SICStus Prolog User's Manual

by the Intelligent Systems Laboratory

Swedish Institute of Computer Science
PO Box 1263
SE-164 29 Kista, Sweden

Release 4.0.2
November 2007

Swedish Institute of Computer Science
sicstus-request@sics.se http://www.sics.se/sicstus/

Table of Contents

Introduction

Prolog is a simple but powerful programming language developed at the University of Marseille [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 39.

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 *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 Standard ISO/IEC 13211-1 (PROLOG: Part 1—General Core) (`http://webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+13211%2D1%2D1995`).

Acknowledgments

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

> Jonas Almgren, Johan Andersson, Stefan Andersson, Nicolas Beldiceanu, Tamás Benkő, Kent Boortz, Dave Bowen, Per Brand, Göran Båge, Vicki Carleson, Mats Carlsson, Per Danielsson, Joakim Eriksson, Jesper Eskilson, Niklas Finne, Lena Flood, György Gyaraki, Dávid Hanák, Seif Haridi, Ralph Haygood, Christian Holzbaur, Tom Howland, Key Hyckenberg, Péter László, Per Mildner, Richard O'Keefe, Greger Ottosson, Dan Sahlin, Peter Schachte, Rob Scott, Thomas Sjöland, Péter Szeredi, Tamás Szeredi, Peter Van Roy, David Warren, Johan Widén, and Emil Åström.

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 on DECsystem-10 Prolog User's Manual by

> D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira, D.H.D. Warren

See Section 10.33 [lib-chr], page 482, for acknowledgments relevant to the CHR constraint solver.

See Section 10.35 [lib-clpqr], page 538, for acknowledgments relevant to the clp(Q,R) constraint solver.

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's 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 148).

+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) (`http://webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+13211%2D1%2D1995`).

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 40. 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 208.

breakpoint spec
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,. . .,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`.

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, ..., 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 coversion can be switched off by the `char_conversion` Prolog flag.

character-type mapping
A function mapping each element of the character code set to one of the character categories (layout, 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.

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, but 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, but 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 enuemrated in Section 4.2.3 [ref-sem-ctr], page 58.

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 199.

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, 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.

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 pre-linked 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 '`;`').

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 152. 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 54).

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.
```

le speci cation
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 80.

oundered 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 spec
A generalized predicate spec 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

Name/[Arity,. . . ,Arity]
the predicates of that name with one of the given arities, 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

```
Head :- Goals, !, 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.

extendible predicate
: An extendible predicate is a dynamic, multifile predicate, to which new clauses can be added by the user.

hook predicate
: A hook predicate is a predicate that somehow alters or customizes the behavior of a hookable predicate.

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

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 interpreted and compiled code, indexing is done on the principal functor of the first argument. Indexing is coarse wrt. large integers and floats.

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`***.

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, but more extensive facilities are available for debugging interpreted code.

invocation box
: Same as procedure box.

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, ***Layout*** is the number of the line where ***Source*** occurs.

  If ***Source*** is a compound term, ***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 list is written as a set of zero or more terms between square brackets. If there are no terms in a list, it is said to be empty, and is written as ‘`[]`’. In this first set of examples, all members of each list are explicitly stated:

```
[aa, bb,cc]   [X, Y]   [Name]   [[x, y], z]
```

  In the second set of examples, only the first several members of each list are explicitly stated, while the rest of the list is represented by a variable on the right-hand side of the “rest of” operator, ‘`|`’:

```
[X | Y]   [a, b, c | Y]   [[x, y] | Rest]
```

  ‘`|`’ is also known as the “list constructor.” The first element of the list to the left of ‘`|`’ is called the head of the list. The rest of the list, including the variable following ‘`|`’ (which represents a list of any length), is called the ***tail*** of the list.

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 A module is a set of predicates in a module-file. 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 148.

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

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

which must appear as the first term in the file.

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 114. 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.

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

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

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

```
| ?- 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,

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

can be written as

```
| ?- 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 47 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 197 for a definition.

pre-linked 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 43.

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 spec
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.

profiledcode
: Virtual code representation of compiled code, instrumented for profiling. A valid value for the `compiling` Prolog flag.

profiling
: The process of gathering execution statistics of parts of the program, essentially counting the 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 Prolog system prompt. For example,

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

refers to the predicate `father/2`. If a query has no variables in it, the system will respond either '`yes`' or '`no`'. If a query contains variables, 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, 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.

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.

SP_term_ref
A "handle" object providing an interface from C to Prolog terms.

spypoint A special case of breakpoint, the ***debugger breakpoint***, intended for interactive debugging. Its simplest form, the ***plain spypoint*** instructs the debugger to stop at all ports of all invocations of a specified predicate. ***Conditional spypoints*** 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 spypoint*** is like a conditional spypoint, but not restricted to a single predicate. See Section 5.6 [Advanced Debugging], page 208.

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 87.

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`.

stream object
: A term denoting an open Prolog stream. See Section 4.6 [ref-iou], page 87.

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 98.

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.

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.

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

uni cation 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 40.

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 is normally 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 instead. A GNU Emacs interface for SICStus Prolog is described later (see Section 3.11 [Emacs Interface], page 29). From a UNIX shell, SICStus Prolog is started by invoking the `sicstus` command-line tool.

Under UNIX, a saved-state *le* can be executed directly by typing:

```
% file argument...
```

This is equivalent to:

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

> **Please note**: As of release 3.7, saved-states do not store the complete path of the binary `sp.exe`. Instead, they call the main executable `sicstus`, which is assumed to be found in the shell's path. If there are several versions of SICStus installed, 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 4.0.2 ...
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 56).

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

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

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

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

| ?- 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 80 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:

```
| ?- 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:

```
| ?- 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:

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

(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:

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

Remember that Prolog terms must terminate with a full stop ('`.`', possibly followed by layout text), 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 there are no variables as in this example, the system answers

```
yes
```

and execution of the query terminates.

If variables are included in the query, the final value of each variable is displayed (except for variables whose names begin with '`_`'). Thus the query

```
| ?- member(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`'.

;
n "rejects" the solution; the development system backtracks (see Section 4.2 [ref-sem], page 56) looking for alternative solutions. If no further solutions can be found it outputs '`no`'.

b invokes a recursive top-level.

`<` In the top-level, a global *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 *n*, the printdepth is set to *n*, treating 0 as infinity. This command works by changing the value of the `toplevel_print_options` Prolog flag.

`^` A local *subterm selector*, initially `[]`, is maintained. The subterm selector provides a way of zooming in to some subterm of each binding. For example, the subterm selector `[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 `[]`. With an argument of 0, the last element of the subterm selector is removed. With an argument of *n* (> 0), *n* is added to the end of the subterm selector. With a list of arguments, the arguments are applied from left to right.

h
? lists available commands.

While the variable bindings are displayed, all variables occurring in the values are replaced by terms of the form `'$VAR'(N)` to yield friendlier variable names. Such names come out as a sequence of letters and digits preceded by '`_`'. The outcome of some queries is shown below.

```
| ?- member(X, [tom,dick,harry]).

X = tom ;
X = dick ;
X = harry ;

no
| ?- member(X, [a,b,f(Y,c)]), member(X, [f(b,Z),d]).

X = f(b,c),
Y = b,
Z = c

| ?- member(X, [f(_),g]).

X = f(_A)
```

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.

3.4.2 Directives

Directives start with the symbol '`:-`'. Any required output must be programmed explicitly; e.g. the directive:

```
:- member(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, 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 don't want terminal interaction in the middle of consulting the file. A useful example would be the use of a directive in a file that consults a whole list of other files, e.g.:

```
:- consult([ bits, bobs, main, tests, data, junk ]).
```

If a directive like this were contained in the file 'myprog', 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, 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. 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.:

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

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

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 you can use the listing/1 predicate to list all the clauses that were successfully read in, e.g.:

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

Please note: The built-in predicates `read/[1,2]` normally raise an exception on syntax errors (see Section 4.15 [ref-ere], page 173). 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 71) 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 173) is called. If undefined or if the call fails the action is governed by the state of the `unknown` Prolog flag, which can be:

`trace`
: which causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity.

`error`
: which causes calls to such predicates to raise an exception (the default state). See Section 4.15 [ref-ere], page 173.

`warning`
: which causes calls to such predicates to display a warning message and then fail.

`fail`
: which causes calls to such predicates to fail.

Calls to predicates that have no clauses are not caught.

The built-in predicate `unknown(`***OldState***`, `***NewState***`)` unifies *OldState* with the current state and sets the state to *NewState*. The built-in predicate `debugging/0` prints the value of this state along with its other information. This state is also controlled by the `unknown` Prolog flag.

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.

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 ***^C***. This ***^C*** interruption has the effect of suspending the execution, and the following message is displayed:

```
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:

a aborts the current computation.

c continues the execution.

e exits from SICStus Prolog, closing all files.

h
? lists available commands.

b invokes a recursive top-level.

d
z
t switch on the debugger. See Chapter 5 [Debug Intro], page 197.

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

3.8 Exiting From The Top-Level

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.

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):

```
| ?- break.
```

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

```
% Break level 1
```

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

If another call of `break/0` is encountered, it moves up to level 2, and so on. To close the break and resume the execution that was suspended, type **^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, 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.

> **Please note**: When creating a saved state with `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 260.

As of SICStus Prolog 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 prefix '`.po`' (for Prolog object), which can be loaded by `load_files/[1,2]`. 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 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.

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 Mac OS X. For information specific to GNU Emacs or XEmacs, see `http://www.gnu.org` and `http://www.xemacs.org` respectively. For information on running Emacs under Windows, see the '`GNU Emacs FAQ For Windows 98/ME/NT/XP and 2000`' at `http://www.gnu.org/software/emacs/windows/ntemacs.html`, much of which applies to both GNU Emacs and XEmacs.

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, and Per Danielsson, 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. More ideas and code have been taken from the SICStus debugger mode by Per Mildner.

3.11.1 Installation

See section "The Emacs Interface" in *SICStus Prolog Release Notes*, for more information about installing the Emacs interface.

There are some differences between GNU Emacs and XEmacs. This will be indicated with Emacs-Lisp comments in the examples.

3.11.1.1 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.

Note to Windows users: '`~/.emacs`' denotes a file '`.emacs`' in whatever Emacs considers to be your *home* directory. See '`GNU Emacs FAQ For Windows 98/ME/NT/XP and 2000`' at `http://www.gnu.org/software/emacs/windows/ntemacs.html`, for details.

Under UNIX, assuming SICStus 4.0.2 was installed in '/usr/local/', add the following line:

```
(load "/usr/local/lib/sicstus-4.0.2/emacs/sicstus_emacs_init")
```

Under Windows, assuming SICStus 4.0.2 was installer in 'C:\Program Files\SICStus Prolog 4.0.2\', add the following line:

```
(load "C:/Program Files/SICStus Prolog
4.0.2/emacs/sicstus_emacs_init")
```

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

3.11.1.2 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.

3.11.1.3 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 'sictus_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:

```
(setq load-path
   (cons (expand-file-name "/usr/local/lib/sicstus-4.0.2/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 4.0.2/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, 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 3.11.4 [Mode Line], page 34.

If the shell command `sicstus` is not available in the default path, 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/bin/sicstus
```

3.11.1.4 Enabling Emacs Support for SICStus Documentation

If you follow the steps in Section Quick Start, above, 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-4.0.2/doc/info/`' on UNIX platforms and '`C:/Program Files/SICStus Prolog 4.0.2/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 4.0.2/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 new enough. You may have to quit and restart Emacs for these changes to take effect.

3.11.2 Basic Configuration

If the following lines are not present in '`~/.emacs`', 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, 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)
```

3.11.3 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⟩.

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 level 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, the debugger will display the source code location of the current goal when it prompts for a debugger command, by overlaying the beginning of the current line of code with an arrow. If `source_info` was `off` when the code was loaded, or if it was asserted or loaded from `user`, the current goal will still be shown but out of context.

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

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-n Go to the next clause.

M-p Go to the previous clause.

M-a Go to beginning of clause.

M-e Go to end of clause.

M-C-c Mark clause.

M-C-a Go to beginning of predicate.

M-C-e Go to end of predicate.

M-C-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.

M-C-n Go to matching right parenthesis.

M-C-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-x SPC
C-u C-x SPC
Set and remove a line breakpoint. This uses the advanced debugger features introduced in SICStus 3.8; see Section 5.6 [Advanced Debugging], page 208.

C-c C-s Insert the *PredSpec* of the current predicate into the code.

C-c C-n Insert the name of the current predicate into the code. This can be useful when writing recursive predicates or predicates with several clauses. See also the `prolog-electric-dot-flag` variable below.

C-c C-v a Convert all variables in a region to anonymous variables. This can also be done using the `Prolog/Transform/All variables to '_'` menu entry. See also the `prolog-electric-underscore-flag` Emacs variable.

C-c ? Help on predicate. This requires the SICStus info files to be installed. If the SICStus info files are installed in a nonstandard way, you may have to change the Emacs variable `prolog-info-predicate-index`.

3.11.4 Mode Line

If working with an application split into several modules, 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 *ModuleName*. If the mode line is missing, 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 `auto-mode-alist`. A mode line can be inserted by choosing `Insert/Module modeline` in the `Prolog` menu.

3.11.5 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 ***M-x set-variable*** ⟨RET⟩ ***VariableName*** ⟨RET⟩ ***Value*** ⟨RET⟩. Note that variable names can be completed by typing a few characters and then pressing ⟨TAB⟩.

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

```
(setq VariableName Value)
```

Note that the Emacs interface is presently not using the '`Customize`' functionality to edit the settings.

The available settings are:

`prolog-system`
: The Prolog system to use. Defaults to `'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 `prolog-system` the variables below named `sicstus-`***something*** will not be used, in some cases corresponding functionality is available through variables named `prolog-`***something***.

`sicstus-version`
: The version of SICStus that is used. Defaults to `'(3 . 8)`. Note that the spaces are significant!

`prolog-use-sicstus-sd`
: Set to `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 `nil`. Note that the source-linked debugging only works if `sicstus-version` is set correctly.

`pltrace-port-arrow-assoc` **obsolescent**

Only relevant for source-linked debugging, this controls how the various ports of invocation boxes (see Section 5.1 [Procedure Box], page 197) map to arrows that point into the current line of code in source code buffers. Initialized as:

```
'(("call" . ">>>") ("exit" . "+++") ("ndexit" . "?++")
  ("redo" . "<<<") ("fail" . "---") ("exception" . "==>"))
```

where `ndexit` is the nondeterminate variant of the Exit port. Do not rely on this variable. It will change in future releases.

`prolog-indent-width`

How many positions to indent the body of a clause. Defaults to `tab-width`, normally 8.

`prolog-paren-indent`

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

```
p :-
        (   q1
        ;   q2,
            q3
        ).
```

Note that the spaces between the parentheses and the code are automatically inserted when ⟨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 `t`.

`prolog-object-end-to-0-flag`

Set to nil to indent the closing `}` of an object definition to `prolog-indent-width`. 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

```
'((sicstus
     ("block" "discontiguous" "dynamic" "initialization"
      "meta_predicate" "mode" "module" "multifile" "public"
      "volatile")))
```

`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 white space 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, 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 anynomous variable. Defaults to `nil`.

`prolog-old-sicstus-keys-flag`
: Set to `t` to enable the key-bindings of the old Emacs interface. These bindings are not used by default since they violate GNU Emacs recommendations. 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`.

3.11.6 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.

3.11.6.1 Font-locking

When editing large files, it might happen that font-locking is not done because the file is too large. Typing ***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, choose `Fontify Buffer` from the `Prolog` menu.

3.11.6.2 Auto-fill Mode

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

```
dynamics([]).        % A list of pit furnace
                     % dynamic instances
```

3.11.6.3 Speed

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

First of all, make sure that '`prolog.el`' and '`sicstus-support.el`' are compiled, i.e. that there is a '`prolog.elc`' and a '`sicstus-support.elc`' file at the same location as the original files. To do the compilation, start Emacs and type ***M-x byte-compile-file*** ⟨RET⟩ ***path*** ⟨RET⟩, where *path* is the path to the '`*.el`' file. Do not be alarmed if there are a few warning messages as this is normal. If all went well, 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 `prolog-use-prolog-tokenizer-flag` to `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 `prolog-parse-mode` variable also affects the speed, `'beg-of-line` being faster than `'beg-of-clause`.

Font locking may be slow. You can turn it off using customization, available through ***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 3.11.2 [Basic Configuration], page 31.

3.11.6.4 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, 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, 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:

```
;; 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 700.

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 ***term***s. 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 40, and Section 4.1.3 [ref-syn-cpt], page 41, respectively.

Foreign data types are discussed in the context of `library(structs)`; see Section 10.22 [lib-structs], page 455.

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   0b1111   0o17  0xf
```

Note that

```
+525
```

is not a valid integer.

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

```
0'A   0'\x41\   0'\101\
```

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 ‘`\`’, the ‘`\`’ denotes the start of an escape sequence with special meaning (see Section 4.1.7.6 [ref-syn-syn-esc], page 54).

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 of up to 65535 characters (other than the null character). An atom 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 an unquoted atom are restricted to a subset of Unicode, see Section 4.1.7.5 [ref-syn-syn-tok], page 51.

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 54), and if the single quote character is included in the sequence it must be escaped, e.g. `'can\'t'`.

The characters that are allowed to occur in an quoted atom are restricted to a subset of Unicode, see Section 4.1.7.5 [ref-syn-syn-tok], page 51.

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 42) 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'   []
```

WARNING: 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, 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 runtime.

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

```
point(X, Y, Z)
```

When we need to refer explicitly to a functor we will normally denote it by the form ***Name***/***Arity***. Thus, the functor '`point`' of arity 3 is denoted

```
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

```
s(np(john), vp(v(likes), np(mary)))
```

would be pictured as the following tree:

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

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

Sometimes it is convenient to write certain functors as ***operators***; binary functors (that is, functors of two arguments) may be declared as ***in x*** operators, and unary functors (that is, functors of one argument) may be declared as either ***pre x*** or ***post x*** operators. Thus it is possible to write

```
X+Y     P;Q     X<Y      +X     P;
```

as optional alternatives to

```
+(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)
```

The use of operators is described fully in Section 4.1.5 [ref-syn-ops], page 43.

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,[])))                                        (A)

[1,2,3]                                                  (B)

[X|L]      [a,b|L]                                       (C)

      .                .
     / \              / \
    X   L            a   .
                        / \
                       b   L                             (D)
```

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) .(a,.(b,L))                                      (E)
```

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

```
[83,73,67,83,116,117,115]
```

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`, 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`, 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 54). As for quoted atoms, if a double quote character is included in the sequence it must be escaped, e.g. `"can\"t"`.

The built-in predicates that print terms (see Section 4.6.4 [ref-iou-tou], page 89) 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 51.

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 layout characters must be expressed with escape sequences (see Section 4.1.7.6 [ref-syn-syn-esc], page 54).

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

```
        +
      /   \
     2     1
```

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 `is/2`; see Section 4.7.2 [ref-ari-eae], page 103.)

Prolog syntax allows operators of three kinds: ***infix***, ***prefix***, and ***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 ***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 *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 `+(a,/(b,c))`. Note that the infix form of the term `/(+(a,b),c)` must be written with explicit parentheses:

```
(a+b)/c
```

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

`xfx`

`xfy`

`yfx`

Operators of type '`xfx`' are not associative: it is required that both of the arguments of the operator be subexpressions of *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 '`xfy`' are right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the *same* precedence as the main operator. Left-associative operators (type '`yfx`') are the other way around.

An atom named ***Name*** is declared as an operator of type ***Type*** and precedence ***Precedence*** by the command

```
:-op(Precedence, Type, Name).
```

An operator declaration can be cancelled by redeclaring the ***Name*** with the same ***Type***, but ***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 47.

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).

```
:-op(500, yfx, [+,-]).                                    (A)

a-b+c                                                     (B)

(a-b)+c                                                   (C)

               +
             /   \
            -     c
           / \
          a   b                                           (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).

```
:-op(600, xfy, .).                                        (E)

a.b.c                                                     (F)

             .
            / \
           a   .
              / \
             b   c                                        (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 `not` were declared as a prefix operator of type `fy`, then

```
not not P
```

would be a permissible way to write `not(not(P))`. If the type were `fx`, the preceding expression would not be legal, although

```
not P
```

would still be a permissible form for `not(P)`.

If these precedence and associativity rules seem rather complex, 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 `op(`***Precedence***`, `***Type***`, `***Name***`)`. `op/3` declares the atom *Name* to be an operator of the stated *Type* and *Precedence*. If *Precedence* is 0, the operator properties of *Name* (if any) are cancelled.

To examine the set of operators currently in force, use `current_op(`***Precedence***`, `***Type***`, `***Name***`)`.

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

   ```
   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 '(', this '(' must be separated from the operator by at least one space or other layout 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

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic, volatile, discontiguous,
                    multifile, block, meta_predicate,
                    initialization ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, \=, is, =.., ==, \==, @<, @>, @=<, @>=,
                            =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, \, /\, \/ ]).
:- op(  400, yfx, [ *, /, //, mod, rem, <<, >> ]).
:- op(  200, xfx, [ ** ]).
:- op(  200, xfy, [ ^ ]).
:- op(  200,  fy, [ +, -, \ ]).
```

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 49, 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 prefix 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 50. 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 layout 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 *comment*s 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 on Section 4.1.7.5 [ref-syn-syn-tok], page 51. 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 165.

In Section 4.1.7.4 [ref-syn-syn-trm], page 50, 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

```
sentence       ::= module : sentence
                 | list                         { 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
                 | goal                         { where goal is not a variable }
body           ::= module : body
                 | body -> body ; body
                 | body -> body
                 | \+ body
                 | body ; body
                 | body , body
                 | goal
goal           ::= term                         { where term is not otherwise a
                                                  body }
grammar-rule   ::= gr-head --> gr-body
gr-head        ::= module : gr-head
                 | gr-head , terminals
                 | non-terminal                 { where non-terminal is not a vari-
                                                  able }
gr-body        ::= module : gr-body
                 | gr-body -> gr-body ; gr-body
                 | gr-body -> gr-body
                 | \+ gr-body
```

<table>
<tr><td></td><td>| gr-body ; gr-body</td><td></td></tr>
<tr><td></td><td>| gr-body , gr-body</td><td></td></tr>
<tr><td></td><td>| non-terminal</td><td></td></tr>
<tr><td></td><td>| terminals</td><td></td></tr>
<tr><td></td><td>| gr-condition</td><td></td></tr>
<tr><td>non-terminal</td><td>::= term</td><td>{ where term is not otherwise a gr-body }</td></tr>
<tr><td>terminals</td><td>::= list | string</td><td></td></tr>
<tr><td>gr-condition</td><td>::= ! | {body}</td><td></td></tr>
<tr><td>module</td><td>::= atom</td><td></td></tr>
</table>

4.1.7.4 Syntax of Terms as Tokens

<table>
<tr><td>term-read-in</td><td>::= subterm(1200) full-stop</td><td></td></tr>
<tr><td>subterm(N)</td><td>::= term(M)</td><td>{ where M is less than or equal to N }</td></tr>
<tr><td>term(N)</td><td>::= op(N,fx) subterm(N-1)</td><td>{ except in the case of a number if subterm starts with a ‘(’, op must be followed by layout-text }</td></tr>
<tr><td></td><td>| op(N,fy) subterm(N)</td><td>{ if subterm starts with a ‘(’, op must be followed by layout-text }</td></tr>
<tr><td></td><td>| subterm(N-1) op(N,xfx) subterm(N-1)</td><td></td></tr>
<tr><td></td><td>| subterm(N-1) op(N,xfy) subterm(N)</td><td></td></tr>
<tr><td></td><td>| subterm(N) op(N,yfx) subterm(N-1)</td><td></td></tr>
<tr><td></td><td>| subterm(N-1) op(N,xf)</td><td></td></tr>
<tr><td></td><td>| subterm(N) op(N,yf)</td><td></td></tr>
<tr><td>term(1000)</td><td>::= subterm(999) , subterm(1000)</td><td></td></tr>
<tr><td>term(0)</td><td>::= functor (arguments)</td><td>{ provided there is no layout-text between the functor and the ‘(’ }</td></tr>
<tr><td></td><td>| (subterm(1200))</td><td></td></tr>
<tr><td></td><td>| { subterm(1200) }</td><td></td></tr>
<tr><td></td><td>| list</td><td></td></tr>
<tr><td></td><td>| string</td><td></td></tr>
<tr><td></td><td>| constant</td><td></td></tr>
<tr><td></td><td>| variable</td><td></td></tr>
<tr><td>op(N,T)</td><td>::= name</td><td>{ where name has been declared as an operator of type T and precedence N }</td></tr>
<tr><td>arguments</td><td>::= subterm(999)</td><td></td></tr>
<tr><td></td><td>| subterm(999) , arguments</td><td></td></tr>
<tr><td>list</td><td>::= []</td><td></td></tr>
<tr><td></td><td>| [listexpr]</td><td></td></tr>
<tr><td>listexpr</td><td>::= subterm(999)</td><td></td></tr>
<tr><td></td><td>| subterm(999) , listexpr</td><td></td></tr>
</table>

	| *subterm(999)* | *subterm(999)*
constant	::= *atom* | *number*
number	::= *unsigned-number*
	| *sign unsigned-number*
	| *sign* `inf`
	| *sign* `nan`
unsigned-number	::= *natural-number* | *unsigned-oat*
atom	::= *name*
functor	::= *name*

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]. 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 layout-char category except that space (character code 32) is allowed despite it being a layout-char.

An additional restriction is that the sequence of characters that makes up a quoted token must be in Normal Form C (NFC) `http://www.unicode.org/reports/tr15/`. This is currently (SICStus Prolog 4.0.1) not enforced. A future version of SICStus Prolog may enforce this restriction or perform this normalization automatically.

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

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]`.

layout-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).

[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 54).

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.

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[2].

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, 39, and 96 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.

token	::= *name*	
	| *natural-number*	
	| *unsigned-float*	
	| *variable*	
	| *string*	
	| *punctuation-char*	
	| *layout-text*	
	| *full-stop*	
name	::= *quoted-name*	
	| *word*	
	| *symbol*	
	| *solo-char*	
	| [*?layout-text*]	
	| { *?layout-text* }	
word	::= *small-letter ?alpha...*	
symbol	::= *symbol-char...*	{ except in the case of a *full-stop* or where the first 2 chars are ‘`/*`’ }

---

[2] In SICStus Prolog 4.0.0 and in SICStus 3 the lower case characters 170 and 186 were incorrectly classified as symbol-char. This was corrected in SICStus Prolog 4.0.1.

<table>
<tr><td>natural-number</td><td>::= digit...</td><td></td></tr>
<tr><td></td><td>| base-prefix alpha...</td><td>{ where each alpha must be digits of the base indicated by base-prefix, treating a,b,... and A,B,... as 10,11,... }</td></tr>
<tr><td></td><td>| 0 ' char-item</td><td>{ yielding the character code for char }</td></tr>
<tr><td>unsigned-float</td><td>::= simple-float</td><td></td></tr>
<tr><td></td><td>| simple-float exp exponent</td><td></td></tr>
<tr><td>simple-float</td><td>::= digit... . digit...</td><td></td></tr>
<tr><td>exp</td><td>::= e | E</td><td></td></tr>
<tr><td>exponent</td><td>::= digit... | sign digit...</td><td></td></tr>
<tr><td>sign</td><td>::= - | +</td><td></td></tr>
<tr><td>variable</td><td>::= underline ?alpha...</td><td></td></tr>
<tr><td></td><td>| capital-letter ?alpha...</td><td></td></tr>
<tr><td>string</td><td>::= " ?string-item... "</td><td></td></tr>
<tr><td>string-item</td><td>::= quoted-char</td><td>{ other than '"' or '\' }</td></tr>
<tr><td></td><td>| ""</td><td></td></tr>
<tr><td></td><td>| \ escape-sequence</td><td></td></tr>
<tr><td>quoted-atom</td><td>::= ' ?quoted-item... '</td><td></td></tr>
<tr><td>quoted-item</td><td>::= quoted-char</td><td>{ other than ''' or '\' }</td></tr>
<tr><td></td><td>| ''</td><td></td></tr>
<tr><td></td><td>| \ escape-sequence</td><td></td></tr>
<tr><td>backquoted-atom</td><td>::= ` <em>?backquoted-item</em>... `</td><td></td></tr>
<tr><td>backquoted-item</td><td>::= quoted-char</td><td>{ other than '`' or '\' }</td></tr>
<tr><td></td><td>| ``</td><td></td></tr>
<tr><td></td><td>| \ escape-sequence</td><td></td></tr>
<tr><td>layout-text</td><td>::= layout-text-item...</td><td></td></tr>
<tr><td>layout-text-item</td><td>::= layout-char | comment</td><td></td></tr>
<tr><td>comment</td><td>::= /* ?char... */</td><td>{ where ?char... must not contain '*/' }</td></tr>
<tr><td></td><td>| % ?char... ⟨LFD⟩</td><td>{ where ?char... must not contain ⟨LFD⟩ }</td></tr>
<tr><td>full-stop</td><td>::= .</td><td>{ the following token, if any, must be layout-text}</td></tr>
<tr><td>char</td><td>::= layout-char</td><td></td></tr>
<tr><td></td><td>| printing-char</td><td></td></tr>
<tr><td>printing-char</td><td>::= alpha</td><td></td></tr>
<tr><td></td><td>| symbol-char</td><td></td></tr>
<tr><td></td><td>| solo-char</td><td></td></tr>
<tr><td></td><td>| punctuation-char</td><td></td></tr>
<tr><td></td><td>| quote-char</td><td></td></tr>
<tr><td>alpha</td><td>::= capital-letter | small-letter | digit | underline</td><td></td></tr>
<tr><td>escape-sequence</td><td>::= b</td><td>{ backspace, character code 8 }</td></tr>
</table>

	| `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*	
	| *backquoted-atom*	
base-pre x	::= `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 }
	| `o` *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)

`\x`***hex-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

   X`,`***Y***

   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 and for the character '`` ` ``' in a backquoted 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 layout 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:

`current_op(`***P***`,`***T***`,`***A***`)` ***ISO***
: atom A is an operator of type T with precedence P

`op(`***P***`,`***T***`,`***A***`)` ***ISO***
: make atom A an operator of type T 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, the sentence is called a ***clause***.

If the body of a clause is empty, 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).

P. (A)

"*P* is true." (B)

"Goal *P* is satisfied." (C)

If the body of a clause is non-empty, 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*. (D)

"*P* is true if *Q* and *R* and *S* are true." (E)

"To satisfy goal *P*, satisfy goals *Q*, *R*, and *S*." (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***. (G)

"Are ***P*** and ***Q*** true?" (H)

"Satisfy goals ***P*** and ***Q***." (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:

`employed(X) :- 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."

`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".

```
concatenate([], L, L).                                  (J)
concatenate([X|L1], L2, [X|L3]) :-
            concatenate(L1, L2, L3).                    (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***.

Certain ones, however, can be modified or totally redefined. These are the hook predicates and the extendible predicates used in message 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. They 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 71). 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, 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 only their first solution is relevant.

4.2.2.2 Extendible Predicates

Extendible predicates exist to enable you to extend or modify SICStus Prolog's message and query handling. These predicates are all defined in the file `library('SU_messages')`.

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 165), and when code is loaded or asserted in the context of modules (see Section 4.11 [ref-mod], page 138). 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, 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*** ***ISO***
: goal *P* is not provable

?X `^` ***:P***
: there exists an *X* such that *P* is provable (used in `setof/3` and `bagof/3`)

`if(`***:P***`,`***:Q***`,`***:R***`)`
: for each solution of *P* succeeds, prove *Q*; if none, prove *R*

`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, 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

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

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 doesn't 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 305.

- 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, 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, it is usually for this reason.

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

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

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

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- X < Y.
```

Furthermore, if the `X >= Y` test fails 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:

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

Unfortunately, this version of `max/3` can give wrong answers, for example:

```
| ?- max(10, 0, 0).
yes
```

The reason is that the query doesn't match the head of the first clause, and so we never executed the `X >= 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 ***not steadfast***. A steadfast version is:

```
max(X, Y, Z) :- X >= Y, !, Z = X.
max(X, Y, Y).
```

4.2.3.2 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).

```
grandfather(X, Z) :-
     (   mother(X, Y)
     |   father(X, Y)
     ),
     father(Y, Z).                                          (A)
```

"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." (B)

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

```
grandfather(X, Z) :- parent(X, Y), father(Y, Z).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).                               (C)
```

For historical reasons, the token '`|`', when used outside a list, is actually an alias for '`;`'. The aliasing is performed when terms are read in, so that (D) is read as if it were (E) thus you can use '`;`' instead of '`|`' for disjunction if you like.

```
a :- b | c.                                                 (D)
```

```
a :- b ; c.                                                 (E)
```

4.2.3.3 If-Then-Else

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

```
(If -> Then ; Else)
```

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

Cuts inside of *If* do not make much sense and are not recommended. If you do use them, their scope is limited to *If* itself.

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

```
(   If_1 -> Then_1
;   If_2 -> Then_2
    ...
;   /* otherwise -> */
    WhenAllElseFails
)
```

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

```
(If -> Then)
```

```
(If -> Then ; fail)
```

That is, the '`->`' operator has nothing to do with, and should not be confused with, logical implication.

`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 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 `setof/3` and `bagof/3` constructs is superfluous and discouraged. Thus, the following two goals are equivalent:

X^P

P

The following construct is meaningful in the context of modules (see Section 4.11 [ref-mod], page 138), meaning "*P* is true in the context of the *M* module":

M: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 ***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 ***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 ***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 `concatenate/3`, declared in Section 4.2.1 [ref-sem-pro], page 56, 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 `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 `concatenate/3`.

```
concatenate([a], [b], [a,b])

concatenate([a], [b], [a,b]):-
   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 definite 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 enumerate, 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 ***uni es*** 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, 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, 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 reconsiders 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) 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).

```
| ?- concatenate(X, Y, [a,b]).                                  (A)

concatenate(X1, Y, [b])                                         (B)

concatenate(X2, Y, [])                                          (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:

```
concatenate([a,b], [], [a,b])
```

If this solution is rejected, 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 `concatenate/3`, against the second clause.

Thus, in the procedural semantics, the set of clauses

```
H :- B1, ..., Bm.
H' :- B1', ..., Bm'.
...
```

are regarded as a ***procedure definition*** for some predicate *H*, and in a query

```
?- G1, ..., Gn.
```

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

```
?- (G1, ..., Gj-1, B1, ..., Bm, Gj+1, ..., Gn)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, the execution backtracks to the most recent successful match in an attempt to find an alternative match. If such a match is found, 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 ore 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

```
?- G1, ..., Gj-1, Gj, Gj+1, ..., Gn.
```

where *Gj* is the first unblocked goal, and matching *Gj* against a clause head causes several blocked goals in *G1*, ..., *Gj-1* to become unblocked, these goals may become reordered. The internal order of any two goals that were blocked on the *same* variable is retained, however.

Another consequence is that a query may be derived consisting entirely of blocked goals. Such a query is said to have ***oundered***. The top-level checks for this condition. If detected, 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 73). 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.

4.2.5 Meta-Calls

If *X* is instantiated to a term that would be acceptable as the body of a clause, the goal `call(X)` is executed exactly as if that term appeared textually in place of the `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 `call(X)` occurs. If *X* is not properly instantiated, an error exception is raised as described in Section 4.2.6 [ref-sem-exc], page 66.

In SICStus 4, `call/1` has been generalized to `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 `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 with the occurs-check is at best linear on the sum of the sizes of the terms being unified, whereas unification without the occurs-check is linear on the size of the smallest of the terms being unified. 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 113), 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 109. Predicates for subsumption and testing (a)cyclicity are available in a library package; see Section 10.24 [lib-terms], page 465. Other predicates usually do not handle cyclic terms well.

4.2.8 Summary of Control Predicates

: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*** — ***ISO***
: goal *P* is not provable

?X* ^ *:P there exists an *X* such that *P* is provable (used in `setof/3` and `bagof/3`)

`block` ***:P*** — ***declaration***
: declaration that predicates specified by *P* should block until sufficiently instantiated

`call(`***:P***`)` — ***ISO***
`call(`***:P***`,...)`
: execute *P* or ***P***`(...)`

`call_cleanup(`***:Goal***`,`***:Cleanup***`)`
: Executes the procedure call *Goal*. When *Goal* succeeds determinately, is cut, fails, or raises an exception, *Cleanup* is executed.

`call_residue_vars(`***:Goal***`,`***?Vars***`)`
: Executes the procedure call *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.

`fail` — ***ISO***
: fail (start backtracking)

`false` same as fail

`freeze(+Var,:Goal)`
: Blocks *Goal* until `nonvar(Var)` holds.

`if(:P,:Q,:R)`
: for each solution of *P* that succeeds, prove *Q*; if none, prove *R*

`once(:P)` ***ISO***
: Find the first solution, if any, of goal *P*.

`otherwise`
: same as true

`repeat` ***ISO***
: succeed repeatedly on backtracking

`true` ***ISO***
: succeed

`when(+Cond,:Goal)`
: block *Goal* until *Cond* holds

4.3 Loading Programs

4.3.1 Overview

There are two ways of loading programs into Prolog—loading source files and loading pre-compiled 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 in-line and cannot be traced.

The compiler operates in three different modes, controlled by the `compiling` Prolog flag. The possible states of the flag are:

`compactcode`
: Compilation produces byte-coded abstract instructions. The default.

`profiledcode`
: Compilation produces byte-coded abstract instructions instrumented to produce execution profiling data. See Section 9.2 [Execution Profiling], page 305. Profiling is not available in runtime systems.

`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 138 for information on the module system).

4.3.2 The Load Predicates

Loading a program is accomplished by one of these predicates

`[]`
`[:File|+Files]`
`load_files(:Files)`
`load_files(:Files, +Options)`
: loads source or PO file(s), whichever is the more recent, according to *Options*.

`compile(:File)`
: loads source file into virtual machine code.

`consult(:File)`
`reconsult(:File)`
: loads source file into interpreted representation.

`ensure_loaded(:File)`
: loads a source and PO file, whichever is more recent, unless the file has already been loaded and it has not been modified since it was loaded.

The following notes apply to all the Load Predicates:

1. The *File* argument must be one of the following:

   an atom 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 80)

   a list of any atom listed above;

   the atom `user`
2. These predicates resolve relative file names in the same way as `absolute_file_name/2`. For information on file names refer to Section 4.5 [ref-fdi], page 80.
3. The above predicates raise an exception if any of the files named in *File* 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, and an appropriate error message is printed.
4. 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`, warnings are printed when a *sentence* (see Section 4.1.7.3 [ref-syn-syn-sen], page 49) containing variables not beginning with '`_`' occurring once only is compiled or consulted.

   `redefine_warnings`
   : If `on`, warnings are printed 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`, warnings are printed when clauses are not together in source files, and the relevant predicate has not been declared `discontiguous`.

5. 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`.
6. If a file being loaded is not a module-file, all the predicates defined in the file are loaded into the source module. The form `load_files(Module:File)` can be used to load the file into the specified module. See Section 4.11.3 [ref-mod-def], page 139, for information about module-files. If a file being loaded *is* a module-file, 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.
7. If there are any directives in the file being loaded, that is, any terms with principal functor `:-/1` or `?-/1`, 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.
8. 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 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.
9. If ***File*** is the atom `user`, or ***File*** is a list, and during loading of the list `user` is encountered, 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 an end-of-file character, or the atom `end_of_file` followed by a full-stop.
10. 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.
11. The current load context (module, file, stream, directory) can be queried using `prolog_load_context/2`.

12. 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 4.15.6 [ref-ere-int], page 184), and then load an altered version of some procedures. If you do this, 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, 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, alternative clauses in the old definition will still be used.

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 spec, causes the specified predicates to become ***multifile***. This means that if more clauses are subsequently loaded from other files for the same predicate, the new clauses will not replace the old ones, but will be added at the end instead. As of release 3, multifile declarations are required in all files from which clauses to a multifile predicate are loaded.

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 58.

If a file containing clauses for a multifile predicate is reloaded, the old clauses from the same file are removed. The new clauses are added at the end.

If a multifile predicate is loaded from a file with no multifile declaration for it, 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, it must also be done so in the other files from which it is loaded. Hook predicates should always be declared as multifile, as this is the convention followed in the library modules.

Multifile declarations *must precede* any other declarations for the same predicate(s)!

4.3.4.2 Dynamic Declarations

A declaration

`:- dynamic` ***:PredSpec***`, ...,` ***:PredSpec***`.` ***ISO***

where each ***PredSpec*** is a predicate spec, 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 152.

4.3.4.3 Volatile Declarations

A declaration

`:- volatile` ***:PredSpec***`, ...,` ***:PredSpec***`.`

where each ***PredSpec*** is a predicate spec, causes the specified predicates to become ***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 run-time, 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 `volatile`, `dynamic` or `multifile` when the saved-state is restored.

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:

`:- discontiguous` ***:PredSpec***`, ...,` ***:PredSpec***`.` ***ISO*** disables such warnings for the predicates specified by each ***PredSpec***. The warnings can also be disabled globally by setting the `discontiguous_warnings` flag to `off`.

4.3.4.5 Block Declarations

The declaration

```
:- 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`, 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.

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:

```
:- meta_predicate :MetaPredSpec, ..., :MetaPredSpec.
```

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 148 and Section 4.11.15 [ref-mod-mne], page 148 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 specs, 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 138, for details.

4.3.4.8 Public Declarations

The only effect of a declaration

```
:- public :PredSpec, ..., :PredSpec.
```

where each ***PredSpec*** is a predicate spec, is to give the SICStus cross-referencer (see Section 9.11 [The Cross-Referencer], page 327) 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 138.

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 Include Declarations

A directive

```
:- include(+Files).                                          ISO
```

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 118.

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).

SICStus Prolog uses the included file name (as opposed to the embedding file name) only in source-linked debugging and error reporting.

4.3.4.11 Initializations

A directive

```
:- initialization :Goal.                                    ISO
```

in a file includes *Goal* to the set 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.

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.

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, [to-
ken|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, *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.

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, *Goal2* is used for further processing, and may be arbitrarily complex.

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.

4.3.6 Predicate List

Detailed information is found in the reference pages for the following:

`[]`
`[:F|+Fs]` same as `load_files([F|Fs])`

`block :P` ***declaration***
: predicates specified by *P* should block until sufficiently instantiated

`user:term_expansion(+Term1, +Layout1, +Tokens1, -Term2, -Layout2, -Tokens2)` ***hook***
: Overrides or complements the standard transformations to be done by `expand_term/2`.

`compile(:F)`
: load compiled clauses from files *F*

`consult(:F)`
`reconsult(:F)`
: load interpreted clauses from files *F*

`expand_term(+T,-X)` ***hookable***
: term *T* expands to term *X* using `user:term_expansion/6` or grammar rule expansion

`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

`ensure_loaded(`***:F***`)` ***ISO***
: load *F* if not already loaded

`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*** ***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***`)`
: restore the state saved in file *F*

`use_module(`***:F***`)`
`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

`volatile` ***:P*** ***declaration***
: predicates specified by *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 le* (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 `save_files/2`, `save_modules/2`, and `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 68.

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 119, any blackboard data (see Section 4.12.9 [ref-mdb-bbd], page 160), database data (see Section 4.12.1 [ref-mdb-bas], page 152), and profiling data (see Section 9.2 [Execution Profiling], page 305); but no information for source-linked debugging.

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*.

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 `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 4.3.4.11 [Initializations], page 75).

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), 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, multifile clauses 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), `load_files(file1)` will load the source ('`.pl`') or the PO, whichever is the most recent. Refer to Section 4.3 [ref-lod], page 68 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

`user:runtime_entry(+S)` ***hook***
: entry point for a runtime system

`save_files(+L,+F)`
: Saves the modules, predicates and clauses and directives in the given files *L* into file *F*

`save_modules(+L,+F)`
: save the modules specifed 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` ***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 *le* 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 *le*, 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, to locate the file specified by (G) each `home` directory is searched in sequence for the file '`.login`'. If '`/usr/jackson/.login`' exists, 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. These predicates are:

```
absolute_file_name/[2,3]
compile/1
consult/1
ensure_loaded/1
load_files/[1,2]
```

```
load_foreign_resource/1
open/[2,3]
reconsult/1
restore/1
save_files/2
save_module/2
save_predicates/2
save_program/[1,2]
see/1
tell/1
use_module/[1,2,3]
```

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, 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 81, clauses like the one following should be placed in the initialization file so that they are automatically available to user programs after startup:

```
:- 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 behaves as if it were a multifile predicate with the following clauses. See Section 4.9.4 [ref-lps-flg], page 119 for more info on the Prolog flag `host_type`. The environment variables `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 run-time. The environment variable `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)`.

```
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
  ...
```

`user:library_directory/1` is undefined at startup, but behaves as if it were a multifile predicate with a single clause defining the location of the Prolog library. The initial value is the same as the value of the environment variable `SP_LIBRARY_DIR`. The predicate may succeed nondeterminately in this search for a library directory.

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 80. Let *FileSpec* be the given or transformed atomic file specification.

The file specification `user` stands for the standard input or output stream, depending on context.

A file specification *FileSpec* other than `user` is subject to *syntactic rewriting*. Depending on the operation, the resulting absolute filename is subject to further processing. Syntactic rewriting is performed wrt. a context directory *Context* (an absolute path), in the following steps:

Under Windows, all '\' characters are converted to '`/`'.

A '`$`***var***' in the beginning of *FileSpec*, followed by '`/`' or the end of the path, is replaced by the absolute path of the value of the environment variable *var*. In addition, under Windows, all '\' characters are converted to '`/`'. If *var* doesn't exist or its value is empty, a permission error is raised.

A relative path that does not begin with '`/`' is made absolute by prepending *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 prepending the root (see below) of *Context*.

A '`~`***user***' in the beginning of *FileSpec*, followed by '`/`' or the end of the path, is replaced by the absolute path of the home directory of *user*. If the home directory of *user* cannot be determined, a permission error is raised.

Under Windows this has not been implemented, instead a permission error is raised.

If the home directory of *user* is a relative path, it is made absolute using *Context* if needed.

A '~' in the beginning of *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, a permission error is raised.

The the home directory of the current user is a relative path it is made absolute using *Context* if needed. In addition, under Windows, all '\' characters are converted to '/'.

Under Windows, the home directory of the current user is determined using the environment variables `HOMEDRIVE` and `HOMEPATH`.

If *FileSpec* is a relative file name, *Context* is prepended to it.

The *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.

driveletter`:/` where *driveletter* is a 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 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, a permission error is raised.

Any trailing '/' is deleted unless it is part of the *root*.

Finally, under Windows, the path is normalized 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 normalize to identical atoms.

A downside to this normalization is that you cannot choose the case used for files created by SICStus Prolog on file systems such as NTFS that are case-preserving (but case-insensitive). For instance, you cannot create a file that has a mixed-case name in the file system, such as '`MyClass.java`'. Such a file would instead have a file system name '`myclass.java`'.

This seldom matters, except for aesthetics, since any Windows application that tries to open a file named '`MyClass.java`' will also find '`myclass.java`'.

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

$VAR1/../local/
   ↦ /opt/local

$VAR2/misc/.
   ↦ /usr/foo/misc

$VAR3/misc/.
   ↦ /home/joe/temp/misc

~joe/../jenny/bin.
   ↦ /home/jenny/bin
```

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`'.

```
/foo/bar
   ↦ C:/foo/bar

foo//../../blip
   ↦ c:/source/blip

$VAR1/../local/
   ↦ //server/docs/local

$VAR2/misc/.
   ↦ c:/source/proj1/foo/misc

$VAR3/misc/.
   ↦ c:/home/temp/misc
```

```
~joe/../jenny/bin.
   ⇗ error
```

4.5.3 List of Predicates

Detailed information is found in the reference pages for the following:

`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

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 ***stream***s. 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[3]. Each stream is used either for input or for output, but 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 (A)
predicate_name/n+1 (B)

[3] At the C level, you can define more general streams, e.g. referring to pipes or to encrypted files.

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 *le speci cation*, 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 80. Stream objects are discussed in Section 4.6.7.1 [ref-iou-sfh-sob], page 93. Reading the state of open streams is discussed in Section 4.6.8 [ref-iou-sos], page 99.

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

- `read(`***-Term***`)`
- `read(+`***Stream***`, `***-Term***`)`
- `read_term(`***-Term***`, +`***Options***`)`
- `read_term(+`***Stream***`, `***-Term***`, +`***Options***`)`

`read_term/[2,3]` offers many options to return extra information about the term.

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 48. The full-stop is removed from the input stream but is not a part of the term that is read.
- `read/[1,2]` does not terminate until the full-stop is encountered. Thus, if you type at top level

  ```
  | ?- read(X)
  ```

  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 43, 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 `prolog_flag/3` or using `read_term/[2,3]`.

If the end of the current input stream has been reached, `read(`***X***`)` will cause *X* to be unified with the atom `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 '`|: `' on return to the top level.

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. (`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.

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]`, you must send the terminating full-stop to the output stream yourself. For example,

```
| ?- write(a), put_code(0'.), nl.
```

If ***Term*** is uninstantiated, it is written as an anonymous variable (an underscore followed by a non-negative integer).

`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 43, 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 39 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 therefore always written in the form:

```
predicate_name(arg1, ..., argn)
```

Atoms output by `write/[1,2]` cannot in general be read back using `read/[1,2]`. For example,

```
| ?- write('a b').
a b
```

If you want to be sure that the atom can be read back by `read/[1,2]`, 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 `yes`. Note also that the printing of quoted atoms is sensitive to character escaping (see Section 4.1.4 [ref-syn-ces], page 43).

`write/[1,2]` and `writeq/[1,2]` treat terms of the form `'$VAR'(`***N***`)` specially: they write '`A`' if N=0, '`B`' if N=1, ...'`Z`' if N=25, '`A1`' if N=26, etc. Terms of this form are generated by `numbervars/3` (see Section 4.8.6 [ref-lte-anv], page 111). Terms of the form `'$VAR'(`***X***`)`, where X is not a number are written as unquoted terms. For example,

```
| ?- writeq(a('$VAR'(0),'$VAR'('Test'))).
a(A,Test)
```

`write_canonical/1` does *not* treat terms of the form `'$VAR'(`***N***`)` specially. It writes square bracket lists using `./2` and `[]` (that is, `[a,b]` is written as '`.(a,.(b,[]))`').

4.6.4.4 Displaying Terms

Like `write_canonical/[1,2]`, `display/1` ignores operator declarations and shows all compound terms in standard prefix form. For example, the command

```
| ?- display(a+b).
```

produces the following:

```
+(a,b)
```

Calling `display/1` is a good way of finding out how Prolog parses a term with several operators. Unlike `write_canonical/[1,2]`, `display/1` does not put quotes around atoms and functors.

4.6.4.5 Using the Portray Hook

By default, the effect of `print/[1,2]` is the same as that of `write/[1,2]`, but you can change its effect by providing clauses for the hook predicate `user:portray/1`.

If *X* is a variable, it is printed using `write(`***X***`)`. Otherwise the user-definable procedure `user:portray(`***X***`)` is called. If this succeeds, it is assumed that *X* has been printed and `print/[1,2]` exits (succeeds).

If the call to `user:portray/1` fails, and if *X* is a compound term, `write/[1,2]` is used to write the principal functor of *X* and `print/[1,2]` is called recursively on its arguments. If *X* is atomic, it is written using `write/[1,2]`.

When `print/[1,2]` has to print a list, say `[X1,X2,...,Xn]`, it passes the whole list to `user:portray/1`. As usual, if `user:portray/1` succeeds, it is assumed to have printed the entire list, and `print/[1,2]` does nothing further with this term. Otherwise `print/[1,2]` writes the list using bracket notation, calling `print/[1,2]` on each element of the list in turn.

Since `[X1,X2,...,Xn]` is simply a different way of writing `.(X1,[X2,...,Xn])`, one might expect `print/[1,2]` to be called recursively on the two arguments `X1` and `[X2,...,Xn]`, giving `user:portray/1` a second chance at `[X2,...,Xn]`. This does *not* happen; lists are a special case in which `print/[1,2]` is called separately for each of `X1,X2,...Xn`.

4.6.4.6 Portraying a Clause

If you want to print a clause, `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 `compile/1`, 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/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.

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.

> **Please note:** The note about "tty-" predicates at the beginning of Section 4.6.5 [ref-iou-cin], page 91 applies here as well.

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 98.

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 a third 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                *
*************************************************************
```

For details, including the code to produce this example, see the example program in the reference page for `format/[2,3]`.

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 699) 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

'$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 95 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.

4.6.7.2 Exceptions Related to Streams

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

instantiation_error
: *Stream* argument is not ground

type_error
: *Stream* is not an input (or output) stream type.

existence_error
: *Stream* is syntactically valid but does not name an open stream.

permission_error
: *Stream* names an open stream but the stream is not open for the required operation, or has reached the end of stream on input, or is binary when text is required, or vice versa.

The reference page for each stream predicate will simply refer to these as "Stream errors" and will go on to detail other exceptions that may be raised for a particular predicate.

4.6.7.3 Suppressing Error Messages

If the fileerrors flag is set to off, 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, 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, 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`.
>
> `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, 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`, 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, `see/1` fails. In addition, if the `fileerrors` flag is set (as it is by default), `see/1` sends an error message to the standard error stream and calls `abort/0`, returning to the top level.
>
> `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`, 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, it is created. If it is not possible to open the file (because of protections, for example), `tell/1` fails. In addition, if the `fileerrors` flag is set (which it is by default), `tell/1` sends an error message to the standard error stream and calls `abort/0`, returning to the top level.

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-iou-sos], page 99).

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 suffix `LE` and `BE` denotes respectiviely 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`.

The encoding to use can be specified when using `open/4` and similar predicates using the option `encoding/1`. When openening 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`.

4.6.7.6 Finding the Current Input Stream

`current_input(`***Stream***`)` unifies *Stream* with the current input stream.

`seeing(`***S***`)` unifies *S* with the current input stream. This is exactly the same as `current_input(`*S*`)`, except that *S* will be unified with a filename if the current input stream was opened by `see/1` (see Section 4.6.7.4 [ref-iou-sfh-opn], page 95).

`seeing/1` 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), 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).

`seeing/1` can be followed by `see/1` to ensure that a section of code leaves the current input unchanged

4.6.7.7 Finding the Current Output Stream

`current_output(`***Stream***`)` unifies *Stream* with the current output

stream.

`telling(`***S***`)` unifies *S* with the current output stream. This is exactly the same as `current_output(`*S*`)`, except that *S* will be unified with a filename if the current output stream was opened by `tell/1`.

A common usage of `telling/1` is

```
tell('Some File Name')
...
telling('Some File Name')
```

`telling/1` should succeed if the current input stream was not changed (or if changed, restored). It succeeds for any filename (including `user`) and any stream object opened by `open/3` (see Section 4.6.7.4 [ref-iou-sfh-opn], page 95), but fails for `user_output` and any stream object opened by `tell/1` (see Section 4.6.7.4 [ref-iou-sfh-opn], page 95). Passing file names to `tell/1` is the only DEC-10 Prolog usage of `telling/1`, so SICStus Prolog is compatible with this usage.

> *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 the three special streams for the standard input, output, and error channels.

`stream_property(`***`Stream`***`, `***`Property`***`)` succeeds if *Stream* is a currently open stream with property *Property*. The three standard channels are *not* ignored.

4.6.7.9 Closing a Stream

`close(`***`X`***`)` closes the stream corresponding to *X*.

If *X* is a stream object, and if the corresponding stream is open, it will be closed; otherwise, `close/1` succeeds immediately, taking no action.

If *X* is a file specification, the corresponding stream will be closed. It is only closed if the file was opened by `see/1` or `tell/1`. In the example

```
see(foo),
...
close(foo)
```

'`foo`' will be closed. However, in the example

```
open(foo, read, S),
...
close(foo)
```

an exception will be raised and '`foo`' will not be closed.

`told/0` closes the current output stream. The current output stream is then set to be `user_output`; that is, the user's terminal.

`seen/0` closes the current input stream. The current input stream is then set to be `user_input`; that is, the user's terminal.

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*.

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 *O set* = 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***`)`
testing whether at end of line on input stream *S*

`at_end_of_stream` ***ISO***
`at_end_of_stream(+`***S***`)` ***ISO***
testing whether end of file is reached for the input stream *S*

`flush_output` ***ISO***
`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***`)` ***ISO***
`get_code(+`***S***`,`***-C***`)` ***ISO***
: *C* is the next character code on text input stream *S*

`nl` ***ISO***
`nl(+`***S***`)` ***ISO***
: send a newline to stream *S*

`peek_byte(+`***C***`)` ***ISO***
`peek_byte(+`***S***`,+`***C***`)` ***ISO***
: 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***`)`
`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 (record) 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)`
: *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)` ***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)`
: 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))` **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(+InChar, +OutChar)` **ISO**
: The mapping of *InChar* to *OutChar* is added to the character-conversion mapping.

`current_char_conversion(?InChar, ?OutChar)` **ISO**
: *InChar* is mapped to *OutChar* in the current character-conversion mapping.

`current_op(?P,?T,?A)` **ISO**
: atom *A* is an operator of type *T* with precedence *P*

`display(+T)`
: write term *T* to the user output stream in prefix notation

`format(+C,:A)`
`format(+S,+C,:A)`
: write arguments *A* on stream *S* according to control string *C*

`op(+P,+T,+A)` **ISO**
: make atom *A* an operator of type *T* with precedence *P*

`user:portray(+T)` **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)` **hookable**
`print(+S,+T)` **hookable**
: display the term *T* on stream *S* using `user:portray/1` or `write/2`

`read(-T)` **ISO**
`read(+S,-T)` **ISO**
: read term *T* from stream *S*

`read_term(-T,+O)` ***ISO***
`read_term(+S,-T,+O)` ***ISO***
: read *T* from stream *S* according to options *O*

`write(+T)` ***ISO***
`write(+S,+T)` ***ISO***
: write term *T* on stream *S*

`write_canonical(+T)` ***ISO***
`write_canonical(+S,+T)` ***ISO***
: write term *T* on stream *S* so that it can be read back by `read/[1,2]`

`writeq(+T)` ***ISO***
`writeq(+S,+T)` ***ISO***
: write term *T* on stream *S*, quoting atoms where necessary

`write_term(+T,+O)` ***ISO,hookable***
`write_term(+S,+T,+O)` ***ISO,hookable***
: writes *T* to *S* according to options *O*

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 103 and Section 4.7.4 [ref-ari-acm], page 104. All of them take arguments of the type *Expr*, which is described in detail in Section 4.7.5 [ref-ari-aex], page 105.

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*.

Term 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`
`type_error`
: Atom given as expression, or float given where integer required.

`evaluation_error`
: Attempting to divide by zero, or undefined function, or function undefined for the given argument.

`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, 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.

Expr1 `=:=` ***Expr2***
: succeeds if the results of evaluating terms *Expr1* and *Expr2* as arithmetic expressions are equal

Expr1 `=\=` ***Expr2***
: succeeds if the results of evaluating terms *Expr1* and *Expr2* as arithmetic expressions are not equal

Expr1 `<` ***Expr2***
: succeeds if the result of evaluating *Expr1* as an arithmetic expression is less than the result of evaluating *Expr2* as an arithmetic expression.

Expr1 `>` ***Expr2***
: succeeds if the result of evaluating *Expr1* as an arithmetic expression *Expr1* is greater than the result of evaluating *Expr2* as an arithmetic expression.

Expr1 `=<` ***Expr2***
: succeeds if the result of evaluating *Expr1* as an arithmetic expression is not greater than the result of evaluating *Expr2* as an arithmetic expression.

Expr1 `>=` ***Expr2***
: succeeds if the result of evaluating *Expr1* as an arithmetic expression is not less than the result of evaluating *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, Prolog raises exceptions of the appropriate type (see Section 4.15.3 [ref-ere-hex], page 174). Some arithmetic operations can also detect overflows. They also raise exceptions, e.g. division by zero results in a domain 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.

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.

`+(`***X***`)`
: The value is *X*.

-X
: The value is the negative of *X*. *ISO*

X+Y
: The value is the sum of *X* and *Y*. *ISO*

X-Y
: The value is the difference between *X* and *Y*. *ISO*

X*Y
: The value is the product of *X* and *Y*. *ISO*

X/Y
: The value is the *float* quotient of *X* and *Y*. *ISO*

X//Y ***ISO***
: The value is the *integer* quotient of *X* and *Y*. The result is always truncated towards zero. *X* and *Y* have to be integers.

X `rem` ***Y*** ***ISO***
: The value is the *integer* remainder after dividing *X* by *Y*, i.e. `integer(`***X***`)-integer(`***Y***`)*(`***X//Y***`)`. The sign of a nonzero remainder will thus be the same as that of the dividend. *X* and *Y* have to be integers.

X `mod` ***Y*** ***ISO***
: The value is *X* modulo *Y*, i.e. `integer(`***X***`)-integer(`***Y***`)*floor(`***X/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)` ***ISO***
The same as `float(integer(X))`. *X* has to be a float.

`float_fractional_part(X)` ***ISO***
The value is the fractional part of *X*, i.e. `X - float_integer_part(X)`. *X* has to be a float.

`float(X)` ***ISO***
The value is the float equivalent of *X*, if *X* is an integer; otherwise, *X* itself.

`X/\Y` ***ISO*** The value is the bitwise conjunction of the integers *X* and *Y*. *X* and *Y* have to be integers.

`X\/Y` ***ISO*** The value is the bitwise disjunction of the integers *X* and *Y*. *X* and *Y* have to be integers.

`X\Y` The value is the bitwise exclusive or of the integers *X* and *Y*.

`\(X)` ***ISO*** The value is the bitwise negation of the integer *X*. *X* has to be an integer.

`X<<Y` ***ISO*** The value is the integer *X* shifted left by *Y* places. *X* and *Y* have to be integers.

`X>>Y` ***ISO*** The value is the integer *X* shifted right by *Y* places. *X* and *Y* have to be integers.

`[X]` 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)` ***ISO***
The value is the absolute value of *X*.

`sign(X)` ***ISO***
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)` ***ISO***
The value is the closest integer to *X*. *X* has to be a float. If *X* is exactly half-way between two integers, it is rounded up (i.e. the value is the least integer greater than *X*).

`truncate(X)` ***ISO***
The value is the closest integer between X and 0. X has to be a float.

`floor(X)` ***ISO***
The value is the greatest integer less or equal to X. X has to be a float.

`ceiling(X)` ***ISO***
The value is the least integer greater or equal to X. X has to be a float.

`sin(X)` ***ISO***
The value is the sine of X.

`cos(X)` ***ISO***
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.

`tanh(X)` The value is the hyperbolic tangent of X.

`coth(X)` The value is the hyperbolic cotangent of X.

`asin(X)` The value is the arc sine of X.

`acos(X)` The value is the arc cosine of X.

`atan(X)` ***ISO***
The value is the arc tangent of X.

`atan2(X,Y)`
The value is the four-quadrant arc tangent of X and Y.

`acot(X)` The value is the arc cotangent of X.

`acot2(X,Y)`
The value is the four-quadrant arc cotangent of X and Y.

`asinh(X)` The value is the hyperbolic arc sine of X.

`acosh(X)` The value is the hyperbolic arc cosine of X.

`atanh(X)` The value is the hyperbolic arc tangent of X.

`acoth(X)` The value is the hyperbolic arc cotangent of X.

`sqrt(X)` ***ISO***
The value is the square root of X.

`log(X)` ***ISO***
The value is the natural logarithm of X.

`log(Base,X)`
The value is the logarithm of X in the base *Base*.

`exp(X)` ***ISO***
The value is the natural exponent of X.

X `**` ***Y*** ***ISO***

`exp(`***X***`,`***Y***`)` The value is X raised to the power of Y.

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:

```
X is "c" - "a"        (A)
X is [99] - [97]      (B)
X is 99 - 97          (C)
```

A cleaner way to do the same thing is

```
X is 0'c - 0'a
```

4.7.6 Predicate Summary

`-`***Y*** `is +`***X*** ***ISO***

Y is the value of arithmetic expression X

`+`***X*** `=:= +`***Y*** ***ISO***

the results of evaluating terms X and Y as arithmetic expressions are equal.

`+`***X*** `=\= +`***Y*** ***ISO***

the results of evaluating terms X and Y as arithmetic expressions are not equal.

`+`***X*** `< +`***Y*** ***ISO***

the result of evaluating X as an arithmetic expression is less than the result of evaluating Y as an arithmetic expression.

`+`***X*** `>= +`***Y*** ***ISO***

the result of evaluating X as an arithmetic expression is not less than the result of evaluating Y as an arithmetic expression.

`+`***X*** `> +`***Y*** ***ISO***

the result of evaluating X as an arithmetic expression X is greater than the result of evaluating Y as an arithmetic expression.

`+`***X*** `=< +`***Y*** ***ISO***

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)` **ISO**
: term *T* is an atom

`atomic(+T)` **ISO**
: term *T* is an atom or a number

`callable(+T)`
: *T* is an atom or a compound term

`compound(+T)` **ISO**
: *T* is a compound term

`float(+N)` **ISO**
: *N* is a floating-point number

`ground(+T)`
: term *T* is a nonvar, and all substructures are nonvar

`integer(+T)` **ISO**
: term *T* is an integer

`mutable(+X)`
: *X* is currently instantiated to a mutable term.

`nonvar(+T)` **ISO**
: term *T* is one of atom, number, compound (that is, *T* is instantiated)

`number(+N)` **ISO**
: *N* is an integer or a float

`simple(+T)`
: *T* is not a compound term; it is either atomic or a var

`var(+T)` **ISO**
: 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 66).

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).
```

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*.

4.8.3 Analyzing and Constructing Lists

To combine two lists to form a third list, use `append(`***Head***`, `***Tail***`, `***List***`)`.

To analyze a list into its component lists in various ways, use `append/3` with *List* instantiated to a proper list. The reference page for `append/3` includes examples of its usage, including backtracking.

To check the length of a list call `length(`***List***`, `***Length***`)`.

To produce a list of a certain length, use `length/2` with *Length* instantiated and *List* uninstantiated or instantiated to a list whose tail is a variable.

To check if a term is the element of a list, use `memberchk(`***Element***`, `***List***`)`.

To enumerate the elements of a list via backtracking, use `member(`***Element***`, `***List***`)`.

To check that a term is NOT the element of a list, use `nonmember(`***Element***`, `***List***`)`, which is equivalent to `\+member(`***Element***`, `***List***`)`.

4.8.4 Converting between Constants and Text

Three predicates convert between constants and lists of character codes: `atom_codes/2`, `number_codes/2`, and `name/2`. Two predicates convert between constants and lists of character atoms: `atom_chars/2`, `number_chars/2`.

`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.

`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, `atom_chars(`***Atom***`, `***Chars***`)` and `number_chars(`***Atom***`, `***Chars***`)` are relations between a constant and a list consisting of the character atoms comprising the printed representation of the constant.

`name/2` converts between a constant and a code-list. Given a code-list, `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 `atom_codes/2` but not by `name/2`. `name/2` is retained for backwards compatibility with other Prologs. New programs should use `atom_codes/2` or `number_codes/2` as appropriate.

