(+JSONCodes, -Term)
json_from_codes(+JSONCodes, -Term, +Options)
  Converts a JSON text, represented as the list of character codes JSONCodes, into the corresponding Prolog term Term.

json_from_atom(+JSONAtom, -Term)
json_from_atom(+JSONAtom, -Term, +Options)
  Converts a JSON text, represented as the character codes of JSONAtom, into the corresponding Prolog term Term.

A small example:

| ?- JSONCodes = "{"foo": 42, "bar": null}"
  json_from_codes(JSONCodes, JSONTerm),
  json_to_atom(JSONTerm, JSONAtom, [compact(true)]).
  JSONCodes = [123,34,102,111,111,34,58,32,52,50|...],
  JSONTerm = json([foo=42,bar= @(null)]),
  JSONAtom = '{"foo":42,"bar":null}' ?
  yes
  | ?-

10.22 JSON mediated process communication—

library(jsonrpc/jsonrpc_server)

This is not a library module proper, but a collection of examples of using JSON to communicate between a non-Prolog parent process and a SICStus sub-process. There are examples of writing the parent process using Python, C#, Java, C, Prolog etc.. These examples provide functionality that is similar to what is available in the language-specific libraries, e.g. library(jasper) and library(prologbeans), but do so in a language-agnostic way. The collection has been part of SICStus Prolog since release 4.5.0, but was made more visible in release 4.7.0. The code is meant to demonstrate possibilities of this process communication method. It is not intended for production use.

The collection can be found in library/jsonrpc and consists of a server written in Prolog, library/jsonrpc/jsonrpc_server.pl, and clients written in C, C#, Java, Prolog, Go, and Python, library/jsonrpc/clients/*.
Following is a number of sessions. Each session has the same structure. The client process first creates a server subprocess, then issues a number of requests to the server and reads the replies, finally shuts down the server. The interactions illustrate functionality like state updates, determinate and nondeterminate Prolog queries, backtracking, failures, and exceptions:

1. Read the initial state.
2. Set state to 4.
3. Read current state.
4. Increment current state.
5. Increment current state again.
6. Call member/2 and ask for the first solution.
7. Ask for the next solution.
8. Commit to the last solution.
9. Call member/2 again and ask for the first solution.
10. Ask for the next solution.
11. Ask for the next solution.
12. Ask for the next solution, which fails.
13. Make a call that throws an error exception.
14. Shut down the server.
These sessions are using Linux or macOS, the % is the shell prompt. On Windows starting the programs will be different but program outputs will be very similar. Recall that $SP_APP_PATH denotes the absolute path to the SICStus development system.

The clients are intentionally written to be very similar in structure, and to have almost identical outputs, regardless of implementation language.

The first session is with the Python client:

```
% python3 "$SP_LIBRARY_DIR/jsonrpc/clients/jsonrpc_client.py" "$SP_APP_PATH"
state ==> State is None
state:=4 ==> State was None
state ==> State is 4

once(Result is StateIn+1, StateOut=Result). ==> Result=5
once(Result is StateIn+1, StateOut=Result). ==> Result=6

Increment=5, once(Result is StateIn+Increment, StateOut=Result). ==> Result=11

Multiplier=10, call(member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc). ==> First Result=111
retry ==> (next) Result=211
cut ==> Result=None

Multiplier=10, call(member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc). ==> First Result=111
retry ==> (next) Result=211
retry ==> (next) Result=311
retry ==> Prolog failed (this is expected)

once(foo is bar). ==> Prolog threw an exception (this is expected): msg=Exception, data=type_error(evaluable,bar/0)

quit ==> Result=Bye
```
The next session is with the Prolog client:

```prolog
\% sicstus --nologo --noinfo \
-1 "$SP_LIBRARY_DIR/jsonrpc/clients/jsonrpc_client.pl"
state == State is @(null)
state:=4 ==> State was @(null)
state ==> State is 4

once('Result is StateIn+1, StateOut=Result.'). ==> Result=5
once('Result is StateIn+1, StateOut=Result.'). ==> Result=6

Increment=5, once('Result is StateIn+Increment, StateOut=Result.'). ==> Result=11

Multiplier=10, call('member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc.'). ==> First Result=111
retry ==> (next) Result=211
cut ==> Result=@(null)

Multiplier=10, call('member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc.'). ==> First Result=111
retry ==> (next) Result=211
retry ==> (next) Result=311
retry ==> Prolog failed (this is expected)

once('foo is bar.'). ==> Prolog threw an exception (this is expected): Exception 'type_error(evaluable,bar/0)'

quit ==> Result='Bye'
```
The next session is with the Java client:

```prolog
\% javac -d "$SP_LIBRARY_DIR/jsonrpc/clients/JSONRPCClient.java"
\% jar cf JSONRPCClient.jar JSONRPCClient*.class
\% java -Dlogging=false -cp JSONRPCClient.jar \n   JSONRPCClient "$SP_APP_PATH"
state ==> State is null
state:=4 ==> State was null
state ==> State is 4

once(Result is StateIn+1, StateOut=Result). ==> Result=5
once(Result is StateIn+1, StateOut=Result). ==> Result=6

Increment=5, once(Result is StateIn+Increment, StateOut=Result). ==> Result=11

Multiplier=10, call(member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc). ==> First Result=111
retry ==> (next) Result=211
cut ==> Result=null

Multiplier=10, call(member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc). ==> First Result=111
retry ==> (next) Result=211
retry ==> (next) Result=311
retry ==> Prolog failed (this is expected)

once(foo is bar). ==> Prolog threw an exception (this is expected): "type_error(evaluable,bar/0)"

quit ==> Result="Bye"
```
The next session is with the C client:

```prolog
% cc "$SP_LIBRARY_DIR/jsonrpc/clients/jsonrpc_client.c" -o jsonrpc_client
% ./jsonrpc_client "$SP_APP_PATH"

state ==> State is null
state:=4 ==> State was null
state ==> State is 4

once(Result is StateIn+1, StateOut=Result). ==> Result=5
once(Result is StateIn+1, StateOut=Result). ==> Result=6

Increment=5, once(Result is StateIn+Increment, StateOut=Result). ==> Result=11

Multiplier=10, call(member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc). ==> First Result=111
retry ==> (next) Result=211
cut ==> Result=null

Multiplier=10, call(member(E,[10,20,30]), Inc is Multiplier*E, Result is StateIn+Inc). ==> First Result=111
retry ==> (next) Result=211
retry ==> (next) Result=311
retry ==> Prolog failed (this is expected)

once(foo is bar). ==> Prolog threw an exception (this is expected)

quit ==> Result="Bye"
```

As mentioned above, there are sample clients in other languages as well.

### 10.23 Process Communication—

library(linda/[server,client])

Linda is a concept for process communication.

For an introduction and a deeper description, see [Carreiro & Gelernter 89a] or [Carreiro & Gelernter 89b], respectively.

One process is running as a server and one or more processes are running as clients. The processes are communicating with sockets and supports networks.

The server is in principle a blackboard on which the clients can write (out/1), read (rd/1) and remove (in/1) data. If the data is not present on the blackboard, then the predicates suspend the process until they are available.
There are some more predicates besides the basic out/1, rd/1 and in/1. The in_noblock/1 and rd_noblock/1 does not suspend if the data is not available—they fail instead. A blocking fetch of a conjunction of data can be done with in/2 or rd/2.

**Example**: A simple producer-consumer. In client 1:

```prolog
producer :-
    produce(X),
    out(p(X)),
    producer.

produce(X) :- ......
```

In client 2:

```prolog
consumer :-
    in(p(A)),
    consume(A),
    consumer.

consume(A) :- ......
```

**Example**: Synchronization

```prolog
..., in(ready), % Waits here until someone does out(ready)
..., 
```

**Example**: A critical region

```prolog
..., in(region_free), % Wait for region to be free
    critical_part,
    out(region_free), % let next one in
..., 
```

**Example**: Reading global data

```prolog
..., rd(data(Data)),
..., 
```

or, without blocking:

```prolog
..., rd_noblock(data(Data)) ->
    do_something(Data)
; write('Data not available!'),nl
),
..., 
```
Example: Waiting for one of several events

\[
\text{in([e(1),e(2),...,e(n)], E),}
\]
% Here is E instantiated to the first tuple that became available
\[
\text{...},
\]

10.23.1 Linda Server

The server is the process running the “blackboard process”. It is an ordinary SICStus process, which can be run on a separate machine if necessary.

To load the package, enter the query

\[
| ?- \text{use_module(library('linda/server')).}
\]

and start the server with \text{linda/[0,1]}.

\text{linda}

Starts a Linda-server in this SICStus. The network address is written to the current output stream as \text{Host:PortNumber}.

\text{linda(:Options)}

Starts a Linda-server in this SICStus. Each option on the list \text{Options} is one of \text{Address-Goal}

where \text{Address} must be unifiable with \text{Host:Port} and \text{Goal} must be instantiated to a goal.

When the linda server is started, \text{Host} and \text{Port} are bound to the server host and port respectively and the goal \text{Goal} is called. A typical use of this would be to store the connection information in a file so that the clients can find the server to connect to.

For backward compatibility, if \text{Options} is not a list, then it is assumed to be an option of the form \text{Address-Goal}.

Before release 3.9.1, \text{Goal} needed an explicit module prefix to ensure it was called in the right module. This is no longer necessary since \text{linda/1} is now a meta-predicate.

\text{accept_hook(Client,Stream,Goal)}

When a client attempts to connects to the server \text{Client} and \text{Stream} will be bound to the IP address of the client and the socket stream connected to the client, respectively. The \text{Goal} is then called, and if it succeeds, then the client is allowed to connect. If \text{Goal} fails, then the server will close the stream and ignore the connection request.

A typical use of this feature would be to restrict the addresses of the clients allowed to connect. If you require bullet proof security, then you would probably need something more sophisticated.

Example:

\[
| ?- \text{linda([(Host:Port)-mypred(Host,Port),}
\]
\text{accept_hook(C,S,should_accept(C,S))]}).
\]
will call `mypred/2` when the server is started. `mypred/2` could start the client-processes, save the address for the clients etc. Whenever a client attempts to connect from a host with IP address `Addr`, a bi-directional socket stream `Stream` will be opened to the client, and `should_accept(Addr, Stream)` will be called to determine if the client should be allowed to connect.

### 10.23.2 Linda Client

The clients are one or more SICStus processes that have connection(s) to the server.

To load the package, enter the query

```prolog
| ?- use_module(library('linda/client')).
```

Some of the following predicates fail if they do not receive an answer from the Linda-server in a reasonable amount of time. That time is set with the predicate `linda_timeout/2`.

- `linda_client(+Address)`
  Establishes a connection to a Linda-server specified by `Address`. The `Address` is of the format `Host:PortNumber` as given by `linda/[0,1]`.
  It is not possible to be connected to two Linda-servers at the same time.
  This predicate can fail due to a timeout.

- `close_client`
  Closes the connection to the server.

- `shutdown_server/0`
  Sends a Quit signal to the server, which immediately stops accepting new connections before `shutdown_server/0` returns. The server continues running after receiving this signal, processing requests from existing clients, until such time as all the clients have closed their connections. It is up to the clients to tell each other to quit. When all the clients are done, the server stops (i.e. `linda/[0,1]` succeeds). Courtesy of Malcolm Ryan. Note that `close_client/0` should be called after `shutdown_server/0`. `shutdown_server/0` will raise an error if there is no connection between the client and the server.
  The behavior of `shutdown_server/0` changed in SICStus Prolog 4.2. In previous releases the server continued to accept new connections after being told to shutdown. Now it immediately stops listening for new connections and releases the listening socket and these server actions happens before the client returns from `shutdown_server/0`.

- `linda_timeout(?OldTime, ?NewTime)`
  This predicate controls Linda’s timeout. `OldTime` is unified with the old timeout and then timeout is set to `NewTime`. The value is either `off` or of the form `Seconds:Milliseconds`. The former value indicates that the timeout mechanism is disabled, that is, eternal waiting. The latter form is the `timeout-time`.

- `out(+Tuple)`
  Places the tuple `Tuple` in Linda’s tuple-space.
in(?Tuple)
Removes the tuple \textit{Tuple} from Linda’s tuple-space if it is there. If not, then the predicate blocks until it is available (that is, someone performs an \texttt{out/1}).

\texttt{in_noblock(?Tuple)}
Removes the tuple \textit{Tuple} from Linda’s tuple-space if it is there. If not, then the predicate fails.
This predicate can fail due to a timeout.

\texttt{in(+TupleList, ?Tuple)}
As \texttt{in/1} but succeeds when either of the tuples in \textit{TupleList} is available. \textit{Tuple} is unified with the fetched tuple. If that unification fails, then the tuple is \textit{not} reinserted in the tuple-space.

\texttt{rd(?Tuple)}
Succeeds if \textit{Tuple} is available in the tuple-space, suspends otherwise until it is available. Compare this with \texttt{in/1}: the tuple is \textit{not} removed.

\texttt{rd_noblock(?Tuple)}
Succeeds if \textit{Tuple} is available in the tuple-space, fails otherwise.
This predicate can fail due to a timeout.

\texttt{rd(+TupleList, ?Tuple)}
As \texttt{in/2} but does not remove any tuples.

\texttt{bagof_rd_noblock(?Template, +Tuple, ?Bag)}
\textit{Bag} is the list of all instances of \textit{Template} such that \textit{Tuple} exists in the tuple-space.
The behavior of variables in \textit{Tuple} and \textit{Template} is as in \texttt{bagof/3}. The variables could be existentially quantified with \texttt{^/2} as in \texttt{bagof/3}.
The operation is performed as an atomic operation.
This predicate can fail due to a timeout.

\textbf{Example}: Assume that only one client is connected to the server and that the tuple-space initially is empty.

\begin{verbatim}
| ?- out(x(a,3)), out(x(a,4)), out(x(b,3)), out(x(c,3)).
| ?- bagof_rd_noblock(C-N, x(C,N), L).

C = _32,
L = [a-3,a-4,b-3,c-3],
N = _52

| ?- bagof_rd_noblock(C, N^x(C,N), L).

C = _32,
L = [a,a,b,c],
N = _48
\end{verbatim}


10.24 List Operations—library(lists)

This library module provides operations on lists. Exported predicates:

select(\?Element, \?Set, \?Residue)

is true when \Set is a list, \Element occurs in \Set, and \Residue is everything in \Set except \Element (things stay in the same order).

selectchk(+Element, +Set, ?Residue)

is to select/3 what memberchk/2 is to member/2. That is, it locates the first occurrence of \Element in \Set, and deletes it, giving \Residue. It is steadfast in \Residue.

append(+ListOfLists, -List)

is true when \ListOfLists is a list \[L1,...,Ln\] of lists, \List is a list, and appending \L1, ..., \Ln together yields \List. \ListOfLists must be a proper list. Additionally, either \List should be a proper list, or each of \L1, ..., \Ln should be a proper list. The behavior on non-lists is undefined. \ListOfLists must be proper because for any given solution, infinitely many more can be obtained by inserting nils ([[]]) into \ListOfList. Could be defined as:

\[
\text{append(Lists, Appended) :-}
\begin{align*}
& \text{foreach(List,Lists),} \\
& \text{fromto(Appended,S0,S,[[]])} \\
& \text{do append(List, S, S0)}
\end{align*}
\]

append(?Prefix, ?Tail1, ?List1, ?Tail2, ?List2)

is true when append(\Prefix, \Tail1, \List1) and append(\Prefix, \Tail2, \List2) are both true. You could call append/3 twice, but that is order-dependent. This will terminate if \Prefix is a proper list or if either \List1 or \List2 is a proper list.

correspond(?X, ?Xlist, ?Ylist, ?Y)

is true when \Xlist and \Ylist are lists, \X is an element of \Xlist, \Y is an element of \Ylist, and \X and \Y are in similar places in their lists. No relation is implied between other elements of \Xlist and \Ylist. For a similar predicate without the cut, see select/4.

delete(+List, +Kill, -Residue)

is true when \List is a list, in which \Kill may or may not occur, and \Residue is a copy of \List with all elements equal to \Kill deleted. To extract a single copy of \Kill, use select(\Kill, \List, \Residue). \Kill and the elements of \List should be sufficiently instantiated for \= to be sound. Could be defined as:

\[
delete(List, Kill, Residue) :-
\begin{align*}
& \text{foreach(X,List),} \\
& \text{fromto(Residue,S0,S,[[]]),} \\
& \text{param(Kill)} \\
& \text{do (X = Kill -> S0 = S ; S0 = [X|S])}
\end{align*}
\]
delete(+List, +Kill, +Count, -Residue)
is true when List is a list, in which Kill may or may not occur, and Count is a
non-negative integer, and Residue is a copy of List with the first Count elements
equal to Kill deleted. If List has fewer than Count elements equal to Count,
all of them are deleted. Kill and the elements of List should be sufficiently
instantiated for \( \neq \) to be sound.

is_list(+List)
succeeds when List is a proper list. That is, List is nil ([]) or a cons cell
([Head|Tail]) whose Tail is a proper list. A variable, or a list whose final tail
is a variable, or a cyclic list, will fail this test.

keys_and_values(?[K1-V1,...,Kn-Vn], ?[K1,...,Kn], ?[V1,...,Vn])
is true when its arguments look like the picture above. It is meant for splitting
a list of Key-Value pairs (such as keysort/2 wants and produces) into separate
lists of Keys and of Values. It may just as well be used for building a list of
pairs from a pair of lists. In fact one usually wants just the keys or just the
values, but you can supply _ as the other argument. For example, suppose you
wanted to sort a list without having duplicates removed. You could do
keys_and_values(RawPairs, RawKeys, _),
keysort(RawPairs, OrdPairs),
keys_and_values(OrdPairs, OrdKeys, _).
Could be defined as:
keys_and_values([], [], []).  
keys_and_values([Key-Value|Pairs], [Key|Keys], [Value|Values]) :-
keys_and_values(Pairs, Keys, Values).

last(+List, -Last)
is true when List is a List and Last is its last element. There is also a
last(?Fore, ?Last, ?List) whose argument order matches append/3. This
could be defined as
last(L, X) :- append(_, [X], L).

nextto(?X, ?Y, ?List)
is true when X and Y appear side-by-side in List. It could be written as
nextto(X, Y, List) :- append(_, [X,Y|_], List).
It may be used to enumerate successive pairs from the list. List should be
proper, otherwise nextto/3 will generate it.

nth0(?N, ?List, ?Elem)
is true when Elem is the Nth member of List, counting the first as element 0.
That is, throw away the first N elements and unify Elem with the next. E.g.
nth0(0, [H|T], H). Either N should be an integer, or List should be proper.

nth1(?N, ?List, ?Element)
is true when Element is the Nth member of List, counting the first as element
1. That is, throw away the first N-1 elements and unify Element with the next
element (the Nth). E.g. nth1(1, [H|T], H). This is just like nth0/3 except
that it counts from 1 instead of 0. Either N should be an integer, or List should
be proper.
nth0(?N, ?List, ?Elem, ?Rest)
unifies Elem with the Nth element of List, counting from 0, and Rest with the other elements. It can be used to select the Nth element of List (yielding Elem and Rest), or to insert Elem before the Nth (counting from 0) element of Rest, when it yields List, e.g. nth0(2, List, c, [a,b,d,e]) unifies List with [a,b,c,d,e]. This can be seen as inserting Elem after the Nth element of Rest if you count from 1 rather than 0. Either N should be an integer, or List or Rest should be proper.

nth1(?N, ?List, ?Elem, ?Rest)
unifies Elem with the Nth element of List, counting from 1, and Rest with the other elements. It can be used to select the Nth element of List (yielding Elem and Rest), or to insert Elem before the Nth (counting from 1) element of Rest, when it yields List, e.g. nth1(2, List, b, [a,c,d,e]) unifies List with [a,b,c,d,e]. Either N should be an integer, or List or Rest should be proper.

one_longer(?Longer, ?Shorter)
is true when
\[
\text{length(Longer,N)}, \text{length(Shorter,M)}, \text{succ(M,N)}
\]
for some integers M, N. It was written to make \{nth0,nth1\}/4 able to find the index, just as same_length/2 is useful for making things invertible.

perm(+List, ?Perm)
is true when List and Perm are permutations of each other. The main use of perm/2 is to generate permutations. You should not use this predicate in new programs; use permutation/2 instead. List must be a proper list. Perm may be partly instantiated.

permutation(?List, ?Perm)
is true when List and Perm are permutations of each other. Unlike perm/2, it will work even when List is not a proper list. Any way, it works by generating permutations of List and unifying them with Perm. Be careful: this is quite efficient, but the number of permutations of an N-element list is N!, and even for a 7-element list that is 5040.

is true when \{A,B\} = \{C,D\}. It is very useful for writing pattern matchers over commutative operators.

proper_length(+List, ?Length)
succeeds when List is a proper list, binding Length to its length. That is, is_list(List), length(List, Length). Will fail for cyclic lists.

remove_dups(+List, ?Pruned)
removes duplicated elements from List, which should be a proper list. If List has non-ground elements, Pruned may contain elements which unify; two elements will remain separate iff there is a substitution which makes them different. E.g. [X,X] -> [X] but [X,Y] -> [X,Y]. The surviving elements, by ascending standard order, is unified with Pruned.
reverse(?List, ?Reversed)
  is true when List and Reversed are lists with the same elements but in opposite orders. Either List or Reversed should be a proper list: given either argument the other can be found. If both are incomplete reverse/2 can backtrack forever trying ever longer lists.

rev(+List, ?Reversed)
  is a version of reverse/2 which only works one way around. Its List argument must be a proper list whatever Reversed is. You should use reverse/2 in new programs, though rev/2 is faster when it is safe to use it.

same_length(?List1, ?List2)
  is true when List1 and List2 are both lists and have the same number of elements. No relation between the values of their elements is implied. It may be used to generate either list given the other, or indeed to generate two lists of the same length, in which case the arguments will be bound to lists of length 0, 1, 2, ... If either List1 or List2 is bound to a proper list, same_length is determinate and terminating.

same_length(?List1, ?List2, ?Length)
  is true when List1 and List2 are both lists, Length is a non-negative integer, and both List1 and List2 have exactly Length elements. No relation between the elements of the lists is implied. If Length is instantiated, or if either List1 or List2 is bound to a proper list, same_length is determinate and terminating.

select(?X, ?Xlist, ?Y, ?Ylist)
  is true when X is the Kth member of Xlist and Y the Kth element of Ylist for some K, and apart from that Xlist and Ylist are the same. You can use it to replace X by Y or vice versa. Either Xlist or Ylist should be a proper list.

selectchk(?X, +Xlist, ?Y, +Ylist)
  is to select/4 as memberchk/2 is to member/2. That is, it finds the first K such that X unifies with the Kth element of Xlist and Y with the Kth element of Ylist, and it commits to the bindings thus found. If you have Keys and Values in "parallel" lists, you can use this to find the Value associated with a particular Key (much better methods exist). Except for argument order, this is identical to correspond/4, but selectchk/4 is a member of a coherent family. Note that the arguments are like the arguments of memberchk/2, twice.

shorter_list(?Short, ?Long)
  is true when Short is a list is strictly shorter than Long. Long doesn’t have to be a proper list provided it is long enough. This can be used to generate lists shorter than Long, lengths 0, 1, 2... will be tried, but backtracking will terminate with a list that is one element shorter than Long. It cannot be used to generate lists longer than Short, because it doesn’t look at all the elements of the longer list.

subseq(?Sequence, ?SubSequence, ?Complement)
  is true when SubSequence and Complement are both subsequences of the list Sequence (the order of corresponding elements being preserved) and every element of Sequence which is not in SubSequence is in the Complement and vice versa.
That is, \( \text{length} (\text{Sequence}) = \text{length} (\text{SubSequence}) + \text{length} (\text{Complement}) \), e.g. \( \text{subseq}([1,2,3,4], [1,3,4], [2]) \). This was written to generate subsets and their complements together, but can also be used to interleave two lists in all possible ways.

**subseq0(+Sequence, ?SubSequence)**

is true when \( \text{SubSequence} \) is a subsequence of \( \text{Sequence} \), but may be \( \text{Sequence} \) itself. Thus \( \text{subseq0}([a,b], [a,b]) \) is true as well as \( \text{subseq0}([a,b], [a]) \). \( \text{Sequence} \) must be a proper list, since there are infinitely many lists with a given \( \text{SubSequence} \).

```prolog
?- setof(X, subseq0([a,b,c],X), Xs).
Xs = [[] , [a] , [a,b] , [a,b,c] , [a,c] , [b] , [b,c] , [c] ]
?- bagof(X, subseq0([a,b,c,d],X), Xs).
Xs = [['a,b,c,d],[b,c,d],[c,d],[d],[a]','[b]','[b,c]','[a,c],[a,b,d],[a,b],[a,b,c]']
```

**subseq1(+Sequence, ?SubSequence)**

is true when \( \text{SubSequence} \) is a proper subsequence of \( \text{Sequence} \), that is it contains at least one element less. \( \text{Sequence} \) must be a proper list, as \( \text{SubSequence} \) does not determine \( \text{Sequence} \).

**sumlist(+Numbers, ?Total)**

is true when \( \text{Numbers} \) is a list of integers, and \( \text{Total} \) is their sum. \( \text{Numbers} \) should be a proper list. Could be defined as:

```prolog
sumlist(Numbers, Total) :-
(  foreach(X,Numbers),
  fromto(0,S0,S,Total)
  do S is S0+X)
).
```

**transpose(?X, ?Y)**

is true when \( \text{X} \) is a list of the form \([X1, ..., X1m], ..., [Xn1, ..., Xnm] \) and \( \text{Y} \) is its transpose, that is, \( \text{Y} = ([X1, ..., Xn1], ..., [X1m, ..., Xnm]) \). We insist that both lists should have this rectangular form, so that the predicate can be invertible. For the same reason, we reject empty arrays with \( m = 0 \) or \( n = 0 \).


is true when

```prolog
append(Prefix, Suffix, List), length(Prefix, Length).
```

The normal use of this is to split a \( \text{List} \) into a \( \text{Prefix} \) of a given \( \text{Length} \) and the corresponding \( \text{Suffix} \), but it can be used any way around provided that \( \text{Length} \) is instantiated, or \( \text{Prefix} \) is a proper list, or \( \text{List} \) is a proper list.

**append_length(?Suffix, ?List, ?Length)**

is true when there exists a list \( \text{Prefix} \) such that \( \text{append_length}(\text{Prefix}, \text{Suffix}, \text{List}, \text{Length}) \) is true. When you don’t want to know the \( \text{Prefix} \), you should call this predicate, because it doesn’t construct the \( \text{Prefix} \) argument, which \( \text{append_length}/4 \) would do.
prefix_length(?List, ?Prefix, ?Length)
is true when
    prefix(List, Prefix) &
    length(Prefix, Length).

The normal use of this is to find the first Length elements of a given List, but it can be used any way around provided that Length is instantiated, or Prefix is a proper list, or List is a proper list. It is identical in effect to append_length(Prefix, _, List, Length).

proper_prefix_length(?List, ?Prefix, ?Length)
is true when
    proper_prefix(List, Prefix) &
    length(Prefix, Length).

The normal use of this is to find the first Length elements of a given List, but it can be used any way around provided that Length is instantiated, or Prefix is a proper list, or List is a proper list. It is logically equivalent to prefix(Prefix, List, Length), Length > 0.

suffix_length(+List, ?Suffix, ?Length)
is true when
    suffix(List, Suffix) &
    length(Suffix, Length).

The normal use of this is to return the last Length elements of a given List. For this to be sure of termination, List must be a proper list. The predicate suffix/2 has the same requirement. If Length is instantiated or Suffix is a proper list, this predicate is determinate.

proper_suffix_length(+List, ?Suffix, ?Length)
is true when
    proper_suffix(List, Suffix) &
    length(Suffix, Length).

The normal use of this is to return the last Length elements of a given List. For this to be sure of termination, List must be a proper list. The predicate proper_suffix/2 has the same requirement. If Length is instantiated or Suffix is a proper list, this predicate is determinate.

rotate_list(+Amount, ?List, ?Rotated)
is true when List and Rotated are lists of the same length, and
    append(Prefix, Suffix, List) &
    append(Suffix, Prefix, Rotated) &
    (  Amount >= 0 & length(Prefix, Amount)
        | Amount =< 0 & length(Suffix, Amount) )

That is to say, List rotated LEFT by Amount is Rotated. If either List or Rotated is bound to a proper list, rotate_list is determinate.
rotate_list(?List, ?Rotated)
is true when rotate_list(1, List, Rotated), but is a bit less heavy-handed.
rotate_list(X, Y) rotates X left one place yielding Y. rotate_list(Y, X) rotates X right one place yielding Y. Either List or Rotated should be a proper list, in which case rotate_list is determinate and terminating.

sublist(+Whole, ?Part, ?Before, ?Length)
sublist(+Whole, ?Part, ?Before)
is true when
- Whole is a list – it must be proper already
- Part is a list
- Whole = Alpha || Part || Omega
- length(Alpha, Before)
- length(Part, Length)
- length(Omega, After)

cons(?Head, ?Tail, ?List)
is true when Head is the head of List and Tail is its tail. i.e. append([Head], Tail, List). No restrictions.

last(?Fore, ?Last, ?List)
is true when Last is the last element of List and Fore is the list of preceding elements, e.g. append(Fore, [Last], List). Fore or Last should be proper. It is expected that List will be proper and Fore unbound, but it will work in reverse too.

head(?List, ?Head)
is true when List is a non-empty list and Head is its head. A list has only one head. No restrictions.

tail(?List, ?Tail)
is true when List is a non-empty list and Tail is its tail. A list has only one tail. No restrictions.

prefix(?List, ?Prefix)
is true when List and Prefix are lists and Prefix is a prefix of List. It terminates if either argument is proper, and has at most N+1 solutions. Prefixes are enumerated in ascending order of length.

proper_prefix(?List, ?Prefix)
is true when List and Prefix are lists and Prefix is a proper prefix of List. That is, Prefix is a prefix of List but is not List itself. It terminates if either argument is proper, and has at most N solutions. Prefixes are enumerated in ascending order of length.

suffix(?List, ?Suffix)
is true when List and Suffix are lists and Suffix is a suffix of List. It terminates only if List is proper, and has at most N+1 solutions. Suffixes are enumerated in descending order of length.
proper_suffix(?List, ?Suffix)
is true when List and Suffix are lists and Suffix is a proper suffix of List. That is, Suffix is a suffix of List but is not List itself. It terminates only if List is proper, and has at most N solutions. Suffixes are enumerated in descending order of length.

segment(?List, ?Segment)
is true when List and Segment are lists and Segment is a segment of List. That is, List = _ <> Segment <> _. Terminates only if List is proper. If Segment is proper it enumerates all solutions. If neither argument is proper, it would have to diagonalize to find all solutions, but it doesn’t, so it is then incomplete. If Segment is proper, it has at most N+1 solutions. Otherwise, it has at most (1/2)(N+1)(N+2) solutions.

proper_segment(?List, ?Segment)
is true when List and Segment are lists and Segment is a proper segment of List. It terminates only if List is proper. The only solution of segment/2 which is not a solution of proper_segment/2 is segment(List,List). So proper_segment/2 has one solution fewer.

cumlist(:Pred, +[X1,...,Xn], ?V0, ?[V1,...,Vn])
cumlist(:Pred, +[X1,...,Xn], +[Y1,...,Yn], ?V0, ?[V1,...,Vn])
cumlist(:Pred, +[X1,...,Xn], +[Y1,...,Yn], +[Z1,...,Zn], ?V0, ?[V1,...,Vn])
cumlist/4 maps a ternary predicate Pred down the list [X1,...,Xn] just as scanlist/4 does, and returns a list of the results. It terminates when the lists runs out. If Pred is bidirectional, it may be used to derive [X1,...,Xn] from V0 and [V1...Vn], e.g. cumlist(plus, [1,2,3,4], 0, /* -> */ [1,3,6,10]) and cumlist(plus, [1,1,1,1], /* <- */ 0, [1,2,3,4]). Could be defined as:
cumlist(Pred, Xs, V0, Cum) :-
  ( foreach(X,Xs),
    foreach(V,Cum),
    fromto(V0,V1,V, _),
    param(Pred)
    do call(Pred,X,V1, V)
  ).

cumlist(Pred, Xs, Ys, V0, Cum) :-
  ( foreach(X,Xs),
    foreach(Y,Ys),
    foreach(V,Cum),
    fromto(V0,V1,V, _),
    param(Pred)
    do call(Pred,X,Y,V1, V)
  ).

cumlist(Pred, Xs, Ys, Zs, V0, Cum) :-
  ( foreach(X,Xs),
    foreach(Y,Ys),
    foreach(Z,Zs),
    foreach(V,Cum),
    fromto(V0,V1,V, _),
    param(Pred)
    do call(Pred,X,Y,Z,V1, V)
  ).

maplist(:Pred, +List)
succeeds when \( \text{Pred}(X) \) succeeds for each element \( X \) of \( \text{List} \). \( \text{List} \) should be a proper list. Could be defined as:

\[
\text{maplist}(\text{Pred}, \text{Xs}) :-
( \text{foreach}(X,\text{Xs}),
  \text{param}(\text{Pred})
  \text{do call}(\text{Pred}, X)
).
\]

maplist(:Pred, +OldList, ?NewList)
succeeds when \( \text{Pred}(\text{Old}, \text{New}) \) succeeds for each corresponding \( \text{Old} \) in \( \text{OldList} \), \( \text{New} \) in \( \text{NewList} \). Either \( \text{OldList} \) or \( \text{NewList} \) should be a proper list. Could be defined as:

\[
\text{maplist}(\text{Pred}, \text{Xs}, \text{Ys}) :-
( \text{foreach}(X,\text{Xs}),
  \text{foreach}(Y,\text{Ys}),
  \text{param}(\text{Pred})
  \text{do call}(\text{Pred}, X, Y)
).
\]
maplist(:Pred, +Xs, ?Ys, ?Zs)
is true when Xs, Ys, and Zs are lists of equal length, and Pred(X, Y, Z) is true for corresponding elements X of Xs, Y of Ys, and Z of Zs. At least one of Xs, Ys, and Zs should be a proper list. Could be defined as:

\[
\text{maplist}(\text{Pred}, \text{Xs}, \text{Ys}, \text{Zs}) :- \\
\quad (\text{foreach}(\text{X}, \text{Xs}), \\
\quad \text{foreach}(\text{Y}, \text{Ys}), \\
\quad \text{foreach}(\text{Z}, \text{Zs}), \\
\quad \text{param}(\text{Pred}) \\
\quad \text{do call}(\text{Pred}, \text{X}, \text{Y}, \text{Z}) \\
\).
\]

map_product(Pred, Xs, Ys, PredOfProduct)
Just as maplist(P, Xs, L) is the analogue of Miranda’s
\[
\text{let } L = [P \ x \mid x \leftarrow Xs]
\]
so map_product(P, Xs, Ys, L) is the analogue of Miranda’s
\[
\text{let } L = [P \ x \ y \mid x \leftarrow Xs; y \leftarrow Ys]
\]
That is, if Xs = [X1,...,Xm], Ys = [Y1,...,Yn], and P(Xi,Yj,Zij), L = [Z11,...,Z1n,Z21,...,Z2n,...,Zm1,...,Zmn]. It is as if we formed the Cartesian product of Xs and Ys and applied P to the (Xi,Yj) pairs. Xs and Ys should be proper lists. Could be defined as:

\[
\text{map_product}(\text{Pred}, \text{Xs}, \text{Ys}, \text{Zs}) :- \\
\quad (\text{foreach}(\text{X}, \text{Xs}), \\
\quad \text{fromto}(\text{Zs}, \text{S0}, \text{S}, []), \\
\quad \text{param}([\text{Ys}, \text{Pred}]) \\
\quad \text{do (} \\
\quad \quad \text{foreach}(\text{Y}, \text{Ys}), \\
\quad \quad \text{fromto}(\text{S0}, [\text{Z} | \text{S1}], \text{S1}, \text{S}), \\
\quad \quad \text{param}([\text{X}, \text{Pred}]) \\
\quad \quad \text{do call}(\text{Pred}, \text{X}, \text{Y}, \text{Z}) \\
\quad \quad ) \\
\).
\]

scanlist(:Pred, [X1,...,Xn], ?V1, ?V)
scanlist(:Pred, [X1,...,Xn], [Y1,...,Yn], ?V1, ?V)
scanlist(:Pred, [X1,...,Xn], [Y1,...,Yn], [Z1,...,Zn], ?V1, ?V)

scanlist/4 maps a ternary relation Pred down a list. The computation is Pred(X1,V1,V2), Pred(X2,V2,V3), ..., Pred(Xn,Vn,V) So if Pred is plus/3, scanlist(plus, [X1,...,Xn], 0, V) puts the sum of the list elements in V. Note that the order of the arguments passed to Pred is the same as the order of the arguments following Pred. This also holds for scanlist/5 and scanlist/6, e.g. scanlist(Pred, Xs, Ys, Zs, V1, V) calls Pred(X3,Y3,Z3,V3,V4). Could be defined as:
scanlist(Pred, Xs, V0, V) :-
  ( foreach(X,Xs),
    fromto(V0,V1,V2,V),
    param(Pred)
  do call(Pred, X, V1, V2)
  ).

scanlist(Pred, Xs, Ys, V0, V) :-
  ( foreach(X,Xs),
    foreach(Y,Ys),
    fromto(V0,V1,V2,V),
    param(Pred)
  do call(Pred, X, Y, V1, V2)
  ).

scanlist(Pred, Xs, Ys, Zs, V0, V) :-
  ( foreach(X,Xs),
    foreach(Y,Ys),
    foreach(Z,Zs),
    fromto(V0,V1,V2,V),
    param(Pred)
  do call(Pred, X, Y, Z, V1, V2)
  ).

some(:Pred, +List)
succeeds when Pred(Elem) succeeds for some Elem in List. It will try all ways of proving Pred for each Elem, and will try each Elem in the List. somechk/2 is to some/2 as memberchk/2 is to member/2.

member(X,L) <- some(=(X), L).
memberchk(X, L) <- somechk(=(X), L).
some(Pred,L) <- member(X, L), call(Pred,X).

This acts on backtracking like member/2; List should be a proper list.

some(:Pred, +[X1,...,Xn], ?[Y1,...,Yn])
is true when Pred(Xi, Yi) is true for some i.

some(:Pred, +[X1,...,Xn], ?[Y1,...,Yn], ?[Z1,...,Zn])
is true when Pred(Xi, Yi, Zi) is true for some i.

somechk(:Pred, +[X1,...,Xn])
is true when Pred(Xi) is true for some i, and it commits to the first solution it finds (like memberchk/2).

somechk(:Pred, +[X1,...,Xn], ?[Y1,...,Yn])
is true when Pred(Xi, Yi) is true for some i, and it commits to the first solution it finds (like memberchk/2).

somechk(:Pred, +[X1,...,Xn], ?[Y1,...,Yn], ?[Z1,...,Zn])
is true when Pred(Xi, Yi, Zn) is true for some i, and it commits to the first solution it finds (like memberchk/2).
convlist(:Rewrite, +OldList, ?NewList)
is a sort of hybrid of maplist/3 and include/3. Each element of NewList
is the image under Rewrite of some element of OldList, and order is
preserved, but elements of OldList on which Rewrite is undefined (fails)
are not represented. Thus if foo(K,X,Y) :- integer(X), Y is X+K. then
convlist(foo(1), [1,a,0,joe(99),101], [2,1,102]). OldList should be a
proper list. Could be defined as:

\[
\text{convlist}(\text{Pred}, \text{Xs}, \text{News}) :-
( \text{foreach}(X, \text{Xs}),
  \text{fromto}(\text{News}, S0, S, []),
  \text{param}(\text{Pred})
  \text{do} (\text{call}(\text{Pred}, X, N) \rightarrow S0 = [N|S] ; S0 = S)
).\]

exclude(:Pred, +Xs, ?SubList)
exclude(:Pred, +Xs, +Ys, ?SubList)
exclude(:Pred, +Xs, +Ys, +Zs, ?SubList)

succeeds when SubList is the sublist of Xs containing all the elements Xi[,Yi[,Zi]]
for which \text{Pred}(Xi[,Yi[,Zi]]) is \text{false}. That is, it removes all the elements satisfying \text{Pred}. Xs, Ys or Zs should be a proper list. Could be defined as:

\[
\text{exclude}(\text{Pred}, \text{Xs}, \text{News}) :-
( \text{foreach}(X, \text{Xs}),
  \text{fromto}(\text{News}, S0, S, []),
  \text{param}(\text{Pred})
  \text{do} (\text{call}(\text{Pred}, X) \rightarrow S0 = S ; S0 = [X|S])
).\]

\[
\text{exclude}(\text{Pred}, \text{Xs}, \text{Ys}, \text{News}) :-
( \text{foreach}(X, \text{Xs}),
  \text{foreach}(Y, \text{Ys}),
  \text{fromto}(\text{News}, S0, S, []),
  \text{param}(\text{Pred})
  \text{do} (\text{call}(\text{Pred}, X, Y) \rightarrow S0 = S ; S0 = [X|S])
).\]

\[
\text{exclude}(\text{Pred}, \text{Xs}, \text{Ys}, \text{Zs}, \text{News}) :-
( \text{foreach}(X, \text{Xs}),
  \text{foreach}(Y, \text{Ys}),
  \text{foreach}(Z, \text{Zs}),
  \text{fromto}(\text{News}, S0, S, []),
  \text{param}(\text{Pred})
  \text{do} (\text{call}(\text{Pred}, X, Y, Z) \rightarrow S0 = S ; S0 = [X|S])
).\]
include(:Pred, +Xs, ?SubList)
include(:Pred, +Xs, +Ys, ?SubList)
include(:Pred, +Xs, +Ys, +Zs, ?SubList)
succeeds when SubList is the sublist of Xs containing all the elements Xi,Yi,Zi for which Pred(Xi,Yi,Zi) is true. That is, it retains all the elements satisfying Pred. Xs, Ys or Zs should be a proper list. Could be defined as:

\[
\text{include(Pred, Xs, News)} \leftarrow \begin{array}{l}
\text{foreach}(X,Xs), \\
\text{fromto}(News,S0,\text{nil},[]) , \\
\text{param}(Pred) \\
\text{do (call(Pred,X) \rightarrow S0 = [X|S] ; S0 = S)} \\
\end{array}
\]

\[
\text{include(Pred, Xs, News)} \leftarrow \begin{array}{l}
\text{foreach}(X,Xs), \\
\text{fromto}(News,S0,\text{nil},[]) , \\
\text{param}(Pred) \\
\text{do (call(Pred,X) \rightarrow S0 = [X|S] ; S0 = S)} \\
\end{array}
\]

\[
\text{include(Pred, Xs, Ys, News)} \leftarrow \begin{array}{l}
\text{foreach}(X,Xs), \\
\text{foreach}(Y,Ys), \\
\text{fromto}(News,S0,\text{nil},[]) , \\
\text{param}(Pred) \\
\text{do (call(Pred,X,Y) \rightarrow S0 = [X|S] ; S0 = S)} \\
\end{array}
\]

\[
\text{include(Pred, Xs, Ys, Zs, News)} \leftarrow \begin{array}{l}
\text{foreach}(X,Xs), \\
\text{foreach}(Y,Ys), \\
\text{foreach}(Z,Zs), \\
\text{fromto}(News,S0,\text{nil},[]) , \\
\text{param}(Pred) \\
\text{do (call(Pred,X,Y,Z) \rightarrow S0 = [X|S] ; S0 = S)} \\
\end{array}
\]

\[
\text{partition(:Pred, +List, ?Less, ?Equal, ?Greater)}
\]
is a relative of include/3 and exclude/3 which has some pretensions to being logical. For each X in List, we call Pred(X,R), and route X to Less, Equal, or Greater according as \(R\) is \(<, =, \text{or} >\).

\[
\text{group(:Pred, +List, ?Front, ?Back)}
\]
is true when append(Front, Back, List), maplist(Pred, Front), and Front is as long as possible.
group(:Pred, +Key, +List, ?Front, ?Back)
  is true when append(Front, Back, List), maplist(call(Pred,Key), Front), and Front is as long as possible. Strictly speaking we don’t need it; group(call(Pred,Key), List, Front, Back) would do just as well.

group(:Pred, +List, ?ListOfLists)
  is true when append(ListOfLists, List), each element ofListOfLists has the form [Head|Tail] such that group(Pred, Head, Tail, Tail, [])], and each element ofListOfLists is as long as possible. For example, if you have a keysorted list, and define same_key(K_-, K_-), then group(same_key, List, Buckets) will divide List up into Buckets of pairs having the same key.

ordered(+List)
  is true when List is a list of terms [T1,T2,...,Tn] such that for all k in 2..n Tk-1 @=< Tk, i.e. T1 @=< T2 @=< T3 ... The output of keysort/2 is always ordered, and so is that of sort/2. Beware: just because a list is ordered does not mean that it is the representation of an ordered set; it might contain duplicates.

ordered(+P, +[T1,T2,...,Tn])
  is true when P(T1,T2) & P(T2,T3) & ... That is, if you take P as a "comparison" predicate like @=<, the list is ordered. This is good for generating prefixes of sequences, e.g. L = [1,_,_,_,_], ordered(times(2), L) yields L = [1,2,4,8,16].

max_member(?Xmax, +[X1,...,Xn])
  unifies Xmax with the maximum (in the sense of @=<) of X1,...,Xn. The list should be proper. If it is empty, the predicate fails quietly. Could be defined as:

  max_member(Maximum, [Head|Tail]) :-
    (    foreach(X,Tail),
      fromto(Head,M0,M,Maximum)
    do (X@=<M0 -> M = M0 ; M = X)
    )

min_member(?Xmin, +[X1,...,Xn])
  unifies Xmin with the minimum (in the sense of @=<) of X1,...,Xn. The list should be proper. If it is empty, the predicate fails quietly. Could be defined as:

  min_member(Minimum, [Head|Tail]) :-
    (    foreach(X,Tail),
      fromto(Head,M0,M,Minimum)
    do (M0@=<X -> M = M0 ; M = X)
    ).

max_member(:P, ?Xmax, +[X1,...,Xn])
  unifies Xmax with the maximum element of [X1,...,Xn], as defined by the comparison predicate P, which should act like @=< . The list should be proper. If it is empty, the predicate fails quietly. Could be defined as:
max_member(Pred, Maximum, [Head|Tail]) :-
  ( foreach(X,Tail),
    fromto(Head,M0,M,Maximum),
    param(Pred)
    do (call(Pred,X,M0) -> M = M0 ; M = X)
  ).

min_member(:P, ?Xmin, +[X1,...,Xn])
unifies Xmin with the minimum element of [X1,...,Xn], as defined by the comp-
parison predicate P, which should act like @=<. The list should be proper. If
it is empty, the predicate fails quietly. Could be defined as:

min_member(Pred, Minimum, [Head|Tail]) :-
  ( foreach(X,Tail),
    fromto(Head,M0,M,Minimum),
    param(Pred)
    do (call(Pred,M0,X) -> M = M0 ; M = X)
  ).

select_min(?Element, +Set, ?Residue)
unifies Element with the smallest (in the sense of @=<) element of Set, and
Residue with a list of all the other elements.

select_min(:Pred, ?Element, +Set, ?Residue)
find the least Element of Set, i.e. Pred(Element,X) for all X in Set.

select_max(?Element, +Set, ?Residue)
unifies Element with the (leftmost) maximum element of the Set, and Residue
to the other elements in the same order.

select_max(:Pred, ?Element, +Set, ?Residue)
find the greatest Element of Set, i.e. Pred(X,Element) for all X in Set.

increasing_prefix(?Sequence, ?Prefix, ?Suffix)
is true when append(Prefix, Suffix, Sequence) and Prefix, together with
the first element of Suffix, forms a monotone non-decreasing sequence, and no
longer Prefix will do. Pictorially,

Sequence = [x1,...,xm,xm+1,...,xn]
Prefix = [x1,...,xm]
Suffix = [xm+1,...,xn]
x1 <= x2 <= ... <= xm <= xm+1
not xm+1 <= (xm+2

This is perhaps a surprising definition; you might expect that the first element
of Suffix would be included in Prefix. However, this way, it means that if
Sequence is a strictly decreasing sequence, the Prefix will come out empty.

increasing_prefix(:Order, ?Sequence, ?Prefix, ?Suffix)
is the same as increasing_prefix/3, except that it uses the binary relation
Order in place of @=<.
decreasing_prefix(?Sequence, ?Prefix, ?Suffix)
decreasing_prefix(:Order, ?Sequence, ?Prefix, ?Suffix)
is the same, except it looks for a decreasing prefix. The order is the converse of the given order. That is, where increasing_prefix/[3,4] check X(R)Y, these routines check Y(R)X.

clumps(+Items, -Clumps)
is true when Clumps is a list of lists such that
• append(Clumps, Items)
• for each Clump in Clumps, all the elements of Clump are identical (==)

Items must be a proper list of terms for which sorting would have been sound. In fact, it usually is the result of sorting.

keyclumps(+Pairs, ?Clumps)
is true when Pairs is a list of pairs and Clumps a list of lists such that
• append(Clumps, Pairs)
• for each Clump in Clumps, all of the Key-Value pairs in Clump have identical (==) Keys.

Pairs must be a proper list of pairs for which keysorting would have been sound. In fact, it usually is the result of keysorting.

clumped(+Items, ?Counts)
is true when Counts is a list of Item-Count pairs such that if clumps(Items, Clumps), then each Item-Count pair in Counts corresponds to an element [Item/*1*/,...,Item/*Count*/] of Clumps. Items must be a proper list of terms for which sorting would have been sound. In fact, it usually is the result of sorting.

keyclumped(+Pairs, ?Groups)
is true when Pairs is a list of Key-Item pairs and Groups is a list of Key-Items pairs such that if keyclumps(Pairs, Clumps), then for each K-[I1,...,In] pair in Groups there is a [K-I1,...,K-In] clump in Clumps. Pairs must be a proper list of pairs for which keysorting would have been sound. In fact, it usually is the result of keysorting.

10.25 External Storage of Terms (LMDB)—library(lmdb)

This library module handles storage and retrieval of terms on files. By using indexing, the store/retrieve operations are efficient also for large data sets. The package is an interface to the Lightning Memory-Mapped Database Manager (LMDB). The API is quite similar to the deprecated library(bdb), and provides essentially the same functionality, although some details differ. The library module is part of SICStus Prolog since release 4.7.0.

10.25.1 Basics
The idea is to get a behavior similar to assert/1, retract/1 and clause/2, but the terms are stored on file instead of in primary memory. Like the Prolog database, LMDB will store variables with attributes or with blocked goals as ordinary variables.
The differences compared with the Prolog database are:

- A database must be explicitly created, and occupies a file system directory. A database must be opened before any access and closed after the last access.
- The functors and the indexing specifications of the terms to be stored have to be given when the database is created. It is possible to index on other parts of the term than just the functor and first argument. (see Section 10.25.7 [LMDB DB-Spec Details], page 635).
- Changes are not guaranteed to affect the database until it is explicitly synced.

10.25.2 Current Limitations

- The terms are not necessarily fetched in the same order as they were stored.
- The number of terms ever inserted in a database cannot exceed $2^{32}-1$.
- A size limit must be set at creation time. An LMDB database cannot grow beyond that limit.

10.25.3 Lightning Memory-Mapped Database Manager (LMDB)

LMDB is a software library that provides a transactional database in the form of a key-value store. LMDB stores arbitrary key/data pairs as byte arrays, has a range-based search capability, and supports multiple data items for a single key. LMDB is fully thread-aware and supports concurrent read/write access from multiple processes and threads.

For details on the underlying technology, we refer to the OpenLDAP home page [https://www.openldap.org/](https://www.openldap.org/).

10.25.4 The DB-Spec—Informal Description

The db-spec defines which functors are allowed and which parts of a term are used for indexing in a database. The syntax of a db-spec is a skeletal goal with no module. The db-spec is a list of atoms and compound terms where the arguments are either + or -. A term can be inserted in the database if there is a spec in the spec list with the same functor.

Multilevel indexing is not supported, terms have to be “flattened”.

Every spec with the functor of the indexed term specifies an indexing. Every argument where there is a + in the spec is indexed on.

The idea of the db-spec is illustrated with a few examples. (A section further down explains the db-spec in a more formal way).

Given a spec of \([f(+,-), , (+,-), g, f(-,+)]\) the indexing works as follows. (The parts with indexing are underlined.)

<table>
<thead>
<tr>
<th>Term</th>
<th>Store</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>g(x,y)</td>
<td>domain error</td>
<td>domain error</td>
</tr>
<tr>
<td>f(A,B)</td>
<td>f(A,B)</td>
<td>instantiation error</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>f(a,b)</td>
<td>f(a,b)</td>
<td>f(a,b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f(a,b)</td>
</tr>
</tbody>
</table>
The specification \([f(\text{-}, \text{-}), f(\text{-}, \text{+})]\) is different from \([f(\text{+}, \text{+})]\). The first specifies that two indices are to be made whereas the second specifies that only one index is to be made on both arguments of the term.

### 10.25.5 Predicates

#### 10.25.5.1 Conventions

The following conventions are used in the predicate descriptions below.

- The predicates that create and open databases can be given a list of options. An **Option** is one of:
  
  **mapsize**\(\,(S)\)
  
  The mapsize, or size limit, of the database being created, in MiB. The default is 10MiB.

  **permission**\(\,(S)\)
  
  The UNIX permission for files of the database being created. Ignored for Windows. The default is octal 664.

  **mode**\(\,(S)\)
  
  The access mode for the database being opened. One of the atoms **read** and **write**. If **read** (the default), no updates are allowed.

- **DBRef** is a reference to an open database. It is returned when the database is opened. The reference becomes invalid after the database has been closed. The reference corresponds to the LMDB environment notion. An environment physically consists of a directory containing the files needed to enable sharing databases between processes.

- **TermRef** is a reference to a term in a given database. The reference is returned when a term is stored. The reference stays valid even after the database has been closed and hence can be stored permanently as part of another term. However, if such references are stored in the database, automatic compression of the database (using **lmdb_compress/[2,3]**) is not possible, in that case the user has to write their own compressing predicate.

- **SpecList** is a description of the indexing scheme; see Section 10.6.7 [The DB-Spec], page 411.

- **Term** is any Prolog term.

- **Iterator** is a non-backtrackable mutable object. It can be used to iterate through a set of terms stored in a database. The iterators are unidirectional.

#### 10.25.5.2 Memory Leaks

In order to avoid memory leaks, databases and iterators should always be closed explicitly. Consider using **lmdb_with_db/[2,3]** and **lmdb_with_iterator/3** to automate the closing/deallocation of these objects.
Please note: a database must not be closed while there are outstanding choices for some lmdb_fetch/3 or lmdbEnumerate/3 goal that refers to that database. Outstanding choices can be removed with a cut (!).

10.25.5.3 The Predicates

\texttt{lmdb_create(+Path, +SpecList)}

\texttt{lmdb_create(+Path, +SpecList, +Options)}

Creates a brand new LMDB database with the given spec list. \texttt{Path} should be a file system path where a new directory can be created. If some file or directory already exists there, an exception is raised.

\texttt{lmdb_open(+Path, -DBRef)}

\texttt{lmdb_open(+Path, -DBRef, +Options)}

Opens an existing database and unifies \texttt{DBRef} with its reference. \texttt{Path} should be a file system path to the directory that holds the database.

\textbf{Please note:} Only one \texttt{DBRef} may be connected to the same \texttt{Path} at the same time in the same process. I.e., you must not call \texttt{lmdb_open/2} twice with the same \texttt{Path} without closing the \texttt{DBRef} before the second call.

\texttt{lmdb_with_db(+Path, :Goal)}

\texttt{lmdb_with_db(+Path, :Goal, +Options)}

Opens an existing database, calls \texttt{Goal} with its reference, as if by \texttt{call(Goal, DBRef)}, and ensures that the database is closed afterwards. \texttt{Goal} may backtrack and succeed multiple times. \texttt{Path} should be a file system path to the directory that holds the database.

\textbf{Please note:} see \texttt{lmdb_open/2} for restrictions on opening the same \texttt{Path} more than once at the same time.

\texttt{lmdb_close(+DBRef)}

Closes the database referenced by \texttt{DBRef}.

\texttt{lmdb_store(+DBRef, +Term, -TermRef)}

Stores \texttt{Term} in the database \texttt{DBRef}. \texttt{TermRef} is unified with a corresponding term reference. The functor of \texttt{Term} must match the functor of a spec in the db-spec associated with \texttt{DBRef}.

\texttt{lmdb_fetch(+DBRef, ?Term, ?TermRef)}

Unifies \texttt{Term} with a term from the database \texttt{DBRef}. At the same time, \texttt{TermRef} is unified with a corresponding term reference. Backtracking over the predicate unifies with all terms matching \texttt{Term}.

If \texttt{TermRef} is not instantiated, both the functor and the instantiatedness of \texttt{Term} must match a spec in the db-spec associated with \texttt{DBRef}.

If \texttt{TermRef} is instantiated, the referenced term is read and unified with \texttt{Term}.

If you simply want to find all matching terms, it is more efficient to use \texttt{lmdb_findall/5} or \texttt{lmdbEnumerate/3}.

\texttt{lmdb_erase(+DBRef, +TermRef)}

Deletes the term from the database \texttt{DBRef} that is referenced by \texttt{TermRef}. 
lmdb_enumerate(+DBRef, ?Term, -TermRef)
Unifies Term with a term from the database DBRef. At the same time, TermRef is unified with a corresponding term reference. Backtracking over the predicate unifies with all terms matching Term.
Implemented by linear search—the db-spec associated with DBRef is ignored.

lmdb_findall(+DBRef, +Template, +Term, :Goal, -Bag)
Unifies Bag with the list of instances of Template in all proofs of Goal found when Term is unified with a matching term from the database DBRef. Both the functor and the instantiatedness of Term must match a spec in the db-spec associated with DBRef. Conceptually, this predicate is equivalent to findall(Template, (lmdb_fetch(DBRef, Term, _), Goal), Bag).

lmdb_compress(+DBRefFrom, +DBRefTo)
Copies the database given by DBRefFrom to a new database given by DBRefTo. The new database will be a compressed version of the first one in the sense that it will not have “holes” resulting from deletion of terms. Deleted term references will also be reused, which implies that references that refer to terms in the old database will be invalid in the new one.
The terms of DBRefFrom will be appended to any existing terms of DBRefTo. Of course, DBRefTo must have an indexing specification that enables the terms in DBRefFrom to be inserted into it.

lmdb_sync(+DBRef)
Flushes any cached information from the database referenced by DBRef to stable storage.

lmdb_make_iterator(+DBRef, +Term, -Iterator)
Creates a new iterator and unifies it with Iterator. The iterator iterates through the terms that would be found by lmdb_fetch(DBRef, Term, _).
Every iterator created by lmdb_make_iterator/3 must be destroyed with lmdb_iterator_done/1.

lmdb_with_iterator(+DBRef, +Term, :Goal)
Creates a new iterator, calls Goal with it, as if by call(Goal, Iterator), and ensures that the iterator is destroyed afterwards. Goal may backtrack and succeed multiple times. See the documentation of lmdb_make_iterator/3.

lmdb_iterator_next(+Iterator, -Term, -TermRef)
Iterator advances to the next term, and Term and TermRef are unified with the term and its reference pointed to by Iterator. If there is no next term, the predicate fails.

lmdb_iterator_done(+Iterator)
Deallocates Iterator, which must not be in use anymore.

lmdb_property(+DBRef, -Property)
Unifies Property with one of the properties of the database given by DBRef, which are:

path(P) The file system path of the database.
speclist(L)
    The SpecList with which the database was created.

mapsize(S)
    The mapsize, or size limit, of the database.

mode(S)  The access mode with which the database was opened. One of the atoms read and write.

Enumerates all four properties on backtracking.

lmdb_export(+DBRef, +ExportFile)
    Exports the database given by DBRef to the text file ExportFile. ExportFile can be imported by lmdb_import/2.

lmdb_import(+DBRef, +ImportFile)
    Imports the text file ImportFile into the database given by DBRef.
    The ImportFile should have been written by lmdb_export/2 or by the corresponding functionality in library(bdb).

10.25.6 Example Session

This example creates a database, opens it, and add some values to it. Finally the values are read.
Chapter 10: The Prolog Library

| ?- lmdb_create(tempdb, [a(+,-)]).
| ?- lmdb_open(tempdb, R, [mode(write)]), assertz(tempdb(R)).
  R = '$lmdb'(4611936272) ?

| ?- tempdb(R), lmdb_store(R, a(b,1), T).
  R = '$lmdb'(4611936272),
  T = '$db_termref'(0) ?

| ?- tempdb(R), lmdb_store(R, a(c,2), T).
  R = '$lmdb'(4611936272),
  T = '$db_termref'(1) ?

| ?- tempdb(R), lmdb_store(R, a(b,X), T).
  R = '$lmdb'(4611936272),
  T = '$db_termref'(2) ?

| ?- tempdb(R), lmdb_enumerate(R, Term, _).
  R = '$lmdb'(4611936272),
  Term = a(b,1) ? ;
  R = '$lmdb'(4611936272),
  Term = a(c,2) ? ;
  R = '$lmdb'(4611936272),
  Term = a(b,_A) ? ;

| ?- tempdb(R), lmdb_findall(R, Term, Term, ground(Term), Bag).
  R = '$lmdb'(4611936272),
  Bag = [a(b,1),a(c,2)] ?

| ?- retract(tempdb(R)), lmdb_close(R).
  R = '$lmdb'(4611936272) ?

Later, in another (or the same) process, the stored data can be read.

| ?- lmdb_open(tempdb, R, [mode(read)]), assertz(tempdb(R)).
  R = '$lmdb'(4560556048) ?

| ?- tempdb(R), lmdb_findall(R, Term, Term, ground(Term), Bag).
  R = '$lmdb'(4560556048),
  Bag = [a(b,1),a(c,2)] ?

| ?- retract(tempdb(R)), lmdb_close(R).
  R = '$lmdb'(4560556048) ?

10.25.7 DB-Spec—Details

A db-spec has the form of a SpecList:
\textit{speclist} = [\textit{spec1}, \ldots, \textit{specM}]
\textit{spec} = \textit{functor} (\textit{argspec1}, \ldots, \textit{argspecN})
\textit{argspec} = + | -

where \textit{functor} is a Prolog atom. The case \(N = 0\) is allowed.

A spec \(F(\textit{argspec1}, \ldots, \textit{argspecN})\) is \textit{applicable} to any nonvar term with principal functor \(F/N\).

When storing a term \(T\), we generate a hash code for every applicable spec in the \(\text{db}\)-spec, and a reference to \(T\) is stored with each of them. (More precisely, with each element of the set of generated hash codes). If \(T\) contains nonvar elements on each \(+\) position in the spec, then the hash code depends on each of these elements. If \(T\) does contain some variables on \(+\) position, then the hash code depends only on the functor of \(T\).

When fetching a term \(Q\), we look for an applicable spec for which there are no variables in \(Q\) on positions marked \(+\). If no applicable spec can be found, a domain error is raised. If no spec can be found where on each \(+\) position a nonvar term occurs in \(Q\), an instantiation error is raised. Otherwise, we choose the spec with the most \(+\) positions, breaking ties by choosing the leftmost one.

The terms that contain nonvar terms on every \(+\) position will be looked up using indexing based on the principal functor of the term and the principal functor of terms on \(+\) positions. The other (more general) terms will be looked up using an indexing based on the principal functor of the term only.

\textbf{10.25.8 Exporting and Importing a Database}

Since the database format of LMDB may change from version to version, it may become necessary to migrate a database when upgrading. Two predicates are provided for this purpose: \texttt{lmdb_export/2} and \texttt{lmdb_import/2}.

To ease migration from the legacy \texttt{library(bdb)} library, files exported with \texttt{bdb:db_export/[2,3]} can be imported into the LMDB format using \texttt{lmdb_import/2}.

\textbf{10.26 Sparse Array Operations—library(logarr)}

This library module provides extendible arrays with logarithmic access time. \textbf{Please note}: the atom \$ is used to indicate an unset element, and the functor \$/4 is used to indicate a subtree. In general, array elements whose principal function symbol is \$ will not work.

Exported predicates:
\begin{itemize}
  \item new_array(-\(A\)) returns a new empty array \(A\).
  \item is_array(+\(A\)) checks whether \(A\) is an array.
\end{itemize}
alist(+Array, -List)
    returns a list of pairs Index-Element of all the elements of Array that have been set.
aref(+Index, +Array, -Element)
    unifies Element with Array[Index], or fails if Array[Index] has not been set.
arefa(+Index, +Array, -Element)
    is as aref/3, except that it unifies Element with a new array if Array[Index] is undefined. This is useful for multidimensional arrays implemented as arrays of arrays.
arefl(+Index, +Array, -Element)
    is as aref/3, except that Element appears as [] for undefined cells.
aset(+Index, +Array, +Element, -NewArray)
    unifies NewArray with the result of setting Array[Index] to Element.

10.27 Mutable, Dense Array Operations—library(mutarray)

Prolog does not have an array type. Instead, one must use lists or trees, with logarithmic access time to the elements at best. This package provides operations on dense, mutable arrays, of elements indexed from 1 and up. It uses an array representation, which admits direct access to the elements and constant-time access to any element. Update operations also take constant time, except under some circumstances when the entire array must be copied, in which case such operations take linear time. Following are the exported predicates, where a mutarray is a mutable (see Section 4.8.9 [ref-lte-mut], page 131), the value of which is accessed and modified by the operations of this package.

is_mutarray(+Array)
    succeeds when Array is a mutarray.
list_to_mutarray(+Items, -Array)
    where Items should be a list of values. Array is unified with a new mutarray populated by the given values.
new_mutarray(-Array)
    unifies Array with a new mutarray of zero length.
new_mutarray(-Array, +Size)
    where Size should be a nonnegative integer. Array is unified with a new mutarray of the given size populated by brand new variables.
mutarray_length(+MutArray, -Length)
    Length is unified in constant time with the length of MutArray.
mutarray_set_length(+MutArray, +Length)
    sets the new length of mutarray MutArray to Length. If the MutArray becomes shorter, this runs in constant time in the best case. Otherwise, new brand new variables are added to the end of the array in linear time.
mutarray_last(+MutArray, -Last)
    Last is unified in constant time with the last element of MutArray. Merely fails if MutArray is of zero length.
mutarray_to_list(+MutArray, -List)
   List is unified with the list of elements of MutArray.

mutarray_gen(+MutArray, -N, -Element)
   is true when Element is the N:\text{th} element of the given MutArray. Iterates over all elements by increasing N.

mutarray_get(+MutArray, +N, -Element)
   unifies Element in constant time with the N:\text{th} element of MutArray.

mutarray_put(+MutArray, +N, +Element)
   sets the N:\text{th} element of MutArray to Element.

mutarray_update(+MutArray, +N, -OldElement, +NewElement)
   unifies OldElement with the N:\text{th} element of MutArray, and sets that element to NewElement.

mutarray_append(+MutArr, +MutArrOrValues)
   where MutArrOrValues should be a mutarray or a list of values, which is appended to the end of MutArr.

10.28  Mutable Dictionary Operations—library(mutdict)

Prolog does not have a dictionary type. Instead, one can use, e.g., AVL trees, with logarithmic access time. This package provides operations on mutable dictionaries. Such a \textit{mutdict} is an unordered collection of items that consist of a key and a value. The only restrictions on items is that the key must be ground. The given \textit{mutdict} can hold at most one item for any given key. It uses a hash table representation, which admits table lookup in constant time, if there are no hash collisions. Update operations also take constant time, except under some circumstances when the entire table must be copied, in which case such operations take linear time. In addition, the hash function takes time linear in the size of the key, and so simple keys are more efficient than compound terms. A \textit{mutdict} is a \textit{mutable} (see Section 4.8.9 [ref-lte-mut], page 131), the value of which is accessed and modified by the operations of this package. Following are the exported predicates:

\textit{is_mutdict}(+Dict)
   succeeds when Dict is a mutdict.

\textit{list_to_mutdict}(+Items, -Dict)
   unifies Dict with a new mutdict populated by Items, each of which should be of the form Key-Value.

\textit{new_mutdict}(-Dict)
   unifies Dict with a new mutdict.

\textit{mutdict_empty}(+Dict)
   succeeds when Dict is an empty mutdict.

\textit{mutdict_size}(+Dict, -Size)
   unifies Size with the number of elements of the mutdict Dict.

\textit{mutdict_gen}(+Dict, -Key, -Value)
   succeeds when Dict is a mutdict where one item consists of Key and Value. Iterates over all items, but in no specific order.
mutdict_get(+Dict, +Key, -Value)
    If there is an item of the mutdict with key Key, then its value is unified with Value. Otherwise, the goal fails.

mutdict_put(+Dict, +Key, +Value)
    If there is an item of the mutdict with key Key, then its value is simply replaced by Value; otherwise, adds the item consisting of Key and Value to the given mutdict.

mutdict_update(+Dict, +Key, -OldValue, +NewValue)
    If there is an item of the mutdict with key Key, then its value is unified with OldValue and replaced by NewValue. Otherwise, the goal fails.

mutdict_delete(+Dict, +Key)
    If there is an item of the mutdict with key Key, then it is deleted; otherwise, the goal merely succeeds.

mutdict_clear(+Dict)
    All items are deleted from the mutdict.

mutdict_keys(+Dict, -Keys)
    Keys is unified with the list of keys of mutdict. The list comes in no specific order, and is free of duplicates.

mutdict_values(+Dict, -Values)
    Values is unified with the list of values of mutdict. The list comes in no specific order, and can contain duplicates.

mutdict_items(+Dict, -Items)
    Items is unified with the list of items of mutdict. Each item is of the form Key-Value. The list comes in no specific order, and is free of duplicates.

mutdict_append(+Dict, +DictOrItems)
    where DictOrItems should be a mutdict or a list of items. For each item of DictOrItems of the form Key-Value, if there is an item of the mutdict with key Key, then its value is simply replaced by Value; otherwise, the item is added to the given mutdict.

10.29 The Objects Package—library(objects)

The SICStus Objects package enables programmers to write object-oriented programs in SICStus Prolog. The objects in SICStus Objects are modifiable data structures that provide a clean and efficient alternative to storing data in the Prolog database.

10.29.1 Introduction

The SICStus Objects package enables programmers to write object-oriented programs in SICStus Prolog. The objects in SICStus Objects are modifiable data structures that provide a clean and efficient alternative to storing data in the Prolog database.

This user’s guide is neither an introduction to object-oriented programming nor an introduction to SICStus Prolog. A number of small, sample programs are described in this manual, and some larger programs are in the demo directory.
10.29.1.1 Using SICStus Objects

One of the basic ideas of object-oriented programming is the encapsulation of data and procedures into objects. Each object belongs to exactly one class, and an object is referred to as an instance of its class. A class definition determines the following things for its objects:

- slots, where an object holds data
- messages, the commands that can be sent to an object
- methods, the procedures the object uses to respond to the messages

All interaction with an object is by sending it messages. The command to send a message to an object has the form

\[ \text{Object MessageOp Message} \]

where Object is an object, MessageOp is one of the message operators (‘<<’, ‘>>’, or ‘<-’) and Message is a message defined for the object’s class. Roughly speaking, the ‘>>’ message operator is used for extracting information from an object, ‘<<’ is for storing information into an object, and ‘<-’ is for any other sort of operation.

For example, using the point class defined in the next section, it would be possible to give the following command, which demonstrates all three message operators.

```
| ?- create(point, PointObj),
  PointObj >> x(InitX),
  PointObj >> y(InitY),
  PointObj << x(2.71828),
  PointObj << y(3.14159),
  PointObj <- print(user_output),
  nl(user_output).

(2.71828,3.14159)
PointObj = point(23461854),
InitX = 1.0,
InitY = 2.0
```

First it binds the variable PointObj to a newly created point object. Then, the two get messages (sent with the ‘>>’ operator) fetch the initial values of the point’s x and y slots, binding the variables InitX and InitY to these values. Next, the two put messages (sent with the ‘<<’ operator) assign new values to the object’s x and y slots. Finally, the send message (sent with the ‘<-’ operator) instructs the point object to print itself to the user_output stream, followed by a newline. Following the goal, we see the point has been printed in a suitable form. Following this, the values of PointObj, InitX, and InitY are printed as usual for goals entered at the top-level prompt.

Because this goal is issued at the top-level prompt, the values of the variables PointObj, InitX and InitY are not retained after the command is executed and their values are displayed, as with any goal issued at the top-level prompt. However, the point object still
exists, and it retains the changes made to its slots. Hence, objects, like clauses asserted to the Prolog database, are more persistent than Prolog variables.

Another basic idea of object-oriented programming is the notion of inheritance. Rather than defining each class separately, a new class can inherit the properties of a more general superclass. Or, it can be further specialized by defining a new subclass, which inherits its properties. (C++ uses the phrase “base class” where we use “superclass.” It also uses “derived class” where we use “subclass.”)

SICStus Objects uses term expansion to translate object-oriented programs into ordinary Prolog. (This is the same technique that Prolog uses for its DCG grammar rules.) As much as possible is done at compile time. Class definitions are used to generate Prolog clauses that implement the class’s methods. Message commands are translated into calls to those Prolog clauses. And, inheritance is resolved at translation time.

SICStus Objects consists of two modules, obj_decl and objects. The obj_decl module is used at compile time to translate the object-oriented features of SICStus Objects. Any file that defines classes or sends messages should include the command

```
:- load_files(library(obj_decl),
              [when(compile_time), if(changed)]).
```

The objects module provides runtime support for SICStus Objects programs. A file that sends messages or asks questions about what classes are defined or to what class an object belongs should include the command:

```
:- use_module(library(objects)).
```

You will probably include both in most files that define and use classes.

**Please note:** A file that loads `library(obj_decl)` currently cannot recursively load another file that loads `library(obj_decl)`, because that would confuse the internal database being used by the package.

If you use the foreign resource linker, splfr, on a Prolog file that uses the objects package, then you must pass it the `--objects` option. This will make splfr understand the package’s syntax extensions.

### 10.29.1.2 Defining Classes

A class definition can restrict the values of any slot to a particular C-style type. It can specify whether a slot is private (the default, meaning that it cannot be accessed except by that methods of that class), protected (like private, except that the slot can also be accessed by subclasses of the class), or public (meaning get and put methods for the slot are generated automatically), and it can specify an initial value. The class definition also may contain method clauses, which determine how instances of the class will respond to messages. A class definition may also specify one or more superclasses and which methods are to be inherited.
The point object created in the previous example had two floating point slots, named \( x \) and \( y \), with initial values of 1.0 and 2.0, respectively. As we have seen, the point class also defined put and get methods for \( x \) and \( y \), as well as a send method for printing the object. The put and get methods for \( x \) and \( y \) can be automatically generated simply by declaring the slots public, but the print method must be explicitly written. In addition, in order to be able to create instances of this class, we must define a create method, as explained in Section 10.29.2.3 [obj-scl-meth], page 647. We also provide a second create method, taking two arguments, allowing us to specify an \( x \) and \( y \) value when we first create a point object.

\[
\begin{align*}
\text{:- class} \ \text{point} &= \ [
\text{public } x: \text{float} = 1.0, \\
\text{public } y: \text{float} = 2.0].
\end{align*}
\]

\[
\text{Self } \leftarrow \text{create}.
\]

\[
\text{Self } \leftarrow \text{create}(X, Y) \leftarrow \\
\quad \text{Self } \ll x(X), \\
\quad \text{Self } \ll y(Y).
\]

\[
\text{Self } \leftarrow \text{print}(\text{Stream}) \leftarrow \\
\quad \text{Self } \gg x(X), \\
\quad \text{Self } \gg y(Y), \\
\quad \text{format}(\text{Stream}, '(~w,~w)', (X, Y)).
\]

\[
\text{:- end_class point}.
\]

The variable name Self in these clauses is arbitrary—any variable to the left of the message operator in the head of a method clause refers to the instance of the class receiving the message.

### 10.29.1.3 Using Classes

Given this definition, the following command creates an instance of the point class, assigning values to its \( x \) and \( y \) slots, and prints a description of the point.

\[
\begin{align*}
| \text{?- create(point(3,4), PointObj),} \\
\quad \text{PointObj } \leftarrow \text{print(user_output).}
\end{align*}
\]

The print message prints \((3.0,4.0)\). The variable PointObj is bound to a Prolog term of the form

\[
\text{point(Address)}
\]

where Address is essentially a pointer to the object.

In general, an object belonging to a class ClassName will be represented by a Prolog term of the form

\[
\text{point(Address)}
\]
ClassName(Address)

The name ClassName must be an atom. This manual refers to such a term as if it were the object, not just a pointer to the object. Users are strongly discouraged from attempting to do pointer arithmetic with the address.

After execution of this command, the point object still exists, but the variable PointObj can no longer be used to access it. So, while objects resemble clauses asserted into the Prolog database in their persistence, there is no automatic way to search for an object. Objects are not automatically destroyed when they are no longer needed. And, there is no automatic way to save an object from one Prolog session to the next. It is the responsibility of the programmer to keep track of objects, perhaps calling the destroy/1 predicate for particular objects that are no longer needed or asserting bookkeeping facts into the Prolog database to keep track of important objects.

10.29.1.4 Looking Ahead

The next few sections of this manual describe the SICStus Objects package in greater detail. In particular, they describe how to define classes, their methods and their slots, and how to reuse class definitions via inheritance. Small sample programs and program fragments are provided for most of the features described.

Experienced Prolog programmers may choose to skip over these sections and look at the sample programs in this package’s demo directory, referring to the reference pages as necessary. Everyone is encouraged to experiment with the sample programs before writing their own programs.

10.29.2 Simple Classes

This section is about simple classes that inherit nothing—neither slots nor methods—from more general superclasses. Everything about these classes is given directly in their definitions, so they are the best starting point for programming with SICStus Objects.

The use of inheritance in defining classes is described in the next section. Classes that inherit properties from superclasses are called derived classes in some systems, such as C++.

In general, the use of inheritance extends the properties of the simple classes in this section.

10.29.2.1 Scope of a Class Definition

A simple class definition begins with a statement of the form

    :- class ClassName = [SlotDef, ...].

The class’s slots are described in the list of SlotDef terms. It is possible, though not often useful, to define a class with no slots, by specifying the empty list. In that case the ‘=’ and the list may be omitted.

The class’s methods are defined following the class/1 directive, by Prolog clauses. Most of this section is about defining and using methods.

The class definition ends with any of the following:
```prolog
:- end_class ClassName.
or
:- end_class.
or the next class/1 directive or the end of the file. The ClassName argument to end_class/1 must match the class name in the corresponding class/1 directive. It is not possible to nest one class definition inside another.

10.29.2.2 Slots
A slot description has the form

Visibility SlotName:SlotType = InitialValue

where Visibility and ' = InitialValue' are optional. Each slot of a class must have a distinct name, given by the atom SlotName. The Visibility, SlotType and InitialValue parts of the slot description are described separately.

Visibility
A slot’s visibility is either private, protected, or public. If its visibility is not specified, then the slot is private. The following example shows all four possibilities:

```prolog
:- class example = [w:integer,
                    private x:integer,
                    protected y:integer,
                    public  z:integer]
```

Slot z is public, y is protected, and both x and w are private.

Direct access to private slots is strictly limited to the methods of the class. Any other access to such slots must be accomplished through these methods. Making slots private will allow you later to change how you represent your class, adding and removing slots, without having to change any code that uses your class. You need only modify the methods of the class to accommodate that change. This is known as information hiding.

Protected slots are much like private slots, except that they can also be directly accessed by subclasses. This means that if you wish to modify the representation of your class, then you will need to examine not only the class itself, but also its subclasses.

Public slots, in contrast, can be accessed from anywhere. This is accomplished through automatically generated get and put methods named for the slot and taking one argument. In the example above, our example class would automatically support a get and put method named z/1. Note, however, that unlike other object oriented programming languages that support them, public slots in SICStus Objects do not violate information hiding. This is because you may easily replace a public slot with your own get and put methods of the same name. In this sense, a public slot is really only a protected slot with automatically generated methods to fetch and store its contents.
Within a method clause, any of the class’s slots can be accessed via the `fetch_slot/2` and `store_slot/2` predicates. These are the only way to access private and protected slots. They may be used to define get and put methods for the class, which provide controlled access to the protected slots. But, they can only be used within the method clauses for the class, and they can only refer to slots of the current class and protected and public slots of superclasses.

In the slot description, `public`, `protected` and `private` are used as prefix operators. The `obj_decl` module redefines the prefix operator `public`, as follows:

```prolog
:- op(600, fy, [public]).
```

Unless you use the obsolete `public/1` directive in your Prolog programs, this should cause no problems.

**Types**

A slot’s type restricts the kinds of values it may contain. The slot is specified in the slot description by one of the following Prolog terms with the corresponding meaning. Most of these will be familiar, but the last four, `address`, `term`, `Class` and `pointer(Type)`, require some additional explanation:

- `integer` signed integer, large enough to hold a pointer
- `integer_64` 64-bit signed integer *since release 4.3*
- `integer_32` 32-bit signed integer
- `integer_16` 16-bit signed integer
- `integer_8` 8-bit signed integer
- `unsigned` unsigned integer, large enough to hold a pointer
- `unsigned_64` 64-bit unsigned integer *since release 4.3*
- `unsigned_32` 32-bit unsigned integer
- `unsigned_16` 16-bit unsigned integer
- `unsigned_8` 8-bit unsigned integer
- `float` 64-bit floating point number
- `float_32` 32-bit floating point number
- `atom` Prolog atom
address  Pointer. The address type is intended for use with foreign code. A slot of this type might store an address returned from a foreign function. That address might, in turn, be used in calling another foreign function. Hence, most Prolog programmers can safely ignore this type.

term  Prolog term. The term type is for general Prolog terms. Such a slot can hold any of the other types. However, if you know a slot will be used to hold only values of a particular type, then it is more efficient to specify that type in the class definition.

Storing a term containing free variables is similar to asserting a clause containing free variables into the Prolog database. The free variables in the term are replaced with new variables in the stored copy. And, when you fetch the term from the slot, you are really fetching a copy of the term, again with new variables.

Class  where Class is the name of a defined class. The class type is for any object in a class defined with SICStus Objects. Such a slot holds an object of its class or one of that class’s descendants, or the null object.

pointer(Type)  where Type is an atom. The pointer type is intended for use with the Structs Package. It is similar to the address type, except that access to this slot yields, and update to this slot expects, a term of arity 1 whose functor is Type and whose argument is the address. Again, most Prolog programmers can safely ignore this type.

Initial Values

A slot description may optionally specify an initial value for the slot. The initial value is the value of the slot in every instance of the class, when the object is first created. The initial value must be a constant of the correct type for the slot.

If an initial value is not specified, then a slot is initialized to a value that depends on its type. All numbers are initialized to 0, of the appropriate type. Atom and term slots are initialized to the empty atom (""). Addresses and pointers are initialized to null pointers. And, objects are initialized to the null object.

More complicated initialization—not the same constant for every instance of the class—must be performed by create methods, which are described later.

The null object

The null object is a special object that is not an instance of any class, but that can be stored in a slot intended for any class of object. This is very much like the NULL pointer in C. This is useful when you do not yet have an object to store in a particular slot.

In Prolog, the null is represented by the atom null.

Note that because the null object is not really an object of any class, you cannot determine its class with class_of/2. Unless noted otherwise, when we write of an object in this document, we do not include the null object.
10.29.2.3 Methods

Some methods are defined by method clauses, between the class/1 directive and the end of the class’s definition. Others are generated automatically. There are three kinds of messages in SICStus Objects, distinguished by the message operator they occur with:

‘>>’ A get message, which is typically used to fetch values from an object’s slots.

‘<<’ A put message, which is typically used to store values in an object’s slots.

‘<-’ A send message, which is used for other operations on or involving an object.

SICStus Objects automatically generates some get and put methods. And, it expects particular message names with the send operator for create and destroy methods. For the most part, however, you are free to use any message operators and any message names that seem appropriate.

A method clause has one of these message operators as the principal functor of its head. Its first argument, written to the left of the message operator, is a variable. By convention, we use the variable Self. Its second argument, written to the right of the message operator, is a term whose functor is the name of the message and whose arguments are its arguments.

For example, in the class whose definition begins as follows, a 0-argument send message named increment is defined. No parentheses are needed in the clause head, because the precedence of the ‘<-’ message operator is lower than that of the ‘:-’ operator.

```prolog
:- class counter = [public count:integer = 0].

Self <- increment :-
    Self >> count (X0),
    X1 is X0 + 1,
    Self << count (X1).
```

Its definition uses the automatically generated get and put methods for the public slot count.

It may look as though this technique is directly adding clauses to the >>/2, <</2 and <-/2 predicates, but the method clauses are transformed by term expansion, at compile time. However, the method clauses have the effect of extending the definitions of those predicates.

Methods are defined by Prolog clauses, so it is possible for them to fail, like Prolog predicates, and it is possible for them to be nondeterminate, producing multiple answers, upon backtracking. The rest of this section describes different kinds of methods.

Get and Put Methods

Get and put methods are generated automatically for each of a class’s public slots. These are 1-argument messages, named after the slots.

In the point class whose definition begins with
:- class point =
    [public x:float=0,
     public y:float=0].

the get and put methods are automatically generated for the \(x\) and \(y\) slots. If the class defines a `create/0` method, then the command

\[
| ?- create(point, PointObj),
    PointObj >> x(OldX),
    PointObj >> y(OldY),
    PointObj << x(3.14159),
    PointObj << y(2.71828).
\]

creates a point object and binds both \(\text{OldX}\) and \(\text{OldY}\) to 0.0\(E+00\), its initial slot values. Then, it changes the values of the \(x\) and \(y\) slots to 3.14159 and 2.71828, respectively. The variable `PointObj` is bound to the point object.

It is possible, and sometimes quite useful, to create get and put methods for slots that do not exist. For example, it is possible to add a polar coordinate interface to the point class by defining get and put methods for \(r\) and \(\theta\), even though there are no \(r\) and \(\theta\) slots. The get methods might be defined as follows:

\[
\text{Self} >> r(R) :-
    \text{Self} >> x(X),
    \text{Self} >> y(Y),
    R \equiv \sqrt{X*X + Y*Y}.
\]

\[
\text{Self} >> \theta(T) :-
    \text{Self} >> x(X),
    \text{Self} >> y(Y),
    T \equiv \tan(Y/X).
\]

The put methods are left as an exercise.

In the rational number class whose definition begins with:

\[
:- \text{class rational} =
    [\text{public num:integer},
     \text{public denom:integer}].
\]

get and put methods are automatically generated for the \(\text{num}\) and \(\text{denom}\) slots. It might be reasonable to add a get method for `float`, which would provide a floating point approximation to the rational in response to that get message. This is left as an exercise.

It is also possible to define get and put methods that take more than one argument. For example, it would be useful to have a put method for the point class that sets both slots of a point object. Such a method could be defined by
Self << point(X,Y) :-
    Self << x(X),
    Self << y(Y).

Similarly, a 2-argument get method for the rational number class might be defined as

Self >> (N/D) :-
    Self >> num(N),
    Self >> denom(D).

Note that the name of the put message is */2/, and that the parentheses are needed because of the relative precedences of the ‘>>’ and ‘/’ operators.

Put messages are used to store values in slots. Get messages, however, may be used either to fetch a value from a slot or to test whether a particular value is in a slot. For instance, the following command tests whether the do_something/2 predicate sets the point object’s x and y slots to 3.14159 and 2.71828, respectively.

| ?- create(point, PointObj),
    do_something(PointObj),
    PointObj >> x(3.14159),
    PointObj >> y(2.71828). |

The fetch_slot/2 predicate can similarly be used to test the value of a slot.

The effects of a put message (indeed, of any message) are not undone upon backtracking. For example, the following command fails:

| ?- create(point, PointObj),
    PointObj << x(3.14159),
    PointObj << y(2.71828),
    fail. |

But, it leaves behind a point object with x and y slots containing the values 3.14159 and 2.71828, respectively. In this, storing a value in an object’s slot resembles storing a term in the Prolog database with assert/1.

Some care is required when storing Prolog terms containing unbound variables in term slots. For example, given the class definition that begins with

:- class prolog_term = [public p_term:term].

Self <- create.

the following command would succeed:
\[ \text{?- create(prolog_term, TermObj),} \]
\[ \text{TermObj << p_term(foo(X,Y)),} \]
\[ \text{X = a,} \]
\[ \text{Y = b,} \]
\[ \text{TermObj >> p_term(foo(c,d)).} \]

The reason is that the free variables in \( \text{foo(X,Y)} \) are renamed when the term is stored in the \text{prolog_term} object’s \text{p_term} slot. This is similar to what happens when such a term is asserted to the Prolog database:

\[ \text{?- retractall(foo(_,_)),} \]
\[ \text{assert(foo(X,Y)),} \]
\[ \text{X = a,} \]
\[ \text{Y = b,} \]
\[ \text{foo(c,d).} \]

However, this goal would fail, because \text{c} and \text{d} cannot be unified:

\[ \text{?- create(prolog_term, TermObj),} \]
\[ \text{TermObj << p_term(foo(X,X)),} \]
\[ \text{TermObj >> p_term(foo(c,d)).} \]

**Direct Slot Access**

Get and put methods are not automatically generated for private and protected slots. Those slots are accessed by the \text{fetch_slot/2} and \text{store_slot/2} predicates, which may only appear in the body of a method clause and which always operate on the object to which the message is sent. It is not possible to access the slots of another object with these predicates.

You may declare a slot to be private or protected in order to limit access to it. However, it is still possible, and frequently useful, to define get and put methods for such a slot.

For example, if numerator and denominator slots of the rational number class were private rather than public, then it would be possible to define put methods to ensure that the denominator is never 0 and that the numerator and denominator are relatively prime. The get methods merely fetch slot values, but they need to be defined explicitly, since the slots are private. The new definition of the rational number class might start as follows:
:- class rational =
    [num:integer=0,
     denom:integer=1].

Self >> num(N) :-
    fetch_slot(num, N).

Self >> denom(D) :-
    fetch_slot(denom, D).

Self >> (N/D) :-
    Self >> num(N),
    Self >> denom(D).

One of the put methods for the class might be

Self << num(NO) :-
    fetch_slot(denom, DO)
    reduce(NO, DO, N, D),
    store_slot(num, N),
    store_slot(denom, D).

where the reduce/4 predicate would be defined to divide NO and DO by their greatest common divisor, producing N and D, respectively.

The definition of reduce/4 and the remaining put methods is left as an exercise. The put methods should fail for any message that attempts to set the denominator to 0.

Send Methods

Messages that do something more than fetch or store slot values are usually defined as send messages. While the choice of message operators is (usually) up to the programmer, choosing them carefully enhances the readability of a program.

For example, print methods might be defined for the point and rational number classes, respectively, as

    Self <- print(Stream) :-
    Self >> x(X),
    Self >> y(Y),
    format(Stream, "(\(\text{~w,~w}\)), [X, Y]).

and

    Self <- print(Stream) :-
    fetch_slot(num, N),
    fetch_slot(denom, D),
    format(Stream, "\(-\text{w/\~w}\), [N, D]).
These methods are used to access slot values. But, the fact that the values are printed to an output stream makes it more reasonable to define them as send messages than get messages.

Frequently send methods modify slot values. For example, the point class might have methods that flip points around the x and y axes, respectively:

```prolog
Self <- flip_x :-
    Self >> y(Y0),
    Y1 is -1 * Y0,
    Self << y(Y1).

Self <- flip_y :-
    Self >> x(X0),
    X1 is -1 * X0,
    Self << x(X1).
```

And, the rational number class might have a method that swaps the numerator and denominator of a rational number object. It fails if the numerator is 0.

```prolog
Self <- invert :-
    fetch_slot(num, N)
    N =\= 0,
    fetch_slot(denom, D)
    store_slot(num, D),
    store_slot(denom, N).
```

These methods modify slot values, but they do not simply store values that are given in the message. Hence, it is more reasonable to use the send operator.

It is possible for a method to produce more than one answer. For example, the class whose definition begins with

```prolog
:- class interval =
    [public lower:integer,
     public upper:integer].
```

might define a send method

```prolog
Self <- in_interval(X) :-
    Self >> lower(L),
    Self >> upper(U),
    between(L, U, X).
```

which uses the `between/3` predicate from `library(between)`. The `in_interval` message will bind X to each integer, one at a time, between the lower and upper slots, inclusive. It fails if asked for too many answers.

The rest of this section describes particular kinds of send messages.
Create and Destroy Methods

Objects are created with the `create/2` predicate. When you define a class, you must specify all the ways that instances of the class can be created. The simplest creation method is defined as

```
Self <- create.
```

If this method were defined for `Class`, then the command

```
| ?- create(Class, Object).
```

would create an instance of `Class` and bind the variable `Object` to that instance. All slots would receive their (possibly default) initial values.

More generally, if the definition for `Class` contains a create method

```
Self <- create(Arguments) :-
  Body.
```

then the command

```
| ?- create(Class(Arguments), Object).
```

will create an instance of `Class` and execute the `Body` of the create method, using the specified `Arguments`. The variable `Object` is bound to the new instance.

If a simple class definition has no create methods, then it is impossible create instances of the class. While the absence of create methods may be a programmer error, that is not always the case. Abstract classes, which are classes that cannot have instances, are often quite useful in defining a class hierarchy.

Create methods can be used to initialize slots in situations when specifying initial slot values will not suffice. (Remember that initial values must be specified as constants at compile time). The simplest case uses the arguments of the create message as initial slot values. For example, the definition of the point class might contain the following create method.

```
Self <- create(X,Y) :-
  Self << x(X),
  Self << y(Y).
```

If used as follows

```
| ?- create(point(3.14159, 2.71828), PointObj),
   PointObj >> x(X),
   PointObj >> y(Y).
```

then it would give `X` and `Y` the values of 3.14159 and 2.71828, respectively.
In some cases, the create method might compute the initial values. The following (partial) class definition uses the `date/1` predicate from `library(date)` to initialize its year, month and day slots.

```prolog
:- class date_stamp =
    [year:integer,
     month:integer,
     day:integer].

Self <- create :-
    date(date(Year, Month, Day)),
    store_slot(year, Year),
    store_slot(month, Month),
    store_slot(day, Day).
```

All three slots are private, so it will be necessary to define get methods in order to retrieve the time information. If no put methods are defined, however, then the date cannot be modified after the `date_stamp` object is created (unless some other method for this class invokes `store_slot/2` itself).

Create methods can do more than initialize slot values. Consider the `named_point` class, whose definition begins as follows:

```prolog
:- class named_point =
    [public name:atom,
     public x:float=1,
     public y:float=0].

Self <- create(Name, X, Y) :-
    Self << name(Name),
    Self << x(X),
    Self << y(Y),
    assert(name_point(Name, Self)).
```

Not only does the `create/3` message initialize the slots of a new `named_point` object, but it also adds a `name_point/2` fact to the Prolog database, allowing each new object to be found by its name. (This create method does not require the `named_point` object to have a unique name. Defining a `uniq_named_point` class is left as an exercise.)

An object is destroyed with the `destroy/1` command. Unlike `create/2`, `destroy/1` does not require that you define a `destroy` method for a class. However, `destroy/1` will send a destroy message (with no arguments) to an object before it is destroyed, if a `destroy` method is defined for the object’s class.

If a `named_point` object is ever destroyed, then the address of the object stored in this `name_point/2` fact is no longer valid. Hence, there should be a corresponding destroy method that retracts it.
Self <- destroy :-
    Self >> name(Name),
    retract(name_point(Name, Self)).

Similar create and destroy methods can be defined for objects that allocate their own separate memory or that announce their existence to foreign code.

**Instance Methods**

Instance methods allow each object in a class to have its own method for handling a specified message. For example, in a push-button class it would be convenient for each instance (each push-button) to have its own method for responding to being pressed.

The declaration

```prolog
:- instance_method Name/Arity, ....
```

inside a class definition states that the message Name/Arity supports instance methods. If the class definition defines a method for this message, then it will be treated as a default method for the message.

The define_method/3 predicate installs a method for an object of the class, and the undefine_method/3 predicate removes that method.

Suppose that the date_stamp class, defined earlier, declared an instance method to print the year of a date_stamp instance.

```prolog
:- instance_method print_year/1.

Self <- print_year(Stream) :-
    Self >> year(Y0),
    Y1 is Y0 + 1970,
    format(Stream, "~d", [Y1]).
```

The arithmetic is necessary because UNIX dates are based on January 1, 1970.

If a particular date_stamp object’s date were to be printed in Roman numerals, then it could be given a different print_year method, using the define_method/3 predicate.

```prolog
| ?- create(date_stamp, DateObj),
   define_method(DateObj,
     print_year(Stream),
     print_roman_year(Stream, DateObj)).
```

If this date_stamp object is created in 1994, then a print_year message sent to it would print the current year as

```
MCMXCVI
```
Defining the predicate `print_roman_year/2` is left as an exercise. It must be able to access the `year` slot of a `date_stamp` object. Because it is not defined by a method clause within the class definition, `print_roman_year/2` cannot use the `get_slot/2` predicate.

None of `instance_method/1`, `define_method/3`, `undefine_method/3` specify a message operator. Instance methods can only be defined for send messages.

10.29.3 Inheritance

This section describes the additional features (and the additional complexity) of defining classes with inheritance in SICStus Objects. Most of what was said about classes in the previous section remains true in these examples.

10.29.3.1 Single Inheritance

The simplest case is when a new class inherits some properties (slots and methods) from a single superclass. That superclass may, in turn, be defined in terms of its superclass, etc. The new class, its superclass, its superclass’s superclass (if any) and so on are all ancestors of the new class.

Class Definitions

The definition of a class with a single superclass begins with a `class/1` directive of the form

```prolog
:- class ClassName = [SlotDef, ...] + SuperClass.
```

where the list of `SlotDef` descriptions may be empty. In that case, the definition can simplified to

```prolog
:- class ClassName = SuperClass.
```

The class `SuperClass` must be a defined class when this definition is given.

In SICStus Objects, a subclass inherits all the slots of its superclass. And, by default, it inherits all the methods of its superclass. The remainder of this section describes what the programmer can do to control this inheritance.

Slots

A class’s slots are a combination of those explicitly defined in its slot description list and the slots it inherits from its superclass. In SICStus Objects, a class inherits all the slots of its superclass. It follows that a class inherits all the slots of all its ancestors.

The programmer’s control over inheritance of slots is limited. It is not possible to rename an inherited slot, nor is it possible to change its type, unless it is a class slot. It is possible to change a slot’s initial value. And, it is possible to effectively change a slot’s visibility.

To change the initial value or the type (when allowed) of a slot, include a new `SlotDef` in the list of slot descriptions for the class, with the same slot name and a new type or initial value. The type of a class slot can only be changed to a subclass of the type of the superclass’s slot. The new initial value must still be a constant of the appropriate type.
The named_point class, defined earlier, could have better been defined from the point class, which began as follows:

```prolog
:- class point =
    [public x:float=0,
     public y:float=0].
```

The definition of the named_point class would then begin with

```prolog
:- class named_point =
    [public name:atom,
     public x:float=1.0] + point.
```

This named_point class has public slots named name, x and y, with the same types and initial values as the earlier named_point definition, which did not use inheritance. This named_point class also inherits all the methods of the point class, which saves us from having to write them again (and maintain them).

A slot that was private or protected in a superclass may be defined as public. This will cause get and put methods to be generated in the subclass. A slot that was public in a superclass may be defined as protected or private, but this does not prevent it from inheriting the get and put methods of the superclass. For that, the uninherit/1 directive, defined below, is needed.

**Methods**

In SICStus Objects, by default, a class inherits all the methods of its superclass. The programmer has more control over the inheritance of methods than the inheritance of slots, however. In particular, methods can be uninherited and they can be redefined.

To prevent a method from being inherited, use the uninherit/1 directive. For example, suppose that the class point is defined as before. That is, its definition begins as follows:

```prolog
:- class point =
    [public x:float=0,
     public y:float=0].
```

Because both slots are public, a put method is automatically generated for each, which allows their values to be changed.

The definition of a new class fixed_point might begin as follows:
:- class fixed_point = point.

:- uninherit
    point << (x/l),
    point << (y/l).

Self <- create(X, Y) :-
    store_slot(x, X),
    store_slot(y, Y).

The parentheses are necessary because of the precedences of the ‘<<’ and ‘/’ operators.

Because the put methods from point are not inherited, no instance of the fixed_point class can change its x and y values once created—unless the class definition contains another method for doing so. The get methods are inherited from point, however.

To redefine a method, simply include method clauses for its message within a class’s definition. The new method clauses replace, or shadow, the inherited method clauses for this class.

Another way to prevent the x and y slots of the fixed_point class from being modified would be to shadow the put methods. For example, they might be redefined as

    Self << x(_) :-
        format(user_error, "cannot modify x slot value.\n.", []),
        fail.

    Self << y(_) :-
        format(user_error, "cannot modify y slot value.\n", []),
        fail.

Now attempts to modify the x or y values of a fixed point object generate a specific error message and fail. A more complicated version would raise an appropriate exception.

Send Super

Even when a superclass’s method is shadowed or uninherited, it is possible to use the superclass’s method inside a method clause for the new class. This makes it possible to define a “wrapper” for the superclass’s method, which invokes the superclass’s method without having to duplicate its code. This technique works with all message types.

Sending a message to a superclass is done with a command of the form

    super MessageOp Message

where MessageOp is one of the message operators (‘<<’, ‘>>’ or ‘<-’) and Message is a message defined for the superclass. A generalization of this syntax may be used to specify which superclass to send the message to. This is discussed in Section 10.29.3.2 [obj-inh-mih], page 659.
Chapter 10: The Prolog Library

Sending a message to a class’s superclass can only be done within a message clause.

10.29.3.2 Multiple Inheritance

It is possible for a class to be defined with more than one superclass. Because the class inherits properties from multiple superclasses, this is referred to as multiple inheritance.

Multiple inheritance is a complex and controversial topic. What should be done about conflicting slot or method definitions? (This is sometimes called a “name clash.”) What should be done about slots that are inherited from two or more superclasses, but that originate with a common ancestor class? (This is sometimes called “repeated inheritance”.) Different systems take different approaches.

SICStus Objects supports multiple inheritance in a limited but still useful way. It does not allow repeated inheritance, and it places all the responsibility for resolving name clashes on the programmer. This section describes the multiple inheritance features of SICStus Objects.

Class Definitions
The definition of a class with multiple superclasses begins with a `class/1` directive of the form

```
:- class ClassName = [SlotDef, ...] + SuperClass + ....
```

The list of slot descriptions and the superclasses to the right of the ‘=’ can appear in any order, without changing the class being defined. In fact, the slot descriptions can be partitioned into more than one list, without changing the class. However, it is best to adopt a fairly simple style of writing class definition and use it consistently.

Just as the slot names in a list of slot descriptions must be distinct, superclass names should not be repeated.

Slots
In SICStus Objects, the programmer has no control over multiple inheritance of slots. All slots from all superclasses are inherited. And, the superclasses should have no slot names in common.

As a consequence, in SICStus Objects no superclasses of a class should have a common ancestor. The only exception would be the unusual case where that common ancestor has no slots.

Methods
By default, all methods are inherited from all superclasses. Any of the superclasses’ methods can be uninherited, as described earlier, by using the `uninherit/1` directive.

If the same message is defined for more than one superclass, however, then you must choose at most one method to inherit for the message. You may choose none. You may do this by defining a new method for the message (shadowing the superclasses’ methods), or by using the `uninherit/1` directive, or by using the `inherit/1` directive.
The following is considered a classic example of multiple inheritance.

```prolog
:- class toy. % no slots in this class

Self >> size(small).
Self >> rolls(false).
:- end_class toy.

:- class truck. % no slots in this class

Self >> size(large).
Self >> rolls(true).
:- end_class truck.
```

The idea expressed in these definitions is that most toys are small and do not roll. On the other hand, most trucks are large, but they do roll. A toy truck shares one feature with each class, but we can hardly expect a compiler to choose the correct one.

The definition of a new class, toy_truck, might begin with

```prolog
:- class toy_truck = toy + truck.
```

Rather than redefine the get methods for size and rolls, we can specify which to inherit in two ways. One way is positive, stating which to inherit, and the other way is negative, stating which not to inherit.

The positive version would be

```prolog
:- inherit
    toy >> (size/1),
    truck >> (rolls/1).
```

This is more convenient when a message is defined in several superclasses, because all but the chosen method are uninherited. And, it is probably easier to understand.

The negative version would be

```prolog
:- uninherit
    toy >> (rolls/1),
    truck >> (size/1).
```

The toy_truck class would exhibit the same behavior with either definition.
It is possible to define methods that access the shadowed or uninherited methods of the superclasses, by sending the message to the superclasses. In the case of multiple inheritance, however, it may be necessary to specify which superclass to send the message to.

The `toy_truck` class, for example, might define these methods:

```
Self >> uninherited_size(S) :-
    super(truck) >> size(S).

Self >> uninherited_rolls(R) :-
    super(toy) >> rolls(R).
```

They provide access to the unchosen methods from `toy_truck`’s superclasses.

While these examples with the `toy_truck` class are clearly “toy” examples, the same techniques can be used in more realistic cases.

**Abstract and Mixin Classes**

While SICStus Objects only supports a limited form of multiple inheritance, its facilities are sufficient for working with so-called *mixin classes*.

The idea is to construct similar classes by first defining a class that contains the things the desired classes have in common. Typically, this will be an *abstract class*, which will have no instances itself. Then, provide the features that differentiate the desired classes with a set of mixin classes.

Mixin classes that have nothing in common can safely be mixed together, to build the desired classes. The mixin classes will usually be abstract classes, also, because they are too specialized for their instances to be useful on their own.

The `date_stam` class defined earlier would make a good mixin class. A similar `time_stamp` class might be (partially) defined as follows:

```
:- class time_stamp =
    [hour:integer,
     minute:integer,
     second:integer].

Self <- create :-
    time(time(Hour, Minute, Second)),
    store_slot(hour, Hour),
    store_slot(minute, Minute),
    store_slot(second, Second).
```

Another mixin class might be used to “register” objects in the Prolog database.
:- class registry = [name:atom].

Self <- create(Name) :-
    Self << name(Name),
    assert(registered(Name, Self)).

Self <- destroy :-
    Self >> name(Name),
    retract(registered(Name, Self)).

The registry mixin class could have been used with the point class to define the named_point class, which was an example from an earlier section.

The ability to send a message to an object’s superclass is useful when working with mixin classes. Suppose the definition of a new class begins with

    :- NewClass = OldClass + date + time + registry.

where OldClass is some previously defined class that lacks the features provided by the date, time and registry classes. (In fact, they should not have any slot names in common.) Then its create method can be defined by

    Self <- create(Name) :-
    super(OldClass) <- create,
    super(date) <- create,
    super(time) <- create,
    super(registry) <- create(Name).

This avoids the need to duplicate the code in the create methods of OldClass and all three mixin classes.

10.29.3.3 Asking About Classes and Objects

It is possible to determine, at run time, what classes are defined, how they are related by inheritance, what class an object belongs to, etc. This section describes the predicates used for those purposes. Most of the predicates involve the class hierarchy, so they are properly described in the section on inheritance. But, several can be useful even in programs that use only simple classes.

Most of these predicates come in pairs, where one predicate involves one class or its direct superclasses, and the other predicate involves all ancestors. For example, the class superclass/2 and class_ancestor/2 predicates connect a currently defined class to its superclass(es) and to all its ancestors, respectively.

In all of these predicates, the ancestors of a class include not only superclasses and their ancestors, but also the class itself. A class cannot be a superclass of itself, by the rules of defining classes. However, it is convenient to consider every class an ancestor of itself, because then we may say that every property of a class is defined in one of its ancestors, without having to say “the class itself or a superclass or a superclass of a superclass, etc.”
Objects

The `class_of/2` predicate is used to test whether an object is of a particular type or to determine the type of an object. Similarly, the `descendant_of/2` predicate relates an object to all ancestors of its class. (Remember that the object’s class is, itself, an ancestor class of the object.)

Both require the first argument (the object) to be instantiated. That is, the predicates cannot be used to find objects of a given class. If you need to search among all the objects of a class, then you must provide a way to do it. One way to do this is to assert a fact connecting the class name to every object, when it is created. The named_point example of the previous section took that idea a step further by allowing each object to have a different name.

The `pointer_object/2` predicate relates an object’s address (a pointer) to the object. Remember that an instance of `Class` is represented by a term of the form

```
Class(Address)
```

The `pointer_object/2` predicate requires that one of its arguments be instantiated, but it may be either one. Hence, just by knowing the address of an object (which possibly was returned by a foreign function) it is possible to determine the object’s type.

Most Prolog programmers can safely ignore the `pointer_object/2` predicate, unless they are using SICStus Objects with foreign functions or with the Structs package.

Classes

The `current_class/1` predicate is used to ask whether a class is currently defined or to get the names of all currently defined classes.

The `class_superclass/2` predicate is used to test whether one class is a superclass of another, or to find a class’s superclasses, or to find a class’s subclasses, or to find all subclass-superclass pairs. The `class_ancestor/2` predicate is used in the same ways for the ancestor relation between currently defined classes.

As an example, the following goal finds all the ancestors of each currently defined class.

```
| ?- setof(C-As, 
    (current_class(C), 
     setof(A, class_ancestor(C,A), As)), 
    L).
```

It binds `L` to a list of terms of the form `Class-AncestorList`, with one term for each currently defined class.

Arguably, this predicate violates the principle of information hiding, by letting you ask about how a class is defined. Therefore, you should generally avoid it. It may be useful, however, in debugging and in building programmer support tools.
Messages

The `message/4` predicate is used to ask whether a message is defined for a class or to find what messages are defined for a class, etc. It does not distinguish between messages whose methods are defined in the class itself and those that are inherited from a superclass.

The `direct_message/4` predicate is used to ask whether a message is not only defined for a class, but whether the method for that message is defined in the class itself. It can also be used to determine which methods are defined in a class. This ability to look inside a class definition makes `direct_message/4` an egregious violator of the principle of information hiding. Thus it, like `class_ancestor/2`, should mainly be confined to use in programmer support applications.

Both `message/4` and `direct_message/4` take the message operator as an argument, along with the class, message name and arity. Hence it is possible to use these predicates to ask about get, put or send messages.

It is not possible to ask about a class's slots, nor should it be. However, it is possible (and quite reasonable) to ask about the get and put messages that are defined for a class. For example, the following goal finds all the 1-argument messages that are defined for both the get and put message operators in the class `Class`.

\[
\text{?}:- \text{setof}(Message, \\
\quad (\text{message(Class}, <<, \text{Msg}, 1), \\
\quad \text{message(Class}, >>, \text{Msg}, 1)), \\
\quad L).
\]

There may or may not be slots corresponding to these messages; that detail is hidden in the definition of `Class`. However, it should be possible to use `Class` as if the slots were there.

As an example, recall the polar coordinate interface to the point class, which defined get and put methods for `r` and `theta`, even though data was represented inside an object by rectangular coordinates `x` and `y`.

10.29.4 Term Classes

Sometimes it is convenient to be able to send messages to ordinary Prolog terms as if they were objects. Prolog terms are easier to create than objects, and unlike objects, they are automatically garbage collected (see Section 10.29.5.2 [obj-tech-lim], page 669). Of course, unlike objects, Prolog terms cannot be modified. However, when a particular class of objects never needs to be dynamically modified, and does not need to be subclassed, it may be appropriate to define it as a term class.

A term class is defined much like an ordinary class; it begins with a `:- class` directive defining the class and its slots, follows with clauses defining the methods for this class, and ends with an `:- end_class` directive, the end of the file, or another `:- class` directive. The only difference is in the form of the `:- class` directive introducing a term class definition.
10.29.4.1 Simple Term Classes

The simplest sort of term class declaration has the following form:

```prolog
:- class ClassName = term(Term).
```

This declares that any term that unifies with Term is an instance of class ClassName. For example, you might declare:

```prolog
:- class rgb_color = term(color(_Red,_Green,_Blue)).
color(R,_G,_B) >> red(R).
color(_R,G,_B) >> green(G).
color(_R,_G,B) >> blue(B).
:- end_class rgb_color.
```

This would declare any term whose principal functor is color and arity is three to be an object of class rgb_color. Given this declaration, entering the goal

```prolog
color(0.5, 0.1, 0.6) >> blue(B)
```

would bind B to 0.6.

Note that you cannot use `create/2` to create a term class instance. Since they are just ordinary terms, you can create them the same way you would create any ordinary Prolog term. Similarly, you cannot modify an existing term class instance.

You may specify a term class as the type of a slot of an ordinary class. This is effectively the same as specifying the type to be `term`. In particular, fetching and storing term class slots is not very efficient. Also, the default value for slots of term class type is `'`; this is because not enough is known about a simple term class to determine a better default. For an explanation of how to avoid these pitfalls, see Section 10.29.4.3 [obj-tcl-tce], page 666.

10.29.4.2 Restricted Term Classes

The intention of the rgb_color class presented above is to represent a color as a triple of floating point numbers between 0.0 and 1.0. But the above definition does not restrict the arguments of the color term in any way: any color/3 term is considered to be an instance of the rgb_color class.

The second form of term class declaration allows you to specify constraints on instances of a term class. The form of such a declaration is as follows:

```prolog
:- class ClassName = term(Term, Constraint).
```

This declares that any term that unifies with Term and satisfies Constraint is an instance of class ClassName. The Constraint term is an ordinary Prolog goal, which will usually share variables with Term.
To extend our \texttt{rgb\_color} class example so that only \texttt{color/3} terms whose arguments are all floats between 0.0 and 1.0 are instances of \texttt{rgb\_color}, we would instead begin the definition as follows:

\begin{verbatim}
:- class rgb\_color =
  term(color(Red,Green,Blue),
    (float(Red), Red >= 0.0, Red =< 1.0,
    float(Green), Green >= 0.0, Green =< 1.0,
    float(Blue), Blue >= 0.0, Blue =< 1.0)).
\end{verbatim}

Note the parentheses around the constraint in this example. Whenever the constraint contains multiple goals separated by commas, you will need to surround the goal with parentheses.

With this definition of the \texttt{rgb\_color} class, only \texttt{color/3} terms whose arguments are all floating point numbers between 0 and 1 inclusive will be considered to be instances of \texttt{rgb\_color}.

\subsection{10.29.4.3 Specifying a Term Class Essence}

As mentioned above, it is possible to specify a term class as the type of a slot of some other object. For example, we might declare

\begin{verbatim}
:- class colored\_rectangle = [
  public origin:point,
  public size:size,
  public color:rgb\_color].
\end{verbatim}

This will store an \texttt{rgb\_color} object (i.e., a \texttt{color/3} term) in the \texttt{color} slot of each \texttt{colored\_rectangle} object. Unfortunately, though, SICStus Objects cannot tell what is the best way to store a term object, and therefore it stores it the same way it stores a slot declared to be of \texttt{term} type: using the Prolog database. This has all the efficiency disadvantages of \texttt{term} slots. In this case, however, we know that all that really needs to be saved in order to save an \texttt{rgb\_color} object is the three arguments. We also know that each of these arguments is a floating point number, and because precision is not terribly critical in representing colors, each of these numbers can be stored as a \texttt{float}, rather than a \texttt{double}. In effect, we know that the \textit{essence} of a \texttt{rgb\_color} object is these three numbers; if we have them, then we can easily construct the \texttt{color/3} term. If we provide this information in the declaration of the \texttt{rgb\_color} class, then SICStus Objects can store instances of the \texttt{rgb\_color} class as 3 separate floats, rather than as a term, significantly improving the performance of creating or destroying a \texttt{colored\_rectangle} object, as well as accessing or modifying its \texttt{color} slot.

The essence of a term class is specified with the following form of \texttt{class} declaration:

\begin{verbatim}
:- class ClassName = term(Term, Constraint, Essence).
\end{verbatim}

where \textit{Essence} is of the form

\[ [\textit{Name1:Type1=i[\textit{Variable1}], \textit{Name2:Type2=i[\textit{Variable2}]}, \ldots} \]
and each Name is a distinct atom naming a slot, each Type is a slot type as specified in Section 10.29.2.2 [obj-scl-slt], page 644, and each Variable is an unbound variable appearing in Term. Providing a term essence not only makes storage of terms in ordinary object slots more efficient, it also gives a name to each “essential” slot of the term class. This allows you to use fetch_slot to fetch the slots of this class.

To extend our rgb_color example, we might introduce the rgb_color class with this declaration:

```prolog
:- class rgb_color =
    term(color(Red,Green,Blue),
        (float(Red), Red >= 0.0, Red =< 1.0,
        float(Green), Green >= 0.0, Green =< 1.0,
        float(Blue), Blue >= 0.0, Blue =< 1.0),
        [red:float=Red, green:float=Green, blue:float=Blue]).
```

This declaration defines the rgb_color class exactly as the example declaration of the previous section: every color/3 term whose arguments are all floating point numbers between 0.0 and 1.0 inclusive are instances of rgb_color. The difference is that with this declaration, ordinary classes that have slots of type rgb_color, such as the colored_rectangle example above, will be stored more efficiently, and their rgb_color slots will be accessed and modified much more efficiently. Also, it will be possible to use fetch_slot(red, Red) in the methods of the rgb_color class to fetch to red component of the message recipient, and similarly for green and blue.

10.29.5 Technical Details

This section will be expanded in future versions of SICStus Objects. For now, it provides a BNF grammar for the syntax of class definitions and a short list of some limitations of SICStus Objects.

10.29.5.1 Syntax of Class Definitions

The following BNF grammar gives a concise description of the syntax of class definitions. It assumes an understanding of Prolog syntax for the following items: variable, atom, compound_term, and constant. Slot types, particularly the address, class and pointer types, were discussed in an earlier section.

```
class_def ::= class_begin { clause | method } class_end

class_begin ::= :- class class_name opt_class_spec .

opt_class_spec ::= empty | = class_spec

class_spec ::= multi_parent_or_slots | term_class_spec

clause ::= head opt_body .
```
head ::= atom | compound_term .

method ::= message_head opt_body .

message_head ::= message_goal

class_end ::= :- end_class opt_class_name .
| empty /* if followed by class_begin or eof */

message ::= atom | compound_term

multi_parent_or_slots ::= parent_or_slots { + parent_or_slots }

parent_or_slots ::= class_name | [] | [ slot_def { , slot_def } ]

slot_def ::= opt_visibility slot_name : slot_type opt_init_value

opt_visibility ::= empty | private | protected | public

opt_init_value ::= empty | = constant

term_class_spec ::= term(term opt_goal_essence)

opt_goal_essence ::= empty | , goal opt_essence

opt_essence ::= empty | , essence

essence ::= [ variable : slot_type { , variable : slot_type } ]

opt_body ::= empty | :- body

body ::= message_or_goal { , message_or_goal }

message_or_goal ::= message_goal | goal

message_goal ::= variable message_operator message

message_operator ::= << | >> | <-

opt_class_name ::= empty | class_name

class_name ::= atom

slot_name ::= atom

slot_type ::= integer
10.29.5.2 Limitations

This section summarizes the current limitations of SICStus Objects.

**Debugging**

When you debug SICStus Objects programs that were compiled using the `obj_decl` module, you are tracing the translated version of your code. This includes all method clauses and (some) message sending commands.

The source-linked debugger cannot connect compiled SICStus Objects code with the source code.

**Garbage Collection**

There is no garbage collection of objects. It is the responsibility of the programmer to keep track of unused objects. In particular, avoid doing the following:

\[
\text{?- create(Class, Object).}
\]

Unless the create message for `Class` made some provision for finding the new object again, it is now lost. It cannot be used, and it cannot be destroyed.

**Multiple Inheritance**

The provisions for multiple inheritance in this version of SICStus Objects are limited. In particular, there is no control over the inheritance of slots, which makes repeated inheritance impossible. However, it does support the mixin style of multiple inheritance.

**Persistence**

While objects are more persistent than Prolog variables, there is no automatic way to save objects from one execution of your program to the next. Hence they are less persistent than the clauses in the Prolog database.

If you need to save a set of objects from one Prolog session to another, then copy the objects to the Prolog database as terms, and save them to a `.sav` file. Then, after you reload the `.sav` file, rebuild the objects. Keep in mind that addresses are not valid from one session to another.
In short, there is no way to avoid initializing objects at run time.

10.29.6 Exported Predicates
The following reference pages, alphabetically arranged, describe the exported SICStus Objects predicates. They can be imported by an embedded command:

    :- use_module(library(objects)).
10.29.6.1  <-/2

Synopsis
+Obj <- +Mesg

Arguments

Obj object
Mesg term

Description
Sends Mesg to Obj. A send message. The class of Obj must have a method defined for this message.

A clause with <-/2 as the principal functor of its head is a method definition clause. Such clauses only occur within the scope of a class definition. They are expanded at compile time.

Exceptions

instantiation_error
either argument is unbound.

domain_error
Mesg is not callable or Obj is not a valid object.

existence_error
Mesg is not a defined message for Obj.

Caveat
For reasons of efficiency, an Existence Error will only be raised if the code that sends the message is compiled with debugging enabled (see debug_message), or if the message is not determined at compile-time. In other circumstances, the message will simply fail.

Calls to the <-/2 predicate will be compiled into more efficient code if the obj_decl module is loaded at compile time.

See Also
<</2, >>/2, direct_message/4, message/4
10.29.6.2  \texttt{\textless \textless 2}

Synopsis

\[ +\text{Obj} \text{\textless \textless +\text{Att} } \]

Arguments

\begin{itemize}
  \item \textit{Obj} \hspace{1cm} \textit{object}
  \item \textit{Att} \hspace{1cm} \textit{term}
\end{itemize}

Description

Sends a message to \texttt{Obj} to store the value of \texttt{Att} in the object. A put message. \texttt{Att} must be an attribute that can be stored in objects of \texttt{Obj}'s class.

A clause with \texttt{\textless \textless 2} as the principal functor of its head is a method definition clause. Such clauses only occur within the scope of a class definition. They are expanded at compile time.

Put methods are automatically generated for public slots.

Exceptions

\begin{itemize}
  \item \texttt{instantiation\_error}
    \hspace{1cm} either argument is unbound.
  \item \texttt{domain\_error}
    \hspace{1cm} \texttt{Mesg} is not \texttt{callable} or \texttt{Obj} is not a valid object.
  \item \texttt{existence\_error}
    \hspace{1cm} \texttt{Mesg} is not a defined message for \texttt{Obj}.
\end{itemize}

Caveat

For reasons of efficiency, an Existence Error exception will only be raised if the code that sends the message is compiled with debugging enabled (see \texttt{debug\_message}), or if the message is not determined at compile-time. In other circumstances, the message will simply fail.

Calls to the \texttt{\textless \textless \textless 2} predicate will be compiled into more efficient code if the \texttt{obj\_decl} module is loaded at compile time.

See Also

\texttt{\textless \textless 2}, \texttt{\textgreater \textgreater \textless \textless 2}, \texttt{direct\_message/4}, \texttt{message/4}, \texttt{store\_slot/2}
10.29.6.3  \texttt{\textasciitilde\textasciitilde}/2

Synopsis
\texttt{+Obj \textasciitilde\textasciitilde +Att}

Arguments
 Obj \hspace{1em} \textit{object}
 Att \hspace{1em} \textit{term}

Description
Sends a message to \textit{Obj} that fetches the value of \textit{Att} from the object. A get message. \textit{Att} must be an attribute to fetch from \textit{Obj}’s class.

A clause with \texttt{\textasciitilde\textasciitilde}/2 as the principal functor of its head is a method definition clause. Such clauses only occur within the scope of a class definition. They are expanded at compile time.

Get methods are automatically generated for public slots.

Exceptions
\begin{itemize}
  \item \texttt{instantiation\_error}
    \begin{itemize}
      \item either argument is unbound.
    \end{itemize}
  \item \texttt{domain\_error}
    \begin{itemize}
      \item \texttt{Mesg} is not \textit{callable} or \textit{Obj} is not a valid object.
    \end{itemize}
  \item \texttt{existence\_error}
    \begin{itemize}
      \item \texttt{Mesg} is not a defined message for \textit{Obj}.
    \end{itemize}
\end{itemize}

Caveat
For reasons of efficiency, an Existence Error exception will only be raised if the code that sends the message is compiled with debugging enabled (see \texttt{debug\_message}), or if the message is not determined at compile-time. In other circumstances, the message will simply fail.

Calls to the \texttt{\textasciitilde\textasciitilde}/2 predicate will be compiled into more efficient code if the \texttt{obj\_decl} module is loaded at compile time.

See Also
\texttt{\textasciitilde}/2, \texttt{\textasciitilde\textasciitilde}/2, \texttt{direct\_message/4}, \texttt{message/4}, \texttt{fetch\_slot/2}
10.29.6.4 class/1  

**Declaration**

**Synopsis**

```prolog
:- class ClassName.

:- class ClassName = [SlotDef, ...].

:- class ClassName = Super.

:- class ClassName = [SlotDef, ...] + Super + ....

:- class ClassName = term(Term).

:- class ClassName = term(Term, Goal).

:- class ClassName = term(Term, Goal, Essence).
```

**Arguments**

- `ClassName`  
  `atom`

- `SlotDef`  
  `term`

- `Super`  
  `atom`

**Description**

The definition of class `ClassName` begins with this `class/1` directive and ends with the next `class/1` directive, the next `end_class/[0,1]` directive, or the end of the file, whichever comes first. All clauses that look like method definitions within the scope of the class definition (that is, which have one of `<-/2`, `</2` or `>>&2` as the principal functors of their heads) are considered method definitions of the class.

You may provide as many slot definitions (`SlotDef`) and superclasses (`Super`) as you like. All superclasses must be previously defined classes.

A slot definition (`SlotDef`) has the form

```
Visibility SlotName:Type = InitialValue
```

where `Visibility` and `=' InitialValue'` are optional.

`Visibility` is either `public`, `protected`, or `private`. If it is omitted, then the slot is private.

`SlotName` must be an atom.

`SlotType` must be one of the following:

- `integer`  
  signed integer, large enough to hold a pointer

- `integer_64`  
  64-bit signed integer

**since release 4.3**
integer_32
  32-bit signed integer
integer_16
  16-bit signed integer
integer_8
  8-bit signed integer
unsigned
  unsigned integer, large enough to hold a pointer
unsigned_64
  64-bit unsigned integer
  \textit{since release 4.3}
unsigned_32
  32-bit unsigned integer
unsigned_16
  16-bit unsigned integer
unsigned_8
  8-bit unsigned integer
float
  64-bit floating point number
float_32
  32-bit floating point number
atom
  Prolog atom
address
  pointer
term
  Prolog term
Class
  pointer to an instance of \textit{Class}, which must be a previously defined class
pointer\text{(}Type\text{)}
  like \textit{address}, except that access to this slot yields, and update of this slot expects, a unary term whose functor is \textit{Type}

\textit{InitialValue} may be any constant appropriate for the slot’s type.

\textit{Term}, if specified, is any compound Prolog term. Class declarations of any of the last three forms introduce a \textit{term class}, which defines any term that unifies with \textit{Term} as an instance of the class being defined.

\textit{Goal}, if specified, is any Prolog goal. This goal may be used to restrict which terms that unify with \textit{Term} will be considered to be instance of the class being defined. The default \textit{Goal} is \textit{true}. Other than when it is \textit{true}, \textit{Goal} will usually share variables with \textit{Term}.

\textit{Essence}, if specified, is a list of terms of the form

\texttt{Variable:Type}

where \texttt{Variable} is a variable appearing somewhere in \textit{Term} and \textit{Type} is one of the possible \textit{Slottype} types listed above. There should be a \texttt{Variable:Type} pair for every variable in \textit{Term}. By specifying an essence, you permit much more space- and time-efficient storage of and access to term slots.
Caveat
Note that every class for which you want to be able to create instances must define at least one create method.

Examples
The following class definition is for a class named point, with two public slots, named x and y. Both slots are of type integer and have initial values of 1 and 2, respectively.

```prolog
:- class point =
    [public x:integer=1,
     public y:integer=2].

Self <- create. :- end_class point.
```

Because the slots are public, they have get and put methods generated automatically. Because the class has a create method defined, it is possible to create an instance with the command

```prolog
| ?- create(point, PointObj).
```

which creates a point object and binds the variable PointObj to it.

Using the point class, we could create a class, named_point, which has an extra public slot, name.

```prolog
:- class named_point =
    [public name:atom] + point.

Self <- create(Name, X, Y) :-
    Self << name(Name),
    Self << x(X),
    Self << y(Y).

:- end_class named_point.
```

The only way to create a named_point object requires specifying values for all three slots.

See Also
end_class/[0,1]

Section 10.29.2 [obj-scl], page 643, Section 10.29.4 [obj-tcl], page 664.
10.29.6.5 class_ancestor/2

Synopsis
class_ancestor(?Class, ?Anc)

Arguments
Class  atom
Anc    atom

Description
Anc is Class or an ancestor class of Class.

See Also
class_superclass/2
10.29.6.6 class_method/1

**Synopsis**

):- class_method +Name/+Arity, ... .

**Arguments**

*Name*    atom
*Arity*   integer

**Description**

Declares that a class’s method for send message *Name/Arity* is an ordinary method, not an instance method.

Used when the class being defined inherits an instance method from a superclass, to allow the class to define a non-instance method for the message. A descendant class may still declare this to be an instance method, so the same message may be an instance method for some classes and an ordinary class method for others.

Must occur within the scope of the class definition. Only applies to send messages.

**See Also**

*instance_method/1*
10.29.6.7 class_superclass/2

Synopsis

class_superclass(?Class, ?Super)

Arguments

Class  atom
Super  atom

Description

Class is an immediate subclass of Super.

See Also

class_ancestor/2
10.29.6.8 class_of/2

Synopsis
class_of(+Obj, -Class)

Arguments
Obj object
Class atom

Description
Class is the class of Obj.

Exceptions
instantiation_error
Obj is unbound.
type_error
Obj is not a valid object.

See Also
pointer_object/2
10.29.6.9 create/2

Synopsis
create(+Descriptor,-Obj)

Arguments
Descriptor
term
Obj
object

Description
Obj is a newly created and initialized object. Descriptor is a term describing the object
to create. After memory is allocated and any slot initializations are performed, a create
message is sent to the object.

The functor of Descriptor indicates the class to create. The arguments of the create message
are the arguments of Descriptor.

Exceptions
instantiation_error
    Descriptor is unbound.
domain_error
    Descriptor is not a valid create descriptor.
resource_error
    unable to allocate enough memory for object.

Caveat
You must have a create/\(N\) method for every arity \(N\) you want to be able to use in creating
instances of a class. This includes arity 0. If no such method exists, then a domain error
will be raised.

Examples
Given the class definition

```prolog
:- class point =
    [public x:integer=1,
     public y:integer=2].

Self <- create.
Self <- create(X, Y) :-
    Self << x(X),
    Self << y(Y).
:- end_class point.
```

the command

```
| ?- create(point, Point1).
```
creates a `point` object, with the default slot values for x and y, and binds variable `Point1` to the new object. The command

```prolog
| ?- create(point(10,15), Point2).
```

creates a `point` object with values 10 and 15 for slots x and y, respectively, and binds variable `Point2` to the new object.

**See Also**

`destroy/1`
10.29.6.10  current_class/1

Synopsis

current_class(*Class)

Arguments

Class  atom

Description

Class is the name of a currently defined class.
10.29.6.11 debug_message/0

Synopsis
:- debug_message.

Description
Prolog clauses following this directive will be compiled to send messages “carefully.”

That is, a message sent to an object that does not understand the message will raise an
exception, which describes both the message and the object receiving it. This also catches
attempts to send an unbound message, to send a message to an unbound object, and similar
errors.

See Also
nodebug_message/0
10.29.6.12 define_method/3

Synopsis
define_method(+Obj, +Message, +Body)

Arguments
Obj object
Message term
Body callable

Description
Installs Body as the method for Message in the instance Obj. Following the execution of this goal, sending Message to Obj will execute Body, rather than the default method or a method previously defined with define_method/3.

Message must have been declared to be an instance method for the class of Obj.

Exceptions
instantiation_error
 any argument is unbound.
type_error
 Obj is not a compound term, or Message or Body is not callable.
domain_error
 Message does not specify an instance method for the class of Obj, or Body include a goal to fetch or store a non-existent slot.

See Also
instance_method/1, undefine_method/3
10.29.6.13 descendant_of/2

Synopsis
descendant_of(+Obj, ?Class)

Arguments

Obj object
Class atom

Description

Obj is an instance of Class or of a descendant of Class.

Exceptions

instantiation_error

Obj is unbound.

type_error

Object is not a valid object.

See Also

class_ancestor/2, class_of/2, class superclass/2
10.29.6.14 destroy/1

Synopsis

destroy(+Obj)

Arguments

Obj object

Description

First, sends a destroy message to Obj, if such a message is defined for its class. A destroy message takes no argument. Unlike create/2, it is possible to destroy instances of a class even if it defines no destroy methods. Finally, disposes of Obj.

Exceptions

instantiation_error

Obj is unbound.

type_error

Object is not a valid object.

See Also

create/2
10.29.6.15  direct_message/4

Synopsis

direct_message(?Class, ?Op, ?Name, ?Arity)

Arguments

Class      atom
Op         message_operator
Name       atom
Arity      integer

Description

Name/Arity is an Op message directly understood (defined rather than inherited) by instances of Class. This predicate is used to test whether a message is defined for a class.

Op is one of <-, >>, or <<, specifying the kind of message.

This predicate violates the principle of information hiding by telling whether the method for a message is defined within a class or inherited. Hence its use in ordinary programs is discouraged. It may be useful, however, during debugging or in developing programming support tools.

See Also

<-/2, <</2, >>/2, message/4
10.29.6.16 end_class/[0,1]  

Synopsis
:- end_class.
:- end_class +ClassName.

Arguments
ClassName  
atom

Description
A class definition continues until the next end_class/[0,1] directive, the next class/1 directive, or the end of the file, whichever comes first.

It is not possible to nest one class definition within another.

All clauses that look like method definitions (that is, which have one of <-/2, </2 or >/2 as the principal functors of their heads) are considered to be method definitions for the class.

Caveat
The argument to end_class/1, if specified, must match the class name of the preceding class/1 directive.

See Also
class/1
**10.29.6.17 fetch_slot/2**

**Synopsis**

`fetch_slot(+SlotName, -Value)`

**Arguments**

- `SlotName` atom
- `Value` term

**Description**

Fetches `Value` from the slot specified by `SlotName`.

This predicate may only appear in the body of a method clause, and it always operates on the object to which that message is sent. It cannot be used to directly access the slots of another object.

**Exceptions**

- `instantiation_error`
  
  `Slot` is unbound.

- `domain_error`
  
  `Slot` is not the name of a slot of the current class.

- `permission_error`
  
  `Slot` is a private slot of a superclass.

**See Also**

`>>/2, store_slot/2`
10.29.6.18 inherit/1  

**Declaration**

Synopsis

`:-(inherit +ClassName +Op +Name/+Arity, ....)

**Arguments**

*ClassName*

  atom

*Op*

  message_operator

*Name*

  atom

*Arity*

  integer

**Description**

*ClassName* names the class from which the message should be inherited, *Op* indicates which kind of message it is, and *Name* and *Arity* indicate the name and arity of the message to be inherited. You may include several inheritance specifications in one directive.

**Caveat**

Be careful of the precedences of the message operator and the `/` operator. You may need to use parentheses.

**Examples**

Suppose classes *toy* and *truck* are defined as follows:

```
:-class toy.
  Self <- create.
  Self >> size(small).
  Self >> rolls(false).
  :- end_class toy.

:-class truck.
  Self <- create.
  Self >> size(small).
  Self >> rolls(true).
  :- end_class truck.
```

Then *toy_truck* inherits its size from *toy* and the fact that it rolls from *truck*:

```
:- class toy_truck = toy + truck.
:- inherit
  toy <- (create/0),
  toy <- (size/1),
  truck <- (rolls/1).
:- end_class toy_truck.
```

Note that this is just a toy example.
See Also
uninherit/1
10.29.6.19 instance_method/1  

declaration

Synopsis

:- instance_method +Name/+Arity.

Arguments

Name  atom  
Arity  integer

Description

The message Name/Arity is declared to support instance methods in a class. This means that instances of this class, and its descendants, may each define their own methods for this message.

A method defined for this message by the class is considered the default method for the message. An instance that does not define its own method uses the default. Defining a new method overrides this default method; there is no need to explicitly remove it.

An instance method is installed in an instance of the class with the define_method/3 predicate. An instance method is removed from an instance of the class, reverting to the default method, with the undefine_method/3 predicate.

Must occur within the scope of the class definition. Only applies to send messages.

See Also

class_method/1, define_method/3, undefine_method/3
10.29.6.20 message/4

Synopsis
message(?Class, ?Op, ?Name, ?Arity)

Arguments
Class atom
Op message_operator
Name atom
Arity integer

Description
Name/Arity is an Op message understood by instances of Class. This predicate is used to
test whether a message is either defined for or inherited by a class.

Op is one of <-, >>, or <<, specifying the kind of message.

See Also
<-/2, <</2, >>/2, direct_message/4
10.29.6.21  nodebug_message/0

Synopsis
:- nodebug_message.

Description
Prolog clauses following this directive are no longer compiled to send messages "carefully."

See Also
debug_message/0
10.29.6.22  pointer_object/2

Synopsis
pointer_object(+Addr,-Obj)
pointer_object(-Addr,+Obj)

Arguments
Addr    integer
Obj     object

Description
Addr is the address of object Obj. This can be used to get the address of an object or to get an object given its address.

Please note: This is a low level operation, passing an invalid address may crash the system.

Exceptions
instantiation_error
  both Obj and Addr are unbound.

type_error
  Addr is not an integer.
10.29.6.23 store_slot/2

**Synopsis**

store_slot(+SlotName, +NewValue)

**Arguments**

SlotName  atom
NewValue  term

**Description**

Stores NewValue in the slot specified by SlotName.

This predicate may only appear in the body of a method clause, and it always operates on the object to which that message is sent. It cannot be used to directly modify the slots of another object.

**Exceptions**

instantiation_error
   either argument is unbound.

type_error
   NewValue is not of the appropriate type for Slotname.

domain_error
   Slotname is not the name of a slot of the current class.

permission_error
   Slotname is a private slot of a superclass.

**See Also**

<</2, fetch_slot/2
10.29.6.24 undefine_method/3

Synopsis

undefine_method(+Obj, +Name, +Arity)

Arguments

Obj     object
Name    atom
Arity   integer

Description

Removes Obj’s current instance method for the Name/Arity message. After executing this goal, sending this message to Obj executes the class’s default method for the message.

Name/Arity must have been declared to be an instance method for the class of Obj.

If Obj has no current instance method for the Name/Arity message, then the predicate has no effect.

Exceptions

instantiation_error
    any argument is unbound.

type_error
    Obj is not a compound term, Name is not an atom, or Arity is not an integer.

domain_error
    Message does not specify an instance method for the class of Obj.

See Also

define_method/3, instance_method/1
10.29.6.25 uninherit/1

Synopsis

:- uninherit +Class +Op +Name/+Arity, ... .

Arguments

| Class     | atom    |
| Op        | message_operator |
| Name      | atom    |
| Arity     | integer |

Description

This prevents the class within whose scope this directive appears from inheriting the Name/Arity method of type Op from ancestor Class.

If Class is unbound, then the specified message is uninherited from all ancestors that define it.

Caveat

Note that if you define a message for your class, then you do not need to uninherit that message from its superclasses: it will automatically be shadowed.

Be careful of the precedences of the message operator and the / operator. You may need to use parentheses.

Examples

    :- uninherit someclass << (foo/1),
       someclass >> (foo/1).

This prevents the get and put methods for the slot foo from being inherited from any ancestors of class someclass. In effect, it makes the foo slot a protected slot for this class.

See Also

inherit/1

10.29.7 Glossary

abstract class

A class that cannot have instances. Abstract classes are helpful in designing a class hierarchy, to contain the common parts of several concrete classes.

ancestor

One of a class’s superclasses, one of its superclasses’s superclasses, etc. Sometimes, for convenience, ancestor includes the class itself, along with its proper ancestors.

child

A synonym for subclass.
**class**

A class is defined by a description of the information its instances contain and the messages they respond to. Every object is an instance of one and only one class.

**concrete class**

A class that can have instances. Most classes are concrete.

**create method**

Specifies what actions should be taken when an instance of a class is created. A create method frequently provides initial slot values or specifies an action to be performed by the new object. A create message is sent to each new object by the `create/2` predicate. A create message is a kind of send message.

**descendant**

One of a class’s subclasses, one of its subclasses’s subclasses, etc. Sometimes the word descendant includes the class itself, along with its proper descendants.

**destroy method**

Specifies what actions should be taken when an instance of a class is destroyed. A destroy message is sent to an object by the `destroy/1` predicate. A destroy message is a kind of send message.

**direct slot access**

Fetching or storing a slot value without sending a message to the object. This should be used with care!

SICStus Objects allows direct access to a class’s slots only within its method definitions, via the `fetch_slot/2` and `store_slot/2` predicates.

**get message**

A message that inquires about some aspect of an object. Typically used to fetch slot values. Get methods are automatically generated for public slots. Get messages are written with the ‘>>’ operator.

**inheritance**

The process by which a class’s slots and methods are determined from an ancestor.

**initial value**

The value a slot is initialized to when an object is created. Every slot has a default initial value, which depends upon its type. You may specify different initial values in a class definition.

**instance**

Another word for object. The word instance draws attention to the class of which the object is an instance.

**instance method**

A method that may be defined differently for each instance of a class. The class may have a default method for this message, which is overridden by installing an instance method for a particular object.

**message**

A command to an object to perform an operation or to modify itself, or an inquiry into some aspect of the object. In SICStus Objects, a message is either
a get message, a put message or a send message. The syntax for sending a message to an object is

\[ \text{Object Operator Message} \]

where \( \text{Operator} \) is one of the following:

\[ >> \text{ get message} \]
\[ << \text{ put message} \]
\[ <- \text{ send message} \]

**method**  
A class’s implementation of a particular message. You send messages to an object, but you define methods for a class.

**method clause**  
A Prolog clause used to define a method for a class. A method clause has one of \( <-/2, <<-/2 \text{ or } >>/2 \) as the principal functor of its head, and it can only appear within the scope of its class’s definition. A method’s definition may contain more than one message clause.

**mixin class**  
A class that is intended to be combined (mixed in) with other classes, via multiple inheritance, to define new subclasses.

**multiple inheritance**  
When a class names more than one superclass. Typically, it inherits slots and methods from each. In SICStus Objects, two different superclasses should not use the same slot name. And, if a message is defined by more than one superclass, then the class definition must specify which method to inherit.

**object**  
A modifiable data item that holds information and responds to messages. Another word for instance.

**parent class**  
A synonym for superclass.

**private slot**  
A private slot is, by default, only accessible within methods of the class itself. Not even the descendants of the class may access its private slots, except through the class’s methods. Get and put methods are not automatically generated for a private slot, so it is only accessed via the methods you define. If the visibility of a slot is not specified, then it is private, rather than public or protected.

**protected slot**  
A protected slot is, by default, only accessible within methods of the class itself and its descendants. Get and put methods are not automatically generated for a protected slot, so it is only accessed via the methods you define. If the visibility of a slot is not specified, then it is private, rather than public or protected.

SICStus Objects **protected** is similar to **protected** in C++.
public slot  A public slot is accessible via its own get and put methods, which are generated for it automatically. If no visibility is specified, then a slot is private, rather than public or protected.

put message  A message that modifies some aspect of an object. Typically used to store slot values. Put methods are automatically generated for public slots. Put messages are written with the ‘<<’ operator.

send message  The most common sort of message. Used for performing an operation on an object or for performing an action that depends upon an object. Send messages are written with the ‘<-’ operator.

send super  When a method for a class executes a shadowed superclass’s method. This allows a class to put a “wrapper” around its superclass’s method, making it unnecessary to duplicate the method just to make a small extension to it.

shadow  When a class defines its own method for a message defined by one of its ancestors, the new method hides or “shadows” the ancestor’s method. The new class’s descendants will inherit its method for that message, rather than its ancestors. That is, a class always inherits the “closer” of two methods for a message.

slot  A part of an instance that holds an individual datum. Like a member of a C struct or a field of a Pascal record.

subclass  A class that is a more specific case of a particular class. This is the opposite of superclass. A class does not name its subclasses; they are inferred.

superclass  A class that is a more general case of a particular class. Each class lists its superclasses.

term class  A class whose instances are represented as ordinary Prolog terms. The functor of these objects need not be the name of the class, and the arity need not be one.

term slot  A slot that can hold any Prolog term.

uninherit  Specify that a method from a superclass should not be inherited. This is similar to shadowing the superclass’s method, but does not specify a replacement for it.

visibility  A slot may be defined to be either public, protected, or private. By default, if no visibility is specified, then a slot is private.

10.30 The ODBC Interface Library–library(odbc)

This library is an interface to an ODBC database driver. For an introduction to ODBC, see http://msdn.microsoft.com/en-us/library/ms715408(VS.85).aspx ("Introduction to ODBC"; Microsoft Web Page). ODBC 3.x is supported.
10.30.1 Overview

ODBC (Open Database Connectivity) is a standard API for using a DBMS (DataBase Management System). By using ODBC you can access data from a multitude of DBMSs without having to know the details of each DBMS.

library(odbc) is a layer on top of ODBC. It has predicates for opening the database, starting and executing a query, and retrieving the results of a query. The ODBC client application, i.e. this library, accesses all ODBC functionality via a service provided by the operating system, the ODBC Driver Manager (DM).

Some operating systems (e.g. Mac OS X and MS Windows) usually come with an ODBC Driver Manager preinstalled. For other, UNIX and UNIX-like, operating systems, unixODBC (http://www.unixodbc.org) is the most common but Mac OS X use iODBC (http://www.iodbc.org).

The ODBC Driver Manager does not, in itself, provide any database functionality. Instead the DM loads a ODBC driver specific to the particular Database Management System (DBMS) (when odbc_db_open/3,4,5 is called).

How to install and configure an ODBC driver is beyond the scope of this document. Please consult the documentation for the particular DBMS you intend to use. Some popular DBMSs are MySQL and PostgreSQL which both provide ODBC drivers for many platforms.

10.30.2 Examples

A few examples will best illustrate how to use library(odbc).

10.30.2.1 Example 1

The first example just verifies that ODBC is working and that some ODBC drivers have been configured in the ODBC Driver Manager.

    :- use_module(library(odbc)).

    example1 :-
      odbc_env_open(EnvHandle),
      odbc_list_DSN(EnvHandle, DSNs),
      odbc_env_close(EnvHandle),
      format('The known DSNs are: ~q~n', [DSNs]).

You begin by opening an environment. This is a handle which can be used for various calls to the ODBC Driver Manager (DM). You then ask the DM about the data sources, i.e. databases, it knows about. If this list is empty you need to install and configure the ODBC drivers appropriate for the database management system that you intend to use.

10.30.2.2 Example 2

This example is a simple SQL query using a fixed SQL string.
:- use_module(library(odbc)).

eexample_select :-
  odbc_env_open(EnvHandle),
  odbc_db_open('MyDatabase', EnvHandle, ConnectionHandle),
  odbc_query_open(ConnectionHandle, StatementHandle),
  odbc_query_execute_sql(StatementHandle,
    'SELECT cookie,soft FROM bakery order by soft',
    ResultSet),
  show_result(ResultSet),
  odbc_query_close(ResultSet),
  odbc_db_close(ConnectionHandle),
  odbc_env_close(EnvHandle).

show_result(ResultSet) :-
  odbc_sql_fetch(ResultSet, Row),
  show_result1(Row, ResultSet).

show_result1([], _ResultSet) :- !.
show_result1(Row, ResultSet) :-
  format('~w~n', [Row]),
  flush_output,
  odbc_sql_fetch(ResultSet, Row1),
  show_result1(Row1, ResultSet).

As always, you begin by opening an environment. You then connect to the database with odbc_db_open/3. The first argument is the identifier for the database in the DBMS. In this scenario, connecting to the database does not require a username and a password. The output from odbc_db_open/3 is an opaque handle on the database.

First, odbc_query_open/2 is used to create an SQL query, which is straightforward. Then, odbc_query_execute_sql/3 is used to execute the SQL query. By executing an SQL query a result set is created. Each consecutive call of odbc_sql_fetch/2 will retrieve one row from the result set.

10.30.2.3 Example 3

This example shows the use of parameter binding. The positional markers (?) in the SQL string are bound to the elements in the list in the third argument of odbc_query_execute_sql/5. The fourth argument is a list of datatypes corresponding to the parameters.
:- use_module(library(odbc)).

test_query2 :-
odbc_env_open('SQL_OV_ODBC3', EnvHandle),
odbc_db_open('MyDatabase', EnvHandle, ConnectionHandle),
odbc_query_open(ConnectionHandle, StatementHandle),
odbc_query_execute_sql(StatementHandle,
    'INSERT INTO scratch (vehicle, wheels) VALUES (?, ?)',
    ['railwaycar', 8],
    ['SQL_VARCHAR', 'SQL_INTEGER'],
    ResultSet),
odbc_query_close(ResultSet),
odbc_db_close(ConnectionHandle),
odbc_env_close(EnvHandle).

10.30.2.4 Example 4
This example is similar to the second, but this time we ask the database what the datatypes of the columns of the table are with odbc_list_data_types/3.

:- use_module(library(odbc)).

test_query3 :-
odbc_env_open(EnvHandle),
odbc_db_open('MyDatabase', EnvHandle, ConnectionHandle),
odbc_query_open(ConnectionHandle, StatementHandle),
odbc_list_data_types(StatementHandle,
    scratch(vehicle, wheels),
    DataTypes),
odbc_query_execute_sql(StatementHandle,
    'INSERT INTO scratch (vehicle, wheels) VALUES (?, ?)',
    ['railwaycar', 8],
    DataTypes,
    ResultSet),
odbc_query_close(ResultSet),
odbc_db_close(ConnectionHandle),
odbc_env_close(EnvHandle).

10.30.3 Datatypes

10.30.3.1 Reading from the database
When reading data from the database the following datatypes are supported, with conversion to the corresponding prolog datatypes.

SQL_CHAR, SQL_VARCHAR etc.
A list of character codes.

SQL_BIT  The integer 0 for false, or 1 for true.
SQL_INTEGER, SQL_TINYINT, SQL_SMALLINT, etc.

An integer.

SQL_REAL, SQL_DOUBLE, SQL_FLOAT

A floating point number.

SQL_DATE

A term date(Year, Month, DayOfMonth), with one-based integer arguments. E.g. date(2012,10,22) means October 22, 2012.

SQL_TIME

A term time(Hour, Minute, Second) with one-based integer arguments. E.g. time(22,11,5) means eleven minutes and five seconds past ten pm.

SQL_TIMESTAMP

A term timestamp(Year, Month, Day, Hour, Minute, Second, Fraction) where the arguments have the same meaning as for SQL_TIME and SQL_TIMESTAMP and Fraction means fractional nanoseconds past, as an integer.

the SQL null value

The atom null.

SQL_BINARY and other binary types
SQL_INTERVAL_HOUR and other interval types
SQL_UTCTIME and SQL_UTCDATETIME

Currently not supported.

Note that atoms with names that start with an upper case letter, like SQL_CHAR must be quoted in Prolog, e.g. 'SQL_CHAR'.

10.30.3.2 Writing to the database

When writing data to the database the following SQL datatypes are supported.

SQL_CHAR, SQL_VARCHAR etc.

A list of character codes, or a list of atoms.

For backwards compatibility only, an atom is also accepted, but note that the atoms null and [] have special meaning (as SQL null value and empty code list, respectively) and more atoms with special meaning may be introduced in the future. For compatibility with some ODBC drivers, the integer 0 and 1 are allowed, meaning "0" and "1".

SQL_BIT

The integer 0 for false, or 1 for true.

SQL_INTEGER, SQL_TINYINT, SQL_SMALLINT, etc.

An integer.

SQL_REAL, SQL_DOUBLE, SQL_FLOAT

A floating point number or a small integer.

SQL_DATE

A term date(Year, Month, DayOfMonth), as above.

SQL_TIME

A term time(Hour, Minute, Second), as above.

SQL_TIMESTAMP

A term timestamp(Year, Month, Day, Hour, Minute, Second, Fraction), as above.
the SQL null value
  The atom null.

SQL_BINARY and other binary types
SQL_INTERVAL_HOUR and other interval types
SQL_UTCTIME and SQL_UTCDATETIME
  Currently not supported.

if a value is out of range for the corresponding SQL type, e.g. a too large integer for SQL_SMALLINT, the result is undefined. Note that atoms with names that start with an upper case letter, like SQL_CHAR must be quoted in Prolog, e.g. 'SQL_CHAR'.

10.30.4 Exceptions
When an error in the ODBC layer occurs, predicates in library(odbc) throw error/2 exceptions. Both arguments of the error/2 exception are the same and has the following form odbc_error(Detail, Goal), where Goal is some goal where the error occurred, and Detail gives more information about the error. The Detail term can have the following form:

data_conversion
  Thrown in case of a error when converting to or from a SICStus data type from or to an ODBC data type.

unsupported_datatype
  Thrown when an SQL data type is unsupported when converting to or from a SICStus data type from or to an ODBC data type.

unknown_datatype
  Thrown when an unknown SQL data type is found when converting to or from a SICStus data type from or to an ODBC data type.

type_error
  Thrown when the Prolog data is of a type incompatible with the SQL data type when converting from a SICStus data type to an ODBC data type.

native_code
  Thrown in case of a error in the native code of library(odbc).

invalid_handle(handle_type, InvalidHandle, ReturnCode)
  Thrown when an invalid handle type is specified.

invalid_handle('HandleType'-HandleType, 'Handle'-Handle)
  Thrown when an invalid handle is specified.

invalid_handle(result_set, ResultSet)
  Thrown when a Result Set handle is invalid.

unknown_connection_option(Options)
  Thrown when an unknown option was given when calling odbc_db_open/[3,4,5].
internal_error
Thrown when an internal error occurs in library(odbc). Please report this to SICStus Support.

diag(ReturnCode, Recs)
Thrown when an error occurs in the ODBC layer, e.g. a SQL syntax error. Recs is bound to the diagnostic records reported from ODBC.

out_of_memory
Thrown when some operation runs out of memory.

there may be other Details and new Details may be added in the future.

10.30.5 Predicates

odbc_env_open(-EnvHandle)
Opens an ODBC environment. Throws an exception if the environment could not be opened.

odbc_db_open(+Dbname,+EnvHandle,-ConnectionHandle)
Opens a database with the name Dbname. The database cannot require a username and a password. ConnectionHandle is an opaque handle for accessing the database.

odbc_db_open(+Dbname,+EnvHandle,+Options,-ConnectionHandle)
Opens a database with the name Dbname. Options should be a list of zero or more of:

username(+Username)
The username for connecting to the database. The default is ''. 

password(+Password)
The password for connection to the database. The default is ''. 

login_timeout(+Timeout)
The number of seconds to wait for a login request to complete. If 0 is used, the login attempt will wait indefinitely. The default is driver-dependent.

connection_timeout(+Timeout)
The number of seconds to wait for any request on the connection to complete. If the Timeout value is 0 (the default), there is no timeout.

raw(+ConnectionOptions)
ConnectionOptions should be a list of atoms. They are passed, terminated by ;, as extra options when opening the database.

ConnectionHandle is an opaque handle for accessing the database.

odbc_db_open(+Dbname,+EnvHandle,+Options,-ConnectionHandle,-ConnectionString)
Like odbc_db_open/4 but also returns the completed connection string returned by the ODBC driver.
Chapter 10: The Prolog Library

odbc_query_open(+ConnectionHandle, -StatementHandle)

Creates a new database query. ConnectionHandle is a handle previously allocated with odbc_db_open/[3,4,5].

odbc_list_DSN(+EnvHandle, -DSNs)

EnvHandle is an opaque database handle. DSNs is unified with a list of all DSNs (Data Source Names). The list elements are X-Y where X is the DSN and Y its description.

odbc_list_data_types(+StatementHandle, +TableDesc, - DataTypes)

Makes a list of the datatypes in a table. StatementHandle is a handle previously allocated with odbc_query_open/2. TableDesc is a description of the table and its columns of the form tablename(columnname1, columnname2, ..., columnnameN), or of the form [tablename, columnname1, columnname2, ..., columnnameN] (the latter form is useful if the table has more than 255 columns). DataTypes is unified with a list of the corresponding datatypes, i.e. on the form [datatype1, datatype2, ... datatypeN].

odbc_current_table(+ConnectionHandle, ?TableName) since release 4.2

Enumerate the proper tables in the database, i.e. tables with attribute 'TABLE_TYPE'("TABLE").

ConnectionHandle is a handle previously allocated with odbc_db_open/[3,4,5]. TableName is the name, as an atom, of the table.

Note that odbc_current_table/2 may exit nondeterminately even if all arguments are instantiated when it is called.

odbc_current_table(+ConnectionHandle, ?TableName, ?Attribute) since release 4.2

Enumerate database tables and their attributes.

ConnectionHandle is a handle previously allocated with odbc_db_open/[3,4,5]. TableName is the name, as an atom, of the table. Attribute is an attribute of the table.

There are two kinds of attributes, derived attributes and raw attributes.

The derived attributes are translations of raw attributes and other information and are in a format that is directly useful. There is currently only one derived attribute,

arity(Value)

The number of columns in the table, as an integer.

This attribute is always present.

the set of derived attributes may be extended in the future.

The raw attributes correspond directly to the (non-null) values returned from the ODBC function SQLTables() and are returned as is, wrapped in a functor with the same name as the attribute, e.g. 'TABLE_CAT'("foo") would be returned for a table in the catalog "foo". Note that the names of the raw attributes are in all uppercase so you need to surround them with single quotes to
prevent their name from being parsed as a variable. Some of the raw attributes are,

'TABLE_CAT'(Value)
Catalog name, as a code list. This attribute corresponds to the TABLE_CAT column, called TABLE_QUALIFIER in ODBC 2.0, as returned from the ODBC function SQLTables().

'TABLE_TYPE'(Value)
Table type, as a code list. This attribute corresponds to the TABLE_TYPE column, as returned from the ODBC function SQLTables(). The standard table types are "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", and "SYNONYM", but there can be data-source-specific types as well.
This attribute is always present.

'REMARKS'(Value)
Table descriptive text, as a code list. This attribute corresponds to the REMARKS column, as returned from the ODBC function SQLTables().

see the ODBC documentation for SQLTables() for the full list of raw attributes and their meaning.

Note that odbc_current_table/3 may exit nondeterminately even if one of more arguments are instantiated when it is called.

odbc_table_column(+ConnectionHandle, ?TableName, ?ColumnName) since release 4.2
Enumerate database table columns.
ConnectionHandle is a handle previously allocated with odbc_db_open/[3,4,5]. TableName is the name, as an atom, of the table. ColumnName is the name, as an atom, of the table.

odbc_table_column(+ConnectionHandle, ?TableName, ?ColumnName, ?Attribute) since release 4.2
Enumerate database table columns and their attributes.
ConnectionHandle is a handle previously allocated with odbc_db_open/[3,4,5]. TableName is the name, as an atom, of the table. ColumnName is the name, as an atom, of the table. Attribute is an attribute of the table.

There are two kinds of attributes, derived attributes and raw attributes.
The derived attributes are translations of raw attributes and other information and are in a format that is directly useful. There is currently only one derived attribute,

nullable(Value)
true if the column is definitely nullable, or false if the column is definitely not nullable. The value is derived from the raw at-
tributes NULLABLE and IS_NULLABLE, see the documentation for SQLColumns() for details.
This attribute is not present if it can not be determined whether the column is nullable.

The set of derived attributes may be extended in the future.
The raw attributes correspond directly to the (non-null) values returned from the ODBC function SQLColumns() and are returned as is, wrapped in a functor with the same name as the attribute, e.g. 'TABLE_CAT'("foo") would be returned for a column in a table in the catalog "foo". Note that the names of the raw attributes are in all uppercase so you need to surround them with single quotes to prevent their name from being parsed as a variable. Some of the raw attributes are,

'REMARKS'(Value)
Column descriptive text, as a code list. This attribute corresponds to the REMARKS column, as returned from the ODBC function SQLColumns().

'ORDINAL_POSITION'(Value)
The ordinal position of the column in the table, starting at 1. This attribute corresponds to the ORDINAL_POSITION column, as returned from the ODBC function SQLColumns(). This attribute is always present.

See the ODBC documentation for SQLColumns() for the full list of raw attributes and their meaning.
Note that odbc_table_column/4 may exit nondeterminately even if one of more arguments are instantiated when it is called.

odbc_query_execute_sql(+StatementHandle, +SQLString, +ParamData, +ParamDataTypes, -ResultSet)
Executes an SQL query. StatementHandle is a handle previously allocated with odbc_query_open/2. SQLString is the SQL statement to be executed. The statement string may contain parameter markers. ParamData is a list of data to be bound to the parameter markers. ParamDataTypes is a list of data types corresponding to the ParamData list. ResultSet is bound to an opaque data structure describing the result of the query.

odbc_query_execute_sql(+StatementHandle, +SQLString, -ResultSet)
StatementHandle is a handle previously allocated with odbc_query_open/2. SQLString is the SQL statement to be executed. ResultSet is bound to an opaque data structure describing the result of the query.

odbc_sql_fetch(+ResultSet, -Row)
Fetch the next row from the result set. ResultSet is the result set from odbc_query_execute_sql/[3,5]. Row is unified with a non-empty list of data constituting a row in the result set, or with [] when there are no more rows. The elements in the Row are in the same order as in the corresponding query.
odbc_query_close(+Query)
Closes the query represented by Query, which can be either a result set, e.g. as returned from odbc_query_execute_sql/[3,5], or it can be a statement handle, as returned from odbc_query_open/2.

odbc_db_close(+ConnectionHandle)
Closes the connection to the database.

odbc_env_close(+EnvHandle)
Frees the environment handle.

10.31 Ordered Set Operations—library(ordsets)
This library module provides operations on sets represented as ordered lists with no duplicates. Thus \{c,r,a,f,t\} would be \[a,c,f,r,t\]. The ordering is defined by the \(<\) family of term comparison predicates, which is the ordering used by sort/2 and setof/3.

The benefit of the ordered representation is that the elementary set operations can be done in time proportional to the sum of the argument sizes rather than their product. You should use the operations defined here in preference to those in library(sets) unless there is a compelling reason why you can’t. Some of the unordered set routines, such as member/2, length/2 and select/3 can be used unchanged on ordered sets; feel free so to use them.

There is no ordset_to_list/2, as an ordered set is a list already. Exported predicates:

is_ordset(+List)
is true when List is a list of terms \[T1,T2,...,Tn\] and the terms are strictly increasing: \(T1 < T2 < ... < Tn\). The output of sort/2 always satisfies this test. Anything which satisfies this test can be given to the predicates in this file, regardless of where you got it.

list_to_ord_set(+List, -Set)
is true when Set is the ordered representation of the set represented by the unordered representation List. The only reason for giving it a name at all is that you may not have realized that sort/2 could be used this way.

ord_add_element(+Set1, +Element, -Set2)
Equivalent to ord_union(Set1, [Element], Set2), but a bit faster.

ord_del_element(+Set1, +Element, -Set2)
Equivalent to ord_subtract(Set1, [Element], Set2), but a bit faster.

ord_disjoint(+Set1, +Set2)
is true when the two ordered sets have no element in common.

ord_intersect(+Set1, +Set2)
is true when the two ordered sets have at least one element in common.

ord_intersection(+Set1, +Set2, -Intersection)
is true when Intersection is the ordered representation of Set1 and Set2, provided that Set1 and Set2 are ordered sets.
ord_intersection(+Set1, +Set2, ?Intersection, ?Difference)
is true when Intersection is the intersection of Set1 and Set2, and Difference is Set2 \ Set1 (like in ord_union/4), provided that Set1 and Set2 are ordered sets.

ord_intersection(+ListOfSets, -Intersection)
is true when ListOfSets is a nonempty proper list of ordered sets and Intersection is their intersection.

ord_member(+Elt, +Set)
is true when Elt is a member of Set. Suggested by Mark Johnson.

ord_nonmember(+Item, +Set)
is true when the given Item is not an element of the given Set.

ord_setsq(+Set1, +Set2)
is true when the two arguments represent the same set. Since they are assumed to be ordered representations, they must be identical.

ord_setproduct(+Set1, +Set2, -Product)
If Set1 and Set2 are ordered sets, Product will be an ordered set of x1-x2 pairs. Note that we cannot solve for Set1 and Set2, because there are infinitely many solutions when Product is empty, and may be a large number in other cases. Could be defined as:

ord_setproduct(Set1, Set2, Product) :-
( foreach(H1,Set1),
  param(Set2),
  fromto(Product,P1,P3,[])
  do ( foreach(H2,Set2),
    param(H1),
    fromto(P1,[H1-H2|P2],P2,P3)
    do true
  )
).

ord_subset(+Set1, +Set2)
is true when every element of the ordered set Set1 appears in the ordered set Set2.

ord_subtract(+Set1, +Set2, -Difference)
is true when Difference contains all and only the elements of Set1 which are not also in Set2.

ord_symdiff(+Set1, +Set2, -Difference)
is true when Difference is the symmetric difference of Set1 and Set2.

ord_disjoint_union(+Set1, +Set2, -Union)
is true when Set1 and Set2 (given to be ordered sets) have no element in common, and Union is their union. The meaning is the same as

ord_disjoint(Set1, Set2),
ord_union(Set1, Set2, Union)
but it is more efficient.
ord_union(+Set1, +Set2, -Union)
    is true when Union is the union of Set1 and Set2. Note that when something occurs in both sets, we want to retain only one copy.

ord_union(+OldSet, +NewSet, -Union, -ReallyNew)
    is true when Union is NewSet U OldSet and ReallyNew is NewSet \ OldSet. This is useful when you have an iterative problem, and you're adding some possibly new elements (NewSet) to a set (OldSet), and as well as getting the updated set (Union) you would like to know which if any of the "new" elements didn't already occur in the set (ReallyNew).

ord_union(+ListOfSets, -Union)
    is true when ListOfSets is given as a proper list of ordered sets and Union is their union. Letting K be the length of ListOfSets, and N the sum of the sizes of its elements, the cost is \( O(N \lg K) \).

ordset_order(+Xs, +Ys, -R)
    is true when \( R \) is \(<\), \(=\), or \(>\) according as Xs is a subset of Ys, equal to Ys, or a superset of Ys. Xs and Ys are ordered sets.

10.32 The PiLLoW Web Programming Library—
library(pillow)

The PiLLoW library ("Programming in Logic Languages on the Web") is a free Internet/WWW programming library for Logic Programming Systems that simplifies the process of writing applications for such environment. The library provides facilities for generating HTML or XML structured documents by handling them as Prolog terms, producing HTML forms, writing form handlers, processing HTML templates, accessing and parsing WWW documents (either HTML or XML), accessing code posted at HTTP addresses, etc.

PiLLoW is documented in its own reference manual, located in http://www.clip.dia.fi.upm.es/Software/pillow/pillow_doc_html/pillow_doc_toc.html (HTML) or http://www.clip.dia.fi.upm.es/Software/pillow/pillow_doc.ps (Postscript). The following points are worth noting wrt. the PiLLoW reference manual:

- PiLLoW is automatically installed with the SICStus Prolog distribution. No extra action needs to be taken.
- PiLLoW comes as a single library module, library(pillow).
  This subsumes the various load_package/1 and use_module/1 queries mentioned in the PiLLoW reference manual.

Further information can be found at the PiLLoW home page, http://clip.dia.fi.upm.es/Software/pillow/pillow.html.

10.33 Plunit Interface—library(plunit)
10.33.1 Introduction

This library module provides a Prolog unit-test framework, initially developed by Jan Wielemaker for SWI-Prolog. The code and documentation was subsequently adapted for SICStus Prolog by SICS. The module is a third-party product, and not part of SICStus Prolog proper.
Automatic testing of software during development is probably the most important Quality Assurance measure. Tests can validate the final system, which is nice for your users. However, most (Prolog) developers forget that it is not just a burden during development.

- Tests document how the code is supposed to be used.
- Tests can validate claims you make on the Prolog implementation. Writing a test makes the claim explicit.
- Tests avoid big applications saying “no” after modifications. This saves time during development, and it saves a lot of time if you must return to the application a few years later or you must modify and debug someone else’s application.

### 10.33.2 A Unit Test Box

Tests are written in normal Prolog. A unit test is a named collection of individual tests, enclosed within the directives:

```prolog
:- begin_tests(Unit[,Options]).
and:

:- end_tests(Unit).
```

They can be embedded inside a normal source module, or be placed in a separate test-file that loads the files to be tested. The individual tests are defined by rules of the form:

```prolog
test(Name[,Options]) :- test-body.
```

where Name is a ground term and Options is a list describing additional properties of the test. Here is a very simple example:

```prolog
:- use_module(library(lists)).

:- begin_tests(lists).

test(reverse) :-
    reverse([a,b], [b,a]).

:- end_tests(lists).
```

The optional second argument of the unit test declaration as well as of the individual test-heads defines additional processing options. The following options are available:

- **blocked(Reason)**
  
The test is currently disabled. Tests are flagged as blocked if they cannot be run for some reason. E.g. they crash Prolog, they rely on some service that is not available, they take too much resources, etc. Tests that fail but do not crash, etc. should be flagged using `fixme(Fixme)`. Reason should be an atom. If this option appears more than once in a list of options, all but one of the occurrences are silently ignored.
fixme(Reason)

Similar to blocked(Fixme), but the test is executed anyway. A summary is printed at the end of the test run. Reason should be an atom.

If this option appears more than once in a list of options, all but one of the occurrences are silently ignored.

condition(Goal)

Precondition for running the test. If the condition fails, then the test is skipped. The condition can be used as an alternative to the setup option. The only difference is that failure of a condition skips the test and is considered an error when using the setup option. Goal should be a callable.

If this option appears more than once in a list of options, the occurrences are combined into a conjunction, in the order they appear.

nondet

Available for individual test rules only. Unless this keyword appears in the option list, nondeterminate success of the test-body is considered an error.

It is an error to specify this more than once in a list of options.

forall(Generator)

Available for individual test rules only. Runs the same test for each solution of Generator. Each run invokes the setup and cleanup handlers. This can be used to run the same test with different inputs. If an error occurs, then the test is reported as ‘name (forall bindings = vars)’, where vars indicates the bindings of variables in Generator, which should be a callable.

It is an error to specify this more than once in a list of options.

setup(Goal)

Goal is run before the test-body. Typically used together with the cleanup option to create and destroy the required execution environment. Goal should be a callable.

If this option appears more than once in a list of options, the occurrences are combined into a conjunction, in the order they appear.

cleanup(Goal)

Goal is always called after completion of the test-body, regardless of whether it fails, succeeds or raises an exception. This option or call_cleanup/2 must be used by tests that require side effects that must be reverted after the test completes. Goal may share variables with a setup option and should be a callable.

If this option appears more than once in a list of options, the occurrences are combined into a conjunction, in the order they appear.
:- use_module(library(file_systems)).

:- begin_tests(hello).

create_file(Tmp) :-
    open(temp(plunit), write, Out, [if_exists(generate_unique_name)],
        current_stream(Tmp, write, Out),
        portray_clause(Out, hello(_World)),
        close(Out).

test(read, [setup(create_file(Tmp)), cleanup(delete_file(Tmp))]) :-
    see(Tmp),
    read(Term),
    seen,
    Term = hello(_).

:- end_tests(hello).

Please note: Do not place directives that load source code between :- begin_tests(Unit[,Options]) and :- end_tests(Unit). Loading source files in this context can cause spurious error messages.

The following options specify how to verify the result of the test-body, and are only available for individual test rules. Unless stated otherwise, it is an error if more than one of them appears in a list of options. In some cases there are additional restrictions on which options can appear together.

true
true(Test)

The test-body as well as the goal Test must succeed. Test defaults to true and should be a callable that typically shares variables with the test-body. This is the same as inserting the test at the end of the conjunction, but makes the test harness print a “wrong answer” message as opposed to a general failure message.

If this option appears more than once in a list of options, the occurrences are combined into a conjunction, in the order they appear.

test(badadd, [true(A =:= 4)]) :-
    A is 1 + 2.

will yield the error message:

! /home/matsc/sicstus4/doc/foo.pl:11:
!   test badadd: wrong answer (compared using =:=)
!   Expected: 4
!   Got: 3
all\((Answer\text{Term} Cmp \text{Instances})\)

Similar to true\((Answer\text{Term} Cmp \text{Instances})\), but used if you want to collect all solutions to a nondeterminate test. Answer\text{Term} should share variables with the test-body. Let All be the list of instances of Answer\text{Term} for each solution. Then the goal Cmp\((All, \text{Instances})\) must succeed. The tests in the example below are equivalent.

It is an error to specify this together with nondet.

\[
\text{test(all1, all(X} \equiv [1,2])) :- \\
(X = 1 ; X = 2).
\]

\[
\text{test(all2, true(Xs} \equiv [1,2])) :- \\
\text{findall(X, (X} = 1 ; X = 2), Xs).
\]

set\((Answer\text{Term} Cmp \text{Instances})\)

Similar to all\((Answer\text{Term} Cmp \text{Instances})\), but sorts the Answer\text{Term} instances before the comparison. The tests in the example below are equivalent.

It is an error to specify this together with nondet.

\[
\text{test(set1, set(X} \equiv [1,2])) :- \\
(X = 2 ; X = 1 ; X = 1).
\]

\[
\text{test(set2, true(Ys} \equiv [1,2])) :- \\
\text{findall(X, (X} = 2 ; X = 1 ; X = 1), Xs), \\
\text{sort(Xs, Ys)}.
\]

fail

The test-body must fail.

It is an error to specify this together with nondet.

exception\((Expected)\)

throws\((Expected)\)

The test-body must raise an exception Raised that is checked wrt. Expected using subsumes_term\((Expected, Raised)\). I.e. the raised exception must be more specific than the specified Expected.

It is an error to specify this together with nondet.

error\((ISO)\)

error\((ISO, Info)\)

A shorthand for exception(error\((ISO, Info)\)). Info defaults to an anonymous variable.

It is an error to specify this together with nondet.

10.33.3 Writing the Test-Body

The test-body is ordinary Prolog code. Without any options, the test-body must be designed to succeed determinately. Any other result is considered a failure. One of the options fail, true, exception, throws or error can be used to specify a different expected result. In this subsection we illustrate typical test-scenarios by testing built-in and library predicates.
10.33.3.1 Determinate Tests

Determinate tests are tests that must succeed exactly once, leaving no choicepoints behind. The test-body supplies proper values for the input arguments and verifies the output arguments. Verification can use test-options or be explicit in the test-body. The tests in the example below are equivalent.

```prolog
test(add1) :-
    A is 1 + 2,
    A =:= 3.

test(add2, [true(A =:= 3)]) :-
    A is 1 + 2.
```

The test engine verifies that the test-body does not leave a choicepoint. We illustrate this using the test below:

```prolog
test(member1) :-
    member(b, [a,b,c]).
```

Although this test succeeds, `member/2` leaves a choicepoint behind, which is reported by the test harness. To make the test silent, use one of the alternatives below.

```prolog
test(member2) :-
    memberchk(b, [a,b,c]).

test(member3) :-
    member(b, [a,b,c]), !.

test(member4, [nondet]) :-
    member(b, [a,b,c]).
```

10.33.3.2 Nondeterminate Tests

Nondeterminate tests succeed zero or more times. Their results can be tested using `findall/3` followed by a value-check. The following are equivalent tests:

```prolog
test(member5) :-
    findall(X, member(X, [a,b,c]), Xs),
    Xs == [a,b,c].

test(member6, true(Xs == [a,b,c])) :-
    findall(X, member(X, [a,b,c]), Xs).
```

10.33.3.3 Tests Expected to Fail

Tests that are expected to fail may be specified using the option `fail` or by negating the test-body using `\+`.
test(is_set) :-
    \+ is_set([a,a]).

test(is_set, [fail]) :-
    is_set([a,a]).

10.33.3.4 Tests Expected to Raise Exceptions
Tests that are expected to raise exceptions may be specified using the option exception(\textit{Expected}) or one of its equivalents, or by wrapping the test in \texttt{on\_exception/3} or \texttt{catch/3}. The following tests are equivalent:

\begin{verbatim}
  test(div01) :-
      on_exception(Excp, A is 1/0, true),
      subsumes_term(error(evaluation_error(zero_divisor),_), Excp).

  test(div02, [error(evaluation_error(zero_divisor))]) :-
      A is 1/0.

  test(div03, [error(evaluation_error(zero_divisor),_)]) :-
      A is 1/0.

  test(div04, [exception(error(evaluation_error(zero_divisor),_))]) :-
      A is 1/0.

  test(div05, [throws(error(evaluation_error(zero_divisor),_))]) :-
      A is 1/0.
\end{verbatim}

10.33.4 Running the Test-Suite
At any time, the tests can be executed by loading the program and running \texttt{run\_tests/[0,1,2]}:

\begin{verbatim}
run_tests
    Run all individual tests of all test-units.

run_tests(Spec)
run_tests(Spec, Options)
    Run only the specified tests. The following options are available:
    quiet      Suppresses informational messages.
    verbose    Prints informational messages, e.g. messages for each successful test. This is the default.
    passed(Count)
    failed(Count)
    skipped(Count)
    Binds \textit{Count} to the number of non-blocked tests that were successful, failed or skipped, respectively. A test is skipped if one of its conditions fails.
\end{verbatim}
The following will quietly run all tests, but print messages and fail if any of the tests were unsuccessful.

\[
\text{\texttt{| ?- run_tests(all, [failed(0), quiet]).}}
\]

\textit{Spec} should be one of:

- The atom \texttt{all}, to run all tests. This is the default i.e. what \texttt{run_tests/0} does.
- a term \texttt{Unit} where \texttt{Unit} is the name of a test-unit, denotes all individual tests of the test-unit \texttt{Unit}, or
- a term \texttt{Unit:Test} where \texttt{Unit} is the name of a test-unit and \texttt{Test} is one of its individual tests, denotes the given test only, or
- a term \texttt{Unit:List} where \texttt{Unit} is the name of a test-unit and \texttt{List} is a list of its individual tests, denotes the given list of tests, or
- finally, a list of terms of one of the above forms.

Running single tests is particularly useful for tracing a test, e.g.:

\[
\text{\texttt{| ?- trace, run_tests(lists:reverse).}}
\]

\texttt{run_tests/[0,1,2]} prints a report during execution. The \texttt{quiet} options suppresses informational messages; in its absence, messages are printed in full as follows.

First, each test-unit report begins with a header:

\[
\textbf{% PL-Unit: Unit}
\]

Then comes a message (success or failure) for all specified tests not marked as \texttt{blocked} or \texttt{fixme}. Success messages are informational; others are error or warning messages. Any errors encountered while executing options are also reported. To close the test-unit, a footer is printed:

\[
\textbf{% done}
\]

After all test-units, a summary report is printed, stating:

- how many tests passed resp. failed and how may test were skipped
- which tests were blocked
- details for each test marked as \texttt{fixme}

### 10.33.5 Tests and Production Systems

Most applications do not want the test-suite to end up in the final application. There are several ways to achieve this. One way is to place all tests in separate files and not to load the tests when creating the production environment. Another way is to wrap each unit test box in a pair of :\texttt{:- if(...), :}- \texttt{endif} directives. For example, the test could be whether the \texttt{plunit} module has been loaded:
:~ if(current_module(plunit)).
:~ begin_tests(Unit[,Options]).
...
:~ end_tests(Unit).
:~ endif.

Alternatively, you can reserve a system property e.g. `enable_unit_tests` to control whether unit tests should be enabled. The property is enabled if you run SICStus Prolog as:

```
% sicstus -Denable_unit_tests=true
```

Then your Prolog source file could have the structure:

```
:~ use_module(library(system), [environ/2]).
...
:~ if(environ(enable_unit_tests, true)).
:~ use_module(library(plunit)).
:~ begin_tests(Unit[,Options])
...
:~ end_tests(Unit)
:~ endif.
```

10.34 Process Utilities—library(process)

This package contains utilities for process creation.

A process is represented by a process reference, a ground compound term. Both SICStus and the operating system maintain a state for each such process reference and they must therefore be released, either explicitly with `process_release/1` or implicitly by `process_wait/[2,3]`. Process references are created with `process_create/[2,3]` if explicitly requested with the `process/1` option. Process references are required in order to obtain the exit status of a process after `process_create/[2,3]` has returned.

Many of the predicates can accept a numeric operating system process id ("PID") but since process ids are subject to re-use by the OS this is less reliable and does not work if the process has already exited.

10.34.1 Examples

The following illustrates some common tasks. The process library is portable and works on all supported platforms, including UNIX, Linux and Windows. However, the examples are by necessity platform dependent. Unless otherwise noted, the examples will work on UNIX and similar systems only.

(If you are looking for something like the old SICStus 3 `system:system/1` and `system:popen/3`, See [unsafe_system], page 726.)

1. Run the `date` command in the standard shell ‘sh’. The output of the command is sent
to the terminal:

| ?- process_create(path(sh),
|    ['-c', date]). |

2. Run the `date` command in the standard shell `sh`. Wait for the command to terminate before returning to Prolog. Fail if the process gets an error. The output of the command is sent to the terminal:

| ?- process_create(path(sh),
|    ['-c', date], [wait(exit(0))]). |

Using `wait/1` option in this way is a convenient way to ensure that the command has finished before Prolog continues.

3. Run the `date` command in the standard shell `sh`. The output of the command is received by Prolog:

| ?- process_create(path(sh),
|    ['-c', date], [stdout(pipe(S))],
|    read_line(S,L), close(S), atom_codes(Date,L). |

...,

Date = 'Fri Jan 24 12:59:26 CET 2014' ?

4. Pipe the output of the `date` command to a file:

| ?- process_create(path(sh),
|    ['-c', [date, '>', file('/tmp/foo.txt')]],
|    read_line(L), close(L), atom_codes(Date,L). |

5. Count the number of words in an atom, using the `wc` command:

| ?- process_create(path(wc), ['-w'],
|    [stdin(pipe(Out))],
|    read_line(S,L), close(S), atom_codes(Date,L). |

...,

N = 3

It may be preferable to let the input or output go via a file. This avoids deadlock in case the stream buffers fill up.

6. Count the number of unique words in a file, piping the output of the `uniq` command to the `wc` command:

| ?- process_create(path(sh),
|    ['-c', ['uniq ', file('/tmp/foo.txt')], ' | wc -w']],
|    [stdout(pipe(Out))],
|    read_line(S,L), close(S), atom_codes(Date,L). |

...,

N = 6

Note that quoting is a problem (and potential security issue), so never pass untrusted data, like file names, to the shell using `-c` (see [Quoting and Security], page 726).

7. Run the `make` command with the `-n` (dry run) option, discarding output, fail if it does not succeed:

| ?- process_create(path(make), ['-n'],
|    [stdout(null), wait(Exit)]),
|    Exit = exit(0). |
By using the wait/1 option, process_create/3 will not return until the subprocess has exited and its exit status is available.

8. Run ls on a home directory in a subshell using the user’s preferred shell:
   
   | ?- process_create('\$SHELL', ['-c', ['ls', ' ', file('~/')]])

9. Run a command with output piped from a file and input provided by Prolog. This is similar to popen('cat > ./myscript.sh',write,S) in SICStus 3. This example also shows one way to create a shell script which is useful when more advanced shell interaction is needed. (The created script outputs the most common line in its input. It is used in the next example.)
   
   | ?- process_create(path(sh), 
   |     ['-c', 
   |        'cat > ./myscript.sh & chmod a+x ./myscript.sh'], 
   |        [stdin(pipe(S))]), 
   |     write(S, '#! /bin/sh
'), 
   |     write(S, 'sort | uniq -c | sort -nr | head -n 1\n'), 
   |     close(S).

Please read [Quoting and Security], page 726, for problems with this approach.

10. Run a shell script with input piped from a file and output read by Prolog. This is similar to popen('.',./myscript.sh < ./somefile.txt',read,S) in SICStus 3.
   
   | ?- open('somefile.txt',write,OF), 
   |    write(OF,'hello
world
hello
hello
'),close(OF), 
   |    process_create(path(sh), 
   |        ['-c', './myscript.sh < ./somefile.txt'], 
   |        read_line(S, L), atom_codes(Line, L), close(S).

   ..., 
   
   Line = ' 3 hello' ?

Please read [Quoting and Security], page 726, for problems with this approach.

11. Run a goal in a SICStus subprocess (UNIX and Windows):
   
   | ?- process_create(application(sicstus), 
   |     ['-f', '--noinfo', '--nologo', 
   |        '--goal', 'read(X), call(X), halt.']
   | , [stdin(pipe(In)), stdout(pipe(Out))]), 
   |    format(In,'~q .~n', ![length(['h,e,l,l,o'], Len), 
   |        format('"q ."n', [Len]))]), 
   |    close(In), read(Out,Answer), close(Out).

   ..., 
   
   Answer = 5

12. Run notepad.exe on a file C:/foo.txt under Windows:
   
   | ?- process_create('\$SYSTEMROOT/notepad.exe', 
   |        [file('C:/foo.txt')])

13. Open a command shell in a separate window under Windows:
   
   | ?- process_create('\$COMSPEC', [],[window(true)])
10.34.1.1 Microsoft Windows Shell

On Windows, it is not possible to pass multiple parameters to a subprocess. When a subprocess is started, it receives exactly one argument and a quoting convention must be used to encode the parameters as the single argument actually passed to the process.

Unfortunately, there is no such universal quoting convention, every program can interpret its (single) argument in any way it sees fit.

Most programs use a convention established by the Microsoft C library. This is the convention used by `process_create/[2,3]` and it usually works well.

However, the command processor on Windows (`cmd.exe`) does not use the common convention and, except for very simple cases, passing arguments to `cmd.exe` will not work reliably.

**Please note:** Passing arguments to `cmd.exe` suffers from the same security vulnerabilities as those described in [Quoting and Security], page 726, below.

If you want to run commands using `cmd.exe`, it is best to create a batch (`'.bat'`) file with your commands and then tell `cmd.exe` to run the batch file.

The following example illustrates how to create a Windows batch file that pipes some output to a file (`COMSPEC` is an environment variable containing the path to `cmd.exe`):

```prolog
?- BatFileName='test.bat',
   open(BatFileName, write, S),
   write(S, 'date /T > "result.txt\n'), close(S),
   process_create('$$COMSPEC', ['/Q', '/C', file(BatFileName)],
                  [wait(exit(0))]),
   open('result.txt', read, R),
   read_line(R,L), close(R), atom_codes(Date,L).
```

```
...,
Date = '2014-01-27 ',
... ?
```

More recent versions of Windows come with a redesigned command line processor, `PowerShell`, which solves the problems associated with the traditional `cmd.exe` command line processor. In particular, it has a very general way to encode command line arguments, using `base-64` encoding. Currently, there is no direct support for PowerShell in this library, but the following example shows how to get the current week day both using a plain text command and with a base-64-encoded command.
| ?- Command = '(get-date).DayOfWeek',
  process_create(path(powershell),
  ['-Command', Command],
  [stdout(pipe(S))]),
  read_line(S,L),atom_codes(Day,L).
...,
Day = 'Monday',
... ?

| ?- EncodedCommand =
  'KABnAGUAdAAtAGQAYQB0AGUAKQAuAEQAYQB5AE8AZgBXAGUAZQBrAA==',
  process_create(path(powershell),
  ['-encodedCommand', EncodedCommand],
  [stdout(pipe(S))]),
  read_line(S,L),atom_codes(Day,L).
...,
Day = 'Monday',
... ?

where the EncodedCommand value was created by encoding the string '(get-date).DayOfWeek' using Base 64. See the PowerShell documentation for details.

### 10.34.2 Quoting and Security

It is easy to get undesired, and possibly harmful, effects if arbitrary data is passed without proper quoting to a shell. For instance, accepting arbitrary file names and passing them as part of a command line to a subshell can cause the shell to execute arbitrary, possibly malicious, code.

The following, vulnerable, predicates suffer from this problem. They are similar to predicates that existed in SICStus 3, and their fragility is one of the reasons process interaction was redesigned in SICStus 4.
% DO NOT USE. This code is vulnerable.  
% Similar to system:system/1 in SICStus 3.
unsafe_system(Cmd) :-
    % pass Cmd to shell, wait for exit, fail on error.
    process_create(path(sh), ['-c', Cmd], [wait(exit(0))]).

% DO NOT USE. This code is vulnerable.  
% Similar to system:popen/3 in SICStus 3.
unsafe_popen(Cmd, Direction, Pipe) :-
    % pass Cmd to shell, do not wait for exit,  
    % connect to stdin or stdout of subprocess.
    ( Direction == read ->
        process_create(path(sh), ['-c', Cmd], [stdout(pipe(Pipe))])
    ; Direction == write ->
        process_create(path(sh), ['-c', Cmd], [stdin(pipe(Pipe))])
    ).

Now consider the task of passing the contents of some file File to a command mycommand. You may think the following is a good idea (it is not!):

% DO NOT USE. This code is vulnerable.
unsafe_command(File, S) :-
    atom_concat('./mycommand < ', File, Cmd),
    unsafe_popen(Cmd, read, S).

That works as expected if the File argument is a plain file with no characters that has special meaning to the shell, e.g.

File = './somefile.txt',  
unsafe_command(File, S), read_line(S,L),close(S).

However, assume that the file name was obtained from some untrusted source and consider the following example:

File = '$(say bohoo)' ,  
unsafe_command(File, S), read_line(S,L),close(S).

depending on the system this can have a quite scary effect, and illustrates how shell meta characters in the constructed command line can lead to potentially dangerous results.

The safest way to interact with the shell is to create shell scripts and pass arguments to the scripts as separate arguments to the shell. E.g.

% A safer version
safer_command(File, S) :-
    % pass the file as the first argument to mycommand.
    process_create(path(sh),
        ['-c', file('./mycommand'), file(File)],
        [stdout(pipe(S))]).
Exported predicates:

```prolog
process_create(+File, +Args)
process_create(+File, +Args, :Options)
```

Start a new process running the program identified by `File` and the arguments specified in `Args`. The standard streams of the new process can be redirected to prolog streams. The exit status of the process can be obtained with `process_wait/[2,3]`.

`File`, is expanded as if by `absolute_file_name/2` (with arguments `access(execute)` and `file_type(executable)`) and is used to locate the file to execute.

The predefined file search path `path/1` (see Section 4.5 [ref-fdi], page 95) is especially useful here since it makes it easy to look up the names of an executable in the directories mentioned by the `PATH` environment variable. To run the Windows command shell `cmd` you would simply specify `path('cmd.exe')` (or `path(cmd)`), to start the UNIX Bash shell you would specify `path(bash)`.

`Args` is a list of argument specifications. Each argument specification is either a simple argument specification, see below, or a non-empty list of simple argument specifications. The expanded value of each element of `Args` is concatenated to produce a single argument to the new process. A **simple argument specification** can be one of:

- an atom The atom name is used as the expanded value. Some operating systems only support 7-bit ASCII characters here. Even when some larger subset of Unicode is used it may not work correctly with all programs.

  ```prolog
  file(File)
  ```

  `File`, an atom, is treated as a file name and subject to an operating system specific transformation to ensure file name syntax and character set is appropriate for the new process. This is especially important under Windows where it ensures that the full Windows Unicode character set can be used.

  **Please note:** The `File` part of `file(File)` is not subject to syntactic rewriting, the argument specification `file/1` only adjusts for differences in file name syntax and character encoding between SICStus and the operating system. You must explicitly call `absolute_file_name/[2,3]` if you want to expand file search paths etc.

`Options` is a list of options:

```prolog
stdin(Spec)
stdout(Spec)
stderr(Spec)
```

Each `Spec` specifies how the corresponding standard stream of the new process should be created. `Spec` can be one of:

```prolog
std
```

The new process shares the (OS level) standard stream with the Prolog process. This is the default. Note that,
especially under Windows, the Prolog process may not have any OS level standard streams, or the OS streams may not be connected to a console or terminal. In such a case you need to use pipe/[1,2] spec, see below, and explicitly read (write) data from (to) the process.

- **null** The stream is redirected to a null stream, i.e. a stream that discards written data and that is always at end of file when read.

```prolog
pipe(Stream) since release 4.0
pipe(Stream, StreamOptions) since release 4.3.2
```

A new Prolog stream is created and connected to the corresponding stream of the new process. StreamOptions is a list of options affecting the created stream. The supported stream options are: type/1, eol/1, and encoding/1, with the same meaning as for open/4 (see Section 11.3.148 [mpg-ref-open], page 1115).

The default, if no stream options are specified, is to use a text stream with the OS default character encoding. This stream must be closed using close/[1,2], it is not closed automatically when the new process exits.

```prolog
wait(-ExitStatus) since release 4.3
```

The call will not return until the sub-process has terminated. ExitStatus will be bound to the exit status of the process, as described for process_wait/2.

```prolog
process(Proc)
```

Proc will be bound to a process reference that can be used in calls to process_wait/[2,3] etc.. This process reference must be released, either explicitly with process_release/1 or implicitly by process_wait/[2,3]. It is often easier to use the wait/1 option if you just want to wait for the process to terminate.

- **detached(Bool)**

  Bool is either true or false. Specifies whether the new process should be “detached”, i.e. whether it should be notified of terminal events such as ^C interrupts. By default a new process is created detached if none of the standard streams are specified, explicitly or implicitly, as std.

- **cwd(CWD)**

  CWD is expanded as if by absolute_file_name/2 and is used as the working directory for the new process.

  By default, the working directory is the same as the Prolog working directory.

- **window(Bool)**

  Bool is either true or false (the default). Specifies whether the process should open in its own window.
Specifying window(true) may give unexpected results if the standard stream options stdin/1, stdout/1 and stderr/1 are specified with anything but their default value std.
Currently only implemented on Windows.

environment(Env)

Env is a list of VAR=VALUE for extra environment variables to pass to the sub-process in addition to the default process environment. VAR should be an atom. VALUE should be an argument specification, as described above. The VALUE is typically an atom but, especially on the Windows platform, it may be necessary to wrap file names in file/1 to ensure file paths are converted to the native format. See Section “System Properties and Environment Variables” in the SICStus Prolog Manual, for more information.

process_wait(+Process, -ExitStatus)
process_wait(+Process, -ExitStatus, +Options)

Wait for a process to exit and obtain the exit status.

Process is either a process reference obtained from process_create/3 or an OS process identifier. Specifying a process identifier is not reliable. The process identifier may have been re-used by the operating system. Under Windows, it is not possible to obtain the exit status using a process identifier if the process has already exited.

ExitStatus is one of:

exit(ExitCode)
The process has exited with exit code ExitCode. By convention processes use exit code zero to signify success and a (positive) non-zero value to specify failure.

killed(SignalNumber)
UNIX only, the process was killed by signal SignalNumber (a positive integer).

timeout
The timeout/1 option was specified and the process did not exit within the specified interval. In this case the process reference is not released, even if the release/1 option is specified.

Options is a list of options:

timeout(Seconds)
Specify a maximum time, in seconds, to wait for the process to terminate. Seconds should be an integer or floating point number or the atom infinite (the default) to specify infinite wait. If the specified timeout interval passes before the process exits, process_wait/3 exits with ExitStatus set to timeout and the process reference is not released.

Currently the UNIX implementation supports only timeout values 0 (zero) and infinite.
release(Bool)
    Bool is either true (the default) or false. Specifies whether the
    process reference should be released when process_wait/3 exits
    successfully.

process_id(-PID)
    Obtain the process identifier of the current (i.e. Prolog) process.

process_id(+Process, -PID)
    Obtain the process identifier of the process reference Process.

is_process(+Thing)
    Returns true if Thing is a process reference that has not been released.

process_release(+Process)
    Release a process reference Process that has previously been obtained from
    process_create/3. This ensures that Prolog and the operating system can
    reclaim any resources associated with the process reference.
    Usually you would not call this. Either do not request the process reference
    when calling process_create/3 or let process_wait/[2,3] reclaim the pro-
    cess reference when the process terminates.

process_kill(+Process)
process_kill(+Process, +SignalSpec)
    Send a signal to the process designated by Process. The signal can either be a
    non-negative integer or a signal name as an (all uppercase) atom.
    The following signal names are accepted under UNIX if the platform defines
    them: SIGABRT, SIGALRM, SIBUS, SIGCHLD, SIGCONT, SIGFPE, SIGHUP, SIGILL,
    SIGINT, SIKILL (the default), SIGPIPE, SIGPOLL, SIGPROF, SIGQUIT, SIGSEGV,
    SIGSTOP, SIGSYS, SIGTERM, SIGTRAP, SIGTSTP, SIGTTIN, SIGTTOU, SIGURG,
    SIGUSR1, SIGUSR2, SIGVTALRM, SIGXCPU and SIGXFSZ. However, many of these
    do not make sense to send as signals.
    Under Windows, which does not have the signal concept, the signal name
    SIGKILL (the default) is treated specially and terminates the process with
    TerminateProcess(Process, -1). Please note: Using process_kill/[2,3]
    on Windows is not recommended. Also, on Windows, the call may throw an
    error if the process has already exited.

10.35 PrologBeans Interface—library(prologbeans)

10.35.1 Introduction
    PrologBeans is a package for integrating Prolog with applications written in other languages.
    Currently Java and .NET are supported. PrologBeans is based on running Prolog as a
    separate server process, and the other part of the application as a client process. This
    makes PrologBeans automatically distributable since the server and the client can run on
    different computers anywhere on the Internet.

    PrologBeans is designed to be used when client applications need to send queries to a
    Prolog server (and less intended for showing a GUI from a Prolog program). One typical
application would be to connect a Java or .NET based web application to a Prolog server (see examples later).

PrologBeans setup where the Prolog application serves several users accessing both via a web application server and a .NET GUI.

The PrologBeans package consists of two parts. The Prolog server is a library module, `library(prologbeans)`. The client is a class library, `prologbeans.jar` for Java (installed on all platforms), and `prologbeans.dll` for .NET (only installed on Microsoft Windows platforms).
10.35.2 Features

The current version of PrologBeans is designed to be used mainly as a connection from the client (Java or .NET) to Prolog. Current features are:

- Socket based communication [Java and .NET]
- Allows the client application and Prolog server to run on different machines [Java and .NET]
- Multiple client applications can connect to same Prolog server [Java and .NET]
- Client applications can make use of several Prolog servers [Java and .NET]
- Allows Java Applets to access Prolog server [Java]
- Platform independent (e.g. any platform where Prolog and Java or .NET exist) [Java and .NET]
- Simplifies the use of Prolog in Java application servers (Tomcat, etc) [Java]
- Prohibits unwanted use of Prolog server by host control (only specified hosts can access the Prolog server) [Java and .NET]
- Supports Java servlet sessions [Java]
- Supports JNDI lookup (Java Naming and Directory Interface) [Java]
- Supports .NET server pages (ASPX) [.NET]

10.35.3 A First Example

This section provides an example to illustrate how PrologBeans can be used. This application has a simple Java GUI where the user can enter expressions that will be evaluated by an expression evaluation server.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import se.sics.prologbeans.*;

public class EvaluateGUI implements ActionListener {

    private JTextArea text = new JTextArea(20, 40);
    private JTextField input = new JTextField(36);
    private JButton evaluate = new JButton("Evaluate");
    private PrologSession session = new PrologSession();

    public EvaluateGUI() throws java.io.IOException {
        if ((Integer.getInteger("se.sics.prologbeans.debug", 0)).intValue() != 0) {
            session.setTimeout(0);
        }
        JFrame frame = new JFrame("Prolog Evaluator");
        Container panel = frame.getContentPane();
        panel.add(new JScrollPane(text), BorderLayout.CENTER);
        JPanel inputPanel = new JPanel(new BorderLayout());
        inputPanel.add(input, BorderLayout.CENTER);
        inputPanel.add(evaluate, BorderLayout.EAST);
        panel.add(inputPanel, BorderLayout.SOUTH);
        text.setEditable(false);
        evaluate.addActionListener(this);
        input.addActionListener(this);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
        session.connect();
    }

    public void actionPerformed(ActionEvent event) {
        try {
            Bindings bindings = new Bindings().bind("E",
                input.getText() + '.");
            QueryAnswer answer = session.executeQuery("evaluate(E,R)", bindings);
            PBTerm result = answer.getValue("R");
            if (result != null) {
                text.append(input.getText() + " = " + result + '\n');
                input.setText("");
            } else {
                text.append("Error: " + answer.getError() + '\n');
            }
        } catch (Exception e) {
            text.append("Error when querying Prolog Server: " +
                e.getMessage() + '\n');
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws java.io.IOException {
        new EvaluateGUI();
    }
}
The Java code above first sets up the GUI with a text area for showing results, a text field for entering expressions, and a button for requesting an evaluation (the constructor EvaluateGUI()). It will also add itself as ActionListener on both the text field and the button. The method actionPerformed(ActionEvent event) will be called whenever the user has pressed RET or clicked on the button. actionPerformed first binds the variable $E$ to the value of the text field, and then sends the query to the Prolog server with session.executeQuery("evaluate(E,R)", bindings). If everything goes well, then the Prolog server will return an answer (bound to $R$), which will be appended to the text area.

```
:- module(evaluate,[main/0,my_predicate/2]).
:- use_module(library(prologbeans)).
:- use_module(library(codesio), [read_from_codes/2]).

%%% Register acceptable queries and start the server (using default port)
main:-
    register_query(evaluate(C,P), my_predicate(C,P)),
    start.

%%% We have received a code list
%%% which needs to be converted into an expression
my_predicate(Chars, P) :-
    read_from_codes(Chars, X),
    P is X.
```

The Prolog code above first defines the module and imports the needed modules. Then, in the main/0 predicate, it configures the server to answer queries on the form evaluate(C,P) and starts the server. The last few lines defines the predicate my_predicate(Chars, P), which is the predicate that performs the evaluation. Note that, here, the expression to evaluate is represented as a code list and must be converted into a term before evaluation.

In general, arbitrary Prolog terms can be passed to the client via this mechanism, including terms containing unbound variables. However, any unbound variables with attributes or blocked goals attached to them will be replaced by plain, brand new variables. This is analogous to the way attributed variables are handled in terms that are written, copied, asserted, gathered as solutions to findall/3 and friends, or raised as exceptions. If the attributes must be passed to the client, then the Prolog code can obtain them by using copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 129).

Please note: the environment variable SP_PATH as used here is meant to be a shorthand (see Section 6.1 [CPL Notes], page 290), and does not need to be set explicitly.

To start the example, first start the Prolog server by going to the %SP_PATH%/library/prologbeans/examples/evaluate (Windows), or $SP_PATH/library/prologbeans/examples/evaluate (UNIX/Linux) directory and type:

```
% sicstus -l evaluate.pl --goal "main."
```

To start the GUI type (from the same directory as above):
> java -classpath "%SP_PATH%\bin\prologbeans.jar;." EvaluateGUI (Windows), or
% java -classpath "$SP_PATH/bin/prologbeans.jar;." EvaluateGUI (UNIX)

10.35.4 Prolog Server Interface

The Prolog interface is based on the idea of a Prolog server that provides its service by answering queries from external applications (typically Java applications). The Prolog interface in PrologBeans is defined in library(prologbeans), which implements the Prolog server and exports the following predicates:

\[ start(\text{+Options}) \]

starts the Prolog server using the options specified. Please note: \( \text{start}/[0,1] \)
will not return until a server shutdown occurs. \text{Options} should be a list of zero or more of:

\[ \text{port(}\text{?Val}) \]

an integer denoting the port number of the Prolog server. The default port, if no port option is present, is 8066. In the case of the default port being used, the Socket Reuse Address bit will be set in the underlying sockets layer. If \text{Val} is a variable, then some unused port will be selected by the OS, the actual port number can be obtained with \text{get_server_property/1}, typically from a \text{server_started} event listener.

\[ \text{accepted_hosts(}\text{+Val}) \]

a list of atoms denoting the hosts (in form of IP-addresses) that are accepted by the Prolog server (default: \{‘127.0.0.1’\}).

\[ \text{session_timeout(}\text{+Val}) \]

an integer denoting the duration of a session in seconds. The session will be removed if it has been inactive more than this timeout when the session garbage collect starts. If the session timeout is set to zero, then there will be no garbage collection of sessions (default: 0).

\[ \text{session_gc_timeout(}\text{+Val}) \]

an integer denoting the minimum time in seconds between two consecutive session garbage collections. If the timeout is set to zero, then there will be no garbage collection of sessions (default: 0).

For example:

\[ \text{:- start([port(7500),}
\text{accepted_hosts([‘127.0.0.1’,’99.8.7.6’])]).} \]

\[ \text{shutdown(\text{+Mode})} \]

shuts down the server and closes the sockets and the streams after processing all available input. There are three modes:

\[ \text{now} \]

as soon as possible (default).
no_sessions

after all sessions have ended (all sessions have either been explicitly removed by request of the client application, or they have been garbage collected). **Please note:** there can still be connections to the Prolog server even when all sessions have ended.

no_connections

after all connections to the Prolog server are closed. **Please note:** there can still be user sessions left when all connections have been closed.

register_query(+Query, :PredicateToCall)

register_query(+Query, :PredicateToCall, +SessionVar)

registers a query and the corresponding goal. Before the registration, any previously registered query matching Query will be removed (as if by unregister_query(Query)). The goal PredicateToCall will be called when a query matching Query is received.

Typically, Query and PredicateToCall share variables that are instantiated by the call, and the instantiated Query is passed back to the client. In general, variable bindings can be arbitrary Prolog terms, including terms containing unbound variables. However, any unbound variables with attributes or blocked goals attached to them will be replaced by plain, brand new variables. This is analogous to the way attributed variables are handled in terms that are written, copied, asserted, gathered as solutions to findall/3 and friends, or raised as exceptions. If the attributes must be passed to the client, then the Prolog code can obtain them by using copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 129).

The goal is called determinately, i.e. it is never backtracked into. If it fails, then the term no is passed to the client instead of the instantiated Query. If it raises an exception E, then the term error(E) is passed to the client instead of the instantiated Query.

Before calling the query, the variable SessionVar, if given, is bound to the id of the current session. Session ids are typically generated in web applications that track users and mark all consecutive web-accesses with the same session id.

unregister_query(+Query)

unregisters all queries matching Query.

session_get(+SessionID, +ParameterName, +DefaultValue, -Value)

returns the value of a given parameter in a given session. If no value exists, then it will return the default value. Arguments:

*SessionID* is the id of the session for which values have been stored

*ParameterName* an atom, is the name of the parameter to retrieve

*DefaultValue* is the value that will be used if no value is stored

*Value* is the stored value or the default value if nothing was stored
session_put(+SessionID, +ParameterName, +Value)
stores the value of the given parameter. Please note: any pre-existing value for this parameter will be overwritten. Note that session_put/3 will not be undone when backtracking (the current implementation is based on assert). Arguments:

SessionID is the id of the session for the values to store

ParameterName an atom, is the name of the parameter to store

Value the value to be stored

register_event_listener(+Event, :PredicateToCall)

register_event_listener(+Event, :PredicateToCall, -Id)
Registers PredicateToCall to be called (as if by once(PredicateToCall)) when the event matching Event occurs (event matching is on principal functor only). If the goal fails or raises an exception, then a warning is written to user_error but the failure or exception is otherwise ignored. Arguments:

Event is the event template; see below.

PredicateToCall an arbitrary goal.

Id becomes bound to a (ground) term that can be used with unregister_event_listener/1 to remove this event listener.

The predefined events are as follows:

session_started(+SessionID)
called before the first call to a query for this session

session_ended(+SessionID)
called before the session is about to be garbage collected (removed)

server_started
called when the server is about to start (enter its main loop)

server_shutdown
called when the server is about to shut down

Attempt to register an event listener for other events than the predefined events will throw an exception.

More than one listeners can be defined for the same event. They will be called in some unspecified order when the event occurs.

unregister_event_listener(+Id)
Unregister a previously registered event listener. The Id is the value returned by the corresponding call to register_event_listener/3. It is an error to attempt to unregister an event listener more than once.
10.35.5 Java Client Interface

The Java interface is centered around the class `PrologSession`, which represents a connection (or session) to a Prolog server. `PrologSession` contains static methods for looking up named `PrologSession` instances using JNDI (Java Naming and Directory Interface) as well as methods for querying the Prolog server. Other important classes are: `QueryAnswer`, which contains the answer for a query sent to the Prolog server; `PBTerm`, which represents a Prolog term; and `Bindings`, which supports stuffing of variable values used in queries.

General information about Java, Servlets and JNDI is available at the Java Technology site: [http://java.sun.com/](http://java.sun.com/)

A brief description of the provided Java classes are presented below. More information about the Java APIs is available in the JavaDoc files on the page [https://sicstus.sics.se/documentation.html](https://sicstus.sics.se/documentation.html).

**PrologSession**

The `PrologSession` object is the connection to the Prolog server. The constructor `PrologSession()` creates a `PrologSession` with the default settings (`host = localhost, port = 8066`).

**QueryAnswer**

The `QueryAnswer` contains the answer (new bindings) for a query (or the error that occurred during the query process).

**PBTerm**

The `PBTerm` object is for representing parsed Prolog terms.

**Bindings**

`Bindings` is used for binding variables to values in a query sent to the Prolog. The values will be automatically stuffed before they are sent to the Prolog server.

10.35.6 Java Examples

The PrologBeans examples for Java can be found in the directory corresponding to the file search path `pbexamples`, defined as if by a clause:

```prolog
user:file_search_path(pbexamples, library('prologbeans/examples')).
```

10.35.6.1 Embedding Prolog in Java Applications

If you have an advanced Prolog application that needs a GUI, then you can write a stand-alone Java application that handles the GUI and set up the Prolog server to call the right predicates in the Prolog application.

An example of how to do this can be found under the `pbexamples(evaluate)` directory (see the example code in Section 10.35.3 [PB First Example], page 733).

Another example of this is `pbexamples(pbtest)`, which illustrates several advanced features like:

- registering several queries
- listening to server events (`server_started`)
• shutting down the Prolog server from Java
• starting up the Prolog server from Java
• using dynamic (OS assigned) ports for the Java/Prolog communication

The example is run by executing the Java program PBTest:

    > java -classpath "%SP_PATH%\bin\prologbeans.jar;:" PBTest (Windows), or
   % java -classpath "$SP_PATH/bin/prologbeans.jar;:" PBTest (UNIX)

10.35.6.2 Application Servers

If you want to get your Prolog application to be accessible from an intranet or the Internet, then you can use this package to embed the Prolog programs into a Java application server such as Tomcat, WebSphere, etc.

An example of how to do this is provided in pbexamples(sessionsum). This example uses sessions to keep track of users so that the application can hold a state for a user session (as in the example below, remember the sum of all expressions evaluated in the session).
The example shows the code of a JSP (Java Server Page). It makes use of the method `PrologSession.getPrologSession(String jndiName, HTTPSession session)`, which uses JNDI to look up a registered `PrologSession`, which is connected to the Prolog server. The variable `session` is in a JSP bound to the current `HttpServletRequest`, and the variable `request` is bound to the current `HTTPRequest`. Since the `HttpServletRequest` object `session` is specified all queries to the Prolog server will contain a session id. The rest of the example shows how to send a query and output the answer.

Example usage of sessions (from the `sessionsum` example) is shown below, and is from `pbexamples('sessionsum/sessionsum.pl'):`
:- module(sessionsum, [main/0, sum/5]).
:- use_module(library(prologbeans)).
:- use_module(library(codesio), [read_from_codes/2]).

%%% Register the acceptable queries (session based)
main: -
    register_query(sum(C, Sum, Average, Count),
        sum(C, Session, Sum, Average, Count),
        Session),
    start.

%%% The sum predicate which gets the information from a session database,
%%% makes some updates and then stores it back in to the session store
%%% (and returns the information back to the application server)
sum(ExprChars, Session, Sum, Average, Count) :-
    session_get(Session, sum, 0, OldSum),
    session_get(Session, count, 0, OldCount),
    read_from_codes(ExprChars, Expr),
    Val is Expr,
    Sum is OldSum + Val,
    Count is OldCount + 1,
    Average is Sum / Count,
    session_put(Session, sum, Sum),
    session_put(Session, count, Count).

In this example a query sum/4 is registered to call sum/5 where one of the variables, Session
will be bound to the session id associated to the query. The sum/5 predicate uses the
session_get/4 predicate to access stored information about the particular session, and
then it performs the evaluation of the expression. Finally, it updates and stores the values
for this session.

10.35.6.3 Configuring Tomcat for PrologBeans

This section will briefly describe how to set up a Tomcat server so that is it possible to test
the example JSPs. Some knowledge about how to run Tomcat and how to set up your own
web application is required. Detailed information about Tomcat is available at http://
jakarta.apache.org/tomcat/.

Assuming that the environment variable CATALINA_HOME is set to the installation di-
rectory of Tomcat, do the following:

1. Create the directory $CATALINA_HOME/webapps/PB_example
2. Copy
   the
   file pbexamples('sessionsum/sessionsum.jsp') to $CATALINA_HOME/webapps/PB_  
   example/sessionsum.jsp
3. Create the directory $CATALINA_HOME/webapps/PB_example/WEB-INF/lib
4. Copy the file $SP_PATH/bin/prologbeans.jar to $CATALINA_HOME/webapps/PB_  
   example/WEB-INF/lib/prologbeans.jar
5. Create the directory $CATALINA_HOME/webapps/PB_example/META-INF

6. Create the file $CATALINA_HOME/webapps/PB_example/META-INF/context.xml with the following content:

```xml
<Context docBase="PB_example">
    <Resource name="prolog/PrologSession" auth="Container"
        type="se.sics.prologbeans.PrologSession"
        factory="org.apache.naming.factory.BeanFactory" />
</Context>
```

7. Create the file $CATALINA_HOME/webapps/PB_example/WEB-INF/web.xml with the following content:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <resource-env-ref>
        <description>
            Object factory for PrologSession instances.
        </description>
        <resource-env-ref-name>
            prolog/PrologSession
        </resource-env-ref-name>
        <resource-env-ref-type>
            se.sics.prologbeans.PrologSession
        </resource-env-ref-type>
    </resource-env-ref>
</web-app>
```

8. Start SICStus, load sessionsum.pl and run main.

9. Start the Tomcat server.

10. In a web browser, enter http://localhost:8080/PB_example/sessionsum.jsp

### 10.35.7 .NET Client Interface

The class `PrologSession` in the .NET interface represents a connection to a Prolog server. `PrologSession` contains methods for establishing a connection and querying the Prolog server. Other important classes are: `QueryAnswer`, which contains the answer for a query sent to the Prolog server; `PBTerm`, which represents a Prolog term; and `Bindings`, which supports stuffing of variable values used in queries.

The `PrologSession` object is the connection to the Prolog server. The constructor `PrologSession()` creates a `PrologSession` with the default settings (`host = localhost`, `port = 8066`).
The interface is almost exactly as the Java version. See the C# source code (library/prologbeans.NET/*.cs) or the JavaDoc, for details.

10.35.8 .NET Examples

The PrologBeans examples for .NET can be found in the directory corresponding to the file search path pbnetexamples, defined as if by a clause:

user:file_search_path(pbnetexamples, library('prologbeans.NET/examples')).

10.35.8.1 C# Examples

.NET Embedding. If you have an advanced Prolog application that needs a GUI, then you can write a stand-alone .NET application that handles the GUI and set up the Prolog server to call the right predicates in the Prolog application.

An example of how to do this can be found under the pbnetexamples('evaluate.NET') directory. This example is the C# version of the example shown in Section 10.35.3 [PB First Example], page 733).

To start the example, first start the Prolog server by going to the pbnetexamples('evaluate.NET') directory and type:

> sicstus -l evaluate.pl --goal "main."

To start the GUI type (from the same directory as above):

> run.bat

Another example of this is pbnetexamples('pbtest.NET'), which illustrates several advanced features like:

- registering several queries
- listening to server events (server_started)
- shutting down the Prolog server from .NET
- starting up the Prolog server from .NET
- using dynamic (OS assigned) ports for the .NET/Prolog communication

The example is run by executing the C# program PBTest:

> PBTest

ASPX Servers Pages. If you want to get your Prolog application to be accessible from an intranet or the Internet, then you can use this package to embed the Prolog programs into a .NET ASP page which can be served by e.g. Internet Information Services.

An example of how to do this is provided in pbnetexamples('prologasp.NET/eval.aspx'). Consult your IIS documentation for how to configure it for an ASPX page. The ASPX example has a number of security vulnerabilities and is for illustrative purposes only. Consult with an expert.
10.35.8.2 Visual Basic Example

A Visual Basic .NET example can be found in `pbnetexamples('vb_examples.NET/calculator')`. It is a simple calculator similar to the first C# `EvaluateGUI` example in Section 10.35.3 [PB First Example], page 733. This example is in the form of a Visual Studio project.

To run the example:

1. Open the project files in Visual Studio .NET
2. Add a reference in Visual Studio .NET to the installed `prologbeans.dll`
3. Start sicstus with the following command:
   ```
   sicstus -l %SP_PATH%/library/prologbeans/examples/evaluate/evaluate --goal "main."
   ```
4. Build and run the example in Visual Studio .NET

10.36 Queue Operations — `library(queues)`

This module provides an implementation of queues, where you can

- create an empty queue
- add an element at either end of a queue
- add a list of elements at either end of a queue
- remove an element from the front of a queue
- remove a list of elements from the front of a queue
- determine the length of a queue
- enumerate the elements of a queue
- recognize a queue
- print a queue nicely

The representation was invented by Mark Johnson of the Center for the Study of Language and Information. All operations are fast.

Exported predicates:

`empty_queue(?Queue)`

is true when `Queue` represents an empty queue. It can be used to test whether an existing queue is empty or to make a new empty queue.

`singleton_queue(?X, ?Queue)`

is true when `Queue` is a queue with just one element `X`.

`portray_queue(+Queue)`

writes a queue out in a pretty form, as `Queue[elements]`. This form cannot be read back in, it is just supposed to be readable. While it is meant to be called only when `is_queue(Queue)` has been established, as by `user:portray(Q) :- is_queue(Q), !, portray_queue(Q).` it is also meant to work however it is called.
is_queue(+Queue)

is true when Queue is a queue. The elements of Queue do not have to be instantiated, and the Back of the Queue may or may not be. It can only be used to recognize queues, not to generate them. To generate queues, use queue_length(Queue, _).

queue_head(+Queue, -Head)

is true when Head is the first element of the given Queue. It does not remove Head from Queue; Head is still there afterwards. It can only be used to find Head, it cannot be used to make a Queue.

queue_tail(?Queue, ?Tail)

is true when Queue and Tail are both queues and Tail contains all the elements of Queue except the first. Note that Queue and Tail share structure, so that you can add elements at the back of only one of them. It can solve for either argument given the other.

queue_cons(?Head, ?Tail, ?Queue)

is true when Head is the head of Queue and Tail is the tail of Queue, that is, when Tail and Queue are both queues, and the elements of the Queue are Head followed by the elements of Tail in order. It can be used in either direction, so

\[
\text{queue_cons}(\text{+Head}, \text{+Q0}, \text{-Q}) \quad \text{adds Head to Q0 giving Q}
\]

\[
\text{queue_cons}(\text{-Head}, \text{-Q}, \text{+Q0}) \quad \text{removes Head from Q0 giving Q}
\]

queue_last(?Last, ?Queue)

is true when Last is the last element currently in Queue. It does not remove Last from Queue; it is still there. This can be used to generate a non-empty Queue. The cost is \(O(|\text{Queue}|)\).

queue_last(+Fore, +Last, -Queue)

is true when Fore and Queue are both lists and the elements of Queue are the elements of Fore in order followed by Last. This is the operation which adds an element at the end of Fore giving Queue; it is not reversible, unlike queue_cons/3, and it side-effects Fore, again unlike queue_cons/3.

append_queue(?List, ?Queue0, ?Queue)

is true when Queue is obtained by appending the elements of List in order at the front of Queue0, e.g. append_queue([a,b,c], Queue[d,e], Queue[a,b,c,d,e]). Use

\[
\text{append_queue}([+X1,...,+Xn], +Q0, -Q) \quad \text{to add X1,...,Xn to Q0 giving Q}\n\]

\[
\text{append_queue}([-X1,...,-Xn], -Q, +Q0) \quad \text{to take X1...Xn from Q0 giving Q}\n\]

The cost is \(O(n)\) and the operation is pure.

queue_append(+Queue0, +List, -Queue)

is true when Queue is obtained by appending the elements of List in order at the rear end of Queue0, e.g. append_queue(Queue[a,b,c], [d,e], Queue[a,b,c,d,e]). This is like queue_last/3; it side-effects Queue0.

list_queue(?List, ?Queue)

is true when Queue is a queue and List is a list and both have the same elements in the same order. list_queue/2 and queue_list/2 are the same except for argument order.
queue_list(?Queue, ?List)
is true when Queue is a queue and List is a list and both have the same elements in the same order. queue_list/2 and list_queue/2 are the same except for argument order.

queue_length(?Queue, ?Length)
is true when Queue is a queue having Length elements. It may be used to determine the Length of a Queue or to make a Queue of given Length.

queue_member(?Element, +Queue)
is true when Element is an element of Queue. It could be made to generate queues, but that would be rather inefficient. It bears the name queue_member/2 because it is prepared to enumerate Elements.

queue_memberchk(+Element, +Queue)
is true when the given Element is an element of Queue. Once it finds a member of Queue which unifies with Element, it commits to it. Use it to check a ground Element.

map_queue(:Pred, +Queue[X1,...,Xn])
succeeds when Pred(Xi) succeeds for each element Xi of the Queue.

map_queue(:Pred, +Queue[X1,...,Xn], ?Queue[Y1,...,Yn])
succeeds when Pred(Xi,Yi) succeeds for each corresponding pair of elements Xi, Yi of the two queues.

map_queue_list(:Pred, ?Queue[X1,...,Xn], ?[Y1,...,Yn])
succeeds when Pred(Xi, Yi) is true for each corresponding pair Xi,Yi of elements of the Queue and the List. It may be used to generate either of the sequences from the other.

map_list_queue(:Pred, ?[X1,...,Xn], ?Queue[Y1,...,Yn])
succeeds when Pred(Xi, Yi) is true for each corresponding pair Xi,Yi of elements of the List and the Queue. It may be used to generate either of the sequences from the other.

some_queue(:Pred, +Queue[X1,...,Xn])
succeeds when Pred(Xi) succeeds for some Xi in the Queue. It will try all ways of proving Pred(Xi) for each Xi, and will try each Xi in the Queue. somechk_queue/2 is to some_queue/2 as memberchk/2 is to member/2; you are more likely to want somechk_queue/2. This acts on backtracking like member/2; Queue should be proper.

some_queue(:Pred, +Queue[X1,...,Xn], ?Queue[Y1,...,Yn])
is true when Pred(Xi, Yi) is true for some i.

somechk_queue(:Pred, +Queue[X1,...,Xn])
is true when Pred(Xi) is true for some i, and it commits to the first solution it finds (like memberchk/2).

somechk_queue(:Pred, +Queue[X1,...,Xn], ?Queue[Y1,...,Yn])
is true when Pred(Xi, Yi) is true for some i, and it commits to the first solution it finds (like memberchk/2).
10.37 Random Number Generator—library(random)

This library module provides a random number generator using algorithm AS 183 from the Journal of Applied Statistics as the basic algorithm. This algorithm is not cryptographically secure.

The state of the random number generator corresponds to a term \texttt{random}(X,Y,Z,B) where \(X\) is an integer in the range \([1,30268]\), \(Y\) is an integer in the range \([1,30306]\), \(Z\) is an integer in the range \([1,30322]\), and \(B\) is a nonzero integer.

The random generator starts with the same default initial state on each run. See \texttt{setrand(1)} for a way to initialize the random number generator with a new state when a more unpredictable initial state is desired.

Exported predicates:

\texttt{getrand(-RandomState)}

returns the random number generator’s current state

\texttt{setrand(+RandomState)}

sets the random number generator’s state to \texttt{RandomState}. \texttt{RandomState} can either be a random state previously obtained with \texttt{getrand/1}, or an arbitrary integer. The latter is useful when you want to initialize the random state to a fresh value. If \texttt{RandomState} is not an integer or a valid random state, it raises an error.

The initial state of the random number generator is always the same. This means that subsequent runs will generate the same sequence of random numbers. This gives reproducible results which is good for testing but not always desirable. One way to get a new initial state is to set it based on the current time, e.g. by calling something like the following predicate early in your program:

\begin{verbatim}
:- use_module(library(system), [now/1]).
init_random_state :-
    now(X),
    setrand(X).
\end{verbatim}

\texttt{maybe(+Probability)}

succeeds determinately with probability \(1/2\), fails with probability \(1/2\). We use a separate "random bit" generator for this test to avoid doing much arithmetic.

\texttt{maybe(+P, +N)}

succeeds determinately with probability \(P/N\), where \(0 =< P =< N\) and \(P\) and \(N\) are integers. If this condition is not met, it fails. It is equivalent to \texttt{random(0, N, X), X < P}, but is somewhat faster.

\texttt{random(-Uniform)}

unifies \texttt{Uniform} with a new random number in \([0.0,1.0)\)
random(+L, +U, −R)
unifies R with a random integer in [L,U) when L and U are integers (note that U will never be generated), or to a random floating number in [L,U) otherwise.

random_member(−Elem, +List)
unifies Elem with a random element of List, which must be proper. Takes O(N) time (average and best case).

random_select(−Elem, −List, −Rest)
unifies Elem with a random element of List and Rest with all the other elements of List (in order). Either List or Rest should be proper, and List should/will have one more element than Rest. Takes O(N) time (average and best case).

random_subseq(+List, −Sbsq, −Cmpl)
unifies Sbsq with a random sub-sequence of List, and Cmpl with its complement. After this, subseq(List, Sbsq, Cmpl) will be true. Each of the 2**|List| solutions is equally likely. Like its name-sake subseq/3, if you supply Sbsq and Cmpl it will interleave them to find List. Takes O(N) time. List should be proper.

random_permutation(−List, −Perm)
unifies Perm with a random permutation of List. Either List or Perm should be proper, and they should/will have the same length. Each of the N! permutations is equally likely, where length(List, N). This takes O(N lg N) time and is bidirectional.

random_perm2(A, B, X, Y)
unifies X,Y = A,B or X,Y = B,A, making the choice at random, each choice being equally likely. It is equivalent to random_permutation([A,B], [X,Y]).

random_numlist(+P, +L, +U, −List)
where P is a probability (0..1) and L=<U are integers unifies List with a random subsequence of the integers L..U, each integer being included with probability P.

10.38 Rem’s Algorithm—library(rem)
This library module maintains equivalence classes using Rem’s algorithm. Exported predicates:

rem_create(+Size, −REM)
creates an equivalence representation function REM which maps each of the nodes 1..Size to itself.

rem_head(−Node, +REM, −Head)
is true when Head is the representative of the equivalence class that Node belongs to in the given REM.

rem_equivalent(−Node1, −Node2, +REM)
is true when Node1 and Node2 belong to the same equivalence class in the given REM.
rem_add_link(?Node1, ?Node2, +OldREM, -NewREM)
is true when adding the equivalence Node1==Node2 to the partition represented by OldREM yields a partition which is represented by NewREM. If Node1 or Node2 is uninstantiated, it will backtrack over all the nodes. It’s not clear how useful this is.

10.39 Generic Sorting—library(samsort)
This library module provides generic sorting. Exported predicates:

samsort(+RawList, -Sorted)
takes a proper list RawList and unifies Sorted with a list having exactly the same elements as RawList but in ascending order according to the standard order on terms.

merge(+List1, +List2, -Merged)
is true when Merged is the stable merge of the two given lists. If the two lists are not ordered, the merge doesn't mean a great deal. Merging is perfectly well defined when the inputs contain duplicates, and all copies of an element are preserved in the output, e.g. merge("122357", "34568", "12233455678").

samsort(:Order, +RawList, -SortedList)
takes a proper list RawList and a binary predicate Order and unifies SortedList with a list having exactly the same elements as RawList but in ascending order according to Order. This is only supposed to work when Order is transitive.

merge(:Order, +List1, +List2, -Merged)
is like merge/3 except that it takes an Order predicate as its first arguments, like all the generalized ordering routines.

samkeysort(+RawList, -Sorted)
takes a proper list RawList of Key-Value pairs, and unifies Sorted with a list having exactly the same elements as RawList but in ascending order according to the standard order on the keys. samkeysort/2 is stable in the sense that the relative position of elements with the same key is maintained.

keymerge(+List1, +List2, -Merged)
is like merge/3 except that it compares only the keys of its input lists.

10.40 Unordered Set Operations—library(sets)
This library module provides operations on sets represented as unordered lists with no repeated elements.

Please note: You should probably not use this module. The ordered representation used in library(ordsets) is much more efficient, but these routines were designed before sort/2 entered the language. Exported predicates:
add_element(+Element, +Set1, -Set2)
is true when Set1 and Set2 are sets represented as unordered lists, and Set2 = Set1 U \{Element\}. It may only be used to calculate Set2 given Element and Set1.

del_element(+Element, +Set1, -Set2)
is true when Set1 and Set2 are sets represented as unordered lists, and Set2 = Set1 \ \{Element\}. It may only be used to calculate Set2 given Element and Set1. If Set1 does not contain Element, Set2 will be identical to Set1 (the old version made a new copy of Set1). If Set1 is not an unordered set, but contains more than one copy of Element, only the first will be removed. If you want to delete all copies of a given element, use lists:delete/3. For a version which fails if Element is not in Set1, use selectchk/3.

disjoint(+Set1, +Set2)
is true when the two given sets have no elements in common. It is the opposite of intersect/2. If either of the arguments is improper, disjoint/2 will fail.

is_set(+List)
is true when List is a proper list that contains no repeated elements.

pairfrom(?Set, ?Element1, ?Element2, ?Residue)
is true when Set is a list, Element1 occurs in list, Element2 occurs in list after Element1, and Residue is everything in Set bar the two Elements. The point of this thing is to select pairs of elements from a set without selecting the same pair twice in different orders.

intersect(+Set1, +Set2)
is true when the two sets have a member in common. It assumes that both sets are known, and that you don’t care which element it is that they share.

subset(+Set1, +Set2)
is true when each member of Set1 occurs in Set2. It can only be used to test two given sets; it cannot be used to generate subsets. There is no predicate for generating subsets as such, but the predicates subseq/3, subseq0/2, subseq1/2 in library(lists) may do what you want (they preserve the order of elements within a list). Could be defined as:

subset(Set1, Set2) :-
(    foreach(X,Set1),
    param(Set2)
    do    memberchk(X,Set2)
  ).

set_order(+Xs, +Ys, -R)
is true when R is <, =, or > according as Xs is a subset of Ys, equivalent to Ys, or a superset of Ys.

seteq(+Set1, +Set2)
is true when each Set is a subset of the other.
list_to_set(+List, -Set)
is true when List and Set are lists, and Set has the same elements as List in the same order, except that it contains no duplicates. The two are thus equal considered as sets.

power_set(+Set, -PowerSet)
is true when Set is a list and PowerSet is a list of lists which represents the power set of the set that Set represents.

intersection(+Set1, +Set2, -Intersection)
is true when all three arguments are lists representing sets, and Intersection contains every element of Set1 which is also an element of Set2, the order of elements in Intersection being the same as in Set1. That is, Intersection represents the intersection of the sets represented by Set1 and Set2. Could be defined as:

intersection(Set1, Set2, Intersection) :-
( foreach(X,Set1),
fromto(Intersection,S0,S,[[]]),
param(Set2)
do (member(X, Set2) -> S0 = [X|S] ; S0 = S)
).

intersection(+ListOfSets, -Intersection)
is true when Intersection is the intersection of all the sets inListOfSets. The order of elements in Intersection is taken from the first set in ListOfSets. This has been turned inside out to minimize the storage turnover. Could be defined as:

intersection([Set1|Sets], Intersection) :-
( foreach(X,Set1),
fromto(Intersection,S0,S,[[]]),
param(Sets)
do ( ( foreach(Set,Sets),
param(X)
do memberchk(X, Set)
) -> S0 = [X|S]
; S0 = S
)
).

subtract(+Set1, +Set2, -Difference)
is like intersect/3, but this time it is the elements of Set1 which are in Set2 that are deleted. Note that duplicated Elements of Set1 which are not in Set2 are retained in Difference. Could be defined as:

subtract(Set1, Set2, Difference) :-
( foreach(X,Set1),
fromto(Difference,S0,S,[[]]),
param(Set2)
do (member(X, Set2) -> S0 = S ; S0 = [X|S])
).
symdiff(+Set1, +Set2, -Difference)
is true when Difference is the symmetric difference of Set1 and Set2, that is, if each element of Difference occurs in one of Set1 and Set2 but not both. The construction method is such that the answer will have no duplicates even if the Sets do.

setproduct(+Set1, +Set2, -CartesianProduct)
is true when Set1 is a set (list) and Set2 is a set (list) and CartesianProduct is a set of Elt1-Elt2 pairs, with a pair for each element Elt1 of Set1 and Elt2 of Set2. Could be defined as:

```
setproduct(Set1, Set2, Product) :-
  ( foreach(H1,Set1),
    param(Set2),
    fromto(Product,P1,P3,[])
  do ( foreach(H2,Set2),
      param(H1),
      fromto(P1,[H1-H2|P2],P2,P3)
      do true
    )
  ).
```

disjoint_union(+Set1, +Set2, -Union)
is true when disjoint(Set1, Set2) and union(Set1, Set2, Union), that is, Set1 and Set2 have no element in common and Union is their union. Could be defined as:

```
disjoint_union(Set1, Set2, Union) :-
  ( foreach(X,Set1),
    fromto(Union,[X|S],S,Set2),
    param(Set2)
  do nonmember(X, Set2)
  ).
```

union(+Set1, +Set2, -Difference)
is true when union(Set1, Set2, Union) and subtract(Set1, Set2, Difference). Could be defined as:
union(Set1, Set2, Union, Difference) :-
( foreach(X,Set1),
  fromto(Union,S0,S,Set2),
  fromto(Difference,T0,T,[[]],)
  param(Set2)
do ( member(X, Set2) -> S0 = S, T0 = T
    ;   S0 = [X|S], T0 = [X|T]
  )
).

union(+ListOfSets, -Union)
is true when Union is the union of all the sets in ListOfSets. It has been arranged with storage turnover in mind. Could be defined as:

union(Sets, Union) :-
( foreach(Set,Sets),
  param(Answer)
do ( foreach(X,Set),
    param(Answer)
do memberchk(X, Answer)
  ),
append(Answer, [], Answer), % cauterize it !,
Union = Answer.

10.41 Socket I/O—library(sockets)

This library package defines a number of predicates for communicating over sockets.

To create a (bi-directional) stream connected to a remote server, use socket_client_open/3.

To open a port for remote clients to connect to, use socket_server_open/[2,3] and to open a stream to a connecting client, use socket_server_accept/4.

To be able to multiplex input and output from several streams (not necessarily socket streams) and incoming connections, use socket_select/7.

When opening a client or server socket a socket address needs to be specified. The address specifies the address family and family-specific information. The following formats are supported for socket addresses:

inet(Nodename,Servname)

Nodename: Servname

This specifies the address for an ordinary internet socket (AF_INET or AF_INET6). Nodename is the internet address of the remote host, as an atom, something like 'sicstus.sics.se' or '193.10.64.51'. The empty nodename ' ' (the default), has special meaning, see the documentation for socket_client_open/3 and socket_server_open/[2,3]. Servname is either a port
number as an atom of decimal digits or as an integer, e.g. '80', or 80; alternatively some well known port names can be used, e.g. 'http'. The set of well known port names is OS specific, portable code should use integer port numbers. Servname can also be a variable when opening a server socket with socket_server_open/[2,3]. In this case a available port is assigned automatically and Servname is bound to it.

unix(Path) \[
\text{since release 4.0.3} \]

A Unix domain (AF_UNIX) socket is opened at the specified file system location. This is only supported on Unix-like platforms. Path is a file-name and is passed to absolute_file_name/2. There may be platform-specific restrictions on the length of the resulting pathname and the file system containing it.

All streams below can be read from as well as written to. All I/O predicates operating on streams can be used, for example get_code/2, get_byte/2, read/2, write/2, format/3, current_stream/3, etc. The predicates that create streams take options similar to open/4, e.g. to specify whether the stream is binary (the default) or text.

socket_client_open(+Addr, -Stream, +Options) \[
\text{since release 4.0.3} \]

Creates a stream Stream connected to address Addr. See above for the allowed address formats. If the nodename is empty ('') then a connection is made to the local machine.

The stream is created using options from Options. Supported options include:

type(binary)
Create a binary stream (the default).

type(text)
Create a text stream. The default encoding is Latin 1.

eof_action(Action)
end of file action, as for open/4.

encoding(ENCODING) \[
\text{since release 4.1} \]
As for open/4. Implies type(text).

eol(Eol) \[
\text{since release 4.1} \]
As for open/4. Implies type(text).

To create a binary stream to some web server sicstus.sics.se, you would do e.g.

| ?- socket_client_open('sicstus.sics.se':80, Stream, [type(binary)]).

or, to make a text (Latin 1) stream to a daytime service in Hong Kong you could do:

| ?- socket_client_open('stdtime.gov.hk':daytime, S, [type(text)]),
  read_line(S, L),
  format('~s', [L]).

See the source code for library('linda/client') for a simple client.
socket_server_open(?Addr, -ServerSocket, +Options)  since release 4.0.3
Create a server socket ServerSocket that listens on address Addr. See above for the allowed address formats. If the nodename is empty ("'") then any remote client machine is allowed to connect unless the option loopback(true) is also specified. Addr can specify an internet address where the port is a variable in which case a free port number is used and Port is bound to it. The common case is that Addr is a numeric port number or a variable that becomes bound to a free port number.

The created server socket should be closed with socket_server_close/1 eventually. Incoming connection can be accepted with socket_server_accept/4 and waited for with socket_select/7. See the source code for library('linda/server') for a simple server that uses this predicate.

Options is a list of options, currently

reuseaddr(Bool)  since release 4.0.3
Bool is either true or false (the default). If true then allow reuse of local addresses. For internet sockets this corresponds to the SO_REUSEADDR socket option. For unix domain sockets this means that the file will be deleted, if present, before opening.

numeric_nodename(Bool)  since release 4.0.3
Bool is either true or false (the default). If true then the nodename of an internet address will be treated as a numerical address and no name lookup will be performed.

numeric_servname(Bool)  since release 4.0.3
Bool is either true or false (the default). If true then the servname of an internet address will be treated as a numerical port number and no lookup of well known port names will be performed.

loopback(Bool)  since release 4.0.3
Bool is either true or false (the default). If true then the nodename will be ignored and the socket will only listen to connection from the loopback device, i.e. the local machine.

socket_server_open(?Port, -ServerSocket)
The same as socket_server_open(Port, ServerSocket, []).

socket_server_accept(+ServerSocket, -Client, -Stream, +StreamOptions)
The first connection to socket ServerSocket is extracted, blocking if necessary. The stream Stream is created on this connection using StreamOptions as for socket_client_open/3. Client will be unified with an atom containing the numerical Internet host address of the connecting client. Note that the stream will be type(binary) unless type(text) is specified either explicitly or implicitly with encoding/1 or other text-only options.

socket_server_close(+ServerSocket)
Close the server socket ServerSocket and stop listening on its port.
socket_select(+ServerSockets, -SReady, +ReadStream, -RReady, +WriteStream, -WReady, +Timeout)

Check for server sockets with incoming connections (i.e. ready for socket_server_accept/4), streams on ReadStream ready for input, and streams on WriteStream ready for output. The streams can be any kind of streams, they need not be socket streams. The ready server sockets are returned (in the same order) in SReady, the ready input streams in RReady, and the ready output streams in WReady.

An input (output) stream is ready for input (output) when an item can be read (written) without blocking. An item is a character for text streams and a byte for binary streams. Note that a stream is considered ready for I/O if the corresponding I/O operation will raise an error (such as if the stream is past end of stream).

Each entry in the input lists ServerSockets, ReadStream, and WriteStream can be either a server socket or stream respectively or a term Term-Entry where Entry is the server socket or stream and Term is some arbitrary term used for book-keeping. If an entry is associated with a term in this way then so will the corresponding ready entry.

If Timeout is instantiated to off, the predicate waits until something is available. If Timeout is a nonzero number (integer or floating point), then the predicate waits at most that number of seconds before returning. For backward compatibility, if Timeout is S:U the predicate waits at most S seconds and U microseconds. If there is a timeout, all ready lists are unified with [].

See the source code for library('linda/server') for a simple server that uses this predicate.

current_host(?HostName)

HostName is unified with the fully qualified name of the machine that the process is executing on. The call will also succeed if HostName is instantiated to the unqualified name of the machine in lower case. Please note: this predicate will fail if there are errors, e.g. if no domain has been configured.

10.42 Statistics Functions—library(statistics)

This library module provides commonly used sample and population statistics functions. In this module, a Sample is simply a proper list of numbers, normally floating-point; Weight is a proper list of numbers and should be of the same length as Sample.

Please note: These functions are plain textbook algorithms and we make no claims about numerical stability, avoiding loss of precision, etc.

Exported predicates:

\[
\text{min}(+\text{Sample}, -\text{Value})
\]

is true when Value is the smallest element of Sample.

\[
\text{max}(+\text{Sample}, -\text{Value})
\]

is true when Value is the largest element of Sample.
min_max(+Sample, -Min, -Max)
is true when Min (Max) is the smallest (largest) element of Sample.

range(+Sample, -Value)
is true when Value is the difference between the largest and smallest elements of Sample.

mode(+Sample, -Values)
is true when Values is the most frequently occurring value(s) in Sample. If there is a unique value with maximum frequency, this value is returned as the only element of Values. Otherwise, Values contains the maximum frequency elements in increasing order. This predicate does not make much sense if the sample is continuous.

mean(+Sample, -Value)
arithmetic_mean(+Sample, -Value)
is true when Value is the arithmetic mean of Sample.

weighted_mean(+Weight, +Sample, -Value)
is true when Value is the arithmetic mean of Sample weighted by Weight.

gеometric_mean(+Sample, -Value)
is true when Value is the geometric mean of Sample.

harmonic_mean(+Sample, -Value)
is true when Value is the harmonic mean of Sample.

central_moment(K, +Sample, -Value)
is true when Value is the K-th central moment of Sample. Also known as the K-th central moment about the mean. K should be positive integer.

skewness(+Sample, -Value)
is true when Value is the skewness of Sample. This is a measure of the asymmetry of its distribution. A sample with negative skew is said to be left-skewed. Most of its mass is on the right of the distribution, with the tail on the left. Vice versa for positive skew. A sample’s skewness is undefined if its variance is zero.

kurtosis(+Sample, -Value)
is true when Value is the excess kurtosis of Sample. This is a measure of the peakedness of its distribution. A high kurtosis indicates that most of the sample’s variance is due to infrequent severe deviations, rather than frequent modest deviations. A sample’s excess kurtosis is undefined if its variance is zero. In this implementation, the kurtosis of the normal distribution is 0.

ml_variance(+Sample, -Value)
population_variance(+Sample, -Value)
is true when Value is the maximum likelihood estimate of the variance of Sample. Also known as the population variance, where the denominator is the length of Sample.
sample_variance(+Sample, -Value)
unbiased_variance(+Sample, -Value)
is true when Value is the unbiased estimate of the variance of Sample. Also known as the sample variance, where the denominator is the length of Sample minus one.

weighted_variance(+Weight, +Sample, -Value)
is true when Value is the weighted (biased) estimate of the variance of Sample.

ml_standard_deviation(+Sample, -Value)
population_standard_deviation(+Sample, -Value)
is true when Value is the maximum likelihood estimate of the standard deviation of Sample. Also known as the population standard deviation, where the denominator is the length of Sample. Equals the square root of the population variance.

sample_standard_deviation(+Sample, -Value)
unbiased_standard_deviation(+Sample, -Value)
is true when Value is the unbiased estimate of the standard deviation of Sample. Also known as the sample standard deviation, where the denominator is the length of Sample minus one. Equals the square root of the sample variance.

weighted_standard_deviation(+Weight, +Sample, -Value)
is true when Value is the weighted (biased) estimate of the variance of Sample. Equals the square root of the weighted (biased) variance.

covariance(+Sample1, +Sample2, -Value)
correlation(+Sample1, +Sample2, -Value)
is true when Value is the covariance of Sample1 and Sample2.

median(+Sample, -Value)
is true when Value is the median of Sample, that is, the value separating the higher half of the sample from the lower half. If there are an even number of observations, then the median is defined to be the smaller middle value. Same as the 0.5-fractile of Sample.

fractile(P, +Sample, -Value)
is true when Value is the P-fractile of Sample, that is, the smallest value in the sample such that the fraction P of the sample is less than or equal to that value. P should be a number in (0.0,1.0].

normalize(+Sample, -Normalized)
is true when Normalized is the normalized Sample, so that Normalized has a mean of 0 and a population standard deviation of 1.

10.43 The Structs Package—library(structs)
The structs package allows Prolog to hold pointers to C data structures, and to access and store into fields in those data structures. Currently, the only representation for a pointer supported by SICStus Prolog is an integer, so it is not possible to guarantee that Prolog
cannot confuse a pointer with an ordinary Prolog term. What this package does is to
represent such a pointer as a term with the type of the structure or array as its functor and
the integer that is the address of the actual data as its only argument. We will refer such
terms as foreign terms.

The package consists of two modules, str_decl and structs. The str_decl module is
used at compile time to translate the structs-related constructs. Any file that defines or
accesses structs should include the command:

    :- load_files(library(str_decl),
                  [when(compile_time), if(changed)]).

The structs module provides runtime support for structs. A file that accesses structs
should include the command:

    :- use_module(library(structs)).

You will probably include both in most files that define and access structs.

**Please note:** A file that loads library(str_decl) currently cannot recursively load another
file that loads library(str_decl), because that would confuse the internal database being
used by the package.

**Important caveats:**

You should not count on future versions of the structs package to continue to
represent foreign terms as compound Prolog terms. In particular, you should
never explicitly take apart a foreign term using unification or functor/3 and
arg/3. You may use the predicate foreign_type/2 to find the type of a foreign
term, and cast/3 (casting a foreign term to address) to get the address part of
a foreign term. You may also use cast/3 to cast an address back to a foreign
term. You should use null_foreign_term/2 to check if a foreign term is null,
or to create a null foreign term of some type.

It should never be necessary to explicitly take apart foreign terms.

### 10.43.1 Foreign Types

There are two sorts of objects that Prolog may want to handle: *atomic* and *compound*. *Atomic* objects include numbers and atoms, and *compound* objects include data structures
and arrays. To be more precise about it, an atomic type is defined by one of the following:

- **integer** signed integer, large enough to hold a pointer.
- **integer_64** 64 bit signed integer. *since release 4.3*
- **integer_32** 32 bit signed integer.
- **integer_16** 16 bit signed integer.
integer_8  
8 bit signed integer.

unsigned  
unsigned integer, large enough to hold a pointer.

unsigned_64  
64 bit unsigned integer.  
*since release 4.3*

unsigned_32  
32 bit unsigned integer.

unsigned_16  
16 bit unsigned integer.

unsigned_8  
8 bit unsigned integer.

float  
64 bit floating-point number.

float_32  
32 bit floating-point number.

atom  
32 bit Prolog atom number. Unique for different atoms, but not consistent across Prolog sessions. The atom is made non garbage collectable. See Section 6.4.2 [Atoms in C], page 302.

string  
A pointer to an encoded string. Represented as an atom in Prolog. **Please note:** This string must not be overwritten, as it constitutes the print name of an atom. Also, the atom and string are made non garbage collectable. See Section 6.4.2 [Atoms in C], page 302.

address  
An untyped pointer. Like `pointer(_), but library(structs)` does no type checking for you. Represented as a Prolog integer.

opaque  
Unknown type. Cannot be represented in Prolog. A pointer to an opaque object may be manipulated.

Compound types are defined by one of the following:

`pointer(Type)`  
a pointer to a thing of type `Type`.

`array(Num, Type)`  
A chunk of memory holding `Num` (an integer) things of type `Type`.

`array(Type)`  
A chunk of memory holding some number of things of type `Type`. This type does not allow bounds checking, so it should be used with great care. It is also not possible to use this sort of array as an element in an array, or in a struct or union.

`struct(Fields)`  
A compound structure. `Fields` is a list of `Field_name:Type` pairs. Each `Field_name` is an atom, and each `Type` is any valid type.
union(Members)

A union as in C. Members is a list of Member_name:Type pairs. Each Member_name is an atom, and each Type is any valid type. The space allocated for one of these is the maximum of the spaces needed for each member. It is not permitted to store into a union (you must get a member of the union to store into, as in C).

C programmers will recognize that the kinds of data supported by this package were designed for the C language. They should also work for other languages, but programmers must determine the proper type declarations in those languages. The table above makes clear the storage requirements and interpretation of each type.

Note that there is one important difference between the structs package and C: the structs package permits declarations of pointers to arrays. A pointer to an array is distinguished from a pointer to a single element. For example

\[
\text{pointer(array(integer_8))}
\]

is probably a more appropriate declaration of a C string type than

\[
\text{pointer(integer_8)}
\]

which is the orthodox way to declare a string in C.

10.43.1.1 Declaring Types

Programmers may declare new named data structures with the following procedure:

\[
:- \text{foreign_type}
\begin{align*}
\text{Type_name} &= \text{Type}, \\
\ldots, \\
\text{Type_name} &= \text{Type}.
\end{align*}
\]

where Type_name is an atom, and Type defines either an atomic or compound type, or is a previously-defined type name.

In Prolog, atomic types are represented by the natural atomic term (integer, float, or atom). Compound structures are represented by terms whose functor is the name of the type, and whose only argument is the address of the data. So a term foo(123456) represents the thing of type foo that exists at machine address 123456. And a term integer(123456) represents the integer that lives in memory at address 123456, not the number 123456.

For types that are not named, a type name is generated using the names of associated types and the dollar sign character (‘$’), and possibly a number. Therefore, users should not use ‘$’ in their type names.

10.43.2 Checking Foreign Term Types

The type of a foreign term may determined by the goal

\[
\text{foreign_type}(\text{+Foreign_term}, -\text{Type_name})
\]
Note that `foreign_type/2` will fail if `Foreign_term` is not a foreign term.

### 10.43.3 Creating and Destroying Foreign Terms

Prolog can create or destroy foreign terms using

```prolog
new(+Type, -Datum),
new(+Type, +Size, -Datum) and
dispose(+Datum)
```

where `Type` is an atom specifying what type of foreign term is to be allocated, and `Datum` is the foreign term. `Type` should be an atomic type or a previously-defined type name. The `Datum` returned by `new/[2,3]` is initialized to all zeroes. `dispose/1` is a dangerous operation, since once the memory is disposed, it may be used for something else later. If `Datum` is later accessed, then the results will be unpredictable. `new/3` is only used to allocate arrays whose size is not known beforehand, as defined by `array(Type)`, rather than `array(Num,Type)`.

### 10.43.4 Accessing and Modifying Foreign Term Contents

Prolog can get or modify the contents of a foreign term with the procedures

```prolog
get_contents(+Datum, ?Part, ?Value)
put_contents(+Datum, +Part, +Value).
```

It can also get a pointer to a field or element of a foreign term with the procedure

```prolog
get_address(+Datum, ?Part, ?Value).
```

For all three of these, `Datum` must be a foreign term, and `Part` specifies what part of `Datum Value` is. If `Datum` is an array, then `Part` should be an integer index into the array, where 0 is the first element. For a pointer, `Part` should be the atom `contents` and `Value` will be what the pointer points to. For a struct, `Part` should be a field name, and `Value` will be the contents of that field. In the case of `get_contents/3` and `get_address/3`, if `Part` is unbound, then `get_contents/3` will backtrack through all the valid parts of `Datum`, binding both `Part` and `Value`. A C programmer might think of the following pairs as corresponding to each other:

```prolog
Prolog: get_contents(Foo, Bar, Baz)
        C: Baz = Foo->Bar

Prolog: put_contents(Foo, Bar, Baz)
        C: Foo->Bar = Baz

Prolog: get_address(Foo, Bar, Baz)
        C: Baz = &Foo->Bar.
```

The hitch is that only atomic and pointer types can be got and put by `get_contents/3` and `put_contents/3`. This is because Prolog can only hold pointers to C structures, not the structures themselves. This is not quite as bad as it might seem, though, since usually
structures contain pointers to other structures, anyway. When a structure directly contains
another structure, Prolog can get a pointer to it with \texttt{get_address/3}.

10.43.5 Casting

Prolog can “cast” one type of foreign term to another. This means that the foreign term is
treated just as if it where the other type. This is done with the following procedure:

\begin{verbatim}
  cast(+Foreign0, +New_type, -Foreign)
\end{verbatim}

where \texttt{Foreign} is the foreign term that is the same data as \texttt{Foreign0}, only is of foreign type
\texttt{New_type}. \texttt{Foreign0} is not affected. This is much like casting in C.

Casting a foreign term to \texttt{address} will get you the raw address of a foreign term. This is
not often necessary, but it is occasionally useful in order to obtain an indexable value to use
in the first argument of a dynamic predicate you are maintaining. An \texttt{address} may also be
cast to a proper foreign type.

This predicate should be used with great care, as it is quite easy to get into trouble with
this.

10.43.6 Null Foreign Terms

“NULL” foreign terms may be handled. The predicate

\begin{verbatim}
  null_foreign_term(+Term, -Type)
  null_foreign_term(-Term, +Type)
\end{verbatim}

holds when \texttt{Term} is a foreign term of \texttt{Type}, but is NULL (the address is 0). At least one of
\texttt{Term} and \texttt{Type} must be bound. This can be used to generate NULL foreign terms, or to
check a foreign term to determine whether or not it is NULL.

10.43.7 Interfacing with Foreign Code

Foreign terms may be passed between Prolog and other languages through the foreign
interface.

To use this, all foreign types to be passed between Prolog and another language must be
declared with \texttt{foreign_type/2} before the \texttt{foreign/}[2,3] clauses specifying the foreign
functions.

The \texttt{structs} package extends the foreign type specifications recognized by the foreign
interface. In addition to the types already recognized by the foreign interface, any atomic
type recognized by the \texttt{structs} package is understood, as well as a pointer to any named
\texttt{structs} type.

For example, if you have a function

\begin{verbatim}
char nth_char(char *string, int n)
{
    return string[n];
}
\end{verbatim}
then you might use it from Prolog as follows:

    :- foreign_type cstring = array(integer_8).

    foreign(nth_char, c, nth_char(+pointer(cstring), +integer, [-integer_8])).

This allows the predicate \texttt{nth_char/3} to be called from Prolog to determine the \texttt{nth} character of a C string.

Note that all existing foreign interface type specifications are unaffected, in particular \texttt{address/[0,1]} continue to pass addresses to and from Prolog as plain integers.

If you use the foreign resource linker, \texttt{splfr}, on a Prolog file that uses the \texttt{structs} package, then you must pass it the \texttt{--structs} option. This will make \texttt{splfr} understand foreign type specifications and translate them into C declarations in the generated header file (see Section 6.2.5 [The Foreign Resource Linker], page 296).

\subsection*{10.43.8 Examining Type Definitions at Run Time}

The above described procedures should be sufficient for most needs. This module does, however, provide a few procedures to allow programmers to access type definitions. These may be a convenience for debugging, or in writing tools to manipulate type definitions.

The following procedures allow programmers to find the definition of a given type:

    type_definition(?Type, ?Definition)
    type_definition(?Type, ?Definition, ?Size)

where \texttt{Type} is an atom naming a type, \texttt{Definition} is the definition of that type, and \texttt{Size} is the number of bytes occupied by a foreign term of this type. \texttt{Size} will be the atom \texttt{unknown} if the size of an object of that type is not known. Such types may not be used as fields in structs or unions, or in arrays. However, pointers to them may be created. If \texttt{Type} is not bound at call time, then these procedures will backtrack through all current type definitions.

A definition looks much like the definition given when the type was defined with \texttt{type/1}, except that it has been simplified. Firstly, intermediate type names have been elided. For example, if \texttt{foo} is defined as \texttt{foo=integer}, and \texttt{bar} as \texttt{bar=foo}, then \texttt{type_definition(bar, integer)} would hold. Also, in the definition of a compound type, types of parts are always defined by type names, rather than complex specifications. So if the type of a field in a struct was defined as \texttt{pointer(fred)}, then it will show up in the definition as \texttt{'$fred'}. Of course, \texttt{type_definition('$fred', pointer(fred))} would hold, also.

The following predicates allow the programmer to determine whether or not a given type is atomic:


atomic_type(?Type)

atomic_type(?Type, ?Primitive_type)

atomic_type(?Type, ?Primitive_type, ?Size)

where Type is an atomic type. See Section 10.43.1 [str-fty], page 760, for the definition of an atomic type. Primitive_type is the primitive type that Type is defined in terms of. Size is the number of bytes occupied by an object of type Type, or the atom unknown, as above. If Type is unbound at call time, then these predicates will backtrack through all the currently defined atomic types.

10.43.9 Tips

1. Most important tip: do not subvert the structs type system by looking inside foreign terms to get the address, or use functor/3 to get the type. This has two negative effects: firstly, if the structs package should change its representation of foreign terms, then your code will not work. But more importantly, you are more likely to get type mismatches, and likely to get unwrapped terms or even doubly wrapped terms where you expect wrapped ones.

2. Remember that a foreign term fred(123456) is not of type fred, but a pointer to fred. Looked at another way, what resides in memory at address 123456 is of type fred.

3. The wrapper put on a foreign term signifies the type of that foreign term. If you declare a type to be pointer(opaque) because you want to view that pointer to be opaque, when you get something of this type, then it will be printed as opaque(456123). This is not very informative. It is better to declare

fred = opaque,
thing = struct([...
   part:pointer(fred),
   ...
  ]).

so that when you get the contents of the part member of a thing, it is wrapped as fred(456123).

10.43.10 Example

The following example shows how to use library(structs) in a simple package for handling integer arrays. We define a module minivec with exported predicates for creating and disposing arrays, accessing its elements, and computing their sum. The summing operation is implemented in C and the rest in Prolog. Arrays are created using the array(Type) foreign type.

Note that the type declaration int32 does not have to be given in the C source code, as it appears in the automatically generated header file minivec_glue.h. Note also how the foreign type specification +pointer(int_array) corresponds to the C type declaration int32 *.
% minivec.pl

:- module(minivec, [ new_array/2, get_array/3, put_array/3, dispose_array/1, sum_array/2 ]).

:- load_files(library(str_decl), [when(compile_time)]).
:- use_module(library(structs)).

:- foreign_type int32 = integer_32,
     int_array = array(int32).

foreign(c_sum_array, c_sum_array(+integer,
   +pointer(int_array),
   [-integer])).

foreign_resource(minivec, [c_sum_array]).

:- load_foreign_resource(minivec).

new_array(Size, array(Size,Mem)) :-
    new(int_array, Size, Mem).

get_array(Index, array(_,Mem), Value) :-
    get_contents(Mem, Index, Value).

put_array(Index, array(_,Mem), Value) :-
    put_contents(Mem, Index, Value).

dispose_array(array(_,Mem)) :-
    dispose(Mem).

sum_array(array(Size,Mem), Sum) :-
    c_sum_array(Size, Mem, Sum).
/* minivec.c */

#include "minivec_glue.h"

SP_integer c_sum_array(SP_integer cnt, int32 *mem)
{
    int i;
    SP_integer sum = 0;

    for (i=0; i<cnt; i++)
        sum += mem[i];
    return sum;
}

A fragment from the generated header file:

/* minivec_glue.h */
#include <sicstus/sicstus.h>
#include <stdlib.h>
typedef int int32;
typedef int32 *(int_array)/* really an unknown-size array */;
extern SP_integer c_sum_array( SP_integer, int32 *);

10.44 Operating System Utilities—library(system)

This package contains utilities for invoking services from the operating system that do not fit elsewhere. This package contains utilities for invoking services from the operating system that does not fit elsewhere.

Exported predicates:
Chapter 10: The Prolog Library

now(-When)
Converts the current date and time as a UNIX timestamp with When.

datetime(-Datime)
Converts Datime with the current date and time as a datime/6 record of the form \textit{datime(Year, Month, Day, Hour, Min, Sec)}. All fields are integers.

datetime(+When, -Datime)
datetime(-When, +Datime)
Converts a time stamp, as obtained by \textit{now/1}, to a \textit{datime/6} record. Can be used in both directions.

sleep(+Seconds)
Puts the SICStus Prolog process asleep for Seconds seconds, where Seconds should be a non-negative number.

environ(?Var, ?Value)
Var is the name of a system property or an environment variable, and Value is its value. Both are atoms. Can be used to enumerate all current system properties and environment variables.
The same as \textit{environ(Var, Value, merged)}.

environ(?Var, ?Value, +Source)
Var is the name of an environment variable or system property, and Value is its value. Both are atoms. Can be used to enumerate all current environment variables and system properties.
Source is one of properties, in which case only system properties are enumerated; environment, in which case only environment variables are enumerated; and merged, in which case both environment variables and system properties are enumerated. When Source is \textit{merged} and an environment variable and a system property have equivalent names, the value of the system property is returned.
On UNIX-like platforms, two names are equivalent if and only if they are identical. On Windows-like platforms, a case insensitive comparison is used.
See Section 4.17.1 [System Properties and Environment Variables], page 224, for more information.

10.45 Tcl/Tk Interface—\texttt{library(tcltk)}

10.45.1 Introduction
This is a basic tutorial for those SICStus Prolog users who would like to add Tcl/Tk user interfaces to their Prolog applications. The tutorial assumes no prior knowledge of Tcl/Tk but, of course, does assume the reader is proficient in Prolog.

Aware that the reader may not have heard of Tcl/Tk, we will start by answering three questions: what is Tcl/Tk? what is it good for? what relationship does it have to Prolog?

10.45.1.1 What Is Tcl/Tk?
Tcl/Tk, as its title suggests, is actually two software packages: Tcl and Tk. Tcl, pronounced \textit{tickle}, stands for \textit{tool command language} and is a scripting language that provides a pro-
gramming environment and programming facilities such as variables, loops, and procedures. It is designed to be easily extensible.

Tk, pronounced *tee-kay*, is just such an extension to Tcl, which is a *toolkit* for windowing systems. In other words, Tk adds facilities to Tcl for creating and manipulating user interfaces based on windows and widgets within those windows.