`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 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 new variables that occur nowhere else. This is precisely the effect that would have been obtained from the definition

```
copy_term(Term, Copy) :-
   recorda(copy, copy(Term), DBref),
   instance(DBref, copy(Temp)),
   erase(DBref),
   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, 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 337) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 63), 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.

`copy_term/2` is efficient enough to use without hesitation if there is no solution that does not require the use of meta-logical predicates. However, for the sake of both clarity and efficiency, such a solution should be sought before using `copy_term/2`.

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 104) 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, fie, foe, X = Y, foe(0,2), 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 63) on variables or modifying their attributes (see Section 10.3 [lib-atts], page 337) 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 \== +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*

`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, `sort/2` and `keysort/2` sort lists into the standard order. `keysort/2` takes a list consisting of key-value pairs and sorts according to the key.

Further sorting predicates are available in `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 three 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 *Mutable* is *Datum*.

`update_mutable(+`***Datum***`,+`***Mutable***`)`
: Updates the current value of the mutable term *Mutable* 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***`)`
: T is an atom or a compound term

`compound(+`***T***`)` ***ISO***
: T is a compound term

`float(+`***N***`)` ***ISO***
: N is a floating-point number

`ground(+`***T***`)`
: term T is a nonvar, and all substructures are nonvar

`integer(+`***T***`)` ***ISO***
: term T is an integer

`mutable(+`***X***`)`
: X is currently instantiated to a mutable term.

`nonvar(+`***T***`)` ***ISO***
: term T is one of atom, number, compound (that is, T is instantiated)

`number(+`***N***`)` ***ISO***
: N is an integer or a float

`simple(+`***T***`)`
: T is not a compound term; it is either atomic or a var

`var(+`***T***`)` ***ISO***
: term T is a variable (that is, T is uninstantiated)

`compare(`***-C***`,+`***X***`,+`***Y***`)` ***meta-predicate***
: C is the result of comparing terms X and Y

`+`***X***` == +`***Y*** ***ISO***
: terms X and Y are strictly identical

`+`***X***` \== +`***Y*** ***ISO***
: terms X and Y are not strictly identical

`+X @< +Y` ***ISO***
: term *X* precedes term *Y* in standard order for terms

`+X @>= +Y` ***ISO***
: term *X* follows or is identical to term *Y* in standard order for terms

`+X @> +Y` ***ISO***
: term *X* follows term *Y* in standard order for terms

`+X @=< +Y` ***ISO***
: term *X* precedes or is identical to term *Y* in standard order for terms

`?T =.. ?L` ***ISO***
: the functor and arguments of term *T* comprise the list *L*

`?X = ?Y` ***ISO***
: terms *X* and *Y* are unified

`+X \= +Y` ***ISO***
: terms *X* and *Y* no not unify

`?=(+X,+Y)`
: *X* and *Y* are either strictly identical or do not unify

`arg(+N,+T,-A)` ***ISO***
: the *N*th argument of term *T* is *A*

`atom_chars(?A,?L)` ***ISO***
: *A* is the atom containing the character atoms in list *L*

`atom_codes(?A,?L)` ***ISO***
: *A* is the atom containing the characters in code-list *L*

`atom_concat(?Atom1,?Atom2,?Atom12)` ***ISO***
: Atom *Atom1* concatenated with *Atom2* gives *Atom12*.

`atom_length(+Atom,-Length)` ***ISO***
: *Length* is the number of characters of the atom *Atom*.

`char_code(?Char,?Code)` ***ISO***
: *Code* is the character code of the one-char atom *Char*.

`copy_term(+T,-C)` ***ISO***
: *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(+Var,-Goal)`
: The goal *Goal* is blocked on the variable *Var*.

`functor(`***?T,?F,?N***`)` ***ISO***
: 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***`)` ***ISO***
: *N* is the numeric representation of list of character atoms *L*

`number_codes(`***?N,?L***`)` ***ISO***
: *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***`)` ***ISO***
: 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*.

`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***`)`
: 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***`)`
: 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 118. 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 10.34.9 [Defining Primitive Constraints], page 524.

`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.

`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(`***SkeletalGoal***`)]`, 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.

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`.

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`.

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/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` ***read-only,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 -a hello world 2001
  ```

  the value will be `[hello,world,'2001']`.

`bounded` ***ISO,read-only,volatile***
: 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` ***ISO,volatile***
: If this flag is `on`, 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` 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 68).

  `compactcode`
  : Compilation produces byte-coded abstract instructions (the default).

  `profiledcode`
  : Compilation produces byte-coded abstract instructions instrumented to produce execution profiling data.

  `debugcode`
  : Compiling is replaced by consulting.

`debugging` ***volatile***

Corresponds to the predicates `debug/0`, `nodebug/0`, `tracee/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` ***ISO,volatile***

The flag `debug`, prescribed by the ISO Prolog standard, is a simplified form of the `debugging` flag:

`off` The debugger is switched off (the default).

`on` The debugger is switched on (to trace mode, if previously switched off).

(The flags `debugging` and `debug` have no effect in runtime systems.)

`double_quotes` ***ISO,volatile***

Governs the interpretation of double quoted strings (see Section 4.1.3.2 [ref-syn-cpt-sli], page 42):

`codes` Code-list comprising the string.

`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 51).

`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 89), to be used in the debugger's messages. The initial value is `[quoted(true),numbervars(true),portrayed(true),max_depth(10)]`.

`discontiguous_warnings`
: `on` or `off`. Enable or disable warning messages when clauses are not together in source files. Initially `on`. (This warning is always disabled 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 unless the global stack is at least *Margin* kilobytes. 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` ***read-only,volatile***
: 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` ***ISO,read-only,volatile***
: One of the flags defining the integer type. In SICStus Prolog its value is `toward_zero`, indicating that the integer division (`(//)/2`) and integer remainder (`rem/2`) arithmetic functions use rounding toward zero; see Section 4.7 [ref-ari], page 103.

`max_arity` ***ISO,read-only,volatile***
: Specifies the maximum arity allowed for a compound term. In SICStus Prolog this is 255.

`max_integer` ***ISO,read-only,volatile***
: Specifies the largest possible integer value. As in SICStus Prolog the range of integers in not bounded, `prolog_flag/3` and `current_prolog_flag/2` will fail, when accessing this flag.

`min_integer` ***ISO,read-only,volatile***
: Specifies the smallest possible integer value. As in SICStus Prolog the range of integers in not bounded, `prolog_flag/3` and `current_prolog_flag/2` will fail, when accessing this flag.

`redefine_warnings`
: `on` or `off`. 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.

Initially `on`. (This warning is always disabled in runtime systems.)

`single_var_warnings`
: `on` or `off`. Enable or disable warning messages when a ***sentence*** (see Section 4.1.7.3 [ref-syn-syn-sen], page 49) containing variables not beginning with '`_`' occurring once only is compiled or consulted. Initially `on`.

`source_info` ***volatile***
: `emacs` or `on` or `off`. If not `off` while source code is being loaded, information about line numbers and file names are stored with the loaded code. If the value is `on` while debugging, 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, 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, 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 173. (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 4.0.2 ...`

  `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 89), 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 143). Corresponds to the predicate `set_module/1`.

`unknown` ***ISO***
: Corresponds to the predicate `unknown/2` (see Chapter 5 [Debug Intro], page 197).

  `trace` Causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity. (This setting is not possible in runtime systems.)

  `fail` Causes calls to such predicates to fail.

  `warning` Causes calls to such predicates to display a warning message and then fail.

  `error` Causes calls to such predicates to raise an exception (the default). See Section 4.15 [ref-ere], page 173.

`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 3 #0: Wed Mar 15 12:29:29 MET 1995'`.

You can use `prolog_flag/2` to enumerate all the *FlagName*s 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 compiled. During loading of a PO file, the corresponding source file name is returned.

`file` Outside included files (see Section 4.3.4.10 [Include Declarations], page 74) 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 compiled/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 148). 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 compiled or loaded from.

`term_position`
A term representing the stream position of the last clause read.

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***
`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*.

4.10 Memory Use and Garbage Collection

4.10.1 Overview

SICStus Prolog uses five data areas: program space, global stack, local stack, choicepoint 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 choicepoint stack contains data representing outstanding choices for some goals or disjunctions. Space on the choicepoint 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 133.

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, 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, the Prolog system attempts to allocate unused space from the other Prolog data areas. If additional space cannot be found, 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]).
:- p(_).
```

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

```
p(X) :- p([X]).
:- 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

`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 `trimcore/0` is called, even though the stack may have shrunk considerably in the meantime. The effect of `trimcore/0` is to reduce the free space in all the data areas as much as possible, and to give the space no longer needed back to the operating system. `trimcore/0` is called each time Prolog returns to the top level or the top of a break level.

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 `statistics/0`.

The output from `statistics/0` looks like this:

```
memory (total)       1718512 bytes
   global stack       185472 bytes:       1784 in use,    183688 free
   local stack         10496 bytes:        308 in use,     10188 free
   trail stack         92750 bytes:        244 in use,     92506 free
   control stack       92722 bytes:        216 in use,     92506 free
   atoms              109849 bytes:       3698 in use,   1044877 free
   program space     1337072 bytes:    1048136 in use,    288936 free
   program space breakdown:
            compiled code              469800 bytes
            predicate                  134624 bytes
            try_node                   129632 bytes
            atom                       123064 bytes
            sw_on_key                   92592 bytes
            incore_info                 43392 bytes
            atom table                  36864 bytes
            interpreted code             7656 bytes
            atom buffer                  2560 bytes
            miscellaneous                2096 bytes
            BDD hash table               1560 bytes
            numstack                     1024 bytes
            FLI stack                    1024 bytes
            SP_malloc                     952 bytes
            int_info                      688 bytes
            source info (itable)          368 bytes
            module                        128 bytes
            source info (lheap)            80 bytes
            foreign resource               32 bytes
    No memory resource errors

       0.000 sec. for 3 global, 21 local, and 0 control space overflows
       0.000 sec. for 0 garbage collections which collected 0 bytes
       0.000 sec. for 0 atom garbage collections which collected 0 atoms 0 bytes
       0.010 sec. for 1 memory defragmentations
       0.420 sec. runtime
      26.100 sec. elapsed time
```

Note the use of indentation to indicate sub-areas. That is, memory contains the program space, global space, and local stack, and the global space contains the global stack 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, 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***`, `***List***`)` 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.

Keyword *List*

`global_stack`
: `[`***size used***`,`***free***`]`
This refers to the global stack, where compound terms are stored. The values are gathered before the list holding the answers is allocated.

`local_stack`
: `[`***size used***`,`***free***`]`
This refers to the local stack, where recursive predicate environments are stored.

`trail`
: `[`***size used***`,`***free***`]`
This refers to the trail stack, where conditional variable bindings are recorded.

`choice`
: `[`***size used***`,`***free***`]`
This refers to the choicepoint stack, where partial states are stored for backtracking purposes.

`core`
`memory`
: `[`***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.

`heap`
`program`
: `[`***size used***`,`***size free***`]`
These refer to the amount of memory allocated for the database, symbol tables, and the like.

`runtime`
: `[`***since start of Prolog***`,`***since previous statistics***`]` These refer to CPU time used while executing, excluding time spent garbage collecting, stack shifting, or in system calls. The second element is the time since the last call to `statistics/2` with this key. It is not affected by calls to `statistics/0`.

`total_runtime`
: `[`***since start of Prolog***`,`***since previous statistics***`]` These refer to total CPU time used while executing, including memory management such as garbage collection but excluding system calls. The second element is the time since the last call to `statistics/2` with this key. It is not affected by calls to `statistics/0`.

`walltime`
: `[`***since start of Prolog***`,`***since previous statistics***`]` These refer to absolute time elapsed. The second element is the time since the last call to `statistics/2` with this key. It is not affected by calls to `statistics/0`.

`garbage_collection`
: `[`***no. of GCs***`,`***bytes freed***`,`***time spent***`]`

`stack_shifts`
: [***no. of global shifts***,***no. of local/trailtrail shifts***,***time spent***]

`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.

`atom_garbage_collection`
: [***no. of AGCs***,***bytes freed***,***time spent***]

`defragmentation`
: [***no. of defragmentations***,***time spent***]

To see an example of the use of each of these keywords, type

```
| ?- 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:

```
:- statistics(runtime, [T0| _]),
   p,
   statistics(runtime, [T1|_]),
   T is T1 - T0,
   format('p/0 took ~3d sec.~n', [T]).
```

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 heap garbage may have to be executed in a failure-driven loop. Failure-driven loops minimize heap usage from iteration to iteration of a loop via SICStus Prolog's automatic recovery of heap 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 heap space.

```
echo :- repeat,
        read(Term),
        echo_term(Term),
        !.

echo_term(Term) :-
        Term == end_of_file.
echo_term(Term) :-
        write(Term), nl,
        fail.
```

Any heap 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

```
echo :- read(Term),
        echo_term(Term).

echo_term(Term) :-
        Term == end_of_file,
        !.
echo_term(Term) :-
        write(Term), nl,
        echo.
```

Without garbage collection the more natural recursive loop accumulates heap garbage that cannot be reclaimed automatically. While it is unlikely that this trivial example will run out of heap space, larger and more practical applications may be unable to use the clearer recursive style without garbage collection. With garbage collection, all inaccessible heap 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,

```
collect(List) :-
        read(Term),
        collect_term(Term, List).

collect_term(Term, []) :-
        Term == end_of_file,
        !.
collect_term(Term, [Term|List0]) :-
        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 `collect/1`:

```
collect(List) :-
        repeat,
        read(Term),
        store_term(Term),
        !,
        collect_terms(List).

store_term(Term) :-
        Term == end_of_file.

store_term(Term) :-
        assertz(term(Term)),
        fail.

collect_terms([M|List]) :-
        retract(term(M)),
        !,
        collect_terms(List).
collect_terms([]).
```

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, 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 heap space before backtracking may not be affected when garbage collection is enabled. Procedures that do use a large amount of heap 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 heap space requires an inordinate amount of processing time, such information is unlikely to be needed.

However, if a program thought to be using much heap space runs out of space or runs inordinately slowly, 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 Heap Expansion

For most programs, the default settings for the garbage collection parameters should suffice. For programs that have high heap 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 heap 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 heap usage and the effectiveness of garbage collecting.

1. Setting the `gc_margin` higher than the default will cause fewer heap 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 heap to expand so that if it does overflow, the resulting garbage collection will significantly disrupt normal processing. This will be especially so if much of the heap 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 heap 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 heap 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 heap garbage generation
The program's rate of heap 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 heap overflows is as follows:

if `gc` is on

```
and the heap is larger than gc_margin kilobytes then
  garbage collect the heap
  if less than gc_margin kilobytes are reclaimed then
    try to expand the heap
  endif
else
  try to expand the heap
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 118). If a garbage collection reclaims at least the `gc_margin` kilobytes of heap space the heap is not expanded after garbage collection completes. Otherwise, the heap is expanded after garbage collection. This expansion provides space for the future heap usage that will presumably occur. In addition, no garbage collection occurs if the heap is smaller than `gc_margin` kilobytes.

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 heap) 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 heap 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.

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 `read_term/3`, when source or PO files are loaded, when `atom_codes/2` is called with a character list, or when `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 heap garbage collection, invoked automatically as well as through a call to the built-in predicate `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.

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 heap 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 `agc_margin` flag.

Atom garbage collection can be invoked explicitly by calling `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 `statistics/2` with the keyword `atoms`. That call returns a list of the form

```
[number of atoms, atom space in use, atom space free]
```

For example,

```
| ?- statistics(atoms, Stats).

Stats = [4313,121062,31032]
```

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.

```
<driver loop> :-
        ...
        repeat,
           maybe_atom_gc,
           <do next iteration>
           ...
        fail.
<driver loop>.
```

where

```
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 didn't 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's better to invoke atom garbage collection after abolishing or retracting a large database than to do so before.

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:

```
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 137.

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 reclaims 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, 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, 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(newvalue)
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()
{
        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, 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 aren't registered when they should be may be reclaimed on atom garbage collection. One normally doesn't 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, `SP_register_atom()` and `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, `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, `SP_unregister_atom()` returns 0. A permanent atom has a reference count of 256. In addition, if an atom is simultaneously registered 256 times, 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; `SP_register_atom()` and `SP_unregister_atom()` return 256.

4.10.8 Summary of Predicates

`garbage_collect`
: force an immediate garbage collection

`garbage_collect_atoms`
: garbage collect atom space

`statistics`
: display various execution statistics

`statistics(`***?K*,*?V***`)`
: the execution statistic with key *K* has value *V*

`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 ***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 141).

There are two kinds of importation:

1. A module ***M1*** may import a specified set of predicates from another module ***M2***. All the specified predicates should be public in ***M2***.
2. A module ***M1*** may import all the public predicates of another module ***M2***.

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, 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 68) will automatically import all the public predicates in that module. Thus the command

```
:- ensure_loaded(library(basics)).
```

will load the basic 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- le*** 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:

```
:- 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 141. Dynamic creation of modules is described in more detail in Section 4.11.9 [ref-mod-dmo], page 144.

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 9.11 [The Cross-Referencer], page 327

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*, this can be done by creating a "driver" file of the following form:

```
:- 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

```
:- ensure_loaded(File).
```

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 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, 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* or its meta-predicate declaration (see Section 4.11.16 [ref-mod-met], page 148) 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 or whose module name expansion requirements have changed. 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. Whenever the possibility of such inconsistency arises, you will be warned at the end of the load that certain modules need to be reloaded. This warning will be repeated at the end of each subsequent load until those modules have been reloaded.

Modules may be saved to a PO file by calling `save_modules(Modules,File)` (see Section 4.4 [ref-sls], page 77).

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,

   ```
   :- mod:foo(X,Y).
   ```

   always calls `foo/2` in module ***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 142, 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 143. For goals appearing in a file (either as goal clauses or as normal clauses), 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, `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 `call/1`. The goal (A) calls the predicate `foo/1` in the source module; this ensures that in the compound goal (B) both occurrences of `foo/1` refer to the same predicate.

```
call(foo(X))                                                    (A)

call(foo(X)), foo(Y)                                            (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`.

```
| ?- assert(m:f(x)).                                              (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 141), 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)).                                              (D)
```

Another built-in predicate that refers to the source module is `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, `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 *File* into a specific module *M* using

```
| ?- compile(M:File).
```

Thus if *File* is a module-file, this command would cause its public predicates to be imported into module *M*. If *File* is a non-module-file, 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 148. 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 148.

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 `user`. That is, the predicates that are available to be called directly by the user are those that are visible in the module `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 `user`. This can be done by prefixing each goal with the module name, as described in Section 4.11.6 [ref-mod-vis], page 141; 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 `set_module/1`; for example,

```
| ?- set_module(mod).
```

This command will cause subsequent goals typed at the top level to be executed with `mod` as their source module.

The name of the type-in module is always displayed, except when it is `user`. If you are running Prolog under the editor interface, the type-in module is displayed in the status line of the Prolog window. If you are running Prolog without the editor interface, the type-in module is displayed before each top-level prompt.

For example, if you are running Prolog without the editor:

```
| ?- 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.

```
| ?- assert(m:f(x)).                                          (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 ***File*** into the module ***M***.

```
| ?- compile(M:File).                                         (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 143) and then calling `compile(`***File***`)`, or executing the command in module ***M*** as in (C).

```
| ?- M:compile(File).                                         (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, 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, 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:

```
:- module(a, []).

:- dynamic m:a/1.
b(1).
m:c([]).
m:d([H|T]) :- q(H), r(T).
m:(e(X) :- s(X), t(X)).
f(X) :- m:(u(X), v(X)).
```

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 711.

Note that if clauses for one of these predicates are to be spread across multiple files, 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 module becomes current as soon as it is encountered, and a module can never lose the property of being current.

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 four recognized replies are as follows:

y means forget the previous definition of `f/3` from `m1` and use the new definition of `f/3` from `m2` instead.

n means 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

`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

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.

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 `basics` module from the library and then remind yourself of what predicates it defines like this:

```
| ?- compile(library(basics)).
% ... loading messages ...

yes
| ?- predicate_property(P, imported_from(basics)).

P = member(_2497,_2498) ;
P = memberchk(_2497,_2498) ;
  .
  .
  .
```

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 predicates exported by module `m`, use

```
| ?- predicate_property(m:Goal, exported).
```

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`:

```
:- module(m1, [f/1]).
:- dynamic f/1.
f(0).
```

Then `f/1` can be manipulated as if it were defined in the current module. For example,

```
| ?- clause(f(X), true).

X = 0
```

The built-in predicate `listing/1` distinguishes predicates that are imported into the current source module by prefixing each clause with the module name. Thus,

```
| ?- listing(f).

m1:f(0).
```

However, `listing/1` does not prefix clauses with their module if they are defined in the source module itself. Note that

```
| ?- listing.
```

can be used to see all the dynamic predicates defined in or imported into the current type-in module. And

```
| ?- listing(m1:_).
```

can be used to see all such predicates that are defined in or imported into module `m1`.

4.11.15 Module Name Expansion

The concept of a source module is explained in Section 4.11.7 [ref-mod-som], page 142. 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* `->` *M*:*X* where *M* is the name of the source module. For example, the goal `call(X)` is expanded into `call(M:X)` and the goal `clause(Head, Body)` is expanded into `clause(M:Head, Body)`.

Module name expansion is avoided if the argument to be expanded is already a `:/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 `meta_predicate` Declaration

Sometimes a user-defined predicate will require module name expansion (see Section 4.11.15 [ref-mod-mne], page 148). This can be specified by providing a `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, 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 respect to the module containing the call to the sort routine. Suppose that the sort routine is

```
mysort(CompareProc, InputList, OutputList)
```

An appropriate `meta_predicate` declaration for this is

```
:- meta_predicate mysort(:, +, -).
```

The significant argument in the `mysort/3` term is the '`:`', which indicates that module name expansion is required for this argument. 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. There are some exceptions to this compile-time transformation rule; the goal is not transformed if either of the following applies:

1. *A* is of the form *Module:Goal*.
2. *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 *Mn:* `...` *:M2:M1:Goal*. For example, consider the following program fragment adapted from the library (see `library(samsort)` for the full program):

```
:- module(samsort, [samsort/3]).

:- meta_predicate
        samsort(:, +, ?),
        sam_sort(+, :, +, +, ?).

samsort(_, [], []) :- !.
samsort(Order, List, Sorted) :-
        sam_sort(List, Order, [], 0, Sorted).
  .
  .
  .
```

Normally, the `sam_sort/5` goal in this example would have the module name of its second argument expanded thus:

```
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

‘`+`’
‘`-`’
‘`?`’ 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.

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, the relevant arguments are expanded. Otherwise, the goal is left unexpanded. Then, if *N/A* is a built-in predicate, 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) :-
        call(Body, M).

call(Var, M) :- \+callable(Var), !,
        must_be(Term, callable, call(M:Var), 1).
call(!, _) :- !,
        % cut relevant choicepoints.
call((A, B), M) :- !,
        call(A, M),
        call(B, M).
call((A -> B), M) :- !,
    (   call(A, M) ->
        call(B, M)
    ).
call((A -> B ; C), M) :- !,
    (   call(A, M) ->
        call(B, M)
    ;   call(C, M)
    ).
call((A ; B), M) :- !,
    (   call(A, M)
    ;   call(B, M)
    ).
call(\+(A), M) :- !,
    (   call(A, M) ->
        fail
    ;   true
    ).
call(_^A, M) :- !,
        call(A, M).
call(if(A,B,C), M) :- !,
     if(call(A, M),
        call(B, M),
        call(C, M)).
call(once(A), M) :- !,
    (   call(A, M) -> true
    ).
call(Goal, M) :-
    (   predicate_property(M:Goal, meta_predicate(Meta)) ->
        functor(Goal, Name, Arity),
        functor(AGoal, Name, Arity),
        annotate_goal(0, Arity, Meta, Goal, AGoal, M),
        call_goal(AGoal, M)
    ;   call_goal(Goal, M)
    ).

call_goal(asserta(X), _) :- !,
        asserta(X).
call_goal(asserta(X,R), _) :- !,
        asserta(X, R).
% and so on for all built-in predicates
call_goal(Goal, M) :-
        (M:Goal :- Body),
```

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 specifed 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 9.4 [Indexing], page 309, 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 68). Dynamic procedures can be modified by adding or deleting individual clauses using the assert and retract procedures.

If a procedure is defined by being compiled, it is static by default. If you need to be able to add, delete, or inspect the individual clauses of such a procedure, you must make the procedure dynamic.

There are two ways to make a procedure dynamic:

If the procedure is to be compiled, it must be declared to be dynamic before it is defined.

If the procedure is to be created by assertions only, 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:

```
:- 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,

```
:- 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 an exception is raised, and the declaration is ignored.

Note that the symbol ':- ' preceding the word 'dynamic' is essential. If this symbol is omitted, 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 compiled, it is considered to be a part of the redefinition of the procedures specified in its argument. Thus, if you compile a file containing only

```
:- dynamic hello/0
```

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 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, 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 reconsult or recompile 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 9.4 [Indexing], page 309.

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 159) 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 `assert/1`, `asserta/1`, and `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.

`assert(`***:C***`)`
: clause *C* is asserted in an arbitrary position in its predicate

`assert(`***:C***`,`***-R***`)`
: as `assert/1`; reference *R* is returned

`asserta(`***:C***`)`
: clause *C* is asserted before existing clauses

`asserta(`***:C***`,`***-R***`)`
: as `asserta/1`; reference *R* is returned

`assertz(`***:C***`)`
: clause *C* is asserted after existing clauses

`assertz(`***:C***`,`***-R***`)`
: as `assertz/1`; reference *R* is returned

Please note: If the term being asserted contains attributed variables (see Section 10.3 [lib-atts], page 337) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 63), those attributes are not stored in the database. To retain the attributes, you can use `copy_term/3` (see Section 4.8.7 [ref-lte-cpt], page 111).

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.

> **Please note:** Removing all of a predicate's clauses by `retract/1` and/or `erase/1` (see Section 4.12.5.1 [ref-mdb-rcd-efu], page 156) 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* properties is to use `abolish/[1,2]`.

`retract(`***:C***`)`
: erase the first dynamic clause that matches *C*

`retractall(`***:H***`)`
: erase every clause whose head matches *H*

`abolish(`***:F***`)`
: abolish the predicate(s) specified by *F*

`abolish(`***:F***`,`***+O***`)`
: abolish the predicate(s) specified by *F* with options *O*

`erase(+R)`
: erase the clause or recorded term (see Section 4.12.8 [ref-mdb-idb], page 159) 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, 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, 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

```
| ?- <do some stuff>
     ( retract(Clause) -> true ),
     <do more stuff without backtracking>.
```

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 152), 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

```
:- dynamic foo/0.
foo :- assertz(foo), fail.
```

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,

```
| ?- compile(user).
| :- dynamic foo/0.
| foo :- assertz(foo), fail.
| ^D
% user compiled in module user, 0.100 sec 2.56 bytes

yes
| ?- foo.  % The asserted clause is not found

no
| ?- foo.  % A later call does find it, however

yes
| ?-
```

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

```
| ?- 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

```
| ?- 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

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:

```
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, 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 152 and Section 9.4 [Indexing], page 309).

`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*.

`bb_get(`***:Key***`, `***?Term***`)`
: If a term is currently stored under *Key*, a copy of it is unified with *Term*. Otherwise, `bb_get/2` silently fails.

`bb_delete(`***:Key***`, `***?Term***`)`
: If a term is currently stored under *Key*, the term is deleted, and a copy of it is unified with *Term*. Otherwise, `bb_delete/2` silently fails.

`bb_update(`***:Key***`, `***?OldTerm***`, `***?NewTerm***`)`
: If a term is currently stored under *Key* and unifies with *OldTerm*, the term is replaced by a copy of *NewTerm*. Otherwise, `bb_update/3` silently fails. This predicate provides an atomic swap operation.

Please note: If the term being stored contains attributed variables (see Section 10.3 [lib-atts], page 337) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 63), those attributes are not stored. To retain the attributes, you can use `copy_term/3` (see Section 4.8.7 [ref-lte-cpt], page 111).

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

`assertz(`***:C***`)` ***ISO***
`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***`)` ***ISO***
`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*** ***ISO,declaration***
: 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***`)` ***ISO***
: 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 337) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 63), 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 111).

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 113).

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***, `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).
```

the call

```
| ?- setof(X, likes(X,Y), S).
```

might produce two alternative solutions via backtracking:

```
X = _872,
Y = beer,
S = [dick,harry,tom] ;

X = _872,
Y = cider,
S = [bill,jan,tom] ;

no
```

The call

```
| ?- setof((Y,S), setof(X,likes(X,Y),S), SS).
```

would then produce

```
Y = _402,
S = _417,
X = _440,
SS = [(beer,[dick,harry,tom]),(cider,[bill,jan,tom])] ;

no
```

4.13.2.1 Existential Quantifier

X ^ *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 `setof/3` and `bagof/3` is superfluous.

Variables occurring in *Generator* will not be treated as free if they are explicitly bound within *Generator* by an existential quantifier. An existential quantification is written:

```
Y^Q
```

meaning “there exists a *Y* such that *Q* is true”, where *Y* is some Prolog variable. For example:

```
| ?- setof(X, Y^likes(X,Y), S).
```

would produce the single result

```
X = _400,
Y = _415,
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=term(X,Y), setof(Z, A^foo(X,Y,Z), L).
```

will treat *X* and *Y* as if they are existentially quantified.

4.13.3 Collecting a Bag of Solutions

`bagof/3` is is exactly the same as `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.

4.13.3.1 Collecting All Instances

`findall/3` is a special case of `bagof/3`, where all free variables in the generator are taken to be existentially quantified. Thus the use of the operator `^` is avoided. Because `findall/3` avoids the relatively expensive variable analysis done by `bagof/3`, using `findall/3` where appropriate rather than `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.

4.13.4 Predicate Summary

?X* ^ *:P there exists an *X* such that *P* is provable (used in `setof/3` and `bagof/3`)

`bagof(`***?X***`,`***:P***`,`***-B***`)` ***ISO***
: *B* is the bag of instances of *X* such that *P* is provable

`findall(`***?T***`,`***:G***`,`***-L***`)` ***ISO***
`findall(`***?T***`,`***:G***`,`***?L***`,`***?R***`)`
: *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

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 lanaguage, 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 constitutent phrases of the string and their hierarchical relationships.

Context-free grammars (CFGs) consist of a series of rules of the form:

```
nt --> body.
```

where *nt* is a non-terminal symbol and 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 *body* is a possible form for a phrase of type *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

```
head --> 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, 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 constitutents 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 disjunction operator can also be written as ‘`|`’ (vertical-bar).)
- 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 168.
- 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 utilitze definte clause grammars:

STEPS:

1. Write a grammar, using `-->/2` to formulate rules.
2. Compile the file containing the grammar rules. The Load Predicates automatically translate the grammar rules into Prolog clauses.
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 75.
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]`. For example,

```
| ?- [grammar].
| ?- phrase(expr(Z), "-2+3*5+1").

Z = 14

| ?- phrase(expr(Z), "-2+3*5", Rest).

Z = 13,
Rest = [] ;

Z = 1,
Rest = "*5" ;

Z = -2,
Rest = "+3*5" ;

no
```

4.14.4 Semantics of Grammar Rules

Grammar rules are best explained in terms of an interpreter. The semantics of `phrase/3` 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, descending into its argument. Note that other built-in predicates are *not* treated this way.

lists
: Treated as terminal symbols.

curly brackets
: Treated as procedure calls.

callable terms with functor N/A
: A grammar rule with head functor *N/A* is looked up using the imaginary predicate `-->/2`, unified against, and its body is interpreted. If none exists, this is treated as a procedure call to a predicate *N/A+2*.

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 305). 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(Term, 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(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),
        S0 = S.                 % unification AFTER call
phrase(NT, M, S0, S) :-
        \+ \+(M:NT --> Rhs), !, % grammar rule exists?
        (M:NT --> Rhs),
        phrase(Rhs, M, 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 `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).                                          (A)
```

```
p(X, S0, S) :-
        q(X, S0, S).                                    (B)
```

```
p(A, B, C) :-
        q(A, B, C).                                     (C)
```

If there is more than one non-terminal on the right-hand side, as in (D) the corresponding input and output arguments are identified, translating into (E):

```
p(X, Y) --> q(X), r(X, Y), s(Y).                        (D)
```

```
p(X, Y, S0, S) :-                                       (E)
    q(X, S0, S1),
    r(X, Y, S1, S2),
    s(Y, S2, S).
```

Terminals are translated using the built-in predicate `=/2`. For instance, (F) is translated into (G):

```
p(X) --> [go, to], q(X), [stop].                        (F)
```

```
p(X, S0, S) :-                                          (G)
    S0 = [go,to|S1],
    q(X, S1, S2),
    S2 = [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], {integer(X), X > 0}, q(X).                (H)
```

```
p(X, S0, S) :-                                                (I)
    S0 = [X|S1],
    integer(X),
    X > 0,
    q(X, S1, S).
```

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), [not] --> [aint].                                      (J)
```

```
is(N, S0, S) :-                                               (K)
    S0 = [aint|S1],
    S = [not|S1].
```

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):

```
args(X, Y) -->                                                (L)
        dir(X), [to], indir(Y) |
        indir(Y), dir(X).
```

```
args(X, Y, S0, S) :-                                          (M)
    (   dir(X, S0, S1),
        S1 = [to|S2],
        indir(Y, S2, S)
    |   indir(Y, S0, S1),
        dir(X, S1, S)
    ).
```

In order to look at these translations, declare the grammar rules dynamic and use `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):

```
 :- dynamic args/4.
```

4.14.5 Summary of Predicates

:Head `-->` ***:Body***
: A possible form for *Head* is *Body*

`expand_term(+`***T***`,-`***X***`)` ***hookable***
: term *T* expands to term *X* using `user:term_expansion/6` or grammar rule expansion

`phrase(`***:P***`, -`***L***`)`
`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*, +*Tokens1*, -*Term2*, -*Layout2*, -*Tokens2*) ***hook***

Overrides 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 raises an exception. For example, if a built-in predicate detects an argument of the wrong type, it raises a `type_error` exception. The manual page description of each built-in predicate lists the kinds of exceptions that can be raised by that built-in predicate.

The default effect of raising 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 is a/2
```

is executed somewhere in a program you get

```
! Type error in argument 2 of is/2
! expected number, but found a
! goal:  A is a/2

| ?-
```

Particular things to notice in this message are:

'`!`'
This character indicates that this is an error message rather than a warning[4] or informational message.

'`Type Error`'
This is the ***exception class***. Every exception raised by the system is categorized into one of a small number of classes. The classes are listed in Section 4.15.4 [ref-ere-err], page 175.

'`goal:`'
The goal that caused the exception to be raised.

4.15.2 Raising Exceptions

You can raise exceptions from your own code using one of the two equivalent built-in predicates:

```
raise_exception(+ExceptionCode)
throw(+ExceptionCode)
```

[4] The difference between an error (including exceptions) and a warning: A ***warning*** is issued if Prolog detects a situation that is likely to cause problems, though it is possible that you intended it. An ***error***, however, indicates that Prolog recognizes a situation where it cannot continue.

The argument to this predicate is the ***exception term***; it is an arbitrary non-variable term of which the principal functor indicates the exception class. You can use the same exception classes as the system (see Section 4.15.4 [ref-ere-err], page 175), or you can use your own exception classes.

See Section 10.27 [lib-types], page 469 for an alternative interface to raising error exceptions, which tries to include line number information for source-linked debugging.

Error messages like the one shown above 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 extended, as described in Section 4.16 [ref-msg], page 185, so that you can have appropriate error messages printed corresponding to your own exception classes.

Please note: If the exception term contains attributed variables (see Section 10.3 [lib-atts], page 337) or suspended goals (see Section 4.2.4 [ref-sem-sec], page 63), 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 111).

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 two ways to do this:

- Trap exceptions to a particular goal by calling `on_exception/3` as described in Section 4.15.3.1 [ref-ere-hex-pgo], page 174.
- 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 175.

4.15.3.1 Protecting a Particular Goal

The built-in predicate `on_exception/3` enables you to handle exceptions to a specific goal:

`on_exception(`***?ExceptionCode***`, `***:ProtectedGoal***`, `***:Handler***`)`

ProtectedGoal is executed. If all goes well, it will behave just as if you had written *ProtectedGoal* without the `on_exception/3` wrapper. If an exception is raised while *ProtectedGoal* is running, 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***, *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 raised with the *ExceptionCode* argument. If this unification succeeds, *Handler* will be executed as if you had written

ExceptionCode=<the actual exception term>,
Handler

If this unification fails, Prolog will keep searching up the ancestor list looking for another exception handler. If it reaches Prolog's top level (or a break level) without having found a call to `on_exception/3` with a matching *ExceptionCode*, 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, the call to `on_exception/3` is also determinate.

The ***ProtectedGoal*** is logically *inside* the `on_exception/3` form, but the ***Handler*** is *not*. If an exception is raised inside the ***Handler***, this `on_exception/3` form will *not* be reactivated. If you want an exception handler that protects itself, you have to program it, perhaps like this:

```
recursive_on_exception_handler(Err, Goal, Handler) :-
    on_exception(Err, Goal,
        recursive_on_exception_handler(Err, Handler, Handler)).
```

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, the ***Handler*** should examine the exception and pass it on if it is not relevant to the current invocation.

In a development system, any previously uncaught exception is caught and an appropriate error message is printed before returning to the top level. In recursive calls to Prolog from C, uncaught exceptions are returned back to C instead. The printing of these and other messages in a development system is handled by the predicate `print_message/2` (see Section 4.16 [ref-msg], page 185).

`catch(`***`ProtectedGoal`***`, `***`ExceptionCode`***`, `***`Handler`***`)` is the same as `on_exception(`***`ExceptionCode`***`, `***`ProtectedGoal`***`, `***`Handler`***`)`.

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 `on_exception/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.

4.15.4 Error Classes

Exceptions raised by the Prolog system are called errors. The set of exception classes used by the system has been kept small. Here is a complete list:

Instantiation Error
: An input argument is insufficiently 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 raised by the built-in predicates is:

```
error(ISO_Error, SICStus_Error)
```

where *ISO_Error* is the error term prescribed by the ISO Prolog standard, while *SICStus_Error* is the part defined by the standard to be implementation dependent. In the case of SICStus Prolog, this is the SICStus error term, which normally contains additional information, such as the goal and the argument number causing the error. Arguments are numbered from 1 upwards.

The list below itemizes the error terms, showing the *ISO_Error* and *SICStus_Error* form of each one, in that order. Note that the SICStus and ISO error terms do not always belong to the same error class, and 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 *AGoal* into a *Goal* proper and a *SourceInfo* descriptor term, indicating the source position of the goal, use:

```
?- goal_source_info(AGoal, Goal, SourceInfo).
```

The reference page gives details about the *SourceInfo* format.

`instantiation_error`
`instantiation_error(Goal,ArgNo)`
: *Goal* was called with insufficiently instantiated variables.

`type_error(`***TypeName***`,`***Culprit***`)`
`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***`)`
`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***`)`
`existence_error(`***Goal***`,`***ArgNo***`,`***ObjectType***`,`***Culprit***`,`***Reserved***`)`
: Something does not exist as indicated by the arguments. If the `unknown` Prolog flag is set to `error`, this error is raised with *ArgNo* set to 0 when an undefined predicate is called.

`permission_error(`***Operation***`,`***ObjectType***`,`***Culprit***`)`
`permission_error(`***Goal***`,`***Operation***`,`***ObjectType***`,`***Culprit***`,`***Reserved***`)`
: The *Operation* is not permitted on *Culprit* of the *ObjectType*.

`context_error(`***ContextType***`,`***CommandType***`)`
`context_error(`***Goal***`,`***ContextType***`,`***CommandType***`)`
: The *CommandType* is not permitted in *ContextType*.

`syntax_error(`***Message***`)`
`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`. In the former case this error is raised only if the `syntax_errors` flag is set to `error`.

`evaluation_error(`***ErrorType***`,`***Culprit***`)`
`evaluation_error(`***Goal***`,`***ArgNo***`,`***ErrorType***`,`***Culprit***`)`
: An incorrect arithmetic expression was evaluated.

`representation_error(`***ErrorType***`)`
`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***`)`
`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***`)`
`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`
`system_error(`***Message***`)`
An error occurred while dealing with the operating system.

Most exception terms include a copy of the *Goal* that raised 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 raise 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

```
instantiation_error(Goal, 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 is Y+1
```

where *Y* is uninstantiated raises the exception

```
instantiation_error(_2298 is _2301+1,2)
```

because the second argument to `is/2` contains a variable.

4.15.4.2 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 `atom/1` and `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 *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 *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 isn't a subtle matter of being off by one.

Most built-in predicates check all their input arguments for type errors.

The *SICStus_Error* term associated with a type error is

```
type_error(Goal, ArgNo, TypeName, Culprit)
```

ArgNo *Culprit* occurs somewhere in the *ArgNo*th argument of *Goal*.

TypeName
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.

Culprit is the actual term being complained about: *TypeName*(*Culprit*) should be false.

For example, suppose we had a predicate

```
date_plus(NumberOfDays, Date0, Date)
```

which held when *Date0* and *Date* were `date(Y,M,D)` records and *NumberOfDays* was the number of days between those two dates. You might see an error term such as

```
type_error(/* Goal     */ date_plus(27, date(18,mar,11), _235),
           /* Argno    */ 2,
           /* TypeName */ integer,
           /* Culprit  */ mar
```

4.15.4.3 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 `open/3` is supposed to be an atom that represents a valid mode for opening a file, such as `read` or `write`. If a number or a compound term is given instead, that is a type error. If an atom is given that is not a valid mode, 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

```
domain_error(Goal, ArgNo, DomainName, 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, it might use `between(1,99)` as the *DomainName*. With respect to the `date_plus` example under **Type Errors**, if the month had been given as 13 it would have passed the type test but would raise a domain error.

For example, the goal

```
open(somefile,rread,S)
```

raises the exception

```
domain_error(open(somefile,rread,_2490),2,'i/o mode',rread,'')
```

The *Message* argument is used to provide extra information about the problem.

4.15.4.4 Evaluation Errors

An evaluation error occurs when an incorrect arithmetic expression was evaluated. The *SICStus_Error* term associated with an evaluation error is

```
evaluation_error(Goal, ArgNo, TypeName, Culprit)
```

This has the same arguments as a type error.

4.15.4.5 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:

```
functor(T, f, 1000)            % maximum arity is 255
atom_codes(X, L)               % if length of L > 65535
```

are all representation errors. Floating-point overflow is a representation error.

The *SICStus_Error* term for a representation error is

```
representation_error(Goal, ArgNo, 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.

4.15.4.6 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. A less obvious example: reading past the end of file marker in a stream is regarded as asking for an object (the next character) that does not exist.

The *SICStus_Error* term associated with an existence error is

```
existence_error(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 raise the exception

```
existence_error(see('../brother/niece'),
        1, file, '/usr/stella/parent/brother/niece',
        errno(20))
```

where the *Message* encodes the system error `ENOTDIR` (some component of the path is not a directory).

As a general rule, if *Culprit* was provided in the goal as some sort of context-sensitive name, the Prolog system will try to resolve it to an absolute name, as shown here, so that you can see whether the problem is just that the name was resolved in the wrong context.

4.15.4.7 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 isn't 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 attempts 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

`permission_error(`***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.

4.15.4.8 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.

The *SICStus_Error* term associated with a context error is

`context_error(`***Goal, ContextType, CommandType***`)`

ContextType
the context in which the command was attempted.

CommandType
the type of command that was attempted.

4.15.4.9 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, that is a consistency error.

The *SICStus_Error* term associated with a consistency error is

```
consistency_error(Goal, Culprit1, Culprit2, Message)
```

Culprit1 One of the conflicting values/operations.

Culprit2 The other conflicting value/operation..

Message Additional information, or `0`, or `''`.

4.15.4.10 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 `read/1` and its variants.

The *SICStus_Error* term associated with a syntax error is

```
syntax_error(Goal, Position, Message, Left, Right)
```

where *Goal* is the goal in question, *Position* identifies the position in the stream where reading started, and *Message* describes the error. Left and right are lists of tokens before and after the error, respectively.

Note that the *Position* is where reading started, not where the error *is*.

`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, a syntax error occurs. Thus, in the case of syntax errors, `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 raised. The behavior can be controlled via the `syntax_errors` Prolog flag as follows. The following values are possible:

`quiet` When a syntax error is detected, nothing is printed, and `read/1` just quietly fails.

`dec10` This provides compatibility with other Prologs: when a syntax error is detected, a syntax error message is printed on `user_error`, and the `read` is repeated. This is the default.

`fail` This provides compatibility with other Prologs. When a syntax error is detected, a syntax error message is printed on `user_error`, and the `read` then fails.

`error` When a syntax error is detected, an exception is raised.

4.15.4.11 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 doesn't 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

```
resource_error(Goal, 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.

Message an operating-system-specific message. Usually it will be `errno(ErrNo)`.

4.15.4.12 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

```
system_error(Message)
```

where *Message* is not further specified.

4.15.5 An Example

Suppose you want a routine that is given a filename and a prompt string. This routine is to open the file if it can; otherwise it is to prompt the user for a replacement name. If the user enters an empty name, 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.)

```
:- use_module(library(prompt), [
        prompted_line/2
   ]).

open_output(FileName, Prompt, Stream) :-
        on_exception(Error,
            open(FileName, write, Stream),
            (   file_error(Error) ->
                print_message(warning, Error),
                retry_open_output(Prompt, Stream)
            ;   raise_exception(Error)
            )).

file_error(domain_error(open(_,_,_), 1, _, _, _)).
file_error(existence_error(open(_,_,_), 1, _, _, _)).
file_error(permission_error(open(_,_,_), _, _, _, _)).

retry_open_output(Prompt, Stream) :-
        prompted_line(Prompt, Chars),
        atom_chars(FileName, Chars),
        FileName \== '',
        open_output(FileName, Prompt, Stream).
```

What this example does *not* catch is as interesting as what it does. All instantiation errors, type errors, context errors, and evaluation errors are re-raised, as they represent errors in the program.

As the previous example shows, you generally do not want to catch *all* exceptions that a particular goal might raise.

4.15.6 Interrupting Execution

There exist more drastic means of interrupting the normal control flow. To invoke a recursive top-level, use:

```
?- break.
```

To exit from Prolog, use:

```
?- halt.
```

To exit from Prolog with return code *Code*, use:

```
?- halt(Code).
```

To abort the execution of the current query and return to the top-level, use:

```
?- abort.
```

> **Please note:** `halt/[0,1]` and `abort/0` are implemented by raising 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.

4.15.7 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)` ***ISO***
: specify a handler *H* for any exception *E* arising in the execution of the goal *P*

`user:error_exception(+Exception)` ***hook***
: *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)` ***ISO***
: raise exception *E*

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:

```
print_message(Severity, 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:

`error` `'! '` for error messages

`warning`	`'* '`	for warning messages
`informational`	`'% '`	for informational messages
`help`	`''`	for help messages
`query`	`''`	for query texts (see Section 4.16.3 [Query Processing], page 189)
`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, 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, 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, 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, which do not fit the default message generation schemes. The latter can also be achieved by adding new clauses to the extendible `'SU_messages':generate_message/3` predicate.

If both the hook and the default method refuses to handle the message, 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.

`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***`)` ***extendible***

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:

```
ask_query(QueryClass, Query, Help, 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), 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 extendible predicate

```
'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:

```
| ?- 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 185). 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 extendible predicate

`'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 ***no***⟨RET⟩, ***RawInput*** becomes `" no" = [32,110,111]`.

Mapping phase: By default, the mapping phase is implemented by the extendible predicate

'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, it is printed. Subsequently, if requested by the *FailureMode* parameter, 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, 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 extendible 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, 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, 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 extendible, 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, 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`^term(`***Term***`,`***Options***`)`
: 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-layout 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, *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, 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 203.

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"])`	`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.

`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, 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***) ***extendible***

Predefined query class characteristics table.

'SU_messages':query_abbreviation(+***QueryClass***, -***Prompt***, -***Pairs***) ***extendible***

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***) ***extendible***

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***) ***extendible***

Predefined query map methods.

4.16.5 Predicate Summary

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***) ***extendible***

determines the mapping from a message term into a sequence of lines of text to be printed

user:generate_message_hook(+***M***,***?S0***,***?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)` ***extendible***
: 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, it is assumed that the query has been processed and nothing further is done.

`'SU_messages':query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)` ***extendible***
: 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)` ***extendible***
: 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)` ***extendible***
: 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.

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 277.

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 :

```
            *--------------------------------------*
   Call     |                                      |    Exit
---------> +  descendant(X,Y) :- offspring(X,Y).  + --------->
            |                                      |
            |  descendant(X,Z) :-                  |
<--------- +     offspring(X,Y), descendant(Y,Z). + <---------
   Fail     |                                      |    Redo
            *-------------------+------------------*
                                |
<------------------------------+
   Exception
```

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 `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 173. 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, 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` ***development***

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.

`zip` ***development***

Same as `debug/0`, except no debugging information is being collected, and so is almost as fast as running with the debugger switched off.

`trace` ***development***

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 203. In particular, you can just type ⟨RET⟩ to creep (or single-step) into your program. If you continue to creep through your program 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 138). You will notice that the debugger stops at all ports. However, if this is not what you want, the next predicate gives full control over the ports at which you are prompted.

`leash(+`***Mode***`)` ***development***

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 208. 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(full).`
Same as `leash([call,exit,redo,fail,exception]).`.

`leash(half).`
Same as `leash([call,redo]).`.

`leash(none).`
Same as `leash([]).`.

The initial value of ***Leashing Mode*** is `[call,exit,redo,fail,exception]` (full leashing).

`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).

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 208.

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 spec *Spec*.

Examples:

```
| ?- spy [user:p, m:q/[2,3]].
| ?- spy m:[p/1, q/1].
```

If you set some spypoints when the debugger is switched off, it will be automatically switched on, entering zip mode.

`nospy` ***:Spec*** ***development***

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 208).

`nospyall` ***development***

Removes all the spypoints that have been set, including the conditional and generic ones.

The commands available when you arrive at a spypoint are described later. See Section 5.5 [Debug Commands], page 203.

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 185. This allows you to trace programs while they are performing file I/O. The basic format is as follows:

```
N S    23     6 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`, 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.

The number following this 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, *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 203). If this particular port is unleashed, 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 in-line are not traced.

Block and Unblock ports are exceptions to the above debugger message format. A message

```
S -  -  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 63). 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 -  -  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⟩).

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.

```
<cr>   creep            c      creep
 l     leap             z      zip
 s     skip             s <i>  skip i
 o     out              o <n>  out n
 q     q-skip           q <i>  q-skip i
 r     retry            r <i>  retry i
 f     fail             f <i>  fail i
 j<p>  jump to port     j<p><i>jump to port i
 d     display          w      write
 p     print            p <i>  print partial
 g     ancestors        g <n>  ancestors n
 t     backtrace        t <n>  backtrace n
 &     blocked goals    & <n>  nth blocked goal
 n     nodebug          =      debugging
 +     spy this         *      spy conditionally
 -     nospy this       \ <i>  remove brkpoint
 D <i> disable brkpoint E <i>  enable brkpoint
 a     abort            b      break
 @     command          u      unify
 e     raise exception  .      find this
 <     reset printdepth < <n>  set printdepth
 ^     reset subterm    ^ <n>  set subterm
 ?     help             h      help
```

c
⟨RET⟩ *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 199), the user is prompted for further interaction. Otherwise, it continues creeping. If leashing is off, creep is the same as *leap* (see below) except that a complete trace is printed on the standard error stream.

l *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.

z *zip* is like *leap*, except no debugging information is being collected while zipping, resulting in significant savings in memory and execution time.

s *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, no message at all will appear until control returns. This includes calls to predicates with spypoints set; they will be masked out during the skip. No debugging information is being collected while skipping.

If you supply an integer argument, 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.

o *out* is a shorthand for skipping to the Exit or Fail port of the immediate ancestor goal. If you supply an integer argument *n*, it denotes skipping to the Exit or Fail port of the *n*th ancestor goal.

q *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*.

r *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, this 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.

f *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, this 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.

j<p> *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, this 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 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, this 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*, 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.

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*, 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*.

& *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 63). The goals are enumerated from 1 and up. If you supply an integer *n*, only that goal will be printed. Each entry is preceded by the goal number followed by the variable name.

n *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

```
spy(Func, Conditions)
```

goal, where *Func* is the predicate spec of the current invocation. For `spy/2`, see Section 5.7 [Breakpoint Predicates], page 237.

- *nospy this* removes all spypoints applicable to the current goal. Equivalent to `nospy` ***Func***, where *Func* is the predicate spec 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 237).

D *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 237).

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 237).

. *nd 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 '`| :- `' 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***, *Pattern* is unified with the current goal and *Body* is executed.

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***, *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 a list of arguments, the arguments are applied from left to right.

?
h *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 237.

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 239 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 237 and in Section 5.9 [Breakpoint Conditions], page 241.

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 spec*. It will return the *breakpoint identi er* (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 spec, 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:

```
| ?- add_breakpoint([pred(foo/2),line('/myhome/bar.pl',123)], _).
% Conditional spypoint for user:foo/2 added, BID=1
```

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 spec 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 spec 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 202. 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 203), 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, 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 `[pred(foo/2), \+port(fail)]` will hold for all ports of `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 you should get rid of the old breakpoints (e.g. using `?- nospyall.`) before you enter a new one.

The `goal(...)` test is a generalization of the `pred(...)` test, as it allows us to check the arguments of the invocation. For example:

```
| ?- add_breakpoint(goal(foo(1,_)), _).
% Conditional spypoint for user:foo/2 added, BID=1
```

The `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 `goal(G)` condition is thus equivalent to the `subsumes(G,`***`CurrentGoal`***`)` test (`subsumes/2` is defined in `library(terms)`, see Section 10.24 [lib-terms], page 465).

In the above example the debugger will stop if `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 `goal` test, and then put further constraints on these variables using the `true` condition:

```
| ?- add_breakpoint([goal(foo(X,_)),true(X>1)], _).
% Conditional spypoint for user:foo/2 added, BID=1
```

Here the first test, `goal`, specifies that we are only interested in invocations of `foo/2`, and names the first argument of the goal as `X`. The second, the `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 `foo/2` is *greater* than `1`. Generally, an arbitrary Prolog goal can be placed inside the `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 (`\+ \+ (...)`). This ensures that the test part has no effect on the variables of the current goal.

Both the `pred` and the `goal` tests may include a module name. In fact, the first argument of `add_breakpoint` is module name expanded, and the (explicit or implicit) module name of this argument is then inherited by default by the `pred`, `goal`, and `true` tests. Notice the module qualification inserted in front of the breakpoint spec of the last example, as shown in the output of the `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 spec 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 138), 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 '`*`' debugger command. This automatically includes `pred(foo/2)` among the conditions and displays the prompt '`Placing spypoint ... with conditions:`', requesting further ones. You enter here the `inv` test with the invocation number in question, resulting in a breakpoint with the `[pred(foo/2),inv(86)]` conditions. If you restart the original top-level goal in debug mode, the debugger immediately positions you at the invocation with the specified number.

Note that when the debugger executes a ***skip*** or a ***zip*** command, no procedure boxes are built. Consequently, the invocation and depth counters are not incremented. If ***skip*** and/or ***zip*** commands were used during the first execution, the suspicious invocation gets an invocation number higher than 86 in the second run. Therefore it is better to supply the `inv(I),true(I>=86)` condition to the '`*`' debugger command, which will bring you to the first call of `foo/2` at, or after invocation number 86 (which still might not be the suspicious invocation).

In the examples, the `inv` test was used both with a numeric and a variable argument (`inv(86)` and `inv(I)`). This is possible because the debugger *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, you can do the following:

```
| ?- 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) ?
```

The above breakpoint spec will cause the debugger to stop at the first invocation of `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 `/home/bob/myprog.pl` contains the following Prolog program:

```
                                              % /home/bob/myprog.pl
```

```
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`, you can use the following breakpoint:

```
| ?- add_breakpoint([pred(q/2),line('/home/bob/myprog.pl',6)], _).
% Conditional spypoint for user:q/2 added, BID=1
```

Generally, the test `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 `line/1` and `file/1` tests, see below) require the presence of source information: the file in question had to be consulted or compiled with the `source_info` Prolog flag switched on (i.e. set to `on` or `emacs`).

If e.g. `q/2` is called only from a single file, the file name need not be mentioned and a `line/1` test suffices: `line(6)`. On the other hand, if we are interested in all invocations of a predicate within a file, we can omit the line number and use the `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 `parent_pred(Pred)` unifies the module name expanded `Pred` with a predicate spec (of form ***Module*:*PredName*/*Arity***) identifying the predicate in which the current invocation resides. The test `parent_pred(Pred,N)` will additionally unify `N` with the serial number of the clause containing the current goal.

For example, assuming the above `myprog.pl` file is consulted, the breakpoint below will cause the execution to stop when the call of `is/2` in the second clause of `q/2` is reached:

```
| ?- 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 inter-
preted 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, 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`:

```
| ?- add_breakpoint([pred(foo/2),(call;exit)], _).
```

```
| ?-
add_breakpoint([pred(foo/2),port(P),true((P=call;P=exit(_)))], _).
```

In both cases we have to use disjunction. In the first example we have a disjunctive breakpoint condition of the two simple tests `port(call)` and `port(exit)` (with the `port` functor omitted). In the second case the disjunction is inside the Prolog test within the `true` test.

Notice that the two examples refer to the Exit port differently. When you use `port(P)`, where `P` is a variable, then, at an exit port, `P` will be unified with either `exit(nondet)` or `exit(det)`, depending on the determinacy of the exited predicate. However, for convenience, the test `port(exit)` will also succeed at Exit ports. So in the first example above, `exit` can be replaced by `exit(_)`, but the `exit(_)` in the second can not be replaced by `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:

```
| ?- add_breakpoint([pred(foo/2),fail], _).
```

The intention here is to have the debugger stop at only the Fail port of `foo/2`. This is very useful if `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 `goal` or `pred` test. Such breakpoints are called *speci c*, as opposed to *generic* ones.

Generic breakpoints are the ones that do not specify a concrete predicate. This can happen when the breakpoint spec does not contain `goal` or `pred` tests at all, or their argument is not sufficiently instantiated. Here are some examples of generic breakpoints:

```
| ?- 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
```

The first breakpoint will stop at all calls in line 6 of the given file, the second at all calls of a predicate `foo`, irrespective of the number of arguments, while the third one will stop at any predicate with `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 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, 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 spec, much like in spy/1, and the second argument is a breakpoint spec. spy/2 will expand the first argument to one or more predicate specs, and for each of these will create a breakpoint, with a pred condition added to the *test* part of the supplied breakpoint spec. 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.

5.6.5 Breakpoint Actions

The action part of a breakpoint spec 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 spec 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`, 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:

```
debugging    leashing            |        Action variables
mode         mode                |  show       command     mode
---------------------------------|--------------------------------
trace        at leashed port     |  print      ask         trace
                                 |
trace        at unleashed port   |  print      proceed     trace
                                 |
debug        -                   |  silent     proceed     debug
                                 |
zip          -                   |  silent     flit        zip
```

It then considers each breakpoint, most recent first, until it finds a breakpoint whose test part succeeds. If such a breakpoint is found, 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, 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`, 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 199. 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 199.

If an action part is nonempty, but it does not set the action variables, 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 spec with an empty test part can be written `-`*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, 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:

```
| ?- spy(foo/2,-[silent,proceed,
                 (   call -> inv(Inv), skip(Inv)
                 ;   true
                 )]).
```

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.

```
| ?- 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 247). Setting `command` to `exception(`***`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 spec.)

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], _).
% Conditional advice point for user:foo/2 added, BID=1
% advice
| ?- foo(4, Y).
Y = 3
% advice
| ?- foo(9, Y).
        3      3 Exit: foo(7,13) ? n
        2      2 Exit: foo(8,21) ?
```

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, 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:

```
| ?- add_breakpoint(pred(foo/2)
                    -[exit,goal(foo(X,Y)),\+true(Y-X<3),leash], _).
% The debugger will first zip -- showing spypoints (zip)
% Conditional spypoint for user:foo/2 added, BID=1
% zip
| ?- foo(4, X).
X = 3
% zip
| ?- foo(9, X).
 *      3      3 Exit: foo(7,13) ? z
 *      2      2 Exit: foo(8,21) ?
```

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, 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, 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, `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, the prescribed action is performed immediately, without executing the spypoint selection process. Otherwise, if `command = proceed`, 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, 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 spec 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 spec, 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 spec. 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 227).

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.

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, we get this:

```
| ?- 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 spec 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`, the test part is extended by a `pred` condition, as exemplified above. Earlier we described some pre-processing steps that the spec goes through, such as moving the module qualification of the spec 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`.

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, 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 `between(`***N;M;I***`)` enumerates all integers such that ***N*** ***I*** ***M***.

```
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.
(...)

?*     41     11 Exit: foo(2,1) ? @
| :- exited_goals.
    26 foo(3,2)
    28 bar(3,1,1)
    31 foo(2,1)
    33 bar(2,1,0)
    36 foo(1,1)
    37 foo(0,0)
    39 foo(1,1)
    41 foo(2,1)
    43 bar(2,1,0)
    46 foo(1,1)
    47 foo(0,0)
?*     41     11 Exit: foo(2,1) ?
```

Note that similar output can be obtained by entering a new break level and calling `exited_goals` from within an `execution_state/2`:

```
% 1
| ?- execution_state(break_level(0), true(exited_goals)).
```

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 `foo/2` that are somewhere within `bar/1`, possibly deeply nested. The following two breakpoints achieve this effect:

```
| ?- 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
```

We added an advice-point for `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 `?- spy(bar/1, -proceed)` the debugger will not stop at the call port of `bar/1` in trace mode. Note that it is perfectly all right to create an advice-point using `spy/2`, although this is a bit of terminological inconsistency.

Further examples of accessing past debugger states can be found in `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 `private(`***Priv***`)` unifies *Priv* with the private information associated with the break level, while the test `goal_private(`***GPriv***`)` unifies *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 `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.

```
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) ? @
| :- append(_,_,L)^memory(L).
        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 `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 `P`, and calls `memberchk/2` (see Section 10.11 [lib-lists], page 367), with the term to be remembered, wrapped in `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 `memory/1` is called for the first time, the private field gets instantiated to `[myterm(Term)|_]`. If later you call `memory/1` with an uninstantiated argument, 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 203). 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 10.36 [lib-fdbg], page 562) 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 SICStus Prolog 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.

```
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 spec, as the *Macro* argument. If the hook succeeds, the term returned in the *Body* argument is substituted for the original test or action. Note that *Body* can not 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.

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.

```
:- 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.

```
user:breakpoint_expansion(invisible,
                          [true(invisible(Settings)),Settings]).

invisible([proceed,silent,NewMode]) :-
        execution_state([mode(M),port(P),inv(Inv),goal_private(GP)]),
        memberchk(mymode(MM), GP),
        (   P == call -> MM = M, NewMode = skip(Inv)
        ;   P = exit(_) -> NewMode = MM
        ;   NewMode = M
        ).
```

The second hook related to breakpoints is `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. *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 189). If the hook succeeds, it should return in *Actions* an action part to be evaluated as the result of the command.

If you want to redefine an existing debugger command, you should study `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, it suffices to know that unrecognized debugger commands are returned as `unknown(Line,Warning)`. Here, `Line` is the code-list typed in, with any leading layout removed, and `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:

```
:- multifile user:debugger_command_hook/2.
user:debugger_command_hook(unknown([0'S|_],_), Actions) :-
        execution_state([port(P),inv(I)]),
        Actions = [Mode,proceed,silent],
        (   P = call -> Mode = skip(I)
        ;   P = redo -> Mode = skip(I)
        ;   Mode = trace
        ).
```

Note that the `silent` action is needed above; otherwise, the trace message will be printed a second time, before continuing the execution.

`library(debugger_examples)` contains some of the above hooks, as well as several others.

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`:

```
:- 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, it stores a term `hide_exit(ground)` in the `goal_private` attribute of the invocation.