**10.45.1.2 What Is Tcl/Tk Good For?**

In combination the Tcl and Tk packages (we will call the combination simply Tcl/Tk) are useful for creating graphical user interfaces (GUIs) to applications. The GUI is described in terms of instances of Tk widgets, created through calls in Tcl, and Tcl scripts that form the glue that binds together the GUI and the application. (If you are a little lost at this point, then all will be clear in a moment with a simple example.)

There are lots of systems out there for adding GUIs to applications so why choose Tcl/Tk? Tcl/Tk has several advantages that make it attractive for this kind of work. Firstly, it is good for rapid prototyping of GUIs. Tcl is an interpreted scripting language. The scripts can be modified and executed quickly, with no compilation phase, so speeding up the development loop.

Secondly, it is easier to use a system based on a scripting language, such as Tcl/Tk, than many of the conventional packages available. For example, getting to grips with the X windows suite of C libraries is not an easy task. Tcl/Tk can produce the same thing using simple scripting with much less to learn. The penalty for this is that programs written in an interpreted scripting language will execute more slowly than those written using compiled C library calls, but for many interfaces that do not need great speed Tcl/Tk is fast enough and its ease of use more than outweighs the loss of speed. In any case, Tcl/Tk can easily handle hundreds of events per mouse movement without the user noticing.

Thirdly, Tcl/Tk is good for making cross-platform GUIs. The Tk toolkit has been ported to native look-and-feel widgets on Mac, PC (Windows), and UNIX (X windows) platforms. You can write your scripts once and they will execute on any of these platforms.

Lastly, the software is distributed under a free software license and so is available in both binary and source formats free of charge.

**10.45.1.3 What Is Tcl/Tks Relationship to SICStus Prolog?**

SICStus Prolog comes with a Prolog library for interfacing to Tcl/Tk. The purpose of the library is to enable Prolog application developers to add GUIs to their applications rapidly and easily.

**10.45.1.4 A Quick Example of Tcl/Tk in Action**

As a taster, we will show you two simple examples programs that use SICStus Prolog with the Tcl/Tk extensions: the ubiquitous “hello world” example; and a very simple telephone book look up example.

You are not expected to understand how these examples work at this stage. They are something for you to quickly type in to see how easy it is to add GUIs to Prolog programs.
through Tcl/Tk. After reading through the rest of this tutorial you will fully understand these examples and be able to write your own GUIs.

Here is the “Hello World” program; also in library('tcltk/examples/ex1.pl'):

```prolog
:- use_module(library(tcltk)).

go :-
  tk_new([name('Example 1')], Interp),
  tcl_eval(Interp, 'button .fred -text "hello world"
            -command { puts "hello world"}', _),
  tcl_eval(Interp, 'pack .fred', _),
  tk_main_loop.
```

SICStus+Tcl/Tk hello world program.

To run it just start up SICStus (under Windows use sicstus, not spwin), load the program, and evaluate the Prolog goal go. The first line of the go clause calls tk_new/2, which creates a Tcl/Tk interpreter and returns a handle Interp through which Prolog will interact with the interpreter. Next a call to tcl_eval/3 is made, which creates a button displaying the 'hello world' text. Next a call is made to tcl_eval/3 that causes the button to be displayed in the main application window. Finally, a call is make to tk_main_loop/0 that passes control to Tcl/Tk, making sure that window events are serviced.

See how simple it is with just a three line Prolog program to create an application window and display a button in it. Click on the button and see what it does.

The reason you should use sicstus under Windows instead of spwin is that the latter does not have the C standard streams (stdin, stdout, stderr) and the Tcl command puts will give an error if there is no stdout.

The previous example showed us how to create a button and display some text in it. It was basically pure Tcl/Tk generated from within Prolog but did not have any interaction with Prolog. The following example demonstrates a simple callback mechanism. A name is typed into a text entry box, a button is pressed, which looks up the telephone number corresponding to the name in a Prolog database, and the telephone number is then displayed.

Here is the code; also in library('tcltk/examples/ex2.pl'):
:- use_module(library(tcltk)).

telephone(fred, '123-456').
telescope(wilbert, '222-2222').
telescope(taxi, '200-0000').
telescope(mary, '00-36-1-666-6666').

go :-
   tk_new([name('Example 2')], T),
   tcl_eval(T, 'entry .name -textvariable name', _),
   tcl_eval(T, 'button .search -text search -command {
       prolog telephone($name,X);
       set result $prolog_variables(X) }', _),
   tcl_eval(T, 'label .result -relief raised -textvariable result', _),
   tcl_eval(T, 'pack .name .search .result -side top -fill x', _),
   tk_main_loop.

Again, to run the example, start up SICStus Prolog, load the code, and run the goal `go`.

You will notice that three widgets will appear in a window: one is for entering the name of the person or thing that you want to find the telephone number for, the button is for initiating the search, and the text box at the bottom is for displaying the result.

Type `fred` into the entry box, hit the search button and you should see the phone number displayed. You can then try the same thing but with `wilbert`, `taxi` or `mary` typed into the text entry box.

What is happening is that when the button is pressed, the value in the entry box is retrieved, then the `telephone/2` predicate is called in Prolog with the entry box value as first argument, then the second argument of telephone is retrieved (by this time bound to the number) and is displayed below the button.

This is a very crude example of what can be done with the Tcl/Tk module in Prolog. For example, this program does not handle cases where there is no corresponding phone number or where there is more than one corresponding phone number. The example is just supposed to wet your appetite, but all these problems can be handled by Prolog +
Tcl/Tk, although with a more sophisticated program. You will learn how to do this in the subsequent chapters.

10.45.1.5 Outline of This Tutorial

Now we have motivated using Tcl/Tk as a means of creating GUIs for Prolog programs, this document goes into the details of using Tcl/Tk as a means of building GUIs for SICStus Prolog applications.

Firstly, Tcl is introduced and its syntax and core commands described. Then the Tk extensions to Tcl are introduced. We show how with Tcl and Tk together the user can build sophisticated GUIs easily and quickly. At the end of this Tcl/Tk part of the tutorial an example of a pure Tcl/Tk program will be presented together with some tips on how to design and code Tcl/Tk GUIs.

The second phase of this document describes the SICStus Prolog tcltk library. It provides extensions to Prolog that allow Prolog applications to interact with Tcl/Tk: Prolog can make calls to Tcl/Tk code and vice versa.

Having reached this point in the tutorial the user will know how to write a Tcl/Tk GUI interface and how to get a Prolog program to interact with it, but arranging which process (the Prolog process or the Tcl/Tk process) is the dominant partner is non-trivial and so is described in a separate chapter on event handling. This will help the user choose the most appropriate method of cooperation between Tcl/Tk and Prolog to suit their particular application.

This section, the Tcl/Tk+Prolog section, will be rounded off with the presentation of some example applications that make use of Tcl/Tk and Prolog.

Then there is a short discussion section on how to use other Tcl extension packages with Tcl/Tk and Prolog. Many such extension packages have been written and when added to Prolog enhanced with Tcl/Tk can offer further functionality to a Prolog application.

The appendices provide a full listing with description of the predicates available in the tcltk SICStus Prolog library, and the extensions made to Tcl/Tk for interacting with Prolog.

Lastly, a section on resources gives pointers to where the reader can find more information on Tcl/Tk.

10.45.2 Tcl

Tcl is an interpreted scripting language. In this chapter, first the syntax of Tcl is described and then the core commands are described. It is not intended to give a comprehensive description of the Tcl language here but an overview of the core commands, enough to get the user motivated to start writing their own scripts.

For pointers to more information on Tcl; see Section 10.45.7 [Resources], page 861.
10.45.2.1 Syntax

A Tcl script consists of a series of strings separated from each other by a newline character. Each string contains a command or series of semi-colon separated commands. A command is a series of words separated by spaces. The first word in a command is the name of the command and subsequent words are its arguments.

An example is:

```
set a 1
set b 2
```

which is a Tcl script of two commands: the first command sets the value of variable `a` to 1, and the second command sets the value of variable `b` to 2.

An example of two commands on the same line separated by a semi-colon is:

```
set a 1; set b 2
```

which is equivalent to the previous example but written entirely on one line.

A command is executed in two phases. In the first phase, the command is broken down into its constituent words and various textual substitutions are performed on those words. In the second phase, the procedure to call is identified from the first word in the command, and the procedure is called with the remaining words as arguments.

There are special syntactic characters that control how the first phase, the substitution phase, is carried out. The three major substitution types are variable substitution, command substitution, and backslash substitution.

Variable substitution happens when a ‘$’ prefixed word is found in a command. There are three types of variable substitution:

- `$name`
  - where `name` is a scalar variable. `name` is simply substituted in the word for its value. `name` can contain only letters, digits, or underscores.

- `$name(index)`
  - where `name` is the name of an array variable and `index` is the index into it. This is substituted by the value of the array element. `name` must contain only letters, digits, or underscores. `index` has variable, command, and backslash substitution performed on it too.

- `{$name}`
  - where `name` can have any characters in it except closing curly bracket. This is more or less the same as `$name` substitution except it is used to get around the restrictions in the characters that can form `name`.

An example of variable substitution is:
set a 1
set b $a

which sets the value of variable a to 1, and then sets the value of variable b to the value of variable a.

Command substitution happens when a word contains an open square bracket, `[ ]`. The string between the open bracket and matching closing bracket are treated as a Tcl script. The script is evaluated and its result is substituted in place of the original command substitution word.

A simple example of command substitution is:

```
set a 1
set b [set a]
```

which does the same as the previous example but using command substitution. The result of a `set a` command is to return the value of a, which is then passed as an argument to `set b` and so variable b acquires the value of variable a.

Backslash substitution is performed whenever the interpreter comes across a backslash. The backslash is an escape character and when it is encountered is causes the interpreter to handle the next characters specially. Commonly escaped characters are `\a` for audible bell, `\b` for backspace, `\f` for form feed, `\n` for newline, `\r` for carriage return, `\t` for horizontal tab, and `\v` for vertical tab. Double-backslash, `\\`, is substituted with a single backslash. Other special backslash substitutions have the following forms:

- \ooo
  - the digits ooo give the octal value of the escaped character
- \xHH
  - the x denotes that the following hexadecimal digits are the value of the escaped character

Any other character that is backslash escaped is simply substituted by the character itself. For example, \W is replaced by W.

A further syntactic construction is used to delay substitution. When the beginning of a word starts with a curly bracket, `{`, it does not do any of the above substitutions between the opening curly bracket and its matching closing curly bracket. The word ends with the matching closing curly bracket. This construct is used to make the bodies of procedures in which substitutions happen when the procedure is called, not when it is constructed. Or it is used anywhere when the programmer does not want the normal substitutions to happen. For example:

```
puts {I have $20}
```

will print the string ‘I have $20’ and will not try variable substitution on the ‘$20’ part.
A word delineated by curly brackets is replaced with the characters within the brackets without performing the usual substitutions.

A word can begin with a double-quote and end with the matching closing double-quote. Substitutions as detailed above are done on the characters between the quotes, and the result is then substituted for the original word. Typically double-quotes are used to group sequences of characters that contain spaces into a single command word.

For example:

```tcl
set name "Fred the Great"
puts "Hello my name is $name"
```

outputs ‘Hello my name is Fred the Great’. The first command sets the value of variable name to the following double-quoted string "Fred the Great". The next command prints its argument, a single argument because it is a word delineated by double-quotes, that has had variable substitution performed on it.

Here is the same example but using curly brackets instead of double-quotes:

```tcl
set name {Fred the Great}
puts {Hello my name is $name}
```

gives the output ‘Hello my name is $name’ because substitutions are suppressed by the curly bracket notation.

And again the same example but without either curly brackets or double-quotes:

```tcl
set name Fred the Great
puts Hello my name is $name
```

simply fails because both set and puts expect a single argument but without the word grouping effects of double-quotes or curly brackets they find that they have more than one argument and throw an exception.

Being a simple scripting language, Tcl does not have any real idea of data types. The interpreter simply manipulates strings. The Tcl interpreter is not concerned with whether those strings contain representations of numbers or names or lists. It is up to the commands themselves to interpret the strings that are passed to them as arguments in any manner those choose.

10.45.2.2 Variables

This has been dealt with implicitly above. A variable has a name and a value. A name can be any string whatsoever, as can its value.

For example,

```tcl
set "Old King Cole" "merry soul"
```
sets the value of the variable named **Old King Cole** to the value **merry soul**. Variable names can also be numbers:

```tcl
set 123 "one two three"
```

sets the variable with name 123 to the value **one two three**. In general, it is better to use the usual conventions — start with a letter then follow with a combination of letters, digits, and underscores — when giving variables names to avoid confusion.

Array variables are also available in Tcl. These are denoted by an array name followed by an array index enclosed in round brackets. As an example:

```tcl
set fred(one) 1
set fred(two) 2
```

will set the variable **fred(one)** to the value 1 and **fred(two)** to the value 2.

Tcl arrays are associative arrays in that both the array name and the array index can be arbitrary strings. This also makes multidimensional arrays possible if the index contains a comma:

```tcl
set fred(one,two) 12
```

It is cheating in that the array is not stored as a multidimensional array with a pair of indices, but as a linear array with a single index that happens to contain a comma.

### 10.45.2.3 Commands

Now that the Tcl syntax and variables have been dealt with, we will now look at some of the commands that are available.

Each command when executed returns a value. The return value will be described along with the command.

A quick word about the **notation** used to describe Tcl commands. In general, a description of a command is the name of the command followed by its arguments separated by spaces. An example is:

```tcl
set varName ?value?
```

which is a description of the Tcl set command, which takes a variable name varName and an optional argument, a value.

Optional arguments are enclosed in question mark, ?, pairs, as in the example.

A series of three dots ... represents repeated arguments. An example is a description of the unset command:

```tcl
unset varName ?varName varName ...
```
which shows that the `unset` command has at least one compulsory argument `varName` but has any number of subsequent optional arguments.

The most used command over variables is the `set` command. It has the form

```
set varName ?value?
```

The value of `value` is determined, the variable `varName` is set to it, and the value is returned. If there is no `value` argument, then the value of the variable is simply returned. It is thus used to set and/or get the value of a variable.

The `unset` command is used to remove variables completely from the system:

```
unset varName ?varName varName ...?
```

which given a series of variable names deletes them. The empty string is always returned.

There is a special command for incrementing the value of a variable:

```
incr varName ?increment?
```

which, given the name of a variable whose value is an integer string, increments it by the amount `increment`. If the `increment` part is left out, then it defaults to 1. The return value is the new value of the variable.

Expressions are constructed from operands and operators and can then be evaluated. The most general expression evaluator in Tcl is the `expr` command:

```
expr arg ?arg arg ... arg?
```

which evaluates its arguments as an expression and returns the value of the evaluation.

A simple example expression is

```
expr 2 * 2
```

which when executed returns the value 4.

There are different classes of operators: arithmetic, relational, logical, bitwise, and choice. Here are some example expressions involving various operators:

<table>
<thead>
<tr>
<th>Class</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>$x * 2</td>
</tr>
<tr>
<td>relational</td>
<td>$x &gt; 2</td>
</tr>
<tr>
<td>logical</td>
<td>($x == $y)</td>
</tr>
<tr>
<td>bitwise</td>
<td>8 &amp; 2</td>
</tr>
<tr>
<td>choice</td>
<td>($a == 1) ? $x : $y</td>
</tr>
</tbody>
</table>

Basically the operators follow the syntax and meaning of their ANSI C counterparts.
Expressions to the `expr` command can be contained in curly brackets in which case the usual substitutions are not done before the `expr` command is evaluated, but the command does its own round of substitutions. So evaluating a script such as:

```
set a 1
expr { ($a==1) : "yes" ? "no" }
```

will evaluate to `yes`.

Tcl also has a whole host of math functions that can be used in expressions. Their evaluation is again the same as that for their ANSI C counterparts. For example:

```
expr { 2*log($x) }
```

will return 2 times the natural log of the value of variable `x`.

Tcl has a notion of `lists`, but as with everything it is implemented through strings. A list is a string that contains words.

A simple list is just a space separated series of strings:

```
set a {one two three four five}
```

will set the variable `a` to the list containing the five strings shown. The empty list is denoted by an open and close curly bracket pair with nothing in between: `{}`.

For the Prolog programmer, there is much confusion between a Prolog implementation of lists and the Tcl implementation of lists. In Prolog we have a definite notion of the printed representation of a list: a list is a sequence of terms enclosed in square brackets (we ignore dot notation for now); a nested list is just another term.

In Tcl, however, a list is really just a string that conforms to a certain syntax: a string of space separated words. But in Tcl there is more than one way of generating such a string. For example,

```
set fred {a b c d}
```

sets `fred` to

"a b c d"

as does

```
set fred "a b c d"
```

because `{a b c d}` evaluates to the string `a b c d`, which has the correct syntax for a list. But what about nested lists? Those are represented in the final list-string as being contained in curly brackets. For example:

```
set fred {a b c {1 2 3} e f}
```
results in \texttt{fred} having the value

"a b c \{1 2 3\} e f"

The outer curly brackets from the \texttt{set} command have disappeared, which causes confusion. The curly brackets within a list denote a nested list, but there are no curly brackets at the top level of the list. (We cannot help thinking that life would have been easier if the creators of Tcl would have chosen a consistent representation for lists, as Prolog and LISP do.)

So remember: a list is really a string with a certain syntax, space separated items or words; a nested list is surrounded by curly brackets.

There are a dozen commands that operate on lists.

\texttt{concat \?list list ...?}

This makes a list out of a series of lists by concatenating its argument lists together. The return result is the list resulting from the concatenation.

\texttt{lindex list index}

returns the \texttt{index}-th element of the \texttt{list}. The first element of a list has an index of 0.

\texttt{linsert list index value \?value ...?}

returns a new list in which the \texttt{value} arguments have been inserted in turn before the \texttt{index}-th element of \texttt{list}.

\texttt{list \?value value \ldots?}

returns a list where each element is one of the \texttt{value} arguments.

\texttt{llength list}

returns the number of elements in list \texttt{list}.

\texttt{lrange list first last}

returns a slice of a list consisting of the elements of the list \texttt{list} from index \texttt{first} until index \texttt{last}.

\texttt{lreplace list first last \?value \ldots value?}

returns a copy of list \texttt{list} but with the elements between indices \texttt{first} and \texttt{last} replaced with a list formed from the \texttt{value} arguments.

\texttt{lsearch \?-exact? \?-glob? \?-regexp? list pattern}
returns the index of the first element in the list that matches the given pattern. The type of matching done depends on which of the switch is present -exact, -glob, -regexp, is present. Default is -glob.

\[
\text{lsort } ?\text{-ascii? ?-integer? ?-real? ?-command command? ?-increasing? ?-decreasing}{? \text{list}}
\]
returns a list, which is the original list list sorted by the chosen technique. If none of the switches supplies the intended sorting technique, then the user can provide one through the -command command switch.

There are also two useful commands for converting between lists and strings:

\[
\text{join list ?joinString?}
\]
which concatenates the elements of the list together, with the separator joinString between them, and returns the resulting string. This can be used to construct filenames; for example:

\[
\begin{aligned}
\text{set a } &\{\{} \text{usr local bin} \\
\text{set filename } &\{\text{join } \$a /\}
\end{aligned}
\]
results in the variable filename having the value /usr/local/bin.

The reverse of the join command is the split command:

\[
\text{split string ?splitChars?}
\]
which takes the string string and splits it into string on splitChars boundaries and returns a list with the strings as elements. An example is splitting a filename into its constituent parts:

\[
\begin{aligned}
\text{set a } &\{\text{split /usr/local/src /}\}
\end{aligned}
\]
gives a the value {{} usr local src}, a list.

Tcl has the four usual classes of control flow found in most other programming languages:

\[
\text{if...elseif...else, while, for, foreach, switch, and eval.}
\]
We go through each in turn.

The general form of an if command is the following:

\[
\begin{aligned}
\text{if } &\text{test1 body1 } ?\text{elseif test2 body2 elseif...? ?else bodyn?}
\end{aligned}
\]
which when evaluated, evaluates expression test1, which if true causes body1 to be evaluated, but if false, causes test2 to be evaluated, and so on. If there is a final else clause, then its bodyn part is evaluated if all of the preceding tests failed. The return result of an if statement is the result of the last body command evaluated, or the empty list if none of the bodies are evaluated.
Conditional looping is done through the `while` command:

```
while test body
```

which evaluates expression `test`, which if true then evaluates `body`. It continues to do that until `test` evaluates to 0, and returns the empty string.

A simple example is:

```
set a 10
while {$a > 0} { puts $a; incr a -1 }
```

which initializes variable `a` with value ten and then loops printing out the value of `a` and decrementing it until its value is 0, when the loop terminates.

The `for` loop has the following form:

```
for init test reinit body
```

which initializes the loop by executing `init`, then each time around the loop the expression `test` is evaluated, which if true causes `body` to be executed and then executes `reinit`. The loop spins around until `test` evaluates to 0. The return result of a `for` loop is the empty string.

An example of a `for` loop:

```
for {set a 10} ($a>0) {incr a -1} {puts $a}
```

which initializes the variable `a` with value 10, then goes around the loop printing the value of `a` and decrementing it as long as its value is greater than 0. Once it reaches 0 the loop terminates.

The `foreach` command has the following form:

```
foreach varName list body
```

where `varName` is the name of a variable, `list` is an instance of a list, and `body` is a series of commands to evaluate. A `foreach` then iterates over the elements of a list, setting the variable `varName` to the current element, and executes `body`. The result of a `foreach` loop is always the empty string.

An example of a `foreach` loop:

```
foreach friend {joe mary john wilbert} {puts "I like $friend"}
```

will produce the output:

```
I like joe
I like mary
I like john
I like wilbert
```
There are also a couple of commands for controlling the flow of loops: `continue` and `break`. `continue` stops the current evaluation of the body of a loop and goes on to the next one. `break` terminates the loop altogether.

Tcl has a general switch statement, which has two forms:

```
switch ?options? string pattern body ?pattern body ... ?
switch ?options? string { pattern body ?pattern body ...? }
```

When executed, the switch command matches its `string` argument against each of the `pattern` arguments, and the `body` of the first matching pattern is evaluated. The matching algorithm depends on the options chosen, which can be one of:

- `-exact` use exact matching
- `-glob` use glob-style matching
- `-regexp` use regular expression matching

An example is:

```
set a rob
switch -glob $a {
    a*z { puts "A to Z"}
    r*b { puts "rob or rab"}
}
```

which will produce the output:

```
rob or rab
```

There are two forms of the `switch` command. The second form has the command arguments surrounded in curly brackets. This is primarily so that multi-line switch commands can be formed, but it also means that the arguments in brackets are not evaluated (curly brackets suppress evaluation), whereas in the first type of switch statement the arguments are first evaluated before the switch is evaluated. These effects should be borne in mind when choosing which kind of switch statement to use.

The final form of control statement is `eval`:

```
eval arg ?arg ...?
```

which takes one or more arguments, concatenates them into a string, and executes the string as a command. The return result is the normal return result of the execution of the string as a command.

An example is:

```
set a b
set b 0
eval set $a 10
```
which results in the variable b being set to 10. In this case, the return result of the eval is 10, the result of executing the string "set b 10" as a command.

Tcl has several commands over strings. There are commands for searching for patterns in strings, formatting and parsing strings (much the same as printf and scanf in the C language), and general string manipulation commands.

Firstly we will deal with formatting and parsing of strings. The commands for this are format and scan respectively.

format formatString ?value value ...?

which works in a similar to C’s printf; given a format string with placeholders for values and a series of values, return the appropriate string.

Here is an example of printing out a table for base 10 logarithms for the numbers 1 to 10:

for {set n 1} {$n <= 10} {incr n} {
    puts [format "log10(%d) = %.4f" $n [expr log10($n)]]
}

which produces the output

\[
\begin{align*}
\ln(1) &= 0.0000 \\
\ln(2) &= 0.3010 \\
\ln(3) &= 0.4771 \\
\ln(4) &= 0.6021 \\
\ln(5) &= 0.6990 \\
\ln(6) &= 0.7782 \\
\ln(7) &= 0.8451 \\
\ln(8) &= 0.9031 \\
\ln(9) &= 0.9542 \\
\ln(10) &= 1.0000
\end{align*}
\]

The reverse function of format is scan:

scan string formatString varName ?varName ...?

which parses the string according to the format string and assigns the appropriate values to the variables. It returns the number of fields successfully parsed.

An example,

\[
\begin{align*}
\text{scan "qty 10, unit cost 1.5, total 15.0" \ \\
cost_per_unit quantity total unit cost} \quad \text{\textbackslash}
\end{align*}
\]

would assign the value 10 to the variable quantity, 1.5 to the variable cost_per_unit and the value 15.0 to the variable total.
There are commands for performing two kinds of pattern matching on strings: one for matching using regular expressions, and one for matching using UNIX-style wildcard pattern matching (globbing).

The command for regular expressions matching is as follows:

```
regexp ?-indices? ?-nocase? exp string ?matchVar? ?subVar subVar ...?
```

where `exp` is the regular expression and `string` is the string on which the matching is performed. The `regexp` command returns 1 if the expression matches the string, 0 otherwise. The optional `-nocase` switch does matching without regard to the case of letters in the string. The optional `matchVar` and `subVar` variables, if present, are set to the values of string matches. In the regular expression, a match that is to be saved into a variable is enclosed in round braces. An example is

```
regexp {([0-9]+)} "I have 3 oranges" a
```

will assign the value 3 to the variable `a`.

If the optional switch `-indices` is present, then instead of storing the matching substrings in the variables, the indices of the substrings are stored; that is a list with a pair of numbers denoting the start and end position of the substring in the string. Using the same example:

```
regexp -indices {([0-9]+)} "I have 3 oranges" a
```

will assign the value "7 7", because the matched numeral 3 is in the eighth position in the string, and indices count from 0.

String matching using the UNIX-style wildcard pattern matching technique is done through the `string match` command:

```
string match pattern string
```

where `pattern` is a wildcard pattern and `string` is the string to match. If the match succeeds, then the command returns 1; otherwise, it returns 0. An example is

```
string match {([a-z] *[0-9])} {a_$_%^-_3}
```

which matches because the command says match any string that starts with a lower case letter and ends with a number, regardless of anything in between.

There is a command for performing string substitutions using regular expressions:

```
regsub ?-all? ?-nocase? exp string subSpec varName
```

where `exp` is the regular expression and `string` is the input string on which the substitution is made, `subSpec` is the string that is substituted for the part of the string matched by the regular expression, and `varName` is the variable on which the resulting string is copied into. With the `-nocase` switch, the matching is done without regard to the case of letters in the
input string. The -all switch causes repeated matching and substitution to happen on the input string. The result of a regsub command is the number of substitutions made.

An example of string substitution is:

```
regsub {#name#} {My name is #name#} Rob result
```

which sets the variable result to the value "My name is Rob". An example of using the -all switch:

```
regsub -all {#name#} {#name#'s name is #name#} Rob result
```

sets the variable result to the value "Rob’s name is Rob" and it returns the value 2 because two substitutions were made.

There are a host of other ways to manipulate strings through variants of the string command. Here we will go through them.

To select a character from a string given the character position, use the string index command. An example is:

```
string index "Hello world" 6
```

which returns w, the 7th character of the string. (Strings are indexed from 0).

To select a substring of a string, given a range of indices use the string range command. An example is:

```
string range "Hello world" 3 7
```

which returns the string "lo wo". There is a special index marker named end, which is used to denote the end of a string, so the code

```
string range "Hello world" 6 end
```

will return the string "world".

There are two ways to do simple search for a substring on a string, using the string first and string last commands. An example of string first is:

```
string first "dog" "My dog is a big dog"
```

find the first position in string "My dog is a big dog" that matches "dog". It will return the position in the string in which the substring was found, in this case 3. If the substring cannot be found, then the value -1 is returned.

Similarly,

```
string last "dog" "My dog is a big dog"
```
will return the value 16 because it returns the index of the last place in the string that the substring matches. Again, if there is no match, then -1 is returned.

To find the length of a string use string length, which given a string simply returns its length.

```
string length "123456"
```

returns the value 6.

To convert a string completely to upper case use string toupper:

```
string toupper "this is in upper case"
```

returns the string "THIS IS IN UPPER CASE".

Similarly,

```
string tolower "THIS IS IN LOWER CASE"
```

returns the string "this is in lower case".

There are commands for removing characters from strings: string trim, string trimright, and string trimleft.

```
string trim string ?chars?
```

which removes the characters in the string chars from the string string and returns the trimmed string. If chars is not present, then whitespace characters are removed. An example is:

```
string string "The dog ate the exercise book" "doe"
```

which would return the string "Th g at th xrcis bk".

string trimleft is the same as string trim except only leading characters are removed. Similarly string trimright removes only trailing characters. For example:

```
string trimright $my_input
```

would return a copy of the string contained in $my_input but with all the trailing whitespace characters removed.

There is a comprehensive set of commands for file manipulation. We will cover only the some of the more important ones here.

To open a file the open command is used:

```
open name ?access?
```

where name is a string containing the filename, and the option access parameter contains a string of access flags, in the UNIX style. The return result is a handle to the open file.
If access is not present, then the access permissions default to "r", which means open for reading only. The command returns a file handle that can be used with other commands. An example of the use of the open command is

```
set fid [open "myfile" "r+"
```

which means open the file \texttt{myfile} for both reading and writing and set the variable \texttt{fid} to the file handle returned.

To close a file simply use

```
close fileId
```

For example,

```
close $fid
```

will close the file that has the file handle stored in the variable \texttt{fid}.

To read from a file, the \texttt{read} command is used:

```
read fileId numBytes
```

which reads \texttt{numBytes} bytes from the file attached to file handle \texttt{fileId}, and returns the bytes actually read.

To read a single line from a file use \texttt{gets}:

```
gets fileId ?varName?
```

which reads a line from the file attached to file handle \texttt{fileId} but chops off the trailing newline. If variable \texttt{varName} is specified, then the string read in is stored there and the number of bytes is returned by the command. If the variable is not specified, then the command returns the string only.

To write to a file, use \texttt{puts}:

```
puts ?-nonewline? ?fileId? string
```

which outputs the string \texttt{string}. If the file handle \texttt{fileId} is present, then the string is output to that file; otherwise, it is printed on \texttt{stdout}. If the switch \texttt{-nonewline} is present, then a trailing newline is not output.

To check if the end of a file has been reached, use \texttt{eof}:

```
eof fileId
```

which, given a file handle \texttt{fileId} returns 1 if the end has been reached, and 0 otherwise.

The are a host of other commands over files and processes, which we will not go into here.

(For extra information on file I/O commands, refer to the Tcl manual pages.)
Tcl provides a way of creating new commands, called procedures, that can be executed in scripts. The arguments of a procedure can be call-by-value or call-by-reference, and there is also a facility for creating new user defined control structures using procedures.

A procedure is declared using the `proc` command:

```
proc name argList body
```

where the name of the procedure is `name`, the arguments are contained in `argList` and the body of the procedure is the script `body`. An example of a procedure is:

```
proc namePrint { first family } {
    puts "My first name is $first"
    puts "My family name is $family"
}
```

which can be called with

```
namePrint Tony Blair
```

to produce the output:

```
My first name is Tony
My family name is Blair
```

A procedure with no arguments is specified with an empty argument list. An example is a procedure that just prints out a string:

```
proc stringThing {} {
    puts "I just print this string"
}
```

Arguments can be given defaults by pairing them with a value in a list. An example here is a counter procedure:

```
proc counter { value { inc 1 } } {
    eval $value + $inc
}
```

which can be called with two arguments like this

```
set v 10
set v [counter $v 5]
```

which will set variable `v` to the value 15; or it can be called with one argument:

```
set v 10
set v [counter $v]
```

in which case `v` will have the value 11, because the default of the argument `inc` inside the procedure is the value 1.
There is a special argument for handling procedures with variable number of arguments, the **args** argument. An example is a procedure that sums a list of numbers:

```prolog
proc sum { args } {
    set result 0;
    foreach n $args {
        set result [expr $result + $n ]
    }
    return $result;
}
```

which can be called like this:

```prolog
sum 1 2 3 4 5
```

which returns the value 15.

The restriction on using defaulted arguments is that all the arguments that come after the defaulted ones must also be defaulted. If **args** are used, then it must be the last argument in the argument list.

A procedure can return a value through the **return** command:

```prolog
return ?options? ?value?
```

which terminates the procedure returning value **value**, if specified, or just causes the procedure to return, if no value specified. (The ?options? part has to do with raising exceptions, which we will will not cover here.)

The return result of a user defined procedure is the return result of the last command executed by it.

So far we have seen the arguments of a procedure are passed using the call-by-value mechanism. They can be passed call by reference using the **upvar** command:

```prolog
upvar ?level? otherVar1 myVar1 ?otherVar2 myVar2 ...?
```

which makes accessible variables somewhere in a calling context with the current context. The optional argument **level** describes how many calling levels up to look for the variable. This is best shown with an example:

```prolog
set a 10
set b 20

proc add { first second } {
    upvar $first f $second s
    expr $f+$s
}
```
which when called with

```prolog
add a b
```

will produce the result 30. If you use call-by-value instead:

```prolog
add $a $b
```

then the program will fail because when executing the procedure `add` it will take the first argument 10 as the level argument, a bad level. (Also variable `20` does not exist at any level.)

New control structures can be generated using the `uplevel` command:

```prolog
uplevel ?level? arg ?arg arg ...?
```

which is like `eval`, but it evaluates its arguments in a context higher up the calling stack. How far up the stack to go is given by the optional `level` argument.

```prolog
proc do { loop condition } {
  set nostop 1

  while { $nostop } {
    uplevel $loop
    if {[uplevel "expr $condition"] == 0} {
      set nostop 0
    }
  }
}
```

which when called with this

```prolog
set x 5
do { puts $x; incr x -1 } { $x > 0 }
```

will print

```
5
4
3
2
1
```

(Please note: this does not quite work for all kinds of calls because of `break`, `continue`, and `return`. It is possible to get around these problem, but that is outside the scope of this tutorial.)

A word about the **scope of variables**. Variables used within procedures are normally created only for the duration of that procedure and have local scope.
It is possible to declare a variable as having global scope, through the `global` command:

```
global name1 ? name2 ...?
```

where `name1`, `name2`, ..., are the names of global variables. Any references to those names will be taken to denote global variables for the duration of the procedure call.

Global variables are those variables declared at the topmost calling context. It is possible to run a `global` command at anytime in a procedure call. After such a command, the variable name will refer to a global variable until the procedure exits.

An example:

```
set x 10

proc fred {} {
    set y 20
    global x
    puts [expr $x + $y]
}

fred
```

will print the result 30 where 20 comes from the local variable `y` and 10 comes from the global variable `x`.

Without the `global x` line, the call to `fred` will fail with an error because there is no variable `x` defined locally in the procedure for the `expr` to evaluate over.

In common with other scripting languages, there is a command for evaluating the contents of a file in the Tcl interpreter:

```
source fileName
```

where `fileName` is the filename of the file containing the Tcl source to be evaluated. Control returns to the Tcl interpreter once the file has been evaluated.

### 10.45.2.4 What We Have Left Out

We have left out a number of Tcl commands as they are outside of the scope of this tutorial. We list some of them here to show some of what Tcl can do. Please refer to the Tcl manual for more information.

- `http` implements the HTTP protocol for retrieving web pages
- `namespaces` a modules systems for Tcl
- `trace` commands can be attached to variables that are triggered when the variable changes value (amongst other things)
- `processes` start, stop, and manage processes
sockets UNIX and Internet style socket management

exception handling

3rd party extension packages
load extension packages into Tcl and use their facilities as native Tcl commands

10.45.3 Tk
Tk is an extension to Tcl. It provides Tcl with commands for easily creating and managing graphical objects, or widgets, so providing a way to add graphical user interfaces (GUIs) to Tcl applications.

In this section we will describe the main Tk widgets, the Tcl commands used to manipulate them, how to give them behaviors, and generally how to arrange them into groups to create a GUI.

10.45.3.1 Widgets
A widget is a “window object”. It is something that is displayed that has at least two parts: a state and a behavior. An example of a widget is a button. Its state is things like what color is it, what text is written it in, and how big it is. Its behavior is things like what it does when you click on it, or what happens when the cursor is moved over or away from it.

In Tcl/Tk there are three parts to creating a useful widget. The first is creating an instance of the widget with its initial state. The second is giving it a behavior by defining how the widget behaves when certain events happen — event handling. The third is actually displaying the widget possibly in a group of widgets or inside another widget — geometry management. In fact, after creating all the widgets for a GUI, they are not displayed until handled by a geometry manager, which has rules about how to calculate the size of the widgets and how they will appear in relation to each other.

10.45.3.2 Types of Widget
In Tcl/Tk there are currently 15 types of widget. In alphabetical order they are (see also library(‘tcltk/examples/widgets.tcl’)):

button a simple press button
canvas is a container for displaying “drawn” objects such as lines, circles, and polygons.
checkbutton a button that hold a state of either on or off
entry a text entry field
frame a widget that is a container for other widgets
label a simple label
listbox a box containing a list of options
menu a widget for creating menu bars
menubutton a button, which when pressed offers a selection of choices
message  a multi-line text display widget

 radiobutton  a button used to form groups of mutually interacting buttons (When one button is pressed down, the others pop up.)

 scale  is like a slider on a music console. It consists of a trough scale and a slider. Moving the slider to a position on the scale sets the overall value of the widget to that value.

 scrollbar  used to add scrollbars to windows or canvases. The scrollbar has a slider, which when moved changes the value of the slider widget.

 text  a sophisticated multi-line text widget that can also display other widgets such as buttons

 toplevel  for creating new standalone toplevel windows. (These windows are containers for other widgets. They are not terminal windows.)
Meet The Main Tk Widgets

10.45.3.3 Widgets Hierarchies

Before going further it is necessary to understand how instances of widgets are named. Widgets are arranged in a hierarchy. The names of widget instances are formed from dot
separated words. The root window is simply . on its own. So for, example, a button widget that is displayed in the root window might have the name .b1. A button that is displayed inside a frame that is displayed inside the root window may have the name .frame1.b1. The frame would have the name .frame1.

Following this notation, it is clear that widgets are both formed in hierarchies, with the dot notation giving the path to a widget, and in groups, all widgets with the same leading path are notionally in the same group.

(It is a similar to the way file systems are organized. A file has a path that shows where to find it in the hierarchical file system. But also files with the same leading path are in the same directory/folder and so are notionally grouped together.)

An instance of a widget is created through the a Tcl command for that widget. The widget command may have optional arguments set for specifying various attributes of the widget that it will have when it is created. The result of a successful widget command is the name of the new widget.

For example, a command to create a button widget named .mybutton that displays the text “I am a button” would look like this:

```
button .mybutton -text "I am a button"
```

and this will return the name .mybutton.

A widget will only be created if all the windows/widgets in the leading path of the new widget also exist, and also that the name of the new widget does not already exist.

For example, the following

```
button .mybutton -text "I am a button"
bbutton .mybutton -text "and so am I"
```

will fail at the second command because there is also a widget named .mybutton from the first command.

The following will also fail

```
button .frame.mybutton -text "I am a button"
```

if there is no existing widget with the name .frame to be the parent of .mybutton.

All this begs the question: why are widgets named and arranged in a hierarchy? Is not a GUI just a bunch of widgets displayed in a window?

This is not generally how GUIs are arranged. For example, they often have a menubar over the top of each window. The menubar contains pulldown menus. The pulldown menus may have cascading menu items that may cascade down several levels. Under the menu bar is the main part of the window that may also be split into several “frames”. A left hand frame may have a set of buttons in it, for example. And so on. From this you can see that the
widgets in GUIs are naturally arranged in a hierarchy. To achieve this in Tcl/Tk instances of widgets are placed in a hierarchy, which is reflected in their names.

Now we will go through each of the widget commands in turn. Each widget command has many options most of which will not be described here. Just enough will be touched on for the reader to understand the basic operation of each widget. For a complete description of each widget and its many options refer to the Tk manual.

10.45.3.4 Widget Creation

As has already been said, a widget is a window object that has state and behavior. In terms of Tcl/Tk a widget is created by calling a widget creation command. There is a specific widget creation for each type of widget.

The widget creation command is supplied with arguments. The first argument is always the name you want to give to the resulting widget; the other arguments set the initial state of the widget.

The immediate result of calling a widget creation command is that it returns the name of the new widget. A side effect is that the instance of the widget is created and its name is defined as in the Tcl interpreter as a procedure through which the widget state can be accessed and manipulated.

This needs an example. We will use the widget creator command button to make a button widget:

```
button .fred -text 'Fred' -background red
```

which creates an instance of a button widget named .fred that will display the text Fred on the button and will have a red background color. Evaluating this command returns the string .fred, the name of the newly created widget.

As a side effect, a Tcl procedure named .fred is created. A call to a widget instance has the following form:

```
widgetName method methodArgs
```

where widgetName is the name of the widget to be manipulated, method is the action to be performed on the widget, and methodArgs are the arguments passed to the method that is performed on the widget.

The two standard methods for widgets are configure and cget. configure - is used to change the state of a widget; for example:

```
.fred configure -background green -text 'Sid'
```

will change the background color of the widget .fred to green and the text displayed to Sid.

cget is used to get part of the state of a widget; for example:
.fred cget -text

will return Sid if the text on the button .fred is Sid.

In addition to these general methods, there are special methods for each widget type. For example, with button widgets you have the flash and invoke methods.

For example,

    .fred invoke

can be called somewhere in the Tcl code to invoke button .fred as though it had been clicked on.

    .fred flash

can be called somewhere in the Tcl code to cause the button to flash.

We will come across some of these special method when we discuss the widgets in detail. For a comprehensive list of widget methods, refer to entry for the appropriate widget creation command in the Tcl/Tk manual.

We now discuss the widget creation command for each widget type.

A label is a simple widget for displaying a single line of text. An example of creating an instance of a label is

    label .l -text "Hello world!"

which simply creates the label named .l with the text ‘Hello world!’ displayed in it. Most widgets that display text can have a variable associated with them through the option -textvariable. When the value of the variable is changed the text changes in the associated label. For example,

    label .l -text "Hello world!" -textvariable mytext

creates a text label called .l displaying the initial text ‘Hello world!’ and associated text variable mytext; mytext will start with the value ‘Hello world!’. However, if the following script is executed:

    set mytext "Goodbye moon!"

then the text in the label will magically change to ‘Goodbye moon!’.

A message widget is similar to a label widget but for multi-line text. As its name suggests it is mostly used for creating popup message information boxes.

An example of a message widget is
message .msg -text "Your data is incorrect.\n\nPlease correct it and try again." -justify center

which will create a message widget displaying the text shown, center justified. The width of the message box can be given through the -width switch. Any lines that exceed the width of the box are wrapped at word boundaries.

Calling the button command creates an instance of a button widget. An example is:

```
button .mybutton -text "hello" -command {puts "howdie!"}
```

which creates a button with name .mybutton that will display the text "hello" and will execute the Tcl script puts "howdie!" (that is print howdie! to the terminal) when clicked on.

Checkbuttons are buttons that have a fixed state that is either on or off. Clicking on the button toggles the state. To store the state, a checkbutton is associated with a variable. When the state of the checkbutton changes, so does that of the variable. An example is:

```
checkbutton .on_or_off -text "I like ice cream" -variable ice
```

which will create a checkbutton with name .on_or_off displaying the text ‘I like ice cream’ and associated with the variable ice. If the checkbutton is checked, then ice will have the value 1; if not checked, then it will have the value 0. The state of the checkbutton can also be changed by changing the state of the variable. For example, executing

```
set ice 0
```

will set the state of .on_or_off to not checked.

Radiobuttons are buttons that are grouped together to select one value among many. Each button has a value, but only one in the button group is active at any one time. In Tcl/Tk this is achieved by creating a series of radiobutton that share an associated variable. Each button has a value. When a radiobutton is clicked on, the variable has that value and all the other buttons in the group are put into the off state. Similarly, setting the value of the variable is reflected in the state of the button group. An example is:

```
radiobutton .first -value one -text one -variable count
radiobutton .second -value two -text two -variable count
radiobutton .third -value three -text three -variable count
```

which creates three radiobuttons that are linked through the variable count. If button .second is active, for example, then the other two buttons are in the inactive state and count has the value two. The following code sets the button group to make the button .third active and the rest inactive regardless of the current state:

```
set count three
```
If the value of count does not match any of the values of the radiobuttons, then they will all be off. For example executing the script

```tcl
set count four
```

will turn all the radiobuttons off.

An entry widget allows input of a one line string. An example of an entry widget:

```tcl
label .l -text "Enter your name"
entry .e -width 40 -textvariable your_name
```

would display a label widget named .l showing the string ‘Enter your name’ and an entry widget named .e of width 40 characters. The value of variable your_name will reflect the string in the entry widget: as the entry widget string is updated, so is the value of the variable. Similarly, changing the value of your_name in a Tcl script will change the string displayed in the entry field.

A scale widget is for displaying an adjustable slider. As the slider is moved its value, which is displayed next to the slider, changes. To specify a scale, it must have -from and -to attributes, which is the range of the scale. It can have a -command option, which is set to a script to evaluate when the value of the slider changes.

An example of a scale widget is:

```tcl
scale .s -from 0 -to 100
```

which creates a scale widget with name .s that will slide over a range of integers from 0 to 100.

There are several other options that scales can have. For example it is possible to display tick marks along the length of the scale through the -tickinterval attribute, and it is possible to specify both vertically and horizontally displayed scales through the -orient attribute.

A listbox is a widget that displays a list of single line strings. One or more of the strings may be selected through using the mouse. Initializing and manipulating the contents of a listbox is done through invoking methods on the instance of the listbox. As examples, the insert method is used to insert a string into a listbox, delete to delete one, and get to retrieve a particular entry. Also the currently selected list items can be retrieved through the selection command.

Here is an example of a listbox that is filled with entries of the form entry N:

```tcl
listbox .l
for { set i 0 } { $i<10 } { incr i } {
    .l insert end "entry $i"
}
```
A listbox may be given a height and/or width attribute, in which case it is likely that not all of the strings in the list are visible at the same time. There are a number of methods for affecting the display of such a listbox.

The `see` method causes the listbox display to change so that a particular list element is in view. For example,

```
.l see 5
```

will make sure that the sixth list item is visible. (List elements are counted from element 0.)

A `scrollbar` widget is intended to be used with any widget that is likely to be able to display only part of its contents at one time. Examples are listboxes, canvases, text widgets, and frames, amongst others.

A scrollbar widget is displayed as a movable slider between two arrows. Clicking on either arrow moves the slider in the direction of the arrow. The slider can be moved by dragging it with the cursor.

The scrollbar and the widget it scrolls are connected through Tcl script calls. A scrollable widgets will have a `scrollcommand` attribute that is set to a Tcl script to call when the widget changes its view. When the view changes the command is called, and the command is usually set to change the state of its associated scrollbar.

Similarly, the scrollbar will have a `command` attribute that is another script that is called when an action is performed on the scrollbar, like moving the slider or clicking on one of its arrows. That action will be to update the display of the associated scrollable widget (which redraws itself and then invokes its `scrollcommand`, which causes the scrollbar to be redrawn).

How this is all done is best shown through an example:

```
listbox .l -yscrollcommand ".s set" -height 10
scrollbar .s -command ".l yview"
for { set i 0 } { $i < 50 } { incr i } {
   .l insert end "entry $i"
}
```

creates a listbox named `.l` and a scrollbar named `.s`. Fifty strings of the form `entry N` are inserted into the listbox. The clever part is the way the scrollbar and listbox are linked. The listbox has its `yscrollcommand` attribute set to the script `".s set"`. What happens is that if the view of `.l` is changed, this script is called with 4 arguments attached: the number of entries in the listbox, the size of the listbox window, the index of the first entry currently visible, and the index of the last entry currently visible. This is exactly enough information for the scrollbar to work out how to redisplay itself. For example, changing the display of the above listbox could result in the following `yscrollcommand` script being called:
.s set 50 10 5 15

which says that the listbox contains 50 elements, it can display 10 at one time, the first
element displayed has index 5 and the last one on display has index 15. This call invokes
the set method of the scrollbar widget .s, which causes it to redraw itself appropriately.

If, instead, the user interacts with the scrollbar, then the scrollbar will invoke its -command
script, which in this example is ".l yview". Before invoking the script, the scrollbar widget
calculates which element should the first displayed in its associated widget and appends its
index to the call. For example, if element with index 20 should be the first to be displayed,
then the following call will be made:

    .l yview 20

which invokes the yview method of the listbox .l. This causes .l to be updated (which
then causes its -yscrollcommand to be called, which updates the scrollbar).

A frame widget does not do anything by itself except reserve an area of the display. Although
this does not seem to have much purpose, it is a very important widget. It is a container
widget; that is, it is used to group together collections of other widgets into logical groups.
For example, a row of buttons may be grouped into a frame, then as the frame is manipulated
so will the widgets displayed inside it. A frame widget can also be used to create large areas
of color inside another container widget (such as another frame widget or a toplevel widget).

An example of the use of a frame widget as a container:

    canvas .c -background red
    frame .f
    button .b1 -text button1
    button .b2 -text button2
    button .b3 -text button3
    button .b4 -text button4
    button .b5 -text button5
    pack .b1 .b2 .b3 .b4 .b5 -in .f -side left
    pack .c -side top -fill both -expand 1
    pack .f -side bottom

which specifies that there are two main widgets a canvas named .c and a frame named .f.
There are also 5 buttons, .b1 through .b5. The buttons are displayed inside the frame.
Then the canvas is displayed at the top of the main window and the frame is displayed at
the bottom. As the frame is displayed at the bottom, then so will the buttons because they
are displayed inside the frame.

(The pack command causes the widgets to be handled for display by the packer geometry
manager. The -fill and -expand 1 options to pack for .c tell the display manager that
if the window is resized, then the canvas is to expand to fill most of the window. You will
learn about geometry managers later in the Geometry Managers section.)
A *toplevel* widget is a new toplevel window. It is a container widget inside which other widgets are displayed. The root toplevel widget has path . — i.e. dot on its own. Subsequent toplevel widgets must have a name that is lower down the path tree just like any other widget.

An example of creating a toplevel widget is:

```plaintext
toplevel .t
```

All the widgets displayed inside .t must also have .t as the root of their path. For example, to create a button widget for display inside the .t toplevel the following would work:

```plaintext
button .t.b -text "Inside 't'"
```

(Attributes, such as size and title, of toplevel widgets can be changed through the `wm` command, which we will not cover in this tutorial. The reader is referred to the Tk manual.)

Yet another kind of container is a *menu widget*. It contains a list of widgets to display inside itself, as a pulldown menu. A simple entry in a menu widget is a `command` widget, displayed as an option in the menu widget, which if chosen executes a Tcl command. Other types of widgets allowed inside a menu widget are radiobuttons and checkboxes. A special kind of menu item is a `separator` that is used to group together menu items within a menu. (It should be noted that the widgets inside a menu widget are special to that menu widget and do not have an independent existence, and so do not have their own Tk name.)

A menu widget is built by first creating an instance of a menu widget (the container) and then invoking the `add` method to make entries into the menu. An example of a menu widget is as follows:

```plaintext
menu .m
  .m add command -label "Open file" -command "open_file"
  .m add command -label "Open directory" -command "open_directory"
  .m add command -label "Save buffer" -command "save_buffer"
  .m add command -label "Save buffer as..." -command "save_buffer_as"
  .m add separator
  .m add command -label "Make new frame" -command "new_frame"
  .m add command -label "Open new display" -command "new_display"
  .m add command -label "Delete frame" -command "delete_frame"
```

which creates a menu widget called .m, which contains eight menu items, the first four of which are commands, then comes a separator widget, then the final three command entries. (Some of you will notice that this menu is a small part of the Files menu from the menubar of the Emacs text editor.)

An example of a checkbox and some radiobutton widget entries:
.m add checkbox -label "Inverse video" -variable inv_vid
.m add radiobutton -label "black" -variable color
.m add radiobutton -label "blue" -variable color
.m add radiobutton -label "red" -variable color

which gives a checkbox displaying ‘Inverse video’, keeping its state in the variable inv_vid, and three radiobuttons linked through the variable color.

Another menu item variant is the cascade variant, which is used to make cascadable menus, i.e. menus that have submenus. An example of a cascade entry is the following:

.m add cascade -label "I cascade" -menu .m.c

which adds a cascade entry to the menu .m that displays the text ‘I cascade’. If the ‘I cascade’ option is chosen from the .m menu, then the menu .m.c will be displayed.

The cascade option is also used to make menubars at the top of an application window. A menubar is simply a menu each element of which is a cascade entry, (for example). The menubar menu is attached to the application window through a special configuration option for toplevel widgets, the -menu option. Then a menu is defined for each of the cascade entry in the menubar menu.

There are a large number of other variants to menu widgets: menu items can display bitmaps instead of text; menus can be specified as tear-off menus; accelerator keys can be defined for menu items; and so on.

A menubutton widget displays like a button, but when activated a menu pops up. The menu of the menubutton is defined through the menu command and is attached to the menubutton. An example of a menubutton:

menubutton .mb -menu .mb.m -text "mymenu"
menu .mb.m
.mb.m add command -label hello
.mb.m add command -label goodbye

which crate a menubutton widget named .mb with attached menu .mb.m and displays the text ‘mymenu’. Menu .mb.m is defined as two command options, one labeled hello and the other labeled goodbye. When the menubutton .mb is clicked on, the menu .mb.m will popup and its options can be chosen.

A canvas widget is a container widget that is used to manage the drawing of complex shapes; for example, squares, circles, ovals, and polygons. (It can also handle bitmaps, text and most of the Tk widgets too.) The shapes may have borders, filled in, be clicked on, moved around, and manipulated.

We will not cover the working of the canvas widget here. It is enough to know that there is a powerful widget in the Tk toolkit that can handle all manner of graphical objects. The interested reader is referred to the Tk manual.
A text widget is another powerful container widget that handles multi-line texts. The textwidget can display texts with varying font styles, sizes, and colors in the same text, and can also handle other Tk widgets embedded in the text.

The text widget is a rich and complicated widget and will not be covered here. The interested reader is referred to the Tk manual.

### 10.45.3.5 Geometry Managers

So far we have described each of the Tk widgets but have not mentioned how they are arranged to be displayed. Tk separates the creating of widgets from the way they are arranged for display. The “geometry” of the display is handled by a “geometry manager”. A geometry manager is handed the set of widgets to display with instructions on their layout. The layout instructions are particular to each geometry manager.

Tk comes with three distinct geometry managers: grid, place, and pack. As might be expected the grid geometry manager is useful for creating tables of widgets, for example, a table of buttons.

The place geometry manager simply gives each widget an X and Y coordinate and places them at that coordinate in their particular parent window.

The pack geometry manager places widgets according to constraints, like “these three button widgets should be packed together from the left in their parent widget, and should resize with the parent”.

(In practice the grid and pack geometry managers are the most useful because they can easily handle events such as resizing of the toplevel window, automatically adjusting the display in a sensible manner. place is not so useful for this.)

Each container widget (the master) has a geometry manager associated with it, which tells the container how to display its sub-widgets (slaves) inside it. A single master has one and only one kind of geometry manager associated with it, but each master can have a different kind. For example, a frame widget can use the packer to pack other frames inside it. One of the slave frames could use the grid manager to display buttons inside it itself, while another slave frame could use the packer to pack labels inside it itself.

The problem is how to display widgets. For example, there is an empty frame widget inside which a bunch of other widgets will be displayed. The pack geometry manager’s solution to this problem is to successively pack widgets into the empty space left in the container widget. The container widget is the master widget, and the widgets packed into it are its slaves. The slaves are packed in a sequence: the packing order.

What the packer does is to take the next slave to be packed. It allocates an area for the slave to be packed into from the remaining space in the master. Which part of the space is allocated depends on instructions to the packer. When the size of the space has been determined, this is sliced off the free space, and allocated to the widget that is displayed in it. Then the remaining space is available to subsequent slaves.
At any one time the space left for packing is a rectangle. If the widget is too small to use up a whole slice from the length or breadth of the free rectangle, then still a whole slice is allocated so that the free space is always rectangular.

It can be tricky to get the packing instructions right to get the desired finished effect, but a large number of arrangements of widgets is possible using the packer.

Let us take a simple example: three buttons packed into the root window. First we create the buttons; see also library('tcltk/examples/ex3.tcl'):

```tcl
button .b1 -text b1
button .b2 -text b2
button .b3 -text b3
```

then we can pack them thus:

```tcl
pack .b1 .b2 .b3
```

which produces a display of the three buttons, one on top of the other, button .b1 on the top, and button .b3 on the bottom.

If we change the size of the text in button .b2 through the command:

```tcl
.b2 config -text "hello world"
```

then we see that the window grows to fit the middle button, but the other two buttons stay their original size.
Middle Button Widens

The packer defaults to packing widgets in from the top of the master. Other directions can be specified. For example, the command:

```
pack .b1 .b2 .b3 -side left
```

will pack starting at the left hand side of the window. The result of this is that the buttons are formed in a horizontal row with the wider button, .b2, in the middle.

Packing From The Left

It is possible to leave space between widgets through the `padding` options to the packer: `-padx` and `-pady`. What these do is to allocate space to the slave that is padded with the padding distances. An example would be:

```
pack .b1 .b2 .b3 -side left -padx 10
```

External Padding

which adds 10 pixels of space to either side of the button widgets. This has the effect of leaving 10 pixels at the left side of button .b1, 20 pixels between buttons .b1 and .b2, 20 pixels between buttons .b2 and .b3, and finally 10 pixels on the right side of button .b3.

That was external padding for spacing widgets. There is also internal padding for increasing the size of widgets in the X and Y directions by a certain amount, through `-ipadx` and `-ipady` options; i.e. internal padding. For example:
pack .b1 .b2 .b3 -side left -ipadx 10 -ipady 10

Internal Padding

instead of spacing out the widgets, will increase their dimensions by 10 pixels in each direction.

Remember that space is allocated to a widget from the currently available space left in the master widget by cutting off a complete slice from that space. It is often the case that the slice is bigger that the widget to be displayed in it.

There are further options for allowing a widget to fill the whole slice allocated to it. This is done through the -fill option, which can have one of four values: none for no filling (default), x to fill horizontally only, y to fill vertically only, and both to fill both horizontally and vertically at the same time.

Filling is useful, for example, for creating buttons that are the same size even though they display texts of differing lengths. To take our button example again, the following code produces three buttons, one on top of each other, but of the same size:

```
button .b1 -text b1
button .b2 -text "hello world"
button .b3 -text b3
pack .b1 .b2 .b3 -fill x
```

Using fill For Evenly Sized Widgets
How does this work? The width of the toplevel windows is dictated by button .b2 because it has the widest text. Because the three buttons are packed from top to bottom, the slices of space allocated to them are cut progressively straight along the top of the remaining space. i.e. each widget gets a horizontal slice of space the same width cut from the top-level widget. Only the wide button .b2 would normally fit the whole width of its slice. But by allowing the other two widgets to fill horizontally, they will also take up the whole width of their slices. The result: 3 buttons stacked on top of each other, each with the same width, although the texts they display are not the same length.

A further common example is adding a scrollbar to a listbox. The trick is to get the scrollbar to size itself to the listbox; see also library('tcltk/examples/ex9a.tcl'):

```tcl
listbox .l
scrollbar .s
pack .l .s -side left
```

![Scrollbar With Listbox, First Try](image)

So far we have a listbox on the left and a tiny scrollbar on the right. To get the scrollbar to fill up the vertical space around it add the following command:

```tcl
pack .s -fill y
```

Now the display looks like a normal listbox with a scrollbar.
Scrollbar With Listbox, Second Try

Why does this work? They are packed from the left, so first a large vertical slice of the master is given to the listbox, then a thin vertical slice is given to the scrollbar. The scrollbar has a small default width and height and so it does not fill the vertical space of its slice. But filling in the vertical direction (through the `pack .s -fill y` command) allows it to fill its space, and so it adjusts to the height of the listbox.

The `fill` packing option specifies whether the widget should fill space left over in its slice of space. A further option to take into account is what happens when the space allocated to the master widget is much greater than the that used by its slaves. This is not usually a problem initially because the master container widget is sized to shrink-wrap around the space used by its slaves. If the container is subsequently resized, however, to a much larger size, then there is a question as to what should happen to the slave widgets. A common example of resizing a container widget is the resizing of a top-level window widget.

The default behavior of the packer is not to change the size or arrangement of the slave widgets. There is an option though through the `expand` option to cause the slices of space allocated to slaves to expand to fill the newly available space in the master. `expand` can have one of two values: 0 for no expansion, and 1 for expansion.

Take the listbox-scrollbar example; see also `library('tcltk/examples/ex10.tcl')`:

```tcl
listbox .l
scrollbar .s
pack .l -side left
pack .s -side left -fill y
```

Initially this looks good, but now resize the window to a much bigger size. You will find that the listbox stays the same size and that empty space appears at the top and bottom of it, and that the scrollbar resizes in the vertical. It is now not so nice.
We can fix part of the problem by having the listbox expand to fill the extra space generated by resizing the window.

```
pack .l -side left -expand 1
```
Scrollbar And Listbox, Almost There

The problem now is that `expand` just expands the space allocated to the listbox, it does not stretch the listbox itself. To achieve that we need to apply the `fill` option to the listbox too.

```
pack .l -side left -expand 1 -fill both
```
Scrollbar And Listbox, Problem Solved Using `fill`

Now whichever way the top-level window is resized, the listbox-scrollbar combination should look good.

If more than one widget has the expansion bit set, then the space is allocated equally to those widgets. This can be used, for example, to make a row of buttons of equal size that resize to fill the widget of their container. Try the following code; see also `library('tcltk/examples/ex11.tcl')`:

```tcl
button .b1 -text "one"
button .b2 -text "two"
button .b3 -text "three"
pack .b1 .b2 .b3 -side left -fill x -expand 1
```
Resizing Evenly Sized Widgets

Now resize the window. You will see that the buttons resize to fill the width of the window, each taking an equal third of the width.

Please note: the best way to get the hang of the packer is to play with it. Often the results are not what you expect, especially when it comes to fill and expand options. When you have created a display that looks pleasing, always try resizing the window to see if it still looks pleasing, or whether some of your fill and expand options need revising.

There is an option to change how a slave is displayed if its allocated space is larger than itself. Normally it will be displayed centered. That can be changed by anchoring it with the -anchor option. The option takes a compass direction as its argument: n, s, e, w, nw, ne, sw, se, or c (for center).

For example, the previous example with the resizing buttons displays the buttons in the center of the window, the default anchoring point. If we wanted the buttons to be displayed at the top of the window, then we would anchor them there thus; see also library('tcltk/examples/ex12.tcl'):

```
button .b1 -text "one"
button .b2 -text "two"
button .b3 -text "three"
pack .b1 .b2 .b3 -side left -fill x -expand 1 -anchor n
```
Each button is anchored at the top of its slice and so in this case is displayed at the top of the window.

The packing order of widget can also be changed. For example,

```tcl
pack .b3 -before .b2
```

will change the positions of .b2 and .b3 in our examples.

### Changing The Packing Order Of Widgets

The *grid geometry manager* is useful for arranging widgets in grids or tables. A grid has a number of rows and columns and a widget can occupy one of more adjacent rows and columns.

A simple example of arranging three buttons; see also library('tcltk/examples/ex14.tcl'):

```tcl
button .b1 -text b1
button .b2 -text b2
button .b3 -text b3
grid .b1 -row 0 -column 0
grid .b2 -row 1 -column 0
grid .b3 -row 0 -column 1 -rowspan 2
```

this will display button .b1 above button .b2. Button .b3 will be displayed in the next column and it will take up two rows.
Using the **grid** Geometry Manager

However, `.b3` will be displayed in the center of the space allocated to it. It is possible to get it to expand to fill the two rows it has using the `-sticky` option. The `-sticky` option says to which edges of its cells a widget “sticks” to, i.e. expands to reach. (This is like the fill and expand options in the pack manager.) So to get `.b3` to expand to fill its space we could use the following:

```tcl
grid .b3 -sticky ns
```

which says stick in the north and south directions (top and bottom). This results in `.b3` taking up two rows and filling them.

---

**grid** Geometry Manager, Cells With Sticky Edges

There are plenty of other options to the grid geometry manager. For example, it is possible to give some rows/columns more “weight” than others, which gives them more space in the master. For example, if in the above example you wanted to allocate 1/3 of the width of the master to column 0 and 2/3 of the width to column 1, then the following commands would achieve that:

```tcl
grid columnconfigure . 0 -weight 1
grid columnconfigure . 1 -weight 2
```

which says that the weight of column 0 for master . (the root window) is 1 and the weight of column 1 is 2. Since column 1 has more weight than column 0 it gets proportionately more space in the master.

It may not be apparent that this works until you resize the window. You can see even more easily how much space is allocated to each button by making expanding them to fill their space through the sticky option. The whole example looks like this; see also `library('tcltk/examples/ex16.tcl')`:
Now resize the window to various sizes and we will see that button .b3 has twice the width of buttons .b1 and .b2.

Changing Row/Column Ratios

The same kind of thing can be specified for each row too via the `grid rowconfigure` command.

For other options and a full explanation of the grid manager see the manual.

`place` simply places the slave widgets in the master at the given x and y coordinates. It displays the widgets with the given width and height. For example (see also `library('tcltk/examples/ex17.tcl')`):
Using The `place` Geometry Manager

will place the buttons `.b1`, `.b2`, and `.b3` along a diagonal 100 pixels apart in both the x and y directions. Heights and widths can be given in absolute sizes, or relative to the size of the master in which case they are specified as a floating point proportion of the master; 0.0 being no size and 1.0 being the size of the master. x and y coordinates can also be specified in a relative way, also as a floating point number. For example, a relative y coordinate of 0.0 refers to the top edge of the master, while 1.0 refers to the bottom edge. If both relative and absolute x and y values are specified, then they are summed.

Through this system the placer allows widgets to be placed on a kind of rubber sheet. If all the coordinates are specified in relative terms, as the master is resized, then so will the slaves move to their new relative positions.

10.45.3.6 Event Handling

So far we have covered the widgets types, how instances of them are created, how their attributes can be set and queried, and how they can be managed for display using geometry managers. What we have not touched on is how to give each widget a behavior.
This is done through event handlers. Each widget instance can be given a window event handler for each kind of window event. A window event is something like the cursor moving into or out of the widget, a key press happening while the widget is active (in focus), or the widget being destroyed.

Event handlers are specified through the `bind` command:

```
bind widgetName eventSequence command
```

where `widgetName` is the name or class of the widget to which the event handler should be attached, `eventSequence` is a description of the event that this event handler will handle, and `command` is a script that is invoked when the event happens (i.e. it is the event handler).

Common event types are:

- **Key**
  - `KeyPress` when a key was pressed
  - `KeyRelease` when a key was released

- **Button**
  - `ButtonPress` when a mouse button was pressed
  - `ButtonRelease` when a mouse button was released

- **Enter** when the cursor moves into a widget
- **Leave** when the cursor moved our of a widget
- **Motion** when the cursor moves within a widget

There are other event types. Please refer to the Tk documentation for a complete list.

The `eventSequence` part of a `bind` command is a list of one or more of these events, each event surrounded by angled brackets. (Mostly, an event sequence consists of handling a single event. Later we will show more complicated event sequences.)

An example is the following:

```
button .b -text "click me"
pack .b
bind .b <Enter> { puts "entering .b" }
```

makes a button `.b` displaying text ‘click me’ and displays it in the root window using the packing geometry manager. The `bind` command specifies that when the cursor enters (i.e. goes onto) the widget, then the text `entering .b` is printed at the terminal.

We can make the button change color as the cursor enters or leaves it like this:
button .b -text "click me" -background red
pack .b
bind .b <Enter> { .b config -background blue }
bind .b <Leave> { .b config -background red }

which causes the background color of the button to change to blue when the cursor enters it and to change back to red when the cursor leaves.

An action can be appended to an event handler by prefixing the action with a + sign. An example is:

    bind .b <Enter> {+puts "entering .b"}

which, when added to the example above, would not only change the color of the button to red when the cursor enters it, but would also print entering .b to the terminal.

A binding can be revoked simply by binding the empty command to it:

    bind .b <Enter> {}

A list of events that are bound can be found by querying the widget thus:

    bind .b

which will return a list of bound events.

To get the current command(s) bound to an event on a widget, invoke bind with the widget name and the event. An example is:

    bind .b <Enter>

which will return a list of the commands bound to the event <Enter> on widget .b.

Binding can be generalized to sequences of events. For example, we can create an entry widget that prints spells rob each time the key sequence ESC r o b happens:

    entry .e
    pack .e
    bind .e <Escape>rob {puts "spells rob"}

(A letter on its own in an event sequence stands for that key being pressed when the corresponding widget is in focus.)

Events can also be bound for entire classes of widgets. For example, if we wanted to perform the same trick for ALL entry widgets, then we could use the following command:

    bind entry <Escape>rob {puts "spells rob"}

In fact, we can bind events over all widgets using all as the widget class specifier.
The event script can have substitutions specified in it. Certain textual substitutions are then made at the time the event is processed. For example, \%x in a script gets the x coordinate of the mouse substituted for it. Similarly, \%y becomes the y coordinate, \%W the dot path of the window on which the event happened, \%K the keysym of the button that was pressed, and so on. For a complete list, see the manual.

In this way it is possible to execute the event script in the context of the event.

A clever example of using the all widget specifier and text substitutions is given in John Ousterhout’s book on Tcl/Tk (see Section 10.45.7 [Resources], page 861):

```tcl
bind all <Enter> {puts "Entering \%W at (\%x, \%y)"}
binding all <Leave> {puts "Leaving \%W at (\%x, \%y)"}
binding all <Motion> {puts "Pointer at (\%x, \%y)"}
```

which implements a mouse tracker for all the widgets in a Tcl/Tk application. The widget’s name and x and y coordinates are printed at the terminal when the mouse enters or leaves any widget, and also the x and y coordinates are printed when the mouse moves within a widget.

### 10.45.3.7 Miscellaneous

There are a couple of other Tk commands that we ought to mention: `destroy` and `update`.

The `destroy` command is used to destroy a widget, i.e. remove it from the Tk interpreter entirely and so from the display. Any children that the widget may have are also destroyed. Anything connected to the destroyed widget, such as bindings, are also cleaned up automatically.

For example, to create a window containing a button that is destroyed when the button is pressed:

```tcl
button .b -text "Die!" -command { destroy . }
pack .b
```

creates a button .b displaying the text ‘Die!’, which runs the command `destroy .` when it is pressed. Because the widget . is the main toplevel widget or window, running that command will kill the entire application associated with that button.

The command `update` is used to process any pending Tk events. An event is not just such things as moving the mouse but also updating the display for newly created and displayed widgets. This may be necessary in that usually Tk draws widgets only when it is idle. Using the `update` command forces Tk to stop and handle any outstanding events including updating the display to its actually current state, i.e. flushing out the pending display of any widgets. (This is analogous to the `fflush` command in C that flushes writes on a stream to disk. In Tk displaying of widgets is “buffered”; calling the `update` command flushes the buffer.)
10.45.3.8 What We Have Left Out
There are a number of Tk features that we have not described but we list some of them here in case the reader is interested. Refer to the Tk manual for more explanation.

- **photo**: creating full color images through the command
- **wm**: setting and getting window attributes
- Selection and focus commands
- Modal interaction
  - (not recommended)
- **send**: sending messages between Tk applications

10.45.3.9 Example pure Tcl/Tk program
To show some of what can be done with Tcl/Tk, we will show an example of part of a GUI for an 8-queens program. Most people will be familiar with the 8-queens problem: how to place 8 queens on a chess board such that they do not attack each other according to the normal rules of chess.

Our example will not be a program to solve the 8-queens problem (that will come later in the tutorial) but just the Tcl/Tk part for displaying a solution. The code can be found in `library('tcltk/examples/ex18.tcl')`.

The way an 8-queens solution is normally presented is as a list of numbers. The position of a number in the list indicates the column the queens is placed at and the number itself indicates the row. For example, the Prolog list `[8, 7, 6, 5, 4, 3, 2, 1]` would indicate 8 queens along the diagonal starting a column 1, row 8 and finishing at column 8 row 1.

The problem then becomes, given this list of numbers as a solution, how to display the solution using Tcl/Tk. This can be divided into two parts: how to display the initial empty chess board, and how to display a queen in one of the squares.

Here is our code for setting up the chess board:
#!/usr/bin/wish

proc setup_board { } {
    # create container for the board
    frame .queens

    # loop of rows and columns
    for {set row 1} {$row <= 8} {incr row} {
        for {set column 1} {$column <= 8} {incr column} {
            # create label with a queen displayed in it
            label .queens.$column-$row -bitmap @bitmaps/q64s.bm -relief flat

            # choose a background color depending on the position of the
            # square; make the queen invisible by setting the foreground
            # to the same color as the background
            if { [expr ($column + $row) % 2] } { 
                .queens.$column-$row config -background #ffff99
                .queens.$column-$row config -foreground #ffff99
            } else {
                .queens.$column-$row config -background #66ff99
                .queens.$column-$row config -foreground #66ff99
            }

            # place the square in a chess board grid
            grid .queens.$column-$row -row $row -column $column -padx 1 -pady 1
        }
    }

    pack .queens
}

setup_board

The first thing that happens is that a frame widget is created to contain the board. Then there are two nested loops that loop over the rows and columns of the chess board. Inside the loop, the first thing that happens is that a label widget is created. It is named using the row and column variables so that it can be easily referenced later. The label will not be used to display text but to display an image, a bitmap of a queen. The label creation command therefore has the special argument -bitmap @q64s.bm, which says that the label will display the bitmap loaded from the file q64s.bm.

The label with the queen displayed in it has now been created. The next thing that happens is that the background color of the label (square) is chosen. Depending on the position of the square it becomes either a “black” or a “white” square. At the same time, the foreground color is set to the background color. This is so that the queen (displayed in the foreground color) will be invisible, at least when the board is first displayed.
The final action in the loop is to place the label (square) in relation to all the other squares for display. A chess board is a simple grid of squares, and so this is most easily done through the grid geometry manager.

After the board has been set up square-by-square it still needs to be displayed, which is done by pack-ing the outermost frame widget.

To create and display a chess board widget, all that is needed is to call the procedure

```proc setup_board
```

which creates the chess board widget.

Once the chess board has been displayed, we need to be able to take a solution, a list of rows ordered by column, and place queens in the positions indicated.

Taking a topdown approach, our procedure for taking a solution and displaying is as follows:

```proc show_solution { solution } {
    clear_board
    set column 1
    foreach row $solution {
        place_queen $column $row
        incr column
    }
}
```

This takes a solution in solution, clears the board of all queens, and then places each queen from the solution on the board.

Next we will handle clearing the board:
The procedure `clear_board` clears the board of queens by calling the procedure `reset_column` for each of the 8 columns on a board. `reset_column` goes through each square of a column and sets the square to `off` through `set_queens`. In turn, `set_queens` sets the foreground color of a square to black if the square is turned `on`, thus revealing the queen bitmap, or sets the foreground color of a square to its background color, thus making the queens invisible, if it is called with something other than `on`.

That handles clearing the board, clearing a column or turning a queen on or off on a particular square.

The final part is `place_queen`:

```procedure
proc place_queen { column row } {
    reset_column $column
    set_queens $column $row on
}
```

This resets a column so that all queens on it are invisible and then sets the square with coordinates given in `row` and `column` to on.

A typical call would be:

```procedure
show_solution "1 2 3 4 5 6 7 6 8"
```
which would display queens along a diagonal. (This is of course not a solution to the 8-queens problem. This Tcl/Tk code only displays possible queens solutions; it does not check if the solution is valid. Later we will combine this Tcl/Tk display code with Prolog code for generating solutions to the 8-queens problem.)

10.45.4 The Tcl/Tk Prolog Library

Now we have covered the wonders of Tcl/Tk, we come to the real meat of the tutorial: how to couple the power of Tcl/Tk with the power of SICStus Prolog.

Tcl/Tk is included in SICStus Prolog by loading a special library. The library provides a bidirectional interface between Tcl/Tk and Prolog.
10.45.4.1 How it Works - An Overview

Before describing the details of the Tcl/Tk library we will give an overview of how it works with the Prolog system.

The Tcl/Tk library provides a loosely coupled integration of Prolog and Tcl/Tk. By this we mean that the two systems, Prolog and Tcl/Tk, although joined through the library, are mostly separate; Prolog variables have nothing to do with Tcl variables, Prolog and Tcl program states are separate, and so on.

The Tcl/Tk library extends Prolog so that Prolog can create a number of independent Tcl interpreters with which it can interact. Basically, there is a predicate, which when executed creates a Tcl interpreter and returns a handle with which Prolog can interact with the interpreter.

Prolog and a Tcl interpreter interact, and so communicate and cooperate, through two ways:

1. One system evaluates a code fragment in the other system and retrieves the result. For example, Prolog evaluates a Tcl code fragment in an attached Tcl interpreter and gets the result of the evaluation in a Prolog variable. Similarly, a Tcl interpreter can evaluate a Prolog goal and get the result back through a Tcl variable.

   This is synchronous communication in that the caller waits until the callee has finished their evaluation and reads the result.

2. One system passing a “message” to the other on an “event” queue.

   This is asynchronous communication in that the receiver of the message can read the message whenever it likes, and the sender can send the message without having to wait for a reply.

The Tk part of Tcl/Tk comes in because an attached Tcl interpreter may be extended with the Tk widget set and so be a Tcl/Tk interpreter. This makes it possible to add GUIs to a Prolog application: the application loads the Tcl/Tk Prolog library, creates a Tcl/Tk interpreter, and sends commands to the interpreter to create a Tk GUI. The user interacts with the GUI and therefore with the underlying Prolog system.

There are two main ways to partition the Tcl/Tk library functions: by function, i.e. the task they perform; or by package, i.e. whether they are Tcl, Tk, or Prolog functions. We will describe the library in terms of the former because it fits in with the tutorial style better, but at the end is a summary section that summarizes the library functions both ways.

Taking the functional approach, the library can be split into six function groups:

- basic functions
  - loading the library
  - creating and destroying Tcl and Tcl/Tk interpreters
- evaluation functions
  - evaluating Tcl expressions from Prolog
  - evaluating Prolog expressions from Tcl
• Prolog event functions
  - handling the Prolog/Tcl event queue
• Tk event handling
• passing control to Tk
• housekeeping functions

We go through each group in turn.

10.45.4.2 Basic Functions

The heart of the system is the ability to create an embedded Tcl interpreter with which the Prolog system can interact. A Tcl interpreter is created within Prolog through a call to \texttt{tcl\_new/1}:

\begin{verbatim}
tcl_new(-TclInterpreter)
\end{verbatim}

which creates a new interpreter, initializes it, and returns a reference to it in the variable \texttt{TclInterpreter}. The reference can then be used in subsequent calls to manipulate the interpreter. More than one Tcl interpreter object can be active in the Prolog system at any one time.

To start a Tcl interpreter extended with Tk, the \texttt{tk\_new/2} predicate is called from Prolog. It has the following form:

\begin{verbatim}
tk_new(+Options, -TclInterpreter)
\end{verbatim}

which returns through the variable \texttt{TclInterpreter} a handle to the underlying Tcl interpreter. The usual Tcl/Tk window pops up after this call is made and it is with reference to that window that subsequent widgets are created. As with the \texttt{tcl\_new/1} predicate, many Tcl/Tk interpreters may be created from Prolog at the same time through calls to \texttt{tk\_new/2}.

The \texttt{Options} part of the call is a list of some (or none) of the following elements:

\textbf{top\_level\_events}
This allows Tk events to be handled while Prolog is waiting for terminal input; for example, while the Prolog system is waiting for input at the top-level prompt. Without this option, Tk events are not serviced while the Prolog system is waiting for terminal input. (For information on Tk events; see Section 10.45.3.6 [Event Handling], page 818).

\textbf{name(+ApplicationName)}
This gives the main window a title \textit{ApplicationName}. This name is also used for communicating between Tcl/Tk applications via the Tcl \texttt{send} command. (\texttt{send} is not covered in this document. Please refer to the Tcl/Tk documentation.)

\textbf{display(+Display)}
(This is X windows specific.) Gives the name of the screen on which to create the main window. If this is not given, then the default display is determined by the \texttt{DISPLAY} environment variable.
An example of using `tk_new/2`:

\[
?- \text{tk\_new([top\_level\_events, name('My SICStus/Tk App')], Tcl)}.
\]

which creates a Tcl/Tk interpreter, returns a handle to it in the variable Tcl and Tk events
are serviced while Prolog is waiting at the top-level prompt. The window that pops up will
have the title My SICStus/Tk App.

The reference to a Tcl interpreter returned by a call to `tk_new/2` is used in the same way
and in the same places as a reference returned by a call to `tcl_new/1`. They are both
references to Tcl interpreters.

To remove a Tcl interpreter from the system, use the `tcl_delete/1` predicate:

\[
\text{tcl\_delete(+TclInterpreter)}
\]

which given a reference to a Tcl interpreter, closes down the interpreter and removes it.
The reference can be for a plain Tcl interpreter or for a Tk enhanced one; `tcl_delete/1`
removes both kinds.

### 10.45.4.3 Evaluation Functions

There are two functions in this category: Prolog extended to be able to evaluate Tcl ex-
pressions in a Tcl interpreter; Tcl extended to be able to evaluate a Prolog expression in
the Prolog system.

There is a mechanism for describing Tcl commands in Prolog as Prolog terms. This is used
in two ways: firstly, to be able to represent Tcl commands in Prolog so that they can be
subsequently passed to Tcl for evaluation; and secondly for passing terms back from Tcl to
Prolog by doing the reverse transformation.

Why not represent a Tcl command as a simple atom or string? This can indeed be done,
but commands are often not static and each time they are called require slightly different
parameters. This means constructing different atoms or strings for each command in Prolog,
which are expensive operations. A better solution is to represent a Tcl command as a
Prolog term, something that can be quickly and efficiently constructed and stored by a
Prolog system. Variable parts to a Tcl command (for example command arguments) can
be passed in through Prolog variables.

In the special command format, a Tcl command is specified as follows.

\[
\text{Command ::= Name}
| \text{codes(code list)}
| \text{write(term)}
| \text{writeq(term)}
| \text{write_canonical(term)}
| \text{format(Fmt, Args)}
| \text{dq(Command)}
| \text{br(Command)}
| \text{sqb(Command)}
\]
\[\text{min(Command)} \quad \text{dot(ListOfNames)} \quad \text{list(ListOfCommands)} \]
\[
\text{Fmt} := \text{atom} \\
\text{Name} := \text{atom} \quad \{ \text{other than []} \} \\
\quad \text{number} \\
\text{ListOfCommands} := [] \\
\quad [\text{Command} \mid \text{ListOfCommands}] \\
\text{ListOfNames} := [] \\
\quad [\text{Name} \mid \text{ListOfNames}] \\
\text{Args} := [] \\
\quad [\text{term} \mid \text{Args}] \\
\]

where

\text{Atom} \quad \text{Number} \quad \text{denote their printed representations}

\text{codes(PrologString)}
\quad \text{denotes the string represented by PrologString (a code list)}

\text{write(Term)}
\quad \text{writeq(Term)}
\quad \text{write_canonical(Term)}
\quad \text{denotes the string that is printed by the corresponding built-in predicate.}

\text{nPlease note: In general it is not possible to reconstruct Term from the string printed by write/1. If Term will be passed back into Prolog, then it therefore safest to use write_canonical(Term).}

\text{format(Fmt, Args)}
\quad \text{denotes the string that is printed by the corresponding built-in predicate}

\text{dq(Command)}
\quad \text{denotes the string specified by Command, enclosed in double quotes}

\text{br(Command)}
\quad \text{denotes the string specified by Command, enclosed in curly brackets}

\text{sqb(Command)}
\quad \text{denotes the string specified by Command, enclosed in square brackets}

\text{min(Command)}
\quad \text{denotes the string specified by Command, immediately preceded by a hyphen}

\text{dot(ListOfName)}
\quad \text{denotes the widget path specified by ListOfName, preceded by and separated by dots}

\text{list(ListOfCommands)}
\quad \text{denotes the TCL list with one element for each element in ListOfCommands. This differs from just using ListOfCommands or br(ListOfCommands) when}
any of the elements contains spaces, braces or other characters treated specially by TCL.

**ListOfCommands**

denotes the string denoted by each element, separated by spaces. In many cases list(ListOfCommands) is a better choice.

Examples of command specifications and the resulting Tcl code:

```
[set, x, 32]
 ⇒ set x 32

[set, x, br([a, b, c])]
 ⇒ set x {a b c}

[dot([panel, value_info, name]), configure, min(text), br(write('$display'/1))]
 ⇒ .panel.value_info.name configure -text {$display/1

['foo bar', baz]
 ⇒ foo bar baz

list(['foo bar', bar])
 ⇒ {foo bar} baz

list(['foo { bar', bar])
 ⇒ foo\ { \bar baz
```

Prolog calls Tcl through the predicate tcl_eval/3, which has the following form:

```
tcl_eval(+TclInterpreter, +Command, -Result)
```

which causes the interpreter TclInterpreter to evaluate the Tcl command Command and return the result Result. The result is a string (a code list) that is the usual return string from evaluating a Tcl command. Command is not just a simple Tcl command string (although that is a possibility) but a Tcl command represented as a Prolog term in the special Command Format (see Section 10.45.4.3 [Evaluation Functions], page 829).

Through tcl_eval/3, Prolog has a method of synchronous communication with an embedded Tcl interpreter and a way of manipulating the state of the interpreter.

An example:

```
| ?- tcl_new(Interp),
   tcl_eval(Interp, 'set x 1', _),
   tcl_eval(Interp, 'incr x', R).
```

which creates a Tcl interpreter the handle of which is stored in the variable Interp. Then variable x is set to the value "1" and then variable x is incremented and the result returned in R as a string. The result will be "2". By evaluating the Tcl commands in separate
tcl_eval/3 calls, we show that we are manipulating the state of the Tcl interpreter and that it remembers its state between manipulations.

It is worth mentioning here also that because of the possibility of the Tcl command causing an error to occur in the Tcl interpreter, two new exceptions are added by the tcltk library:

\[
\begin{align*}
tcl\_error(\text{Goal}, \text{Message}) \\
tk\_error(\text{Goal}, \text{Message})
\end{align*}
\]

where Message is a code list detailing the reason for the exception. Also two new user:portray_message/2 rules are provided so that any such uncaught exceptions are displayed at the Prolog top level as

\[
\begin{align*}
[\text{TCL ERROR: Goal - Message}] \\
[\text{TK ERROR: Goal - Message}]
\end{align*}
\]

respectively.

These exception conditions can be raised/caught/displayed in the usual way through the built-in predicates raise_exception/3, on_exception/1, and portray_message/2.

As an example, the following Prolog code will raise such an exception:

\[
| ?- tcl\_new(X), tcl\_eval(X, 'wilbert', R).
\]

which causes a tcl_error/2 exception and prints the following:

\[
\{\text{TCL ERROR: tcl\_eval/3 - invalid command name "wilbert"}\}
\]

assuming that there is no command or procedure defined in Tcl called wilbert.

The Tcl interpreters created through the SICStus Prolog Tcl/Tk library have been extended to allow calls to the underlying Prolog system.

To evaluate a Prolog expression in the Prolog system from a Tcl interpreter, the new prolog Tcl command is invoked. It has the following form:

\[
\text{prolog PrologGoal}
\]

where PrologGoal is the printed form of a Prolog goal. This causes the goal to be executed in Prolog. It will be executed in the user module unless it is prefixed by a module name. Execution is always determinate.

The return value of the command either of the following:

"1" if execution succeeded,
"0" if execution failed.

If succeeded (and "1" was returned), then any variable in PrologGoal that has become bound to a Prolog term will be returned to Tcl in the Tcl array named prolog_variables.
with the variable name as index. The term is converted to Tcl using the same conversion as used for Tcl commands (see Section 10.45.4.3 [Evaluation Functions], page 829). As a special case the values of unbound variables and variables with names starting with ‘_’, are not recorded and need not conform to the special command format, this is similar to the treatment of such variables by the Prolog top level.

An example:

```prolog
test_callback(Result) :-
    tcl_new(Interp),
    tcl_eval(Interp,
        'if {[[prolog "foo(X,Y,Z)"] == 1} \n
        {list $prolog_variables(X) \n
        $prolog_variables(Y) \n
        $prolog_variables(Z)}',
        Result),
    tcl_delete(Interp).

foo(1, bar, [a, b, c]).
```

When called with the query:

```
| ?- test_callback(Result).
```

will succeed, binding the variable `Result` to:

"1 bar {a b c}"

This is because execution of the `tcl_eval/3` predicate causes the execution of the `prolog` command in Tcl, which executes `foo(X, Y, Z)` in Prolog making the following bindings:

- \( X = 1 \)
- \( Y = \text{bar} \)
- \( Z = [a, b, c] \)

The bindings are returned to Tcl in the associative array `prolog_variables` where `prolog_variables(X)` is "1", `prolog_variables(Y)` is "bar", and `prolog_variables(Z)` is "a b c". Then Tcl goes on to execute the `list` command as

```
list "1" "bar" "a b c"
```

which returns the result

"1 bar {a b c}"

(remember: nested lists magically get represented with curly brackets) which is the string returned in the `Result` part of the Tcl call, and is ultimately returned in the `Result` variable of the top-level call to `test_callback(Result)`.

If an error occurs during execution of the `prolog` Tcl command, then a `tcl_error/2` exception will be raised. The message part of the exception will be formed from the string ‘Exception during Prolog execution: ’ appended to the Prolog exception message. An example is the following:

```
| ?- tcl_new(T), tcl_eval(T, 'prolog wilbert', R).
```
which will print

```prolog
{TCL ERROR: tcl_eval/3 - Exception during Prolog execution:
   wilbert  existence_error(wilbert,0,procedure,user:wilbert/0,0)}
```

at the Prolog top-level, assuming that the predicate wilbert/0 is not defined on the Prolog side of the system. (This is a tcl_error exception containing information about the underlying exception, an existence_error exception, which was caused by trying to execute the non-existent predicate wilbert.)

### 10.45.4.4 Event Functions

Another way for Prolog to communicate with Tcl is through the predicate tcl_event/3:

```prolog
tcl_event(+TclInterpreter, +Command, -Events)
```

This is similar to tcl_eval/3 in that the command Command is evaluated in the Tcl interpreter TclInterpreter, but the call returns a list of events in Events rather than the result of the Tcl evaluation. Command is again a Tcl command represented as a Prolog term in the special Command Format described previously (see Section 10.45.4.3 [Evaluation Functions], page 829).

This begs the questions what are these events and where does the event list come from? The Tcl interpreters in the SICStus Prolog Tcl/Tk library have been extended with the notion of a Prolog event queue. (This is not available in plain standalone Tcl interpreters.) The Tcl interpreter can put events on the event queue by executing a prolog_event command. Each event is a Prolog term. So a Tcl interpreter has a method of putting Prolog terms onto a queue, which can later be picked up by Prolog as a list as the result of a call to tcl_event/3. (It may be helpful to think of this as a way of passing messages as Prolog terms from Tcl to Prolog.)

A call to tcl_event/3 blocks until there is something on the event queue.

A second way of getting Prolog events from a Prolog event queue is through the tk_next_event/[2,3] predicates. These have the form:

```prolog
tk_next_event(+TclInterpreter, -Event)
tk_next_event(+Options, +TclInterpreter, -Event)
```

where TclInterpreter reference to a Tcl interpreter and Event is the Prolog term at the head of the associated Prolog event queue. (The Options feature will be described below in the Housekeeping section when we talk about Tcl and Tk events; see Section 10.45.4.7 [Housekeeping], page 839.)

(We will meet tk_next_event/[2,3] again later when we discuss how it can be used to service Tk events; see Section 10.45.4.5 [Servicing Tk Events], page 837).

If the interpreter has been deleted, then the empty list `[]` is returned.
The Tcl interpreters under the SICStus Prolog library are extended with a command, prolog_event, for adding events to a Prolog event queue.

The prolog_event command has the following form:

    prolog_event Terms ...

where Terms are strings that contain the printed representation of Prolog terms. These are stored in a queue and retrieved as Prolog terms by tcl_event/3 or tk_next_event/[2,3] (described above).

An example of using the prolog_event command:

    test_event(Event) :-
        tcl_new(Interp),
        tcl_event(Interp, [prolog_event, dq(write(zap(42)))], Event),
        tcl_delete(Interp).

with the query:

    | ?- test_event(Event).

will succeed, binding Event to the list [zap(42)].

This is because tcl_event converts its argument using the special Command Format conversion (see Section 10.45.4.3 [Evaluation Functions], page 829), which yields the Tcl command prolog_event "zap(42)". This command is evaluated in the Tcl interpreter referenced by the variable Interp. The effect of the command is to take the string given as argument to prolog_event (in this case "zap(42)") and to place it on the Tcl to Prolog event queue. The final action of a tcl_event/3 call is to pick up any strings on the Prolog queue from Tcl, add a trailing full stop and space to each string, and parse them as Prolog terms, binding Event to the list of values, which in this case is the singleton list [zap(42)]. (The queue is a list the elements of which are terms put there through calls to prolog_event).

If any of the Term-s in the list of arguments to prolog_event is not a valid representation of a Prolog term, then an exception is raised in Prolog when it is converted from the Tcl string to the Prolog term using read. To ensure that Prolog will be able to read the term correctly it is better to always use write_canonical and to ensure that Tcl is not confused by special characters in the printed representation of the Prolog term it is best to wrap the list with list.

A safer variant that safely passes any term from Prolog via Tcl and back to Prolog is thus:

    test_event(Term, Event) :-
        tcl_new(Interp),
        tcl_event(Interp, list([prolog_event, write_canonical(Term)]), Event),
        tcl_delete(Interp).
As an example of using the Prolog event system supplied by the \texttt{tcltk} library, we will return to our 8-queens example but now approaching from the Prolog side rather than the Tcl/Tk side:

\begin{verbatim}
:- use_module(library(tcltk)).

setup :-
  tk_new([name('SICStus+Tcl/Tk - Queens')], Tcl),
  tcl_eval(Tcl, 'source queens.tcl', _),
  tk_next_event(Tcl, Event),
  ( Event = next -> go(Tcl),
    ; closedown(Tcl)
 ).

closedown(Tcl) :-
  tcl_delete(Tcl).

go(Tcl) :-
  tcl_eval(Tcl, 'clear_board', _),
  queens(8, Qs),
  show_solution(Qs, Tcl),
  tk_next_event(Tcl, Event),
  ( Event = next -> fail
    ; closedown(Tcl)
  ).

go(Tcl) :-
  tcl_eval(Tcl, 'disable_next', _),
  tcl_eval(Tcl, 'clear_board', _),
  tk_next_event(Tcl, _Event),
  closedown(Tcl).
\end{verbatim}

This is the top-level fragment of the Prolog side of the 8-queens example. It has three predicates: \texttt{setup/0}, \texttt{closedown/1}, and \texttt{go/1}. \texttt{setup/0} simply creates the Tcl interpreter, loads the Tcl code into the interpreter using a call to \texttt{tcl_eval/3} (which also initializes the display) but then calls \texttt{tk_next_event/2} to wait for a message from the Tk side.

The Tk part that sends \texttt{prolog_event}-s to Prolog looks like this:

\begin{verbatim}
button .next -text next -command \{prolog_event next\}
pack .next

button .stop -text stop -command \{prolog_event stop\}
pack .stop
\end{verbatim}

that is two buttons, one that sends the atom \texttt{next}, the other that sends the atom \texttt{stop}. They are used to get the next solution and to stop the program respectively.
So if the user presses the next button in the Tk window, then the Prolog program will receive a next atom via a prolog_event/tk_next_event pair, and the program can proceed to execute go/1.

go/1 is a failure driven loop that generates 8-queens solutions and displays them. First it generates a solution in Prolog and displays it through a tcl_eval/3 call. Then it waits again for a Prolog events via tk_next_event/2. If the term received on the Prolog event queue is next, then corresponding to the user pressing the “next solution” button, then fail is executed and the next solution found, thus driving the loop.

If the stop button is pressed, then the program does some tidying up (clearing the display and so on) and then executes closedown/1, which deletes the Tcl interpreter and the corresponding Tk windows altogether, and the program terminates.

This example fragment show how it is possible for a Prolog program and a Tcl/Tk program to communicate via the Prolog event queue.

10.45.4.5 Servicing Tcl and Tk events

The notion of an event in the Prolog+Tcl/Tk system is overloaded. We have already come across the following kinds of events:

- Tk widget events captured in Tcl/Tk through the bind command
- Prolog queue events controlled through the tcl_event/3, tk_next_event(2,3), and prolog_event functions

It is further about to be overloaded with the notion of Tcl/Tk events. It is possible to create event handlers in Tcl/Tk for reacting to other kinds of events. We will not cover them here but describe them so that the library functions are understandable and in case the user needs these features in an advanced application.

There are the following kinds of Tcl/Tk events:

- idle events happen when the Tcl/Tk system is idle
- file events happen when input arrives on a file handle that has a file event handler attached to it
- timer events happen when a Tcl/Tk timer times out
- window events when something happens to a Tk window, such as being resized or destroyed

The problem is that in advanced Tcl/Tk applications it is possible to create event handlers for each of these kinds of event, but they are not normally serviced while in Prolog code. This can result in unresponsive behavior in the application; for example, if window events are not serviced regularly, then if the user tries to resize a Tk window, then it will not resize in a timely fashion.
The solution to this is to introduce a Prolog predicate that passes control to Tk for a while so that it can process its events, \texttt{tk\_do\_one\_event/[0,1]}. If an application is unresponsive because it is spending a lot of time in Prolog and is not servicing Tk events often enough, then critical sections of the Prolog code can be sprinkled with calls to \texttt{tk\_do\_one\_event/[0,1]} to alleviate the problem.

\texttt{tk\_do\_one\_event/[0,1]} has the following forms:

\begin{verbatim}
    tk\_do\_one\_event
    tk\_do\_one\_event(+Options)
\end{verbatim}

which passes control to Tk to handle a single event before passing control back to Prolog. The type of events handled is passed through the \textit{Options} variable, a list of atoms that are event types.

The \textit{Options} list can contain the following atoms:

\begin{itemize}
    \item \texttt{tk\_dont\_wait}
        \begin{itemize}
            \item do not wait for new events, process only those that are ready
        \end{itemize}
    \item \texttt{tk\_window\_events}
        \begin{itemize}
            \item process window events
        \end{itemize}
    \item \texttt{tk\_file\_events}
        \begin{itemize}
            \item process file events
        \end{itemize}
    \item \texttt{tk\_timer\_events}
        \begin{itemize}
            \item process timer events
        \end{itemize}
    \item \texttt{tk\_idle\_events}
        \begin{itemize}
            \item process Tcl\_DoWhenIdle events
        \end{itemize}
    \item \texttt{tk\_all\_events}
        \begin{itemize}
            \item process any event
        \end{itemize}
\end{itemize}

Calling \texttt{tk\_do\_one\_event/0} is equivalent to a call to \texttt{tk\_do\_one\_event/1} with the \texttt{tk\_all\_events} and \texttt{tk\_dont\_wait} flags.

A call to either of these predicates succeeds only if an event of the appropriate type happens in the Tcl/Tk interpreter. If there are no such events, then \texttt{tk\_do\_one\_event/1} will fail if the \texttt{tk\_dont\_wait} wait flag is present, as will \texttt{tk\_do\_one\_event/0}, which has that flag set implicitly.

If the \texttt{tk\_dont\_wait} flag is not set, then a call to \texttt{tk\_do\_one\_event/1} will block until an appropriate Tk event happens (in which case it will succeed).

It is straight forward to define a predicate that handles all Tk events and then returns:

\begin{verbatim}
    tk\_do\_all\_events :-
        tk\_do\_one\_event, !,
        tk\_do\_all\_events.
\end{verbatim}
The predicate \texttt{tk\_next\_event/[2,3]} is similar to \texttt{tk\_do\_one\_event/[0,1]} except that it processes Tk events until at least one Prolog event happens. (We came across this predicate before when discussing Prolog event queue predicates. This shows the overloading of the notion event where we have a predicate that handles both Tcl/Tk events and Prolog queue events.)

It has the following forms:

\begin{verbatim}
  tk_next_event(+TclInterpreter, -Event)
  tk_next_event(+Options, +TclInterpreter, -Event)
\end{verbatim}

The Prolog event is returned in the variable \textit{Event} and is the first term on the Prolog event queue associated with the interpreter \textit{TclInterpreter}. (Prolog events are initiated on the Tcl side through the new Tcl command \texttt{prolog\_event}, covered earlier; see Section 10.45.4.4 [Event Functions], page 834).

\section*{10.45.4.6 Passing Control to Tk}

There is a predicate for passing control completely over to Tk, the \texttt{tk\_main\_loop/0} command. This passes control to Tk until all windows in all the Tcl/Tk interpreters in the Prolog have have been destroyed:

\begin{verbatim}
  tk\_main\_loop
\end{verbatim}

\section*{10.45.4.7 Housekeeping functions}

Here we will described the functions that do not fit into any of the above categories and are essentially housekeeping functions.

There is a predicate that returns a reference to the main window of a Tcl/Tk interpreter:

\begin{verbatim}
  tk_main_window(+TclInterpreter, -TkWindow)
\end{verbatim}

which given a reference to a Tcl interpreter \textit{TclInterpreter}, returns a reference to its main window in \textit{TkWindow}.

The window reference can then be used in \texttt{tk\_destroy\_window/1}:

\begin{verbatim}
  tk_destroy_window(+TkWindow)
\end{verbatim}

which destroys the window or widget referenced by \textit{TkWindow} and all of its sub-widgets.

The predicate \texttt{tk\_make\_window\_exist/1} also takes a window reference:

\begin{verbatim}
  tk_make_window_exist(+TkWindow)
\end{verbatim}

which causes the window referenced by \textit{TkWindow} in the Tcl interpreter \textit{TclInterpreter} to be immediately mapped to the display. This is useful because normally Tk delays displaying new information for as long as possible (waiting until the machine is idle, for example), but using this call causes Tk to display the window immediately.
There is a predicate for determining how many main windows, and hence Tcl/Tk interpreters (excluding simple Tcl interpreters), are currently in use:

\[
\text{tk\_num\_main\_windows}(-\text{NumberOfWindows})
\]

which returns an integer in the variable \text{NumberOfWindows}.

**10.45.4.8 Summary**

The functions provided by the SICStus Prolog Tcl/Tk library can be grouped in two ways: by function, and by package.

By function, we can group them like this:

- **basic functions**
  
  \text{tcl\_new/1}
  
  create a Tcl interpreter

  \text{tcl\_delete/1}
  
  remove a Tcl interpreter

  \text{tk\_new/2}
  
  create a Tcl interpreter with Tk extensions

- **evaluation functions**

  \text{tcl\_eval/3}
  
  evaluate a Tcl expression from Prolog

  \text{prolog}
  
  evaluate a Prolog expression from Tcl

- **Prolog event queue functions**

  \text{tcl\_event/3}
  
  evaluate a Tcl expression and return a Prolog queue event list

  \text{tk\_next\_event/[2,3]}
  
  pass control to Tk until a Prolog queue event happens and return the head of the queue

  \text{prolog\_event}
  
  place a Prolog term on the Prolog event queue from Tcl

- **servicing Tcl and Tk events**

  \text{tk\_do\_one\_event/[0,1]}
  
  pass control to Tk until one Tk event is serviced

  \text{tk\_next\_event/[2,3]}
  
  also services Tk events but returns when a Prolog queue event happens and returns the head of the queue

- **passing control completely to Tk**

  \text{tk\_main\_loop/0}
  
  control passed to Tk until all windows in all Tcl/Tk interpreters are gone

- **housekeeping**

  \text{tk\_main\_window/2}
  
  return reference to main in of a Tcl/Tk interpreter
Chapter 10: The Prolog Library

**tk_destroy_window/1**
destroy a window or widget

**tk_make_window_exist/1**
force display of a window or widget

**tk_num_main_windows/1**
return a count of the total number of Tk main windows existing in the system

By package, we can group them like this:

- **predicates for Prolog to interact with Tcl interpreters**
  
  \[\text{tcl_new/1}\]
  
create a Tcl interpreter

  \[\text{tcl_delete/1}\]
  
  remove a Tcl interpreter

  \[\text{tcl_eval/3}\]
  
evaluate a Tcl expression from Prolog

  \[\text{tcl_event/3}\]
  
evaluate a Tcl expression and return a Prolog event list

- **predicates for Prolog to interact with Tcl interpreters with Tk extensions**
  
  \[\text{tk_new/2}\]
  
  create a Tcl interpreter with Tk extensions

  \[\text{tk_do_one_event/[0,1]}\]
  
  pass control to Tk until one Tk event is serviced

  \[\text{tk_next_event/[2,3]}\]
  
  also services Tk events but returns when a Prolog queue event happens and returns the head of the queue

  \[\text{tk_main_loop/0}\]
  
  control passed to Tk until all windows in all Tcl/Tk interpreters are gone

  \[\text{tk_main_window/2}\]
  
  return reference to main in of a Tcl/Tk interpreter

  \[\text{tk_destroy_window/1}\]
  
  destroy a window or widget

  \[\text{tk_make_window_exist/1}\]
  
  force display of a window or widget

  \[\text{tk_num_main_windows/1}\]
  
  return a count of the total number of Tk main windows existing in the system

- **commands for the Tcl interpreters to interact with the Prolog system**
  
  \[\text{prolog}\]
  
  evaluate a Prolog expression from Tcl
In the next section we will discuss how to use the tcltk library to build graphical user interfaces to Prolog applications. More specifically we will discuss the ways in which cooperation between Prolog and Tcl/Tk can be arranged: how to achieve them, and their benefits.

10.45.5 Putting It All Together

At this point we now know Tcl, the Tk extensions, and how they can be integrated into SICStus Prolog through the tcltk library module. The next problem is how to get all this to work together to produce a coherent application. Because Tcl can make Prolog calls and Prolog can make Tcl calls it is easy to create programming spaghetti. In this section we will discuss some general principles of organizing the Prolog and Tcl code to make writing applications easier.

The first thing to do is to review the tools that we have. We have two programming systems: Prolog and Tcl/Tk. They can interact in the following ways:

- Prolog evaluates a Tcl expression in a Tcl interpreter, using tcl_eval
- Tcl evaluates a Prolog expression in the Prolog interpreter, using prolog
- Prolog evaluates a Tcl expression in a Tcl interpreter and waits for a Prolog event, using tcl_event
- Prolog waits for a Prolog event from a Tcl interpreter, using tk_next_event
- Tcl sends a Prolog predicate to Prolog on a Prolog event queue using prolog_event

With these interaction primitives there are three basic ways in which Prolog and Tcl/Tk can be organized:

1. Tcl the master, Prolog the slave: program control is with Tcl, which makes occasional calls to Prolog, through the prolog function.
2. Prolog the master, Tcl the slave: program control is with Prolog, which makes occasional call to Tcl through the tcl_eval function
3. Prolog and Tcl share control: program control is shared with Tcl and Prolog interacting via the Prolog event queue, through tcl_event, tk_next_event, and prolog_event.

These are three ways of organizing cooperation between Tcl/Tk and Prolog to produce an application. In practice an application may use only one of these methods throughout, or may use a combination of them where appropriate. We describe them here so that the developer can see the different patterns of organization and can pick those relevant to their application.

10.45.5.1 Tcl The Master, Prolog The Slave

This is the classical way that GUIs are bolted onto applications. The slave (in this case Prolog) sits mostly idle while the user interacts with the GUI, for example filling in a form. When some action happens in the GUI that requires information from the slave (a form
submit, for example), the slave is called, performs a calculation, and the GUI retrieves the result and updates its display accordingly.

In our Prolog+Tcl/Tk setting this involves the following steps:

- start Prolog and load the Tcl/Tk library
- load Prolog application code
- start a Tcl/Tk interpreter through \texttt{tk\_new/2}
- set up the Tk GUI through calls to \texttt{tcl\_eval/3}
- pass control to Tcl/Tk through \texttt{tk\_main\_loop}

Some of The Tk widgets in the GUI will have “callbacks” to Prolog, i.e. they will call the \texttt{prolog} Tcl command. When the Prolog call returns, the values stored in the \texttt{prolog\_variables} array in the Tcl interpreter can then be used by Tcl to update the display.

Here is a simple example of a callback. The Prolog part is this:

\begin{verbatim}
:- use\_module(library(tcltk)).

hello('world').

go :-
    tk\_new([], Tcl),
    tcl\_eval(Tcl, 'source simple.tcl', _),
    tk\_main\_loop.
\end{verbatim}

which just loads the \texttt{library(tcltk)}, defines a \texttt{hello/1} data clause, and \texttt{go/0}, which starts a new Tcl/Tk interpreter, loads the code \texttt{simple.tcl} into it, and passes control to Tcl/Tk.

The Tcl part, \texttt{simple.tcl}, is this:

\begin{verbatim}
label .l -textvariable tvar
button .b -text "push me" -command { call\_and\_display }
pack .l .b -side top

proc call\_and\_display {} {
    global tvar

    prolog "$\text{hello}(X)$"
    set tvar $\text{prolog\_variables}(X)$
}
\end{verbatim}

which creates a label, with an associated text variable, and a button, that has a call back procedure, \texttt{call\_and\_display}, attached to it. When the button is pressed, \texttt{call\_and\_display} is executed, which simply evaluates the goal \texttt{hello(X)} in Prolog and the text variable of the label \texttt{.l} to whatever \texttt{X} becomes bound to, which happens to be ‘world’. In short, pressing the button causes the word ‘world’ to be displayed in the label.
Having Tcl as the master and Prolog as the slave, although a simple model to understand and implement, does have disadvantages. The Tcl command prolog is determinate, i.e. it can return only one result with no backtracking. If more than one result is needed, then it means either performing some kind of all-solutions search and returning a list of results for Tcl to process, or asserting a clause into the Prolog clause store reflecting the state of the computation.

Here is an example of how an all-solutions search can be done. It is a program that calculates the outcome of certain ancestor relationships; i.e. enter the name of a person, click on a button and it will tell you the mother, father, parents or ancestors of that person.

The Prolog portion looks like this (see also library('tcltk/examples/ancestors.pl')):

```prolog
:- use_module(library(tcltk)).

go :- tk_new([name('ancestors')], X),
     tcl_eval(X, 'source ancestors.tcl', _),
     tk_main_loop,
     tcl_delete(X).

father(ann, fred).
father(fred, jim).
mother(ann, lynn).
mother(fred, lucy).
father(jim, sam).

parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

all_ancestors(X, Z) :- findall(Y, ancestor(X, Y), Z).
all_parents(X, Z) :- findall(Y, parent(X, Y), Z).
```

This program consists of three parts. The first part is defined by go/0, the now familiar way in which a Prolog program can create a Tcl/Tk interpreter, load a Tcl file into that interpreter, and pass control over to the interpreter.

The second part is a small database of mother/father relationships between certain people through the clauses mother/2 and father/2.

The third part is a set of “rules” for determining certain relationships between people: parent/2, ancestor/2, all_ancestors/2 and all_parents/2.

The Tcl part looks like this (see also library('tcltk/examples/ancestors.tcl')):
Chapter 10: The Prolog Library

% ancestors.pl

#!/usr/bin/wish

# set up the tk display

# construct text filler labels
label .search_for -text "SEARCHING FOR THE" -anchor w
label .of -text "OF" -anchor w
label .gives -text "GIVES" -anchor w

# construct frame to hold buttons
frame .button_frame

# construct radio button group
radiobutton .mother -text mother -variable type -value mother
radiobutton .father -text father -variable type -value father
radiobutton .parents -text parents -variable type -value parents
radiobutton .ancestors -text ancestors -variable type -value ancestors

# add behaviors to radio buttons
.mother config -command { one_solution mother $name}
.father config -command { one_solution father $name}
.parents config -command { all_solutions all_parents $name}
.ancestors config -command { all_solutions all_ancestors $name}

# create entry box and result display widgets
entry .name -textvariable name
label .result -text ">>> result <<<" -relief sunken -anchor nw -justify left

# pack buttons into button frame
pack .mother .father .parents .ancestors -fill x -side left -in .button_frame

# pack everything together into the main window
pack .search_for .button_frame .of .name .gives .result -side top -fill x

# now everything is set up
% ancestors.pl

# defined the callback procedures

# called for one solution results
proc one_solution { type name } {  
    if [prolog "${type}('$name', R)"] {  
        display_result $prolog_variables(R)  
    } else {  
        display_result ""  
    }  
}

# called for all solution results
proc all_solutions { type name } {  
    prolog "${type}('$name', R)"  
    display_result $prolog_variables(R)  
}

# display the result of the search in the results box
proc display_result { result } {  
    if { $result != "" } {  
        # create a multiline result  
        .result config -text $result  
    } else {  
        .result config -text "*** no result ***"  
    }  
}

Ancestors Calculator

This program is in two parts. The first part sets up the Tk display, which consists of four radiobuttons to choose the kind of relationship we want to calculate, an entry box to put
the name of the person we want to calculate the relationship over, and a label in which to display the result.

Each radio buttons has an associated callback. Clicking on the radio button will invoke the appropriate callback, apply the appropriate relationship to the name entered in the text entry box, and display the result in the results label.

The second part consists of the callback procedures themselves. There are actually just two of them: one for a single solution calculation, and one for an all-solutions calculation. The single solution callback is used when we want to know the mother or father as we know that a person can have only one of each. The all-solutions callback is used when we want to know the parents or ancestors as we know that these can return more than one results. (We could have used the all-solutions callback for the single solutions cases too, but we would like to illustrate the difference in the two approaches.) There is little difference between the two approaches, except that in the single solution callback, it is possible that the call to Prolog will fail, so we wrap it in an if ... else construct to catch this case. An all-solutions search, however, cannot fail, and so the if ... else is not needed.

But there are some technical problems too with this approach. During a callback Tk events are not serviced until the callback returns. For Prolog callbacks that take a very short time to complete this is not a problem, but in other cases, for example during a long search call when the callback takes a significant time to complete, this can cause problems. Imagine that, in our example, we had a vast database describing the parent relationships of millions of people. Performing an all-solutions ancestors search could take a long time. The classic problem is that the GUI no longer reacts to the user until the callback completes.

The solution to this is to sprinkle tk_do_one_event/[0,1] calls throughout the critical parts of the Prolog code, to keep various kinds of Tk events serviced.

If this method is used in its purest form, then it is recommended that after initialization and passing of control to Tcl, Prolog do not make calls to Tcl through tcl_eval/3. This is to avoid programming spaghetti. In the pure master/slave relationship it is a general principle that the master only call the slave, and not the other way around.

10.45.5.2 Prolog The Master, Tk The Slave

The second approach is to have Prolog be the master and Tk the slave. This is suitable when heavy processing is done in the Prolog code and Tk is used mostly to display the state of the computation in some way rather than as a traditional GUI; i.e. during computation Prolog often makes calls to Tk to show some state, but the user rarely interacts with the application.

In our Prolog+Tcl/Tk setting this involves the following steps:

- start Prolog and load the Tcl/Tk library
- load Prolog application code
- start a Tcl/Tk interpreter through tk_new/2
- set up the Tk GUI through calls to tcl_eval/3
- Prolog calls tcl_eval to update the Tk display
- values are passed to Prolog through the Result string of tcl_eval

Again it its purest form, Prolog makes calls to Tcl, but Tcl does not make calls to Prolog. The result of a call to Tcl is either passed back through the Result variable of a tcl_eval/3 call.

A good example of this is the Tcl/Tk display for our 8-queens problem, that we saw earlier; see Section 10.45.3.9 [Queens Display], page 822.

We will now fill out the example by presenting the Prolog master part. The Prolog program calculates a solution to the 8-queens problem and then makes calls Tcl/Tk to display the solution. In this way Tcl/Tk is the slave, just being used as a simple display.

We have already seen the Tcl/Tk part, but here is the Prolog part for generating a solution and displaying it:
:- use_module(library(tcltk)).
:- use_module(library(lists)).

go :-
    tk_new([name('SICStus+Tcl/Tk - Queens')], Tcl),
    tcl_eval(Tcl, 'source queens.tcl', _),
    tk_next_event(Tcl, Event),
    queens(8, Qs),
    reverse(L, LR),
    tcl_eval(Tcl, [show_solution, br(LR)], _),
    fail.
go.

queens(N, Qs) :-
    range(1, N, Ns),
    queens(Ns, [], Qs).

queens(UnplacedQs, SafeQs, Qs) :-
    select(Q, UnplacedQs, UnplacedQs1),
    \+ attack(Q, SafeQs),
    queens(UnplacedQs1, [Q|SafeQs], Qs).
    queens([], Qs, Qs).

attack(X, Xs) :- attack(X, 1, Xs).

attack(X, N, [Y|_Ys]) :- X is Y + N.
attack(X, N, [Y|_Ys]) :- X is Y - N.
attack(X, N, [_Y|Ys]) :-
    N1 is N + 1,
    attack(X, N1, Ys).

range(M, N, [M|Ns]) :-
    M < N,
    M1 is M + 1,
    range(M1, N, Ns).
range(N, N, [N]).

:- go.

All this simply does it to create a Tcl/Tk interpreter, load the Tcl code for displaying
queens into it, generate a solution to the 8-queens problem as a list of integers, and then
calls show_solution/2 in the Tcl interpreter to display the solution. At the end of first
clause for go/0 is a fail clause that turns go/0 into a failure driven loop. The result of this
is that the program will calculate all the solutions to the 8-queens problem, displaying them
rapidly one after the other, until there are none left.
10.45.5.3 Prolog And Tcl Interact through Prolog Event Queue

In the previous two methods, one of the language systems was the master and the other slave, the master called the slave to perform some action or calculation, the slave sits waiting until the master calls it. We have seen that this has disadvantages when Prolog is the slave in that the state of the Prolog call is lost. Each Prolog call starts from the beginning unless we save the state using message database manipulation through calls to assert and retract.

Using the Prolog event queue, however, it is possible to get a more balanced model where the two language systems cooperate without either really being the master or the slave.

One way to do this is the following:

- Prolog is started
- load Tcl/Tk library
- load and set up the Tcl side of the program
- Prolog starts a processing loop
- it periodically checks for a Prolog event and processes it
- Prolog updates the Tcl display through tcl_eval calls

What can processing a Prolog event mean? Well, for example, a button press from Tk could tell the Prolog program to finish or to start processing something else. The Tcl program is not making an explicit call to the Prolog system but sending a message to Prolog. The Prolog system can pick up the message and process it when it chooses, in the meantime keeping its run state and variables intact.

To illustrate this, we return to the 8-queens example. If Tcl/Tk is the master and Prolog the slave, then we have shown that using a callback to Prolog, we can imagine that we hit a button, call Prolog to get a solution and then display it. But how do we get the next solution? We could get all the solutions, and then use Tcl/Tk code to step through them, but that does not seem satisfactory. If we use the Prolog is the master and Tcl/Tk is the slave model, then we have shown how we can use Tcl/Tk to display the solutions generate from the Prolog side: Prolog just make a call to the Tcl side when it has a solution. But in this model Tcl/Tk widgets do not interact with the Prolog side; Tcl/Tk is merely an add-on display to Prolog.

But using the Prolog event queue we can get the best of both worlds: Prolog can generate each solution in turn as Tcl/Tk asks for it.

Here is the code on the Prolog side that does this. (We have left out parts of the code that have not changed from our previous example, see Section 10.45.3.9 [Queens Display], page 822).
:- use_module(library(tcltk)).
:- use_module(library(lists)).

setup :-
tk_new([name('SICStus+Tcl/Tk - Queens')], Tcl),
tcl_eval(Tcl, 'source queens2.tcl', _),
tk_next_event(Tcl, Event),
( Event = next -> go(Tcl)
    ; closedown(Tcl)
).

closedown(Tcl) :-
tcl_delete(Tcl).

go(Tcl) :-
tcl_eval(Tcl, 'clear_board', _),
queens(8, Qs),
show_solution(Qs),
tk_next_event(Tcl, Event),
( Event = next -> fail
    ; closedown(Tcl)
).

go(Tcl) :-
tcl_eval(Tcl, 'disable_next', _),
tcl_eval(Tcl, 'clear_board', _),
tk_next_event(Tcl, _Event),
closedown(Tcl).

show_solution(Tcl, L) :-
tcl(Tcl),
reverse(L, LR),
tcl_eval(Tcl, [show_solution, br(LR)], _),
tk_do_all_events.

Notice here that we have used \texttt{tk\_next\_event/2} in several places. The code is executed by calling \texttt{setup/0}. As usual, this loads in the Tcl part of the program, but then Prolog waits for a message from the Tcl side. This message can either be \texttt{next}, indicating that we want to show the next solution, or \texttt{stop}, indicating that we want to stop the program.

If \texttt{next} is received, then the program goes on to execute \texttt{go/1}. What this does it to first calculate a solution to the 8-queens problem, displays the solution through \texttt{show\_solution/2}, and then waits for another message from Tcl/Tk. Again this can be either \texttt{next} or \texttt{stop}. If \texttt{next}, then the program goes into the failure part of a failure driven loop and generates and displays the next solution.

If at any time \texttt{stop} is received, then the program terminates gracefully, cleaning up the Tcl interpreter.
On the Tcl/Tk side all we need are a couple of buttons: one for sending the next message, and the other for sending the stop message.

```tcl
button .next -text next -command {prolog_event next}
pack .next

button .stop -text stop -command {prolog_event stop}
pack .stop
```

(We could get more sophisticated. We might want it so that when the button it is depressed until Prolog has finished processing the last message, when the button is allowed to pop back up. This would avoid the problem of the user pressing the button many times while the program is still processing the last request. We leave this as an exercise for the reader.)

### 10.45.5.4 The Whole 8-Queens Example

To finish off, we our complete 8-queens program.

Here is the Prolog part, which we have covered in previous sections. The code is in `library('tcltk/examples/8-queens.pl')`:

```prolog
:- use_module(library(tcltk)).
:- use_module(library(lists)).

setup :-
    tk_new([name('SICStus+Tcl/Tk - Queens')], Tcl),
    tcl_eval(Tcl, 'source 8-queens.tcl', _),
    tk_next_event(Tcl, Event),
    ( Event = next -> go(Tcl)
    ;   closedown(Tcl)
    ).

closedown(Tcl) :-
    tcl_delete(Tcl).

go(Tcl) :-
    tcl_eval(Tcl, 'clear_board', _),
    queens(8, Qs),
    show_solution(Tcl,Qs),
    tk_next_event(Tcl, Event),
    ( Event = next -> fail
    ;   closedown(Tcl)
    ).
go(Tcl) :-
    tcl_eval(Tcl, 'disable_next', _),
    tcl_eval(Tcl, 'clear_board', _),
    tk_next_event(Tcl, _Event),
closedown(Tcl).
```

% 8-queens.pl
% 8-queens.pl

queens(N, Qs) :-
    range(1, N, Ns),
    queens(Ns, [], Qs).

queens(UnplacedQs, SafeQs, Qs) :-
    select(Q, UnplacedQs, UnplacedQs1),
    \+ attack(Q, SafeQs),
    queens(UnplacedQs1, [Q|SafeQs], Qs).
    queens([], Qs, Qs).

attack(X, Xs) :- attack(X, 1, Xs).

attack(X, N, [Y|_Ys]) :- X is Y + N.
attack(X, N, [Y|_Ys]) :- X is Y - N.
attack(X, N, [Y|_Ys]) :-
    N1 is N + 1,
    attack(X, N1, Ys).

range(M, N, [M|Ns]) :-
    M < N,
    M1 is M + 1,
    range(M1, N, Ns).
range(N, N, [N]).

show_solution(Tcl, L) :-
    reverse(L, LR),
    tcl_eval(Tcl, [show_solution, br(LR)], _),
    tk_do_all_events.

tk_do_all_events :-
    tk_do_one_event, !,
    tk_do_all_events.

:- setup.

And here is the Tcl/Tk part, which we have covered in bits and pieces but here is the whole thing. We have added an enhancement where when the mouse is moved over one of the queens, the squares that the queen attacks are highlighted. Move the mouse away and the board reverts to normal. This is an illustration of how the Tcl/Tk bind feature can be used. The code is in library('tcltk/examples/8-queens.tcl'):
#!/usr/bin/wish
# create an 8x8 grid of labels
proc setup_display { } {
    frame .queens -background black
    pack .queens

    for {set y 1} {$y <= 8} {incr y} {
        for {set x 1} {$x <= 8} {incr x} {

            # create a label and display a queen in it
            label .queens.$x-$y -bitmap @bitmaps/q64s.bm -relief flat

            # color alternate squares with different colors
            # to create the chessboard pattern
            if { [expr ($x + $y) % 2] } {
                .queens.$x-$y config -background #ffff99
            } else {
                .queens.$x-$y config -background #66ff99
            }

            # set foreground to the background color to
            # make queen image invisible
            .queens.$x-$y config -foreground [.queens.$x-$y cget -background]

            # bind the mouse to highlight the squares attacked by a
            # queen on this square
            bind .queens.$x-$y <Enter> "highlight_attack on $x $y"
            bind .queens.$x-$y <Leave> "highlight_attack off $x $y"

            # arrange the queens in a grid
            grid .queens.$x-$y -row $y -column $x -padx 1 -pady 1
        }
    }
}
# 8-queens.tcl

```
# clear a whole column
proc reset_column { column } {
    for {set y 1} {$y <= 8} {incr y} {
        set_queens $column $y ""
    }
}

# place or unplace a queen
proc set_queens { x y v } {
    if { $v == "Q" } {
        .queens.$x-$y config -foreground black
    } else {
        .queens.$x-$y config -foreground [.queens.$x-$y cget -background]
    }
}

# place a queen on a column
proc place_queen { x y } {
    reset_column $x
    set_queens $x $y Q
}

# clear the whole board by clearing each column in turn
proc clear_board { } {
    for { set x 1 } {$x <= 8} {incr x} {
        reset_column $x
    }
}

# given a solution as a list of queens in column positions
# place each queen on the board
proc show_solution { solution } {
    clear_board
    set x 1
    foreach y $solution {
        place_queen $x $y
        incr x
    }
}
```
# 8-queens.tcl

proc highlight_square { mode x y } {
    # check if the square we want to highlight is on the board
    if { $x < 1 || $y < 1 || $x > 8 || $y > 8 } { return };

    # if turning the square on make it red,
    # otherwise determine what color it should be and set it to that
    if { $mode == "on" } { set color red } else {
        if { ![expr ($x + $y) % 2] } { set color "#ff99ff" } else {
            set color "#66ff99"
        }
    }

    # get the current settings
    set bg [.queens.$x-$y cget -bg ]
    set fg [.queens.$x-$y cget -fg ]

    # if the current foreground and background are the same
    # there is no queen there
    if { $bg == $fg } {
        # no queens
        .queens.$x-$y config -bg $color -fg $color
    } else {
        .queens.$x-$y config -bg $color
    }
}

proc highlight_attack { mode x y } {
    # get current colors of square at x y
    set bg [.queens.$x-$y cget -bg ]
    set fg [.queens.$x-$y cget -fg ]

    # no queen there, give up
    if { $bg == $fg } { return };

    # highlight the square the queen is on
    highlight_square $mode $x $y

    # highlight vertical and horizontal
    for { set i 1 } {$i <= 8} {incr i} {
        highlight_square $mode $x $i
        highlight_square $mode $i $y
    }

    # highlight diagonals
    for { set i 1 } {$i <= 8} {incr i} {
        highlight_square $mode [expr $x+$i] [expr $y+$i]
        highlight_square $mode [expr $x-$i] [expr $y-$i]
        highlight_square $mode [expr $x+$i] [expr $y-$i]
        highlight_square $mode [expr $x-$i] [expr $y+$i]
    }
}
proc disable_next {} {
    .next config -state disabled
}

setup_display

# button for sending a 'next' message
button .next -text next -command {prolog_event next}
pack .next

# button for sending a 'stop' message
button .stop -text stop -command {prolog_event stop}
pack .stop
10.45.6 Quick Reference

10.45.6.1 Command Format Summary

Command ::= Name
| codes(code list)
| write(term)
| writeq(term)
| write_canonical(term)
| format(Fmt, Args)
| dq(Command)
| \text{br}(\text{Command}) |
| \text{sqb}(\text{Command}) |
| \text{min}(\text{Command}) |
| \text{dot}(\text{ListOfNames}) |
| \text{list}(\text{ListOfCommands}) |

\text{Fmt} ::= \text{atom} \\
\text{Name} ::= \text{atom} \hspace{1cm} \{ \text{other than } [\hspace{1cm} ] \} \\
| \text{number} |
\text{ListOfCommands} ::= [ ] \\
| [ \text{Command} \mid \text{ListOfCommands} ] |
\text{ListOfNames} ::= [ ] \\
| [ \text{Name} \mid \text{ListOfNames} ] |
\text{Args} ::= [ ] \\
| [ \text{term} \mid \text{Args} ] |

where

\text{Atom} \hspace{1cm} \text{Number} \hspace{1cm} \text{denote their printed representations} \hspace{1cm} \\
\text{codes}(\text{PrologString}) \\
\hspace{1cm} \text{denotes the string represented by } \text{PrologString} \hspace{1cm} \text{(a code list)} \hspace{1cm} \\
\text{write}(\text{Term}) \\
\text{writeq}(\text{Term}) \\
\text{write\_canonical}(\text{Term}) \\
\hspace{1cm} \text{denotes the string that is printed by the corresponding built-in predicate} \hspace{1cm} \\
\text{format}(\text{Fmt}, \text{Args}) \\
\hspace{1cm} \text{denotes the string that is printed by the corresponding built-in predicate} \hspace{1cm} \\
\text{dq}(\text{Command}) \\
\hspace{1cm} \text{denotes the string specified by } \text{Command} \hspace{1cm} \text{, enclosed in double quotes} \hspace{1cm} \\
\text{br}(\text{Command}) \\
\hspace{1cm} \text{denotes the string specified by } \text{Command} \hspace{1cm} \text{, enclosed in curly brackets} \hspace{1cm} \\
\text{sqb}(\text{Command}) \\
\hspace{1cm} \text{denotes the string specified by } \text{Command} \hspace{1cm} \text{, enclosed in square brackets} \hspace{1cm} \\
\text{min}(\text{Command}) \\
\hspace{1cm} \text{denotes the string specified by } \text{Command} \hspace{1cm} \text{, immediately preceded by a hyphen} \hspace{1cm} \\
\text{dot}(\text{ListOfName}) \\
\hspace{1cm} \text{denotes the widget path specified by } \text{ListOfName} \hspace{1cm} \text{, preceded by and separated by dots} \hspace{1cm} \\
\text{list}(\text{ListOfCommands}) \\
\hspace{1cm} \text{denotes the TCL list with one element for each element in } \text{ListOfCommands} \hspace{1cm} \\
\text{ListOfCommands} \\
\hspace{1cm} \text{denotes the string denoted by each element, separated by spaces} \hspace{1cm}
10.45.6.2 Predicates for Prolog to Interact with Tcl Interpreters

`tcl_new(-TclInterpreter)`  
Create a Tcl interpreter and return a handle to it in the variable `Interpreter`.

`tcl_delete(+TclInterpreter)`  
Given a handle to a Tcl interpreter in variable `TclInterpreter`, it deletes the interpreter from the system.

`tcl_eval(+TclInterp, +Command, -Result)`  
Evaluates the Tcl command term given in `Command` in the Tcl interpreter handle provided in `TclInterpreter`. The result of the evaluation is returned as a string in `Result`.

`tcl_event(+TclInterp, +Command, -Events)`  
Evaluates the Tcl command term given in `Command` in the Tcl interpreter handle provided in `TclInterpreter`. The first Prolog events arising from the evaluation is returned as a list in `Events`. Blocks until there is something on the event queue.

10.45.6.3 Predicates for Prolog to Interact with Tcl Interpreters with Tk Extensions

`tk_new(+Options, -Interp)`  
Create a Tcl interpreter with Tk extensions.  
`Options` should be a list of options described following:

`top_level_events`  
This allows Tk events to be handled while Prolog is waiting for terminal input; for example, while the Prolog system is waiting for input at the top-level prompt. Without this option, Tk events are not serviced while the Prolog system is waiting for terminal input.

`name(+ApplicationName)`  
This gives the main window a title `ApplicationName`. This name is also used for communicating between Tcl/Tk applications via the Tcl `send` command.

`display(+Display)`  
(This is X windows specific.) Gives the name of the screen on which to create the main window. If this is not given, then the default display is determined by the `DISPLAY` environment variable.

`tk_do_one_event`  
`tk_do_one_event(+Options)`  
Passes control to Tk to handle a single event before passing control back to Prolog. The type of events handled is passed through the `Options` variable, a list of event types and options.  
The `Options` list can contain the following atoms:

`tk_dont_wait`  
do not wait for new events, process only those that are ready
Chapter 10: The Prolog Library

\[ \text{tk_window_events} \]
process window events

\[ \text{tk_file_events} \]
process file events

\[ \text{tk_timer_events} \]
process timer events

\[ \text{tk_idle_events} \]
process Tcl_DoWhenIdle events

\[ \text{tk_all_events} \]
process any event

Calling tk_do_one_event/0 is equivalent to a call to tk_do_one_event/1 with all flags set. If the tk_dont_wait flag is set and there is no event to handle, then the call will fail.

\[ \text{tk_next_event(+Options, -Event)} \]

\[ \text{tk_next_event(+Options, +TclInterpreter, -Event)} \]
These predicates are similar to tk_do_one_event/[0,1] except that they processes Tk events until is at least one Prolog event happens, when they succeed binding Event to the first term on the Prolog event queue associated with the interpreter TclInterpreter.

\[ \text{tk_main_loop} \]
Pass control to Tk until all windows in all Tcl/Tk interpreters are gone.

\[ \text{tk_main_window(+TclInterpreter, -TkWindow)} \]
Return in TkWindow a reference to the main window of a Tcl/Tk interpreter with handle passed in TclInterpreter.

\[ \text{tk_destroy_window(+TkWindow)} \]
Destroy a window or widget.

\[ \text{tk_make_window_exist(+TkWindow)} \]
Force display of a window or widget.

\[ \text{tk_num_main_windows(-NumberOfWindows)} \]
Return in NumberOfWindows the total number of Tk main windows existing in the system.

10.45.6.4 Commands for Tcl Interpreters to Interact with The Prolog System

\[ \text{prolog} \]
Evaluate a Prolog expression from Tcl.

\[ \text{prolog_event} \]
Place a Prolog term on the Prolog event queue from inside Tcl.

10.45.7 Resources

We do not know of any resources out there specifically for helping with creating Prolog applications with Tcl/Tk interfaces. Instead we list here some resources for Tcl/Tk, which may help readers to build the Tcl/Tk side of the application.
10.45.7.1 Web Sites
The home of Tcl/Tk is at:

http://www.tcl.tk

Also visit the Tcl/Tk wiki:

https://wiki.tcl-lang.org/

10.45.7.2 Manual Pages
Complete manual pages in various formats and for various versions of the Tcl/Tk library can be found at the Tcl/Tk web site:

http://www.tcl.tk/

10.46 Term Utilities—library(terms)
This library module provides miscellaneous operations on terms. Exported predicates:

subsumeschk(+General, +Specific)

is true when Specific is an instance of General. It does not bind any variables.

This predicate is identical to the built-in subsumes_term/2 and it is only present for backwards compatibility.

subsumes(+General, +Specific)

is true when Specific is an instance of General. It will bind variables in General (but not those in Specific, except when +General and +Specific share variables) so that General becomes identical to Specific.

In many cases, binding variable is not really desirable, in which case subsumes_term/2 should be used instead. If unification is in fact wanted, it may be better to make this explicit in your code by using subsumes_term/2 followed by an explicit unification, e.g. subsumes_term(G,S), G=S.

variant(+Term, +Variant)

is true when Term and Variant are identical modulo renaming of variables, provided Term and Variant have no variables in common.

term_subsumer(+Term1, +Term2, -Term)

binds Term to a most specific generalization of Term1 and Term2. Using Plotkin’s algorithm [Machine Intelligence 5, 1970], extended by Dan Sahlin to handle cyclic structures.

term_hash(+Term, -Hash)

Equivalent to term_hash(Term, [], Hash).

term_hash(+Term, +Options, -Hash)

term_hash/[2,3,4] is provided primarily as a tool for the construction of sophisticated Prolog clause access schemes. Its intended use is to generate hash values for terms that will be used with first argument clause indexing, yielding
compact and efficient multi-argument or deep argument indexing. *Options* is a list of options,

```
algorithm(Algorithm)
```

*Algorithm* specifies which hash function to use. An atom, one of,

```
default
```

This is currently the same as *jenkins*. This is the default. If we ever see a need to change the default hash algorithm again then the algorithm denoted by *default* may change but the algorithm denoted by the other names, like *sicstus-4.0.5*, will not change.

```
jenkins
```

Based on the algorithm “lookup3” by Bob Jenkins, see http://burtleburtle.net/bob/hash/doobs.html.

```
hsieh
```

Based on the algorithm “SuperFastHash” by Paul Hsieh, see http://www.azillionmonkeys.com/qed/hash.html. Despite the name neither this nor any other choice of algorithm significantly affects the speed of term_hash/3.

```
sdbm
```

Based on the well known algorithm “sdbm”.

```
'sicstus-4.0.4'
```

This is the algorithm used up to SICStus Prolog 4.0.4 (inclusive). It is only present to provide backwards compatibility. It is not as good as any of the above algorithms. Note that this atom needs to be quoted. This algorithm produces hash values that may differ between platforms.

```
'sicstus-4.0.5'
```

This is the same as *jenkins*. I.e. the default since SICStus Prolog 4.0.5. Note that this atom needs to be quoted.

there are some other (not as good) algorithms available for the curious, see the source for detail. Unless otherwise noted, the hash value will be identical across runs and platforms.

```
range(Range)
```

The resulting hash value will be non-negative and less than the upper bound specified by *Range*. *Range* should be either a positive integer, or an atom, one of,

```
infinite
```
Do not constrain the hash value. Currently all hash algorithms produce an unsigned 32-bit integer.

**smallint**

Ensure the resulting hash value is a small integer. This is the same as specifying a range of $2^{28}$ on 32-bit platforms and $2^{60}$ on 64-bit platforms.

**smallint32**

Ensure the resulting hash value is in the 32-bit platform range of small integers, i.e. the same as a range of $2^{28}$.

**default**

The same as `smallint32`. This is the default. This ensures that, by default, the same hash value is computed for the same term on both 32-bit and 64-bit platforms.

**depth(‌Depth)‌**

Specifies how deep to descend into the term when calculating the hash value. If `Depth` is a non-negative integer the subterms up to depth `Depth` of `Term` are used in the computation. Alternatively, if `Depth` is the atom `infinite`, all subterms of `Term` are relevant in computing `Hash`. In the latter case `Term` must be acyclic. In this context the `depth` of a term is defined as follows: the (principal functor of) the term itself has depth 1, and an argument of a term with depth `i` has depth `i+1`. Note that this is similar to, but not the same as, the value computed by `term_depth/2`. For legacy reasons a `Depth` of `-1` is treated the same as `infinite`.

**if_var(‌IfVar‌)‌**

Specifies what to do if a variable is encountered in the term (i.e. to the specified depth). `IfVar` should be an atom, one of,

- **error**
  
  An instantiation error is thrown.

- **ignore**
  
  The variable is ignored and the hash algorithm continues with the other parts of the term.

- **value(Value)**
  
  The hash algorithm stops, the intermediate hash result is discarded and `Hash` is bound to `Value`. There is no restrictions on `Value`, it need not be an integer or even be ground.

- **default**
  
  This is the same as `value(_),` i.e. `term_hash/3` just succeeds without binding `Hash`. This is the default. This is useful when the hash value us used for first-argument indexing. This ensures that if the (possibly variable-valued) hash values for `Term1` and `Term2`
Chapter 10: The Prolog Library

are Hash1 and Hash2, respectively, then if Term1 and Term2 are unifiable (to the specified depth) then so are Hash1 and Hash2. For other use cases it is probably more appropriate to specify if_var(error).

term_hash(+Term, +Depth, +Range, -Hash)
Equivalent to term_hash(Term, [depth(Depth), range(Range)], Hash).

term_variables_set(+Term, -Variables)
since release 4.3
True if Variables is the (ordered) set of variables occurring in Term.
This was called term_variables/2 prior to SICStus Prolog 4.3 but now term_variables/2 is a built-in with different meaning, due to alignment with the ISO Prolog standard.

term_variables_bag(+Term, -Variables)
True if Variables is the list of variables occurring in Term, in first occurrence order.
This predicate has been superseded by the built-in term_variables/2 and it is only present for backwards compatibility.
The name is an historical accident, the result is not really a bag (i.e. multiset).

cyclic_term(+X)
True if X is infinite (cyclic). Runs in linear time.

term_order(+X, +Y, -R)
is true when X and Y are arbitrary terms, and R is <, =, or > according as X \&\& Y, X == Y, or X @> Y. This is the same as compare/3, except for the argument order.

contains_term(+Kernel, +Expression)
is true when the given Kernel occurs somewhere in the Expression. It can only be used as a test; to generate sub-terms use sub_term/2.

free_of_term(+Kernel, +Expression)
is true when the given Kernel does not occur anywhere in the Expression. NB: if the Expression contains an unbound variable, this must fail, as the Kernel might occur there. Since there are infinitely many Kernels not contained in any Expression, and also infinitely many Expressions not containing any Kernel, it doesn’t make sense to use this except as a test.

occurrences_of_term(+Kernel, +Expression, -Tally)
is true when the given Kernel occurs exactly Tally times in Expression. It can only be used to calculate or test Tally; to enumerate Kernels you’ll have to use sub_term/2 and then test them with this routine. If you just want to find out whether Kernel occurs in Expression or not, use contains_term/2 or free_of_term/2.

contains_var(+Variable, +Term)
is true when the given Term contains at least one sub-term which is identical to the given Variable. We use == to check for the variable (contains_term/2 uses =) so it can be used to check for arbitrary terms, not just variables.
free_of_var(+Variable, +Term)
is true when the given Term contains no sub-term identical to the given Variable
(which may actually be any term, not just a var). For variables, this is precisely
the "occurs check" which is needed for sound unification.

occurrences_of_var(+Variable, +Term, -Tally)
is true when the given Variable occurs exactly Tally times in Term. It can
only be used to calculate or test Tally; to enumerate Variables you'll have to use
sub_term/2 and then test them with this routine. If you just want to find
out whether Variable occurs in Term or not, use contains_var/2 or free_of_var/2.

sub_term(?Kernel, +Term)
is true when Kernel is a sub-term of Term. It enumerates the sub-terms of
Term in an arbitrary order. Well, it is defined that a sub-term of Term will be
enumerated before its own sub-terms are (but of course some of those sub-terms
might be elsewhere in Term as well).

depth_bound(+Term, +Bound)
is true when the term depth of Term is no greater than Bound, that is, when
constructor functions are nested no more than Bound deep. Later variable
bindings may invalidate this bound. To find the (current) depth, use term_depth/2.

length_bound(?List, +Bound)
is true when the length of List is no greater than Bound. It can be used to
enumerate Lists up to the bound.

size_bound(+Term, +Bound)
is true when the number of constant and function symbols in Term is (currently)
at most Bound. If Term is non-ground, later variable bindings may invalidate
this bound. To find the (current) size, use term_size/2.

term_depth(+Term, -Depth)
calculates the Depth of a Term, using the definition

\[
\text{term_depth(Var)} = 0 \\
\text{term_depth(Const)} = 0 \\
\text{term_depth(F(T1,\ldots,Tn)}) = 1+\text{max(term_depth(T1),\ldots,term_depth(Tn))}
\]

Could be defined as:

\[
\text{term_depth}(X, \text{Depth}) :- \\
\text{simple}(X), !, \text{Depth} = 0. \\
\text{term_depth}(X, \text{Depth}) :- \\
( \text{foreacharg}(A,X), \\
\text{fromto}(0,D0,D,\text{Depth0}) \\
\text{do term_depth}(A, D1), \\
D \text{ is } \text{max}(D0,D1) \\
), \\
\text{Depth} \text{ is } \text{Depth0}+1.
\]
term_size(+Term, -Size)
calculates the Size of a Term, defined to be the number of constant and function
symbol occurrences in it. Could be defined as:

\[
\text{term_size}(X, \text{Size}) :-
\begin{align*}
\text{var}(X), &!, \text{Size} = 0. \\
\text{term_size}(X, \text{Size}) :-
\text{simple}(X), &!, \text{Size} = 1. \\
\text{term_size}(X, \text{Size}) :-
( \text{foreacharg}(A, X), \\
\text{fromto}(1, S0, S, \text{Size}) \\
\text{do } \text{term_size}(A, S1), \\
S &\text{ is } S0+S1 \\
). 
\end{align*}
\]

\[
\text{same_functor}(\forall T1, \forall T2)
\]
is true when \( T1 \) and \( T2 \) have the same principal functor. If one of the terms is
a variable, it will be instantiated to a new term with the same principal functor
as the other term (which should be instantiated) and with arguments being new
distinct variables. If both terms are variables, an error is reported.

\[
\text{same_functor}(\forall T1, \forall T2, \forall N)
\]
is true when \( T1 \) and \( T2 \) have the same principal functor, and their common
arity is \( N \). Like \text{same_functor}/3, at least one of \( T1 \) and \( T2 \) must be bound,
or an error will be reported.

\[
\text{same_functor}(\forall T1, \forall T2, \forall F, \forall N)
\]
is true when \( T1 \) and \( T2 \) have the same principal functor, and their common
funcctor is \( F/N \). Given \( T1 \) (or \( T2 \)) the remaining arguments can be computed.
Given \( F \) and \( N \), the remaining arguments can be computed. If too many argu-
ments are unbound, an error is reported.

### 10.47 Meta-Call with Limit on Execution Time—
\text{library}(timeout)

This module contains utilities for computing a goal with limit on execution time.

As of release 4.4, this library no longer uses a foreign resource, and it can be used by more
than one SICStus instance in the same process.

Exported predicates:

\[
\text{time_out}(:\text{Goal, } +\text{Time, } -\text{Result})
\]
The \text{Goal} is executed as if by \text{call}/1. If computing any solution takes more
than \text{Time} milliseconds, the goal will be aborted, as if by \text{throw}/1, and \text{Result}
unified with the atom \text{time_out}. If the goal succeeds within the specified time,
\text{Result} is unified with the atom \text{success}. \text{Time} must be a positive integer, less
than 2147483647.

Currently, time is measured in runtime (as opposed to walltime), i.e. the time
does not increment while the program is waiting, e.g. during a blocking read.
Ideally, the measured runtime should be thread-specific, i.e. it should not be affected by computations done in other threads in the process (of course, thread-specific time is the same as process runtime for a single-threaded process). Thread-specific runtime measurement is only implemented on Windows.

The precision of the timeout interval is usually not better than several tens of milliseconds. This is due to limitations in the timing mechanisms used to implement library(timeout).

### 10.48 Dense Arrays as Binary Trees—library(trees)

This library module provides updatable binary trees with logarithmic access time. Exported predicates:

- `gen_label(?Index, +Tree, ?Value)`
  - Assumes that Tree is a proper binary tree, and is true when Value is the Index-th element in Tree. Can be used to enumerate all Values by ascending Index.

- `get_label(+Index, +Tree, -Label)`
  - Treats the tree as an array of N elements and returns the Index-th. If Index < 1 or > N it simply fails, there is no such element. As Tree need not be fully instantiated, and is potentially unbounded, we cannot enumerate Indices.

- `list_to_tree(+List, -Tree)`
  - Takes a given proper List of N elements and constructs a binary Tree where get_label(K, Tree, Lab) < = Lab is the Kth element of List.

- `map_tree(:Pred, +OldTree, ?NewTree)`
  - Is true when OldTree and NewTree are binary trees of the same shape and Pred(Old,New) is true for corresponding elements of the two trees.

- `put_label(+Index, +OldTree, -Label, -NewTree)`
  - Constructs a new tree the same shape as the old which moreover has the same elements except that the Index-th one is Label. Unlike the "arrays" of library(arrays), OldTree is not modified and you can hang on to it as long as you please. Note that O(lg N) new space is needed.

- `put_label(+Index, +OldTree, -OldLabel, -NewTree, +NewLabel)`
  - Is true when OldTree and NewTree are trees of the same shape having the same elements except that the Index-th element of OldTree is OldLabel and the Index-th element of NewTree isNewLabel. You can swap the <Tree,Label> argument pairs if you like, it makes no difference.

- `tree_size(+Tree, -Size)`
  - Calculates the number of elements in the Tree. All trees made by list_to_tree/2 that are the same size have the same shape.

- `tree_to_list(+Tree, -List)`
  - Is the converse operation to list_to_tree/2. Any mapping or checking operation can be done by converting the tree to a list, mapping or checking the list, and converting the result, if any, back to a tree. It is also easier for a human to read a list than a tree, as the order in the tree goes all over the place.
10.49 Type Checking—library(types)

This library module provides more and better type tests. For the purposes of this library, we first define an abstract type \texttt{typeterm}, as follows:

\[
typeterm ::= \text{atom} \\
            \mid \text{atomic} \\
            \mid \text{boolean} \\
            \mid \text{callable} \\
            \mid \text{character} \\
            \mid \text{character\_code} \\
            \mid \text{compound} \\
            \mid \text{db\_reference} \\
            \mid \text{float} \\
            \mid \text{float(rangeterm)} \\
            \mid \text{ground} \\
            \mid \text{integer} \\
            \mid \text{integer(rangeterm)} \\
            \mid \text{list} \\
            \mid \text{list(Type)} \\
            \mid \text{mutable} \\
            \mid \text{nonvar} \\
            \mid \text{number} \\
            \mid \text{number(rangeterm)} \\
            \mid \text{oneof}(L) \\
            \mid \text{order} \\
            \mid \text{pair} \\
            \mid \text{pred\_spec} \\
            \mid \text{pred\_spec\_tree} \\
            \mid \text{proper\_list} \\
            \mid \text{proper\_list}(Type) \\
            \mid \text{simple} \\
            \mid \text{term} \\
            \mid \text{var} \\
            \mid \text{var\_or}(Type) \\
\]

\[
rangeterm ::= \text{between}(L,U) \\
             \mid \geq (L) \\
             \mid > (L) \\
             \mid < (L) \\
             \mid \leq (L) \\
             \mid = (L) \\
             \mid \neq (L) \\
\]

Culprit information:

These predicates takes arguments that are used when reporting the reason and location of errors. The arguments are:
Goal must be a callable term, without (:)/2 module wrapping, with arity at least ArgNo.

ArgNo must be a non-negative integer, where zero means no specific argument position.

Culprit the term that has the offending value.

Exported predicates:

must_be(+Term, +Type, +Goal, +ArgNo)
checks whether the Term belongs to the indicated Type, which should be a typeterm. If it doesn’t, several different error exceptions can be thrown: the Term may not be instantiated enough to tell yet (Instantiation Error); it may be instantiated when an unbound variable was expected (Uninstantiation Error); it may be definitely not of the right type (Type Error); it may be of the right type but not representable (Representation Error); or it may be of the right type but in the wrong domain (Domain Error). If an error exception is thrown, it will include Goal and ArgNo and, if possible, the line of code in the scope of which the error occurred. See Section 4.15.4 [ref-ere-err], page 200.

illarg(+ErrorTerm, +Goal, +ArgNo)
illarg(+ErrorTerm, +Goal, +ArgNo, +Culprit)
is the way to raise an error exception, if you would like the exception to pinpoint the line of code in the scope of which the error occurs. This is especially useful in the context of source-linked debugging. Culprit defaults to argument number ArgNo of Goal. These three arguments are passed to the exception being raised, if appropriate. ErrorTerm should be one of the following. See Section 4.15.4 [ref-ere-err], page 200.

var An Instantiation error is raised.

type(ErrorType)
Same as must_be(Culprit, ErrorType, Goal, ArgNo).

domain(ErrorType,ErrorDomain)
First, the type is checked by must_be(Culprit, ErrorType, Goal, ArgNo). If the type is valid, a Domain Error is raised with the expected domain being ErrorDomain.

force_type(ExpType)
A Type Error is raised.

context(ContextType,CommandType)
A Context Error is raised.

existence(ObjType,Culprit,Message)
An Existence Error is raised.

permission(Operation,ObjType,Message)
A Permission Error is raised.

representation(ErrorType)
A Representation Error is raised.
Chapter 10: The Prolog Library

10.50 Unweighted Graph Operations—library(ugraphs)

This library module provides operations on directed graphs. An unweighted directed graph (ugraph) is represented as a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by `keysort/2` with unique keys) and the neighbors of each vertex are also in standard order (as produced by `sort/2`), and every neighbor appears as a vertex even if it has no neighbors itself.

An undirected graph is represented as a directed graph where for each edge $(U, V)$ there is a symmetric edge $(V, U)$.

An edge $(U, V)$ is represented as the term $U-V$.

A vertex can be any term. Two vertices are distinct iff they are not identical ($\equiv$).

A path is represented as a list of vertices. No vertex can appear twice in a path.

Exported predicates:

- `vertices_edges_to_ugraph(+Vertices, +Edges, -Graph)`
  - is true if `Vertices` is a proper list of vertices, `Edges` is a proper list of edges, and `Graph` is a graph built from `Vertices` and `Edges`. `Vertices` and `Edges` may be in any order. The vertices mentioned in `Edges` do not have to occur explicitly in `Vertices`. `Vertices` may be used to specify vertices that are not connected to any edges.

- `vertices(+Graph, -Vertices)`
  - unifies `Vertices` with the vertices in `Graph`. Could be defined as:
    ```prolog
    vertices(Graph, Vertices) :-
    (   foreach(V-,Graph),
        foreach(V,Vertices)
        do true
    ).
    ```

- `edges(+Graph, -Edges)`
  - unifies `Edges` with the edges in `Graph`. Could be defined as:
edges(Graph, Edges) :-
( foreach(V1-Neibs,Graph),
  fromto(Edges,S0,S,[[]])
do ( foreach(V2,Neibs),
    param(V1),
    fromto(S0,[V1-V2|S1],S1,S)
do true )
).

add_vertices(+Graph1, +Vertices, -Graph2)
is true if Graph2 is Graph1 with Vertices added to it.

del_vertices(+Graph1, +Vertices, -Graph2)
is true if Graph2 is Graph1 with Vertices and all edges to and from Vertices removed from it.

add_edges(+Graph1, +Edges, -Graph2)
is true if Graph2 is Graph1 with Edges and their "to" and "from" vertices added to it.

del_edges(+Graph1, +Edges, -Graph2)
is true if Graph2 is Graph1 with Edges removed from it.

transpose_ugraph(+Graph, -Transpose)
is true if Transpose is the graph computed by replacing each edge (u,v) in Graph by its symmetric edge (v,u). It can only be used one way around. The cost is $O(N \log N)$.

neighbors(+Vertex, +Graph, -Neighbors)
neighbours(+Vertex, +Graph, -Neighbors)
is true if Vertex is a vertex in Graph and Neighbors are its neighbors.

complement(+Graph, -Complement)
Complement is the complement graph of Graph, i.e. the graph that has the same vertices as Graph but only the edges that are not in Graph.

compose(+G1, +G2, -Composition)
computes Composition as the composition of two graphs, which need not have the same set of vertices.

transitive_closure(+Graph, -Closure)
computes Closure as the transitive closure of Graph in $O(N^3)$ time.

transitive_reduction(+Graph, -Reduction) since release 4.3.3
computes Reduction as the transitive reduction of Graph.
Aho et al. let GraphT be the transitive closure of Graph. Then an edge uv belongs to the transitive reduction iff uv belongs to Graph but not to the composition of Graph and GraphT. In this construction, the edges of the composition represent pairs of vertices connected by paths of length two or more.
symmetric_closure(+Graph, -Closure)
computes Closure as the symmetric closure of Graph, i.e. for each edge (u,v) in
Graph, add its symmetric edge (v,u). Approx. $O(N \log N)$ time. This is useful
for making a directed graph undirected. Could be defined as:

```prolog
symmetric_closure(Graph, Closure) :-
    transpose_ugraph(Graph, Transpose),
    ( foreach(V-Neibs1,Graph),
      foreach(V-Neibs2,Transpose),
      foreach(V-Neibs,Closure)
    do  ord_union(Neibs1, Neibs2, Neibs)
    ).
```

top_sort(+Graph, -Sorted)
finds a topological ordering of Graph and returns the ordering as a list of Sorted
vertices. Fails iff no ordering exists, i.e. iff the graph contains cycles. Approx.
$O(N \log N)$ time.

max_path(+V1, +V2, +Graph, -Path, -Cost)
is true if Path is a list of vertices constituting a longest path of cost Cost from
V1 to V2 in Graph, there being no cyclic paths from V1 to V2. Takes $O(N^2)$
time.

min_path(+V1, +V2, +Graph, -Path, -Length)
is true if Path is a list of vertices constituting a shortest path of length Length
from V1 to V2 in Graph. Takes $O(N^2)$ time.

min_paths(+Vertex, +Graph, -Tree)
is true if Tree is a tree of all the shortest paths from Vertex to every other vertex
in Graph. This is the single-source shortest paths problem. The algorithm is
straightforward.

path(+Vertex, +Graph, -Path)
is given a Graph and a Vertex of that Graph, and returns a maximal Path
rooted at Vertex, enumerating more Paths on backtracking.

reduce(+Graph, -Reduced)
is true if Reduced is the reduced graph for Graph. The vertices of the reduced
graph are the strongly connected components of Graph. There is an edge in
Reduced from u to v iff there is an edge in Graph from one of the vertices in u
to one of the vertices in v. A strongly connected component is a maximal set
of vertices where each vertex has a path to every other vertex. Algorithm from
"Algorithms" by Sedgewick, page 482, Tarjan’s algorithm.

reachable(+Vertex, +Graph, -Reachable)
is given a Graph and a Vertex of that Graph, and returns the set of vertices
that are Reachable from that Vertex. Takes $O(N^2)$ time.

random_ugraph(+P, +N, -Graph)
where P is a probability, unifies Graph with a random graph of N vertices where
each possible edge is included with probability P.
min_tree(+Graph, -Tree, -Cost)
    is true if Tree is a spanning tree of an undirected Graph with cost Cost, if it
    exists. Using a version of Prim's algorithm.

max_cliques(+Graph, -Cliques)
    since release 4.3.3
    is true if Cliques is the set of the maximal cliques of the undirected graph Graph.
    That is, all subsets of vertices such that (i) each pair of vertices in any listed
    subset is connected by an edge, and (ii) no listed subset can have any additional
    vertex added to it. Using a version of the Bron-Kerbosch algorithm.

10.51 An Inverse of numbervars/3—library(varnumbers)

The built-in predicate numbervars/3 makes a term ground by binding the variables in it to
subterms of the form 'VAR'(N) where N is an integer. Most of the calls to numbervars/3
look like

    numbervars(Term, 0, _)

which can be abbreviated to

    numbervars(Term)

if you use this package.

varnumbers/3 is a partial inverse to numbervars/3:

    varnumbers(Term, N0, Copy)

unifies Copy with a copy of Term in which subterms of the form 'VAR'(N) where N
is an integer not less than N0 (that is, subterms which might have been introduced by
numbervars/3 with second argument N0) have been consistently replaced by new variables.
Since 0 is the usual second argument of numbervars/3, there is also

    varnumbers(Term, Copy)

This provides a facility whereby a Prolog-like data base can be kept as a term. For example,
we might represent append/3 thus:

    Clauses = [
        (append([], 'VAR'(0), 'VAR'(0)) :- true),
        (append([ 'VAR'(0)|'VAR'(1), 'VAR'(2), ['VAR'(0)|'VAR(3)]] :-
            append('VAR'(1), 'VAR'(2), 'VAR'(3)))
    ]

and we might access clauses from it by doing

    prove(Goal, Clauses) :-
        member(Clause, Clauses),
        varnumbers(Clause, (Goal:-Body)),
        prove(Goal).
Exported predicates:

\[
\text{numbervars}(\text{+Term}) \quad \text{makes } \text{Term} \text{ ground by binding variables to subterms } \$VAR'(N) \text{ with values of } N \text{ ranging from 0 up.}
\]

\[
\text{vargnumbers}(\text{+Term}, \text{-Copy}) \quad \text{succeeds when } \text{Term} \text{ was a term producing by calling numbervars(} \text{Term} \text{) and } \text{Copy} \text{ is a copy of } \text{Term} \text{ with such subterms replaced by variables.}
\]

\[
\text{vargnumbers}(\text{+Term}, \text{+NO}, \text{-Copy}) \quad \text{succeeds when } \text{Term} \text{ was a term produced by calling numbervars(} \text{Term}, \text{NO}, \text{N} \text{) (so that all subterms } \$VAR'(X) \text{ have integer}(X), X \geq \text{NO} \text{) and } \text{Copy} \text{ is a copy of } \text{Term} \text{ with such subterms replaced by variables.}
\]

10.52 Weighted Graph Operations—library(wgraphs)

This library module provides operations on weighted directed graphs. A weighted directed graph (wgraph) is represented as a list of (vertex-edgelist) pairs, where the pairs are in standard order (as produced by \text{keysort}/2 with unique keys), the edgelist is a list of (neighbor-weight) pair also in standard order (as produced by \text{keysort}/2 with unique keys), every weight is a nonnegative integer, and every neighbor appears as a vertex even if it has no neighbors itself.

An undirected graph is represented as a directed graph where for each edge \((U,V)\) there is a symmetric edge \((V,U)\).

An edge \((U,V)\) is represented as the term \(U-V\).

A vertex can be any term. Two vertices are distinct iff they are not identical (==).

A path is represented as a list of vertices. No vertex can appear twice in a path.

Exported predicates:

\[
\text{vertices}/2, \text{edges}/2, \text{add_vertices}/3, \text{neighbors}/3, \text{neighbours}/3
\]

Re-exported from library(wgraphs).

\[
\text{wgraph_to_ugraph}(\text{+WeightedGraph}, \text{-Graph}) \quad \text{is true if } \text{Graph} \text{ has the same vertices and edges as } \text{WeightedGraph}, \text{ except the edges of } \text{Graph} \text{ are unweighted. Could be defined as:}
\]

wgraph_to_ugraph(WGraph, Graph) :-
(  foreach(V-WNeibs,WGraph),
    foreach(V-Neibs,Graph)
  do (  foreach(V1-_,WNeibs),
    foreach(V1,Neibs)
      do true
    )
).

ugraph_to_wgraph(+Graph, -WeightedGraph)
  is true if WeightedGraph has the same vertices and edges as Graph, except the
  edges of WeightedGraph all have weight 1. Could be defined as:
  ugraph_to_wgraph(Graph, WGraph) :-
(  foreach(V-Neibs,Graph),
    foreach(V-WNeibs,WGraph)
  do (  foreach(V1,Neibs),
    foreach(V1-1,WNeibs)
      do true
    )
).

ugraph_to_wgraph(+SubGraph, +WeightedGraph, -WeightedSubGraph)
  is true if WeightedSubGraph has the same vertices and edges as SubGraph and
  the same weights as the corresponding edges in WeightedGraph.

vertices_edges_to_wgraph(+Vertices, +Edges, -WeightedGraph)
  is true if Vertices is a proper list of vertices, Edges is a proper list of edges,
  and WeightedGraph is a graph built from Vertices and Edges. Vertices and
  Edges may be in any order. The vertices mentioned in Edges do not have to
  occur explicitly in Vertices. Vertices may be used to specify vertices that are
  not connected to any edges.

del_vertices(+WeightedGraph1, +Vertices, -WeightedGraph2)
  is true if WeightedGraph2 is WeightedGraph1 with Vertices and all edges to
  and from Vertices removed from it.

add_edges(+WeightedGraph1, +Edges, -WeightedGraph2)
  is true if WeightedGraph2 is WeightedGraph1 with Edges and their "to" and
  "from" vertices added to it.

del_edges(+WeightedGraph1, +Edges, -WeightedGraph2)
  is true if WeightedGraph2 is WeightedGraph1 with Edges removed from it.

transpose_wgraph(+WeightedGraph, -Transpose)
  is true if Transpose is the graph computed by replacing each edge \((u,v)\) in
  WeightedGraph by its symmetric edge \((v,u)\). It can only be used one way
  around. The cost is \(O(N \log N)\).

transitive_closure(+WeightedGraph, -Closure)
  computes Closure as the transitive closure of WeightedGraph in \(O(N^3)\) time.
  Uses Floyd’s algorithm and fragments of Barney Pell’s code.
symmetric_closure(+WeightedGraph, -Closure)
computes Closure as the symmetric closure of WeightedGraph, i.e. for each
edge \((u,v)\) in WeightedGraph, add its symmetric edge \((v,u)\). Approx \(O(N \log N)\) time. This is useful for making a directed graph undirected.

top_sort(+Graph, -Sorted)
finds a topological ordering of a Graph and returns the ordering as a list of
Sorted vertices. Fails iff no ordering exists, i.e. iff the graph contains cycles.
Takes \(O(N \log N)\) time.

max_path(+V1, +V2, +WeightedGraph, -Path, -Cost)
is true if Path is a list of vertices constituting a longest path of cost Cost from
V1 to V2 in WeightedGraph, there being no cyclic paths from V1 to V2. Takes
\(O(N^2)\) time.

min_path(+V1, +V2, +WeightedGraph, -Path, -Cost)
is true if Path is a list of vertices constituting a shortest path with total cost
Cost from V1 to V2 in WeightedGraph. Takes \(O(N^2)\) time.

min_paths(+Vertex, +WeightedGraph, -Tree)
is true if Tree is a tree of all the shortest paths from Vertex to every other
vertex in WeightedGraph. This is the single-source shortest paths problem.
Using Dijkstra’s algorithm.

path(+Vertex, +WeightedGraph, -Path)
is given a WeightedGraph and a Vertex of that WeightedGraph, and returns a
maximal Path rooted at Vertex, enumerating more Paths on backtracking.

reduce(+WeightedGraph, -Reduced)
is true if Reduced is the reduced graph for WeightedGraph. The vertices of
the reduced graph are the strongly connected components of WeightedGraph.
There is an edge in Reduced from u to v iff there is an edge in WeightedGraph
from one of the vertices in u to one of the vertices in v. A strongly connected
component is a maximal set of vertices where each vertex has a path to every
other vertex. Algorithm from "Algorithms" by Sedgewick, page 482, Tarjan’s
algorithm.

reachable(+Vertex, +WeightedGraph, -Reachable)
is given a WeightedGraph and a Vertex of that WeightedGraph, and returns the
set of vertices that are Reachable from that Vertex. Takes \(O(N^2)\) time.

random_wgraph(+P, +N, +W, -WeightedGraph)
where \(P\) is a probability, unifies WeightedGraph with a random graph with ver-
tices \(1..N\) where each possible edge is included with probability \(P\) and random
weight in \(1..W\).

min_tree(+WeightedGraph, -Tree, -Cost)
is true if Tree is a minimum-Cost spanning tree of an undirected WeightedGraph
with cost Cost, if it exists. Using Kruskal’s algorithm.
10.53 Parsing and Generating XML—library(xml)

This is a package for parsing XML with Prolog, which provides Prolog applications with a simple “Document Value Model” interface to XML documents. A description of the subset of XML that it supports can be found at: http://www.binding-time.co.uk/xmlpl.html

The package, originally written by Binding Time Ltd., is in the public domain and unsupported. To use the package, enter the query:

\[
| ?- use_module(library(xml)).
\]

The package represents XML documents by the abstract data type `document`, which is defined by the following grammar:

\[
document ::= \{ \text{well-formed document } \}
\]

\[
xml(attributes, content)
\]

\[
| \{ \text{malformed document } \}
\]

\[
malformed(attributes, content)
\]

\[
attributes ::= []
\]

\[
| [\text{name=char-data}]attributes
\]

\[
content ::= []
\]

\[
| [\text{cterm}|content]
\]

\[
cterm ::= \text{pcdata(char-data)} \{ \text{text} \}
\]

\[
| \text{comment(char-data)} \{ \text{an XML comment} \}
\]

\[
| \{ \text{a Namespace} \}
\]

\[
namespace(\text{URI, prefix, element})
\]

\[
| \{ <\text{tag}>..</\text{tag}> encloses content or <\text{tag} />\text{content} \}
\]

\[
| \text{instructions(name, char-data)}
\]

\[
| \text{out_of_context(tag)} \{ \text{tag is not closed} \}
\]

\[
| \text{unparsed(char-data)} \{ \text{text that hasn’t been parsed} \}
\]

\[
\text{doctype}(\text{tag, doctype-id})
\]

\[
\text{tag ::= atom} \{ \text{naming an element} \}
\]

\[
\text{name ::= atom} \{ \text{not naming an element} \}
\]

\[
\text{URI ::= atom} \{ \text{giving the URI of a namespace} \}
\]

\[
\text{char-data ::= code list}
\]

\[
\text{doctype-id ::= public(char-data, char-data)}
\]

\[
| \text{public(char-data, dtd-literals)}
\]

\[
| \text{system(char-data)}
\]
system(char-data,dtd-literals)
| local
| local,dtd-literals
dtd-literals ::= []
| [dtd_literal(char-data)|dtd-literals]

The following predicates are exported by the package:

xml_parse(?Chars, ?Document)
xml_parse(?Chars, ?Document, +Options)

Either parses Chars, a code list, to Document, a document. Chars is not required to represent strictly well-formed XML. Or generates Chars, a code list, from Document, a document. If Document is not a valid document term representing well-formed XML, an exception is raised. In the second usage of the predicate, the only option available is format/1.

Options is a list of zero or more of the following, where Boolean must be true or false:

format(Boolean)
   Indent the element content (default true).

extended_characters(Boolean)
   Use the extended character entities for XHTML (default true).

remove_attribute_prefixes(Boolean)
   Remove namespace prefixes from attributes when it’s the same as the prefix of the parent element (default false).

xml_subterm(+Term, ?Subterm)

Unifies Subterm with a sub-term of Term, a document. This can be especially useful when trying to test or retrieve a deeply-nested subterm from a document.

xml_pp(+Document)

“Pretty prints” Document, a document, on the current output stream.

10.54 Zinc Interface—library(zinc)

MiniZinc is a free and open-source constraint modeling language, developed at Monash University in collaboration with Data61 Decision Sciences and the University of Melbourne. Models are compiled into FlatZinc, a solver input language that is understood by a wide range of solvers, including SICStus Prolog. See http://www.minizinc.org for more information.

This library provides an interpreter for FlatZinc programs (see Section 10.54.2 [FlatZinc], page 880), and also for MiniZinc programs (see Section 10.54.3 [MiniZinc], page 889). The library interface was inspired by the MiniZinc and FlatZinc libraries of The ECLiPSe Constraint Programming System. It is compatible with MiniZinc 2.5.5 distribution.
10.54.1 Prerequisites

Let \$MINIZINC_DIR\ denote the directory containing the MiniZinc distribution.

To add SICStus Prolog as a MiniZinc and FlatZinc solver, two additions to \$MINIZINC_DIR\ are required:

- Copy or symbolically link the SICStus specific global constraint definition directory \$SP_LIBRARY_DIR/zinc/globals). The copy or symbolic link should be named \$MINIZINC_DIR/share/minizinc/sicstus).
- Copy or symbolically link the configuration file \$SP_LIBRARY_DIR/zinc/sicstus.msc). The copy or symbolic link should be named \$MINIZINC_DIR/share/minizinc/solvers/sicstus.msc).

Finally, make sure that \$MINIZINC_DIR/bin\ is included in the environment variable PATH.

10.54.2 FlatZinc

The FlatZinc interpreter described here is based on “Specification of FlatZinc, version 1.6”, available at http://www.minizinc.org/specifications.html.

A FlatZinc program can be run directly using fzn_run_file/[1,2] and fzn_run_stream/[1,2], as well as with spfz, a simple command-line tool interface to fzn_run_file/[1,2] (for details, see Section 13.3 [too-spfz], page 1393). For example, a program for solving the 4 Queens problem, located in library('zinc/examples/queen4.fzn'), can be run by the following goal:

| ?- fzn_run_file(library('zinc/examples/queen4')).

or command:

% spfz \$SP_LIBRARY_DIR/zinc/examples/queen4

The following solution is then written on the current output stream:

q = array1d(1..4, [2, 4, 1, 3]);

----------

Note the ten consecutive dashes after the solution.

The following goal can be used to find all solutions:

| ?- fzn_run_file(library('zinc/examples/queen4'), [solutions(all)]).

or command:

% spfz \$SP_LIBRARY_DIR/zinc/examples/queen4 -a

The following solutions are then written on the current output stream:
Chapter 10: The Prolog Library

\[
q = \text{array1d}(1..4, [2, 4, 1, 3]);
\]
\[
q = \text{array1d}(1..4, [3, 1, 4, 2]);
\]

Note the ten consecutive equal signs after all solutions have been found.

FlatZinc programs are not intended to be written (or read) by humans, but can be generated by the minizinc command-line tool with the -c option, for example (queen.mzn and queen4.dzn can be found in library('zinc/examples')):

\[
\text{minizinc --solver sicstus -c queen.mzn queen4.dzn}
\]

The resulting FlatZinc program queen4.fzn can then be run as described above. minizinc without the -c option first generates a temporary FlatZinc file, then runs it, then deletes it:

\[
\text{minizinc --solver sicstus queen.mzn queen4.dzn}
\]

or, simplest of all:

\[
\text{minizinc --solver sicstus -D n=4 queen.mzn}
\]

It is also possible to just load a FlatZinc program into SICStus by fzn_load_file/2 and fzn_load_stream/2. The loaded FlatZinc program can then be processed further from within SICStus, e.g. by retrieving some FlatZinc variables using fzn_identifier/3 and posting additional library(clpfd) constraints or applying a Prolog labeling predicate on those variables.

Finally, it is possible to load and run MiniZinc programs directly from within SICStus by using the predicates described in Section 10.54.3 [MiniZinc], page 889.

10.54.2.1 Exported Predicates

The predicates described here operate on a data structure FznState representing a FlatZinc program and consisting of the following members:

- A table that maps identifiers of the FlatZinc program to Prolog terms.
- A list containing all domain variables of the FlatZinc program, except those with an is_defined_var annotation.
- A list containing all domain variables of the FlatZinc program that may be written on the current output stream.
- A goal representing the constraint part of the FlatZinc program.
- A goal representing the solve part of the FlatZinc program.
- A counter denoting the number of solutions found by the FlatZinc program.
This data structure can be constructed from a FlatZinc program by the predicates \texttt{fzn\textunderscore load\textunderscore stream/2} and \texttt{fzn\textunderscore load\textunderscore file/2} described next, or directly from a MiniZinc program (see Section 10.54.3 [MiniZinc], page 889).

\texttt{fzn\textunderscore load\textunderscore stream(+FznStream, -FznState)}

\textit{FznStream} is a FlatZinc input stream. \textit{FznState} is a FlatZinc state that is initialized with respect to \textit{FznStream}.

\textit{Exceptions:} Exceptions regarding errors in \textit{FznStream} (see Section 10.54.4 [Zinc Errors], page 895).

\texttt{fzn\textunderscore load\textunderscore file(+FznFile, -FznState)}

\textit{FznFile} is a FlatZinc file (extension defaults to .fzn). \textit{FznState} is a FlatZinc state that is initialized with respect to \textit{FznFile}. This predicate is just a wrapper around \texttt{fzn\textunderscore load\textunderscore stream/2} handling stream opening and closing.

\textit{Exceptions:}
- Exceptions related to the opening of \textit{FznFile} for reading.
- Exceptions regarding errors in \textit{FznFile} (see Section 10.54.4 [Zinc Errors], page 895).

Consider the following FlatZinc program for solving the 4 Queens problem located in \texttt{library('zinc/examples/queen4.fzn')}.

(Note that FlatZinc programs are not intended to be written (or read) by humans, but rather to be automatically generated. One method to generate FlatZinc programs is described in Section 10.54.3 [MiniZinc], page 889.)
int: n = 4;

array[1 .. 4] of var 1 .. 4: q::output_array([ 1 .. 4 ]);  

constraint int_lin_ne([ 1, -1 ], [ q[1], q[2] ], 1);
constraint int_ne(q[1], q[2]);

constraint int_lin_ne([ 1, -1 ], [ q[1], q[2] ], -1);
constraint int_lin_ne([ 1, -1 ], [ q[1], q[3] ], 2);
constraint int_ne(q[1], q[3]);

constraint int_lin_ne([ 1, -1 ], [ q[1], q[3] ], -2);
constraint int_lin_ne([ 1, -1 ], [ q[1], q[4] ], 3);
constraint int_ne(q[1], q[4]);

constraint int_lin_ne([ 1, -1 ], [ q[1], q[4] ], -3);
constraint int_lin_ne([ 1, -1 ], [ q[2], q[3] ], 1);
constraint int_ne(q[2], q[3]);

constraint int_lin_ne([ 1, -1 ], [ q[2], q[3] ], -1);
constraint int_lin_ne([ 1, -1 ], [ q[2], q[4] ], 2);
constraint int_ne(q[2], q[4]);

constraint int_lin_ne([ 1, -1 ], [ q[2], q[4] ], -2);
constraint int_lin_ne([ 1, -1 ], [ q[3], q[4] ], 1);
constraint int_ne(q[3], q[4]);

constraint int_lin_ne([ 1, -1 ], [ q[3], q[4] ], -1);

solve satisfy;

A FlatZinc state Queen4State representing the program above can be constructed by typing:
| ?- fzn_load_file(library('zinc/examples/queen4'), Queen4State).
The predicates presented next are used to query an already initialized FlatZinc state.

\[\text{fzn\_post(+FznState)}\]

Posts the constraints of the FlatZinc program represented by \(FznState\). May fail if the constraints are inconsistent.

\[\text{fzn\_solve(+FznState)}\]

Runs the solve and output parts of the FlatZinc program represented by \(FznState\) to find and display an (optimal) solution. Fails if the constraints of the FlatZinc program are inconsistent. Generates the next solution upon backtracking.

\[\text{fzn\_output(+FznState)}\]

Outputs the values of the variables in \(FznState\) that have been annotated with \text{output\_var/0} or \text{output\_array/1}.

*Exceptions:* An *instatiation error* if the output variables are not instantiated.

Consider again the FlatZinc program \texttt{queen4.fzn} described above and the following goal at the Prolog top level:

\[
| ?- \text{fzn\_load\_file(library('zinc/examples/queen4'), Queen4State), fzn\_post(Queen4State), fzn\_solve(Queen4State)}.\\
\]

The first line initializes \(Queen4State\) with respect to \texttt{queen4.fzn}. The second and third line posts the constraints of \texttt{queen4.fzn} and runs the solve and output parts of \texttt{queen4.fzn}, respectively. The following is written on the current output stream:

\[
q = \text{array1d(1..4, [2, 4, 1, 3])};\\
----------\\
\]

Upon backtracking the solve and output parts of \(Queen4State\) are rerun, which means that the following is written on the current output stream:

\[
q = \text{array1d(1..4, [3, 1, 4, 2])};\\
----------\\
\]

\[\text{fzn\_identifier(+FznState, +Id, -Value)}\]

\(FznState\) is a FlatZinc state initialized with respect to some FlatZinc program and \(Id\) is an identifier of the FlatZinc program. Unifies the FlatZinc value of \(Id\) with \(Value\) according to the following translation scheme:

- A \texttt{bool} is translated into a Prolog integer: \texttt{false} is translated into 0 and \texttt{true} is translated into 1.
- An \texttt{int} is translated into a Prolog integer.
- A \texttt{float} is translated into a Prolog float.
- An integer range or an integer set is translated into a library(clpfd) FD set term (see Section 10.10.9.3 [FD Set Operations], page 481).
A non-integer set is translated into a sorted Prolog list containing the (translated) elements of the set.

An array is translated into a Prolog list containing the (translated) elements of the array. Ordering is preserved such that the \( n \)th element of the array is the \( n \)th element of the list.

A \texttt{var int} is translated into a \texttt{library(clpfd)} domain variable (see Section 10.10.3 [CLPFD Interface], page 428).

A \texttt{var bool} is translated into a \texttt{library(clpfd)} domain variable with the domain \( 0..1 \) (see Section 10.10.3 [CLPFD Interface], page 428).

Exceptions: An existence error if \texttt{Id} is not an identifier of \texttt{FznState}.

\begin{verbatim}
ofzn_objective(+FznState, -Objective)
  FznState is a FlatZinc state initialized with respect to some FlatZinc program.
  Unifies Objective with a domain variable representing the FlatZinc objective.

  Exceptions: An existence error if there is no objective in FznState.
\end{verbatim}

A possible use of \texttt{fzn_identifier/3} is to post additional \texttt{library(clpfd)} constraints or to apply a Prolog labeling predicate on the FlatZinc variables. For example, given the 4 Queens problem in \texttt{queen4.fzn} described above, the following goal labels the variables to find all solutions:

\begin{verbatim}
| ?- use_module(library(clpfd)).
| ?- fzn_load_file(library('zinc/examples/queen4'), Queen4State),
   fzn_post(Queen4State),
   fzn_identifier(Queen4State, q, Q),
   findall(_, (labeling([], Q), fzn_output(Queen4State)), _).
\end{verbatim}

Given this goal, the following is written on the current output stream:

\begin{verbatim}
q = array1d(1..4, [2, 4, 1, 3]);
----------
q = array1d(1..4, [3, 1, 4, 2]);
----------
\end{verbatim}

To avoid symmetric solutions where the chess board is rotated 180 degrees, the following goal posts an additional symmetry breaking constraint on the first two queens:

\begin{verbatim}
| ?- fzn_load_file(library('zinc/examples/queen4'), Queen4State),
   fzn_post(Queen4State),
   fzn_identifier(Queen4State, q, Q),
   Q = [Q1, Q2|_], Q1 #< Q2,
   findall(_, (labeling([], Q), fzn_output(Queen4State)), _).
\end{verbatim}

Given this goal, the following is written on the current output stream:

\begin{verbatim}
q = array1d(1..4, [2, 4, 1, 3]);
----------
\end{verbatim}
Note that, now, only the first one of the previous two solutions is displayed.

The following two predicates can be used to run a FlatZinc program in one go. They both take as optional argument a list `Options`, which can be used to change the default behavior of the execution. This list may contain zero or more of the following:

- `search(Method)` since release 4.3
  where `Method` must be one of the atoms `bab` and `restart`. Tells the solver which optimization algorithm to use: branch-and-bound (the default), or to restart the search each time a new solution is found. The corresponding `spfz` option is `-search Method`.

- `free_search(Boolean)` since release 4.7.0
  where `Boolean` must be `true` or `false` (default). If `true`, tells the solver to ignore any search annotation. The corresponding `spfz` option is `-f`.

- `solutions(NumberOfSolutions)` where `NumberOfSolutions` must be an integer greater than zero or the atom `all`. Describes the number of solutions to search for; the default is 1. The corresponding `spfz` options are `-n N` and `-a`.

- `output(File)` where `File` must be the name of a writable file. Causes any output written on the current output stream to be directed to `File`. The corresponding `spfz` option is `-o File`.

- `ozn_file(File)` since release 4.2.3
  where `File` must be the name of an existing file, containing the MiniZinc output commands that `minizinc` should use. If not given, then the solutions will be printed unformatted.

- `statistics(Boolean)` where `Boolean` must be `true` or `false` (default). The corresponding `spfz` option is `-s`. If `true`, a block of statistics is written on the current output stream at the end of execution. Each line of the block has the format:

  ```
  %%%mzn-stat: name=value
  ```

  The block is terminated by a line:

  ```
  %%%mzn-stat-end
  ```

  The `name` and its meaning is one of the following:

  - `failures` The number of times a failure was encountered during search.
  - `variables` The number of domain variables created.
  - `propagators` The number of propagators created.
  - `propagations` The number of propagator invocations.
  - `initTime` Initialization time (in seconds).
solveTime
Solving time (in seconds).

timeout(Time)
where Time should be an integer greater than zero. Stops the computation if it has not finished before Time milliseconds has elapsed. In any event, the best solution found so far is reported. The corresponding spfz option is \(-t\) Time.

\texttt{fzn\_run\_stream}(\texttt{FznStream})
\texttt{fzn\_run\_stream}(\texttt{FznStream}, \texttt{Options})
\texttt{FznStream} is a FlatZinc input stream and \texttt{Options} is a list of options as described above. Performs the following steps:
1. Loads the FlatZinc program (\texttt{fzn\_load\_stream/2}), initializing a FlatZinc state.
2. Posts the constraints of the FlatZinc program (\texttt{fzn\_post/1}).
3. Runs the solve part of the FlatZinc program (\texttt{fzn\_solve/1}).
4. Outputs the values of the variables that have been annotated with \texttt{output\_var/0} or \texttt{output\_array/1}.

The two final steps are repeated until the number of solutions as specified in \texttt{Options} have been found or until no more solutions can be found. At this point, if the whole search space have been explored, then ten consecutive equal signs are output on a separate line.

\textit{Exceptions:}
- A \textit{type error} if the number of solutions to search for is not greater than zero nor the atom \texttt{all}.
- Exceptions regarding errors in \texttt{FznStream} (see Section 10.54.4 [Zinc Errors], page 895).

\texttt{fzn\_run\_file}(\texttt{FznFile})
\texttt{fzn\_run\_file}(\texttt{FznFile}, \texttt{Options})
\texttt{FznFile} is a FlatZinc program file (extension defaults to .fzn) and \texttt{Options} is a list of options as described above. This predicate is just a wrapper around \texttt{fzn\_run\_stream/[1,2]} handling stream opening and closing.

\textit{Exceptions:}
- Exceptions related to the opening of \texttt{FznFile} for reading.
- A \textit{type error} if the number of solutions to search for is not greater than zero nor the atom \texttt{all}.
- Exceptions regarding errors in \texttt{FznFile} (see Section 10.54.4 [Zinc Errors], page 895).

The next predicate can be used to write the constraints of a FlatZinc program to a file, in the format of \texttt{library(clpfd)}. 
fzn_dump(+FznState, +File)
fzn_dump(+FznState, +Options, +File)

_FznState_ is a FlatZinc state initialized with respect to some FlatZinc program and _File_ is a writable file (extension defaults to `.pl`). Writes the constraints of _FznState_ to _File_ in the format of `library(clpfd)`.

_Options_ is a list containing zero or more of the following (currently, this is the only available option):

`variables(ListOfVarDef)`

where _ListOfVarDef_ is a list of elements of the form _Id_= _Var_ where _Id_ is a FlatZinc identifier and _Var_ is a Prolog variable. Means that _Var_ is unified with the value of _Id_ after the FlatZinc program is loaded and that _Id_= _Var_ is included in a list of arguments to `query/1` that is written to _File_. Default is `ListOfVarDef= [vars=Vars]`, with the meaning that _Vars_ is a list containing all variables of the FlatZinc state, in the order they were introduced.

_Exceptions:_ Exceptions related to the opening of _File_ for writing.

Consider again the FlatZinc program `queen4.fzn` described above and the following goal at the Prolog top level:

```prolog
| ?- fzn_load_file(library('zinc/examples/queen4'), Queen4State),
   fzn_dump(Queen4State, [variables([q=Q])], queen4).
```

The file `queen4.pl` then contains the following:
queen4.pl

:- use_module(library(clpfd)).

query([q=[A,B,C,D]]) :-
  domain([A,B,C,D], 1, 4),
  C#\=D,
  B#\=D,
  B#\=C,
  A#\=D,
  A#\=C,
  A#\=B,
  scalar_product([1,-1], [C,D], #\=, 1),
  scalar_product([1,-1], [C,D], #\=, -1),
  scalar_product([1,-1], [B,D], #\=, 2),
  scalar_product([1,-1], [B,D], #\=, -2),
  scalar_product([1,-1], [B,C], #\=, 1),
  scalar_product([1,-1], [B,C], #\=, -1),
  scalar_product([1,-1], [A,D], #\=, 3),
  scalar_product([1,-1], [A,D], #\=, -3),
  scalar_product([1,-1], [A,C], #\=, 2),
  scalar_product([1,-1], [A,C], #\=, -2),
  scalar_product([1,-1], [A,B], #\=, 1),
  scalar_product([1,-1], [A,B], #\=, -1).

10.54.3 MiniZinc

The predicates described here make it possible to load and run MiniZinc programs directly from within SICStus. In this way, the predicates described here are essentially wrappers for the predicates described in Section 10.54.2 [FlatZinc], page 880, providing a more high-level interface.

10.54.3.1 Exported Predicates

The following predicates all take as (optional) argument a list Options, which can be used to change the default behavior of the execution. This list may contain zero or more of the following:

data_file(MznDatFile)
  where MznDatFile must be a MiniZinc data file. Means that MznDatFile is passed to minizinc -c.

parameters(ListOfParDef)
  where ListOfParDef is a list of elements of the form Id=Value where Id is a MiniZinc identifier and Value is a MiniZinc value. Means that all elements are written to a temporary file, which is passed to minizinc -c.

post(Boolean)
  where Boolean must be true or false. If true (the default), then the constraints of the MiniZinc program are posted directly and a separate call to
\texttt{fzn\_post/1} (see Section 10.54.2.1 [FlatZinc Exported Predicates], page 881) is not necessary. (Only usable with \texttt{mzn\_load\_file/3} and \texttt{mzn\_load\_model/3}.)

\texttt{search(\textit{Method})} \texttt{\since{} release 4.3}
where \textit{Method} must be one of the atoms \texttt{bab} and \texttt{restart}. Tells the solver which optimization algorithm to use: branch-and-bound (the default), or to restart the search each time a new solution is found. (Only usable with \texttt{mzn\_run\_file/2} and \texttt{mzn\_run\_model/2}.)

\texttt{free\_search(\textit{Boolean})} \texttt{\since{} release 4.7.0}
where \textit{Boolean} must be \texttt{true} or \texttt{false} (default). If \texttt{true}, tells the solver to ignore any search annotation. (Only usable with \texttt{mzn\_run\_file/2} and \texttt{mzn\_run\_model/2}.)

\texttt{solutions(\textit{NumberOfSolutions})}
where \textit{NumberOfSolutions} must be an integer greater than zero or the atom \texttt{all}. Describes the number of solutions to search for, default is 1. (Only usable with \texttt{mzn\_run\_file/2} and \texttt{mzn\_run\_model/2}.)

\texttt{output(\textit{File})}
where \textit{File} must be the name of a writable file. Causes any output written on the current output stream to be directed to \textit{File}. (Only usable with \texttt{mzn\_run\_file/2} and \texttt{mzn\_run\_model/2}.)

\texttt{fzn\_file(\textit{File})} \texttt{\since{} release 4.2.3}
where \textit{File} must be the name of a writable file. The translated FlatZinc program will be written to the given file. Otherwise, a temporary file will be used and erased afterwards.

\texttt{ozn\_file(\textit{File})} \texttt{\since{} release 4.2.3}
where \textit{File} must be the name of a writable file. The MiniZinc output commands will be written to the given file. Otherwise, a temporary file will be used and erased afterwards.

\texttt{optimise(\textit{Boolean})} \texttt{\since{} release 4.2.3}
\texttt{optimize(\textit{Boolean})} \texttt{\since{} release 4.2.3}
where \textit{Boolean} must be \texttt{true} (the default) or \texttt{false}. If \texttt{false}, then \texttt{--no-optimize} is passed to \texttt{minizinc -c}.

\texttt{option(\textit{Value})} \texttt{\since{} release 4.5}
Causes an extra option \textit{Value} to be passed to the MiniZinc driver. (Only usable with \texttt{mzn\_run\_file/2} and \texttt{mzn\_run\_model/2}.)

\texttt{statistics(\textit{Boolean})}
where \textit{Boolean} must be \texttt{true} or \texttt{false} (default). If \texttt{true}, then the following statistics are written on the current output stream (see the built-in \texttt{statistics/[0,2]} and \texttt{fd\_statistics/[0,2]} of \texttt{library(clpfd)} for more detailed information on their meaning):

\texttt{runtime} \text{Total running time (milliseconds), including parsing the FlatZinc program.}
Chapter 10: The Prolog Library 891

solvetime
Running time (milliseconds) for posting the constraints and performing the search.
solutions
The number of solutions found.
constraints
The number of constraints created.
backtracks
The number of times a contradiction was found by a domain being wiped out, or by a global constraint signaling failure.
prunings
The number of times a domain was pruned.
(Only usable with mzn_run_file/2 and mzn_run_model/2.)
timeout(Time)
where Time should be an integer greater than zero. Stops the computation if it has not finished before Time milliseconds has elapsed. In any event, the best solution found so far is reported. Measuring starts after the call to minizinc -c has finished. (Only usable with mzn_run_file/2 and mzn_run_model/2.)
variables(ListOfVarDef)
where ListOfVarDef is a list of elements of the form Id=Var where Id is a MiniZinc identifier and Var is a Prolog variable. Means that Var is unified with the value of Id after the MiniZinc program is loaded. (Only usable with mzn_load_file/3 and mzn_load_model/3).

The first two predicates can be used to run a MiniZinc program in one go.

mzn_run_file(+MznFile)
mzn_run_file(+MznFile, +Options)
MznFile is a MiniZinc program file and Options is a list of options as described above. Runs the MiniZinc program on MznFile, and writes the result onto the file given in the option/1 option if given, or onto the current output stream otherwise. This is done by first calling minizinc -c, and then interpreting its output with fzn_run_stream/[1,2] (see Section 10.54.2.1 [FlatZinc Exported Predicates], page 881). Fails if the constraints of the MiniZinc program are inconsistent.

Exceptions:
- Exceptions related to the opening of MznFile for reading.
- A system error if mzn2fzn is unsuccessful. This error will include any information produced by mzn2fzn on its standard error stream.
- A type error if the number of solutions to search for is not greater than zero nor the atom all.
- Exceptions regarding errors in MznFile (see Section 10.54.4 [Zinc Errors], page 895) although these are most probably already handled by mzn2fzn.
mzn_run_model(+MznModel)
mzn_run_model(+MznModel, +Options)

*MznModel* is a MiniZinc program specified by a list of strings as explained below and *Options* is a list of options as described above. Runs the MiniZinc program *MznModel*, and writes the result onto the file given in the *option/1* option if given, or onto the current output stream otherwise. This is done by first calling *mzn2fzn* and interpreting its output with *fzn_run_stream/[1,2]* (see Section 10.54.2.1 [FlatZinc Exported Predicates], page 881). The MiniZinc program specification *MznModel* must be a list of strings (list of character codes) where each element must specify one line of the MiniZinc program. For example, a MiniZinc program for the N Queens problem can be specified as follows:

```prolog
NQueens = ["int: n;", "array [1..n] of var 1..n: q;", "constraint forall (i in 1..n, j in i+1..n)", 
"solve satisfy;", 
"output ['"A solution to the ", show(n), 
"Queens problem: ", show(q), 
"\n'];"]
```

**Note** that backslashes and double quotes must be escaped with an additional backslash.

**Exceptions:**

- A *system error* if *mzn2fzn* is unsuccessful. This error will include any information produced by *mzn2fzn* on its standard error stream.
- A *type error* if the number of solutions to search for is not greater than zero nor the atom *all*.
- Exceptions regarding errors in *MznModel* (see Section 10.54.4 [Zinc Errors], page 895) although these are most probably already handled by *mzn2fzn*.

Consider the following MiniZinc program for solving the N Queens problem located in *library('zinc/examples/queen.mzn')*:
queen.mzn

```prolog
int: n;

array [1..n] of var 1..n: q;

constraint
    forall (i in 1..n, j in i+1..n) ( 
        q[i] != q[j] /
        q[i] + i != q[j] + j /
        q[i] - i != q[j] - j
    );

solve satisfy;

output ["A solution to the ", show(n), " Queens problem: ", show(q), ",n"];```

Consider now the following goal at the Prolog top level:

```prolog
| ?- mzn_run_file(library('zinc/examples/queen'),
    [data_file(library('zinc/examples/queen4.dzn'))]).
```

Since `library('zinc/examples/queen4.dzn')` contains the single line

`n = 4;`

the following is written on the current output stream:

```
A solution to the 4 Queens problem: [2, 4, 1, 3]
----------
```

The initialization `n = 4` can also be passed using the `parameter/1` option. So the following goal is equivalent to the one above:

```prolog
| ?- mzn_run_file(library('zinc/examples/queen'), [parameters([n=4])]).
```

Finally, the following goal finds all solutions to the 4 Queens problem:

```prolog
| ?- mzn_run_file(library('zinc/examples/queen'),
    [parameters([n=4]), solutions(all)]).
```

Given this goal, the following is written on the current output stream:

```
A solution to the 4 Queens problem: [2, 4, 1, 3]
----------
A solution to the 4 Queens problem: [3, 1, 4, 2]
----------
==========
```

The next two predicates can be used to construct a FlatZinc state (see Section 10.54.2.1 [FlatZinc Exported Predicates], page 881).
mzn_load_file(+MznFile, -FznState)
mzn_load_file(+MznFile, +Options, -FznState)

MznFile is a MiniZinc program file and Options is a list of options as described above. Initializes a FlatZinc state FznState with respect to MznFile. May fail if post(true) and the constraints are inconsistent.

Exceptions:
- Exceptions related to the opening of MznFile for reading.
- A system error if mzn2fzn is unsuccessful. This error will include any information produced by mzn2fzn on its standard error stream.
- An existence error if an Id of the variables/1 option is not an identifier of FznState.
- Exceptions regarding errors in MznFile (see Section 10.54.4 [Zinc Errors], page 895) although these are most probably already handled by mzn2fzn.

mzn_load_model(+MznModel, -FznState)
mzn_load_model(+MznModel, +Options, -FznState)

MznModel is a MiniZinc program specified by a list of strings as explained for mzn_run_model/[1,2] above and Options is a list of options as described above. Initializes a FlatZinc state FznState with respect to MznModel. May fail if post(true) and the constraints are inconsistent.

Exceptions:
- A system error if mzn2fzn is unsuccessful. This error will include any information produced by mzn2fzn on its standard error stream.
- An existence error if an Id of the variables/1 option is not an identifier of FznState.
- Exceptions regarding errors in MznModel (see Section 10.54.4 [Zinc Errors], page 895) although these are most probably already handled by mzn2fzn.

The following Prolog goal constructs a FlatZinc state representing the 4 Queens problem:

```
| ?- mzn_load_file(library('zinc/examples/queen'),
                   [parameters([n=4])], Queen4State).
```

See Section 10.54.2.1 [FlatZinc Exported Predicates], page 881, for more information on FlatZinc states and how they can be queried. A very useful option to mzn_load_file/3 and mzn_load_model/3 is the variables/1 option, which can be used to unify values of MiniZinc identifiers with Prolog variables (this option can be used in place of several calls to fzn_identifier/3). For example, the following goal posts an additional symmetry breaking constraint and labels the variables using a Prolog goal that finds all remaining solutions to the 4 Queens problem:

```
| ?- mzn_load_file(library('zinc/examples/queen'),
                   [parameters([n=4]), variables([q=Q])],
                   Queen4State),
                   Q = [Q1, Q2|_], Q1 #< Q2,
                   findall(_, (labeling([], Q), fzn_output(Queen4State)), _).
```
Given this goal, the following is written on the current output stream:

```prolog
q = array1d(1..4, [2, 4, 1, 3]);
```

Finally, the following predicate can be used to translate a MiniZinc file to a FlatZinc by a direct call to `mzn2fzn`.

```prolog
mzn_to_fzn(+MznFile, +FznFile)
mzn_to_fzn(+MznFile, +Options, +FznFile)
```

`MznFile` is a MiniZinc program file and `Options` is a list of options as described above. Calls `mzn2fzn`, whose result is written to `FznFile`.

Exceptions:
- Exceptions related to the opening of `MznFile` for reading as well as the opening of `FznFile` for writing.
- A **system error** if `mzn2fzn` is unsuccessful. This error will include any information produced by `mzn2fzn` on its standard error stream.
- Exceptions regarding errors in `MznFile` (see Section 10.54.4 [Zinc Errors], page 895) although these are most probably already handled by `mzn2fzn`.

### 10.54.4 Error Messages

The following is a list of exceptions that may be generated by the predicates described in Section 10.54.2.1 [FlatZinc Exported Predicates], page 881, and in Section 10.54.3.1 [MiniZinc Exported Predicates], page 889, when there is an error in the FlatZinc or MiniZinc input.

- A **syntax error** occurs when the parser cannot continue. For example, the FlatZinc code:

  ```prolog
  array[1..2] of int a = [1, 2];
  ```

  generates the following error (since there must be a colon between `int` and `a`):

  ```prolog
  ! Item ending on line 1:
  ! Syntax error
  ! expected ':' but found 'ident(a)'
  ```

  The line number indicates the ending line of the item containing the error. **Note that** this means that the error may be on a preceding line, if the item occupies several lines.

- A **consistency error** occurs when the same identifier is used multiple times. For example, the FlatZinc code:

  ```prolog
  bool : b = false;
  bool : b = true;
  ```

  generates the following error:

  ```prolog
  ! Item ending on line 2:
  ! Consistency error: 'b' is already defined
  ! previous definition of b was 'bool : b = false'
  ! cannot redefine b to 'bool : b = true'
  ```
• An *existence error* occurs when an identifier or a constraint is used without being previously defined. For example, the FlatZinc code:

```prolog
bool : b = a;
```
may generate the following error:

```
! Item ending on line 2:
! Existence error
! 'a' is not defined
```

Another example, the FlatZinc code:

```prolog
var int : a;
var int : b;
constraint distance(a, b, 1);
```
may generate the following error:

```
! Item ending on line 4:
! Existence error
! 'distance/3' is not defined
```

• A *type error* occurs when a value is of the wrong type. For example, the FlatZinc code:

```prolog
var float : f;
```
generates the following error (since only finite domain integer variables are supported):

```
! Item ending on line 2:
! Type error
! 'f' must be a member of 'int'
```

Another example, the FlatZinc code:

```prolog
array[1..2] of float : a = [2.1, 3];
```
generates the following error (since an array of floats cannot contain integers):

```
! Item ending on line 2:
! Type error
! '3' must be a member of 'float'
```

A type error also occurs when an array index is out of bounds. For example, the FlatZinc code:

```prolog
array[1..2] of int : a = [1, 2];
int : i = a[3];
```
generates the following error:

```
! Item ending on line 3:
! Type error in array index
! index evaluates to 3 but must be in 1..2
```

### 10.54.5 Limitations

**Domain variables**

Only variables with finite integer domains are supported. This includes Boolean variables, which are considered finite integer domain variables with the domain 0..1. Domain variables declared to be of type `var int` are initially given the finite integer domain `inf..sup`, and are given maximally wide bounded domains.
before any search is performed on them, as well as before certain constraints that demand bounded domains are posted on them.

**Ground set values**

Although set variables are not supported, ground set values are. For example, the MiniZinc global constraint `sum_pred/4` takes as second argument an array of such ground set values.

**Solve annotations**

- The solve annotations currently recognized are `bool_search`, `int_search`, and `seq_search`.
- The FlatZinc specification describes several exploration strategies. Currently, the only recognized exploration strategies are `complete` and `fail`.
- Variables not included in any solve or `is_defined_var` annotation are labeled with a default first-fail domain splitting heuristic. This corresponds to `labeling/2` of `library(clpfd)` with the option list `[ff, bisect]`.
- The choice method `indomain_random` as described in the FlatZinc specification uses `random_member/2` of `library(random)`. The random generator of SICStus is initialized using the same seed on each start up, meaning that the same sequence will be tried for `indomain_random` on each start up. This behavior can be changed by setting a different random seed using `setrand/1` of `library(random)`.

**Constraint annotations**

Constraint annotations of the form `domain`, `bounds`, and `value` are recognized by the FlatZinc interpreter. Any other constraint annotation is ignored.

**Variable annotations**

The following variable annotations are recognized. Any other variable annotation is ignored:

- `output_var` since release 4.2
  the variable may be written on the current output stream.
- `output_array` since release 4.2
  the variable array may be written on the current output stream.
- `is_defined_var`
  the variable will not be considered in any default labeling (such as when the search annotations do not include all variables)
11 Prolog Reference Pages

11.1 Reading the Reference Pages

11.1.1 Overview
The reference pages for SICStus Prolog built-in predicates conform to certain conventions concerning

- mode annotations
- predicate annotations
- argument types

These are particularly important in utilizing the Synopsis and Arguments fields of each reference page. The Synopsis field consists of the goal template(s) with mode annotations and a brief description of the purpose of the predicate. For example, consider this excerpt from the reference page for assert/[1,2]:

**Synopsis**
These predicates add a dynamic clause, `Clause`, to the Prolog database. They optionally return a database reference in `Ref`:

assert(+Clause)

assert(+Clause, -Ref)

It is undefined whether `Clause` will precede or follow the clauses already in the database.

The Arguments field lists, for each meta-variable name in the template, its argument type, (e.g. callable), a brief description (sometimes omitted), and an indication (‘:’) if it does module name expansion. For example,

**Arguments**
:Clause       callable A valid dynamic Prolog clause.
Ref           db_reference a database reference, which uniquely identifies the newly asserted Clause.

11.1.2 Mode Annotations
The mode annotations are useful to tell whether an argument is input or output or both. They also describe formally the instantiation pattern to the call that makes the call to the built-ins determinate.

The mode annotations in the above example are ‘+’ and ‘-’. Following is a complete description of the mode annotations you will find in the reference pages:

‘+’ Input argument. This argument will be inspected by the predicate, and affects the behavior of the predicate. An exception is raised if the argument is not of
the expected type. Note that an input argument can be an unbound variable in some cases.

`-` Output argument. This argument is unified with the output value of the predicate. An output argument is only tested to be of the same type as the possible output value if this is prescribed by the ISO standard, or if such testing is deemed helpful to the user.

`?` An argument that could be either input or output. This mode annotation is normally only used for predicates that behave as pure relations and do not type test their arguments.

If the synopsis of a predicate has more than one mode declaration, then the first (the topmost) that satisfies the types (of a goal instance) is the one to be applied (to that goal instance).

All built-in predicates of arity zero are determinate (with the exception of repeat/0).

For input arguments, an exception will be raised if the argument is not of the specified type.

For output arguments, an exception might be raised if the argument is nonvar, and not of the specified type. The generated value of the argument will be of the specified type.

### 11.1.3 Predicate Annotation

This section describes the annotations of predicates and how they are indicated in the reference pages for predicates of each given annotation. The annotations appear to the right of the title of the reference page.

**hookable** The behavior of the predicate can be customized/redefined by defining one or more hooks. The mode and type annotations of a hookable predicate might not be absolute, since hooks added by the user can change the behavior.

**hook** The predicate is user defined, and is called by a hookable builtin. Typically, it is undefined initially, belongs to the user module, and if defined by the user, it is best defined as multife, so that new clauses can be added by different software modules. For a hook, the mode and type annotations should be seen as guidelines to the user who wants to add his own hook; they describe how the predicate is used by the system.

**declaration** You cannot call these directly but they can appear in files as ‘:- declaration’ and give information to the compiler. The goal template is preceded by ‘:-’ in the Synopsis.

**development** A predicate that is defined in the development system only, i.e. not in runtime systems.

**ISO** A predicate that is part of the ISO Prolog Standard.
**deprecated**

A predicate that is not recommended in new code and that could be withdrawn in a future release.

**Meta-predicates and operators** are recognizable by the implicit conventions described below.

- **Meta-predicates** are predicates that need to assume some module. The reference pages of these predicates indicate which arguments are in a module expansion position by prefixing such arguments by `:` in the **Arguments** field. That is, the argument can be preceded by a module prefix (an **atom** followed by a colon). For example:

  ```prolog
  assert(mod:a(1), Ref)
  ```

  If no module prefix is supplied, then it will implicitly be set to the calling module. If the module prefix is a variable, then an instantiation error will be raised. If it is not an atom, then a type error will be raised. So in any meta-predicate reference page the following exceptions are implicit:

  **Exceptions**

  - **instantiation_error**
    - A module prefix is written as a variable.
  - **type_error**
    - A module prefix is not an atom.

- Whenever the name of a built-in predicate is defined as **operator**, the name is presented in the **Synopsis** as an operator, for example

  ```prolog
  :- initialization +Goal
  +Term1 @> +Term2
  ```

  It is thus always possible to see if a name is an operator or not. The predicate can, of course, be written using the canonical representation, even when the name is an operator. Thus (A) and (B) can be written as (C) and (D), respectively:

  ```prolog
  :- initialization(+Goal)  \( \text{(C)} \)
  @>(+Term1, +Term2) \( \text{(D)} \)
  ```

### 11.1.4 Argument Types

The argument section describes the type/domain of each argument of a **solution** to the given predicate. That is, for a predicate to succeed, it must be possible to instantiate the given argument to a term of the described type/domain.

If it is a `+` argument, then the predicate always tests if the argument is of the right type/domain. Usually, input arguments must also be instantiated to some extent. Such details are documented for each input argument.

Many built-in and library predicates take an `+Options` argument, which must be given as a proper list of terms specifying what the predicate should do, typically as the last argument. As a general rule for such option lists, if the same option occurs more than once, then the last occurrence overrides previous ones.
11.1.4.1 Simple Types

The simple argument types are those for which type tests are provided. They are summarized in Section 11.2.23 [mpg-top-typ], page 922.

If an output argument is given the type var, then it means that that argument is not used by the predicate in the given instantiation pattern.

11.1.4.2 Extended Types

Following is a list of argument types that are defined in terms of the simple argument types. This is a formal description of the types/domains used in the Arguments sections of the reference pages for the built-ins. The rules are given in BNF (Backus-Naur form).

\[
\begin{align*}
bbkey & ::= \text{atom} \mid \text{integer} \{\text{where the integer is small}\} \\
stream\_object & ::= \text{term} \{\text{as defined in Section 4.6.7.1 [ref-iov-sfh-sob], page 109,}\} \\
term & ::= \{\text{any Prolog term}\} \\
list\ of\ \text{Type} & ::= [] \mid [\text{Type} \mid \text{list}\ of\ \text{Type}] \\
var\ or\ \text{Type} & ::= \text{var} \mid \text{Type} \\
one\ of\ [\text{Element} \mid \text{Rest}] & ::= \text{Element} \mid \text{one}\ of\ \text{Rest} \\
arity & ::= \{\text{an integer X in the range 0..255}\} \\
byte & ::= \{\text{an integer X in the range 0..255}\} \\
char & ::= \{\text{an atom consisting of a single character}\} \\
chars & ::= \text{list}\ of\ \text{char} \\
code & ::= \{\text{an integer X }\geq 0\} \\
codes & ::= \text{list}\ of\ \text{code} \\
order & ::= \{\text{an atom, one of }[<,=,>]\} \\
pair & ::= \text{term}\-\text{term} \\
simple\_pred\_spec & ::= \text{atom}/\text{arity} \\
pred\_spec & ::= \text{simple}\_pred\_spec \mid \text{atom:pred}\_spec \\
pred\_spec\_forest & ::= [] \mid \text{pred}\_spec \\
& \quad \mid [\text{pred}\_spec\_forest \mid \text{pred}\_spec\_forest] \\
& \quad \mid \text{pred}\_spec\_forest, \text{pred}\_spec\_forest \\
pred\_spec\_tree & ::= \text{pred}\_spec \mid \text{list}\ of\ \text{pred}\_spec \\
foreign\_spec & ::= \text{callable} \{\text{all arguments being foreign}\_arg\} \\
foreign\_arg & ::= +\text{interf}\_arg\_type \mid -\text{interf}\_arg\_type \mid [-\text{interf}\_arg\_type] \\
interp\_arg\_type & ::= \text{integer} \mid \text{float} \mid \text{atom} \\
& \quad \mid \text{term} \mid \text{codes} \mid \text{string} \mid \text{address} \mid \text{address}(\text{atom}) \\
& \quad \{\text{see the description in Section 6.2.3 [Conversions between Prolog Arguments and C Types], page 293,}\} \\
file\_spec & ::= \text{atom} \mid \text{atom(file}\_spec) \\
expr & ::= \{\text{everything that is accepted as second argument to is/2;} \\
& \quad \text{see the description of arithmetic expressions in Section 4.7.5 [refari-aex], page 120.}\}
\end{align*}
\]
11.1.5 Exceptions

The Exceptions field of the reference page consists of a list of exception type names, each followed by a brief description of the situation that causes that type of exception to be raised. The following example comes from the reference page for `assert/[1,2]`:

**Exceptions**

instantiation_error

If `Head` (in `Clause`) or `M` is uninstantiated.

type_error

If `Head` is not of type callable, or if `M` is not an atom, or if `Body` is not a valid clause body.

For input arguments, an exception will be raised if the argument is not of the specified type.

For output arguments, an exception might be raised if the argument is nonvar, and not of the specified type. The generated value of the argument will be of the specified type.

11.1.6 Other Fields

The Backtracking field, if included, describes how the predicate behaves on backtracking. If this field is omitted, then the predicate is determinate (succeeds at most once).

The See Also field contains cross references to related predicates and/or manual sections.

Reference pages may also include Comments, Examples, and Tips fields, when appropriate.

11.2 Topical List of Prolog Built-Ins

Following is a complete list of SICStus Prolog built-in predicates, arranged by topic. A predicate may be included in more than one list.

11.2.1 All Solutions

`?X :- P` there exists an `X` such that `P` is provable (used in `setof/3` and `bagof/3`)

`bagof(?X, :P, -B)`

`B` is the bag of instances of `X` such that `P` is provable

`findall(?T, :G, -L)`

`ISO`


`L` is the list of all solutions `T` for the goal `G`, concatenated with `R` or with the empty list

`setof(?X, :P, -S)`

`ISO`

`S` is the set of instances of `X` such that `P` is provable

11.2.2 Arithmetic

`-Y is +X` `ISO`

`Y` is the value of arithmetic expression `X`
\[ +X =:= +Y \]  
the results of evaluating terms \(X\) and \(Y\) as arithmetic expressions are equal.

\[ +X =\not=} +Y \]  
the results of evaluating terms \(X\) and \(Y\) as arithmetic expressions are not equal.

\[ +X < +Y \]  
the result of evaluating \(X\) as an arithmetic expression is less than the result of evaluating \(Y\) as an arithmetic expression.

\[ +X \geqslant +Y \]  
the result of evaluating \(X\) as an arithmetic expression is not less than the result of evaluating \(Y\) as an arithmetic expression.

\[ +X > +Y \]  
the result of evaluating \(X\) as an arithmetic expression \(X\) is greater than the result of evaluating \(Y\) as an arithmetic expression.

\[ +X \leqslant +Y \]  
the result of evaluating \(X\) as an arithmetic expression is not greater than the result of evaluating \(Y\) as an arithmetic expression.

### 11.2.3 Character I/O

**at_end_of_line**  
\[ \text{at_end_of_line}(+S) \]  
testing whether at end of line on input stream \(S\)

**at_end_of_stream**  
\[ \text{at_end_of_stream}(+S) \]  
testing whether end of file is reached for the input stream \(S\)

**flush_output**  
\[ \text{flush_output}(+S) \]  
flush the output buffer for stream \(S\)

**get_byte**  
\[ \text{get_byte}(-C) \]  
\[ \text{get_byte}(+S,-C) \]  
\(C\) is the next byte on binary input stream \(S\)

**get_char**  
\[ \text{get_char}(-C) \]  
\[ \text{get_char}(+S,-C) \]  
\(C\) is the next character atom on text input stream \(S\)

**get_code**  
\[ \text{get_code}(-C) \]  
\[ \text{get_code}(+S,-C) \]  
\(C\) is the next character code on text input stream \(S\)

**nl**  
\[ \text{nl}(+S) \]  
send a newline to stream \(S\)

**peek_byte**  
\[ \text{peek_byte}(+C) \]  
\[ \text{peek_byte}(+S,+C) \]  
looks ahead for next input byte on the binary input stream \(S\)
peek_char(+C) \[ ISO \]
peek_char(+S,+C) \[ ISO \]
looks ahead for next input character atom on the text input stream S

peek_code(+C) \[ ISO \]
peek_code(+S,+C) \[ ISO \]
looks ahead for next input character code on the text input stream S

put_byte(+C) \[ ISO \]
put_byte(+S,+C) \[ ISO \]
write byte C to binary stream S

put_char(+C) \[ ISO \]
put_char(+S,+C) \[ ISO \]
write character atom C to text stream S

put_code(+C) \[ ISO \]
put_code(+S,+C) \[ ISO \]
write character code C to text stream S

skip_byte(+C) \[ ISO \]
skip_byte(+S,+C) \[ ISO \]
skip input on binary stream S until after byte C

skip_char(+C) \[ ISO \]
skip_char(+S,+C) \[ ISO \]
skip input on text stream S until after char C

skip_code(+C) \[ ISO \]
skip_code(+S,+C) \[ ISO \]
skip input on text stream S until after code C

skip_line \[ ISO \]
skip_line(+S) \[ ISO \]
skip the rest input characters of the current line on the input stream S

11.2.4 Control

:P, Q \[ ISO \]
prove P and Q

:P; Q \[ ISO \]
prove P or Q

+M: P \[ ISO \]
call P in module M

:P->:Q; :R \[ ISO \]
if P succeeds, prove Q; if not, prove R

:P->:Q \[ ISO \]
if P succeeds, prove Q; if not, fail

! \[ ISO \]
cut any choices taken in the current procedure
\+ :P
  goal \( P \) is not provable

\(?X \leftarrow :P\)
  there exists an \( X \) such that \( P \) is provable (used in \texttt{setof/3} and \texttt{bagof/3})

\( \text{block} :P \)
  declaration
  declaration that predicates specified by \( P \) should block until sufficiently instantiated

call(\( :P \))
  ISO

call(\( :P,\ldots \))
  ISO
  execute \( P \) or \( P(...) \)

call\_cleanup(\( :Goal, :Cleanup \))
  executes \( Goal \). When \( Goal \) succeeds determinately, is cut, fails, or raises an exception, \( Cleanup \) is executed.

call\_residue\_vars(\( :Goal, ?Vars \))
  executes \( Goal \). \( Vars \) is unified with the list of new variables created during the call that remain unbound and have blocked goals or attributes attached to them.

\(+\text{Iterators do} :Body\)
  executes \( Body \) iteratively according to \texttt{Iterators}

\texttt{fail}
  fail (start backtracking)

\texttt{false}
  same as \texttt{fail}

\texttt{freeze(+Var, :Goal)}
  Blocks \( Goal \) until \texttt{nonvar(Var)} holds.

\texttt{if(\( :P, :Q, :R \))}
  for each solution of \( P \) that succeeds, prove \( Q \); if none, prove \( R \)

\texttt{once(\( :P \))}
  ISO
  Find the first solution, if any, of goal \( P \).

\texttt{otherwise}
  same as \texttt{true}

\texttt{repeat}
  succeed repeatedly on backtracking

\texttt{true}
  succeed

\texttt{when(+Cond, :Goal)}
  block \( Goal \) until \( Cond \) holds

\subsection{Database}

\texttt{abolish(\( :F \))}
  ISO
  abolish the predicate(s) specified by \( F \)
abolish(:F, +O)
abolish the predicate(s) specified by F with options O

assert(:C)
assert(:C, -R)
clause C is asserted; reference R is returned

asserta(:C)
asserta(:C, -R)
clause C is asserted before existing clauses; reference R is returned

assertz(:C)
assertz(:C, -R)
clause C is asserted after existing clauses; reference R is returned

bb_delete(:Key, -Term)
Delete from the blackboard Term stored under Key.

bb_get(:Key, -Term)
Get from the blackboard Term stored under Key.

bb_put(:Key, +Term)
Store Term under Key on the blackboard.

bb_update(:Key, -OldTerm, +NewTerm)
Replace OldTerm by NewTerm under Key on the blackboard.

clause(:P, ?Q)
clause(:P, ?Q, ?R)
there is a clause for a dynamic predicate with head P, body Q, and reference R

current_key(?N, ?K)
N is the name and K is the key of a recorded term

dynamic :P
predicates specified by P are dynamic declaration,ISO

erase(+R)
erase the clause or record with reference R

instance(+R, -T)
T is an instance of the clause or term referenced by R

recorda(+K, +T, -R)
make term T the first record under key K; reference R is returned

recorded(?K, ?T, ?R)
term T is recorded under key K with reference R

recordz(+K, +T, -R)
make term T the last record under key K; reference R is returned

retract(:C)
ISO
erase the first dynamic clause that matches C

retractall(:H)
ISO
erase every clause whose head matches H
11.2.6 Debugging

add_breakpoint(+Conditions, -BID)

   Creates a breakpoint with Conditions and with identifier BID.

user:breakpoint_expansion(+Macro, -Body)

   defines debugger condition macros

coverage_data(?Data)

   Data is the coverage data accumulated so far

coverage_data(?Data) since release 4.2

current_breakpoint(?Conditions, ?BID, ?Status, ?Kind, ?Type)

   There is a breakpoint with conditions Conditions, identifier BID, enabledness Status, kind Kind, and type Type.

debug

   switch on debugging

disable_breakpoints(+BIDs)

   Disables the breakpoints specified by BIDs.

enable_breakpoints(+BIDs)

   Enables the breakpoints specified by BIDs.

user:error_exception(+Exception)

   Exception is an exception that traps to the debugger if it is switched on.

execution_state(+Tests)

   Tests are satisfied in the current state of the execution.

execution_state(+FocusConditions, +Tests)

   Tests are satisfied in the state of the execution pointed to by FocusConditions.

leash(+M)

   set the debugger’s leashing mode to M

nodebug

   switch off debugging

nospy(:P)

   remove spypoints from the procedure(s) specified by P

nospyall

   remove all spypoints

notrace

   switch off debugging (same as nodebug/0)

nozip

   switch off debugging (same as nodebug/0)
print_coverage since release 4.2, development

print_coverage(?Data) since release 4.2, development

The coverage data Data is displayed in a hierarchical format. Data defaults to the coverage data accumulated so far.

print_profile since release 4.2, development

print_profile(?Data) since release 4.2, development

The profiling data Data is displayed in a format similar to gprof(1). Data defaults to the profiling data accumulated so far.

profile_data(?Data) since release 4.2, development

Data is the profiling data accumulated so far.

profile_reset since release 4.2, development

All profiling data is reset.

remove_breakpoints(+BIDs) development

Removes the breakpoints specified by BIDs.

spy(:P) development

spy(:P,:C)

set spypoints on the procedure(s) specified by P with conditions C

trace development

switch on debugging and start tracing immediately

unknown(-O,+N) development

Changes action on undefined predicates from O to N.

user:unknown_predicate_handler(+G,+M,-N) hook

handle for unknown predicates.

zip development

switch on debugging in zip mode

11.2.7 Errors and Exceptions

abort abort execution of the program; return to current break level

break start a new break level to interpret commands from the user

catch(:P,?E,:H) ISO

specify a handler H for any exception E arising in the execution of the goal P

user:.error_exception(+Exception) hook, development

Exception is an exception that traps to the debugger if it is switched on.

goal_source_info(+AGoal, -Goal, -SourceInfo)

Decomposes the annotated goal AGoal into a Goal proper and the SourceInfo descriptor term, indicating the source position of the goal.

halt ISO

halt(C) ISO

exit from Prolog with exit code C

on_exception(?E,:P,:H)

specify a handler H for any exception E arising in the execution of the goal P
raise_exception(+E)  
    raise exception E  
throw(+E)  
    raise exception E  
unknown(?OldValue, ?NewValue)  
    access the unknown Prolog flag and print a message  
user:unknown_predicate_handler(+Goal, +Module, -NewGoal)  
    tell Prolog to call Module:NewGoal if Module:Goal is undefined  

11.2.8 Filename Manipulation  

absolute_file_name(+R, -A)  
    hookable  
absolute_file_name(+R, -A, +O)  
    hookable  
    expand relative filename R to absolute file name A using options specified in O  
user:file_search_path(+F, -D)  
    hook  
    directory D is included in file search path F  
user:library_directory(-D)  
    hook  
    D is a library directory that will be searched  

11.2.9 File and Stream Handling  

byte_count(+S, -N)  
    N is the number of bytes read/written on binary stream S  
character_count(+S, -N)  
    N is the number of characters read/written on text stream S  
close(+F)  
    ISO  
close(+F, +O)  
    ISO  
    close file or stream F with options O  
current_input(-S)  
    ISO  
    S is the current input stream  
current_output(-S)  
    ISO  
    S is the current output stream  
    ISO  
    S is a stream open on file F in mode M  
line_count(+S, -N)  
    N is the number of lines read/written on text stream S  
line_position(+S, -N)  
    N is the number of characters read/written on the current line of text stream S  
open(+F, +M, -S)  
    ISO  
open(+F, +M, -S, +O)  
    ISO  
    file F is opened in mode M, options O, returning stream S
open_null_stream(+S)
    new output to text stream S goes nowhere
prompt(-O,+N)
    queries or changes the prompt string of the current input stream
see(+F)
    make file F the current input stream
seeing(-N)
    the current input stream is named N
seek(+S,+O,+M,+N)
    seek to an arbitrary byte position on the stream S
seen
    close the current input stream
set_input(+S)
    select S as the current input stream
set_output(+S)
    select S as the current output stream
set_stream_position(+S,+P)
    P is the new position of stream S
stream_code(?S,?C)
    Converts between Prolog and C representations of a stream
stream_position(+S,-P)
    P is the current position of stream S
stream_position_data(?Field,?Position,?Data)
    The Field field of the stream position term Position is Data.
stream_property(?Stream, ?Property)
    Stream Stream has property Property.
tell(+F)
    make file F the current output stream
telling(-N)
    to file N
told
    close the current output stream

11.2.10 Foreign Interface

foreign(+F,-P)
    hook
foreign(+F,-L,-P)
    hook
    function F in language L is attached to P
foreign_resource(+R,-L)
    hook
    resource R defines foreign functions in list L
load_foreign_resource(+R)
    hookable
    load foreign resource R
stream_code(?S,?C)
    Converts between Prolog and C representations of a stream
unload_foreign_resource(+R)

unload foreign resource \( R \)

11.2.11 Grammar Rules

:Head --> :Body
A possible form for Head is Body

expand_term(+T,-X)

\( T \) expands to \( X \) using user:term_expansion/6 or grammar rule expansion

phrase(:P,-L)

\( P \) or the empty list is what remains of list \( L \) after phrase \( P \) has been found

user:term_expansion(+Term1,+Layout1,+Tokens1,-Term2,-Layout2,-Tokens2)

hook

Overrides or complements the standard transformations to be done by expand_term/2.

11.2.12 Hook Predicates

user:breakpoint_expansion(+Macro,-Body)

hook,development

defines debugger condition macros

user:debugger_command_hook(+DCommand,-Actions)

hook,development

Allows the interactive debugger to be extended with user-defined commands.

user:error_exception(=Exception)

hook

Exception is an exception that traps to the debugger if it is switched on.

user:file_search_path(+F,-D)

hook
directory \( D \) is included in file search path \( F \)

foreign(+F,-P)
foreign(+F,-L,-P)

Describes the interface between Prolog and the foreign Routine

foreign_resource(+R,-L)

resource \( R \) defines foreign functions in list \( L \)

user:generate_message_hook(+M,?S0,?S)

hook

A way for the user to override the call to 'SU_messages':generate_message/3 in print_message/2.

goal_expansion(+Term1,+Layout1,+Module,-Term2,-Layout2)

hook

Defines transformations on goals while clauses are being compiled or asserted, and during meta-calls.

user:library_directory(-D)

hook

\( D \) is a library directory that will be searched

user:message_hook(+S,+M,+L)

hook

Overrides the call to print_message_lines/3 in print_message/2.
user:portray(+T)
A way for the user to override the default behavior of print/1.

user:portray_message(+S,+M)

hook
Tells print_message/2 what to do.

user:query_hook(+QueryClass, +Query, +QueryLines, +Help, +HelpLines, -Answer)

hook
Called by ask_query/4 before processing the query. If this predicate succeeds, then it is assumed that the query has been processed and nothing further is done.

user:query_class_hook(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)

hook
Provides the user with a method of overriding the call to 'SU_messages':query_class/5 in the preparation phase of query processing. This way the default query class characteristics can be changed.

user:query_input_hook(+InputMethod, +Prompt, -RawInput)

hook
Provides the user with a method of overriding the call to 'SU_messages':query_input/3 in the input phase of query processing. This way the implementation of the default input methods can be changed.

user:query_map_hook(+MapMethod, +RawInput, -Result, -Answer)

hook
Provides the user with a method of overriding the call to 'SU_messages':query_map/4 in the mapping phase of query processing. This way the implementation of the default map methods can be changed.

user:runtime_entry(+M)

hook
This predicate is called upon start-up and exit of stand alone applications.

user:term_expansion(+Term1, +Layout1, +Tokens1, -Term2, -Layout2, -Tokens2)

hook
Overrides or complements the standard transformations to be done by expand_term/2.

user:unknown_predicate_handler(+G,+M,-N)

hook
hook to trap calls to unknown predicates

11.2.13 List Processing

?T =.. ?L
the functor and arguments of term T comprise the list L

append(?A,?B,?C)
the list C is the concatenation of lists A and B

keysort(+L,-S)
the list L sorted by key yields S

length(?L,?N)
the length of list L is N

member(?X,?L)
X is a member of L
memberchk(+X,+L)
  X is a member of L

nonmember(+X,+L)
  X is not a member of L

sort(+L,-S)  \( ISO \)
  sorting the list L into order yields S

11.2.14 Loading Programs

[]
[:F|+Fs] same as \texttt{load_files([F|Fs])}

block :\( P \)  \( declaration \)
  predicates specified by \( P \) should block until sufficiently instantiated

\texttt{compile(:F)}
  load compiled clauses from files \( F \)

\texttt{consult(:F)}
\texttt{reconsult(:F)}
  load interpreted clauses from files \( F \)

\texttt{expand_term(+T,-X)}  \( hookable \)
  term \( T \) expands to term \( X \) using \texttt{user:term_expansion/6} or grammar rule expansion

\texttt{goal_expansion(+Term1, +Layout1, +Module, -Term2, -Layout2)}  \( hook \)
  Defines transformations on goals while clauses are being compiled or asserted, and during meta-calls.

discontiguous :\( P \)  \( declaration, ISO \)
  clauses of predicates \( P \) do not have to appear contiguously

dynamic :\( P \)  \( declaration, ISO \)
  predicates specified by \( P \) are dynamic

elif(:Goal)  \( declaration \)
  Provides an alternative branch in a sequence of conditional compilation directives.

dynamic :\( P \)  \( declaration \)
  Provides an alternative branch in a sequence of conditional compilation directives.

endif  \( declaration \)
  Terminates a sequence of conditional compilation directives.

\texttt{ensure_loaded(:F)}  \( ISO \)
  load \( F \) if not already loaded

\texttt{if(:Goal)}  \( declaration \)
  Starts a sequence of conditional compilation directives for conditionally including parts of a source file.
include(+F)  declaration, ISO
   include the source file(s) F verbatim

initialization :G  declaration, ISO
   declares G to be run when program is started

load_files(:F)  declaration
load_files(:F,+O)  declaration
   load files according to options O

meta_predicate :P  declaration
   declares predicates P that are dependent on the module from which they are called

mode :P  declaration
   NO-OP: document calling modes for predicates specified by P

module(+M,+L)  declaration
module(+M,+L,+O)  declaration
   module M exports predicates in L, options O

multifile :P  declaration, ISO
   the clauses for P are in more than one file

public :P  declaration
   NO-OP: declare predicates specified by P public

restore(+F)  declaration
   restore the state saved in file F

user:term_expansion(+Term1,+Layout1,+Tokens1,-Term2,-Layout2,-Tokens2)  hook
   Overrides or complements the standard transformations to be done by expand_term/2.

use_module(:F)  declaration
use_module(:F,+I)  declaration
   import the procedure(s) I from the module file F

use_module(?M,:F,+I)  declaration
   import I from module M, loading module file F if necessary

volatile :P  declaration
   predicates specified by P are not to be included in saves

11.2.15 Memory

garbage_collect  declaration
   force an immediate garbage collection

garbage_collect_atoms  declaration
   garbage collect atom space

statistics  declaration
   display various execution statistics
statistics(?K,?V)
the execution statistic with key \( K \) has value \( V \)

**trimcore** reduce free stack space to a minimum

### 11.2.16 Messages and Queries

**ask_query(+QueryClass, +Query, +Help, -Answer)**  
hookable
Prints the question \( Query \), then reads and processes user input according to \( QueryClass \), and returns the result of the processing, the abstract answer term \( Answer \). The \( Help \) message is printed in case of invalid input.

**user:message_hook(+M,+S,+L)**  
hook
intercept the printing of a message

'SU_messages':**generate_message(+M,+SO,+S)**  
hook
determines the mapping from a message term into a sequence of lines of text to be printed

**user:generate_message_hook(+M,+SO,+S)**  
hook
intercept message before it is given to 'SU_messages':generate_message/3

**goal_source_info(+AGoal, -Goal, -SourceInfo)**
Decomposes the annotated goal \( AGoal \) into a \( Goal \) proper and the \( SourceInfo \) descriptor term, indicating the source position of the goal.

**user:portray_message(+Severity,+Message)**  
hook
Tells \( print_message/2 \) what to do.

**print_message(+S,+M)**  
hookable
print a message \( M \) of severity \( S \)

**print_message_lines(+S,+P,+L)**
print the message lines \( L \) to stream \( S \) with prefix \( P \)

'SU_messages':**query_abbreviation(+T,-P)**  
hook
specifies one letter abbreviations for responses to queries from the Prolog system

**user:query_hook(+QueryClass, +Query, +QueryLines, +Help, +HelpLines, -Answer)**
hook
Called by \( ask_query/4 \) before processing the query. If this predicate succeeds, then it is assumed that the query has been processed and nothing further is done.

'SU_messages':**query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)**  
hook
Access the parameters of a given \( QueryClass \).

**user:query_class_hook(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)**
hook
Provides the user with a method of overriding the call to 'SU_messages':query_class/5 in the preparation phase of query processing. This way the default query class characteristics can be changed.
'SU_messages':query_input(+InputMethod, +Prompt, -RawInput)        hook
Implements the input phase of query processing.

user:query_input_hook(+InputMethod, +Prompt, -RawInput)        hook
Provides the user with a method of overriding the call to 'SU_messages':query_input/3 in the input phase of query processing. This way the implementation of the default input methods can be changed.

'SU_messages':query_map(+MapMethod, +RawInput, -Result, -Answer)        hook
Implements the mapping phase of query processing.

user:query_map_hook(+MapMethod, +RawInput, -Result, -Answer)        hook
Provides the user with a method of overriding the call to 'SU_messages':query_map/4 in the mapping phase of query processing. This way the implementation of the default map methods can be changed.

### 11.2.17 Modules

current_module(?M)           declaration
M is the name of a current module

current_module(?M, ?F)       declaration
F is the name of the file in which M’s module declaration appears

meta_predicate :P              declaration
declares predicates P that are dependent on the module from which they are called

module(+M,+L)                  declaration
declaration that module M exports predicates in L, options O

module(+M,+L,+O)               declaration

save_modules(+L,+F)     save the modules specified in L into file F

set_module(+M)               declaration
make M the type-in module

use_module(:F)               declaration
import the module file(s) F, loading them if necessary

use_module(:F,+I)            declaration
import the procedure(s) I from the module file F

use_module(?M,:F,+I)        declaration
import I from module M, loading module file F if necessary

### 11.2.18 Program State

current_atom(?A)        declaration
backtrack through all atoms

current_module(?M)          declaration
M is the name of a current module
current_module(?M, ?F)
   F is the name of the file in which M’s module declaration appears

current_predicate(:A/?N) ISO
current_predicate(?A, :P)
   A is the name of a predicate with most general goal P and arity N

current_prolog_flag(?F, ?V) ISO
   V is the current value of Prolog flag F

listing  list all dynamic procedures in the type-in module
listing(:P)  list the dynamic procedure(s) specified by P

predicate_property(:P, ?Prop)
   Prop is a property of the loaded predicate P

prolog_flag(?F, ?V)
   V is the current value of Prolog flag F

prolog_flag(+F, =O, +N)
   O is the old value of Prolog flag F; N is the new value

prolog_load_context(?K, ?V)
   find out the context of the current load

set_module(+M)
   make M the type-in module

set_prolog_flag(+F, +N) ISO
   N is the new value of Prolog flag F

source_file(?F)
   F is a source file that has been loaded into the database

source_file(:P, ?F)
   P is a predicate defined in the loaded file F

unknown(-O, +N) development
   Changes action on undefined predicates from O to N.

11.2.19 Saving Programs

initialization :G declaration, ISO
   declares G to be run when program is started

load_files(:F)
load_files(:F, +O)
   load files according to options O

user:runtime_entry(+S) hook
   entry point for a runtime system

save_files(+L, +F)
   saves the modules, predicates, clauses and directives in the given files L into
   file F
save_modules(+L,+F)
    save the modules specified in L into file F

save_predicates(:L,+F)
    save the predicates specified in L into file F

save_program(+F)
save_program(+F,:G)
    save all Prolog data into file F with startup goal G

volatile :P 
    declares predicates specified by P to not be included in saves.

11.2.20 Term Comparison

compare(-C,+X,+Y) 
    C is the result of comparing terms X and Y

+X == +Y 
    terms X and Y are strictly identical

+X \== +Y 
    terms X and Y are not strictly identical

+X @< +Y 
    term X precedes term Y in standard order for terms

+X @>= +Y 
    term X follows or is identical to term Y in standard order for terms

+X @> +Y 
    term X follows term Y in standard order for terms

+X @=\< +Y 
    term X precedes or is identical to term Y in standard order for terms

11.2.21 Term Handling

?T =.. ?L 
    the functor and arguments of term T comprise the list L.

?X = ?Y 
    terms X and Y are unified.

+X \=< +Y 
    terms X and Y no not unify.

?=+(X,+Y) 
    X and Y are either strictly identical or do not unify.

acyclic_term(+T) 
    term T is a finite (acyclic) term.

arg(+N,+T,\-A) 
    the Nth argument of term T is A.
atom_chars(?A, ?L)  
A is the atom containing the character atoms in list L.

atom_codes(?A, ?L)  
A is the atom containing the characters in code list L.

atom_concat(?Atom1, ?Atom2, ?Atom12)  
Atom Atom1 concatenated with Atom2 gives Atom12.

atom_length(+Atom, -Length)  
Length is the number of characters of the atom Atom.

char_code(?Char, ?Code)  
Code is the character code of the one-char atom Char.

copy_term(+T, -C)  
C is a copy of T in which all variables have been replaced by new variables.

copy_term(+T, -C, -G)  
C is a copy of T in which all variables have been replaced by new variables, and G is a goal for reinstating any attributes in C.

create_mutable(+Datum, -Mutable)  
Mutable is a new mutable term with current value Datum.

dif(+X, +Y)  
X and Y are constrained to be different.

frozen(+Term, -Goal)  
Goal is the conjunction of all goals blocked on some variable in Term.

functor(?T, ?F, ?N)  
the principal functor of term T has name F and arity N.

get_mutable(-Datum, +Mutable)  
The current value of the mutable term Mutable is Datum.

name(?A, ?L)  
the code list of atom or number A is L.

number_chars(?N, ?L)  
N is the numeric representation of list of character atoms L.

number_codes(?N, ?L)  
N is the numeric representation of code list L.

numbervars(+T, +M, -N)  
number the variables in term T from M to N-1.

The characters of SubAtom form a sublist of the characters of Atom, such that the number of characters preceding SubAtom is Before, the number of characters after SubAtom is After, and the length of SubAtom is Length.

subsumes_term(General, Specific)  
Specific is an instance of General.  
since release 4.3, ISO
term_variables(+Term,-Variables) \textit{since release 4.3, ISO}

Variables is the set of variables that occur in Term, in first occurrence order.

unify_with_occurs_check(?X, ?Y) \textit{ISO}

True if X and Y unify to a finite (acyclic) term.

\subsection*{11.2.22 Term I/O}

char_conversion(+InChar, +OutChar) \textit{ISO}

The mapping of InChar to OutChar is added to the character-conversion mapping.

current_char_conversion(?InChar, ?OutChar) \textit{ISO}

InChar is mapped to OutChar in the current character-conversion mapping.

current_op(?P, ?T, ?A) \textit{ISO}

atom A is an operator of type T with precedence P

display(+T)

write term T to the user output stream in functional notation

format(+C,:A)

format(+S,+C,:A)

write arguments A on stream S according to control string C

op(+P,+T,+A) \textit{ISO}

make atom A an operator of type T with precedence P

user:portray(+T) \textit{hook}

tell print/[1,2] and write_term/[2,3] what to do

portray_clause(+C)

portray_clause(+S,+C)

write clause C to the stream S

print(+T) \textit{hookable}

print(+S,+T) \textit{hookable}

display the term T on stream S using user:portray/1 or write/2

read(-T) \textit{ISO}

read(+S,-T) \textit{ISO}

read term T from stream S

read_term(-T,+O) \textit{ISO}

read_term(+S,-T,+O) \textit{ISO}

read T from stream S according to options O

write(+T) \textit{ISO}

write(+S,+T) \textit{ISO}

write term T on stream S

write_canonical(+T) \textit{ISO}

write_canonical(+S,+T) \textit{ISO}

write term T on stream S so that it can be read back by read/[1,2]
writeq(+T)
writeq(+S,+T)
    write term T on stream S, quoting atoms where necessary
write_term(+T,+O)
write_term(+S,+T,+O)
    writes T to S according to options O

11.2.23 Type Tests

atom(+T)
    term T is an atom
atomic(+T)
    term T is an atom or a number
callable(+T)
    T is an atom or a compound term
compound(+T)
    T is a compound term
db_reference(+X)
    X is a db_reference
    since release 4.1
float(+N)
    N is a floating-point number
ground(+T)
    term T is a nonvar, and all substructures are nonvar
integer(+T)
    term T is an integer
mutable(+X)
    X is a mutable term
nonvar(+T)
    term T is one of atom, number, compound (that is, T is instantiated)
number(+N)
    N is an integer or a float
simple(+T)
    T is not a compound term; it is either atomic or a var
var(+T)
    term T is a variable (that is, T is uninstantiated)

11.3 Built-In Predicates

The following reference pages, alphabetically arranged, describe the SICStus Prolog built-in predicates.

For a functional grouping of these predicates including brief descriptions, see Section 11.2 [mpg-top], page 903.
In many cases, the heading of a reference page, as well as an entry in a list of built-in predicates, will be annotated with keywords. These annotations are defined in Section 11.1.3 [mpg-ref-cat], page 900.

Further information about categories of predicates and arguments, mode annotations, and the conventions observed in the reference pages is found in Section 11.1 [mpg-ref], page 899.
11.3.1 abolish/[1,2]

Synopsis
abolish(+Predicates)
abolish(+Predicates, +Options)

Removes procedures from the Prolog database.

Arguments
:Predicates
pred_spec or pred_spec_tree
A predicate specification, or a list of such.
Note that the default is to only allow a single predicate specification, see tree/1 option below.

Options list of term, must be ground
A list of zero or more of the following:

force(Boolean)
Specifies whether SICStus Prolog is to abolish the predicate even if it is static (true), or only if it is dynamic (false). The latter is the default.

tree(Boolean)
Specifies whether the first argument should be a pred_spec_tree (true), or a pred_spec (false). The latter is the default.

Description
Removes all procedures specified. After this command is executed the current program functions as if the named procedures had never existed. That is, in addition to removing all the clauses for each specified procedure, abolish/[1,2] removes any properties that the procedure might have had, such as being dynamic or multifile. You cannot abolish built-in procedures.

It is important to note that retract/1, retractall/1, and erase/1 only remove clauses, and only of dynamic procedures. They don’t remove the procedures themselves or their properties properties (such as being dynamic or multifile). abolish/[1,2], on the other hand, remove entire procedures along with any clauses and properties.

The procedures that are abolished do not become invisible to a currently running procedure.

Space occupied by abolished procedures is reclaimed. The space occupied by the procedures is reclaimed.

Procedures must be defined in the source module before they can be abolished. An attempt to abolish a procedure that is imported into the source module will cause a permission error. Using a module prefix, ‘M:’, procedures in any module may be abolished.
Abolishing a foreign procedure destroys only the link between that Prolog procedure and the associated foreign code. The foreign code that was loaded remains in memory. This is necessary because Prolog cannot tell which subsequently-loaded foreign files may have links to the foreign code. The Prolog part of the foreign procedure is destroyed and reclaimed.

Specifying an undefined procedure is not an error.

**Exceptions**

- **instantiation_error**
  - if one of the arguments is not instantiated enough.

- **type_error**
  - *Predicates* is not a valid tree of predicate specifications, or a *Name* is not an atom or an *Arity* not an integer.

- **domain_error**
  - if an *Arity* is specified as an integer outside the range 0-255.

- **permission_error**
  - if a specified procedure is built-in, or imported into the source module, or static when *force(true)* is not in effect.

**Examples**

```
| ?- [user].
% compiling user...
| foo(1,2).
| `D
% compiled user in module user, 10 msec -80 bytes
yes
| ?- abolish(foo).
! Type error in argument 1 of abolish/1
! expected pred_spec, but found foo
! goal: abolish(user:foo)
| ?- abolish(foo,[tree(true)]).
! Permission error: cannot abolish static user:foo/2
! goal: abolish(user:foo,[tree(true)])
| ?- abolish(foo/2).
! Permission error: cannot abolish static user:foo/2
! goal: abolish(user:foo/2)
| ?- abolish(foo/2,[force(true)]).
yes
% source_info
```

**Comments**

abolish/1 is part of the ISO Prolog standard; abolish/2 is not.

**See Also**

dynamic/1, erase/1, retract/1, retractall/1.
11.3.2 abort/0

Synopsis
abort

Abandons the current execution and returns to the beginning of the current break level or terminates the enclosing query, whichever is closest.

Description
Fairly drastic predicate that is normally only used when some error condition has occurred and there is no way of carrying on, or when debugging.

Often used via the debugging option a or the ^C interrupt option a.

abort/0 is implemented by raising a reserved exception, which has handler at the top level; see Section 4.15.7 [ref-ere-int], page 211.

Tips
Does not close any files that you may have opened. When using see/1 and tell/1, (rather than open/3, set_input/1, and set_output/1), close files yourself to avoid strange behavior after your program is aborted and restarted.

Exceptions
Does not throw errors, but is implemented by throwing a reserved exception.

See Also
halt/[0,1], break/0, runtime_entry/1, Section 4.15.7 [ref-ere-int], page 211.
### 11.3.3 absolute_file_name/[2,3]

**hookable**

**Synopsis**

```
absolute_file_name(+RelFileSpec, -AbsFileName)

absolute_file_name(+RelFileSpec, -AbsFileName, +Options)
```

Unifies `AbsFileName` with the absolute filename that corresponds to the relative file specification `RelFileSpec`.

**Arguments**

*RelFileSpec*

- `file_spec`, must be ground
  - A valid file specification. See below for details.

*AbsFileName*

- `atom`
  - Corresponding absolute filename.

*Options*  

- `list of term`, must be ground
  - A list of zero or more of the following. The default is the empty list:

  **extensions(Ext)**
  
  Has no effect if `FileSpec` contains a file extension. `Ext` is an atom or a list of atoms, each atom representing an extension (e.g. `'.pl'`) that should be tried when constructing the absolute file name. The extensions are tried in the order they appear in the list. Default value is `Ext = ['']`, i.e. only the given `FileSpec` is tried, no extension is added. To specify `extensions(''` or `extensions([])` is equal to not giving any extensions option at all.

  When case-normalization is applied to the `FileSpec` e.g. on Windows, each atom in `Ext` will also be case-normalized before use. That is, on Windows, specifying `extensions(['.pl'])` will typically give the same result as `extensions(['.PL'])`. Prior to release 4.3 the `extensions/1` option was always case sensitive, also on Windows.

  **file_type(Type)**
  
  Picks an adequate extension for the operating system currently running, which means that programs using this option instead of `extensions(Ext)` will be more portable between operating systems. This extension mechanism has no effect if `FileSpec` contains a file extension. `Type` must be one of the following atoms:

  **text**
  
  - Imply `extensions([])`. `FileSpec` is a file without any extension. (Default)

  **source**
  
  - `implies extensions(['.pro','.pl','']).` `FileSpec` is a Prolog source file, maybe with a ``.pro` or ``.pl` extension.

```
object implies extensions(['.po']). FileSpec is a Prolog object file.

saved_state implies extensions(['.sav', '']) . FileSpec is a saved state, maybe with a '.sav' extension.

foreign_resource FileSpec is a foreign language shared object file, maybe with a system dependent extension.

executable since release 4.0.2 FileSpec is an executable file, maybe with a system dependent extension.

directory implies extensions([]). This option has two effects. First, for an access option other than access(none) the file must exist and be a directory. Second, the returned file name will end in slash (/)

Only when this option is present can absolute_file_name/3 return the name of an existing directory with an access option other than access(none) without raising an exception.

glob(Glob) Match file names against a pattern. RelFileSpec will be expanded to a directory and AbsFileName will be the absolute path to each child that matches both the Glob pattern and any other filtering option, like access/1, extensions/1, file_type/1, .... The special children . and .. will never be returned. The Glob should be an atom specifying a glob pattern consisting of characters interpreted as follows:

- A '*' matches any sequence of zero or more characters.
- A '?' matches exactly one character.
- A '{', '}', '[', ']' currently matches themselves but are reserved for future expansion of the allowable patterns.
- Any other character matches itself.

With the options solutions(all) and file_errors(fail) this can be used to enumerate the contents of a directory.

access(Mode) Mode must be an atom or a list of atoms. If a list is given, then AbsFileName must obey every specified option in the list. This makes it possible to combine a read and write, or write and exist check, into one call. If AbsFileName specifies a directory and an access option other than access(none) is specified, then a permission error is signaled unless file_type(directory) is also specified.
Each atom must be one of the following:

- **read**  
  `AbsFileName` must be readable and exist.

- **write**  
  If `AbsFileName` exists, then it must be writable. If it doesn’t exist, then it must be possible to create.

- **append**  
  If `AbsFileName` exists, then it must be writable. If it doesn’t exist, then it must be possible to create.

- **exist**  
  The file represented by `AbsFileName` must exist.

- **execute**  
  The file represented by `AbsFileName` must be executable and exist. This is ignored if `file_type(directory)` is also specified.

- **search**  
  The directory represented by `AbsFileName` must be searchable and exist. This is ignored unless `file_type(directory)` is also specified.

- **none**  
  The file system is not accessed to determine existence or access properties of `AbsFileName`. The first absolute file name that is derived from `FileSpec` is returned. Note that if this option is specified, then no existence exceptions can be raised. (Default)

**Please note:** Most current file systems have complex access control mechanisms, such as access control lists (ACLs). These mechanisms make it hard to determine the effective access permissions, short of actually attempting the file operations in question. With networked file systems it may in fact be impossible to determine the effective access rights.

Therefore, a simplified access control model is used by `absolute_file_name/3` and elsewhere in SICStus.

On UNIX systems only the “classical” access control information is used, i.e. the read/write/execute “bits” for owner/group/other. Under Windows only the “FAT” access control information is used, i.e. a file may be marked as read-only. A file is deemed executable if its extension is one of `.cmd`, `.bat` or if it is classified as an executable by the Win32 API `GetBinaryType`.

This may change to more faithfully reflect the effective permissions in a future release.

```prolog
file_errors(Val)
fileerrors(Val)
```

`Val` is one of the following, where the default is determined by the current value of the `fileerrors` Prolog flag:

- **error**  
  Raise an exception if a file derived from `FileSpec` has the wrong permissions, that is, can’t be accessed at all,
or doesn’t satisfy the access modes specified with the access option. This is the default if the Prolog flag fileerrors is set to its default value, on.

**fail**

Fail if a file derived from FileSpec has the wrong permissions. Normally an exception is raised, which might not always be a desirable behavior, since files that do obey the access options might be found later on in the search. When this option is given, the search space is guaranteed to be exhausted. This is the default if the Prolog flag fileerrors is set to off.

**solutions(Val)**

Val is one of the following:

**first**

As soon as a file derived from FileSpec is found, commit to that file. Makes absolute_file_name/3 determinate. (Default)

**all**

Return each file derived from FileSpec that is found. The files are returned through backtracking. This option is probably most useful in combination with the option file_errors(fail).

**relative_to(FileOrDirectory)**

FileOrDirectory should be an atom, and controls how to resolve relative filenames. If it is '', then file names will be treated as relative to the current working directory. If a regular, existing file is given, then file names will be treated as relative to the directory containing FileOrDirectory. Otherwise, file names will be treated as relative to FileOrDirectory.

If absolute_file_name/3 is called from a goal in a file being loaded, then the default is the directory containing that file, accessible from the load context (prolog_load_context/2). Otherwise, the default is the current working directory.

You can use file_systems:current_directory/1 to obtain the current working directory from a goal in a file being loaded.

**if_user(Val)** since release 4.3

controls how to resolve the special file name user. Val is one of the following:

**file**

Treat the name user like any other name, e.g. like open/3 does. This is the default.

**user**

Unifies AbsFileName with the atom user and ignores the other options. This corresponds to the behavior prior to SICStus Prolog 4.3.

**error**

Treat the name user as a non-existing file, subject to the file_errors/1 option.
Description

If FileSpec is user, and the option if_user(file) is not in effect, then special processing takes place, see the description of the if_user/1 option, above. Otherwise (the default), unifies AbsFileName with the first absolute file name that corresponds to the relative file specification FileSpec and that satisfies the access modes given by Options.

The functionality of absolute_file_name/3 is most easily described as multi-phase process, in which each phase gets an infile from the preceding phase, and constructs one or more outfiles to be consumed by the succeeding phases. The phases are:

1. Syntactic rewriting
2. Pattern expansion
3. Extension expansion
4. Access checking

The first phase and each of the expansion phases modifies the infile and produces variants that will be fed into the succeeding phases. The functionality of all phases but the first are decided with the option list. The last phase checks if the generated file exists, and if not asks for a new variant from the preceding phases. If the file exists, but doesn’t obey the access mode option, then a permission exception is raised. If the file obeys the access mode option, then absolute_file_name/3 commits to that solution, subject to the solutions option, and unifies AbsFileName with the file name. For a thorough description, see below.

Note that the relative file specification FileSpec may also be of the form Path(FileSpec), in which case the absolute file name of the file FileSpec in one of the directories designated by Path is returned (see the description of each phase below).

Syntactic rewriting
This phase translates the relative file specification given by FileSpec into the corresponding absolute file name. The rewrite is done wrt. the value of the relative_to option. There can be more than one solution, in which case the outfile becomes the solutions in the order they are generated. If the following phases fails, and there are no more solutions, then an existence exception is raised.

FileSpec can be a file search path, e.g. library('lists.pl'). It can also refer to system properties, environment variables and the home directory of users. See Section 4.5.2 [ref-fdi-syn], page 99, for a description of syntactic rewriting.

Pattern expansion
If the glob/1 option was specified all matching children of the directory will be enumerated. See the glob option.

Extension expansion
See the extensions and file_type options.

Access checking
See the access option.
Final stage
As a final stage, if file_type(directory) is specified, then the file is suffixed with slash. Otherwise, trailing slash will be removed except for root directories, such as ‘/’ under UNIX or ‘c:/’ under Windows.

Backtracking
Can find multiple solutions only if the solutions(all) option is used.

Exceptions
instantiation_error
Any of the Options arguments or RelFileSpec is not ground.
type_error
In Options or in RelFileSpec.
domain_error
Options contains an undefined option.
existence_error
RelFileSpec is syntactically valid but does not correspond to any file and an access option other than access(none) was given.
permission_error
RelFileSpec names an existing file but the file does not obey the given access mode.

Comments
If an option is specified more than once, then the rightmost option takes precedence. This provides for a convenient way of adding default values by putting these defaults at the front of the list of options. If absolute_file_name/3 succeeds, and the file access option was one of {read, write, append}, then it is guaranteed\(^1\) that the file can be opened with open/[3,4]. If the access option was exist, then the file does exist, but might be both read and write protected.

If file_type(directory) is not given, then the file access option is other than none, and a specified file refers to a directory, then absolute_file_name/3 signals a permission error.

absolute_file_name/[2,3] is sensitive to the fileerrors Prolog flag, which determines whether the predicate should fail or raise permission errors when encountering files with the wrong permission. Failing has the effect that the search space always is exhausted.

If RelFileSpec contains ‘.’ components, then these are resolved by removing directory components from the pathname, not by accessing the file system. This can give unexpected results, e.g. when soft links or mount points are involved.

This predicate is used for resolving file specification by built-in predicates that open files.

\(^1\) To the extent that the access permissions can be precisely determined. See the access/1 option above.
Examples

To check whether the file \texttt{my\_text} exists in the home directory, with one of the extensions `\texttt{.text}` or `\texttt{.txt}`, and is both writable and readable:

\begin{verbatim}
| ?- absolute_file_name('~/my\_text', File,  
  [extensions(['.text', '.txt']),  
  access([read,write])]).
\end{verbatim}

To check whether the directory \texttt{bin} exists in the home directory:

\begin{verbatim}
| ?- absolute_file_name('~/bin', Dir,  
  [file\_type(directory),  
  access(exist)]).
\end{verbatim}

Here \texttt{Dir} would get a slash terminated value, such as \texttt{/home/joe/}.

To list all files in the current directory:

\begin{verbatim}
| ?- findall(File, absolute_file_name('.', File,  
  [glob('*'),  
  solutions(all), file\_errors(fail)]), Files).
\end{verbatim}

To list all directories in the parent of the current directory containing the string “sicstus”:

\begin{verbatim}
| ?- findall(File, absolute_file_name('..', File,  
  [glob('*sicstus*'), file\_type(directory),  
  solutions(all), file\_errors(fail)]), Files).
\end{verbatim}

To find a file \texttt{cmd.exe} in any of the “usual places” where executables are found, i.e. by looking through the \texttt{PATH} environment variable:

\begin{verbatim}
| ?- absolute_file_name(path('cmd\_exe'), File,  
  [access(exist)])).
\end{verbatim}

This uses the predefined file search path \texttt{path/1}, Section 4.5 [ref-fdi], page 95.

See Also

\texttt{file\_search\_path/2}, \texttt{prolog\_load\_context/2}, Section 4.5 [ref-fdi], page 95, Section 4.9.4 [ref-lps-flg], page 136.
11.3.4 acyclic_term/1

Synopsis

acyclic_term(+Term)

Term is currently instantiated to a finite (acyclic) term.

Arguments

Term term

Description

True if X is finite (acyclic). Runs in linear time.

Examples

| ?- X = g(Y), acyclic_term(f(X,X)).
  X = g(Y) ?
  yes
| ?- X = g(X), acyclic_term(X).
  no
| ?- X = g(X), acyclic_term(f(X)).
  no

Exceptions

None.

See Also

Section 4.8.2 [ref-lte-act], page 127.
11.3.5  add_breakpoint/2

Synopsis

add_breakpoint(+Conditions, -BID)

Creates a breakpoint with Conditions and with identifier BID.

Arguments

:Conditions  
  term.  
  Breakpoint conditions.

BID  integer  
  Breakpoint identifier.

Exceptions

instantiation_error  
  Conditions not instantiated enough.

type_error  
  Conditions not a proper list of callable term.

domain_error  
  Conditions not a proper list of valid breakpoint conditions

context_error  
  Attempt to put a breakpoint on true/0 or fail/0.

See Also

Section 5.6.1 [Creating Breakpoints], page 243, Section 5.7 [Breakpoint Predicates], page 272.
11.3.6  \(/2\)                ISO

Synopsis
+\(P\) , +\(Q\)

Arguments
:\(P\)    \textit{callable}, must be nonvar
:\(Q\)    \textit{callable}, must be nonvar

Description
This is not normally regarded as a built-in predicate, since it is part of the syntax of the
language. However, it is like a built-in predicate in that you can say \texttt{call(\((P\ ,\ Q)\))} to
execute \(P\) and then \(Q\), with identical effect if it does not contain any cuts.

The scope of cuts in \(P\) and \(Q\) extends to the containing clause.

Backtracking
Depends on \(P\) and \(Q\).

Exceptions
Call errors (see Section 4.2.6 [ref-sem-exec], page 77).

See Also
Section 4.2 [ref-sem], page 61.
11.3.7 append/3

Synopsis
append(?List1, ?List2, ?List3)

Arguments
List1 list of term  
List2 list of term  
List3 list of term

A list consisting of List1 followed by List2.

Description
Appends lists List1 and List2 to form List3:

| ?- append([a,b], [a,d], X). 
X = [a,b,a,d]  
| ?- append([a], [a], [a]). 
no  
| ?- append(2, [a], X). 
no

Takes List3 apart:

| ?- append(X, [e], [b,e,e]). 
X = [b,e]  
| ?- append([b|X], [e,r], [b,o,r,e,r]). 
X = [o,r]  
| ?- append(X, Y, [h,i]). 
X = [], Y = [h,i]  
X = [h], Y = [i]  
X = [h,i], Y = []  
no
Backtracking

Suppose \( L \) is bound to a proper list. That is, it has the form \([T_1, \ldots, T_n]\) for some \( n \). In that instance, the following things apply:

1. \( \text{append}(L, X, Y) \) has at most one solution, whatever \( X \) and \( Y \) are, and cannot backtrack at all.
2. \( \text{append}(X, Y, L) \) has at most \( n+1 \) solutions, whatever \( X \) and \( Y \) are, and though it can backtrack over these it cannot run away without finding a solution.
3. \( \text{append}(X, L, Y) \), however, can backtrack indefinitely if \( X \) and \( Y \) are variables.

Examples

The following examples are perfectly ordinary uses of \( \text{append}/3 \):

To enumerate adjacent pairs of elements from a list:

\[
\text{next_to}(X, Y, (#in#) \text{List3}) :-
\text{append}(\_, [X,Y|\_], \text{List3}).
\]

To check whether Word1 and Word2 are the same except for a single transposition. (\( \text{append}/5 \) in \text{library(lists)} would be better for this task.)

\[
\text{one_transposition}(\text{Word1}, \text{Word2}) :-
\text{append}(\text{Prefix}, [X,Y|\text{Suffix}], \text{Word1}),
\text{append}(\text{Prefix}, [Y,X|\text{Suffix}], \text{Word2}).
\]

| ?- \text{one_transposition}("fred", X). \\
| X = "rfed" ; \\
| X = "ferd" ; \\
| X = "frde" ; \\
| no

Given a list of words and commas, to backtrack through the phrases delimited by commas:

\[
\text{comma_phrase}(\text{List3}, \text{Phrase}) :-
\text{append}(
\text{F}, ['|\text{Rest}], \text{List3}),
!',
( \text{Phrase} = \text{F}
; \text{comma_phrase}(
\text{Rest}, \text{Phrase})
).
\]

\[
\text{comma_phrase}(\text{List3}, \text{List3}).
\]

| ?- \text{comma_phrase}([this,is,'
,um,'
,an,
\text{example}], X). \\
| X = [this,is] ; \\
| X = [um] ; \\
| X = [an,example] ; \\
| no
Exceptions
None.