```
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, it issues a `hide` action, which is a breakpoint macro expanding to `[silent,proceed]`.

```
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, 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).
| ^D
% 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 114) 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(:, ?).
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(:, ?).
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, 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***`)` ***development***

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 241, 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, a consistency error is signalled. Module name expansion is performed for certain tests, as described below.

Both the original and the transformed breakpoint spec is recorded by the debugger. The original is returned in `current_breakpoint/5`, while the transformed spec 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.

`spy(`***:PredSpec***`, `***:Spec***`)` ***development***

Adds a conditional spypoint with a breakpoint spec formed by adding `pred(`***Pred***`)` to the test part of *Spec*, for each predicate *Pred* designated by the generalized predicate spec *PredSpec*.

`current_breakpoint(`***:Spec***`, `***?BID***`, `***?Status***`, `***?Kind***`, `***?Type***`)` ***development***

There is a breakpoint with breakpoint spec *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,

`remove_breakpoints(+`***BIDs***`)` ***development***
`disable_breakpoints(+`***BIDs***`)` ***development***
`enable_breakpoints(+`***BIDs***`)` ***development***

Removes, disables or enables the breakpoints with identifiers specified by *BIDs*. *BIDs* can be a number, a list of numbers or one of the atoms: `all`, `debugger`, `advice`. The atoms specify all breakpoints, debugger type breakpoints and advice type breakpoints, respectively.

`execution_state(`***:Tests***`)` ***development***

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 `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 241) occurs in a call of `execution_state/1` from outside the debugger.

`execution_state(+`***FocusConditions***`, `***:Tests***`)` ***development***

Tests are satisfied in the state of the execution pointed to by *FocusConditions* (see Section 5.9.7 [Past States], page 247). An exception is raised if there is a non-backtraced test among *Tests*.

Note that the predicate arguments holding a breakpoint spec (*Spec* or *Tests* above) are subject to module name expansion. The first argument within simple tests `goal(_)`, `pred(_)`, `parent_pred(_)`, `parent_pred(_,_)`, `ancestor(_,_)`, and `true(_)` will inherit the module name from the (module name expanded) breakpoint spec/tests predicate argument, if there is no explicit module qualification within the simple test. Within the `proceed(`***Old***`,`***New***`)` and `flit(`***Old***`,`***New***`)` command value settings, *Old* will get the module name from the `goal` or `pred` condition by default, while *New* from the whole breakpoint spec argument.

The following hook predicate can be used to customize the behavior of the interactive debugger.

`debugger_command_hook(+`***DCommand***`,`***?Actions***`)` ***hook,development***
`user:debugger_command_hook(+`***DCommand***`,`***?Actions***`)`

This predicate is called for each debugger command SICStus Prolog reads in. The first argument is the abstract format of the debugger command *DCommand*, as returned by the query facility (see Section 4.16.3 [Query Processing], page 189). If it succeeds, *Actions* is taken as the list of actions (see Section 5.9.6 [Action Conditions], page 246) to be done for the given debugger command. If it fails, the debugger command is interpreted in the standard way.

Note that if a line typed in in response to the debugger prompt can not be parsed as a debugger command, `debugger_command_hook/2` is called with the term `unknown(Line,Warning)`. Here, `Line` is the code-list typed in, with any leading layout removed, and `Warning` is a warning message. This allows the user to define new debugger commands, see Section 5.6.10 [Hooks Related to Breakpoints], page 231 for an example.

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 spec 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, the breakpoint search is said to have completed successfully, otherwise it is said to have failed.

If a breakpoint has been selected, its action part is evaluated, normally setting some debugger action variables. If the action part fails, 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, `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`, 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 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, 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 217 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, `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`, 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`***`)`, no procedure box is required by the spypoint part. If the advice-point does require the building of a procedure box, 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 247.

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.

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 enquire 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`, 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 the `subsumes(`***MGoal***`,`***CurrentGoal***`)` test (`subsumes/2` is defined in `library(terms)`, see Section 10.24 [lib-terms], page 465).

`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 230.

`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 *I*th 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 predicate in the same way as the `pred` test does, see the notes in the previous section (Section 5.9.1 [Goal Tests], page 242).

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, 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 197. 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:

```
add_breakpoint(Tests-[port(Port),Actions], BID).
```

`bid(`***BID***`)` The breakpoint being examined has a breakpoint identifier *BID*. (*BID* = `none` if no breakpoint was selected.)

`mode(`***Mode***`)`
: *Mode* is the value of the `mode` variable, which normally reflects the current debugger mode.

`command(`***Command***`)`
: *Command* is the value of the `command` variable, which is the command to be executed by default, if the breakpoint is selected.

`show(`***Show***`)`
: *Show* 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 `mode(trace)`, if it occurs in the tests, *checks* if the current debugger mode is `trace`. On the other hand, if the same term occurs within the action part, it *sets* the debugger mode to `trace`.

To support the querying of the action variables in the action part, the following breakpoint condition is provided:

`get(`***ActVar***`)`
: Equivalent to *ActVar*, where this is an action variable test, i.e. one of the terms `mode(`***Mode***`)`, `command(`***Command***`)`, `show(`***Show***`)`. It has this meaning in the action part as well.

For the `port`, `mode`, `command` and `show` conditions, the condition can be replaced by its argument, if that is not a variable. For example the condition `call` can be used instead of `port(call)`. Conditions matching the terms listed above as valid port values will be converted to a `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 247.

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.

`break_level(`***N***`)`
: We are at (or focused on) break level *N* (*N* = 0 for the outermost break level).

`max_inv(`***MaxInv***`)`
: The last invocation number used within the current break level is *MaxInv*. Note that this invocation number may not be present in the backtrace (because the corresponding call exited determinately).

`private(`***Priv***`)`
: The private information associated with the break level is *Priv*. Similarly to `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 *Priv* be used as an open ended list, see Section 5.6.9 [Storing User Information in the Backtrace], page 230.

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 `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 *Cond* is true, i.e. `once(`***Cond***`)` is executed and the condition is satisfied if and only if this completes successfully. If an exception is raised during execution, an error message is printed and the condition fails.

  The substitutions done on executing *Cond* are carried out. *Cond* is subject to module name expansion. If used in the test part of spypoint conditions, 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`

`[]` A condition that is always true. Useful e.g. in conditionals.

`false` A condition that is always 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(`***Mode***`)`
: Set the debugger mode to *Mode*.

`command(`***Command***`)`
: Set the command to be executed to *Command*.

`show(`***Show***`)`
: Set the show method to *Show*.

The values admissible for *Mode*, *Command* and *Show* are described in Section 5.9.9 [Action Variables], page 247.

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 237). Alternatively, a list containing such terms can be used. If a given condition occurs multiple times, only the last one is considered. The order of conditions within the list does not matter.

`break_level(`***BL***`)`
: Focus on the current invocation of break level *BL*. *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 *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.

`breakpoint_expansion(+`***Macro***`, -`***Body***`)` ***hook,development***
`user:breakpoint_expansion(+`***Macro***`, -`***Body***`)`
: This predicate is called with each (non-composite) breakpoint test or action, as its first argument. If it succeeds, the term returned in the second argument (`Body`) is substituted for the original condition. The expansion is done at the time the breakpoint is added.

  Note that *Body* can be composite, but it cannot be of form ***Tests-Actions***. This means that the whole *Body* will be interpreted as being in either the test or the action part, depending on the context.

  The built-in breakpoint conditions can not be redefined using this predicate.

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 spec. If the command value is created during run time, 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

```
Tests - [goal(foo(X)),...,proceed(_,bar(X))]
```

  should be avoided, and

```
Tests - [goal(foo(X)),...,proceed(foo(Y),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`.

`exception(`***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 203.

`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 203.

`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 203.

`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 203.

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, 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.

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.

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

`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` ***development***

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`)

`profile_data(`***:Spec***`,`***?Selection***`,`***?Resolution***`,`***?Data***`)` ***development***
: *Data* is the profiling data collected from the instrumented predicates covered by *Spec* with selection and resolution *Selection* and *Resolution* respectively.

`profile_reset(`***:Spec***`)` ***development***
: The profiling counters for the instrumented predicates covered by *Spec* are zeroed.

`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

6 Mixing C/C++ and Prolog

SICStus Prolog provides a bi-directional, procedural interface for program parts written in C and Prolog. 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 C functions to predicates.

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 10.22 [lib-structs], page 455 and Section 10.13 [lib-objects], page 378) 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 to set this environment variable, but its value will be used, as a fall-back, at runtime if no explicit boot path is given when initializing a runtime or development system. In this chapter, the environment variable

`SP_PATH` is used as a shorthand for the SICStus Prolog installation directory, whose default location for SICStus 4.0.2 is '`/usr/local/lib/sicstus-4.0.2`') for UNIX and '`C:\Program Files\SICStus Prolog 4.0.2`' for Windows. See Section 13.1 [too-sicstus], page 1156.

> **Windows note**: Explicit use of the `SP_PATH` variable is discouraged, since Windows applications can find out for themselves where they were started from. `SP_PATH` is only used if the directory where '`sprt4-0-2.dll`' is loaded from does not contain `sp-4.0.2` (a directory), '`sprt.sav`', or '`spre.sav`'. If `SP_PATH` is used, SICStus expects it to be set such that `%SP_PATH%\bin` contains '`sprt.sav`' or '`spre.sav`'. See Section 6.7.2 [Runtime Systems on Target Machines], page 277.

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`, the macro `SP_errno` will return a value describing the error condition:

```
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, 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 `representation_error(...,...,mis_encoded_string)`.

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 1045.

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 '`~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 260), to form a ***linked foreign resource***. A linked foreign resource exists in two different flavors, ***static*** and ***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 (***pre-linked***), or at runtime. Pre-linking can be done using static or dynamic 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 pre-linked, only the predicate names are bound; otherwise, runtime-linking is attempted (using `dlopen()`, `LoadLibrary()`, or similar).

6.2.2 Conversion Declarations

Conversion declaration predicates:

`foreign_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 260). 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:

```
foreign_resource('terminal', [scroll,pos_cursor,ask]).
```

specifies that functions `scroll()`, `pos_cursor()` and `ask()` are to be found in the resource '`terminal`'.

`foreign(+`***CFunctionName***`, +`***Predicate***`)` ***hook***
`foreign(+`***CFunctionName***`, +`***Language***`, +`***Predicate***`)` ***hook***

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:

```
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 raise an exception if an input arguments is uninstantiated (`instantiation_error/2`) or has the wrong type (`type_error/4`) or domain (`domain_error/4`). 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.

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: long` The argument should be a number. It is converted to a C `long` and passed to the C function.

`Prolog: +float`
`C: double` The argument should be a number. It is converted to a C `double` and passed to the C function.

`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, 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: long *` The C function is passed a reference to an uninitialized `long`. 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.

`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.

`Prolog: [-integer]`
`C: long` ***F*** `()`

The C function should return a `long`. 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.

`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, it will be unloaded, as if `unload_foreign_resource(`***Resource***`)` were called, before *Resource* is loaded.

`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, it is unlinked from the Prolog load image.

If no resource named ***ResourceName*** is currently loaded, 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.

> **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 255). `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.

6.2.5.1 Customizing `splfr` under UNIX

The `splfr` tool is implemented as a Perl script and can be customized in order to adapt to local variations. *Do not attempt this unless you know what you are doing*. Customization is done by editing their common configuration file `spconfig-`***`version`***. Follow these instructions:

1. Locate the configuration file `spconfig-`***`version`***. It should be located in the same directory as `splfr`.
2. Make a copy for `spconfig-`***`version`***; let's call it `hacked_splfr.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 flags.
4. You may now use the modified `spconfig-`***`version`*** together with `splfr` like this:

   ```
   % splfr [...] --config=/path/to/hacked_splfr.config
   ```
5. Replace '`/path/to`' with the actual path to the hacked configuration file.

6.2.5.2 Creating Dynamic 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.

6.2.6 Init and Deinit Functions

An init function and/or a deinit function can be declared by `foreign_resource/2`. If this is the case, these functions should have the prototype:

`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()`), 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, 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 27. 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:

```
                                                      % 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 long and having a long as the only parameter. The conversion declarations in 'ex.pl' state that these functions form the foreign resource ex.

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.

Dynamic linking of foreign resources can also be used by runtime systems. On some platforms, however, the executable must not be ***stripped*** for dynamic linking to work, i.e. its symbol table must remain.

6.3 Calling C++ from Prolog

Functions in C++ files that should be called from Prolog must use C linkage, e.g.

```
extern "C" {
void myfun(long i)
{...};
};
```

To build a dynamic linked foreign resource with C++ code, you may (depending on platform) have to explicitly include certain libraries. E.g. on Sparc/SunOS 5.X using `gcc`:

```
% splfr .... -LD -L/usr/gnu/lib/gcc-lib/sparc-sun-solaris2.4/2.7.0 -
lgcc
```

The library path is installation dependent, of course.

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 arguments 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 of SICStus Prolog 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.

It is important to understand the rules governing the scope of SP_term_refs in conjunction with calls from Prolog to C and vice versa:

> When a C function called from Prolog returns, all SP_term_refs passed to the function or dynamically created by the function become invalid.
>
> When terms are passed to C as a result of calling Prolog, those terms and any SP_term_refs created since the start of the query are only valid until backtracking into the query or an enclosing one.

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()`.

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.

`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 run-time 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 run-time is not thread safe in general.

A dynamic foreign resource that is used by multiple SICStus run-times in the same process may need to maintain a global state that is kept separate for each SICStus run-time. Each SICStus run-time 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 run-time.

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 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 a more complicated scheme is used. You find the predicate definition as above, but you don't 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, 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 is to use `SP_read_from_string()` (see Section 6.4.3 [Creating Prolog Terms], page 263) 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.

6.5.3 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*, 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, 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, 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.

6.5.3.1 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 from SICStus 3.8 and earlier).

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. Under UNIX, `library(timeout)` currently uses `SIGVTALRM`.

When `func` is called, it cannot call any SICStus API functions except `SP_event()`. Note that `func` will be called in the main thread.

6.5.4 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, 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.

To propagate failure to Prolog, call `SP_fail()`. Upon return, Prolog will backtrack.

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.

6.6 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.

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.

```
#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_opti
{
  stdio_t_stream *s;

  s = (stdio_t_stream *)*puser_data;
  /* we can ignore SPIO_DEVICE_CLOSE_OPTION_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)

long SPCDECL 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;
```

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 implementions 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 1042, 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.

`int` ***`user_read`*** `()`
: Should fill a buffer with data available from the stream. See Section 12.3.103 [cpg-ref-user_read], page 1151.

`int` ***`user_write`*** `()`
: Should write data from a buffer to the stream. See Section 12.3.104 [cpg-ref-user_write], page 1153.

`int` ***`user_flush_output`*** `()`
: Should flush the (output) stream.

`int` ***`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 1056, Section 12.3.57 [cpg-ref-SP_next_stream], page 1100 and Section 12.3.40 [cpg-ref-SP_get_stream_user_data], page 1082.

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 286). In custom built systems, they may be called in the hook function `SU_initialize()`. See Section 6.7.3 [The Application Builder], page 280.

6.6.3.1 Writing User-stream Hooks

The user-stream hook is, if defined, called during `SP_initialize()`. It has the following prototype:

```
SP_stream *user_stream_hook(void *user_data, int which)
```

If the hook is not defined, 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), 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 271.

See Section 12.3.90 [cpg-ref-SP_set_user_stream_hook], page 1135, for details.

6.6.3.2 Writing User-stream Post-hooks

If defined, 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:

```
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 12.3.91 [cpg-ref-SP_set_user_stream_post_hook], page 1136, 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. The Prolog flags `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 system contains the compiler.
- The Prolog flags `discontiguous_warnings`, `redefine_warnings`, `single_var_warnings` have no effect. The user is not prompted in the event of name clashes.
- The `informational` Prolog flag is off by default, suppressing informational messages.
- No profiler. The predicates `profile_data/4` and `profile_reset/1` are unavailable.
- No signal handling except as installed by `SP_signal()`.

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 280).

the runtime kernel
: This is a shared object or a DLL, usually '`$SP_PATH/../libsprt4-0-2.so`' under UNIX, or '`%SP_PATH%\..\sprt4-0-2.dll`' under Windows.

the (extended) runtime library
: The saved-state '`$SP_PATH/bin/sprt.sav`' contains the built-in predicates written in Prolog. It is restored into the program at runtime by the function `SP_initialize()`. Extended runtime systems restore '`$SP_PATH/bin/spre.sav`' instead, available from SICS as an add-on product.

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 286).

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 281.

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.

Next, in order for SICStus to be able to locate all relevant files, the following directory structure should be used.

```
myapp.exe
sp-4.0.2/
+--- libsprt4-0-2.so

+--- sicstus-4.0.2/
     +--- bin/
     |    +--- sprt.sav
     +--- library/
          +--- <files from $SP_PATH/library>
```

If support for multiple SICStus instances is needed, the run-times named e.g. '`libsprt4-0-2_instance_01_.so`' need to be available as well, in the same place as '`libsprt4-0-2.so`'.

If SICStus Prolog is installed on the target machine, a symbolic link named '`sp-4.0.2`' can be used, in which case it should point to the directory of the SICStus installation that contains the '`libsprt4-0-2.so`' (or equivalent).

'`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 run-time (e.g., '`libsprt4-0-2.so`') as well as the boot file '`sprt.sav`' ('`spre.sav`'). In addition, application specific files, e.g. a '`.sav`' file, can be found using the automatically set environment variables `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.

On some platforms, the executable will not be able to locate '`sprt.sav`' ('`spre.sav`') unless the environment variable `SP_PATH` is set. If the example above is rooted in '`/home/joe`', `SP_PATH` should be set to '`/home/joe/lib/sicstus-4.0.2`'. This is not needed on platforms where `SP_APP_DIR` or `SP_RT_DIR` can be used, e.g. on Linux, Solaris or Win32 and most others.

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 'libsprt4-0-2.so'. If the '--static' option is used, this is not necessary. Setting LD_LIBRARY_PATH is not needed, nor recommended, on Linux and Solaris (and Win32).

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-4.0.2':

'../*.{a,so,sl,dylib}'
'bin/sprt.sav'
'bin/spre.sav'
'bin/prologbeans.jar'
'library/*.{tcl,po,pl}'
    Except 'license.pl'!

'library/*/*.{s.o,so,sl,dylib}'
'library/*/*.{po,pl}'
'sp_platform'
    (Located with InstallSICStus)

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
sprt4-0-2.dll

sp-4.0.2\
+--- bin\
|    +--- sprt.sav
+--- library\
     +--- <files from %SP_PATH%\library>
```

If support for multiple SICStus instances is needed, the run-times named e.g. 'sprt4-0-2_instance_01_.dll' need to be available as well, in the same place as 'sprt4-0-2.dll'.

'myapp.exe' is typically created by a call to spld:

```
% spld --main=user [...] -o ./myapp.exe
```

If the directory containing 'sprt4-0-2.dll' contains a directory called sp-4.0.2, 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-4.0.2/bin'), as in the picture. Furthermore, the initial library_directory/1 fact will be set to the same directory with sp-4.0.2/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(system) and library(random), your directory structure may look like:

```
myapp.exe
sprt4-0-2.dll

sp-4.0.2\
+--- bin\
|    +--- sprt.sav
+--- library\
     +--- random.po
     +--- system.po
     +--- x86-win32-nt-4 \
          +--- random.dll
          +--- system.dll
```

The `sp*` files can also be put somewhere else in order to be shared by several applications provided the '`sprt4-0-2.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%`':

'`bin\sprt.sav`'
'`bin\spre.sav`'
'`bin\prologbeans.jar`'
'`bin\*.dll`'
'`bin\*.po`'
'`library\*.{tcl,po,pl,bas}`'
    Except '`license.pl`'!

'`library\*\*.dll`'
'`library\*\*.{po,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.

6.7.3.1 Customizing `spld` under UNIX

The `spld` tool is implemented as a Perl script and can be customized in order to adapt to local variations. *Do not attempt this unless you know what you are doing.* Customization is done by editing their common configuration file `spconfig-`***`version`***. 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's 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. Together with static linking, this gives an all-in-one executable, an executable 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 run-time and its run-time saved state.

The keys to this feature are:

Static linking. By linking an application with a static version of the SICStus run-time, you avoid any dependency on e.g. '`sprt4-0-2.dll`' (Windows) or '`libsprt4-0-2.so`' (UNIX). Note that, as of SICStus 3.9, static linking is supported under Windows.

If the application needs foreign resources (predicates written in C code), as used for example by `library(system)` and `library(clpfd)`, 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 run-time 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 *le*=***name*** where *le* 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 *le* during run-time. A typical choice of *name* would be the base name, i.e. without directories, of *le*, 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 run-time system that consists of a single file '`main.pl`' that looks like:

% main.pl

```
:- use_module(library(system)).
:- 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 date:'),nl,
   datime(Date),             % from system
   write(Date),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(system)` and `library(clpfd)` and other Prolog libraries needed by `library(clpfd)`.

```
% sicstus -i -f
SICStus 4.0.2 ...
Licensed to SICS
| ?- compile(main).
% compiling .../main.pl...
% ... loading several library modules
| ?- save_program('main.sav').
% .../main.sav created in 201 msec

| ?- halt.
```

Finally, tell `spld` to build an executable statically linked with the SICStus run-time and the foreign resources needed by `library(system)` and `library(clpfd)`. Also, embed the Prolog run-time saved-state and the application specific saved-state just created. Note that the '`random`' foreign resource is needed since it is used by `library(clpfd)`.

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,system,random
```

The arguments are as follows:

'`--output=main.exe`'
: This tells `spld` where to put the resulting executable.

'`--static`'
: Link statically with the SICStus run-time and foreign resources (`system` and `clpfd` in this case).

'`--embed-rt-sav`'
: This option embeds the SICStus run-time '`.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 run-time 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`
`system`
`random`
: These tell `spld` to link with the foreign resources (that is, C-code) associated with `library(system)`, `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 date:
datime(2006,4,21,1,4,55)
3 != 9
bash-2.04$
```

6.7.3.3 Examples

1. The character-based SICStus development system executable (`sicstus`) can be created using

   ```
   % spld --main=prolog -o sicstus
   ```

   This will create a development system that is dynamically linked and has no pre-linked foreign resources.

2.

   ```
   % spld --static -D --resources=random -o main -ltk8.0 -ltcl8.0
   ```

   This will create a statically linked executable called `main` that has the resource `random` pre-linked (statically). The linker will receive '`-ltk8.0 -ltcl8.0`', which will work under UNIX (if Tcl/Tk is installed correctly) but will probably fail under Windows.

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 281, with the addition of a foreign resource of our own.

% foo.pl

```
:- use_module(library(system)).
:- use_module(library(clpfd)).

:- load_foreign_resource(bar).

% 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 date:'),nl,
   datime(Date),             % from system
   write(Date),nl,
   ( all_different([3,9]) ->        % from clpfd
       write('3 != 9'),nl
   ; otherwise ->
       write('3 = 9!?'),nl
   ),
   '$pint'(4711).                   % from our own foreign resource 'bar'
```

% bar.pl

```
foreign(print_int, '$pint'(+integer)).
foreign_resource(bar, [print_int]).
```

/* bar.c */

```
#include <sicstus/sicstus.h>
#include <stdio.h>
#include "bar_glue.h"

extern void print_int(long a);

void print_int(long a)
{
  printf("a=%lu\n", a);
}
```

To create the saved-state `foo.sav` we will compile the file '`foo.pl`' and save it with `save_program('foo.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(foo), save_program('foo.sav'), halt."***

We also need a static foreign resource to embed in our all-in-one executable.

% ***splfr --static bar.c bar.pl***

Lastly 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=foo.sav=/mystuff/foo.sav --output=foo
```

In this case four foreign resource names are extracted from the '`.sav`' file: `bar`, `clpfd`, `random` and `system`. The source file '`foo.pl`' loads the foreign resource named `bar`. It also uses `library(system)` module, which loads the foreign resource named `system`, and `library(clpfd)` module, which loads the foreign resources named `clpfd` and `random`.

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, a function `user_main()` must be supplied:

```
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, initialize command line arguments so that they can be accessed by the `argv` Prolog flag, and load the Runtime Library.

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 input stream as a terminal, even if it does not appear to be a terminal. Same as the '`-i`' option in development systems (see Section 3.1 [Start], page 21).

You may also call `SP_set_memalloc_hooks()` before calling `SP_initialize()`. This will define one layer of Prolog's memory manager, in case your application has special requirements.

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's 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 under Windows

There are three ready-made runtime systems provided with the distributions, '`%SP_PATH%\bin\sprt.exe`', '`%SP_PATH%\bin\sprtw.exe`', and '`%SP_PATH%\bin\sprti.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`, the given startup goal is run. Then, `user:runtime_entry(start)` is run. The program exits with 0 upon normal temination and with 1 on failure or exception.

The program `sprti` assumes that the standard streams are connected to a terminal, even if they do not seem to be (useful under Emacs, for example). `sprtw` is a windowed executable, corresponding to `spwin`.

6.8 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:

% train.pl

```
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 <sicstus/sicstus.h>

void write_path(SP_term_ref path)
{
  char *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
     reserved and should always be NULL */
  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");
      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");
      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:

```
% sicstus
SICStus 4.0.2 ...
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:

```
% spld --main=user train.c -o train.exe
```

And finally, run the executable:

```
% ./train
Path: Stockholm -> Katrineholm -> Hallsberg -> Kumla -> Orebro
Path: Stockholm -> Vasteras -> Orebro
```

6.8.2 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, the function should raise an exception instead.

/* math.c */

```
#include <math.h>
#include <stdio.h>
#include <sicstus/sicstus.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 4.0.2 ...
Licensed to SICS
| ?- [math].
% compiling /home/san/pl/math.pl...
% /home/san/pl/math.pl compiled, 10 msec 816 bytes

| ?- sqrt(5.0,X).

X = 2.23606797749979

| ?- sqrt(a,X).
! Type error in argument 1 of user:sqrt/2
! number expected, but a found
! goal:  sqrt(a,_143)

| ?- sqrt(-5,X).
! Domain error in argument 1 of user:sqrt/1
! expected '>=0.0', found -5.0
! goal:  sqrt(-5.0)
```

The above example used the foreign language interface with dynamic linking. To statically link '`math.s.o`' with the Prolog emulator, the following steps would have been taken:

```
% splfr -S math.pl math.c -lm
SICStus 4.0.2 ...
Licensed to SICS
% spXxQwsr.c generated, 0 msec

% spld -D -o mathsp --resources=./math.s.o
SICStus 4.0.2 ...
Licensed to SICS
% spYdLTgi1.c generated, 0 msec

Created "mathsp"
% ./mathsp
SICStus 4.0.2 ...
Licensed to SICS
| ?- [math].
% compil-
ing /a/filur/export/labs/isl/sicstus/jojo/sicstus38p/math.pl...
% com-
piled /a/filur/export/labs/isl/sicstus/jojo/sicstus38p/math.pl in mod-
ule user, 0 msec 960 bytes

| ?- sqrt(5.0,X).

X = 2.23606797749979
```

6.8.3 Stream 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 has been closed when the foreign resource is unloaded.

7 Interfacing .NET and Java

SICStus Prolog provides a uniform way of interfacing to Java and .NET clients via the ***PrologBeans*** (see Section 10.40 [lib-prologbeans], page 682) 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.

8 Multiple SICStus Run-Times in a Process

It is possible to have more than one SICStus run-time 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 Run-Times], page 299).

Even though the SICStus run-time can only be run in a single thread, it is now possible to start several SICStus run-times, optionally each in its own thread.

SICStus run-times are rather heavy weight and you should not expect to be able to run more than a handful.

8.1 Memory Considerations

The most pressing restriction when using more than one SICStus run-time in a process is that (on 32bit machines) all these run-times must compete for the the *address-constrained* range of virtual memory, typically the lower 256MB of memory.

This is worsened by the fact that, on some platforms, each SICStus run-time will attempt to grow its memory area as needed, leading to fragmentation. A fix that removes the restriction on useable memory is planned for a later release.

One way to avoid the fragmentation issue to some extent is to make each SICStus run-time preallocate a large enough memory area so it will not have to grow during run-time. This can be effected by setting the environment variables `GLOBALSTKSIZE`, `PROLOGINITSIZE` and `PROLOGMAXSIZE`.

On some platforms, currently `Linux` and Windows, the default bottom memory manager layer will pre-allocate as large a chunk of *address-constrained* memory as possible when the SICStus run-time is initialized. In order to use more than one run-time you therefore should set the environment variable `PROLOGMAXSIZE` to limit this greedy pre-allocation.

```
bash> GLOBALSTKSIZE=10MB; export GLOBALSTKSIZE;
bash> PROLOGINITSIZE=20MB; export PROLOGINITSIZE;
bash> PROLOGMAXSIZE=30MB; export PROLOGMAXSIZE;
```

You can use `statistics/2` to try to determine suitable values for thes, but it is bound to be a trial-and-error process.

8.2 Multiple SICStus Run-Times in C

Unless otherwise noted, this section documents the behavior when using dynamic linking to access a SICStus run-time.

The key implementation feature that makes it possible to use multiple run-times is that all calls from C to the SICStus API (`SP_query()`, etc.) go through a dispatch vector. Two run-times 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, use the C macro SICStusDISPATCHVAR instead; see below.

8.2.1 Using a Single SICStus Run-Time

When building an application with spld, by default only one SICStus run-time can be loaded in the process. This is similar to the case in SICStus versions prior to 3.9. For most applications built with spld, the changes necessary to support multiple SICStus run-times 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 by SP_initialize(). 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_memalloc_hooks() also set up the dispatch vector to allow them to be called before SP_initialize(). However, only SP_initialize() is 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 Run-Time

Using more than one SICStus run-time in a process is only supported when the dynamic library version of the SICStus run-time is used (e.g, sprt4-0-2.dll, libsprt4-0-2.so).

An application that wants to use more than one SICStus run-time 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 run-time 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 run-times.

There are several steps needed to access a SICStus run-time from an application built with '--multi-sp-aware'.

1. You must obtain the dispatch vector of the initial SICStus run-time 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 run-time dynamic library (sprt4-0-2.dll under Windows and, typically, libsprt4-0-2.so under UNIX).
2. You must ensure that SICStusDISPATCHVAR expands to something that references the dispatch vector obtained in step 1.

   The C pre-processor 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 run-time you can call the SICStus API function SP_load_sicstus_run_time() to load additional run-times.

```
SICSTUS_API_STRUCT_TYPE *SP_get_dispatch(void *reserved);
```

SP_get_dispatch() returns the dispatch vector of the SICStus run-time. The argument reserved should be NULL. This function can be called from any thread.

```
typedef SICSTUS_API_STRUCT_TYPE *SP_get_dispatch_type(void *);

int SP_load_sicstus_run_time(SP_get_dispatch_type **ppfunc, void **phandle);
```

SP_load_sicstus_run_time() loads a new SICStus run-time. SP_load_sicstus_run_time() returns zero if a new run-time could not be loaded. If a new run-time could be loaded a non-zero value is returned and the address of the SP_get_dispatch() function of the newly loaded SICStus run-time 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 run-time 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 run-time is loaded. Instead, the SP_get_dispatch() of the run-time itself is returned. In particular, the first time SP_load_sicstus_run_time() is called on the initial SICStus run-time, and if this happens before the initial SICStus run-time is initialized, no new run-time is loaded.

Calling SP_load_sicstus_run_time() from a particular run-time can be done from any thread.

An application that links statically with the SICStus run-time should not call SP_load_sicstus_run_time().

You should not use pre-linked foreign resources when using multiple SICStus run-times in the same process.

8.3 Foreign Resources and Multiple SICStus Run-Times

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 run-time. An alternative mode makes it possible for a foreign resource to be shared between several SICStus run-times in the same process.

Unless otherwise noted, this section documents the behavior when using dynamically linked foreign resources. That is, shared objects (.so-files) under UNIX, dynamic libraries (DLLs) under Windows.

8.3.1 Foreign Resources Supporting Only One SICStus Run-Time

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 run-time.

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 run-times. Requiring a foreign resource to maintain its global state on a per SICStus run-time basis would be an incompatible change. Instead, by default, only the first SICStus run-time that loads a foreign resource will be allowed to use it. If a subsequent SICStus run-time (in the same process) tries to load the foreign resource, an error will be reported to the second SICStus run-time.

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 run-time 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 260.

8.3.2 Foreign Resources Supporting Multiple SICStus Run-Times

A foreign resource that wants to be shared between several SICStus run-times must somehow know which SICStus run-time is calling it so that it can make callbacks using the SICStus API into the right SICStus run-time. In addition, the foreign resource may have global variables that should have different values depending on which SICStus run-time 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 run-time 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 260.

8.3.2.1 Simplified Support for Multiple SICStus Run-Times

To make it simpler to convert old foreign resources, there is an intermediate level of support for multiple SICStus run-times. This level of support makes it possible for several SICStus run-times to call the foreign resource, but a mutual exclusion lock ensures that only one SICStus run-time at a time can execute code in the foreign resource. That is, the mutex is locked upon entry to any function in the foreign resource and unlocked when the function returns. This makes it possible to use a global variable to hold the SICStus dispatch vector, in much the same way as is done when only a single SICStus run-time is supported. In addition, a special hook function in the foreign resource will be called every time the foreign resource is entered. This hook function can then make arrangements to ensure that any global variables are set up as appropriate.

To build a foreign resource in this way, use `splfr --exclusive-access`. In addition to including the generated header file, your code needs to define the context switch function. If the resource is named *resname*, the context switch hook should look like:

```
void sp_context_switch_hook_resname(int entering)
```

The context switch hook will be called with the SICStus API dispatch vector already set up, so calling any SICStus API function from the context switch hook will work as expected. The argument `entering` will be non-zero when a SICStus run-time is about to call a function in the foreign resource. The hook will be called with `entering` zero when the foreign function is about to return to SICStus.

It is possible to specify a name for the context switch hook with the `splfr` option '`--context-hook=`***name***'. If you do not require a context switch hook you can specify the `splfr` option '`--no-context-hook`'.

Due to the use of mutual exclusion lock to protect the foreign resource, there is a remote possibility of dead-lock. This would happen if the foreign resource calls back to SICStus and then passes control to a different SICStus run-time in the same thread, which then calls the foreign resource. For this reason it is best to avoid '`--exclusive-access`' for foreign resources that makes call-backs into Prolog.

The new SICStus API function `SP_foreign_stash()` provides access to a location where the foreign resource can store anything that is specific to the calling SICStus run-time. The location is specific to each foreign resource and each SICStus run-time. See Section 6.4.7.3 [OS Threads], page 267.

C code compiled by `splfr --exclusive-access` will have the C pre-processor macro `SP_SINGLE_THREADED` defined to a non-zero value.

Some of the foreign resources in the SICStus library use this technique; see for instance `library(system)`.

8.3.2.2 Full Support for Multiple SICStus Run-Times

To fully support multiple SICStus run-times, a foreign resource should be built with `splfr --multi-sp-aware`.

C code compiled by `splfr --multi-sp-aware` will have the C pre-processor macro `MULTI_SP_AWARE` defined to a non-zero value.

Full support for multiple SICStus run-times means that more than one SICStus run-time 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 run-time. 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 run-times 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 run-times:

`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:

```
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(long 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 long 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`, 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 267, for details.

Most foreign resources that come with SICStus fully support multiple SICStus run-times. 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 pre-processor, see the files in '`library/clpfd/`'.

8.4 Multiple Run-Times and Threads

Perhaps the primary reason to use more than one SICStus run-time in a process is to have each run-time running in a separate thread. To this end, a few mutual exclusion primitives are available. See Section 6.4.7 [Operating System Services], page 266, for details on mutual exclusion locks.

> **Please note**: the SICStus run-time is not thread safe in general. See Section 6.5.3 [Calling Prolog Asynchronously], page 268, 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.

- Don't 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 59). 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, 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, it is usually for this reason.
- Replace cuts by if-then-else constructs if the test is simple enough (see Section 9.9 [Conditionals and Disjunction], page 321).
- 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).
- Don't 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. The SICStus Prolog compiler has the capability of instrumenting compiled code with ***counters***, which are initially zero and incremented whenever the flow of control passes a given point in the compiled code. This way the number of calls, backtracks, choicepoints created, etc., can be counted for the instrumented predicates, and an estimate of the time spent in individual clauses and disjuncts can be calculated.

Gauge is a graphical user interface for inspecting execution profiles. It is available as a library module (see Section 10.39 [lib-gauge], page 679).

The original version of the profiling package was written by M.M. Gorlick and C.F. Kesselman at the Aerospace Corporation [Gorlick & Kesselman 87].

Only compiled code can be instrumented. To get an execution profile of a program, the compiler must first be told to produce instrumented code. This is done by issuing the query:

```
| ?- prolog_flag(compiling,_,profiledcode).
```

after which the program to be analyzed can be compiled as usual. Any new compiled code will be instrumented while the `compiling` Prolog flag has the value `profiledcode`.

The profiling data is generated by simply running the program. The predicate `profile_data/4` (see below) makes available a selection of the data as a Prolog term. The predicate `profile_reset/1` zeroes the profiling counters for a selection of the currently instrumented predicates. For more information, see the respective reference page.

`profile_data(`***:Spec***`,`***?Selection***`,`***?Resolution***`,`***-Data***`)` ***development***
: *Data* is the profiling data collected from the instrumented predicates covered by *Spec* with selection and resolution *Selection* and *Resolution* respectively.

`profile_reset(`***:Spec***`)` ***development***
: The profiling counters for the instrumented predicates covered by *Spec* are zeroed.

9.3 The Cut

9.3.1 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 9.6 [The Determinacy Checker], page 311).

9.3.2 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, 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, 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 9.9 [Conditionals and Disjunction], page 321 for more information on optimization using conditionals).

9.3.3 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, 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 9.6 [The Determinacy Checker], page 311.

9.3.4 Terminating a Backtracking Loop

Cut is also commonly used in conjunction with the generate-and-test programming paradigm. For example, consider the predicate `find_solution/1` defined by

```
find_solution(X) :-
        candidate_solution(X),
        test_solution(X),
        !.
```

where `candidate_solution/1` generates possible answers on backtracking. The intent is to stop generating candidates as soon as one is found that satisfies `test_solution/1`. If the cut were omitted, 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, 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.

9.4 Indexing

9.4.1 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, 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 dynamic predicates as well as static predicates are indexed. The programming hints given in this section apply equally to static and dynamic code.

9.4.2 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, 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.
```

9.4.3 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, 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, 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, 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, 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

```
| ?- 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, this would not work.

9.5 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:

```
member(Element, [Element|_]).
member(Element, [_|Rest]) :-
        member(Element, Rest).
```

`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

```
| ?- 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.

9.6 The Determinacy Checker

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, but 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

9.6.1 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

```
:- load_files(library(detcheck),
              [when(compile_time), if(changed)]).
```

Once this line is added, every time that file is loaded, it will be checked for unwanted nondeterminacy.

9.6.2 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

`:- nondet` ***`name/arity`***`.`

in your source file. This is similar to a `dynamic` or `discontiguous` declaration. You may have multiple `nondet` declarations, and a single declaration may mention several predicates, separating them by commas.

Similarly, a predicate *P/N* may be classified as nondeterminate by the checker, whereas in reality it is determinate. This may happen e.g. if *P/N* calls a dynamic predicate that in reality never has more than one clause. To prevent false alarms asiring from this, you can inform the checker about determinate predicates by declarations of the form:

`:- det` ***`name/arity`***`.`

If you wish to include `det` and `nondet` declarations in your file and you plan to use the stand-alone determinacy checker, you must include the line

```
:- load_files(library(nondetdecl),
              [when(compile_time), if(changed)]).
```

near the top of each file that contains such declarations. If you use the integrated determinacy checker, you do not need (and should not have) this line.

9.6.3 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, 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, 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, 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.
```

9.6.4 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

```
:- nondet parent/2.
```

before the clauses for `parent/2`. If run again after modifying file, 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`.

9.6.5 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 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, 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`' *i le* option. `spdet` will execute any operator declaration it encounters.

9.6.6 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's 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 can't 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).

9.7 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,

```
% 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.

9.7.1 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 isn't 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.

9.7.2 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:

```
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.

Don't 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.

9.8 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.

```
ident_univ(X, Y) :-
        var(X),                 % If X is a variable,
        !,
        var(Y),                 % so must Y be, and
        samevar(X, Y).          % they must be the same.
ident_univ(X, Y) :-             % If X is not a variable,
        nonvar(Y),              % neither may Y be;
        X =.. [F|L],            % they must have the
        Y =.. [F|M],            % 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
        var(Y),                 % leaves Y unbound,
        !,                      % they were not the same
        fail.                   % 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, 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

```
arg(1, Term, First),
arg(3, Term, Third),
arg(4, Term, Fourth)
```

than to write

```
Term =.. [_,First,_,Third,Fourth|_]
```

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:

```
add_date(OldItem, Date, NewItem) :-
        OldItem =.. [item,Type,Ship,Serial],
        NewItem =.. [item,Type,Ship,Serial,Date].
```

However, this could be expressed more clearly and more efficiently as

```
add_date(OldItem, Date, NewItem) :-
        OldItem = item(Type,Ship,Serial),
        NewItem = item(Type,Ship,Serial,Date).
```

or even

```
add_date(item(Type,Ship,Serial),
         Date,
         item(Type,Ship,Serial,Date)
        ).
```

9.9 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(_Ch, 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`

`float/1`

`ground/1`

`integer/1`

`nonvar/1`

`number/1`

`simple/1`

`var/1`

`</2`

`=</2`

`=:=/2`

```
=\=/2
>=/2
>/2
@</2
@=</2
==/2
\==/2
@>=/2
@>/2
```

This optimization is actually somewhat more general than what is described above. A sequence of guarded clauses:

```
Head1 :- Guard1, !, Body1.
...
Headm :- Guardm, !, Bodym.
Headn :- Bodym.
```

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.

9.10 Programming Examples

The rest of this chapter contains a number of simple examples of Prolog programming, illustrating some of the techniques described above.

9.10.1 Simple List Processing

The goal `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 `member(`***X***`,`***L***`)` is true if *X* is one of the elements of list *L*. The goal `reverse(`***L1***`,`***L2***`)` is true if list *L2* consists of the elements of list *L1* in reverse order.

```
concatenate([], L, L).
concatenate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).

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

reverse(L, L1) :- reverse_concatenate(L, [], L1).

reverse_concatenate([], L, L).
reverse_concatenate([X|L1], L2, L3) :-
        reverse_concatenate(L1, [X|L2], L3).
```

9.10.2 Family Example (descendants)

The goal `descendant(`***X***`,`***Y***`)` is true if *Y* is a descendant of *X*.

```
descendant(X, Y) :- offspring(X, Y).
descendant(X, Z) :- offspring(X, Y), descendant(Y, Z).

offspring(abraham, ishmael).
offspring(abraham, isaac).
offspring(isaac, esau).
offspring(isaac, jacob).
```

If for example the query

```
| ?- descendant(abraham, X).
```

is executed, Prolog's backtracking results in different descendants of Abraham being returned as successive instances of the variable *X*, i.e.

```
X = ishmael
X = isaac
X = esau
X = jacob
```

9.10.3 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 `compare/3` for case analysis.

The goal `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.

```
get_assoc(Key, t(K,V,L,R), Val) :-
        compare(Rel, Key, K),
        get_assoc(Rel, Key, V, L, R, Val).

get_assoc(=, _, Val, _, _, Val).
get_assoc(<, Key, _, Tree, _, Val) :-
        get_assoc(Key, Tree, Val).
get_assoc(>, Key, _, _, Tree, Val) :-
        get_assoc(Key, Tree, Val).
```

9.10.4 Differentiation

The goal `d(`***E1***`, `***X***`, `***E2***`)` is true if expression *E2* is a possible form for the derivative of expression *E1* with respect to *X*.

```
d(X, X, D) :- atomic(X), !, D = 1.
d(C, X, D) :- 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) :- integer(N), N1 is N-1, d(U, X, DU).
d(-U, X, -DU) :- d(U, X, DU).
```

9.10.5 Use of Meta-Logical Predicates

This example illustrates the use of the meta-logical predicates `var/1`, `arg/3`, and `functor/3`. The goal `variables(`***Term***`, `***L***`, [])` instantiates variable ***L*** to a list of all the variable occurrences in ***Term***. E.g.:

```
| ?- variables(d(U*V, X, DU*V+U*DV), L, []).

L = [U,V,X,DU,V,U,DV]

variables(X, [X|L0], L) :- var(X), !, L = L0.
variables(T, L0, L) :-
%       nonvar(T),
        functor(T, _, A),
        variables(0, A, T, L0, L).

variables(A, A, _, L0, L) :- !, L = L0.
variables(A0, A, T, L0, L) :-
%       A0<A,
        A1 is A0+1,
        arg(A1, T, X),
        variables(X, L0, L1),
        variables(A1, A, T, L1, L).
```

9.10.6 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 `my_clause/1`, e.g.:

```
my_clause( (grandparent(X, Z) :- parent(X, Y), parent(Y, Z)) ).
```

A unit clause will be represented by a term such as

```
my_clause( (parent(john, mary) :- true) ).
```

The mini-interpreter consists of three clauses:

```
execute((P,Q)) :- !, execute(P), execute(Q).
execute(P) :- predicate_property(P, built_in), !, P.
execute(P) :- my_clause((P :- Q)), execute(Q).
```

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. `!`, `(P;Q)`, `(P->Q)`, `(P->Q;R)`, `(\+ P)`, and `if(P,Q,R)`.

9.10.7 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

```
| ?- phrase(sentence(P), [every,man,that,lives,loves,a,woman]).
```

P will get instantiated to

```
all(X):(man(X)&lives(X) => exists(Y):(woman(Y)&loves(X,Y)))
```

where `:`, `&` and `=>` are infix operators defined by

```
:- op(900, xfx, =>).
:- op(800, xfy, &).
:- op(550, xfy, :). /* predefined */
```

The grammar follows:

```
sentence(P) --> noun_phrase(X, P1, P), verb_phrase(X, P1).

noun_phrase(X, P1, P) -->
        determiner(X, P2, P1, P), noun(X, P3), rel_clause(X, P3, P2).
noun_phrase(X, P, P) --> name(X).

verb_phrase(X, P) --> trans_verb(X, Y, P1), noun_phrase(Y, P1, P).
verb_phrase(X, P) --> intrans_verb(X, P).

rel_clause(X, P1, P1&P2) --> [that], verb_phrase(X, P2).
rel_clause(_, P, P) --> [].

determiner(X, P1, P2, all(X):(P1=>P2)) --> [every].
determiner(X, P1, P2, exists(X):(P1&P2)) --> [a].

noun(X, man(X)) --> [man].
noun(X, woman(X)) --> [woman].

name(john) --> [john].

trans_verb(X, Y, loves(X,Y)) --> [loves].
intrans_verb(X, lives(X)) --> [lives].
```

9.11 The Cross-Referencer

9.11.1 Introduction

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, it also traces from `user:runtime_entry/1`. If individual module-files are being checked, 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.

9.11.2 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`' *i le* 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, 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.

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 138) modules. The following packages are provided:

`aggregate` (see Section 10.1 [lib-aggregate], page 332)
: provides an aggregation operator for data-base-style queries.

`assoc` (see Section 10.2 [lib-assoc], page 335)
: uses unbalanced binary trees trees to implement "association lists", i.e. extendible finite mappings from terms to terms.

`atts` (see Section 10.3 [lib-atts], page 337)
: 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 343)
: uses AVL trees to implement "association lists", i.e. extendible finite mappings from terms to terms.

`bags` (see Section 10.5 [lib-bags], page 346)
: defines operations on bags, or multisets

`bdb` (see Section 10.6 [lib-bdb], page 350)
: 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 358)
: provides some means of generating integers.

`codesio` (see Section 10.8 [lib-codesio], page 359)
: defines I/O predicates that read from, or write to, a code-list.

`file_systems` (see Section 10.9 [lib-file_systems], page 360)
: accesses files and directories.

`heaps` (see Section 10.10 [lib-heaps], page 365)
: implements binary heaps, the main application of which are priority queues.

`lists` (see Section 10.11 [lib-lists], page 367)
: provides basic operations on lists.

`logarr` (see Section 10.12 [lib-logarr], page 378)
: provides an implementation of extendible arrays with logarithmic access time.

`objects` (see Section 10.13 [lib-objects], page 378)
provides a package for object-oriented programming, and can be regarded as a high-level alternative to `library(structs)`.

`ordsets` (see Section 10.14 [lib-ordsets], page 440)
defines operations on sets represented as lists with the elements ordered in Prolog standard order.

`process` (see Section 10.15 [lib-process], page 442)
provides process creation primitives.

`queues` (see Section 10.16 [lib-queues], page 446)
defines operations on queues (FIFO stores of information).

`random` (see Section 10.17 [lib-random], page 449)
provides a random number generator.

`rem` (see Section 10.18 [lib-rem], page 450)
provides Rem's algorithm for maintaining equivalence classes.

`samsort` (see Section 10.19 [lib-samsort], page 451)
provides generic sorting.

`sets` (see Section 10.20 [lib-sets], page 451)
defines operations on sets represented as lists with the elements unordered.

`sockets` (see Section 10.21 [lib-sockets], page 453)
provides an interface to sockets.

`structs` (see Section 10.22 [lib-structs], page 455)
provides access to C data structures, and can be regarded as a low-level alternative to `library(objects)`.

`system` (see Section 10.23 [lib-system], page 464)
provides access to operating system services.

`terms` (see Section 10.24 [lib-terms], page 465)
provides a number of operations on terms.

`timeout` (see Section 10.25 [lib-timeout], page 467)
Meta-call with limit on execution time.

`trees` (see Section 10.26 [lib-trees], page 468)
uses binary trees to represent non-extendible arrays with logarithmic access time. The functionality is very similar to that of `library(logarr)`, but `library(trees)` is slightly more efficient if the array does not need to be extendible.

`types` (see Section 10.27 [lib-types], page 469)
Provides type checking.

`ugraphs` (see Section 10.28 [lib-ugraphs], page 471)
Provides an implementation of directed and undirected graphs with unlabeled edges.

`varnumbers` (see Section 10.29 [lib-varnumbers], page 473)
An inverse of `numbervars/3`.

`wgraphs` (see Section 10.30 [lib-wgraphs], page 474)
: provides an implementation of directed and undirected graphs where each edge has an integral weight.

`xml` (see Section 10.31 [lib-xml], page 476)
: provides an XML parser.

`linda/client` (see Section 10.32 [lib-linda], page 478)
`linda/server` (see Section 10.32 [lib-linda], page 478)
: provides an implementation of the Linda concept for process communication.

`chr` (see Section 10.33 [lib-chr], page 482)
: provides Constraint Handling Rules

`clpfd` (see Section 10.34 [lib-clpfd], page 491)
: provides constraint solving over Finite (Integer) Domains

`clpq` (see Section 10.35 [lib-clpqr], page 538)
`clpr` (see Section 10.35 [lib-clpqr], page 538)
: provides constraint solving over Q (Rationals) or R (Reals)

`fdbg` (see Section 10.36 [lib-fdbg], page 562)
: provides a debugger for finite domain constraint programs

`pillow` (see Section 10.37 [lib-pillow], page 586)
: The PiLLoW Web Programming Library,

`tcltk` (see Section 10.38 [lib-tcltk], page 586)
: An interface to the Tcl/Tk language and toolkit.

`gauge` (see Section 10.39 [lib-gauge], page 679)
: is a profiling tool for Prolog programs with a graphical interface based on `tcltk`.

`prologbeans` (see Section 10.40 [lib-prologbeans], page 682)
: Access Prolog from Java.

`comclient` (see Section 10.41 [lib-comclient], page 694)
: An interface to Microsoft COM automaton objects.

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 <*Department*,*TotalArea*> pairs. The Prolog equivalent of this might be

```
dept_office_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^O^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,O),
                I^office(I,Dept,Area,O), min(_,Occupant)).
```

Consider queries like

```
aggregate(sum(Pay), Person^pay(Person,Pay), TotalPay)
```

where for some reason `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

```
aggregate(Template, Discriminator, Generator, Results)
```

does.

Operations available:

`count`
: `sum(1)`

`sum(`***E***`)`
: sum of values of *E*

`min(`***E***`)`
: minimum of values of *E*

`min(`***E***`,`***X***`)`
: `min(`***E***`)` with corresponding instance of *X*

`max(`***E***`)`
: maximum of values of *E*

`max(`***E***`,`***X***`)`
: `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.

```
bagof(X, G, B) :- aggregate(bag(X),    G, L).
```

```
setof(X, G, B) :- aggregate(set(X), X, G, L).
```

Exported predicates:

`forall(`***:Generator***`, `***:Goal***`)`
: succeeds when *Goal* is provable for each true instance of *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 Generator in turn, we make a copy of *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 `\+ \+ p(1), \+ \+ p(2), \+ \+ p(3)`.

  The trick in `foreach/2` is to ensure that the variables of *Goal* which do not occur in *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 generalisation of `setof/3` which lets you compute sums, minima, maxima, and so on.

`aggregate(+`***Template***`, `***:Generator***`, `***-Result***`)`
: is a generalisation 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.

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 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 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 recognised 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`.

`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

```
:- attribute AttributeSpec, ..., AttributeSpec.
```

where each *AttributeSpec* has the form (*Name*/*Arity*). There must be at most one such declaration in a module *Module*.

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 `user:goal_expansion/3` 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. The meaning of the '`+`'/'`-`' prefix is documented below:

Module`:get_atts(`***-Var***`, `***?AccessSpec***`)`
: Gets the attributes of *Var* according to *AccessSpec*. If *AccessSpec* is unbound, it will be bound to a list of all set attributes of *Var*. Non-variable terms cause a type error to be raised. The prefixes in the *AccessSpec* have the following meaning:

  `+(`***Attribute***`)`
  : The corresponding actual attribute must be present and is unified with *Attribute*.

`-(`***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* according to *AccessSpec*. Non-variable terms cause a type error to be raised. 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.

`-(`***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 non-variable term, 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*.

`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 non-variable term, `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*.

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`, `call_residue/2`) 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***`)` ***hook***

This predicate is called in each module that contains an attribute declaration, when an interpretation of the attributes as a goal is needed, for example in `frozen/2` and `call_residue/2`. 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***`)` ***hook***

This predicate is called by the Prolog top level and by the built-in predicate `call_residue/2` 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 substition 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.

Similarly during the execution of `call_residue(`***Goal***`,`***Residue***`)`, when *Goal* has succeeded, `project_attributes/2` is called. After that, all attributed variables created during the execution of *Goal* are located. For each such variable, `attribute_goal/2` produces a term representing the constraint encoded by the attribute, and *Residue* is unified with the list of all such terms.

The exact definition of `project_attributes/2` is constraint system dependent, but see Section 10.34.6 [Answer Constraints], page 516 and see Section 10.35.5 [CLPQR Projection], page 551 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.

% domain.pl

```
:- 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
                                            % because of attributes
                                            % in other modules

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
                                            % verify_attributes/3
        ).
```

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 th calls to `verify_attribute/3`.

Assuming that the code resides in the file '`domain.pl`', we can load it via:

```
| ?- use_module(domain).
```

Let's test it:

```
| ?- 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])

| ?- 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])

| ?- 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)).

:- 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]
        ).
verify_attributes(_, _, []).

attribute_goal(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 corresponding `verify_attributes/3` predicates would typically refer to the attributes from other known solvers/modules via the module prefix in ***Module***`:get_atts/2`.

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 `library(assoc)`, which provides similar functionality based on binary trees that are not kept balanced.

Exported predicates:

`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 is_avl/1) being `@<` than any non-variable. in strict point of fact you *can* construct an AVL tree with variables as keys, but `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 `avl_to_list/2`.

`avl_range(+`***AVL***`, -`***Range***`)`
: unifies *Range* with the ordered set representation of the range of the AVL (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 intertible 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.

`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, Val, 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 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.

`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, +Inc, +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:

`bag`
: creates an empty bag

`bag(E,M,B)`
: extends the bag *B* with a *new* element *E* which occurs with multiplicity *M*, and which precedes all elements of *B* 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

```
bag(f(a,Y), 1, bag(f(X,b), 1, 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,

```
bag(f(a,b), 1, bag(f(a,b), 1, bag))
```

is not the representation of any bag, that bag is represented by

```
bag(f(a,b), 2, 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:

`is_bag(+Bag)`
: recognises proper well-formed bags. You can pass variables to `is_bag/1`, and it will reject them.

`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 *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 | 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 recognise 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

```
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_min/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

```
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 lg 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 *Di erence* is the multiset difference of *Bag1* and *Bag2*.

10.6 External Storage of Terms (Berkeley DB)— `library(bdb)`

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 `assert/1`, `retract/1` and `clause/2`, but the terms are stored on files instead of in primary memory.

The differences compared with the Prolog database are:

A *database* must be opened before any access and closed after the last access. (There are special predicates for this: `db_open/[4,5]` and `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 357).

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 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 (`db_store/3`, `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 352). 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 databas, 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.)

Term	*Store*	*Fetch*
`g(x,y)`	domain error	domain error
`f(A,B)`	`f(A,B)`	instantiation error
	`-`	
`f(a,b)`	`f(a,b) f(a,b)`	`f(a,b)`
	`- - - -`	`- -`
`[a,b]`	`.(a,.(b,[]))`	`.(a,.(b,[]))`
	`- -`	`- -`
`g`	`g`	`g`
	`-`	`-`

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 357.

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

`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. *EnvName* is not subject to `absolute_file_name/3` conversion.

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. *DBName* is not subject to `absolute_file_name/3` conversion.

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 `db_enumerate/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 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]`.

  In SICStus 3.12.0 `bdb:export/[2,3]` is available instead of `db_export/[2,3]`.

`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]`.

  In SICStus 3.12.0 `bdb:import/[2,3]` is available instead of `db_import/[2,3]`.

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), db_enumerate(DBRef, X, _).
X = a(b,1),
DBRef = '$db'(1077241400) ? ;
X = a(c,2),
DBRef = '$db'(1077241400) ? ;
no

| ?- db_current(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 maked `+`. 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 the spec with the most `+` postitions in it breaking ties by choosing the leftmost one.

The terms that contain nonvar terms on every `+` postition will be looked up using indexing based on the principal functor of the term and the principal functor of terms on `+` postitions. 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 353).

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 (see Section 10.6.3 [Berkeley DB], page 351), 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(?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]`. This is not yet as general as it ought to be: if *Lower* and *Upper* are not both integers, *List* must be proper, and if *Lower* and *Upper* are both variables, at least one element of List must be an integer. If *Lower* = *Upper+1*, `numlist(Lower, Upper, [])` is true.

10.8 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 head and tail of the list, 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 head and tail of the list, 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 head and tail of the list, 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_from_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 *S0* and *S* are unified with the head and tail of the list, respectively. `with_output_to_codes/4` also passes the stream in the *Stream* argument. It can be used only by *Goal* for writing.

10.9 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 in `library(directory)` 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. [PM] 4.0 this was never true (it always, and still, barfs if taking property of a non-existing object)

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 725, for details.

Exported predicates:

`rename_file(+OldName, +NewName)`
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)`
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 permisison 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.

  `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:datime/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(., readable).
```

See `directory_property/2`, above, for a list of properties.

`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.

`current_directory(-Directory)`
`current_directory(-Directory, +NewDirectory)`
: *Directory* is unified with the current working directory and the working directory is set to *NewDirectory*.

10.10 Heap Operations—`library(heaps)`

A heap is a labelled 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 `heap(N,Free,Tree)` where *N* is the number of elements in the tree, *Free* is a list of integers which specifies unused positions in the tree, and *Tree* is a tree made of:

`heap`
: terms for empty subtrees and

`heap(Key,Datum,Lson,Rson)`
: terms for the rest

The nodes of the tree are notionally numbered like this:

```
                                 1
                     2      3
               4                   6                      5                      7
            8        12        10        14         9         13         11        15
      ..   ..   ..   ..   ..   ..   ..   ..   ..   ..   ..   ..   ..   ..   ..   ..
```

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 subtreee 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 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).
```

10.11 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*. The *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 behaviour 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*.

`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 *Pre x* 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***`)`. If *List* is not proper, `delete/3` will *fail*. *Kill* and the elements of *List* should be sufficiently instantiated for `\=` to be sound.

`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. If ***List*** is not proper, `delete/4` may *fail*. ***Kill*** and the elements of ***List*** should be sufficiently instantiated for `\=` 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, 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, _).
```

`last(+`***List***`, `***-Last***`)`
: is true when ***List*** is a ***List*** and ***Last*** is its last element. 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 ***N***th 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 ***Elem*** is the ***Nth*** member of ***List***, counting the first as element 1. That is, throw away the first ***N-1*** elements and unify ***Elem*** 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

```
length(Longer,N), length(Shorter,M), 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 permuations of each other. Unlike `perm/2`, it will work even when *List* is not a proper list. It even acts in a marginally sensible way when *Perm* isn't proper either, but it will still backtrack forever. 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.

`perm2(`***?A***`,`***?B***`, `***?C***`,`***?D***`)`
: 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.

`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]*.

`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, ...

`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 `memberhck/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, `length(Sequence) = length(SubSequence)+length(Complement)`, e.g. `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 *SubSequence* is a subsequence of *Sequence*, but may be *Sequence* itself. Thus `subseq0([a,b], [a,b])` is true as well as `subseq0([a,b], [a])`. *Sequence* must be a proper list, since there are infinitely many lists with a given *SubSequence*.

```
?- 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],[],[c],[b,d],[b],[b,c],[a,c,d],
      [a,d],[a],[a,c],[a,b,d],[a,b],[a,b,c]]
```

`subseq1(+`***Sequence***`, `***?SubSequence***`)`
: is true when *SubSequence* is a proper subsequence of *Sequence*, that is it contains at least one element less. *Sequence* must be a proper list, as *SubSequence* does not determine *Sequence*.

`sumlist(+`***Numbers***`, `***?Total***`)`
: is true when *Numbers* is a list of integers, and *Total* is their sum.

`transpose(`***?X***`, `***?Y***`)`
: is true when *X* is a list of the form *[[X11,...,X1m],...,[Xn1,...,Xnm]]* and *Y* is its transpose, that is, *Y = [[X11,...,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*.

`append_length(`***?Prefix***`, `***?Suffix***`, `***?List***`, `***?Length***`)`
: is true when

    ```
    append(Prefix, Suffix, List), length(Prefix, Length).
    ```

    The normal use of this is to split a *List* into a *Pre x* of a given *Length* and the corresponding *Su x*, but it can be used any way around provided that *Length* is instantiated, or *Pre x* is a proper list, or *List* is a proper list.

`append_length(`***?Suffix***`, `***?List***`, `***?Length***`)`
: is true when there exists a list *Pre x* such that `append_length(`***Prefix***`, `***Suffix***`, `***List***`, `***Length***`)` is true. When you don't want to know the *Pre x*, you should call this predicate, because it doesn't construct the *Pre x* argument, which `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 *Pre x* 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 *Pre x* 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. If *Length* is instantiated or *Su x* 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. If *Length* is instantiated or *Su x* 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*. *Amount* must already be instantiated. As it is a strict input, it must come first.

`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.

`sublist(+`***Whole***`, `***?Part***`, `***?Before***`, `***?Length***`, `***?After***`)`

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 *Pre x* are lists and *Pre x* 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 *Pre x* are lists and *Pre x* is a proper prefix of *List*. That is, *Pre x* 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 *Su x* are lists and *Su x* 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 *Su x* are lists and *Su x* is a proper suffix of *List*. That is, *Su x* 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 diagonalise 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]***`)`
: 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 either list 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])`.

`maplist(`***:Pred***`, +`***List***`)`
: succeeds when *Pred(X)* succeeds for each element *X* of *List*.

`maplist(`***:Pred***`, +`***OldList***`, `***?NewList***`)`
: succeeds when *Pred(Old,New)* succeeds for each corresponding *Old* in *OldList*, *New* in *NewList*. Either *OldList* or *NewList* should be a proper list.

`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.

`map_product(Pred, Xs, Ys, PredOfProduct)`
: Just as `maplist(P, Xs, L)` is the analogue of Miranda's

```
let L = [ P x | x <- Xs ]
```

so `map_product(P, Xs, Ys, L)` is the analogue of Miranda's

```
let L = [ P x y | x <- Xs; y <- 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.

`scanlist(`***:Pred***`, +`***List***`, `***?V1***`, `***?V***`)`
: maps a ternary relation *Pred* down a list. If the list is *[X1,X2,...,Xn]*, the computation is *Pred(X1,V1,V2)*, *Pred(X2,V2,V3)*, ..., *Pred(Xn,Vn,V)* So if *Pred* is `plus/3`, `scanlist(plus, List, 0, V)` puts the sum of the list elements in *V*. *List* should be a proper list. 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).

`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.

`exclude(`***:Pred***`, +`***List***`, `***?SubList***`)`
: succeeds when *SubList* is the *SubList* of *List* containing all the elements for which *Pred(Elem)* is *false*. That is, it removes all the elements satisfying *Pred*. *List* should be a proper list.

`exclude(`***:Pred***`, +`***Xs***`, +`***Ys***`, +`***Es***`)`
: is true when Es is the sublist of Xs containing all the elements Xi for which Pred(Xi, Yi) is *false*. Xs should be a proper list, and Ys should be at least as long as Xs (it may be longer).

`exclude(`***:Pred***`, +`***Xs***`, +`***Ys***`, +`***Zs***`, +`***Es***`)`
: is true when *Es* is the sublist of *Xs* containing all the elements *Xi* for which *Pred(Xi, Yi, Zi)* is *false*. *Xs* should be a proper list, and *Ys* and *Zs* should be at least as long as *Xs* (they may be longer).

`include(`***:Pred***`, +`***List***`, `***?SubList***`)`
: succeeds when *SubList* is the *SubList* of *List* containing all the elements for which *Pred(Elem)* is *true*. That is, it retains all the elements satisfying *Pred*. *List* should be a proper list.

`include(`***:Pred***`, +`***Xs***`, +`***Ys***`, +`***Es***`)`
: is true when *Es* is the sublist of *Xs* containing all the elements *Xi* for which *Pred(Xi, Yi)* is *true*. *Xs* should be a proper list, and *Ys* should be at least as long as *Xs* (it may be longer).

`include(`***:Pred***`, +`***Xs***`, +`***Ys***`, +`***Zs***`, +`***Es***`)`
: is true when *Es* is the sublist of *Xs* containing all the elements *Xi* for which *Pred(Xi, Yi, Zi)* is *false*. *Xs* should be a proper list, and *Ys* and *Zs* should be at least as long as *Xs* (they may be longer).

`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 `<`, `=`, or `>` .

`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 of *ListOfLists* has the form *[Head|Tail]* such that `group(Pred, Head, Tail, Tail, [])`, and each element of *ListOfLists* 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*. If the list is empty, it fails quietly.

`min_member(`***?Xmin***`, +`***[X1,...,Xn]***`)`
: unifies *Xmin* with the minimum (in the sense of `@=<`) of *X1*,...,*Xn*. If the list is empty, it fails quietly.

`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 `@=<` .

`min_member(`***:P***`, `***?Xmin***`, +`***[X1,...,Xn]***`)`
: unifies *Xmin* with the minimum element of *[X1,...,Xn]*, as defined by the comparison predicate *P*, which should act like `@=<` .

`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***`, `***?Tail***`)`

is true when `append(Prefix, Tail, Sequence)` and *Pre x*, together with the first element of *Tail*, forms a monotone non-decreasing sequence, and no longer Prefix will do. Pictorially,

```
Sequence = [x1,...,xm,xm+1,...,xn]
Prefix   = [x1,...,xm]
Tail     = [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 *Tail* would be included in *Pre x*. However, this way, it means that if Sequence is a strictly decreasing sequence, the *Pre x* will come out empty.

`increasing_prefix(`***:Order***`, `***?Sequence***`, `***?Prefix***`, `***?Tail***`)`

is the same as `increasing_prefix/3`, except that it uses the binary relation *Order* in place of `¯<`.

`decreasing_prefix(`***?Sequence***`, `***?Prefix***`, `***?Tail***`)`
`decreasing_prefix(`***:Order***`, `***?Sequence***`, `***?Prefix***`, `***?Tail***`)`

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.

10.12 Array Operations—`library(logarr)`

This libary module provides extendible arrays with logarithmic access time. **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:

new_array(-*A*) returns a new empty array *A*.

`is_array(+`***A***`)`
: checks whether *A* is an array.

`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*.

10.13 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.

10.13.1 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.

10.13.1.1 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

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 Prolog prompt.

Because this goal is issued at the Prolog 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 Prolog 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.

10.13.1.2 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 10.13.2.3 [obj-scl-meth], page 385. 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.

```
:- class point =
        [public x:float = 1.0,
         public y:float = 2.0].

Self <- create.

Self <- create(X, Y) :-
        Self << x(X),
        Self << y(Y).

Self <- print(Stream) :-
        Self >> x(X),
        Self >> y(Y),
        format(Stream, '(~w,~w)', [X,Y]).

:- 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.

10.13.1.3 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.

```
| ?- create(point(3,4), PointObj),
     PointObj <- print(user_output).
```

The print message prints `(3.0,4.0)`. The variable `PointObj` is bound to a Prolog term of the form

```
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

```
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.

10.13.1.4 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.

10.13.2 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.

10.13.2.1 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:

```
:- 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.

10.13.2.2 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, the slot is private. The following example shows all four possibilities:

```
:- 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 accomodate 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, 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:

```
:- 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`
: long signed integer

`integer_32`
: 32-bit signed integer

`integer_16`
: 16-bit signed integer

`integer_8`
: 8-bit signed integer

`unsigned`
: long unsigned integer

`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`
: Long address. 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, 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, 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.

10.13.2.3 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.

```
:- 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, 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 `OldX` and `OldY` to 0.0E+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:

```
Self >> r(R) :-
        Self >> x(X),
        Self >> y(Y),
        R2 is X*X + Y*Y,
        sqrt(R2, R).

Self >> theta(T) :-
        Self >> x(X),
        Self >> y(Y),
        A is Y/X,
        atan(A, T).
```

This assumes that `library(math)`, which defines the `sqrt/2` and `atan/2` predicates, has been loaded. The put methods are left as an exercise.