See Also
Section 4.8.3 [ref-lte-acl], page 128, library(lists).
11.3.8 arg/3

Synopsis

\[ \text{arg}(+\text{ArgNum}, +\text{Term}, -\text{Arg}) \]

unifies \text{Arg} with the \text{ArgNum}th argument of term \text{Term}.

Arguments

\text{ArgNum} \quad \text{integer}, \text{must be nonvar and non-negative.}

\text{Term} \quad \text{compound}, \text{must be nonvar}

\text{Arg} \quad \text{term}

Description

The arguments are numbered from 1 upwards.

Exceptions

instantiation_error

\text{if ArgNum} \text{or Term is unbound.}

type_error

\text{if ArgNum} is not an integer, \text{or Term is not compound.}

domain_error

\text{if ArgNum} is an integer less than zero.

Examples

\[
| ?- \text{arg}(2, \text{foo}(a,b,c), X).
\]

\[
X = b
\]

See Also

functor/3, =../2, Section 4.8.2 [ref-lte-act], page 127.
11.3.9 ask_query/4  

Synopsis

ask_query(+QueryClass, +Query, +Help, -Answer)

Prints the question Query, then reads and processes user input according to QueryClass, and returns the result of the processing, the abstract answer term Answer. The Help message may be printed in case of invalid input.

Arguments

<table>
<thead>
<tr>
<th>QueryClass</th>
<th>term, must be nonvar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Determines the allowed values for the atom Answer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A message term.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Help</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A message term.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Answer</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>See QueryClass</td>
</tr>
</tbody>
</table>

Description

All queries made by the system are handled by calling this predicate.

First ask_query/4 calls query_hook/6 with the same arguments plus the Query and Help arguments converted to format-command lines. If this call succeeds, then it overrides all further processing done by ask_query/4. Otherwise, the query is processed in the following way:

- Preparation phase: The parameters of the query processing, defined by QueryClass (Prompt, InputMethod, MapMethod and FailureMode) are retrieved using the four step procedure described above. That is, the following alternatives are tried:
  - user:query_class_hook/5;
  - 'SU_messages':query_class/5;
  - the built-in copy of query_class/5.

- Input phase: The user is prompted with Prompt, input is read according to InputMethod, and the result is returned in RawInput. The four step procedure is used for performing this phase, the predicates tried are the following:
  - user:query_input_hook/3;
  - 'SU_messages':query_input/3;
  - the built-in copy of query_input/3.

- Mapping phase: The RawInput returned by the input phase is mapped to the Answer of the query. This mapping is defined by the MapMethod parameter, and the result of the conversion is returned in Result, which can be:
- success—the mapping was successful, Answer is valid;
- failure—the mapping was unsuccessful, the query has to be repeated;
- failure(Warning)—same as failure, but first the given warning message has to be printed.

The four step procedure is used for performing this phase, the predicates tried are the following:
- user:query_map_hook/4;
- 'SU_messages':query_map/4;
- the built-in copy of query_map/4.

If the mapping phase succeeds, then ask_query/4 returns with the Answer delivered by this phase.
- If the mapping does not succeed, then the query has to be repeated. If the Result returned by the mapping contains a warning message, then it is printed using print_message/2. FailureMode specifies whether to print the help message and whether to re-print the query text. Subsequently, the input and mapping phases are called again, and this is repeated until the mapping is successful.

Exceptions
instantiation_error
    QueryClass, Query, or Help uninstantiated.

type_error
    QueryClass not an atom.

domain_error
    QueryClass not a valid query class.

See Also
Section 4.16.3 [Query Processing], page 216.
11.3.10 assert/[1,2]

Synopsis

These predicates add a dynamic clause, Clause, to the Prolog database. They optionally return a database reference in Ref:

assert(+Clause)

assert(+Clause, -Ref)

It is undefined whether Clause will precede or follow the clauses already in the database.

Arguments

:Clause callable, must be nonvar
   A valid dynamic Prolog clause.
Ref db_reference, must be var
   A database reference, which uniquely identifies the newly asserted Clause.

Description

Clause must be of the form:

   Head
or  Head :- Body
or  M:Clause

where Head is of type callable and Body is a valid clause body. If specified, then M must be an atom.

assert(Head) means assert the unit-clause Head. The exact same effect can be achieved by assert((Head :- true)).

If Body is uninstantiated, then it is taken to mean call(Body). For example, (A) is equivalent to (B):

    | ?- assert((p(X) :- X)).       (A)
    | ?- assert((p(X) :- call(X))). (B)

Ref should be uninstantiated; a range exception is signaled if Ref does not unify with its return value. This exception is signaled after the assert has been completed.

The procedure for Clause must be dynamic or undefined. If it is undefined, then it is set to be dynamic.

When an assert takes place, the new clause is immediately seen by any subsequent call to the procedure. However, if there is a currently active call of the procedure at the time the clause is asserted, then the new clause is not encountered on backtracking by that call. See Section 4.12.1 [ref-mdb-bas], page 176, for further explanation of what happens when currently running code is modified.
Any uninstantiated variables in the Term will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

**Exceptions**

**instantiation_error**

- *Head* (in *Clause*) or *M* is uninstantiated.

**type_error**

- *Head* is not a callable, or *M* is not an atom, or *Body* is not a valid clause body.

**permission_error**

- the procedure corresponding to *Head* is not dynamic

**uninstantiation_error**

- *Ref* is not a variable

**See Also**

Section 4.12.4 [ref-mdb-acd], page 179.
11.3.11 asserta/[1,2]

Synopsis
These predicates add a dynamic clause, Clause, to the Prolog database. They optionally return a database reference in Ref:

asserta(+Clause)

asserta(+Clause, -Ref)

Clause will precede all existing clauses in the database.

Arguments
:Clause callable, must be nonvar
A valid dynamic Prolog clause.

Ref db_reference, must be var
A database reference, which uniquely identifies the newly asserted Clause.

Description
Clause must be of the form:

\[
\begin{align*}
\text{Head} \\
\text{or} & \quad \text{Head} :\text{- Body} \\
\text{or} & \quad M:\text{Clause}
\end{align*}
\]

where Head is of type callable and Body is a valid clause body. If specified, then M must be an atom.

asserta(Head) means assert the unit-clause Head. The exact same effect can be achieved by asserta((Head :- true)).

If Body is uninstantiated, then it is taken to mean call(Body). For example, (A) is equivalent to (B):

| l ?- asserta((p(X) :- X)). \quad (A) \\
| l ?- asserta((p(X) :- call(X))). \quad (B)

Ref should be uninstantiated; a range exception is signaled if Ref does not unify with its return value. This exception is signaled after the assert has been completed.

The procedure for Clause must be dynamic or undefined. If it is undefined, then it is set to be dynamic.

When an assert takes place, the new clause is immediately seen by any subsequent call to the procedure. However, if there is a currently active call of the procedure at the time the clause is asserted, then the new clause is not encountered on backtracking by that call. See Section 4.12.1 [ref-mdb-bas], page 176, for further explanation of what happens when currently running code is modified.
Any uninstantiated variables in the *Term* will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

**Exceptions**

*instantiation_error*

*Head* (in *Clause*) or *M* is uninstantiated.

*type_error*

*Head* is not a callable, or *M* is not an atom, or *Body* is not a valid clause body.

*permission_error*

the procedure corresponding to *Head* is not dynamic

*uninstantiation_error*

*Ref* is not a variable

**See Also**

Section 4.12.4 [ref-mdb-acd], page 179.
11.3.12 assertz/[1,2]  

Synopsis

These predicates add a dynamic clause, Clause, to the Prolog database. They optionally return a database reference in Ref:

assertz(+Clause)

assertz(+Clause, -Ref)

Clause will follow all existing clauses in the database.

Arguments

:Clause callable, must be nonvar
   A valid dynamic Prolog clause.

Ref db_reference, must be var
   A database reference, which uniquely identifies the newly asserted Clause.

Description

Clause must be of the form:

   Head
   or  Head :- Body
   or  M:Clause

where Head is of type callable and Body is a valid clause body. If specified, then M must be an atom.

assertz(Head) means assert the unit-clause Head. The exact same effect can be achieved by assertz((Head :- true)).

If Body is uninstantiated, then it is taken to mean call(Body). For example, (A) is equivalent to (B):

| ?- assertz((p(X) :- X)). | (A) |
| ?- assertz((p(X) :- call(X))). | (B) |

Ref should be uninstantiated; a range exception is signaled if Ref does not unify with its return value. This exception is signaled after the assert has been completed.

The procedure for Clause must be dynamic or undefined. If it is undefined, then it is set to be dynamic.

When an assert takes place, the new clause is immediately seen by any subsequent call to the procedure. However, if there is a currently active call of the procedure at the time the clause is asserted, then the new clause is not encountered on backtracking by that call.

See Section 4.12.1 [ref-mdb-bas], page 176, for further explanation of what happens when currently running code is modified.
Any uninstantiated variables in the Term will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

**Exceptions**

**instantiation_error**

- Head (in Clause) or M is uninstantiated.

**type_error**

- Head is not a callable, or M is not an atom, or Body is not a valid clause body.

**permission_error**

- the procedure corresponding to Head is not dynamic

**uninstantiation_error**

- Ref is not a variable

**Examples**

| ?- assertz(mammal(kangaroo)). |
| yes |
| ?- assertz(mammal(whale), Ref). |
| Ref = '$ref'(1258504,210) ? RET |
| yes |
| ?- listing(mammal). |
| mammal(kangaroo). |
| mammal(whale). |
| yes |

**See Also**

Section 4.12.4 [ref-mdb-acd], page 179.
11.3.13 at_end_of_line/[0,1]

Synopsis

at_end_of_line

at_end_of_line(+Stream)

Test whether end of line has been reached for the current input stream or for the input stream Stream.

Arguments

Stream stream_object, must be ground
A valid Prolog input stream, defaults to the current input stream.

Description

Succeeds when end of line is reached for the specified input stream. An input stream reaches end of line when all the characters except LFD of the current line have been read, or when the end of the stream has been reached (or passed). This predicate will peek ahead to determine the result, and will block if necessary until a character becomes available.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

at_end_of_stream/[0,1], skip_line/[0,1], set_input/1.
11.3.14 at_end_of_stream/[0,1]

Synopsis

at_end_of_stream

at_end_of_stream(+Stream)

Tests whether the end has been reached for the current input stream or for the input stream Stream.

Arguments

Stream stream object, must be ground

A valid Prolog input stream, defaults to the current input stream.

Description

Checks whether the end has been reached for the specified input stream. An input stream reaches the end when all items (characters or bytes) except ‘EOF’ (-1) of the stream have been read. It remains at the end after ‘EOF’ has been read.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

Comments

Note that at_end_of_stream/[0,1] never blocks. If reading ahead would block, then at_end_of_stream/[0,1] will fail, even if the stream is actually at its end. If you want to ensure that the end-of-stream condition is always properly detected, even if that entails blocking until further input is possible, then you can use peek_code/[1,2] or peek_byte/[1,2], e.g. something like the following:

```
blocking_at_end_of_stream(Alias) :-
  atom(Alias),
  stream_property(S, alias(Alias)), !,
  blocking_at_end_of_stream(S).
blocking_at_end_of_stream(S) :-
  at_end_of_stream(S), !.
blocking_at_end_of_stream(S) :-
  stream_property(S, type(text)), !,
  peek_code(S, -1).
blocking_at_end_of_stream(S) :-
  stream_property(S, type(binary)), !,
  peek_byte(S, -1).
```

This is similar to what SICStus 3 and some other Prolog implementations do.

Please note: The design of at_end_of_stream/[0,1] makes it inherently unreliable. It is present only for ISO standards compliance. It is better to read or peek until one of the end of file indications is returned.
See Also

at_end_of_line/[0,1].
11.3.15 atom/1

Synopsis

atom(+Term)

Succeeds if Term is currently instantiated to an atom.

Arguments

Term  term

Examples

| ?- atom(pastor).

yes
| ?- atom(Term).

no
| ?- atom(1).

no
| ?- atom('Time').

yes

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.16 atom_chars/2

Synopsis

atom_chars(+Atom, -Chars)
atom_chars(-Atom, +Chars)

Chars is the chars comprising the printed representation of Atom.

Arguments

Chars chars
The chars comprising the printed representation of Atom.

Atom atom
The atom containing exactly those characters, even if the characters look like
the printed representation of a number.

Description

Initially, either Atom must be instantiated to an atom, or Chars must be instantiated to a proper chars.

Any atom that can be read or written by Prolog can be constructed or decomposed by atom_chars/2.

Exceptions

instantiation_error
Atom is uninstantiated and Chars is not instantiated enough.

type_error
Atom is not an atom or Chars cannot be unified with a chars.

representation_error
Chars is a list corresponding to an atom that can’t be represented

The check of Chars when Atom is instantiated was added in release 4.3 for alignment with
the ISO Prolog standard. Previous releases simply failed in this case instead of reporting
an error for malformed Chars.

See Also

atom_codes/2.
11.3.17 atom_codes/2

Synopsis

atom_codes(+Atom, -Codes)
atom_codes(-Atom, +Codes)

*Codes* is the *codes* comprising the printed representation of *Atom*.

Arguments

*Codes*  
*codes*
  
The *codes* comprising the printed representation of *Atom*.

*Atom*  
*atom*
  
The atom containing exactly those characters, even if the characters look like the printed representation of a number.

Description

Initially, either *Atom* must be instantiated to an atom, or *Codes* must be instantiated to a proper *codes*.

Any atom that can be read or written by Prolog can be constructed or decomposed by atom_codes/2.

Exceptions

instantiation_error
  
  *Atom* is uninstantiated and *Codes* is not instantiated enough

type_error
  
  *Atom* is not an atom or *Codes* cannot be unified with a list of integers

representation_error
  
  An element of *Codes* is an invalid character code, or *Codes* is a list corresponding to an atom that can’t be represented

The check of *Codes* when *Atom* is instantiated was added in release 4.3 for alignment with the ISO Prolog standard. Previous releases simply failed in this case instead of reporting an error for malformed *Codes*.

See Also

atom_chars/2.
11.3.18 atom_concat/3  

Synopsis

\[
\text{atom_concat}(+\text{Atom1},+\text{Atom2},-\text{Atom12})
\]

\[
\text{atom_concat}(-\text{Atom1},-\text{Atom2},+\text{Atom12})
\]

The characters of the atom \text{Atom1} concatenated with those of \text{Atom2} are the same as the characters of atom \text{Atom12}.

Arguments

\[\begin{align*}
\text{Atom1} & \quad \text{atom} \\
\text{Atom2} & \quad \text{atom} \\
\text{Atom12} & \quad \text{atom}
\end{align*}\]

Description

Initially, either both \text{Atom1} and \text{Atom2}, or \text{Atom12}, must be instantiated to atoms. If only \text{Atom12} is instantiated, then nondeterminately enumerates all possible atom-pairs that concatenate to the given atom, e.g.:

\[
| ?- \text{atom_concat}(A, B, 'ab').
\]

\[
\begin{align*}
A &= '', \\
B &= ab ? ; \\
A &= a, \\
B &= b ? ; \\
A &= ab, \\
B &= '' ; \\
\text{no}
\end{align*}
\]

Exceptions

\text{instantiation_error}

More than one argument uninstantiated.

\text{type_error}

An instantiated argument is not an atom.

\text{representation_error}

\text{Atom12} is too long to be represented.

See Also

\text{atom_length/2, sub_atom/5}.  

ISO
11.3.19 atom_length/2

Synopsis

atom_length(+Atom, -Length)

Length is the number of characters of the atom Atom.

Arguments

Atom atom, must be nonvar
Length integer

Exceptions

instantiation_error
    Atom is uninstantiated

type_error
    Atom is not an atom, or Length cannot be unified with an integer

domain_error
    Length < 0

See Also

atom_length/2, atom_concat/3, sub_atom/5.
11.3.20 atomic/1

Synopsis
atomic(+Term)

Succeeds if Term is currently instantiated to an atom or a number.

Arguments
Term term

Examples
| ?- atomic(9).
  yes
| ?- atomic(a).
  yes
| ?- atomic("a").
  no
| ?- assert(foo(1), Ref), atomic(Ref).
  no

Exceptions
None.

See Also
Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.21 bagof/3

Synopsis

\[ \text{bagof}(+\text{Template}, +\text{Generator}, -\text{Set}) \]

Like \text{setof/3} except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. This relaxation saves time and space in execution.

Arguments

- **Template** term
- **Generator** callable, must be nonvar
  - A goal to be proved as if by \text{call/1}.
- **Set** list of term, non-empty set

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

Examples

See \text{findall/3} for examples that illustrate the differences among \text{findall/3}, \text{setof/3}, and \text{bagof/3}.

See Also

\text{findall/3}, \text{setof/3}, ^/2, Section 4.13 [ref-all], page 186.
11.3.22 bb_delete/2

Synopsis

bb_delete(:Key, -Term)

If a term is currently stored under Key, then the term is deleted, and a copy of it is unified with Term. Otherwise, bb_delete/2 silently fails.

Arguments

:Key       bbkey
Term       term

Exceptions

instantiation_error
    Key is not instantiated

type_error
    Key is not an atom or a small integer.

See Also

Section 4.12.9 [ref-mdb-bbd], page 184.
11.3.23  bb_get/2

Synopsis

bb_get(:Key, -Term)

If a term is currently stored under Key, then a copy of it is unified with Term. Otherwise, bb_get/2 silently fails.

Arguments

:Key  bbkey
Term  term

Exceptions

instantiation_error
  Key is not instantiated

type_error
  Key is not an atom or a small integer.

See Also

Section 4.12.9 [ref-mdb-bbd], page 184.
11.3.24  bb_put/2

Synopsis

bb_put(:Key, +Term)

A copy of Term is stored under Key in the source module blackboard. Any previous term stored under the same Key is simply deleted.

Arguments

:Key       bbkey
Term       term

Description

Any uninstantiated variables in the Term will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

Exceptions

instantiation_error
  Key is not instantiated

type_error
  Key is not an atom or a small integer.

See Also

Section 4.12.9 [ref-mdb-bbd], page 184.
11.3.25  bb_update/3

Synopsis

bb_update(:Key, -OldTerm, +NewTerm)

If a term is currently stored under Key and unifies with OldTerm, then the term is replaced by a copy of NewTerm. Otherwise, bb_update/3 silently fails. This predicate provides an atomic swap operation.

Arguments

:Key        bbkey
OldTerm     term
NewTerm     term

Description

Any uninstantiated variables in the Term will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

Exceptions

instantiation_error
   Key is not instantiated

type_error
   Key is not an atom or a small integer.

See Also

Section 4.12.9 [ref-mdb-bbd], page 184.
11.3.26 block/1
declaration

Synopsis
:- block +BlockSpec

Specifies conditions for blocking goals of the predicates referred to by BlockSpec.

Arguments
:BlockSpec
callable, must be ground
Goal template or list of goal templates, of the form f(Arg1, Arg2,...). Each Argn is one of:
‘-’ part of a block condition
‘?’ otherwise

Description
When a goal for a block declared predicate is to be executed, the block specs are interpreted as conditions for blocking the goal, and if at least one condition evaluates to true, then the goal is blocked.

A block condition evaluates to true iff all arguments specified as ‘-’ are uninstantiated, in which case the goal is blocked until at least one of those variables is instantiated. If several conditions evaluate to true, then the implementation picks one of them and blocks the goal accordingly.

The recommended style is to write the block declarations in front of the source code of the predicate they refer to. Indeed, they are part of the source code of the predicate, and must precede the first clause. For example, with the definition:

:- block merge(-,?,-), merge(?,-,-).

merge([], Y, Y).
merge(X, [], X).
merge([H|X], [E|Y], [H|Z]) :- H @< E, merge(X, [E|Y], Z).
merge([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).

calls to merge/3 having uninstantiated arguments in the first and third position or in the second and third position will suspend.

The behavior of blocking goals for a given predicate on uninstantiated arguments cannot be switched off, except by abolishing or redefining the predicate.

Exceptions
Exceptions in the context of loading code are printed as error messages.

instantiation_error
BlockSpec not ground.
type_error
   BlockSpec not a valid specification.

context_error
   Declaration appeared in a goal.

permission_error
   Declaration appeared as a clause.

See Also
Section 4.3.4.5 [Block Declarations], page 84.
11.3.27 break/0

devlopment

Synopsis
break

takes the current execution to be interrupted; enters next break level.

Description
The first time break/0 is called, it displays the message

```
  % Break level 1
  % 1
  | ?-
```

The system is then ready to accept input as though it were at top level. If another call to break/0 is encountered, then it moves up to level 2, and so on. The break level is displayed on a separate line before each top-level prompt.

To close a break level and resume the execution that was suspended, type `^D`. break/0 then succeeds, and execution of the interrupted program is resumed.

Changes can be made to a running program while in a break level. Any change made to a procedure will take effect the next time that procedure is called. See Section 4.12.5 [ref-mdb-rcc], page 179, for details of what happens if a procedure that is currently being executed is redefined. When a break level is entered, the debugger is turned off (although leashing and spypoints are retained). When a break level is exited, the debugging state is restored to what it was before the break level was entered.

Often used via the debugging option b.

Exceptions
Catches otherwise uncaught exceptions and issues an error message.

See Also
abort/0, halt/[0,1], Section 3.9 [Nested], page 27.
11.3.28  \texttt{breakpoint\_expansion/2} \hspace{1cm} \textit{hook, development}

Synopsis

\begin{verbatim}
:- multifile user:breakpoint\_expansion/2.

user:breakpoint\_expansion(+Macro, -Body)
\end{verbatim}

Defines debugger condition macros.

Arguments

\begin{itemize}
\item \textit{Macro} \hspace{1cm} \textit{term}
  \begin{itemize}
  \item Breakpoint test or action.
  \end{itemize}
\item \textit{Body} \hspace{1cm} \textit{term}
  \begin{itemize}
  \item Expanded breakpoint test or action, may be composite.
  \end{itemize}
\end{itemize}

Exceptions

Exceptions are treated as failures, except an error message is printed as well.

See Also

Section 5.9 [Breakpoint Conditions], page 277.
11.3.29  byte_count/2

Synopsis
byte_count(+Stream, -Count)

Obtains the total number of bytes either input from or output to the open binary stream
Stream and unifies it with Count.

Arguments
Stream  stream_object, must be ground
         A valid open binary stream.
Count  integer
         The resulting byte count of the stream.

Description
A freshly opened stream has a byte count of 0. When a byte is input from or output to a
Prolog stream, the byte count of the Prolog stream is increased by one.

The count is reset by set_stream_position/2.

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also
byte_count/2, line_count/2, line_position/2, stream_position/2, set_stream_position/2, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.30  call/[1,2,\ldots,255]  \textit{ISO}

Synopsis

\texttt{call(+P)}

Proves (executes) \(P\).

\texttt{call(+P,?Q,\ldots)}

Executes the goal obtained by augmenting \(P\) by the remaining arguments.

Arguments

\begin{itemize}
  \item \texttt{:P} \textit{callable}, must be nonvar
  \item \texttt{Q} \textit{term} ...
\end{itemize}

Description

If \(P\) is instantiated to an atom or compound term, then the goal \texttt{call(P)} is executed exactly as if that term appeared textually in its place, except that any cut (‘!’) occurring in \(P\) only cuts alternatives in the execution of \(P\). Only \texttt{call/1..8} are required by ISO.

Backtracking

Depends on \(P\).

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

Examples

\texttt{maplist/2} in \texttt{library(lists)} is defined as:

\begin{verbatim}
maplist(Pred, Xs) :-
  (  foreach(X,Xs),
      param(Pred)
    do call(Pred, X)
  ).
\end{verbatim}

See Also

Section 4.2.5 [ref-sem-cal], page 77.
11.3.31 call_cleanup/2

\[ \text{call_cleanup}(+\text{Goal}, +\text{Cleanup}) \]

**Synopsis**

Executes the procedure call \text{Goal}. When \text{Goal} succeeds determinately, is cut, fails, or raises an exception, \text{Cleanup} is executed.

**Arguments**

: \text{Goal} callable, must be nonvar

: \text{Cleanup} callable, must be nonvar

**Description**

This construction can be used to ensure that \text{Cleanup} is executed as soon as \text{Goal} has completed execution, no matter how it finishes. In more detail:

When \text{call_cleanup/2} with a continuation \text{C} is called or backtracked into, first \text{Goal} is called or backtracked into. Then there are four possibilities:

1. \text{Goal} succeeds determinately, possibly leaving some blocked subgoals. \text{Cleanup} is executed with continuation \text{C}.
2. \text{Goal} succeeds with some alternatives outstanding. Execution proceeds to \text{C}. If a cut that removes the outstanding alternatives is encountered, then \text{Cleanup} is executed with continuation to proceed at the cut. Also, if an exception \text{E} that will be caught by an ancestor of the \text{call_cleanup/2 Goal} is raised, then \text{Cleanup} is executed with continuation \text{raise_exception}(\text{E}).
3. \text{Goal} fails. \text{Cleanup} is executed with continuation \text{fail}.
4. \text{Goal} raises an exception \text{E}. \text{Cleanup} is executed with continuation \text{raise_exception}(\text{E}).

In a typical use of \text{call_cleanup/2}, \text{Cleanup} succeeds determinately after performing some side effect; otherwise, unexpected behavior may result.

Note that the Prolog top-level operates as a read-execute-fail loop, which backtracks into or cuts the query when the user types ; or RET respectively. Also, some predicates, such as \text{halt/[0,1]} and \text{abort/0}, are implemented in terms of exceptions. All of these circumstances can trigger the execution of \text{Cleanup}.

**Backtracking**

Depends on the arguments.

**Exceptions**

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

**See Also**

Section 4.2 [ref-sem], page 61.
11.3.32 call_residue_vars/2

\[ \text{call_residue_vars}(+\text{Goal}, -\text{Vars}) \]

**Synopsis**

Executes the procedure call \text{Goal}, unifying \text{Vars} with the list of residual variables that have blocked goals or attributes attached to them.

**Arguments**

:Goal callable, must be nonvar

Vars list of var

**Description**

\text{Goal} is executed as if by \text{call/1}. \text{Vars} is unified with the list of new variables created during the call that remain unbound and have blocked goals or attributes attached to them. For example:

\[ | ?- \text{call_residue_vars}((\text{dif}(X,f(Y)), X=f(Z)), \text{Vars}). \]

\[ X = f(Z), \]
\[ \text{Vars} = [Z,Y], \]
\[ \text{prolog:dif}(f(Z),f(Y)) ? \]

**Backtracking**

Depends on \text{Goal}.

**Exceptions**

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

**See Also**

Section 4.2.4 [ref-sem-sec], page 74.
11.3.33 callable/1

**Synopsis**
callable(+Term)

Succeeds if *Term* is currently instantiated to an atom or a compound term.

**Arguments**
Term  term

**Examples**
| ?- callable(a).

  yes
| ?- callable(a(1,2,3)).

  yes
| ?- callable([1,2,3]).

  yes
| ?- callable(1.1).

  no

**Exceptions**
None.

**See Also**
Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.34 catch/3

Synopsis

\texttt{catch(+ProtectedGoal, -Exception, +Handler)}

Specify an exception handler for \textit{ProtectedGoal}, and call \textit{ProtectedGoal}, as described in Section 4.15 [ref-ere], page 197.

Arguments

:\textit{ProtectedGoal} callable, must be nonvar

\textit{Exception} term

:\textit{Handler} callable, must be nonvar

Examples

Fail on exception:

\begin{verbatim}
:- meta_predicate fail_on_exception(0).
fail_on_exception(C):-
catch(C, E, print_exception_then_fail(C, E)).

print_exception_then_fail(C, E) :-
    format(user_error, 'Exception occurred while calling ^q:^n', [C]),
    print_message(warning, E),
    fail.
\end{verbatim}

Backtracking

Depends on \textit{ProtectedGoal} and \textit{Handler}.

Exceptions

None.

See Also

Section 4.15 [ref-ere], page 197.
11.3.35 char_code/2

Synopsis

char_code(+Char, -Code)

char_code(-Char, +Code)

**Code** is the character code comprising the printed representation of **Char**.

**Arguments**

*Char* char

The char whose code is **Code**.

*Code* code

The code corresponding to **Char**.

**Description**

Initially, at least one argument must be instantiated.

**Exceptions**

instantiation_error

Char and Code are both uninstantiated

type_error

Char is not a char or Code is not an integer.

representation_error

Code is not a code.

**See Also**

atom_codes/2, number_codes/2.
11.3.36 char_conversion/2

Synopsis

char_conversion(+InChar, +OutChar)

The mapping of InChar to OutChar is added to the character-conversion mapping.

Arguments

InChar char, must be nonvar
OutChar char, must be nonvar

Description

The mapping of InChar to OutChar is added to the character-conversion mapping. This means that in all subsequent term and program input operations any unquoted occurrence of InChar will be replaced by OutChar. The rationale for providing this facility is that in some extended character sets (such as Japanese JIS character sets) the same character can appear several times and thus have several codes, which the users normally expect to be equivalent. It is advisable to always quote the arguments of char_conversion/2.

Any previous mapping of InChar is replaced by the new one.

Please note: the mapping is global, as opposed to being local to the current module, Prolog text, or otherwise.

Exceptions

instantiation_error
    An argument is uninstantiated

type_error
    An argument is not a char

See Also

Chapter 2 [Glossary], page 7.
11.3.37 character_count/2

Synopsis

character_count(+Stream, -Count)

Obtains the total number of characters either input from or output to the open text stream Stream and unifies it with Count.

Arguments

Stream  stream_object, must be ground
        A valid open text stream
Count    integer
        The resulting character count of the stream

Description

A freshly opened text stream has a character count of 0. When a character is input from or output to a non-interactive Prolog stream, the character count of the Prolog stream is increased by one. Character count for an interactive stream reflects the total character input from or output to any interactive stream, i.e. all interactive streams share the same counter.

A n1/[0,1] operation also increases the character count of a stream by one.

The count is reset by set_stream_position/2.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

byte_count/2, line_count/2, line_position/2, stream_position/2, set_stream_position/2, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.38 clause/[2,3]  

**Synopsis**

clause(+Head, -Body)

clause(+Head, -Body, -Ref)

clause(-Head, -Body, +Ref)

Searches the database for a clause whose head matches `Head` and whose body matches `Body`.

**Arguments**

:Head  callable
    A term whose functor names a dynamic procedure.

Body  callable

Ref  `db_reference`

**Description**

Initially, at least one of `Head` and `Ref` must be instantiated.

In the case of unit-clauses, `Body` is unified with `true`.

If a procedure consists entirely of unit-clauses, then there is no point in calling `clause/2` on it. It is simpler and faster to call the procedure.

In `clause/3`, either `Head` or `Ref` must be instantiated. If `Ref` is instantiated, then `(Head :- Body)` is unified with the clause identified by `Ref`. (If this clause is a unit-clause, then `Body` is unified with `true`.)

If the predicate did not previously exist, then it is created as a dynamic predicate and `clause/2` fails. If `Ref` is not instantiated, then `clause/3` behaves exactly like `clause/2` except that the database reference is returned.

By default, clauses are accessed with respect to the source module.

**Backtracking**

Can be used to backtrack through all the clauses matching a given `Head` and `Body`. It fails when there are no (or no further) matching clauses in the database.

**Exceptions**

- **instantiation_error**
  
  Neither `Head` nor `Ref` is instantiated.

- **type_error**
  
  `Head` is not a callable, or `Ref` is not a well-formed `db_reference`

- **permission_error**
  
  Procedure is not dynamic.
existence_error

Ref is a well-formed db_reference but does not correspond to an existing clause or record.

Comments

If clause/[2,3] is called on an undefined procedure, then it fails, but before failing it makes the procedure dynamic. This can be useful if you wish to prevent unknown procedure catching from happening on a call to that procedure.

It is not a limitation that Head is required to be instantiated in clause(Head, Body), because if you want to backtrack through all clauses for all dynamic procedures, then this can be achieved by:

| ?- predicate_property(P,dynamic), clause(P,B).

If there are clauses with a given name and arity in several different modules, or if the module for some clauses is not known, then the clauses can be accessed by first finding the module(s) by means of current_predicate/2. For example, if the procedure is f/1:

| ?- current_predicate(_,M:f(_)), clause(M:f(X),B).

clause/3 will only access clauses that are defined in, or imported into, the source module, except that the source module can be overridden by explicitly naming the appropriate module. For example:

| ?- assert(foo:bar,R).

R = '$ref'(771292,1)

| ?- clause(H,B,'$ref'(771292,1)).

no

| ?- clause(foo:H,B,'$ref'(771292,1)).

H = bar,
B = true

Accessing a clause using clause/2 uses first argument indexing when possible, in just the same way that calling a procedure uses first argument indexing. See Section 9.5 [Indexing], page 360.

clause/2 is part of the ISO Prolog standard; clause/3 is not.

See Also

instance/2, assert/[1,2], dynamic/1, retract/1, Section 4.12.6 [ref-mdb-acl], page 181.
11.3.39 close/[1,2]  ISO

Synopsis

close(+Stream)

close(+Stream, +Options)

closes the stream corresponding to Stream.

Arguments

Stream  stream_object, must be ground
Stream or file specification.

Options  list of term, must be ground
A list of zero or more of the following:

force(Boolean)
  Specifies whether SICStus Prolog is to close the stream forcefully,
even in the presence of errors (true), or not (false). The latter is
the default. Currently this option has no effect.

direction(+Direction)
  Direction is an atom specifying the direction or directions to close.
  One of:
    input  Close only the input direction, if open.
    output Close only the output direction, if open.
    all    Close all directions. This is the default.

if stream is not open in the specified direction then the call to
open/4 does nothing.
Closing a single direction is mainly useful when dealing with bidi-
rectional streams, such as sockets.

Description

If Stream is a stream object, then if the corresponding stream is open, then it will be closed
in the specified directions; otherwise, an error exception is raised.

If Stream is a file specification, then the corresponding stream will be closed in the specified
directions, provided that the file was opened by see/1 or tell/1.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iov-sfh-est], page 109), plus:

instantiation_error
  Options is not instantiated enough.

type_error
  Options is not a proper list.
domain_error
   Options contains an invalid option.

permission_error
   File not opened by see/1 or tell/1.

domain_error
   Stream is neither a filename nor a stream.

Examples

In this example, foo will be closed:

    see(foo),
    ...
    close(foo)

However, in this example, a permission error will be raised and foo will not be closed:

    open(foo, read, S),
    ...
    close(foo)

Here, close(S) should have been used.

See Also

see/1, tell/1, open/[3,4], Section 4.6.7 [ref-iou-sfh], page 109, Section 10.41 [lib-sockets], page 754.
11.3.40 compare/3

Synopsis

\[
\text{compare}(-\text{Order}, +\text{Term1}, +\text{Term2})
\]
succeeds if the result of comparing terms \text{Term1} and \text{Term2} is \text{Order}

Arguments

\begin{align*}
\text{Order} & \quad \text{order} \\
= & \quad \text{if } \text{Term1} \text{ is identical to } \text{Term2}, \\
< & \quad \text{if } \text{Term1} \text{ is before } \text{Term2} \text{ in the standard order}, \\
> & \quad \text{if } \text{Term1} \text{ is after } \text{Term2} \text{ in the standard order}.
\end{align*}

\text{Term1} \quad \text{term} \\
\text{Term2} \quad \text{term}

Description

The standard order is described in Section 4.8.8 [ref-lte-cte], page 130. Note that the standard order is not, in general, well defined for cyclic terms.

The goal (A) is equivalent to (B):

\[
\begin{align*}
| ?- \text{compare} (=, \text{Term1}, \text{Term2}). & \quad \text{(A)} \\
| ?- (\text{Term1} == \text{Term2}). & \quad \text{(B)}
\end{align*}
\]

The following query succeeds, binding \text{R} to \text{<}, because 1 comes before 2 in the standard order.

\[
| ?- \text{compare}(\text{R}, 1, 2).
\]

\text{R} = \text{<}

If \text{Order} is supplied, and is not one of \text{<}, \text{>}, \text{or} =, then an error is thrown, as follows.

Exceptions

\begin{align*}
\text{type_error} & \quad \text{atom, Order} \\
& \quad \text{if } \text{Order} \text{ is neither a variable nor an atom.}
\end{align*}

\begin{align*}
\text{domain_error} & \quad \text{order, Order} \\
& \quad \text{if } \text{Order} \text{ is an atom but not } \text{<}, \text{ =}, \text{or } \text{>}.
\end{align*}

These errors were added in SICStus Prolog 4.3 for alignment with the ISO Prolog standard. Previous versions of SICStus Prolog simply failed instead of reporting an error.

See Also

\text{@</2, @=</2, @>/2, @>=/2, SP_compare()}, Section 4.8.8 [ref-lte-cte], page 130.
11.3.41 compile/1

Synopsis
compile(+Files)

Compiles the specified Prolog source file(s) into memory.

Arguments
:Files file_spec or list of file_spec, must be ground
A file specification or a list of file specifications; extensions optional.

Description
This predicate is defined as if by:

\[
\text{compile}(\text{Files}) :- \\
\quad \text{load_files}(\text{Files}, [\text{load_type(source)},\text{compilation_mode(compile)}]).
\]

Exceptions
See load_files/[2,3].

See Also
Section 4.3.2 [ref-lod-lod], page 80.
11.3.42 compound/1

Synopsis

compound(+Term)

Term is currently instantiated to a compound term.

Arguments

Term term

Examples

| ?- compound(9).

no

| ?- compound(a(1,2,3)).

yes

| ?- compound("a").

yes

| ?- compound([1,2]).

yes

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.43 consult/1

Synopsis

consult(+Files)

Consults the specified Prolog source file(s) into memory.

Arguments

:Files file_spec or list of file_spec, must be ground

A file specification or a list of file specifications; extensions optional.

Description

This predicate is defined as if by:

\[
\text{consult}(\text{Files}) :\text{-} \\
\quad \text{load_files(Files, [load_type(source), compilation_mode(consult)]).}
\]

Exceptions

See load_files/[2,3].

See Also

Section 4.3.2 [ref-lod-lod], page 80.
**11.3.44 copy_term/[2,3]**

**ISO**

**Synopsis**

```prolog
copy_term(+Term, -Copy)
```

Unifies *Copy* with a copy of *Term* in which all variables have been replaced by brand new variables, and all mutables by brand new mutables.

```prolog
copy_term(+Term, -Copy, -Body)
```

Furthermore, if *Term* contains variables with goals blocked on them, or variables with attributes that can be interpreted as a goal (see Section 10.3 [lib-atts], page 391), then *Body* is unified with the conjunction of such goals. If no such goals are present, then *Body* is unified with the atom *true*. The idea is that executing *Body* will reinstate blocked goals and attributes on the variables in *Copy* equivalent to those on the variables in *Term*.

**Arguments**

- **Term** term
- **Copy** term
- **Body** callable

**Description**

Independent copies are substituted for any mutable terms in *term*. It behaves as if defined by:

```prolog
copy_term(X, Y) :-
    assert('copy of'(X)),
    retract('copy of'(Y)).
```

The implementation of *copy_term/2* endeavors to conserve space by not copying ground subterms.

When you call *clause/[2,3]* or *instance/2*, you get a new copy of the term stored in the database, in precisely the same sense that *copy_term/2* gives you a new copy.

**Examples**

- A naive way to attempt to find out whether one term is a copy of another:
  ```prolog
  identical_but_for_variables(X, Y) :-
      \+ \+ (\n      numbervars(X, 0, N),
      numbervars(Y, 0, N),
      X = Y\n  ).
  ```

  This solution is sometimes sufficient, but will not work if the two terms have any variables in common.

- If you want the test to succeed even when the two terms do have some variables in common, then you need to copy one of them; for example,
identical_but_for_variables(X, Y) :-
\+ \+ (  
  copy_term(X, Z),
  numbervars(Z, 0, N),
  numbervars(Y, 0, N),
  Z = Y  
).

- An example of copy_term/3. Suppose that you want to make copy_term/3 aware of the attribute tfs/1 in some module. Then with the module file:

  :- module(foo, []).
  :- use_module(library(atts)).
  :- attribute tfs/1.

  attribute_goal(X, put_atts(X, tfs(Y))) :-
    get_atts(X, tfs(Y)).

the following query works:

  | ?- foo:put_atts(X, tfs(ind)), copy_term(f(X), Copy, Body).
  Body = foo:put_atts(_A, tfs(ind)),
  Copy = f(_A),
  put_atts(X, tfs(ind)) ? RET
  yes

Comments
copy_term/2 is part of the ISO Prolog standard; copy_term/3 is not.

Exceptions
None.

See Also
Section 4.8.7 [ref-lte-cpt], page 129.
11.3.45 coverage_data/1

Synopsis
coverage_data(-Data)

since release 4.2

Data is the coverage data accumulated so far.

Arguments
Data     list of coverage_pair
where:
coverage_pair ::= counter(filename,pred_spec,clauseno,lineno)-tagged_hits
filename     ::= atom {file containing coverage site}
clauseno    ::= integer {file relative clause number containing coverage site}
lineno      ::= integer {line of code containing coverage site}
tagged_hits ::= det(hits) {no nondet calls made from site}
| nondet(hits) {some nondet calls made from site}
hits        ::= integer {number of times that coverage site was hit}

Description
The coverage data accumulated so far is collected into a term of type list of coverage_pair
and unified with Data.

Please note: A given line of code can contain more than one coverage site.

Exceptions
None.

See Also
Section 9.3 [Coverage Analysis], page 356.
11.3.46 create Mutable/2

Synopsis

create Mutable(+Datum, -Mutable)

Mutable is a new mutable term with initial value Datum.

Arguments

Datum term, must be nonvar

Mutable mutable

Exceptions

instantiation_error

Datum is uninstantiated

See Also

Section 4.8.9 [ref-lte-mut], page 131.
11.3.47 current_atom/1

Synopsis

current_atom(?Atom)

Atom is a currently existing atom.

Arguments

Atom atom

Backtracking

If Atom is uninstantiated, then current_atom/1 can be used to enumerate all known atoms. The order in which atoms are bound to Atom on backtracking corresponds to the times of their creation.

Comments

Note that the predicate atom/1 is recommended for determining whether a term is an atom, as current_atom/1 will succeed if Atom is uninstantiated as well.

Exceptions

None.

See Also

Section 4.12.8 [ref-mdb-idb], page 183.
11.3.48 current_breakpoint/5

**Synopsis**

current_breakpoint(-Conditions, -BID, -Status, -Kind, -Type)

There is a breakpoint with conditions Conditions, identifier BID, enabledness Status, kind Kind, and type Type.

**Arguments**

:Conditions  

  term  

  Breakpoint conditions.

BID  

  integer  

  Breakpoint identifier.

Status  

  one of [on, off]  

  on for enabled breakpoints and off for disabled ones

Kind  

  one of [plain(MFunc), conditional(MFunc), generic]  

  MFunc is the module qualified functor of the specific breakpoint.

Type  

  one of [debugger, advice]

**Exceptions**

instantiation_error  

  Conditions not instantiated enough.

type_error  

  Conditions not a proper list of callable term.

domain_error  

  Conditions not a proper list of valid breakpoint conditions

**See Also**

Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 260, Section 5.7 [Breakpoint Predicates], page 272.
11.3.49  current_char_conversion/2

Synopsis

current_char_conversion(?InChar, ?OutChar)

*InChar* is currently mapped to *OutChar* in the character-conversion mapping, where the two are distinct.

**Arguments**

*InChar*  char

*OutChar*  char

**Exceptions**

type_error

An argument is instantiated not to a char.

**See Also**

Chapter 2 [Glossary], page 7.
11.3.50 current_input/1

Synopsis

current_input(-Stream)

unifies Stream with the current input stream.

Arguments

Stream : stream_object

Description

Stream is the current input stream. The current input stream is also accessed by the C variable SP_curin.

Exceptions

domain_error

Stream is instantiated not to a valid stream.

See Also

open/3, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.51 current_key/2

Synopsis

\texttt{current_key(?KeyName, ?KeyTerm)}

Succeeds when \texttt{KeyName} is the name of \texttt{KeyTerm}, and \texttt{KeyTerm} is a recorded key.

Arguments

\texttt{KeyName} \hspace{1em} \texttt{atomic}

One of:

- \texttt{KeyTerm}, if \texttt{KeyTerm} is atomic; or
- the principal functor of \texttt{KeyTerm}, if \texttt{KeyTerm} is a compound term.

\texttt{KeyTerm} \hspace{1em} \texttt{term}

The most general form of the key for a currently recorded term.

Description

This predicate can be used to enumerate in undefined order all keys for currently recorded terms through backtracking.

Backtracking

Enumerates all keys through backtracking.

Exceptions

None.

See Also

Section 4.12.8 [ref-mdb-idb], page 183.
11.3.52 current_module/[1,2]

Synopsis

\texttt{current\_module(?ModuleName)}

Queries whether a module is “current” or backtracks through all of the current modules.

\texttt{current\_module(?ModuleName, ?AbsFile)}

Associates modules with their module files.

\textbf{Arguments}

\textit{ModuleName} \hspace{1em} \texttt{atom}

\textit{AbsFile} \hspace{1em} \texttt{atom}

Absolute filename in which the module is defined.

\textbf{Description}

A loaded module becomes “current” as soon as some predicate is defined in it, and a module can never lose the property of being current.

It is possible for a current module to have no associated file, in which case \texttt{current\_module/1} will succeed on it but \texttt{current\_module/2} will fail. This arises for the special module \texttt{user} and for dynamically-created modules (see Section 4.11 [ref-mod], page 161).

If its arguments are not correct, or if \textit{Module} has no associated file, then \texttt{current\_module/2} simply fails.

\textbf{Backtracking}

\texttt{current\_module/1} backtracks through all of the current modules. The following command will print out all current modules:

\begin{verbatim}
| ?- current_module(Module), writeq(Module), nl, fail.
\end{verbatim}

\texttt{current\_module/2} backtracks through all of the current modules and their associated files.

\textbf{Exceptions}

\texttt{type\_error}

\textbf{Examples}

\begin{verbatim}
| ?- findall(M, current\_module(M),Ms).
Ms = [chr,user,prolog,'SU\_messages',clpfd] ? RET
yes
| ?- findall(M-F, current\_module(M,F),MFs).
MFs = ['SU\_messages'+'/'+src/sicstus/matsc/sicstus4/Utils/x86\_linux-glibc2.3/bin/sp-4.1.0/sicstus-4.1.0/library/SU\_messages.pl'] ?
yes
\end{verbatim}
See Also
Section 4.11.13 [ref-mod-ilm], page 169.
11.3.53  current_op/3

Synopsis

current_op(\text{?Precedence}, \text{?Type}, \text{?Name})

Succeeds when the atom \text{Name} is currently an operator of type \text{Type} and precedence \text{Precedence}.

Arguments

\text{Precedence}
\begin{itemize}
\item integer, in the range 1-1200
\end{itemize}

\text{Type}
\begin{itemize}
\item one of [\text{xfx}, \text{xfy}, \text{yfx}, \text{fx}, \text{xf}, \text{yf}]
\end{itemize}

\text{Name}
\begin{itemize}
\item atom
\end{itemize}

Description

None of the arguments need be instantiated at the time of the call; that is, this predicate can be used to find the precedence or type of an operator or to backtrack through all operators.

To add or remove an operator, use \text{op/3}.

Exceptions

\begin{itemize}
\item type_error
\begin{itemize}
\item \text{Name} not an atom or \text{Type} not an atom or \text{Precedence} not an integer.
\end{itemize}
\item domain_error
\begin{itemize}
\item \text{Precedence} not between 1-1200, or \text{Type} not one of listed atoms.
\end{itemize}
\end{itemize}

Examples

See Also

\text{op/3}, Section 4.1.5 [ref-syn-ops], page 47.
11.3.54 current_output/1

Synopsis

current_output(-Stream)

unifies Stream with the current output stream.

Arguments

Stream stream_object

Description

Stream is the current output stream. The current output stream is also accessed by the C variable SP_curout.

Exceptions

domain_error
Stream is instantiated not to a valid stream.

See Also

open/[3,4], Section 4.6.7 [ref-iou-sfh], page 109.
11.3.55 current_predicate/[1,2]

Synopsis

current_predicate(?PredSpec)

Unifies PredSpec with a predicate specifications of the form Name/Arity.

current_predicate(?Name, ?Term)

Unifies Name with the name of a user-defined predicate, and Term with the most general term corresponding to that predicate.

Arguments

:PredSpec  pred_spec
Name        atom
:Term       callable

Description

If you have loaded the predicates `foo/1` and `foo/3` into Prolog, then `current_predicate/2` would return the following:

```
| ?- current_predicate(foo, T).

T = foo(_A) ;
T = foo(_A,_B,_C) ;

no
```

Examples

- The following goals can be used to backtrack through every predicate in your program.
  
  ```
  | ?- current_predicate(Name, Module:Term).
  
  | ?- current_predicate(Module:PredSpec).
  ```

- If a module is specified, then `current_predicate/[1,2]` only succeeds for those predicates that are defined in the module. It fails for those predicates that are imported into a module.

```
| ?- current_predicate(m:P).

```

will backtrack through all predicates P that are defined in module m. To backtrack through all predicates imported by a module use `predicate_property/2` (see Section 4.9.1 [ref-lps-ove], page 135).

To find out whether a predicate is built-in, use `predicate_property/2`. 

ISO
% Is there a callable predicate named gc?

| ?- current_predicate(gc, Term).
| no
| ?- predicate_property(gc, Prop)

Prop = built_in

Exceptions

instantiation_error
type_error

in PredSpec

Comments

current_predicate/1 is part of the ISO Prolog standard; current_predicate/2 is not.

See Also

predicate_property/2, Section 4.9.1 [ref-lps-ove], page 135.
11.3.56 current_prolog_flag/2

Synopsis

current_prolog_flag(?FlagName, ?Value)

same as prolog_flag(FlagName, Value), except that current_prolog_flag(FlagName, Value) type checks FlagName.

Arguments

FlagName atom
Value term

Exceptions

type_error
FlagName is not an atom.
domain_error
FlagName is not a valid flag name.

See Also

prolog_flag/[2,3], set_prolog_flag/2, Section 4.9.4 [ref-lps-flg], page 136.
11.3.57 current_stream/3

Synopsis

current_stream(?AbsFile, ?Mode, ?Stream)

Stream is a stream, which is currently open on file AbsFile in mode Mode.

Arguments

AbsFile atom
    Absolute filename.

Mode for streams opened with open/[3,4] this is one of [read, write, append].
    For other streams Mode may have other values.

Stream stream_object

Description

- None of the arguments need be initially instantiated.

- Ignores certain pre-defined streams, e.g. the streams initially associated with user_input, user_output and user_error will not be recognized or generated by current_stream/3.

  This is unlike stream_property/2, which can backtrack over all streams, including the pre-defined ones.

Backtracking

Can be used to backtrack through all open streams.

Exceptions

None.

See Also

open/[3,4], Section 4.6.7 [ref-iou-sfh], page 109.
11.3.58  !/0

Synopsis
!
Cut.

Description
When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, then the parent goal will fail (the parent goal is the one that matched the head of the clause containing the cut).

Exceptions
None.

See Also
Section 4.2.3.1 [ref-sem-ctr-cut], page 64.
11.3.59  db_reference/1

Synopsis

db_reference(+Term)  

since release 4.1

Term is currently instantiated to a compound term with principal functor '$ref'/2 denoting a unique reference to a dynamic clause.

Arguments

Term  term

Examples

| ?- db_reference(9).

no

| ?- db_reference(_X).

no

| ?- assertz(foo(a), R), db_reference(R).

R = '$ref'(1816730,128)

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.60 debug/0

**Synopsis**

d debug

Turns on the debugger in debug mode.

**Description**

d debug/0 turns the debugger on and sets it to debug mode. Turning the debugger on in debug mode means that it will stop at the next spypoint encountered in the current execution.

The effect of this predicate can also be achieved by typing the letter d after a ^C interrupt (see Section 3.7 [Execution], page 26).

**Exceptions**

None.

**See Also**

Section 5.2 [Basic Debug], page 233.
11.3.61 debugger_command_hook/2

Synopsis

:- multifile user:debugger_command_hook/2.

user:debugger_command_hook(+DCommand, -Actions)

Allows the interactive debugger to be extended with user-defined commands. See Section 5.5 [Debug Commands], page 237.

Arguments

DCommand  \text{term}

Actions  \text{term}

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

Section 5.7 [Breakpoint Predicates], page 272.
11.3.62 debugging/0

**Synopsis**

debugging

Prints out current debugging state

**Description**

debugging/0 displays information on the terminal about the current debugging state. It shows

- The top-level state of the debugger, which is one of
  
  - **debug**: The debugger is on but will not show anything or stop for user interaction until a spypoint is reached.
  
  - **trace**: The debugger is on and will show everything. As soon as you type a goal, you will start seeing a debugging trace. After printing each trace message, the debugger may or may not stop for user interaction: this depends on the type of leashing in force (see below).
  
  - **zip**: The debugger is on but will not show anything or stop for user interaction until a spypoint is reached. The debugger does not even keep any information of the execution of the goal till the spypoint is reached and hence you will not be able to see the ancestors of the goal when you reach the spypoint.
  
  - **off**: The debugger is off.

The top-level state can be controlled by the predicates `debug/0`, `nodebug/0`, `trace/0`, `notrace/0`, `zip/0`, `nozip/0`, and `prolog_flag/3`.

- The type of leashing in force. When the debugger prints a message saying that it is passing through a particular port (one of Call, Exit, Redo, Fail, or Exception) of a particular procedure, it stops for user interaction only if that port is leashed. The predicate `leash/1` can be used to select which of the seven ports you want to be leashed.

- All the current spypoints. Spypoints are controlled by the predicates `spy/[1,2]`, `nospy/1`, `nospyall/0`, `add_breakpoint/2`, `disable_breakpoints/1`, `enable_breakpoints/1`, and `remove_breakpoints/1`.

**Exceptions**

None.

**See Also**

Section 5.2 [Basic Debug], page 233.
11.3.63 dif/2

Synopsis
dif(+X,+Y)

Constrains \(X\) and \(Y\) to represent different terms i.e. to be non-unifiable.

Arguments

\(X\) \hspace{1cm} \text{term}
\(Y\) \hspace{1cm} \text{term}

Description

Calls to \texttt{dif/2} either succeed, fail, or are blocked depending on whether \(X\) and \(Y\) are sufficiently instantiated. It is defined as if by:

\[
\text{dif}(X, Y) :- \text{when}(\neq(X,Y), X\neq Y).
\]

Exceptions

None.

See Also

Section 4.2.4 [ref-sem-sec], page 74.
11.3.64 disable_breakpoints/1

Synopsis
disable_breakpoints(+BIDs)

Disables the breakpoints specified by BIDs.

Arguments
BIDs list of integer, must be ground
   Breakpoint identifiers.

Exceptions
instantiation_error
    in BIDs

See Also
Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 260, Section 5.7 [Breakpoint Predicates], page 272.
11.3.65 discontiguous/1

**Declaration, ISO**

**Synopsis**

```prolog
:- discontiguous +PredSpecs
```

Declares the clauses of the predicates defined by `PredSpecs` to be discontiguous in the source file (suppresses compile-time warnings).

**Arguments**

`:PredSpecs`  
pred_spec_forest, must be ground

A predicate specification, or a list of such, or a sequence of such separated by commas.

**Comments**

discontiguous is not an ISO predefined prefix operator.

**Exceptions**

Exceptions in the context of loading code are printed as error messages.

- **instantiation_error**
  
  PredSpecs not ground.

- **type_error**
  
  PredSpecs not a valid pred_spec_forest.

- **domain_error**
  
  Some arity is an integer < 0.

- **representation_error**
  
  Some arity is an integer > 255.

- **context_error**
  
  Declaration appeared in a goal.

- **permission_error**
  
  Declaration appeared as a clause.

**See Also**

Section 4.3.4.4 [Discontiguous Declarations], page 84.
11.3.66 display/1

Synopsis

display(+Term)

Writes Term on the standard output stream, without quoting atoms, in functional notation, without treating `$VAR'/1 terms specially.

Since quoting is never used, even when needed for reading the term back in, the standard predicate write_canonical/1 is often preferable.

Arguments

Term term

Description
display(Term) is equivalent to:

\[
\text{write_term(Term, [ignore_ops(true)])}
\]

Examples

| ?- display(a+b).
| +(a,b)
yes

| ?- read(X), display(X), nl.
| : a + b * c.
| +(a,*(b,c))

\[X = a+b*c\]

| ?-

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

Section 4.6.4 [ref-iou-tou], page 104.
11.3.67 do/2

Synopsis
(+Iterator do +Body)

Arguments
+Iterator iterator, must be nonvar
:Body callable, must be nonvar

Description
This control structure reduces the need to write auxiliary predicates performing simple iterations. It iterates Body until Iterator’s termination condition is true.

A iterator is a term of one of the following forms:

fromto(First, In, Out, Last)
In the first iteration, In=First. In the n:th iteration, In is the value that Out had at the end of the (n-1):th iteration. In and Out are local variables in Body. The termination condition is Out=Last.

foreach(X,List)
Iterate Body with X ranging over all elements of List. X is a local variable in Body. Can also be used for constructing a list. The termination condition is Tail = [], where Tail is the suffix of List that follows the elements that have been iterated over.

foreacharg(X,Struct)
foreacharg(X,Struct, I)
Iterate Body with X ranging over all arguments of Struct and I ranging over the argument number, 1-based. X and I are local variables in Body. Cannot be used for constructing a term. So the termination condition is true iff all arguments have been iterated over.

count(I, MinExpr, Max)
This is normally used for counting the number of iterations. Let Min take the value integer(MinExpr). Iterate Body with I ranging over integers from Min. I is a local variable in Body. The termination condition is I = Max, i.e. Max can be and typically is a variable.

for(I, MinExpr, MaxExpr)
This is used when the number of iterations is known. Let Min take the value integer(MinExpr), let Max take the value integer(MaxExpr), and let Past take the value max(Min, Max+1). Iterate Body with I ranging over integers from Min to max(Min, Max) inclusive. I is a local variable in Body. The termination condition is I = Past.

param(X) For declaring variables in Body global, i.e. shared with the context. X can be a single variable, or a list of them. The termination condition is true. Please note: By default, variables in Body have local scope.
iterator, iterator

The iterators are iterated synchronously; that is, they all take their first value for the first execution of Body, their second value for the second execution of Body, etc. The order in which they are written does not matter, and the set of local variables in Body is the union of those of the iterators. The termination condition is the conjunction of those of the iterators.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

See Also

Section 4.2.3.5 [ref-sem-ctr-dol], page 68.
11.3.68 dynamic/1

dynamic is a declaration, ISO

Synopsis

:- dynamic +PredSpecs

Declares the clauses of the predicates defined by PredSpecs to be dynamic.

Arguments

:PredSpecs

pred_spec_forest, must be ground

A predicate specification, or a list of such, or a sequence of such separated by commas.

Comments

dynamic is not an ISO predefined prefix operator.

To declare a grammar rule gram/n dynamic, the arity of PredSpecs must be n+2.

Exceptions in the context of loading code are printed as error messages.

Exceptions

Exceptions in the context of loading code are printed as error messages.

instantiation_error

PredSpecs not ground.

type_error

PredSpecs not a valid pred_spec_forest.

domain_error

Some arity is an integer < 0.

representation_error

Some arity is an integer > 255.

context_error

Declaration appeared in a goal.

permission_error

Declaration appeared as a clause.

See Also

Section 4.3.4.2 [Dynamic Declarations], page 84.
11.3.69  enable_breakpoints/1

Synopsis

enable_breakpoints(+BIDs)

Enables the breakpoints specified by BIDs.

Arguments

BIDs  list of integer, must be ground

Breakpoint identifiers.

Exceptions

instantiation_error

in BIDs

type_error

See Also

Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 260, Section 5.7 [Breakpoint Predicates], page 272.
11.3.70 ensure_loaded/1

Synopsis

ensure_loaded(+Files)

Loads the specified Prolog source and/or object file(s) into memory, if not already loaded and up to date.

Arguments

:Files file_spec or list of file_spec, must be ground

A file specification or a list of file specifications; extension optional.

Description

The recommended style is to use this predicate for non-module files only, but if any module files are encountered, then their public predicates are imported.

This predicate is defined as if by:

\[
\text{ensure_loaded}(\text{Files}) :\neg \\
\phantom{\text{ensure_loaded}(\text{Files})} \text{load_files} (\text{Files, [if(changed)]}).
\]

Exceptions

See load_files/[2,3].

See Also

Section 4.3.2 [ref-lod-lod], page 80.
11.3.71 :=/2

Synopsis

+Expr1 := +Expr2

Succeeds if the results of evaluating Expr1 and Expr2 are equal.

Arguments

Expr1 expr, must be ground
Expr2 expr, must be ground

Description

Evaluates Expr1 and Expr2 as arithmetic expressions and compares the results.

Exceptions

Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

Examples

| ?- 1.0 + 1.0 =:= 2.

yes

| ?- "a" =:= 97.

yes

See Also

Section 4.7 [ref-ari], page 119,
11.3.72 erase/1

Synopsis
 erase(+Ref)

Erases from the database the dynamic clause or recorded term referenced by Ref.

Arguments
 Ref db_reference, must be nonvar

Description
Erases from the database the dynamic clause or recorded term referenced by Ref.

Ref must be a database reference to an existing clause or recorded term.

erase/1 is not sensitive to the source module; that is, it can erase a clause even if that clause is neither defined in nor imported into the source module.

Exceptions
 instantiation_error
   If Ref is not instantiated.

type_error
   If Ref is not a database reference.

existence_error
   if Ref is not a database reference to an existing clause or recorded term.

Examples
See Also
Section 4.12.5 [ref-mdb-rcd], page 179.
### 11.3.73 error_exception/1

**Synopsis**

```prolog
:- multifile user:error_exception/1.

user:error_exception(+Exception)
```

Tells the debugger to enter trace mode on exceptions matching `Exception`.

**Arguments**

- `Exception` term

**Exceptions**

None.

**See Also**

Section 5.11 [Exceptions Debug], page 285.
**11.3.74 execution_state/[1,2] development**

**Synopsis**

`execution_state(+Tests)`

*Tests* are satisfied in the current state of the execution.

`execution_state(+FocusConditions, +Tests)`

*Tests* are satisfied in the state of the execution pointed to by *FocusConditions*.

**Arguments**

*FocusConditions*  
`term`

:*Tests*  
`term`

**Exceptions**

`instantiation_error`

An argument not instantiated enough.

`type_error`

An argument not a proper list of callable term.

`domain_error`

An argument not a proper list of valid conditions and tests.

**See Also**

Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 260, Section 5.7 [Breakpoint Predicates], page 272.
11.3.75 ^/2

Synopsis

+X ^ +P

Equivalent to “there exists an X such that P is true”, thus X is normally an unbound variable. The use of the explicit existential quantifier outside setof/3 and bagof/3 is superfluous.

Arguments

X  
:term

:P  
callable, must be nonvar

Description

Equivalent to simply calling P.

Backtracking

Depends on P.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

Examples

Using bagof/3 without and with the existential quantifier:

    | ?- bagof(X, foo(X,Y), L).

    X = _3342,
    Y = 2,
    L = [1,1] ;

    X = _3342,
    Y = 3,
    L = [2] ;

    no

    | ?- bagof(X, Y^foo(X,Y), L).

    X = _3342,
    Y = _3361,
    L = [1,1,2] ;

    no

See Also

setof/3, bagof/3, Section 4.13 [ref-all], page 186.
11.3.76 expand_term/2

Synopsis

\begin{verbatim}
expand_term(+Term1, -Term2)
\end{verbatim}

Transforms source file terms into Prolog clauses before they are compiled. Normally called by the compiler, but can be called directly. The transform can be customized by defining \texttt{user:term_expansion/6}.

When a source file is loaded, except by \texttt{:- include}, \texttt{expand_term/2} is called with the virtual clauses \texttt{beginning_of_file} before and \texttt{end_of_file} after the real Prolog clauses, to give \texttt{user:term_expansion/6} an opportunity to perform some action at the beginning and end of a source file. \textbf{Please note}: the virtual clause \texttt{beginning_of_file} is “seen” before any module declaration, i.e. before the source module has been updated.

Arguments

\begin{verbatim}
Term1    term
Term2    term
\end{verbatim}

Description

Usually called by the built-in predicates that read code and not directly by user programs. In particular used to translate grammar rules, written with \texttt{-->/2}, into ordinary Prolog clauses, written with \texttt{:-/2}. If \texttt{Term1} is a grammar rule, then \texttt{Term2} is the corresponding clause. Otherwise \texttt{Term2} is simply \texttt{Term1} unchanged.

Calls \texttt{user:term_expansion/6}.

Exceptions

Prints messages for exceptions raised by \texttt{user:term_expansion/6}.

See Also

\texttt{phrase/[2,3], -->/2}, Section 4.3.5 [ref-lod-exp], page 87.
11.3.77 fail/0

Synopsis
fail
Always fails.

Exceptions
None.

See Also
Section 4.2 [ref-sem], page 61.
11.3.78 false/0

Synopsis
false
Always fails (same as fail/0).

Exceptions
None.

See Also
Section 4.2 [ref-sem], page 61.
11.3.79 file_search_path/2

Synopsis

`:multifile user:file_search_path/2.`

`user:file_search_path(+PathAlias, +DirSpec)`

Defines a symbolic name for a directory or a path. Used by predicates taking `file_spec` as input argument.

Arguments

- **PathAlias**  
  atom
  An atom that represents the path given by `DirSpec`.

- **DirSpec**  
  file_spec
  Either an atom giving the path to a file or directory, or `PathAlias(DirSpec)`, where `PathAlias` is defined by another `file_search_path/2` rule.

Description

The `file_search_path` mechanism provides an extensible way of specifying a sequence of directories to search to locate a file. For instance, if a filename is given as a structure term, `library(between)`. The principle functor of the term, `library`, is taken to be another `file_search_path/2` definition of the form

```
file_search_path(library, LibPath)
```

and file `between` is assumed to be relative to the path given by `LibPath`. `LibPath` may also be another structure term, in which case another `file_search_path/2` fact gives its definition. The search continues until the path is resolved to an atom.

There may also be several definitions for the same `PathAlias`. Certain predicates, such as `load_files/[1,2]` and `absolute_file_name/[2,3]`, search all these definitions until the path resolves to an existing file.

There are several predefined search paths, such as `application`, `runtime`, `library`, `system`. These are tried before the user-defined ones.

The predicate is undefined at startup, but behaves as if it were a multifile predicate with the following clauses. The system properties `SP_APP_DIR` and `SP_RT_DIR` expand respectively to the absolute path of the directory that contains the executable and the directory that contains the SICStus runtime (see Section 4.17.1 [System Properties and Environment Variables], page 224), `SP_TEMP_DIR` expand to a directory suitable for storing temporary files.
file_search_path(library, Path) :-
    library_directory(Path).
file_search_path(system, Platform) :-
    prolog_flag(host_type, Platform).
file_search_path(application, '$SP_APP_DIR').
file_search_path(runtime, '$SP_RT_DIR').
file_search_path(temp, '$SP_TEMP_DIR').
file_search_path(path, Dir) :-
    ... backtracks through the $PATH environment variable ...

Examples

| ?- [user].
% compiling user...
| :- multifile user:file_search_path/2.
| user:file_search_path(home, '/usr/joe_bob').
| user:file_search_path(review, home('movie/review')).
| end_of_file.
% compiled user in module user, 0 msec 768 bytes
yes
| ?- compile(review(blob)).
% compiling /usr/joe_bob/movie/review/blob.pl

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

absolute_file_name/[2,3], library_directory/1, load_files/[1,2], Section 4.5 [ref-fdi], page 95, Section 4.9.4 [ref-lps-flg], page 136, Section 4.17.1 [System Properties and Environment Variables], page 224.
11.3.80 findall/[3,4]  

Synopsis

\[
\text{findall}(\text{Template}, \text{Generator}, -\text{List})
\]

\[
\text{findall}(\text{Template}, \text{Generator}, -\text{List}, +\text{Remainder})
\]

List is the list of all the instances of Template for which the goal Generator succeeds, appended to Remainder. Remainder defaults to the empty list.

Arguments

Template  term
:Generator  callable, must be nonvar
   A goal to be proved as if by \text{call/1}.

List  list of term

Remainder  list of term

Description

A special case of bagof/3, where all free variables in the generator are taken to be existentially quantified, as if by means of the ’ \sim \ ‘ operator. Contrary to bagof/3 and setof/3, if there are no instances of Template such that Generator succeeds, then List = Remainder.

Because findall/[3,4] avoids the relatively expensive variable analysis done by bagof/3, using findall/[3,4] where appropriate rather than bagof/3 can be considerably more efficient.

Please note: If the instances being gathered contain attributed variables (see Section 10.3 [lib-atts], page 391) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 74), then those variables are replaced by brand new variables, without attributes, in List. To retain the attributes, you can use copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 129).

Backtracking

bagof/3 can succeed nondeterminately, generating alternative values for Set corresponding to different instantiations of the free variables of Generator.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

Examples

To illustrate the differences among findall/3, setof/3, and bagof/3:
% user compiled in module user, 0.100 sec 352 bytes

yes
| ?- bagof(X, foo(X,Y), L).
Y = 2,
L = [1,1] ? ;

Y = 3,

no
| ?- bagof(X, Y^foo(X,Y), L).
L = [1,1,2] ? ;

no
| ?- findall(X, foo(X,Y), L).
L = [1,1,2] ? ;

no
| ?- findall(X, foo(X,Y), L, S).
L = [1,1,2|S] ? ;

no
| ?- setof(X, foo(X,Y), L).
X = _3342,
Y = 2,
L = [1] ;

X = _3342,
Y = 3,
L = [2] ;

no
Comments
findall/3 is part of the ISO Prolog standard; findall/4 is not.

See Also
bagof/3, setof/3, ^/2, Section 4.13 [ref-all], page 186.
11.3.81 float/1

Synopsis

float(+Term)

Term is currently instantiated to a float.

Arguments

Term term

Examples

| ?- float(Term1).  
  
  no
  
  | ?- float(5.2).
  
  yes

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.82 flush_output/[0,1]  ISO

Synopsis

flush_output

flush_output(+Stream)

Forces the buffered output of the stream Stream (defaults to the current output stream) to be sent to the associated device.

Arguments

Stream stream_object, must be ground
   A valid Prolog stream, defaults to the current output stream.

Description

Sends the current buffered output of an output stream Stream to the actual output device, which is usually a disk or a tty device.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
   An error occurred in flushing out the buffered output.

Examples

See Also

Section 4.6.7 [ref-iou-sfh], page 109.
11.3.83 foreign/[2,3]  

Synopsis

:- discontiguous foreign/2, foreign/3.

foreign(+Routine, +ForeignSpec)
foreign(+Routine, +Language, +ForeignSpec)

Describes the interface between Prolog and the foreign Routine. Used by load_foreign_resource/1.

Arguments

Routine  atom, must be nonvar
An atom that names a foreign code Routine.

Language  atom, must be nonvar
An atom that names the Language in which Routine is written. Can only be c.

ForeignSpec
foreign_spec, must be ground
A ground term of the form PredName(Argspec, . . ., Argspec) as described in Section 6.2.3 [Conversions between Prolog Arguments and C Types], page 293. Each Argspec should be a foreign_arg.

Description

The user has to define a foreign/[2,3] fact for every foreign function that is to be called from Prolog. Note that Routine does not have to be the same as PredicateName. Arguments are passed to the foreign function as specified in ForeignSpec.

+type specifies that an argument is to be passed to the foreign function.

-type specifies that an argument is to be received from the foreign function.

[-type] argument is used to obtain the return value of a foreign function call. At most one “return value” argument can be specified.

The foreign/[2,3] facts are used only in the context of a load_foreign_resource/1 command and can be removed once the foreign files are loaded.

Contrary to most hook predicates which reside in the user module, foreign/[2,3] facts will only be looked up in the source module of the loading command.

Exceptions

Error handling is performed by load_foreign_resource/1.

See Also

load_foreign_resource/1, Section 6.2 [Calling C from Prolog], page 291.
11.3.84 foreign_resource/2

Synopsis
:- discontiguous foreign_resource/2.
foreign_resource(+ResourceName, +ForeignFunctions)

Describes the foreign functions in ResourceName to interface to.

Arguments

ResourceName
atom, must be nonvar

ForeignFunctions
list of atom, must be ground
A list of foreign function symbols that will be obtained from ResourceName.

Description

The user has to define a foreign_resource/2 fact for every foreign resource that is to be loaded into Prolog. The ForeignFunctions gives the list of foreign symbols that are to be found in the given foreign resource. When a foreign resource is loaded using load_foreign_resource/1, Prolog looks for a foreign_resource/2 fact for that foreign resource and finds the address of each symbol listed in that fact. Prolog also expects a foreign/[2,3] definition for each symbol in the second argument of that fact.

The foreign_resource/2 facts are used only in the context of a load_foreign_resource/1 command and can be removed once the foreign resource has been loaded.

Contrary to most hook predicates which reside in the user module, load_foreign_resource/1 will look for foreign_resource/2 facts defined in its source module.

Exceptions

Error handling is performed by load_foreign_resource/1.

See Also
load_foreign_resource/1, foreign/[2,3], Section 6.2 [Calling C from Prolog], page 291.
11.3.85 format/[2,3]

Synopsis

format(+Control, +Arguments)

format(+Stream, +Control, +Arguments)

Interprets the Arguments according to the Control string and prints the result on Stream.

Arguments

Stream stream_object, must be ground
   Defaults to the current output stream.

Control chars or codes or atom, must be ground
   A string, which can contain control sequences of the form ‘~N<c>’:
   <c> a format control option
   N optional; if given, must be ‘*’ or an integer.
   Any characters that are not part of a control sequence are written to the specified output stream.

:Arguments
   list of term, must be proper list
   List of arguments, which will be interpreted and possibly printed by format control options.

Description

If a parameter N can be specified, then it can be either an integer, specified as an optional minus sign followed by a sequence of decimal digits, or the character ‘*’.

If the parameter is specified as ‘*’, then the value used will be the truncated integer value of the next element from Arguments interpreted as a numerical expression.

The following control options cause formatted printing of the next element from Arguments to the current output stream.

‘~a’ The argument is an atom. The atom is printed without quoting.

‘~Nc’ (Print character.) The argument is a number that will be interpreted as a code. N defaults to one and is interpreted as the number of times to print the character. If N is zero, or negative, then the character is not printed.

‘~Ne’ ‘~NE’ (Print float in exponential notation.) The argument is a float, which will be printed in exponential notation with one digit before the decimal point and N digits after it. If N is zero, or negative, then one digit appears after the decimal point. A sign and at least two digits appear in the exponent, which is introduced by the letter used in the control sequence. N defaults to 6.
   The magnitude of N must be less than 100000, but useful values are much smaller than that.
~~NF~~ (Print float in fixed-point notation.) The argument is a float, which will be printed in fixed-point notation with $N$ digits after the decimal point. $N$ may be zero, or negative, in which case a single zero appears after the decimal point. At least one digit appears before the decimal point and at least one after it. $N$ defaults to 6.

The magnitude of $N$ must be less than 100000, but useful values are much smaller than that.

~~Ng~~ (Print float in generic notation.) The argument is a float, which will be printed in ‘f’ or ‘e’ (or ‘E’ if ‘G’ is used) notation with $N$ significant digits. If $N$ is zero, then one significant digit is printed. ‘E’ notation is used if the exponent from its conversion is less than -4 or greater than or equal to $N$, otherwise ‘f’ notation. Trailing zeroes are removed from the fractional part of the result. A decimal point and at least one digit after it always appear. $N$ defaults to 6.

The magnitude of $N$ must be less than 100000, but useful values are much smaller than that.

~~Nh~~ (Print float precisely.) The argument is a float, which will be printed in ‘f’ or ‘e’ (or ‘E’ if ‘H’ is used) notation with $d$ significant digits, where $d$ is the smallest number of digits that will yield the same float when read in. ‘E’ notation is used if $N<0$ or if the exponent is less than $-N-1$ or greater than or equal to $N+d$, otherwise ‘f’ notation. $N$ defaults to 3.

The intuition is that for numbers like 123000000.0, at most $N$ consecutive zeroes before the decimal point are allowed in ‘f’ notation. Similarly for numbers like 0.000000123.

‘E’ notation is forced by using ‘~1H’. ‘F’ is forced by using ‘~999H’.

The magnitude of $N$ must be less than 100000, but useful values are much smaller than that.

~~Nd~~ (Print decimal.) The argument is an integer. $N$ is interpreted as the number of digits after the decimal point. If $N$ is 0 or missing, then no decimal point will be printed.

$N$ must be non-negative.

~~ND~~ (Print decimal.) The argument is an integer. Identical to ‘~Nd’ except that ‘,’ will separate groups of three digits to the left of the decimal point.

~~Nr~~ (Print radix.) The argument is an integer. $N$ is interpreted as a radix, $2 \leq N \leq 36$. If $N$ is missing, then the radix defaults to 8. The letters ‘a–z’ will denote digits larger than 9.

~~NR~~ (Print radix.) The argument is an integer. Identical to ‘~Nr’ except that the letters ‘A–Z’ will denote digits larger than 9.

~~Ns~~ (Print string.) The argument is a code list. If $N$ is zero, or negative, then no characters are output.
If \( N \) is positive, then the first \( N \) characters of the code list will be written, and
if the code list is exhausted, then extra \( \text{SPC} \) characters will be written, for a
total of \( N \) characters output.

`~i` (Ignore.) The argument, which may be of any type, is ignored.

`~k` (Print canonical.) The argument may be of any type. The argument will be
passed to `write_canonical/1` (see Section 4.6.4 [ref-iou-tou], page 104).

`~p` (Print.) The argument may be of any type. The argument will be passed to
`print/1` (see Section 4.6.4 [ref-iou-tou], page 104).

`~q` (Print quoted.) The argument may be of any type. The argument will be
passed to `writeq/1` (see Section 4.6.4 [ref-iou-tou], page 104).

`~w` (Write.) The argument may be of any type. The argument will be passed to
`write/1` (see Section 4.6.4 [ref-iou-tou], page 104).

`~O` (Call.) The argument \( \text{Arg} \) is a goal, which will be called as if by \(+ \ \text{Arg} +\) and
is expected to print on the current output stream. If the goal performs other
side effects, then the behavior is undefined.

`~` (Print tilde.) Takes no argument. Prints `~`.

`~N` (Print newline.) Takes no argument. Prints \( N \) newlines. \( N \) defaults to 1.
If \( N \) is negative or zero, then no newlines are output.

The following control sequences set column boundaries and specify padding. A column is
defined as the available space between two consecutive column boundaries on the same line.
A boundary is initially assumed at line position 0. The specifications only apply to the line
currently being written.

When a column boundary is set (`~|` or `~+`) and there are fewer characters written in the
column than its specified width, the remaining space is divided equally amongst the pad
sequences (`~t`) in the column. If there are no pad sequences, then the column is space
padded at the end.

If `~` specifies a position preceding the current position, then the boundary is set
at the current position.

`~N` Set a column boundary at line position \( N \). \( N \) defaults to the current position.
The control sequence `~N` (with explicit \( N \)) will not work correctly on bi-
directional streams\(^1\), e.g. those created by `library(sockets)`. A workaround
for bi-directional streams may be to use `~|` (without explicit \( N \)) and `~N+`, see
below.

`~N+` Set a column boundary at \( N \) positions past the previous column boundary. \( N \)
defaults to 8.

---

\(^1\) This is because streams only have one counter for line position, and for bidirectional streams that counter
tracks the input direction (see Section 12.3.40 [cpg-ref-SP_get_stream_counts], page 1310).
Specify padding in a column. \texttt{N} is the fill character code. \texttt{N} may also be specified as \texttt{C} where \texttt{C} is the fill character. The default fill character is \texttt{SPC}. Any (\texttt{~t}) after the last column boundary on a line is ignored.

**Exceptions**

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

- \texttt{consistency_error}
  - Wrong number of Arguments.
- \texttt{type_error}
- \texttt{domain_error}
  - Argument of the wrong type or domain.

**Examples**

```
| ?- Pi=3.14159265, format(\texttt{~e~2E~0E\n'}, [Pi,Pi,Pi]).
  3.141593e+00 3.14E+00 3.0E+00

| ?- Pi=3.14159265, format(\texttt{~f~2F~0F\n'}, [Pi,Pi,Pi]).
  3.141593, 3.14, 3.0

| ?- format(\texttt{~g~2G~0G\n'}, [1.23456789e+10, 3.14159265, 0.0123]).
  1.23457e+10 3.1 0.01

| ?- F = 123000.0, G = 0.000123, 
  format(\texttt{~h~h~2h~2H~iH\n'}, [F,G,F,G,3.14]).
  123000.0 0.000123 1.23e+05 1.23E-04 3.14E+00

| ?- format(\texttt{Hello~1d world!\n'}, [42]).
  Hello 4.2 world!

| ?- format(\texttt{Hello~d world!\n'}, [42]).
  Hello 42 world!

| ?- format(\texttt{Hello~1D world!\n'}, [12345]).
  Hello 1,234.5 world!

| ?- format(\texttt{Hello~2r world!\n'}, [15]).
  Hello 1111 world!

| ?- format(\texttt{Hello~16r world!\n'}, [15]).
  Hello f world!

| ?- format(\texttt{Hello~16R world!\n'}, [15]).
  Hello F world!
```
| ?- format('Hello `4s `4s!
', ["new","world"]). Hello new world! |
| ?- format('Hello `s world!
', ["new"]). Hello new world! |
| ?- format('Hello `i`s world!
', ["old","new"]). Hello new world! |
| ?- format('Hello `k world!
', [[a,b,c]]). Hello '.(a,'.'(b,'.'(c,[])) world! |
| ?- assert((portray([X|Y]) :- print(cons(X,Y)))). |
| ?- format('Hello `p world!
', [[a,b,c]]). Hello cons(a,cons(b,cons(c,[])) world! |
| ?- format('Hello `q world!
', [['A','B']]). Hello ['A','B'] world! |
| ?- format('Hello `w world!
', [['A','B']]). Hello [A,B] world! |
| ?- format('Hello `@ world!
', [write(new)]). Hello new world! |
| ?- format('Hello `~ world!
', []). Hello ~ world! |
| ?- format('Hello `n world!
', []). Hello world! |
| ?- format("^t NICE TABLE ^t61^n", []). format("^t61^n", []). format("^t-a^20|t-a^20+a-t^20+t^61^n", [Right aligned,'Centered','Left aligned'], format("^t-d^20|t-d^20+d-t^20+t^61^n", [123,45,678]), format("^t-d^20|t-d^20+d-t^20+t^61^n", [1,2345,6789]), format("^t^61^n", []). |

********************************************************* NICE TABLE *********************************************************

* Right aligned Centered Left aligned *
* 123 45 678 *
* 1 2345 6789 *

********************************************************* NICE TABLE *********************************************************
Chapter 11: Prolog Reference Pages 1037

| 1-1 Documentation supplement for Quintus Prolog Release 1.5 ........... 2
| 1-2 Finding all solutions ......................................... 3
| 1-3 Searching for a file in a library ............................. 4
| 1-4 New Built-in Predicates ....................................... 5
| 1-4-1 write_canonical (?Term) ....................................... 5
| 1-7 File Specifications .......................................... 17
| 1-7-1 multifile(+PredSpec) ...................................... 18

See Also
Section 4.6.4 [ref-iou-tou], page 104.
11.3.86 freeze/2

Synopsis
freeze(+Flag, +Goal)

Blocks Goal until Flag is bound.

Arguments
Flag : term
:Goal : callable, must be nonvar

Description
Defined as if by:

\[
\text{freeze}(X, \text{Goal}) :- \text{when} \left( \text{nonvar}(X) \right), \text{Goal}.
\]

or

\[
\text{:- block freeze}(-, ?).
\]

\[
\text{freeze}(-, \text{Goal}) :- \text{Goal}.
\]

Backtracking
Depends on Goal.

Exceptions
Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

See Also
Section 4.2.4 [ref-sem-sec], page 74.
11.3.87 frozen/2

Synopsis

frozen(+Term, -Goal)

Goal is unified with the conjunction of all goals blocked on some variable that occurs in Term.

Arguments

Term term
Goal callable

Description

If no variable in Term blocks any goal or has attributes that can be interpreted as a goal (see Section 10.3 [lib-atts], page 391), then Goal is unified with the atom true. Otherwise, Goal is unified with the conjunction of all such goals.

Exceptions

None.

See Also

Section 4.2.4 [ref-sem-sec], page 74.
**11.3.88 functor/3**

**Synopsis**

functor(+Term, -Name, -Arity)

functor(-Term, +Name, +Arity)

Succeeds if the principal functor of term Term has name Name and arity Arity.

**Arguments**

Term term

Name atom

Arity arity

**Description**

There are two ways of using this predicate:

1. If Term is initially instantiated, then
   - if Term is a compound term, then Name and Arity are unified with the name and arity of its principal functor.
   - otherwise, Name is unified with Term, and Arity is unified with 0.
2. If Term is initially uninstantiated, then Name and Arity must both be instantiated, and
   - if Arity is an integer in the range 1..255, then Name must be an atom, and Term becomes instantiated to the most general term having the specified Name and Arity; that is, a term with distinct variables for all of its arguments.
   - if Arity is 0, then Name must be atomic, and it is unified with Term.

**Exceptions**

instantiation_error

Term and either Name or Arity are uninstantiated.

type_error

Name is not atomic, or Arity is not an integer, or Name is not an atom when Arity > 0.

domain_error

Arity is an integer < 0.

representation_error

Term is uninstantiated and Arity > 255.
Examples

?- functor(foo(a,b), N, A).

N = foo,
A = 2

?- functor(X, foo, 2).

X = foo(_A, _B)

?- functor(X, 2, 0).

X = 2

See Also

arg/3, name/2, =../2, Section 4.8.2 [ref-lte-act], page 127.
11.3.89 garbage_collect/0

Synopsis
garbage_collect

Invokes the garbage collector.

Description
This predicate invokes the garbage collector to reclaim data structures on the Prolog stack that are no longer accessible to the computation.

Examples
In the code fragment:

\[ \text{cycle}(X) :- \text{big_goal}(X, X1), \text{cycle}(X1). \]

if \text{cycle} \((X)\) is to run for a long time, and if \text{big_goal} \((X, X1)\) generates a lot of garbage, then rewrite the code like this:

\[ \text{cycle}(X) :- \text{big_goal}(X, X1), !, \text{garbage_collect}, \text{cycle}(X1). \]

Tips
Use of the ‘!, garbage_collect’ idiom is only desirable when you notice that your code does frequent garbage collections. It will allow the garbage collector to collect garbage more effectively, and the cycle will run without demanding increasing amounts of memory.

Exceptions
None.

See Also
Section 4.10 [ref-mgc], page 144.
**11.3.90 garbage_collect_atoms/0**

**Synopsis**

`garbage_collect_atoms`

Invokes the atom garbage collector.

**Description**

This predicate invokes the atom garbage collector to discard atoms that are no longer accessible to the computation, reclaiming their space.

**Tips**

A program can use the `atoms` keyword to `statistics/2` to determine if a call to `garbage_collect_atoms/0` would be appropriate.

**Exceptions**

None.

**See Also**

Section 4.10 [ref-mgc], page 144.
11.3.91  generate_message/3

Synopsis

:- multifile 'SU_messages':generate_message/3.
'SU_messages':generate_message(+MessageTerm, -S0, -S)

For a given MessageTerm, generates a list composed of Control-Arg pairs and the atom nl. This can be translated into a nested list of Control-Arg pairs, which can be used as input to print_message_lines/3.

Arguments

MessageTerm
term
  May be any term.

S0  list of pair
  The resulting list of Control-Args pairs.

S   list of pair
  The remaining list.

Description

Clauses for 'SU_messages':generate_message/3 underlie all messages from Prolog. They may be examined and altered. They are found in library('SU_messages').

The purpose of this predicate is to allow you to redefine how Prolog’s messages are displayed. For example, to translate all the messages from English into some other language.

This predicate should not be modified if all you want to do is modify or add a few messages: user:generate_message_hook/3 is provided for that purpose.

The Prolog system uses the built-in predicate print_message/2 to print all its messages. When print_message/2 is called, it calls user:generate_message_hook(Message,L,[]) to generate the message. If that fails, then 'SU_messages':generate_message(Message,L,[]) is called instead. If that succeeds, then L is assumed to have been bound to a list whose elements are either Control-Args pairs or the atom nl. Each Control-Arg pair should be such that the call

  format(user_error, Control, Args)

is valid. The atom nl is used for breaking the message into lines. Using the format specification ‘~n’ (new-line) is discouraged, since the routine that actually prints the message (see user:message_hook/3 and print_message_lines/3) may need to have control over newlines.

'SU_messages':generate_message/3 is not included by default in runtime systems, since end-users of application programs should probably not be seeing any messages from the Prolog system.
If there is a call to `print_message/2` when when `SU_messages':generate_message/3` does not succeed for some reason, then the message term itself is printed, for example:

```
| ?- print_message(error,unexpected_error(37)).
! unexpected_error(37)
```

'SU_messages':generate_message/3 failed because the message term was not recognized. In the following example `print_message/2` is being called by the default exception handler:

```
| ?- write(A,B).
! Instantiation error in argument 1 of write/2
! goal: write(_2107,_2108)
```

**Examples**

```prolog
:- multifile user:generate_message_hook/3.
user:generate_message_hook(hello_world) -->
    ['hello world'-[],nl].
```

Note that the terminating `nl` is required.

**Exceptions**

`print_message/2` checks that the generated list is a valid parse.

**See Also**

Section 4.16 [ref-msg], page 212.
11.3.92 generate_message_hook/3

Synopsis
:- multifile user:generate_message_hook/3.

user:generate_message_hook(+MessageTerm, -S0, -S)

A way for the user to override the call to 'SU_messages':generate_message/3 in print_message/2.

Arguments
MessageTerm
term
    May be any term.

S0 list of pair
    The resulting list of Control-Args pairs.

S list of pair
    The remaining list.

Description
For a given MessageTerm, generates the list of Control-Args pairs required for print_message_lines/3 to format the message for display.

This is the same as 'SU_messages':generate_message/3 except that it is a hook. It is intended to be used when you want to override particular messages from the Prolog system, or when you want to add some messages. If you are using your own exception classes (see raise_exception/1), then it may be useful to provide generate_message_hook clauses for those exceptions so that the print_message/2 (and thus the default exception handler that calls print_message/2) can print them out nicely.

The Prolog system uses the built-in predicate print_message/2 to print all its messages. When print_message/2 is called, it calls user:generate_message_hook(Message,L,[]) to generate the message. If that fails, then 'SU_messages':generate_message(Message,L,[]) is called instead. If that succeeds, then L is assumed to have been bound to a list whose elements are either Control-Args pairs or the atom nl. Each Control-Arg pair should be such that the call

    format(user_error, Control, Args)

is valid. The atom nl is used for breaking the message into lines. Using the format specification ‘\n’ (new-line) is discouraged, since the routine that actually prints the message (see user:message_hook/3 and print_message_lines/3) may need to have control over newlines.
Examples

```prolog
:- multifile user:generate_message_hook/3.
user:generate_message_hook(hello_world) -->
    ['hello world'-[],nl].
```

Note that the terminating `nl` is required.

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

Section 4.16 [ref-msg], page 212.
11.3.93  get_byte/[1,2]  ISO

Synopsis
get_byte(-Byte)
get_byte(+Stream, -Byte)

Unifies Byte with the next byte from Stream or with -1 if there are no more bytes.

Arguments
Stream  stream_object, must be ground
        valid input binary stream, defaults to the current input stream.
Byte    byte or -1

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

type_error
        Byte is an invalid byte.

permission_error
        Trying to read beyond end of Stream

See Also
Section 4.6.5 [ref-iou-cin], page 107.
11.3.94 get_char/[1,2]  

Synopsis

get_char(-Char)

get_char(+Stream, -Char)

Unifies Char with the next char from Stream or with end_of_file if there are no more characters.

Arguments

Stream stream_object, must be ground.
Valid input text stream, defaults to the current input stream.

Char char or one of [end_of_file]

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
   Trying to read beyond end of Stream

See Also

Section 4.6.5 [ref-iou-cin], page 107.
11.3.95 get_code/[1,2]  

**Synopsis**

get_code(-Code)

get_code(+Stream, -Code)

Unifies Code with the next code from Stream or with -1 if there are no more characters.

**Arguments**

*Stream* stream_object, must be ground

Valid input text stream, defaults to the current input stream.

*Code* code or -1

**Exceptions**

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

**permission_error**

Trying to read beyond end of Stream

**See Also**

Section 4.6.5 [ref-iou-cin], page 107.
11.3.96 get_mutable/2

Synopsis

get_mutable(-Datum, +Mutable)

*Datum* is the current value of the mutable term *Mutable*.

Arguments

*Datum* term, must be nonvar

*Mutable* mutable, must be nonvar

Exceptions

instantiation_error

Mutable is uninstantiated.

type_error

Mutable is not a mutable.

See Also

Section 4.8.9 [ref-lte-mut], page 131.
11.3.97 goal_expansion/5

Synopsis

\[ M:goal\_expansion(+Goal1, +Layout1, +Module, -Goal2, -Layout2) \]

Defines transformations on goals while clauses are being compiled or asserted, and during meta-calls at runtime.

Arguments

- **Goal1** \( \text{callable} \)
  Goal to transform.

- **Layout1** \( \text{term} \)
  Layout of goal to transform.

- **Module** \( \text{atom} \)
  Source module of goal to transform.

- **Goal2** \( \text{callable} \)
  Transformed goal.

- **Layout2** \( \text{term} \)
  Layout of transformed goal.

Description

Defines transformations on goals while clauses are being consulted, compiled or asserted, \textit{after} any processing by \texttt{user:term_expansion/6} of the terms being read in. It is called for every simple \texttt{Goal1}, defined in \( M \), in the source module \texttt{Module} found while traversing the clause bodies. Typically, \texttt{Module} has imported the predicate \texttt{Goal1} from module \( M \) but it happens also if \texttt{Module} uses an explicit module prefix, i.e. \( M:Goal1 \).

If it succeeds, then \texttt{Goal1} is replaced by \texttt{Goal2}; otherwise, \( Goal1 = Goal2 \). \texttt{Goal2} may be an arbitrarily complex goal, and \( M:goal\_expansion/5 \) is recursively applied to the expansion and its subgoals.

\textbf{Please note}: the arguments of meta-predicates such as \texttt{call/1}, \texttt{setof/3} and \texttt{on_exception/3} are \textit{not} subject to such compile-time processing. Instead the expansion is performed at runtime, see below.

The above description holds even if \texttt{Goal1} is exported but not defined in the module \( M \), i.e. it is possible to define a goal expansion for an exported, but otherwise undefined, predicate. However, in general, it is better to provide an ordinary predicate definition as a fallback, e.g. to be able to handle meta-calls if the goal expansion is not defined at runtime.

This predicate is also used to resolve any meta-calls to \texttt{Goal1} at runtime via the same mechanism. If the transformation succeeds, then \texttt{Goal2} is simply called instead of \texttt{Goal1}. Otherwise, if \texttt{Goal1} is a goal of an existing predicate, then that predicate is invoked. Otherwise, error recovery is attempted by \texttt{user:unknown\_predicate\_handler/3}. 
**M:goal_expansion/5** can be regarded as a macro expansion facility. It is used for this purpose to support the interface to attributed variables in `library(atts)`, which defines the predicates `M:get_atts/2` and `M:put_atts/2` to access module-specific variable attributes. These “predicates” are actually implemented via the `M:goal_expansion/5` mechanism. This has the effect that calls to the interface predicates are expanded at compile time to efficient code.

For accessing aspects of the load context, e.g. the name of the file being compiled, the predicate `prolog_load_context/2` (see Section 4.9.5 [ref-lps-lco], page 143) can be used. Note that `prolog_load_context/2` only gives meaningful results during compile (or consult) time. This means that when a meta call is goal expanded, at runtime, the load context will not be available, and there is no reliable way for a goal expansion to distinguish between these cases.

The goal expansion may happen both at compile time (the normal case) at runtime (for meta calls). In some cases the compiler may try to avoid meta calls by calling goal expansion also for meta calls. This all means that the code implementing goal expansion should be present at both compile time and runtime. It also implies that goal expansion should not misbehave if it is called more times than expected.

`Layout1` and `Layout2` are for supporting source-linked debugging in the context of goal expansion. The predicate should construct a suitable `Layout2` compatible with `Term2` that contains the line number information from `Layout1`. If source-linked debugging of `Term2` is not important, then `Layout2` should be `[]`. The recording of source info is affected by the `source_info` prolog flag (see Section 4.9.4 [ref-lps-flg], page 136).

**Exceptions**

Exceptions are treated as failures, except an error message is printed as well.

**See Also**

Section 4.3.5 [ref-lod-exp], page 87, Chapter 2 [Glossary], page 7.
11.3.98 goal_source_info/3

Synopsis

goal_source_info(+AGoal, -Goal, -SourceInfo)

Decompose the AGoal annotated goal into a Goal proper and the SourceInfo descriptor term, indicating the source position of the goal.

Arguments

AGoal callable, must be nonvar
Goal callable
SourceInfo term

Description

Annotated goals occur in most of error message terms, and carry information on the Goal causing the error and its source position. The SourceInfo term, retrieved by goal_source_info/3 will be one of the following:

[] The goal has no source information associated with it.
fileref(File,Line) The goal occurs in file File, line Line.
clauseref(File,MFunc,ClauseNo,CallNo,Line) The goal occurs in file File, within predicate MFunc, clause number ClauseNo, call number CallNo and virtual line number Line. Here, MFunc is of form Module:Name/Arity, calls are numbered textually and the virtual line number shows the position of the goal within the listing of the predicate MFunc, as produced by listing/1. Such a term is returned for goals occurring in interpreted predicates, which do not have “real” line number information, e.g. because they were entered from the terminal, or created dynamically.

Exceptions

instantiation_error Goal is uninstantiated
type_error Goal is not a callable

See Also

Section 4.16 [ref-msg], page 212.
11.3.99 \( \textit{>}/2 \)

**Synopsis**

\(+Expr1 > +Expr2\)

Succeeds if the result of evaluating \(Expr1\) is strictly \textit{greater than}\ the result of evaluating \(Expr2\).

**Arguments**

- \(Expr1\) \(\text{expr}\), must be ground
- \(Expr2\) \(\text{expr}\), must be ground

**Description**

Evaluates \(Expr1\) and \(Expr2\) as arithmetic expressions and compares the results.

**Exceptions**

Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

**Examples**

\[
| \text{?- "g" > "g".} \\
| \text{no} \\
| \text{?- 4*2 > 15/2.} \\
| \text{yes}
\]

**See Also**

Section 4.7 [ref-ari], page 119,
\textbf{11.3.100 ground/1} \hspace{1cm} \textit{ISO}

\textbf{Synopsis}

\texttt{ground(+Term)}

\textit{Term} is currently instantiated to a ground term.

\textbf{Arguments}

\texttt{Term \hspace{1cm} term}

\textbf{Description}

Tests whether \textit{X} is completely instantiated, i.e. free of unbound variables. In this context, mutable terms are treated as nonground, so as to make \texttt{ground/1} a monotone predicate.

\textbf{Examples}

\begin{verbatim}
| ?- ground(9).
  yes
| ?- ground(major(tom)).
  yes
| ?- ground(a(1,Term,3)).
  no
| ?- ground("a").
  yes
| ?- ground([1,foo(Term)]).
  no
\end{verbatim}

\textbf{Exceptions}

None.

\textbf{See Also}

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.101 halt/[0,1]  ISO

Synopsis

halt

halt(+ExitCode)

Causes an exit from the running process.

Arguments

ExitCode  integer, must be nonvar

Exit status code. Only the lower 8 bits of this value is used.

Description

Causes an exit from the running process with exit code ExitCode. ExitCode defaults to zero which, by convention, signifies a successful exit from the process.

halt/[0,1] is implemented by raising a reserved exception, which is handled at the top level; see Section 4.15.7 [ref-ere-int], page 211.

Exceptions

instantiation_error

ExitCode is uninstantiated.

type_error

ExitCode is not an integer.

See Also

abort/0, break/0, runtime_entry/1, Section 4.15.7 [ref-ere-int], page 211.
11.3.102 if/3

Synopsis
if(+P,+Q,+R)

If P then Q else R, for all solution of P.

Arguments
: P callable, must be nonvar
: Q callable, must be nonvar
: R callable, must be nonvar

Description
Analogous to
if P then Q else R

but differs from P -> Q ; R in that if(P, Q, R) explores all solutions to the goal P. There is a small time penalty for this—if P is known to have only one solution of interest, then the form P -> Q ; R should be preferred.

This is normally regarded as part of the syntax of the language, but it is like a built-in predicate in that you can write call(if(P,Q,R)), with identical effect if it does not contain any cuts.

Cuts in P do not make sense, but are allowed, their scope being the goal P. The scope of cuts in Q and R extends to the containing clause.

Backtracking
Depends on the arguments.

Exceptions
Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

See Also
Section 4.2 [ref-sem], page 61.
11.3.103  

**Synopsis**

\(+P \rightarrow +Q\)

When occurring other than as the first argument of a disjunction operator \(;/2\), this is equivalent to:

\[ P \rightarrow Q ; \text{fail}. \]

**Arguments**

: \(P\)  
callable, must be nonvar

: \(Q\)  
callable, must be nonvar

**Description**

This is not normally regarded as a built-in predicate, since it is part of the syntax of the language. However, it is like a built-in predicate in that you can say `call((P \rightarrow Q))`, with identical effect if it does not contain any cuts.

`'\rightarrow'` cuts away any choice points in the execution of \(P\)

Note that the operator precedence of `'\rightarrow'` is greater than 1000, so it dominates commas. Thus, in:

\[
\begin{align*}
  f & :- p, q \rightarrow r, s. \\
  f.
\end{align*}
\]

`'\rightarrow'` cuts away any choices in \(p\) or in \(q\), but unlike cut (!) it does not cut away the alternative choice for \(f\).

Cuts in \(P\) do not make sense, but are allowed, their scope being the goal \(P\). The scope of cuts in \(Q\) extends to the containing clause.

**Backtracking**

Depends on \(Q\).

**Exceptions**

Call errors (see Section 4.2.6 [ref-sem-exec], page 77).

**See Also**

Section 4.2 [ref-sem], page 61.
11.3.104 include/1

declaration, ISO

Synopsis

:- include +Files

Literally embed the Prolog clauses and directives in Files into the file being loaded. The file or files will be opened with default options.

Arguments

:Files file_spec or list of file_spec, must be ground

A file specification or a list of file specifications; extension optional.

Description

The effect is such as if the declaration itself was replaced by the text in the Files. Including some files is thus different from loading them in several respects:

- The embedding file counts as the source file of the predicates loaded, e.g. with respect to the built-in predicate source_file/2; see Section 4.9.3 [ref-lps-apf], page 136.
- Some clauses of a predicate can come from the embedding file, and some from included files.
- When including a file twice, all the clauses in it will be entered twice into the program (although this is not very meaningful).

Comments

include is not an ISO predefined prefix operator.

Exceptions

Exceptions in the context of loading code are printed as error messages. See also load_files/[2,3].

context_error

Declaration appeared in a goal.

permission_error

Declaration appeared as a clause.

See Also

Section 4.3.4.11 [Include Declarations], page 86.
11.3.105 initialization/1

**Declaration, ISO**

**Synopsis**

```prolog
:- initialization +Goal
```

Declares that \textit{Goal} is to be run when the file in which the declaration appears is loaded into a running system, or when a stand-alone program or runtime system that contains the file is started up.

**Arguments**

:Goal callable, must be nonvar

**Description**

Callable at any point during loading of a file. That is, it can be used as a directive, or as part of a goal called at load time. The initialization goal will be run as soon as the loading of the file is completed. That is at the end of the load, and notably after all other directives appearing in the file have been run.

\texttt{save_program/[1,2]} saves initialization goals in the saved state, so that they will run when the saved state is restored. When they run, they have access to the load context (\texttt{prolog_load_context/2}), just like other goals appearing in directives.

\textit{Goal} is associated with the file loaded and a module. When a file, or module, is going to be reloaded, all goals earlier installed by that file or in that module, are removed. This is done before the actual load, thus allowing a new initialization \textit{Goal} to be specified, without creating duplicates.

**Comments**

\texttt{initialization} is not an ISO predefined prefix operator.

**Exceptions**

- \texttt{instantiation_error}
  - The argument \textit{Goal} is not instantiated.

- \texttt{context_error}
  - Initialization appeared in a goal.

- \texttt{permission_error}
  - Initialization appeared as a clause.

**See Also**

Section 4.3.4.12 [Initializations], page 87.
11.3.106 instance/2

Synopsis

instance(+Ref, -Term)

Unifies Term with the most general instance of the dynamic clause or recorded term indicated by the database reference Ref.

Arguments

Ref      db_reference, must be nonvar
Term     term

Description

Ref must be instantiated to a database reference to an existing clause or recorded term. instance/2 is not sensitive to the source module and can be used to access any clause, regardless of its module.

Exceptions

instantiation_error
  if Ref is not instantiated

type_error
  if Ref is not a syntactically valid database reference

existence_error
  if Ref is a syntactically valid database reference but does not refer to an existing clause or recorded term.

Examples

| ?- assert(foo:bar,R).  
  | R = '$ref'(771292,1)
  | ?- instance('$ref'(771292,1),T).
  | T = (bar:-true)
  | ?- clause(H,B,'$ref'(771292,1)).
  | no
  | ?- clause(foo:H,B,'$ref'(771292,1)).

H = bar,
B = true

See Also

Section 4.12.6 [ref-mdb-acl], page 181.
11.3.107 integer/1

Synopsis

integer(+Term)

Term is currently instantiated to an integer.

Arguments

Term term

Examples

| ?- integer(5).
  yes |
| ?- integer(5.0).
  no |

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.108 is/2

Synopsis

- Term is + Expression

Evaluates Expression as an arithmetic expression, and unifies the resulting number with Term.

Arguments

Expression

expr, must be ground.

An expression made up of:

- functors representing arithmetic operations
- numbers
- variables bound to numbers or arithmetic expressions

Term

number

Exceptions

Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

Examples

| ?- X is 2 * 3 + 4. |

X = 10

| ?- Y = 32.1, X is Y * Y. |

X = 1030.41
Y = 32.1

| ?- Arity is 3 * 8, X is 4 + Arity + (3 * Arity * Arity). |

Arity = 24
X = 1756

| ?- X is 6/0. |

! Domain error in argument 2 of is/2
! expected an integer not equal to 0, but found 0
! goal: _98 is 6/0
| ?- X is "a".
|     X = 97
| ?- X is 4 * 5, Y is X * 4.
|     X = 20,
|     Y = 80

Comments
If a variable in an arithmetic expression is bound to another arithmetic expression (as opposed to a number) at runtime, then the cost of evaluating that expression is much greater. It is approximately equal to the cost of call/1 of an arithmetic goal.

See Also
Section 4.7 [ref-ari], page 119.
11.3.109  keysort/2

Synopsis

keysort(+Pairs, -Sorted)

Sorts the elements of the list Pairs into ascending standard order (see Section 4.8.8.2 [ref-lte-cte-sot], page 130) with respect to the key of the pair structure.

Arguments

Pairs    list of pair, must be a proper list of proper pairs
Sorted   list of pair

Description

The list Pairs must consist of terms of the form Key-Value. Multiple occurrences of pairs with the same key are not removed.

(The time taken to do this is at worst order ($N \log N$) where $N$ is the length of the list.)

Note that the elements of Pairs are sorted only according to the value of Key, not according to the value of Value.

keysort/2 is stable in the sense that the relative position of elements with the same key is maintained.

Sorted is type checked since release 4.3 for alignment with the ISO Prolog standard. Previous releases simply failed instead of reporting an error for malformed Sorted.

Exceptions

instantiation_error
   If Pairs is not properly instantiated

type_error
   If Pairs is not a proper list of pair.

type_error
   If Sorted cannot be unified with a list of pair.

Examples

| ?- keysort([3-a,1-b,2-c,1-a,1-b], X).

X = [1-b,1-a,1-b,2-c,3-a]

|?- keysort([2-1, 1-2], [1-2, 2-1]).

yes

See Also

Section 4.8.8.3 [ref-lte-cte-sor], page 131.
11.3.110 leash/1  

**Synopsis**

leash(+Mode)

Starts leashing on the ports given by Mode.

**Arguments**

Mode  

list of one of [call, exit, redo, fail, exception, all, half, loose, tight, off], must be ground

A list of the ports to be leashed. A single keyword can be given without enclosing it in a list.

**Description**

Some of the keywords denote a set of ports:

all  

Stands for all five port.

half  

Stands for the Exception, Call and Redo ports.

loose  

Stands for the Exception and Call ports.

tight  

Stands for all ports but Exit.

off  

Stands for no ports.

The leashing mode only applies to procedures that do not have spypoints on them, and it determines which ports of such procedures are leashed. By default, all five ports are leashed. On arrival at a leashed port, the debugger will stop to allow you to look at the execution state and decide what to do next. At unleashed ports, the goal is displayed but program execution does not stop to allow user interaction.

**Exceptions**

instantiation_error  

Mode is not ground

domain_error  

Mode is not a valid leash specification

**Examples**

| ?- leash(off).

turns off all leashing; now when you creep you will get an exhaustive trace but no opportunity to interact with the debugger. You can get back to the debugger to interact with it by pressing ^c t.

| ?- leash([call, redo]).
leashes on the Call and Redo ports. When creeping, the debugger will now stop at every Call and Redo port to allow you to interact.

See Also

Section 5.2 [Basic Debug], page 233.
11.3.111 length/2

Synopsis
length(?List, ?Integer)

Integer is the length of List. If List is instantiated to a proper list of term, or Integer to an integer, then the predicate is determinate.

Arguments
List list of term
Integer integer, non-negative

Description
If List is a list of indefinite length (that is, either a variable or of the form [...|X], where X is a variable) and if Integer is bound to an integer, then List is made to be a list of length Integer with unique variables used to “pad” the list. If List cannot be made into a list of length Integer, then the call fails.

\[
| ?- List = [a,b|X], length(List, 4).
\]

List = [a,b,_A,_B],
X = [_A,_B] ;

| ?-

If List is bound, and is not a list, then length/2 simply fails.

If List is a cyclic list, and Integer is bound to a non-negative integer, then length/2 simply fails.

If List is a cyclic list, and Integer is a variable, then the behavior is unspecified and may change in the some future version.

Backtracking
If Integer is unbound, then it is unified with all possible lengths for the list List.

Exceptions
type_error
   Integer is not an integer
domain_error
   Integer < 0
Examples

| ?- length([1,2], 2). 
| yes |
| ?- length([1,2], 0). 
| no |
| ?- length([1,2], X). 
| X = 2 ; |
| no |

See Also

Section 4.8.3 [ref-lte-acl], page 128, library(lists).
11.3.112 \(</2\)  

**Synopsis**  
+Expr1 < +Expr2  

Evaluates Expr1 and Expr2 as arithmetic expressions. The goal succeeds if the result of evaluating Expr1 is strictly less than the result of evaluating Expr2.

**Arguments**  
Expr1 expr, must be ground  
Expr2 expr, must be ground  

**Description**  
Evaluates Expr1 and Expr2 as arithmetic expressions and compares the results.

**Exceptions**  
Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

**Examples**  

\[ ?- 23 + 2.2 < 23 - 2.2. \]

\[ \text{yes} \]

\[ ?- X = 31, Y = 25, X + Y < X - Y \]

\[ \text{no} \]

**See Also**  
Section 4.7 [ref-ari], page 119,
11.3.113 library_directory/1

Synopsis

:- multifile user:library_directory/1.

user:library_directory(+DirSpec)

Defines a library directory. Used by predicates taking file_spec as input argument.

Arguments

DirSpec    file_spec
Either an atom giving the path to a file or directory, or PathAlias(DirSpec), where PathAlias is defined by a file_search_path/2 rule.

Description

These facts define directories to search when a file specification library(File) is expanded to the full path, in addition to the predefined library path, which is tried first.

The file_search_path mechanism is an extension of the library_directory scheme and is preferred.

Examples

| ?- [user].
| % compiling user...
| :- multifile user:library_directory/1.
| library_directory('/usr/joe_bob/prolog/libs').
| end_of_file.
| % compiled user in module user, 0 msec 384 bytes
| yes
| ?- ensure_loaded(library(flying)).
| % loading file /usr/joe_bob/prolog/libs/flying.qof

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

absolute_file_name/[2,3], file_search_path/2, load_files/[1,2],
11.3.114 line_count/2

Synopsis

line_count(+Stream, -Count)

Obtains the total number of lines either input from or output to the open text stream Stream and unifies it with Count.

Arguments

Stream stream_object, must be ground
   A valid open text stream.
Count integer
   The resulting line count of the stream.

Description

A freshly opened stream has a line count of 0, i.e. this predicate counts the number of newlines seen. When a line is input from or output to a non-interactive Prolog stream, the line count of the Prolog stream is increased by one. Line count for an interactive stream reflects the total line input from or output to any interactive stream, i.e. all interactive streams share the same counter.

The count is reset by set_stream_position/2.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

byte_count/2, character_count/2, line_position/2, stream_position/2, set_stream_position/2, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.115 line_position/2

Synopsis

line_position(+Stream, -Count)

Obtains the total number of characters either input from or output to the current line of the open text stream Stream and unifies it with Count.

Arguments

Stream stream_object, must be ground
A valid open text stream.

Count integer
The resulting line count of the stream.

Description

A fresh line has a line position of 0, i.e. this predicate counts the length of the current line. Line count for an interactive stream reflects the total line input from or output to any interactive stream, i.e. all interactive streams share the same counter.

The count is reset by set_stream_position/2.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

byte_count/2, character_count/2, line_count/2, stream_position/2, set_stream_position/2, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.116 listing/[0,1]

Synopsis

`listing(+PredSpecs)`

Prints the clauses of all the interpreted procedures currently in the type-in module of the Prolog database, or of `PredSpecs`, to the current output stream, using `portray_clause/1`.

Arguments

`:PredSpecs`

- `pred_spec_tree`
  - A predicate specification, or a list of such.

Exceptions

- `type_error`
  - `PredSpecs` of the wrong type.

Examples

You could list the entire program to a file using the command

```prolog
| ?- tell(file), listing, told.
```

Note that `listing/[0,1]` does not work on compiled procedures.

`listing/1` is dependent on the source module. As a special case,

```prolog
| ?- listing(mod:__).
```

will list all the dynamic predicates in module mod. However, `listing/0` is not dependent on the source module; it refers instead to the type-in module.

Variables may be included in predicate specifications given to `listing/1`. For example, you can list clauses for `f` in any current module with:

```prolog
| ?- listing(_,f).
```

See Also

Section 4.11 [ref-mod], page 161.
11.3.117 load_files/[1,2]

Synopsis
load_files(+Files)

load_files(+Files, +Options)

[+Files]

Loads the specified Prolog source and/or object file(s) into memory. Subsumes all other load predicates.

Arguments

:Files file_spec or list of file_spec, must be ground
A file specification or a list of file specifications; extensions optional.

Options list of term, must be ground
A list of zero or more options of the form:

if(X)  true (the default) to always load, or changed to load only if the file has not yet been loaded or if it has been modified since it was last loaded. A non-module file is not considered to have been previously loaded if it was loaded into a different module. The file specification user is never considered to have been previously loaded.

when(When) always (the default) to always load, or compile_time to load only if the goal is not in the scope of another load_files/[1,2] directive occurring in a PO file.
The latter is intended for use when the file only defines predicates that are needed for proper term or goal expansion during compilation of other files.

load_type(LoadType) source to load source files only, object to load object (PO) files only, or latest (the default) to load any type of file, whichever is newest. If the file specification is user, then source is forced.

imports(Imports) all (the default) to import all exported predicates if the file is a module file, or a list of predicates to import.

compilation_mode(Mode) compile to translate into compiled code, consult to translate into static, interpreted code, or assert_all to translate into dynamic, interpreted code.
The default is the compilation mode of any ancestor load_files/[1,2] goal, or compile otherwise. Note that Mode has no effect when a PO file is loaded, and that it is recommended to
use `assert_all` in conjunction with `load_type(source)`, to ensure that the source file will be loaded even in the presence of a PO file.

In addition the `open/4` options `encoding/1`, `encoding_signature/1` and `eol/1` can be specified. These will be used if the Prolog code is loaded from a source file. See Section 11.3.148 [mpg-ref-open], page 1115, for details.

**Description**

`load_files/[1,2]` reads Prolog clauses, in source or precompiled form, and adds them to the Prolog database, after first deleting any previous versions of the predicates they define. Clauses for a single predicate must all be in the same file unless that predicate is declared to be multifile.

If a source file contains directives, that is, terms with principal functor `:-/1` or `?-/1`, then these are executed as they are encountered. Initialization goals specified with `initialization/1` are executed after the load.

A non-module source file can be loaded into any module by `load_files/[1,2]`, but the module of the predicates in a precompiled file is fixed at the time it is created.

**Exceptions**

- `instantiation_error`
  - `Files` or `Options` is not ground.
- `type_error`
  - In `Files` or `Options`.
- `domain_error`
  - Illegal option in `Options`.
- `existence_error`
  - A specified file does not exist. If the `fileerrors` flag is `off`, then the predicate fails instead of raising this exception.
- `permission_error`
  - A specified file not readable. If the `fileerrors` flag is `off`, then the predicate fails instead of raising this exception.

While loading clauses from a PO file, clauses for an existing multifile predicate were encountered, but were compiled in a way different from the existing clauses. In this case, the existing clauses remain untouched, the multifile clauses from the PO file are simply ignored, the load continues, and an exception is raised at the end.

**Examples**

Several of the other built-in predicates of this category could be defined in terms of `load_files/2`:
[File|Files] :-
    load_files([File|Files]).

consult(Files) :-
    load_files(Files, [load_type(source), compilation_mode(consult)]).

ensure_loaded(Files) :-
    load_files(Files, [if(changed)]).

use_module(File) :-
    load_files(File, [if(changed)]).

use_module(File, Imports) :-
    load_files(File, [if(changed), imports(Imports)]).

Code that is only needed at compile-time, e.g. for operator declarations or compile-time expansion, is conveniently loaded with the following idiom:

    :- load_files(library(obj_decl), [when(compile_time), if(changed)]).

See Also
Section 4.3.2 [ref-lod-lod], page 80.
11.3.118 load_foreign_resource/1

Synopsis

load_foreign_resource(:Resource)

Load the foreign resource Resource into Prolog. Relies on the hook predicates foreign_resource/2 and foreign/[2,3].

Arguments

:Resource  file_spec, must be ground

The foreign resource to be loaded. The file extension can be omitted.

Description

load_foreign_resource/1 takes a foreign resource and loads it into Prolog.

The extension can be omitted from the filename given in the Resource argument.

Uses the foreign/[2,3] and foreign_resource/2 facts defined by the user to make the connection between a Prolog procedure and the foreign function. In this context, the resource name is derived from Resource name by deleting any leading path and extension from the absolute file name of Resource.

When loading the foreign resource, it looks for a foreign_resource/2 fact for the resource name. For each symbol in that fact, it looks for a foreign/[2,3] fact that gives the name of the Prolog procedure associated with the foreign symbol and the argument specification.

Contrary to most hook predicates which reside in the user module, load_foreign_resource/1 will look for foreign_resource/2 and foreign/[2,3] facts defined in its source module.

Foreign resources are created with the splfr tool (see Section 6.2.5 [The Foreign Resource Linker], page 296).

Exceptions

Errors in the specification of foreign/[2,3] and foreign_resource/2 will all be reported when load_foreign_resource/1 is called.

instantiation_error

Resource not ground.

type_error

Resource not an atom, or argument of a declared fact of the wrong type.

domain_error

Invalid argument of foreign/[2,3] fact.

existence_error

Resource does not exist as a foreign resource, or Resource does not have a foreign_resource/2 fact, or declared function does not exist, or declared function does not have a foreign/[2,3] fact.
domain_error
Invalid option to foreign_resource/2.

consistency_error
Function declared twice with clashing declarations.

permission_error
Attempt to redefine built-in predicate.

Examples
library(codesio) contains a foreign resource consisting of three foreign functions, one
init function, and one deinit function. The Prolog source file contains the following lines of
code:

    :- dynamic foreign/2, foreign_resource/2.

    foreign(codes_to_stream, '$codes_to_stream'(codes,-address('SP_stream'))).
    foreign(open_buf_stream, '$open_buf_stream'(address('SP_stream'))).
    foreign(stream_to_codes, '$stream_to_codes'(address('SP_stream'),-term,-term)).

    foreign_resource(codesio, [
        init(codesio_init),
        deinit(codesio_deinit),
        codes_to_stream,
        open_buf_stream,
        stream_to_codes
    ]).

    :- load_foreign_resource(library(system(codesio))).

Comments
Note that the foreign declarations are needed by other operations as well and should not
be abolished after loading the foreign resource.

See Also
unload_foreign_resource/1, foreign_resource/2, foreign/[2,3], Section 6.2.1 [Foreign
Resources], page 291, Section 6.2 [Calling C from Prolog], page 291.
11.3.119 member/2

Synopsis

\[\text{member(?Element, ?List)}\]

is true if \(\text{Element}\) occurs in the \(\text{List}\). It may be used to test for an element or to enumerate all the elements by backtracking. Indeed, it may be used to generate the \(\text{List}\)!

Arguments

- \(\text{Element}\) term
- \(\text{List}\) list of term

Description

In the context of this predicate, a term occurs in a list if it can be unified with an element of the list.

Backtracking

On backtracking, an attempt is made to unify \(\text{Element}\) with successive elements of \(\text{List}\). If \(\text{List}\) is not a proper list, then on backtracking it is unified with lists of ever increasing length.

Examples

```
| ?- member(foo(X), [foo(1), bar(2), foo(3)]).
  X = 1 ? ;
  X = 3 ? ;
  no
```

Exceptions

None.

See Also

Section 4.8.3 [ref-lte-acl], page 128, library(lists).
11.3.120 memberchk/2

Synopsis

memberchk(?Element, ?List)

is true if the given Element occurs in the given List. Its purpose is to test for membership. Normally, the two arguments are ground.

Arguments

Element term
List list of term

Description

In the context of this predicate, a term occurs in a list if it can be unified with an element of the list.

Backtracking

The predicate is determinate and commits to the first successful unification, if any.

Examples

| ?- memberchk(bar, [foo,bar,baz]).  
  yes |

Exceptions

None.

See Also

Section 4.8.3 [ref-lte-acl], page 128, library(lists).
11.3.121 message_hook/3

Synopsis

:- multifile user:message_hook/3.

user:message_hook(+Severity, +MessageTerm, +Lines)

Overrides the call to `print_message_lines/3` in `print_message/2`. A way for the user to intercept the `Message` of type `Severity`, whose translations is `Lines`, before it is actually printed.

Arguments

`Severity` one of `[informational, warning, error, help, silent]`

`MessageTerm` term

`Lines` list of list of pair

Is of the form `[Line1, Line2, ...]`, where each `Linei` is of the form `[Control_1-Args_1, Control_2-Args_2, ...].`

Description

After a message is parsed, but before the message is written, `print_message/2` calls

`user:message_hook(+Severity, +MsgTerm, +Lines)`

If the call to `user:message_hook/3` succeeds, then `print_message/2` succeeds without further processing. Otherwise the built-in message portrayal is used. It is often useful to have a message hook that performs some action and then fails, allowing other message hooks to run, and eventually allowing the message to be printed as usual.

Exceptions

An exception raised by this predicate causes an error message to be printed and then the original message is printed using the default message text and formatting.

See Also

Section 4.16 [ref-msg], page 212.
**11.3.122 meta_predicate/1 declaration**

**Synopsis**

```prolog
:- meta_predicate +MetaSpec
```

Provides for module name expansion of arguments in calls to the predicate given by `MetaSpec`. All `meta_predicate/1` declarations should be at the beginning of a module.

**Arguments**

`:MetaSpec` **callable**, must be ground

Goal template or list of goal templates, of the form `functor(Arg1, Arg2, ...)`.

Each `Argn` is one of:

- `'.'` requires module name expansion
  
  If the argument will be treated as a goal, then it is better to explicitly indicate this using an integer, see the next item.

- `nsuppressed`
  
  a non-negative integer.

  This is a special case of `'.'` that means that the argument can be made into a goal by adding `nsuppressed` additional arguments. E.g., if the argument will be passed to `call/1`, then 0 (zero) should be used.

As another example, the meta_predicate declaration for the built-in `call/3` would be `:- meta_predicate call(2,?,?)`, since `call/3` will add two arguments to its first argument in order to to construct the goal to invoke.

An integer is treated the same as `'.'` above by the SICStus runtime. Other tools, such as the cross referencer (see Section 9.12 [The Cross-Referencer], page 379) and the SICStus Prolog IDE (see Section 3.11 [SPIDER], page 29), will use this information to better follow predicate references in analyzed source code.

- `'*'`
- `'+'
- `-'
- `'?'
  
  ignored

**Exceptions**

Exceptions in the context of loading code are printed as error messages.

- `instantiation_error`
  
  `MetaSpec` not ground.

- `type_error`
  
  `MetaSpec` not a valid specification.

- `context_error`
  
  Declaration appeared in a goal.
permission_error

Declaration appeared as a clause.

Examples

Consider a sort routine, `mysort/3`, to which the name of the comparison predicate is passed as an argument:

```prolog
mysort(LessThanOrEqual, InputList, OutputList) :-
    ...
    /** LessThanOrEqual is called exactly like the built-in @=</2
     ( call(LessThanOrEqual, Term1, Term2) -> ... ; ... ),
    ...
```

An appropriate `meta_predicate` declaration for `mysort/3` is:

```prolog
:- meta_predicate mysort(2, +, -).
```

since the first argument, `LessThanOrEqual`, will have two additional arguments added to it (by `call/3`) when invoked.

This means that whenever a goal `mysort(A, B, C)` appears in a clause, it will be transformed at load time into `mysort(M:A, B, C)`, where `M` is the source module. The transformation will happen unless:

1. `A` has an explicit module prefix, or
2. `A` is a variable and the same variable appears in the head of the clause in a module-name-expansion position.

See Also

Section 4.3.4.6 [Meta-Predicate Declarations], page 85, Section 4.11.15 [ref-mod-mne], page 171.
11.3.123 mode/1

Synopsis
:- mode +Mode

Currently a dummy declaration.

Arguments
:Mode term

Exceptions
Exceptions in the context of loading code are printed as error messages.

correct_error
   Declaration appeared in a goal.

permission_error
   Declaration appeared as a clause.

See Also
Section 4.3.4.9 [Mode Declarations], page 86.
11.3.124 module/[2,3]  

**Declaration**

**Synopsis**

:- module(+ModuleName, +PublicPred).

:- module(+ModuleName, +PublicPred, +Options).

Declares the file in which the declaration appears to be a module file named *ModuleName*, with public predicates *PublicPred*. Must appear as the first term in the file.

**Arguments**

*ModuleName*

*atom*, must be nonvar

*PublicPred*

list of simple_pred_spec, must be ground

List of predicate specifications of the form *Name*/Arity.

*Options*

list of term, must be ground

A list of zero or more options of the form:

hidden(Boolean)

*Boolean* is *false* (the default) or *true*. In the latter case, tracing of the predicates of the module is disabled (although spypoints can be set), and no source information is generated at compile time.

**Description**

The definition of a module is not limited to a single file, because a module file may contain commands to load other files. If *myfile*, a module file for *ModuleName*, contains an embedded command to load *yourfile* and if *yourfile* is not itself a module file, then all the predicates in *yourfile* are loaded into module *ModuleName*.

**Exceptions**

Exceptions in the context of loading code are printed as error messages.

instantiation_error

Declaration not ground.

type_error

An argument has the wrong type.

domain_error

Some arity is an integer < 0, or invalid option given.

representation_error

Some arity is an integer > 255.

context_error

Declaration appeared in a goal, or not first in the file being loaded.

permission_error

Declaration appeared as a clause.
Examples
A module declaration from the Prolog library:

```prolog
:- module(varnumbers, [ numbervars/1, varnumbers/2, varnumbers/3 ], [ hidden(true) ]).
```

See Also
Section 4.3.4.7 [Module Declarations], page 85, Section 4.11 [ref-mod], page 161.
11.3.125 multifile/1            declaration, ISO

Synopsis

:- multifile +PredSpecs

Declares the clauses of the predicates defined by PredSpecs to be multifile in the source file (suppresses compile-time warnings).

Arguments

:PredSpecs

pred_spec_forest, must be ground
A predicate specification, or a list of such, or a sequence of such separated by commas.

Description

By default, all clauses for a predicate are expected to come from just one file. This assists with reloading and debugging of code. Declaring a predicate multifile means that its clauses can be spread across several different files. This is independent of whether or not the predicate is declared dynamic.

Should precede all the clauses for the specified predicates in the file.

There should be a multifile declaration for a predicate \( P \) in every file that contains clauses for \( P \). If a multifile predicate is dynamic, then there should be a dynamic declaration in every file containing clauses for the predicate.

When a file containing clauses for a multifile predicate \( (P) \) is reloaded, the clauses for \( P \) that previously came from that file are removed. Then the new clauses for \( P \) (which may be the same as the old ones) are added to the end of the definition of the multifile predicate.

If a multifile declaration is found for a predicate that has already been defined in another file (without a multifile declaration), then this is considered to be a redefinition of that predicate. Normally this will result in a multiple-definition style-check warning (see style_check/1).

The predicate source_file/2 can be used to find all the files containing clauses for a multifile predicate.

Comments

multifile is not an ISO predefined prefix operator.

Exceptions

Exceptions in the context of loading code are printed as error messages.

instantiation_error

PredSpecs not ground.

type_error

PredSpecs not a valid pred_spec_forest.
domain_error
  Some arity is an integer < 0.

representation_error
  Some arity is an integer > 255.

context_error
  Declaration appeared in a goal.

permission_error
  Declaration appeared as a clause.

See Also
Section 4.3.4.1 [Multifile Declarations], page 83.
11.3.126 mutable/1

Synopsis
mutable(+Term)

Succeeds if Term is currently instantiated to a mutable term.

Arguments
Term term

Exceptions
None.

See Also
Section 4.8.1.1 [ref-lte-met-typ], page 126, Section 4.8.9 [ref-lte-mut], page 131.
11.3.127 name/2  

**Synopsis**

name(+Constant, -Codes)

name(-Constant, +Codes)

*Codes* is the list consisting of the *codes* comprising the printed representation of *Constant*.

**Arguments**

*Constant*  atomic  
*Codes*  codes

**Description**

Initially, either *Constant* must be instantiated to a number or an atom, or *Codes* must be instantiated to a proper *codes*.

If *Codes* is instantiated to a proper *codes* that corresponds to the correct syntax of a number, then *Constant* will be unified with that number.

Else if *Codes* is instantiated to a proper *codes*, then *Constant* will be unified with the atom containing exactly those characters.

Else, *Constant* should be instantiated to a number or atom, and *Codes* will be unified with the *codes* that make up its printed representation.

There are atoms for which `name(Constant, Codes)` is true, but which will not be constructed if `name/2` is called with *Constant* uninstantiated. One such atom is the atom '1976'. It is recommended that new programs use `atom_codes/2` or `number_codes/2`, as these predicates do not have this ambiguity.

**Exceptions**

- **instantiation_error**
  
  *Constant* is uninstantiated and *Codes* is not instantiated enough

- **type_error**
  
  If *Constant* is a compound term

- **representation_error**
  
  An element of *Codes* is an invalid character code, or *Codes* is a list corresponding to a number or atom that can’t be represented

**Examples**

```
| ?- name(foo, L).
L = [102,111,111]

| ?- name('Foo', L).
L = [70,111,111]
```
| ?- name(431, L).
L = [52,51,49]
| ?- name(X, [102,111,111]).
X = foo
| ?- name(X, [52,51,49]).
X = 431
| ?- name(X, "15.0e+12").
X = 1.5E+13

**See Also**

Section 4.8.4 [ref-lte-c2t], page 128.
11.3.128 nl/[0,1]  

Synopsis

nl

nl(+Stream)

Terminates the current output record on the current output stream or on Stream.

Arguments

Stream stream

+stream_object, must be ground

A valid output text stream, defaults to the current output stream.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

Section 4.6.6 [ref-iou-cou], page 108.
11.3.129 nodebug/0

Synopsis

nodebug

Turns the debugger off.

Exceptions

None.

See Also

Section 5.2 [Basic Debug], page 233.
11.3.130 nonmember/2

Synopsis

nonmember(?Element, ?List)

is true if the given Element does not occur in the given List. Its purpose is to test for membership. Normally, the two arguments are ground.

Arguments

Element  term
List      list of term

Description

In the context of this predicate, a term occurs in a list if it can be unified with an element of the list.

Backtracking

The predicate is determinate and either succeeds or fails. It never binds variables.

Examples

| ?- nonmember(bar, [foo,bar,baz]).
  no

Exceptions

None.

See Also

Section 4.8.3 [ref-lte-acl], page 128, library(lists).
11.3.131 nonvar/1  

Synopsis

nonvar(+Term)

Term is currently instantiated.

Arguments

Term term

Examples

\[
\begin{align*}
\text{| ?- nonvar(foo(X,Y)).} & \quad \text{true ; no} \\
\text{| ?- nonvar([X,Y]).} & \quad \text{true ; no} \\
\text{| ?- nonvar(X).} & \quad \text{no} \\
\text{| ?- Term = foo(X,Y), nonvar(Term).} & \quad \text{true ; no}
\end{align*}
\]

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.132 nospy/1

Synopsis

nospy +PredSpecs

Any spypoints (plain and conditional) on the predicates represented by PredSpecs are removed.

Arguments

:PredSpecs
    pred_spec_tree
    A predicate specification, or a list of such.

Exceptions

instantiation_error
type_error
domain_error

    if a PredSpec is not a valid procedure specification

See Also

Section 5.2 [Basic Debug], page 233, Section 5.3 [Plain Spypoint], page 235.
11.3.133 nospyall/0

Synopsis

nospyall

Removes all the spypoints (including the generic ones) that have been set.

Exceptions

None.

See Also

Section 5.2 [Basic Debug], page 233.
11.3.134 \(
=\not_/2
\)

Synopsis

\(+Expr1 =\not_+Expr2\)

Succeeds if the results of evaluating \(Expr1\) and \(Expr2\) are not equal.

Arguments

\(Expr1\) expr, must be ground
\(Expr2\) expr, must be ground

Description

Evaluates \(Expr1\) and \(Expr2\) as arithmetic expressions and compares the results.

Exceptions

Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

Examples

\[
\begin{align*}
| \mspace{1em} ?- & \ 7 =\not_14/2. \\
& \text{no} \\
| \mspace{1em} ?- & \ 7 =\not_15/2. \\
& \text{yes}
\end{align*}
\]

See Also

Section 4.7 [ref-ari], page 119,
11.3.135  

**Synopsis**

+Expr1 =< +Expr2

Succeeds if the result of evaluating Expr1 is *less than or equal* to the result of evaluating Expr2.

**Arguments**

<table>
<thead>
<tr>
<th>Expr1</th>
<th>expr, must be ground</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr2</td>
<td>expr, must be ground</td>
</tr>
</tbody>
</table>

**Description**

Evaluates Expr1 and Expr2 as arithmetic expressions and compares the results.

**Exceptions**

Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

**Examples**

| ?- 42 =< 42. |
|---|---|
| yes |
| ?- "b" =< "a". |
| no |

**Comments**

Note that the symbol ‘=<’ is used here rather than ‘<=’, which is used in some other languages. One way to remember this is that the inequality symbols in Prolog are the ones that cannot be thought of as looking like arrows. The ‘<’ or ‘>’ always points at the ‘=’.

**See Also**

Section 4.7 [ref-ari], page 119,
11.3.136  \( \geq/2 \)  

**Synopsis**

\(+Expr1 \geq +Expr2\)

Succeeds if the results of evaluating \( Expr1 \) and \( Expr2 \) are equal.

**Arguments**

- \( Expr1 \) expr, must be ground
- \( Expr2 \) expr, must be ground

**Description**

Succeeds if the result of evaluating \( Expr1 \) is greater than or equal to the result of evaluating \( Expr2 \).

**Exceptions**

Arithmetic errors (see Section 4.7.3 [ref-ari-exc], page 119).

**Examples**

```
| ?- 42 \( \geq \) 42.

yes
```

```
| ?- "b" \( \geq \) "a".

yes
```

**See Also**

Section 4.7 [ref-ari], page 119,
**11.3.137 \+/1**

*ISO*

**Synopsis**
\+/ +P

Fails if the goal P has a solution, and succeeds otherwise. Equivalent to:

\[
\text{call}(P) \rightarrow \text{fail} ; \text{true}.
\]

except that the use of call/1 often can be avoided.

**Arguments**

:P *callable*, must be nonvar

**Description**

This is not normally regarded as a built-in predicate, since it is part of the syntax of the language. However, it is like a built-in predicate in that you can say `call((\+ P))`.

Cuts in P do not make sense, but are allowed, their scope being the goal P.

**Comments**

Remember that with prefix operators such as this one it is necessary to be careful about spaces if the argument starts with a ‘(’. For example:

```prolog
| ?- \+ (P,Q).
```

is this operator applied to the conjunction of P and Q, but

```prolog
| ?- \+ (P,Q).
```

would require a predicate \+/2 for its solution. The prefix operator can however be written as a functor of one argument; thus

```prolog
| ?- \+((P,Q)).
```

is also correct.

**Exceptions**

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

**See Also**

Section 4.2 [ref-sem], page 61.
11.3.138 \ /=/2

Synopsis

+Term1 \ /= +Term2

Term1 and Term2 do not unify.

Arguments

Term1 term
Term2 term

Description

The same as \+/ X = Y; i.e. X and Y are not unifiable.

Exceptions

None.

See Also

Chapter 2 [Glossary], page 7.
11.3.139 notrace/0

Synopsis

notrace

Turns the debugger off.

Exceptions

None.

See Also

Section 5.2 [Basic Debug], page 233.
11.3.140 nozip/0

Synopsis

nozip

Turns the debugger off.

Exceptions

None.

See Also

Section 5.2 [Basic Debug], page 233.
11.3.141  number/1

Synopsis

number(+Term)

Term is currently instantiated to a number.

Arguments

Term  term

Examples

| ?- number(5.2).
   yes
   | ?- number(5).
   yes

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.142 number_chars/2

Synopsis

number_chars(+Number, -Chars)

number_chars(-Number, +Chars)

Chars is the chars comprising the printed representation of Number.

Arguments

Number  number
Chars    chars

Description

Initially, either Number must be instantiated to a number, or Chars must be instantiated to a proper chars.

If Chars is instantiated to a chars that corresponds to the correct syntax of a number, then Number will be unified with that number.

Else, Number should be instantiated to a number, and Chars will be unified with the chars that make up its printed representation.

Exceptions

instantiation_error
  Number is uninstantiated and Chars is not instantiated enough

type_error
  Number is not a number or Chars cannot be unified with a chars

representation_error
  Chars is a list corresponding to a number that can’t be represented

syntax_error
  Chars does not correspond to a syntactically valid number

The check of Chars when Number is instantiated was added in release 4.3 for alignment with the ISO Prolog standard. Previous releases simply failed in this case instead of reporting an error for malformed Chars.

See Also

number_codes/2.
11.3.143 number_codes/2

Synopsis

number_codes(+Number, -Codes)

number_codes(-Number, +Codes)

*Codes* is the *codes* comprising the printed representation of *Number*.

Arguments

*Number* number

*Codes* codes

Description

Initially, either *Number* must be instantiated to a number, or *Codes* must be instantiated to a proper *codes*.

If *Codes* is instantiated to a *codes* that corresponds to the correct syntax of a number, then *Number* will be unified with that number.

Else, *Number* should be instantiated to a number, and *Codes* will be unified with the *codes* that make up its printed representation.

Exceptions

instantiation_error

*Number* is uninstantiated and *Chars* is not instantiated enough

type_error

*Number* is not a number or *Codes* cannot be unified with a list of integers

representation_error

An element of *Codes* is an invalid character code, or *Codes* is a list corresponding to a number that can’t be represented

syntax_error

*Codes* does not correspond to a syntactically valid number

The check of *Codes* when *Number* is instantiated was added in release 4.3 for alignment with the ISO Prolog standard. Previous releases simply failed in this case instead of reporting an error for malformed *Codes*.

Examples

```
| ?- number_codes(foo, L).

! Type error in argument 1 of number_codes/2
! expected a number, but found foo
! goal:  number_codes(foo,-104)
```
| ?- number_codes(431, L).
L = [52,51,49]
| ?- number_codes(X, [102,111,111]).
! Syntax error in number_codes/2
! number syntax
! in line 0
| ?- number_codes(X, [52,51,49]).
X = 431
| ?- number_codes(X, "15.0e+12").
X = 1.5E+13

See Also
number_chars/2.
11.3.144 numbervars/3

Synopsis

numbervars(+Term, +FirstVar, -LastVar)

instantiates each of the variables in Term to a term of the form '$VAR'(N).

Arguments

Term: term
FirstVar: integer, must be nonvar
LastVar: integer

Description

FirstVar is used as the value of N for the first variable in Term (starting from the left). The second distinct variable in Term is given a value of N satisfying “N is FirstVar+1”; the third distinct variable gets the value FirstVar+2, and so on. The last variable in Term has the value LastVar-1.

Notice that in the example below, write_canonical/1 is used rather than writeq/1. This is because writeq/1 treats terms of the form '$VAR'(N) specially; it writes ‘A’ if N=0, ‘B’ if N=1, . . . ‘Z’ if N=25, ‘A1’ if N=26, etc. That is why, if you type the goal in the example below, then the variable bindings will also be printed out as follows:

```
Term = foo(W,W,X),
  A = W,
  B = X
```

Exceptions

instantiation_error
  FirstVar is uninstantiated

type_error
  FirstVar is not an integer

Examples

```
| ?- Term = foo(A, A, B), numbervars(Term, 22, _), write_canonical(Term).
  foo( '$VAR'(22), '$VAR'(22), '$VAR'(23))
```

See Also

Section 4.8.6 [ref-lte-anv], page 129, write_term/[2,3].
11.3.145 on_exception/3

Synopsis

\[
\text{on_exception}(-\text{Exception}, +\text{ProtectedGoal}, +\text{Handler})
\]

same as:

\[
\text{catch}(\text{ProtectedGoal}, \text{Exception}, \text{Handler})
\]

Specify an exception handler for \text{ProtectedGoal}, and call \text{ProtectedGoal}, as described in Section 4.15 [ref-ere], page 197.

Arguments

Exception term

:ProtectedGoal callable, must be nonvar

:Handler callable, must be nonvar

Examples

Fail on exception:

\[
\text{- meta_predicate fail_on_exception(0).}
\]

\[
\text{fail_on_exception(C):-}
\]

\[
\text{on_exception(E, C, print_exception_then_fail(C, E)).}
\]

\[
\text{print_exception_then_fail(C, E):-}
\]

\[
\text{format(user_error, 'Exception occurred while calling `q:'\n, \[C\]),}
\]

\[
\text{print_message(warning, E),}
\]

\[
\text{fail.}
\]

Backtracking

Depends on \text{ProtectedGoal} and \text{Handler}.

Exceptions

None.

See Also

Section 4.15 [ref-ere], page 197.
11.3.146 once/1

Synopsis
once(+P)

Equivalent to:

\[ \text{call}(P) \rightarrow \text{true} ; \text{fail}. \]

except that the use of \texttt{call/1} often can be avoided.

Arguments
: \(P\) callable, must be nonvar

Cuts in \(P\) do not make sense, but are allowed, their scope being the goal \(P\).

Exceptions
Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

See Also
Section 4.2 [ref-sem], page 61.
11.3.147 op/3

Synopsis

\( \text{op}(+\text{Precedence}, +\text{Type}, +\text{Name}) \)

declares \text{Name} to be an operator of the stated \text{Type} and \text{Precedence}.

Arguments

\text{Precedence} \quad \text{integer}, must be nonvar and in the range 1-1200

\text{Type} \quad \text{one of } [\text{xfx}, \text{xfy}, \text{yfx}, \text{fx}, \text{fy}, \text{xf}, \text{yf}], \text{must be nonvar}

\text{Name} \quad \text{atom or list of atom, must be ground}

Description

Operators are a notational convenience to read and write Prolog terms. You can define new
operators using \text{op}/3.

The \text{Precedence} of an operator is used to disambiguate the way terms are parsed. The
general rule is that the operator with the highest precedence is the principal functor.

The \text{Type} of an operator decides the position of an operator and its associativity. In the
atom that represents the type the character ‘f’ represents the position of the operator. For
example, a type ‘fx’ says that the operator is a prefix operator. The character ‘y’ indicates
that the operator is associative in that direction. For example, an operator of type ‘xfy’ is
a right-associative, infix operator.

To cancel the operator properties of \text{Name} (if any) set \text{Precedence} to 0.

\textbf{Please note:} operators are \textit{global}, as opposed to being local to the current module, Prolog
text, or otherwise.

Exceptions

instantiation_error
    An argument is not ground

type_error
    \text{Precedence} is not an integer or \text{Type} is not an integer or an operator is not an
    atom

domain_error
    \text{Precedence} is not in the range 1-1200, or \text{Type} is invalid,

permission_error
    Attempt to redefine the operator ','

See Also

current_op/3, Section 4.1.5 [ref-syn-ops], page 47.
11.3.148 open/[3,4] 

Synopsis
open(+FileSpec, +Mode, -Stream)
open(+FileSpec, +Mode, -Stream, +Options)

Creates a Prolog stream by opening the file FileSpec in mode Mode with options Options.

Arguments
FileSpec file_spec, must be ground
A file specification.

Mode one of [read, write, append], must be nonvar
An atom specifying the open mode of the target file. One of:
read open FileSpec for input.
write open FileSpec for output. A new file is created if FileSpec does not exist. If the file already exists, then it is set to empty and its previous contents are lost.
append opens FileSpec for output. If FileSpec already exists, then it adds output to the end of it. If not, then a new file is created, as for the write mode.

Options list of term, must be ground
A list of zero or more of the following.
type(+T) Specifies whether the stream is a text or binary stream. Default is text.
reposition(+Boolean)
Specifies whether repositioning is required for the stream (true), or not (false). The latter is the default.
For text streams reposition(true) affects the default eol/1 and encoding_signature/1 options, see below. Also, not all encodings supports this option (see Section 4.6.7.5 [ref-impl-ref-std-enc], page 111).
alias(+A) Specifies that the atom A is to be an alias for the stream.
eof_action(+Action)
Specifies what action is to be taken when the end of stream has already been reported (by returning -1 or end_of_file), and a further attempt to input is made. Action can have the following values:
error An exception is raised. This is the default.
eof_code An end of stream indicator (-1 or end_of_file) is returned again.
reset  The stream is considered not to be past end of stream and another attempt is made to input from it.

encoding(Encoding)
   Specifies the encoding to use if the stream is opened in text mode, as an atom. The default is ‘ISO-8859-1’, the 8 bit subset of Unicode, i.e. “ISO-8859-1” (Latin 1) (see Section 4.6.7.5 [ref-iou-sfh-enc], page 111).
   Overridden by the encoding_signature/1 option, see below.

coding_signature(+Boolean)
   Specifies whether an encoding signature should be used (true), or not (false). An encoding signature is a special byte sequence that identifies the encoding used in the file. The most common case is one of the Unicode signatures, often called “byte order mark” (BOM).

   A Unicode signature is a special byte sequence that can be used to distinguish between several UTF encoding variants, such as “UTF-8”, “UTF-16-BE” and “UTF-16-LE”.

   If reposition(true) is specified, then encoding_signature/1 defaults to false for both streams opened in write mode and streams opened in read mode.
   If reposition(true) is not specified, and if the file is opened in mode read, then encoding_signature/1 defaults to true.

   When encoding_signature/1 option is true additional heuristics will be used if no Unicode signature is detected. Only if neither a Unicode signature nor these heuristics specifies a character encoding will the encoding/1 option, if any, be used.

   The method used for selecting character encoding when a text file is opened in mode read is the first applicable item in the following list:

   1. If the encoding_signature/1 option is true: If a byte order mark is detected, then it will be used to select between the encodings “UTF-8”, “UTF-16” or “UTF-32” with suitable endianness.

   2. If the encoding_signature/1 option is true: If an Emacs style ‘-*- coding: coding-system-*’ is present on the first non-empty line of the file, then it will be used.

   3. If an option encoding(ENCODING) Is supplied, then the specified encoding will be used.

   4. As a final fallback, “ISO-8859-1” (Latin 1) will be used.

   the character encoding selected in this way will be used if it is recognized, otherwise an error exception is raised.

If reposition(true) is not specified, and if the file is opened in mode write, then it depends on the character encoding whether
an encoding signature will be output by default or not. If you want to force an encoding signature to be output for those encodings that supports it, then you can specify `encoding_signature(true)`. Conversely, if you want to prevent an encoding signature from being output, then you can explicitly specify `encoding_signature(false)`. All UTF encodings supports an encoding signature in the form of a BOM. “UTF-8” does not write a BOM unless you explicitly specify `encoding_signature(true)`, the 16 and 32 bit UTF encodings, e.g. “UTF-16 BE”, “UTF-32 LE” writes a BOM unless explicitly requested not to with `encoding_signature(false)`.

If the file is opened in mode append, then `encoding_signature/1` defaults to `false`.

**eol(Eol)** Specifies how line endings in the file should be handled if the stream is opened in text mode.

In Prolog code, end of line is always represented by the character `\n`, which has character code 10, i.e. the same as ASCII Line Feed (LFD). The representation in the file may be different, however. *Eol* can have the following values:

- **lf** Line Feed (LF, character code 10) is used to specify a end of line. This can be used for both read mode and write mode streams.
- **crlf** A two character sequence Carriage Return (CR, character code 13) followed by Line Feed (LF, character code 10) is used to specify a end of line. This can be used for both read mode and write mode streams.
- **auto** Translate both the two character sequence CR LF and single CR or LF into an end of line character. This can be used only for read mode streams.
- **default** Use a default end of line convention. This is the default. If `reposition(true)` is specified, then this uses `lf` for both streams opened in write mode and streams opened in read mode, on all platforms.

If `reposition(true)` is not specified, then under UNIX, this uses `lf` for streams opened in write mode and `auto` for streams opened in read mode. Under Windows, this uses `crlf` for streams opened in write mode and `auto` for streams opened in read mode. This can be used for both read mode and write mode streams.

**if_exists(+Action)**

Specifies what should happen if the file already exists. Only valid if *Mode* is `write` or `append`. *Action* can have the following values:
**default**
The file is overwritten or appended to, according to the `Mode` argument. This is the default.

**error**
An exception is raised.

**generate_unique_name**
If a file named `FileSpec` already exists, then `FileSpec` is rewritten so that it refers to a non-existing file. `FileSpec` is rewritten by adding digits at the end of the file name (but before any extension). The generated name, `RealName` can be obtained by using `stream_property(Stream, file_name(RealName))` on the resulting stream. See the example below.

With this option `open/4` will never open an existing file but it may still fail to find a unique name. `open/4` may fail to find a unique name if there are thousands of files with similar names. In that case `open/4` behaves as if `if_exists(error)` had been passed.

**Description**
If `FileSpec` is a valid file specification, then the file that it denotes is opened in mode `Mode`.

The resulting stream is unified with `Stream`.

`Stream` is used as an argument to Prolog input and output predicates.

`Stream` can also be converted to the corresponding foreign representation through `stream_code/2` and used in foreign code to perform input/output operations.

On Windows, where file names are usually subject to case-normalization, the file will be created with the same case as in `FileSpec`. As an example, `open('HelloWorld.txt', write, S), stream_property(S, file_name(Name)), close(S)`. will create a file with the mixed case name `HelloWorld.txt` whereas the stream property will reflect the case-normalized name `.../helloworld.txt`. Prior to release 4.3 the file would have been created in the file system with the case-normalized name `helloworld.txt`.

**Exceptions**

**instantiation_error**
`FileSpec` or `Mode` is not instantiated. `Options` argument is not instantiated enough.

**type_error**
`FileSpec` or `Mode` is not an atom type. `Options` is not a list type or an element in `Options` is not a correct type for open options or

**domain_error**
`Mode` is not one of `read`, `write` or `append`. `Options` has an undefined option or an element in `Options` is out of the domain of the option.

**uninstantiation_error**
`Stream` is not a variable
existence_error
    The specified FileSpec does not exist.

permission_error
    Cannot open FileSpec with specified Mode and Options.

system_error
    Unexpected error detected by the operating system

Examples
The following example creates two log files, both based on the base name my.log. The files
will be written to a directory suitable for temporary files (see Section 4.5.1.3 [ref-fdi-fsp-pre],
page 98).

    | ?- open(temp('my.log'), write, S1, [if_exists(generate_unique_name)]),
      open(temp('my.log'), write, S2, [if_exists(generate_unique_name)]),
      stream_property(S1, file_name(N1)),
      stream_property(S2, file_name(N2)),
      format('Logging to ~a and ~a~n', [N1, N2]),
      ...

Under UNIX this would produce something like:

    Logging to /tmp/my.log and /tmp/my1886415233.log

See Also
Section 4.6.7 [ref-iou-sfh], page 109.
11.3.149 open_null_stream/1

Synopsis

open_null_stream(-Stream)

opens an output text stream that is not connected to any file and unifies its stream object with Stream.

Arguments

Stream stream_object

Description

Characters or terms that are sent to this stream are thrown away. This predicate is useful because various pieces of local state are kept for null streams: the predicates character_count/2, line_count/2, and line_position/2 can be used on these streams.

Exceptions

uninstantiation_error

Stream is not a variable

system_error

Unexpected error detected by the operating system

See Also

Section 4.6.7 [ref-iou-sfh], page 109.
11.3.150 \(/2 \\

**Synopsis**

\( +Q ; +R \)

Disjunction: Succeeds if \( Q \) succeeds or \( R \) succeeds.

\( +P \rightarrow +Q ; +R \)

*If \( P \) then \( Q \) else \( R \), using first solution of \( P \) only.*

**Arguments**

- \( P \) **callable**, must be nonvar
- \( Q \) **callable**, must be nonvar
- \( R \) **callable**, must be nonvar

**Description**

These are normally regarded as part of the syntax of the language, but they are like a built-in predicate in that you can write \( \text{call}((Q ; R)) \) or \( \text{call}((P \rightarrow Q ; R)) \), with identical effect if they do not contain any cuts.

By default, the character ‘|’ (vertical bar) can be used as an alternative to the infix operator ‘;’. This equivalence does not hold when ‘|’ has been declared as an operator.

Using ‘|’ as an alternative to the infix operator ‘;’ is not recommended. A future version of the ISO Prolog standard is likely to define ‘|’ as an operator and with such an operator definition the ‘|’ will no longer be equivalent to ‘;’.

The operator precedences of the ‘;’ and ‘\( \rightarrow \)’ are both greater than 1000, so that they dominate commas.

Cuts in \( P \) do not make sense, but are allowed, their scope being the goal \( P \). The scope of cuts in \( Q \) and \( R \) extends to the containing clause.

**Backtracking**

For the if-then-else construct: if \( P \) succeeds and \( Q \) then fails, then backtracking into \( P \) does not occur. A cut in \( P \) does not make sense. ‘\( \rightarrow \)’ acts like a cut except that its range is restricted to within the disjunction: it cuts away \( R \) and any choice points within \( P \). ‘\( \rightarrow \)’ may be thought of as a “local cut”.

**Exceptions**

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

**See Also**

Section 4.2 [ref-sem], page 61.
**11.3.151 otherwise/0**

**Synopsis**

`otherwise`

Always succeeds (same as `true/0`).

**Tips**

Useful for laying out conditionals in a readable way.

**Exceptions**

None.

**See Also**

Section 4.2 [ref-sem], page 61.
11.3.152 peek_byte/[1,2]  

Synopsis
peek_byte(-Byte)
peek_byte(+Stream, -Byte)

looks ahead for next input byte on the input stream Stream.

Arguments
Stream  stream_object, must be ground
  A valid input binary stream, defaults to the current input stream.
Byte     byte or -1
  The resulting next input byte available on the stream.

Description
peek_byte/[1,2] looks ahead of the next input byte of the specified input stream and unifies the byte with Byte. The peeked byte is still available for subsequent input on the stream.

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

type_error
  Byte is an invalid byte.
permission_error
  Trying to read beyond end of Stream

See Also
Section 4.6.5 [ref-iou-cin], page 107.
11.3.153 peek_char/[1,2]  
ISO

Synopsis

peek_char(-Char)

peek_char(+Stream, -Char)

looks ahead for next input character on the current input stream or on the input stream Stream.

Arguments

Stream  stream object, must be ground
        A valid input text stream.
Char  char or one of [end_of_file]
        The resulting next input character available on the stream.

Description

peek_char/[1,2] looks ahead of the next input character of the specified input stream and unifies the character with Char. The peeked character is still available for subsequent input on the stream.

Comments

It is safe to call peek_char/[1,2] several times without actually inputting any character. For example:

    | ?- peek_char(X), peek_char(X), get_char(X).
    |:
      X = a

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
    Trying to read beyond end of Stream

See Also

Section 4.6.5 [ref-iou-cin], page 107.
11.3.154 peek_code/[1,2]  

Synopsis

peek_code(-Code)
peek_code(+Stream, -Code)

looks ahead for next input character on the input stream Stream.

Arguments

Stream stream_object, must be ground
   A valid input text stream, defaults to the current input stream.

Code code or -1
   The resulting next input character available on the stream.

Description

peek_code/[1,2] looks ahead of the next input character of the specified input stream and unifies the character with Code. The peeked character is still available for subsequent input on the stream.

Comments

Comments

It is safe to call peek_code/[1,2] several times without actually inputting any character. For example:

    ?- peek_code(X), peek_code(X), get_code(X).
    X = 97

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
   Trying to read beyond end of Stream

See Also

Section 4.6.5 [ref-iou-cin], page 107.
11.3.155 phrase/[2,3]

Synopsis

\[ \text{phrase}(+\text{PhraseType}, +\text{List}) \]

\[ \text{phrase}(+\text{PhraseType}, +\text{List}, -\text{Rest}) \]

Used in conjunction with a grammar to parse or generate strings.

**Arguments**

\( :\text{PhraseType} \)

\[ \text{callable}, \text{must be nonvar} \]

Name of a phrase type.

\( \text{List} \)

\[ \text{list of term} \]

A list of symbols — tokens or codes.

\( \text{Rest} \)

\[ \text{list of term} \]

A suffix of \( \text{List} \); what remains of \( \text{List} \) after \( \text{PhraseType} \) has been found. Defaults to \( [] \).

**Description**

This predicate is a convenient way to start execution of grammar rules. Runs through the grammar rules checking whether there is a path by which \( \text{PhraseType} \) can be rewritten as \( \text{List} \).

If \( \text{List} \) is bound, then this goal corresponds to using the grammar for parsing. If \( \text{List} \) is unbound, then this goal corresponds to using the grammar for generation.

\( \text{phrase}/[2,3] \) succeeds when the portion of \( \text{List} \) between the start of \( \text{List} \) and the start of \( \text{Rest} \) is a phrase of type \( \text{PhraseType} \) (according to the current grammar rules), where \( \text{PhraseType} \) is either a non-terminal or, more generally, a grammar rule body.

\( \text{phrase}/[2,3] \) allows variables to occur as non-terminals in grammar rule bodies, just as \( \text{call}/1 \) allows variables to occur as goals in clause bodies.

**Backtracking**

Depends on \( \text{PhraseType} \).

**Exceptions**

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

**Examples**

Here is a simple grammar that parses an arithmetic expression (made up of digits and operators) and computes its value. Create a file containing the following rules:
grammar.pl

expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.

In the last rule, C is the character code of a decimal digit.

This grammar can now be used to parse and evaluate an expression:

\[ \text{grammar} \].
\[ \text{phrase(expr(Z), "-2+3*5+1"} \].

\ Z = 14 \n
\[ \text{phrase(expr(Z), "-2+3*5", Rest} \].

\ Z = 13, \nRest = [] ;

\ Z = 1, \nRest = "*5" ;

\ Z = -2, \nRest = "+3*5" ;

\ no \n
See Also

Section 4.3.5 [ref-lod-exp], page 87, Section 4.14 [ref-gru], page 189.
11.3.156 portray/1

Synopsis

:- multifile user:portray/1.

user:portray(+Term)

A way for the user to over-ride the default behavior of print/1.

Arguments

Term  term

Description

If user:portray/1 is defined, then the predicates listed below performing term output will call it on the term itself and on every non-variable subterm \( T \). If user:portray/1 succeeds, then it is assumed to have written \( T \). If it fails, then the calling predicate will write the principal functor of \( T \) and treat the arguments of \( T \) recursively.

Note that on lists ([[..]) user:portray/1 will be called on the whole list to user:portray/1 and, if that call fails, on each list element, but not on every suffix of the list.

Note that a variable written from within user:portray/1 may be written with a different name than that used by the surrounding write predicate.

The affected predicates are:

print/[1,2]
write_term/[2,3]
  when used with the option portrayed(true)
goals during debugging
  controlled by the debugger_print_options Prolog flag, whose value by default includes portrayed(true)
top-level variable bindings
  controlled by the toplevel_print_options Prolog flag, whose value by default includes portrayed(true)

Exceptions

Exceptions are treated as failures, except an error message is also printed.

See Also

Section 4.6.4 [ref-iou-tou], page 104, Section 4.9.4 [ref-lps-flg], page 136.
11.3.157 portray_clause/[1,2]

Synopsis

portray_clause(+Clause)

portray_clause(+Stream, +Clause)

Writes Clause to the current output stream. Used by listing/[0,1].

Arguments

Stream stream
   stream_object, must be ground
   A valid open Prolog stream, defaults to the current output stream.

Clause term

Description

The operation used by listing/[0,1]. Clause is written to Stream, in exactly the format in which listing/[0,1] would have written it, including a terminating full stop.

If you want to print a clause, then this is almost certainly the command you want. By design, none of the other term output commands puts a full stop after the written term. If you are writing a file of facts to be loaded by the Load Predicates, then use portray_clause/[1,2], which attempts to ensure that the clauses it writes out can be read in again as clauses.

The output format used by portray_clause/[1,2] and listing/[0,1] has been carefully designed to be clear. We recommend that you use a similar style. In particular, never put a semicolon (disjunction symbol) at the end of a line in Prolog.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

Examples

| ?- portray_clause((X:- a -> b ; c)).
- :-
   ( a ->
     b
     ; c
   ).
| ?- portray_clause((X:- a -> (b -> c ; d ; e); f)). 
  _ :-
   ( a ->
    ( b ->
     c
     ; d
     ; e
    )
    ; f
   ).

| ?- portray_clause((a:-b)). 
  a :-
  b.

| ?- portray_clause((a:-b,c)). 
  a :-
  b,
  c.

| ?- portray_clause((a:-!,(b,c))). 
  a :-
  b,!,
  c.

See Also

listing/[0,1], Section 4.6.4 [ref-iou-tou], page 104.
11.3.158  portray_message/2

Synopsis

:- multifile user:portray_message/2.

user:portray_message(+Severity, +MessageTerm)

Called by print_message/2 before processing the message. If this succeeds, then it is assumed that the message has been processed and nothing further is done.

Arguments

Severity one of [informational,warning,error,help,silent]

MessageTerm term

Exceptions

An exception raised by this predicate causes an error message to be printed and then the original message is printed using the default message text and formatting.

See Also

Section 4.16 [ref-msg], page 212.
11.3.159 predicate_property/2

Synopsis

predicate_property(?Callable, ?PredProperty)

Unifies PredProperty with a predicate property of an existing predicate, and Callable with the most general term that corresponds to that predicate.

Arguments

:Callable callable
   The skeletal specification of a loaded predicate.

PredProperty
   term
   The various properties associated with Callable. Each loaded predicate will have one or more of the properties:
   
   • one of the atoms built_in (for built-in predicates) or compiled or interpreted (for user defined predicates) or fd_constraint for FD predicates see Section 10.10.10 [Defining Primitive Constraints], page 485.
   • the atom dynamic for predicates that have been declared dynamic (see Section 4.3.4.2 [Dynamic Declarations], page 84),
   • the atom multifile for predicates that have been declared multifile (see Section 4.3.4.1 [Multifile Declarations], page 83),
   • the atom volatile for predicates that have been declared volatile (see Section 4.3.4.3 [Volatile Declarations], page 84),
   • the atom jittable for predicates that are amenable to JIT compilation,
   • the atom jitted for predicates that have been JIT compiled,
   • one or more terms (block Term) for predicates that have block declarations (see Section 4.3.4.5 [Block Declarations], page 84),
   • the atom exported or terms imported_from(ModuleFrom) for predicates exported or imported from modules (see Section 4.11 [ref-mod], page 161),
   • the term (meta_predicate Term) for predicates that have meta-predicate declarations (see Section 4.11.16 [ref-mod-met], page 171).

Description

If Callable is instantiated, then predicate_property/2 successively unifies PredProperty with the various properties associated with Callable.

If PredProperty is bound to a valid predicate property, then predicate_property/2 successively unifies Callable with the skeletal specifications of all loaded predicates having PredProperty.

If Callable is not a loaded predicate or PredProperty is not a valid predicate property, then the call fails.

If both arguments are unbound, then predicate_property/2 can be used to backtrack through all currently defined predicates and their corresponding properties.
Examples

- Predicates acquire properties when they are defined:

  ```prolog
  | ?- [user].
  | :-( dynamic p/1.
  | p(a).
  | end_of_file.
  \% user compiled 0.117 sec 296 bytes
  yes
  | ?- predicate_property(p(_), Property).

  Property = dynamic ;

  Property = interpreted ;

- To backtrack through all the predicates $P$ imported into module $m$ from any module:

  ```prolog
  | ?- predicate_property(m:P, imported_from(_)).
  ```

- To backtrack through all the predicates $P$ imported into module $m_1$ from module $m_2$:

  ```prolog
  | ?- predicate_property(m1:P, imported_from(m2)).
  ```

- To backtrack through all the predicates $P$ exported by module $m$:

  ```prolog
  | ?- predicate_property(m:P, exported).
  ```

- A variable can also be used in place of a module atom to find the names of modules having a predicate and property association:

  ```prolog
  | ?- predicate_property(M:f, imported_from(m1)).
  ```

  will return all modules $M$ that import $f/0$ from $m_1$.

Exceptions

None.

See Also

Section 4.9.1 [ref-lps-ove], page 135.
11.3.160  print/[1,2]  

Synopsis

print(+Stream, +Term)

print(+Term)

W...
11.3.161 print_coverage/[0,1]  

Synopsis

print_coverage

Prints the coverage data accumulated so far, to the current output stream, in a hierarchical format.

print_coverage(+Data)

Prints the coverage data Data, to the current output stream, in a hierarchical format. Data should be of type list of coverage_pair; see coverage_data/1.

Arguments

Data list of coverage_pair

Description

The output is formatted hierarchically into blocks of lines, one block per source file. A typical block looks like:

```
/home/matsc/tmp/primes.pl
  user:integers/3
    clause 1
      5:  100
      8:  99
    clause 2
      9:  1
  user:primes/2
    clause 1
      1:  1
      2:  1
      3:  *1
  user:remove/3
    clause 1
      16: 436
    clause 2
      17: 411
      20: *337
    clause 3
      21: 74
      22: *74
  user:sift/2
    clause 1
      11: 1
    clause 2
      12: 25
      13: *25
      14: *25
```
This block lists all the coverage sites for the given file. They are distributed over 4 predicates, 8 clauses, and 16 active lines of code. The coverage site on line 8 was hit 99 times. The coverage site on line 20 was hit 337 times, making at least one nondet call. And so on.

The variant `print_coverage/1` is useful e.g. if you want to somehow filter the execution coverage computed by `coverage_data/1` before printing it.

**Exceptions**

None.

**See Also**

Section 9.3 [Coverage Analysis], page 356. The collected coverage information can be presented by the SICStus Prolog IDE, SPIDER (see Section 3.11 [SPIDER], page 29). The Emacs interface also has commands for code coverage highlighting of the current buffer (C-c C-o, or use the Prolog menu; see Section 3.12.3 [Usage], page 34).
11.3.162 print_message/2

Synopsis

\texttt{print_message(+Severity, +MessageTerm)}

Print a \textit{Message} of a given \textit{Severity}. The behavior can be customized using the hooks \texttt{user:portray_message/2}, \texttt{user:generate_message_hook/3} and \texttt{user:message_hook/3}.

Arguments

\textit{Severity} \quad \texttt{atom}, must be nonvar

Unless the default system portrayal is overridden with \texttt{user:message_hook/3}, \textit{Severity} must be one of:

<table>
<thead>
<tr>
<th>Value</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>informational</td>
<td><code>%</code></td>
</tr>
<tr>
<td>warning</td>
<td><code>*</code></td>
</tr>
<tr>
<td>error</td>
<td><code>!</code></td>
</tr>
<tr>
<td>help</td>
<td></td>
</tr>
<tr>
<td>query</td>
<td></td>
</tr>
<tr>
<td>silent</td>
<td>no prefix</td>
</tr>
</tbody>
</table>

\textit{MessageTerm} \quad \texttt{term}

Description

First \texttt{print_message/2} calls \texttt{user:portray_message/2} with the same arguments. If this does not succeed, then the message is processed in the following phases:

- Message generation phase: the abstract message term \textit{Message} is formatted, i.e. converted to a format-command list. First the hook predicate \texttt{user:generate_message_hook/3} is tried, then if it does not succeed, then '\SU_messages':generate_message/3 is called. The latter predicate is defined in terms of definite clause grammars in \texttt{library('SU\_messages')}). If that also does not succeed, then the built-in default conversion is used, which gives the following result:

\['\text{'\texttt{q'}-}\left[\text{\texttt{Message}},\text{\texttt{nl}}\right]\]

- Line splitting transformation: the format-command list is converted to format-command lines—the list is broken up into a list of lists, each list containing format-commands for one line.

- Message printing phase: The text of the message (format-command lines generated in the previous stage) is printed. First the hook predicate \texttt{user:message_hook/3} is tried, then, if it does not succeed, then the built-in predicate \texttt{print_message_lines/3} is called for the \texttt{user_error} stream.

An unhandled exception message \textit{E} calls \texttt{print_message(error, E)} before returning to the top level. The convention is that an error message is the result of an unhandled exception.
Thus, an error message should only be printed if `raise_exception/1` does not find a handler and unwinds to the top level.

All messages from the system are printed using this predicate. Means of intercepting these messages before they are printed are provided.

`print_message/2` always prints to `user_error`. Messages can be redirected to other streams using `user:message_hook/3` and `print_message_lines/3`.

Silent messages do not get translated or printed, but they can be intercepted with `user:portray_message/2` and `user:message_hook/3`.

**Exceptions**

- `instantiation_error`
- `type_error`
- `domain_error`
  
  in Severity

**See Also**

Section 4.16 [ref-msg], page 212.
11.3.163 print_message_lines/3

Synopsis

print_message_lines(+Stream, +Severity, +Lines)

Print the Lines to Stream, preceding each line with a prefix corresponding to Severity.

Arguments

Stream stream_object, must be ground
    Any valid output stream.
Severity one of [query, help, informational, warning, error, silent, term]
Lines list of list of pair
    Must be of the form [Line1, Line2, ...], where each Linei must be of the form [Control_1-Args_1, Control_2-Args_2, ...].

Description

If Severity is a valid severity, then the prefix will be as described for print_message/2, otherwise Severity itself will be used as the prefix. If Severity is query, then no newline is written after the last line (otherwise, a newline is written).

This predicate is intended to be used in conjunction with user:message_hook/3. After a message is intercepted using user:message_hook/3, this command is used to print the lines. If the hook has not been defined, then the arguments are those provided by the system.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

\[\text{instantiation\_error}\]
\[\text{type\_error}\]
\[\text{in Lines.}\]

Examples

Suppose you want to intercept messages and force them to go to a different stream:

\[
\text{user:message\_hook}(\text{Severity}, \text{Message}, \text{Lines}):-
\]
\[
\text{my\_stream(MyStream),}
\]
\[
\text{print\_message\_lines(MyStream, Severity, Lines).}
\]

See Also

Section 4.16 [ref-msg], page 212.
11.3.164 print_profile/[0,1]  

Synopsis

print_profile

Prints the profiling data accumulated so far, to the current output stream, in a format similar to gprof(1).

print_profile(+Data)

Prints the profiling data Data, to the current output stream, in a format similar to gprof(1). Data should be of type list of profile_pair; see profile_data/1.

Arguments

Data       list of profile_pair

Description

The output is formatted into blocks of lines. There is one block per predicate with profiling data. A typical block looks like:

```
6667/11582 user:extract_index_2/5
4915/11582 user:safe_insns/5
174446 21862 11582 user:safe_insns/5
*10280/37221 user:safe_insn/1
4915/11582 user:safe_insns/5
```

This block concerns user:safe_insns/5. We are told that 174446 virtual instructions were executed and 21862 choicepoints were accessed, and that it was called 11582 times. There are two callers: user:extract_index_2/5 and user:safe_insns/5 itself, which called user:safe_insns/5 6667 and 4915 times respectively. Finally user:safe_insns/5 accounts for 10280 out of the 37221 calls to user:safe_insn/1 and, as we already know, for 4915 out of the 11582 calls to itself. The ‘*’ in front of 10280 tells us that for at least one of the 10280 calls, user:safe_insn/1 left a choicepoint behind, which could be a case of unwanted nondeterminacy; see Section 9.7 [The Determinacy Checker], page 363. If user:safe_insn/1 had been declared nondet (see Section 10.19 [lib-is_directives], page 565), ‘%’ would have been used instead of ‘*’. If it had been declared det or semidet, ‘!’ would have been used.

The variant print_profile/1 is useful e.g. if you want to somehow filter the execution profile computed by profile_data/1 before printing it.

Exceptions

None.

See Also

Section 9.2 [Execution Profiling], page 355.
### 11.3.165 profile_data/1

**Synopsis**

`profile_data(-Data)`  
Since release 4.2

*Data* is the profiling data accumulated so far.

**Arguments**

*Data*  
*list of profile_pair*

where:

- `profile_pair`  
  ::= `caller-profile_info`

- `profile_info`  
  ::= `counter(list of callee_pair, insns, chpts, tagged_calls)`

- `insns`  
  ::= `integer` \{virtual instructions executed\}

- `chpts`  
  ::= `integer` \{choicepoints accessed\}

- `callee_pair`  
  ::= `callee-tagged_calls`

- `tagged_calls`  
  ::= `det(calls)` \{all calls were determinate\}

  | `nondet(calls)` \{not all calls were determinate\}

- `calls`  
  ::= `integer` \{number of calls\}

- `caller`  
  ::= `pred_spec`

- `callee`  
  ::= `pred_spec`

**Description**

The profiling data accumulated so far is collected into a term of type *list of profile_pair* and unified with *Data*.

**Please note:** The number of instructions and choicepoints are not counted for interpreted code, so `insns` and `chpts` will be 0 for such predicates.

**Please note:** In a *list of callee_pair*, `callee` is not necessarily unique. This happens if the given `caller` code contains more than one call to `callee`.

**Please note:** The `calls` of a `profile_info` can be greater than the total `calls` of its *list of callee_pair*. This happens e.g. if `caller` occurred in a metacall context.

**Exceptions**

None.

**See Also**

Section 9.2 [Execution Profiling], page 355.
11.3.166 profile_reset/0

Synopsis

profile_reset

Resets all profiling data.

Exceptions

None.

See Also

Section 9.2 [Execution Profiling], page 355.
11.3.167 prolog_flag/[2,3]

Synopsis

prolog_flag(?FlagName, ?Value)

FlagName is a flag, which currently is set to Value.

prolog_flag(+FlagName, -OldValue, +NewValue)

Unifies the current value of FlagName with OldValue and then sets the value of the flag to NewValue. The available Prolog flags are listed in Section 4.9.4 [ref-lps-flg], page 136.

Arguments

FlagName atom, must be nonvar and a legal flag in prolog_flag/3
Value term
OldValue term
NewValue term, must be nonvar and belong to proper type/domain

Description

To inspect the value of a flag without changing it, use prolog_flag/2 or the following idiom, where FlagName is bound to one of the valid flags above.

| ?- prolog_flag(FlagName, Value, Value). |

Use prolog_flag/2 to query and set_prolog_flag/2 or prolog_flag/3 to set values.

prolog_flag/3 can be used to save flag values so that one can return a flag to its previous state. For example:

...  
prolog_flag(debugging,Old,on), % Save in Old and set  
...  
prolog_flag(debugging,_,Old), % Restore from Old  
...

Backtracking

prolog_flag/2 enumerates all valid flagnames of a given current value, or all pairs of flags and their current values.

Exceptions

instantiation_error

In prolog_flag/3, FlagName unbound, or NewValue unbound and not identical to OldValue.

type_error

FlagName is not an atom.
domain_error
In prolog_flag/3, FlagName bound to an atom that does not represent a supported flag, or NewValue bound to a term that does not represent a valid value for FlagName.

permission_error
In prolog_flag/3, NewValue not identical to OldValue for a read-only flag.

Examples

| ?- prolog_flag(X,Y).  
  X = bounded,  
  Y = false ? RET  
  yes

| ?- prolog_flag(X,Y,Y).  
  ! Instantiation error in argument 1 of prolog_flag/3  
  ! goal: prolog_flag(_94,_95,_95)

| ?- prolog_flag(source_info,X,X).  
  X = on ? RET  
  yes

See Also

current_prolog_flag/2, set_prolog_flag/2, Section 4.9.4 [ref-lps-flg], page 136.
11.3.168 prolog_load_context/2

Synopsis

prolog_load_context(?Key, ?Value)

Finds out the context of the current load. The available context keys are described in Section 4.9.5 [ref-lps-lco], page 143.

Arguments

Key atom
Value term

Description

You can call prolog_load_context/2 from an embedded command or from term_expansion/6 to find out the context of the current load. If called outside the context of a load, then it simply fails.

Backtracking

Can be used to backtrack through all keys and values.

Exceptions

None.

See Also

load_files/[2,3], Section 4.9.5 [ref-lps-lco], page 143.
11.3.169 prompt/2

Synopsis

prompt(-OldPrompt, +NewPrompt)

Queries or changes the prompt string of the current input stream or an input stream Stream.

Arguments

OldPrompt
  atom
  The old prompt atom.

NewPrompt
  atom, must be nonvar
  The new prompt atom.

Description

A prompt atom is a sequence of characters that indicates the Prolog system is waiting for input when a “Read” or “Get” predicate is called. If an input stream connected to a terminal is waiting for input at the beginning of a line (at line position 0), then the prompt atom will be printed through an output stream associated with the same terminal.

Prolog sets the prompt to ‘|’: ’ for every new top-level query. This is the prompt that can be changed by invoking prompt/2.

Unlike state changes such as those implemented as prolog flags, the scope of a prompt change is a goal typed at the toplevel. Therefore, the change is in force only until returning to the toplevel (prompt = ‘| ?- ’).

To query the current prompt atom, OldPrompt and NewPrompt should be the same unbound variable.

To set the prompt, NewPrompt should be an instantiated atom.

The “Load” predicates change the prompt during the time operations are performed: If a built-in loading predicate is performed on user (such as compile(user), etc.), then the prompt is set to ‘| ’. This prompt is not affected by prompt/2.

Exceptions

instantiation_error

type_error
  NewPrompt is not an atom

See Also

Section 4.6.3.2 [ref-iou-tin-cpr], page 104.
**11.3.170 public/1**

*declaration*

**Synopsis**

`:- public +Term`

Currently a dummy declaration.

**Arguments**

`:Term    term`

**Exceptions**

Exceptions in the context of loading code are printed as error messages.

- `context_error`
  Declaration appeared in a goal.

- `permission_error`
  Declaration appeared as a clause.

**See Also**

Section 4.3.4.8 [Public Declarations], page 86.
11.3.171 put_byte/[1,2]  

Synopsis

put_byte(+Byte)

put_byte(+Stream, +Byte)

Writes the byte Byte to Stream.

Arguments

Stream  stream_object, must be ground
        A valid output binary stream, defaults to the current output stream.

Byte    byte, must be nonvar

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

instantiation_error

type_error
        Byte is not a byte.

permission_error
        There is an error in the bottom layer of write function of the stream.

type_error
        Byte is not a byte

See Also

Section 4.6.6 [ref-iou-cou], page 108.
11.3.172 put_char/[1,2]  

Synopsis

put_char(+Char)

put_char(+Stream, +Char)

The char Char is written to Stream.

Arguments

Stream stream_object, must be ground
   A valid output text stream, defaults to the current output stream.

Char char, must be nonvar

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

type_error
   Char is not a char

permission_error
   There is an error in the bottom layer of write function of the stream.

See Also

Section 4.6.6 [ref-iou-cou], page 108.
11.3.173 put_code/[1,2]  

Synopsis

put_code(+Code)

put_code(+Stream, +Code)

The code Code is written to the stream Stream.

Arguments

Stream  stream_object, must be ground

   A valid output text stream, defaults to the current output stream.

Code  code, must be nonvar

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

instantiation_error

type_error

   Code is not an integer

permission_error

   There is an error in the bottom layer of write function of the stream.

representation_error

   Code is not a code

See Also

Section 4.6.6 [ref-iou-cou], page 108.
11.3.174 query_abbreviation/3

Synopsis

:- multifile 'SU_messages':query_abbreviation/3.

'SU_messages':query_abbreviation(+QueryClass, -Prompt, -Pairs)

A way to specify one letter abbreviations for responses to queries from the Prolog System.

Arguments

QueryClass

atom
The query class being defined.

Prompt

atom
The prompt to be used, typically indicating appropriate abbreviations.

Pairs

list of pair
A list of word-abbreviation pairs, defining the characters accepted and the corresponding abstract answers.

Description

This predicate defines a query class with the given prompt, the line input method, the char(Pairs) map method and help_query failure mode. The predicate is actually implemented by the first clause of 'SU_messages':query_class/5:

    query_class(QueryClass, Prompt, line, char(Pairs), help_query) :-
        query_abbreviation(QueryClass, Prompt, Pairs), !.

Prolog only asks for keyboard input in a few different ways. These are enumerated in the clauses for 'SU_messages':query_abbreviation/3. These clauses specify valid abbreviations for a given key word. For example,

    query_abbreviation(yes_or_no, ' (y or n) ', [yes=-[-1,0'y], no="n"]): - !.

a French translator might decide that the letters 'O' and 'o' are reasonable abbreviations for 'oui' (yes), and therefore write

    query_abbreviation(yes_or_no, ' (y or n) ', [yes=-[-1,0'o,0'o], no="n"]): - !.

Exceptions

ask_query/4 checks the output arguments.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.175 query_class/5

Synopsis

:- multifile 'SU_messages':query_class/5.

'SU_messages':query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)

Access the parameters of a given QueryClass.

Arguments

QueryClass

term

Determines the allowed values for the atom Answer.

Prompt

atom

The prompt to display at the terminal.

InputMethod

term

A ground term, which specifies how to obtain input from the user.

MapMethod

term

A ground term, which specifies how to process the input to get the abstract answer to the query.

FailureMode

term

An atom determining what to print in case of an input error, before re-querying the user. Possible values are:

- help_query - print the help message and print the query text again;
- help - only print the help message;
- query - only print the query text;
- none - do not print anything.

Description

For the list of default input- and map methods, see the “Default Input Methods” and “Default Map Methods” subsections in Section 4.16.3 [Query Processing], page 216.

Exceptions

ask_query/4 checks the output arguments.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.176  query_class_hook/5

Synopsis

:- multifile user:query_class_hook/5.

user:query_class_hook(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)

Provides the user with a method of overriding the call to 'SU_messages':query_class/5 in the preparation phase of query processing. This way the default query class characteristics can be changed.

Arguments

QueryClass  
  term
  Determines the allowed values for the atom Answer.

Prompt  
  atom
  The prompt to display at the terminal.

InputMethod  
  term
  The input method to use.

MapMethod  
  term
  The map method to use.

FailureMode  
  term
  The failure mode to use.

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.177 query_hook/6

Synopsis

::- multifile 'SU_messages':query_hook/6.
'SU_messages':query_hook(+QueryClass, +Prompt, +PromptLines, +Help, +HelpLines, -Answer)

Provides a method of overriding Prolog’s default keyboard based input requests.

Arguments

QueryClass  term
Determines the allowed values for the atom Answer.

Prompt  term
A message term.

PromptLines  list of pair
The message generated from the Prompt message term.

Help  term
A message term.

HelpLines  list of pair
The message generated from the Help message term.

Answer  term
See QueryClass

Description

This provides a way of overriding Prolog’s default method of interaction. If this predicate fails, then Prolog’s default method of interaction is invoked.

The default method first prints out the prompt, then if the response from the user is not one of the allowed values, then the help message is printed.

It is useful to compare this predicate to user:message_hook/3, since this explains how you might use the Prompt, PromptLines, Help, HelpLines.

Exceptions

An exception raised by this predicate causes an error message to be printed and then the default method of interaction is invoked. In other words, exceptions are treated as failures.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.178 query_input/3

Synopsis

:- multifile 'SU_messages':query_input/3.

'SU_messages':query_input(+InputMethod, +Prompt, -RawInput)

Implements the input phase of query processing. The user is prompted with Prompt, input is read according to InputMethod, and the result is returned in RawInput.

Arguments

InputMethod term
The input method to use.

Prompt atom
The prompt to display at the terminal.

RawInput term

Exceptions

ask_query/4 checks the output arguments.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.179  query_input_hook/3

Synopsis

:- multifile user:query_input_hook/3.

user:query_input_hook(+InputMethod, +Prompt, -RawInput)

Provides the user with a method of overriding the call to 'SU_messages':query_input/3 in the input phase of query processing. This way the implementation of the default input methods can be changed.

Arguments

*InputMethod*  
*term*  
The input method to use.

*Prompt*  
*atom*  
The prompt to display at the terminal.

*RawInput*  
*term*  

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.180  query_map/4

Synopsis

:- multifile 'SU_messages':query_map/4.

'SU_messages':query_map(+MapMethod, +RawInput, -Result, -Answer)

Implements the mapping phase of query processing. The RawInput, received from query_input/3, is mapped to the abstract answer term Answer.

Arguments

MapMethod
term
The map method to use.

RawInput  atom
As received from query_input/3.

Result one of [success, failure, failure(Warning)]
Result of conversion.

Answer one of [success, failure, failure(Warning)]
Abstract answer term.

Exceptions

ask_query/4 checks the output arguments.

See Also
Section 4.16.3 [Query Processing], page 216.
11.3.181 query_map_hook/4

Synopsis

:- multifile user:query_map_hook/4.

user:query_map_hook(+MapMethod, +RawInput, -Result, -Answer)

Provides the user with a method of overriding the call to 'SU_messages':query_map/4 in the map phase of query processing. This way the implementation of the default map methods can be changed.

Arguments

MapMethod  term
The map method to use.

RawInput  atom
As received from query_input/3.

Result  one of [success,failure,failure(Warning)]
Result of conversion.

Answer  one of [success,failure,failure(Warning)]
Abstract answer term.

Exceptions

All error handling is done by the predicates extended by this hook.

See Also

Section 4.16.3 [Query Processing], page 216.
11.3.182 raise_exception/1

Synopsis
raise_exception(+Exception)

If Exception matches one of the SICStus error terms listed in Section 4.15.4 [ref-ere-err], page 200, then the corresponding error term error(ISO_Error, SICStus_Error) is thrown. Otherwise, Exception is thrown as is.

New code should prefer the ISO-standard conformant throw/1.

Please note: For backward compatibility reasons, some Exception terms are automatically transformed into their corresponding ISO error/2 terms. The standard conformant throw/1 does not perform any such transformation, and is generally preferable.

Arguments
Exception term, must be nonvar

Exceptions
instantiation_error
Exception is unbound.

See Also
Section 4.15 [ref-ere], page 197, throw/1.
11.3.183 read/[1,2]  ISO

Synopsis

read(-Term)
read(+Stream, -Term)

Reads the next term from Stream and unifies it with Term. Same as:
read_term(Term, [])
read_term(Stream, Term, [])

Arguments

Stream  stream_object, must be ground
         A valid Prolog input stream, defaults to the current input stream.
Term     term
         The term to be read.

Description

Term must be followed by a full stop. The full stop is removed from the input stream and
is not a part of the term that is read. The term is read with respect to current operator
declarations.

Does not finish until the full stop is encountered. Thus, if you type at top level

| ?- read(X)

then you will keep getting prompts (first ‘|: ’, and five spaces thereafter) every time you
type RET, but nothing else will happen, whatever you type, until you type a full stop.

If a syntax error is encountered, then the action taken depends on the current value of the
syntax_errors Prolog flag.

If the end of the current input stream has been reached, then Term will be unified with
the atom end_of_file. Further calls to read/[1,2] for the same stream will then raise an
exception, unless the stream is connected to the terminal. The characters read are subject
to character-conversion.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:
syntax_error
         A syntax error was found.

Examples
See Also

read_term/[2,3], char_conversion/2, Section 4.6.3.1 [ref-iou-tin-trm], page 103, Section 4.9.4 [ref-lps-flg], page 136.
11.3.184 read_line/[1,2]

Synopsis
read_line(-Line)

read_line(+Stream, -Line)

Reads one line of input from Stream, and unifies the codes with Line. On end of file, Line is unified with end_of_file.

Arguments
Stream stream_object, must be ground
A valid input text stream, defaults to the current input stream.
Line codes or one of [end_of_file]

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
Trying to read beyond end of Stream

See Also
at_end_of_line/[0,1], Section 4.6.5 [ref-iou-cin], page 107.
11.3.185 read_term/[2,3]  ISO

Synopsis

read_term(-Term, +Options)
read_term(+Stream, -Term, +Options)

Read a term from Stream, optionally returning extra information about the term.

Arguments

Stream  stream_object, must be ground
       A valid Prolog input stream, defaults to the current input stream.

Term     term
       The term that is read.

Options  list of term, must be ground, except Vars, Names, and Layout as described below.
       A list of zero or more of the following:

syntax_errors(Val)
       Controls what action to take on syntax errors. Val must be one of the values allowed for the syntax_errors Prolog flag. The default is set by that flag. See Section 4.9.4 [ref-lps-flg], page 136.

variables(Vars)
       Vars is bound to the list of variables in the term input, in left-to-right traversal order.

variable_names(Names)
       Names is bound to a list of Name=Var pairs, where each Name is an atom indicating the name of a non-anonymous variable in the term, and Var is the corresponding variable. The elements of the list are in the same order as in Term, i.e. in left-to-right traversal order.

singletons(Names)
       Names is bound to a list of Name=Var pairs, one for each non-anonymous variable appearing only once in the term. The elements of the list are in the same order as in Term, i.e. in left-to-right traversal order.

cycles(Boolean)
       Boolean must be true or false. If selected, then any occurrences of @/2 in the term read in are replaced by the potentially cyclic terms they denote as described below. Otherwise (the default), Term is just unified with the term read in.

The notation used when this option is selected is @([Template,Substitution]) where Substitution is a list of Var=Term pairs where the Var occurs in Template or in one of the Terms. This notation stands for
the instance of Template obtained by binding each Var to its corresponding Term. The purpose of this notation is to provide a finite printed representation of cyclic terms. This notation is not used by default, and \( \text{\#}/2 \) has no special meaning except in this context.

Terms can be written in this notation using \texttt{write_term/[2,3]} (see Section 11.3.253 [mpg-ref-write_term], page 1249).

\texttt{layout(Layout)}

\texttt{Layout} is bound to a \textit{layout term} corresponding to \textit{Term} (see Chapter 2 [Glossary], page 7).

\texttt{consume_layout(Boolean)}

\texttt{Boolean} must be \textit{true} or \textit{false}. If this option is \textit{true}, then \texttt{read_term/[2,3]} will consume the \textit{layout-text-item} that follows the terminating ‘.’ (this \textit{layout-text-item} can either be a \textit{layout-char} or a \textit{comment} starting with a ‘%’). If the option is \textit{false}, then the \textit{layout-text-item} will remain in the input stream, so that subsequent character input predicates will see it. The default of the \texttt{consume_layout} option is \textit{false}.

\textbf{Description}

The characters read are subject to character-conversion.

\textbf{Exceptions}

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

\texttt{syntax_error}

A syntax error was found.

\texttt{instantiation_error}

\texttt{type_error}

\texttt{domain_error}

An illegal option was specified.

\textbf{Examples}

\begin{verbatim}
| ?- read_term(T, [variable_names(L)]).
|:
| append([\_U|\_X], \_Y, [\_U|\_Z]) :- append(X,Y,Z).
| L = ['U'=\_A,'X'=\_B,'Y'=\_C,'Z'=\_D],
| T = (append([\_A|\_B], \_C, [\_A|\_D]) :- append(\_B, \_C, \_D))
\end{verbatim}
| ?- read_term(T, [layout(L), variable_names(Va), singletons(S)]).  |
|: [  |
|   foo(X),  |
|   X = Y  |
| ].  |

L = [35,[36,36],[36,[37,37,37],38]],
S = ['Y'=\_A],
T = [foo(_B),_B=_A],
Va = ['X'=\_B,'Y'=\_A]

| ?- read_term(T, [consume_layout(false)]), get_code(C).  |
|: 1.  |

C = 10,
T = 1

| ?- read_term(T, [consume_layout(true)]), get_code(C).  |
|: 1.  |
|: a  |

C = 97,
T = 1

See Also
read/[1,2], char_conversion/2, Section 4.6.3.1 [ref-iou-tin-trm], page 103, Section 4.9.4 [ref-lps-flg], page 136.
11.3.186  reconsult/1

Synopsis

reconsult(+Files)

same as:

consult(Files)

Arguments

:Files  file_spec or list of file_spec, must be ground

A file specification or a list of file specifications; extensions optional.

Exceptions

See load_files/[2,3].

See Also

Section 4.3.2 [ref-lod-lod], page 80.
11.3.187 recorda/3

Synopsis

`recorda(+Key, +Term, -Ref)`

records the `Term` in the internal database as the first item for the principal functor of `Key`; a database reference to the newly-recorded term is returned in `Ref`.

Arguments

- `Key`: term, must be nonvar
- `Term`: term
- `Ref`: db_reference, must be var

Description

If `Key` is a compound term, then only its principal functor is significant. That is, `foo(1)` represents the same key as `foo(n)`.

Any uninstantiated variables in the `Term` will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

Exceptions

- `instantiation_error`
  - `Key` is not instantiated
- `uninstantiation_error`
  - `Ref` is not a variable

Examples

See Also

Section 4.12.8 [ref-mdb-idb], page 183.
11.3.188  recorded/3

Synopsis

recorded(-Key, -Term, +Ref)

recorded(?Key, ?Term, ?Ref)

Searches the internal database for a term recorded under the principal functor of Key that unifies with Term, and whose database reference unifies with Ref.

Arguments

<table>
<thead>
<tr>
<th>Key</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>term</td>
</tr>
<tr>
<td>Ref</td>
<td>db_reference</td>
</tr>
</tbody>
</table>

Description

If Ref is instantiated, then Key and Term are unified with the key and term associated with Ref. Otherwise, if Key is a compound term, then only its principal functor is significant. That is, foo(1) represents the same key as foo(n).

Backtracking

Can be used to backtrack through all the matching terms recorded under the specified key.

Exceptions

<table>
<thead>
<tr>
<th>type_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref is not a database reference</td>
</tr>
</tbody>
</table>

Examples

See Also

Section 4.12.8 [ref-mdb-idb], page 183.
11.3.189  recordz/3

Synopsis

recordz(+Key, +Term, -Ref)

records the Term in the internal database as the last item for the principal functor of Key; a database reference to the newly-recorded term is returned in Ref.

Arguments

Key  term, must be nonvar
Term  term
Ref  db_reference, must be var

Description

If Key is a compound term, then only its principal functor is significant. That is, foo(1) represents the same key as foo(n).

Any uninstantiated variables in the Term will be replaced by brand new, unattributed variables (see Section 4.2.4 [ref-sem-sec], page 74).

Exceptions

instantiation_error
    Key is not instantiated

uninstantiation_error
    Ref is not a variable

Examples

See Also

Section 4.12.8 [ref-mdb-idb], page 183.
11.3.190 remove_breakpoints/1

Synopsis

remove_breakpoints(+BIDs)

Removes the breakpoints specified by BIDs.

Arguments

BIDs list of integer, must be ground
Breakpoint identifiers.

Exceptions

instantiation_error
type_error in BIDs

See Also

Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 260, Section 5.7 [Breakpoint Predicates], page 272.
11.3.191 repeat/0

Synopsis

repeat

Succeeds immediately when called and whenever reentered by backtracking.

Description

Generally used to simulate the looping constructs found in traditional procedural languages.

Generates an infinite sequence of backtracking choices. In sensible code, repeat/0 is hardly ever used except in repeat loops. A repeat loop has the structure

\[
\text{Head} :- \\
\quad \ldots \\
\quad \text{save}_\text{state}(\text{OldState}), \\
\quad \text{repeat}, \\
\quad \text{generate}(\text{Datum}), \\
\quad \text{action}(\text{Datum}), \\
\quad \text{test}(\text{Datum}), \\
\quad !, \\
\quad \text{restore}_\text{state}(\text{OldState}), \\
\quad \ldots
\]

The purpose is to repeatedly perform some action on elements that are somehow generated, e.g. by reading them from a stream, until some test becomes true. Usually, generate, action, and test are all determinate. Repeat loops cannot contribute to the logic of the program. They are only meaningful if the action involves side effects.

The easiest way to understand the effect of repeat/0 is to think of failures as “bouncing” back off them causing re-execution of the later goals.

Repeat loops are not often needed; usually recursive procedure calls will lead to code that is easier to understand as well as more efficient. There are certain circumstances, however, in which repeat/0 will lead to greater efficiency. An important property of SICStus Prolog is that all runtime data is stored in stacks so that any storage that has been allocated during a proof of a goal is recovered immediately on backtracking through that goal. Thus, in the above example, any space allocated by any of the actions is very efficiently reclaimed. When an iterative construct is implemented using recursion, storage reclamation will only be done by the garbage collector.

Tips

In the most common use of repeat loops, each of the calls succeeds determinately. It can be confusing if calls sometimes fail, so that backtracking starts before the test is reached, or if calls are nondeterminate, so that backtracking does not always go right back to repeat/0.

Note that the repeat loop can only be useful if one or more of the actions involves a side effect — either a change to the data base (such as an assertion) or an I/O operation. Otherwise you would do the same thing each time around the loop (which would never terminate).
Backtracking
Succeeds repeatedly until backtracking is terminated by a cut or an exception.

Exceptions
None.

See Also
Section 4.2 [ref-sem], page 61.
11.3.192 restore/1

Synopsis

restore(+FileSpec)

Restores a saved state.

Arguments

FileSpec  file_spec, must be ground

The name of a saved state, ‘.sav’ extension optional.

Description

The system is returned to the program state previously saved to the file denoted by FileSpec with start-up goal Goal. restore/1 may succeed, fail or raise an exception depending on Goal.

Exceptions

instantiation_error

type_error

In FileSpec.

existence_error

The specified file does not exist. If the fileerrors Prolog flag is off, then the predicate fails instead of raising this exception.

permission_error

A specified file is not readable. If the fileerrors Prolog flag is off, then the predicate fails instead of raising this exception.

Examples

| ?- save_program(state, format('Restored!
',[])).
% /home/matsc/sicstus4/Bips/state.sav created in 0 msec
yes

| ?- restore(state).
% restoring /home/matsc/sicstus4/Bips/state.sav...
% /home/matsc/sicstus4/Bips/state.sav restored in 10 msec 16 bytes
Restored!
yes

See Also

save_program/[1,2], Section 3.10 [Saving], page 28, Section 4.4 [ref-sls], page 92, Section 4.4.2 [ref-sls-sst], page 93, Section 4.9.4 [ref-lps-flg], page 136.
11.3.193 retract/1

Synopsis
retract(+Clause)

Removes the first occurrence of dynamic clause Clause from module M.

Arguments
:Clause callable, must be nonvar
   A valid Prolog clause.

Description
retract/1 erases the first clause in the database that matches Clause. Clause is retracted
in module M if specified. Otherwise, Clause is retracted in the source module.

retract/1 is nondeterminate. If control backtracks into the call to retract/1, then suc-
cessive clauses matching Clause are erased. If and when no clauses match, then the call to
retract/1 fails.

If the predicate did not previously exist, then it is created as a dynamic predicate and
retract/1 fails.

Clause must be of one of the forms:
  • Head
  • Head :- Body
  • Module:Clause

where Head is of type callable and the principal functor of Head is the name of a dynamic
procedure. If specified, then Module must be an atom.

retract(Head) means retract the unit-clause Head. The exact same effect can be achieved
by retract((Head :- true)).

Body may be uninstantiated, in which case it will match any body. In the case of a unit-
clause it will be bound to true. Thus, for example,

    | ?- retract((foo(X) :- Body)), fail.

is guaranteed to retract all the clauses for foo/1, including any unit-clauses, providing of
course that foo/1 is dynamic.

Backtracking
Can be used to retract all matching clauses through backtracking.

Exceptions
instantiation_error
   Head (in Clause) or M is uninstantiated.
type_error
  Head is not a callable, or M is not an atom, or Body is not a valid clause body.

permission_error
  the procedure corresponding to Head is not dynamic

See Also
retractall/1, Section 4.12.5 [ref-mdb-rcd], page 179.
11.3.194 retractall/1

Synopsis
retractall(+Head)

Removes every clause in module $M$ whose head matches $Head$.

Arguments
:Head callable, must be nonvar
   Head of a Prolog clause.

Description
$Head$ must be instantiated to a term that looks like a call to a dynamic procedure. For example, to retract all the clauses of $foo/3$, you would write

$$\mid \ ?- \ retractall(foo(_,_,_)).$$

$Head$ may be preceded by a $M$: prefix, in which case the clauses are retracted from module $M$ instead of the calling module.

retractall/1 is useful for erasing all the clauses of a dynamic procedure without forgetting that it is dynamic; abolish/1 will not only erase all the clauses, but will also forget absolutely everything about the procedure. retractall/1 only erases the clauses. This is important if the procedure is called later on.

Since retractall/1 erases all the dynamic clauses whose heads match $Head$, it has no choices to make, and is determinate. If there are no such clauses, then it succeeds trivially. None of the variables in $Head$ will be instantiated by this command.

If the predicate did not previously exist, then it is created as a dynamic predicate and retractall/1 succeeds.

Exceptions

- instantiation_error
  $Head$ or $Module$ is uninstantiated.

- type_error
  $Head$ is not a callable.

- permission_error
  the procedure corresponding to $Head$ is not dynamic.

See Also
retract/1, Section 4.12.5 [ref-mdb-rcl], page 179.
11.3.195 save_files/2

Synopsis

save_files(+SourceFiles, +File)

Any code loaded from SourceFiles, as well as from any file included by them, is saved into File in PO format.

Arguments

SourceFiles

    file_spec or list of file_spec, must be ground

    A file specification or a list of file specifications; extensions optional.

File

    file_spec, must be ground

    A file specification, ‘.po’ extension optional.

Description

Any module declarations, predicates, multifile clauses, or directives encountered in SourceFiles, as well as from any file included by them, are saved in object format into the file denoted by File. Source file information as provided by source_file/[1,2] for the relevant predicates and modules is also saved.

File can later be loaded by load_files/[1,2], at which time any saved directives will be re-executed. If any of the SourceFiles declares a module, then FileSpec too will behave as a module file and export the predicates listed in the first module declaration encountered in SourceFiles. See Section 4.4 [ref-sls], page 92.

Exceptions

instantiation_error

    SourceFiles or File is not bound.

type_error

    SourceFiles or File is not a valid file specification.

permission_error

    File is not writable.

See Also

load_files/[1,2], Section 3.10 [Saving], page 28, Section 4.4 [ref-sls], page 92, Section 4.4.3 [ref-sls-ssl], page 94.
11.3.196 save_modules/2

Synopsis

\texttt{save\_modules(+Modules, +File)}

Saves all predicates in \texttt{Modules} in PO format to \texttt{File}.

Arguments

\begin{description}
\item[Modules] atom or list of atom, must be ground
An atom representing a current module, or a list of such atoms representing a list of modules.
\item[File] file\_spec, must be ground
A file specification, ‘.po’ extension optional.
\end{description}

Description

The module declarations, predicates, multifile clauses and initializations belonging to \texttt{Modules} are saved in object format into the file denoted by \texttt{File}. Source file information and embedded directives (except initializations) are not saved.

The PO file produced can be loaded using \texttt{load\_files/[1,2]}. When multiple modules are saved into a file, loading that file will import only the first of those modules into the module in which the load occurred.

Exceptions

\begin{description}
\item[instantiation\_error] \texttt{Modules} or \texttt{File} is not bound.
\item[type\_error] \texttt{Modules} is not a valid list of module names, or a single module name, or \texttt{File} is not a valid file specification.
\item[permission\_error] \texttt{File} is not writable.
\item[existence\_error] A given module is not a current module.
\end{description}

See Also

\texttt{load\_files/[1,2]}, Section 3.10 [Saving], page 28, Section 4.4 [ref-sls], page 92, Section 4.4.3 [ref-sls-ssl], page 94.
11.3.197 save_predicates/2

Synopsis

save_predicates(+PredSpecs, +File)

Saves all predicates in PredSpecs in PO format to File.

 Arguments

:PredSpecs
    pred_spec_tree
    A predicate specification, or a list of such.
File
    file_spec, must be ground
    A file specification, ‘.po’ extension optional.

Description

save_predicates/2 saves the current definitions of all the predicates specified by the list of predicate specifications in PO format into a file. The module of the predicates saved in the PO file is fixed, so it is not possible to save a predicate from any module foo, and reload it into module bar. Source file information and embedded directives are not saved. A typical use of this would be to take a snapshot of a table of dynamic facts.

The PO file that is written out can be loaded using load_files/[1,2].

Please note: if PredSpecs contains specifications for which no matching predicate can be found, then a warning is issued, and the file is written anyway. Also, no built-in predicates are saved.

Exceptions

instantiation_error
    PredSpecs is not instantiated enough, or File is not bound.
type_error
    PredSpecs is not a valid tree of predicate specifications, or File is not a valid file specification, or a Name is not an atom or an Arity is not an integer.
domain_error
    if an Arity is specified as an integer outside the range 0-255.
permission_error
    File is not writable.

See Also

load_files/[1,2], Section 3.10 [Saving], page 28, Section 4.4 [ref-sls], page 92, Section 4.4.3 [ref-sls-ssl], page 94.
11.3.198  save_program/[1,2]

Synopsis

save_program(+File)

save_program(+File, +Goal)

Saves the state of the current execution in object format to File. A goal, Goal, to be called upon execution/restoring of the saved state, may be specified.

Arguments

File file spec, must be ground
A file specification, .sav extension optional.

:Goal callable, must be nonvar
A goal, defaults to true.

Description

save_program/[1,2] creates a binary representation of all predicates in all modules existing in the system. However, it does not save the user’s pre-linked code. It also saves such states of the system as operator definitions, Prolog flags, debugging and advice state, initializations, and dependencies on foreign resources.

The resulting file can be restored using restore/1.

Any unbound variables in Goal with attributes or blocked goals attached to them will be replaced by plain, brand new variables. This is analogous to the way attributed variables are handled in terms that are written, copied, asserted, gathered as solutions to findall/3 and friends, or raised as exceptions. To retain the attributes, you can use copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 129).

Exceptions

instantiation_error
    File or Goal is not bound.

type_error
    File is not a valid file specification, or Goal is not a callable.

permission_error
    File is not writable.
Examples

?- save_program(state, format('Restored!\n',[])).
% /home/matsc/sicstus4/Bips/state.sav created in 0 msec
yes

?- restore(state).
% restoring /home/matsc/sicstus4/Bips/state.sav...
% /home/matsc/sicstus4/Bips/state.sav restored in 10 msec 16 bytes
Restored!
yes

See Also

restore/1, Section 3.10 [Saving], page 28, Section 4.4 [ref-sls], page 92, Section 4.4.2 [ref-sls-sst], page 93.
11.3.199 see/1

Synopsis
see(+FileOrStream) Makes file FileOrStream the current input stream.

Arguments
FileOrStream
file_spec or stream_object, must be ground

Description
If there is an open input stream associated with FileOrStream, and that stream was opened by see/1, then it is made the current input stream. Otherwise, the specified file is opened for input in text mode with default options and made the current input stream.

Different file names (that is, names that do not unify) represent different streams (even if they correspond to the same file). Therefore, assuming food and ./food represent the same file, the following sequence will open two streams, both connected to the same file.

    see(food)
    ...
    see('./food')

It is important to remember to close streams when you have finished with them. Use seen/0 or close/[1,2].

Exceptions
instantiation_error
FileOrStream is not instantiated enough.

existence_error
FileOrStream specifies a nonexisting file, and the fileerrors Prolog flag is on.

permission_error
FileOrStream is a stream not currently open for input, or FileOrStream specifies a file with insufficient access permission, and the fileerrors Prolog flag is on.

domain_error
FileOrStream is neither a file_spec nor a stream_object.

See Also
seen/0, open/[3,4], current_input/1, Section 4.6.7 [ref-iot-sfh], page 109, Section 4.9.4 [ref-lps-flg], page 136.
11.3.200 seeing/1

Synopsis

seeing(-FileOrStream)

Unifies FileOrStream with the current input stream or file.

Arguments

FileOrStream
  file_spec or stream_object

Description

Exactly the same as current_input(FileOrStream), except that FileOrStream will be unified with a filename if the current input stream was opened by see/1 (Section 4.6.7 [ref-iou-sfh], page 109).

Can be used to verify that FileNameOrStream is still the current input stream as follows:

% nonvar(FileNameOrStream),
see(FileNameOrStream),
...
seeing(FileNameOrStream)

If the current input stream has not been changed (or if changed, then restored), then the above sequence will succeed for all file names and all stream objects opened by open/[3,4]. However, it will fail for all stream objects opened by see/1 (since only filename access to streams opened by see/1 is supported). This includes the stream object user_input (since the standard input stream is assumed to be opened by see/1, and so seeing/1 would return user in this case).

If FileOrStream is instantiated to a value that is not the identifier of the current input stream, then seeing(FileOrStream) simply fails.

Can be followed by see/1 to ensure that a section of code leaves the current input unchanged:

% var(OldFileNameOrStream),
seeing(OldFileNameOrStream),
...
see(OldFileNameOrStream)

The above is analogous to its stream-object-based counterpart,

% var(OldStream),
current_input(OldStream),
...
set_input(OldStream)
Both of these sequences will always succeed regardless of whether the current input stream was opened by see/1 or open/[3,4].

Exceptions

None.

See Also

see/1, open/[3,4], current_input/1, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.201 seek/4

Synopsis

\texttt{seek(+Stream, +Offset, +Method, -NewLocation)}

Seeks to an arbitrary position in \textit{Stream}.

Arguments

\textit{Stream} \hspace{1em} stream\_object, must be ground

A valid Prolog stream.

\textit{Offset} \hspace{1em} integer, must be nonvar

The offset, in \textit{items}, to seek relative to the specified \textit{Method}. Items are bytes for binary streams, characters for text streams.

\textit{Method} \hspace{1em} one of [\textit{bof, current, eof}], must be nonvar

Where start seeking, one of the following:

\textit{bof} \hspace{1em} Seek from beginning of the file stream.

\textit{current} \hspace{1em} Seek from current position of the file stream.

\textit{eof} \hspace{1em} Seek from end of the file stream.

\textit{NewLocation} \hspace{1em} integer

The offset from beginning of the file after seeking operation.

Description

Sets the current position of the file stream \textit{Stream} to a new position according to \textit{Offset} and \textit{Method}. If \textit{Method} is:

\textit{bof} \hspace{1em} then the new position is set to \textit{Offset} \textit{items} from beginning of the file stream.

\textit{current} \hspace{1em} then the new position is \textit{Offset} plus the current position of \textit{Stream}.

\textit{eof} \hspace{1em} then the new position is \textit{Offset}, plus the current size of the stream.

Avoid using this \textit{Method}. Determining the size of the stream may be expensive or unsupported for some streams.

Positions and offsets are measured in \textit{items}, bytes for binary streams and characters for text streams. Note that there may not be any simple relationship between the number of characters read and the byte offset of a text file.

After applying this operation on a text stream, the line counts and line position aspects of the stream position of \textit{Stream} will be undefined.

The term “file” above is used even though the stream may be connected to other seekable objects that are not files, e.g. an in-memory buffer.
Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

instantiation_error
   Offset or Method is not instantiated.

type_error
   Stream is not a stream object, or Offset is not an integer, or Method is not an atom.

domain_error
   Method is not one of bof, current or eof, or the resulting position would refer to an unsupported location. Some streams support setting the position past the current end of the stream, in this case the stream is padded with zero bytes or characters as soon as an item is written to the new location.

permission_error
   Seeking was not possible. Common reasons include: the stream has not been opened with reposition(true), the stream is a text stream that does not implement seeking, or an I/O error happened during seek.

See Also
stream_position/2, set_stream_position/2, open/[3,4], byte_count/2, character_count/2, line_count/2, line_position/2, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.202 seen/0

Synopsis
seen

Closes the current input stream.

Description
Current input stream is set to be user_input; that is, the user’s terminal.

Always succeeds

Exceptions
None.

Examples

See Also
see/1, close/[1,2], current_input/1, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.203  set_input/1

Synopsis

set_input(+Stream)

makes Stream the current input stream.

Arguments

Stream  stream_object, must be ground
        A valid input stream.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

see/1, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.204 set_module/1

Synopsis

set_module(+ModuleName)

Changes the type-in module (see Section 4.11.8 [ref-mod-tyi], page 166) to ModuleName. Thus subsequent top-level goals use ModuleName as their source module.

Arguments

ModuleName

  atom, must be nonvar

  The name of a module.

Description

If ModuleName is not a current module, then a warning message is issued, but the type-in module is changed nonetheless.

Calling set_module/1 from a command embedded in a file that is being loaded does not affect the loading of clauses from that file. It only affects subsequent goals that are typed at top level.

Exceptions

instantiation_error

type_error

Examples

See Also

Section 4.11 [ref-mod], page 161, Section 4.11.8 [ref-mod-tyi], page 166.
11.3.205 set_output/1

Synopsis

set_output(+Stream)

makes Stream the current output stream.

Arguments

Stream stream_object, must be ground
A valid output stream.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

tell/1, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.206 set_prolog_flag/2

Synopsis

set_prolog_flag(+FlagName, +Value)

same as:

prolog_flag(FlagName, _, Value)

Arguments

FlagName atom, must be nonvar
Value term, must be nonvar and belong to proper type/domain

Exceptions

instantiation_error
  An argument is unbound.

type_error
  FlagName is not an atom, or Value has the wrong type.

domain_error
  FlagName is not a valid flag name, or Value is not a valid value for it.

permission_error
  The flag is read-only.

See Also

current_prolog_flag/2, prolog_flag/[2,3], Section 4.9.4 [ref-lps-flg], page 136.
11.3.207 set_stream_position/2  

**Synopsis**

set_stream_position(+Stream, +Position)

Sets the current position of `Stream` to `Position`.

**Arguments**

*Stream*  
stream_object, must be ground
An open stream.

*Position*  
term
Stream position object representing the current position of `Stream`.

**Description**

`set_stream_position/2` repositions the stream pointer, and also the other counts, such as byte, character, and line counts and line position. It may only be used on streams that have been opened with the open/4 option `reposition(true)`.

**Please note:** A stream position object is represented by a special Prolog term. The only safe way of obtaining such an object is via `stream_position/2` or `stream_property/2`. You should not try to construct, change, or rely on the form of this object. It may change in subsequent releases.

**Exceptions**

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

instantiation_error
domain_error

*Position* is not a valid stream position object.

**See Also**

`stream_position/2`, `stream_property/2`, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.208 setof/3

Synopsis

\texttt{setof(+Template, +Generator, -Set)}

Returns the non-empty set \textit{Set} of all instances of \textit{Template} such that \textit{Generator} is provable.

Arguments

\begin{itemize}
  \item \textit{Template} term
  \item \textit{Generator} callable, must be \texttt{nonvar}
    \begin{itemize}
      \item A goal to be proved as if by \texttt{call/1}.
    \end{itemize}
  \item \textit{Set} list of term
\end{itemize}

Description

\textit{Set} is a non-empty set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 4.8.8 [ref-lte-cte], page 130). If there are no instances of \textit{Template} such that \textit{Generator} is satisfied, then \texttt{setof/3} simply fails.

Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case \textit{Set} will only provide an imperfect representation of what is in reality an infinite set.

If \textit{Generator} is instantiated, but contains uninstantiated variables that do not also appear in \textit{Template}, then \texttt{setof/3} can succeed nondeterminately, generating alternative values for \textit{Set} corresponding to different instantiations of the free variables of \textit{Generator}. (It is to allow for such usage that \textit{Set} is constrained to be non-empty.)

If \textit{Generator} is of the form \texttt{A^\text{-}B}, then all the variables in \texttt{A} are treated as being existentially quantified.

\textbf{Please note:} If the instances being gathered contain attributed variables (see Section 10.3 [lib-atts], page 391) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 74), then those variables are replaced by brand new variables, without attributes, in the \textit{Set}. To retain the attributes, you can use \texttt{copy_term/3} (see Section 4.8.7 [ref-lte-cpt], page 129).

Backtracking

\texttt{setof/3} can succeed nondeterminately, generating alternative values for \textit{Set} corresponding to different instantiations of the free variables of \textit{Generator}.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

Examples

See \texttt{findall/3} for examples that illustrate the differences among \texttt{findall/3}, \texttt{setof/3}, and \texttt{bagof/3}.
See Also

findall/3, bagof/3, ~\(/2, Section 4.13 [ref-all], page 186.
11.3.209  simple/1

Synopsis

\texttt{simple(+Term)}

\textit{Term} is currently not instantiated to a compound term.

Arguments

\texttt{Term} \hspace{1cm} \textit{term}

Examples

\begin{verbatim}
| ?- simple(9).
   yes
| ?- simple(_X).
   yes
| ?- simple("a").
   no
\end{verbatim}

Exceptions

None.

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.210 skip_byte/[1,2]

Synopsis

skip_byte(+Byte)

skip_byte(+Stream, +Byte)

read up to and including the first occurrence of Byte on the current input stream or on the input stream Stream.

Arguments

Stream stream_object, must be ground
A valid input binary stream, defaults to the current input stream.

Byte byte

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

type_error
Byte is an invalid byte.

permission_error
Trying to read beyond end of Stream

type_error
Byte is not a byte

See Also

Section 4.6.5 [ref-iou-cin], page 107.
11.3.211  skip_char/[1,2]

Synopsis

skip_char(+Char)

skip_char(+Stream, +Char)

Read up to and including the first occurrence of Char on the current input stream or on
the input stream Stream.

Arguments

Stream  stream_object, must be ground
       A valid input text stream.

Char    char

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
       Trying to read beyond end of Stream

type_error
       Char is not a char

See Also

Section 4.6.5 [ref-iou-cin], page 107.
11.3.212 skip_code/[1,2]

Synopsis
skip_code(+Code)

skip_code(+Stream, +Code)

read up to and including the first occurrence of Code on the current input stream or on the input stream Stream.

Arguments
Stream stream_object, must be ground
A valid input text stream, defaults to the current input stream.

Code code

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
Trying to read beyond end of Stream

representation_error
Code is not a code

See Also
Section 4.6.5 [ref-iou-cin], page 107.
11.3.213  skip_line/[0,1]

Synopsis

skip_line

skip_line(+Stream)

Skip the remaining input characters on the current line on Stream.

Arguments

Stream  stream_object, must be ground
        A valid input text stream, defaults to the current input stream.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

permission_error
        Trying to read beyond end of Stream

See Also

at_end_of_line/[0,1], Section 4.6.5 [ref-iou-cin], page 107.
11.3.214 sort/2

Synopsis

sort(+List, -Sorted)

Sorts the elements of the list List into the ascending standard order, and removes any multiple occurrences of an element. The resulting sorted list is unified with the list Sorted.

Arguments

List list of term, must be a proper list
Sorted list of term

Sorted is type checked since release 4.3 for alignment with the ISO Prolog standard. Previous releases simply failed instead of reporting an error for malformed Sorted.

Exceptions

instantiation_error
type_error
        List is not a proper list
type_error
        Sorted cannot be unified with a proper list

Examples

| ?- sort([a,X,1,a(X),a,a(X)], L).  
  L = [X,1,a,a(X),a(X)]

(The time taken to do this is at worst order \(N \log N\) where \(N\) is the length of the list.)

See Also

Section 4.8.8.3 [ref-lte-cte-sor], page 131.
11.3.215  source_file/[1,2]

Synopsis

source_file(?AbsFile)

source_file(?Pred, ?AbsFile)

AbsFile is the absolute name of a loaded file, and Pred is a predicate with clauses in that file. AbsFile will be user if the special file specification user was loaded, and Pred is a predicate with clauses from user.

Arguments

:Pred   callable
       Selected predicate specification.

AbsFile   atom
       Absolute filename.

Description

Loaded files include compiled, consulted, restored, PO loaded and pre-linked files.

If AbsFile is bound and not the name of a loaded file, or if Pred is bound and not the name of a loaded predicate, then source_file(AbsFile) simply fails.

To find any predicates defined in a given file, use the form:

    source_file(M:P, File)

Examples

Suppose that the startup file `~/.sicstusrc` defines a predicate user:cd/1. Then upon startup:

```prolog
| ?- source_file(F).
F = '/src/sicstus/matstc/sicstus4/Utils/x86-linux-glibc2.3/bin/sp-4.1.0/sicstus-4.1.0/library/SU_messages.pl' ? ;
F = '/home/matsc/.sicstusrc' ? ;
no
```

```prolog
| ?- source_file(P,F).
F = '/home/matsc/.sicstusrc',
P = cd(_A) ? ;
no
```

```prolog
| ?- source_file('SU_messages':P,F).
F = '/src/sicstus/matstc/sicstus4/Utils/x86-linux-glibc2.3/bin/sp-4.1.0/sicstus-4.1.0/library/SU_messages.pl',
P = query_class(_A,_B,_C,_D,_E) ? RET
yes
```
Exceptions
None.

See Also
Section 4.9.3 [ref-lps-apf], page 136.
**11.3.216 spy/[1,2]**

### Synopsis

spy +PredSpecs

Sets plain spypoints on all the predicates represented by `PredSpecs`.

`spy(+PredSpecs, +Conditions)`

Sets conditional spypoints on all the predicates represented by `PredSpecs`.

### Arguments

`:PredSpecs`

`pred_spec_tree`

A predicate specification, or a list of such.

`:Conditions`

`term`, must be ground

Spypoint conditions.

### Description

Turns debugger on in debug mode, so that it will stop as soon as it reaches a spypoint. Turning off the debugger does not remove spypoints. Use `nospy/1` or `nospyall/0` to explicitly remove them.

If you use the predicate specification form `Name` but there are no clauses for `Name` (of any arity), then a warning message will be displayed and no spypoint will be set.

```
| ?- spy test.
* spy user:test - no matching predicate
```

### Exceptions

`instantiation_error`

`type_error`

`domain_error`

if a `PredSpec` is not a valid procedure specification

### See Also

Section 5.2 [Basic Debug], page 233, Section 5.3 [Plain Spypoint], page 235, Section 5.7 [Breakpoint Predicates], page 272.
Synopsis

statistics

Displays statistics relating to memory usage and execution time.

statistics(?Keyword, ?List)

Obtains individual statistics.

Arguments

Keyword   atom
  Statistics key (see Section 4.10.1.2 [ref-mgc-ove-sta], page 146).

List      list of integer
  List of statistics.

Description

statistics/0 displays various statistics relating to memory usage, runtime and garbage collection, including information about which areas of memory have overflowed and how much time has been spent expanding them. The printing is handled by print_message/2.

Garbage collection statistics are initialized to zero when a Prolog session starts. The statistics increase until the session is over.

statistics/2 is usually used with Keyword instantiated to a keyword such as runtime and List unbound. The predicate then binds List to a list of statistics related to the keyword. It can be used in programs that depend on current runtime statistical information for their control strategy, and in programs that choose to format and write out their own statistical summaries.

Exceptions

type_error
  Invalid keyword.

domain_error

Examples

To report information on the runtime of a predicate p/0, add the following to your program:

    :- statistics(runtime, [T0|_]),
    p,
    statistics(runtime, [T1|_]),
    T is T1 - T0,
    format('p/0 took ~3d sec.~n', [T]).

See Also

Section 4.10.1.2 [ref-mgc-ove-sta], page 146, Section 4.16 [ref-msg], page 212.
11.3.218  stream_code/2

Synopsis

stream_code(-Stream, +CStream)
stream_code(+Stream, -CStream)

Converts between Prolog representation, Stream, and C representation, CStream, of a stream.

Arguments

Stream  stream_object
        A valid Prolog stream.
CStream  integer
        Representing an SP_stream * pointer.

Description

At least one argument must be ground. stream_code/2 is used when there are input/output related operations performed on the same stream in both Prolog code and foreign code. The CStream value can be used as the stream argument to any of the SP_* functions taking a stream argument.

Exceptions

instantiation_error
        Both Stream and CStream unbound.
type_error
        Stream or CStream is not a stream type or CStream is not an integer type.
existence_error
        Stream is syntactically valid but does not name an open stream or CStream is of integer type but does not name a pointer to a stream.

See Also

Section 6.6.1 [Prolog Streams], page 311.
11.3.219 stream_position/2

Synopsis
stream_position(+Stream, -Position)

True when Position represents the current position of Stream.

Arguments
Stream stream_object, must be ground
An open stream.
Position term
Stream position object representing the current position of Stream.

Description
Byte, character, and line counts and line position determine the position of the pointer in the stream. Such information is found by using byte_count/2, character_count/2, line_count/2 and line_position/2. A stream position object packages this information as a single Prolog terms. You can retrieve this information from a stream position object using stream_position_data/3. Do not rely on the form of this object in any other way.

Standard term comparison of two stream position objects for the same stream will work as one expects. When SP1 and SP2 refer to positions in the same stream, SP1 @< SP2 if and only if SP1 is before SP2 in the stream.

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

Examples

See Also
Section 4.6.7 [ref-iou-sfh], page 109.
11.3.220 stream_position_data/3

Synopsis

\texttt{stream_position_data(?Field, \_Position, \_Value)}

Value is the value of the Field field of stream position object Position.

Arguments

\textit{Field} \quad one of \texttt{[byte_count, line_count, character_count, line_position]}

Note that \texttt{byte_count} is meaningful only for binary streams and that the other values are meaningful only for text streams.

\textit{Position} \quad \texttt{term}

Stream position object representing the current position of Stream.

\textit{Value} \quad \texttt{integer}

Backtracking

Can be used to backtrack over the fields.

Exceptions

None.

See Also

Section 4.6.7 \texttt{[ref-iou-sfh]}, page 109.
11.3.221 stream_property/2

Synopsis

stream_property(?Stream, ?Property)

Stream Stream has stream property Property.

Arguments

Stream stream_object

Property term

A stream property, one of the following:

file_name(F)

F is the file name associated with the Stream.

mode(M)

Stream has been opened in mode M.

id(ID) since release 4.2

Stream has the unique identity ID. The identity of a stream is a positive integer that is never re-used during the life-time of the SICStus process. This is unlike the compound term Stream which is likely to be re-used for some new stream after the original stream denoted by Stream has been closed.

input

Stream is an input stream. Note that both input and output stream properties are set for bidirectional streams.

output

Stream is an output stream. Note that both input and output stream properties are set for bidirectional streams.

alias(A)

Stream has an alias A.

position(P)

P is a term representing the current stream position of Stream. Only guaranteed to be available if the stream has been opened with the open/4 option reposition(true).

Same as stream_position(Stream, P) except that the latter can be called on any stream, regardless of the value of the reposition/1 open/4 option.

end_of_stream(E)

E describes the position of the input stream Stream, with respect to the end of stream. If not all characters have been read, or if peeking ahead to determine this fact would block, then E is unified with not; otherwise, (all characters read) but no end of stream indicator (-1 or end_of_file) was reported yet, then E is unified with at; otherwise, E is unified with past.

eof_action(A)

A is the end-of-file action applicable to Stream, cf. the eof_action option of open/4.
type(T) Stream is of type T, one of text, binary, cf. the type option of open/4.

input_encoding(CS) since release 4.3
output_encoding(CS) since release 4.3
Stream is a text stream with encoding CS in the input direction, cf. the encoding option of open/4. Note that the encoding used may be different from the encoding option passed to open/4 if a byte order mark or other information was used to determine the real encoding of the file, cf. the encoding_signature option of open/4.

encoding(CS)
Stream is a text stream open in direction input, with input encoding CS or Stream is a text stream open in direction output but not in direction input, with output encoding CS.
Note that, for bi-directional streams, the encoding/1 property reflects the input_encoding/1.

eol(EOL) Stream is a text stream with end of line convention EOL, cf. the eol option of open/4.

encoding_signature(ES)
If Stream is a text stream, then ES is determined as follows:
If the file contents was used to determine the character encoding, then ES will be true. Typically this is the result of opening, in mode read, a text file that contains a byte order mark or some other information that lets open/[3,4] determine a suitable encoding, cf. the encoding_signature option of open/4.
Otherwise, if the stream is open in direction output, then ES will be as specified when the file was opened.

reposition(REPOSITION)
REPOSITION is true if it is possible to set the position of the stream with set_stream_position/2, cf. the reposition option of open/4.

interactive since release 4.1
Stream is an interactive stream.
Most streams have only a subset of these properties set.
More properties may be added in the future.

**Backtracking**
Can be used to backtrack over all currently open streams, including the standard input/output/error streams, and all their properties. See Section 4.6.7.8 [ref-iou-sfh-bos], page 113.

**Exceptions**
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:
domain_error
   Stream is not a valid stream object, or Property is not a valid stream property.

See Also
Section 4.6.7 [ref-iou-sfh], page 109.
11.3.222 sub_atom/5

Synopsis

sub_atom(+Atom, -Before, -Length, -After, -SubAtom)

The characters of SubAtom form a sublist of the characters of Atom, such that the number of characters preceding SubAtom is Before, the number of characters after SubAtom is After, and the length of SubAtom is Length.

Arguments

Atom atom, must be nonvar
The atom from which a part is selected.

Before integer
The number of characters preceding SubAtom.

Length integer
The number of characters of SubAtom.

After integer
The number of characters following SubAtom.

SubAtom atom
The selected part of Atom.

Description

Capable of nondeterminately enumerating all sub-atoms and their all possible placements, e.g.:

| ?- sub_atom(abrakadabra, Before, _, After, ab).

After = 9,
Before = 0 ? ;

After = 2,
Before = 7 ? ;

no

Exceptions

instantiation_error
Atom is uninstantiated.

type_error
Atom is not an atom. Before, Length, or After, if instantiated, is not an integer. SubAtom, if instantiated, is not an atom.

domain_error
Before, Length, or After, if instantiated, is negative.
See Also

atom_length/2, atom_concat/3.
11.3.223 `subsumes_term/2`  

**Synopsis**

`subsumes_term(+General, +Specific)`

is true when `Specific` is an instance of `General`. It does not bind any variables.

**Arguments**

- `General`  any term.
- `Specific`  any term.

**Description**

True iff there is a substitution that makes `General` identical to `Specific` and that does not affect `Specific`.

The predicate is determinate and either succeeds or fails. It never binds variables.

The predicate does not take any constraints, variable attributes, or blocked goals into account when determining whether a substitution exists. This can be seen in the last two examples.

**Examples**
| ?- subsumes_term(a, a).  
  yes  

| ?- subsumes_term(f(X,Y), f(Z,Z)).  
  yes  

| ?- subsumes_term(f(Z,Z), f(X,Y)).  
  no  

| ?- subsumes_term(g(X), g(f(X))).  
  no  

| ?- subsumes_term(X, f(X)).  
  no  

| ?- subsumes_term(X, Y), subsumes_term(Y, f(X)).  
  yes  

| ?- when(nonvar(X), X=a), subsumes_term(X, b), X = a.  
  X = a ?  
  yes  

| ?- when(nonvar(X), X=a), subsumes_term(X, b), X = b.  
  no  

Exceptions
None.

See Also
Section 4.8.1.2 [ref-lte-met-usu], page 126.
11.3.224 tell/1

Synopsis

tell(+FileOrStream)

Makes file FileOrStream the current output stream.

Arguments

FileOrStream

file_spec or stream_object, must be ground

Description

If there is an open output stream associated with FileOrStream, and that stream was opened by tell/1, then it is made the current output stream. Otherwise, the specified file is opened for output in text mode with default options and made the current output stream.

Different file names (that is, names that do not unify) represent different streams (even if they correspond to the same file). Therefore, assuming food and ./food represent the same file, the following sequence will open two streams, both connected to the same file.

```
tell(food)
...
tell('./food')
```

It is important to remember to close streams when you have finished with them. Use told/0 or close/[1,2].

Exceptions

instantiation_error

FileOrStream is not instantiated enough.

existence_error

FileOrStream specifies a nonexisting file, and the fileerrors Prolog flag is on.

permission_error

FileOrStream is a stream not currently open for output, or FileOrStream specifies a file with insufficient access permission, and the fileerrors Prolog flag is on.

domain_error

FileOrStream is neither a file_spec nor a stream_object.

See Also

told/0, open/[3,4], current_output/1, Section 4.6.7 [ref-iou-sfh], page 109, Section 4.9.4 [ref-lps-flg], page 136.
11.3.225 telling/1

Synopsis
telling(-FileOrStream)

Unifies FileOrStream with the current output stream or file.

Arguments

FileOrStream
  file_spec or stream_object

Description

Exactly the same as current_output(FileOrStream), except that FileOrStream will be
unified with a filename if the current output stream was opened by tell/1 (Section 4.6.7
[ref-ion-sfh], page 109).

Can be used to verify that FileNameOrStream is still the current output stream as follows:

    % nonvar(FileNameOrStream),
    tell(FileNameOrStream),
    ...
    telling(FileNameOrStream)

If the current output stream has not been changed (or if changed, then restored), then the
above sequence will succeed for all file names and all stream objects opened by open/[3,4].
However, it will fail for all stream objects opened by tell/1 (since only filename access to
streams opened by tell/1 is supported). This includes the stream object user_output
(since the standard output stream is assumed to be opened by tell/1, and so telling/1
would return user in this case).

If FileOrStream is instantiated to a value that is not the identifier of the current output
stream, then telling(FileOrStream) simply fails.

Can be followed by tell/1 to ensure that a section of code leaves the current output
unchanged:

    % var(OldFileNameOrStream),
    telling(OldFileNameOrStream),
    ...
    tell(OldFileNameOrStream)

The above is analogous to its stream-object-based counterpart,

    % var(OldStream),
    current_output(OldStream),
    ...
    set_output(OldStream)
Both of these sequences will always succeed regardless of whether the current output stream was opened by \texttt{tell/1} or \texttt{open/[3,4]}.

\textbf{Exceptions}

None.

\textbf{See Also}

\texttt{tell/1}, \texttt{open/[3,4]}, \texttt{current\_input/1}, Section 4.6.7 [ref-ionsf], page 109.
11.3.226 ==/2

Synopsis
+Term1 == +Term2

Succeeds if Term1 and Term2 are identical terms.

Arguments
Term1 term
Term2 term

Examples
For example, the query

| ?- X == Y.

fails (answers ‘no’) because X and Y are distinct uninstantiated variables. However, the query

| ?- X = Y, X == Y.

succeeds because the first goal unifies the two variables.

Exceptions
None.

See Also
Section 4.8.8 [ref-lte-cte], page 130.
11.3.227 term_expansion/6  

Synopsis

:- multifile user:term_expansion/6.

user:term_expansion(+Term1, +Layout1, +Tokens1, -Term2, -Layout2, -Tokens2)

Overrides or complements the standard transformations to be done by expand_term/2.

Arguments

Term1 term  
Term to transform.

Layout1 term  
Layout term of Term1.

Tokens1 list of atom  

Term2 term  
Transformed term.

Layout2 term  
Layout term of Term2.

Tokens2 list of atom

Description

expand_term/2 calls this hook predicate first; if it succeeds, then the standard grammar rule expansion is not tried.

Tokens1 is a list of atoms, each atom uniquely identifying an expansion. It is used to look up what expansions have already been applied to the clause or goal. The tokens are defined by the user, and should simply be added to the input list, before expansions recursively are applied. This token list can for instance be used to avoid cyclic expansions. The token dcg is reserved and denotes grammar rule expansion. Tokens2 should be unified with [Token|Tokens1].

Layout1 and Layout2 are for supporting source-linked debugging in the context of clause expansion. The predicate should construct a suitable Layout2 compatible with Term2 that contains the line number information from Layout1. If source-linked debugging of Term2 is not important, then Layout2 should be []. The recording of source info is affected by the source_info prolog flag (see Section 4.9.4 [ref-lps-flg], page 136).

A clause of this predicate should conform to the following template, where convert(Term1, Layout1, Expansion, Layout) should be a goal that performs the actual transformation. Token should be the atom uniquely identifying this particular transformation rule. Tokens2 should be unified with [Token|Tokens1].
user:term_expansion(Term1, Layout1, Tokens1, Term2, Layout2, Tokens2) :-
  nonmember(Token, Tokens1),
  convert(Term1, Layout1, Expansion, Layout),
  !, % commit to this expansion
  Term2 = Expansion,
  Layout2 = Layout,
  Tokens2 = [Token|Tokens1].

This hook predicate may return a list of terms rather than a single term. Each of the terms
in the list is then treated as a separate clause (or directive).

This hook predicate may also be used to transform queries entered at the terminal in
response to the ‘| ?- ’ prompt. In this case, it will be called with Term1 = ?-(Query)
and should succeed with Term2 = ?-(ExpandedQuery).

For accessing aspects of the load context, e.g. the name of the file being compiled, the
predicate prolog_load_context/2 (see Section 4.9.5 [ref-lps-lco], page 143) can be used.

Exceptions
Exceptions are treated as failures, except an error message is printed also.

See Also
Section 4.3.5 [ref-lod-exp], page 87, Chapter 2 [Glossary], page 7.
11.3.228 \( @>/2 \)  

**Synopsis**

\[ +\text{Term}1 \ @> +\text{Term}2 \]

Succeeds if \( \text{Term}1 \) is *after* \( \text{Term}2 \) in the standard order.

**Arguments**

\[ \text{Term}1 \quad \text{term} \]

\[ \text{Term}2 \quad \text{term} \]

**Exceptions**

None.

**See Also**

Section 4.8.8 [ref-lte-cte], page 130.
11.3.229 \( @</2 \)  

**Synopsis**

\(+\text{Term1} @< +\text{Term2}\)

Succeeds if \(\text{Term1}\) is *before* \(\text{Term2}\) in the standard order.

**Arguments**

\(\text{Term1} \quad \text{term}\)
\(\text{Term2} \quad \text{term}\)

**Exceptions**

None.

**See Also**

Section 4.8.8 [ref-lte-cte], page 130.
11.3.230 \==/2

Synopsis

+Term1 \== +Term2

Succeeds if Term1 and Term2 are non-identical terms.

Arguments

Term1 term
Term2 term

Exceptions

None.

See Also

Section 4.8.8 [ref-lte-cte], page 130.
11.3.231  @=</2

Synopsis

+Term1  @=<  +Term2

Succeeds if Term1 is not after Term2 in the standard order.

Arguments

Term1       term
Term2       term

Exceptions

None.

See Also

Section 4.8.8 [ref-lte-cte], page 130.
11.3.232 \( \geq /2 \)

Synopsis

\(+Term1 \geq +Term2\)

Succeeds if \( Term1 \) is not before \( Term2 \) in the standard order.

Arguments

\( Term1 \quad \text{term} \)
\( Term2 \quad \text{term} \)

Exceptions

None.

See Also

Section 4.8.8 [ref-lte-cte], page 130.
11.3.233  ?=/2

Synopsis

?=+(Term1,Term2)

Succeeds if Term1 and Term2 are identical terms, or if they are syntactically non-unifiable.

Arguments

Term1  term
Term2  term

Comments

Succeeds if and only if dif(Term1,Term2) does not block.

Exceptions

None.

See Also

Section 4.8.1.2 [ref-lte-met-usu], page 126.
11.3.234 term_variables/2

Synopsis

`term_variables(+Term, -Variables)` since release 4.3 True if Variables is the list of variables occurring in Term, without duplicates, in first occurrence order.

Arguments

- **Term** Any term, a cyclic term is also accepted.
- **Variables** The variables in the term. Must be a variable or a list.

Exceptions

- **type_error** Variables is not a variable or a list.

Examples

```prolog
| ?- term_variables(f(A, B, A), Vs).
Vs = [A, B] ? yes
| ?- term_variables(f(a, b, a), Vs).
Vs = [] ? yes
| ?- T=[A, B|T], term_variables(f(C, T), Vs).
T = [A, B, A, B, A, B, A, B, A, B|...],
Vs = [C, A, B] ? yes
```

See Also

Section 4.8.2 [ref-lte-act], page 127.
11.3.235 throw/1

Synopsis

throw(+Exception)

Exception is thrown as an exception.

Arguments

Exception term, must be nonvar

Exceptions

instantiation_error

Exception is unbound.

See Also

Section 4.15 [ref-ere], page 197, on_exception/1.
11.3.236 told/0

Synopsis
told
told
Closes the current output stream.

Description

Current output stream is set to be user_output; that is, the user’s terminal.

Always succeeds

Exceptions
None.

Examples

See Also
tell/1, close/[1,2], current_output/1, Section 4.6.7 [ref-iou-sfh], page 109.
11.3.237 trace/0

**Synopsis**

trace

Turns on the debugger in trace mode.

**Description**

The debugger will start showing goals as soon as the first call is reached, and it will stop
to allow you to interact as soon as it reaches a leashed port (see `leash/1`). Setting the
debugger to trace mode means that every time you type a query, the debugger will start by
creeing.

The effect of this predicate can also be achieved by typing the letter t after a `^C` interrupt
(see Section 3.7 [Execution], page 26).

**Exceptions**

None.

**See Also**

Section 5.2 [Basic Debug], page 233.
11.3.238 trimcore/0

Synopsis
trimcore

Force reclamation of memory in all of Prolog’s data areas.

Description
Trims the stacks, reclaims any dead clauses and predicates, defragmentizes Prolog’s free memory, and endeavors to return any unused memory to the operating system.

The system property PROLOGKEEPSIZE can be used to define a lower bound on the amount of memory to be retained. Also, the system property PROLOGINITSIZE can be used to request that an initial amount of memory be allocated. This initially allocated memory will not be touched by trimcore/0.

When trimming a given stacks, trimcore/0 will retain at least the amount of space initially allocated for that stack.

trimcore/0 is called each time Prolog returns to the top level or the top of a break level, except it does not trim the stacks then.

Exceptions
None.

See Also
Section 4.10.1.1 [ref-mgc-ove-rsp], page 146, Section 4.17.1 [System Properties and Environment Variables], page 224.
11.3.239 true/0

Synopsis
true

Always succeeds.

Exceptions
None.

See Also
Section 4.2 [ref-sem], page 61.
11.3.240  =/2

Synopsis
+Term1 = +Term2

unifies Term1 and Term2.

Arguments
Term1    term
Term2    term

Description
This is defined as if by the clause ‘Z = Z.’.

If =/2 is not able to unify Term1 and Term2, then it will simply fail.

Exceptions
None.

See Also
Chapter 2 [Glossary], page 7, Section 4.2.7 [ref-sem-occ], page 78.
11.3.241 unify_with_occurs_check/2

ISO

Synopsis
unify_with_occurs_check(+Term1, +Term2)

$Term1$ and $Term2$ unify to a finite (acyclic) term.

Arguments

$Term1$ term

$Term2$ term

Description
Runs in almost linear time.

Exceptions
None.

See Also
Chapter 2 [Glossary], page 7.
11.3.242 =../2

ISO

Synopsis

+Term =.. -List

-Term =.. +List

Unifies List with a list whose head is the atomic term corresponding to the principal functor of Term and whose tail is a list of the arguments of Term.

Arguments

Term term any term
List list of term and not empty

Description

If Term is uninstantiated, then List must be instantiated either to a proper list whose head is an atom, or to a list of length 1 whose head is a number.

This predicate is not strictly necessary, since its functionality can be provided by arg/3 and functor/3, and using the latter two is usually more efficient.

Examples

| ?- product(0, n, n-1) =.. L.  
  L = [product,0,n,n-1]  
| ?- n-1 =.. L.  
  L = [-,n,1]  
| ?- product =.. L.  
  L = [product]

Exceptions

instantiation_error

Term is unbound and List is not instantiated enough.

type_error

List is not a proper list, or the head of List is not atomic, or the head of List is a number and the tail of List is not empty.

domain_error

List is the empty list.

representation_error

Term is uninstantiated and List is longer than 256.
See Also

functor/3, arg/3, Section 4.8.2 [ref-lte-act], page 127.
11.3.243 unknown/2

devolution

Synopsis

unknown(-OldAction, +NewAction)

Unifies OldAction with the current action on unknown procedures, i.e. the current value of the unknown Prolog flag, sets the current action to NewAction, and prints a message about the change.

Arguments

OldAction   one of [error, fail, trace, warning]
NewAction   one of [error, fail, trace, warning], must be nonvar

Description

This is merely a front-end to the unknown Prolog flag, which see.

Note that:

| ?- unknown(Action, Action).

just returns Action without changing it.

Procedures that are known to be dynamic just fail when there are no clauses for them.

Exceptions

instantiation_error
type_error
domain_error

Invalid NewAction.

See Also

Section 3.6 [Undefined Predicates], page 26, Section 4.15 [ref-ere], page 197, Section 4.9.4 [ref-lps-flg], page 136.
11.3.244 unknown_predicate_handler/3

Synopsis

:- multifile user:unknown_predicate_handler/3.

user:unknown_predicate_handler(+Goal, +Module, -NewGoal)

User definable hook to trap calls to unknown predicates.

Arguments

Goal         callable
  The goal to trap.

Module       atom
  Any atom that is a current module

NewGoal      callable
  The goal to call instead.

Description

When Prolog comes across a call to an unknown predicate, Prolog makes a call to
user:unknown_predicate_handler/3 with the first two arguments bound. Goal is bound
to the call to the undefined predicate and Module is the module in which that predicate
is supposed to be defined. If the call to user:unknown_predicate_handler/3 succeeds,
then Prolog replaces the call to the undefined predicate with the call to Module:NewGoal.
Otherwise, the action taken is governed by the unknown Prolog flag.

Exceptions

Exceptions are treated as failures, except an error message is printed.

Examples

The following clause gives the same behavior as setting unknown(_,fail):

    unknown_predicate_handler(_, _, fail).

The following clause causes calls to undefined predicates whose names begin with ‘xyz_’ in
module m to be trapped to my_handler/1 in module n. Predicates with names not beginning
with this character sequence are not affected.

    unknown_predicate_handler(G, m, n:my_handler(G)) :-
        functor(G,N,_),
        atom_concat(xyz_, _, N).

See Also

Section 3.6 [Undefined Predicates], page 26, Section 4.15 [ref-ere], page 197, Section 4.9.4
[ref-lps-flg], page 136.
11.3.245  unload_foreign_resource/1  hookable

Synopsis

unload_foreign_resource(:Resource)

Unload the foreign resource Resource from Prolog. Relies on the hook predicates foreign_resource/2 and foreign/[2,3].

Arguments

:Resource file_spec, must be ground
  The foreign resource to be unloaded. The file extension can be omitted.

Exceptions

instantiation_error
  Resource not ground.

type_error
  Resource not an atom.

existence_error
  Resource does not exist as a foreign resource.

See Also

load_foreign_resource/1, foreign_resource/2, foreign/[2,3], Section 6.2.1 [Foreign Resources], page 291, Section 6.2 [Calling C from Prolog], page 291.
### 11.3.246 update_mutable/2

**Synopsis**

`update_mutable(+Datum, +Mutable)`

Updates the current value of the mutable term `Mutable` to become `Datum`.

**Arguments**

- `Datum` term, must be nonvar
- `Mutable` mutable, must be nonvar

**Exceptions**

- `instantiation_error`
  - `Datum` or `Mutable` is uninstantiated.
- `type_error`
  - `Mutable` is not a mutable.

**See Also**

Section 4.8.9 [ref-lte-mut], page 131.
11.3.247 use_module/[1,2,3]

Synopsis

use_module(+File)

Loads the module file(s) File, if not already loaded and up-to-date imports all exported predicates.

use_module(+File, +Imports)

Loads module file File, if not already loaded and up-to-date imports according to Imports.

use_module(+Module, -File, +Imports)

Module is already loaded and up-to-date. Imports according to Imports.

use_module(-Module, +File, +Imports)

Module has not been loaded, or is out-of-date. Loads Module from File and imports according to Imports.

Arguments

:File file_spec or list of file_spec, must be ground Any legal file specification. Only use_module/1 accepts a list of file specifications, file extensions optional.

Imports list of simple_pred_spec or one of [all], must be ground Either a list of predicate specifications in the Name/Arity form to import into the calling module, or the atom all, meaning all predicates exported by the module are to be imported.

Module atom The module name in File, or a variable, in which case the module name is returned.

Description

Loads each specified file except the previously loaded files that have not been changed since last loaded. All files should be module files; if they are not, then warnings are issued. All the exported predicates of the modules are imported into the calling module (or module M if specified).

use_module/2 imports only the predicates in Imports when loading File. If an attempt is made to import a predicate that is not public, then a warning is issued. If File is not a module file, then nothing is imported.

use_module/3 allows Module to be imported into another module without requiring that its source file (File) be known, as long as the Module already exists in the system.

Generally, use_module/3 is similar to use_module/[1,2], except that if Module is already in the system, then Module, or predicates from Module, are simply imported into the calling module, and File is not loaded again. If Module does not already exist in the system, then File is loaded, and use_module/3 behaves like use_module/2, except that Module is unified,
after the file has been loaded, with the actual name of the module in File. If Module is a variable, then File must exist, and the module name in File is returned.

*use_module/1* is similar to *ensure_loaded/1* except that all files should be module files; if they are not, then warnings are issued.

An attempt to import a predicate may fail or require intervention by the user because a predicate with the same name and arity has already been defined in, or imported into, the loading module (or module $M$ if specified). Details of what happens in the event of such a *name clash* are given in Section 4.11.2 [ref-mod-bas], page 162.

After loading the module file, the source module will attempt to import all the predicates in *Imports*. *Imports* must be a list of predicate specifications in *Name/Arity* form. If the file is not a module file, then nothing is imported. If any of the predicates in *Imports* are not public predicates, then a warning is issued, but the predicates are imported nonetheless. This lack of strictness is for convenience; if you forget to declare a predicate to be public, then you can supply the necessary declaration and reload its module, without having to reload the module that has imported the predicate.

While *use_module/1* may be more convenient at the top level, *use_module/2* is recommended in files because it helps document the interface between modules by making the list of imported predicates explicit.

For consistency, *use_module/2* has also been extended so that the *Imports* may be specified as the term *all*, in which case it behaves the same as *use_module/1*, importing the entire module into the caller.

**Exceptions**

See also *load_files/[2,3]*.

*instantiation_error*  
File or Imports is not ground.

*type_error*  
In File or Imports.

**Examples**

*use_module/[1,2]* could be defined as:

```
use_module(Files) :-
    load_files(Files, [if(changed)]).
```

```
use_module(File, Imports) :-
    load_files([File], [if(changed),imports(Imports)]).
```

*use_module/3* can be used to access the (primary) module name of a module file:
| ?- use_module(Module, library(clpfd), all).
% loading /src/sicstus/matsc/sicstus4/Utils/x86-linux-glibc2.3/bin/sp-4.1.0/sicstus-4.1.0/library/clpfd.po...
[...]
% loaded /src/sicstus/matsc/sicstus4/Utils/x86-linux-glibc2.3/bin/sp-4.1.0/sicstus-4.1.0/library/clpfd.po in module clpfd, 830 msec 496796 bytes
Module = clpfd ?
yes

See Also

Section 4.3.2 [ref-lod-lod], page 80.
11.3.248 \texttt{var/1} \hspace{1em} \textit{ISO}

Synopsis
\texttt{var(+Term)}

\textit{Term} is currently uninstantiated.

Arguments
\textit{Term} \hspace{1em} \texttt{term}

Examples
\begin{verbatim}
| ?- var(foo(X,Y)).  
  no
| ?- var([X,Y]).     
  no
| ?- var(X).         
  true ;
  no
| ?- Term = foo(X,Y), var(Term).
  no
\end{verbatim}

Exceptions
None.

See Also
Section 4.8.1.1 [ref-lte-met-typ], page 126.
11.3.249 volatile/1

Declares PredSpecs to be volatile. Clauses of volatile predicates are not saved by the `save_*` predicates.

Arguments

:PredSpecs

pred_spec_forest, must be ground
A predicate specification, or a list of such, or a sequence of such separated by commas.

Exceptions

Exceptions in the context of loading code are printed as error messages.

instantiation_error

PredSpecs not ground.

type_error

PredSpecs not a valid pred_spec_forest.

domain_error

Some arity is an integer < 0.

representation_error

Some arity is an integer > 255.

context_error

Declaration appeared in a goal.

permission_error

Declaration appeared as a clause.

See Also

Section 4.3.4.3 [Volatile Declarations], page 84.
11.3.250  when/2

Synopsis
when(+Condition,+Goal)

Blocks Goal until the Condition is true.

Arguments

Condition  callable, must be nonvar and one of:
  nonvar(X)       False until X is nonvar.
  ground(X)      False until X is ground.
  ==(X,Y)        False while dif(X,Y) would block.
  Condition;Condition  True if at least one condition is true.
  Condition;Condition  True if both conditions are true.

:Goal  callable, must be nonvar

Backtracking
Depends on Goal.

Examples

    | ?- when(((nonvar(X);?=(X,Y)),ground(T)), process(X,Y,T)).

Exceptions
Call errors (see Section 4.2.6 [ref-sem-exc], page 77).

See Also
Section 4.2.4 [ref-sem-sec], page 74.
11.3.251 write/[1,2]  

Synopsis

write(+Stream, +Term)

write(+Term)

Writes Term on Stream, without quoting atoms.

Arguments

Stream stream_object, must be ground
  A valid open Prolog stream, defaults to the current output stream.

Term term

Description

write(Term) is equivalent to:

    write_term(Term, [numbervars(true)])

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also

Section 4.6.4 [ref-iou-tou], page 104.
11.3.252 write_canonical/[1,2]  

**Synopsis**

write_canonical(+Stream, +Term)

write_canonical(+Term)

Writes Term on Stream, quoting atoms, in functional notation, without treating '$VAR'/1 terms specially.

**Arguments**

*Stream*    stream object, must be ground

A valid open Prolog stream, defaults to the current output stream.

*Term*    term

**Description**

This predicate is provided so that Term, if written to a file, can be read back by read/[1,2] regardless of special characters in Term or prevailing operator declarations.

write_canonical(Term) is equivalent to:

\[
\text{write_term(Term, [quoted(true),ignore_ops(true),quoted_charset(portable)])}
\]

**Exceptions**

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

**Examples**

To contrast write/[1,2] and write_canonical/[1,2]:

\[
\text{| ?- write('A' + '$VAR'(0) + [a]).} \\
\text{\{A+A+[a]\}} \\
\text{\| ?- write_canonical('A' + '$VAR'(0) + [a]).} \\
\text{\{+(+(\text{'A'},'$VAR'(0)),(a,[]))\}} \\
\]

**See Also**

Section 4.6.4 [ref-iou-tou], page 104.
11.3.253 write_term/[2,3]  hookable, ISO

Synopsis
write_term(+Stream, +Term, +Options)
write_term(+Term, +Options)

W rites Term on Stream, subject to +Options.

Arguments

Stream  stream_object, must be ground
A valid open Prolog stream, defaults to the current output stream.

Term    term

Options  list of term.
A list of zero or more of the following, where Boolean must be true or false (false is the default).

quoted(Boolean)
If selected, then atoms and functors are quoted where necessary to make the result acceptable as input to read/1. write_canonical/1, writeq/1, and portray_clause/1 select this.
Any output produced by write_term/2 with the option quoted(true) will be in Normal Form C, as defined by Unicode. See Section 4.1.7.5 [ref-syn-syn-tok], page 56, for further details.

ignore_ops(Boolean)
If selected, then Term is written in standard functional notation instead of using operators. write_canonical/1 and display/1 select this.

portrayed(Boolean)
If selected, then user:portray/1 is called for each non-variable subterm. print/1 selects this.

variable_names(Names)  since release 4.3
Names should be a list of Name=Var pairs, where each Name is an atom indicating the name to be used if Var is a variable occurring in the written term.
This argument has the same form as the corresponding read_term/[2,3] option and provides a convenient and safe way to preserve variable names when writing a previously read term.

numbervars(Boolean)
If selected, then terms of the form '$VAR'(N) where N is an integer >= 0 are treated specially (see numbervars/3). print/1, write/1, writeq/1, and portray_clause/1 select this.
legacy_numbervars(Boolean)  since release 4.3
If selected, then terms of the form \'$VAR'(N)\' where N is an integer \(\geq 0\), an atom, or a code list, are treated specially, in a way consistent with versions prior to release 4.3, as follows.
If N is an integer \(\geq 0\), then the behavior is as for the numbervars/1 option. Otherwise the characters of the atom or code list are written instead of the term.
The preferred way to specify variable names is with the variable_names/1 option.
cycles(Boolean)
If selected, then the potentially cyclic term is printed in finite \(\oplus/2\) notation, as discussed for read_term/[2,3] (see Section 11.3.185 [mpg-ref-read_term], page 1163).
indented(Boolean)
If selected, then the term is printed with the same indentation as is used by portray_clause/1 and listing/[0,1].
max_depth(Depth)
Depth limit on printing. Depth is an integer. 0 (the default) means no limit.
quoted_charset(Charset)
Only relevant if quoted(true) holds. Charset should be a legal value of the quoted_charset Prolog flag, where it takes its default value from. write_canonical/1 selects the value portable. See Section 4.9.4 [ref-lps-flg], page 136.
float_format(Spec)
How to print floats. Spec should be an atom of the form \`~NC`, like one of the format/[2,3] character sequences for printing floats. The default is \`~H`.
priority(Prio)
The term is printed as if in the context of an associative operator of precedence Prio, where Prio is an integer. The default is 1200. See Section 4.1.5 [ref-syn-ops], page 47.

Description
This predicate subsumes the predicates that output terms except portray_clause/[1,2], which additionally prints a period and a newline, and removes module prefixes that are redundant wrt. the current type-in module.

During debugging, goals are written out by this predicate with options given by the debugger_print_options Prolog flag.
Top-level variable bindings are written out by this predicate with options given by the toplevel_print_options Prolog flag.
Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109), plus:

- instantiation_error
- type_error
- domain_error

in Options.

Examples
How certain options affect the output of write_term/2:

\[
\begin{align*}
| ?- & \text{write_term('a b', [quoted(true)]).} \\
& 'a b' \\
| ?- & \text{write_term(a+b, [ignore_ops(true)]).} \\
& +(a,b) \\
| ?- & \text{write_term(f('VAR'(2)), [numbervars(true)].)} \\
& f(C) \\
| ?- & \text{write_term(A+B, [variable_names([Hello=A, World=B])].)} \\
& \text{Hello+World}
\end{align*}
\]

If your intention is to name variables such as that generated by read_term/2 with the variable_names option, then this can be done by simply passing on the variable_names option to write_term/2:

\[
\begin{align*}
| ?- & \text{read_term(T, [variable_names(Names)]),} \\
& \text{write_term(T, [variable_names(Names),quoted(true)].),} \\
& \text{nl,} \\
& \text{fail.} \\
& : a(X, Y). \\
& a(X, Y) \\
& \text{no}
\end{align*}
\]

There is, currently, no documented way to write single-occurrence variables as _, e.g. as done by portray_clause/1 (see Section 11.3.157 [mpg-ref-portray_clause], page 1129).

See Also
Section 4.6.4 [ref-iou-tou], page 104, Section 4.9.4 [ref-lps-flg], page 136, user:portray/1.
11.3.254 writeq/[1,2]  

Synopsis
writeq(+Stream, +Term)
writeq(+Term)

Writes Term on Stream, quoting atoms.

Arguments
Stream  stream_object, must be ground
       A valid open Prolog stream, defaults to the current output stream.
Term    term

Description
writeq(Term) is equivalent to:

write_term(Term, [quoted(true),numbervars(true)])

Exceptions
Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 109).

See Also
Section 4.6.4 [ref-iou-tou], page 104.
11.3.255 zip/0

development

Synopsis

zip

Turns on the debugger in zip mode.

Description

zip/0 turns the debugger on and sets it to zip mode. Turning the debugger on in zip mode means that it will stop at the next spypoint encountered in the current execution. Until the spypoint is reached, it does not keep any information of the execution of the goal, and hence you will not be able to see the ancestors of the goal when you reach the spypoint.

The effect of this predicate can also be achieved by typing the letter z after a ^C interrupt (see Section 3.7 [Execution], page 26).

Exceptions

None.

See Also

Section 5.2 [Basic Debug], page 233.
12 C Reference Pages

12.1 Return Values and Errors
Many, but not all, C functions return one of the codes `SP_SUCCESS` for success, `SP_FAILURE` for failure, `SP_ERROR` if an error condition occurred, or if an uncaught exception was raised during a call from C to Prolog. If the value is `SP_ERROR`, then the macro `SP_errno` will return a value (an integer) describing the error condition.

The function `SP_error_message()` returns a pointer to the diagnostic message corresponding to a specified error number.

See Section 11.1.1 [mpg-ref-ove], page 899, for a description of the conventions observed in the Reference Pages for Prolog predicates. C function Reference Pages differ primarily in the synopsis. Also, the Reference Page for each C function documents its return values.

The following function annotations are used in the Reference Pages:

- **hook**: The function is user defined and is called in some specific context.
- **macro**: The function is defined as a C macro.
- **preinit**: It is only meaningful to call the function before initializing the Prolog engine.

12.2 Topical List of C Functions

12.2.1 C Errors

`SP_error_message()`

gets the corresponding error message from an error number obtained from `SP_error`

12.2.2 I/O

`SP_get_byte()`

gets a byte from a Prolog binary input stream

`SP_get_code()`

gets a character code from a Prolog input text stream

`SP_unget_byte()`

`SP_unget_code()`

Ungets a byte or character, respectively.

`SP_fprintf()`

`SP_printf()`

prints formatted output on a Prolog output text stream

`SP_put_byte()`

`SP_put_bytes()`

Writes one or more bytes to a Prolog output binary stream.
SP_put_code()
SP_put_codes()
SP_put_encoded_string()
Wrote one or more characters to a Prolog output text stream.

### 12.2.3 Exceptions

SP_exception_term()
fetches the Prolog term representing the most recently raised exception

SP_fail()
propagates failure to Prolog

SP_on_fault()
provide a scope for faults

SP_raise_exception()
propagates an exception to Prolog

SP_raise_fault()
raise a fault

### 12.2.4 Files and Streams

SP_fopen()
opens a file as a Prolog stream

SP_fclose()
closes a Prolog stream

SP_flush_output()
flushes output on a Prolog output stream

SP_load()
same as load_files/1

SP_create_stream()
makes a new Prolog stream

SP_restore()
same as restore/1

SP_set_user_stream_hook() 
SP_set_user_stream_post_hook() 
provide hooks for setting up standard streams

### 12.2.5 Foreign Interface

SP_atom_from_string()
returns the encoded string representing a Prolog atom

SP_atom_length()
returns the length of the encoded string representing a Prolog atom

SP_close_query()
closes a Prolog query opened from C by SP_open_query()
SP_cons_functor()
SP_cons_functor_array()
creates a Prolog compound term from C

SP_cons_list()
creates a Prolog list from C

SP_cut_query()
commits to the current solution of a Prolog query opened from C by SP_open_query()

SP_define_c_predicate()
defines a Prolog predicate linked to a C function

SP_exception_term()
returns the Prolog term to C corresponding to the most recent Prolog error

SP_get_address()
fetches an integer representing a pointer in an SP_term_ref

SP_get_arg()
fetches a specified argument of a compound term in an SP_term_ref

SP_get_atom()
fetches an atom from an SP_term_ref

SP_get_current_dir()
obtain name of current working directory

SP_get_float()
fetches a floating point number from an SP_term_ref

SP_get_functor()
fetches the name and arity of a term in an SP_term_ref

SP_get_integer()
fetches an integer in an SP_term_ref

SP_get_integer_bytes()
fetches an arbitrarily sized integer in an SP_term_ref

SP_get_list()
fetches the head and tail of a list in an SP_term_ref

SP_get_list_codes()
fetches a code list in an SP_term_ref

SP_get_list_n_codes()
fetches the first part of a code list in an SP_term_ref

SP_get_list_n_bytes()
fetches the first part of a byte list in an SP_term_ref

SP_get_number_codes()
fetches a number encoded as a code list in an SP_term_ref

SP_get_string()
fetches the encoded string representing a Prolog atom in an SP_term_ref
SP_next_solution()  
gets the next solution, if any, to an open Prolog query

SP_open_query()  
opens a Prolog query from C

SP_pred()  
fetches an identifier for a Prolog predicate

SP_predicate()  
fetches an identifier a Prolog predicate

SP_put_address()  
assigns a pointer to an SP_term_ref

SP_put_atom()  
assigns an atom to an SP_term_ref

SP_put_float()  
assigns a floating point number to an SP_term_ref

SP_put_functor()  
assigns a new compound term to an SP_term_ref

SP_put_integer()  
assigns an integer to an SP_term_ref

SP_put_integer_bytes()  
assigns an arbitrarily sized integer to an SP_term_ref

SP_put_list()  
assigns a new list to an SP_term_ref

SP_put_list_codes()  
assigns a code list to an SP_term_ref

SP_put_list_n_codes()  
assigns the first part of a code list to an SP_term_ref

SP_put_list_n_bytes()  
assigns the first part of a byte list to an SP_term_ref

SP_put_number_codes()  
assigns a number encoded as a code list to an SP_term_ref

SP_put_string()  
assigns the atom represented by an encoded string to an SP_term_ref

SP_put_term()  
assigns the value of an SP_term_ref to another SP_term_ref

SP_put_variable()  
assigns a Prolog variable to an SP_term_ref

SP_query()  
makes a determinate query to a Prolog predicate, committing to the solution
SP_query_cut_fail()
  makes a determinate query to a Prolog predicate for side effects only

SP_read_from_string()
  assigns a Prolog term read from a string to an SP_term_ref

SP_set_current_dir()
  set name of current working directory

SP_string_from_atom()
  returns a null-terminated string corresponding to a Prolog atom

12.2.6 Initialization

SP_deinitialize()
  shuts down the Prolog engine

SP_force_interactive()  preinit
  consider standard streams to be interactive streams, even if they appear not to
  be TTY streams

SP_initialize()  macro
  initializes the Prolog engine

SP_set_argv()
  sets the argv Prolog flag.

SP_set_user_stream_hook()  preinit
SP_set_user_stream_post_hook()  preinit
  provide hooks for setting up standard streams

SU_initialize()  hook
  called before initializing the Prolog engine in applications built with
  --userhook

12.2.7 Memory Management

SP_calloc()
  Allocates memory for an array of elements, and clears the allocated memory.

SP_foreign_stash()  macro
  provide a memory location unique to the current foreign resource instance

SP_free()
  Deallocate a piece of memory.

SP_malloc()
  Allocates a piece of memory.

SP_mutex_lock()
  Locks a mutex.

SP_mutex_unlock()
  Unlocks a mutex.

SP_realloc()
  Changes the size of an allocated piece of memory.
SP_register_atom() prevents an atom from being discarded by atom garbage collection even if not referenced by Prolog code.

SP_strdup() makes a copy of a string in allocated memory.

SP_unregister_atom() enables an atom to be discarded during atom garbage collection if not referenced by Prolog code.

12.2.8 Signal Handling

SP_signal() install a signal handler.

SP_event() schedules a function for execution in the main thread in contexts where queries cannot be issued.

12.2.9 Terms in C

SP_compare() compares two terms using Prolog's standard term order.

SP_new_term_ref() returns an SP_term_ref, which can be used to hold a Prolog term in C.

SP_unify() unifies two Prolog terms.

12.2.10 Type Tests

SP_is_atom() tests whether an SP_term_ref contains an atom.

SP_is_atomic() tests whether an SP_term_ref contains an atomic term.

SP_is_compound() tests whether an SP_term_ref contains a compound term.

SP_is_float() tests whether an SP_term_ref contains a floating point number.

SP_is_integer() tests whether an SP_term_ref contains a Prolog integer.

SP_is_list() tests whether an SP_term_ref contains a list cell.

SP_is_number() tests whether an SP_term_ref contains an integer or a floating point number.

SP_is_variable() tests whether an SP_term_ref contains a Prolog variable.
SP_term_type() returns the type of the term in an SP_term_ref

12.3 API Functions

The following reference pages, alphabetically arranged, describe the SICStus Prolog API functions.
12.3.1 SP_atom_from_string()

Synopsis

#include <sicstus/sicstus.h>

SP_atom
SP_atom_from_string(char const *str);

Finds the Prolog atom whose characters are encoded by str.

Arguments

str The characters comprising the atom.

Return Value

The SP_atom, if str is a valid internal character encoding, and 0 otherwise.

See Also

Section 6.4.1 [Creating and Manipulating SP_term_refs], page 301.
12.3.2 SP_atom_length()

Synopsis

```c
#include <sicstus/sicstus.h>

size_t
SP_atom_length(SP_atom atom);
```

Obtains the length of the encoded string representing a Prolog atom.

Arguments

`atom` The atom to inspect.

Return Value

The length if `atom` is valid, and 0 otherwise.

Description

Same as `strlen(SP_string_from_atom(a))`, but runs in \(O(1)\) time.

See Also

Section 6.4.1 [Creating and Manipulating SP_term_refs], page 301.
12.3.3 SP_calloc()

Synopsis

#include <sicstus/sicstus.h>

void *
SP_calloc(size_t nmemb,
         size_t size);

Allocates a block of at least \texttt{size} \times \texttt{nmemb}. The first \texttt{size} \times \texttt{nmemb} bytes are set to zero.

Arguments

- \texttt{nmemb} : How many items to allocate.
- \texttt{size} : Size of each item.

Return Value

The pointer, if allocation was successful, otherwise \texttt{NULL}.

See Also

Section 6.4.7.1 [OS Memory Management], page 305.
12.3.4 SP_close_query()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_close_query(SP_qid query);
```

Discard the current solution to the given query, and close it.