In the rational number class whose definition begins with

```
:- class rational =
        [public num:integer,
         public denom:integer].
```

get and put methods are automatically generated for the `num` and `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:

```
| ?- create(prolog_term, TermObj),
     TermObj << p_term(foo(X,Y)),
     X = a,
     Y = b,
     TermObj >> p_term(foo(c,d)).
```

The reason is that the free variables in `foo(X,Y)` are renamed when the term is stored in the `prolog_term` object's `p_term` slot. This is similar to what happens when such a term is asserted to the Prolog database:

```
| ?- retractall(foo(_,_)),
     assert(foo(X,Y)),
     X = a,
     Y = b,
     foo(c,d).
```

However, this goal would fail, because `c` and `d` cannot be unified:

```
| ?- create(prolog_term, TermObj),
     TermObj << p_term(foo(X,X)),
     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 `fetch_slot/2` and `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, 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 `N0` and `D0` 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, "(~w,~w)", [X, Y]).
```

and

```
Self <- print(Stream) :-
        fetch_slot(num, N),
        fetch_slot(denom, D),
        format(Stream, "~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:

```
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.

```
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

```
:- class interval =
        [public lower:integer,
         public upper:integer].
```

might define a send method

```
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*, 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.
```

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, 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).
```

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.

```
:- 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, 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:

```
:- 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, 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

`:- 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, 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.

```
:- 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, it could be given a different `print_year` method, using the `define_method/3` predicate.

```
| ?- create(date_stamp, DateObj),
     define_method(DateObj,
    print_year(Stream),
    print_roman_year(Stream, DateObj)).
```

If this `date_stamp` object is created in 1994, a `print_year` message sent to it would print the current year as

```
MCMXCIV
```

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.

10.13.3 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.

10.13.3.1 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

```
:- class ClassName = [SlotDef, ...] +  SuperClass.
```

where the list of *SlotDef* descriptions may be empty. In that case, the definition can simplified to

```
:- 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:

```
:- class point =
        [public x:float=0,
         public y:float=0].
```

The definition of the `named_point` class would then begin with

```
:- 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:

```
:- 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/1),
        point << (y/1).

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 10.13.3.2 [obj-inh-mih], page 397.

Sending a message to a class's superclass can only be done within a message clause.

10.13.3.2 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, 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.

```
:- 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

```
:- 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

```
:- 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

```
:- 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_stamp 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.

10.13.3.3 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, 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*.

```
| ?- setof(Message,
   (message(Class, <<, Msg, 1),
    message(Class, >>, Msg, 1)),
   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`.

10.13.4 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 10.13.5.2 [obj-tech-lim], page 407). Of course, unlike objects, Prolog terms cannot be modified. However, when a particular class of objects never needs to be dynamically modified, and doesn't 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.

10.13.4.1 Simple Term Classes

The simplest sort of term class declaration has the following form:

```
:- class ClassName = term(Term).
```

This declares that any term that unifies with *Term* is an instance of class *ClassName*. For example, you might declare:

```
:- 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

```
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'd 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 specifing 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 10.13.4.3 [obj-tcl-tce], page 404.

10.13.4.2 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:

```
:- 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 `rgb_color` class example so that only `color/3` terms whose arguments are all floats between 0.0 and 1.0 are instances of `rgb_color`, we would instead begin the definition as follows:

```
:- 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)).
```

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 `rgb_color` class, only `color/3` terms whose arguments are all floating point numbers between 0 and 1 inclusive will be considered to be instances of `rgb_color`.

10.13.4.3 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

```
:- class colored_rectangle = [
       public origin:point,
       public size:size,
       public color:rgb_color].
```

This will store an `rgb_color` object (i.e., a `color/3` term) in the `color` slot of each `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 `term` type: using the Prolog database. This has all the efficiency disadvantages of `term` slots. In this case, however, we know that all that really needs to be saved in order to save an `rgb_color` object is the three arguments. We also know that each of these arguments is a floating point number, and because precision isn't terribly critical in representating colors, each of these numbers can be stored as a `float`, rather than a `double`. In effect, we know that the *essence* of a `rgb_color` object is these three numbers; if we have them, we can easily construct the `color/3` term. If we provide this information in the declaration of the `rgb_color` class, SICStus Objects can store instances of the `rgb_color` class as 3 separate floats, rather than as a term, significantly improving the performance of creating or destroying a `colored_rectangle` object, as well as accessing or modifying its `color` slot.

The essence of a term class is specified with the following form of `class` declaration:

```
:- class ClassName = term(Term, Constraint, Essence).
```

where *Essence* is of the form

```
[Name1:Type1=i[Variable1], Name2:Type2=i[Variable2], ...]
```

and each *Name* is a distinct atom naming a slot, each *Type* is a slot type as specified in Section 10.13.2.2 [obj-scl-slt], page 383, and each *Variable* is an unbound variable appering 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:

```
:- 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`.

10.13.5 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.

10.13.5.1 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
                       | short
                       | char
                       | unsigned_short
                       | unsigned_char
                       | float
                       | double
                       | atom
                       | address
                       | term
                       | class_name
                       | pointer(atom)
```

10.13.5.2 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-level 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:

```
| ?- 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, copy the objects to the Prolog database as terms, and save them to a QOF file. Then, after you reload the QOF 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.

10.13.6 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)).
```

10.13.6.1 `<-/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` exception 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`

10.13.6.2 <</2

Synopsis

+Obj << *+Att*

Arguments

Obj *object*

Att *term*

Description

Send a message to `Obj` to store the value of `Att` in the object. A put message. `Att` must be an attribute that can be stored in objects of `Obj`'s class.

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.

Put methods are automatically generated for public slots.

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` exception 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`, `store_slot/2`

10.13.6.3 `>>/2`

Synopsis

+Obj >> *+-Att*

Arguments

Obj *object*

Att *term*

Description

Send a message to *Obj* that fetches the value of *Att* from the object. A get message. *Att* must be an attribute to fetch from *Obj*'s class.

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.

Get methods are automatically generated for public slots.

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` exception 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`, `fetch_slot/2`

10.13.6.4 `class/1` *declaration*

Synopsis

`:- 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, the slot is private.

SlotName must be an atom.

SlotType must be one of the following:

`integer`
: long signed integer

`integer_32`
: 32-bit signed integer

`integer_16`
: 16-bit signed integer

`integer_8`
: 8-bit signed integer

`unsigned`
: long unsigned integer

`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`
: Long address.

`term`
: Prolog term.

Class
: Pointer to an instance of *Class*, which must be a previously defined class.

`pointer(`***Type***`)`
: Like `address`, except that access to this slot yields, and update of this slot expects, a unary term whose functor is *Type*

InitialValue may be any constant appropriate for the slot's type.

Term, if specified, is any compound Prolog term. Class declarations of any of the last three forms introduce a *term class*, which defines any term that unifies with *Term* as an instance of the class being defined.

Goal, if specified, is any Prolog goal. This goal may be used to restrict which terms that unify with *Term* will be considered to be instance of the class being defined. The default *Goal* is `true`. Other than when it is `true`, *Goal* will usually share variables with *Term*.

Essence, if specified, is a list of terms of the form

Variable`:`***Type***

where *Variable* is a variable apprearing somewhere in *Term* and *Type* is one of the possible *Slottype* types listed above. There should be a *Variable*:*Type* pair for every variable in *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.

```
:- 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

```
| ?- 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`.

```
:- 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 10.13.2 [obj-scl], page 382, Section 10.13.4 [obj-tcl], page 402.

10.13.6.5 `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`

10.13.6.6 `class_method/1` *declaration*

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 descendent 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`

10.13.6.7 `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`

10.13.6.8 `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`

10.13.6.9 `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, a domain error will be raised.

Examples

Given the class definition

```
:- 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

```
| ?- 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`

10.13.6.10 `current_class/1`

Synopsis

`current_class(*`***`Class`***`)`

Arguments

Class *atom*

Description

`Class` is the name of a currently defined class.

10.13.6.11 `debug_message/0` *declaration*

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`

10.13.6.12 `define_method/3`

Synopsis

`define_method(+`***Obj***`, +`***Message***`, +`***Body***`)`

Arguments

Obj *object*

Message *term*

Body *callable*

Description

Install *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`

10.13.6.13 `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`

10.13.6.14 `destroy/1`

Synopsis

`destroy(+`***`Obj`***`)`

Arguments

Obj *object*

Description

Dispose of *Obj*.

First send 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.

Exceptions

`instantiation_error`
: *Obj* is unbound.

`type_error`
: *Object* is not a valid object.

See Also

`create/2`

10.13.6.15 `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`

10.13.6.16 `end_class/[0,1]` *declaration*

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`

10.13.6.17 `fetch_slot/2`

Synopsis

`fetch_slot(+`***SlotName***`, -`***Value***`)`

Arguments

SlotName *atom*

Value *term*

Description

Fetch *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`

10.13.6.18 `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`

10.13.6.19 `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`

10.13.6.20 `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`

10.13.6.21 `nodebug_message/0` *declaration*

Synopsis

`:- nodebug_message.`

Description

Prolog clauses following this directive are no longer compiled to send messages "carefully."

See Also

`debug_message/0`

10.13.6.22 `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.

Exceptions

`instantiation_error`
: both *Obj* and *Addr* are unbound.

`type_error`
: *Addr* is not an integer.

10.13.6.23 store_slot/2

Synopsis

store_slot(+***SlotName***, +***NewValue***)

Arguments

SlotName *atom*

NewValue *term*

Description

Store *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

10.13.6.24 `undefine_method/3`

Synopsis

`undefine_method(+`***`Obj`***`, +`***`Name`***`, +`***`Arity`***`)`

Arguments

Obj *object*

Name *atom*

Arity *integer*

Description

Remove *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, 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`

10.13.6.25 `uninherit/1` *declaration*

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, the specified message is uninherited from all ancestors that define it.

Caveat

Note that if you define a message for your class, 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`

10.13.7 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

  Object Operator Message

where *Operator* is one of the following:

`>>` get message

`<<` put message

`<-` 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` 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, 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, 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, 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, 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, a slot is private.

10.14 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 realised 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 *Di erence* 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_seteq(+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.

`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 *Di erence* contains all and only the elements of *Set1* which are not also in *Set2*.

`ord_symdiff(+Set1, +Set2, -Difference)`
: is true when *Di erence* 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.

10.15 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. 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. Run `ls` on a home directory in a subshell under UNIX:

```
| ?- absolute_file_name('$SHELL', Shell),
     absolute_file_name('~/', Dir),
     process_create(Shell, ['-c', [ ls, ' ', file(Dir) ]]).
```

Run `notepad.exe` on a file '`C:/foo.txt`' under Windows:

```
| ?- absolute_file_name('$SYSTEMROOT/notepad.exe', Prog),
     process_create(Prog, [file('C:/foo.txt')]).
```

Exported predicates:

`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 argument `access(execute)`) and is used to locate the file to execute.

The predefined file search path `path/1` (see Section 4.5 [ref-fdi], page 80) 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')`, 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 speci cation* 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.

`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:

`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:

`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` 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.

`pipe(`***Stream***`)`

A new Prolog (text) stream is created and connected to the corresponding stream of the new process. It is currently not possible to request binary streams or to specify a character set different from the OS default. This stream must be closed using `close/[1,2]`, it is not closed automatically when the new process exits.

`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]`.

`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.

`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 process 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`, `SIGBUS`, `SIGCHLD`, `SIGCONT`, `SIGFPE`, `SIGHUP`, `SIGILL`, `SIGINT`, `SIGKILL` (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)`.

10.16 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
- recognise 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 recognise 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

```
queue_cons(+Head, +Q0, -Q)      adds Head to Q0 giving Q
queue_cons(-Head, -Q, +Q0)      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(|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

```
append_queue([+X1,...,+Xn], +Q0, -Q) to add X1,...,Xn to Q0 giving Q
append_queue([-X1,...,-Xn], -Q, +Q0) to take X1...Xn from Q0 giving Q
```

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`).

10.17 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.

The state of the random number generator corresponds to a term `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.

Exported predicates:

`getrand(`***-RandomState***`)`
: returns the random number generator's current state

`setrand(+`***RandomState***`)`
: sets the random number generator's state to *RandomState*. If *RandomState* is not a valid random state, it reports an error, and then fails.

`maybe`
: 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.

`maybe(+`***Probability***`)`
: succeeds determinately with probability Probability, fails with probability *1-Probability*. Arguments =< 0 always fail, >= 1 always succeed.

`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 `random(0, N, X), X < P`, but is somewhat faster.

`random(`***-Uniform***`)`
: unifies *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*.

10.18 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.

10.19 Generic Sorting—library(samsort)

This library module provides generic sorting. Exported predicates:

`samsort(+`***RawList***`, -`***Sorted***`)`
: is given 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***`)`
: is given a proper list *RawList* and a binary predicate *Order* (note that it may be an *N-ary* predicate with the first *N-2* arguments already filled in) and unifies *SortedList* with a sorted version of *RawList*. This is only supposed to work when Orderis transitive.

`merge(`***:Order***`, +`***List1***`, +`***List2***`, -`***Merged***`)`
: is like `merge/3` except that it takes an *Order* predicate as its first arguments, like all the generalised ordering routines. samkeysort(+*RawList*, -*Sorted*) is given 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.

`keymerge(+`***List1***`, +`***List2***`, -`***Merged***`)`
: is like `merge/3` except that it compares only the keys of its input lists. Note that it will not work properly when *Merged* is already instantiated.

10.20 Unordered Set Operations—library(sets)

This library module provides operations on sets represented as unordered lists with no repeated elements. 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.

`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*.

`intersection(+ListOfSets, -Intersection)`
: is true when *Intersection* is the intersection of all the sets in *ListOfSets*. The order of elements in *Intersection* is taken from the first set in *ListOfSets*. This has been turned inside out to minimise the storage turnover.

`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 *Di erence*.

`symdiff(+`***`Set1`***`, +`***`Set2`***`, -`***`Difference`***`)`
is true when *Di erence* is the symmetric difference of *Set1* and *Set2*, that is, if each element of *Di erence* 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 for each element *Elt1* of *Set1* and *Elt2* of *Set2*.

`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 command and *Union* is their union.

`union(+`***`Set1`***`, +`***`Set2`***`, -`***`Union`***`)`
is true when `subtract(Set1,Set2,Diff)` and `append(Diff,Set2,Union)`, that is, when *Union* is the elements of *Set1* that do not occur in *Set2*, followed by all the elements of *Set2*.

`union(+`***`Set1`***`, +`***`Set2`***`, -`***`Union`***`, -`***`Difference`***`)`
is true when `union(Set1, Set2, Union)` and `subtract(Set1, Set2, Difference)`.

`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.

10.21 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` 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 necesessarily socket streams) and incoming connections, use `socket_select/7`.

All streams below can be read from as well as written on. 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`***`)`
Creates a stream *Stream* connected to address *Addr*. *Addr* can be:

`Host``:`***`Port`***
Connect to the machine with address *Host* (a host name or host address) at port *Port* (a port number or service name). The *Host* should be an atom, e.g., `'www.sics.se'`. The *Port* is either a port number as an integer or atom, 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.

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`.

To create a binary stream to some web server `www.sics.se`, you would do e.g.,

```
| ?- socket_client_open('www.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(`***?Port***`, `***-ServerSocket***`)`
: Create a server socket *ServerSocket* that listens on port *Port*. Port can be either an integer port number or an atomic service name, see `socket_client_open/3` for details. Port can also be a variable in which case a free port number is used and *Port* is bound to it. 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.

`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 explicitly specified.

`socket_server_close(+`***ServerSocket***`)`
: Close the server socket *ServerSocket* and stop listening on its port.

`socket_select(+`***ServerSockets***`,`***-SReady***`, +`***ReadStreams***`,`***-RReady***`, +`***WriteStreams***`,`***-WReady***`, +`***Timeout***`)`
: Check for server sockets with incoming connections (i.e., ready for `socket_server_accept/4`), streams on *ReadStreams* ready for input, and streams on *WriteStreams* 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*, *ReadStreams*, and *WriteStreams* 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.

10.22 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 isn't possible to guarantee that Prolog can't 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.

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.

10.22.1 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` Long signed integer.

`integer_32`
32 bit signed integer.

`integer_16`
16 bit signed integer.

`integer_8`
8 bit signed integer.

`unsigned` long unsigned integer.

`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 263.

`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 263.

`address` An untyped address. 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 long pointer to a thing of type *Type*.

`array(`***Num***`,T`***ype***`)`
: 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

```
pointer(array(integer_8))
```

is probably a more appropriate declaration of a C string type than

```
pointer(integer_8)
```

which is the orthodox way to declare a string in C.

10.22.1.1 Declaring Types

Programmers may declare new named data structures with the following procedure:

```
:- foreign_type
    Type_name = Type,
    ...,
    Type_name = Type.
```

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 memeory 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.

10.22.2 Checking Foreign Term Types

The type of a foreign term may determined by the goal

`foreign_type(+`***Foreign_term***`, -`***Type_name***`)`

Note that `foreign_type/2` will fail if *Foreign_term* is not a foreign term.

10.22.3 Creating and Destroying Foreign Terms

Prolog can create or destroy foreign terms using

`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. The *Datum* returned by `new/[2,3]` is not initialized in any way. `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, 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(`***Type***`,`***Size***`)`.

10.22.4 Accessing and Modifying Foreign Term Contents

Prolog can get or modify the contents of a foreign term with the procedures

`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

`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, ***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, `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: 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 isn't 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 `get_address/3`.

10.22.5 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:

`cast(+`***Foreign0***`, +`***New_type***`, -`***Foreign***`)`

where ***Foreign*** is the foreign term that is the same data as ***Foreign0***, only is of foreign type ***New_type***. ***Foreign0*** is not affected. This is much like casting in C.

Casting a foreign term to `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 `address` may also be casted to a proper foreign type.

This predicate should be used with great care, as it is quite easy to get into trouble with this.

10.22.6 Null Foreign Terms

"NULL" foreign terms may be handled. The predicate

`null_foreign_term(+`***Term***`, -`***Type***`)`
`null_foreign_term(-`***Term***`, +`***Type***`)`

holds when *Term* is a foreign term of *Type*, but is NULL (the address is 0). At least one of *Term* and *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.

10.22.7 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 `foreign_type/2` before the `foreign/[2,3]` clauses specifying the foreign functions.

The `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 `structs` package is understood, as well as a pointer to any named `structs` type.

For example, if you have a function

```
char nth_char(string, n)
    char *string;
    int n;
    {
        return string[n];
    }
```

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 `nth_char/3` to be called from Prolog to determine the nth character of a C string.

Note that all existing foreign interface type specifications are unaffected, in particular `address/[0,1]` continue to pass addresses to and from Prolog as plain integers.

If you use the foreign resource linker, `splfr`, on a Prolog file that uses the `structs` package, you must pass it the '`--structs`' option. This will make `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 260).

10.22.8 Examining Type Definitions at Runtime

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 *Type* is an atom naming a type, *Definition* is the definition of that type, and *Size* is the number of bytes occupied by a foreign term of this type. *Size* will be the atom `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 *Type* is not bound at call time, these procedures will backtrack through all current type definitions.

A definition looks much like the definition given when the type was defined with `type/1`, except that it has been simplified. Firstly, intermediate type names have been elided. For example, if `foo` is defined as `foo=integer`, and `bar` as `bar=foo`, `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 `pointer(fred)`, it will show up in the definition as `'$fred'`. Of course, `type_definition('$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 10.22.1 [str-fty], page 456 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, these predicates will backtrack through all the currently defined atomic types.

10.22.9 Tips

1. Most important tip: don't 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, 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, 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)`.

10.22.10 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"

long c_sum_array(long cnt, int32 *mem)
{
  int i;
  long sum = 0;

  for (i=0; i<cnt; i++)
    sum += mem[i];
  return sum;
}
```

 # session

```
% splfr --struct minivec.pl minivec.c
% sicstus -l minivec
% compiling /home/matsc/sicstus4/Suite/minivec.pl...
% [...]
% compiled /home/matsc/sicstus4/Suite/minivec.pl in mod-
ule minivec, 30 msec 68388 bytes
SICStus 4.0.2 ...
Licensed to SICS
| ?- new_array(4, A),
     put_array(0,A,1),
     put_array(1,A,10),
     put_array(2,A,100),
     put_array(3,A,1000),
     sum_array(A,S),
     dispose_array(A).
A = array(4,int_array(1264224)),
S = 1111
```

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 long c_sum_array( long, int32 *);
```

10.23 Operating System Utilities—`library(system)`

This package contains utilities for invoking services from the operating system that does not fit elsewhere.

Exported predicates:

`now(`***-When***`)`
: Unifies the current date and time as a UNIX timestamp with *When*.

`datime(`***-Datime***`)`
: Unifies *Datime* with the current date and time as a `datime/6` record of the form `datime(`***Year***`,`***Month***`,`***Day***`,`***Hour***`,`***Min***`,`***Sec***`)`. All fields are integers.

`datime(+`***When***`,`***-Datime***`)`
`datime(`***-When***`,+`***Datime***`)`
: Convert a time stamp, as obtained by `now/1`, to a `datime/6` record. Can be used in both directions.

`sleep(+`***Seconds***`)`
: Puts the SICStus Prolog process asleep for *Second* seconds, where *Seconds* should be a non-negative number.

`environ(`***?Var***`, `***?Value***`)`
: *Var* is the name of an environment variable, and *Value* is its value. Both are atoms. Can be used to enumerate all current environment variables.

10.24 Term Utilities—`library(terms)`

This library module provides miscellaneous operations on terms. Exported predicates:

`subsumeschk(+`***General***`, +`***Specific***`)`
: is true when *Speci c* is an instance of *General*. It does not bind any variables.

`subsumes(+`***General***`, +`***Specific***`)`
: is true when *Speci c* is an instance of *General*. It will bind variables in *General* (but not those in *Speci c*) so that *General* becomes identical to *Speci c*.

`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 generalisation of *Term1* and *Term2*. Using Plotkin's algorithm [Machine Intelligence 5, 1970], extended by Dan Sahlin to handle cyclic structures.

`term_hash(+`***Term***`, `***-Hash***`)`
: If *Term* is ground, an integer hash value corresponding to *Term* is unified with *Hash*. Otherwise, the goal just succeeds.

`term_hash(+`***Term***`, +`***Depth***`, +`***Range***`, `***-Hash***`)`
: If *Term* is instantiated to the given *Depth*, an integer hash value in the range *[0,Range)* corresponding to *Term* is unified with *Hash*. Otherwise, the goal just succeeds.

`term_variables(+`***Term***`, `***-Variables***`)`
: True if *Variables* is the set of variables occurring in *Term*.

`term_variables_bag(+`***Term***`, `***-Variables***`)`
: True if *Variables* is the list of variables occurring in *Term*, in first occurrence order.

`acyclic_term(+X)`
: True if *X* is finite (acyclic). Runs in linear time.

`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(+Term, +Variable, -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
    term_depth(Var) = 0
    term_depth(Const) = 0
    term_depth(F(T1,...,Tn)) = 1+max(term_depth(T1),...,term_depth(Tn))
```

`term_size(+Term, -Size)`
: calculates the *Size* of a *Term*, defined to be the number of constant and function symbol occurrences in it.

`same_functor(?T1, ?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.

`same_functor(?T1, ?T2, ?N)`
: is true when *T1* and *T2* have the same principal functor, and their common arity is *N*. Like `same_functor/3`, at least one of *T1* and *T2* must be bound, or an error will be reported.

`same_functor(?T1, ?T2, ?F, ?N)`
: is true when *T1* and *T2* have the same principal functor, and their common functor 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 arguments are unbound, an error is reported.

10.25 Meta-Call with Limit on Execution Time—`library(timeout)`

Exported predicates:

`time_out(:Goal, +Time, -Result)`
: The *Goal* is executed as if by `call/1`. If computing any solution takes more than *Time* milliseconds, the goal will be aborted and *Result* unified with the atom `time_out`. If the goal succeeds within the specified time, *Result* is unified

with the atom `success`. *Time* must be a number between (not including) 0 and 2147483647.

The time is measured in process virtual time under UNIX. Under Windows NT/2000/XP, as of SICStus 3.10, thread virtual time is used, which is the same as process virtual time for single-threaded processes. Under Windows 98/ME the time is measured in real time (wall time), due to limitations in those operating systems.

The precision of the time out interval is usually not better than several tens of milliseconds. This is due to limitations in the timing mechanisms used to implement '`library(timeout)`'.

`time_out/3` is implemented by raising and handling `time_out` exceptions, so any exception handler in the scope of *Goal* must be prepared to pass on `time_out` exceptions. The following incorrect example shows what can happen otherwise:

```
| ?- time_out(on_exception(Q,(repeat,false),true), 1000, Res).
Q = time_out,
Res = success
```

10.26 Updatable Binary Trees—`library(trees)`

This libary 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 *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* is *NewLabel*. 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.

10.27 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 *typeterm*, as follows:

typeterm	::= `atom`
	| `atomic`
	| `callable`
	| `character`
	| `character_code`
	| `compound`
	| `float`
	| `float(`***rangeterm***`)`
	| `ground`
	| `integer`
	| `integer(`***rangeterm***`)`
	| `list`
	| `list(`***Type***`)`
	| `nonvar`
	| `number`
	| `number(`***rangeterm***`)`
	| `oneof(`***L***`)`
	| `pair`
	| `proper_list`
	| `proper_list(`***Type***`)`
	| `simple`
	| `term`
	| `var`
rangeterm	::= `between(L,U)`
	| `>=(L)`
	| `>(L)`
	| `=\=(L)`

Exported predicates:

`must_be(+Term, +Type, +Goal, +ArgNo)`
: checks whether the *Term* belongs to the indicated *Type*, which should be a *typeterm*. If it isn't, there are two cases: the *Term* may not be instantiated enough to tell yet, in which case an Instantiation Error will be raised, or the *Term* may be definitely not of the type, in which case a Type Error is raised. You should use this in commands with side effects, and should arrange that if this predicate does not succeed the side effect(s) will not take place. If an exception is raised, it will pinpoint the line of code in the scope of which the error occurs, if possible.

`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 175.

  `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.

  `evaluation(ErrorType)`
  : An Evaluation Error is raised.

  `consistency(Culprit1,Culprit2,Message)`
  : A Consistency Error is raised.

  `syntax(Pos,Msg,Tokens,AfterError)`
  : A Syntax Error is raised.

  `resource(Resource)`
  : A Resource Error is raised.

`system(`***Message***`)`
A System Error is raised.

10.28 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 (`==`).

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 list of vertices, *Edges* is a 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*.

`edges(+`***Graph***`, `***-Edges***`)`
unifies *Edges* with the edges in *Graph*.

`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.

`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.

`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.

10.29 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:

`numbervars(+Term)`
: makes *Term* ground by binding variables to subterms `'$VAR'(N)` with values of *N* ranging from 0 up.

`varnumbers(+Term, -Copy)`
: xo succeeds when *Term* was a term producing by calling `numbervars(Term)` and *Copy* is a copy of *Term* with such subterms replaced by variables.

`varnumbers(+Term, +N0, -Copy)`
: succeeds when *Term* was a term produced by calling `numbervars(Term, N0, N)` (so that all subterms `'$VAR'(X)` have `integer(X)`, `X >= N0`) and *Copy* is a copy of *Term* with such subterms replaced by variables.

10.30 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 `keysort/2` with unique keys), the edgelist is a list of *(neighbor-weight)* pair also in standard order (as produced by `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:

`vertices/2`
`edges/2`
`add_vertices/3`
`neighbors/3`
`neighbours/3`
: Re-exported from `library(wgraphs)`.

`wgraph_to_ugraph(+WeightedGraph, -Graph)`
: is true if *Graph* has the same vertices and edges as *WeightedGraph*, except the edges of *Graph* are unweighted.

`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.

`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 list of vertices, *Edges* is a 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 vertices *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.

10.31 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://homepages.tesco.net/binding-time/xml.pl.html

The package, originally written by Binding Time Ltd., is in the public domain and unsupported. To use the package, enter the query:

The package represents XML documents by the abstract data type *document*, which is defined by the following grammar:

<table>
<tr><td>document</td><td>::= <code>xml(attributes,content)</code></td><td>{ well-formed document }</td></tr>
<tr><td></td><td><code>| malformed(attributes,content)</code></td><td>{ malformed document }</td></tr>
<tr><td>attributes</td><td>::= <code>[]</code></td><td></td></tr>
<tr><td></td><td><code>| [name=char-data|attributes]</code></td><td></td></tr>
<tr><td>content</td><td>::= <code>[]</code></td><td></td></tr>
<tr><td></td><td><code>| [cterm|content]</code></td><td></td></tr>
<tr><td>cterm</td><td>::= <code>pcdata(char-data)</code></td><td>{ text }</td></tr>
<tr><td></td><td><code>| comment(char-data)</code></td><td>{ an XML comment }</td></tr>
<tr><td></td><td><code>| namespace(URI,prefix,element)</code></td><td>{ a Namespace }</td></tr>
<tr><td></td><td><code>| element(tagattributes,content)</code></td><td>{ <tag>..</tag> encloses content or <tag /> if empty }</td></tr>
</table>

	| `instructions(`***name***`,`***char-data***`)`	{ A PI <? *name char-data* ?> }
	| `cdata(`***char-data***`)`	{ <![CDATA[*char-data*]]> }
	| `doctype(`***tag***`,`***doctype-id***`)`	{ DTD <!DOCTYPE .. > }
	| `unparsed(`***char-data***`)`	{ text that hasn't been parsed }
	| `out_of_context(`***tag***`)`	{ *tag* is not closed }
tag	::= *atom*	{ naming an element }
name	::= *atom*	{ not naming an element }
URI	::= *atom*	{ giving the URI of a namespace }
char-data	::= *code-list*	
doctype-id	::= `public(`***char-data***`,`***char-data***`)`	
	| `public(`***char-data***`,`***dtd-literals***`)`	
	| `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.

10.32 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, 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:

```
producer :-
       produce(X),
       out(p(X)),
       producer.

produce(X) :- .....
```

In client 2:

```
consumer :-
       in(p(A)),
       consume(A),
       consumer.

consume(A) :- .....
```

Example: Synchronization

```
       ...,
       in(ready),  %Waits here until someone does out(ready)
       ...,
```

Example: A critical region

```
          ...,
          in(region_free),  % wait for region to be free
          critical_part,
          out(region_free), % let next one in
          ...,
```

Example: Reading global data

```
          ...,
          rd(data(Data)),
          ...,

     or, without blocking:
          ...,
          rd_noblock(data(Data)) ->
                do_something(Data)
          ;     write('Data not available!'),nl
          ),
          ...,
```

Example: Waiting for one of several events

```
          ...,
          in([e(1),e(2),...,e(n)], E),
     %  Here is E instantiated to the first tuple that became available
          ...,
```

10.32.1 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

```
| ?- use_module(library('linda/server')).
```

and start the server with `linda/[0,1]`.

`linda`
: Starts a Linda-server in this SICStus. The network address is written to the current output stream as *Host:PortNumber*.

`linda(`***:Options***`)`
: Starts a Linda-server in this SICStus. Each option on the list *Options* is one of

Address-Goal
: where *Address* must be unifiable with *Host:Port* and *Goal* must be instantiated to a goal.

When the linda server is started, *Host* and *Port* are bound to the server host and port respectively and the goal *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 *Options* is not a list, it is assumed to be an option of the form ***Address*-*Goal***.

In SICStus before 3.9.1, *Goal* needed an explicit module prefix to ensure it was called in the right module. This is no longer necessary since `linda/1` is now a meta-predicate.

`accept_hook(`***Client*,*Stream*,*Goal***`)`
: When a client attempts to connects to the server *Client* and *Stream* will be bound to the IP address of the client and the socket stream connected to the client, respectively. The *Goal* is then called, and if it succeeds, the client is allowed to connect. If *Goal* fails, 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, you would probably need something more sophisticated.

Example:

```
| ?- linda([(Host:Port)-mypred(Host,Port),
             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.

10.32.2 Linda Client

The clients are one or more SICStus processes that have connection(s) to the server.

To load the package, enter the query

```
| ?- use_module(library('linda/client')).
```

Some of the following predicates fail if they don't 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 keeps running after receiving this signal, 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.

`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 *Tuple* from Linda's tuple-space if it is there. If not, the predicate blocks until it is available (that is, someone performs an `out/1`).

`in_noblock(`***?Tuple***`)`

Removes the tuple *Tuple* from Linda's tuple-space if it is there. If not, the predicate fails.

This predicate can fail due to a timeout.

`in(+`***TupleList***`, `***?Tuple***`)`

As `in/1` but succeeds when either of the tuples in *TupleList* is available. *Tuple* is unified with the fetched tuple. If that unification fails, the tuple is *not* reinserted in the tuple-space.

`rd(`***?Tuple***`)`

Succeeds if *Tuple* is available in the tuple-space, suspends otherwise until it is available. Compare this with `in/1`: the tuple is *not* removed.

`rd_noblock(`***?Tuple***`)`

Succeeds if *Tuple* is available in the tuple-space, fails otherwise.

This predicate can fail due to a timeout.

`rd(+`***TupleList***`, `***?Tuple***`)`

As `in/2` but does not remove any tuples.

`bagof_rd_noblock(`***?Template***`, +`***Tuple***`, `***?Bag***`)`

Bag is the list of all instances of *Template* such that *Tuple* exists in the tuple-space.

The behavior of variables in *Tuple* and *Template* is as in `bagof/3`. The variables could be existentially quantified with `^/2` as in `bagof/3`.

The operation is performed as an atomic operation.

This predicate can fail due to a timeout.

Example: Assume that only one client is connected to the server and that the tuple-space initially is empty.

```
| ?- 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
```

10.33 Constraint Handling Rules—`library(chr)`

This section is written by Tom Schrijvers, K.U. Leuven, and adjustments by Jan Wielemaker.

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].

10.33.1 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 10.33.2 [CHR Syntax and Semantics], page 483 we present the syntax of CHR in Prolog and explain informally its operational semantics. Next, Section 10.33.3 [CHR in Prolog Programs], page 485 deals with practical issues of writing and compiling Prolog programs containing CHR. Section 10.33.4 [CHR Debugging], page 487 explains the currently primitive CHR debugging facilities. Section 10.33.4.3 [CHR Debugging Predicates], page 488 provides a few useful predicates to inspect the constraint store and Section 10.33.5 [CHR

Examples], page 489 illustrates CHR with two example programs. Finally, Section 10.33.6 [CHR Guidelines], page 490 concludes with a few practical guidelines for using CHR.

10.33.2 Syntax and Semantics

10.33.2.1 Syntax

The syntax of CHR rules is the following:

rules	::= *rule rules*
rules	::= *empty*
rule	::= *name actual_rule pragma* `.`
name	::= *atom* `@`
name	::= *empty*
actual_rule	::= *simpli cation_rule*
actual_rule	::= *propagation_rule*
actual_rule	::= *simpagation_rule*
simpli cation_rule	::= *head* `<=>` *guard body*
propagation_rule	::= *head* `==>` *guard body*
simpagation_rule	::= *head* `\` *head* `<=>` *guard body*
head	::= *constraints*
constraints	::= *constraint constraint_id*
constraints	::= *constraint constraint_id* `,` *constraints*
constraint	::= *compound_term*
constraint_id	::= *empty*
constraint_id	::= `#` *variable*
guard	::= *empty*
guard	::= *goal* `|`
body	::= *goal*
pragma	::= *empty*
pragma	::= `pragma` *actual_pragmas*
actual_pragmas	::= *actual_pragma*
actual_pragmas	::= *actual_pragma* `,` *actual_pragmas*
actual_pragma	::= `passive(`***variable***`)`

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.

10.33.2.2 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 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, the rule is committed and the body of the rule executed. If not all the necessary passive constraint can be found, 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, 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:

simpli cation
: 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* `<=>` *constraints_1*, *body

  Namely, in the simpagation form:

  constraints_1* `\` *constraints_2* `<=>` *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:

`passive(`***Identifier***`)`
: The constraint in the head of a rule *Identi er* 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 `chr_option/2` declaration:

```
:- chr_option(Option,Value).
```

and may appear in the file anywhere after the first constraints declaration.

Available options are:

`check_guard_bindings`
: This option controls whether guards should be checked for (illegal) variable bindings or not. Possible values for this option are `on`, to enable the checks, and `off`, to disable the checks. If this option is on, 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.

`optimize`
: This option controls the degree of optimization. Possible values are `full`, to enable all available optimizations, and `off` (the default), to disable all optimizations. If optimization is enabled, debugging must be disabled.

`debug`
: This options enables or disables the possibility to debug the CHR code. Possible values are `on` (the default) and `off`. See Section 10.33.4 [CHR Debugging], page 487 for more details on debugging.

10.33.3 CHR in Prolog Programs

10.33.3.1 Embedding in Prolog Programs

The CHR constraints defined in a '`.pl`' file are associated with a module. The default module is `user`. One should never load different '`.pl`' files with the same CHR module name.

10.33.3.2 Constraint Declaration

Every constraint used in CHR rules has to be declared with a `chr_constraint/1` declaration by the *constraint speci er*. 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.

```
:- chr_constraint foo/1.
:- chr_constraint bar/2, baz/3.
```

In its extended form, a constraint specifier is ***c*(*A_1*,...,*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.

```
:- chr_constraint get_value(+,?).
:- chr_constraint domain(?int,+list(int)),
                  alldifferent(?list(int)).
```

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), 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:

```
:- 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, you can define an alias as follows:

```
:- chr_type lli == list(list(int)).
```

10.33.3.3 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:

```
:- 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:

```
:- module(zebra, [ zebra/0 ]).
:- use_module(library(chr)).

:- chr_constraint ...
```

10.33.4 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`.

10.33.4.1 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 constraints tries a rule with possibly some passive constraints. The try port is entered just before committing to the rule.

`apply` An active constraints commits to a rule with possibly some passive constraints. The apply port is entered just after committing to the rule.

10.33.4.2 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
        g       ancestors
        n       nodebug
        b       break
        a       abort
        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 toplevel. See `break/0`.

`abort` Exit to the toplevel. See `abort/0`.

`fail` Insert failure in execution.

`help` Print the above available debug options.

10.33.4.3 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 deactivated 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 deactivated 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 10.33.4.1 [CHR Ports], page 487 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), the Prolog toplevel displays the constraint store at the end of each query. If `off`, the toplevel 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.

10.33.5 Examples

Here are two example constraint solvers written in CHR.

1. The program below defines a solver with one constraint, `leq/2`, which is a less-than-or-equal constraint, also known as a partial order constraint.

   ```
   :- 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 `leq/2` constraint in a query, e.g.:

   ```
   | ?- 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.

```
:- 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 `dom/2` constraint in a query, e.g.:

```
| ?- dom(A,[1,2,3]), dom(A,[3,4,5]).
A = 3
```

Finally, Martin Keser's WebCHR package at http://bruckner.informatik.uni-ulm.de/webchr/ contains more than 40 example programs for SICStus 4, complete with documentation and example queries.

10.33.6 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 constraints 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:

```
constraint \ constraint <=> 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.

10.34 Constraint Logic Programming over Finite Domains—library(clpfd)

10.34.1 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:

- Two classes of constraints are handled internally: primitive constraints and global constraints.
- The constraints described in this chapter are automatically translated to conjunctions of primitive and global library constraints.
- The truth value of a primitive constraint can be reflected into a 0/1-variable (reification).
- New primitive constraints can be added by writing so-called indexicals.
- New global constraints can be written in Prolog, by means of a programming interface.

This library fully supports multiple SICStus run-times 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 10.34.2 [CLPFD Interface], page 492. A description of all constraints that the solver provides is contained in Section 10.34.3 [Available Constraints], page 495. The predicates for searching for solution are documented in Section 10.34.4 [Enumeration Predicates], page 513. The predicates for getting execution statistics are documented in Section 10.34.5 [Statistics Predicates], page 516. A few example programs are given in Section 10.34.10 [Example Programs], page 531. Finally, Section 10.34.11 [Syntax Summary], page 535 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 10.34.6 [Answer Constraints], page 516. The solver's execution mechanism and primitives are described in Section 10.34.7 [The Constraint System], page 517. How to add new global constraints via a programming interface is described in Section 10.34.8 [Defining Global Constraints], page 518. How to define new primitive constraints with indexicals is described in Section 10.34.9 [Defining Primitive Constraints], page 524.

10.34.1.1 Referencing this Software

When referring to this implementation of clp(FD) in publications, please use the following reference:

Carlsson M., Ottosson G., Carlson B. *An Open-Ended Finite Domain Constraint Solver*, Proc. Programming Languages: Implementations, Logics, and Programs, 1997.

10.34.1.2 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].

10.34.2 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. Hence, only small integers and unbound variables are allowed in finite domain constraints.

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, an overflow condition is issued. This is expressed as silent failure or as a representation error, subject to the `overflow` option of `fd_flag/3`.

The domain of all variables gets smaller and smaller as more constraints are added. If a domain becomes empty, the accumulated constraints are unsatisfiable, and the current computation branch fails. 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, the garbage collector will preserve all constraint data that is attached to them.

Please note: if a term containing domain variables is copied, asserted, collected as a solution to `findall/3` and friends, or raised as an exception, those domain variables will be replaced by brand new variables in the copy. To retain the domain variables and any attached constraints, you can use `copy_term/3` (see Section 4.8.7 [ref-lte-cpt], page 111).

The heart of the constraint solver is a scheduler for indexicals [Van Hentenryck et al. 92] and global constraints. Both entities act as coroutines performing incremental constraint solving or entailment checking. They wake up by changes in the domains of its arguments. All constraints provided by this package are implemented as indexicals or global constraints. New constraints can be defined by the user.

Indexicals are reactive functional rules, which take part in the solver's basic constraint solving algorithm, whereas each global constraint is associated with its particular constraint solving algorithm. The solver maintains two scheduling queues, giving priority to the queue of indexicals.

The feasibility of integrating the indexical approach with a Prolog based on the WAM was clearly demonstrated by Diaz's clp(FD) implementation [Diaz & Codognet 93], one of the fastest finite domains solvers around.

10.34.2.1 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 not any constraints that may be attached to them.

10.34.2.2 A Constraint Satisfaction Problem

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 pre-defined 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).

```
:- 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
```

10.34.2.3 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 ***rei cation***. Several frequently used operations can be defined in terms of reified constraints, such as blocking implication [Saraswat 90] and the cardinality operator [Van Hentenryck & Deville 91], to name a few. A reified constraint is written:

```
| ?- Constraint #<=> B.
```

where *Constraint* is reifiable. As an example of a constraint that uses reification, consider `exactly(X,L,N)`, which is true if X occurs exactly N times in the list L. It can be defined thus:

```
exactly(_, [], 0).
exactly(X, [Y|L], N) :-
    X #= Y #<=> B,
    N #= M+B,
    exactly(X, L, M).
```

10.34.3 Available Constraints

This section describes the classes of constraints that can be used with this solver.

10.34.3.1 Arithmetic Constraints

?Expr RelOp ?Expr

defines an arithmetic constraint. The syntax for *Expr* and *RelOp* is defined by a grammar (see Section 10.34.11.2 [Syntax of Arithmetic Expressions], page 537). Note that the expressions are not restricted to being linear. Constraints over non-linear expressions, however, will usually yield less constraint propagation than constraints over linear expressions.

Arithmetic constraints can be reified as e.g.:

```
| ?- X in 1..2, Y in 3..5, X#=<Y #<=> B.
B = 1,
X in 1..2,
Y in 3..5
```

Linear arithmetic constraints, except equalities, maintain bound-consistency and their reified versions detect bound-entailment and -disentailment; see Section 10.34.7 [The Constraint System], page 517.

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, 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.

`sum(+Xs, +RelOp, ?Value)`

where *Xs* is a list of integers or domain variables, *RelOp* is a relational symbol as above, and *Value* is an integer or a domain variable. True if `sum(Xs)`

RelOp Value. Cannot be reified. Corresponds roughly to `sumlist/2` in `library(lists)`.

`scalar_product(+`***Coeffs***`, +`***Xs***`, +`***RelOp***`, `***?Value***`)`
`scalar_product(+`***Coeffs***`, +`***Xs***`, +`***RelOp***`, `***?Value***`, +`***Options***`)`

where *Coe s* 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 an integer or a domain variable. True if `sum(`***Coeffs*Xs***`)` ***RelOp Value***. Cannot be reified.

Options is a list that may include the following option. It can be used to control the level of consistency used by the constraint.

`consistency(`***Cons***`)`
The value is one of the following:

`domain` The constraint will maintain arc-consistency. **Please note**: This option is only meaningful if *RelOp* is `#=`, and requires that any domain variables have finite bounds.

`bound`
`value` The constraint will try to maintain bound-consistency (the default).

The following constraints constrain a value to be the minimum (maximum) of a given list of values.

`minimum(`***?Value***`, +`***Xs***`)`
where *Xs* is a list of integers or domain variables, and *Value* is an integer or a domain variable. True if *Value* is the minimum of *Xs*. Cannot be reified. Corresponds to `min_member/2` in `library(lists)`.

`maximum(`***?Value***`, +`***Xs***`)`
where *Xs* is a list of integers or domain variables, and *Value* is an integer or a domain variable. True if *Value* is the maximum of *Xs*. Cannot be reified. Corresponds to `max_member/2` in `library(lists)`.

10.34.3.2 Membership Constraints

`domain(+`***Variables***`, +`***Min***`, +`***Max***`)`
where *Variables* is a list of domain variables or integers, *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***. Cannot be reified.

?X `in +`***Range***
defines a membership constraint. *X* is an integer or a domain variable and *Range* is a *ConstantRange* (see Section 10.34.11.1 [Syntax of Indexicals], page 535). True if *X* is an element of the range.

?X* in_set *+FDSet
: defines a membership constraint. *X* is an integer or a domain variable and ***FDSet*** is an FD set term (see Section 10.34.8.3 [FD Set Operations], page 521). True if *X* is an element of the FD set.

`in/2` and `in_set/2` constraints can be reified. They maintain arc-consistency and their reified versions detect arc-entailment and -disentailment; see Section 10.34.7 [The Constraint System], page 517.

10.34.3.3 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 #= 4 #\/ Y #= 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 10.34.9 [Defining Primitive Constraints], page 524. The following propositional combinators are available:

#\ *:Q*
: True if the constraint *Q* is false.

:P* #/\ *:Q
: True if the constraints *P* and *Q* are both true.

:P* #\ *:Q
: True if exactly one of the constraints *P* and *Q* is true.

:P* #\/ *:Q
: True if at least one of the constraints *P* and *Q* is true.

:P* #=> *:Q
:Q* #<= *:P
: True if the constraint *Q* is true or the constraint *P* is false.

:P* #<=> *:Q
: True if the constraints *P* and *Q* are both true or both false.

Note that the reification scheme introduced in Section 10.34.2.3 [Reified Constraints], page 494 is a special case of a propositional constraint.

10.34.3.4 Combinatorial Constraints

The constraints listed here are sometimes called symbolic constraints. They are currently not reifiable. Unless documented otherwise, they maintain (at most) bound-consistency in their arguments; see Section 10.34.7 [The Constraint System], page 517.

`global_cardinality(+Xs,+Vals)`
`global_cardinality(+Xs,+Vals,+Options)`

where *Xs* = $[X_1,\ldots,X_d]$ is a list of integers or domain variables, and *Vals* = $[K_1 - V_1,\ldots,K_n - V_n]$ is a list of pairs where each key K_i is a unique integer and V_i is a domain variable or an integer. True if every element of *Xs* is equal to some key and for each pair $K_i - V_i$, exactly V_i elements of *Xs* are equal to K_i.

If either *Xs* or *Vals* is ground, and in many other special cases, `global_cardinality/[2,3]` maintains arc-consistency, but generally, bound-consistency cannot be guaranteed. An arc-consistency algorithm [Regin 96] is used, roughly linear in the total size of the domains.

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.

  `bound`
  : 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 the domain bound of a variable is changed;

  `val`
  : to wake up when a variable becomes ground.

`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, 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 $Matrix[1,p_1] + \cdots + Matrix[d,p_d]$.

  With this option, an arc-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.

`element(?X,+List,?Y)`

where *X* and *Y* are integers or domain variables and *List* is a list of integers or domain variables. True if the *X*:th element of *List* is *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 arc-consistency in *X* and bound-consistency in *List* and *Y*. Corresponds to `nth1/3` in `library(lists)`.

`table(+Tuples,+Extension)`
`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 or integers, each also of length *n*. The constraint holds if every *Tuple* in *Tuples* occurs in the *Extension*.

For convenience, *Extension* may contain *ConstantRange* (see Section 10.34.11.1 [Syntax of Indexicals], page 535) expressions in addition to integers.

Options is a list of zero or more of the following. It can be used to control the waking and pruning conditions of the constraint:

`consistency(Cons)`

The value is one of the following:

`domain` The constraint will maintain arc-consistency (the default).

`bound` The constraint will maintain bound-consistency.

`value` The constraint will wake up when a variable has become ground, and only prune a variables when its domain has been reduced to a singleton.

`table/[2,3]` is implemented in terms of the following, more general constraint, with which arbitrary relations can be defined compactly:

`case(+Template, +Tuples, +Dag)`
`case(+Template, +Tuples, +Dag, +Options)`

Template is an arbitrary non-ground Prolog term. 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,Successors)`, where *X* is a place-holder variable. The set of all *X* should equal the set of variables in *Template*. The first node in the list is the *root node*. Let *rootID* denote its ID.

Nodes are either *internal nodes* or *leaf nodes*. In the former case, *Successors* is a list of terms `(Min..Max)-ID2`, where the *ID2* refers to a child node. In the latter case, *Successors* is a list of terms `(Min..Max)`. In both cases, the `(Min..Max)` should form disjoint intervals.

ID is a unique, integer identifier of a node.

Each path from the root node to a leaf node corresponds to one set of tuples admitted by the relation expressed by the constraint. Each variable in *Template* should occur exactly once on each path, and there must not be any cycles.

Options is a list of zero or more of the following. It can be used to control the waking and pruning conditions of the constraint, as well as to identify the leaf nodes reached by the tuples:

`leaves(TLeaf,Leaves)`

TLeaf is a place-holder variable. *Leaves* is a list of variables of the same length as *Tuples*. This option effectively extends the relation by one argument, corresponding to the ID of the leaf node reached by a particular tuple.

`on(`***Spec***`)` Specifies how eagerly the constraint should react to domain changes of *X*.

`prune(`***Spec***`)`
Specifies the extent to which the constraint should prune the domain of *X*.

Spec is one of the following, where *X* is a place-holder variable occurring in *Template* or equal to *TLeaf*:

`dom(`***X***`)` wake up when the domain of *X* has changed, resp. perform full pruning on *X*. This is the default for all variables mentioned in the constraint.

`min(`***X***`)` wake up when the lower bound of *X* has changed, resp. prune only the lower bound of *X*.

`max(`***X***`)` wake up when the upper bound of *X* has changed, resp. prune only the upper bound of *X*.

`minmax(`***X***`)`
wake up when the lower or upper bound of *X* has changed, resp. prune only the bounds of *X*.

`val(`***X***`)` wake up when *X* has become ground, resp. only prune *X* when its domain has been reduced to a singleton.

`none(`***X***`)` ignore domain changes of *X*, resp. never prune *X*.

The constraint holds if *path(rootID,Tuple,Leaf)* holds for each *Tuple* in *Tuples* and *Leaf* is the corresponding element of *Leaves* if given (otherwise, *Leaf* is a free variable).

path(ID,Tuple,Leaf) holds if *Dag* contains a term `node(`***ID,Var,Successors***`)`, *Var* is the unique *k*:th element of *Template*, *i* is the *k*:th element of *Tuple*, and:

The node is an internal node, and

1. *Successors* contains a term `(`***Min***`..`***Max***`)-`***Child***,
2. *Min* ***i*** *Max*, and
3. *path(Child,Tuple,Leaf)* holds; or

The node is a leaf node, and

1. *Successors* contains a term `(`***Min***`..`***Max***`)`,
2. *Min* ***i*** *Max*, and *Leaf* = *ID*.

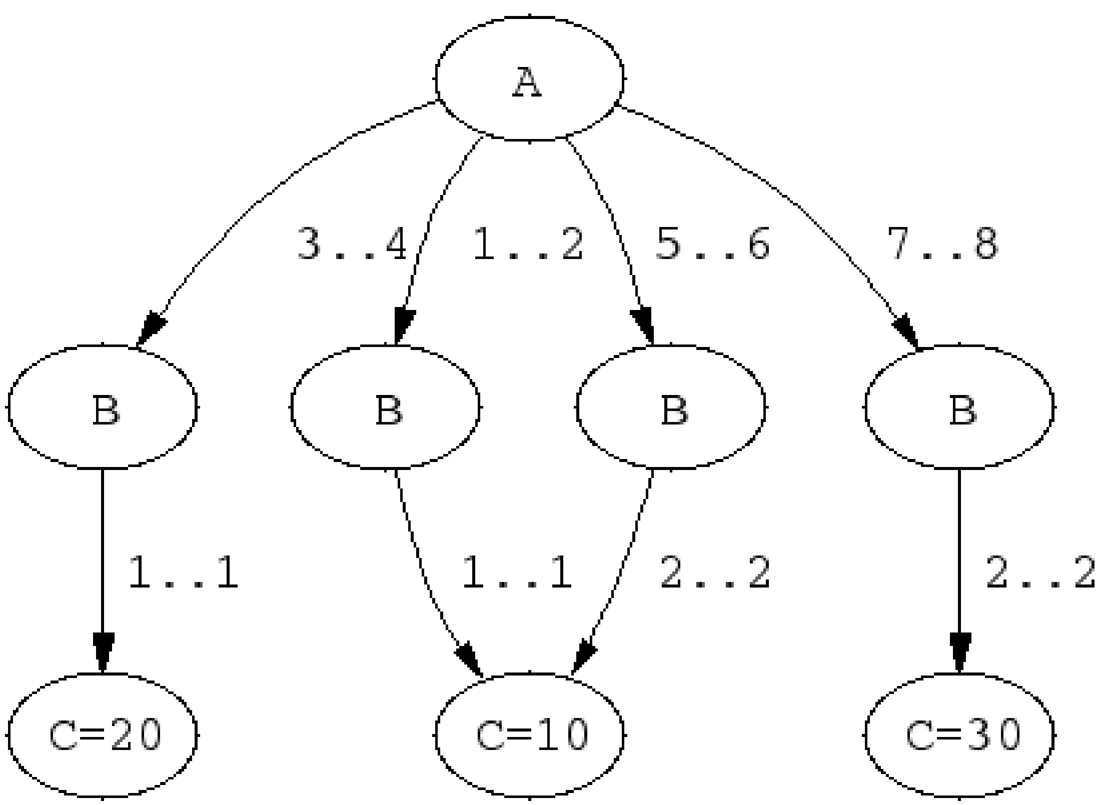
For example, recall that `element(`***X,L,Y***`)` wakes up when the domain of *X* or the lower or upper bound of *Y* has changed, performs full pruning of *X*, but only prunes the bounds of *Y*. The following two constraints:

```
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 as well as wrt. pruning and waking. The fourth argument illustrates the leaf feature:

```
elts(X, Y, Z, L) :-
    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)])],
         [on(dom(A)),on(minmax(B)),on(minmax(C)),
          prune(dom(A)),prune(minmax(B)),prune(minmax(C)),
          leaves(_,[L])]).
```

The DAG of the previous example has the following shape:



DAG corresponding to `elts/4`

A couple of sample queries:

```
| ?- elts(X, Y, Z, L).
L in 5..7,
X in 1..8,
Y in 1..2,
Z in 10..30

| ?- elts(X, Y, Z, L), Z #>= 15.
L in 6..7,
X in(3..4)\/(7..8),
Y in 1..2,
Z in 20..30

| ?- elts(X, Y, Z, L), Y = 1.
Y = 1,
L in 5..6,
X in 1..4,
Z in 10..20

| ?- elts(X, Y, Z, L), L = 5.
Z = 10,
X in(1..2)\/(5..6),
Y in 1..2
```

`all_different(+`***Variables***`)`
`all_different(+`***Variables***`, +`***Options***`)`
`all_distinct(+`***Variables***`)`
`all_distinct(+`***Variables***`, +`***Options***`)`

where *Variables* is a list of domain variables or integers. 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.

Options is a list of zero or more of the following:

`consistency(`***Cons***`)`
Which algorithm to use, one of the following:

`domain` The default for `all_distinct/[1,2]` and `assignment/[2,3]`. an arc-consistency algorithm [Regin 94] is used, roughly linear in the total size of the domains. Implies `on(dom)`.

`bound` a bound-consistency algorithm [Mehlhorn 00] is used. This algorithm is nearly linear in the number of variables and values. 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 becomes ground.

`nvalue(`***?N***`, +`***Variables***`)`

where *Variables* is a list of domain variables with finite bounds or integers, and N is an integer or a domain variable. True if N is the number of distinct values taken by *Variables*. Approximates bound-consistency in N and arc-consistency in *Variables*. Can be thought of as a relaxed version of `all_distinct/2`.

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 $Xs = [X_1,\ldots,X_n]$ and $Ys = [Y_1,\ldots,Y_n]$ are lists of domain variables or integers. True if all $X_i, Y_i \in [1,n]$ and $X_i = j \Leftrightarrow Y_j = i$.

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`, `circuit(`***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 $n \times n$ matrix, represented as a list of lists. The cost of the constraint is $Matrix[1,X_1] + \cdots + Matrix[n,X_n]$.

With this option, an arc-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 can be thought of as constraining n nodes in a graph to form a Hamiltonian circuit. The nodes are numbered from 1 to n. The circuit starts in node 1, visits each node, and returns to the origin.

`circuit(+Succ)`
`circuit(+Succ, +Pred)`

where *Succ* is a list of length n of domain variables or integers. The i:th element of *Succ* (*Pred*) is the successor (predecessor) of i in the graph. True if the values form a Hamiltonian circuit.

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:

`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, or integers.

Let L be a global resource limit (by default 1, but see below), and:

$$a = \min(S_1; :::; S_n)$$
$$b = \max(S_1 + D_1; :::; S_n + D_n)$$
$$H_{ij} = \begin{cases} H_j; \text{ if } S_j \quad i < S_j + D_j \\ 0; \text{ otherwise} \end{cases}$$

The constraint holds if:

1. For every task i, *Si+Di=Ei*.
2. $$8a \quad i < b : H_{i1} + \quad + H_{in} \quad L$$

Options is a list of zero or more of the following, where *Boolean* must be `true` or `false` (`false` is the default, except for the `bounds_only` option):

`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

`Ti-Tj #= Dij`

where *Ti* and *Tj* should be task identifiers, and *Dij* should be a a domain variable (or an integer), denoting:

$$S_i \quad S_j = D_{ij} \wedge D_{ij} \; 2 \; r$$

`global(Boolean)`

if `true`, 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.

`cumulatives(+`***Tasks***`,+`***Machines***`)`
`cumulatives(+`***Tasks***`,+`***Machines***`,+`***Options***`)`

A task is represented by a term `task(`O_i, D_i, E_i, H_i, M_i`)` where O_i is the start time, D_i the duration, E_i the end time, H_i the resource consumption, and M_i a machine identifier.

A machine is represented by a term `machine(`M_j, L_j`)` where M_j is the identifier and L_j is the resource limit of the machine.

All fields are domain variables with bounded domains, or integers. L_j must be an integer. D_i must be non-negative, but H_i may be either positive or negative. Negative resource consumption is interpreted as resource production.

Options is a list of zero or more of the following, where *Boolean* must be `true` or `false` (`false` is the default):

`bound(`***B***`)` If `lower` (the default), each resource limit is treated as a lower bound. If `upper`, each resource limit is treated as an upper bound.

`prune(`***P***`)` If `all` (the default), the constraint will try to prune as many variables as possible. If `next`, only variables that occur in the first non-ground task term (wrt. the order given when the constraint was posted) can be pruned.

`generalization(`***Boolean***`)`
If `true`, extra reasoning based on assumptions on machine assignment will be done to infer more.

`task_intervals(`***Boolean***`)`
If `true`, extra global reasoning will be performed in an attempt to infer more.

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 $Xs = [X_1, \ldots, X_n]$, $Ps = [P_1, \ldots, P_n]$, and $Ys = [Y_1, \ldots, Y_n]$ are lists of domain variables or integers. The constraint holds if the following are true:

Ys is in ascending order.

Ps is a permutation of $[1, n]$.

$\forall i \in [1, n] : X_i = Y_{(P_i)}$

In practice, the underlying algorithm [Mehlhorn 00] is likely to achieve bound-consistency, and is guaranteed to do so if *Ps* is ground or completely free.

The following constraints model a set or lines or rectangles, respectively, so that no pair of objects overlap:

`disjoint1(+Lines)`
`disjoint1(+Lines,+Options)`

where *Lines* is a list of terms $F(S_j,D_j)$ or $F(S_j,D_j,T_j)$, S_j and D_j are domain variables with finite bounds or integers denoting the origin and length of line j respectively, F is any functor, and the optional T_j is an atomic term denoting the type of the line. T_j 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):

`decomposition(Boolean)`
: if `true`, an attempt is made to decompose the constraint each time it is resumed.

`global(Boolean)`
: if `true`, a redundant algorithm using global reasoning is used to achieve more complete pruning.

`wrap(Min,Max)`
: If used, 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(`T_1`,`T_2`,`D`)`
: This option imposes a minimal distance D between the end point of any line of type T_1 and the origin of any line of type T_2. D should be a positive integer or `sup`. If `sup` is used, all lines of type T_2 must be placed before any line of type T_1.

  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.

The file `library('clpfd/examples/bridge.pl')` contains an example where `disjoint1/2` is used for scheduling non-overlapping tasks.

`disjoint2(+Rectangles)`
`disjoint2(+Rectangles,+Options)`

where *Rectangles* is a list of terms $F(S_{j1},D_{j1},S_{j2},D_{j2})$ or $F(S_{j1},D_{j1},S_{j2},D_{j2},T_j)$, S_{j1} and D_{j1} are domain variables with finite bounds or integers denoting the origin and size of rectangle j in the X dimension, S_{j2} and D_{j2} are the values for the Y dimension, F is any functor, and the optional T_j is an atomic term denoting the type of the rectangle. T_j 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):

`decomposition(`***Boolean***`)`
: If `true`, an attempt is made to decompose the constraint each time it is resumed.

`global(`***Boolean***`)`
: If `true`, a redundant algorithm using global reasoning is used to achieve more complete pruning.

`wrap(`***Min1***`,`***Max1***`,`***Min2***`,`***Max2***`)`
: *Min1* and *Max1* should be either integers or the atoms `inf` and `sup` respectively. If they are integers, 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 $\boldsymbol{S}(\boldsymbol{j}_1)$ 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, 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 $\boldsymbol{S}(\boldsymbol{j}_2)$ variables to be inside *[Min2,Max2-1]*.

: If all four are integers, the space is a toroid wrapping around both dimensions.

`margin(`$\boldsymbol{T}_1$`,`$\boldsymbol{T}_2$`,`$\boldsymbol{D}_1$`,`$\boldsymbol{D}_2$`)`
: This option imposes minimal distances $\boldsymbol{D}_1$ in the X dimension and $\boldsymbol{D}_2$ in the Y dimension between the end point of any rectangle of type $\boldsymbol{T}_1$ and the origin of any rectangle of type $\boldsymbol{T}_2$. $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ should be positive integers or `sup`. If `sup` is used, all rectangles of type $\boldsymbol{T}_2$ must be placed before any rectangle of type $\boldsymbol{T}_1$ 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***`)`
: Let the *assignment dimension* and the *temporal dimension* denote the two dimensions, no matter which is the X and which is the Y dimension. If *Boolean* is `true`, a redundant algorithm is used to achieve more complete pruning for the following case:

  All rectangles have size 1 in the assignment dimension.

  Some rectangles have the same origin and size in the temporal dimension, and that origin is not yet fixed.

: The following example shows an artificial placement problem involving 25 rectangles including four groups of rectangles whose left and right borders must be aligned. If `Synch` is `true`, it can be solved with first-fail labeling in 23 backtracks. If `Synch` is `false`, 60 million backtracks do not suffice to solve it.

```
ex([O1,Y1a,Y1b,Y1c,
    O2,Y2a,Y2b,Y2c,Y2d,
    O3,Y3a,Y3b,Y3c,Y3d,
    O4,Y4a,Y4b,Y4c],
   Synch) :-
        domain([Y1a,Y1b,Y1c,
                Y2a,Y2b,Y2c,Y2d,
                Y3a,Y3b,Y3c,Y3d,
                Y4a,Y4b,Y4c], 1, 5),
        O1 in 1..28,
        O2 in 1..26,
        O3 in 1..22,
        O4 in 1..25,
        disjoint2([t(1,1,5,1),    t(20,4,5,1),
                   t(1,1,4,1),    t(14,4,4,1),
                   t(1,2,3,1),    t(24,2,3,1),
                   t(1,2,2,1),    t(21,1,2,1),
                   t(1,3,1,1),    t(14,2,1,1),
                   t(O1,3,Y1a,1),
                   t(O1,3,Y1b,1),
                   t(O1,3,Y1c,1),
                   t(O2,5,Y2a,1),
                   t(O2,5,Y2b,1),
                   t(O2,5,Y2c,1),
                   t(O2,5,Y2d,1),
                   t(O3,9,Y3a,1),
                   t(O3,9,Y3b,1),
                   t(O3,9,Y3c,1),
                   t(O3,9,Y3d,1),
                   t(O4,6,Y4a,1),
                   t(O4,6,Y4b,1),
                   t(O4,6,Y4c,1)],
                  [synchronization(Synch)]).
```

The file `library('clpfd/examples/squares.pl')` contains an example where `disjoint2/2` is used for tiling squares.

The following constraints express the fact that several vectors of domain variables are in ascending lexicographic order:

`lex_chain(+`***`Vectors`***`)`
`lex_chain(+`***`Vectors`***`,+`***`Options`***`)`

where *Vectors* is a list of vectors (lists) of domain variables with finite bounds or integers. The constraint holds if *Vectors* are in ascending lexicographic order.

Options is a list of zero or more of the following:

`op(`***`Op`***`)` If *Op* is the atom `#=<` (the default), the constraints holds if *Vectors* are in non-descending lexicographic order. If *Op* is the atom `#<`, the constraints holds if ***Vectors*** are in strictly ascending lexicographic order.

`increasing`
: This option imposes the additional constraint that each vector in ***Vectors*** be sorted in strictly ascending order.

`among(`***`Least`***`,`***`Most`***`,`***`Values`***`)`
: If given, ***Least*** and ***Most*** should be integers such that 0 *Least* *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***.

Unless the `increasing/0` or `among/3` options are given, the underlying algorithm [Carlsson & Beldiceanu 02] guarantees arc-consistency.

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, extended with counter operations on its arcs. The point is that it is very much easier to come up with such a checker that to come up with a filtering algorithm for the constraint of interest.

`automaton(...)`
: The constraint has the form ***ctr*** according to the grammar shown below, which describes its abstract syntax. The arguments are:

sequence
: The sequence of terms of interest.

template
: A template for an item of the sequence. 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***. The automaton is not driven by the ***sequence*** itself, but by ***signature***, which ranges over an alphabet, defined in the following argument. In addition to `automaton/8`, you must call a constraint that maps ***sequence*** to ***signature***.

nodes
: The nodes of the automaton, classified as source, sink or internal.

arcs
: The arcs (transitions) of the automaton. 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
: For ***k*** counters, a list of ***k*** variables.

initial
: For ***k*** counters, a list of ***k*** initial values, usually instantiated.

nal
: For ***k*** counters, a list of ***k*** final values, usually uninstantiated.

Abstract syntax:

ctr ::= `automaton(`***`sequence`***`, `***`template`***`, `***`signature`***`,`

	nodes, ***arcs***, ***counters***, ***initial***, ***final***)	
sequence	::= *list of template*	{all of which of the same shape}
template	::= *term*	{most general shape of the *sequence*} {its variables should be local to *ctr*}
signature	::= *list of variable*	
nodes	::= *list of nodespec*	{all of which of the same shape}
nodespec	::= `source(`***node***`)`	{the initial state}
	| `sink(`***node***`)`	{an accept state}
node	::= *atomic*	
arcs	::= *list of arc*	{all of which of the same shape}
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)*	
	| *(conditional ; conditional)*	
exprs	::= *list of Expr*	{of same length as *counters*} {*Expr* as defined in Section 10.34.11.2 [Syntax of Arithmetic Expressions], page 537} {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 *ctr*}
initial	::= *list of integer*	{of same length as *counters*}
nal	::= *list of variable*	{of same length as *counters*}

If no counters are used, the arguments *counters*, *initial* and *nal* 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:

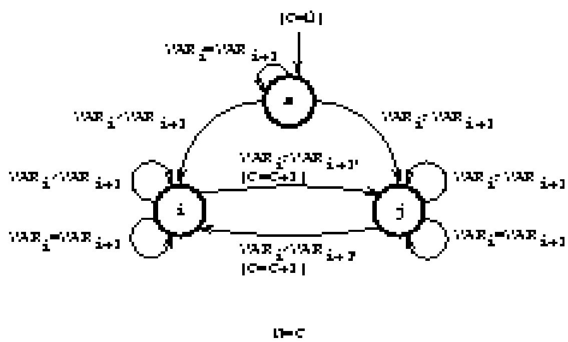
signature is the signature corresponding to *sequence*.

The finite automaton encoded by ***nodes*** and ***arcs*** stops in an accept state.

Any counter arithmetic on the transitions map their ***initial*** values to the ***nal*** values.

Here is an example. Suppose that you want to define the predicate `inflexion(`***N***`,`***L***`)` 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/2`

The following piece of code encodes this using `automaton/8`. 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/2`). Two transitions increment the counter. All states are accept states.

```
inflexion(N, VARIABLES) :-
        inflexion_signature(VARIABLES, SIGNATURE),
        automaton(_, _, SIGNATURE,
                  [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]).

inflexion_signature([], []).
inflexion_signature([_], []) :- !.
inflexion_signature([VAR1,VAR2|VARs], [S|Ss]) :-
        S in 0..2,
        VAR1 #> VAR2 #<=> S #= 0,
        VAR1 #= VAR2 #<=> S #= 1,
        VAR1 #< VAR2 #<=> S #= 2,
        inflexion_signature([VAR2|VARs], Ss).
```

A couple of queries:

```
| ?- inflexion(N, [1,1,4,8,8,2,7,1]).
N = 3 ? <RET>
yes

| ?- length(L,4), domain(L,0,1), inflex-
ion(2,L), labeling([],L).
L = [0,1,0,1] ? ;
L = [1,0,1,0] ? ;
no
```

This constraint was introduced in [Beldiceanu, Carlsson & Petit 04].

10.34.3.5 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 10.34.8 [Defining Global Constraints], page 518. 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 predicates***; see Section 10.34.9 [Defining Primitive Constraints], page 524. They can optionally be reified.

10.34.4 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 or an integer. Assigns, in increasing order via backtracking, a feasible value to *X*.

`labeling(`***:Options***`, +`***Variables***`)`
: where *Variables* is a list of domain variables or integers 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***`)`
`maximize(`***:Goal***`,`***?X***`)`
: Uses a branch-and-bound algorithm with restart 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, at which time *Goal* and *X* are unified with values corresponding to the optimal solution.

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), and whether all solutions or a single, optimal solution should be found. The options are divided into four groups. One option may be selected per group. Also, the number of assumptions (choices) made during the search can be collected. Finally, a discrepancy limit can be imposed.

The following options control the order in which the next variable is selected for assignment.

`leftmost`
: The leftmost variable is selected. This is the default.

`min`
: The leftmost variable with the smallest lower bound is selected.

`max`
: The leftmost variable with the greatest upper bound is selected.

`ff`
: The first-fail principle is used: the leftmost variable with the smallest domain is selected.

`ffc`
: The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by (a) selecting the variable

that has the most constraints suspended on it and (b) selecting the leftmost one.

`variable(`***Sel***`)`
: *Sel* is a predicate to select the next variable. Given *Vars*, the variables that remain to label, it will be called as *Sel(Vars,Selected,Rest)*.

  Sel is expected to succeed determinately, unifying *Selected* and *Rest* with the selected variable and the remaining list, respectively.

  Sel should be a callable term, optionally with a module prefix, and the arguments *Vars,Selected,Rest* will be appended to it. For example, if *Sel* is `mod:sel(Param)`, it will be called as `mod:sel(Param,Vars,Selected,Rest)`.

The following options control the way in which choices are made for the selected variable *X*:

`step`
: Makes a binary choice between ***X*** `#=` ***B*** and ***X*** `#\=` ***B***, where *B* is the lower or upper bound of *X*. This is the default.

`enum`
: Makes a multiple choice for *X* corresponding to the values in its domain.

`bisect`
: Makes a binary choice between ***X*** `#=<` ***M*** and ***X*** `#>` ***M***, where *M* is the midpoint of the domain of *X*. This strategy is also known as domain splitting.

`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)`, 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 control whether all solutions should be enumerated by backtracking or whether a single solution that minimizes (maximizes) *X* is returned, if one exists.

`all` All solutions are enumerated. This is the default.

`minimize(`***X***`)`
`maximize(`***X***`)`
: Uses a branch-and-bound algorithm to find 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 this option with the `time_out/2` option (see below).

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.

A limit on the discrepancy of the search 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.

Finally, a time limit on the search can be imposed:

`time_out(`***Time***`,`***Flag***`)`
: This is equivalent to a goal `time_out(labeling(...),Time,Flag)` (see Section 10.25 [lib-timeout], page 467). Furthermore, if combined with the `minimize(V)` or `maximize(V)` option, and the time limit is reached, the values of *Variables* and *V* will be those of the best solution found.

For example, to enumerate solutions using a static variable ordering, use:

```
| ?- constraints(Variables),
     labeling([], Variables).
     %same as [leftmost,step,up,all]
```

To minimize a cost function using branch-and-bound search, a dynamic variable ordering using the first-fail principle, and domain splitting exploring the upper part of domains first, use:

```
| ?- constraints(Variables, Cost),
     labeling([ff,bisect,down,minimize(Cost)], Variables).
```

The file `library('clpfd/examples/tsp.pl')` contains an example of user-defined variable and value choice heuristics.

10.34.5 Statistics Predicates

The following predicates can be used to get execution statistics.

`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 `statistics/2`.

For each of the possible keys *Key*, *Value* is unified with the current value of a counter, which is simultaneously zeroed. The following counters are maintained. See Section 10.34.7 [The Constraint System], page 517, for details of what they all mean:

`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 signalling failure. Other causes of backtracking, such as failed Prolog tests, are not covered by this counter.

`constraints`
: The number of constraints created.

`fd_statistics`

Displays on the standard error stream a summary of the above statistics. All counters are zeroed.

10.34.6 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:

`clpfd:full_answer` ***hook***

If false (the default), the answer constraint, as well as constraints projected by `clpfd:project_attributes/2`, `clpfd:attribute_goal/2` and their callers, only contain the domains of the query variables. If true, those constraints contain the domains and any attached constraints of all variables. Initially defined as a dynamic predicate with no clauses.

10.34.7 The Constraint System

10.34.7.1 Definitions

The constraint system is based on domain constraints and indexicals. A ***domain constraint*** is an expression $X::I$, where X is a domain variable and I is a nonempty set of integers.

A set S of domain constraints is called a ***store***. $D(X,S)$, the ***domain*** of X in S, is defined as the intersection of all I such that $X::I$ belongs to S. The store is ***contradictory*** if the domain of some variable is empty; otherwise, it is ***consistent***. A consistent store S' is an ***extension*** of a store S iff, for all variables X, $D(X,S')$ is contained in $D(X,S)$.

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 ***arc-consistent wrt.*** S iff, for each variable X_i and value V_i in $D(X_i,S)$, there exist values V_j in $D(X_j,S)$, $1 \le j \le n \wedge i \neq j$, such that $C(V_1,\ldots,V_n)$ is true.

A constraint C is ***arc-entailed by*** S iff, for all values V_j in $D(X_j,S)$, $1 \le j \le n$, $C(V_1,\ldots,V_n)$ is true.

Let $D'(X,S)$ denote the interval $[\min(D(X,S)),\max(D(X,S))]$.

A constraint C is ***bound-consistent wrt.*** S iff, for each variable X_i, there exist values V_j and W_j in $D'(X_j,S)$, $1 \le j \le n, i \neq j$, such that $C(V_1,\ldots,\min(D(X_i,S)),\ldots,V_n)$ and $C(W_1,\ldots,\max(D(X_i,S)),\ldots,W_n)$ are both true.

A constraint C is ***bound-entailed by*** S iff, for all values V_j in $D'(X_j,S)$, $1 \le j \le n$, $C(V_1,\ldots,V_n)$ is true.

Finally, a constraint is ***arc-disentailed (bound-disentailed) by*** S iff its negation is arc-entailed (bound-entailed) by S.

10.34.7.2 Pitfalls of Interval Reasoning

In most circumstances, arithmetic constraints maintain bound-consistency and detect bound-entailment and -disentailment. There are cases where a bound-consistency maintaining constraint may detect a contradiction when the constraint is not yet bound-disentailed, as the following example illustrates. Note that `X #\= Y` maintains arc-consistency if both arguments are constants or variables:

```
| ?- X+Y #= Z, X=1, Z=6, Y in 1..10, Y #\= 5.
no
| ?- X+Y #= Z #<=> B, X=1, Z=6, Y in 1..10, Y #\= 5.
X = 1,
Z = 6,
Y in(1..4)\/(6..10),
B in 0..1
```

Since `1+5#=6` holds, `X+Y #= Z` is not bound-disentailed, although any attempt to make it bound-consistent wrt. the store results in a contradictory store.

10.34.8 Defining Global Constraints

10.34.8.1 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:

`clpfd:dispatch_global(+`***Constraint***`, +`***State0***`, `***-State***`, `***-Actions***`)` ***extendible***

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`.

State0 and *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, *Actions* should be unified with a list of requests to the solver. Each request should be of the following form:

`exit` The constraint has become entailed, and ceases to exist.

`fail` The constraint has become disentailed, causing the solver to backtrack.

X `=` ***V*** The solver binds *X* to *V*.

X `in` ***R*** The solver constrains *X* to be a member of the *ConstantRange R* (see Section 10.34.11.1 [Syntax of Indexicals], page 535).

X `in_set` ***S***

The solver constrains *X* to be a member of the FD set *S* (see Section 10.34.8.3 [FD Set Operations], page 521).

`call(`***Goal***`)`
: The solver calls the goal or constraint *Goal*, which should be module prefixed unless it is a built-in predicate or an exported predicate of the `clpfd` module.

  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.

`fd_global(`***:Constraint***`, +`***State***`, +`***Susp***`)`
`fd_global(`***:Constraint***`, +`***State***`, +`***Susp***`, +`***Options***`)`
: where *Constraint* is a constraint goal, *State* is its initial state, and *Susp* is a term encoding how the constraint should wake up in response to domain changes. This predicate posts the constraint.

  Susp is a list of *F(Var)* terms where *Var* is a variable to suspend on and *F* is a functor encoding when to wake up:

  `dom(`***X***`)` wake up when the domain of *X* has changed

  `min(`***X***`)` wake up when the lower bound of *X* has changed

  `max(`***X***`)` wake up when the upper bound of *X* has changed

  `minmax(`***X***`)`
  wake up when the lower or upper of *X* has changed

  `val(`***X***`)` wake up when *X* has become ground

  Options is a list of zero or more of the following:

  `source(`***Term***`)`
  By default, the symbolic form computed by `copy_term/3`, and shown in the answer constraint if `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`, 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), 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`, 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, the solver will no longer trust the constraint solving method to be idempotent.

For an example of usage, see Section 10.34.8.4 [A Global Constraint Example], page 523.

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.

`debug` Controls the visibility of constraint propagation. Possible values are `on` and `off` (the default). For iternal use by `library(fdbg)`.

10.34.8.2 Reflection Predicates

The constraint solving method needs access to information about the current domains of variables. This is provided by the following predicates, which are all constant time operations.

`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 (or an integer). *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 (or an integer). *Max* is unified with the upper bound of the current domain of *X*, i.e. an integer or the atom `sup` denoting infinity.

`fd_size(?X, ?Size)`

where *X* is a domain variable (or an integer). *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 (or an integer). *Set* is unified with an FD set term denoting the internal representation of the current domain of *X*; see below.

`fd_dom(?X, ?Range)`

where *X* is a domain variable (or an integer). *Range* is unified with a *ConstantRange* (see Section 10.34.11.1 [Syntax of Indexicals], page 535) denoting the the current domain of *X*.

`fd_degree(?X, ?Degree)`

where *X* is a domain variable (or an integer). *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.

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 variables that can be reached from *Var* via constraints posted so far.

`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 posted so far. Thus, `fd_closure/2` is the transitive closure of `fd_neighbors/2`.

10.34.8.3 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. 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 10.34.11.1 [Syntax of Indexicals], page 535) *Range*.

`fdset_to_range(+Set, -Range)`
: *Range* is a constant interval, a singleton constant set, or a union of such, denoting 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, *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)`
: *Di erence* 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`.

10.34.8.4 A 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 integers and 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 10.34.2.3 [Reified Constraints], page 494.

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  ↦  exactly(I, X[1]...X[i-1],X[i+1]...X[m], N-1):
[2] |= X[i]\=I ↦  exactly(I, X[1]...X[i-1],X[i+1]...X[m], N):
[3] |= N=0     ↦  X[1]\=I ... X[m]\=I
[4] |= N=m     ↦  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,N0), state(Xs,N), Actions) :-
        exactly_solver(I, Xs0, Xs, N0, N, Actions).

exactly_solver(I, Xs0, Xs, N0, N, Actions) :-
        ex_filter(Xs0, Xs, N0, N, I),
        length(Xs, M),
        (   N=:=0 -> Actions = [exit|Ps], ex_neq(Xs, I, Ps)
        ;   N=:=M -> Actions = [exit|Ps], ex_eq(Xs, I, Ps)
        ;   N>0, N<M -> Actions = []
        ;   Actions = [fail]
        ).
```

```
                                                         % exactly.pl
% 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
```

10.34.9 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.

10.34.9.1 Indexicals

An *indexical* is a reactive functional rule of the form X `in` R, where R is a set valued *range expression* (see below). See Section 10.34.11.1 [Syntax of Indexicals], page 535, 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. When a propagating indexical fires, R is evaluated in the current store S, which is extended by adding the new domain constraint $X::S(R)$ to the store, where $S(R)$ denotes the value of R in S. When a checking indexical fires, it checks if $D(X;S)$ is contained in $S(R)$, and if so, the constraint corresponding to the indexical is detected as entailed.

10.34.9.2 Range Expressions

A range expression has one of the following forms, where R_i denote range expressions, T_i denote integer valued *term expressions*, $S(T_i)$ denotes the integer value of T_i in S, X denotes a variable, I denotes an integer, and S denotes the current store.

`dom(`X`)` evaluates to $D(X;S)$

`{`T_1; nldots; T_n`}`
: evaluates to `{`$S(T_1)$;:::;$S(T_n)$`}`. Any T_i containing a variable that is not "quantified" by `unionof/3` will cause the indexical to suspend until this variable has been assigned.

T_1`..`T_2 evaluates to the interval between $S(T_1)$ and $S(T_2)$.

R_1`/\`R_2 evaluates to the intersection of $S(R_1)$ and $S(R_2)$

R_1`\/`R_2 evaluates to the union of $S(R_1)$ and $S(R_2)$

`\`R_2 evaluates to the complement of $S(R_2)$

R_1`+`R_2
R_1`+`T_2 evaluates to $S(R_2)$ or $S(T_2)$ added pointwise to $S(R_1)$

`-`R_2 evaluates to $S(R_2)$ negated pointwise

R_1`-`R_2
R_1`-`T_2
T_1`-`R_2 evaluates to $S(R_2)$ or $S(T_2)$ subtracted pointwise from $S(R_1)$ or $S(T_1)$

R_1 `mod` R_2
R_1 `mod` T_2 evaluates to $S(R_1)$ pointwise modulo $S(R_2)$ or $S(T_2)$

R_1 `?` R_2 evaluates to $S(R_2)$ if $S(R_1)$ is a non-empty set; otherwise, evaluates to the empty set. This expression is commonly used in the context `(`R_1 `? (inf..sup) \/` R_3`)`, which evaluates to $S(R_3)$ if $S(R_1)$ is an empty set; otherwise, evaluates to `inf..sup`. As an optimization, R_3 is not evaluated while the value of R_1 is a non-empty set.

`unionof(`X`,`R_1`,`R_2`)`
: evaluates to the union of $S(E_1);:::;S(E_N)$, 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 10.34.10.2 [N Queens], page 532, for an example of usage.

  Please note: if $S(R_1)$ is infinite, 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 `unionof/3`, the indexical will suspend until this variable has been assigned.

10.34.9.3 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.

`min(`X`)` evaluates to the minimum of $D(X;S)$

`max(`X`)` evaluates to the maximum of $D(X;S)$

`card(`X`)` evaluates to the size of $D(X;S)$

X evaluates to the integer value of X. The indexical will suspend until X is assigned.

I an integer

`inf` minus infinity

`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 quotient of $S(T_1)$ and $S(T_2)$, rounded up, where $S(T_2)$ must be positive

T_1`/<`T_2 evaluates to the quotient of $S(T_1)$ and $S(T_2)$, rounded down, where $S(T_2)$ must be positive

T_1 `mod` T_2 evaluates to the modulo of $S(T_1)$ and $S(T_2)$

10.34.9.4 Monotonicity of Indexicals

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 10.34.9.6 [Execution of Propagating Indexicals], page 529 and Section 10.34.9.7 [Execution of Checking Indexicals], page 530.

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.

10.34.9.5 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 bound-consistency, which is significantly cheaper to maintain than arc-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 what domain constraints should be added to the constraint store when the constraint is posted. Therefore the FD predicate contains a set of indexicals, each representing a domain constraint to be added to the constraint store. The actual domain constraint depends on the constraint store itself. For example, the third indexical in the above FD predicate prescribes the domain constraint '`T :: 3..13`' if the store contains '`X :: 1..5, Y :: 2..8`'. As the domain of some variables gets smaller, the indexical may enforce a new, stricter constraint on some other variables. Therefore such an indexical (called a propagating indexical) can be viewed as an agent 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 agent does not enforce any constraints. When the indexical becomes evaluable the resulting domain constraint is added to the store. The agent then waits and reacts to changes in the domains of variables occurring in the indexical by re-evaluating it and adding the new, stricter constraint to the store. Eventually the computation reaches a phase when no further refinement of the store can result in a more precise constraint (the indexical is entailed by the store), and then the agent can cease to exist.

A necessary condition for the FD predicate to be correctly defined is the following: for any store mapping each variable to a singleton domain the execution of the indexicals should succeed without contradiction exactly when the predicate is intended to be true.

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 arc-consistency:

```
plusd(X,Y,T) +:
        X in dom(T) - dom(Y),
        Y in dom(T) - dom(X),
        T in dom(X) + dom(Y).

| ?- X in {1}\/{3}, Y in {10}\/{20}, plusd(X, Y, T).
X in{1}\/{3},
Y in{10}\/{20},
T in{11}\/{13}\/{21}\/{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, 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, 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 clause consists of propagating indexicals for solving *C*.

Head `-:` ***Indexicals***`.`
: The clause consists of propagating indexicals for solving the negation of *C*.

Head `+?` ***Indexical***`.`
: The clause consists of a single checking indexical for testing entailment of *C*.

Head `-?` ***Indexical***.
: The clause consists of a single checking indexical for testing entailment of the negation of *C*.

When a constraint is reified, the solver spawns two reactive agents 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 agent 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 agent is dismissed. The waiting is implemented by means of the coroutining facilities of SICStus Prolog.

As an example of a constraint with all methods defined, consider the following library constraint defining a disequation between two domain variables:

```
'x\\=y'(X,Y) +:
        X in \{Y},
        Y in \{X}.
'x\\=y'(X,Y) -:
        X in dom(Y),
        Y in dom(X).
'x\\=y'(X,Y) +?
        X in \dom(Y).
'x\\=y'(X,Y) -?
        X in {Y}.
```

The following sections provide more precise coding rules and operational details for indexicals. ***X*** `in` ***R*** denotes an indexical corresponding to a constraint ***C***. ***S*** denotes the current store.

10.34.9.6 Execution of Propagating Indexicals

Consider the definition of a constraint ***C*** containing a propagating indexical ***X*** `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, ***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*** `in` ***R*** thus involves the following:

- If ***D***(***X;S***) is disjoint from ***S***(***R***), a contradiction is detected.
- If ***D***(***X;S***) is contained in ***S***(***R***), ***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, ***X*** :: ***S***(***R***) is added to the store (***X*** 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 `dom(`***Y***`)` or `card(`***Y***`)` in ***R*** has been updated
2. the lower bound of a variable ***Y*** that occurs as `min(`***Y***`)` in ***R*** has been updated
3. the upper bound of a variable ***Y*** that occurs as `max(`***Y***`)` in ***R*** has been updated

10.34.9.7 Execution of Checking Indexicals

Consider the definition of a constraint ***C*** containing a checking indexical ***X*** `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, ***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*** `in` ***R*** thus involves the following:

If ***D***(***X;S***) is contained in ***S***(***R***), 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***, 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 `dom(`***Y***`)` or `card(`***Y***`)` in ***R*** has been pruned
3. the lower bound of a variable ***Y*** that occurs as `min(`***Y***`)` in ***R*** has been increased
4. the upper bound of a variable ***Y*** that occurs as `max(`***Y***`)` in ***R*** has been decreased

10.34.9.8 Goal Expanded Constraints

The arithmetic, membership, and propositional constraints described earlier are transformed at compile time to conjunctions of goals of library constraints.

Although space economic (linear in the size of the source code), the expansion of a constraint to library goals can have an overhead compared to expressing the constraint in terms of indexicals. Temporary variables holding intermediate values may have to be introduced, and the grain size of the constraint solver invocations can be rather small. The translation of constraints to library goals has been greatly improved in the current version, so these problems have virtually disappeared. However, for backward compatibility, an implementation by compilation to indexicals of the same constraints is also provided.

The following two constructions are semantically equivalent:

```
Head +: LinExpr RelOp LinExpr.

Head :- LinExpr RelOp LinExpr.
```

where *Head* only contains unique variables mentioned in the linear arithmetic expressions *LinExpr* (see Section 10.34.11.1 [Syntax of Indexicals], page 535). The alternative version of `sum/8` in Section 10.34.10.1 [Send More Money], page 532 illustrates this technique.

Similarly, the following two constructions are semantically equivalent:

```
Pred(X,Y) +: element(X, CList, Y).

Pred(X,Y) :- element(X, CList, Y).
```

where *CList* is a ground list of integers. This technique is used in some demo programs in `library('clpfd/examples')`. **Please note**: the generated indexical will assume that the domains of *X* and *Y* do not contain values incompatible with *CList*.

Similarly, the following two constructions are semantically equivalent:

```
Pred(X1,...,Xn) +: table(CTable).

Pred(X1,...,Xn) :- table([[X1,...Xn]], CTable).
```

where *CTable* is an *n*-ary relation given by extension, as for the constraint `table/[2,3]`. **Please note**: the generated indexical will assume that the domains of *X1,...Xn* do not contain values incompatible with *CTable*.

In the body of an FD predicate, `element/3` and `table/1` expressions expand to indexicals recursively built up from `switch/2` and `unionof/3` expressions. For example, the following constraint:

```
p(X, Y) +: table([[1,1],[2,1..2],[3,1..3]]).
```

expands to:

```
q(X, Y) +:
        X in unionof(B,dom(Y),switch(B,[1-{1,2,3},2-{2,3},3-{3}])),
        Y in unionof(B,dom(X),switch(B,[1-{1},2-{1,2},3-{1,2,3}])).
```

10.34.10 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')).
| ?- bench.
```

10.34.10.1 Send More Money

Let us return briefly to the Send More Money problem (see Section 10.34.2.2 [A Constraint Satisfaction Problem], page 493). Its sum/8 predicate will expand to a space-efficient conjunction of library constraints. A faster but more memory consuming version is defined simply by changing the neck symbol of sum/8 from ':-' to '+:', thus turning it into an FD predicate:

```
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.
```

10.34.10.2 N Queens

The problem is to place N queens on an NxN 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 no_threat/3 constraint and holds between all pairs (X,Y) of queens. It could be defined as

```
no_threat(X, Y, I) :-
        X   #\= Y,
        X+I #\= Y,
        X-I #\= Y.
```

However, this formulation introduces new temporary domain variables and creates twelve fine-grained indexicals. Worse, the disequalities only maintain bound-consistency and so may miss some opportunities for pruning elements in the middle of domains.

A better idea is to formulate 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, 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 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 `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.

```
:- 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} \/ {Y+I} \/ {Y-I}),
     Y in \({X} \/ {X+I} \/ {X-I}).

/*
% version 2: strong but very inefficient version
no_threat(X, Y, I) +:
    X in unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
    Y in unionof(B,dom(X),\({B} \/ {B+I} \/ {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} \/ {B+I} \/ {B-I})),
    Y in (4..card(X)) ? (inf..sup) \/
          unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
*/

| ?- queens(8, L, [ff]).
L = [1,5,8,6,3,7,2,4]
```

10.34.10.3 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:

Task	*Duration*	*Resource*
`t1`	16	2
`t2`	6	9
`t3`	13	3
`t4`	7	7
`t5`	5	10
`t6`	18	1
`t7`	4	11

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].

```
:- use_module(library(clpfd)).
:- use_module(library(lists), [append/3]).

schedule(Ss, End) :-
        length(Ss, 7),
        Ds = [16, 6,13, 7, 5,18, 4],
        Rs = [ 2, 9, 3, 7,10, 1,11],
        domain(Ss, 1, 30),
        domain([End], 1, 50),
        after(Ss, Ds, End),
        cumulative(Ss, Ds, Rs, 13),
        append(Ss, [End], Vars),
        labeling([minimize(End)], Vars). % label End last

after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).

%% End of file

| ?- schedule(Ss, End).
Ss = [1,17,10,10,5,5,1],
End = 23
```

10.34.11 Syntax Summary

10.34.11.1 Syntax of Indexicals

X	::= *variable*	{ domain variable }
Constant	::= *integer*	
	| `inf`	{ minus infinity }
	| `sup`	{ plus infinity }
Term	::= *Constant*	
	| *X*	{ suspend until assigned }

	| `min(`***X***`)`	{ min. of domain of X }
	| `max(`***X***`)`	{ max. of domain of X }
	| `card(`***X***`)`	{ size of domain of X }
	| `-` *Term*	
	| *Term* `+` *Term*	
	| *Term* `-` *Term*	
	| *Term* `*` *Term*	
	| *Term* `/>` *Term*	{ division rounded up }
	| *Term* `/<` *Term*	{ division rounded down }
	| *Term* `mod` *Term*	
TermSet	::= `{`***Term***`,...,`***Term***`}`	
Range	::= *TermSet*	
	| `dom(`***X***`)`	{ domain of X }
	| *Term* `..` *Term*	{ interval }
	| *Range* `/\` *Range*	{ intersection }
	| *Range* `\/` *Range*	{ union }
	| `\` *Range*	{ complement }
	| `-` *Range*	{ pointwise negation }
	| *Range* `+` *Range*	{ pointwise addition }
	| *Range* `-` *Range*	{ pointwise subtraction }
	| *Range* `mod` *Range*	{ pointwise modulo }
	| *Range* `+` *Term*	{ pointwise addition }
	| *Range* `-` *Term*	{ pointwise subtraction }
	| *Term* `-` *Range*	{ pointwise subtraction }
	| *Range* `mod` *Term*	{ pointwise modulo }
	| *Range* `?` *Range*	
	| `unionof(`***X***`,`***Range***`,`***Range***`)`	
	| `switch(`***Term***`,`***MapList***`)`	
ConstantSet	::= `{`***integer***`,...,`***integer***`}`	
ConstantRange	::= *ConstantSet*	
	| *Constant* `..` *Constant*	
	| *ConstantRange* `/\` *ConstantRange*	
	| *ConstantRange* `\/` *ConstantRange*	
	| `\` *ConstantRange*	
MapList	::= `[]`	
	| `[`***integer***`-`***ConstantRange***`|`***MapList***`]`	
CTable	::= `[]`	
	| `[`***CRow***`|`***CTable***`]`	
CRow	::= `[]`	
	| `[`***integer***`|`***CRow***`]`	
	| `[`***ConstantRange***`|`***CRow***`]`	
CList	::= `[]`	
	| `[`***integer***`|`***CList***`]`	

Indexical	::= *X* `in` *Range*	
Indexicals	::= *Indexical*	
	| *Indexical* `,` *Indexicals*	
ConstraintBody	::= *Indexicals*	
	| *LinExpr RelOp LinExpr*	
	| `element(`***X***`,`***CList***`,`***X***`)`	
	| `table(`***CTable***`)`	
Head	::= *term*	{ a compound term with unique variable args }
TellPos	::= *Head* `+:` *ConstraintBody*	
TellNeg	::= *Head* `-:` *ConstraintBody*	
AskPos	::= *Head* `+?` *Indexical*	
AskNeg	::= *Head* `-?` *Indexical*	
ConstraintDef	::= *TellPos*.	
	?(*TellNeg*.)	
	?(*AskPos*.)	
	?(*AskNeg*.)	

10.34.11.2 Syntax of Arithmetic Expressions

X	::= *variable*	{ domain variable }
N	::= *integer*	
LinExpr	::= *N*	{ linear expression }
	| *X*	
	| *N* `*` *X*	
	| *N* `*` *N*	
	| *LinExpr* `+` *LinExpr*	
	| *LinExpr* `-` *LinExpr*	
Expr	::= *LinExpr*	
	| *Expr* `+` *Expr*	
	| *Expr* `-` *Expr*	
	| *Expr* `*` *Expr*	
	| *Expr* `/` *Expr*	{ integer division }
	| *Expr* `mod` *Expr*	
	| `min(`***Expr***`,`***Expr***`)`	
	| `max(`***Expr***`,`***Expr***`)`	
	| `abs(`***Expr***`)`	
RelOp	::= `#=` | `#\=` | `#<` | `#=<` | `#>` | `#>=`	

10.34.11.3 Operator Declarations

```
:- op(1200, xfx, [+:,-:,+?,-?]).
:- op(760, yfx, #<=>).
:- op(750, xfy, #=>).
:- op(750, yfx, #<=).
:- op(740, yfx, #\/).
:- op(730, yfx, #\).
:- op(720, yfx, #/\).
:- op(710, fy, #\).
:- op(700, xfx, [in,in_set]).
:- op(700, xfx, [#=,#\=,#<,#=<,#>,#>=]).
:- op(550, xfx, ..).
:- op(500, fy, \).
:- op(490, yfx, ?).
:- op(400, yfx, [/>,/<]).
```

10.35 Constraint Logic Programming over Rationals or Reals—`library([clpq,clpr])`

10.35.1 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.

10.35.1.1 Referencing this Software

When referring to this implementation of clp(Q,R) in publications, you should use the following reference:

> Holzbaur C., ***OFAI clp(q,r) Manual***, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

10.35.1.2 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.

10.35.2 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 145). You can avoid the interactive resolution dialog if the importation is skipped, e.g. via: `use_module(library(clpq),[]),use_module(library(clpr),[])`.

10.35.2.1 Notational Conventions

Throughout this chapter, the prompts `clp(q) ?-` and `clp(r) ?-` are used to differentiate between clp(Q) and 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 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 10.35.5.1 [CLPQR Variable Ordering], page 552. The approximative nature of floating point numbers may also produce numerical differences between the text in this manual and the actual results of clp(R), for a given edition of the software.

10.35.2.2 Solver Predicates

The solver interface for both Q and R consists of the following predicates, which are exported from `module(linear)`.

`{+`***Constraint***`}`

Constraint is a term accepted by the 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 clp(Q) and clp(R) were loaded

```
| ?- clpr:{Ar+Br=10}, Ar=Br, clpq:{Aq+Bq=10}, Aq=Bq.

Aq = 5,
Ar = 5.0,
Bq = 5,
Br = 5.0
```

Although clp(Q) and clp(R) are independent modules, you are asking for trouble if you (accidently) share variables between them:

```
| ?- 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
```

This is because both solvers eventually compute values for the variables and Reals are incompatible with Rationals.

Here is the constraint grammar:

```
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 }
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 10.35.7 [CLPQR Monash Examples], page 558.

`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=\=5).

{A=<4}

clp(q) ?- {A =< 4}, entailed(A=\=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, *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, *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.

```
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:

```
minimize(Expr) :- inf(Expr, Expr).
```

`maximize(+Expr)`

Computes the supremum of the linear expression *Expr* and equates it with the expression.

```
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 10.35.8 [CLPQR MIP], page 558.

```
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: `abs(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 *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 `minimize/1`, where the general solution is exhibited.

`bb_inf/5` works properly for non-strict inequalities only! Disequations (`=\=`) and higher dimensional strict inequalities (`>`,`<`) are beyond its scope. Strict bounds on the decision variables are honored however:

```
clp(q) ?- {X >= Y+Z, Y > 1, Z > 1}, bb_inf([Y,Z],X,Inf,Vertex,0).

Inf = 4,
Vertex = [2,2],
{Y>1},
{Z>1},
{X-Y-Z>=0}
```

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 >= 1}` for integral `X` and `Y`;
- contemplating the use of clp(FD).

`ordering(+Spec)`

Provides a means to control one aspect of the presentation of the answer constraints; see Section 10.35.5.1 [CLPQR Variable Ordering], page 552.

`dump(+Target, -NewVars, -CodedAnswer)`

Reflects the constraints on the target variables into a term, where *Target* and *NewVars* are lists of variables of equal length and *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 *NewVars* (see Section 10.35.5.2 [CLPQR Turning Answers into Terms], page 553).

```
clp(q) ?- {A+B =< 10, A>=4},
          dump([A,B],Vs,Cs),
          dump([B],Bp,Cb).

Cb = [_A=<6],
Bp = [_A],
Cs = [_B>=4,_C+_B=<10],
Vs = [_C,_B],
{A>=4},
{A+B=<10}
```

The current version of `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 `dump/3` currently has no means to collect global constraints . . .

```
q(X) :-
        {X>=10},
        {sin(Z)>3}.

clp(r) ?- q(X), dump([X],V,C).

C = [_A>=10.0],
V = [_A],
clpr:{3.0-sin(_B)<0.0},
{X>=10.0}
```

`projecting_assert/1(`***:Clause***`)`

If you use the database, the clauses you assert might have constraints associated with their variables. Use this predicate instead of `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.

10.35.2.3 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.

```
clp(r) ?- {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

```
clp(r) ?- {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 clp(R) coincides. In general, equations involving interpreted functors, `+/2` in this case, must be fed to the solver explicitly:

```
clp(r) ?- X=3.0+1.0, X=4.0.

no
```

Further, variables known by clp(R) may be bound directly to floats only. Likewise, variables known by clp(Q) may be bound directly to rational numbers only; see Section 10.35.9.1 [CLPQR Fragments and Bits], page 561. Failing to do so is rewarded with an exception:

```
clp(q) ?- {2*A+3*B=C/2}, C=10.0, A=B.
! Type error in argument 2 of = /2
! 'a rational number' expected, but 10.0 found
! goal:  _254=10.0
```

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`, you don't have to worry about such details.

10.35.2.4 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 10.35.5 [CLPQR Projection], page 551.

10.35.3 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, 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, 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 systems 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*.

```
clp(r) ?- {sin(cos(X)) = 1/2}.

X = 1.0197267436954502
```

If we change the equation into an inequation, clp(Q,R) gives up:

```
clp(r) ?- {sin(cos(X)) < 1/2}.

clpr:{sin(cos(X))-0.5<0.0}
```

The following is easy again:

```
clp(r) ?- {sin(X+2+2)/sin(4+X) = Y}.

Y = 1.0
```

And so is this:

```
clp(r) ?- {(X+Y)*(Y+X)/X = Y*Y/X+99}.

{Y=49.5-0.5*X}
```

An ancient symbol manipulation benchmark consists in rising the expression `X+Y+Z+1` to the 15th power:

```
clp(q) ?- {exp(X+Y+Z+1,15)=0}.
clpq:{Z^15+Z^14*15+Z^13*105+Z^12*455+Z^11*1365+Z^10*3003+...
          ... polynomial continues for a few pages ...
          =0}
```

Computing its roots is another story.

10.35.3.1 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:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.

clpq:{Y*2-X^2*2+Y*X*2+X*2+1=0}
```

Equating *X* and *Y* collapses the expression completely and even determines the values of the two variables:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.

X = -1/4,
Y = -1/4
```

10.35.3.2 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 * 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 = abs(Y)`
: Residuates unless *Y* is ground.

`X = pow(Y,Z)`, `X = exp(Y,Z)`
: Residuates unless any pair of two of the three variables is ground. Example:

```
clp(r) ?- { 12=pow(2,X) }.

X = 3.5849625007211565

clp(r) ?- { 12=pow(X,3.585) }.

X = 1.9999854993443926

clp(r) ?- { X=pow(2,3.585) }.

X = 12.000311914286545
```

`X = sin(Y)`
: Residuates unless *X* or *Y* is ground. Example:

```
clp(r) ?- { 1/2 = sin(X) }.

X = 0.5235987755982989
```

`X = cos(Y)`
: Residuates unless *X* or *Y* is ground.

`X = tan(Y)`
: Residuates unless *X* or *Y* is ground.

10.35.4 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, 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.

% library('clpqr/examples/root')

```
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 = 1572584048032918633353217/1111984844349868137938112

clp(q) ?- root(7,R),print_decimal(R,70).
1.4142135623 7309504880 1688724209 6980785696 7187537694 8073176679
7379907324

R = 4946041176255201878775086487573351061418968498177 /
    3497379255757941172020851852070562919437964212608
```

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.

% 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`.

```
clp(q) ?- e(1000,E).

E = 7149056228932760213666809592072842334290744221392610955845565494
    3708750229467761730471738895197792271346693089326102132000338192
    0131874187833985420922688804220167840319199699494193852403223700
    5853832741544191628747052136402176941963825543565900589161585723
    4023097417605004829991929283045372355639145644588174733401360176
    9953973706537274133283614740902771561159913069917833820285608440
    3104966899999651928637634656418969027076699082888742481392304807
    9484725489080844360397606199771786024695620205344042765860581379
    3538290451208322129898069978107971226873160872046731879753034549
    3130492167474809196348846916421782850086985668680640425192038155
    4902863298351349469211627292865440876581064873866786120098602898
    8799130098877372097360065934827751120659213470528793143805903554
    7928682131082164366007016698761961066948371407368962539467994627
    1374858249110795976398595034606994740186040425117101588480000000
    0000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000000000000000000000
    /
    2629990810403002651095959155503002285441272170673105334466808931
    6863103901346024240326549035084528682487048064823380723787110941
    6809235187356318780972302796570251102928552003708556939314795678
    1978390674393498540663747334079841518303636625888963910391440709
    0887345797303470959207883316838346973393937778363411195624313553
    8835644822353659840936818391050630360633734935381528275392050975
    7271468992840907541350345459011192466892177866882264242860412188
    0652112744642450404625763019639086944558899249788084559753723892
    1643188991444945360726899532023542969572584363761073528841147012
    2634218045463494055807073778490814692996517359952229262198396182
    1838930043528583109973872348193806830382584040536394640895148751
    0766256738740729894909630785260101721285704616818889741995949666
    6303289703199393801976334974240815397920213059799071915067856758
    6716458821062645562512745336709063396510021681900076680696945309
    3660590933279867736747926648678738515702777431353845466199680991
    73361873421152165477774911660108200059
```

The decimal expansion itself looks like this:

```
clp(q) ?- e(1000, E), print_decimal(E, 1000).
2.
7182818284 5904523536 0287471352 6624977572 4709369995 9574966967
6277240766 3035354759 4571382178 5251664274 2746639193 2003059921
8174135966 2904357290 0334295260 5956307381 3232862794 3490763233
8298807531 9525101901 1573834187 9307021540 8914993488 4167509244
7614606680 8226480016 8477411853 7423454424 3710753907 7744992069
5517027618 3860626133 1384583000 7520449338 2656029760 6737113200
7093287091 2744374704 7230696977 2093101416 9283681902 5515108657
4637721112 5238978442 5056953696 7707854499 6996794686 4454905987
9316368892 3009879312 7736178215 4249992295 7635148220 8269895193
6680331825 2886939849 6465105820 9392398294 8879332036 2509443117
3012381970 6841614039 7019837679 3206832823 7646480429 5311802328
7825098194 5581530175 6717361332 0698112509 9618188159 3041690351
5988885193 4580727386 6738589422 8792284998 9208680582 5749279610
4841984443 6346324496 8487560233 6248270419 7862320900 2160990235
3043699418 4914631409 3431738143 6405462531 5209618369 0888707016
7683964243 7814059271 4563549061 3031072085 1038375051 0115747704
1718986106 8739696552 1267154688 9570350354
```

10.35.5 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:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=12.682503013196973*_A-11.682503013196971*P},
{Mp= -(_A)+1.01*P},
{_B=2.01*_A-1.01*P},
{_C=3.0301*_A-2.0301*P},
{_D=4.060401000000001*_A-3.0604009999999997*P},
{_E=5.101005010000001*_A-4.10100501*P},
{_F=6.152015060100001*_A-5.152015060099999*P},
{_G=7.213535210701001*_A-6.213535210700999*P},
{_H=8.285670562808011*_A-7.285670562808009*P},
{_I=9.368527268436091*_A-8.36852726843609*P},
{_J=10.462212541120453*_A-9.46221254112045*P},
{_K=11.566834666531657*_A-10.566834666531655*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.

10.35.5.1 Variable Ordering

In general, there are many ways to express the same linear relationship between variables. clp(Q,R) does not care to distinguish between them, but the user might. The predicate `ordering(+Spec)` gives you some control over the variable ordering. Suppose that instead of *B*, you want *Mp* to be the defined variable:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

This is achieved with:

```
clp(r) ?- mg(P,12,0.01,B,Mp), ordering([Mp]).

{Mp= -0.0788487886783417*B+0.08884878867834171*P}
```

One could go one step further and require *P* to appear before (to the left of) *B* in an addition:

```
clp(r) ?- mg(P,12,0.01,B,Mp), ordering([Mp,P]).

{Mp=0.08884878867834171*P-0.0788487886783417*B}
```

Spec in `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, `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.

```
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp).

{B= -12.682503013196973*Mp+1.1268250301319698*P}

clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp), ordering(P < Mp).

{P=0.8874492252651537*B+11.255077473484631*Mp}
```

10.35.5.2 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:

```
clp(r) ?- {2*A+B+C=10,C-
D=E,A<10}, dump([A,B,C,D,E],[a,b,c,d,e],Constraints).

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}
```

10.35.5.3 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')
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

% 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:

```
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:

```
  |
2 -    *    *
  |
  |
1 -    *
  |
  |
0 -----|----*----*----
       1    2    3
```

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.

10.35.6 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):

% 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,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61] ? <RET>
```

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,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61] ? <RET>
```

10.35.7 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
```

10.35.8 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:

```
NAME:         flugpl
ROWS:         18
COLUMNS:      18
INTEGER:      11
NONZERO:      46
BEST SOLN:    1201500 (opt)
LP SOLN:      1167185.73
SOURCE:       Harvey M. Wagner
              John W. Gregory (Cray Research)
              E. Andrew Boyd (Rice University)
APPLICATION:  airline model
COMMENTS:     no integer variables are binary
```

% library('clpqr/examples/mip')

```
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]
```

10.35.9 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].

10.35.9.1 Fragments and Bits

Rationals. The internal data structure for rational numbers is `rat(`***Num***`,`***Den***`)`. *Den* is always positive, i.e. the sign of the rational number is the sign of *Num*. Further, *Num* and *Den* are relative prime. Note that integer *N* looks like `rat(`***N***`,1)` in this representation. You can control printing of terms with `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 runtime. The mechanism to realize this idea is to use `dump/3` for the expansion of `{}/1`, via the goal and term expansion hook predicates.

Asserting with Constraints. If you use the database, the clauses you assert might have constraints associated with their variables. You should use `projecting_assert/1` instead of `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).
test(A, B) :-
        {A+B=<rat(33,1)}

| ?- test(A,B).

{A+B=<33}
```

10.35.9.2 Bugs

The fuzzy comparison of floats is the source for all sorts of weirdness. If a result in R surprises you, 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.

10.36 Finite Domain Constraint Debugger—`library(fdbg)`

10.36.1 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.

FDBG defines the following prefix operator:

```
:- op(400, fy, #).
```

The presence of FDBG affects the translation and execution, but not the semantics, of subsequently loaded arithmetic constraints.

10.36.2 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.

10.36.2.1 Events

An FDBG event can (currently) belong to one of the two following major classes:

constraint event
: A global 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 10.36.2.3 [FDBG Visualizers], page 563) gets called with a representation of the event (a Prolog term) as extra arguments.

Note that it is *not possible* to debug indexicals with FDBG. What's more, any domain narrowings done by indexicals happen unnoticed, making FDBG output harder to follow. On the other hand, arithmetical constraints (like `X #> 0`) are translated to global constraints instead of indexicals after consulting `library(fdbg)`, and therefore don't lead to any misunderstandings. For this latter reason it is advisable to load `library(fdbg)` *before* any user programs.

10.36.2.2 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 10.36.2.1 [FDBG Events], page 562.

10.36.2.3 Visualizers

A visualizer is a Prolog predicate reacting to FDBG events (see Section 10.36.2.1 [FDBG Events], page 562). 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 10.36.3.1 [FDBG Options], page 565), 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 10.36.3.1 [FDBG Options], page 565) is the following:

for global constraint awakenings: `fdbg_show`

for labeling events: `fdbg_label_show`

For details on built-in visualizers, see Section 10.36.3.3 [FDBG Built-In Visualizers], page 567.

10.36.2.4 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 10.36.2.6 [FDBG Name Auto-Generation], page 564.

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 10.36.4.2 [FDBG Writing Visualizers], page 573.

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 10.36.2.3 [FDBG Visualizers], page 563.

10.36.2.5 Selectors

A ***selector*** is a Prolog term denoting a (path to a) subterm of a given term ***T***. Let ***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	*subterm(T,S)*
`[]`	*T*
`[...,`***N***`]`	*N*th argument of the compound term ***subterm(T,`[...]`)***
`[...,#`***N***`]`	*N*th element of the list ***subterm(T,`[...]`)***

10.36.2.6 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:

   ```
   fdbg_assign_name(bar(A, [B, C], foobar(D, E)), foo)
   ```

   will create the following name/term entries:

Name	*Term/Variable*	*Selector*
`foo`	`bar(A, [B, C], foobar(D, E))`	`[]`
`foo_1`	`A`	`[1]`
`foo_2_1`	`B`	`[2,#1]`
`foo_2_2`	`C`	`[2,#2]`
`foo_3_1`	`D`	`[3,1]`
`foo_3_2`	`E`	`[3,2]`

   See Section 10.36.3.2 [FDBG Naming Terms], page 567.
2. If, during the annotation of a term (see Section 10.36.3.5 [FDBG Annotation], page 569) an unnamed constraint variable is found, 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.

10.36.2.7 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 10.36.3.5 [FDBG Annotation], page 569), and doesn't 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 10.36.4.1 [FDBG Customizing Output], page 572. The default format of the legend is the following:

```
list_2 = 0..3
list_3 = 0..3
list_4 = 0..3
fdvar_2 = 0..3 -> 1..3
```

Here, we see four variables, with initial domains 0..3, but the domain of the (previously unnamed) variable `fdvar_2` is narrowed by the constraint (not shown here) to 1..3.

A legend is automatically printed by the built-in visualizer `fdbg_show`, but it can be easily printed from user defined visualizers too.

10.36.2.8 The `fdbg_output` Stream

The `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.

10.36.3 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 `fdbg_show` could be too detailed and therefore hard to analyze. See Section 10.36.4 [FDBG Advanced Usage], page 572.

10.36.3.1 FDBG Options

FDBG is switched on and off with the predicates:

`fdbg_on`
`fdbg_on(:Options)`

Turns on FDBG by putting advice-points on several predicates of the CLP(FD) module. *Options* is a single option or a list of options; see Section 10.36.3.1 [FDBG Options], page 565. The empty list is the default value.

`fdbg_on/[0,1]` can be called safely several times consecutively; only the first call will have an effect.

`fdbg_off` Turns the debugger off by removing the previously installed advice-points.

`fdbg_on/1` accepts the following options:

`file(`***Filename***`,` ***Mode***`)`
: Tells FDBG to attach the stream alias `fdbg_output` to the file called *Filename* opened in mode *Mode*. *Mode* can either be `write` or `append`. The file specified is opened on a call to `fdbg_on/1` and is closed on a call to `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, the stream alias `fdbg_output` is attached to the current output stream.

`constraint_hook(`***Goal***`)`
: Tells FDBG to extend *Goal* with two (further) arguments and call it on the exit port of the global constraint dispatcher (`dispatch_global_fast/4`).

`no_constraint_hook`
: Tells FDBG not to use any constraint hook.

  If none of the above two options is used, the default is `constraint_hook(fdbg:fdbg_show)`.

`labeling_hook(`***Goal***`)`
: Tells FDBG to extend *Goal* with three (further) 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, the default is `labeling_hook(fdbg:fdbg_label_show)`.

For both `constraint_hook` and `labeling_hook`, *Goal* should be a visualizer, either built-in (see Section 10.36.3.3 [FDBG Built-In Visualizers], page 567) 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 10.36.4.2 [FDBG Writing Visualizers], page 573, for more details on these two options.

10.36.3.2 Naming Terms

Naming is a procedure of associating names with terms and variables; see Section 10.36.2.4 [FDBG Names of Terms], page 563. 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, use a default (generated) name, and return it in *Name*. See Section 10.36.2.6 [FDBG Name Auto-Generation], page 564.

`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.

10.36.3.3 Built-In Visualizers

The default visualizers are generic predicates to display FDBG events (see Section 10.36.2.1 [FDBG Events], page 562) in a well readable form. These visualizers naturally don't exploit any problem specific information—to have more "fancy" output, you have to write your own visualizers; see Section 10.36.4.2 [FDBG Writing Visualizers], page 573. To use these visualizers, pass them in the appropriate argument to `fdbg_on/1`; see Section 10.36.3.1 [FDBG Options], page 565, or call them directly from user defined visualizers.

`fdbg_show(+`***Constraint***`, +`***Actions***`)`
: This visualizer produces a trace output of all woken global constraints, in which a line showing the constraint is followed by a legend (see Section 10.36.2.7 [FDBG Legend], page 565) 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 = {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 doesn't have anything more to do: it exits.

Note that when you pass `fdbg_show/2` 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), ...]).
```

10.36.3.4 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, 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,
            [exit,_1001 in_set[[3|3]]]) ? A [2,4]

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.
```

W 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.

10.36.3.5 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 don't 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 pre-defined labeling predicates if possible. If you define your own labeling predicates and you want to use them even in the debugging session, 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 10.36.3.3 [FDBG Built-In Visualizers], page 567)—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 10.36.4.2 [FDBG Writing Visualizers], page 573.

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 FDBG won't be able to distinguish labeling events from other FDBG events any more.

10.36.3.6 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(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
```

Then, the magic sequence solver is loaded:

```
| ?- consult(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 mod-
ule 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_gcc(4,L,[enum]).
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 '`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 displayed block, `scalar_product/4` removes infeasible domain values from `list_3` and `list_4`, thus adjusting their upper bounds. The legend shows the domains before and after pruning. Note also that the constraint is rewritten to a more readable form:

```
<list_2>+2*<list_3>+3*<list_4>#=4
    list_2 = 0..3
    list_3 = 0..3 -> 0..2
    list_4 = 0..3 -> 0..1
```

The following block shows the initial labeling events, trying the value 0 for `list_1`:

```
Labeling [22, <list_1>]: starting in range 0..3.
Labeling [22, <list_1>]: indomain_up: <list_1> = 0
```

This immediately leads to a dead end:

```
global_cardinality([0,<list_2>,<list_3>,<list_4>],
                   [0-0,1-<list_2>,2-<list_3>,3-<list_4>])
    list_2 = 0..3
    list_3 = 0..2
    list_4 = 0..1
    Constraint failed.
```

We backtrack on `list_1`, trying instead the value 1. This leads to the following propagation steps:

```
Labeling [22, <list_1>]: indomain_up: <list_1> = 1

global_cardinality([1,<list_2>,<list_3>,<list_4>],
                   [0-1,1-<list_2>,2-<list_3>,3-<list_4>])
    list_2 = 0..3 -> 1..3
    list_3 = 0..2
    list_4 = 0..1

<list_2>+2*<list_3>+3*<list_4>#=4
    list_2 = 1..3
    list_3 = 0..2 -> 0..1
    list_4 = 0..1
```

However, we do not yet have a solution, so we try the first feasible value for `list_2`, which is 2. This is in fact enough to solve the goal. In the last two propagation steps, the constraint exits, which means that it holds no matter what value any remaining variable takes (in this example, there are none):

```
Labeling [29, <list_2>]: indomain_up: <list_2> = 2

global_cardinality([1,2,<list_3>,<list_4>],[0-1,1-2,2-<list_3>,3-<list_4>])
    list_3 = 0..1 -> {1}
    list_4 = 0..1 -> {0}

global_cardinality([1,2,1,0],[0-1,1-2,2-1,3-0])
    Constraint exited.

0#=0
    Constraint exited.
```

10.36.4 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.

10.36.4.1 Customizing Output

The printing of variable names is customized by defining the following hook predicate.

`fdbg:fdvar_portray(`***Name***`, `***Var***`, `***FDSet***`)` ***hook***

This hook predicate is called whenever an annotated constraint variable (see Section 10.36.3.5 [FDBG Annotation], page 569) 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 stored in this compoun, but it can be easily determined with a call to `fd_set/2`. (Although these two sets may be the same if the constraint didn't narrow it.)

If `fdbg:fdvar_portray/3` is undefined or fails the default representation is printed, which is *Name* between angle brackets.

The printing of legend lines is customized by defining the following hook predicate.

`fdbg:legend_portray(`***Name*, *Var*, *FDSet***`)` ***hook***

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 `fdbg:fdvar_portray/3`. Note that a prefix of four spaces and a closing newline character is always printed by FDBG.

If `fdbg:fdvar_portray/3` is undefined or fails the default representation is printed, which is

Name `=` ***RangeNow*** `[ ->` ***RangeAfter*** `]`

The arrow and *RangeAfter* are only printed if the constraint narrowed the domain of *Var*.

The following example will print a list of all possible values instead of the range for each variable in the legend:

```
:- 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', [Name, L])
        ;   format('~p = ~p -> ~p', [Name, L0, L])
        ).
```

10.36.4.2 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

MyGlobalVisualizer`([+`***Arg1***`, +`***Arg2***`, ...] +`***Constraint***`, +`***Actions***`)`

This visualizer is passed in the `constraint_hook` option. It must have at least two arguments, these are the following:

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 don't 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 10.36.4.6 [FDBG Debugging Global Constraints], page 580, for an example.

labeling visualizer

`MyLabelingVisualizer([+Arg1, +Arg2, ...] +Event, +ID, +Var)`

This visualizer is passed in the `labeling_hook` option. It must have at least three arguments, these are the following:

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

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 10.36.3.5 [FDBG Annotation], page 569.

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)`
`fdbg_annotate(+Term0, +Actions, -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 10.36.2.6 [FDBG Name Auto-Generation], page 564)

Var is the variable itself

FDSet 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*.

10.36.4.3 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_legend/[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 `fdbg_annotate/[3,4]`);
2. remove multiple `exit` and/or `fail` commands;
3. remove all ground actions, translating those that will cause failure into `fail(Action)`;
4. substitute all other narrowings with an `fdvar/3` compound term per variable.

The transformed action list may contain the following terms:

`exit`
: the constraint exits

`fail`
: the constraint fails due to a `fail` action

`fail(Action)`
: the constraint fails because of *Action*

`call(Goal)`
: *Actions* originally contained this action. FDBG can't do anything with that but to inform the user about it.

`fdvar(Name, Var, FDSet)`
: *Actions* also narrowed some variables that didn't appear in the *Vars* list, this is one of them. The meaning of the arguments is the usual, described in Section 10.36.4.2 [FDBG Writing Visualizers], page 573. This should normally not happen.

AnythingElse
: *Actions* contained unrecognized actions too, these are copied unmodified. This shouldn't happen!

10.36.4.4 Showing Selected Constraints (simple version)

Sometimes the programmer is not interested in every global 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 `all_different/1` and `all_distinct/1` only:

```
%% spec_filter(+Constraint, +Actions):  Call fdbg_show for all constraints
%%   for which intresting_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_distinct(_)).
```

Here is a session using the visualizer. Note that the initialization part (`domain/3` events), are filtered out, leaving only the `all_different/1` constraints:

```
| ?- [library('clpfd/examples/suudoku')].
[...]
| ?- fdbg_on(constraint_hook(spec_filter)).
% The clp(fd) debugger is switched on
% advice
| ?- suudoku([], 1, P).
all_different([1,<fdvar_1>,<fdvar_2>,8,<fdvar_3>,
               4,<fdvar_4>,<fdvar_5>,<fdvar_6>])
    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_different([7,6,2,5,8,4,1,3,9])
    Constraint exited.

P = [...] ;
no
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off
```

Note that the failure of `spec_filter/2` doesn't cause any unwanted output.

10.36.4.5 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.

```
:- use_module(library(lists), [append/3]).

%% 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:
%%     ctr_name            - matches all constraints of the given name
%%     ctr_name/arity      - matches constraints with the given name and arity
%%     ctr_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.
%%     fdbg_on([constraint_hook(filter_events([count/4],my_show))]).
filter_events(-CtrSpecs, Visualizer, Constraint, Actions) :- !,
        \+ show_constraint(CtrSpecs, Constraint),
        add_args(Visualizer, [Constraint, Actions], Goal),
        call(Goal).
filter_events(CtrSpecs, Visualizer, Constraint, Actions) :-
        show_constraint(CtrSpecs, Constraint),
        add_args(Visualizer, [Constraint, Actions], Goal),
        call(Goal).

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.

add_args(Goal0, NewArgs, Goal) :-
        Goal0 =.. [F|Args0],
        append(Args0, NewArgs, Args),
        Goal =.. [F|Args].
```

Here is a session using the visualizer, filtering out everything but `all_different/1` constraints:

```
| ?- [library('clpfd/examples/suudoku')].
[...]
| ?- fdbg_on(constraint_hook(filter_events([all_different/1]))).
% The clp(fd) debugger is switched on
% advice
| ?- suudoku([], 1, P).
all_different([1,<fdvar_1>,<fdvar_2>,8,<fdvar_3>,
               4,<fdvar_4>,<fdvar_5>,<fdvar_6>])
    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_different([7,6,2,5,8,4,1,3,9])
    Constraint exited.

P = [...] ;
no
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off
```

In the next session, all constraints named `all_different` 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).

```
| ?- [library('clpfd/examples/suudoku')].
[...]
| ?- fdbg_on(constraint_hook(filter_events(-
[all_different],fdbg_show))).
% The clp(fd) debugger is switched on
% advice
| ?- suudoku([], 1, P).
domain([1,<fdvar_1>,<fdvar_2>,8,<fdvar_3>,
        4,<fdvar_4>,<fdvar_5>,<fdvar_6>],1,9)
    fdvar_1 = inf..sup -> 1..9
    fdvar_2 = inf..sup -> 1..9
    fdvar_3 = inf..sup -> 1..9
    fdvar_4 = inf..sup -> 1..9
    fdvar_5 = inf..sup -> 1..9
    fdvar_6 = inf..sup -> 1..9
    Constraint exited.
    Constraint exited.

[...]

domain([2,<fdvar_46>,5,<fdvar_47>,<fdvar_48>,
       <fdvar_49>,<fdvar_50>,<fdvar_51>,9],1,9)
    fdvar_46 = inf..sup -> 1..9
    fdvar_47 = inf..sup -> 1..9
    fdvar_48 = inf..sup -> 1..9
    fdvar_49 = inf..sup -> 1..9
    fdvar_50 = inf..sup -> 1..9
    fdvar_51 = inf..sup -> 1..9
    Constraint exited.

P = [...] ;
no
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off
```

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/suudoku')].
[...]
| ?- fdbg_on(constraint_hook(filter_events(-
[domain(_,1,9),all_different(_)]))).
% The clp(fd) debugger is switched on
% advice
| ?- suudoku([], 1, P).
P = [...] ;
no
% advice
| ?- fdbg_off.
% The clp(fd) debugger is switched off
```

10.36.4.6 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:

`fdbg_guard(`***:Goal***`, +`***Constraint***`, +`***Actions***`)`

A constraint visualizer that does no output, but notifies the user by calling *Goal* if a solution is lost through domain narrowings. Naturally you have to inform `fdbg_guard/3` about the solution in question—stating which variables should have which values. To use `fdbg_guard/3`, first:

1. Set it up as a visualizer by calling:

   `fdbg_on([..., constraint_hook(fdbg_guard(`***Goal***`)), ...])`

   As usual, the two other arguments will be supplied by the FDBG core when calling `fdbg_guard/3`.

2. At the beginning of your program, form a pair of lists ***Xs-Vs*** where *Xs* is the list of variables and *Vs* is the list of values in question. This pair should then be assigned the name `fdbg_guard` using:

   `| ?- `***`fdbg_assign_name(Xs-Vs, fdbg_guard).`***

When these steps have been taken, `fdbg_guard/3` will watch the domain changes of *Xs* done by each global constraint *C*. Whenever *Vs* is in the domains of *Xs* at entry to *C*, but not at exit from *C*, *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(clpfd)).
:- use_module(library(fdbg)).

queens(L, N) :-
        length(L, N),
        domain(L, 1, N),
        constrain_all(L),
        labeling([ff,enum], 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).

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),
     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. FDBG is told to watch for the disappearance of the first correct solution, `[2,4,1,3]`. First, we get two incorrect solutions before FDBG wakes up, because in these cases the given good solution was made impossible by a labeling event. The second branch of labeling does not by itself remove the solution, but at some point on that branch the bad constraint does remove it, so `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 use the '`A`' debugger command (see Section 10.36.3.4 [FDBG Debugger Commands], page 568) to print the annotated form of the goal containing the culprit constraint.

For clarity, the labeling events were not turned off in the session below.

This information can be used to track down why the buggy `no_threat/3` performed the invalid pruning.

```
| ?- missing(L, [2,4,1,3]).
% The clp(fd) debugger is switched on
Labeling [8, <board_1>]: starting in range 1..4.
Labeling [8, <board_1>]: indomain_up: <board_1> = 1

Labeling [13, <board_2>]: starting in range {2}\/{4}.
Labeling [13, <board_2>]: dual: <board_2> = 2

L = [1,2,3,4] ? ;
Labeling [13, <board_2>]: dual: <board_2> = 4

L = [1,4,2,3] ? ;
Labeling [13, <board_2>]: failed.

Labeling [8, <board_1>]: indomain_up: <board_1> = 2


FDBG Guard:
  4 was removed from <board_2>

Culprit constraint:

no_threat(2,<board_2>,1)
    board_2 = 1..4 -> {3}
    Constraint exited.

% The debugger will first creep -- showing everything (trace)
23  2 Exit: clpfd:dispatch_global_fast(no_threat(2,_1001,1),0,0,
            [exit,_1001 in_set[[3|3]]]) ? A

clpfd:dispatch_global_fast(no_threat(2,<board_2>,1),0,0,
                           [exit,<board_2> in_set[[3|3]]])
    board_2 = 1..4

23  2 Exit: clpfd:dispatch_global_fast(no_threat(2,_1001,1),0,0,
            [exit,_1001 in_set[[3|3]]]) ? A [2,4]

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.

23  2 Exit: clpfd:dispatch_global_fast(no_threat(2,_1001,1),0,0,
            [exit,_1001 in_set[[3|3]]]) ? a
% Execution aborted
% advice,source_info
| ?- fdbg_off.
% The clp(fd) debugger is switched off
```

10.36.4.7 Code of the Built-In Visualizers

Now that you know everything about writing visualizers, it might be worth having a look at the code of the built-in visualizers, `fdbg_show/2` and `fdbg_label_show/3`.

```
fdbg_show(Constraint, Actions) :-
        fdbg_annotate(Constraint, Actions, AnnotC, CVars),
        print(fdbg_output, AnnotC),
        nl(fdbg_output),
        fdbg_legend(CVars, Actions),
        nl(fdbg_output).

fdbg_label_show(start, I, Var) :-
        fdbg_annotate(Var, AVar, _),
        (   AVar = fdvar(Name, _, Set)
        ->  fdset_to_range(Set, Range),
            format(fdbg_output,
                   'Labeling [~p, <~p>]: starting in range ~p.~n',
                   [I,Name,Range])
        ;   format(fdbg_output,
                   'Labeling [~p, <>]: starting.~n',
                   [I])
        ).
fdbg_label_show(fail, I, Var) :-
        (   var(Var)
        ->  lookup_or_set_name(Var, Name),
            format(fdbg_output,
                   'Labeling [~p, <~p>]: failed.~n~n',
                   [I,Name])
        ;   format(fdbg_output,
                   'Labeling [~p, <>]: failed.~n~n',
                   [I])
        ).
fdbg_label_show(step(Step), I, Var) :-
        (   var(Var)
        ->  lookup_or_set_name(Var, Name),
            format(fdbg_output,
                   'Labeling [~p, <~p>]: ~p~n~n',
                   [I,Name,Step])
        ;   format(fdbg_output,
                   'Labeling [~p, <>]: ~p~n~n',
                   [I,Step])
        ).

lookup_or_set_name(Term, Name) :-
        fdbg_get_name(Term, Name), !.
lookup_or_set_name(Term, Name) :-
        fdbg_assign_name(Term, Name).
```

As you can see, they are quite simple, thanks to the extensive set of support predicates also available to the user.

10.37 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`.

10.38 Tcl/Tk Interface—`library(tcltk)`

10.38.1 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?

10.38.1.1 What Is Tcl/Tk?

Tcl/Tk, as its title suggests, is actually two software packages: Tcl and Tk. Tcl, pronounced *tickle*, stands for *tool command language* and is a scripting language that provides a programming 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.

10.38.1.2 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, 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.

10.38.1.3 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.

10.38.1.4 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')`:

```
:- 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`.

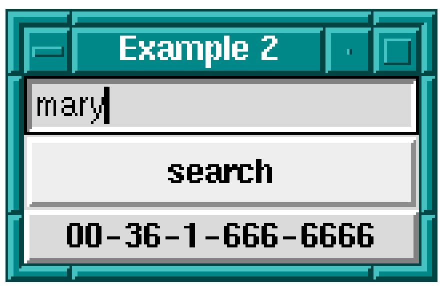
The previous example showed us how to create a button and display some text in it. It was basically pure Tcl/Tk generated from within Prolog but did not have any interaction with Prolog. The following example demonstrates a simple callback mechanism. A name is typed into a text entry box, a button is 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').
telephone(wilbert, '222-2222').
telephone(taxi, '200-0000').
telephone(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.
```



SICStus+Tcl/Tk telephone number lookup

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.

10.38.1.5 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.

10.38.2 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 10.38.7 [Resources], page 679.

10.38.2.1 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:

```
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 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:

```
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:

```
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.