Arguments

query The query, created by SP_open_query().

Return Value

SP_SUCCESS for success, SP_ERROR if an error condition occurred.

Description

This will discard the choices created since the corresponding SP_open_query(), and then backtrack into the query, throwing away any current solution, like the goal !, fail. The given argument does not have to be the innermost open query; any open queries in its scope will also be closed.

See Also

Section 6.5.2 [Finding Multiple Solutions of a Call], page 307.
12.3.5 SP_compare()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_compare(SP_term_ref x,
    SP_term_ref y)
```

Compares two terms.

Arguments

- `x` The one term to compare
- `y` The other term to compare

Return Value

-1 if `x` @< `y`, 0 if `x` == `y`, and 1 if `x` @> `y`.

See Also

Section 4.8.8 [ref-lte-cte], page 130, Section 6.4.6 [Unifying and Comparing Terms], page 305.
12.3.6 SP_cons_functor()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_cons_functor(SP_term_ref term,
                 SP_atom name,
                 int arity,
                 SP_term_ref arg, ...);
```

Assigns to `term` a reference to a compound term whose arguments are the values of `arg`... If `arity` is 0, assigns the Prolog atom whose canonical representation is `name`. This is similar to calling `../2` with the first argument unbound and the second argument bound.

Arguments

- `term` The SP_term_ref to be assigned
- `name` The name of the functor
- `arity` The arity of the functor
- `arg`... The arguments

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.7 SP_cons_functor_array()

Synopsis

#include <sicstus/sicstus.h>

int
SP_cons_functor_array(SP_term_ref term,
                      SP_atom name,
                      int arity,
                      SP_term_ref *arg);

Assigns to term a reference to a compound term whose arguments are the elements of arg. If arity is 0, assigns the Prolog atom whose canonical representation is name. This is similar to calling =../2 with the first argument unbound and the second argument bound.

Arguments

term   The SP_term_ref to be assigned
name   The name of the functor
arity  The arity of the functor
arg    The argument array

Return Value
Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also
Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.8 SP_cons_list()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_cons_list(SP_term_ref term,
             SP_term_ref head,
             SP_term_ref tail);
```

Assigns to `term` a reference to a Prolog list whose head and tail are the values of `head` and `tail`.

Arguments

- `term`  The SP_term_ref to be assigned
- `head`  The head of the new list
- `tail`  The tail of the new list

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.9 SP_create_stream()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_create_stream(
    void *user_data,
    void const *user_class,
    spio_t_simple_device_read *user_read,
    spio_t_simple_device_write *user_write,
    spio_t_simple_device_flush_output *user_flush_output,
    spio_t_simple_device_seek *user_seek,
    spio_t_simple_device_close *user_close,
    spio_t_simple_device_interrupt *user_interrupt,
    spio_t_simple_device_ioctl *user_ioctl,
    spio_t_bits create_stream_options,
    SP_stream **pstream);
```

Create a Prolog stream that will call user defined functions to perform stream operations.

Arguments

- **user_data** This is a pointer to arbitrary user specified data. It is passed to all user defined stream methods. It must not be NULL.
- **user_class** Arbitrary pointer. This is used with `SP_get_stream_user_data()`, which see.
- **user_read** If non-NULL then this is an input stream. See Section 12.3.106 [cpg-ref-user_read], page 1384, for details.
- **user_write** If non-NULL then this is an output stream. See Section 12.3.107 [cpg-ref-user_write], page 1386, for details.
  Note that both **user_read** and **user_write** can be specified, signifying a bidirectional stream.
- **user_flush_output** Will be called to flush output on the stream. Ignored if **user_write** is NULL.
  Can be NULL if the stream need not be flushed, e.g. if **user_write** always ensures that any output reaches its destination immediately. See Section 12.3.105 [cpg-ref-user_flush_output], page 1382, for details.
- **user_seek** Reserved, should be NULL.
- **user_close** Closes the stream. See Section 12.3.104 [cpg-ref-user_close], page 1380, for details.
- **user_interrupt** Reserved, should be NULL.
- **user_ioctl** Reserved, should be NULL.
- **args** Reserved, should be NULL.
create_stream_options
The following bits can be set:

SP_CREATE_STREAM_OPTION_BINARY
This is a binary stream. The user_read and user_write methods transfer bytes.

SP_CREATE_STREAM_OPTION_TEXT
This is a TEXT stream. The user_read and user_write methods transfer wide characters.

SP_CREATE_STREAM_OPTION_AUTOFLUSH
After writing to this stream prolog predicates will do a flush_output/1. In essence this ensures that the stream behaves as if it were unbuffered.

SP_CREATE_STREAM_OPTION_INTERACTIVE
Treat this stream as an interactive stream. Implies SP_CREATE_STREAM_OPTION_AUTOFLUSH.

SP_CREATE_STREAM_OPTION_EOF_ON_EOF
SP_CREATE_STREAM_OPTION_RESET_ON_EOF
These correspond to the open/4 options eof_action(eof) and eof_action(reset) respectively. The default is to give an error if reading after reaching end of file.

Exactly one of SP_CREATE_STREAM_OPTION_BINARY and SP_CREATE_STREAM_OPTION_TEXT must be set.

pstream This is assigned to the created SICStus stream on success. It should be closed with SP_fclose() or close/[1,2].

Return Value
On success, *pstream is assigned, and SPIO_S_NOERR or some other success code is returned. You should use the SPIO_FAILED() macro to determine if the return value signifies failure or success.

See Also
Section 6.6.2 [Defining a New Stream], page 313.
12.3.10 SP_cut_query()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_cut_query(SP_qid query);
```

Commit to the current solution to the given query, and close it.

Arguments

`query` The query, created by `SP_open_query()`.

Return Value

`SP_SUCCESS` for success, `SP_FAILURE` for failure, `SP_ERROR` if an error condition occurred.

Description

This will discard the choices created since the corresponding `SP_open_query()`, like the goal `!`. The current solution is retained in the arguments until backtracking into any enclosing query. The given argument does not have to be the innermost open query; any open queries in its scope will also be cut.

See Also

Section 6.5.2 [Finding Multiple Solutions of a Call], page 307.
### 12.3.11 SP_define_c_predicate()

#### Synopsis

```c
#include <sicstus/sicstus.h>

typedef int
SP_CPredFun(SP_term_ref goal,
void *stash);

int
SP_define_c_predicate(char *name,
    int arity,
    char *module,
    SP_CPredFun *proc,
    void *stash);
```

Defines a Prolog predicate such that when the Prolog predicate is called it will call a C function with a term corresponding to the Prolog goal.

#### Arguments

- **name**: The predicate name.
- **arity**: The predicate arity.
- **module**: The predicate module name.
- **proc**: The function.
- **stash**: See below.

#### Return Value

Nonzero on success, and 0 otherwise.

#### Description

The Prolog predicate `module:name/arity` will be defined (the module `module` must already exist). The `stash` argument can be anything and is simply passed as the second argument to the C function `proc`.

The C function should return `SP_SUCCESS` for success and `SP_FAILURE` for failure. The C function may also call `SP_fail()` or `SP_raise_exception()` in which case the return value will be ignored.

#### Examples

Here is an end-to-end example of the above:

```prolog
% square.pl

foreign_resource(square, [init(square_init)]).

:- load_foreign_resource(square).
```
// square.c

#include <sicstus/sicstus.h>

static int square_it(SP_term_ref goal, void *stash)
{
    SP_integer arg1;
    SP_term_ref tmp = SP_new_term_ref();
    SP_term_ref square_term = SP_new_term_ref();

    // goal will be a term like square(42,X)
    SP_get_arg(1,goal,tmp);  // extract first arg
    if (!SP_get_integer(tmp,&arg1))
        return SP_FAILURE;  // type check first arg
    SP_put_integer(square_term, arg1*arg1);
    SP_get_arg(2,goal,tmp);  // extract second arg

    // Unify output argument.
    // SP_put_integer(tmp,...) would *not* work!
    return (SP_unify(tmp, square_term) ? SP_SUCCESS : SP_FAILURE);
}

void square_init(int when)
{
    (void)when;  // unused
    // Install square_it as user:square/2
    SP_define_c_predicate("square", 2, "user", square_it, NULL);
}

% splfr square.pl square.c
% sicstus -f -l square
% compiling /home/matstc/tmp/square.pl...
% loading foreign resource /home/matstc/tmp/square.so in module user
% compiled /home/matstc/tmp/square.pl in module user, 0 msec 816 bytes
SICStus 4.7.0 ...
Licensed to SICS
| ?- square(4711, X).
X = 22193521 ?
yes
| ?- square(not_an_int, X).
no

See Also
See Section 6.2 [Calling C from Prolog], page 291.
12.3.12 SP_deinitialize()

Synopsis

```c
#include <sicstus/sicstus.h>

void
SP_deinitialize(void);
```

Shuts down the Prolog engine.

Description

SP_deinitialize() will make a best effort to restore the system to the state it was in at the time of calling SP_initialize(). This involves unloading foreign resources, shutting down the emulator, and deallocating memory used by Prolog.

SP_deinitialize() is idempotent i.e. it is a no-op unless SICStus has actually been initialized.

See Also

Section 6.7.4.1 [Initializing the Prolog Engine], page 330.
12.3.13 SP_error_message()

Synopsis

```
#include <sicstus/sicstus.h>
char const *
SP_error_message(int errnum);
```

Obtains a pointer to the diagnostic message corresponding to a specified error number.

Arguments

`errnum`  The error number.

Return Value

A pointer to the diagnostic message.

See Also

Section 6.1 [CPL Notes], page 290.
12.3.14 SP_event()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_event(int (*func)(void*), void *arg)
```

Schedules a function for execution in the main thread in contexts where queries cannot be issued.

Arguments

- **func**: The function to schedule for execution.
- **arg**: Its argument.

Return Value

Nonzero on success, and 0 otherwise.

Description

If you wish to call Prolog back from a signal handler that has been installed with `SP_signal` or a thread other than the thread that called `SP_initialize()`, that is, the main thread, you cannot use `SP_query()` etc. directly. The call to Prolog has to be delayed until such time that the Prolog execution can accept an interrupt and the call has to be performed from the main thread (the Prolog execution thread). This function serves this purpose, and installs `func` to be called from Prolog (in the main thread) when the execution can accept a callback.

A queue of functions, with corresponding arguments, is maintained; that is, if several calls to `SP_event()` occur before Prolog can accept an interrupt, the functions are queued and executed in turn at the next possible opportunity. A `func` installed with `SP_event()` will not be called until SICStus is actually running. One way of ensuring that all pending functions installed with `SP_event()` are run is to call, from the main thread, some dummy goal, such as,

```c
SP_query_cut_fail(SP_predicate("true",0,"user"));
```

While `SP_event()` is safe to call from any thread, it is not safe to call from arbitrary signal handlers. If you want to call `SP_event()` when a signal is delivered, you need to install your signal handler with `SP_signal()`.

Note that `SP_event()` is one of the very few functions in the SICStus API that can safely be called from another thread than the main thread.

Depending on the value returned from `func`, the interrupted Prolog execution will just continue (`SP_SUCCESS`) or backtrack (`SP_FAILURE` or `SP_ERROR`). An exception raised by `func`, using `SP_raise_exception()`, will be processed in the interrupted Prolog execution. If `func` calls `SP_fail()` or `SP_raise_exception()` the return value from `func` is ignored.
and handled as if `func` returned `SP_FAILURE` or `SP_ERROR`, respectively. In case of failure or exception, the event queue is flushed.

It is generally not robust to let `func` raise an exception or (even worse) fail. The reason is that not all Prolog code is written such that it gracefully handles being interrupted. If you want to interrupt some long-running Prolog code, it is better to let the event handler set a flag (in C) and let your Prolog code test the flag (using a foreign predicate) in some part of your code that is executed repeatedly.

**Examples**

How to install the predicate `user:event_pred/1` as the signal handler for `SIGUSR1` and `SIGUSR2` signals.

The function `signal_init()` installs the function `signal_handler()` as the primary signal handler for the signals `SIGUSR1` and `SIGUSR2`. That function invokes the predicate as the actual signal handler, passing the signal number as an argument to the predicate.

```c
SP_pred_ref event_pred;

static int signal_event(void *handle)
{
    int signal_no = (int) handle;
    SP_term_ref x=SP_new_term_ref();
    int rc;

    SP_put_integer(x, signal_no); // Should not give an error
    rc = SP_query_cut_fail(event_pred, x);
    if (rc == SP_ERROR && SP_exception_term(x))
        SP_raise_exception(x); // Propagate any raised exception
    return rc;
}

static void signal_handler(int signal_no)
{
    SP_event(signal_event, (void *)signal_no);
}

void signal_init(void)
{
    event_pred = SP_predicate("prolog_handler",1,"user");

    SP_signal(SIGUSR1, signal_handler);
    SP_signal(SIGUSR2, signal_handler);
}
```
See Also
Section 6.5.4 [Calling Prolog Asynchronously], page 309, SP_signal(), SP_fail(), SP_raise_exception().
12.3.15 SP_exception_term()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_exception_term(SP_term_ref term);
```

Retracts the current pending exception term, if it exists, and assigns it to `term`.

Arguments

`term` The SP_term_ref to assign.

Return Value

1 if an exception term was retracted and assigned, and 0 otherwise.

See Also

Section 6.5.5 [Exception Handling in C], page 310.
12.3.16 SPExpand_file_name()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SPExpand_file_name(
    char const *relpath,
    char const *cwd,
    spio_t_bits options,
    char **pabspath);
```

Expand a file name into an absolute path.

Arguments

relpath
The relative path to expand. It should be an encoded string. The path is subject to syntactic rewriting, as if by absolute_file_name/2.

cwd
If the relpath is a relative path, it is expanded relative to cwd, unless cwd is NULL. If cwd is NULL, a relative relpath is expanded relative to the SICStus working directory (as returned by SP_get_current_dir()).

options
The following option bits can be set:

- **SP_EXPAND_FILE_NAME_OPTION_DIR**
  The relpath is expanded as a directory, i.e. *pabspath will be slash terminated.

- **SP_EXPAND_FILE_NAME_OPTION_NO_CWD**
  An error is returned if the relpath is not an absolute path after syntactic rewriting.

- **SP_EXPAND_FILE_NAME_OPTION_NO_ENV**
  Do not expand system properties and environment variables during syntactic rewriting.

- **SP_EXPAND_FILE_NAME_OPTION_NO_HOME**
  Do not expand ‘~’ and ‘~user’ during syntactic rewriting.

- **SP_EXPAND_FILE_NAME_OPTION_ROOT_SLASH**
  If the expanded value would refer to the root directory, return a slash terminated absolute path, as if SP_EXPAND_FILE_NAME_OPTION_DIR had been set. By default, an error is returned if the expanded absolute path would refer to a root directory and SP_EXPAND_FILE_NAME_OPTION_DIR is not set.

- **SP_EXPAND_FILE_NAME_OPTION_ROOT_DOT**
  If the expanded value would refer to the root directory, return an absolute path terminated with ‘/.’. By default, an error is returned
if the expanded absolute path would refer to a root directory and
SP_EXPAND_FILE_NAME_OPTION_DIR is not set.

\texttt{pabspath} On success, \texttt{*pabspath} is set to the expanded path. This value is allocated with \texttt{SP_malloc()} and should be freed with \texttt{SP_free()}.

**Return Value**

On success, \texttt{*pabspath} is set to the expanded path and \texttt{SPIO_S_NOERR} or some other success code is returned.

On failure, an error code is returned.

**See Also**

Section 12.3.29 [cpg-ref-SP_get_current_dir], page 1298. See Section 4.5.2 [ref-fdi-syn], page 99, for a description of syntactic rewriting. Section 6.4.7.2 [OS File System], page 306.
12.3.17 SP_fail()

Synopsis

#include <sicstus/sicstus.h>

void
SP_fail(SP_term_ref term);

Fails in the scope of Prolog calling C.

Arguments

term The SP_term_ref whose value will be the exception term.

Description

This function is normally used in the context of a call from Prolog to C, and will cause Prolog to backtrack on return from the call.

Please note: this should only be called right before returning to Prolog.

See Also

Section 6.5.5 [Exception Handling in C], page 310.
12.3.18 SP_fclose()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_fclose(
    SP_stream *stream,
    spio_t_bits close_options);
```

Close the stream.

Arguments

- **stream**
  The stream to close unless the `SP_FCLOSE_OPTION_USER_STREAMS` is set, see below.

- **close_options**
  The following bits can be set:

  - **SP_FCLOSE_OPTION_READ**
    Close the specified directions. If neither of these options is specified, the stream is closed in all opened directions, i.e. as if both options were specified. If the stream is not opened in a direction specified by an option, that option is ignored.
    Note that it is possible to close only one direction of a bidirectional stream. The return value will tell whether the stream is still open; see below.

  - **SP_FCLOSE_OPTION_WRITE**
    Close the specified directions. If neither of these options is specified, the stream is closed in all opened directions, i.e. as if both options were specified. If the stream is not opened in a direction specified by an option, that option is ignored.
    Note that it is possible to close only one direction of a bidirectional stream. The return value will tell whether the stream is still open; see below.

  - **SP_FCLOSE_OPTION_FORCE**
    Close the specified direction forcibly, i.e. without flushing buffers etc. This also ensures that the close finishes quickly, i.e. does not block.

  - **SP_FCLOSE_OPTION_NO_FSYNC**
    Do not use OS `fclose()` or similar when closing a stream in write direction, i.e. do not wait for written data to reach the disk.
    By default, closing a stream will try to ensure that all written data have been stored on disk before the call returns. This makes stream handling more robust, e.g. if the process crashes shortly after closing the stream. However, waiting for data to reach the disk is sometimes very slow (e.g. on some Linux configurations), in which case this flag can be used to speed things up, at the cost of somewhat reduced robustness.

  - **SP_FCLOSE_OPTION_NONBLOCKING**
    You should avoid using this option.
    Pass non-blocking option to lower level routines, including the call to `SP_flush_output()` that is issued when non-forcibly closing write direction.
One possible use for this option is to perform a best effort close, which falls back to using \texttt{SP_FCLOSE_OPTION_FORCE} only if ordinary close would block.

\texttt{SP_FCLOSE_OPTION_USER_STREAMS}

In this case the stream should not be a stream but instead be the user class of a user defined stream. When this option is passed, all currently opened streams of that class is closed, using the remaining option flags. E.g. to close all user defined streams of class my\_class in the read direction only do: \texttt{SP\_fclose((SP\_stream*)my\_class,SP\_FCLOSE\_OPTION\_USER\_STREAMS|SP\_FCLOSE\_OPTION\_READ)}.

**Return Value**

On success, all specified directions has been closed. Since some direction may still be open, there are two possible return values on success:

\textbf{SPIO\_S\_NOERR}

The stream is still valid, some direction is still not closed.

\textbf{SPIO\_S\_DEALLOCATED}

The stream has been deallocated and cannot be used further. All directions have been closed.

On failure, returns a SPIO error code. Error codes with special meaning for \texttt{SP\_fclose()} are the same as for \texttt{SP\_flush\_output()}, which see. Other error codes may also be returned.

**See Also**

Section 12.3.19 [cpg-ref-SP\_flush\_output], page 1286. Section 6.6.1 [Prolog Streams], page 311.
12.3.19 SP_flush_output()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_flush_output(
    SP_stream *stream,
    spio_t_bits flush_options);
```

Ensure that all buffered data reaches its destination.

Arguments

- `stream` The stream to flush. This stream should be open for writing.
- `flush_options` The following bits can be set:

  - `SP_FLUSH_OUTPUT_OPTION_NO_FSYNC`
    - If this is set, flush will not wait for data to reach the disk. See the `SP_fclose()` option `SP_FCLOSE_OPTION_NO_FSYNC` for more information.

  - `SP_FLUSH_OUTPUT_OPTION_NONBLOCKING`
    - If this is set, the function should return quickly or with a `SPIO_E_WOULD_BLOCK` code.
    - Can return `SPIO_E_NOT_SUPPORTED` if the stream cannot support non-blocking flush.

  - `SP_FLUSH_OUTPUT_OPTION_AUTOFLUSH`
    - Only flush stream if it has AUTOFLUSH enabled.

Return Value

On success, all buffered data should have been written and `SPIO_S_NOERR` or some other success code returned.

On failure, returns a SPIO error code. Error codes with special meaning for `SP_flush_output()`:

- `SPIO_E_END_OF_FILE`
  - Returned if it is not possible to write more data onto the stream, e.g. some underlying device has been closed.

- `SPIO_E_WOULD_BLOCK`
  - `SP_FLUSH_OUTPUT_OPTION_NONBLOCKING` was set but the operation would block.

- `SPIO_E_NOT_SUPPORTED`
  - Some unsupported option, e.g. `SP_FLUSH_OUTPUT_OPTION_NONBLOCKING`, was passed.
Other error codes may also be returned.

**See Also**

Section 6.6.1 [Prolog Streams], page 311.
12.3.20 SP_fopen()

Synopsis

#include <sicstus/sicstus.h>

spio_t_error_code
SP_fopen(
  char const *pathname,
  void *reserved,
  spio_t_bits options,
  SP_stream **pstream);

Opens a file and creates a SICStus stream reading and/or writing to it.

Arguments

pathname  The path to the file as an encoded string. It is expanded by SP_expand_file_name() unless the option SP_FOPEN_OPTION_NOEXPAND is specified, in which case the path must already have been expanded by SPExpand_file_name().

reserved  Reserved, should be NULL.

read_options  The following bits can be set:

SP_FOPEN_OPTION_READ
  Open the file for reading. The file must exist.

SP_FOPEN_OPTION_WRITE
  Open the file for writing. The file is overwritten if it exists. The file is created if it does not exist.

SP_FOPEN_OPTION_APPEND
  Open the file for writing but start writing at the end of the file if it exists. The file is created if it does not exist.

SP_FOPEN_OPTION_BINARY
  Open the file as a binary (byte) stream.

SP_FOPEN_OPTION_TEXT
  Open the file as a text stream. The default character encoding is Latin 1 (i.e. the 8 bit subset of Unicode). The default end of line convention is OS specific.

SP_FOPEN_OPTION_AUTOFLUSH
  After writing to this stream, Prolog predicates will do a flush_output/1. In essence this ensures that the stream behaves as if it were unbuffered.

SP_FOPEN_OPTION_INTERACTIVE
  Treat this stream as an interactive stream. Implies SP_CREATE_STREAM_OPTION_AUTOFLUSH.
If this is set, flush and close of the stream will not wait for data to reach the disk. See the `SP_fclose()` option `SP_FCLOSE_OPTION_NO_FSYNC` for more information.

The `pathname` has already been expanded with `SP_expand_file_name()` or something similar. This implies that `pathname` is an absolute path. If this option is not specified, `pathname` is expanded with `SP_expand_file_name()` before use.

On successful return, `*pstream` will be set to the created stream.

**Return Value**

On success, `*pstream` will be set to the created stream and `SPIO_S_NOERR` or some other success code returned.

On failure, some SPIO failure code will be returned. Error codes with special meaning for `SP_fopen()`:

- `SPIO_E_FILE_NOT_FOUND`
  The file does not exist.

- `SPIO_E_FILE_ACCESS`
  Insufficient permissions to open or create the file.

- `SPIO_E_OPEN_ERROR`
  Generic error during open.

Other error codes may also be returned.

**See Also**

Section 6.6.1 [Prolog Streams], page 311.
12.3.21 SP_foreign_stash()

Synopsis

#include <sicstus/sicstus.h>

void *
SP_foreign_stash();

Obtains a storage location that is unique to the calling foreign resource.

Return Value
The location, initially set to NULL.

Description
A dynamic foreign resource that is used by multiple SICStus runtimes in the same process may need to maintain a global state that is kept separate for each SICStus runtime. Each SICStus runtime maintains a location (containing a void*) for each foreign resource. A foreign resource can then access this location to store any data that is specific to the calling SICStus runtime.

You can use SP_foreign_stash() to get access to a location, where the foreign resource can store a void*. Typically this would be a pointer to a C struct that holds all information that need to be stored in global variables. This struct can be allocated and initialized by the foreign resource init function, it should be deallocated by the foreign resource deinit function.

SP_foreign_stash() is only available for use in dynamic foreign resources.

Examples
The value returned by SP_foreign_stash() is only valid until the next SICStus API call. The correct way to initialize the location pointed at by SP_foreign_stash() is therefore:

```
struct my_state {...};

init_my_foreign_resource(...) {
    struct my_state *p = SP_malloc(sizeof(struct my_state));
    (*SP_foreign_stash()) = (void*)p;
}
```

The following example is incorrect; SP_malloc() may be called between the time SP_foreign_stash() is called and the time its return value is used:

```
// WRONG
(*SP_foreign_stash()) = SP_malloc(sizeof(struct my_state));
```

See Also
Section 6.4.7.3 [OS Threads], page 306.
12.3.22 SP_fprintf()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_fprintf(
    SP_stream *stream,
    char const *fmt, ...);
```

Formatted output on the Prolog stream `stream`.

Arguments

- `stream` The stream. Must be a text stream open for output.
- `fmt` The format string. This uses the same syntax as the C library `printf` functions.
- `...` The data to format.

Return Value

On success, all data has been written and `SPIO_S_NOERR` or some other success code returned.

On failure, returns an error code without transferring any data. Error codes with special meaning for `SP_fprintf()`:

- `SPIO_E_PARAMETER_ERROR` The underlying C library function reported an error while formatting the string.

Other error codes may also be returned.

Description

First the formatting operation will be performed. The resulting string will be assumed to be in internal encoding, and will then be output using the `SP_put_encoded_string()` function. This means e.g. that the `%c` printf conversion specification can only be used for ASCII characters, and the strings included using a `%s` specification should also be encoded strings.

See Also

Section 6.6.1 [Prolog Streams], page 311.
12.3.23 SP_free()

Synopsis

```c
#include <sicstus/sicstus.h>

void
SP_free(void *ptr);
```

Disposes of the block referenced by `ptr`, which must have been obtained by a call to `SP_malloc()` or `SP_realloc()`, and must not have been released by a call to `SP_free()` or `SP_realloc()`.

Arguments

`ptr` Block to dispose of.

See Also

See Section 6.4.7.1 [OS Memory Management], page 305.
12.3.24 SP_get_address()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_get_address(SP_term_ref term,
               void **p);
```

Assigns to *p the pointer that corresponds to a Prolog integer

Arguments

term The SP_term_ref holding the value

p The location to assign

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.25 SP_get_arg()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_get_arg(int i,
                SP_term_ref term,
                SP_term_ref arg);

Assigns to `arg` the `i`-th argument of a compound term. This is similar to calling `arg/3`.

Arguments

- `i` The (one-based) argument number
- `term` The SP_term_ref holding the compound term
- `arg` The SP_term_ref to be assigned

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.26 SP_get_atom()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_get_atom(SP_term_ref term,
            SP_atom *a);
```

Assigns to *a the canonical representation of a Prolog atom

Arguments

- `term`: The SP_term_ref holding the value
- `a`: The location to assign

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.27  SP_get_byte()

Synopsis

#include <sicstus/sicstus.h>

spio_t_error_code
SP_get_byte(
    SP_stream *stream);

Read a byte from a binary stream.

Arguments

stream  The stream. Must be a binary stream open for input.

Return Value

On success, the byte just read will be returned, cast to a spio_t_error_code. The value returned on successful return will never be negative.

On failure, returns an error code, recognizable with SPIO_FAILED(). Error codes are always negative.

Description

Note that SP_get_byte() is implemented as a macro and may evaluate the stream argument more than once.

See Also

Section 12.3.28 [cpg-ref-SP_get_code], page 1297. Section 6.6.1 [Prolog Streams], page 311.
12.3.28  SP_get_code()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_get_code(
    SP_stream *stream);
```

Read a character code from a text stream.

Arguments

`stream`  The stream. Must be a text stream open for input.

Return Value

On success, the character just read will be returned, cast to a `spio_t_error_code`. The value returned on successful return will never be negative.

On failure, returns an error code, recognizable with `SPIO_FAILED()`. Error codes are always negative.

Description

Note that `SP_get_code()` is implemented as a macro and may evaluate the `stream` argument more than once.

See Also

Section 12.3.27 [cpg-ref-SP_get_byte], page 1296. Section 6.6.1 [Prolog Streams], page 311.
12.3.29 SP_get_current_dir()

Synopsis

```
#include <sicstus/sicstus.h>

char *
SP_get_current_dir(void);
```

Obtains an encoded string containing the absolute, slash (/) terminated, path to the current working directory. The return value is allocated with SP_malloc and should be freed with SP_free.

Return Value

The string on success and NULL on error.

See Also

Section 12.3.92 [cpg-ref-SP_set_current_dir], page 1367. Section 6.4.7.2 [OS File System], page 306.
12.3.30 SP_get DISPATCH()  

Synopsis  

```c
#include <sicstus/sicstus.h>

SICSTUS_API_STRUCT_TYPE *
SP_get_dispatch(void *reserved);
```

Arguments  

`reserved`  
Reserved, should be NULL.

Return Value  

Returns the dispatch vector of the SICStus runtime.

Description  

This function can be called from any thread.

This function is special in that it is not accessed through the dispatch vector; instead, it is exported in the ordinary manner from the SICStus runtime dynamic library (`sprt4-7-0.dll` under Windows and, typically, `libsprt4-7-0.so` under UNIX).

The address of this function is typically obtained by linking to the SICStus runtime library or, when using multiple SICStus runtimes, by a call to `SP_load_sicstus_run_time()`.

See Also  

Chapter 8 [Multiple SICStus Runtimes], page 349.
12.3.31 SP_get_float()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_get_float(SP_term_ref term,
             double *f);
```

Assigns to *f the float that corresponds to a Prolog number.

If the term is an integer that does not fit in a `double`, then the call will fail.

Arguments

- `term` The SP_term_ref holding the value
- `f` The location to assign

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.32 SP_get_functor()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_get_functor(SP_term_ref term,
               SP_atom *name,
               int *arity);
```

Assigns to *name and *arity the canonical representation and arity of the principal functor of a Prolog compound term. If the value of term is an atom, then that atom is assigned to *name and 0 is assigned to *arity. This is similar to calling functor/3 with the first argument bound to a compound term or an atom and the second and third arguments unbound.

Arguments

- `term` The SP_term_ref holding the value
- `name` The location to assign to the functor name
- `arity` The location to assign to the functor arity

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.33 SP_get_integer()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_get_integer(SP_term_ref term,
               SP_integer *i);
```

Assigns to *i the integer that corresponds to a Prolog number. The value must fit in *i for the operation to succeed.

Arguments

term The SP_term_ref holding the value

i The location to assign

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.34 SP_get_integer_bytes()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_get_integer_bytes(SP_term_ref term, 
                      void *buf, 
                      size_t *pbuf_size, 
                      int native);
```

Extracts from term an an arbitrarily sized integer.

Arguments

- **term**: The SP_term_ref holding the integer
- **buf**: The buffer receiving the integer
- **pbuf.size**: Should point at the size of buf
- **native**: See the description below

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

Description

In the following, assume that the integer referred to by term requires a minimum of size bytes to store (in twos-complement representation).

1. If term does not refer to a Prolog integer, zero is returned and the other arguments are ignored.
2. If *pbuf.size is less than size, then *pbuf.size is updated to size and zero is returned.
   The fact that *pbuf.size has changed can be used to distinguish insufficient buffer size from other possible errors. By calling SP_get_integer_bytes() with *pbuf.size set to zero, you can determine the buffer size needed; in this case, buf is ignored.
3. *pbuf.size is set to size.
4. If native is zero, buf is filled with the twos complement representation of the integer, with the least significant bytes stored at lower indices in buf. Note that all of buf is filled, even though only size bytes was needed.
5. If native is non-zero, buf is assumed to point at a native *pbuf.size byte integral type. On most platforms, native integer sizes of two (16-bit), four (32 bit) and eight (64 bytes) bytes are supported. Note that *pbuf.size == 1, which would correspond to signed char, is not supported with native.
6. If an unsupported size is used with native, zero is returned.

Examples

The following example gets a Prolog integer into a (presumably 64 bit) long long C integer.
\begin{verbatim}
long long x; // C99, GCC supports this
size_t sz = sizeof x;
if (!SP_get_integer_bytes(tr, &x, &sz, 1)) // 1 for native
  .. error handling ..
  .. use x .. // sz may have decreased
\end{verbatim}

See Also
Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.35 SP_get_list()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_get_list(SP_term_ref term,
             SP_term_ref head,
             SP_term_ref tail);
```

Assigns to `head` and `tail` the head and tail of a Prolog list.

Arguments

term The SP_term_ref holding the list

head The SP_term_ref to be assigned the head

tail The SP_term_ref to be assigned the tail

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.36 SP_get_list_codes()

Synopsis

#include <sicstus/sicstus.h>

int
SP_get_list_codes(SP_term_ref term,
        char const **s);

Assigns to *s a zero-terminated array containing an encoded string that corresponds to the given Prolog code list. The array is subject to reuse by other support functions, so if the value is going to be used on a more than temporary basis, it must be moved elsewhere.

Arguments

term The SP_term_ref holding the code list
s The location to assign

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.37 SP_get_list_n_bytes()

Synopsis

#include <sicstus/sicstus.h>

int
SP_get_list_n_bytes(SP_term_ref term,
                     SP_term_ref tail,
                     size_t n,
                     size_t *w,
                     unsigned char *s);

Copies into the byte array s the initial elements of term, which should hold a list of integers in the range [0,255], so that at most n bytes are used. The number of bytes actually written is assigned to *w. tail is set to the remainder of the list. The array s must have room for at least n bytes.

Arguments

term The SP_term_ref holding the list
tail The SP_term_ref to be assigned the remainder of the list
n Max number of bytes to use
w Location to assign to number of bytes actually used
s The location to assign to the encoded string

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.38 SP_get_list_n_codes()

Synopsis

#include <sicstus/sicstus.h>

int
SP_get_list_n_codes(SP_term_ref term,
            SP_term_ref tail,
            size_t n,
            size_t *w,
            char *s);

Copies into \texttt{s} the encoded string representing the character codes in the initial elements of list \texttt{term}, so that at most \texttt{n} bytes are used. The number of bytes actually written is assigned to \texttt{*w}. \texttt{tail} is set to the remainder of the list. The array \texttt{s} must have room for at least \texttt{n} bytes.

Please note: The array \texttt{s} is never NUL-terminated. Any zero character codes in the list \texttt{term} will be converted to the overlong UTF-8 sequence \texttt{0xC0 0x80}.

Arguments

\begin{description}
\item \texttt{term} The SP\_term\_ref holding the code list
\item \texttt{tail} The SP\_term\_ref to be assigned the remainder of the list
\item \texttt{n} Max number of bytes to use
\item \texttt{w} Location to assign to number of bytes actually used
\item \texttt{s} The location to assign to the encoded string
\end{description}

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.39 SP_get_number_codes()

Synopsis

```c
#include <sicstus/sicstus.h>

int SP_get_number_codes(SP_term_ref term,
                         char const **s);
```

Assigns to *s a zero-terminated array of characters corresponding to the printed representation of a Prolog number. The array is subject to reuse by other support functions, so if the value is going to be used on a more than temporary basis, it must be moved elsewhere.

Arguments

term The SP_term_ref holding the number
s The location to assign to the array

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.40 SP_get_stream_counts()

Synopsis

#include <sicstus/sicstus.h>

spio_t_error_code
SP_get_stream_counts(
   SP_stream *stream,
   spio_t_offset *pitem_count,
   spio_t_offset *pnewline_count,
   spio_t_offset *pline_length,
   spio_t_bits options);

Obtain the stream counters.

Arguments

stream    The stream.

item_count
If pitem_count is NULL it is not used, otherwise it is used as follows.
On success, *pitem_count is assigned to the number of items read from an
input-only or bidirectional stream or with the number of items written to a
output-only stream.
For binary streams, an item is a byte, for text streams it is a character.

pnewline_count
If pnewline_count is NULL it is not used, otherwise it is used as follows.
On success *pnewline_count is assigned to the number of newlines read from an
input-only or bidirectional text stream or with the number of newlines written
to a output-only text stream.
For binary streams, *pnewline_count is set to zero.

pline_length
If pline_length is NULL it is not used, otherwise it is used as follows.
On success, *pline_length is assigned to the number of characters read on
the current line from an input-only or bidirectional text stream or with the
characters written on the current line to a output-only text stream.
For binary streams, *pline_length is set to zero.

options    The following bits can be set:

SP_GET_STREAM_COUNTS_OPTION_READ
Return the real input counts of a read-only or bidirectional stream.

SP_GET_STREAM_COUNTS_OPTION_WRITE
Return the real output counts of a write-only stream.
Currently, the call will fail with SPIO_E_NOT_SUPPORTED if the
stream is bidirectional and SP_GET_STREAM_COUNTS_OPTION_WRITE
is specified. This is because there is only one set of counters for
each stream and these are used to count in the input direction of bidirectional streams. This may be changed in a future release.

At most one of SP_GET_STREAM_COUNTS_OPTION_READ and SP_GET_STREAM_COUNTS_OPTION_WRITE can be specified. If neither is specified then default behavior is as follows

- If stream is interactive, a common set of counts shared by all interactive streams is returned.
- If stream is write-only, the output counts are returned.
- Otherwise, the stream is read-only or bidirectional and the input counts are returned.

**Return Value**

On success, SPIO_S_NOERR or some other success code is returned.

On failure, returns a SPIO error code. Error codes with special meaning for SP_get_stream_counts():

- SPI0_E_NOT_READ
  SP_GET_STREAM_COUNTS_OPTION_READ was specified but stream is not an input stream.
- SPI0_E_NOT_WRITE
  SP_GET_STREAM_COUNTS_OPTION_WRITE was specified but stream is not an output stream.
- SPI0_E_NOT_SUPPORTED
  SP_GET_STREAM_COUNTS_OPTION_WRITE was specified but stream is a bidirectional stream.

**Description**

There is only one set of counters for each stream. For a bidirectional stream, these counters only count in the input direction and the output direction does not affect the counts.

There is a common set of stream counters for all interactive streams. By default, these will be returned if stream is interactive instead of the real counts. This behavior can be changed with the options argument, see above.

**See Also**

Section 6.6.1 [Prolog Streams], page 311.
12.3.41 SP_get_stream_user_data()

Synopsis

```
#include <sicstus/sicstus.h>

spio_t_error_code
SP_get_stream_user_data(
    SP_stream *stream,
    void const *user_class,
    void **puser_data);
```

Get the user data of a user defined stream of a particular class.

Arguments

- **stream**
  - An arbitrary stream. It is legal, and often useful, to call `SP_get_stream_user_data()` on a stream even if it is not known whether the stream is in fact a user defined stream of a particular class.

- **puser_data**
  - On success, `puser_data` will be set to the `user_data` value used when the stream was created.

Return Value

- On success, `puser_data` is assigned and `SPIO_S_NOERR` or some other success code is returned.

- On failure, e.g. if the stream was not created with this `user_class`, an error code is returned.

Description

This function is used in order to recognize streams of a particular type (or class). At the same time as it verifies the type of stream it also returns the `user_data` which gives the caller a handle to the internal state of the user defined stream.

The following sample illustrates how all streams of a particular class can be found and closed. This function mimics the behavior of the `SP_FCLOSE_OPTION_USER_STREAMS` option to `SP_fclose`, see Section 12.3.18 [cpg-ref-SP_fclose], page 1284.
spio_t_error_code close_streams(void const *user_class, int force)
{
    spio_t_error_code ecode = SPIO_E_ERROR;
    SP_stream *stream;
    SP_stream *next_stream;
    void *user_data;
    spio_t_bits fclose_options = 0;

    if (force) fclose_options |= SP_FCLOSE_OPTION_FORCE;

    stream = NULL;    /* means start of list of stream */
    do
    {
        /* Note: We need to do this before closing stream */
        ecode = SP_next_stream(stream, &next_stream);
        if (SPIO_FAILED(ecode)) goto barf;

        if (stream != NULL)
        {
            if (SPIO_SUCCEEDED(SP_get_stream_user_data(stream, user_class, &user_data)))
            {
                /* This is the right class of stream, close it */
                ecode = SP_fclose(stream, fclose_options);
                if (SPIO_FAILED(ecode))
                {
                    if (!force) goto barf; /* ignore error if force */
                }
            }
        }
        stream = next_stream;
    }
    while (stream != NULL);

    return SPIO_S_NOERR;

    barf:
        return ecode;
    }

See Also
Section 12.3.9 [cpg-ref-SP_create_stream], page 1270. Section 6.6.2 [Defining a New Stream], page 313.
12.3.42 SP_get_string()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_get_string(SP_term_ref term,
              char const **s);
```

Assigns to *s a pointer to the encoded string representing the name of a Prolog atom. This string must not be modified by the calling function.

Arguments

term The SP_term_ref holding the value
s The location to assign

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.4 [Accessing Prolog Terms], page 304.
12.3.43 SP_getenv()

Synopsis

```
#include <sicstus/sicstus.h>

void *
SP_getenv(const char *name);
```

Retrieves the value of the system property, or if no such system property exists the environment variable, with the given name.

The memory for the returned value is managed by Prolog and should be freed with SP_free().

Arguments

- `name`: The name of the system property or environment variable.

Return Value

The value of the system property or environment variable, if set. `NULL` if neither a system property nor an environment variable of that name exists, or if an error occurs.

See Also

Section 4.17.1 [System Properties and Environment Variables], page 224.
12.3.44 SP_initialize()  

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_initialize(int argc,
               char **argv,
               SP_options *options);
```

Initializes the Prolog engine.

Arguments

- `argc`: The number of elements of the `argv` vector.
- `argv`: A vector of strings that can be accessed by `prolog_flag(argv,X)`. This argument is copied by `SP_initialize()` so it can be discarded by the caller. May be passed as NULL if `argc` is zero.
  Each entry should be an encoded string, i.e. encoded using ‘UTF-8’. This may not be the encoding used by the operating system when invoking `main()`. A better alternative is to pass zero for `argc`, NULL for `argv` and use `SP_set_argv()` to pass the `argv` entries.
- `options`: A pointer to an option block. In most cases it suffice to pass NULL.
  An option block can be initialized with `SP_OPTIONS_STATIC_INITIALIZER` and its `options` field set to point to a `SP_option` array. Each `SP_option` is a typed value. Currently the only type is `SP_option_type_system_property`, for setting initial system properties (see Section 4.17.1 [System Properties and Environment Variables], page 224).
  To pass the system properties `foo` and `bar`, with values `true` and `hello`, respectively, you would do something like this

```c
        ```
int res;
SP_options opts = SP_OPTIONS_STATIC_INITIALIZER;
SP_option props[2];

opts.noptions = 0;
opts.options = &props;
props[opts.noptions].type = SP_option_type_system_property;
props[opts.noptions].u.prop.key   = "foo";
props[opts.noptions].u.prop.value = "true";
opts.noptions++;

props[opts.noptions].type = SP_option_type_system_property;
props[opts.noptions].u.prop.key   = "bar";
props[opts.noptions].u.prop.value = "hello";
opts.noptions++;

res = SP_initialize(argv, argc, &opts);
if (res != SP_SUCCESS) {
    ... /* error handling */
}

...
12.3.45 SP_is_atom()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_is_atom(SP_term_ref term);
```

Determines whether the value of `term` is a Prolog atom.

Arguments

- `term` The SP_term_ref to be inspected

Return Value

- 1 if it is a atom and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.46 SP_is_atomic()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_is_atomic(SP_term_ref term);
```

Determines whether the value of `term` is a Prolog atomic term.

Arguments

`term` The SP_term_ref to be inspected

Return Value

1 if it is an atomic term and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.47 SP_is_compound()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_is_compound(SP_term_ref term);
```

Determines whether the value of `term` is a Prolog compound term.

Arguments

term The SP_term_ref to be inspected

Return Value

1 if it is a compound term and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.48 SP_is_float()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_is_float(SP_term_ref term);

Determines whether the value of term is a Prolog float.

Arguments

    term The SP_term_ref to be inspected

Return Value

1 if it is a float and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.49 SP_is_integer()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_is_integer(SP_term_ref term);
```

Determines whether the value of `term` is a Prolog integer.

Arguments

`term` The SP_term_ref to be inspected

Return Value

1 if it is a integer and 0 otherwise.

Please note: `SP_is_integer()` will return true also for integers that are too large to be passed to `SP_get_integer()`. In this case you will need to use `SP_get_integer_bytes()` to obtain the value.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.50 SP_is_list()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_is_list(SP_term_ref term);
```

Determines whether the value of `term` is a Prolog list cell, i.e. a compound term with functor `/2`.

Arguments

term The SP_term_ref to be inspected

Return Value

1 if the argument is a list cell and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.51 SP_is_number()

Synopsis

#include <sicstus/sicstus.h>

int
SP_is_number(SP_term_ref term);

Determines whether the value of term is a Prolog number.

Arguments

term The SP_term_ref to be inspected

Return Value

1 if it is a number and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.52 SP_is_variable()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_is_variable(SP_term_ref term);
```

Determines whether the value of `term` is a Prolog variable.

**Arguments**

`term` The SP_term_ref to be inspected

**Return Value**

1 if it is a variable and 0 otherwise.

**See Also**

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.53 SP_load()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_load(char const *filename);
```

Calls `load_files/1`.

Arguments

`filename` The file name, which is treated as a Prolog atom.

Return Value

See `SP_query_cut_fail()`.

See Also

Section 6.7.4.2 [Loading Prolog Code], page 331.
12.3.54 SP_load_sicstus_run_time()

Synopsis

Since release 4.0.3

```
#include <sicstus/sicstus.h>

int
SP_load_sicstus_run_time(SP_get_dispatch_type **pfuncp,
                           void *reserved);
```

Arguments

- `pfuncp`: On success the address of the `SP_get_dispatch()` function of the newly loaded SICStus runtime is stored at this address.
- `reserved`: Reserved, should be `NULL`.

Return Value

Positive if a new runtime could be loaded, non-positive on error. **Please note:** The return value was incorrectly described prior to SICStus Prolog 4.0.5. Also note that, for historical reasons, the meaning of the return value is different from the convention used by most of the SICStus Prolog C API.

Description

`SP_load_sicstus_run_time()` loads a new SICStus runtime.

As a special case, if `SP_load_sicstus_run_time()` is called from a SICStus runtime that has not been initialized (with `SP_initialize()`) and that has not previously been loaded as the result of calling `SP_load_sicstus_run_time()`, no new runtime is loaded. Instead, the `SP_get_dispatch()` of the runtime itself is returned. In particular, the first time `SP_load_sicstus_run_time()` is called on the initial SICStus runtime, and if this happens before the initial SICStus runtime is initialized, no new runtime is loaded.

Calling `SP_load_sicstus_run_time()` from a particular runtime can be done from any thread.

An application that links statically with the SICStus runtime should not call `SP_load_sicstus_run_time()`.

You should not use pre-linked foreign resources when using multiple SICStus runtimes in the same process.

For an example of loading and using multiple SICStus runtimes, see `library/jasper/spnative.c` that implements this functionality for the Java interface Jasper.

See Also

Chapter 8 [Multiple SICStus Runtimes], page 349.
12.3.55 SP_malloc()

Synopsis

```c
#include <sicstus/sicstus.h>

void *
SP_malloc(size_t size);
```

Allocates a block of at least `size` bytes.

Arguments

- `size` Requested number of bytes.

Return Value

NULL on failure, the pointer otherwise.

See Also

See Section 6.4.7.1 [OS Memory Management], page 305.
12.3.56 SP_mutex_lock()

Synopsis

```c
#include <sicstus/sicstus.h>

static SP_mutex volatile mutex = SP_MUTEX_INITIALIZER;

int
SP_mutex_lock(SP_mutex *pmx);
```

Locks the mutex.

Return Value

Zero on error, non-zero on success.

Examples

```c
static SP_mutex volatile my_mutex = SP_MUTEX_INITIALIZER;
// only access this counter with my_mutex locked
int volatile protected_counter = 0;

// returns the new value of protected_counter
int increment_the_counter(void)
{
    int new_value;

    if(SP_mutex_lock(&my_mutex) == 0) goto error_handling;
    // No other thread can update protected_counter here
    new_value = protected_counter+1;
    protected_counter = new_value;
    if (SP_mutex_unlock(&my_mutex) == 0) goto error_handling;
    return new_value;

    error_handling:
    ...
}
```

See Also

Section 6.4.7.3 [OS Threads], page 306.
12.3.57 SP_mutex_unlock()

Synopsis

```
#include <sicstus/sicstus.h>

static SP_mutex volatile mutex = SP_MUTEX_INITIALIZER;

int
SP_mutex_unlock(SP_mutex *pmx);
```

Unlocks the mutex.

Return Value

Zero on error, non-zero on success.

Description

The number of unlocks must match the number of locks and only the thread that performed the lock can unlock the mutex.

Examples

See the example of SP_mutex_lock().

See Also

Section 6.4.7.3 [OS Threads], page 306.
12.3.58 SP_new_term_ref()

Synopsis

#include <sicstus/sicstus.h>

    SP_term_ref
    SP_new_term_ref(void);

Creates a new SP_term_ref, initialized to the empty list [].

Return Value

The new SP_term_ref.

See Also

Section 6.4.1 [Creating and Manipulating SP_term_refs], page 301.
12.3.59 SP_next_solution()

Synopsis

#include <sicstus/sicstus.h>

int
SP_next_solution(SP_qid query);

Look for the next solution to the given query.

Arguments

query The query, created by SP_open_query().

Return Value

SP_SUCCESS for success, SP_FAILURE for failure, SP_ERROR if an error condition occurred.

Description

This will cause the Prolog engine to backtrack over any current solution of an open query
and look for a new one. The given argument must be the innermost query that is still open,
i.e. it must not have been terminated explicitly by SP_close_query() or SP_cut_query().
Only when the return value is SP_SUCCESS are the values in the query arguments valid, and
will remain so until backtracking into this query or an enclosing one.

See Also

Section 6.5.2 [Finding Multiple Solutions of a Call], page 307.
12.3.60  SP_next_stream()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_next_stream(SP_stream *stream, SP_stream **pnext);
```

Iterate through all Prolog streams.

Arguments

- `stream` If this is `NULL` then `pnext` is set to the first stream in the list of streams. If this is non-`NULL` then the stream following `stream` in the list of streams is returned in `pnext`.

- `pnext` The returned stream is returned in `pnext`.

Return Value

On success, `pnext` is assigned, and SPIO_S_NOERR or some other success code is returned. You should use the SPIO_FAILED() macro to determine if the return value signifies failure or success.

When `stream` is the last stream `pnext` is set to `NULL`.

This function can be used to iterate over all Prolog streams. One way to use this is together with SP_get_stream_user_data to find all currently open user defined streams of a particular type.

See Also

Section 6.6 [SICStus Streams], page 311.
12.3.61  SP_open_query()

Synopsis

    #include <sicstus/sicstus.h>

    SP_qid
    SP_open_query(SP_pred_ref predicate,
                  SP_term_ref arg1,
                  ...);

Sets up a query for use by SP_next_solution(), SP_close_query(), SP_cut_query().

Arguments

predicate  The predicate to call.
arg1...    The arguments to pass.

Return Value

The query identifier if successful, otherwise 0.

Description

Note that if a new query is opened while another is already open, the new query must be
terminated before exploring the solutions of the old one. That is, queries must be strictly
nested.

See Also

Section 6.5.2 [Finding Multiple Solutions of a Call], page 307.
12.3.62 SP_pred()  

Synopsis

```c
#include <sicstus/sicstus.h>

SP_pred_ref
SP_pred(SP_atom name_atom,  
SP_integer arity,  
SP_atom module_atom);
```

Returns a pointer to the predicate definition.

Arguments

- `name_atom`  
  Predicate name.
- `arity`  
  Arity.
- `module_atom`  
  Module name.

Return Value

The reference if the predicate is found, NULL otherwise with error code `PRED_NOT_FOUND`.

Description

Faster than `SP_predicate()`.

See Also

Section 6.5 [Calling Prolog from C], page 306.
12.3.63 SP_predicate()

Synopsis

```c
#include <sicstus/sicstus.h>

SP_pred_ref
SP_predicate(char *name_string,
             SP_integer arity,
             char *module_string);
```

Returns a pointer to the predicate definition.

Arguments

- `name_string`: Predicate name.
- `arity`: Arity.
- `module_string`: Module name, optional. NULL and "" (the empty string) both denote the type-in module (see Section 4.11.8 [ref-mod-tyi], page 166).

Return Value

The reference if the predicate is found, NULL otherwise with error code `PRED_NOT_FOUND` or, if one of the string arguments are malformed, `INV_STRING`.

Description

Slower than `SP_pred()`.

See Also

Section 6.5 [Calling Prolog from C], page 306.
12.3.64 SP_printf()

Synopsis

```
#include <sicstus/sicstus.h>

spio_t_error_code
SP_printf(
    char const *fmt,
    ...);
```

Same as SP_fprintf(SP_stdout, fmt, ...).

See Also

Section 12.3.22 [cpg-ref-SP_fprintf], page 1291.
12.3.65  SP_put_address()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_put_address(SP_term_ref term,
                  void *pointer);
```

Assigns to `term` a reference to a Prolog integer representing a pointer.

Arguments

- `term`  The SP_term_ref to be assigned
- `pointer`  The pointer

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.66 SP_put_atom()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_put_atom(SP_term_ref term,
            SP_atom atom);
```

Assigns to `term` a reference to a Prolog atom.

Arguments

term The SP_term_ref to be assigned
atom The atom

Return Value
Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also
Section 6.4.3 [Creating Prolog Terms], page 303.
**12.3.67 SP_put_byte()**

**Synopsis**

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_put_byte(
    SP_stream *stream,
    int item);
```

Write a byte to a binary stream.

**Arguments**

- `stream`: The stream. Must be a binary stream open for output.

**Return Value**

On success, the written byte will be returned, cast to a `spio_t_error_code`. The value returned on successful return will never be negative.

On failure, returns an error code, recognizable with `SPIO_FAILED()`. Error codes are always negative.

**Description**

Note that `SP_put_byte()` is implemented as a macro and may evaluate the arguments more than once. For the same reason, no error checking is performed on the arguments.

**See Also**

Section 12.3.69 [cpg-ref-SP_put_code], page 1342. Section 6.6.1 [Prolog Streams], page 311.
12.3.68  SP_put_bytes()

Synopsis

```
#include <sicstus/sicstus.h>

spio_t_error_code
SP_put_bytes(
    spio_t_uint8 const *bytes,
    size_t byte_count,
    spio_t_bits options);
```

Write several bytes to a binary stream.

Arguments

- `stream`  The stream. Must be a binary stream open for output.
- `bytes`  A pointer to the data to write.
- `byte_count`  The number of bytes to write.
- `options`  The following bits can be set:

  - `SP_PUT_BYTES_OPTION_NONBLOCKING`
    Write the bytes without blocking.

Return Value

On success, all data has been written and `SPIO_S_NOERR` or some other success code returned.

On failure, returns an error code without transferring any data. Error codes with special meaning for `SP_put_bytes()`:

- `SPIO_E_WOULD_BLOCK`
  `SP_PUT_BYTES_OPTION_NONBLOCKING` was set but the operation would block.

Other error codes may also be returned.

See Also

Section 12.3.67 [cpg-ref-SP_put_byte], page 1340. Section 6.6.1 [Prolog Streams], page 311.
12.3.69 SP_put_code()

Synopsis

```
#include <sicstus/sicstus.h>

spio_t_error_code
SP_put_code(
    SP_stream *stream,
    int item);
```

Write a character code to a text stream.

Arguments

`stream` The stream. Must be a text stream open for output.

Return Value

On success, the written character will be returned, cast to a `spio_t_error_code`. The value returned on successful return will never be negative.

On failure, returns an error code, recognizable with `SPIO_FAILED()`. Error codes are always negative.

Description

Note that `SP_put_code()` is implemented as a macro and may evaluate the arguments more than once. For the same reason, no error checking is performed on the arguments.

See Also

Section 12.3.67 [cpg-ref-SP_put_byte], page 1340. Section 6.6.1 [Prolog Streams], page 311.
12.3.70 SP_put_codes()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_put_codes(
       SP_stream *stream,
       spio_t_wchar const *codes,
       size_t code_count,
       spio_t_bits options);
```

Write several codes to a text stream.

Arguments

- `stream`  The stream. Must be a text stream open for output.
- `codes`   A pointer to the data to write.
- `code_count`  The number of character codes to write. Note that this is the number of character codes, not the number of bytes.
- `options`   The following bits can be set:

  - `SP_PUT_CODES_OPTION_NONBLOCKING`  Write the codes without blocking.

Return Value

On success, all data has been written and `SPIO_S_NOERR` or some other success code returned.

On failure, returns an error code without transferring any data. Error codes with special meaning for `SP_put_codes()`:

- `SPIO_E_WOULD_BLOCK`  `SP_PUT_CODES_OPTION_NONBLOCKING` was set but the operation would block.

Other error codes may also be returned.

See Also

Section 12.3.69 [cpg-ref-SP_put_code], page 1342. Section 6.6.1 [Prolog Streams], page 311.
12.3.71 SP_put_encoded_string()

Synopsis
#include <sicstus/sicstus.h>

spio_t_error_code
SP_put_encoded_string(
    SP_stream *stream,
    spio_t_wchar const *encoded_string,
    spio_t_bits options);

Write an encoded string to a text stream.

Arguments
stream The stream. Must be a text stream open for output.
encoded_string An encoded string to write.
options The following bits can be set:
    SP_PUT_ENCODED_STRING_OPTION_NONBLOCKING
        Write the string without blocking.

Return Value
On success, all data has been written and SPIO_S_NOERR or some other success code returned.

On failure, returns an error code without transferring any data. Error codes with special
meaning for SP_put_encoded_string():

SPIO_E_WOULD_BLOCK
    SP_PUT_ENCODED_STRING_OPTION_NONBLOCKING was set but the operation
    would block.

Other error codes may also be returned.

See Also
Section 12.3.70 [cpg-ref-SP_put_codes], page 1343. Section 6.6.1 [Prolog Streams], page 311.
12.3.72 SP_put_float()

Synopsis

#include <sicstus/sicstus.h>

int
SP_put_float(SP_term_ref term,
    double f);

Assigns to term a reference to a float.

Arguments

term The SP_term_ref to be assigned
f The float (must be finite)

Return Value
Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also
Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.73 SP_put_functor()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_put_functor(SP_term_ref term,
               SP_atom name,
               int arity);
```

Assigns to `term` a reference to a compound term with all the arguments unbound variables. If `arity` is 0, assigns the Prolog atom whose canonical representation is `name`. This is similar to calling `functor/3` with the first argument unbound and the second and third arguments bound to an atom and an integer, respectively.

Arguments

- `term` The SP_term_ref to be assigned
- `name` The name of the functor
- `arity` The arity of the functor

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.74 SP_put_integer()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_put_integer(SP_term_ref term,
                SP_integer i);
```

Assigns to `term` a reference to an integer.

Arguments

- `term` The SP_term_ref to be assigned
- `i` The integer

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.75 SP_put_integer_bytes()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_put_integer_bytes(SP_term_ref term,
                    void *buf,
                    size_t buf_size,
                    int native);
```

Assigns to `term` a reference to an arbitrarily sized integer.

Arguments

term The SP_term_ref to be assigned

buf

- If `native` is zero, `buf` consists of the `buf_size` bytes of the two's complement representation of the integer. Less significant bytes are at lower indices.
- If `native` is nonzero, `buf` is a pointer to the native `buf_size` bytes integer type.

buf_size The size of `buf`

native See above. Supported native sizes typically include two, four and eight (64bit) bytes.

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.76 SP_put_list()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_put_list(SP_term_ref term);

Assigns to term a reference to a Prolog list whose head and tail are both unbound variables.

Arguments

term       The SP_term_ref to be assigned

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.77 SP_put_list_codes()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_put_list_codes(SP_term_ref term,
                      SP_term_ref tail,
                      char const *s);

Assigns to term a Prolog code list represented by the encoded string s, prepended to the value of tail.

Arguments

term       The SP_term_ref to be assigned
tail      The tail of the code list
s          The string to convert

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.78  SP_put_list_n_bytes()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_put_list_n_bytes(SP_term_ref term,
                        SP_term_ref tail,
                        size_t n,
                        unsigned char const *s);

Assigns to \texttt{term} a list of integers represented by the first \texttt{n} elements of the byte array \texttt{s}, prepended to the value of \texttt{tail}.

Arguments

term  The SP\_term\_ref to be assigned
tail  The tail of the list
n     The number of bytes of \texttt{s} to convert
s     The byte array to convert

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.79 SP_put_list_n_codes()

Synopsis

#include <sicstus/sicstus.h>

int
SP_put_list_n_codes(SP_term_ref term,
                     SP_term_ref tail,
                     size_t n,
                     char const *s);

Assigns to term a Prolog code list represented by the first n bytes of the encoded string s, prepended to the value of tail.

Please note: Some characters may be encoded using more than one byte so the number of characters may be less than n.

Arguments

    term        The SP_term_ref to be assigned
    tail        The tail of the code list
    n           The number of character codes of s to convert
    s           The string to convert

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.80  SP_put_number_codes()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_put_number_codes(SP_term_ref term,
                        char const *s);

Assigns to term a reference to a Prolog number obtained by parsing s as if by number_codes/2.

Arguments

    term  The SP_term_ref to be assigned
    s     The string to parse

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.81 SP_put_string()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_put_string(SP_term_ref term,
              char const *string);
```

Assigns to `term` a reference to a Prolog atom.

Arguments

- **term**: The SP_term_ref to be assigned
- **string**: The string corresponding to the atom

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.82 SP_put_term()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_put_term(SP_term_ref to,
            SP_term_ref from);
```

Assigns to to the value of from.

Arguments

to The SP_term_ref to be assigned
from The SP_term_ref whose value is accessed

Return Value

Zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.83 SP\_put\_variable()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_put_variable(SP_term_ref term);
```

Assigns to `term` a reference to a new unbound Prolog variable.

**Arguments**

`term` The SP\_term\_ref to be assigned

**Return Value**

Zero if the call fails (as far as failure can be detected), and a nonzero value otherwise.

**See Also**

Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.84 SP_query()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_query(SP_pred_ref predicate,
         SP_term_ref arg1,
         ...);
```

Calls a predicate, committing to its first solution.

Arguments

- **predicate**  The predicate to call.
- **arg1...**  The arguments to pass.

Return Value

- **SP_SUCCESS** if the goal succeeded,
- **SP_FAILURE** if it failed,
- **SP_ERROR** if an error condition occurred.

Description

Use this if you are only interested in the first solution. It will create a goal from the
predicate definition and the arguments, call it, and commit to the first solution found, if
any. If it returns **SP_SUCCESS**, values in the query arguments valid, and will remain so until
backtracking into any enclosing query.

See Also

Section 6.5.1 [Finding One Solution of a Call], page 307.
12.3.85 SP_query_cut_fail()

Synopsis

#include <sicstus/sicstus.h>

int
SP_query_cut_fail(SP_pred_ref predicate,
                 SP_term_ref arg1,
                 ...);

Calls a predicate for side effects, reclaiming any storage used.

Arguments

predicate  The predicate to call.
arg1...    The arguments to pass.

Return Value

SP_SUCCESS if the goal succeeded, SP_FAILURE if it failed, and SP_ERROR if an error condition occurred.

Description

Call this if you are only interested in the side effects of a predicate. It will try to prove the predicate, cut away the rest of the solutions, and finally fail. This will reclaim the storage used after the call, and throw away any solution found.

See Also

Section 6.5.1 [Finding One Solution of a Call], page 307.
12.3.86 SP_raise_exception()

Synopsis

#include <sicstus/sicstus.h>

void
SP_raise_exception(SP_term_ref term);

Raises an exception in the scope of Prolog calling C.

Arguments

term The SP_term_ref whose value will be the exception term.

Description

The exception will be stored as pending. This function is normally used in the context of
a call from Prolog to C, and will cause the exception to be propagated to Prolog on return
from the call. The effect is as if raise_exception/1 was called with the term as argument.

Please note: this should only be called right before returning to Prolog.

See Also

Section 6.5.5 [Exception Handling in C], page 310.
12.3.87 SP_read_from_string()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_read_from_string(SP_term_ref t,
                      const char *string,
                      SP_term_ref values[])
```

Assigns to `t` the result of reading a term from the its textual representation `string`. Variables that occur in the term are bound to the corresponding term in `val`.

Arguments

- `term` The SP_term_ref to assign.
- `string` The string to read from.
- `values` The SP_term_refs to bind variables to. The vector is terminated by 0 (zero). `values` may be NULL, which is treated as an empty vector.

Return Value
Nonzero on success, and 0 otherwise.

Description
The variables in the term are ordered according to their first occurrence during a depth first traversal in increasing argument order. That is, the same order as used by terms:term_variables_bag/2 (see Section 10.46 [lib-terms], page 862). Variables that do not have a corresponding entry in `vals` are ignored. Entries in `vals` that do not correspond to a variable in the term are ignored.

The string should be encoded using the SICStus Prolog internal encoding.

Examples
This example creates the term `foo(X,42,42,X)` (without error checking):
SP_term_ref x = SP_new_term_ref();
SP_term_ref y = SP_new_term_ref();
SP_term_ref term = SP_new_term_ref();
SP_term_ref vals[] = {x,y,x,0}; // zero-terminated

SP_put_variable(x);
SP_put_integer(y,42);

SP_read_from_string(term, "foo(A,B,B,C).", vals);
#if 0
A corresponds to vals[0] (x),
B to vals[1] (y),
C to vals[2] (x).
A and C therefore both are bound to
the variable referred to by x.
B is bound to the term referred to by y (42).
So term refers to a term foo(X,42,42,X).
#endif

See Section 6.5.6 [Reading a goal from a string], page 311, for an example of using SP_read_from_string() to call an arbitrary goal.

See Also
Section 6.4.3 [Creating Prolog Terms], page 303.
12.3.88 SP_realloc()

Synopsis

```
#include <sicstus/sicstus.h>

void *
SP_realloc(void *ptr,
            size_t size);
```

Changes the size of the block referenced by `ptr` to `size` bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. The block referenced by `ptr` must have been obtained by a call to `SP_malloc()` or `SP_realloc()`, and must not have been released by a call to `SP_free()` or `SP_realloc()`.

Arguments

- `ptr` The current block.
- `size` Requested number of bytes of the new block.

Return Value

`NULL` on failure, the pointer otherwise.

See Also

See Section 6.4.7.1 [OS Memory Management], page 305.
12.3.89  SP_register_atom()

Synopsis

    #include <sicstus/sicstus.h>

    int
    SP_register_atom(SP_atom atom);

Registers the atom atom with the Prolog memory manager by incrementing its reference counter.

Arguments

atom The atom to register

Return Value

1 if atom is valid, and 0 otherwise.

See Also

Section 6.4.1 [Creating and Manipulating SP_term_refs], page 301.
12.3.90 SP_restore()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_restore(char const *filename);
```

Calls `restore/1`.

Arguments

`filename` The file name, which is treated as a Prolog atom.

Return Value

See `SP_query_cut_fail()`.

See Also

Section 6.7.4.2 [Loading Prolog Code], page 331.
12.3.91 SP_set_argv()

Synopsis

#include <sicstus/sicstus.h>

int
SP_set_argv(int argc,
    char **argv,
    spio_t_bits options);

Initializes the argv prolog flag.

Arguments

argc  The number of elements of the argv vector.

argv  A vector of strings that can be accessed by prolog_flag(argv,X). This argument
    is copied by SP_set_argv() so it can be discarded by the caller. May be
    passed as NULL if argc is zero.

The encoding used when converting the entries to atoms is determined by the
options argument.

options  If options is zero then each entry in argv is assumed to be an encoded string,
    i.e. encoded using ‘UTF-8’. This is the same as passing argv and argc to SP_
    initialize() except that SP_initialize() will not report failure even if the
    argv entries are not suitable as atom names. Note that UTF-8 may not be the
    encoding used by the operating system when invoking main().

If options is SP_SET_ARGV_OPTION_SYSTEM_ENCODING then each entry in argv
    is assumed to be encoded using some system encoding. This is only useful on
    Unix-like systems since it is preferable, and trivial, to obtain and pass a UTF-8
    argv vector on Windows.

The system encoding used by SP_set_argv() will be the first character encoding specified by the following environment variables:

    SP_CTYPE
    LC_ALL (only on Unix)
    LC_CTYPE (only on Unix)
    LANG (only on Unix)

if none of these are set then the operating system will be queried in some
platform specific way.

Please note: For best results on Unix-like platforms you should use a locale
based on UTF-8.

Return Value

Zero if the argument entries cannot be converted to atoms, and a nonzero value otherwise.
Description

See Also

Section 6.7.4.1 [Initializing the Prolog Engine], page 330.
12.3.92 SP_set_current_dir()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_set_current_dir(char const *dir);
```

Makes a directory pointed to by `dir` to become the current working directory. `path` should be an encoded string.

Arguments

`dir` Name of the directory to become current.

Return Value

On success, `SPIO_S_NOERR` or some other success code is returned.

On failure, an error code is returned and the working directory is not changed.

See Also

Section 12.3.29 [cpg-ref-SP_get_current_dir], page 1298. Section 6.4.7.2 [OS File System], page 306.
12.3.93 SP_set_user_stream_hook()  

Synopsis

```c
#include <sicstus/sicstus.h>

typedef SP_stream *
SP_UserStreamHook(void *user_data, int which);

SP_UserStreamHook *
SP_set_user_stream_hook(SP_UserStreamHook *hook, void *user_data);
```

Sets the user-stream hook to `hook`. Must be called before `SP_initialize()`.

Arguments

- **hook**  
  It is called three times, one for each stream. The `which` argument indicates which stream it is called for. The value of `which` is one of:

  - `SP_STREAMHOOK_STDIN`: Create stream for standard input.
  - `SP_STREAMHOOK_STDOUT`: Create stream for standard output.
  - `SP_STREAMHOOK_STDERR`: Create stream for standard error.

  The hook should return a standard SICStus text I/O stream, as described in Section 6.6.2 [Defining a New Stream], page 313.

- **user_data**  
  An arbitrary pointer that will be passed to the hook.

See Also

Section 6.6.3 [Hookable Standard Streams], page 317.
12.3.94 SP_set_user_stream_post_hook()  

Synopsis

```
#include <sicstus/sicstus.h>

typedef SP_stream *
SP_UserStreamPostHook(void *user_data, int which, SP_stream *str);

SP_UserStreamPostHook *
SP_set_user_stream_post_hook(SP_UserStreamPostHook *hook, void *user_data);
```

Sets the user-stream post-hook to `hook`. Must be called before `SP_initialize()`.

Arguments

`hook`  The user-stream post-hook is, if defined, called after all the streams have been defined, once for each of the three standard streams. It has a slightly different prototype:

```
void user_stream_post_hook(void *user_data, int which, SP_stream *str)
```

where `user_data` is the value passed to `SP_set_user_stream_post_hook` and where `str` is a pointer to the corresponding `SP_stream` structure. There are no requirements as to what this hook must do; the default behavior is to do nothing at all.

The post-hook is intended to be used to do things that may require that all streams have been created.

`user_data`  An arbitrary pointer that will be passed to the hook.

See Also

Section 6.6.3 [Hookable Standard Streams], page 317.
12.3.95 SP_signal()

Synopsis

```c
#include <sicstus/sicstus.h>

typedef void
SP_SigFun (int sig, void *user_data);

SP_SigFun
SP_signal(int sig,
    SP_SigFun fun, void *user_data);
```

Installs a function `fun` as a handler for the signal `sig`. It will be called with `sig` and `user_data` as arguments.

Arguments

- `sig` The signal
- `fun` The function
- `user_data` An extra, user defined value passed to the function.

Return Value

`SP_SIG_ERR` if an error occurs error. On success, some value different from `SP_SIG_ERR`.

Description

When the OS delivers a signal `sig` for which `SP_signal(sig,func,...)` has been called, SICStus will *not* call `fun` immediately. Instead the call to `fun` will be delayed until it is safe for Prolog to do so, in much the same way that functions installed by `SP_event()` are handled.

Since the signal handling function `fun` will not be called immediately upon delivery of the signal to the process it only makes sense to use `SP_signal()` to handle certain asynchronous signals such as `SIGINT`, `SIGUSR1`, `SIGUSR2`. Other asynchronous signals handled specially by the OS, such as `SIGCHLD` are not suitable for handling via `SP_signal()`. Note that the development system installs a handler for ‘SIGINT’, and, under Windows, ‘SIGBREAK’, to catch keyboard interrupts. As of release 4.4, `library(timeout)` no longer uses any signals.

When `fun` is called it may only call other (non SICStus) C code and `SP_event()`. Note that `fun` will be called in the main thread.

If `fun` is one of the special constants `SP_SIG_IGN` or `SP_SIG_DFL`, then one of two things happens:

1. If a signal handler for `sig` has already been installed with `SP_signal()`, then the SICStus OS-level signal handler is removed and replaced with, respectively, `SIG_IGN` or `SIG_DFL`.
2. If a signal handler has not been installed with `SP_signal()`, then `SP_signal()` does nothing and returns `SP_SIG_ERR`. 
A signal handler installed by a foreign resource should be uninstalled in the deinit function for the foreign resource. This is to prevent the handler in the foreign resource from being called after the code of the foreign resource has been unloaded (e.g. by `unload_foreign_resource/1`).

Note that `SP_signal()` is not suitable for installing signal handlers for synchronous signals like `SIGSEGV`.

**See Also**

`SP_event()`, Section 6.5.4.1 [Signal Handling], page 310.
12.3.96 SP_strdup()

Synopsis

#include <sicstus/sicstus.h>

void *
SP_strdup(const char *str);

Allocates a string, which is a duplicates of the given string. The memory for the new string is managed by Prolog.

Arguments

str The given string.

Return Value

The pointer, if allocation was successful, otherwise NULL.

See Also

Section 6.4.7.1 [OS Memory Management], page 305.
12.3.97  SP_string_from_atom()

Synopsis

    #include <sicstus/sicstus.h>

    char const *
    SP_string_from_atom(SP_atom atom);

Obtains the encoded string holding the characters of a Prolog atom. This string must not be modified by the calling function.

Arguments

atom    The atom to inspect.

Return Value

The encoded string if atom is valid, and 0 otherwise.

See Also

Section 6.4.1 [Creating and Manipulating SP_term_refs], page 301.
12.3.98 SP_term_type

Synopsis

#include <sicstus/sicstus.h>

int
SP_term_type(SP_term_ref term);

Determines the type of the value of term.

Arguments

term The SP_term_ref to be inspected

Return Value

One of:

SP_TYPE_VARIABLE
    a variable

SP_TYPE_INTEGER
    an integer

SP_TYPE_FLOAT
    a float

SP_TYPE_ATOM
    an atom

SP_TYPE_COMPOUND
    a compound term

SP_TYPE_ERROR
    an error occurred.

See Also

Section 6.4.5 [Testing Prolog Terms], page 304.
12.3.99 SP_unget_byte()

Synopsis

#include <sicstus/sicstus.h>

spio_t_error_code
SP_unget_byte(
    SP_stream *stream,
    int item);

Push back a byte so it can be read again by subsequent read operations.

Arguments

stream The stream. Must be a binary stream open for input.

item The byte to push back. This must be the byte that was most recently read from stream, e.g. with SP_get_byte(). As a special case, -1 can be put back if the last read operation returned end of file, i.e., SPIO_E_END_OF_FILE.

Return Value

On success, the byte has been pushed back and will be read by the next read operation. SPIO_S_NOERR or some other success code is returned.

On failure, an error code is returned.

See Also

Section 12.3.27 [cpg-ref-SP_get_byte], page 1296. Section 6.6.1 [Prolog Streams], page 311.
12.3.100  SP_unget_code()

Synopsis

```c
#include <sicstus/sicstus.h>

spio_t_error_code
SP_unget_code(
    SP_stream *stream,
    int item);
```

Push back a character so it can be read again by subsequent read operations.

Arguments

- `stream` The stream. Must be a text stream open for input.
- `item` The character to push back. This must be the same character that was most recently read from `stream`, e.g. with `SP_get_code()`. As a special case, -1 can be put back if the last read operation returned end of file, i.e., `SPIO_E_END_OF_FILE`.

Return Value

On success, the character has been pushed back and will be read by the next read operation. `SPIO_S_NOERR` or some other success code is returned.

On failure, returns an error code.

See Also

Section 12.3.28 [cpg-ref-SP_get_code], page 1297. Section 6.6.1 [Prolog Streams], page 311.
12.3.101 SP_unify()

Synopsis

   #include <sicstus/sicstus.h>

   int
       SP_unify(SP_term_ref x,
                 SP_term_ref y)

Unifies two terms.

Arguments

x      The one term to unify
y      The other term to unify

Return Value

1 if they unify, and 0 otherwise.

Description

Bear in mind that the unification may unblock some goals. such goals are not run in the scope of SP_unify(); they remain pending until the next Prolog goal is run.

See Also

Section 6.4.6 [Unifying and Comparing Terms], page 305.
12.3.102 SP_unregister_atom()

Synopsis

```c
#include <sicstus/sicstus.h>

int
SP_unregister_atom(SP_atom atom);
```

Unregisters the atom `atom` with the Prolog memory manager by incrementing its reference counter.

Arguments

- `atom` The atom to unregister

Return Value

1 if `atom` is valid, and 0 otherwise.

See Also

Section 6.4.1 [Creating and Manipulating SP_termRefs], page 301.
12.3.103 SU_initialize()

Synopsis

```c
int SU_initialize(int argc, char *argv[])
```

In applications built with `--userhook`, `SU_initialize()` is called by the main program before `SP_initialize()`. Its purpose is to call interface functions, which must be called before `SP_initialize()`. It is not meaningful to specify this option if `--main=user` or `--main=none` is given.

Arguments

- `argc` Number of command-line arguments.
- `argv` The command-line arguments, should not be modified.

Return Value

Zero on success, and nonzero otherwise. If a non-zero value is returned, the application system exits with the return value as error code.

See Also

Section 6.7.3 [The Application Builder], page 323.
12.3.104 user_close()

Synopsis

```c
spio_t_error_code
user_close(
    void **puser_data,
    spio_t_bits close_options
);
```

This is the prototype for one of the methods of user defined streams. It is used when SICStus wants to close one or both directions of a user defined stream.

Arguments

*puser_data

A pointer to the same value as was passed to `SP_create_stream()`. On successful return, if the stream has been closed and any resources freed, then *puser_data should be set to NULL.

If user_close fails, it can still set *puser_data to NULL to signify that the stream is no longer usable.

close_options

The following bits can be set:

SPIO_DEVICE_CLOSE_OPTION_READ

The read direction should be closed. Only set if the device was created as an input or bidirectional device.

SPIO_DEVICE_CLOSE_OPTION_WRITE

The write direction should be closed. Only set if the device was created as an output or bidirectional device.

SPIO_DEVICE_CLOSE_OPTION_FORCE

The specified directions should be closed without attempting to flush any data. Among other things this option may be passed if a previous call to user_close returned an error.

Note that a bidirectional stream should only close the directions specified by the close_options. Also note that user_close for a bidirectional stream may be called several times and that the same direction flag, e.g. SPIO_DEVICE_CLOSE_OPTION_READ may be specified more than once, even if that direction has already been closed successfully.

Once a call to user_close has set *puser_data to NULL, none of the device methods will be called again. Note that a *puser_data may be set to NULL even when a failure code is returned. This is useful if the failure is unrecoverable.

There is no option to specify non-blocking close, it is expected that user_close will finish quickly. To make this more likely, user_flush_output is called before non-forcibly closing an output stream.
Return Value

On success, return SPIO_S_NOERR or some other success code and set *puser_data if and only if the user data and any other resources have been freed.

On failure, return a SPIO error code. Error codes with special meaning for user_close:

SPIO_E_END_OF_FILE
   Returned if there were buffered data and it is not possible to write more data onto the stream, e.g. some underlying device has been closed.

Other error codes may also be returned.

Description

Should close one or all directions depending on the close_options. If all directions have been closed, the user data should be deallocated and *puser_data set to NULL.

See Also

Section 12.3.9 [cpg-ref-SP_create_stream], page 1270. Section 6.6.2 [Defining a New Stream], page 313.
12.3.105 user_flush_output()

Synopsis

```c
spio_t_error_code
user_flush_output(
    void *user_data,
    spio_t_bits flush_options
);
```

This is the prototype for one of the methods of user defined streams. It is used when SICStus wants to write data to the user defined stream.

Arguments

- `user_data` The same value as was passed to `SP_create_stream()`.

- `flush_options` The following bits can be set:
  
<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIO_DEVICE_FLUSH_OPTION_NONBLOCKING</td>
<td>If this is set, the function should return quickly or with a SPIO_E_WOULD_BLOCK code.</td>
</tr>
<tr>
<td></td>
<td>If your <code>user_flush_output</code> will never block, you can ignore this value.</td>
</tr>
<tr>
<td></td>
<td>You should return SPIO_E_NOT_SUPPORTED if <code>user_flush_output</code> cannot support non-blocking flush.</td>
</tr>
</tbody>
</table>

Return Value

On success, all buffered data should have been written and SPIO_S_NOERR or some other success code returned.

On failure, return a SPIO error code. Error codes with special meaning for `user_flush_output`:

- SPIO_E_END_OF_FILE
  Returned if it is not possible to write more data onto the stream, e.g. some underlying device has been closed.

- SPIO_E_WOULD_BLOCK
  SPIO_DEVICE_FLUSH_OPTION_NONBLOCKING was set but the operation would block.

- SPIO_E_NOT_SUPPORTED
  Some unsupported option, e.g. SPIO_DEVICE_FLUSH_OPTION_NONBLOCKING, was passed.

Other error codes may also be returned.

Description

Should ensure that any buffered data is transmitted to its destination. Can be passed as NULL.
See Also

Section 12.3.9 [cpg-ref-SP_create_stream], page 1270. Section 6.6.2 [Defining a New Stream], page 313.
12.3.106 user_read()

Synopsis

```c
spio_t_error_code
user_read(
    void *user_data,
    void *buf,
    size_t *pbuf_size,
    spio_t_bits read_options
);
```

This is the prototype for one of the methods of user defined streams. It is used when SICStus need to obtain more data from the user defined stream.

Arguments

**user_data** The same value as was passed to `SP_create_stream()`.

**buf** Points to a buffer allocated by the caller.

**pbuf_size** Points to the size of the buffer. The buffer is always large enough to hold at least one byte (for binary streams) or one character (for text streams). When this function returns successfully, `*pbuf_size` should be set to the number of bytes stored in the buffer, which should always be positive for successful return. Note that buffer size is measured in bytes also for text streams.

**read_options** The following bits can be set:

- **SPIO_DEVICE_READ_OPTION_BINARY**
  This is always specified if the device was created as a binary device.
  The buffer should be filled with up to `*pbuf_size` bytes.

- **SPIO_DEVICE_READ_OPTION_TEXT**
  This is always specified if the device was created as a text device.
  The buffer should be filled with wide characters, i.e. `spio_t_wchar`. Note that `*buf_size` is size in bytes, not in characters.

- **SPIO_DEVICE_READ_OPTION_NONBLOCKING**
  If this is set then the function should return quickly, either with some data read or with a `SPIO_E_WOULD_BLOCK` code.
  If your `user_read` will never block, you can ignore this value.
  You should return `SPIO_E_NOT_SUPPORTED` if `user_read` cannot support non-blocking read.

Return Value

On success, `*pbuf_size` should be assigned and `SPIO_S_NOERR` or some other success code returned.

On failure, return a SPIO error code. Error codes with special meaning for `user_read`:
SPIO_E_END_OF_FILE
Return this when there are no more data to read.

SPIO_E_WOULD_BLOCK
SPIODEVICE_READ_OPTION_NONBLOCKING was set but the operation would block.

SPIO_E_NOT_SUPPORTED
Some unsupported option, e.g. SPIODEVICE_READ_OPTION_NONBLOCKING, was passed.

Other error codes may also be returned.

Description
Should fill buf with up to buf_size bytes of data. Data should be either bytes, for a binary device, or spio_t wchar (32 bit) wide characters, for a text device.

See Also
Section 12.3.9 [cpg-ref-SP_create_stream], page 1270. Section 6.6.2 [Defining a New Stream], page 313.
12.3.107 user_write()

Synopsis

```c
spio_t_error_code
user_write(
    void *user_data,
    void const *buf,
    size_t *pbuf_size,
    spio_t_bits write_options
);
```

This is the prototype for one of the methods of user defined streams. It is used when SICStus wants to write data to the user defined stream.

Arguments

- **user_data** The same value as was passed to `SP_create_stream()`.
- **buf** Points to a buffer allocated by the caller containing the data to be written.
- **pbuf_size** Points to the size of the buffer, always positive. When this function returns successfully, `*pbuf_size` should be set to the number of bytes actually written, which should always be positive for successful return.
  
  Note that buffer size is measured in bytes also for text streams.
- **write_options**
  
  The following bits can be set:

  - `SPIODEVICE_WRITE_OPTION_BINARY`
    
    This is always specified if the device was created as a binary device. The buffer contains `*pbuf_size` bytes.
  - `SPIODEVICE_WRITE_OPTION_TEXT`
    
    This is always specified if the device was created as a text device. The buffer contains wide characters, i.e. `spio_t_wchar`. Note that `*buf_size` is size in bytes, not in characters.
  - `SPIODEVICE_WRITE_OPTION_NONBLOCKING`
    
    If this is set, the function should return quickly, either with some data written or with a `SPIO_E_WOULD_BLOCK` code.
    
    If your `user_write` will never block, you can ignore this value.
    
    You should return `SPIO_E_NOT_SUPPORTED` if `user_write` cannot support non-blocking write.

Return Value

On success, `*pbuf_size` should be assigned to with the number of bytes written and `SPIOS_NOERR` or some other success code returned. On success, something must have been written, e.g. `*pbuf_size` must be set to a positive value.

On failure, return a SPIO error code. Error codes with special meaning for `user_write`: 
SPIO_E_END_OF_FILE
   Returned if it is not possible to write more data onto the stream, e.g. some underlying device has been closed.

SPIO_E_WOULD_BLOCK
   SPIO_DEVICE_WRITE_OPTION_NONBLOCKING was set but the operation would block.

SPIO_E_NOT_SUPPORTED
   Some unsupported option, e.g. SPIO_DEVICE_WRITE_OPTION_NONBLOCKING, was passed.

Other error codes may also be returned.

**Description**

Should write up to *buf_size bytes of data from buf. Data could be either bytes, for a binary device, or wide characters, for a text device.

**See Also**

Section 12.3.9 [cpg-ref-SP_create_stream], page 1270. Section 6.6.2 [Defining a New Stream], page 313.
13 Command Reference Pages

The reference pages for the SICStus Prolog command line tools follow.

sicstus(1)  
SICStus Prolog Development System

mzn-sicstus(1)  
Shortcut for MiniZinc with SICStus back-end

spfz(1)  
FlatZinc Interpreter

spdet(1)  
Determinacy Checker

spld(1)  
SICStus Prolog Application Builder

splfr(1)  
SICStus Prolog Foreign Resource Linker

splm(1)  
SICStus Prolog License Manager

spxref(1)  
Cross Referencer
13.1 sicstus — SICStus Prolog Development System

Synopsis

    % sicstus [options] [-- argument...]  

Description

The prompt ‘| ?- ’ indicates that the execution is in top-level mode. In this mode, Prolog
queries may be issued and executed interactively. To exit from the top level and return to
the shell, either type ^D at the top level, or call the built-in predicate halt/0, or use the e
(exit) command following a ^C interruption.

Under Windows, sicstus.exe is a console-based program that can run in a command
prompt window, whereas spwin.exe runs in its own window and directs the Prolog standard
streams to that window. spwin.exe is a “windowed” executable.

Options

- -f  
    Fast start. Do Not read any initialization file on startup. If the option is
    omitted and the initialization file exists, then SICStus Prolog will consult it on
    startup after running any initializations and printing the version banners. The
    initialization file is .sicstusrc or sicstus.ini in the users home directory,
    i.e. “/.sicstusrc or “/sicstus.ini. See Section 4.5.2 [ref-fdi-syn], page 99,
    for an explanation of how a file specification starting with ‘~/’ is interpreted.

- -i  
    Forced interactive. Prompt for user input, even if the standard input stream
does not appear to be a terminal.

- -m  
    Use malloc() et al. for memory allocations.

--noinfo  
    Start with the informational Prolog flag set to off initially, suppressing in-
    formational messages. The flag is set before any prolog-file or initialization file
    is loaded or any saved state is restored.

--nologo  
    Start without the initial version message.

- -l prolog-file  
    Ensure that the file prolog-file is loaded on startup. This is done before any
    initialization file is loaded. The -l option can be specified more than once, and
    all files will be loaded in the order specified.

- -r saved state  
    Restore the saved state saved state on startup. This is done before any prolog-
    file or initialization file is loaded. Only one -r option is allowed.

--goal Goal  
    Read a term from the text Goal and pass the resulting term to call/1 after all
    files have been loaded. As usual Goal should be terminated by a full stop (‘.’).
    Only one --goal option is allowed.

-Dvar=value  
    Sets the system property var to value value. Most system properties take their
default value from the environment but often it is convenient to pass a system
property directly instead of setting the corresponding environment variable. See Section 4.17.1 [System Properties and Environment Variables], page 224, for details.

--locale name
Sets the process locale to the given locale name. The process locale primarily affects the character encoding used for the standard streams.
The default, also available by specifying ‘default’ as the name, is to inherit the locale from the environment.
This option is not supported on Windows.

--no-locale
Do not inherit the process locale from the environment.
This is, in effect, the default on Windows.

-Xrs
Reduce use of OS-signals.
On UNIX-like platforms, several OS signals are handled specially in a development system. The option -Xrs, prevents this and keeps the OS default behavior.
On both UNIX-like platforms and Windows, the development system will install handlers for the signal SIGINT (corresponding to a C-c keyboard interrupt). On Windows, a signal handler will also be added for SIGBREAK (signalled when the console window is closed). The handling of SIGINT and SIGBREAK is not affected by -Xrs.

--help
Display a help message and exit.

-- argument... since release 4.0.3
-a argument...
where the arguments can be retrieved from Prolog by prolog_flag(argv, Args), which will unify Args with argument... represented as a list of atoms.

Files
file.pl
file.pro Prolog source file
file.po Prolog object file
file.sav Prolog saved state file
.sicstusrc sicstus.ini
SICStus Prolog initialization file, looked up in the home directory

See Also
Section 3.1 [Start], page 21, Section 4.17.1 [System Properties and Environment Variables], page 224.
13.2 mzn-sicstus — Shortcut for MiniZinc with SICStus back-end

Synopsis

% mzn-sicstus [options] mznfile

Description

This tool is a shortcut for invoking minizinc(1) with SICStus as the FlatZinc interpreter and with the appropriate global constraint definitions.

Options
See minizinc(1).

See Also
Section 10.54 [lib-zinc], page 879.
13.3 spfz — FlatZinc Interpreter

Synopsis

```
```

Description

This tool interprets the FlatZinc (`.fzn`) file `fznfile` with options taken from the command line.

Options

- `-help`, `--help`, `-?`
  print help message

- `--version`
  print version

- `-a`
  return all solutions (equal to `-n 0`)

- `-f`
  solver is free to ignore search strategy (does nothing)

- `-n N`
  number of solutions (0 = all) (default: 1)

- `-o ofile`
  file to send output to (default: standard output stream)

- `-p P`
  number of cores available (does nothing)

- `-r R`
  random seed

- `-s`
  emit statistics

- `-t`, `-time T`
  time (in ms) cutoff (default: no cutoff)

- `-search S`
  optimization method is `S`, one of `bab` (the default) and `restart`

See Also

Section 10.54 [lib-zinc], page 879.
13.4 spdet — Determinacy Checker

Synopsis

% spdet [-r] [-d] [-D] [-i ifile] fspec...

Description

The determinacy checker can help you spot unwanted nondeterminacy in your programs. This tool examines your program source code and points out places where nondeterminacy may arise.

Options

- **-r**  
  Process files recursively, fully checking the specified files and all the files they load.

- **-d**  
  Print out declarations that should be added.

- **-D**  
  Print out all needed declarations.

- **-i ifile**  
  An initialization file, which is loaded before processing begins.

See Also

Section 9.7 [The Determinacy Checker], page 363.
13.5 spld — SICStus Prolog Application Builder

Synopsis

% spld [ Option | InputFile ] ...

Description

The application builder, spld, is used for creating stand-alone executables. See Section 6.7.3 [The Application Builder], page 323, for an overview.

spld takes the files specified on the command line and combines them into an executable file, much like the UNIX ld or the Windows link commands.

Note that no pathnames passed to spld should contain spaces. Under Windows, this can be avoided by using the short version of pathnames as necessary.

Options

The input to spld can be divided into Options and Files, which can be arbitrarily mixed on the command line. Anything not interpreted as an option will be interpreted as an input file. Do not use spaces in any file or option passed to spld. Under Windows you can use the short file name for files with space in their name. The following options are available:

-?
--help       Prints out a summary of all options. This may document more options than those described in this manual.

-v
--verbose    Print detailed information about each step in the compilation/linking sequence. Multiple occurrences increase verbosity.

-vv          Same as -v -v.

--version    Prints out the version number of spld and exits successfully.

-o
--output=filename
       Specify output file name. The default depends on the linker (e.g. a.out on UNIX systems).

-E
--extended-rt
       Create an extended runtime system. In addition to the normal set of built-in runtime system predicates, extended runtime systems include the compiler. Extended runtime systems require the extended runtime library, available from SICS as an add-on product. Extended runtime systems need access to license information; see Section 6.7.3.4 [Extended Runtime Systems], page 328.

-D
--development
       Create a development system (with top level, debugger, compiler, etc.). The default is to create a runtime system. Implies --main=prolog.
--main=type

Specify what the executable should do upon startup. The possible values are:

**prolog**

Implies -D. The executable will start the Prolog top level. This is the default if -D is specified and no '.sav', '.pl', or '.po' files are specified.

**user**

The user supplies his/her own main program by including C-code (object file or source), which defines a function `user_main()`. This option is not compatible with -D. See Section 6.7.4 [User-defined Main Programs], page 330.

**restore**

The executable will restore a saved state created by `save_program/[1,2]`. This is the default if a '.sav' file is found among Files. It is only meaningful to specify one '.sav' file. If it was created by `save_program/2`, then the given startup goal is run. Then the executable will any Prolog code specified on the command line. Finally, the goal `user:runtime_entry(start)` is run. The executable exits with 0 upon normal termination and with 1 on failure or exception. Not compatible with -D.

**load**

The executable will load any Prolog code specified on the command line, i.e. files with extension '.pl' or '.po'. This is the default if there are '.pl' or '.po' but no '.sav' files among Files. Finally, the goal `user:runtime_entry(start)` is run. The executable exits with 0 upon normal termination and with 1 on failure or exception. Not compatible with -D. Note that this is almost like --main=restore except that no saved state will be restored before loading the other files.

**none**

No main function is generated. The main function must be supplied in one of the user supplied files. Not compatible with -D.

--window

*Win32 only.* Create a windowed executable. A console window will be opened and connected to the Prolog standard streams. If --main=user is specified, then `user_main()` should not set the user-stream hooks. C/C++ source code files specified on the command line will be compiled with -DSP_WIN=1 if this option is given.

--moveable
--no-moveable

Controls whether to hard-code certain paths into the executable in order for it to find the SICStus libraries and bootfiles etc.

Under UNIX, if --no-moveable is specified, then paths are hard-coded into executables in order for them to find the SICStus libraries and bootfiles. Two paths are normally hard-coded; the value of SP_PATH and, where possible, the runtime library search path using the -R linker option (or equivalent). If the linker does not support the -R option (or an equivalent), then a wrapper script is generated instead, which sets LD_LIBRARY_PATH (or equivalent).
The --moveable option turns off this behavior, so the executable is not dependent on SICStus being installed in a specific place. On most platforms the executable can figure out where it is located and so can locate any files it needs, e.g. using SP_APP_DIR and SP_RT_DIR. On some UNIX platforms, however, this is not possible. In these cases, --moveable is in effect, the executable will rely on the system properties and environment variables (SP_PATH (see Section 4.17.1 [System Properties and Environment Variables], page 224) and LD_LIBRARY_PATH etc.) to find all relevant files.

Under Windows, --moveable is always on, since Windows applications do not need to hard-code paths in order for them to find out where they are installed. On UNIX platforms, --moveable is the default (as of release 4.2) but can be turned off with --no-moveable. See Section 6.7.2 [Runtime Systems on Target Machines], page 319, for more information on how SICStus locates its libraries and bootfiles.

-S --static Link statically with SICStus runtime and foreign resources. When --static is specified, a static version of the SICStus runtime will be used and any SICStus foreign resources specified with --resources will be statically linked with the executable. In addition, --static implies --embed-rt-sav, --embed-sav-file and --resources-from-sav.

Even with --static, spld will go with the linker’s default, which is usually dynamic. If you are in a situation where you would want spld to use a static library instead of a dynamic one, then you will have to hack into spld’s configuration file spconfig-version (normally located in <installdir>/bin). We recommend that you make a copy of the configuration file and specify the new configuration file using --config=<file>. A typical modification of the configuration file for this purpose may look like:

```
[...]
TCLLIB=-Bstatic -L/usr/local/lib -ltk8.0 -ltcl8.0 -Bdynamic [...]
```

Use the new configuration file by typing

```
% spld [...] -S --config=/home/joe/hacked_spdconfig [...]  
```

The SICStus runtime depends on certain OS support that is only available in dynamically linked executables. For this reason it will probably not work to try to tell the linker to build a completely static executable, i.e. an executable that links statically also with the C library and that cannot load shared objects.

--shared Create a shared library runtime system instead of an ordinary executable. Implies --main=none.

Can be combined with --static to create a all-in-one shared library runtime system.

--resources=ResourceList

ResourceList is a comma-separated list of resource names, describing which resources should be prelinked with the executable. Names can be either simple
resource names, for example TclTk, or they can be complete paths to a foreign resource (with or without extensions). Example

```
% spld [...] --resources=tcltk,clpfd,/home/joe/foobar.so
```

This will cause `library(tcltk), library(clpfd), and /home/joe/foobar.so` to be prelinked with the executable. See also the option `--respath` below.

It is also possible to embed a data resource, that is, the contents of an arbitrary data file that can be accessed at runtime.

It is possible to embed any kind of data, but, currently, only `restore/1` knows about data resources. For this reason it only makes sense to embed `'.sav' files.

The primary reason to embed files within the executable is to create an all-in-one executable, that is, an executable file that does not depend on any other files and that therefore is easy to run on machines without SICStus installed.

See Section 6.7.3.2 [All-in-one Executables], page 324, for more information.

```
--resources-from-sav

--no-resources-from-sav

When embedding a saved state as a data resource (with `--resources` or `--embed-sav-file`), this option extracts information from the embedded saved state about the names of the foreign resources that were loaded when the saved state was created. This is the default for static executables when no other resource is specified except the embedded saved state. This option is only supported when a saved state is embedded as a data resource. See Section 6.7.3.2 [All-in-one Executables], page 324, for more information.

Use `--no-resources-from-sav` to ensure that this feature is not enabled.

```
--respath=Path

Specify additional paths used for searching for resources. `Path` is a list of search-paths, colon separated under UNIX, semicolon separated under Windows. `spld` will always search the default library directory as a last resort, so if this option is not specified, then only the default resources will be found. See also the `--resources` option above.

```
--config=ConfigFile

Specify another configuration file. This option is not intended for normal use. The file name may not contain spaces.