10.38.2.2 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,

```
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:

```
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:

```
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:

```
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.

10.38.2.3 Commands

Now that the Tcl syntax and variables have been 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:

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:

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, 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 thats value is an integer string, increments it by the amount ***increment***. If the ***increment*** part is left out, 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:

arithmetic	`$x * 2`
relational	`$x > 2`
logical	`($x == $y) \|\| ($x == $z)`
bitwise	`8 & 2`
choice	`($a == 1) ? $x : $y`

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 `fred` having the value

```
"a b c {1 2 3} e f"
```

The outer curly brackets from the `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 can't 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.

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.

> `lindex` ***list index***

returns the *index*-th element of the *list*. The first element of a list has an index of 0.

> `linsert` ***list index value ?value ...?***

returns a new list in which the *value* arguments have been inserted in turn before the *index*-th element of *list*.

> `list` ***?value value ...?***

returns a list where each element is one of the *value* arguments.

> `llength` ***list***

returns the number of elements in list *list*.

> `lrange` ***list first last***

returns a slice of a list consisting of the elements of the list *list* from index *rst* until index *last*.

> `lreplace` ***list first last ?value ... value?***

returns a copy of list *list* but with the elements between indices *rst* and *last* replaced with a list formed from the *value* arguments.

> `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`'.

> `lsort` ***?***`-ascii`***? ?***`-integer`***? ?***`-real`***? ?***`-command` ***command? ?***`-increasing`***? ?***`-decreasing{`***? 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, the user can provide one through the '`-command` ***command***' switch.

There are also two useful commands for converting between lists and strings:

> `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:

```
set a {{} usr local bin}
set filename [join $a /]
```

results in the variable `filename` having the value `/usr/local/bin`.

The reverse of the `join` command is the `split` command:

`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:

```
set a [split /usr/local/src /]
```

gives `a` the value `{{} usr local src}`, a list.

Tcl has the four usual classes of *control flow* found in most other programming languages:

`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:

`if` ***test1 body1 ?elseif test2 body2 elseif ...? ?else bodyn?***

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, 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 matchinig

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

```
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
```

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,

```
scan "qty 10, unit cost 1.5, total 15.0" \
     "qty %d, unit cost %f, total %f"    \
     quantity cost_per_unit total
```

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, 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, 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.

The 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 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, 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, -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, 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, 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 `myfile` for both reading and writing and set the variable `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 `fid`.

To read from a file, the `read` command is used:

`read` ***fileId numBytes***

which reads ***numBytes*** bytes from the file attached to file handle ***leId***, and returns the bytes actually read.

To read a single line from a file use `gets`:

`gets` ***fileId ?varName?***

which reads a line from the file attached to file handle ***leId*** but chops off the trailing newline. If variable ***varName*** is specified, the string read in is stored there and the number of bytes is returned by the command. If the variable is not specified, the command returns the string only.

To write to a file, use `puts`:

`puts` ***?***`-nonewline`***? ?fileId? string***

which outputs the string ***string***. If the file handle ***leId*** is present, the string is output to that file; otherwise, it is printed on `stdout`. If the switch '`-nonewline`' is present, a trailing newline is not output.

To check if the end of a file has been reached, use `eof`:

`eof` ***fileId***

which, given a file handle ***leId*** 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:

```
proc sum { args } {
    set result 0;

    foreach n $args {
     set result [expr $result + $n ]
    }

    return $result;
}
```

which can be called like this:

```
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, it must be the last argument in the argument list.

A procedure can return a value through the `return` command:

```
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:

```
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:

```
set a 10
set b 20

proc add { first second } {
    upvar $first f $second s
    expr $f+$s
}
```

which when called with

```
add a b
```

will produce the result 30. If you use call-by-value instead:

```
add $a $b
```

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` doesn't exist at any level.)

New control structures can be generated using the `uplevel` command:

```
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.

```
proc do { loop condition } {
    set nostop 1

    while { $nostop } {
        uplevel $loop
        if {[uplevel "expr $condition"] == 0} {
            set nostop 0
         }
    }
}
```

which when called with this

```
set x 5
do { puts $x; incr x -1 } { $x > 0 }
```

will print

```
5
4
3
2
1
```

(**Please note**: this doesn't 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 *leName* 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.

10.38.2.4 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

10.38.3 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.

10.38.3.1 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.

10.38.3.2 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

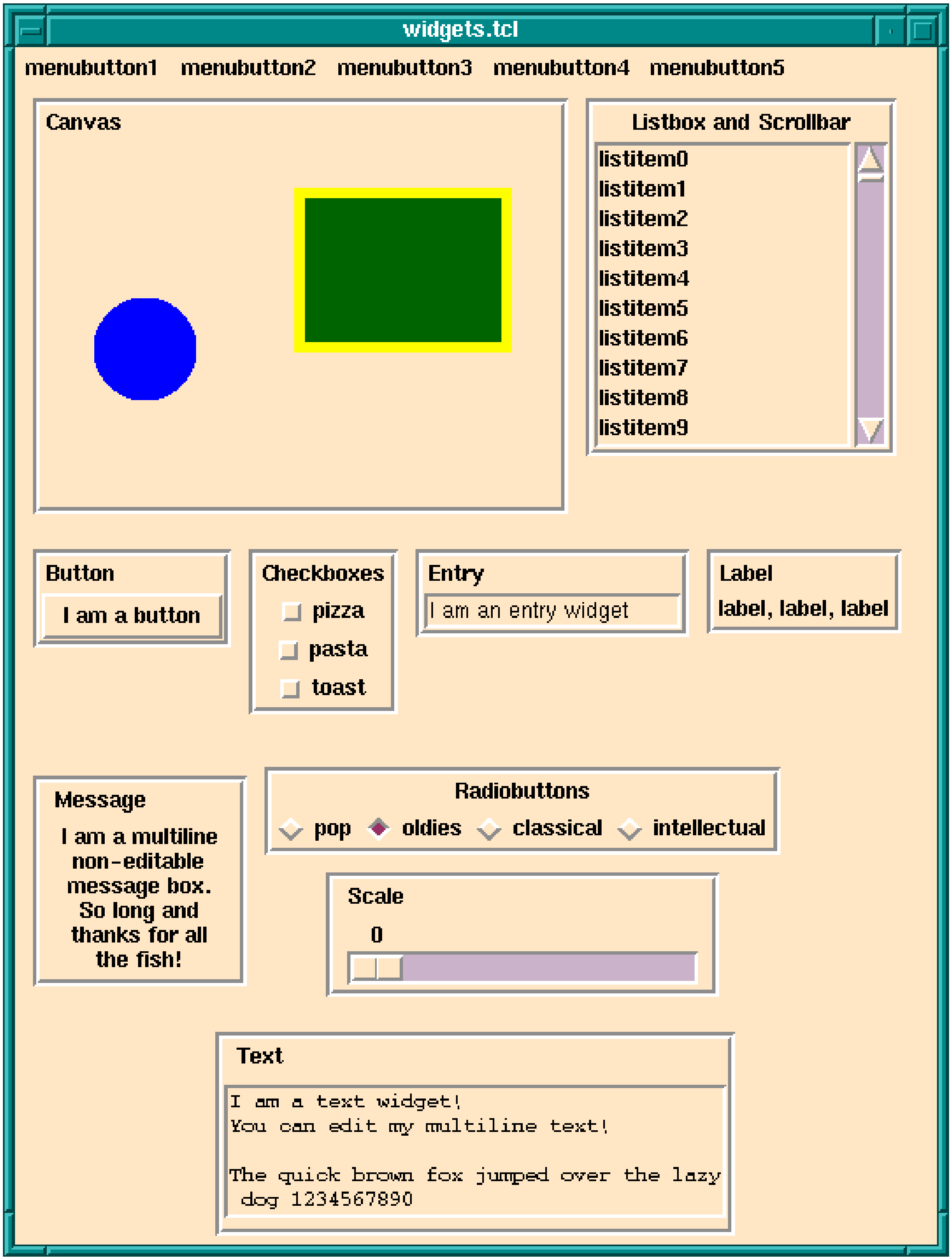
`radiobutton`
: a button used to form groups of mutually interacting buttons (When one button is pressed down, the others pop up.)

`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.

`scollbar` 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

10.38.3.3 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 notionaly 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 notionaly grouped together.)

An instance of a widget is created through the a Tcl command for that widget. The widget command my 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"
button .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? Isn’t 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 my 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.

10.38.3.4 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!"
```

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\n \
              Please 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, `ice` will have the value 1; if not checked, 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, 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, they will all be off. For example executing the script

```
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:

```
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:

```
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`:

```
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 scollbar 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, 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, 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, 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:

```
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:

```
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:

```
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 menu bar 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 menu button:

```
menubutton .mb -menu .mb.m -text "mymenu"
menu .mb.m
.mb.m add command -label hello
.mb.m add command -label goodbye
```

which crates 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 labelled `hello` and the other labelled `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.

10.38.3.5 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, 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.

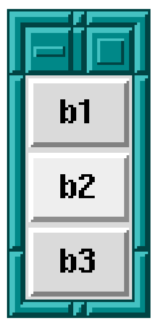
Let us take a simple example: three buttons packed into the root window. First we create the buttons; see also `library('tcltk/examples/ex3.tcl')`:

```
button .b1 -text b1
button .b2 -text b2
button .b3 -text b3
```

then we can pack them thus:

```
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.



Three Plain Buttons

If we change the size of the text in button `.b2` through the command:

```
.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')`:

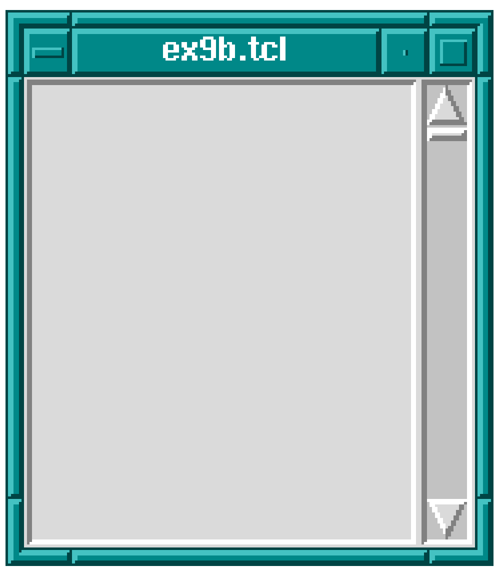
```
listbox .l
scrollbar .s
pack .l .s -side left
```

Scrollbar With Listbox, First Try

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:

```
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.

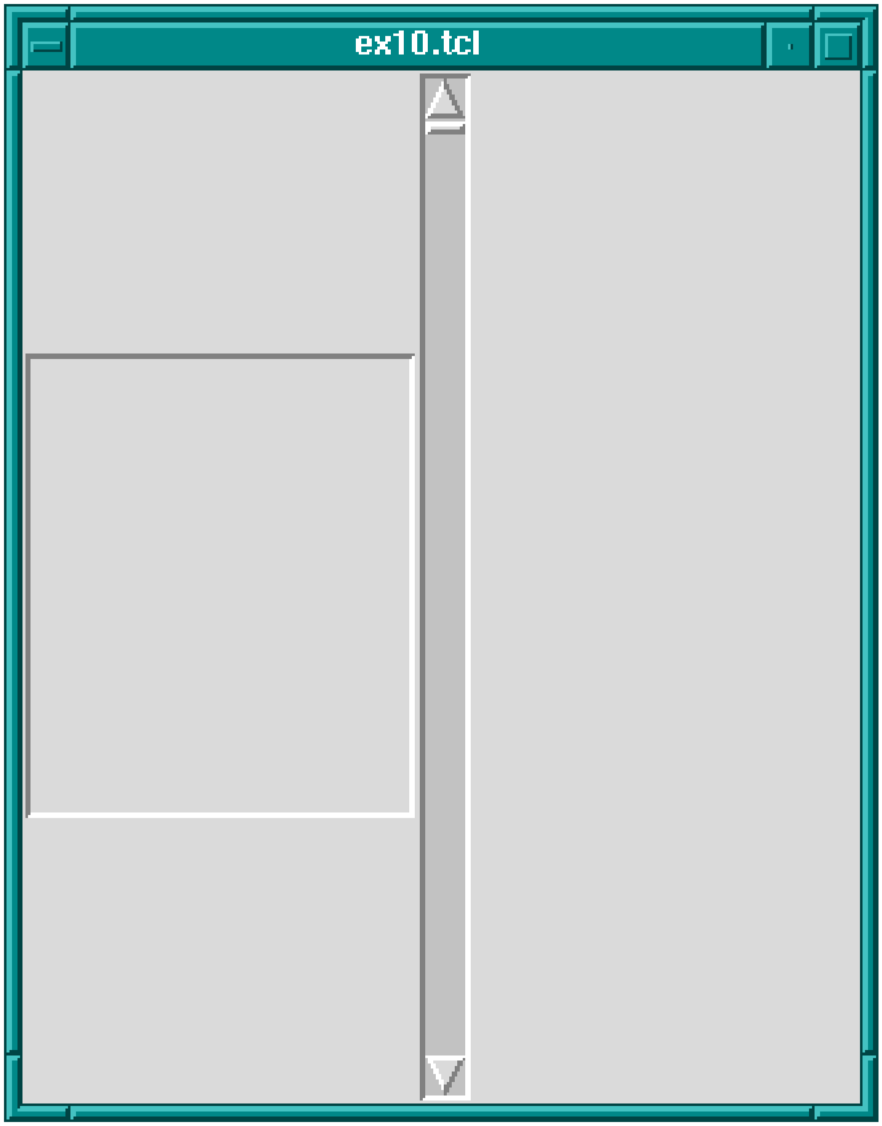
The `fill` packing option specifies whether the widget should fill space left over in its slice of space. A further option to take into account is what happens when the space allocated to the master widget is much greater than the that used by its slaves. This is not usually a problem initially because 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 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')`:

```
listbox .l
scrollbar .s
pack .l -side left
pack .s -side left -fill y
```

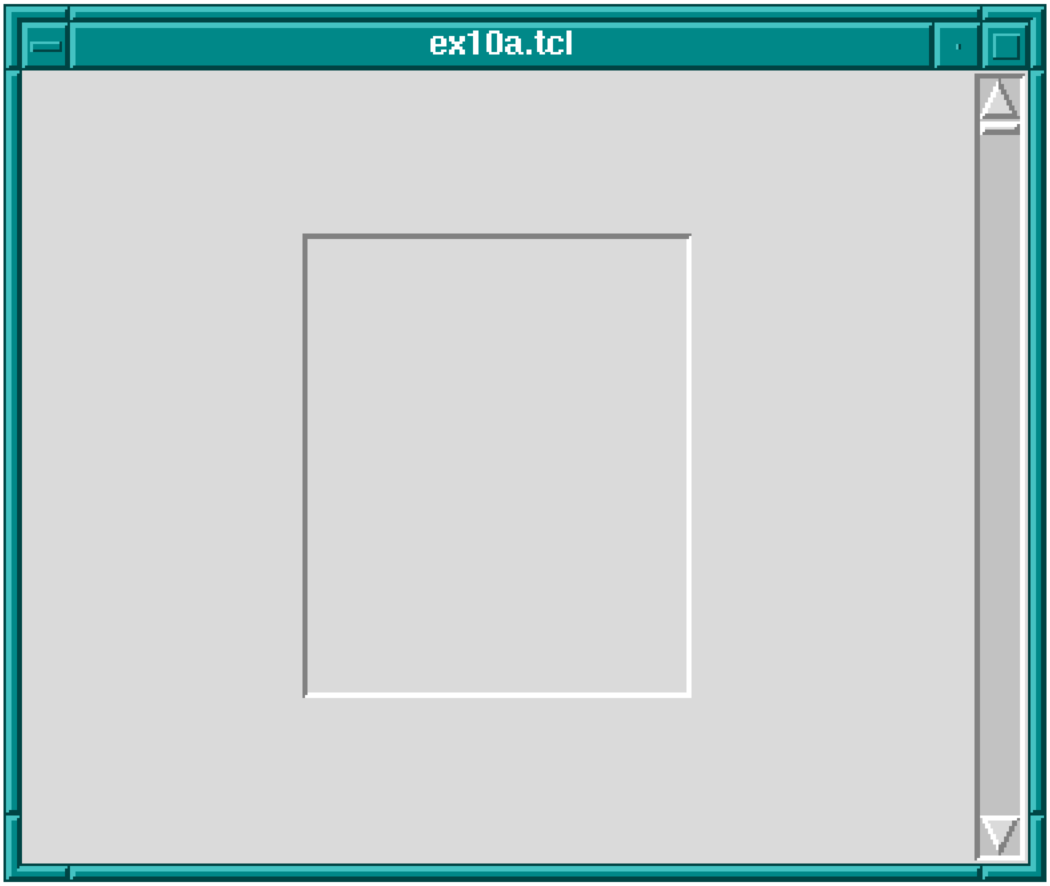
Initially this looks good, but now resize the window to a much bigger size. You will find that the listbox stays the same size and that empty space appears at the top and bottom of it, and that the scrollbar resizes in the vertical. It is now not so nice.



Scrollbar And Listbox, Problems With Resizing

We can fix part of the problem by having the listbox expand to fill the extra space generated by resizing the window.

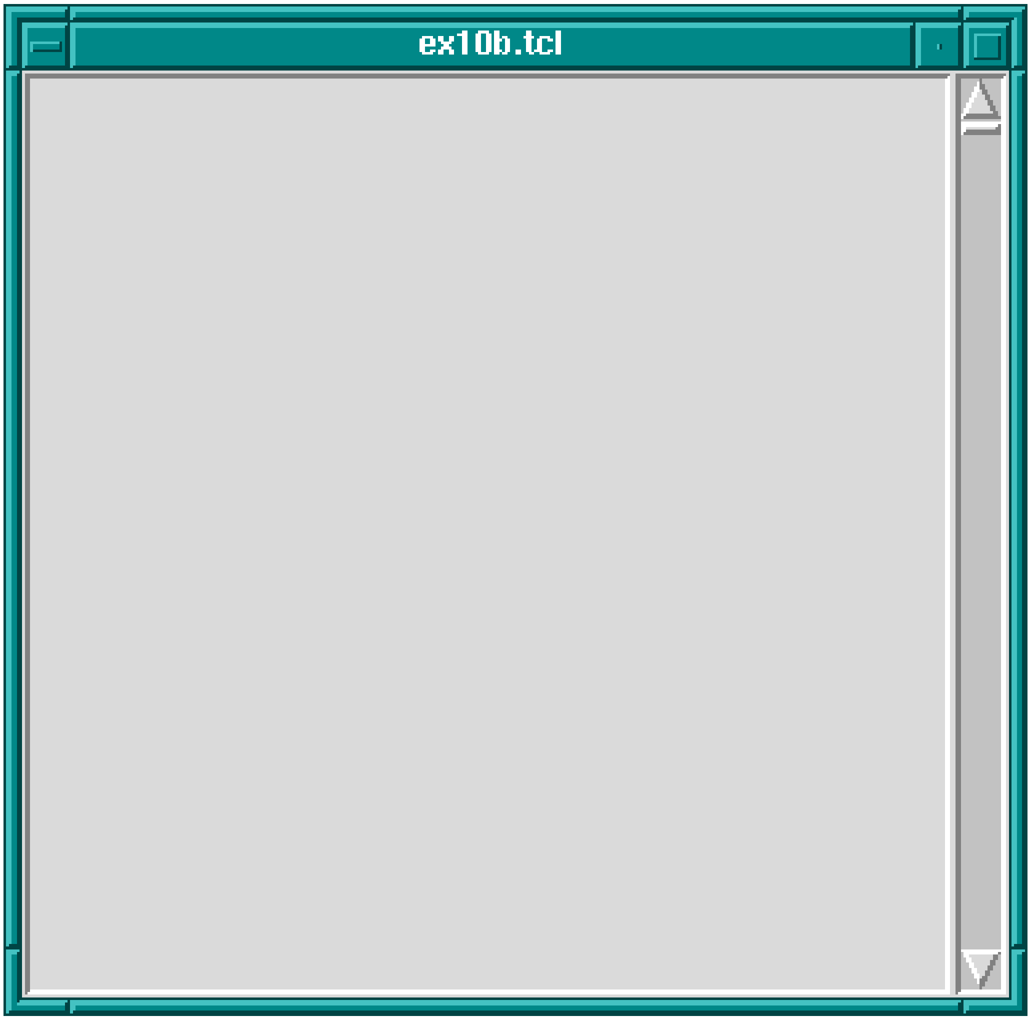
```
pack .l -side left -expand 1
```



Scrollbar And Listbox, Almost There

The problem now is that `expand` just expands the space allocated to the listbox, it doesn't 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
```

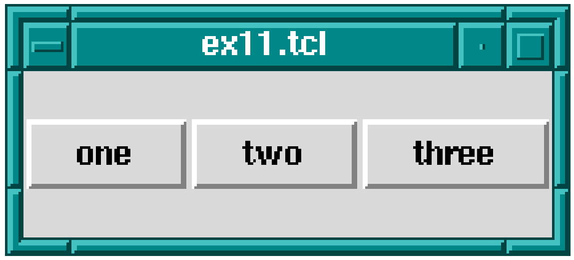


Scrollbar And Listbox, Problem Solved Using `fill`

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, 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')`:

```
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

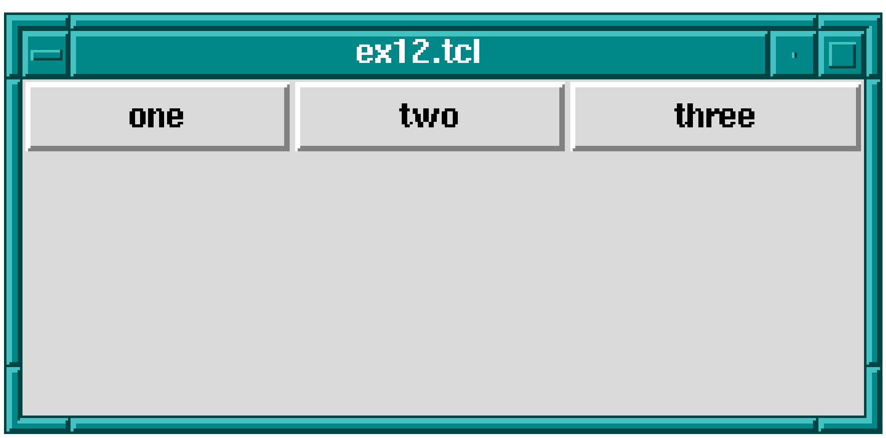
Now resize the window. You will see that the buttons resize to fill the width of the window, each taking an equal third of the width.

> **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, 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
```



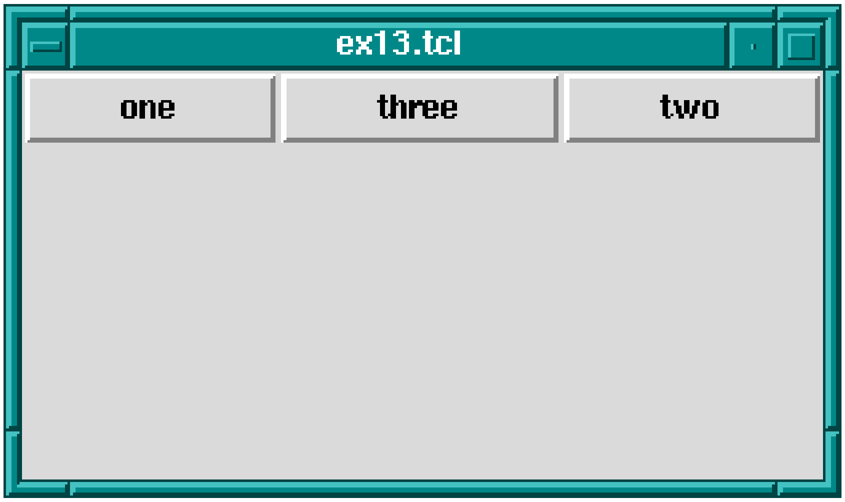
Anchoring Widgets

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,

```
pack .b3 -before .b2
```

will change the positions of `.b2` and `.b3` in our examples.



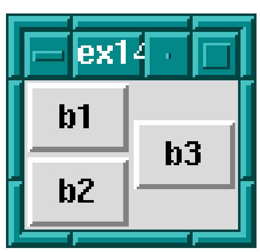
Changing The Packing Order Of Widgets

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')`:

```
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.

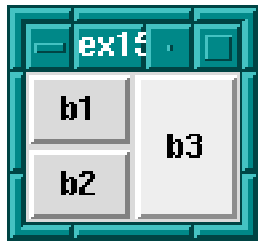


Using the `grid` Geometry Manager

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:

```
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, the following commands would achieve that:

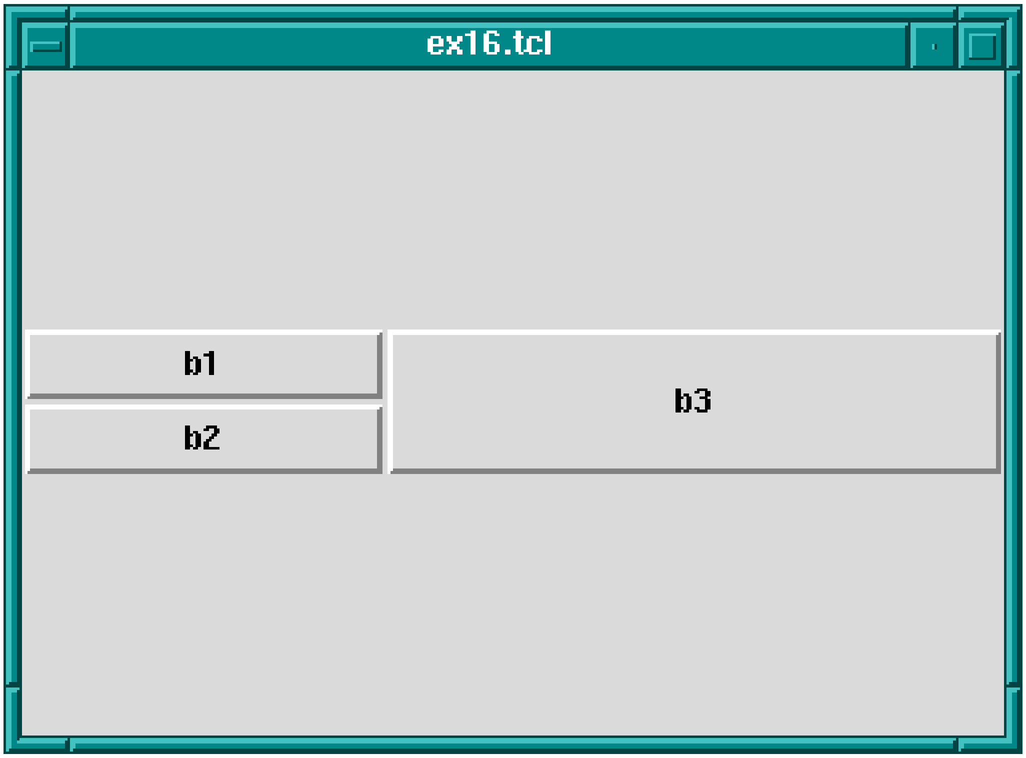
```
grid columnconfigure . 0 -weight 1
grid columnconfigure . 1 -weight 2
```

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')`:

```
button .b1 -text b1
button .b2 -text b2
button .b3 -text b3
grid .b1 -row 0 -column 0 -sticky nsew
grid .b2 -row 1 -column 0 -sticky nsew
grid .b3 -row 0 -column 1 -rowspan 2 -sticky nsew
grid columnconfigure . 0 -weight 1
grid columnconfigure . 1 -weight 2
```

Now resize the window to various sizes and we will see that button `.b3` has twice the width of buttons `.b1` and `.b2`.



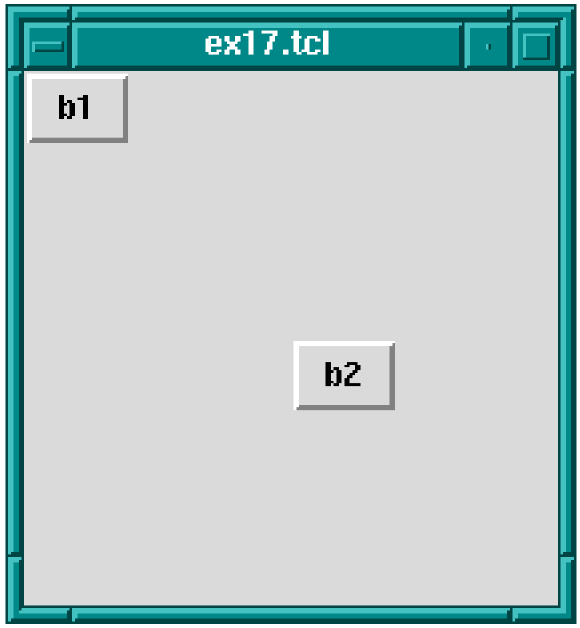
Changing Row/Column Ratios

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')`):

```
button .b1 -text b1
button .b2 -text b2
button .b3 -text b3
place .b1 -x 0 -y 0
place .b2 -x 100 -y 100
place .b3 -x 200 -y 200
```



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, 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.

10.38.3.6 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, *eventSqueuence* 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 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 10.38.7 [Resources], page 679):

```
bind all <Enter> {puts "Entering %W at (%x, %y)"}
bind all <Leave> {puts "Leaving %W at (%x, %y)"}
bind 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.

10.38.3.7 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 `destroy`-ed. 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:

```
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.)

10.38.3.8 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

10.38.3.9 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:

% ex18.pl

```
#! /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

```
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:

```
proc clear_board { } {
    for { set column 1 } {$column <= 8} {incr column} {
        reset_column $column
    }
}

proc reset_column { column } {
    for {set row 1 } { $row <= 8 } {incr row} {
        set_queens $column $row off
    }
}

proc set_queens { column row state } {
    if { $state == "on" } {
        .queens.$column-$row config -foreground black
    } else {
        .queens.$column-$row config
        -foreground [.queens.$column-$row cget -background]
    }
}
```

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.

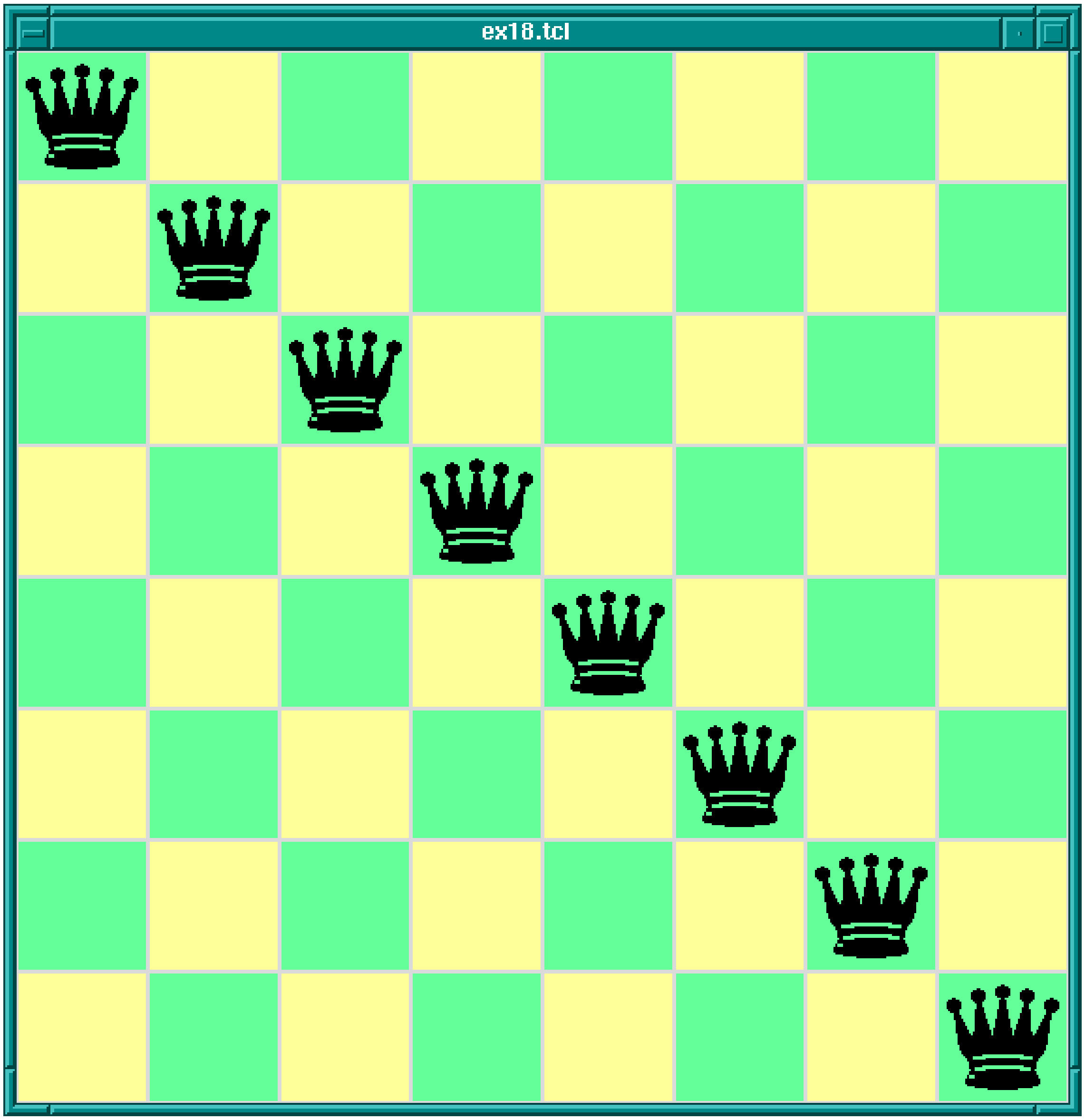
The final part is `place_queen`:

```
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:

```
show_solution "1 2 3 4 5 6 7 6 8"
```



8-Queens Display In Tcl/Tk

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 doesn't 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.)

10.38.4 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.

10.38.4.1 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.

10.38.4.2 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 `tcl_new/1`:

```
tcl_new(-TclInterpreter)
```

which creates a new interpreter, initializes it, and returns a reference to it in the variable *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 `tk_new/2` predicate is called from Prolog. It has the following form:

```
tk_new(+Options, -TclInterpreter)
```

which returns through the variable *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 `tcl_new/1` predicate, many Tcl/Tk interpreters may be created from Prolog at the same time through calls to `tk_new/2`.

The *Options* part of the call is a list of some (or none) of the following elements:

`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 Prolog 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 10.38.3.6 [Event Handling], page 635).

`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. (`send` is not covered in this document. Please refer to the Tcl/Tk documentation.)

`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, the default display is determined by the `DISPLAY` environment variable.

An example of using `tk_new/2`:

```
| ?- 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 Prolog 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:

`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.

10.38.4.3 Evaluation Functions

There are two functions in this category: Prolog extended to be able to evaluate Tcl expressions 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.

Command	::= *Name*
	| `chars(`***code-list***`)`
	| `write(`***term***`)`
	| `writeq(`***term***`)`
	| `write_canonical(`***term***`)`
	| `format(`***Fmt***`,`***Args***`)`
	| `dq(`***Command***`)`
	| `br(`***Command***`)`
	| `sqb(`***Command***`)`

```
                  | min(Command)
                  | dot(ListOfNames)
                  | list(ListOfCommands)
                  | ListOfCommands
Fmt               ::= atom
Name              ::= atom                    { other than [] }
                  | number
ListOfCommands    ::= []
                  |  [ Command | ListOfCommands
                  ]
ListOfNames       ::= []
                  | [ Name | ListOfNames ]
Args              ::= []
                  | [ term | Args ]
```

where

Atom
Number denote their printed representations

`chars(`***PrologString***`)`
: denotes the string represented by *PrologString* (a code-list)

`write(`***Term***`)`
`writeq(`***Term***`)`
`write_canonical(`***Term***`)`
: denotes the string that is printed by the corresponding built-in predicate.

> **Please 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 it therefore safest to use `write_canonical(`***Term***`)`.

`format(`***Fmt***`, `***Args***`)`
: denotes the string that is printed by the corresponding built-in predicate

`dq(`***Command***`)`
: denotes the string specified by *Command*, enclosed in double quotes

`br(`***Command***`)`
: denotes the string specified by *Command*, enclosed in curly brackets

`sqb(`***Command***`)`
: denotes the string specified by *Command*, enclosed in square brackets

`min(`***Command***`)`
: denotes the string specified by *Command*, immediately preceded by a hyphen

`dot(`***ListOfName***`)`
: denotes the widget path specified by *ListOfName*, preceded by and separated by dots

`list(`***ListOfCommands***`)`
: 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 10.38.4.3 [Evaluation Functions], page 646).

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:

```
tcl_error(Goal, Message)
tk_error(Goal, Message)
```

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

```
[TCL ERROR: Goal - Message]
[TK ERROR: Goal - Message]
```

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:

```
{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:

```
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), 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 10.38.4.3 [Evaluation Functions], page 646). 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 threatment of such variables by the Prolog top-level.

An example:

```
test_callback(Result) :-
    tcl_new(Interp),
    tcl_eval(Interp,
        'if {[prolog "foo(X,Y,Z)"] == 1} \\
            {list $prolog_variables(X) \\
                  $prolog_variables(Y) \\
                  $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 = 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, 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

```
{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`.)

10.38.4.4 Event Functions

Another way for Prolog to communicate with Tcl is through the predicate `tcl_event/3`:

```
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 10.38.4.3 [Evaluation Functions], page 646).

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:

```
tk_next_event(+TclInterpreter, -Event)
tk_next_event(+ListOrBitMask, +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 ***ListOrBitMask*** feature will be described below in the Housekeeping section when we talk about Tcl and Tk events; see Section 10.38.4.7 [Housekeeping], page 656.).

(We will meet `tk_next_event/[2,3]` again later when we discuss how it can be used to service Tk events; see Section 10.38.4.5 [Servicing Tk Events], page 654).

If the interpreter has been deleted, 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 10.38.4.3 [Evaluation Functions], page 646), 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, 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 `tcltk` library, we will return to our 8-queens example but now approaching from the Prolog side rather than the Tcl/Tk side:

```
:- 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).
```

This is the top-level fragment of the Prolog side of the 8-queens example. It has three predicates: `setup/0`, `closedown/1`, and `go/1`. `setup/0` simply creates the Tcl interpreter, loads the Tcl code into the interpreter using a call to `tcl_eval/3` (which also initialises the display) but then calls `tk_next_event/2` to wait for a message from the Tk side.

The Tk part that sends `prolog_event`-s to Prolog looks like this:

```
button .next -text next -command {prolog_event  next}
pack .next

button .stop -text stop -command {prolog_event stop}
pack .stop
```

that is two buttons, one that sends the atom `next`, the other that sends the atom `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, 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`, 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, 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 altoegther, 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.

10.38.4.5 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, 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, `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, critical sections of the Prolog code can be sprinkled with calls to `tk_do_one_event/[0,1]` to alleviate the problem.

`tk_do_one_event/[0,1]` has the following forms:

```
tk_do_one_event
tk_do_one_event(+ListOrBitMask)
```

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 *ListOrBitMask* variable. As indicated, this is either a list of atoms that are event types, or a bit mask as specified in the Tcl/Tk documentation. (The bit mask should be avoided for portability between Tcl/Tk versions.)

The *ListOrBitMask* list can contain the following atoms:

`tk_dont_wait`
: don't wait for new events, process only those that are ready

`tk_x_events`
`tk_window_events`
: process window events

`tk_file_events`
: process file events

`tk_timer_events`
: process timer events

`tk_idle_events`
: process `Tk_DoWhenIdle` events

`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.

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, `tk_do_one_event/1` will fail if the `tk_dont_wait` wait flag is present, as will `tk_do_one_event/0`, which has that flag set implicitly.

If the `tk_dont_wait` flag is not set, a call to `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:

```
tk_do_all_events :-
    tk_do_one_event, !,
    tk_do_all_events.
tk_do_all_events.
```

The predicate `tk_next_event/[2,3]` is similar to `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:

```
tk_next_event(+TclInterpreter, -Event)
tk_next_event(+ListOrBitMask, +TclInterpreter, -Event)
```

The Prolog event is returned in the variable *Event* and is the first term on the Prolog event queue associated with the interpreter *TclInterpreter*. (Prolog events are initiated on the Tcl side through the new Tcl command `prolog_event`, covered earlier; see Section 10.38.4.4 [Event Functions], page 651).

10.38.4.6 Passing Control to Tk

There is a predicate for passing control completely over to Tk, the `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:

```
tk_main_loop
```

10.38.4.7 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:

```
tk_main_window(+TclInterpreter, -TkWindow)
```

which given a reference to a Tcl interpreter *TclInterpreter*, returns a reference to its main window in *TkWindow*.

The window reference can then be used in `tk_destroy_window/1`:

```
tk_destroy_window(+TkWindow)
```

which destroys the window or widget referenced by *TkWindow* and all of its sub-widgets.

The predicate `tk_make_window_exist/1` also takes a window reference:

```
tk_make_window_exist(+TkWindow)
```

which causes the window referenced by *TkWindow* in the Tcl interpreter *TclInterpreter* to be immediately mapped to the display. This is useful because normally Tk delays displaying new information for a 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:

```
tk_num_main_windows(-NumberOfWindows)
```

which returns an integer in the variable *NumberOfWindows*.

10.38.4.8 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

`tcl_new/1`
: create a Tcl interpreter

`tcl_delete/1`
: remove a Tcl interpreter

`tk_new/2`
: create a Tcl interpreter with Tk extensions

evaluation functions

`tcl_eval/3`
: evaluate a Tcl expression from Prolog

`prolog`
: evaluate a Prolog expression from Tcl

Prolog event queue functions

`tcl_event/3`
: evaluate a Tcl expression and return a Prolog queue event list

`tk_next_event/[2,3]`
: pass control to Tk until a Prolog queue event happens and return the head of the queue

`prolog_event`
: place a Prolog term on the Prolog event queue from Tcl

servicing Tcl and Tk events

`tk_do_one_event/[0,1]`
: pass control to Tk until one Tk event is serviced

`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

`tk_main_loop/0`
: control passed to Tk until all windows in all Tcl/Tk interpreters are gone

housekeeping

`tk_main_window/2`
: return reference to main in of a Tcl/Tk interpreter

`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

`tcl_new/1`
: create a Tcl interpreter

`tcl_delete/1`
: remove a Tcl interpreter

`tcl_eval/3`
: evaluate a Tcl expression from Prolog

`tcl_event/3`
: evaluate a Tcl expression and return a Prolog event list

predicates for Prolog to interact with Tcl interpreters with Tk extensions

`tk_new/2` create a Tcl interpreter with Tk extensions

`tk_do_one_event/[0,1]`
: pass control to Tk until one Tk event is serviced

`tk_next_event/[2,3]`
: also services Tk events but returns when a Prolog queue event happens and returns the head of the queue

`tk_main_loop/0`
: control passed to Tk until all windows in all Tcl/Tk interpreters are gone

`tk_main_window/2`
: return reference to main in of a Tcl/Tk interpreter

`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

commands for the Tcl interpreters to interact with the Prolog system

`prolog`
: evaluate a Prolog expression from Tcl

`prolog_event`
: place a Prolog term on the Prolog event queue 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 co-operation between Prolog and Tcl/Tk can be arranged: how to achieve them, and their benefits.

10.38.5 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 my 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.

10.38.5.1 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 `tk_new/2`
- set up the Tk GUI through calls to `tcl_eval/3`
- pass control to Tcl/Tk through `tk_main_loop`

Some of The Tk widgets in the GUI will have “callbacks” to Prolog, i.e. they will call the `prolog` Tcl command. When the Prolog call returns, the values stored in the `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:

```
:- use_module(library(tcltk)).

hello('world').

go :-
    tk_new([], Tcl),
    tcl_eval(Tcl, 'source simple.tcl', _),
    tk_main_loop.
```

which just loads the `library(tcltk)`, defines a `hello/1` data clause, and `go/0`, which starts a new Tcl/Tk interpreter, loads the code `simple.tcl` into it, and passes control to Tcl/Tk.

The Tcl part, `simple.tcl`, is this:

```
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 "hello(X)"
    set tvar $prolog_variables(X)
}
```

which creates a label, with an associated text variable, and a button, that has a call back procedure, `call_and_display`, attached to it. When the button is pressed, `call_and_` `display` is executed, which simply evaluates the goal `hello(X)` in Prolog and the text variable of the label `.l` to whatever `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 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')`):

```
:- 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')`):

% 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
```

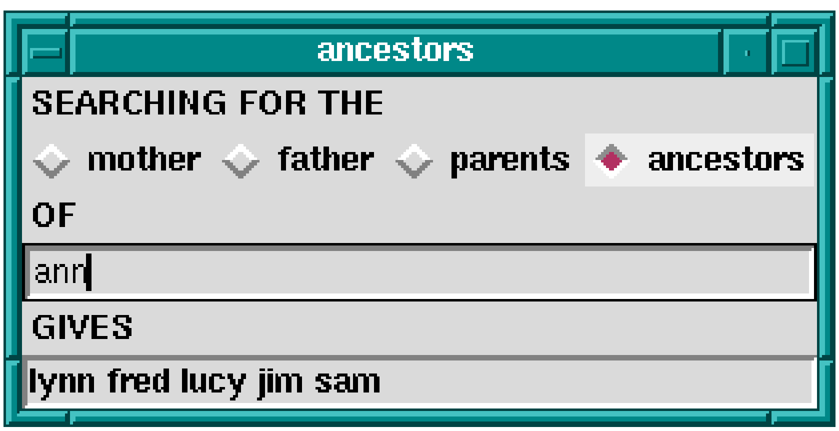
% ancestors.pl

```
# 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, 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.

10.38.5.2 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 10.38.3.9 [Queens Display], page 638.

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.

10.38.5.3 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, 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 doesn't seem satisfactory. If we use the Prolog is the master and Tcl/Tk is the slave model, 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 mearly 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 haven't changed from our previous example, see Section 10.38.3.9 [Queens Display], page 638).

```
:- 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 `tk_next_event/2` in several places. The code is executed by calling `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 `next`, indicating that we want to show the next solution, or `stop`, indicating that we want to stop the program.

If `next` is received, the program goes on to execute `go/1`. What this does it to first calculate a solution to the 8-queens problem, displays the solution through `show_solution/2`, and then waits for another message from Tcl/Tk. Again this can be either `next` or `stop`. If `next`, the the program goes into the failure part of a failure driven loop and generates and displays the next solution.

If at any time `stop` is received, 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.

```
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.)

10.38.5.4 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')`:

% 8-queens.pl

```
:- 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

```
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.
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')`:

 # 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 "#ffff99" } 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 sqaure 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]
    }
}
```

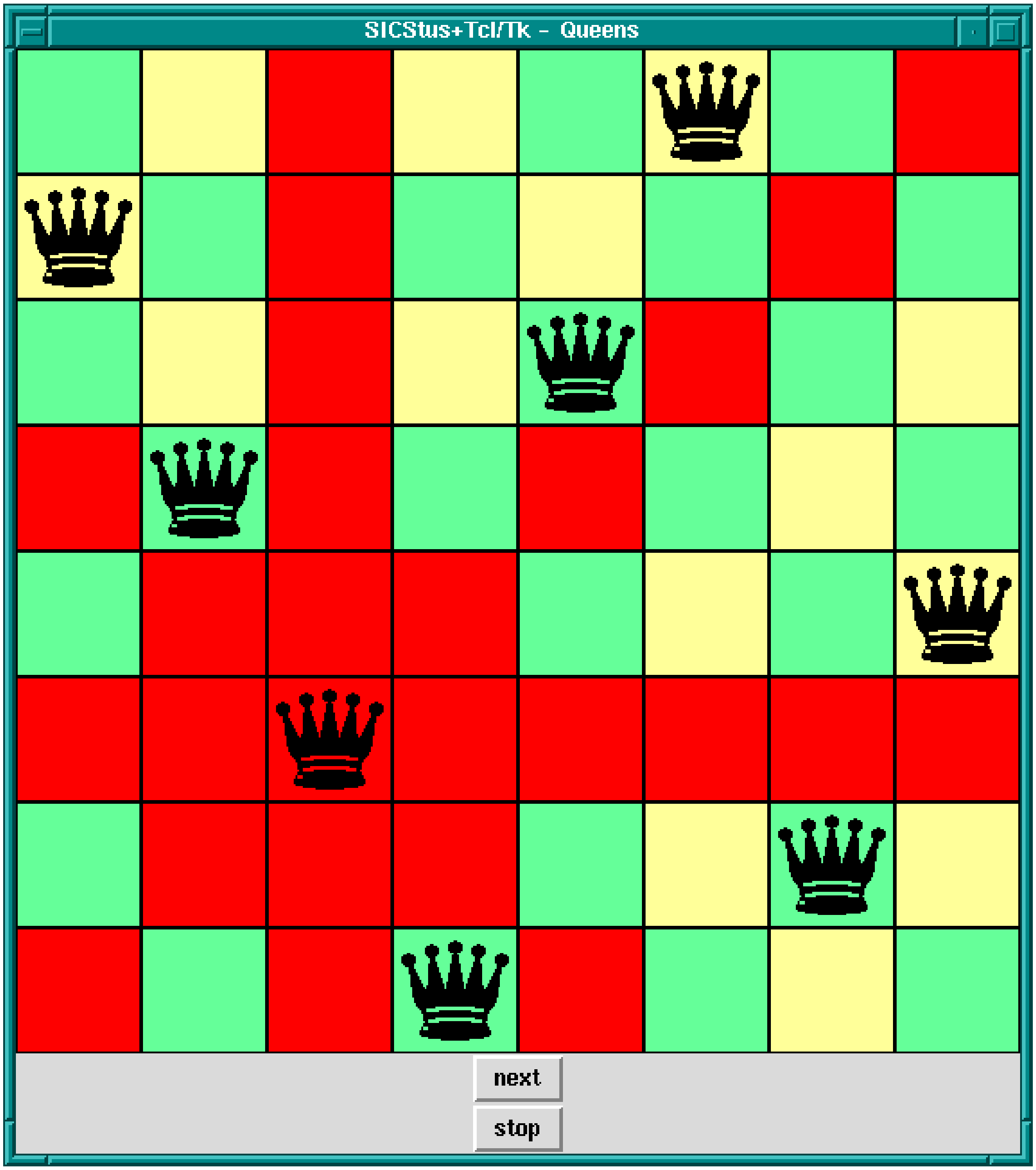
 # 8-queens.tcl

```
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
```



8-Queens Solution, Attacked Squares Highlighted

10.38.6 Quick Reference

10.38.6.1 Command Format Summary

Command ::= *Name*
| `chars(`***`code-list`***`)`
| `write(`***`term`***`)`
| `writeq(`***`term`***`)`
| `write_canonical(`***`term`***`)`
| `format(`***`Fmt`***`,`***`Args`***`)`
| `dq(`***`Command`***`)`

```
                   | br(Command)
                   | sqb(Command)
                   | min(Command)
                   | dot(ListOfNames)
                   | list(ListOfCommands)
                   | ListOfCommands
Fmt                ::= atom
Name               ::= atom                      { other than [] }
                   | number
ListOfCommands     ::= []
                   |  [ Command | ListOfCommands
                   ]
ListOfNames        ::= []
                   | [ Name | ListOfNames ]
Args               ::= []
                   | [ term | Args ]
```

where

Atom
Number denote their printed representations

`chars(`***PrologString***`)`
denotes the string represented by *PrologString* (a code-list)

`write(`***Term***`)`
`writeq(`***Term***`)`
`write_canonical(`***Term***`)`
denotes the string that is printed by the corresponding built-in predicate

`format(`***Fmt***`, `***Args***`)`
denotes the string that is printed by the corresponding built-in predicate

`dq(`***Command***`)`
denotes the string specified by *Command*, enclosed in double quotes

`br(`***Command***`)`
denotes the string specified by *Command*, enclosed in curly brackets

`sqb(`***Command***`)`
denotes the string specified by *Command*, enclosed in square brackets

`min(`***Command***`)`
denotes the string specified by *Command*, immediately preceded by a hyphen

`dot(`***ListOfName***`)`
denotes the widget path specified by *ListOfName*, preceded by and separated by dots

`list(`***ListOfCommands***`)`
denotes the TCL list with one element for each element in *ListOfCommands*.

ListOfCommands
denotes the string denoted by each element, separated by spaces

10.38.6.2 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.

10.38.6.3 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 Prolog 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, the default display is determined by the `DISPLAY` environment variable.

`tk_do_one_event`
`tk_do_one_event(+ListOrBitMask)`
: Passes control to Tk to handle a single event before passing control back to Prolog. The type of events handled is passed through the *ListOrBitMask* variable. As indicated, this is either a list of atoms that are event types, or a bit mask as specified in the Tcl/Tk documentation. (The bit mask should be avoided for portability between Tcl/Tk versions.)

  The *ListOrBitMask* list can contain the following atoms:

`tk_dont_wait`
: don't wait for new events, process only those that are ready

`tk_x_events`
`tk_window_events`
: process window events

`tk_file_events`
: process file events

`tk_timer_events`
: process timer events

`tk_idle_events`
: process `Tk_DoWhenIdle` events

`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, the call will fail.

`tk_next_event(+`***TclInterpreter***`, -`***Event***`)`
`tk_next_event(+`***ListOrBitMask***`, +`***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*.

`tk_main_loop`
: Pass control to Tk until all windows in all Tcl/Tk interpreters are gone.

`tk_main_window(+`***TclInterpreter***`, -`***TkWindow***`)`
: Return in *TkWindow* a reference to the main window of a Tcl/Tk interpreter with handle passed in *TclInterpreter*.

`tk_destroy_window(+`***TkWindow***`)`
: Destroy a window or widget.

`tk_make_window_exist(+`***TkWindow***`)`
: Force display of a window or widget.

`tk_num_main_windows(`***-NumberOfWindows***`)`
: Return in *NumberOfWindows* the total number of Tk main windows existing in the system.

10.38.6.4 Commands for Tcl Interpreters to Interact with The Prolog System

`prolog`
: Evaluate a Prolog expression from Tcl.

`prolog_event`
: Place a Prolog term on the Prolog event queue from inside Tcl.

10.38.7 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.

10.38.7.1 Web Sites

The home of Tcl/Tk is at:

`http://tcl.sourceforge.net`

The Tcl Developer Xchange site is at:

`http://www.tcl.tk`

10.38.7.2 Books

There are a surprising number of books on Tcl/Tk, extensions to Tcl/Tk, and Tk as an extension to other languages. Here we mention just a few of the well-known books that will get you started with building Tcl/Tk GUIs, which can then be interfaced to your Prolog applications.

Brent Welch, ***Practical Programming in Tcl and Tk***. Prentice Hall, 1999. 3rd Edition ISBN: 0-13-022028-0 `http://www.beedub.com/book/`

John Ousterhout, ***Tcl and the Tk Toolkit***. Addison-Wesley, 1994, ISBN 0-201-63337-X

Paul Raines, ***Tcl/Tk Pocket Reference***, 1st Ed., Oct. 1998, ISBN 1-56592-498-3

Paul Raines & Jeff Tranter, ***Tcl/Tk in a Nutshell***, 1st Ed., March 1999, 1-56592-433-9

Also visit the '`books`' section of the Tcl/Tk web site:

`http://resource.tcl.tk/resource/doc/books/`

10.38.7.3 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://resource.tcl.tk/resource/doc/manual/`

10.38.7.4 Usenet News Groups

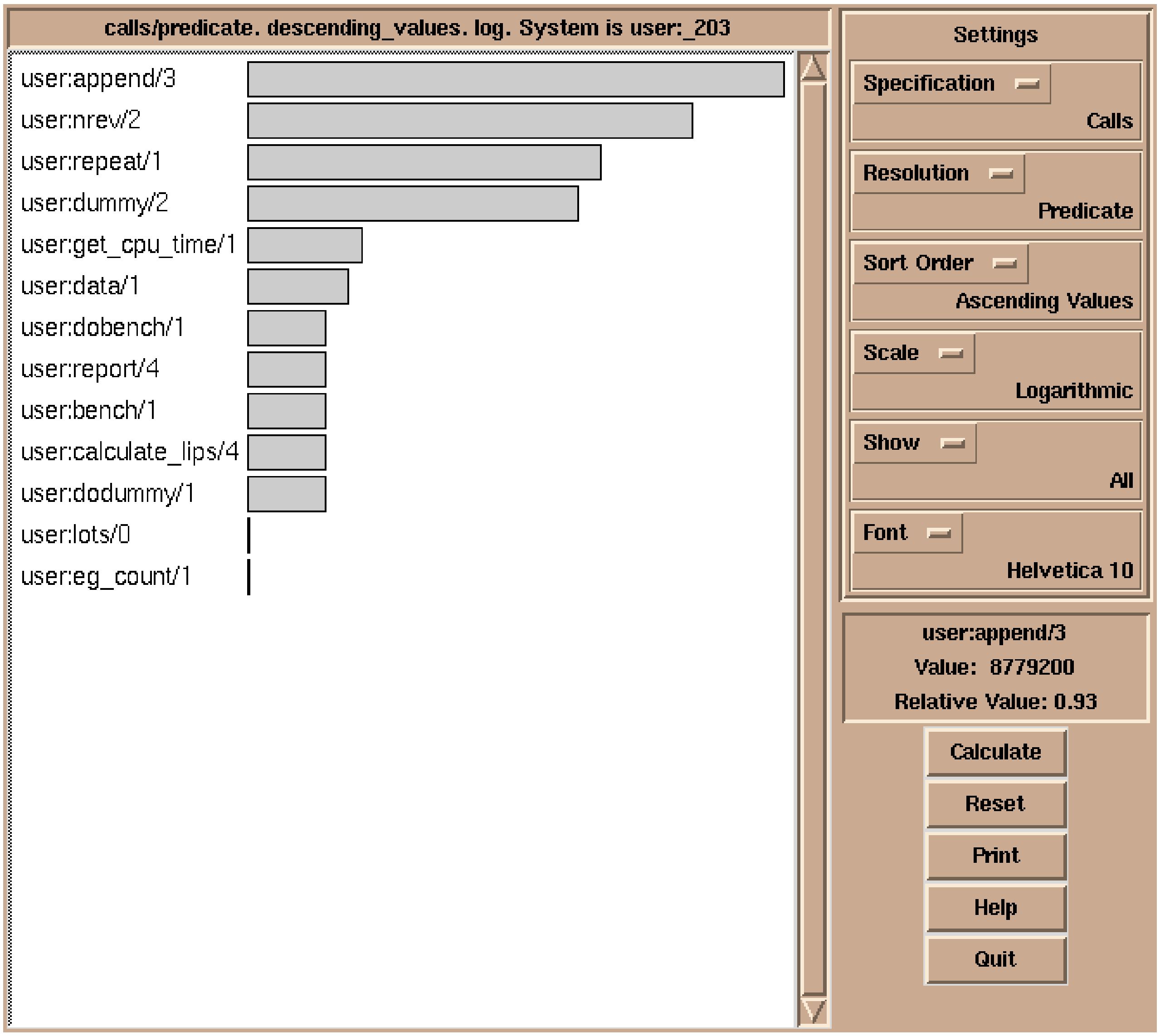
The newsgroup for everything Tcl is

`news:comp.lang.tcl`

10.39 The Gauge Profiling Tool—`library(gauge)`

The Gauge library package is a graphical interface to the SICStus built-in predicates `profile_data/4` and `profile_reset/1`. See Section 9.2 [Execution Profiling], page 305, for more information about execution profiling. The interface is based on Tcl/Tk (see Section 10.38 [lib-tcltk], page 586).

`view(:Spec)`
: Creates a graphical user interface for viewing the profile data for the predicates covered by the generalized predicate spec *Spec*. For example, the call `view([user:_,m2:_])`, will bring up the graphical user interface on the predicates contained in the modules `user` and `m2`. When the display first comes up it is blank except for the control panel. A screen shot is shown below.



Gauge graphical user interface

The menus and buttons on the control panel are used as follows:

Specification
: Selects what statistics to display. One of:

  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).

Resolution
Selects the level of resolution. One of:

Predicate Compute results on a per predicate basis. Predicates are denoted ***Module:Name/Arity***.

Clause Compute results on a per clause basis. Clauses are denoted ***Module:Name/Arity/ClauseNo***.

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.

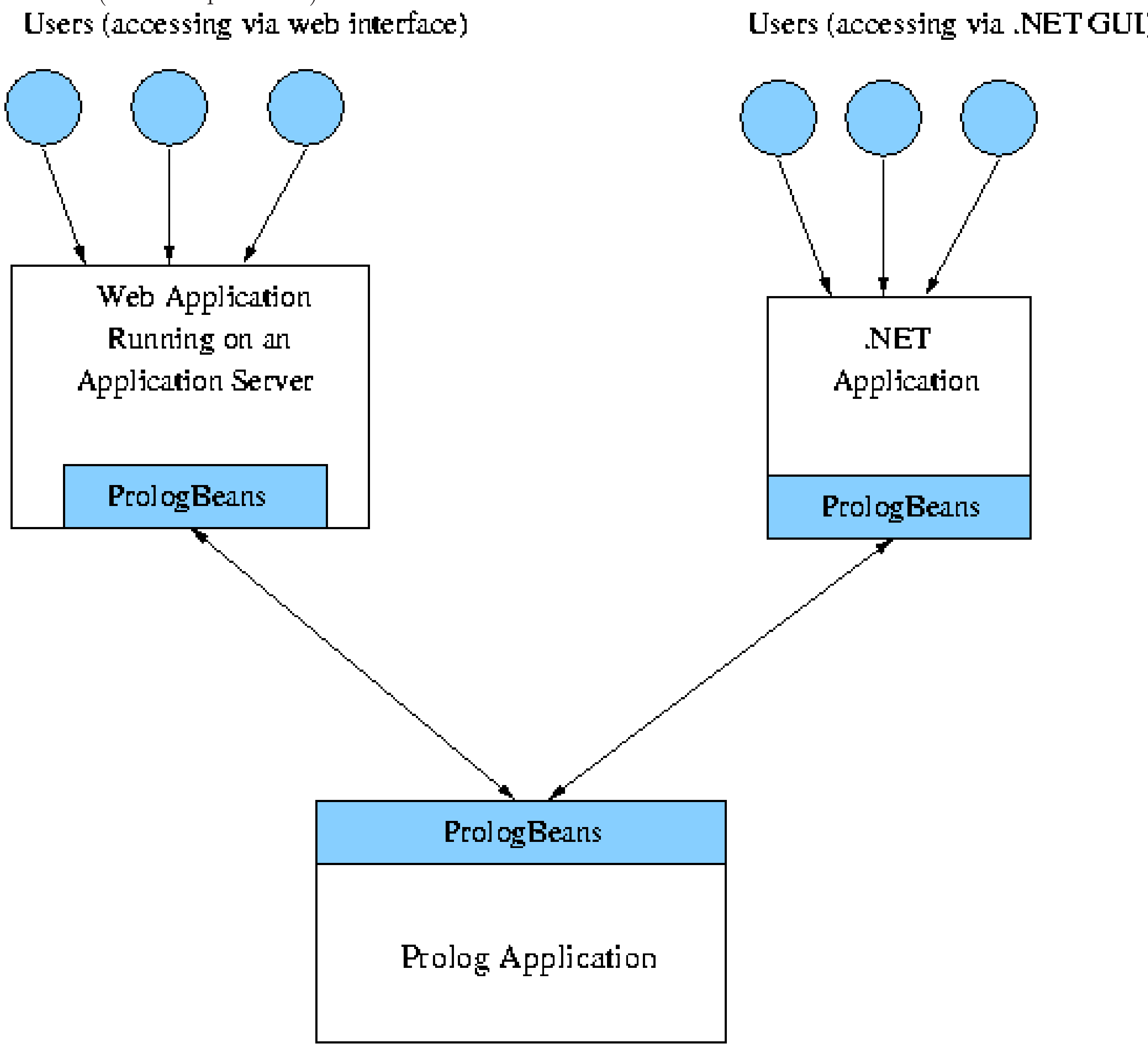
By clicking on the bars of the histogram, the figures are displayed in the *Value Info* window.

10.40 PrologBeans Interface—`library(prologbeans)`

10.40.1 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 and '`prologbeans.dll`' for .NET (MS Windows only).

10.40.2 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]

10.40.3 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();
    }
```

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, 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 the expression to evaluate is represented as a code-list and must be converted into a term before evaluation.

Please note: the environment variable `SP_PATH` as used here is meant to be a shorthand for the SICStus Prolog installation directory, and does not need to be set explicitly.

To start the example, first start the Prolog server by going to the '`pbexamples(evaluate)`' 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)

10.40.4 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`
`start(+`***Options***`)`

starts the Prolog server using the options specified. **Please note**: `start/[0,1]` will not return until a server shutdown occurs. ***Options*** should be a list of zero or more of:

`port(`***?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 Adress bit will be set in the underlying sockets layer. If ***Val*** is a variable, some unused port will be selected by the OS, the actual port number can be obtained with `get_server_property/1`, typically from a `server_started` event listener.

`accepted_hosts(+`***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']`).

`session_timeout(+`***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 there will be no garbage collect on sessions (default: `0`).

`session_gc_timeout(+`***Val***`)`
: an integer denoting the minimum time in seconds between two consecutive session garbage collections. If the timeout is set to zero there will be no garbage collect on sessions (default: `0`).

For example:

```
:- start([port(7500),
          accepted_hosts(['127.0.0.1','99.8.7.6'])]).
```

`shutdown`
`shutdown(+Mode)`

shuts down the server and closes the sockets and the streams after processing all available input. There are three modes:

`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`***`, +`***`SessionVar`***`)`
: registers a query and the corresponding predicate. Before the registration any previously registered query matching `Query` will be removed (as if `unregister_query(Query)` was called). The predicate, `PredicateToCall` will be called, as if by `once(PredicateToCall)`, when a query matching `Query` is received. 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, 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`***`, `***`-Id`***`)`
`register_event_listener(+`***`Event`***`, `***`:PredicateToCall`***`)`
: 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 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.

10.40.5 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/

A brief description of the provided Java classes are presented below. More information about the Java APIs is available in the JavaDoc files in the directory http://www.sics.se/sicstus/docs/.

`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.

10.40.6 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:

```
user:file_search_path(pbexamples, library('prologbeans/examples')).
```

10.40.6.1 Embedding Prolog in Java Applications

If you have an advanced Prolog application that needs a GUI 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 10.40.3 [PB First Example], page 683).

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)

10.40.6.2 Application Servers

If you want to get your Prolog application to be accessible from an intranet or the Internet 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).

```
<%@ page import = "se.sics.prologbeans.*" %>
<html>
<head><title>Sum Calculator</title></head>
<body bgcolor="white">
<font size=4>Prolog Sum Calculator, enter expression to evaluate:
<form><input type=text name=query></form>
<%
   PrologSession pSession =
   PrologSession.getPrologSession("prolog/PrologSession", session);

   String evQuery = request.getParameter("query");
   String output = "";
   if (evQuery != null) {
     Bindings bindings = new Bindings().bind("E",evQuery + '.');
     QueryAnswer answer =
        pSession.executeQuery("sum(E,Sum,Average,Count)", bindings);
     PBTerm average = answer.getValue("Average");
     if (average != null) {
        PBTerm sum = answer.getValue("Sum");
        PBTerm count = answer.getValue("Count");

        output = "<h4>Average =" + average + ", Sum = "
        + sum + " Count = " + count + "</h4>";
     } else {
        output = "<h4>Error: " + answer.getError() + "</h4>";
     }
  }
%>
<%= output  %><br></font>
<p><hr>Powered by SICStus Prolog
</body></html>
```

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 `HTTPSession`, and the variable *request* is bound to the current `HTTPRequest`. Since the `HTTPSession` 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 use a predicate `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.

10.40.6.3 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 directory 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:

```
<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 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

10.40.7 .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.

Detailed documentation on the .NET APIs of PrologBeans is available in the PrologBeans.NET documentation files in the directory http://www.sics.se/sicstus/docs or locally at '%SP_PATH%/doc/html/prologbeans.NET' (where SP_PATH is the path to your SICStus Prolog installation directory).

10.40.8 .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')).
```

10.40.8.1 C# Examples

.NET Embedding. If you have an advanced Prolog application that needs a GUI 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 10.40.3 [PB First Example], page 683).

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 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.

10.40.8.2 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 10.40.3 [PB First Example], page 683. 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

10.41 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@sics.se) if you have suggestions for how this library could be improved.

10.41.1 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.

10.41.2 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:

```
comclient_invoke_method_proc(App, [workbooks, add])
```

or as

```
comclient_invoke_method_fun(App, workbooks, WorkBooks),
comclient_invoke_method_proc(WorkBooks, add),
comclient_release(WorkBooks)
```

10.41.3 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 *stu* 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:

```
| ?- comclient_clsid_from_progid('Excel.Application, CLSID),
     comclient_progid_from_clsid(CLSID, ProgID).
CLSID = '{00024500-0000-0000-C000-000000000046}',
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.

```
| ?- comclient_iid_from_name('IDispatch', IID).
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*.

```
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*.

```
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 datastructures 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.

10.41.4 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),
   %% Visuall Basic: app.visible = 1
   comclient_invoke_put(App, visible, 1),
   %% VB: app.workbooks.add
   comclient_invoke_method_proc(App, [workbooks, add]),
   %% VB: 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,
   %% VB: .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(_),

   %% VB: .range A1:O15 .Clear
   comclient_invoke_method_proc(ActiveSheet, [range(Range),clear]),

   %% Avoid Excel query "do you want to save..."
   %%  VB: 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(_),
```

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 metavariable 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 isn’t 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, 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 isn't 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. Most hooks must be defined in module `user`. For a hook, the mode and type annotations should be seen as guide-lines to the user who wants to add his own hook; they describe how the predicate is used by the system.

extendible A multifile predicate, to which new clauses can be added by the user. For such a predicate, the mode and type annotations should be seen as guide-lines to the user who wants to extend the predicate; 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.

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:

```
assert(mod:a(1), Ref)
```

If no module prefix is supplied, it will implicitly be set to the calling module. If the module prefix is a variable, an instantiation error will be raised. If it is not an atom 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

```
:- initialization +Goal                                    (A)
+Term1 @> +Term2                                           (B)
```

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:

```
:- initialization(+Goal)                                   (C)
@>(+Term1, +Term2)                                         (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, 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.

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 720.

If an output argument is given the type ***var***, 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).

db_reference ::= *term* {with principal functor `'$ref'/2`}
stream_object ::= *term* {as defined in Section 4.6.7.1 [ref-iou-sfh-sob], page 93}
term ::= {any Prolog term}
list of Type ::= `[]` | `[`***Type*****|*****list of Type***`]`
one of [Element|Rest] ::= *Element* | *one of Rest*
arity ::= {an *integer* *X* in the range 0..255}
byte ::= {an *integer* *X* in the range 0..255}
char ::= {an *atom* consisting of a single character}
chars ::= *list of char*
code ::= {an *integer* *X* > 0}
codes ::= *list of code*
pair ::= ***term*-*term***
simple_pred_spec ::= ***atom*/*arity***
pred_spec ::= *simple_pred_spec* | ***atom*:*pred_spec***
pred_spec_forest ::= `[]` | *pred_spec*
| `[`***pred_spec_forest*****|*****pred_spec_forest***`]`
| ***pred_spec_forest*,*pred_spec_forest***
pred_spec_tree ::= *pred_spec* | *list of pred_spec*
foreign_spec ::= *callable* {all arguments being *foreign_arg*}
foreign_arg ::= `+`***interf_arg_type*** | `-`***interf_arg_type*** | `[-`***interf_arg_type***`]`
interf_arg_type ::= `integer` | `float` | `atom`
| `term` | `codes` | `string` | `address` | `address(`***atom***`)`
{see the description in Section 6.2.3 [Conversions between Prolog Arguments and C Types], page 256}
le_spec ::= *atom* | ***atom*(*file_spec*)**
expr ::= {everything that is accepted as second argument to `is/2`; see the description of arithmetic expressions in Section 4.7.5 [ref-ari-aex], page 105.}

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 isn't 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, 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***`)` ***ISO***
: *B* is the bag of instances of *X* such that *P* is provable

`findall(`***?T***`,`***:G***`,`***-L***`)` ***ISO***
`findall(`***?T***`,`***:G***`,`***?L***`,`***?R***`)`
: *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

11.2.3 Character I/O

`at_end_of_line`
`at_end_of_line(+`***S***`)`
: testing whether at end of line on input stream *S*

`at_end_of_stream` ***ISO***
`at_end_of_stream(+`***S***`)` ***ISO***
: testing whether end of file is reached for the input stream *S*

`flush_output` ***ISO***
`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)` ***ISO***
`get_code(+S,-C)` ***ISO***
: *C* is the next character code on text input stream *S*

`nl` ***ISO***
`nl(+S)` ***ISO***
: send a newline to stream *S*

`peek_byte(+C)` ***ISO***
`peek_byte(+S,+C)` ***ISO***
: 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)`
`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 (record) 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*** — ***ISO***
: goal *P* is not provable

?X* ^ *:P there exists an *X* such that *P* is provable (used in `setof/3` and `bagof/3`)

`block` ***:P*** — ***declaration***
: declaration that predicates specified by *P* should block until sufficiently instantiated

`call(`***:P***`)` — ***ISO***
`call(`***:P***`,...)`
: execute *P* or ***P***`(...)`

`call_cleanup(`***:Goal***`,`***:Cleanup***`)`
: Executes the procedure call *Goal*. When *Goal* succeeds determinately, is cut, fails, or raises an exception, *Cleanup* is executed.

`call_residue_vars(`***:Goal***`,`***?Vars***`)`
: Executes the procedure call *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.

`fail` — ***ISO***
: fail (start backtracking)

`false` same as fail

`freeze(+`***Var***`,`***:Goal***`)`
: Blocks *Goal* until `nonvar(`***Var***`)` holds.

`if(`***:P***`,`***:Q***`,`***:R***`)`
: for each solution of *P* that succeeds, prove *Q*; if none, prove *R*

`once(`***:P***`)` — ***ISO***
: Find the first solution, if any, of goal *P*.

`otherwise`
: same as true

`repeat` ***ISO***
: succeed repeatedly on backtracking

`true` ***ISO***
: succeed

`when(+`***Cond***`,`***:Goal***`)`
: block *Goal* until *Cond* holds

11.2.5 Database

`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

`assertz(`***:C***`)` ***ISO***
`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***`)` ***ISO***
`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*** ***ISO,declaration***
: 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)` ***ISO***
: erase the first dynamic clause that matches *C*

`retractall(:H)`
: erase every clause whose head matches *H*

11.2.6 Debugging

`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

`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` ***development***
: 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`)

`profile_data(`***:Spec,?Selection,?Resolution,?Data***`)` ***development***
: *Data* is the profiling data collected from the instrumented predicates covered by *Spec* with selection and resolution *Selection* and *Resolution* respectively.

`profile_reset(`***:Spec***`)` ***development***
: The profiling counters for the instrumented predicates covered by *Spec* are zeroed.

`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***
: *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***`)` ***ISO***
raise exception *E*

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

`current_stream(`***?F,?M,?S***`)`
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)` ***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)`
: 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))` ***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

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)` ***hookable***
: term *T* expands to term *X* using `user:term_expansion/6` or grammar rule expansion

`phrase(:P, -L)`
`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, +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 over-ride 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, 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` ***ISO***
: 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)`
: sorting the list *L* into order yields *S*

11.2.14 Loading Programs

`[]`
`[:F|+Fs]` same as `load_files([F|Fs])`

`block :P` ***declaration***
: predicates specified by *P* should block until sufficiently instantiated

`user:term_expansion(+Term1, +Layout1, +Tokens1, -Term2, -Layout2, -Tokens2)` ***hook***
: Overrides or complements the standard transformations to be done by `expand_term/2`.

`compile(:F)`
: load compiled clauses from files *F*

`consult(:F)`
`reconsult(:F)`
: load interpreted clauses from files *F*

`expand_term(+T,-X)` ***hookable***
: term *T* expands to term *X* using `user:term_expansion/6` or grammar rule expansion

`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

`ensure_loaded(:F)` ***ISO***
: load *F* if not already loaded

`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*** ***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***`)`
: restore the state saved in file *F*

`use_module(`***:F***`)`
`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

`volatile` ***:P*** ***declaration***
: predicates specified by *P* are not to be included in saves

11.2.15 Memory

`garbage_collect`
: force an immediate garbage collection

`garbage_collect_atoms`
: garbage collect atom space

`statistics`
: 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)` ***extendible***
: determines the mapping from a message term into a sequence of lines of text to be printed

`user:generate_message_hook(+M,?S0,?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)` ***extendible***
: 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, it is assumed that the query has been processed and nothing further is done.

`'SU_messages':query_class(+QueryClass, -Prompt, -InputMethod, -MapMethod, -FailureMode)` ***extendible***
: 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)` ***extendible***
: 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)` ***extendible***
: 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***`)`
: *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 specifed 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

11.2.18 Program State

`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***
`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

`user:runtime_entry(+`***S***`)` ***hook***
: entry point for a runtime system

`save_files(+`***L***`,+`***F***`)`
: Saves the modules, predicates and clauses and directives in the given files *L* into file *F*

`save_modules(+`***L***`,+`***F***`)`
: save the modules specifed 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*** ***declaration***
: declares predicates specified by *P* to not be included in saves.

11.2.20 Term Comparison

`compare(`***-C***`,+`***X***`,+`***Y***`)` ***meta-predicate***
: C is the result of comparing terms *X* and *Y*

`+`***X***` == +`***Y*** ***ISO***
: terms *X* and *Y* are strictly identical

+*X* \== +*Y* **ISO**
: terms *X* and *Y* are not strictly identical

+*X* @< +*Y* **ISO**
: term *X* precedes term *Y* in standard order for terms

+*X* @>= +*Y* **ISO**
: term *X* follows or is identical to term *Y* in standard order for terms

+*X* @> +*Y* **ISO**
: term *X* follows term *Y* in standard order for terms

+*X* @=< +*Y* **ISO**
: term *X* precedes or is identical to term *Y* in standard order for terms

11.2.21 Term Handling

?T =.. *?L* **ISO**
: the functor and arguments of term *T* comprise the list *L*

?X = *?Y* **ISO**
: terms *X* and *Y* are unified

+*X* \= +*Y* **ISO**
: terms *X* and *Y* no not unify

?=(+*X*,+*Y*)
: *X* and *Y* are either strictly identical or do not unify

arg(+*N*,+*T*,*-A*) **ISO**
: the *N*th argument of term *T* is *A*

atom_chars(*?A*,*?L*) **ISO**
: *A* is the atom containing the character atoms in list *L*

atom_codes(*?A*,*?L*) **ISO**
: *A* is the atom containing the characters in code-list *L*

atom_concat(*?Atom1*,*?Atom2*,*?Atom12*) **ISO**
: Atom *Atom1* concatenated with *Atom2* gives *Atom12*.

atom_length(+*Atom*,*-Length*) **ISO**
: *Length* is the number of characters of the atom *Atom*.

char_code(*?Char*,*?Code*) **ISO**
: *Code* is the character code of the one-char atom *Char*.

copy_term(+*T*,*-C*) **ISO**
: *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(+Var,-Goal)`
: The goal *Goal* is blocked on the variable *Var*.

`functor(?T,?F,?N)` ***ISO***
: 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)` ***ISO***
: *N* is the numeric representation of list of character atoms *L*

`number_codes(?N,?L)` ***ISO***
: *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)` ***ISO***
: 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*.

`unify_with_occurs_check(?X, ?Y)` ***ISO***
: True if *X* and *Y* unify to a finite (acyclic) term.

11.2.22 Term I/O

`char_conversion(+InChar, +OutChar)` ***ISO***
: The mapping of *InChar* to *OutChar* is added to the character-conversion mapping.

`current_char_conversion(?InChar, ?OutChar)` ***ISO***
: *InChar* is mapped to *OutChar* in the current character-conversion mapping.

`current_op(?P,?T,?A)` ***ISO***
: atom *A* is an operator of type *T* with precedence *P*

`display(+T)`
: write term *T* to the user output stream in prefix notation

`format(+C,:A)`
`format(+S,+C,:A)`
: write arguments *A* on stream *S* according to control string *C*

`op(+P,+T,+A)` ***ISO***
: make atom *A* an operator of type *T* with precedence *P*

`user:portray(+T)` ***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)` ***hookable***
`print(+S,+T)` ***hookable***
: display the term *T* on stream *S* using `user:portray/1` or `write/2`

`read(-T)` ***ISO***
`read(+S,-T)` ***ISO***
: read term *T* from stream *S*

`read_term(-T,+O)` ***ISO***
`read_term(+S,-T,+O)` ***ISO***
: read *T* from stream *S* according to options *O*

`write(+T)` ***ISO***
`write(+S,+T)` ***ISO***
: write term *T* on stream *S*

`write_canonical(+T)` ***ISO***
`write_canonical(+S,+T)` ***ISO***
: write term *T* on stream *S* so that it can be read back by `read/[1,2]`

`writeq(+T)` ***ISO***
`writeq(+S,+T)` ***ISO***
: write term *T* on stream *S*, quoting atoms where necessary

`write_term(+T,+O)` ***ISO,hookable***
`write_term(+S,+T,+O)` ***ISO,hookable***
: writes *T* to *S* according to options *O*

11.2.23 Type Tests

`atom(+T)` ***ISO***
: term *T* is an atom

`atomic(+T)` ***ISO***
: term *T* is an atom or a number

`callable(+T)`
: *T* is an atom or a compound term

`compound(+T)` ***ISO***
: *T* is a compound term

`float(+N)` ***ISO***
: *N* is a floating-point number

`ground(+T)`
: term *T* is a nonvar, and all substructures are nonvar

`integer(+T)` ***ISO***
: term *T* is an integer

`mutable(+X)`
: *X* is currently instantiated to a mutable term.

`nonvar(+T)` ***ISO***
: term *T* is one of atom, number, compound (that is, *T* is instantiated)

`number(+N)` ***ISO***
: *N* is an integer or a float

`simple(+T)`
: *T* is not a compound term; it is either atomic or a var

`var(+T)` ***ISO***
: 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 703.

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 700.

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 699.

11.3.1 abolish/[1,2] *[ISO]*

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.

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 ground.

`type_error`
: if a *Name* is not an atom or an *Arity* not an integer.

`domain_error`
: if a *PredSpec* is not a valid procedure specification, or 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.

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 4.15.6 [ref-ere-int], page 184.

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.

See Also

`halt/[0,1]`, `break/0`, `runtime_entry/1`, Section 4.15.6 [ref-ere-int], page 184.

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 the absolute filename that corresponds to the relative file specification *RelFileSpec*.

Arguments

RelFileSpec
: *le_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.

  `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`
    `file` implies `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`
: *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 themself but are reserved for future expansion of the allowable patterns.

  Any other character matches itself.

  Please note: Currently glob pattern matching is *case sensitive*, even under Windows (where all expanded file names are subjected to case-normalization). This means that a pattern containing upper case Latin 1 characters or lower case non-Latin 1 characters will never match a file name under Windows. This may be corrected in the future.

  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, *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`
`append` If *AbsFileName* exists, it must be writable. If it doesn't exist, it must be possible to create.

`exist` The file represented by *AbsFileName* must exist.

`execute`
`executable`
The file represented by *AbsFileName* must be executable and exist. This is ignored if `file_type(directory)` is also specified.

`search`
`searchable`
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, 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 makes 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.

`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 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 `''`, file names will be treated as relative to the current working directory. If a regular, existing file is given, 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, the default is the directory containing that file. Otherwise, the default is the current working directory.

Description

If *FileSpec* is `user`, then *AbsFileName* is unified with `user`; this "file name" stands for the standard input or output stream, depending on context. Otherwise, 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:

Syntactic rewriting

Pattern expansion

Extension expansion

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, a permission exception is raised. If the file obeys the access mode option, `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, an existence exception is raised.

FileSpec can be a file search paths, e.g. `library('lists.pl')`. It can also refer to environment variables and the home directory of users. See Section 4.5.2 [ref-fdi-syn], page 84, 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, 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 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}`, it is guaranteed[1] that the file can be opened with `open/[3,4]`. If the access option was `exist`, the file does exist, but might be both read and write protected.

If `file_type(directory)` is not given, 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, these are resolved by removing directory components from the pathname, not by acessing 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.

Examples

To check whether the file '`my_text`' exists in the home directory, with one of the extensions '`.text`' or '`.txt`', and is both writable and readable:

```
| ?- absolute_file_name('~/my_text', File,
                         [extensions(['.text','.txt']),
                         access([read,write])]).
```

To check whether the directory '`bin`' exists in the home directory:

```
| ?- absolute_file_name('~/bin', Dir,
                         [file_type(directory),
                         access(exist)]).
```

Here *Dir* would get a slash terminated value, such as `/home/joe/`.

To list all files in the current directory:

[1] To the extent that the access permissions can be precisely determined. See the **`access/1`** option above.

```
| ?- findall(File, absolute_file_name('.', File,
                          [glob('*'),
                          solutions(all), file_errors(fail)]), Files).
```

To list all directories in the parent of the current directory containing the string “sicstus”:

```
| ?- findall(File, absolute_file_name('..', File,
                          [glob('*sicstus*'),file_type(directory),
                          solutions(all), file_errors(fail)]), Files).
```

To find a file ‘cmd.exe’ in any of the “usual places” where executables are found, i.e. by looking through the PATH environment variable:

```
| ?- absolute_file_name(path('cmd.exe'), File,
                          [access(exist)]).
```

This uses the predefined file search path path/1, Section 4.5 [ref-fdi], page 80.

See Also

file_search_path/2, Section 4.5 [ref-fdi], page 80.

11.3.4 `add_breakpoint/2` *[development]*

Synopsis

`add_breakpoint(+`***Conditions***`, `***-BID***`)`

Creates a breakpoint with *Conditions* and with identifier *BID*.

Arguments

:Conditions
: *term*, must be ground

  Breakpoint conditions.

BID
: *integer*

  Breakpoint identifier.

See Also

Section 5.6.1 [Creating Breakpoints], page 208, Section 5.7 [Breakpoint Predicates], page 237.

11.3.5 `,/2` *[ISO]*

Synopsis

+P `,` *+Q*

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*** `,` ***Q***`))` to execute *P* and then *Q*.

Backtracking

Depends on *P* and *Q*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.6 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 [*T1*,...,*Tn*] for some *n*. In that instance, the following things apply:

1. `append(L, X, Y)` has at most one solution, whatever *X* and *Y* are, and cannot backtrack at all.
2. `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. `append(X, L, Y)`, however, can backtrack indefinitely if *X* and *Y* are variables.

Examples

The following examples are perfectly ordinary uses of `append/3`:

To enumerate adjacent pairs of elements from a list:

```
next_to(X, Y, (*in*) List3) :-
        append(_, [X,Y|_], List3).
```

To check whether Word1 and Word2 are the same except for a single transposition. (`append/5` in `library(lists)` would be better for this task.)

```
one_transposition(Word1, Word2) :-
        append(Prefix, [X,Y|Suffix], Word1),
        append(Prefix, [Y,X|Suffix], Word2).

| ?- 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:

```
comma_phrase(List3, Phrase) :-
        append(F, [',' |Rest], List3),
        !,
        (   Phrase = F
        ;   comma_phrase(Rest, Phrase)
        ).
comma_phrase(List3, List3).

| ?- comma_phrase([this,is,',',um,',',an, example], X).
X = [this,is] ;
X = [um] ;
X = [an,example] ;
no
```

See Also

Section 4.8.3 [ref-lte-acl], page 110, `library(lists)`.

11.3.7 arg/3 *[ISO]*

Synopsis

`arg(+`***ArgNum***`, +`***Term***`, -`***Arg***`)`

unifies *Arg* with the *ArgNum*th argument of term *Term*.

Arguments

ArgNum *integer*, must be nonvar and positive

Term *compound*, must be nonvar

Arg *term*

Description

The arguments are numbered from 1 upwards.

Exceptions

`instantiation_error`
: if *ArgNum* or *Term* is unbound.

`type_error`
: if *ArgNum* is not an integer.

Examples

```
| ?- arg(2, foo(a,b,c), X).

X = b
```

See Also

`functor/3`, `=../2`, Section 4.8.2 [ref-lte-act], page 110.

11.3.8 `ask_query/4` *[hookable]*

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

QueryClass
: *term*, must be nonvar

  Determines the allowed values for the atom *Answer*.

Query
: *term*

  A message term.

Help
: *term*

  A message term.

Answer
: *term*

  See *QueryClass*

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 189.

11.3.9 assert/[1,2] *[ISO]*

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*

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, *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 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 signalled if *Ref* does not unify with its return value. This exception is signalled after the assert has been completed.

The procedure for *Clause* must be dynamic or undefined. If it is undefined, 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, the new clause is not encountered on backtracking by that call. See Section 4.12.1 [ref-mdb-bas], page 152 for further explanation of what happens when currently running code is modified.

Any uninstantiated variables in the *Clause* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

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.

`permission_error`
: if the procedure corresponding to *Head* is not static.

See Also

Section 4.12.4 [ref-mdb-acd], page 154.

11.3.10 `asserta/[1,2]` *[ISO]*

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*

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, *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 it is taken to mean `call(Body)`. For example, (A) is equivalent to (B):

```
| ?- asserta((p(X) :- X)).                                        (A)
| ?- asserta((p(X) :- call(X))).                                  (B)
```

Ref should be uninstantiated; a range exception is signalled if *Ref* does not unify with its return value. This exception is signalled after the assert has been completed.

The procedure for *Clause* must be dynamic or undefined. If it is undefined, 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, the new clause is not encountered on backtracking by that call. See Section 4.12.1 [ref-mdb-bas], page 152 for further explanation of what happens when currently running code is modified.

Any uninstantiated variables in the *Clause* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

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.

`permission_error`
: if the procedure corresponding to *Head* is not static.

See Also

Section 4.12.4 [ref-mdb-acd], page 154.

11.3.11 assertz/[1,2] *[ISO]*

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*

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, *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 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 signalled if *Ref* does not unify with its return value. This exception is signalled after the assert has been completed.

The procedure for *Clause* must be dynamic or undefined. If it is undefined, 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, the new clause is not encountered on backtracking by that call. See Section 4.12.1 [ref-mdb-bas], page 152 for further explanation of what happens when currently running code is modified.

Any uninstantiated variables in the *Clause* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

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.

`permission_error`
: if the procedure corresponding to *Head* is not static.

See Also

Section 4.12.4 [ref-mdb-acd], page 154.

11.3.12 at_end_of_line/[0,1]

Synopsis

`at_end_of_line`

`at_end_of_line(+Stream)`

Test whether end of line (record) 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 (record) 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.

Is also true whenever `at_end_of_stream/[0,1]` is true.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94), plus:

`existence_error`

Some operating system dependent error occurred in reading.

Comments

Coding with `at_end_of_line/[0,1]` to check for end of line is more portable among different operating systems than checking end of line by the input character code.

See Also

`at_end_of_stream/[0,1]`, `skip_line/[0,1]`, `get0/[1,2]`, `set_input/1`.

11.3.13 `at_end_of_stream/[0,1]` *[ISO]*

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 if 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 94), plus:

`existence_error`

Some operating system dependent error occurred in reading.

Comments

`at_end_of_stream/[0,1]` peeks ahead for next input item if there is no item available on the buffer of the specified input stream.

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 actally at its end. If you want ensure that end of stream condition is always properly detected, even if that entails blocking until further input is possible, you can use `peek_code/[1,2]` or `peek_byte/[1,2]`.

See Also

`at_end_of_line/[0,1]`.

11.3.14 `atom/1` *[ISO]*

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
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.15 `atom_chars/2` *[ISO]*

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 and *Chars* are both uninstantiated

`type_error`
Atom is not a number or *Chars* is not a list, or *Chars* is not a *chars*

`representation_error`
Chars is a list corresponding to an atom that can't be represented

See Also

`atom_codes/2`.

11.3.16 `atom_codes/2` *[ISO]*

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* and *Codes* are both uninstantiated

`type_error`
: *Atom* is not an atom or *Codes* is not a list

`domain_error`
: *Codes* is not a *codes*

`representation_error`
: *Codes* is a list corresponding to an atom that can't be represented

See Also

`atom_chars/2`.

11.3.17 atom_concat/3 *[ISO]*

Synopsis

`atom_concat(+Atom1,+Atom2,-Atom12)`

`atom_concat(-Atom1,-Atom2,+Atom12)`

The characters of the atom *Atom1* concatenated with those of *Atom2* are the same as the characters of atom *Atom12*.

Arguments

Atom1 *atom*

Atom2 *atom*

Atom12 *atom*

Description

Initially, either both *Atom1* and *Atom2*, or *Atom12*, must be instantiated to atoms. If only *Atom12* is instantiated, nondeterminately enumerates all possible atom-pairs that concatenate to the given atom, e.g.:

```
| ?- atom_concat(A, B, 'ab').

A = '',
B = ab ? ;

A = a,
B = b ? ;

A = ab,
B = ''  ;

no
```

Exceptions

`instantiation_error`
: More than one argument uninstantiated.

`type_error`
: An instantiated argument is not an atom.

`representation_error`
: *Atom12* is too long to be represented.

See Also

`atom_length/2`, `sub_atom/5`.

11.3.18 `atom_length/2` *[ISO]*

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

`domain_error`
: *Length* < 0

See Also

`atom_length/2`, `atom_concat/3`, `sub_atom/5`.

11.3.19 `atomic/1` *[ISO]*

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
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.20 bagof/3 *[ISO]*

Synopsis

`bagof(+Template, +Generator, -Set)`

Like `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 `call/1`.

Set *list of term*, non-empty set

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

Examples

See `findall/3` for examples that illustrate the differences among `findall/3`, `setof/3`, and `bagof/3`.

See Also

`findall/3`, `setof/3`, `^/2`, Section 4.13 [ref-all], page 162.

11.3.21 bb_delete/2

Synopsis

`bb_delete(+Key, -Term)`

If a term is currently stored under *Key*, the term is deleted, and a copy of it is unified with *Term*. Otherwise, `bb_delete/2` silently fails.

Arguments

:Key *atomic*, must be nonvar

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 160.

11.3.22 `bb_get/2`

Synopsis

`bb_get(+`***Key***`, -`***Term***`)`

If a term is currently stored under *Key*, a copy of it is unified with *Term*. Otherwise, `bb_get/2` silently fails.

Arguments

:Key *atomic*, must be nonvar

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 160.

11.3.23 `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 *atomic*, must be nonvar

Term *term*

Description

Any uninstantiated variables in the *Term* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

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 160.

11.3.24 `bb_update/3`

Synopsis

`bb_update(+Key, -OldTerm, +NewTerm)`

If a term is currently stored under *Key* and unifies with *OldTerm*, the term is replaced by a copy of *NewTerm*. Otherwise, `bb_update/3` silently fails. This predicate provides an atomic swap operation.

Arguments

:Key *atomic*, must be nonvar

OldTerm *term*

NewTerm *term*

Description

Any uninstantiated variables in the *NewTerm* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

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 160.

11.3.25 `(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`, 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`, 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

`instantiation_error`

BlockSpec not ground.

`context_error`

“declaration appeared in query”

See Also

Section 4.3.4.5 [Block Declarations], page 73.

11.3.26 `break/0` *[development]*

Synopsis

`break`

causes 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, 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-rcd], page 155 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***.

See Also

`abort/0`, `halt/[0,1]`, Section 3.9 [Nested], page 27.

11.3.27 `breakpoint_expansion/2` *[development,hook]*

Synopsis

`:- multifile user:breakpoint_expansion/2.`

`user:breakpoint_expansion(+`***`Macro`***`, `***`-Body`***`)`

Defines debugger condition macros.

Arguments

Macro *term*

Breakpoint test or action.

Body *term*

Expanded breakpoint test or action, may be composite.

See Also

Section 5.9 [Breakpoint Conditions], page 241.

11.3.28 `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 94).

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 93.

11.3.29 `call/_53553` *[ISO]*

Synopsis

`call(+P)`

Proves (executes) *P*.

`call(+P,?Q,...)`

Executes the goal obtained by augmenting *P* by the remaining arguments.

Arguments

:P *callable*, must be nonvar

Q term ...

Description

If *P* is instantiated to an atom or compound term, then the goal `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*.

Backtracking

Depends on *P*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2.5 [ref-sem-cal], page 66.

11.3.30 `call_cleanup/2`

`call_cleanup(+Goal, +Cleanup)`

Synopsis

Executes the procedure call *Goal*. When *Goal* succeeds determinately, is cut, fails, or raises an exception, *Cleanup* is executed.

Arguments

:Goal *callable*, must be nonvar

:Cleanup *callable*, must be nonvar

Description

This construction can be used to ensure that *Cleanup* is executed as soon as *Goal* has completed execution, no matter how it finishes. In more detail:

When `call_cleanup/2` with a continuation *C* is called or backtracked into, first *Goal* is called or backtracked into. Then there are four possibilities:

1. *Goal* succeeds determinately, possibly leaving some blocked subgoals. *Cleanup* is executed with continuation *C*.
2. *Goal* succeeds with some alternatives outstanding. Execution proceeds to *C*. If a cut that removes the outstanding alternatives is encountered, *Cleanup* is executed with continuation to proceed after the cut. Also, if an exception *E* that will be caught by an ancestor of the `call_cleanup/2` *Goal* is raised, *Cleanup* is executed with continuation `raise_exception(E)`.
3. *Goal* fails. *Cleanup* is executed with continuation `fail`.
4. *Goal* raises an exception *E*. *Cleanup* is executed with continuation `raise_exception(E)`.

In a typical use of `call_cleanup/2`, *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 `halt/[0,1]` and `abort/0`, are implemented in terms of exceptions. All of these circumstances can trigger the execution of *Cleanup*.

Backtracking

Depends on the arguments.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.31 `call_residue_vars/2`

`call_residue_vars(+Goal, -Vars)`

Synopsis

Executes the procedure call *Goal*, unifying *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

Goal is executed as if by `call/1`. *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:

```
| ?- call_residue_vars((dif(X,f(Y)), X=f(Z)), Vars).

X = f(Z),
Vars = [Z,Y],
prolog:dif(f(Z),f(Y)) ?
```

Backtracking

Depends on *Goal*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2.4 [ref-sem-sec], page 63.

11.3.32 `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
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.33 `catch/3` *[ISO]*

Synopsis

`catch(+`***`ProtectedGoal`***`, `***`-Exception`***`, +`***`Handler`***`)`

same as:

`on_exception(`***`Exception`***`, `***`ProtectedGoal`***`, `***`Handler`***`)`

Arguments

:ProtectedGoal
: *callable*, must be nonvar

Exception
: *term*

:Handler
: *callable*, must be nonvar

Backtracking

Depends on *ProtectedGoal* and *Handler*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.15 [ref-ere], page 173.

11.3.34 `char_code/2` *[ISO]*

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`.

11.3.35 `char_conversion/2` *[ISO]*

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.

Exceptions

`instantiation_error`
`type_error`

See Also

Chapter 2 [Glossary], page 7.

11.3.36 `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-tty Prolog stream, the character count of the Prolog stream is increased by one. Character count for a tty stream reflects the total character input from or output to the tty since the tty is opened to any stream.

A `nl/[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 94).

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 93.

11.3.37 `clause/[2,3]` *[ISO]*

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, `(`***Head*** `:-` ***Body***`)` is unified with the clause identified by *Ref*. (If this clause is a unit-clause, *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, `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 of type callable. *Ref* is not a syntactically valid database reference.

`permission_error`
  Procedure is not dynamic.

`existence_error`
: *Ref* is a well-formed database reference but does not correspond to an existing clause or record.

Comments

If `clause/[2,3]` is called on an undefined procedure 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 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, 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 9.4 [Indexing], page 309.

See Also

`instance/2`, `assert/[1,2]`, `dynamic/1`, `retract/1`, Section 4.12.6 [ref-mdb-acl], page 157.

11.3.38 `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 bidirectional streams, such as sockets.

Description

If *Stream* is a stream object, then if the corresponding stream is open, it will be closed in the specified directions; otherwise, an error exception is raised.

If *Stream* is a file specification, 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-iou-sfh-est], page 94), plus:

`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 93, Section 10.21 [lib-sockets], page 453.

11.3.39 `compare/3`

Synopsis

`compare(`***-Order***`, +`***Term1***`, +`***Term2***`)`

succeeds if the result of comparing terms *Term1* and *Term2* is *Order*

Arguments

Order *one of* `[<,=,>]`

- `=` if *Term1* is identical to *Term2*,
- `<` if *Term1* is before *Term2* in the standard order,
- `>` if *Term1* is after *Term2* in the standard order.

Term1 *term*

Term2 *term*

Description

The standard total order is described in Section 4.8.8 [ref-lte-cte], page 113.

The goal (A) is equivalent to (B):

```
| ?- compare(=, Term1, Term2).                    (A)

|?- (Term1 == Term2).                             (B)
```

The following query succeeds, binding *R* to `<`, because 1 comes before 2 in the standard order.

```
 | ?- compare(R, 1, 2).

R = <
```

If *Order* is supplied, and is not one of `<`, `>`, or `=`, `compare/3` simply fails.

See Also

`@</2`, `@=</2`, `@>/2`, `@>=/2`, `SP_compare()`, Section 4.8.8 [ref-lte-cte], page 113.

11.3.40 compile/1

Synopsis

`compile(+Files)`

Compiles the specified Prolog source file(s) into memory.

Arguments

:Files *le_spec* or *list of le_spec*, must be ground
: A file specification or a list of file specifications; extensions optional.

Description

This predicate is defined as if by:

```
compile(Files) :-
        load_files(Files, [load_type(source),compilation_mode(compile)]).
```

Exceptions

See `load_files/[2,3]`.

See Also

Section 4.3.2 [ref-lod-lod], page 69.

11.3.41 compound/1 *[ISO]*

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
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.42 `consult/1`

Synopsis

`consult(+`***`Files`***`)`

Consults the specified Prolog source file(s) into memory.

Arguments

:Files *le_spec* or *list of le_spec*, must be ground

A file specification or a list of file specifications; extensions optional.

Description

This predicate is defined as if by:

```
consult(Files) :-
        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 69.

11.3.43 `copy_term/[2,3]` *[ISO]*

Synopsis

`copy_term(+`***Term***`, -`***Copy***`)`

Makes a copy of *Term* in which all variables have been replaced by new variables that occur nowhere outside the newly created term.

`copy_term(+`***Term***`, -`***Copy***`, -`***Body***`)`

Furthermore, if *Term* contains attributed variables, unifies *Body* with a term such that executing *Body* will reinstate equivalent attributes on the variables in *Copy*. Otherwise, *Body* is unified with `true`.

Arguments

Term *term*

Copy *term*

Body *callable*

Description

Independent copies are substituted for any mutable terms in *term*. It behaves as if defined by:

```
copy_term(X, Y) :-
        assert('copy of'(X)),
        retract('copy of'(Y)).
```

The implementation of `copy_term/2` conserves 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:

```
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, 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
   ).
```

See Also

Section 4.8.7 [ref-lte-cpt], page 111.

11.3.44 `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 114.

11.3.45 `current_atom/1`

Synopsis

`current_atom(?Atom)`

Atom is a currently existing atom.

Arguments

Atom *atom*

Backtracking

If *Atom* is uninstantiated, `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.

See Also

Section 4.12.8 [ref-mdb-idb], page 159.

11.3.46 `current_breakpoint/5` *[development]*

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]`

See Also

Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 225, Section 5.7 [Breakpoint Predicates], page 237.

11.3.47 `current_char_conversion/2` *[ISO]*

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*

See Also

Chapter 2 [Glossary], page 7.

11.3.48 `current_input/1` *[ISO]*

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`.

See Also

`open/[3,4]`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.49 `current_key/2`

Synopsis

`current_key(`***?KeyName*, *?KeyTerm***`)`

Succeeds when *KeyName* is the name of *KeyTerm*, and *KeyTerm* is a recorded key.

Arguments

KeyName *atomic*

One of:

KeyTerm, if *KeyTerm* is atomic; or

the principal functor of *KeyTerm*, if *KeyTerm* is a compound term.

KeyTerm *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.

See Also

Section 4.12.8 [ref-mdb-idb], page 159.

11.3.50 `current_module/[1,2]`

Synopsis

`current_module(`***?ModuleName***`)`

Queries whether a module is "current" or backtracks through all of the current modules.

`current_module(`***?ModuleName***`, `***?AbsFile***`)`

Associates modules with their module-files.

Arguments

ModuleName
  atom

AbsFile *atom*
  Absolute filename in which the module is defined.

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 `current_module/1` will succeed on it but `current_module/2` will fail. This arises for the special module `user` and for dynamically-created modules (see Section 4.11 [ref-mod], page 138).

If its arguments are not correct, or if ***Module*** has no associated file, `current_module/2` simply fails.

Backtracking

`current_module/1` backtracks through all of the current modules. The following command will print out all current modules:

```
| ?- current_module(Module), writeq(Module), nl, fail.
```

`current_module/2` backtracks through all of the current modules and their associated files.

Exceptions

`type_error`

See Also

Section 4.11.13 [ref-mod-ilm], page 146.

11.3.51 `current_op/3` *[ISO]*

Synopsis

`current_op(`***?Precedence***`,` ***?Type***`,` ***?Name***`)`

Succeeds when the atom *Name* is currently an operator of type *Type* and precedence *Precedence*.

Arguments

Precedence
: *integer*, in the range 1-1200

Type
: *one of* `[xfx, xfy, yfx, fx, xf, yf]`

Name
: *atom*

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 `op/3`.

Exceptions

`type_error`
: *Name* not an atom or *Type* not an atom or *Precedence* not an integer.

`domain_error`
: *Precedence* not between 1-1200, or *Type* not one of listed atoms.

Examples

See Also

`op/3`, Section 4.1.5 [ref-syn-ops], page 43.

11.3.52 `current_output/1` *[ISO]*

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`.

See Also

`open/[3,4]`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.53 current_predicate/[1,2] *[ISO]*

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, `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, `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 118).

To find out whether a predicate is built-in, use `predicate_property/2`.

```
% 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*

See Also

`predicate_property/2`, Section 4.9.1 [ref-lps-ove], page 118.

11.3.54 current_prolog_flag/2 *[ISO]*

Synopsis

`current_prolog_flag(`***?FlagName***`,` ***?Value***`)`

same as:

`prolog_flag(`***FlagName***`,` ***Value***`)`

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`.

11.3.55 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 the three special streams for the standard input, output, and error channels.

Backtracking

Can be used to backtrack through all open streams.

See Also

`open/[3,4]`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.56 `!/0` *[ISO]*

Synopsis

`!`

Cut.

Description

When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the parent goal will fail (the parent goal is the one that matched the head of the clause containing the cut).

See Also

Section 4.2.3.1 [ref-sem-ctr-cut], page 59.

11.3.57 `debug/0` *[development]*

Synopsis

`debug`

Turns on the debugger in debug mode.

Description

`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).

See Also

Section 5.2 [Basic Debug], page 199.

11.3.58 `debugger_command_hook/2` *[development,hook]*

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 203.

Arguments

DCommand
: *term*

Actions
: *term*

See Also

Section 5.7 [Breakpoint Predicates], page 237.

11.3.59 debugging/0 *[development]*

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`.

See Also

Section 5.2 [Basic Debug], page 199.

11.3.60 `dif/2`

Synopsis

`dif(+X,+Y)`

Constrains *X* and *Y* to represent different terms i.e. to be non-unifiable.

Arguments

X *term*

Y *term*

Description

Calls to `dif/2` either succeed, fail, or are blocked depending on whether *X* and *Y* are sufficiently instantiated. It is defined as if by:

```
dif(X, Y) :- when(?=(X,Y), X\==Y).
```

See Also

Section 4.2.4 [ref-sem-sec], page 63.

11.3.61 `disable_breakpoints/1` *[development]*

Synopsis

`disable_breakpoints(+`***BIDs***`)`

Disables 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 225, Section 5.7 [Breakpoint Predicates], page 237.

11.3.62 `(discontiguous)/1` *[ISO,declaration]*

Synopsis

`:- 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.

Exceptions

`type_error`
`context_error`

"declaration appeared in query"

See Also

Section 4.3.4.4 [Discontiguous Declarations], page 72.

11.3.63 display/1

Synopsis

`display(+Term)`

Writes *Term* on the standard output stream, without quoting atoms, without operator notation, without treating `'$VAR'/1` terms specially.

Arguments

Term *term*

Description

`display(Term)` is equivalent to:

```
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 94).

See Also

Section 4.6.4 [ref-iou-tou], page 89.

11.3.64 `(dynamic)/1` *[ISO,declaration]*

Synopsis

`:- dynamic` ***+PredSpec***

Declares the clauses of the predicates defined by *PredSpecs* to be dynamic.

Arguments

:PredSpec *pred_spec_forest*, must be ground

A predicate specification, or a list of such, or a sequence of such separated by commas.

Exceptions

`type_error`

`context_error`

“declaration appeared in query”

`permission_error`

Cannot redefine built-in predicate.

Comments

To declare a grammar rule *gram/n* dynamic, the arity of *PredSpec* must be *n*+2.

See Also

Section 4.3.4.2 [Dynamic Declarations], page 72.

11.3.65 `enable_breakpoints/1` *[development]*

Synopsis

`enable_breakpoints(+`***BIDs***`)`

Enables 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 225, Section 5.7 [Breakpoint Predicates], page 237.

11.3.66 `ensure_loaded/1` *[ISO]*

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 *le_spec* or *list of le_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, their public predicates are imported.

This predicate is defined as if by:

```
ensure_loaded(Files) :-
        load_files(Files, [if(changed)]).
```

Exceptions

See `load_files/[2,3]`.

See Also

Section 4.3.2 [ref-lod-lod], page 69.

11.3.67 `=:=` /2 *[ISO]*

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 104).

Examples

```
| ?- 1.0 + 1.0 =:= 2.

yes
| ?- "a" =:= 97.

yes
```

See Also

Section 4.7 [ref-ari], page 103

11.3.68 `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 155.

11.3.69 `error_exception/1` *[development,hook]*

Synopsis

`:- multifile user:error_exception/1.`

`user:error_exception(+Exception)`

Tells the debugger to enter trace mode on exceptions matching *Exception*.

Arguments

Exception *term*

See Also

Section 5.11 [Exceptions Debug], page 250.

11.3.70 `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*

See Also

Section 5.6.7 [Built-in Predicates for Breakpoint Handling], page 225, Section 5.7 [Breakpoint Predicates], page 237.

11.3.71 ^ /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 66).

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 162.

11.3.72 `expand_term/2` *[hookable]*

Synopsis

`expand_term(+`***Term1***`, -`***Term2***`)`

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 `user:term_expansion/6`.

When a source file is compiled, `expand_term/2` is called with the virtual clauses `beginning_of_file` before and `end_of_file` after the real Prolog clauses, to give `user:term_expansion/6` an opportunity to perform some action at the beginning and end of a source file. **Please note**: the virtual clause `beginning_of_file` is "seen" *before* any module declaration, i.e. before the source module has been updated.

Arguments

Term1 *term*

Term2 *term*

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 `-->/2`, into ordinary Prolog clauses, written with `:-/2`. If *Term1* is a grammar rule, then *Term2* is the corresponding clause. Otherwise *Term2* is simply *Term1* unchanged.

Calls `user:term_expansion/6`.

Exceptions

Prints messages for exceptions raised by `user:term_expansion/6`.

See Also

`phrase/[2,3]`, `-->/2`, Section 4.3.5 [ref-lod-exp], page 75.

11.3.73 `fail/0` *[ISO]*

Synopsis

`fail`

Always fails.

See Also

Section 4.2 [ref-sem], page 56.

11.3.74 `false/0`

Synopsis

`false`

Always fails (same as `fail/0`).

See Also

Section 4.2 [ref-sem], page 56.

11.3.75 `file_search_path/2` *[hook]*

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 *le_spec* as input argument.

Arguments

PathAlias *atom*

An atom that represents the path given by *DirSpec*.

DirSpec *le_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(basics)`. 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 `basics` 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. See Section 4.9.4 [ref-lps-flg], page 119 for more info on the Prolog flag `host_type`. The environment variables `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 run-time.

```
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
```

See Also

absolute_file_name/[2,3], library_directory/1, load_files/[1,2], Section 4.5 [ref-fdi], page 80.

11.3.76 `findall/[3,4]` *[ISO]*

Synopsis

`findall(+`***Template***`, +`***Generator***`, `***-List***`)`

`findall(+`***Template***`, +`***Generator***`, `***-List***`, +`***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 `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 '^' 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.

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 66).

Examples

To illustrate the differences among `findall/3`, `setof/3`, and `bagof/3`:

```
| ?- [user].
| foo(1,2).
| foo(1,2).
| foo(2,3).
|
% user compiled in module user, 0.100 sec 352 bytes

yes
| ?- 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

| ?- findall(X, foo(X,Y), L).

X = _3342,
Y = _3384,
L = [1,1,2] ;

no

| ?- setof(X, foo(X,Y), L).

X = _3342,
Y = 2,
L = [1] ;

X = _3342,
Y = 3,
L = [2] ;

no
```

See Also

`bagof/3`, `setof/3`, `^/2`, Section 4.13 [ref-all], page 162.

11.3.77 `float/1` *[ISO]*

Synopsis

`float(+`***Term***`)`

Term is currently instantiated to a float.

Arguments

Term *term*

Examples

```
| ?- float(Term1).

no
| ?- float(5.2).

yes
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.78 `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 94), plus:

`permission_error`

An error occurred in flushing out the buffered output.

Examples

See Also

Section 4.6.7 [ref-iou-sfh], page 93.

11.3.79 `foreign/[2,3]` *[hook]*

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 term of the form ***PredName***(*Argspec*, *Argspec*, ...) as described in Section 6.2.3 [Conversions between Prolog Arguments and C Types], page 256.

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.

See Also

`load_foreign_resource/1`, Section 6.2 [Calling C from Prolog], page 254.

11.3.80 `foreign_resource/2` *[hook]*

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.

See Also

`load_foreign_resource/1`, `foreign/[2,3]`, Section 6.2 [Calling C from Prolog], page 254.

11.3.81 `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 a non-negative 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 <*n*> can be specified, then it can be the character '`*`'. In this case <*n*> will be taken as the next argument from *Arguments*.

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.

'`~`***`N`***`c`' (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.

'`~`***`N`***`e`'
'`~`***`N`***`E`' (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, 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.

'`~`***`N`***`f`'
'`~`***`N`***`F`' (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, in which case a zero appears after the decimal point. At least one digit appears before it and at least one after it. *N* defaults to 6.

'~**N**g'
'~**N**G' (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, 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.

'~**N**h'
'~**N**H' (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`'.

'~**N**d' (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, no decimal point will be printed.

'~**N**D' (Print decimal.) The argument is an integer. Identical to '~**N**d' except that '`,`' will separate groups of three digits to the left of the decimal point.

'~**N**r' (Print radix.) The argument is an integer. *N* is interpreted as a radix, 2 ***N*** 36. If *N* is missing the radix defaults to 8. The letters '`a-z`' will denote digits larger than 9.

'~**N**R' (Print radix.) The argument is an integer. Identical to '~**N**r' except that the letters '`A-Z`' will denote digits larger than 9.

'~**N**s' (Print string.) The argument is a code-list. Exactly *N* characters will be printed. *N* defaults to the length of the string.

'`~i`' (Ignore.) The argument, which may be of any type, is ignored. Example:

'`~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 89).

'`~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 89).

'`~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 89).

'`~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 89).

‘~@’ (Call.) The argument *Arg* is a goal, which will be called as if by `\+ \+ Arg` and is expected to print on the current output stream. If the goal performs other side-effects, the behavior is undefined.

‘~~’ (Print tilde.) Takes no argument. Prints ‘~’.

‘~*N*n’ (Print newline.) Takes no argument. Prints *N* newlines. *N* defaults to 1.

‘~N’ (Print Newline.) Prints a newline if not at the beginning of a line.

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, the column is space padded at the end.

If ‘~|’ or ‘~+’ specifies a position preceding the current position, the boundary is set at the current position.

‘~*N*|’ Set a column boundary at line position *N*. *N* defaults to the current position.

‘~*N*+’ Set a column boundary at *N* positions past the previous column boundary. *N* defaults to 8.

‘~*N*t’ Specify padding in a column. *N* is the fill character code. *N* may also be specified as **‘*C*** where *C* is the fill character. The default fill character is ⟨SPC⟩. Any (‘~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 94), plus:

`consistency_error`
Wrong number of arguments, or argument of the wrong type.

Examples

```
| ?- Pi=3.14159265, format('~e ~2E ~0E\n', [Pi,Pi,Pi]).
3.141593e+00 3.14E+00 3.0E+00

| ?- Pi=3.14159265, format('~f, ~2F, ~0F\n', [Pi,Pi,Pi]).
3.141593, 3.14, 3.0

| ?- format('~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('~h ~h ~2h ~2H ~-1H\n', [F,G,F,G,3.14]).
123000.0 0.000123 1.23e+05 1.23E-04 3.14E+00
```

```
| ?- format('Hello ~1d world!\n', [42]).
Hello 4.2 world!

| ?- format('Hello ~d world!\n', [42]).
Hello 42 world!

| ?- format('Hello ~1D world!\n', [12345]).
Hello 1,234.5 world!

| ?- format('Hello ~2r world!\n', [15]).
Hello 1111 world!

| ?- format('Hello ~16r world!\n', [15]).
Hello f world!

| ?- format('Hello ~16R world!\n', [15]).
Hello F world!

| ?- format('Hello ~4s ~4s!\n', ["new","world"]).
Hello new  worl!

| ?- format('Hello ~s world!\n', ["new"]).
Hello new world!

| ?- format('Hello ~i~s world!\n', ["old","new"]).
Hello new world!

| ?- format('Hello ~k world!\n', [[a,b,c]]).
Hello .(a,.(b,.(c,[]))) world!

| ?- assert((portray([X|Y]) :- print(cons(X,Y)))).
| ?- format('Hello ~p world!\n', [[a,b,c]]).
Hello cons(a,cons(b,cons(c,[]))) world!

| ?- format('Hello ~q world!\n', [['A','B']]).
Hello ['A','B'] world!

| ?- format('Hello ~w world!\n', [['A','B']]).
Hello [A,B] world!

| ?- format('Hello ~@ world!\n', [write(new)]).
Hello new world!

| ?- format('Hello ~~ world!\n', []).
Hello ~ world!

| ?- format('Hello ~n world!\n', []).
Hello
 world!
```

```
| ?-    format('~`*t NICE TABLE ~`*t~61|~n', []),
        format('*~t*~61|~n', []),
        format('*~t~a~20|~t~a~t~20+~a~t~20+~t*~61|~n',
               ['Right aligned','Centered','Left aligned']),
        format('*~t~d~20|~t~d~t~20+~d~t~20+~t*~61|~n',
               [123,45,678]),
        format('*~t~d~20|~t~d~t~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                *
*************************************************************

| ?-
    format('Table of Contents ~t ~a~72|~*n', [i,3]),
    format('~tTable of Contents~t~72|~*n', 2),
    format("1. Documentation supplement for ~s~1f \c
      ~`.t ~d~72|~*n", ["Quintus Prolog Release ",1.5,2,2]),
    format("~t~*+~w Definition of the term \"loaded\" \c
      ~`.t ~d~72|~n", [3,1-1,2]),
    format("~t~*+~w Finding all solutions ~`.t ~d~72|~n", [3,1-2,3]),
    format("~t~*+~w Searching for a file in a library \c
      ~`.t ~d~72|~n", [3,1-3,4]),
    format("~t~*+~w New Built-in Predicates ~`.t ~d~72|~n", [3,1-
4,5]),
    format("~t~*+~w write_canonical (?Term) ~`.t ~d~72|~n", [7,1-4-
1,5]),
    format("~*+.~n~*+.~n~*+.~n", [20,20,20]),
    format("~t~*+~w File Specifications ~`.t ~d~72|~n", [3,1-7,17]),
    format("~t~*+~w multifile(+PredSpec) ~`.t ~d~72|~n", [7,1-7-
1,18]).
```

```
                    Table of Contents

1. Documentation supplement for Quintus Prolog Re-
lease 1.5 ........... 2

   1-1 Defini-
tion of the term "loaded" ............................... 2
   1-2 Finding all solu-
tions ......................................... 3
   1-3 Searching for a file in a li-
brary ............................. 4
   1-4 New Built-in Predi-
cates ..................................... 5
       1-4-
1 write_canonical (?Term) ................................ 5
                    .
                    .
                    .
   1-7 File Specifica-
tions .......................................... 17
       1-7-1 multi-
file(+PredSpec) .................................. 18
```

See Also

Section 4.6.4 [ref-iou-tou], page 89.

11.3.82 `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:

```
freeze(X, Goal) :- when(nonvar(X), Goal).
```

or

```
:- block freeze(-, ?).
freeze(_, Goal) :- Goal.
```

Backtracking

Depends on *Goal*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2.4 [ref-sem-sec], page 63.

11.3.83 `frozen/2`

Synopsis

`frozen(+`***Var***`,`***-Goal***`)`

Goal is unified with the conjunction of goals blocked on *Var*.

Arguments

Var *term*, must be var

Goal *callable*

Description

If some goal is blocked on the variable *Var*, or *Var* has attributes that can be interpreted as a goal (see Section 10.3 [lib-atts], page 337), then that goal is unified with *Goal*. If no goals are blocked, *Goal* is unified with the atom `true`. If more than one goal is blocked, a conjunction is unified with *Goal*.

Exceptions

`type_error`
Var is not a variable.

See Also

Section 4.2.4 [ref-sem-sec], page 63.

11.3.84 `functor/3` *[ISO]*

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, *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, *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 110.

11.3.85 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:

```
cycle(X) :- big_goal(X, X1), cycle(X1).
```

if `cycle/1` is to run for a long time, and if `big_goal/2` generates a lot of garbage, then rewrite the code like this:

```
cycle(X) :- big_goal(X, X1), !, garbage_collect, 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.

See Also

Section 4.10 [ref-mgc], page 125.

11.3.86 `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.

See Also

Section 4.10 [ref-mgc], page 125.

11.3.87 `generate_message/3` *[extendible]*

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` underly 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 the displayal of Prolog's messages. 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, `'SU_messages':generate_message(Message,L,[])` is called instead. If that succeeds, *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, 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

```
:- multifile user:generate_message_hook/3.
user:generate_message_hook(hello_world) -->
        ['hello world'-[],nl].
```

Note that the terminating `nl` is required.

See Also

Section 4.16 [ref-msg], page 185.

11.3.88 `generate_message_hook/3` *[hook]*

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`) 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, `'SU_messages':generate_message(Message,L,[])` is called instead. If that succeeds, *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

```
:- multifile user:generate_message_hook/3.
user:generate_message_hook(hello_world) -->
        ['hello world'-[],nl].
```

Note that the terminating `nl` is required.

See Also

Section 4.16 [ref-msg], page 185.

11.3.89 `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 94), plus:

`type_error`
: *Byte* is an invalid byte.

`existence_error`
: Trying to read beyond end of *Stream*

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.90 `get_char/[1,2]` *[ISO]*

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 94), plus:

`existence_error`
Trying to read beyond end of *Stream*

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.91 `get_code/[1,2]` *[ISO]*

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 94), plus:

`existence_error`

Trying to read beyond end of *Stream*

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.92 `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 114.

11.3.93 `goal_expansion/5` *[hook]*

Synopsis

M`:goal_expansion(+`***Goal1***`, +`***Layout1***`, +`***Module***`, -`***Goal2***`, -`***Layout2***`)`

Defines transformations on goals while clauses are being compiled or asserted, and during meta-calls.

Arguments

Goal1 *callable*
: Goal to transform.

Layout1 *term*
: Layout of goal to transform.

Module *atom*
: Source module of goal to transform.

Goal2 *callable*
: Transformed goal.

Layout2 *term*
: Layout of transformed goal.

Description

Defines transformations on goals while clauses are being consulted, compiled or asserted, *after* any processing by `user:term_expansion/6` of the terms being read in. It is called for every simple *Goal1* in the source module *Module* found while traversing the clause bodies. Typically, *Module* has imported the predicate *Goal1* from module *M*.

If it succeeds, *Goal1* is replaced by *Goal2*; otherwise, *Goal1* = *Goal2*. *Goal2* may be an arbitrarily complex goal, and ***M***`:goal_expansion/5` is recursively applied to its subgoals.

> **Please note**: the arguments of built-in meta-predicates such as `call/1`, `setof/3` and `on_exception/3` are *not* subject to such compile-time processing.

This predicate is also used to resolve any meta-calls to *Goal1* at runtime via the same mechanism. If the transformation succeeds, *Goal2* is simply called instead of *Goal1*. Otherwise, if *Goal1* is a goal of an existing predicate, that predicate is invoked. Otherwise, error recovery is attempted by `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 123) can be used.

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, *Layout2* should be `[]`.

Exceptions

Exceptions are treated as failures, except an error message is printed also.

See Also

Section 4.3.5 [ref-lod-exp], page 75.

11.3.94 `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 185.

11.3.95 `>` /2 *[ISO]*

Synopsis

+Expr1 `>` *+Expr2*

Succeeds if the result of evaluating *Expr1* is strictly *greater 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 104).

Examples

```
| ?- "g" > "g".

no

| ?- 4*2 > 15/2.

yes
```

See Also

Section 4.7 [ref-ari], page 103

11.3.96 `ground/1`

Synopsis

`ground(+Term)`

Term is currently instantiated to a ground term.

Arguments

Term *term*

Description

Tests whether *X* is completely instantiated, i.e. free of unbound variables. In this context, mutable terms are treated as nonground, so as to make `ground/1` a monotone predicate.

Examples

```
| ?- ground(9).

yes
| ?- ground(major(tom)).

yes
| ?- ground(a(1,Term,3)).

no
| ?- ground("a").

yes
| ?- ground([1,foo(Term)]).

no
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.97 `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 has handler at the top level; see Section 4.15.6 [ref-ere-int], page 184.

Exceptions

`instantiation_error`

`type_error`

ExitCode is not an integer.

See Also

`abort/0`, `break/0`, `runtime_entry/1`, Section 4.15.6 [ref-ere-int], page 184.

11.3.98 `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, 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))`.

Cuts in *P* do not make sense, but they 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 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.99 `->` /2 *[ISO]*

Synopsis

`+P -> +Q`

When occurring other than as the first argument of a disjunction operator (`;/2`), this is equivalent to:

```
P -> Q ; 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 -> Q))`.

'`->`' cuts away any choice points in the execution of *P*

Note that the operator precedence of '`->`' is greater than 1000, so it dominates commas. Thus, in:

```
f :- p, q -> r, s.
f.
```

'`->`' 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 they 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-exc], page 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.100 `include/1` *[ISO,declaration]*

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 *le_spec* or *list of le_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 118.
- 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).

Exceptions

`instantiation_error`
: ***Files*** not ground.

`context_error`
: "declaration appeared in query"

See Also

Section 4.3.4.10 [Include Declarations], page 74.

11.3.101 `(initialization)/1` *[ISO,declaration]*

Synopsis

`:- initialization +`***Goal***

Declares that *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.

`save_program/[1,2]` saves initialization goals in the saved state, so that they will run when the saved state is restored.

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 *Goal* to be specified, without creating duplicates.

Exceptions

`instantiation_error`
The argument *Goal* is not instantiated.

See Also

Section 4.3.4.11 [Initializations], page 75.

11.3.102 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 157.

11.3.103 `integer/1` *[ISO]*

Synopsis

`integer(+`***Term***`)`

Term is currently instantiated to an integer.

Arguments

Term *term*

Examples

```
| ?- integer(5).

yes
| ?- integer(5.0).

no
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.104 is/2 *[ISO]*

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

Evaluation errors.

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 103.

11.3.105 `keysort/2`

Synopsis

`keysort(+`***List1***`, -`***List2***`)`

Sorts the elements of the list *List1* into ascending standard order (see Section 4.8.8.2 [ref-lte-cte-sot], page 113) with respect to the key of the pair structure.

Arguments

List1 *list of pair*, must be a proper list of proper pairs

List2 *list of pair*

Description

The list *List1* must consist of terms of the form *Key-Value*. Multiple occurrences of any term 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 *List1* are sorted *only* according to the value of *Key*, *not* according to the value of *Value*.

`keysort` is stable in the sense that the relative position of elements with the same key is maintained.

Exceptions

`instantiation_error`
: If *List1* is not properly instantiated

`type_error`
: If *List1* is not a list of key-value 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 114.

11.3.106 `leash/1` *[development]*

Synopsis

`leash(+`***Mode***`)`

Starts leashing on the ports given by *Mode*.

Arguments

Mode *one of* `[all]` or *list of one of* `[call,exit,redo,fail,exception]`, must be ground

Either the atom `all`, or a list of the ports to be leashed..

Description

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([]).
```

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 199.

11.3.107 `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, 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]`) 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*, 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, `length/2` simply fails.

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 110, `library(lists)`.

11.3.108 < /2 *[ISO]*

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 104).

Examples

```
| ?- 23 + 2.2  <  23 - 2.2.

yes
| ?- X = 31, Y = 25, X + Y  <  X - Y

no
```

See Also

Section 4.7 [ref-ari], page 103

11.3.109 `library_directory/1` *[hook]*

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.

The predicate is undefined at startup, but behaves as if it were a multifile predicate with a single clause defining the location of the Prolog library. The initial value is the same as the value of the environment variable `SP_LIBRARY_DIR`. The predicate may succeed nondeterminately in this search for a library directory.

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
...
```

See Also

`absolute_file_name/[2,3]`, `file_search_path/2`, `load_files/[1,2]`, Section 4.5 [ref-fdi], page 80.

11.3.110 `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-tty Prolog stream, the line count of the Prolog stream is increased by one. Line count for a tty stream reflects the total line input from or output to the tty since the tty is opened to any stream.

The count is reset by `set_stream_position/2`.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94).

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 93.

11.3.111 `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 a tty stream reflects the total line input from or output to the tty since the tty is opened to any stream.

The count is reset by `set_stream_position/2`.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94).

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 93.

11.3.112 `listing/[0,1]`

Synopsis

`listing`

`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

```
| ?- 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,

```
| ?- 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:

```
| ?- listing(_:f).
```

See Also

Section 4.11 [ref-mod], page 138.

11.3.113 `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 *le_spec* or *list of le_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 `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 is `user`, `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 specfied. These will be used if the Prolog code is loaded from a source file. See Section 11.3.144 [mpg-ref-open], page 902, 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`, the predicate fails instead of raising this exception.

`permission_error`
: A specified file is protected. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

See Also

Section 4.3.2 [ref-lod-lod], page 69.

11.3.114 `load_foreign_resource/1` *[hookable]*

Synopsis

`load_foreign_resource(`***:Resource***`)`

Load the foreign resource ***Resource*** into Prolog. Relies on the hook predicates `foreign_resource/2` and `foreign/3`.

Arguments

:Resource *le_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/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/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/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 260).

Exceptions

Errors in the specification of `foreign/[2,3]` and `foreign_resource/2` will all be reported when `load_foreign_resource/1` is called.

See Also

`foreign/[2,3]`, Section 6.2.1 [Foreign Resources], page 255, Section 6.2 [Calling C from Prolog], page 254.

11.3.115 `member/2`

Synopsis

`member(`***?Element***`,` ***?List***`)`

is true if *Element* occurs in the *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 *List*!

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

On backtracking, an attempt is made to unify *Element* with successive elements of *List*. If *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
```

See Also

Section 4.8.3 [ref-lte-acl], page 110, `library(lists)`.

11.3.116 `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
```

See Also

Section 4.8.3 [ref-lte-acl], page 110, `library(lists)`.

11.3.117 `message_hook/3` *[hook]*

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, `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 185.

11.3.118 (meta_predicate)/1 *[declaration]*

Synopsis

`:- 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

‘+’

‘-’

‘?’ ignored

Exceptions

`type_error`

`domain_error`

in *MetaSpec*

`context_error`

“declaration appeared in query”

Examples

Consider a sort routine, `mysort/3`, to which the name of the comparison predicate is passed as an argument:

```
mysort(CompareProc, InputList, OutputList)
```

If *CompareProc* is module sensitive, an appropriate `meta_predicate` declaration for `mysort/3` is:

```
:- meta_predicate mysort(:, +, -).
```

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 73, Section 4.11.15 [ref-mod-mne], page 148.

11.3.119 `(mode)/1` *[declaration]*

Synopsis

`:- mode +`***`Mode`***

Currently a dummy declaration.

Arguments

:Mode *term*

See Also

Section 4.3.4.9 [Mode Declarations], page 74.

11.3.120 `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*.

If the export list is not properly specified, there will be a warning or error message at compile time.

Exceptions

`instantiation_error`

`type_error`
: *PredSpecList* is not a *list of simple_pred_spec*

`context_error`
: "declaration appeared in query", or TODO: declaration appears other than as the first term in a file being loaded.

See Also

Section 4.3.4.7 [Module Declarations], page 74, Section 4.11 [ref-mod], page 138.

11.3.121 (multifile)/1 *[ISO,declaration]*

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, 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.

Exceptions

`instantiation_error`
: *PredSpecs* not ground.

`type_error`
: Either name or arity in *PredSpec* has the wrong type

`domain_error`
: *Arity* not in the range 0..255.

`context_error`

“declaration appeared in query”, or TODO: declaration appears other than as the first term in a file being loaded.

See Also

Section 4.3.4.1 [Multifile Declarations], page 71.

11.3.122 `mutable/1`

Synopsis

`mutable(+Term)`

Succeeds if *Term* is currently instantiated to a mutable term.

Arguments

Term *term*

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109, Section 4.8.9 [ref-lte-mut], page 114.

11.3.123 `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 *Constant* is initially instantiated to an atom or number, *Codes* will be unified with the *codes* that make up its printed representation.

If *Constant* is uninstantiated and *Codes* is initially instantiated to a *codes* that corresponds to the correct syntax of a number (either integer or float), *Constant* will be bound to that number; otherwise *Constant* will be instantiated to an atom containing exactly those characters.

There are atoms for which `name(`***`Const`***`,`***`CharList`***`)` is true, but which will not be constructed if `name/2` is called with *Const* 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 inconsistency.

Exceptions

`instantiation_error`
    If *Constant* and *Codes* are both uninstantiated

`type_error`
    If *Constant* is not a constant

`domain_error`
    Codes is not a *codes*

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 110.