```
--conf VAR=VALUE since release 4.0.3

Override values from the configuration file. Can occur multiple times. For instance, `--conf CC=/usr/bin/gcc` would override the default C compiler.

```
--cflag=CFlag

`CFlag` is an option to pass to the C-compiler. This option can occur multiple times.

The current behavior is that if `CFlag` contains commas, then each comma-separated part is treated as a separate compiler option. This may change in the future, so instead you should use multiple occurrences of `--cflag`. To turn off splitting at commas and treat `CFlag` as a single option even it contains a comma, you can pass the option `--conf SPLIT_OPT_CFLAG=0`. This can be useful with certain options to the `gcc` compiler.
--cxx

This enables improved handling of C++ code. It is automatically enabled if any
C++ source code is detected.

On some platforms the C++ compiler will be used for linking instead of the C
compiler. On some platforms you may need to pass extra options to ensure the
C++ library is available.

It can be disabled with --nocxx, e.g. if the C++ support causes some backward
compatibility problem.

-- since release 4.0.3
--LD
-LD

Do not process the rest of the command line, but send it directly to the linker
step. Note that linking is often performed by the compiler.

--sicstus=Executable

spld relies on using SICStus during some stages of its execution. The default
is the development system installed with the distribution. Executable can be
used to override this, in case the user wants to use another development system.

--interactive
-i

Only applicable with --main=load or --main=restore. Calls SP_force_
interactive() (see Section 6.7.4.1 [Initializing the Prolog Engine], page 330)
before initializing SICStus.

--userhook

This option allows you to define your own version of the SU_initialize() function. SU_initialize() is called by the main program before SP_
initialize(). Its purpose is to call interface functions that must be called be-
fore SP_initialize(), such as SP_set_user_stream_hook(). It is not mean-
ingful to specify this option if --main=user or --main=none is given.

--locale=LOCALE

By default, on UNIX platforms, the executable created by spld sets the process
locale from the environment.

Setting the process locale from the environment can suppressed by passing the
--no-locale option to spld. This corresponds to the behavior prior to release
4.3.

An explicit locale that the process should set on initialization, can be passed
with the --locale option to spld.

The valid locale names depends on the operating system. Typically you can
use the locale utility, with the -a option, to list all valid locale names.
Specify the installation path for third-party software for foreign resources, such as jasper, that have special dependencies. This is mostly useful under Windows. Under UNIX, the installation script manages this automatically.

--keep
Keep temporary files and interface code and rename them to human-readable names. Not intended for the casual user, but useful if you want to know exactly what code is generated.

--nocompile
Do Not compile, just generate code. This may be useful in Makefiles, for example to generate the header file in a separate step. Implies --keep.

--namebase=namebase
Use namebase to construct the name of generated files. This defaults to spldgen_ or, if --static is specified, spldgen_s_.

--embed-rt-sav
--no-embed-rt-sav
--embed-rt-sav will embed the SICStus runtime '.sav' file into the executable. This is off by default unless --static is specified. It can be forced on (off) by specifying --embed-rt-sav (--no-embed-rt-sav).

--embed-sav-file
--no-embed-sav-file
--embed-sav-file will embed any '.sav' file passed to spld into the executable. This is just a shorthand for avoiding the ugly data resource syntax of the --resources option. This is the default when --static is specified. It can be forced on (off) by specifying --embed-sav-file (--no-embed-sav-file). A file ./foo/bar.sav will be added with the data resource name '/bar.sav', i.e. as if --resources=./foo/bar.sav=/bar.sav had been specified.

--license-file=LicenseFile
Specify the path to the license information needed by extended runtime systems. Only relevant with --extended-rt. See Section 6.7.3.4 [Extended Runtime Systems], page 328, for details.

--embed-license
--no-embed-license
Controls whether to embed the license information in the executable. --no-embed-license is the default. Only relevant with --extended-rt. See Section 6.7.3.4 [Extended Runtime Systems], page 328, for details.

--multi-sp-aware
Compile the application with support for using more than one SICStus runtime in the same process. Not compatible with --static or prelinked foreign resources. See Section 8.2 [Multiple SICStus Runtimes in C], page 349, for details.
There may be additional, undocumented, options, some of which may be described with the --help option.

**Files**

Arguments to spld not recognized as options are assumed to be input files and are handled as follows:

- `*.pro`
- `*.pl`
- `*.po`

These are interpreted as names of files containing Prolog code and will be passed to SP_load() at runtime (if --main is load or restore). **Please note:** If the intention is to make an executable that works independently of the working directory at run time, then avoid relative file names, for they will be resolved at run time, not at spld time. Use absolute file names instead, SP_APP_DIR, SP_LIBRARY_DIR, or embed a `.sav` file as a data resource, using --resource.

- `*.sav`

These are interpreted as names of files containing saved states and will be passed to SP_restore() at runtime if --main=restore is specified, subject to the above caveat about relative file names.

It is not meaningful to give more than one `.sav` argument.

- `*.so`
- `*.sl`
- `*.s.o`
- `*.o`
- `*.obj`
- `*.dll`
- `*.dylib`
- `*.c`
- `*.C`

These files are assumed to be input files to the linker and will be passed on unmodified.

- `*.cc`
- `*.cpp`
- `*.c++`

These files are assumed to be C++ source code and will be compiled by the C++ compiler before being passed to the linker.

Prior to release 4.7, these files would be compiled with the C compiler. The --nocxx option can be used if the legacy behavior is desired.

If an argument is still not recognized, then it will be passed unmodified to the linker.

**See Also**

See Section 6.7.3 [The Application Builder], page 323.
13.6 splfr — SICStus Prolog Foreign Resource Linker

Synopsis

% splfr [ Option | InputFile ] ...

Description

The foreign resource linker, splfr, is used for creating foreign resources (see Section 6.2.1 [Foreign Resources], page 291). splfr reads terms from a Prolog file, applying op declarations and extracting any foreign_resource/2 fact with first argument matching the resource name and all foreign/[2,3] facts. Based on this information, it generates the necessary glue code, and combines it with any additional C or object files provided by the user into a linked foreign resource. The output file name will be the resource name with a suitable extension.

Options

The input to splfr can be divided into Options and InputFiles and they can be arbitrarily mixed on the command line. Anything not interpreted as an option will be interpreted as an input file. Exactly one of the input files should be a Prolog file. The following options are available:

-?
  --help     Prints out a summary of all options.

-v
  --verbose  Print detailed information about each step in the compilation/linking sequence. Multiple occurrences increase verbosity.

-vv
  Same as -v -v.

--version
  Prints out the version number of spld and exits successfully.

--config=ConfigFile
  Specify another configuration file. This option is not intended for normal use. The file name may not contain spaces.

--conf VAR=VALUE since release 4.0.3
  Override values from the configuration file. Can occur multiple times. For instance, ‘--conf CC=/usr/bin/gcc’ would override the default C compiler.

--cflag=CFlag
  CFlag is an option to pass to the C-compiler (or C++ compiler). This option can occur multiple times.

  The current behavior is that if CFlag contains commas, then each comma-separated part is treated as a separate compiler option. This may change in the future, so instead you should use multiple occurrences of --cflag. To turn off splitting at commas and treat CFlag as a single option even it contains a comma, you can pass the option --conf SPLIT_OPT_CFLAG=0. This can be useful with certain options to the gcc compiler.
--cxxflag=CXXFlag \textit{since release 4.7.0}

CXXFlag is an option to pass to the C++ compiler (in addition to any C flags specified with \textit{--cflag}). This option can occur multiple times.

--cxx \textit{since release 4.7.0}

This enables improved handling of C++ code. It is automatically enabled if any C++ source code is detected.

On some platforms the C++ compiler will be used for linking instead of the C compiler. On some platforms you may need to pass extra options to ensure the C++ library is available.

It can be disabled with \textit{--nocxx}, e.g. if the C++ support causes some backward compatibility problem.

--LD \textit{since release 4.0.3}

-LD \textit{since release 4.0.3}

Do not process the rest of the command line, but send it directly to the compiler/linker. Note that linking is often performed by the compiler.

--sicstus=Executable

splfr relies on using SICStus during some stages of its execution. The default is the development system installed with the distribution. \textit{Executable} can be used to override this, in case the user wants to use another development system.

--keep

Keep temporary files and interface code and rename them to human-readable names. Not intended for the casual user, but useful if you want to know exactly what code is generated.

--resource=ResourceName

Specify the resource’s name. This defaults to the basename of the Prolog source file found on the command line.

-o, --output=OutputFileName

Specify output file name. This defaults to the name of the resource, suffixed with the platform’s standard shared object suffix (i.e. ‘.so’ on most UNIX dialects, ‘.dll’ under Windows). The use of this option is discouraged, except to change the output directory.

-S

--static

Create a statically linked foreign resource instead of a dynamically linked one, which is the default. A statically linked foreign resource is a single object file, which can be prelinked into a Prolog system. See also the spld tool, Section 6.7.3 \textit{[The Application Builder]}, page 323.

--no-rpath

Under UNIX, the default is to embed into the shared object all linker library directories for use by the dynamic linker. For most UNIX linkers this corresponds to adding a \texttt{-Rpath} for each \texttt{-Lpath}. The \textit{--no-rpath} option inhibits this.

--nocompile

Do Not compile, just generate code. This may be useful in Makefiles, for example to generate the header file in a separate step. Implies \textit{--keep}. 
--namebase=namebase

namebase will be used as part of the name of generated files. The default name
base is the resource name (e.g. as specified with --resource). If --static is
specified, then the default namebase is the resource name followed by '_s'.

--header=headername

Specify the name of the generated header file. The default is namebase_glue.h.
All C files that define foreign functions or that call SICStus API functions should
include this file. Among other things the generated header file includes proto-
types corresponding to the foreign/declarations in the Prolog code.

--multi-sp-aware

Create a (dynamic) foreign resource that can be loaded by several SICStus run-
times in the same process, at the same time. See Section 8.3 [Foreign Resources
and Multiple SICStus Runtimes], page 351, for details.

--moveable

Do Not embed paths into the foreign resource.
On platforms that support it, i.e. some versions of UNIX, the default behavior
of splfr is to add each directory dir specified with -Ldir to the search path
used by the runtime loader (using the SysV ld -R option or similar). The option
--moveable turns off this behavior. For additional details, see the correspond-
ing option to spld (see Section 6.7.3 [The Application Builder], page 323).

--structs

The Prolog source file uses library(structs). This option makes splfr un-
derstand foreign type specifications and translate them into C declarations in
the generated header file. See See Section 10.43 [lib-structs], page 759.

--objects

The Prolog source file uses library(objects). This option makes splfr un-
derstand that library's syntax extensions. See See Section 10.29 [lib-objects],
page 639.

there may be additional, undocumented, options, some of which may be described with the
--help option.

Files

Arguments to spld not recognized as options are assumed to be input files and are handled
as follows:

'*.pro'

The Prolog file containing the relevant declarations. Exactly one such argument
should be given.
 These files are assumed to be input files to the linker and will be passed on unmodified.

 These files are assumed to be C source code and will be compiled by the C-compiler before being passed to the linker.

 These files are assumed to be C++ source code and will be compiled by the C++-compiler before being passed to the linker.

 Prior to release 4.7, these files would be compiled with the C compiler. The --nocxx option can be used if the legacy behavior is desired.

 See Also

 Section 6.2.5 [The Foreign Resource Linker], page 296.
13.7 splm — SICStus Prolog License Manager

Synopsis

% splm -i Site
% splm -a LicensedProduct ExpirationDate Code

Description

SICStus Prolog requires a license code to run. You should have received from SICS your site name, the expiration date and the code. This information is normally entered during installation, but it can also be entered later on by means of this command-line tool.

Under Windows, splm must be run by a user with Administrative rights. The windowed version of SICStus (spwin.exe) has a menu item for license entry, making splm unnecessary under Windows.

Please note: when using spwin.exe for changing the license information, it too must be run with Administrative rights.

Files

library/license.pl

See Also

Section 3.1 [Start], page 21.
13.8 spxref — Cross Referencer

Synopsis

% spxref [-R] [-v] [-c] [-i ifile] [-w wfile] [-x xfile] [-m mfile] [-u ufile] fspec ...

Description

The main purpose is to find undefined predicates and unreachable code. To this end, it begins by looking for initializations, hooks and public directives to start tracing the reachable code from. If an entire application is being checked, then it also traces from user:runtime_entry/1. If individual module files are being checked, then it also traces from their export lists.

Options

File arguments should be given as atoms or as -, denoting the standard output stream.

-R Check an application, i.e. follow user:runtime_entry/1, as opposed to module declarations.
-c Generate standard compiler style error messages.
-v Verbose output. This echoes the names of the files being read.
-i ifile An initialization file, which is loaded before processing begins.
-w wfile Warning file. Warnings are written to the standard error stream by default.
-x xfile Generate a cross-reference file. This is not generated by default.
-m mfile Generate a file indicating which predicates are imported and which are exported for each file. This is not generated by default.
-u ufile Generate a file listing all the undefined predicates. This is not generated by default.

See Also

Section 9.12 [The Cross-Referencer], page 379.
References

[Aggoun & Beldiceanu 90]

[Aggoun & Beldiceanu 93]

[Beldiceanu, Carlsson, Flener & Pearson 10]

[Beldiceanu, Carlsson & Petit 04]

[Beldiceanu, Carlsson & Rampon 05]

[Beldiceanu & Contejean 94]

[Bryant 86]


[Carlsson 90]

[Carlsson & Beldiceanu 02]

[Carlsson, Beldiceanu & Martin 08]

[Carreiro & Gelernter 89a]

[Carreiro & Gelernter 89b]
[Clocksin & Mellish 81]

[Colmerauer 90]

[Diaz & Codognet 93]

[Fruehwirth 98]

[Gorlick & Kesselman 87]

[Hanak et al. 04]

[Heintze et al. 87]

[Holzbaur 92a]

[Holzbaur 94]

[Jaffar & Michaylov 87]

[Kowalski 74]

[Kowalski 79]

[Letort, Beldiceanu & Carlsson 14]
[Lopez-Ortiz 03]
A Lopez-Ortiz, CG Quimper, J Tromp, P van Beek, A fast and simple algorithm for bounds consistency of the alldifferent constraint, IJCAI 2003.

[Mehlhorn 00]

[O’Keefe 90]

[Ousterhout 94]


[Schrijvers & Demoen 04]

[Sellmann 02]

[Razakarison, Carlsson, Beldiceanu & Simonis 13]

[Robinson 65]

[Roussel 75]

[Schimpf 2002]

[Sterling & Shapiro 86]

[Van Hentenryck 89]
[Van Hentenryck et al. 95]


[Warren 83]

Predicate Index

! 1/0 (built-in, ref page)...................... 1001
! 0, cut........................................ 64

# 436
#/2 (clpfd).................................... 436
#< /2 (clpfd)................................... 436
#<=/2 (clpfd)................................. 436
#>=/2 (clpfd)................................. 436
#/>2 (clpfd).................................... 436
#>=/2 (clpfd)................................. 436
#</2 (clpfd).................................... 436
#</2 (clpfd).................................... 436
#</2 (clpfd).................................... 436
#</2 (clpfd).................................... 436
#</2 (clpfd).................................... 436

, 436
'SU_messages':generate_message/3.......... 215
'SU_messages':query_abbreviation/3........ 222
'SU_messages':query_class/5............... 222
'SU_messages':query_input/3.............. 222
'SU_messages':query_map/4.................. 222

, 436
, 436
, 436
, 436
, 436
, 436
, 436
, 436
, 436
, 436

:/2 (built-in, ref page).................... 936
:/2, conjunction.............................. 63

- 67
-2 ;/2, if then else.......................... 67
-2, if then.................................... 67
-2 (built-in, ref page)...................... 1059

: 74
:/1, directive................................ 24

; 1121
;/2 (built-in, ref page).................... 1121
;/2, disjunction.............................. 66

< 671
<-2 (objects)................................. 671
</2 (built-in, ref page)................... 1071
</2 (built-in, ref page)................... 672

= 127
= ./2 (built-in).............................. 127
= ./2 (built-in, ref page).................. 1235
= /2 (built-in)............................... 126
= /2 (built-in, ref page)................... 1233
=: /2 (built-in, ref page)................... 1015
= /2 (built-in, ref page)................... 1101
= /2 (built-in)............................... 131
= /2 (built-in, ref page)................... 1218
= /2 (built-in, ref page)................... 1100

> 1055
>/2 (built-in, ref page)..................... 1055
>=/2 (built-in, ref page)................... 1102
>=/2 (objects)................................. 673

? 23
?-/1, query................................. 23
?= /2 (built-in)............................. 127
?= /2 (built-in, ref page).................. 1226

^ 131
^/2 (built-in)............................... 131
^/2 (built-in, ref page).................... 1222
^/2 (built-in)............................... 131
^/2 (built-in, ref page).................... 1224
^/2 (built-in)............................... 131
^/2 (built-in, ref page).................... 1221
^/2 (built-in)............................... 131
^/2 (built-in, ref page).................... 1225

^/2, existential quantifier................... 74
^/2, existential quantifier................... 188
^/2, existential quantifier................... 1019
Predicate Index 1415

B

bag_add_element/4 (bags) .................. 404
bag_del_element/4 (bags) .................. 404
bag_intersect/2 (bags) ..................... 404
bag_intersection/2 (bags) .................. 404
bag_max/2 (bags) .......................... 403
bag_max/3 (bags) .......................... 403
bag_min/2 (bags) .......................... 403
bag_subtract/3 (bags) ........................ 404
bag_to_list/2 (bags) ........................ 402
bag_to_ord_set/2 (bags) .................... 402
bag_to_ord_set/3 (bags) .................... 402
bag_to_set/2 (bags) ........................ 402
bag_to_set/3 (bags) ........................ 403
bag_union/2 (bags) ........................ 404
bag_union/3 (bags) ........................ 404
bagof/3 (built-in) .......................... 188
bagof/3 (built-in, ref page) .............. 958
bagof_rd_noblock/3 (linda_client) ........ 613
bb_delete/2 (built-in) ...................... 184
bb_delete/2 (built-in, ref page) ........... 950
bb_get/2 (built-in) ........................ 184
bb_get/2 (built-in, ref page) ............. 960
bb_inf/3,5 (clpq) .......................... 503
bb_put/2 (built-in) ........................ 184
bb_put/2 (built-in, ref page) .............. 961
bb_update/3 (built-in) ..................... 184
bb_update/3 (built-in, ref page) .......... 962
begin_tests/[1,2] (plunit declaration) ... 715
between/3 (between) ....................... 412
bin_packing/2 (clpfd) ..................... 454
block/1 (built-in, ref page) ............... 963
block/1 (declaration) ........................ 84
blocked/1 (plunit option) .................. 715
bool_and/2 (clpfd) ........................ 442
bool_channel/4 (clpfd) .................... 442
bool_or/2 (clpfd) .......................... 442
bool_xor/2 (clpfd) ........................ 442
break/0 (built-in) .......................... 27, 82, 211
break/0 (built-in, ref page) .............. 965
breakpoint_expansion/2 (hook, ref page) ... 966
breakpoint_expansion/2 (user, hook) ..... 267, 282
byte_count/2 (built-in) .................... 111, 114
byte_count/2 (built-in, ref page) .......... 967
char_code/2 (built-in, ref page) ........... 973
char_conversion/2 (built-in, ref page) .... 974
character_count/2 (built-in) ............... 111, 114
character_count/2 (built-in, ref page) ... 975
checkbag/2 (bags) .......................... 402
chr_constraint/1 (CHR declaration) ....... 417
chr_flag/3 (chr) ............................ 420
chr_leash/1 (chr) ........................... 420
chr_notrace/0 (chr) ........................ 420
chr_option/2 (CHR declaration) .......... 416
chr_show_store/1 (chr) ........................ 420
chr_trace/0 (chr) ........................... 420
chr_type/1 (CHR declaration) ............. 417
circuit/[1,2] (clpfd) ....................... 449
class/1 (objects) .......................... 674
class/ancestor/2 (objects) ................ 677
class_method/1 (objects) .................. 678
class_of/2 (objects) ........................ 680
classスーパーclass/2 (objects) .......... 679
class/2,3 (built-in) ......................... 181
clause/2,3 (built-in) ....................... 976
cleanup/1 (plunit option) .................. 716
close/[1,2] (built-in, ref page) .......... 978
close/1 (built-in) .......................... 114
close_all_streams/0 (file systems) ....... 558
close_client/0 (linda_client) ............. 612
clpfd:dispatch_global/4 .................. 478
clpfd:full_answer/0 ....................... 478
clumped/2 (lists) .......................... 629
clumps/2 (lists) ........................... 629
comclient_clsid_from__progid/2 (comclient) 529
comclient_create_instance/2 (comclient) 529
comclient_equal/2 (comclient) ............ 529
comclient_exception_code/2 (comclient) ... 530
comclient_exception__culprit/2 (comclient) 530
comclient_exception__description/2 (comclient) 530
comclient_get_active__object/2 (comclient) 529
comclient_iid_from_name/2 (comclient) ... 529
comclient_invoke_method__fun/3 (comclient) 529
comclient_invoke_method__proc/2 (comclient) 529
comclient_invoke_put/3 (comclient) ....... 529
comclient_is_exception/1 (comclient) .... 530
comclient_is_object/1 (comclient) ....... 528
comclient_name_from_iid/2 (comclient) ... 529
comclient_progid_from__clsid/2 (comclient) 529
comclient_release/1 (comclient) .......... 530
comclient_valid_object/1 (comclient) ..... 529
compare/3 (built-in, ref page) .......... 980
compile/1 (built-in) ........................ 163
compile/1 (built-in, ref page) .......... 981

call/[1,2,..,255] (built-in, ref page) ...... 968
call_cleanup/2 (built-in, ref page) ....... 969
call_residue-vars/2 (built-in, ref page) .... 970
callable/1 (built-in, ref page) .......... 971
case/[3,4] (clpfd) ......................... 443
cast/1 (structs) ............................ 764
catch/3 (built-in) .......................... 198
catch/3 (built-in, ref page) .............. 972
central_moment/3 (statistics) ............. 758
char_code/2 (built-in) ........................ 128
directory_must_exist/1 (file_systems) . . . 557
directory_must_exist/2 (file_systems) . . . 557
directory_property/[2,3] (file_systems) . . . 559
disable_breakpoints/1 (built-in) . . . 274
disable_breakpoints/1 (built-in, ref page) . . . 1007
discontiguous/1 (built-in, ref page) . . . 1008
disjoint/2 (sets) . . . 84
disjoint/2 (sets, declaration) . . . 751
disjoint/2 (sets, reference) . . . 753

disjoint1/1,2 (clpfd) . . . 452
disjoint2/1,2 (clpfd) . . . 452

disjoint_union/3 (sets) . . . 478
display/1 (built-in) . . . 106
display/1 (built-in, ref page) . . . 1009

disable/1 (structures) . . . 763
do/2 (built-in, ref page) . . . 1010
do/2, do loop . . . 68
domain/3 (clpfd) . . . 435
dump/3 (clpqr) . . . 504
dynamic/1 (built-in, ref page) . . . 1012
dynamic/1 (built-in, ref page, declaration) . . . 84

edges/2 (wgraphs) . . . 871
edges/2 (wgraphs, declaration) . . . 875
element/2 (clpfd) . . . 442
element/3 (clpfd) . . . 442

elif/1 (conditional directive) . . . 88
else/0 (conditional directive) . . . 88
empty_assoc/1 (assoc) . . . 389
empty_avl/1 (avl) . . . 398
empty_bag/1 (bags) . . . 403
empty_fdset/1 (clpfd) . . . 482
empty_interval/2 (clpfd) . . . 482
empty_queue/1 (queue) . . . 745
enable_breakpoints/1 (built-in) . . . 274
enable_breakpoints/1 (built-in, ref page) . . . 1013
end_class/[0,1] (objects) . . . 689
end_tests/1 (plunit declaration) . . . 715

enotify/0 (conditional directive) . . . 88

ensure_loaded/1 (built-in) . . . 163
ensure_loaded/1 (built-in, ref page) . . . 1014
entailed/1 (clpqr) . . . 502
environ/[2,3] (system) . . . 769
erase/1 (built-in) . . . 179
erase/1 (built-in, ref page) . . . 1016
error/1 (plunit option) . . . 718
error/2 (plunit option) . . . 718
error_exception/1 (hook, ref page) . . . 1017
error_exception/1 (user, hook) . . . 198, 285
exception/1 (plunit option) . . . 718
exclude/[3,4,5] (lists) . . . 625
execution_state/[1,2] (built-in) . . . 260, 262, 274

F
false/0 (built-in, ref page) . . . 1022
fail/0 (plunit option) . . . 718
false/0 (built-in, ref page) . . . 1022
fd_batch/1 (clpfd) . . . 431
fd_closure/2 (clpfd) . . . 481
fd_degree/2 (clpfd) . . . 481
fd_domain/3 (clpfd) . . . 480
fd_global/[3,4] (clpfd) . . . 479
fd_max/2 (clpfd) . . . 480
fd_min/2 (clpfd) . . . 480
fd_neighbors/2 (clpfd) . . . 481
fd_purge/1 (clpfd) . . . 431
fd_set/2 (clpfd) . . . 481
fd_set_failures/2 (clpfd) . . . 481
fd_size/2 (clpfd) . . . 481
fd_statistics/[0,2] (clpfd) . . . 477
fd_var/1 (clpfd) . . . 480
fdg annotate/[3,4] (fdg) . . . 545
fdg assign_name/2 (fdg) . . . 536
fdg current_name/2 (fdg) . . . 536
fdg get_name/2 (fdg) . . . 537
fdg guard/3 (fdg) . . . 551
fdg label show/3 (fdg) . . . 537
fdg labeling step/2 (fdg) . . . 539
fdg legend/[1,2] (fdg) . . . 545
fdg off/0 (fdg) . . . 535
fdg on/[0,1] (fdg) . . . 535
fdg show/2 (fdg) . . . 537
fdg start labeling/1 (fdg) . . . 539
fdg transform actions/3 (fdg) . . . 545
fdset add element/3 (clpfd) . . . 482
fdset complement/2 (clpfd) . . . 483
fdset del element/3 (clpfd) . . . 483
fdset disjoint/2 (clpfd) . . . 483
fdset eq/2 (clpfd) . . . 483
fdset intersect/2 (clpfd) . . . 483
fdset intersection/[2,3] (clpfd) . . . 483
fdset interval/3 (clpfd) . . . 482
fdset max/2 (clpfd) . . . 482
fdset member/2 (clpfd) . . . 483
fdset min/2 (clpfd) . . . 482
fdset parts/4 (clpfd) . . . 482
fdset singleton/2 (clpfd) . . . 482
fdset size/2 (clpfd) . . . 482
fdset subset/2 (clpfd) . . . 483
fdset subtract/3 (clpfd) . . . 483
fdset to list/2 (clpfd) . . . 482
fdset to range/2 (clpfd) . . . 482
fdset union/[2,3] (clpfd) . . . 483
min/2 (statistics) ........................................ 757
min_assoc/3 (assoc) ........................................ 390
min_max/3 (statistics) ...................................... 758
min_member/[2,3] (lists) ................................. 627
min_path/5 (ugraphs) ...................................... 873
min_path/5 (wgraphs) ...................................... 877
min_paths/3 (ugraphs) ...................................... 873
min_paths/3 (wgraphs) ...................................... 877
min_tree/3 (ugraphs) ....................................... 877
min_tree/3 (wgraphs) ....................................... 877
minimize/[2,3] (clpfd) .................................... 471
minimize/1 (clpqr) ......................................... 503
minimum/2 (clpfd) ......................................... 434
minimum_arg/2 (clpfd) .................................... 435
ml_standard_deviation/2 (statistics) ..................... 759
ml_variance/2 (statistics) ................................. 758
mode/1 (built-in, ref page) ............................... 1086
mode/1 (declaration) ....................................... 86
mode/2 (statistics) ......................................... 758
module/[2,3] (built-in, ref page) ......................... 1087
module/[2,3] (declaration) ............................... 85, 164
multi_cumulative/[2,3] (clpfd) ......................... 451
multifile/1 (built-in, ref page) ......................... 1089
multifile/1 (declaration) .................................. 83
must_be/4 (types) ......................................... 870
mutable/1 (built-in) ........................................ 132
mutable/1 (built-in, ref page) ......................... 1091
mutarray_append/2 (mutarray) ......................... 638
mutarray_gen/3 (mutarray) ............................... 638
mutarray_get/3 (mutarray) ............................... 638
mutarray_last/2 (mutarray) .............................. 637
mutarray_length/2 (mutarray) ......................... 637
mutarray_put/3 (mutarray) ............................... 638
mutarray_set_length/2 (mutarray) ...................... 637
mutarray_to_list/2 (mutarray) ......................... 638
mutarray_update/4 (mutarray) ......................... 638
mutdict_append/2 (mutdict) ............................. 639
mutdict_clear/1 (mutdict) ............................... 639
mutdict_delete/2 (mutdict) ............................. 639
mutdict_empty/1 (mutdict) ............................... 638
mutdict_gen/3 (mutdict) ................................. 638
mutdict_get/3 (mutdict) ................................. 639
mutdict_items/2 (mutdict) ............................... 639
mutdict_keys/2 (mutdict) ............................... 639
mutdict_put/3 (mutdict) ................................. 639
mutdict_size/1 (mutdict) ............................... 638
mutdict_update/4 (mutdict) ............................. 639
mutdict_values/2 (mutdict) ............................. 639
mzn_load_file/[2,3] (zinc) ............................. 894
mzn_run_file/[1,2] (zinc) .............................. 891
mzn_run_model/[1,2] (zinc) ............................ 892
mzn_to_fzn/[2,3] (zinc) ................................. 895

M
make_directory/1 (file_systems) ......................... 557
make_sub_bag/2 (bags) .................................. 404
map_assoc/2 (assoc) .................................... 390
map_assoc/3 (assoc) .................................... 390
map_list_queue/3 (queues) .............................. 747
map_product/5 (lists) .................................. 623
map_queue/[2,3] (queues) .............................. 747
map_queue_list/[2,3] (queues) ......................... 747
map_tree/3 (trees) ....................................... 868
mapbag/2 (bags) .......................................... 402
mapbag/3 (bags) .......................................... 402
maplist/[2,3,4] (lists) ................................. 622
max/2 (statistics) ........................................ 757
max_assoc/3 (assoc) .................................... 390
max_cliques/2 (ugraphs) ................................ 874
max_member/[2,3] (lists) ............................... 627
max_path/5 (ugraphs) .................................. 873
max_path/5 (wgraphs) .................................. 877
maximize/[2,3] (clpfd) ................................. 471
maximize/1 (clpqr) .................................... 503
maximum/2 (clpfd) ...................................... 435
maximum_arg/2 (clpfd) .................................. 435
maybe/[0,1,2] (random) ................................ 748
mean/2 (statistics) ....................................... 758
median/2 (statistics) ..................................... 759
member/2 (built-in) ...................................... 128
member/2 (built-in, ref page) ......................... 1081
member/3 (bags) .......................................... 403
memberchk/2 (built-in) .................................. 128
memberchk/2 (built-in, ref page) .................... 1082
memberchk/3 (bags) ..................................... 403
merge/[3,4] (amsort) .................................... 750
message/4 (objects) ..................................... 694
message_hook/3 (hook, ref page) ..................... 1083
message_hook/3 (user, hook) ......................... 216
meta_predicate/1 (built-in, ref page) ....... 1084
meta_predicate/1 (declaration) ....................... 85, 171
min/2 (statistics) ........................................ 757

SICStus Prolog
<table>
<thead>
<tr>
<th>Predicate Index 1423</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_term/[2,3] (built-in, ref page)</td>
</tr>
<tr>
<td>read_term_from_codes/3 (codesio)</td>
</tr>
<tr>
<td>reconsult/1 (built-in, ref page)</td>
</tr>
<tr>
<td>recorda/3 (built-in)</td>
</tr>
<tr>
<td>recorda/3 (built-in, ref page)</td>
</tr>
<tr>
<td>recorded/3 (built-in, ref page)</td>
</tr>
<tr>
<td>recordz/3 (built-in)</td>
</tr>
<tr>
<td>recordz/3 (built-in, ref page)</td>
</tr>
<tr>
<td>reduce/2 (ugraphs)</td>
</tr>
<tr>
<td>reduce/2 (ugraphs)</td>
</tr>
<tr>
<td>register_event_listener/[2,3] (prologbeans)</td>
</tr>
<tr>
<td>register_query/[2,3] (prologbeans)</td>
</tr>
<tr>
<td>regular/2 (clpfd)</td>
</tr>
<tr>
<td>relation/3 (clpfd)</td>
</tr>
<tr>
<td>rem_add_link/4 (rem)</td>
</tr>
<tr>
<td>rem_create/2 (rem)</td>
</tr>
<tr>
<td>rem_equivalent/3 (rem)</td>
</tr>
<tr>
<td>rem_head/3 (random)</td>
</tr>
<tr>
<td>remove.breakpoints/1 (built-in)</td>
</tr>
<tr>
<td>remove.breakpoints/1 (built-in, ref page)</td>
</tr>
<tr>
<td>remove.dups/2 (lists)</td>
</tr>
<tr>
<td>rename.directory/2 (file_systems)</td>
</tr>
<tr>
<td>rename.file/2 (file_systems)</td>
</tr>
<tr>
<td>repeat/0 (built-in, ref page)</td>
</tr>
<tr>
<td>repeat/1 (between)</td>
</tr>
<tr>
<td>restore/1 (built-in)</td>
</tr>
<tr>
<td>restore/1 (built-in, ref page)</td>
</tr>
<tr>
<td>retract/1 (built-in)</td>
</tr>
<tr>
<td>retract/1 (built-in, ref page)</td>
</tr>
<tr>
<td>retractall/1 (built-in)</td>
</tr>
<tr>
<td>retractall/1 (built-in, ref page)</td>
</tr>
<tr>
<td>rev/2 (lists)</td>
</tr>
<tr>
<td>reverse/2 (lists)</td>
</tr>
<tr>
<td>rotate.list/[2,3] (lists)</td>
</tr>
<tr>
<td>run_tests/[0,1,2] (plunit)</td>
</tr>
<tr>
<td>runtime_entry/1 (user, hook)</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>Predicate Index 1423</th>
</tr>
</thead>
<tbody>
<tr>
<td>same_functuor/[2,3,4] (terms)</td>
</tr>
<tr>
<td>same_length/[2,3] (lists)</td>
</tr>
<tr>
<td>samkeysort/2 (samsort)</td>
</tr>
<tr>
<td>sample.standard_deviation/2 (statistics)</td>
</tr>
<tr>
<td>sample.variance/2 (statistics)</td>
</tr>
<tr>
<td>samsort/[2,3] (samsort)</td>
</tr>
<tr>
<td>sat/1 (clpb)</td>
</tr>
<tr>
<td>save.files/2 (built-in)</td>
</tr>
<tr>
<td>save.files/2 (built-in, ref page)</td>
</tr>
<tr>
<td>save_modules/2 (built-in)</td>
</tr>
<tr>
<td>save_modules/2 (built-in, ref page)</td>
</tr>
<tr>
<td>save.predicates/2 (built-in)</td>
</tr>
<tr>
<td>save.predicates/2 (built-in, ref page)</td>
</tr>
<tr>
<td>save.program/[1,2] (built-in)</td>
</tr>
<tr>
<td>save.program/[1,2] (built-in, ref page)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predicate Index 1423</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar.product/[4,5] (clpfd)</td>
</tr>
<tr>
<td>scalar.product.reif/[5,6] (clpfd)</td>
</tr>
<tr>
<td>scanlist/[4,5,6] (lists)</td>
</tr>
<tr>
<td>see/1 (built-in)</td>
</tr>
<tr>
<td>see/1 (built-in, ref page)</td>
</tr>
<tr>
<td>seeing/1 (built-in)</td>
</tr>
<tr>
<td>seeing/1 (built-in, ref page)</td>
</tr>
<tr>
<td>seek/4 (built-in)</td>
</tr>
<tr>
<td>seek/4 (built-in, ref page)</td>
</tr>
<tr>
<td>seen/0 (built-in)</td>
</tr>
<tr>
<td>seen/0 (built-in, ref page)</td>
</tr>
<tr>
<td>segment/2 (lists)</td>
</tr>
<tr>
<td>select/3 (lists)</td>
</tr>
<tr>
<td>select/4 (lists)</td>
</tr>
<tr>
<td>select_max/[3,4] (lists)</td>
</tr>
<tr>
<td>select_min/[3,4] (lists)</td>
</tr>
<tr>
<td>selectchk/3 (lists)</td>
</tr>
<tr>
<td>selectchk/4 (lists)</td>
</tr>
<tr>
<td>seq.precede.chain/[1,2] (clpfd)</td>
</tr>
<tr>
<td>session.get/4 (prologbeans)</td>
</tr>
<tr>
<td>session.put/3 (prologbeans)</td>
</tr>
<tr>
<td>set/1 (plunit option)</td>
</tr>
<tr>
<td>set.input/1 (built-in)</td>
</tr>
<tr>
<td>set.input/1 (built-in, ref page)</td>
</tr>
<tr>
<td>set.module/1 (built-in)</td>
</tr>
<tr>
<td>set.module/1 (built-in, ref page)</td>
</tr>
<tr>
<td>set.order/3 (sets)</td>
</tr>
<tr>
<td>set.output/1 (built-in)</td>
</tr>
<tr>
<td>set.output/1 (built-in, ref page)</td>
</tr>
<tr>
<td>set.prolog.flag/2 (built-in, ref page)</td>
</tr>
<tr>
<td>set.stream.position/2 (built-in)</td>
</tr>
<tr>
<td>set.stream.position/2 (built-in, ref page)</td>
</tr>
<tr>
<td>seteq/2 (sets)</td>
</tr>
<tr>
<td>setof/3 (built-in)</td>
</tr>
<tr>
<td>setof/3 (built-in, ref page)</td>
</tr>
<tr>
<td>setproduct/3 (sets)</td>
</tr>
<tr>
<td>setrand/1 (random)</td>
</tr>
<tr>
<td>setup/1 (plunit option)</td>
</tr>
<tr>
<td>shorter.list/2 (lists)</td>
</tr>
<tr>
<td>shutdown/[0,1] (prologbeans)</td>
</tr>
<tr>
<td>shutdown.server/0 (linda_client)</td>
</tr>
<tr>
<td>simple/1 (built-in, ref page)</td>
</tr>
<tr>
<td>singleton.queue/2 (queues)</td>
</tr>
<tr>
<td>size_bound/2 (terms)</td>
</tr>
<tr>
<td>skewness/2 (statistics)</td>
</tr>
<tr>
<td>skip.byte/[1,2] (built-in)</td>
</tr>
<tr>
<td>skip.byte/[1,2] (built-in, ref page)</td>
</tr>
<tr>
<td>skip.char/[1,2] (built-in)</td>
</tr>
<tr>
<td>skip.char/[1,2] (built-in, ref page)</td>
</tr>
<tr>
<td>skip.code/[1,2] (built-in)</td>
</tr>
<tr>
<td>skip.code/[1,2] (built-in, ref page)</td>
</tr>
<tr>
<td>skip.line/[0,1] (built-in)</td>
</tr>
<tr>
<td>skip.line/[0,1] (built-in, ref page)</td>
</tr>
<tr>
<td>sleep/1 (system)</td>
</tr>
<tr>
<td>smt/1 (clpfd)</td>
</tr>
<tr>
<td>socket.client.open/3 (sockets)</td>
</tr>
<tr>
<td>socket.select/7 (sockets)</td>
</tr>
</tbody>
</table>
socket_server_accept/4 (sockets) .................................. 756
socket_server_close/1 (sockets) .................................. 756
socket_server_open/2,3 (sockets) ................................. 756
solve/2 (clpfd) ......................................................... 475
some/2,3,4 (lists) ...................................................... 624
some_queue/2,3 (queues) .............................................. 747
somebag/2 (bags) ....................................................... 402
somechk/2,3,4 (lists) ................................................. 624
somechk_queue/2,3 (queues) ...................................... 747
somechkbag/2 (bags) ................................................. 402
sort/2 (built-in) ......................................................... 131
sort/2 (built-in, ref page) ........................................ 1200
sorting/3 (clpfd) ....................................................... 440
source_file/1,2 (built-in) ........................................... 136
source_file/1,2 (built-in, ref page) ............................. 1201
spy/1,2 (built-in) ....................................................... 235, 273
spy/1,2 (built-in, ref page) ......................................... 1203
start/0,1 (prologbeans) .............................................. 736
statistics/0,2 (built-in) ............................................. 146, 152
statistics/0,2 (built-in, ref page) ............................... 1204
store_slot/2 (objects) ............................................... 697
stream_code/2 (built-in) ............................................. 109, 312
stream_code/2 (built-in, ref page) ............................... 1205
stream_position/2 (built-in) ....................................... 115
stream_position/2 (built-in, ref page) ......................... 1206
stream_position_data/3 ............................................. 1207
stream_property/2 (built-in) ...................................... 114
stream_property/2 (built-in, ref page) ......................... 1208
sub_atom/5 (built-in) ............................................... 129
sub_atom/5 (built-in, ref page) ................................... 1211
sub_term/2 (terms) .................................................... 866
subcircuit/1,2 (clpfd) .............................................. 449
sublist/3,4,5 (lists) ................................................... 620
subseq/3 (lists) ......................................................... 617
subseq0/2 (lists) ....................................................... 618
subseq1/2 (lists) ....................................................... 618
subset/2 (sets) .......................................................... 751
subsumes/2 (terms) .................................................... 862
subsumes_term/2 (built-in) ....................................... 127
subsumes_term/2 (built-in, ref page) ........................... 1213
subsumeschk/2 (terms) .............................................. 862
subtract/3 (sets) ....................................................... 752
suffix/2 (lists) ........................................................ 620
suffix_length/3 (lists) .............................................. 619
sum/3 (clpfd) .......................................................... 434
sumlist/2 (lists) ....................................................... 618
sup/2,4 (clpqr) ........................................................ 502
symdiff/3 (sets) ......................................................... 753
symmetric_all_different/1 (clpfd) ............................... 440
symmetric_all_distinct/1 (clpfd) ................................ 440
symmetric_closure/2 (ugraphs) ................................ 873
symmetric_closure/2 (wgraphs) ................................ 877

T

table/2,3 (clpfd) ...................................................... 443
tail/2 (lists) .......................................................... 620
tau/2 (clp) .............................................................. 423
tcl_delete/1 (tcltk) .................................................. 829, 860
tcl_eval/3 (tcltk) ..................................................... 831, 860
tcl_event/3 (tcltk) ................................................... 834, 860
tcl_new/1 (tcltk) ..................................................... 828, 860
tell/1 (built-in) ....................................................... 111
telling/1 (built-in, ref page) ..................................... 1215
telling/1 (built-in, ref page) ..................................... 1216
term_depth/2 (terms) ................................................ 866
term Expansion/6 (hook, ref page) ......................... 1219
term expansion/6 (user, hook) ................................. 87
term_hash/[2,3,4] (terms) .................................... 862
term_order/3 (terms) ................................................ 865
term_size/2 (terms) ................................................... 867
term_subsumer/3 (terms) .......................................... 862
term_variables/2 (built-in) .................................... 127
term_variables/2 (built-in, ref page) ......................... 1227
term_variables/3 (aggregate) .................................. 388
term_variables_bag/2 (terms) .................................. 865
term_variables_set/2 (terms) .................................. 865
test/[1,2] (plunit declaration) ................................. 715
test_sub_bag/2 (bags) ............................................. 404
throw/1 (built-in) ..................................................... 197
throw/1 (built-in, ref page) ..................................... 1228
throws/1 (plunit option) .......................................... 718
time_out/3 (timeout) ................................................. 867
tk_destroy_window/1 (tcltk) .................................. 839, 861
tk_do_one_event/[0,1] (tcltk) .................................. 838, 860
tk_main_loop/0 (tcltk) .......................................... 839, 861
tk_main_window/2 (tcltk) ....................................... 839, 861
tk_make_window_exist/1 (tcltk) ............................... 839, 861
tk_new/2 (tcltk) ..................................................... 828, 860
tk_next_event/[2,3] (tcltk) ..................................... 834, 839, 861
tk_num_main_windows/1 (tcltk) .................................. 840, 861
told/0 (built-in) ..................................................... 114
told/0 (built-in, ref page) ....................................... 1229
top_sort/2 (ugraphs) .............................................. 873
top_sort/2 (wgraphs) ............................................... 877
trace/0 (built-in) ..................................................... 234
trace/0 (built-in, ref page) ..................................... 1230
transitive_closure/2 (ugraphs) ................................ 872
transitive_closure/2 (wgraphs) ................................ 876
transitive_reduction/2 (ugraphs) ............................. 872
transpose/2 (lists) ................................................... 618
transpose/ugraph/2 (ugraphs) ................................... 872
transpose/ugraph/2 (wgraphs) .................................. 876
tree_size/2 (trees) ................................................... 868
tree_to_list/2 (trees) .............................................. 868
true/0 (built-in) ...................................................... 146
true/0 (built-in, ref page) ....................................... 1232
true/0 (plunit option) ............................................. 717
true/1 (plunit option) ............................................. 717
type_definition/[2,3] (structures) ......................... 705
Keystroke Index

&
& (debugger command) .......... 241, 538

*
* (debugger command) .......... 241

+
+ (debugger command) .......... 241

-
- (debugger command) .......... 241

.
. (debugger command) .......... 242

;
; (top-level command) .......... 23

<
< (debugger command) .......... 242
< (top-level command) .......... 23

=
= (debugger command) .......... 241

?
? (debugger command) .......... 243
? (interruption command) ....... 27
? (top-level command) .......... 24

@
@ (debugger command) .......... 242

[ [ (debugger command) .......... 240

] ] (debugger command) .......... 241

^ ^ (debugger command) .......... 242
^ (top-level command) .......... 24

\ \ (debugger command) .......... 241

A a (debugger command) .......... 242
a (interruption command) ....... 27
A (debugger command) .......... 538

B b (debugger command) .......... 242
b (interruption command) ....... 27
b (top-level command) .......... 23

C c (debugger command) .......... 238
c (interruption command) ....... 27
C-c < (emacs command) .......... 36
C-c ? (emacs command) .......... 37
C-c C< (emacs command) .......... 37
C-c C-b (emacs command) ....... 35
C-c C-c (emacs command) ....... 37
C-c C-c b (emacs command) .... 35
C-c C-c f (emacs command) .... 35
C-c C-c p (emacs command) .... 35
C-c C-c r (emacs command) .... 35
C-c C-d (emacs command) ....... 36
C-c C-g (emacs command) ....... 36
C-c C-n (emacs command) ....... 37
C-c C-o (emacs command) ....... 36
C-c C-p (emacs command) ....... 35
C-c C-r (emacs command) ....... 35
C-c C-s (emacs command) ....... 37
C-c C-t (emacs command) ....... 36
C-c C-v a (emacs command) .... 37
C-c C-z (emacs command) ....... 36
C-c RET (emacs command) ...... 37
C-M-a (emacs command) ....... 36
C-M-c (emacs command) ....... 36
C-M-e (emacs command) ....... 36
C-M-h (emacs command) ....... 36
C-M-n (emacs command) ....... 36
C-M-p (emacs command) ....... 36
C-u C-c C-d (emacs command) .......................... 36
C-u C-c C-g (emacs command) .......................... 36
C-u C-c C-o (emacs command) .......................... 36
C-u C-c C-t (emacs command) .......................... 36
C-u C-c C-z (emacs command) .......................... 36
C-u C-x RET (emacs command) .......................... 37
C-u C-x SPC (emacs command) .......................... 37
C-x SPC (emacs command) ................................. 37

D
d (debugger command) .................................. 240
d (interruption command) ................................. 27
D (debugger command) ..................................... 241

E
e (debugger command) .................................. 242
e (interruption command) ................................. 27
E (debugger command) ..................................... 242

F
f (debugger command) .................................. 239

G
g (debugger command) .................................. 240

H
h (debugger command) .................................. 243
h (interruption command) ................................. 27
h (top-level command) .................................... 24

J
j<p> (debugger command) ................................ 239

L
l (debugger command) .................................. 238

M
M-; (emacs command) .................................... 36
M-{ (emacs command) ..................................... 36
M-} (emacs command) ..................................... 36
M-a (emacs command) ..................................... 36
M-e (emacs command) ..................................... 36
M-h (emacs command) ..................................... 36
M-RET (emacs command) .................................. 37

N
n (debugger command) .................................. 241
n (top-level command) .................................... 23

O
o (debugger command) .................................... 239

P
p (debugger command) .................................... 240

Q
q (debugger command) .................................... 239

R
r (debugger command) .................................... 239
RET (debugger command) .................................. 238
RET (top-level command) .................................. 23

S
s (debugger command) .................................... 238

T
t (debugger command) .................................... 240
t (interruption command) ................................. 27

U
u (debugger command) .................................... 242

V
v (debugger command) .................................... 241

W
w (debugger command) .................................... 240
W (debugger command) ..................................... 538

Y
y (top-level command) .................................... 23

Z
z (debugger command) .................................... 238
z (interruption command) ................................. 27
Book Index

! | 64
!/0 (built-in, ref page) | 1001
!/0, cut | 64

# | 423
#/ /2 (clpfd) | 436
#/ (clpfd) | 436
#/ <= /2 (clpfd) | 436
#/ => /2 (clpfd) | 436
#/ = /2 (clpfd) | 433
#/ = /2 (all_different/2 option) | 438
#/ = /2 (all_distinct/2 option) | 438
#/ = /2 (clpfd) | 433
#\ /1 (clpfd) | 436
#\ /2 (clpfd) | 436
#\ /2 (clpfd) | 436
#\ /2 (clpfd) | 436
#\ = /2 (clpfd) | 433

, | 105
'SU_messages': generate_message/3 | 215
'SU_messages': query_abbreviation/3 | 222
'SU_messages': query_class/5 | 222
'SU_messages': query_input/3 | 222
'SU_messages': query_map/4 | 222

* | 423
* /2, boolean eor | 423
* /2, multiplication (evaluable) | 121
** /2, float power (evaluable) | 124

+ | 417
+(CHR mode) | 417
+ /1, identity (evaluable) | 121
+ /2, addition (evaluable) | 121
+ /2, boolean ior | 423
+, mode annotation | 899

, | 44
, atom | 44
, /2 (built-in, ref page) | 936
, /2 (built-in, ref page) | 936
, /2 (iterator) | 73,1011
, /2 (when/2 condition) | 1246
, /2, conjunction | 63

- | 417
- (CHR mode) | 417
- /1, negation (evaluable) | 121
- /2, subtraction (evaluable) | 121
-, mode annotation | 899
-->, grammar rules | 189
--/2 (debugger show control) | 283
--> /2 ;/2, if then else | 67
--> /2, if then | 67
--> ;/2, if then else | 67
--> /2 (built-in, ref page) | 1059

. | 122
./2, identity (evaluable) | 122
., functor | 46
.emacs Emacs initialization file | 32

/ | 121
/2, floating division (evaluable) | 121
// /2, integer truncated
// division (evaluable) | 121
/\ /2, bitwise conjunction (evaluable) | 122

: | 74
: /2, module qualifier | 74
: use in meta_predicate declaration | 171
: use in Module:Goal | 164
:- /1, directive | 24

; | 121
; /2 (built-in, ref page) | 121
; /2 (when/2 condition) | 1246
; /2, disjunction | 66
<
< /2, boolean less .................................. 423
<, arithemetic less than .................................. 120
</2 (objects) .................................................. 671
</2 (built-in, ref page) ...................................... 1071
<< /2, left shift (evaluable) .................................. 122
<</2 (objects) .................................................. 672
=
=. /2 (built-in) .................................................. 127
=/.2 (map method) ............................................. 1235
=/2 (built-in) .................................................. 126
=/2 (built-in, ref page) ...................................... 1233
=/2 (clpfd:dispatch_global/4 request) ................. 479
==/2, boolean equal ........................................... 423
==, arithemetic equal ........................................ 120
==/2 (built-in, ref page) ................................... 1015
==/2, boolean less or equal .................................. 423
==, arithemetic less or equal ................................ 120
==/2 (built-in, ref page) ................................... 1101
==/2 (built-in) .................................................. 131
==/2 (built-in, ref page) ................................... 1218
=/= /2, boolean not equal .................................... 423
=/=, arithemetic not equal ................................... 120
>/2, boolean less or equal .................................. 423
>/= /2, boolean greater or equal .................................. 423
>=, arithemetic greater or equal ................................ 120
>=/2 (built-in, ref page) ................................... 1102
>> /2, right shift (evaluable) .................................. 122
>>/2 (objects) .................................................. 673
>
>/2, boolean greater ........................................... 423
>, arithemetic greater than .................................. 120
>/2 (built-in, ref page) ...................................... 1055
>=/2, boolean greater or equal .................................. 423
>=, arithemetic greater or equal ................................ 120
>=/2 (built-in, ref page) ................................... 1102
>> /2, right shift (evaluable) .................................. 122
>>/2 (objects) .................................................. 673
?
? (CHR mode) .................................................. 417
?, mode annotation ............................................ 899
?-/1, query .................................................... 23
?=/2 (built-in) .................................................. 127
?=/2 (built-in, ref page) .................................... 1226
?=/2 (when/2 condition) ...................................... 1246

?/2 (built-in) .................................................. 131
?/2 (built-in, ref page) ...................................... 1222
?=/2 (built-in) .................................................. 131
?=/2 (built-in, ref page) ...................................... 1224
?=/2 (built-in) .................................................. 131
?=/2 (built-in, ref page) ...................................... 1221
?=/2 (built-in) .................................................. 131
?=/2 (built-in, ref page) ...................................... 1225

\, empty grammar body ........................................ 190
\, empty list ..................................................... 46
\/0 (debugger condition) ..................................... 281
^/2, boolean existential quantifier ....................... 423
^/2, existential quantifier ..................................... 74
^/2, generic power (evaluable) ................................ 125
^/2 (built-in) .................................................. 188
^/2 (built-in, ref page) ...................................... 1019
^/2 (input method) ............................................. 221
^, anonymous variable ......................................... 44
^, empty grammar body ........................................ 190
^, empty list ..................................................... 46
^/0 (debugger condition) ..................................... 281
^/2, not provable ................................................ 68
^/1 (built-in, ref page) ...................................... 1103
^/2, bitwise disjunction (evaluable) ....................... 122
^/2 (built-in) .................................................. 127
^/2 (built-in, ref page) ...................................... 1104
^=/2 (built-in) .................................................. 131
^=/2 (built-in, ref page) ...................................... 1223
^, escape sequence ............................................. 60
\, escape sequence ............................................. 60
\/1, bitwise negation (evaluable) ......................... 122
\/2, bitwise exclusive or (evaluable) ................. 122
\^/2 (built-in) .................................................. 188
\^/2 (built-in, ref page) ...................................... 1103
\^/2 (input method) ............................................. 221
\^, not provable ................................................ 68
\^/1 (built-in, ref page) ...................................... 1103
\^/2, bitwise disjunction (evaluable) ....................... 122
\= /2 (built-in) .................................................. 127
\= /2 (built-in, ref page) ...................................... 1104
\= /2 (built-in) .................................................. 131
\= /2 (built-in, ref page) ...................................... 1223
\, escape sequence ............................................. 60
\\ (escape sequence) .......................................... 60
\a (escape sequence) .......................................... 60
\b (escape sequence) .......................................... 60
\d (escape sequence) .......................................... 60
\e (escape sequence) .......................................... 60
\f (escape sequence) .......................................... 60
\LFD (escape sequence) ...................................... 60
\n (escape sequence) .......................................... 60
\octal-digit...\ (escape sequence) ......................... 60
add_breakpoint/2 (built-in, ref page) ........................................... 935
add_edges/3 (ugraphs) ......................................................... 872
add_edges/3 (wgraphs) ......................................................... 876
add_element/3 (sets) .......................................................... 751
add_vertices/3 (ugraphs) ...................................................... 872
add_vertices/3 (wgraphs) ...................................................... 875
address, socket .................................................................. 754
advice breakpoint .................................................................. 243, 258
advice point (definition) .......................................................... 7
advice/0 (debugger condition) .................................................... 258, 281
tag_count (statistics key) .......................................................... 151
tag_freed (statistics key) .......................................................... 151
tag_margin (prolog flag) .......................................................... 137, 157, 302
tag_nb Freed (statistics key) ........................................................ 151
tag_time (statistics key) ............................................................ 151
aggregate/3 (aggregate) ........................................................... 388
aggregate/4 (aggregate) ........................................................... 388
aggregate_all/3 (aggregate) ....................................................... 388
aggregate_all/4 (aggregate) ....................................................... 388
aggregate ................................................................. 385
alias, of a stream .................................................................. 109
alias, stream (definition) .......................................................... 12, 19
alias/1 (open/4 option) .......................................................... 109, 1115
alias/1 (stream property) .......................................................... 1208
all (absolute_file_name/3 solutions) ........................................ 930
all (labeling/2 option) .............................................................. 474
all (maximize/3 option) ............................................................. 471
all (minimize/3 option) ............................................................. 471
all solutions, predicates for ....................................................... 189, 903
all/1 (plunit option) ................................................................. 718
all/different/1,2 (clpfd) .............................................................. 438
all/different/1,2 (clpfd) .............................................................. 439
all/different_except_0/1 (clpfd) .................................................. 439
all/distinct/1,2 (clpfd) ............................................................... 438
all/distinct_except_1/1 (clpfd) ...................................................... 439
all_equal/1 (clpfd) ................................................................. 438
all_equal/2 (clpfd) ................................................................. 438
all/1, absolute value (evaluable) .................................................. 99
absolute path ...................................................................... 100
absolute_file_name/[2,3] (built-in, ref page) ................................... 927
accept (top-level command) ...................................................... 927
acceptable_hosts/1 (start/1 option) .......................................... 736
access to streams, random ......................................................... 115
access/1 (absolute_file_name/3 option) .................................... 928
accumulating parameter ............................................................ 368
acos /1, (evaluable) ................................................................. 124
acosh /1, (evaluable) ............................................................... 124
acot /1, (evaluable) ................................................................. 124
acot2 /2, (evaluable) ............................................................... 124
acoth /1, (evaluable) ............................................................... 124
action condition, breakpoint ..................................................... 281
action execution, breakpoint ...................................................... 254
action variables, debugger ......................................................... 252, 282
action, breakpoint ................................................................. 244, 252
acyclic_term/1 (built-in, ref page) ............................................. 934
acyclic_term/2 (built-in) .......................................................... 127
add_breakpoint/2 (built-in) ....................................................... 273
\r (escape sequence) ............................................................... 60
\t (escape sequence) ............................................................... 60
\v (escape sequence) ............................................................... 60
\xhex-digit\ (escape sequence) .................................................. 60
| { visionary/2 (built-in, ref page) ........................................... 927
| /
|, list separator .................................................................. 46
~
\r, list separator .................................................................. 46
\t, boolean not .................................................................... 422
\n
A
abolish (definition) ................................................................. 7
abolish/[1,2] (built-in) ............................................................. 179
abolish/[1,2] (built-in, ref page) ............................................. 924
abort (CHR debug command) ................................................... 420
abort (debugger command) ...................................................... 242
abort/0 (built-in) ................................................................. 211
abort/0 (built-in, ref page) ..................................................... 926
abort/0 (debugger command control) ....................................... 284
abs /1, absolute value (evaluable) ............................................. 123
advice point (definition) .......................................................... 7
ancestors (definition) .............................................................. 137, 157, 302
ancestors (debugger command) ................................................. 240
ancestors (debugger command) ................................................. 240
ancestors (definition) .............................................................. 7
analysis, coverage ................................................................ 356
ancestor goal ...................................................................... 236
ancestor/2 (debugger condition) .............................................. 264, 278
ancestors (CHR debug command) .......................................... 420
ancestors (debugger command) ................................................. 240
ancestors (definition) .............................................................. 7
anonymous variable ................................................................ 44
anonymous variable (definition) ............................................... 7
ansi_conformance ................................................................. 5
anti_first_fail (labeling/2 option) ............................................. 472
any (absolute_file_name/3 file type) ....................................... 927
anystretchocc/1 (automaton/9 option) ...................................... 462
avl_member/3 (avl) .......................... 399
avl_min/2 (avl) .............................. 398
avl_min/3 (avl) .............................. 398
avl_next/3 (avl) ............................. 399
avl_next/4 (avl) ............................. 399
avl_prev/3 (avl) ............................. 399
avl_prev/4 (avl) ............................. 399
avl_range/2 (avl) ........................... 398
avl_size/2 (avl) ............................. 399
avl_store/4 (avl) ......................... 400
avl_to_list/2 (avl) .......................... 398
avoiding append ................................ 369

B
bab (labeling/2 option) ......................... 474
backtrace .................................. 240, 262
backtrace (debugger command) ............. 240
backtrace (definition) ....................... 7
backtracking .................................. 75, 76
backtracking (definition) ................. 7
backtracking, terminating a loop ......... 459
backtracks (fd_statistics/2 option) ...... 477
backtracks (zinc option value) .......... 891
backward-paragraph (emacs command) ... 36
bag ............................................. 401
bag_add_element/4 (bags) ................. 404
bag_del_element/4 (bags) ................. 404
bag_intersect/2 (bags) ...................... 404
bag_intersection/2 (bags) ................. 404
bag_max/2 (bags) ......................... 403
bag_max/3 (bags) ......................... 403
bag_min/2 (bags) ......................... 403
bag_subtract/3 (bags) ...................... 404
bag_to_list/2 (bags) ....................... 402
bag_to_ord_set/2 (bags) .................... 402
bag_to_ord_set/3 (bags) .................... 402
bag_to_set/2 (bags) ........................ 402
bag_to_set/3 (bags) ........................ 403
bag_union/2 (bags) ......................... 404
bag_union/3 (bags) ......................... 404
bagof/3 (built-in) .......................... 188
bagof/3 (built-in, ref page) ............. 958
bagof_rd_noblock/3 (linda_client) ...... 613
bb_delete/2 (built-in) .................... 184
bb_delete/2 (built-in, ref page) ....... 959
bb_get/2 (built-in) .......................... 184
bb_get/2 (built-in, ref page) .......... 960
bb_inf/[3,5] (clpqr) ....................... 503
bb_put/2 (built-in) .......................... 184
bb_put/2 (built-in, ref page) .......... 961
bb_update/3 (built-in) .................... 184
bb_update/3 (built-in, ref page) ....... 962
bbkey (argument type) ...................... 902
begin_tests/[1,2] (plunit declaration) .. 715
best (labeling/2 option) .................... 474
best (maximize/3 option) .................. 471
best (minimize/3 option) .................. 471
between/3 (between) ...................... 412
bid/1 (debugger condition) ............. 261, 280
bin_packing/2 (clpfd) .................... 454
binary tree ................................. 389, 398
binding (definition) ...................... 7
bisect (labeling/2 option) ............... 472
blackboard .................................. 184
block (predicate property) .............. 1132
block declaration ......................... 84
block/0 (debugger port value) .......... 279
block/1 (built-in, ref page) ............. 963
block/1 (declaration) ..................... 84
block/1 (predicate property) .......... 136
blocked goal ................................. 237
blocked goal (definition) ............... 7
blocked goals (debugger command) ...... 241, 538
blocked/1 (plunit option) ............... 715
body (definition) ........................... 8
body of a clause ............................ 61
bof (seek/4 method) ....................... 1185
bool_and/2 (clpfd) ......................... 442
bool_channel/4 (clpfd) ................... 442
bool_or/2 (clpfd) ........................... 442
bool_xor/2 (clpfd) .......................... 442
bound/1 (cumulatives/3 option) ......... 450
bounded (prolog flag) ..................... 137
bounding_box/2 (geost/[2,3,4] option) .. 456
bounds (all_different/2 consistency/1 value) ............. 438
bounds (all_distinct/2 consistency/1 value) .......... 438
bounds (global_cardinality/3 consistency/1 value) ........ 437
bounds (scalar_cardinality/5 consistency/1 value) ........ 434
bounds-consistent ......................... 486
bounds-disentailed ....................... 486
bounds-entailed .......................... 486
box, invocation ............................ 233
box, invocation (definition) .......... 13
box, procedure ............................ 231
box, procedure (definition) ........... 16
break (CHR debug command) .......... 420
break (debugger command) ............. 242
break (top-level command) ............. 23
break/0 (built-in) ......................... 27, 82, 211
break/0 (built-in, ref page) .......... 965
C
C errors, functions for .................................. 1255
C functions for Exceptions .......................... 1256
C functions for File and Stream Handling ...... 1256
C functions for Foreign Interface ............. 1256
C functions for I/O ................................. 1255
C functions for initialization .................. 1259
C functions for memory management ....... 1259
C functions for signal handling .............. 1260
C functions for terms ............................ 1260
C functions for type tests ........................ 1260
C functions, return values, errors .......... 1255

break_level/1 (debugger condition) ... 280, 282
breakpoint (definition) ......................... 8
breakpoint action .............................. 244, 252
breakpoint action condition ................. 281
breakpoint action execution ................. 254
breakpoint condition ....................... 244
breakpoint conditions .................. 277
breakpoint handling predicates .......... 272
breakpoint identifier ..................... 244
breakpoint processing .................... 275
breakpoint specification ................. 243
breakpoint specification (definition) .. 8
breakpoint test .............................. 244
breakpoint test condition ................. 277
breakpoint, advice ...................... 243, 258
breakpoint, debugger ..................... 243
breakpoint, generic ....................... 251
breakpoint, line .......................... 37
breakpoint, setting ....................... 37
breakpoint, specific .................... 251
breakpoint_expansion/2 (hook, ref page) ... 966
breakpoint_expansion/2 (user, hook) ..... 267, 282
buffer (definition) ......................... 8
builder, application ...................... 323
built-in operators ......................... 49
built-in operators, list of ............... 51
built-in predicate (definition) .......... 8
built-in predicates, annotations ...... 900
built-in predicates, list of .......... 903
built_in (predicate property) ........ 1132
built_in/0 (predicate property) ......... 135
Button (Tk event type) .................. 819
button (Tk widget) ...................... 793
ButtonPress (Tk event type) .......... 819
ButtonRelease (Tk event type) ........ 819
byte (argument type) ..................... 902
byte list (definition) .................... 8
byte_count/2 (built-in) ............... 111, 114
byte_count/2 (built-in, ref page) .... 967
cache_size/1 (db_open/5 option) ........ 408
call (CHR port) .......................... 419
call (leasing mode) .................... 234
call errors ................................ 77
call exceptions ................. 77
call, procedure ....................... 61, 76
call/[1,2,...,255] (built-in, ref page) . 968
call/0 (debugger port value) ........... 279
call/1 (clpfd:dispatch_global/4 request) ... 479
call_cleanup/2 (built-in, ref page) .... 969
call_residue_vars/2 (built-in, ref page) ... 970
callable term (definition) ................. 8
callable/1 (built-in, ref page) ........ 971
calling Prolog asynchronously .......... 309
calling Prolog from C ................... 306
CallSpec .................................. 528
canonical representation of atoms .... 302
canvas (Tk widget) ................... 793
card/2, boolean cardinality ........... 423
case-normalized path .................. 100
case/3,4] (clpfd) ...................... 443
cast/1 (structs) ...................... 764
Casting .................................. 764
catch/3 (built-in) ..................... 198
catch/3 (built-in, ref page) ........ 972
catching unknown procedures ........... 178
ceiling 1, (evaluable) .............. 123
central_moment/3 (statistics) ..... 758
changing prompt ...................... 104
changing type-in module .......... 166
char (argument type) ................. 902
char list (definition) .......... 8
char/1 (map method) ................. 221
char_code/2 (built-in) .............. 128
char_code/2 (built-in, ref page) .... 973
char_conversion (prolog flag) ...... 8, 137
char_conversion/2 (built-in, ref page) .... 974
character code (definition) .......... 8
character code set (definition) .... 8
character codes, arithmetic and ...... 125
character codes, lists of ............... 46
character encoding .................. 111
character escaping .................. 47
character I/O, predicates for .......... 115, 904
character set ...................... 56, 111
character, EOF ...................... 5
character, interrupt .......... 5
character-converson .................. 974, 990
character-converson mapping (definition) . 8
character-type mapping (definition) . 8
character_count/2 (built-in) ........ 111, 114
command (debugger command) ..................... 242
command-line arguments ............................. 21
command/1 (debugger condition) 253, 280, 281
commands, debug ................................. 237
comment-dwim (emacs command) .................... 36
communication, process ............................. 609, 754
compactcode (compiling flag value) ............... 80, 137
compactcode (definition) ............................ 9
compare/3 (built-in, ref page) ..................... 980
comparison of terms ................................ 130
comparison, of numbers ............................. 120
compilation, JIT .................................... 80, 227
compilation_mode/1 (load_files/2 option) ........ 1076
compile (definition) ................................ 9
compile-buffer (emacs command) ..................... 35
compile-file (emacs command) ....................... 35
compile-predicate (emacs command) ................. 35
compile-region (emacs command) ..................... 35
compile/1 (built-in) ................................ 163
compile/1 (built-in, ref page) .................... 981
compiled (predicate property) ..................... 1132
compiled/0 (predicate property) ................... 135
compiling .......................................... 21
compiling (prolog flag) 9, 80, 137
complement/2 (ugraphs) ............................. 872
compose/3 (ugraphs) ................................ 872
compound term (definition) ......................... 9
compound, terms .................................... 45
compound/1 (built-in, ref page) ................... 982
computation rule .................................. 76
ComValue .......................................... 527
concat (Tcl command) ............................... 780
concepts, FDBG ..................................... 532
condition, breakpoint ................................ 244
condition/1 (plunit option) ......................... 716
conditional compilation .............................. 88
conditional compilation (definition) ................ 9
conditional spypoint ................................ 244
conditionals ........................................ 373
conditions, breakpoint .............................. 277
conformance, ANSI ................................... 5
conjunction ........................................... 63
conjunction (definition) ............................ 9
connection_timeout/1 (odbc_db_open/4 option) ...... 708
cons/3 (lists) ....................................... 620
consistency errors .................................. 207
consistency/1 (all_different/2 option) ............ 438
consistency/1 (all_distinct/2 option) ............. 438
consistency/1 (assignment/3 option) ............... 440
consistency/1 (global_cardinality/3 option) ....... 437
consistency/1 (scalar_product/5 option) .......... 434
consistency_error/[3,4] (error class) .............. 202
consistent store ..................................... 429
console-based executable (definition) .............. 9
constant ............................................. 43
constant (definition) ................................ 9
constraint ........................................... 428
constraint checker .................................... 461
constraint event ..................................... 532
constraint, global ................................... 478
constraints (fd_statistics/2 option) ............... 477
constraints (zinc option value) ..................... 891
constraints, forgetting ................................ 431
constraints, posting .................................. 430
consult (definition) ................................... 9
consult-buffer (emacs command) ..................... 35
consult-file (emacs command) ....................... 35
consult-predicate (emacs command) ................. 35
consult-region (emacs command) ..................... 35
consult/1 (built-in, ref page) ..................... 983
consulting .......................................... 21, 22
consume_layout/1 (read_term/[2,3] option) ....... 1164
contains_term/2 (terms) ............................. 865
contains_var/2 (terms) ............................... 865
context errors ....................................... 207
context, load ........................................ 143
context-free grammars ............................... 189
context_error/[2,3] (error class) ................. 202
contradictory store .................................. 429
core (statistics key) ................................ 149
corners/1 (goest/[2,3,4] option) .................. 455
correlation/3 (statistics) ......................... 759
correspond/4 (lists) ................................. 614
cos/1, (evaluable) ................................... 123
cosh/1, (evaluable) .................................. 123
cost/2 (assignment/3 option) ........................ 440
cost/2 (global_cardinality/3 option) .............. 437
cot/1, (evaluable) ................................... 123
coth/1, (evaluable) .................................. 124
count/3, (iterator) ................................... 73
count/3 (iterator) ................................. 1010
count/4 (clpfd) ................................. 437
counter ............................................. 355
counterseq/1 (automaton/9 option) ............... 463
dif/2 (built-in) ........................................... 127
dif/2 (built-in, ref page) .............................. 1006
diffn/[1,2] (clpfd) ....................................... 453
direct_message/4 (objects) ............................. 688
direction/1 (close/2 option) ............................ 978
directive .................................................... 21, 23, 61, 62
directive (definition) .................................... 10
directives, in files being compiled .................... 82
directories .................................................. 556
directory
  (absolute_file_name/3 file type) ................ 928
directory (load context key) ........................... 143
directory specification .................................. 96
directory_exists/1 (file_systems) ...................... 557
directory_exists/2 (file_systems) ...................... 557
directory_member_of_directory/2
  (file_systems) ........................................... 558
directory_member_of_directory/3
  (file_systems) ........................................... 558
directory_member_of_directory/4
  (file_systems) ........................................... 558
  directory_members_of_
    directory/[1,2,3] (file_systems) ................ 558
directory_must_exist/1 (file_systems) ............... 557
directory_must_exist/2 (file_systems) ............... 557
directory_property/[2,3] (file_systems) ............. 559
disable this (debugger command) ...................... 242
disable_breakpoints/1 (built-in) ...................... 274
disable_breakpoints/1
  (built-in, ref page) ................................... 1007
discontiguous declaration .............................. 84
discontiguous/1 (built-in, ref page) ................. 1008
discontiguous/1 (declaration) ......................... 84
discontiguous_warnings
  (prolog flag) ........................................... 81, 84, 138, 319
discrepancy/1 (labeling/2 option) .................... 474
disjoint/2 (sets) ........................................ 751
disjoint_union/3 (sets) ................................ 753
disjoint1/[1,2] (clpfd) ................................ 452
disjoint2/[1,2] (clpfd) ................................ 452
disjunction .............................................. 66, 373
disjunction (definition) ................................ 10
disjunctive/1 (geost/[2,3,4] option) ................. 455
dispatch_global/4 (clpfd) .............................. 478
display (debugger command) ............................ 240
display/0 (debugger show control) .................... 283
display/1 (built-in) ..................................... 106
display/1 (built-in, ref page) ......................... 1009
display/1 (tk_new/2 option) ............................ 828
displaying statistics ................................... 146
dispose/1 (structs) ..................................... 763
distinctions among write predicates ................... 105
div/2, integer floored
  division (evaluable) .................................. 121
do loop ..................................................... 68
do loop (definition) .................................... 10
do/2 (built-in, ref page) .............................. 1010
do/2, do loop ............................................ 68
dom (all_different/2 on/1 value) ...................... 439
dom (all_distinct/2 on/1 value) ....................... 439
dom (global_cardinality/3 on/1 value) ............... 437
dom/1 (fd_global/[3,4] option) ....................... 479
dom_w_deg (labeling/2 option) ......................... 472
domain (all_different/2
  consistency/1 value) .................................. 438
domain (all_distinct/2
  consistency/1 value) .................................. 438
domain (global_cardinality/3
  consistency/1 value) .................................. 437
domain (scalar_product/5
  consistency/1 value) .................................. 434
domain errors ............................................ 204
domain variable ......................................... 428
domain, finite ........................................... 428
domain-consistent ....................................... 486
domain-disentailed ..................................... 486
domain-entailed ......................................... 486
domain/3 (clpfd) ........................................ 435
domain_error/[2,4] (error class) ....................... 202
double_quotes (prolog flag) ........................... 19, 47, 138
down (labeling/2 option) ............................... 473
dpgc_count (statistics key) ........................... 151
dpgc_time (statistics key) ............................ 151
dump/3 (clpqr) .......................................... 504
dynamic (predicate property) ......................... 1132
dynamic code, indexing of ............................ 178
dynamic code, semantics of ............................ 176
dynamic creation of modules ........................... 167
dynamic declaration ..................................... 84
dynamic predicate (definition) ....................... 10
dynamic predicates, importing ......................... 170
dynamic resource ........................................ 291
dynamic, procedures and declarations ................ 177
dynamic/0 (predicate property) ....................... 136
dynamic/1 (built-in, ref page) ....................... 1012
dynamic/1 (declaration) ............................... 84
dynamic_programming/1
  (geost/[2,3,4] option) ............................... 455
Eclipse ........................................ 29
edges/2 (ugraphs) ............................. 871
dges/2 (ugraphs) ............................. 875
effect, side (definition) ................... 17
efficiency and specifying streams ........ 102
efficiency, increasing ..................... 355
element/2 (clpfd) ............................ 442
element/3 (clpfd) ............................ 442
e1f/1 (conditional directive) .......... 88
e1se/0 (conditional directive) .......... 88
eMacs initialization file .emacs ........ 32
eMacs interface ................................ 32
em1 list (definition) ..................... 14
em1_assoc/1 (assoc) ...................... 389
em1_av1/1 (avl) ........................... 398
em1_bag/1 (bags) ............................ 403
em1_funcet/1 (clpfd) .................... 482
em1_inter1al/2 (clpfd) .................. 482
em1_queue/1 (queues) .................... 745
enable this (debugger command) .... 242
enable breakpoints/1 (built-in) ....... 274
enable breakpoints/1 (built-in, ref page) ............ 1013
enabling and disabling garbage collection ... 154
encoded string ......................... 290
encoded string (definition) .......... 11
encoding .................................. 11
encoding, UTF-8 .......................... 290
encoding/1 (open/4 option) .......... 1116
encoding/1 (stream property) ....... 1209
encoding_signature/1 (open/4 option) ...... 1116
encoding_signature/1 (stream property) .......... 1209
end of line .................................. 949
end of stream ............................ 950
end-of-file on character input .......... 107
end-of-file, characters ................. 965
end_class/[0,1] (objects) ............ 689
end_of_file .................................. 103
end_of_file atom .......................... 1160
end_of_stream/1 (stream property) .... 1208
end_tests/1 (plunit declaration) ..... 715
endif/0 (conditional directive) .......... 88
ensure_loaded/1 (built-in) ............ 163
ensure_loaded/1 (built-in, ref page) .... 1014
entailed/1 (clpqr) ....................... 502
entailments (fd_statistics/2 option) .... 477
Enter (Tk event type) .................. 819
entry (Tk widget) ....................... 793
enum (labeling/2 option) .......... 472
enumerating solutions to a goal ...... 187, 188
environ/[2,3] (system) .................. 769
environment variables ................. 224
environment/1 (db_open/5 option) ...... 408
eof (seek/4 method) ................. 1185
eof (Tcl command) .................... 788
eof_action/1 (open/4 option) ........ 1115
eof_action/1 (stream property) .... 1208
eof_code (open/4 eof_action value) ..... 1115
EOF character .............................. 5
eol/1 (open/4 option) .................. 1117
eol/1 (stream property) .............. 1209
Epoch ........................................ 559
equality of floats ...................... 120
equality, arithmetic ................. 120
equality, unification ................. 126
erase/1 (built-in) ....................... 179
erase/1 (built-in, ref page) .......... 1016
error (absolute_file_name/3 fileerrors value) .... 206
error (absolute_file_name/3 fileerrors) 929
error (delete_directory/2 if_nonempty option value) ...... 557
error (open/4 eof_action value) .... 1115
error (open/4 if_exists value) ...... 1118
error (overflow FD flag value) ..... 480
error (syntax_errors flag value) .... 141, 208
error (unknown flag value) .......... 142, 199
erro classes .............................. 200
error handling .......................... 285
error term ............................... 197
error, syntax ............................ 25
error/1 (plunit option) ............. 718
error/2 (plunit option) ............. 718
error_exception/1 (hook, ref page) .... 1017
error_exception/1 (user, hook) ...... 198, 285
errors ...................................... 197
ers, arithmetics ......................... 119
ers, calls .................................. 77
ers, consistency ....................... 207
ers, context .............................. 207
ers, domain ............................. 204
ers, evaluation ......................... 205
ers, existence ............................ 206
ers, instantiation ...................... 203
ers, permission ......................... 206
ers, representation .................... 205
ers, resource ............................ 208
ers, streams ............................ 109
ers, syntax .............................. 208
ers, system .............................. 209
ers, type ................................. 204
ers, uninstantiation ................... 203
escape sequence ......................... 60
escape sequence (definition) ...... 11
file, initialization ............................................ 98
file, module (definition) .................................. 15
file, PO (definition) ......................................... 16
file/1 (debugger condition) ................................. 249, 279
file_errors/1
  (absolute_file_name/3 option) ......................... 927
file_errors/1 (read_term/[2,3] option) .................. 206
file_exists/1 (file_systems) ............................... 557
file_exists/2 (file_systems) .............................. 557
file_member_of_directory/[2,3,4]
  (file_systems) ........................................... 558
file_members_of_directory/[1,2,3]
  (file_systems) ........................................... 559
file_must_exist/1 (file_systems) ......................... 557
file_must_exist/2 (file_systems) ......................... 557
file_name/1 (stream property) ......................... 1208
file_property/[2,3] (file_systems) ..................... 560
file_search_path/2 (hook, ref page) .................... 1023
file_search_path/2 (user, hook) ......................... 96
file_spec (argument type) ................................ 902
file_type/1
  (absolute_file_name/3 option) ......................... 927
fileerrors (prolog flag) ................................ 81, 110, 139, 206, 207, 929, 932, 1173, 1182, 1215
fileerrors/1
  (absolute_file_name/3 option) ......................... 929
fileerrors/1
  (read_term/[2,3] option) ............................... 206, 207
filename manipulation, predicates for .................. 101, 910
fileref/2 (source information descriptor) .............. 1054
files .................................................................. 556
files, appending to existing ................................ 110
files, creating new ......................................... 110
files, opening ............................................... 110
files, searching for in a library ......................... 95
find this (debugger command) ............................. 242
find_chr_constraint/1 (chr) ............................... 420
findall/[3,4] (built-in) .................................... 188
findall/[3,4] (built-in, ref page) ....................... 1025
finding nondeterminacy .................................... 363
finite domain .................................................. 428
first (absolute_file_name/3 solutions) ................. 930
first_bound/2 (clpfld) ...................................... 471, 473
first_fail (labeling/2 option) ............................ 471
fixall/2 (geost/[2,3,4] option) ......................... 456
fixme/1 (plunit option) .................................... 716
flit/0 (debugger command control) ...................... 283
flit/2 (debugger command control) ...................... 283
float (CHR type) ............................................ 417
float/1, coercion (evaluable) ............................ 122
float/1 (built-in, ref page) .............................. 1028

fd_purge/1 (clpfld) ........................................... 431
fd_set/2 (clpfld) ............................................ 481
fd_setfailures/2 (clpfld) .................................. 481
fd_size/2 (clpfld) ............................................ 481
fd_statistics/[0,2] (clpfld) ............................... 477
fd_var/1 (clpfld) ............................................ 480
FD predicate .................................................. 470, 489
FD set ........................................................ 481
fdbg_annotate/[3,4] (fdbg) .............................. 545
fdbg_assign_name/2 (fdbg) ............................... 536
fdbg_current_name/2 (fdbg) .............................. 536
fdbg_get_name/2 (fdbg) .................................... 537
fdbg_guard/3 (fdbg) ........................................ 551
fdbg_label_show/3 (fdbg) ................................ 537
fdbg_labeling_step/2 (fdbg) ............................. 539
fdbg_legend/[1,2] (fdbg) .................................. 545
fdbg_off/0 (fdbg) ............................................ 535
fdbg_on/[0,1] (fdbg) ....................................... 535
fdbg_output .................................................. 535
fdbg_show/2 (fdbg) ......................................... 537
fdbg_start_labeling/1 (fdbg) ............................ 539
fdbg_transform_actions/3 (fdbg) ....................... 545
FDDB concepts ............................................... 532
FDDB event ................................................... 532
FDDB output stream ........................................ 535
fdset_add_element/3 (clpfld) ............................ 482
fdset_complement/2 (clpfld) ............................. 483
fdset_del_element/3 (clpfld) ............................. 483
fdset_disjoint/2 (clpfld) ................................ 483
fdset_eq/2 (clpfld) ......................................... 483
fdset_intersect/2 (clpfld) ................................ 483
fdset_intersection/[2,3] (clpfld) ....................... 483
fdset_interval/3 (clpfld) ................................ 482
fdset_max/2 (clpfld) ....................................... 482
fdset_member/2 (clpfld) ................................... 482
fdset_min/2 (clpfld) ....................................... 482
fdset_parts/4 (clpfld) .................................... 482
fdset_singleton/2 (clpfld) ................................ 483
fdset_size/2 (clpfld) ...................................... 482
fdset_subset/2 (clpfld) ................................... 483
fdset_subtract/3 (clpfld) ................................ 483
fdset_to_list/2 (clpfld) .................................. 482
fdset_to_range/2 (clpfld) ................................ 482
fdset_union/[2,3] (clpfld) ................................ 483
fdvar_portray/3 (fdbg, hook) ............................ 543
fetch_slot/2 (objects) ..................................... 690
ff (labeling/2 option) ..................................... 471
ffc (labeling/2 option) .................................... 472
file (load context key) ................................... 143
File and Stream Handling, C functions for ............ 1256
file and stream handling, predicates for .............. 116, 910
file specification .......................................... 96, 99
file specification (definition) ......................... 11
file, closing ................................................. 114
### G

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>garbage collection</td>
<td>139, 145, 152, 924</td>
</tr>
<tr>
<td>garbage collection (definition)</td>
<td>11</td>
</tr>
<tr>
<td>garbage collection, atoms</td>
<td>156</td>
</tr>
<tr>
<td>garbage collection, enabling and disabling</td>
<td>154</td>
</tr>
<tr>
<td>garbage collection, invoking directly</td>
<td>156</td>
</tr>
<tr>
<td>garbage collection, monitoring</td>
<td>156</td>
</tr>
<tr>
<td>garbage_collect/0 (built-in)</td>
<td>156</td>
</tr>
<tr>
<td>garbage_collect/0 (built-in, ref page)</td>
<td>1042</td>
</tr>
<tr>
<td>garbage_collect_atoms/0 (built-in)</td>
<td>157</td>
</tr>
<tr>
<td>garbage_collect_atoms/0 (built-in, ref page)</td>
<td>1043</td>
</tr>
<tr>
<td>gauge</td>
<td>561</td>
</tr>
<tr>
<td>gc (prolog flag)</td>
<td>139, 154</td>
</tr>
<tr>
<td>gc_count (statistics key)</td>
<td>150</td>
</tr>
<tr>
<td>gc_freed (statistics key)</td>
<td>150</td>
</tr>
<tr>
<td>gc_margin (prolog flag)</td>
<td>139, 156</td>
</tr>
<tr>
<td>gc_time (statistics key)</td>
<td>151</td>
</tr>
<tr>
<td>gc_trace (prolog flag)</td>
<td>139, 154, 155</td>
</tr>
<tr>
<td>gcd/2, greatest common</td>
<td>123</td>
</tr>
<tr>
<td>divisor (evaluable)</td>
<td>123</td>
</tr>
<tr>
<td>gen_assoc/3 (assoc)</td>
<td>389</td>
</tr>
<tr>
<td>gen_int/1 (between)</td>
<td>413</td>
</tr>
<tr>
<td>gen_label/3 (trees)</td>
<td>868</td>
</tr>
<tr>
<td>gen_nat/1 (between)</td>
<td>412</td>
</tr>
<tr>
<td>generalization/1 (cumulatives/3 option)</td>
<td>450</td>
</tr>
<tr>
<td>generalized predicate specification (definition)</td>
<td>11</td>
</tr>
<tr>
<td>generate-and-test, use with cut</td>
<td>359</td>
</tr>
<tr>
<td>generate_debug_info (CHR flag)</td>
<td>420</td>
</tr>
<tr>
<td>generate_message/3 (hook, ref page)</td>
<td>1044</td>
</tr>
<tr>
<td>generate_message/3 (SU_messages)</td>
<td>215</td>
</tr>
<tr>
<td>generate_message_hook/3 (hook, ref page)</td>
<td>1046</td>
</tr>
<tr>
<td>generate_message_hook/3 (user, hook)</td>
<td>215</td>
</tr>
<tr>
<td>generate_unique_name (open/4)</td>
<td>1118</td>
</tr>
<tr>
<td>if_exists value</td>
<td></td>
</tr>
<tr>
<td>generic breakpoint</td>
<td>251</td>
</tr>
<tr>
<td>geometric_mean/2 (statistics)</td>
<td>758</td>
</tr>
<tr>
<td>geost/[2,3,4] (clpfd)</td>
<td>454</td>
</tr>
<tr>
<td>get/1 (debugger condition)</td>
<td>280</td>
</tr>
<tr>
<td>get_address/3 (structs)</td>
<td>763</td>
</tr>
<tr>
<td>get_assoc/3 (assoc)</td>
<td>389</td>
</tr>
<tr>
<td>get_atts/2 (Module)</td>
<td>391</td>
</tr>
<tr>
<td>get_byte/1[1,2] (built-in)</td>
<td>107</td>
</tr>
<tr>
<td>get_byte/1[1,2] (built-in, ref page)</td>
<td>1048</td>
</tr>
<tr>
<td>get_char/1[1,2] (built-in)</td>
<td>107</td>
</tr>
<tr>
<td>get_char/1[1,2] (built-in, ref page)</td>
<td>1049</td>
</tr>
<tr>
<td>get_code/1[1,2] (built-in)</td>
<td>107</td>
</tr>
<tr>
<td>get_code/1[1,2] (built-in, ref page)</td>
<td>1050</td>
</tr>
<tr>
<td>get_contents/3 (structs)</td>
<td>763</td>
</tr>
<tr>
<td>get_label/3 (trees)</td>
<td>868</td>
</tr>
<tr>
<td>get_mutable/2 (built-in)</td>
<td>132</td>
</tr>
</tbody>
</table>
### H

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt/0 [0,1] (built-in)</td>
<td>211</td>
</tr>
<tr>
<td>halt/0 [0,1] (built-in, ref page)</td>
<td>1057</td>
</tr>
<tr>
<td>handling, error</td>
<td>285</td>
</tr>
<tr>
<td>handling, exception</td>
<td>285</td>
</tr>
<tr>
<td>handling, interrupt</td>
<td>309</td>
</tr>
<tr>
<td>handling, signal</td>
<td>309</td>
</tr>
<tr>
<td>harmonic_mean/2 (statistics)</td>
<td>758</td>
</tr>
<tr>
<td>head (definition)</td>
<td>12</td>
</tr>
<tr>
<td>head of a clause</td>
<td>61</td>
</tr>
<tr>
<td>head/2 (lists)</td>
<td>620</td>
</tr>
<tr>
<td>help (CHR debug command)</td>
<td>420</td>
</tr>
<tr>
<td>help (debugger command)</td>
<td>243</td>
</tr>
<tr>
<td>help (top-level command)</td>
<td>24</td>
</tr>
<tr>
<td>hidden module</td>
<td>163</td>
</tr>
<tr>
<td>hidden/1 (module/3 option)</td>
<td>163, 1087</td>
</tr>
<tr>
<td>hide/0 (debugger condition)</td>
<td>282</td>
</tr>
<tr>
<td>hook (function annotation)</td>
<td>1255</td>
</tr>
<tr>
<td>hook (predicate annotation)</td>
<td>900</td>
</tr>
<tr>
<td>hook predicate</td>
<td>83</td>
</tr>
<tr>
<td>hook predicate (definition)</td>
<td>12, 63</td>
</tr>
<tr>
<td>hook predicates</td>
<td>912</td>
</tr>
<tr>
<td>hookable (predicate annotation)</td>
<td>900</td>
</tr>
<tr>
<td>hookable predicate (definition)</td>
<td>12</td>
</tr>
<tr>
<td>host_type (prolog flag)</td>
<td>98, 139</td>
</tr>
<tr>
<td>host_type (prolog flag, volatile)</td>
<td>1023</td>
</tr>
</tbody>
</table>

### I

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O, C functions for</td>
<td>1255</td>
</tr>
<tr>
<td>id/1 (stream property)</td>
<td>1208</td>
</tr>
<tr>
<td>id3 (table/3 order/1 value)</td>
<td>443</td>
</tr>
<tr>
<td>idempotent/1 (fd_global/[3,4] option)</td>
<td>480</td>
</tr>
<tr>
<td>identifier, breakpoint</td>
<td>244</td>
</tr>
<tr>
<td>if (Tcl command)</td>
<td>781</td>
</tr>
<tr>
<td>if/1 (conditional directive)</td>
<td>88</td>
</tr>
<tr>
<td>if/1 (load_files/2 option)</td>
<td>1076</td>
</tr>
<tr>
<td>if/3 (built-in, ref page)</td>
<td>1058</td>
</tr>
<tr>
<td>if/3, soft cut</td>
<td>68</td>
</tr>
<tr>
<td>if_exists/1 (open/4 option)</td>
<td>1117</td>
</tr>
<tr>
<td>if_nonempty/1</td>
<td>557</td>
</tr>
<tr>
<td>if_nonempty option value</td>
<td>557</td>
</tr>
<tr>
<td>ignore (delete_directory/2 option)</td>
<td>577</td>
</tr>
<tr>
<td>ignore (delete_directory/2)</td>
<td>557</td>
</tr>
<tr>
<td>ignore_ops/1 (write_term/[2,3] option)</td>
<td>1249</td>
</tr>
<tr>
<td>IID</td>
<td>527</td>
</tr>
<tr>
<td>illarg/[3,4] (types)</td>
<td>870</td>
</tr>
<tr>
<td>impact (labeling/2 option)</td>
<td>472</td>
</tr>
<tr>
<td>import (definition)</td>
<td>13</td>
</tr>
<tr>
<td>importation (importation, predicates from another module)</td>
<td>162, 164</td>
</tr>
<tr>
<td>imported_from (predicate property)</td>
<td>1132</td>
</tr>
<tr>
<td>imported_from/1 (predicate property)</td>
<td>136</td>
</tr>
<tr>
<td>importing dynamic predicates</td>
<td>170</td>
</tr>
<tr>
<td>importing predicates into modules</td>
<td>163</td>
</tr>
<tr>
<td>imports/1 (load_files/2 option)</td>
<td>1076</td>
</tr>
<tr>
<td>in/1 (linda_client)</td>
<td>613</td>
</tr>
<tr>
<td>in/2 (clpfdf)</td>
<td>435</td>
</tr>
<tr>
<td>in/2 (clpfdf:dispatch_global/4 request)</td>
<td>479</td>
</tr>
<tr>
<td>in/2 (linda_client)</td>
<td>613</td>
</tr>
<tr>
<td>in_noblock/1 (linda_client)</td>
<td>613</td>
</tr>
<tr>
<td>in_set/2 (clpfdf)</td>
<td>435</td>
</tr>
<tr>
<td>indexed (definition)</td>
<td>13</td>
</tr>
<tr>
<td>indexing (definition)</td>
<td>178</td>
</tr>
<tr>
<td>indexing, determinacy detection</td>
<td>361</td>
</tr>
<tr>
<td>indexing, dynamic code</td>
<td>361</td>
</tr>
<tr>
<td>indomain/1 (clpfdf)</td>
<td>470</td>
</tr>
<tr>
<td>inf. minus infinity</td>
<td>429</td>
</tr>
<tr>
<td>inf/2/[2,4] (clpqr)</td>
<td>502</td>
</tr>
<tr>
<td>inf/2/[2,4] (clpqr)</td>
<td>45, 48</td>
</tr>
<tr>
<td>information, source</td>
<td>35, 141</td>
</tr>
<tr>
<td>informational (prolog flag)</td>
<td>139, 319, 1390</td>
</tr>
<tr>
<td>inherit/1 (objects)</td>
<td>691</td>
</tr>
<tr>
<td>init function</td>
<td>292</td>
</tr>
<tr>
<td>init function (definition)</td>
<td>13</td>
</tr>
<tr>
<td>initialization</td>
<td>87</td>
</tr>
<tr>
<td>initialization (definition)</td>
<td>13</td>
</tr>
<tr>
<td>initialization file</td>
<td>98</td>
</tr>
<tr>
<td>initialization of saved states</td>
<td>93</td>
</tr>
<tr>
<td>initialization, C functions for</td>
<td>1259</td>
</tr>
<tr>
<td>initialization/1 (built-in, ref page)</td>
<td>1061</td>
</tr>
<tr>
<td>initialization/1 (declaration)</td>
<td>87</td>
</tr>
<tr>
<td>input</td>
<td>102</td>
</tr>
<tr>
<td>input and output of characters</td>
<td>107</td>
</tr>
<tr>
<td>input and output of terms</td>
<td>102</td>
</tr>
<tr>
<td>input and output, streams</td>
<td>102</td>
</tr>
<tr>
<td>input stream, current</td>
<td>109</td>
</tr>
<tr>
<td>input, current</td>
<td>102</td>
</tr>
<tr>
<td>input, term</td>
<td>103</td>
</tr>
</tbody>
</table>
M

macro (function annotation) .................. 1255
macro expansion ................................ 1052
main thread ...................................... 309
make_directory/1 (file_systems) ............. 557
make_sub_bag/2 (bags) ......................... 404
map_assoc/2 (assoc) ........................... 390
map_assoc/3 (assoc) ........................... 390
map_list_queue/3 (queue) ..................... 747
map_product/5 (lists) .......................... 623
map_queue/2,3 (queue) ......................... 747
map_queue_list/3 (queue) ...................... 747
map_tree/3 (trees) ............................... 868
mapbag/2 (bags) .................................. 402
mapbag/3 (bags) .................................. 402
maplist/2,3,4 (lists) ............................ 622
mapsize/1 (lmdb property) ..................... 631
mapsize/1 (lmdb option) ......................... 634
margin/3 (disjoint/2 option) .................. 452
margin/4 (disjoint/2 option) .................. 453
mark_paragraph (emacs command) ............. 36
max (all_different/2 on/1 value) ................ 439
max (all_distinct/2 on/1 value) ............... 439
max (labeling/2 option) ......................... 471
max /2, maximum value (evaluable) ............ 123
max/1 (fd_global/[3,4] option) .............. 479
max/2 (statistics) ............................... 757
max/arity (prolog flag) ......................... 139
max_assoc/3 (assoc) ............................ 390
max_cliques/2 (ugraphs) ....................... 874
max_depth/1 (write_term/[2,3] option) ....... 1250
max_integer (prolog flag) ...................... 139
max_inv/1 (debugger condition) ............... 263
max_member/2,3 (lists) ......................... 627
max_path/5 (ugraphs) ............................ 873
max_path/5 (wgraphs) ............................ 877
max_regret (labeling/2 option) ............... 472
max_tagged_integer (prolog flag) ............. 18
maximize/2,3 (clpfd) ............................ 471
maximize/1 (clpqr) ............................. 503
maximize/1 (labeling/2 option) ............... 473
maximum/2 (clpfd) .............................. 435
maximum_arg/2 (clpfd) ........................ 435
maybe/[0,1,2] (random) ......................... 748
mean/2 (statistics) .............................. 758
median (labeling/2 option) ..................... 473
median/2 (statistics) ......................... 759
member/2 (built-in) ............................. 128
member/2 (built-in, ref page) ................. 1081
member/3 (bags) .................................. 403
memberchk/2 (built-in) ......................... 128
memberchk/2 (built-in, ref page) ............. 1082
memberchk/3 (bags) ............................. 403
minimize/1 (clpqr) ........................................ 503
minimize/1 (labeling/2 option) .................. 473
minimum/2 (clpfd) .................................. 434
minimum_arg/2 (clpfd) ............................. 435
minmax (all_different/2 on/1 value) ............. 439
minmax (all_distinct/2 on/1 value) ............... 439
minmax
 (global_cardinality/3 on/1 value) .......... 437
minmax/1 (fd_global/[3,4] option) ............. 479
minus infinity, inf .................................. 429
mixing C/C++ and Prolog ......................... 289
ml_standard_deviation/2 (statistics) .......... 759
ml_variance/2 (statistics) ......................... 758
mod/2, integer floored
 remainder (evaluable) ......................... 122
mode annotations .................................. 899
mode declaration .................................. 86
mode declarations ................................ 173
mode spec .......................................... 5
mode/1 (built-in, ref page) .................... 1086
mode/1 (debugger condition) ............. 253, 280, 281
mode/1 (declaration) .............................. 86
mode/1 (lmdb option) .......................... 631, 634
mode/1 (stream property) ........... 1208
mode/2 (statistics) ............................... 758
modification, database .......................... 176
modularity, procedural .......................... 161
module (definition) ...................... 14, 161
module (load context key) .................... 143
module declaration ............................ 85, 164
module file (definition) ..................... 15
module files ...................................... 162
module files, converting into .............. 163
module files, loading ...................... 163, 164
module name expansion ...................... 5, 85, 171
module name expansion (definition) ...... 15
module name expansion, exceptions .......... 901
module prefixes on clauses .................... 167
module, declaration ............................ 162
module, hidden .................................. 163
module, source .................................. 165
module, source (definition) ................. 18
module, type-in .................................. 142, 165
module, type-in (definition) ................. 20
module/[2,3] (built-in, ref page) .......... 1087
module/[2,3] (declaration) .......... 85, 164
module/1 (debugger condition) ............ 277
modules, currently loaded .................... 168, 169
modules, debugging ............................ 168
modules, defining .............................. 162
modules, dynamic creation of ............... 167
modules, exporting predicates from ........ 162
modules, importing predicates into .......... 163
modules, loading ................................ 163
modules, loading
merge/[3,4] (samsort) ................. 750
message (Tk widget) ............................. 793
menu (Tk widget) .................................. 793
memory (statistics key) ...................... 149
Memory management, C functions for .. 1259
memory statistics ............................... 146
memory, general description ................. 144
memory, predicates for ...................... 161, 915
memory, reclamation ........................... 924
memory_buckets (statistics key) .......... 151
memory_culprit (statistics key) .......... 151
memory_free (statistics key) ................ 149
memory_used (statistics key) ............. 149
menu (Tk widget) .................................. 793
menus and queries, predicates for ........ 223, 916
messages, debugging ........................... 236
meta-call ........................................... 77
meta-call (definition) ......................... 14
meta-logical (definition) .................... 126
meta-logical predicate (definition) ....... 14
meta-logical predicates ....................... 126
meta-predicate (definition) .................. 14
meta-predicate declaration ................... 85
meta-predicate (predicate property) ...... 1132
meta_predicate (predicate property) ...... 1132
meta_predicate/1 (built-in, ref page) ..... 1084
meta_predicate/1 (declaration) ............. 85, 171
meta_predicate/1 (predicate property) .... 136
method/1 (table/3 option) ..................... 443
method/3 (Java method identifier) ......... 501
middle (labeling/2 option) ..................... 473
min (all_different/2 on/1 value) .......... 439
min (all_distinct/2 on/1 value) .......... 439
min (labeling/2 option) ....................... 471
min/2, minimum value (evaluable) ........... 123
min/1 (fd_global/[3,4] option) ............. 479
min/2 (statistics) ............................... 757
min_assoc/3 (assoc) ............................. 390
min_integer (prolog flag) ...................... 140
min_max/3 (statistics) ......................... 758
min_member/[2,3] (lists) ....................... 627
min_path/5 (ugraphs) .......................... 873
min_path/5 (wgraphs) ......................... 877
min_paths/3 (ugraphs) ......................... 873
min_paths/3 (wgraphs) ......................... 877
min_tagged_integer (prolog flag) .......... 18, 140
min_tree/3 (ugraphs) .......................... 874
min_tree/3 (wgraphs) ......................... 877
minimize/2,3 (clpfd) ......................... 471
minimize/1 (clpqr) .............................. 503
modules, name clashes ........................................... 168
modules, predicates defined in ................................ 169
modules, predicates exported from ............................ 169
modules, predicates for ........................................ 169
modules, predicates imported into ............................. 170
modules, source .................................................. 171
modules, visibility rules ....................................... 164
monitoring garbage collection .................................. 154
most general unifier ............................................. 76
most_constrained (labeling/2 option) ......................... 472
Motion (Tk event type) .......................................... 819
msb /1, most significant bit (evaluable) .................... 123
multi_cumulative/[2,3] (clpfd) ................................ 451
multifile (predicate property) ................................ 1132
multifile declaration ........................................... 83
multifile predicate (definition) ............................... 15
multifile/0 (predicate property) .............................. 136
multifile/1 (built-in, ref page) ............................... 1089
multifile/1 (declaration) ....................................... 83
multiset .................................................................... 401
must_be/4 (types) .................................................. 870
mutable .............................................................. 528
mutable term ......................................................... 132
mutable term (definition) ....................................... 15
mutable/1 (built-in) ............................................... 132
mutable/1 (built-in, ref page) ................................. 1091
mutarray_append/2 (mutarray) ................................. 638
mutarray_gen/3 (mutarray) ...................................... 638
mutarray_get/3 (mutarray) ...................................... 638
mutarray_last/2 (mutarray) ..................................... 637
mutarray_length/2 (mutarray) .................................. 637
mutarray_put/3 (mutarray) ...................................... 638
mutarray_set_length/2 (mutarray) ......................... 637
mutarray_to_list/2 (mutarray) ............................... 638
mutarray_append/2 (mutarray) ............................... 638
mutdict_append/2 (mutdict) .................................... 639
mutdict_clear/1 (mutdict) ...................................... 639
mutdict_delete/2 (mutdict) ..................................... 639
mutdict_empty/1 (mutdict) ..................................... 638
mutdict_gen/3 (mutdict) ........................................ 638
mutdict_get/3 (mutdict) ........................................ 639
mutdict_items/2 (mutdict) ...................................... 639
mutdict_keys/2 (mutdict) ....................................... 639
mutdict_put/3 (mutdict) ........................................ 639
mutdict_size/1 (mutdict) ...................................... 638
mutdict_update/4 (mutdict) .................................... 639
mutdict_values/2 (mutdict) .................................... 639
mutex ..................................................................... 306
mutual exclusion locks .......................................... 306
mzn-sicstus(1) (command line tool) ......................... 1392
mzn_load_file/[2,3] (zinc) ..................................... 894
mzn_run_file/[1,2] (zinc) ...................................... 891
mzn_run_model/[1,2] (zinc) .................................... 892
mzn_to_fzn/[2,3] (zinc) .......................................... 895

N

name auto-generation ........................................... 534
name clash .......................................................... 168
name clash (definition) ......................................... 15
name expansion, module ........................................ 171
name variable (debugger command) ......................... 538
name, of a functor ................................................ 45
name/1 (tk_new/2 option) ....................................... 828
name/2 (built-in) .................................................. 128
name/2 (built-in, ref page) .................................... 1092
names of terms ..................................................... 533, 536
natural (CHR type) ................................................ 417
neighbours/3 (ugraphs) .......................................... 872
neighbours/3 (wgraphs) ......................................... 872
neighbours/3 (wgraphs) ......................................... 875
nested execution ................................................... 27
network path ........................................................ 100
new/[2,3] (structs) ............................................... 763	new_mutarray/1 (mutarray) .................................... 637	new_mutarray/2 (mutarray) .................................... 637	new_mutdict/1 (mutdict) ...................................... 638
nextto/3 (lists) ...................................................... 615
nl/[0,1] (built-in) .................................................. 108	nl/[0,1] (built-in, ref page) .................................... 1094
noaux (table/3 method/1 value) ................................. 443
nodebug (CHR debug command) ............................... 420
nodebug (debugger command) .................................. 241
nodebug/0 (built-in) .............................................. 235
nodebug/0 (built-in, ref page) ................................. 1095
nodebug_message/0 (objects) ................................. 695
non-backtraced tests .............................................. 277
non_det/0 (plunit option) ........................................ 716
 nondeterminacy, declaring ................................... 364
 nondeterminacy, finding ....................................... 303
 nondeterminate (definition) ................................. 15
 none (--main option value) .................................. 1396
 none (absolute_file_name/3 access) ....................... 929
 nonmember/2 (built-in) ......................................... 128
 nonmember/2 (built-in, ref page) ......................... 1096
 nonvar/1 (built-in, ref page) ................................ 1097
 nonvar/1 (when/2 condition) ................................. 1246
 normalize/2 (statistics) ....................................... 759
 nospy this (debugger command) ............................. 241
 nospy/1 (built-in) ............................................... 236
 nospyall/0 (built-in) .......................................... 236
 nospyall/0 (built-in, ref page) ............................. 1099
 notation .......................................................... 5, 53
notrace/0 (built-in) .............................................. 235
notrace/0 (built-in, ref page) ................................. 1105
ord_seteq/2 (ordsets) .................. 713
ord_setproduct/3 (ordsets) .............. 713
ord_subset/2 (ordsets) .................. 713
ord_subtract/3 (ordsets) ................. 713
ord_symdiff/3 (ordsets) .................. 713
ord_union/[2,3,4] (ordsets) ............... 714
order (argument type) .................... 902
order on terms, standard .................. 130
order/1 (table/3 option) .................. 443
ordered/[1,2] (lists) ..................... 627
ordering/1 (clpqr) ........................ 504, 514
ordset_order/3 (ordsets) .................. 714
os_data (prolog flag) .................... 140
otherwise/0 (built-in, ref page) ......... 1122
out (debugger command) ................... 239
out/1 (linda_client) ....................... 612
output .................................. 102
output stream, current ..................... 109
output, current .......................... 102
output, flushing .......................... 114
output, term ................................ 104
output/0 (stream property) ............... 1208
output/1 (zinc option) .................... 886, 890
output_encoding/1 (stream property) ..... 1209
overflow (FD flag) ........................ 480
overlap/1 (geost/[2,3,4] option) ........ 456
ozn_file/1 (zinc option) .................. 886, 890

P
pair (argument type) ...................... 902
pair (definition) ........................ 16
pairfrom/4 (sets) ......................... 751
pallet_loading/1 (geost/[2,3,4] option) 456
param/1, (iterator) ........................ 73
param/1 (iterator) ......................... 1010
parameter, accumulating ................... 368
parameters/1 (zinc option) ............... 889
parconflict/1 (geost/[2,3,4] option) ...... 455
parent (definition) ....................... 16
parent_clause/1 (debugger condition) .......... 250, 279
parent_clause/2 (debugger condition) ........ 250, 279
parent_clause/3 (debugger condition) .......... 250, 279
parent_inv/1 (debugger condition) ....... 264, 278
parent_pred/1 (debugger condition) ....... 249, 279
parent_pred/2 (debugger condition) ....... 249, 279
parsing phrases .......................... 1126
partial list (definition) .................. 14
partition/5 (lists) ....................... 626
passive/1 (CHR pragma) ................... 416
password/1 (odbc_db_open/4 option) ........ 708
path root ................................ 99
path, absolute ................................ 99, 100
path, case-normalized ........................ 100
path, network ................................ 100
path, UNC .................................. 100
path/1 (lmdb property) .................... 633
path/3 (ugraphs) .......................... 873
path/3 (wgraphs) .......................... 877
peek_byte/[1,2] (built-in) ................. 107
peek_byte/[1,2] (built-in, ref page) ...... 1123
peek_char/[1,2] (built-in) ................. 107
peek_char/[1,2] (built-in, ref page) ...... 1124
peek_code/[1,2] (built-in) ................ 107
peek_code/[1,2] (built-in, ref page) ...... 1125
perm/2 (lists) ............................. 616
perm2/4 (lists) ............................ 616
permission errors .......................... 206
permission/1 (lmdb option) ............... 631
permission_error/[3,5] (error class) ....... 202
permutation/1 (keysorting/3 option) ....... 441
permutation/2 (lists) ...................... 616
phrase/[2,3] (built-in) ..................... 191
phrase/[2,3] (built-in, ref page) .......... 1126
pi/0, pi (evaluable) ....................... 125
placement of cut ......................... 359
plain spypoint ............................ 235, 243
platform_data (prolog flag) .............. 140
pltrace-break (emacs command) ........... 37
plus infinity, sup ......................... 429
PO file (definition) ...................... 16
PO files .................................. 92
PO files, loading .......................... 94
PO files, saving ........................... 94
point, advice (definition) ............... 7
pointer_object/2 (objects) ............... 696
polymorphism/1 (geost/[2,3,4] option) ...... 456
population_standard_
    deviation/2 (statistics) .............. 759
population_variance/2 (statistics) ........ 758
port (definition) .......................... 16
port debugger ............................. 232
port, debugger ............................ 279
port/1 (debugger condition) .............. 250, 279
port/1 (start/1 option) .................... 736
portable (quoted_charset flag value) ..... 138
portray/1 (hook, ref page) ............... 1128
portray/1 (user, hook) .................... 106, 523
portray_assoc/1 (assoc) .................. 390
portray_avl/1 (avl) ....................... 399
portray_bag/1 (bags) ...................... 401
portray_clause/[1,2] (built-in) .......... 106
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>portray_clause/[1,2]</td>
<td>1129</td>
</tr>
<tr>
<td>(built-in, ref page)</td>
<td></td>
</tr>
<tr>
<td>portray_message/2 (hook, ref page)</td>
<td>1131</td>
</tr>
<tr>
<td>portray_message/2 (user, hook)</td>
<td>215</td>
</tr>
<tr>
<td>portray_queue/1 (queues)</td>
<td>745</td>
</tr>
<tr>
<td>portrayed/1 (write_term/[2,3] option)</td>
<td>1249</td>
</tr>
<tr>
<td>position in a stream</td>
<td>115</td>
</tr>
<tr>
<td>position, stream</td>
<td>114</td>
</tr>
<tr>
<td>position, stream (definition)</td>
<td>19</td>
</tr>
<tr>
<td>position/1 (stream property)</td>
<td>1208</td>
</tr>
<tr>
<td>post/1 (zinc option)</td>
<td>889</td>
</tr>
<tr>
<td>postfix operators</td>
<td>45, 48</td>
</tr>
<tr>
<td>posting constraints</td>
<td>430</td>
</tr>
<tr>
<td>power_set/2 (sets)</td>
<td>752</td>
</tr>
<tr>
<td>precedence (definition)</td>
<td>16</td>
</tr>
<tr>
<td>precedence, of operators</td>
<td>48</td>
</tr>
<tr>
<td>precedences/1 (cumulative/2 option)</td>
<td>449</td>
</tr>
<tr>
<td>precedences/1</td>
<td></td>
</tr>
<tr>
<td>(multi_cumulatives/3 option)</td>
<td>452</td>
</tr>
<tr>
<td>pred/1 (debugger condition)</td>
<td>243, 277</td>
</tr>
<tr>
<td>pred_spec (argument type)</td>
<td>902</td>
</tr>
<tr>
<td>pred_spec/forest (argument type)</td>
<td>902</td>
</tr>
<tr>
<td>pred_spec_tree (argument type)</td>
<td>902</td>
</tr>
<tr>
<td>predicate</td>
<td>61</td>
</tr>
<tr>
<td>predicate (definition)</td>
<td>16</td>
</tr>
<tr>
<td>predicate declaration</td>
<td>83</td>
</tr>
<tr>
<td>predicate specification (definition)</td>
<td>16</td>
</tr>
<tr>
<td>predicate specification,</td>
<td></td>
</tr>
<tr>
<td>generalized (definition)</td>
<td>11</td>
</tr>
<tr>
<td>predicate, built-in (definition)</td>
<td>8</td>
</tr>
<tr>
<td>predicate, dynamic (definition)</td>
<td>10</td>
</tr>
<tr>
<td>predicate, FD</td>
<td>470, 489</td>
</tr>
<tr>
<td>predicate, foreign (definition)</td>
<td>11</td>
</tr>
<tr>
<td>predicate, hook</td>
<td>83</td>
</tr>
<tr>
<td>predicate, hook (definition)</td>
<td>12</td>
</tr>
<tr>
<td>predicate, hookable (definition)</td>
<td>12</td>
</tr>
<tr>
<td>predicate, meta-logical (definition)</td>
<td>14</td>
</tr>
<tr>
<td>predicate, multifile (definition)</td>
<td>15</td>
</tr>
<tr>
<td>predicate, public</td>
<td>162</td>
</tr>
<tr>
<td>predicate, static (definition)</td>
<td>18</td>
</tr>
<tr>
<td>predicate, steadfast (definition)</td>
<td>18</td>
</tr>
<tr>
<td>predicate, undefined</td>
<td>26, 142, 1238</td>
</tr>
<tr>
<td>predicate_property/2 (built-in)</td>
<td>135, 170</td>
</tr>
<tr>
<td>(built-in, ref page)</td>
<td></td>
</tr>
<tr>
<td>predicates defined in modules</td>
<td>1132</td>
</tr>
<tr>
<td>predicates exported from modules</td>
<td>169</td>
</tr>
<tr>
<td>predicates for all solutions</td>
<td>189, 903</td>
</tr>
<tr>
<td>predicates for arithmetic</td>
<td>125, 903</td>
</tr>
<tr>
<td>predicates for character I/O</td>
<td>115, 904</td>
</tr>
<tr>
<td>predicates for control</td>
<td>78, 905</td>
</tr>
<tr>
<td>predicates for database</td>
<td>185, 906</td>
</tr>
<tr>
<td>predicates for debugging</td>
<td>285, 908</td>
</tr>
<tr>
<td>predicates for execution state</td>
<td>211, 909</td>
</tr>
<tr>
<td>predicates for file and stream handling</td>
<td>116, 910</td>
</tr>
<tr>
<td>predicates for filename manipulation</td>
<td>101, 910</td>
</tr>
<tr>
<td>predicates for foreign interface</td>
<td>911</td>
</tr>
<tr>
<td>predicates for grammar rules</td>
<td>196, 912</td>
</tr>
<tr>
<td>predicates for list processing</td>
<td>135, 913</td>
</tr>
<tr>
<td>predicates for loading programs</td>
<td>91, 914</td>
</tr>
<tr>
<td>predicates for looking at terms</td>
<td>126</td>
</tr>
<tr>
<td>predicates for memory</td>
<td>161, 915</td>
</tr>
<tr>
<td>predicates for messages and queries</td>
<td>223, 916</td>
</tr>
<tr>
<td>predicates for modules</td>
<td>176, 917</td>
</tr>
<tr>
<td>predicates for processing lists</td>
<td>128</td>
</tr>
<tr>
<td>predicates for program state</td>
<td>143, 917</td>
</tr>
<tr>
<td>predicates for saving programs</td>
<td>95, 918</td>
</tr>
<tr>
<td>predicates for term comparison</td>
<td>133, 919</td>
</tr>
<tr>
<td>predicates for term handling</td>
<td>133, 919</td>
</tr>
<tr>
<td>predicates for term I/O</td>
<td>118, 921</td>
</tr>
<tr>
<td>predicates for type tests</td>
<td>126, 132, 922</td>
</tr>
<tr>
<td>predicates imported into modules</td>
<td>170</td>
</tr>
<tr>
<td>predicates, annotations</td>
<td>900</td>
</tr>
<tr>
<td>predicates, assertion and retraction</td>
<td>176</td>
</tr>
<tr>
<td>predicates, breakpoint handling</td>
<td>272</td>
</tr>
<tr>
<td>predicates, debugging</td>
<td>233</td>
</tr>
<tr>
<td>predicates, hook</td>
<td>912</td>
</tr>
<tr>
<td>predicates, importing dynamic</td>
<td>170</td>
</tr>
<tr>
<td>predicates, meta-logical</td>
<td>126</td>
</tr>
<tr>
<td>predicates, private</td>
<td>162</td>
</tr>
<tr>
<td>predicates, public</td>
<td>162</td>
</tr>
<tr>
<td>predicates, read</td>
<td>103</td>
</tr>
<tr>
<td>predicates, write</td>
<td>104</td>
</tr>
<tr>
<td>prefix operators</td>
<td>45, 48</td>
</tr>
<tr>
<td>prefix/2 (lists)</td>
<td>620</td>
</tr>
<tr>
<td>prefix_length/3 (lists)</td>
<td>619</td>
</tr>
<tr>
<td>preinit (function annotation)</td>
<td>1255</td>
</tr>
<tr>
<td>prelinked foreign resource (definition)</td>
<td>16</td>
</tr>
<tr>
<td>prelinked resource</td>
<td>292</td>
</tr>
<tr>
<td>principal functor</td>
<td>127</td>
</tr>
<tr>
<td>principal functor (definition)</td>
<td>45</td>
</tr>
<tr>
<td>print (debugger command)</td>
<td>240</td>
</tr>
<tr>
<td>print/[1,2] (built-in)</td>
<td>106</td>
</tr>
<tr>
<td>print/[1,2] (built-in, ref page)</td>
<td>1134</td>
</tr>
<tr>
<td>print/0 (debugger show control)</td>
<td>283</td>
</tr>
<tr>
<td>print_coverage/[0,1] (built-in)</td>
<td>357</td>
</tr>
<tr>
<td>print_coverage/[0,1]</td>
<td></td>
</tr>
<tr>
<td>(built-in, ref page)</td>
<td>1135</td>
</tr>
<tr>
<td>print_message/2 (built-in)</td>
<td>215</td>
</tr>
<tr>
<td>print_message/2 (built-in, ref page)</td>
<td>1137</td>
</tr>
<tr>
<td>print_message_lines/3 (built-in)</td>
<td>216</td>
</tr>
<tr>
<td>print_message_lines/3</td>
<td></td>
</tr>
<tr>
<td>(built-in, ref page)</td>
<td></td>
</tr>
<tr>
<td>print_profile/[0,1] (built-in)</td>
<td>1139</td>
</tr>
<tr>
<td>print_profile/[0,1]</td>
<td>356</td>
</tr>
<tr>
<td>(built-in, ref page)</td>
<td>1140</td>
</tr>
<tr>
<td>Procedure</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Printing clauses</td>
<td>106</td>
</tr>
<tr>
<td>Printing, formatted</td>
<td>108</td>
</tr>
<tr>
<td>Priority/1 (write_term/[2,3] option)</td>
<td>1250</td>
</tr>
<tr>
<td>Private predicates</td>
<td>162</td>
</tr>
<tr>
<td>Private/1 (debugger condition)</td>
<td>265, 281</td>
</tr>
<tr>
<td>Proc (Tcl command)</td>
<td>789</td>
</tr>
<tr>
<td>Procedural modularity</td>
<td>161</td>
</tr>
<tr>
<td>Procedural semantics</td>
<td>74</td>
</tr>
<tr>
<td>Procedure</td>
<td>63</td>
</tr>
<tr>
<td>Procedure (definition)</td>
<td>16</td>
</tr>
<tr>
<td>Procedure box</td>
<td>231</td>
</tr>
<tr>
<td>Procedure box (definition)</td>
<td>16</td>
</tr>
<tr>
<td>Procedure call</td>
<td>61, 76</td>
</tr>
<tr>
<td>Procedure definition</td>
<td>76</td>
</tr>
<tr>
<td>Procedures, dynamic and static</td>
<td>177</td>
</tr>
<tr>
<td>Procedures, redefining during execution</td>
<td>82</td>
</tr>
<tr>
<td>Procedures, removing properties</td>
<td>924</td>
</tr>
<tr>
<td>Procedures, self-modifying</td>
<td>176, 181</td>
</tr>
<tr>
<td>Proceed (redefine_warnings flag value)</td>
<td>141</td>
</tr>
<tr>
<td>Proceed/0 (debugger command control)</td>
<td>283</td>
</tr>
<tr>
<td>Proceed/2 (debugger command control)</td>
<td>283</td>
</tr>
<tr>
<td>Process communication</td>
<td>609, 754</td>
</tr>
<tr>
<td>Process_create/[2,3] (process)</td>
<td>728</td>
</tr>
<tr>
<td>Process_id/1 (process)</td>
<td>731</td>
</tr>
<tr>
<td>Process_id/2 (process)</td>
<td>731</td>
</tr>
<tr>
<td>Process_kill/[1,2] (process)</td>
<td>731</td>
</tr>
<tr>
<td>Process_release/1 (process)</td>
<td>731</td>
</tr>
<tr>
<td>Process_wait/[2,3] (process)</td>
<td>730</td>
</tr>
<tr>
<td>Processing, breakpoint</td>
<td>275</td>
</tr>
<tr>
<td>Profile_data/1 (built-in)</td>
<td>356</td>
</tr>
<tr>
<td>Profile_data/1 (built-in, ref page)</td>
<td>1141</td>
</tr>
<tr>
<td>Profile_reset/0 (built-in)</td>
<td>356, 357</td>
</tr>
<tr>
<td>Profile_reset/0 (built-in, ref page)</td>
<td>1142</td>
</tr>
<tr>
<td>Profiling</td>
<td>561</td>
</tr>
<tr>
<td>Profiling (definition)</td>
<td>16</td>
</tr>
<tr>
<td>Profiling (prolog flag)</td>
<td>138, 319</td>
</tr>
<tr>
<td>Profiling, execution</td>
<td>355</td>
</tr>
<tr>
<td>ProgID</td>
<td>527</td>
</tr>
<tr>
<td>Program (definition)</td>
<td>16</td>
</tr>
<tr>
<td>Program (statistics key)</td>
<td>149</td>
</tr>
<tr>
<td>Program space</td>
<td>145</td>
</tr>
<tr>
<td>Program state</td>
<td>28</td>
</tr>
<tr>
<td>Program state, predicates for</td>
<td>143, 917</td>
</tr>
<tr>
<td>Program, loading</td>
<td>79</td>
</tr>
<tr>
<td>Programming in logic</td>
<td>1</td>
</tr>
<tr>
<td>Project_attributes/2 (Module)</td>
<td>393</td>
</tr>
<tr>
<td>Projecting_assert/1 (clpqr)</td>
<td>505</td>
</tr>
<tr>
<td>Prolog (~main option value)</td>
<td>1396</td>
</tr>
<tr>
<td>Prolog (quoted_charset flag value)</td>
<td>138</td>
</tr>
<tr>
<td>Prolog-backward-list (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-beginning-of-clause</td>
<td>36</td>
</tr>
<tr>
<td>(emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-debug-on (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-end-of-clause (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-end-of-predicate (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-forward-list (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-help-on-predicate (emacs command)</td>
<td>37</td>
</tr>
<tr>
<td>Prolog-insert-next-clause</td>
<td>37</td>
</tr>
<tr>
<td>(emacs command)</td>
<td>37</td>
</tr>
<tr>
<td>Prolog-insert-predicate-template</td>
<td>37</td>
</tr>
<tr>
<td>(emacs command)</td>
<td>37</td>
</tr>
<tr>
<td>Prolog-insert-predspec (emacs command)</td>
<td>37</td>
</tr>
<tr>
<td>Prolog-mark-clause (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-mark-predicate (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-trace-on (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog-variables-to-anonymous</td>
<td>37</td>
</tr>
<tr>
<td>(emacs command)</td>
<td>37</td>
</tr>
<tr>
<td>Prolog-zip-on (emacs command)</td>
<td>36</td>
</tr>
<tr>
<td>Prolog_flag/[2,3] (built-in)</td>
<td>103</td>
</tr>
<tr>
<td>Prolog_flag/[2,3] (built-in, ref page)</td>
<td>1143</td>
</tr>
<tr>
<td>Prolog_load_context/2</td>
<td>1145</td>
</tr>
<tr>
<td>(built-in, ref page)</td>
<td>1145</td>
</tr>
<tr>
<td>PROLOGICSIZE (system property)</td>
<td>227</td>
</tr>
<tr>
<td>PROLOGINITSIZE (system property)</td>
<td>146, 227</td>
</tr>
<tr>
<td>PROLOGKEEPSIZE (system property)</td>
<td>146, 227</td>
</tr>
<tr>
<td>PROLOGMAXSIZE (system property)</td>
<td>227</td>
</tr>
<tr>
<td>Prompt, changing</td>
<td>104</td>
</tr>
<tr>
<td>Prompt/2 (built-in)</td>
<td>104</td>
</tr>
<tr>
<td>Prompt/2 (built-in, ref page)</td>
<td>1146</td>
</tr>
<tr>
<td>Propagating indexicals</td>
<td>487</td>
</tr>
<tr>
<td>Proper list (definition)</td>
<td>14</td>
</tr>
<tr>
<td>Proper_length/2 (lists)</td>
<td>616</td>
</tr>
<tr>
<td>Proper_prefix/2 (lists)</td>
<td>620</td>
</tr>
<tr>
<td>Proper_prefix_length/3 (lists)</td>
<td>619</td>
</tr>
<tr>
<td>Proper_segment/2 (lists)</td>
<td>621</td>
</tr>
<tr>
<td>Proper_suffix/2 (lists)</td>
<td>621</td>
</tr>
<tr>
<td>Proper_suffix_length/3 (lists)</td>
<td>619</td>
</tr>
<tr>
<td>Proper, stream</td>
<td>1208</td>
</tr>
<tr>
<td>Property, stream (definition)</td>
<td>19</td>
</tr>
<tr>
<td>Prototype, function</td>
<td>5</td>
</tr>
<tr>
<td>Prune/1 (cumulatives/3 option)</td>
<td>450</td>
</tr>
<tr>
<td>Prunings (fd_statistics/2 option)</td>
<td>477</td>
</tr>
<tr>
<td>Prunings (zinc option value)</td>
<td>891</td>
</tr>
<tr>
<td>Public declaration</td>
<td>86</td>
</tr>
<tr>
<td>Public predicate</td>
<td>162</td>
</tr>
<tr>
<td>Public predicates</td>
<td>162</td>
</tr>
<tr>
<td>Public/1 (built-in, ref page)</td>
<td>1147</td>
</tr>
<tr>
<td>Public/1 (declaration)</td>
<td>86</td>
</tr>
<tr>
<td>Put_assoc/4 (assoc)</td>
<td>391</td>
</tr>
<tr>
<td>Put_atts/2 (Module)</td>
<td>392</td>
</tr>
<tr>
<td>Put_byte/[1,2] (built-in)</td>
<td>108</td>
</tr>
<tr>
<td>Put_byte/[1,2] (built-in, ref page)</td>
<td>1148</td>
</tr>
<tr>
<td>Put_chars/[1,2] (built-in)</td>
<td>108</td>
</tr>
</tbody>
</table>
put_char/[1,2] (builtin, ref page) .................. 1149
put_code/[1,2] (builtin) .................................. 108
put_code/[1,2] (builtin, ref page) .............. 1150
put_contents/3 (structs) ............................... 763
put_label/[4,6] (trees) ................................ 868
puts (Tcl command) ...................................... 788

Q

qskip/1 (debugger mode control) ................. 284
quantifier, existential .................................. 188
quasi-skip (debugger command) ................... 239
query ...................................................... 21, 23, 62
query (definition) ....................................... 17
query, floundered (definition) ..................... 11
query_abbreviation/3 (hook, ref page) ....... 1151
query_abbreviation/3 (SU_messages) .......... 222
query_class/5 (hook, ref page) ...................... 1152
query_class/5 (SU_messages) ......................... 222
query_class_hook/5 (hook, ref page) .......... 1153
query_class_hook/5 (user, hook) ................. 222
query_hook/6 (hook, ref page) ..................... 1154
query_hook/6 (user, hook) ............................ 222
query_input/3 (hook, ref page) ...................... 1155
query_input/3 (SU_messages) ......................... 222
query_input_hook/3 (hook, ref page) ........... 1156
query_input_hook/3 (user, hook) ................. 222
query_map/4 (hook, ref page) ...................... 1157
query_map/4 (SU_messages) ............................ 222
query_map_hook/4 (hook, ref page) ............. 1158
query_map_hook/4 (user, hook) ..................... 222
queue_append/3 (queues) .............................. 746
queue_cons/3 (queues) ................................. 746
queue_head/2 (queues) ................................. 746
queue_last/[2,3] (queues) ......................... 747
queue_length/2 (queues) .............................. 747
queue_list/2 (queues) ................................. 747
queue_member/2 (queues) ............................. 747
queue_memberchk/2 (queues) .......................... 747
queue_tail/2 (queues) ................................. 746
quiet (syntax_errors flag value) ................... 142, 208
quote characters, in atoms ......................... 44
quoted/1 (write_term/[2,3] option) ............. 1249
quoted/1, write_term/[2,3] option ................. 105
quoted_charset (prolog flag) ....................... 138, 1250
quoted_charset/1 (write_term/[2,3] option) .... 1250

R

radiobutton (Tk widget) ................................. 794
raise exception (debugger command) .......... 242
raise/1 (debugger command control) .......... 284
raise_exception/1 (built-in) ...................... 210
raise_exception/1 (built-in, ref page) ....... 1159
random access to streams ......................... 115
random/[1,3] (random) ................................. 748
random_member/2 (random) ......................... 749
random_numlist/4 (random) ......................... 749
random_perm2/4 (random) ............................. 749
random_permutation/2 (random) ................. 749
random_select/3 (random) ........................... 749
random_subseq/3 (random) ......................... 749
random_ugraph/3 (ugraphs) ......................... 873
random_wgraph/4 (wgraphs) ......................... 877
random of floats .................................... 119
range of integers .................................... 111
range/2 (statistics) ................................. 758
range_to_fdset/2 (clpfd) ......................... 482
raw/1 (odbc_db_open/4 option) ..................... 708
rd/[1,2] (linda_client) ............................... 613
rd_noblock/1 (linda_client) ................. 613
reachable/3 (ugraphs) ............................... 873
reachable/3 (wgraphs) ............................... 877
read (absolute_file_name/3 access) ............. 929
read (open/[3,4] mode) ............................. 1115
read (Tcl command) ................................ 788
read predicates ...................................... 103
read/[1,2] (built-in) .................................. 103
read/[1,2] (built-in, ref page) ............. 1160
read_from_codes/2 (codesio) ...................... 524
read_line/[1,2] (built-in, ref page) .......... 1162
read_record/[1,2] (csv) ............................ 526
read_record_from_codes/[2,3] (csv) ............ 526
read_records/[1,2] (csv) ......................... 526
read_term/[2,3] (built-in) ......................... 103
read_term/[2,3] (built-in, ref page) .......... 1163
read_term_from_codes/3 (codesio) ............... 524
reading a goal from a string ...................... 311
reading in ............................................ 22
reclamation, space .................................. 146
reconsult .............................................. 285
reconsult/1 (built-in, ref page) ............... 1166
record/a (built-in) .................................. 138
record/a (built-in, ref page) ...................... 1167
recorded keys ........................................ 184
recorded/3 (built-in, ref page) ............... 1168
recordz/3 (built-in) ................................ 184
recordz/3 (built-in, ref page) .................... 1169
recursion (definition) .............................. 17
red cut ................................................. 65
redefine_warnings (prolog flag) ................. 81, 141, 319
redefining procedures, during execution 82
redo (CHR port) 419
redo (leashing mode) 234
redo/0 (debugger port value) 279
redo/1 (debugger command control) 284
reduce/2 (ugraphs) 873
reduce/2 (ugraphs) 877
reexit/1 (debugger command control) 284
reference page conventions 899
reference, term 289
regexp (Tcl command) 785
region (definition) 17
register_event_listener/[2,3]
  (prologbeans) 738
register_query/[2,3] (prologbeans) 737
regex (Tcl command) 785
regular/2 (clpfd) 468
refication 433
reject (redefine_warnings flag value) 141
reject (top-level command) 23
relation/3 (clpfd) 442
relative_to/1
  (absolute_file_name/3 option) 930
rem/2, integer truncated
  remainder (evaluable) 121
rem_add_link/4 (rem) 750
rem_create/2 (rem) 749
rem_equivalent/3 (rem) 749
rem_head/3 (random) 749
remove (CHR port) 419
remove this (debugger command) 241
remove_attribute_prefixes/1
  (xml_parse/3 option) 879
remove_breakpoints/1 (built-in) 261, 274
remove_breakpoints/1
  (built-in, ref page) 1170
remove_dups/2 (lists) 616
rename_directory/2 (file_systems) 557
rename_file/2 (file_systems) 556
repeat/0 (built-in, ref page) 1171
repeat/1 (between) 413
reposition/1 (open/4 option) 1115
reposition/1 (stream property) 1209
representation errors 205
representation_error/[1,3]
  (error class) 202
reset (open/4 eof_action value) 1116
reset printdepth (debugger command) 242
reset printdepth (top-level command) 23
reset subterm (debugger command) 243
reset subterm (top-level command) 24
resource errors 208
resource, colored 451
resource, cumulative 451
resource, data 1309
resource, dynamic 201
resource, foreign 201
resource, foreign (definition) 11
resource, linked foreign (definition) 14
resource, prelinked 292
resource, static 201
resource_error/[1,2] (error class) 203
rest of list, |` 46
restart (labeling/2 option) 474
restore (--main option value) 1309
restore/1 (built-in) 28, 93
restore/1 (built-in, ref page) 1173
restoring 28
restrictions, operator syntax 50
resumptions (fd_statistics/2 option) 477
retract/1 (built-in) 179
retract/1 (built-in, ref page) 1174
retractall/1 (built-in) 179
retractall/1 (built-in, ref page) 1176
retry (debugger command) 239
retry/1 (debugger command control) 284
return (Tcl command) 790
rev/2 (lists) 617
reverse/2 (lists) 617
rewriting, syntactic 99
rotate_list/[2,3] (lists) 619
round /1, (evaluable) 123
rule (definition) 17
rule, computation 76
rule, search 76
run-prolog (emacs command) 37
run_tests/[0,1,2] (plunit) 720
running 21
runtime (statistics key) 148
runtime (zinc option value) 890
runtime kernel (definition) 17
runtime system 5, 318
runtime system (definition) 17
runtime system, extended (definition) 11
runtime_entry/1 (user, hook) 1396
S

same Functor/[2,3,4] (terms) ............ 867
same_length/[2,3] (lists) .................. 617
samkeysort/2 (samsort) .................. 750
sample_standard_deviation/2
  (statistics) ............................. 759
sample_variance/2 (statistics) .......... 759
samsort/[2,3] (samsort) ................. 750
set/1 (clpb) ............................... 423
satisfy (labeling/2 option) .............. 473
save_files/2 (built-in) ................... 28, 94
save_files/2 (built-in, ref page) ..... 1177
save_modules/2 (built-in) ............... 28, 94
save_modules/2 (built-in, ref page) .. 1178
save_predicates/2 (built-in) ............. 28, 94
save_predicates/2 (built-in, ref page) .. 1179
save_program/[1,2] (built-in) .......... 28, 93
save_program/[1,2]
  (built-in, ref page) .................... 1180
saved state ................................ 28, 93
saved state (definition) ................. 17
saved states, initialization of .......... 93
saved_state
  (absolute_file_name/3 file type) .... 928
saving ..................................... 28
saving database ........................... 92
saving PO files ............................ 94
saving programs, predicates for ....... 95, 918
scalar_product/[4,5] (clpfd) ............ 434
scalar_product/4 (case/4 option) ..... 444
scalar_product_ref/[5,6] (clpfd) ....... 434
scale (Tk widget) ......................... 794
scan (Tcl command) ....................... 784
scanlist/[4,5,6] (lists) ................. 623
scrollbar/[4,5,6] (Tk widget) .......... 794
scope of variables ....................... 62
search (absolute_file_name/3 access) .. 929
search rule ................................ 76
search/1 (zinc option) .................. 886, 890
searchable
  (absolute_file_name/3 access) ....... 929
searching, for a file in a library ....... 95
seconds since the Epoch .................. 559
see/1 (built-in) ........................... 111
see/1 (built-in, ref page) ............... 1182
seeing/1 (built-in) ....................... 112
seeing/1 (built-in, ref page) .......... 1183
seek/4 (built-in) ......................... 115
seek/4 (built-in, ref page) .......... 1185
seen/0 (built-in) ......................... 114
seen/0 (built-in, ref page) ............. 1187
segment/2 (lists) ......................... 621
select/3 (lists) ........................... 614
select/4 (lists) .......................... 617
select_max/[3,4] (lists) ................. 628
select_min/[3,4] (lists) ................. 628
selectchk/3 (lists) ....................... 614
selectchk/4 (lists) ....................... 617
selector ................................... 534
selector, subterm ......................... 24, 250
selector, subterm (definition) .......... 19
self-modifying procedures ............... 176, 181
semantics ................................... 61
semantics (definition) .................... 17
semantics of dynamic code ............... 176
semantics of grammar rules ............. 192
semantics, declarative ................. 74
semantics, procedural ................. 74
sentence .................................... 61
sentence (definition) .................... 17
sentences ................. 52
seq_precede_chain/[1,2] (clpfd) ....... 470
sequence, escape .......................... 60
sequence, escape (definition) .......... 11
servlet ...................................... 733
session_gc_timeout/1 (start/1 option) .. 736
session_get/4 (prologbeans) .......... 737
session_put/3 (prologbeans) .......... 738
session_timeout/1 (start/1 option) .... 736
set (Tcl command) ......................... 778
set_printdepth (debugger command) ..... 242
set_printdepth (top-level command) .... 23
set_subterm (debugger command) ....... 243
set_subterm (top-level command) ....... 24
set, character ............................ 56
set, FD ........................................ 481
set/1 (plunit option) ..................... 718
set_input/1 (built-in) ................... 110
set_input/1 (built-in, ref page) ....... 1188
set_module/1 (built-in) ................. 166
set_module/1 (built-in, ref page) ..... 1189
set_order/3 (sets) ....................... 751
set_output/1 (built-in) ................. 110
set_output/1 (built-in, ref page) ..... 1190
set_prolog_flag/2 (built-in, ref page) .. 1191
set_stream_position/2 (built-in) ....... 115
set_stream_position/2
  (built-in, ref page) .................... 1192
seteq/2 (sets) ............................. 751
setof/3 (built-in) ......................... 187
setof/3 (built-in, ref page) ............ 1193
setproduct/3 (sets) ...................... 753
setrand/1 (random) ....................... 748
sets, collecting solutions to a goal .... 186
setting a breakpoint ..................... 37
setup/1 (plunit option) .................. 716
<table>
<thead>
<tr>
<th>Function/Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>start/[0,1] (prologbeans)</td>
<td>736</td>
</tr>
<tr>
<td>state, program</td>
<td>28</td>
</tr>
<tr>
<td>state, saved (definition)</td>
<td>17</td>
</tr>
<tr>
<td>state/2 (automaton/9 option)</td>
<td>463</td>
</tr>
<tr>
<td>static predicate (definition)</td>
<td>18</td>
</tr>
<tr>
<td>static procedures</td>
<td>177</td>
</tr>
<tr>
<td>static resource</td>
<td>291</td>
</tr>
<tr>
<td>statistics, displaying</td>
<td>146</td>
</tr>
<tr>
<td>statistics, memory</td>
<td>146</td>
</tr>
<tr>
<td>statistics/[0,2] (built-in)</td>
<td>146, 152</td>
</tr>
<tr>
<td>statistics/[0,2] (built-in, ref page)</td>
<td>1204</td>
</tr>
<tr>
<td>statistics/1 (zinc option)</td>
<td>886, 890</td>
</tr>
<tr>
<td>steadfast predicate (definition)</td>
<td>18</td>
</tr>
<tr>
<td>step (labeling/2 option)</td>
<td>472</td>
</tr>
<tr>
<td>store, consistent</td>
<td>429</td>
</tr>
<tr>
<td>store, contradictory</td>
<td>429</td>
</tr>
<tr>
<td>store_slot/2 (objects)</td>
<td>697</td>
</tr>
<tr>
<td>stream (definition)</td>
<td>19</td>
</tr>
<tr>
<td>stream (load context key)</td>
<td>143</td>
</tr>
<tr>
<td>stream alias</td>
<td>109</td>
</tr>
<tr>
<td>stream alias (definition)</td>
<td>19</td>
</tr>
<tr>
<td>stream errors</td>
<td>109</td>
</tr>
<tr>
<td>stream exceptions</td>
<td>109</td>
</tr>
<tr>
<td>stream object</td>
<td>109</td>
</tr>
<tr>
<td>stream object (definition)</td>
<td>19</td>
</tr>
<tr>
<td>stream position</td>
<td>114</td>
</tr>
<tr>
<td>stream position (definition)</td>
<td>19</td>
</tr>
<tr>
<td>stream position information for terminal I/O</td>
<td>115</td>
</tr>
<tr>
<td>stream position object</td>
<td>115</td>
</tr>
<tr>
<td>stream property</td>
<td>1208</td>
</tr>
<tr>
<td>stream property (definition)</td>
<td>19</td>
</tr>
<tr>
<td>stream, closing</td>
<td>114</td>
</tr>
<tr>
<td>stream, current</td>
<td>113</td>
</tr>
<tr>
<td>stream, end of</td>
<td>950</td>
</tr>
<tr>
<td>stream, position objects</td>
<td>1192, 1206</td>
</tr>
<tr>
<td>stream_code/2 (built-in)</td>
<td>109, 312</td>
</tr>
<tr>
<td>stream_code/2 (built-in, ref page)</td>
<td>1205</td>
</tr>
<tr>
<td>stream_object (argument type)</td>
<td>902</td>
</tr>
<tr>
<td>stream_position/2 (built-in)</td>
<td>115</td>
</tr>
<tr>
<td>stream_position/2 (built-in, ref page)</td>
<td>1206</td>
</tr>
<tr>
<td>stream_position_data/3 (built-in, ref page)</td>
<td>1207</td>
</tr>
<tr>
<td>stream_property/2 (built-in)</td>
<td>114</td>
</tr>
<tr>
<td>stream_property/2 (built-in, ref page)</td>
<td>1208</td>
</tr>
<tr>
<td>streams</td>
<td>109</td>
</tr>
<tr>
<td>streams, current input</td>
<td>110, 112, 113</td>
</tr>
<tr>
<td>streams, current input and output</td>
<td>102</td>
</tr>
<tr>
<td>streams, current output</td>
<td>110, 112, 113</td>
</tr>
<tr>
<td>streams, null</td>
<td>111</td>
</tr>
<tr>
<td>streams, opening</td>
<td>110</td>
</tr>
<tr>
<td>streams, random access to</td>
<td>115</td>
</tr>
<tr>
<td>streams, specifying</td>
<td>102</td>
</tr>
<tr>
<td>streams, standard</td>
<td>102</td>
</tr>
<tr>
<td>stretchmaxlen/2 (automaton/9 option)</td>
<td>462</td>
</tr>
<tr>
<td>stretchminlen/2 (automaton/9 option)</td>
<td>463</td>
</tr>
<tr>
<td>stretchocc/2 (automaton/9 option)</td>
<td>462</td>
</tr>
<tr>
<td>stretchoccmod/3 (automaton/9 option)</td>
<td>462</td>
</tr>
<tr>
<td>strict/1 (diffln/2 option)</td>
<td>453</td>
</tr>
<tr>
<td>string (definition)</td>
<td>19, 46</td>
</tr>
<tr>
<td>string first (Tcl command)</td>
<td>786</td>
</tr>
<tr>
<td>string index (Tcl command)</td>
<td>786</td>
</tr>
<tr>
<td>string last (Tcl command)</td>
<td>786</td>
</tr>
<tr>
<td>string length (Tcl command)</td>
<td>787</td>
</tr>
<tr>
<td>string match (Tcl command)</td>
<td>785</td>
</tr>
<tr>
<td>string range (Tcl command)</td>
<td>786</td>
</tr>
<tr>
<td>string string (Tcl command)</td>
<td>787</td>
</tr>
<tr>
<td>string tolower (Tcl command)</td>
<td>787</td>
</tr>
<tr>
<td>string toupper (Tcl command)</td>
<td>787</td>
</tr>
<tr>
<td>string trim (Tcl command)</td>
<td>787</td>
</tr>
<tr>
<td>string trimright (Tcl command)</td>
<td>787</td>
</tr>
<tr>
<td>structure, control</td>
<td>63</td>
</tr>
<tr>
<td>structure, control (definition)</td>
<td>9</td>
</tr>
<tr>
<td>SU_initialize() (C function)</td>
<td>1379, 1399</td>
</tr>
<tr>
<td>sub_atom/5 (built-in)</td>
<td>129</td>
</tr>
<tr>
<td>sub_atom/5 (built-in, ref page)</td>
<td>1211</td>
</tr>
<tr>
<td>sub_term/2 (terms)</td>
<td>866</td>
</tr>
<tr>
<td>subcircuit/[1,2] (clpfd)</td>
<td>449</td>
</tr>
<tr>
<td>subcircuit/1 (assignment/3 option)</td>
<td>440</td>
</tr>
<tr>
<td>sublist/[3,4,5] (lists)</td>
<td>620</td>
</tr>
<tr>
<td>subseq/3 (lists)</td>
<td>617</td>
</tr>
<tr>
<td>subseq0/2 (lists)</td>
<td>618</td>
</tr>
<tr>
<td>subseq1/2 (lists)</td>
<td>618</td>
</tr>
<tr>
<td>subset/2 (sets)</td>
<td>751</td>
</tr>
<tr>
<td>subsume/2 (terms)</td>
<td>862</td>
</tr>
<tr>
<td>subsume_term/2 (built-in)</td>
<td>127</td>
</tr>
<tr>
<td>subsume_term/2 (built-in, ref page)</td>
<td>1213</td>
</tr>
<tr>
<td>subsumeschk/2 (terms)</td>
<td>862</td>
</tr>
<tr>
<td>subterm selector</td>
<td>24, 250</td>
</tr>
<tr>
<td>subterm selector (definition)</td>
<td>19</td>
</tr>
<tr>
<td>subtract/3 (sets)</td>
<td>752</td>
</tr>
<tr>
<td>success (labeling/2 time_out/2 option value)</td>
<td>475</td>
</tr>
<tr>
<td>suffix/2 (lists)</td>
<td>620</td>
</tr>
<tr>
<td>suffix_length/3 (lists)</td>
<td>619</td>
</tr>
<tr>
<td>sum/3 (clpfd)</td>
<td>434</td>
</tr>
<tr>
<td>sumlist/2 (lists)</td>
<td>618</td>
</tr>
<tr>
<td>sup, plus infinity</td>
<td>429</td>
</tr>
<tr>
<td>sup/[2,4] (clpqr)</td>
<td>502</td>
</tr>
<tr>
<td>suppress (redefine warnings flag value)</td>
<td>141</td>
</tr>
<tr>
<td>switch (Tcl command)</td>
<td>783</td>
</tr>
<tr>
<td>symdiff/3 (sets)</td>
<td>753</td>
</tr>
<tr>
<td>symmetric_all_different/1 (clpfd)</td>
<td>440</td>
</tr>
<tr>
<td>symmetric_all_distinct/1 (clpfd)</td>
<td>440</td>
</tr>
</tbody>
</table>
### T

<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>table/[2,3] (clpfd)</td>
<td>443</td>
</tr>
<tr>
<td>tables, data</td>
<td>360</td>
</tr>
<tr>
<td>tail/2 (lists)</td>
<td>620</td>
</tr>
<tr>
<td>tan/1, (evaluable)</td>
<td>123</td>
</tr>
<tr>
<td>tanh/1, (evaluable)</td>
<td>124</td>
</tr>
<tr>
<td>task_intervals/1 (cumulatives/3 option)</td>
<td>451</td>
</tr>
<tr>
<td>task_intervals/1 (geost/[2,3,4] option)</td>
<td>455</td>
</tr>
<tr>
<td>taut/2 (clpb)</td>
<td>423</td>
</tr>
<tr>
<td>tcl_delete/1 (tcltk)</td>
<td>829,860</td>
</tr>
<tr>
<td>tcl_eval/3 (tcltk)</td>
<td>831,860</td>
</tr>
<tr>
<td>tcl_event/3 (tcltk)</td>
<td>834,860</td>
</tr>
<tr>
<td>tcl_new/1 (tcltk)</td>
<td>828,860</td>
</tr>
<tr>
<td>tell/1 (built-in)</td>
<td>111</td>
</tr>
<tr>
<td>tell/1 (built-in, ref page)</td>
<td>1215</td>
</tr>
<tr>
<td>term (argument type)</td>
<td>902</td>
</tr>
<tr>
<td>term (definition)</td>
<td>19</td>
</tr>
<tr>
<td>term comparison, predicates for</td>
<td>133,919</td>
</tr>
<tr>
<td>term conversions</td>
<td>301</td>
</tr>
<tr>
<td>term handling, predicates for</td>
<td>133,919</td>
</tr>
<tr>
<td>term I/O, predicates for</td>
<td>118,921</td>
</tr>
<tr>
<td>term input</td>
<td>103</td>
</tr>
<tr>
<td>term names</td>
<td>533, 536</td>
</tr>
<tr>
<td>term output</td>
<td>104</td>
</tr>
<tr>
<td>term reference</td>
<td>289</td>
</tr>
<tr>
<td>term, atomic (definition)</td>
<td>7</td>
</tr>
<tr>
<td>term, callable (definition)</td>
<td>8</td>
</tr>
<tr>
<td>term, compound (definition)</td>
<td>9</td>
</tr>
<tr>
<td>term, cyclic</td>
<td>1163,1250</td>
</tr>
<tr>
<td>term, error</td>
<td>197</td>
</tr>
<tr>
<td>term, exception</td>
<td>197,310</td>
</tr>
<tr>
<td>term, indexed</td>
<td>406,630</td>
</tr>
<tr>
<td>term, layout</td>
<td>1164</td>
</tr>
<tr>
<td>term, layout (definition)</td>
<td>13</td>
</tr>
<tr>
<td>term, mutable</td>
<td>132</td>
</tr>
<tr>
<td>term, mutable (definition)</td>
<td>15</td>
</tr>
<tr>
<td>term, simple (definition)</td>
<td>17</td>
</tr>
<tr>
<td>term/1 (input method)</td>
<td>221</td>
</tr>
<tr>
<td>term_depth/2 (terms)</td>
<td>866</td>
</tr>
<tr>
<td>term_expansion/6 (hook, ref page)</td>
<td>1219</td>
</tr>
<tr>
<td>term_expansion/6 (user, hook)</td>
<td>87</td>
</tr>
<tr>
<td>term_hash/[2,3,4] (terms)</td>
<td>862</td>
</tr>
<tr>
<td>term_order/3 (terms)</td>
<td>865</td>
</tr>
<tr>
<td>term_position (load context key)</td>
<td>143</td>
</tr>
<tr>
<td>term_size/2 (terms)</td>
<td>867</td>
</tr>
<tr>
<td>term_subsumers/3 (terms)</td>
<td>862</td>
</tr>
<tr>
<td>term_variables/2 (built-in)</td>
<td>127</td>
</tr>
<tr>
<td>term_variables/3 (aggregate)</td>
<td>388</td>
</tr>
<tr>
<td>term_variables_bag/2 (terms)</td>
<td>865</td>
</tr>
<tr>
<td>term_variables_set/2 (terms)</td>
<td>865</td>
</tr>
<tr>
<td>terminal I/O, stream position information for</td>
<td>115</td>
</tr>
<tr>
<td>terminating a backtracking loop</td>
<td>359</td>
</tr>
<tr>
<td>Terms in C, C functions for</td>
<td>1260</td>
</tr>
<tr>
<td>terms, arguments of</td>
<td>127</td>
</tr>
<tr>
<td>terms, as sentences</td>
<td>52</td>
</tr>
<tr>
<td>terms, comparison of</td>
<td>130</td>
</tr>
<tr>
<td>terms, compound</td>
<td>45</td>
</tr>
<tr>
<td>terms, cyclic</td>
<td>78</td>
</tr>
<tr>
<td>terms, input and output of</td>
<td>102</td>
</tr>
<tr>
<td>terms, ordering on</td>
<td>130</td>
</tr>
<tr>
<td>terms, predicates for looking at</td>
<td>126</td>
</tr>
<tr>
<td>terse (gc_trace flag value)</td>
<td>139</td>
</tr>
<tr>
<td>test condition, breakpoint</td>
<td>277</td>
</tr>
<tr>
<td>test, breakpoint</td>
<td>244</td>
</tr>
<tr>
<td>test/[1,2] (plunit declaration)</td>
<td>715</td>
</tr>
</tbody>
</table>