11.3.124 `nl/[0,1]` *[ISO]*

Synopsis

`nl`

`nl(+`***`Stream`***`)`

Terminates the current output record on the current output stream or on *Stream*.

Arguments

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 94), plus:

`existence_error`
Some operating system dependent error occurred in writing.

`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 92.

11.3.125 `nodebug/0` *[development]*

Synopsis

`nodebug`

Turns the debugger off.

See Also

Section 5.2 [Basic Debug], page 199.

11.3.126 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
```

See Also

Section 4.8.3 [ref-lte-acl], page 110, `library(lists)`.

11.3.127 nonvar/1 *[ISO]*

Synopsis

`nonvar(+Term)`

Term is currently instantiated.

Arguments

Term *term*

Examples

```
| ?- nonvar(foo(X,Y)).

true ;
no
| ?- nonvar([X,Y]).

true ;
no
| ?- nonvar(X).

no
| ?- Term = foo(X,Y), nonvar(Term).

true ;
no
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.128 (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`

See Also

Section 5.2 [Basic Debug], page 199, Section 5.3 [Plain Spypoint], page 201.

11.3.129 `nospyall/0` *[development]*

Synopsis

`nospyall`

Removes all the spypoints (including the generic ones) that have been set.

See Also

Section 5.2 [Basic Debug], page 199.

11.3.130 `=\=` /2 *[ISO]*

Synopsis

+Expr1 `=\=` *+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 104).

Examples

```
| ?- 7 =\= 14/2.

no
| ?- 7 =\= 15/2.

yes
```

See Also

Section 4.7 [ref-ari], page 103

11.3.131 =< /2 *[ISO]*

Synopsis

+Expr1 =< *+Expr2*

Succeeds if the result of evaluating *Expr1* is *less than or equal* to 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 104).

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 103

11.3.132 `>=` /2 *[ISO]*

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

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 104).

Examples

```
| ?- 42 >= 42.

yes

| ?- "b" >= "a".

yes
```

See Also

Section 4.7 [ref-ari], page 103

11.3.133 `(\+)/1` *[ISO]*

Synopsis

`\+ +P`

Fails if the goal *P* has a solution, and succeeds otherwise. Equivalent to:

```
P -> fail ; true.
```

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 they 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:

```
| ?- \+ (P,Q).
```

is this operator applied to the conjunction of *P* and *Q*, but

```
| ?- \+(P,Q).
```

would require a predicate `\+ /2` for its solution. The prefix operator can however be written as a functor of one argument; thus

```
| ?- \+((P,Q)).
```

is also correct.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.134 `\=` /2 *[ISO]*

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.

See Also

Chapter 2 [Glossary], page 7.

11.3.135 `notrace/0` *[development]*

Synopsis

`notrace`

Turns the debugger off.

See Also

Section 5.2 [Basic Debug], page 199.

11.3.136 `nozip/0` *[development]*

Synopsis

`nozip`

Turns the debugger off.

See Also

Section 5.2 [Basic Debug], page 199.

11.3.137 `number/1` *[ISO]*

Synopsis

`number(+`***`Term`***`)`

Term is currently instantiated to a number.

Arguments

Term *term*

Examples

```
| ?- number(5.2).

yes
| ?- number(5).

yes
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.138 `number_chars/2` *[ISO]*

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* (containing no variables).

If *Number* is initially instantiated to a number, *Chars* will be unified with the *chars* that make up its printed representation.

If *Number* is uninstantiated and *Chars* is initially instantiated to a *chars* that corresponds to the correct syntax of a number, *Number* will be bound to that number; otherwise `number_chars/2` will simply fail.

Exceptions

`instantiation_error`
: *Number* and *Chars* are both uninstantiated

`type_error`
: *Number* is not a number or *Chars* is not a list, or *Chars* is not 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

See Also

`number_codes/2`.

11.3.139 `number_codes/2` *[ISO]*

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 *Number* is initially instantiated to a number, *Codes* will be unified with the *codes* that make up its printed representation.

If *Number* is uninstantiated and *Codes* is initially instantiated to a *codes* that corresponds to the correct syntax of a number, *Number* will be bound to that number; otherwise `number_codes/2` will simply fail.

Exceptions

`instantiation_error`
: *Number* and *Codes* are both uninstantiated

`type_error`
: *Number* is not a number or *Codes* is not a list

`domain_error`
: *Codes* is not a *codes*

`representation_error`
: *Codes* is a list corresponding to a number that can't be represented

`syntax_error`
: *Chars* does not correspond to a syntactically valid number

Examples

```
| ?- number_codes(foo, L).

no

| ?- number_codes(431, L).

L = [52,51,49]
```

```
| ?- number_codes(X, [102,111,111]).

no

| ?- number_codes(X, [52,51,49]).

X = 431

| ?- number_codes(X, "15.0e+12").

X = 1.5e+13
```

See Also

`number_chars/2`.

11.3.140 `numbervars/3` *[meta_logic]*

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, `display/1` is used rather than `write/1`. This is because `write/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, 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, _), display(Term).
foo($VAR(22),$VAR(22),$VAR(23))
```

See Also

Section 4.8.6 [ref-lte-anv], page 111, `listing/[0,1]`, `write_term/[2,3]`.

11.3.141 `on_exception/3`

Synopsis

`on_exception(`***-Exception***`, +`***ProtectedGoal***`, +`***Handler***`)`

Specify an exception handler for *ProtectedGoal*, and call *ProtectedGoal*, as described in Section 4.15 [ref-ere], page 173.

Arguments

Exception *term*

:ProtectedGoal
callable, must be nonvar

:Handler *callable*, must be nonvar

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

Examples

Fail on exception:

```
:-meta_predicate fail_on_exception(:).
fail_on_exception(C):-
    on_exception(E,C,print_exception_then_fail(C,E)).

print_exception_then_fail(C,E):-
    format('Exception occured while calling ~q:~n', [C]),
    print_message(warning,E),
    fail.
```

Backtracking

Depends on *ProtectedGoal* and *Handler*.

See Also

Section 4.15 [ref-ere], page 173.

11.3.142 `once/1` *[ISO]*

Synopsis

`once(+P)`

Equivalent to:

```
P -> true ; fail.
```

Arguments

:P *callable*, must be nonvar

Cuts in *P* do not make sense, but they are allowed, their scope being the goal *P*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.143 `op/3` *[ISO]*

Synopsis

`op(+`***Precedence***`, +`***Type***`, +`***Name***`)`

declares *Name* to be an operator of the stated *Type* and *Precedence*.

Arguments

Precedence
: *integer*, must be nonvar and in the range 1-1200

Type
: *one of* `[xfx,xfy,yfx,fx,fy,xf,yf]`, must be nonvar

Name
: *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 `op/3`.

The *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 *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 *Name* (if any) set *Precedence* to 0.

Exceptions

`instantiation_error`
: An argument is not ground

`type_error`
: *Precedence* is not an integer or *Type* is not an integer or an operator is not an atom

`domain_error`
: *Precedence* is not in the range 1-1200, or *Type* is invalid,

`permission_error`
: Attempt to redefine the operator `','`

See Also

`current_op/3`, Section 4.1.5 [ref-syn-ops], page 43.

11.3.144 `open/[3,4]` *[ISO]*

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 *le_spec* or *integer*, must be ground

A file specification or file descriptor.

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, adds output to the end of it. If not, a new file is created.

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.

`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 96).

Overridden by the `encoding_signature/1` option, see below.

`encoding_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 the file is opened in mode read then `encoding_signature/1` defaults to `true`. When `encoding_signature(true)` is specified 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` (the default): If a byte order mark is detected it will be used to select between the encodings “UTF-8”, “UTF-16” or “UTF-32” with suitable endianess.
2. If the `encoding_signature/1` option is `true` (the default): 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, 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 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 you can specify `encoding_signature(true)`. Conversely, if you want to prevent an encoding signature from being output 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.

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, *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 obtain by using `stream_property(`***`Stream`***`, file_name(RealName))` on the resulting stream. See the example below.

Description

If *FileSpec* is a valid file specification, 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.

The following example creates two log files, both based on the 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 83).

```
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
```

Exceptions

`instantiation_error`
: *FileSpec* or *Mode* is not instantiated. *Options* argument is not ground.

`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.

`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.

`existence_error`
: The specified *FileSpec* does not exist.

`permission_error`
: Can not open *FileSpec* with specified *Mode* and *Options*.

`resource_error`
: There are too many files opened.

See Also

Section 4.6.7 [ref-iou-sfh], page 93.

11.3.145 `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.

See Also

Section 4.6.7 [ref-iou-sfh], page 93.

11.3.146 `;/2` *[ISO]*

Synopsis

`+P ; +Q`

Disjunction: Succeeds if *P* succeeds *or* *Q* succeeds.

`+P -> +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 `call((P ; Q))` or `call((P -> Q ; R))`.

The character '`|`' (vertical bar) can be used as an alternative to '`;`'.

The operator precedences of the '`;`' and '`->`' are both greater than 1000, so that they dominate commas.

Cuts in *P* do not make sense, but they 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, backtracking into *P* does not occur. A cut in *P* does not make sense. '`->`' acts like a cut except that its range is restricted to within the disjunction: it cuts away *R* and any choice points within *P*. '`->`' may be thought of as a "local cut".

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.2 [ref-sem], page 56.

11.3.147 `otherwise/0`

Synopsis

`otherwise`

Always succeeds (same as `true/0`).

Tips

Useful for laying out conditionals in a readable way.

See Also

Section 4.2 [ref-sem], page 56.

11.3.148 `peek_byte/[1,2]` *[ISO]*

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 94), plus:

`type_error`
: *Byte* is an invalid byte.

`existence_error`
: Attempt to read past end of file, or some operating system dependent error occurred in reading.

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.149 `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).
|: a

X = a
```

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94), plus:

`existence_error`

Attempt to read past end of file, or some operating system dependent error occurred in reading.

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.150 `peek_code/[1,2]` *[ISO]*

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).
|: a

X = 97
```

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94), plus:

`existence_error`

Attempt to read past end of file, or some operating system dependent error occurred in reading.

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.151 `phrase/[2,3]`

Synopsis

`phrase(+`***PhraseType***`, +`***List***`)`

`phrase(+`***PhraseType***`, +`***List***`, `***-Rest***`)`

Used in conjunction with a grammar to parse or generate strings.

Arguments

:PhraseType
: *callable*, must be nonvar

  Name of a phrase type.

List
: *list of term*

  A list of symbols — tokens or ***codes***.

Rest
: *list of term*

  A tail of *List*; what remains of *List* after *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 *PhraseType* can be rewritten as *List*.

If *List* is bound, this goal corresponds to using the grammar for parsing. If *List* is unbound, this goal corresponds to using the grammar for generation.

`phrase/[2,3]` succeeds when the portion of *List* between the start of *List* and the start of *Rest* is a phrase of type *PhraseType* (according to the current grammar rules), where *PhraseType* is either a non-terminal or, more generally, a grammar rule body.

`phrase/[2,3]` allows variables to occur as non-terminals in grammar rule bodies, just as `call/1` allows variables to occur as goals in clause bodies.

Backtracking

Depends on *PhraseType*.

Exceptions

Call errors (see Section 4.2.6 [ref-sem-exc], page 66).

See Also

Section 4.3.5 [ref-lod-exp], page 75, Section 4.14 [ref-gru], page 165.

11.3.152 `portray/1` *[hook]*

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, it is assumed to have written *T*. If it fails, 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 tail of the list.

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 printed also

See Also

Section 4.6.4 [ref-iou-tou], page 89.

11.3.153 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_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, 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, 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 94).

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 89.

11.3.154 `portray_message/2` *[hook]*

Synopsis

`:- multifile user:portray_message/2.`

`user:portray_message(+`***`Severity`***`, +`***`MessageTerm`***`)`

Called by `print_message/2` before processing the message. If this succeeds, 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 185.

11.3.155 `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 10.34.9 [Defining Primitive Constraints], page 524.

the atom `dynamic` for predicates that have been declared dynamic (see Section 4.3.4.2 [Dynamic Declarations], page 72),

the atom `multifile` for predicates that have been declared multifile (see Section 4.3.4.1 [Multifile Declarations], page 71),

the atom `volatile` for predicates that have been declared volatile (see Section 4.3.4.3 [Volatile Declarations], page 72),

one or more terms `(block `***Term***`)` for predicates that have block declarations (see Section 4.3.4.5 [Block Declarations], page 73),

the atom `exported` or terms `imported_from(`***ModuleFrom***`)` for predicates exported or imported from modules (see Section 4.11 [ref-mod], page 138),

the term `(meta_predicate `***Term***`)` for predicates that have meta-predicate declarations (see Section 4.11.16 [ref-mod-met], page 148).

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, 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:

```
| ?- [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:

```
| ?- predicate_property(m:P, imported_from(_)).
```

To backtrack through all the predicates `P` imported into module `m1` from module `m2`:

```
| ?- predicate_property(m1:P, imported_from(m2)).
```

To backtrack through all the predicates `P` exported by module `m`:

```
| ?- 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:

```
| ?- predicate_property(M:f, imported_from(m1)).
```

will return all modules `M` that import `f/0` from `m1`.

See Also

Section 4.9.1 [ref-lps-ove], page 118.

11.3.156 `print/[1,2]` *[hookable]*

Synopsis

`print(+`***Stream***`, +`***Term***`)`

`print(+`***Term***`)`

Writes *Term* on the standard output stream, without quoting atoms, calling `user:portray/1` on subterms.

Arguments

Stream *stream_object*, must be ground

A valid open Prolog stream, defaults to the current output stream.

Term *term*

Description

`print(Term)` is equivalent to:

```
write_term(Term, [portrayed(true),numbervars(true)])
```

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94).

See Also

Section 4.6.4 [ref-iou-tou], page 89, `user:portray/1`.

11.3.157 print_message/2 *[hookable]*

Synopsis

`print_message(+Severity, +MessageTerm)`

Print a *Message* of a given *Severity*. The behavior can be customized using the hooks `user:portray_message/2`, `user:generate_message_hook/3` and `user:message_hook/3`.

Arguments

Severity *atom*, must be nonvar

Unless the default system portrayal is overidden with `user:message_hook/3`, *Severity* must be one of:

Value	Prefix
`informational`	'% '
`warning`	'* '
`error`	'! '
`help`	
`query`	
`silent`	no prefix

MessageTerm
term

Description

First `print_message/2` calls `user:portray_message/2` with the same arguments. If this does not succeed, the message is processed in the following phases:

Message generation phase: the abstract message term *Message* is formatted, i.e. converted to a format-command list. First the hook predicate `user:generate_message_hook/3` is tried, then if it does not succeed, `'SU_messages':generate_message/3` is called. The latter predicate is defined in terms of definite clause grammars in `library('SU_messages')`. If that also does not succeed, then the built-in default conversion is used, which gives the following result:

```
['~q'-[Message],nl]
```

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 `user:message_hook/3` is tried, then, if it does not succeed, the built-in predicate `print_message_lines/3` is called for the `user_error` stream.

An unhandled exception message *E* calls `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 185.

11.3.158 `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, the prefix will be as described for `print_message/2`, otherwise *Severity* itself will be used as the prefix. If *Severity* is `query`, 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, the arguments are those provided by the system.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94), plus:

`instantiation_error`
`type_error`
in *Lines*.

Examples

Suppose you want to intercept messages and force them to go to a different stream:

```
user:message_hook(Severity, Message, Lines):-
    my_stream(MyStream),
    print_message_lines(MyStream, Severity, Lines).
```

See Also

Section 4.16 [ref-msg], page 185.

11.3.159 `profile_data/4` *[development]*

Synopsis

`profile_data(+`***Spec***`, `***?Selection***`, `***?Resolution***`, `***?Data***`)`

Data is profiling data collected from the predicates covered by the generalized predicate spec *Spec*.

Arguments

:Spec *pred_spec_tree*
: A predicate specification, or a list of such.

Selection *one of* `[calls,choice_points,instructions]`
: The kind of profiling data to be collected.

Resolution
: *one of* `[predicate,clause]`
: The level of resolution of the profiling data.

Data *list of pair*
: Will be instantiated to a list of key-value pairs.

Description

The *Selection* argument determines the kind of profiling data to be collected. If uninstantiated, the predicate will backtrack over its possible values, which are:

`calls` All instances of entering a clause by a procedure call are counted. This is equivalent to counting all procedure calls *that have not been determined to fail by indexing on the first argument*.

`choice_points`
: All instances of accessing a choicepoint are counted. This occurs, roughly, when there are more than one possibly matching clauses for a procedure call, when a disjunction is entered, as well as when profiled code is re-entered upon backtracking.

`instructions`
: The (number of virtual machine instructions) executed in the selected piece of code, plus 2*(number of choiceopint accesses).

The *Resolution* argument determines the level of resolution of the profiling data to be collected. If uninstantiated, the predicate will backtrack over its possible values, which are:

`predicate`
: *Data* is a list of terms ***Module:Name/Arity-Count***, where *Count* is a sum of the corresponding counts per clause.

`clause` *Data* is a list of terms ***Module:Name/Arity/ClauseNo-Count***.

Backtracking

Can be used to backtrack over all profiling data for the given *Spec*.

See Also

Section 9.2 [Execution Profiling], page 305.

11.3.160 `profile_reset/1` *[development]*

Synopsis

`profile_reset(+`***Spec***`)`

Zeroes all counters for predicates covered by the generalized predicate spec ***Spec***.

Arguments

:Spec *pred_spec_tree*

A predicate specification, or a list of such.

See Also

Section 9.2 [Execution Profiling], page 305.

11.3.161 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 119.

Arguments

FlagName *atom*, must be nonvar 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 `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

See Also

`current_prolog_flag/2`, `set_prolog_flag/2`, Section 4.9.4 [ref-lps-flg], page 119.

11.3.162 `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 123.

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, it simply fails.

See Also

`load_files/[2,3]`, Section 4.9.5 [ref-lps-lco], page 123.

11.3.163 `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), 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.), 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 89.

11.3.164 `(public)/1` *[declaration]*

Synopsis

`:- public +`***Term***

Currently a dummy declaration.

Arguments

:Term *term*

See Also

Section 4.3.4.8 [Public Declarations], page 74.

11.3.165 `put_byte/[1,2]` *[ISO]*

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 94), 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.

See Also

Section 4.6.6 [ref-iou-cou], page 92.

11.3.166 `put_char/[1,2]` *[ISO]*

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 94), 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 92.

11.3.167 `put_code/[1,2]` *[ISO]*

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 94), plus:

`instantiation_error`

`type_error`

Code is not a *code*.

`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 92.

11.3.168 `query_abbreviation/3` *[extendible]*

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,0'Y], no-"nN"]) :- !.
```

a French translator might decide that the letters '`O`' and '`o`' are reasonable abreviations for '`oui`' (yes), and therefore write

```
query_abbreviation(yes_or_no, ' (y or n) ', [yes-[-1,0'o,0'O], no-"nN"]) :- !.
```

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.169 query_class/5 *[extendible]*

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 189.

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.170 `query_class_hook/5` *[hook]*

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.

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.171 `query_hook/6` *[hook]*

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, 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, 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 interation is invoked. In other words, exceptions are treated as failures.

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.172 `query_input/3` *[extendible]*

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*

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.173 `query_input_hook/3` *[hook]*

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*

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.174 `query_map/4` *[extendible]*

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.

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.175 `query_map_hook/4` *[hook]*

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.

See Also

Section 4.16.3 [Query Processing], page 189.

11.3.176 `raise_exception/1`

Synopsis

`raise_exception(+Exception)`

Raise an exception (that might be intercepted by `on_exception/3`).

Arguments

Exception *term*, must be nonvar

Exceptions

`instantiation_error`
: *Exception* is unbound.

See Also

Section 4.15 [ref-ere], page 173.

11.3.177 `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.

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)
```

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, the action taken depends on the current value of the `syntax_errors` Prolog flag. See Section 4.9.4 [ref-lps-flg], page 119.

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 94), 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 88.

11.3.178 `read_line/[0,1]`

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 *list of code* or *one of* [`end_of_file`]

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94), plus:

`existence_error`

Trying to read beyond end of *Stream*.

See Also

`at_end_of_line/[0,1]`, Section 4.6.5 [ref-iou-cin], page 91.

11.3.179 `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 119.

`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.

`singletons(`***Names***`)`
: *Names* is bound to a list of ***Name***`=`***Var*** pairs, one for each variable appearing only once in the term and whose name does not begin with '`_`'.

`cycles(`***Boolean***`)`
: *Boolean* must be `true` or `false`. If selected, any occurrences of `@/2` in the term read in are replaced by the potentially cyclic terms they denote as described above. Otherwise (the default), *Term* is just unified with the term read in.

`layout(`***Layout***`)`
: *Layout* is bound to a *layout term* corresponding to *Term* (see Chapter 2 [Glossary], page 7).

`consume_layout(`***Boolean***`)`
: *Boolean* must be `true` or `false`. If this option is `true`, `read_term/[2,3]` will consume the *layout-text-item* that follows the terminating '`.`' (this *layout-text-item* can either be a *layout-char* or a

comment starting with a '%'). If the option is `false`, the *layout-text-item* will remain in the input stream, so that subsequent character input predicates will see it. The default of the `consume_layout` option is `false`.

Description

The characters read are subject to character-conversion.

Exceptions

Stream errors (see Section 4.6.7.2 [ref-iou-sfh-est], page 94), plus:

`syntax_error`
: A syntax error was found.

`instantiation_error`
`domain_error`
`domain_error`
: An illegal option was specified.

Examples

```
| ?- 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))
```

```
| ?- 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 88.

11.3.180 `reconsult/1`

Synopsis

`reconsult(+Files)`

same as:

`consult(Files)`

Arguments

:Files *le_spec* or *list of le_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 69.

11.3.181 `recorda/3`

Synopsis

`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*.

Arguments

Key *atomic*, must be nonvar

Term *term*

Ref *db_reference*

Description

If *Key* is a compound term, 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 new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

Exceptions

`instantiation_error`
: *Key* is not instantiated

Examples

See Also

Section 4.12.8 [ref-mdb-idb], page 159.

11.3.182 `recorded/3`

Synopsis

`recorded(`***-Key***`, `***-Term***`, +`***Ref***`)`

`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*.

Arguments

Key *atomic*

Term *term*

Ref *db_reference*

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, 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

`type_error`
: *Ref* is not a database reference

Examples

See Also

Section 4.12.8 [ref-mdb-idb], page 159.

11.3.183 `recordz/3`

Synopsis

`recordz(+`***Key***`, +`***Term***`, -`***Ref***`)`

records the *Term* in the internal database as the last item for the key *Key*; a database reference to the newly-recorded term is returned in *Ref*.

Arguments

Key *atomic*, must be nonvar

Term *term*

Ref *db_reference*

Description

If *Key* is a compound term, 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 new private variables, along with copies of any subgoals blocked on these variables (see Section 4.2.4 [ref-sem-sec], page 63).

Exceptions

`instantiation_error`
Key is not instantiated

Examples

See Also

Section 4.12.8 [ref-mdb-idb], page 159.

11.3.184 `remove_breakpoints/1` *[development]*

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 225, Section 5.7 [Breakpoint Predicates], page 237.

11.3.185 `repeat/0` *[ISO]*

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

```
Head :-
        ...
        save_state(OldState),
        repeat,
          generate(Datum),
          action(Datum),
          test(Datum),
        !,
        restore_state(OldState),
        ...
```

The purpose is to repeatedly perform some ***action*** on elements that are somehow ***generate***d, 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 run-time 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.

See Also

Section 4.2 [ref-sem], page 56.

11.3.186 `restore/1`

Synopsis

`restore(+`***FileSpec***`)`

Restores a saved-state.

Arguments

FileSpec *le_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`, the predicate fails instead of raising this exception.

`permission_error`
: A specified file is protected. If the `fileerrors` Prolog flag is `off`, the predicate fails instead of raising this exception.

See Also

`save_program/[1,2]`, Section 4.4 [ref-sls], page 77, Section 4.4.2 [ref-sls-sst], page 78.

11.3.187 `retract/1` *[ISO]*

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`, successive clauses matching *Clause* are erased. If and when no clauses match, the call to `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, *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`
: 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.

`permission_error`
: if the procedure corresponding to ***Head*** is not static.

See Also

`retractall/1`, Section 4.12.5 [ref-mdb-rcd], page 155.

11.3.188 `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

```
| ?- 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, it succeeds trivially. None of the variables in *Head* will be instantiated by this command.

Exceptions

`instantiation_error`
: if *Head* or *Module* is uninstantiated.

`type_error`
: if *Head* is not of type callable.

`permission_error`
: if the procedure corresponding to *Head* is not static.

See Also

`retract/1`, Section 4.12.5 [ref-mdb-rcd], page 155.

11.3.189 `save_files/2`

Synopsis

`save_files(+`***`SourceFiles`***`, +`***`File`***`)`

Any code loaded from *SourceFiles* is saved into *File* in PO format.

Arguments

SourceFiles
: *le_spec* or *list of le_spec*, must be ground

  A file specification or a list of file specifications; extensions optional.

File
: *le_spec*, must be ground

  A file specification, '`.po`' extension optional.

Description

Any module declarations, predicates, multifile clauses, or directives encountered in *SourceFiles* 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, *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 77.

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, or a predicate is built-in, TODO.

See Also

`load_files/[1,2]`, Section 4.4 [ref-sls], page 77, Section 4.4.3 [ref-sls-ssl], page 79.

11.3.190 `save_modules/2`

Synopsis

`save_modules(+`***Modules***`, +`***File***`)`

Saves all predicates in *Modules* in PO format to *File*.

Arguments

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.

File — *file_spec*, must be ground

A file specification, '`.po`' extension optional.

Description

The module declarations, predicates, multifile clauses and initializations belonging to *Modules* are saved in object format into the file denoted by *File*. Source file information and embedded directives (except initializations) are *not* saved.

The PO file produced can be loaded using `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

`instantiation_error`
: *Modules* or *File* is not bound.

`type_error`
: *Modules* is not a valid list of module names, or a single module name, or *File* is not a valid file specification.

`permission_error`
: *File* is not writable.

`existence_error`
: A given module is not a current module.

See Also

`load_files/[1,2]`, Section 4.4 [ref-sls], page 77, Section 4.4.3 [ref-sls-ssl], page 79.

11.3.191 `save_predicates/2`

Synopsis

`save_predicates(+`***PredSpecs***`, +`***File***`)`

Saves all predicates in *PredSpecs* in PO format to *File*.

Arguments

:PredSpecs
: *pred_spec_tree*

  A list of predicate specifications.

File
: *le_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]`.

Exceptions

`instantiation_error`
: *PredSpecs* or *File* is not bound.

`type_error`
: *PredSpecs* is not a valid list of predicate specifications, or *File* is not a valid file specification.

`permission_error`
: *File* is not writable, or a predicate is built-in, TODO.

`existence_error`
: A predicate is undefined, TODO.

See Also

`load_files/[1,2]`, Section 4.4 [ref-sls], page 77, Section 4.4.3 [ref-sls-ssl], page 79.

11.3.192 `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 *le_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`.

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.

See Also

`restore/1`, Section 4.4 [ref-sls], page 77, Section 4.4.2 [ref-sls-sst], page 78.

11.3.193 `see/1`

Synopsis

`see(+`***FileOrStream***`)` Makes file *FileOrStream* the current input stream.

Arguments

FileOrStream
: *le_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* not currently open for input, and the `fileerrors` Prolog flag is `on`.

`domain_error`
: *FileOrStream* is neither a filename nor a stream.

See Also

`seen/0`, `open/[3,4]`, `current_input/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.194 `seeing/1`

Synopsis

`seeing(`***`-FileOrStream`***`)`

Unifies *FileOrStream* with the current input stream or file.

Arguments

FileOrStream
: *le_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 93).

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), 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, `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]`.

See Also

`see/1`, `open/[3,4]`, `current_input/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.195 `seek/4`

Synopsis

`seek(+`***`Stream`***`, +`***`Offset`***`, +`***`Method`***`, `***`-NewLocation`***`)`

Seeks to an arbitrary position in *Stream*.

Arguments

Stream — *stream_object*, must be ground
A valid Prolog stream.

O set — *integer*, must be nonvar
The offset, in *items*, to seek relative to the specified *Method*. Items are bytes for binary streams, characters for text streams.

Method — *one of* `[bof,current,eof]`, must be nonvar
Where start seeking, one of the following:

- `bof` — Seek from beginning of the file stream.
- `current` — Seek from current position of the file stream.
- `eof` — Seek from end of the file stream.

NewLocation
integer
The offset from beginning of the file after seeking operation.

Description

Sets the current position of the file stream *Stream* to a new position according to *O set* and *Method*. If *Method* is:

`bof` — the new position is set to *O set items* from beginning of the file stream.

`current` — the new position is *O set* plus the current position of *Stream*.

`eof` — the new position is *O set*, plus the current size of the stream.
Avoid using this *Method*. Determining the size of the stream may be expensive or unsupported for some streams.

Positions and offsets are measured in *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 *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 94), plus:

`instantiation_error`
: *O set* or *Method* is not instantiated.

`type_error`
: *Stream* is not a stream object, or *O set* 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 supports 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 93.

11.3.196 `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

Examples

See Also

`see/1`, `close/[1,2]`, `current_input/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.197 `set_input/1` *[ISO]*

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 94).

See Also

`see/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.198 `set_module/1`

Synopsis

`set_module(+ModuleName)`

Changes the type-in module (see Section 4.11.8 [ref-mod-tyi], page 143) 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, a warning message is printed, 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 138, Section 4.11.8 [ref-mod-tyi], page 143.

11.3.199 `set_output/1` *[ISO]*

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 94).

See Also

`tell/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.200 `set_prolog_flag/2` *[ISO]*

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

`prolog_flag/3`.

11.3.201 set_stream_position/2 *[ISO]*

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 94), 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 93.

11.3.202 `setof/3` *[ISO]*

Synopsis

`setof(+`***`Template`***`, +`***`Generator`***`, `***`-Set`***`)`

Returns the non-empty set *Set* of all instances of *Template* such that *Generator* is provable.

Arguments

Template *term*

:Generator
: *callable*, must be nonvar
: A goal to be proved as if by `call/1`.

Set *list of term*

Description

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 113). If there are no instances of *Template* such that *Generator* is satisfied, then `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 *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.)

If *Generator* is of the form *A*`^`*B* then all the variables in *A* are treated as being existentially quantified.

Backtracking

`setof/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 66).

Examples

See `findall/3` for examples that illustrate the differences among `findall/3`, `setof/3`, and `bagof/3`.

See Also

`findall/3`, `bagof/3`, `^/2`, Section 4.13 [ref-all], page 162.

11.3.203 simple/1

Synopsis

`simple(+Term)`

Term is currently not instantiated to a compound term.

Arguments

Term *term*

Examples

```
| ?- simple(9).

yes
| ?- simple(_X).

yes
| ?- simple("a").

no
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.204 `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 94), plus:

`type_error`
: *Byte* is an invalid byte.

`existence_error`
: Attempt to read past end of file, or some operating system dependent error occurred in reading.

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.205 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 94), plus:

`existence_error`

Attempt to read past end of file, or some operating system dependent error occurred in reading.

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.206 `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 94), plus:

`existence_error`

Attempt to read past end of file, or some operating system dependent error occurred in reading.

See Also

Section 4.6.5 [ref-iou-cin], page 91.

11.3.207 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 94), plus:

`existence_error`
Trying to read beyond end of *Stream*.

Comments

Coding with `skip_line/[0,1]` and `at_end_of_line/[0,1]` to handle line input is more portable among different operating systems than checking end of line by the input character code.

See Also

`at_end_of_line/[0,1]`, Section 4.6.5 [ref-iou-cin], page 91.

11.3.208 `sort/2`

Synopsis

`sort(+`***List1***`, -`***List2***`)`

Sorts the elements of the list *List1* into the ascending standard order, and removes any multiple occurrences of an element. The resulting sorted list is unified with the list *List2*.

Arguments

List1 *list of term*, must be a proper list

List2 *list of term*

Exceptions

`instatiation_error`
`type_error`
List1 is not 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 114.

11.3.209 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.

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 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)
```

See Also

Section 4.9.3 [ref-lps-apf], page 118.

11.3.210 (spy)/[1,2] *[development]*

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 199, Section 5.3 [Plain Spypoint], page 201, Section 5.7 [Breakpoint Predicates], page 237.

11.3.211 `statistics/[0,2]`

Synopsis

`statistics`

Displays statistics relating to memory usage and execution time.

`statistics(`***?Keyword***`, `***?List***`)`

Obtains individual statistics.

Arguments

Keyword *atom*
: Statistics key.

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`
`domain_error`
: Invalid keyword.

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 [ref-mgc], page 125, Section 4.16 [ref-msg], page 185.

11.3.212 `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 270.

11.3.213 `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 94).

Examples

See Also

Section 4.6.7 [ref-iou-sfh], page 93.

11.3.214 `stream_position_data/3`

Synopsis

stream_position_data(*?Field*, *?Position*, *?Value*)

Value is the value of the *Field* field of stream position object *Position*.

Arguments

Field *one of* `[byte_count,line_count,character_count,line_position]`

Note that `byte_count` is meaningful only for binary streams and that the other values are meaningful only for text streams.

Position *term*

Stream position object representing the current position of *Stream*.

Value *integer*

Backtracking

Can be used to backtrack over the fields.

See Also

Section 4.6.7 [ref-iou-sfh], page 93.

11.3.215 `stream_property/2` *[ISO]*

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*.

`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`.

`encoding(CS)`
: *Stream* is a text stream with encoding *CS*, 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`.

`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`.

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 98.

Exceptions

`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 93.

11.3.216 `sub_atom/5` *[ISO]*

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`.

11.3.217 tell/1

Synopsis

`tell(+FileOrStream)`

Makes file *FileOrStream* the current output stream.

Arguments

FileOrStream
: *le_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* not currently open for output, and the `fileerrors` Prolog flag is `on`.

`domain_error`
: *FileOrStream* is neither a filename nor a stream.

See Also

`told/0`, `open/[3,4]`, `current_output/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.218 `telling/1`

Synopsis

`telling(`***-FileOrStream***`)`

Unifies *FileOrStream* with the current output stream or file.

Arguments

FileOrStream
: *le_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-iou-sfh], page 93).

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), 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, `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 `tell/1` or `open/[3,4]`.

See Also

`tell/1`, `open/[3,4]`, `current_input/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.219 `==` /2 *[ISO]*

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

See Also

Section 4.8.8 [ref-lte-cte], page 113.

11.3.220 `term_expansion/6` *[hook]*

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, 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, *Layout2* should be `[]`.

A clause of this predicate should conform to the following template, where *convert(Term1,Term2,Layout1,Layout2)* 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, Lay1, Tokens1, Term2, Lay2, [To-
ken|Tokens1]) :-
nonmember(Token, Tokens1),
convert(Term1, Lay1, Term2, Lay2), !.
```

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.

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 123) 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 75.

11.3.221 `@>` /2 *[ISO]*

Synopsis

+Term1 `@>` *+Term2*

Succeeds if ***Term1*** is *after* ***Term2*** in the standard order.

Arguments

Term1 *term*

Term2 *term*

See Also

Section 4.8.8 [ref-lte-cte], page 113.

11.3.222 `@<` /2 *[ISO]*

Synopsis

`+`*Term1* `@<` `+`*Term2*

Succeeds if ***Term1*** is *before* ***Term2*** in the standard order.

Arguments

Term1 *term*

Term2 *term*

See Also

Section 4.8.8 [ref-lte-cte], page 113.

11.3.223 `\==` /2 *[ISO]*

Synopsis

`+Term1 \== +Term2`

Succeeds if *Term1* and *Term2* are *non-identical terms*.

Arguments

Term1 *term*

Term2 *term*

See Also

Section 4.8.8 [ref-lte-cte], page 113.

11.3.224 `@=<` /2 *[ISO]*

Synopsis

+Term1 `@=<` *+Term2*

Succeeds if ***Term1*** is *not after* ***Term2*** in the standard order.

Arguments

Term1 *term*

Term2 *term*

See Also

Section 4.8.8 [ref-lte-cte], page 113.

11.3.225 `@>=` /2 *[ISO]*

Synopsis

+Term1 `@>=` *+Term2*

Succeeds if ***Term1*** is *not before* ***Term2*** in the standard order.

Arguments

Term1 *term*

Term2 *term*

See Also

Section 4.8.8 [ref-lte-cte], page 113.

11.3.226 `?= /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.

See Also

Section 4.8.1.2 [ref-lte-met-usu], page 109.

11.3.227 `throw/1` *[ISO]*

Synopsis

`throw(+`***`Exception`***`)`

same as

`raise_exception(`***`Exception`***`)`

Arguments

Exception *term*, must be nonvar

Exceptions

`instantiation_error`
: *Exception* is unbound.

See Also

Section 4.15 [ref-ere], page 173.

11.3.228 `told/0`

Synopsis

`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

Examples

See Also

`tell/1`, `close/[1,2]`, `current_output/1`, Section 4.6.7 [ref-iou-sfh], page 93.

11.3.229 `trace/0` *[development]*

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 creeping.

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).

See Also

Section 5.2 [Basic Debug], page 199.

11.3.230 `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 attempts to return any unused memory to the operating system. It is called automatically at every top-level query, except the stacks are not trimmed then.

See Also

Section 4.10 [ref-mgc], page 125.

11.3.231 `true/0` *[ISO]*

Synopsis

`true`

Always succeeds.

See Also

Section 4.2 [ref-sem], page 56.

11.3.232 `=/2` *[ISO]*

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*, it will simply fail.

See Also

Chapter 2 [Glossary], page 7, Section 4.2.7 [ref-sem-occ], page 66.

11.3.233 `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.

See Also

Chapter 2 [Glossary], page 7.

11.3.234 =.. /2 *[ISO]*

Synopsis

+Term =.. *-List*

-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*.

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 instantiated and *List* has an uninstantiated tail.

`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 110.

11.3.235 `unknown/2` *[development]*

Synopsis

`unknown(`***-OldAction***`, +`***NewAction***`)`

Unifies *OldAction* with the current action on unknown procedures, and then sets the current action to *NewAction*.

Arguments

OldAction *one of* `[error,fail,trace,warning]`

NewAction
  one of `[error,fail,trace,warning]`, must be nonvar

Description

This action determines what happens when an undefined predicate is called, and `user:unknown_predicate_handler/3` does not handle the goal, by accessing the `unknown` Prolog flag:

`trace` Causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity.

`fail` Causes calls to such predicates to fail.

`warning` Causes calls to such predicates to display a warning message and then fail.

`error` Causes calls to such predicates to raise an exception (the default). See Section 4.15 [ref-ere], page 173.

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 173.

11.3.236 unknown_predicate_handler/3 *[hook]*

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. See Section 4.9.4 [ref-lps-flg], page 119.

Exceptions

Exceptions are treated as failures, except an error message is printed.

Examples

The following clause gives the same behaviour 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 173.

11.3.237 `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 114.

11.3.238 `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 *le_spec* or *list of le_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, although this is not currently enforced. 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*.

`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, *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, *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, *File* must exist, and the module name in *File* is returned.

Special case of `load_files/2` and is defined as

```
use_module(File) :-
    load_files(File, [if(changed)]).

use_module(File, Imports) :-
    load_files(File, [if(changed),imports(Imports)]).
```

`use_module/1` is similar to `ensure_loaded/1` except that all files must be module-files.

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 139.

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 any of the predicates in *Imports* are not public predicates, an error message is printed, but the predicates are imported nonetheless. This lack of strictness is for convenience; if you forget to declare a predicate to be public, 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

`instantiation_error`
: *File* or *Imports* is not ground.

`type_error`
: In *File* or *Imports*.

`existence_error`
: A specified file does not exist. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

`permission_error`
: A specified file is protected. If the `fileerrors` flag is `off`, the predicate fails instead of raising this exception.

Examples

See Also

Section 4.3.2 [ref-lod-lod], page 69.

11.3.239 `var/1` *[ISO]*

Synopsis

`var(+Term)`

Term is currently uninstantiated.

Arguments

Term *term*

Examples

```
| ?- var(foo(X,Y)).

no
| ?- var([X,Y]).

no
| ?- var(X).

true ;

no
| ?- Term = foo(X,Y), var(Term).

no
```

See Also

Section 4.8.1.1 [ref-lte-met-typ], page 109.

11.3.240 `(volatile)/1` *[declaration]*

Synopsis

`:- volatile +`***PredSpecs***

Declares *PredSpecs* to be volatile. 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

`type_error`
`context_error`
: "declaration appeared in query"

See Also

Section 4.3.4.3 [Volatile Declarations], page 72.

11.3.241 `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 both conditions are true.

`Condition``;`***`Condition`***
: True if at least one condition is 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 66).

See Also

Section 4.2.4 [ref-sem-sec], page 63.

11.3.242 `write/[1,2]` *[ISO]*

Synopsis

`write(+Stream, +Term)`

`write(+Term)`

Writes *Term* on the standard output 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 94).

See Also

Section 4.6.4 [ref-iou-tou], page 89.

11.3.243 `write_canonical/[1,2]` *[ISO]*

Synopsis

`write_canonical(+`***`Stream`***`, +`***`Term`***`)`

`write_canonical(+`***`Term`***`)`

Writes *Term* on the standard output stream, quoting atoms, without operator 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:

```
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 94).

Examples

To contrast `write/[1,2]` and `write_canonical/[1,2]`:

```
| ?- write({'A' + '$VAR'(0) + [a]}).

{A+A+[a]}

| ?- write_canonical({'A' + '$VAR'(0) + [a]}).

{}(+(+('A','$VAR'(0)),.(a,[])))
```

See Also

Section 4.6.4 [ref-iou-tou], page 89.

11.3.244 `write_term/[2,3]` *[ISO,hookable]*

Synopsis

`write_term(+Stream, +Term, +Options)`

`write_term(+Term, +Options)`

Writes *Term* on the standard output 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*, must be ground

A list of zero or more of the following, where *Boolean* must be `true` or `false` (`false` is the default).

`quoted(Boolean)`
: If selected, 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 51 for further details.

`ignore_ops(Boolean)`
: If selected, *Term* is written in standard parenthesized notation instead of using operators. `write_canonical/1` and `display/1` select this.

`portrayed(Boolean)`
: If selected, `user:portray/1` is called for each non-variable subterm. `print/1` selects this.

`numbervars(Boolean)`
: If selected, terms of the form `'$VAR'(N)` where *N* is an integer >= 0, an atom, or a code-list, are treated specially (see `numbervars/3`). `print/1`, `write/1`, `writeq/1`, and `portray_clause/1` select this.

`cycles(Boolean)`
: If selected, the potentially cyclic term is printed in finite `@/2` notation, as discussed above.

`indented(Boolean)`
: If selected, 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 119.

`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 43.

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 94), plus:

`instantiation_error`
`type_error`
`domain_error`
: in *Options*.

Examples

How certain options affect the output of `write_term/2`:

```
| ?- write_term('a b', [quoted(true)]).
'a b'

| ?- write_term(a+b, [ignore_ops(true)]).
+(a,b)

| ?- write_term(f('$VAR'(2)),
                    [numbervars(true)].)
f(C)

| ?- write_term(f('$VAR'('C')),
                    [numbervars(true)]).

f(C)
```

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 defining a predicate like:

```
var_to_names([]) :- !.
var_to_names([=(Name,Var)|RestofPairs]) :-
     ( var(Var) ->
          Var = '$VAR'(Name)
     ; true
     ),
     var_to_names(RestofPairs).

| ?- read_term([variable_names(Names)], X),
     var_to_names(Names),
     write_term(X, [numbervars(true)]),
     nl,
     fail.
|: a(X, Y).
a(X, Y).

no
```

See Also

Section 4.6.4 [ref-iou-tou], page 89, `user:portray/1`.

11.3.245 writeq/[1,2] *[ISO]*

Synopsis

`writeq(+`***Stream***`, +`***Term***`)`

`writeq(+`***Term***`)`

Writes *Term* on the standard output 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 94).

See Also

Section 4.6.4 [ref-iou-tou], page 89.

11.3.246 `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).

See Also

Section 5.2 [Basic Debug], page 199.

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`, 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 699 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()`
Writes 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()` ***[macro]***
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()` ***[preinit]***
`SP_set_user_stream_post_hook()` ***[preinit]***
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()`
: terminates a nondeterminate Prolog query opened from C

`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()`
: 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_memalloc_hooks()` ***[preinit]***
: provide the memory management bottom layer

`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()`
: Deallocates 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_set_memalloc_hooks()` ***[preinit]***
: provide the memory management bottom layer

`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

`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 262.

12.3.2 SP_atom_length()

Synopsis

```
#include <sicstus/sicstus.h>

int
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 262.

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 `size * nemb`. The first `size * nmemb` bytes are set to zero.

Arguments

nmemb How many items to allocate.

size Size of each item.

Return Value

The pointer, if allocation was successful, otherwise `NULL`.

See Also

Section 6.4.7.1 [OS Memory Management], page 266.

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 267.

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 113, Section 6.4.6 [Unifying and Comparing Terms], page 266.

12.3.6 SP_cons_functor()

Synopsis

```
#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 263.

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 263.

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 263.

12.3.9 SP_create_stream()

Synopsis

```
#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 12.3.103 [cpg-ref-user_read], page 1151 for details.

user_write If non-NULL then this is an output stream. See Section 12.3.104 [cpg-ref-user_write], page 1153 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 12.3.102 [cpg-ref-user_flush_output], page 1149 for details.

user_seek Reserved, should be `NULL`.

user_close Closes the stream. See Section 12.3.101 [cpg-ref-user_close], page 1147 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 271.

12.3.10 SP_cut_query()

Synopsis

```
#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 267.

12.3.11 SP_define_c_predicate()

Synopsis

```
#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:

% 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)
{
  long arg1;
  SP_term_ref tmp = SP_new_term_ref();
  SP_term_ref square_term = SP_new_term_ref();
  long the_square;

  // 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);
}
```

 # terminal

```
% splfr square.pl square.c
% sicstus -f -l square
% compiling /home/matsc/tmp/square.pl...
%  loading foreign resource /home/matsc/tmp/square.so in module user
% compiled /home/matsc/tmp/square.pl in module user, 0 msec 816 bytes
SICStus 4.0.2 ...
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 254.

12.3.12 SP_deinitialize()

Synopsis

```
#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 286.

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 253.

12.3.14 SP_event()

Synopsis

```
#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,

```
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 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 your code test a flag 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.

```
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(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 6.5.3 [Calling Prolog Asynchronously], page 268, `SP_signal()`, `SP_fail()`, `SP_raise_exception()`.

12.3.15 SP_exception_term()

Synopsis

```
#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 execption term was retracted and assigned, and 0 otherwise.

See Also

Section 6.5.4 [Exception Handling in C], page 269.

12.3.16 SP_expand_file_name()

Synopsis

```
#include <sicstus/sicstus.h>

spio_t_error_code
SP_expand_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 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.

pabspath On success, *pabspath is set to the expanded path. This value is allocated with SP_malloc() and should be freed with SP_free().

Return Value

On success, *pabspath is set to the expanded path and 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 1069. See Section 4.5.2 [ref-fdi-syn], page 84, for a description of syntactic rewriting. Section 6.4.7.2 [OS File System], page 266.

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.

See Also

Section 6.5.4 [Exception Handling in C], page 269.

12.3.18 SP_fclose()

Synopsis

```
#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`
`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_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 e ort* close, which falls back to using `SP_FCLOSE_OPTION_FORCE` only if ordinary close would block.

`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: `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:

`SPIO_S_NOERR`
: The stream is still valid, some direction is still not closed.

`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 `SP_fclose()` are the same as for `SP_flush_output()`, which see. Other error codes may also be returned.

See Also

Section 12.3.19 [cpg-ref-SP_flush_output], page 1058. Section 6.6.1 [Prolog Streams], page 270.

12.3.19 SP_flush_output()

Synopsis

```
#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.

ush_options
: The following bits can be set:

  `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 270.

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 `SP_expand_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.

SP_FOPEN_OPTION_NOEXPAND
: 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.

pstream 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 270.

12.3.21 SP_foreign_stash() *[macro]*

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 run-times in the same process may need to maintain a global state that is kept separate for each SICStus run-time. Each SICStus run-time 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 run-time.

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 267.

12.3.22 SP_fprintf()

Synopsis

```
#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 270.

12.3.23 `SP_free()`

Synopsis

```
#include <sicstus/sicstus.h>

void
SP_free(void *ptr);
```

Disposees 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 266.

12.3.24 SP_get_address()

Synopsis

```
#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 264.

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 264.

12.3.26 SP_get_atom()

Synopsis

```
#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 264.

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 1068. Section 6.6.1 [Prolog Streams], page 270.

12.3.28 SP_get_code()

Synopsis

```
#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 1067. Section 6.6.1 [Prolog Streams], page 270.

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 12.3.88 [cpg-ref-SP_set_current_dir], page 1132. Section 6.4.7.2 [OS File System], page 266.

12.3.30 SP_get_float()

Synopsis

```
#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.

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 264.

12.3.31 SP_get_functor()

Synopsis

```
#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 264.

12.3.32 SP_get_integer()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_get_integer(SP_term_ref term,
               long *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 264.

12.3.33 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.

```
{
  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
}
```

See Also

Section 6.4.4 [Accessing Prolog Terms], page 264.

12.3.34 SP_get_list()

Synopsis

```
#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 264.

12.3.35 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 264.

12.3.36 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 264.

12.3.37 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 `s` the encoded string representing the character codes in the initial elements of list `term`, 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.

Please note: The array `s` is never `NUL`-terminated. Any zero character codes in the list `term` will be converted to the overlong UTF-8 sequence `0xC0 0x80`.

Arguments

term The SP_term_ref holding the code-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 264.

12.3.38 SP_get_number_codes()

Synopsis

```
#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 264.

12.3.39 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
: 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
: 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 left undefined.

pline_length
: 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 left undefined.

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 behaviour 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()`:

`SPIO_E_NOT_READ`
: `SP_GET_STREAM_COUNTS_OPTION_READ` was specified but `stream` is not an input stream.

`SPIO_E_NOT_WRITE`
: `SP_GET_STREAM_COUNTS_OPTION_WRITE` was specified but `stream` is not an output stream.

`SPIO_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 behaviour can be changed with the `options` argument, see above.

See Also

Section 6.6.1 [Prolog Streams], page 270.

12.3.40 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 1056.

```
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 1042. Section 6.6.2 [Defining a New Stream], page 271.

12.3.41 SP_get_string()

Synopsis

```
#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 264.

12.3.42 SP_initialize() *[macro]*

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_initialize(int argc,
      char **argv,
      void *reserved);
```

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)`.

reserved Should be `NULL`

Return Value

`SP_SUCCESS` if initialization was successful. If initialization was successful, further calls to `SP_initialize()` will be no-ops (and return `SP_SUCCESS`).

Description

This must be done before any interface functions are called, except those annotated as *[preinit]*. The function will allocate data areas used by Prolog, initialize command line arguments so that they can be accessed by the `argv` Prolog flag, and load the Runtime Library.

See Also

Section 6.7.4.1 [Initializing the Prolog Engine], page 286.

12.3.43 SP_is_atom()

Synopsis

```
#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 265.

12.3.44 SP_is_atomic()

Synopsis

```
#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 265.

12.3.45 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 265.

12.3.46 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 265.

12.3.47 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.

See Also

Section 6.4.5 [Testing Prolog Terms], page 265.

12.3.48 SP_is_list()

Synopsis

```
#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 it is a list and 0 otherwise.

See Also

Section 6.4.5 [Testing Prolog Terms], page 265.

12.3.49 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 265.

12.3.50 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 265.

12.3.51 SP_load()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_load(char const *filename);
```

Calls `load_files/1`.

Arguments

lename 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 286.

12.3.52 SP_malloc()

Synopsis

```
#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 266.

12.3.53 SP_mutex_lock()

Synopsis

```
#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

```
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 267.

12.3.54 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 267.

12.3.55 SP_new_term_ref()

Synopsis

```
#include <sicstus/sicstus.h>

SP_term_ref
SP_new_term_ref(void);
```

Creates a new SP_term_ref.

Return Value

The new SP_term_ref.

See Also

Section 6.4.1 [Creating and Manipulating SP_term_refs], page 262.

12.3.56 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 267.

12.3.57 SP_next_stream()

Synopsis

```
#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 269.

12.3.58 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 267.

12.3.59 SP_pred()

Synopsis

```
#include <sicstus/sicstus.h>

SP_pred_ref
SP_pred(SP_atom name_atom,
long 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 267.

12.3.60 SP_predicate()

Synopsis

```
#include <sicstus/sicstus.h>

SP_pred_ref
SP_predicate(char *name_string,
     long 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 143).

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 267.

12.3.61 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 1062.

12.3.62 SP_put_address()

Synopsis

```
#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 263.

12.3.63 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 263.

12.3.64 SP_put_byte()

Synopsis

```
#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 12.3.66 [cpg-ref-SP_put_code], page 1109. Section 6.6.1 [Prolog Streams], page 270.

12.3.65 SP_put_bytes()

Synopsis

```
#include <sicstus/sicstus.h>

spio_t_error_code
SP_put_bytes(
  SP_stream *stream,
  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 12.3.64 [cpg-ref-SP_put_byte], page 1107. Section 6.6.1 [Prolog Streams], page 270.

12.3.66 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 12.3.64 [cpg-ref-SP_put_byte], page 1107. Section 6.6.1 [Prolog Streams], page 270.

12.3.67 SP_put_codes()

Synopsis

```
#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 12.3.66 [cpg-ref-SP_put_code], page 1109. Section 6.6.1 [Prolog Streams], page 270.

12.3.68 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 12.3.67 [cpg-ref-SP_put_codes], page 1110. Section 6.6.1 [Prolog Streams], page 270.

12.3.69 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

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 263.

12.3.70 SP_put_functor()

Synopsis

```
#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 263.

12.3.71 SP_put_integer()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_put_integer(SP_term_ref term,
               long 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 263.

12.3.72 SP_put_integer_bytes()

Synopsis

```
#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 twos 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 263.

12.3.73 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 263.

12.3.74 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 263.

12.3.75 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 `term` a list of integers represented by the first `n` elements of the byte array `s`, prepended to the value of `tail`.

Arguments

term The SP_term_ref to be assigned

tail The tail of the list

n The number of bytes of `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 263.

12.3.76 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 263.

12.3.77 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`.

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 263.

12.3.78 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 263.

12.3.79 SP_put_term()

Synopsis

```
#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 263.

12.3.80 SP_put_variable()

Synopsis

```
#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 conversion fails (as far as failure can be detected), and a nonzero value otherwise.

See Also

Section 6.4.3 [Creating Prolog Terms], page 263.

12.3.81 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, and `SP_ERROR` if an error condition occurred.

Description

Use this 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 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 267.

12.3.82 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 is 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 267.

12.3.83 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.

See Also

Section 6.5.4 [Exception Handling in C], page 269.

12.3.84 SP_read_from_string()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_read_from_string(SP_term_ref t,
                    const char *string,
                    SP_term_ref values[])
```

Assigns to `tt` 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 occurence during a depth first traversal in increasing argument order. That is, the same order as used by `terms:term_variables_bag/2` (see Section 10.24 [lib-terms], page 465). 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 [Calling Prolog from C], page 267, for an example of using `SP_read_from_string()` to call an arbitrary goal.

See Also

Section 6.4.3 [Creating Prolog Terms], page 263.

12.3.85 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 266.

12.3.86 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 262.

12.3.87 SP_restore()

Synopsis

```
#include <sicstus/sicstus.h>

int
SP_restore(char const *filename);
```

Calls `restore/1`.

Arguments

lename 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 286.

12.3.88 SP_set_current_dir()

Synopsis

```
#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 1069. Section 6.4.7.2 [OS File System], page 266.

12.3.89 SP_set_memalloc_hooks() *[preinit]*

Synopsis

```
#include <sicstus/sicstus.h>

typedef int
SP_InitAllocHook(size_t alignment,
                 void *cookie);
typedef void
SP_DeinitAllocHook(void *cookie);
typedef void *
SP_AllocHook(size_t size,
     size_t *actual_sizep,
     void *cookie);
typedef int
SP_FreeHook(void *ptr,
    size_t size,
    int force,
    void *cookie);
int
SP_set_memalloc_hooks(int hint,
      SP_InitAllocHook *init_alloc_hook,
      SP_DeinitAllocHook *deinit_alloc_hook,
      SP_AllocHook *alloc_hook,
      SP_FreeHook *free_hook,
      void *cookie);
```

Defines the Prolog memory manager's bottom layer. Must be called *before* `SP_initialize()`.

Arguments

hint is reserved for future extensions. It should be zero.

init_alloc_hook
: is called initially. `alignment` is guaranteed to be a power of 2, and is used by `alloc_hook`. `earliest_start` (inclusive) and `latest_end` (exclusive) are the bounds within which address-constrained memory blocks must fit. Both are aligned according to `alignment` and non-zero. The function can do whatever initialization that this layer of memory management wants to do. It should return non-zero if it succeeds, zero if the memory manager bottom layer could not be initialized, in which case initialization of the SICStus run-time will fail.

deinit_alloc_hook
: is called by `SP_deinitialize()` when the Prolog engine shuts down. The function can do any necessary cleaning up.

alloc_hook
: must allocate and return a pointer to a piece of memory that contains at least `size` bytes aligned at a multiple of `alignment`. The actual size of the piece of

memory should be returned in `*actual_sizep`. Should return `NULL` if it cannot allocate a suitable piece of memory. Note that the memory returned need not be aligned as long as there is room for an aligned block of at least `size` bytes.

free_hook is called with a pointer to a piece of memory to be freed and its size as returned by `alloc_hook`. If `force` is non-zero, `free_hook` must accept the piece of memory; otherwise, it only accepts it if it is able to return it to the operating system. `free_hook` should return non-zero iff it accepts the piece of memory. Otherwise, the upper layer will keep using the memory as if it were not freed.

cookie can be used for any state needed by the memory hook functions. The value passed to `SP_set_memalloc_hooks()` is passed to each hook function. One possible use is to keep track of multiple SICStus run-times within the same process.

Return Value

Non-zero on success, Zero on error, e.g. if called after `SP_initialize()`.

Description

The default bottom layers look at the environment variables `PROLOGINITSIZE`, `PROLOGINCSIZE`, `PROLOGKEEPSIZE` and `PROLOGMAXSIZE`. They are useful for customizing the default memory manager. If you redefine the bottom layer, you can choose to ignore these environment variables. See Section 13.1 [too-sicstus], page 1156.

See Also

Section 6.7.4.1 [Initializing the Prolog Engine], page 286.

12.3.90 SP_set_user_stream_hook() *[preinit]*

Synopsis

```
#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 271.

user_data An arbitrary pointer that will be passed to the `hook`.

See Also

Section 6.6.3 [Hookable Standard Streams], page 275.

12.3.91 SP_set_user_stream_post_hook() *[preinit]*

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 275.

12.3.92 SP_signal()

Synopsis

```
#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 `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.

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. Under UNIX, `library(timeout)` currently uses `SIGVTALRM`.

When `func` is called it may only call other (non SICStus) C code and `SP_event()`. Note that `func` 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 6.5.3.1 [Signal Handling], page 268.

12.3.93 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, otehrwise `NULL`.

See Also

Section 6.4.7.1 [OS Memory Management], page 266.

12.3.94 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 262.

12.3.95 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

See Also

Section 6.4.5 [Testing Prolog Terms], page 265.

12.3.96 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 1067. Section 6.6.1 [Prolog Streams], page 270.

12.3.97 SP_unget_code()

Synopsis

```
#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 1068. Section 6.6.1 [Prolog Streams], page 270.

12.3.98 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 266.

12.3.99 SP_unregister_atom()

Synopsis

```
#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_term_refs], page 262.

12.3.100 SU_initialize() *[hook]*

Synopsis

```
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

0 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 280.

12.3.101 `user_close()`

Synopsis

```
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 1042. Section 6.6.2 [Defining a New Stream], page 271.

12.3.102 `user_flush_output()`

Synopsis

```
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()`.

ush_options
: The following bits can be set:

  `SPIO_DEVICE_FLUSH_OPTION_NONBLOCKING`
  : If this is set, the function should return *quickly* or with a `SPIO_E_WOULD_BLOCK` code.

    If your `user_flush_output` will never block, you can ignore this value.

    You should return `SPIO_E_NOT_SUPPORTED` if `user_flush_output` cannot support non-blocking flush.

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 1042. Section 6.6.2 [Defining a New Stream], page 271.

12.3.103 user_read()

Synopsis

```
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`
: `SPIO_DEVICE_READ_OPTION_NONBLOCKING` was set but the operation would block.

`SPIO_E_NOT_SUPPORTED`
: Some unsupported option, e.g. `SPIO_DEVICE_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 1042. Section 6.6.2 [Defining a New Stream], page 271.

12.3.104 user_write()

Synopsis

```
spio_t_error_code
user_write(
  void *user_data,
  void *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:

`SPIO_DEVICE_WRITE_OPTION_BINARY`
This is always specified if the device was created as a binary device. The buffer contains `*pbuf_size` bytes.

`SPIO_DEVICE_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.

`SPIO_DEVICE_WRITE_OPTION_NONBLOCKING`
If this is set, the 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 `SPIO_S_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 1042. Section 6.6.2 [Defining a New Stream], page 271.

13 Command Reference Pages

The reference pages for the SICStus Prolog command line tools follow, in alphabetical order.

`sicstus(1)`
: SICStus Prolog Development System

`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] [-a argument...]***

Description

The prompt '`| ?-`' indicates that the execution of is 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. Don't read any initialization file on startup. If the option is omitted and the initialization file exists, 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 84, 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.

'`--noinfo`'
: Start with the `informational` Prolog flag set to `off` initially, suppressing informational messages. The flag is set before any ***prolog- le*** 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- le*** is loaded on startup. This is done before any initialization file is loaded. Only one '`-l`' option is allowed.

'`-r` ***saved-state***'
: Restore the saved-state ***saved-state*** on startup. This is done before any ***prolog- le*** 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.

'`--help`'
: Display a help message and exit.

'-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.

'-B[***abspath***]'
: Creates a saved-state for a development system. This option is not needed for normal use. If *abspath* is given, it specifies the absolute pathname for the saved-state. **Please note**: There must not be a space before the path, lest it be interpreted as a separate option.

'-R[***abspath***]'
: Equivalent to the '-B' option, except that it builds a saved-state for a runtime system instead.

Environment

The following environment variables can be set before starting SICStus Prolog. Some of these override the default sizes of certain areas. The sizes are given in bytes, but may be followed by 'K' or 'M' meaning kilobytes or megabytes respectively.

`SP_PATH`
: This environment variable 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 253.

`GLOBALSTKSIZE`
: Governs the initial size of the global stack.

`LOCALSTKSIZE`
: Governs the initial size of the local stack.

`CHOICESTKSIZE`
: Governs the initial size of the choicepoint stack.

`TRAILSTKSIZE`
: Governs the initial size of the trail stack.

`PROLOGINITSIZE`
: Governs the size of Prolog's initial memory allocation.

`PROLOGMAXSIZE`
: Defines a limit on the amount of data space that Prolog will use.

`PROLOGINCSIZE`
: Governs the amount of space Prolog asks the operating system for in any given memory expansion.

`PROLOGKEEPSIZE`
: Governs the size of space Prolog retains after performing some computation. By default, Prolog gets memory from the operating system as the user program executes and returns all free memory back to the operating system when the user program does not need it any more. If the programmer knows that her program, once it has grown to a certain size, is likely to need as much memory for

future computations, she can advise Prolog not to return all the free memory back to the operating system by setting this variable. Only memory that is allocated above and beyond `PROLOGKEEPSIZE` is returned to the OS; the rest will be kept.

`SP_ULIMIT_DATA_SEGMENT_SIZE`
: Sets the maximum size of the data segment of the Prolog process. The value can be unlimited or a numeric value as described in the first paragraph in this section. A numeric value of zero (0) is equivalent to unlimited. Only used on Unix.

In addition the following environment variables 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_RT_DIR`
: The full path to the directory that contains the SICStus run-time. If the application has linked statically to the SICStus run-time, `SP_RT_DIR` is the same as `SP_APP_DIR`. Also available as the `runtime` file search path.

`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.

Files

‘***file***`.pl`’ Prolog source file

‘***file***`.po`’ Prolog Object file

‘`.sicstusrc`’
‘`sicstus.ini`’
: SICStus Prolog initialization file, looked up in the home directory

See Also

Section 3.1 [Start], page 21.

13.2 `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 9.6 [The Determinacy Checker], page 311.

13.3 `spld` — SICStus Prolog Application Builder

Synopsis

> % ***spld [Option | InputFile] ...***

Description

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.

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.

'`-v`'
'`--verbose`'
: Print detailed information about each step in the compilation/linking sequence.

'`-vv`' Be very verbose. Prints everything printed by '`--verbose`', and switches on verbose flags (if possible) to the compiler and linker.

'`--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.

'`-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 286.

`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`, 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 temination 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 temination 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, `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`'

Under UNIX, paths are normally hardcoded into executables in order for them to find the SICStus libraries and bootfiles. Two paths are normally hardcoded; 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), 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 some platforms the executable can figure out where it is located and so can locate any files it need, e.g. using `SP_APP_DIR` and `SP_RT_DIR`. On some UNIX platforms, however, this is not possible. In these cases, if this option is given, the executable will rely on environment variables (`SP_PATH` (see Section 13.1 [too-sicstus], page 1156) and `LD_LIBRARY_PATH`, etc.) to find all relevant files.

Under Windows, this option is always on, since Windows applications do not need to hardcode paths in order for them to find out where they're installed. See Section 6.7.2.2 [Runtime Systems on Windows Target Machines], page 279, for more information on how SICStus locates its libraries and bootfiles.

'-S'
'--static'
: Link statically with SICStus run-time and foreign resources. When '--static' is specified, a static version of the SICStus run-time 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, 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_spldconfig [...]
```

  The SICStus run-time 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. Not compatible with '--static'. Implies '--main=none'.

  Not supported on all platforms.

'--resources=*ResourceList*'
: *ResourceList* is a comma-separated list of resource names, describing which resources should be pre-linked 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 pre-linked 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 run-time.

  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 281, 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 281, 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, 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.

'--cflag=***CFlag***'
: *CFlag* is a comma-separated list of options to send to the C-compiler. Any commas in the list will be replaced by spaces. This option can occur multiple times.

'-LD'
: Do not process the rest of the command-line, but send it directly to the linker step.

'--sicstus=***Executable***'
: `spld` relies on using SICStus during some stages of its execution. The default is the SICStus-executable installed with the distribution. ***Executable*** can be used to override this, in case the user wants to use another SICStus executable.

'--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 286) before initializing SICStus.

'--more-memory'
'--no-more-memory'
: Applies platform specific tricks to ensure that the Prolog stacks can use close to 256MB (on 32bit architectures). Currently only affects x86 Linux where it circumvents the default 128MB limit. Ignored on other platforms. Not compatible with '--shared'. Somewhat experimental since the required linker

options are not well documented. This is the default on Linux. It can be forced on (off) by specifying '`--more-memory`' ('`--no-more-memory`').

'`--userhook`'
: This option enables 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, which must be called before `SP_initialize()`, such as `SP_set_memalloc_hooks()`. It is not meaningful to specify this option if '`--main=user`' or '`--main=none`' is given.

'`--with_jdk=`***DIR***'
'`--with_tcltk=`***DIR***'
'`--with_tcl=`***DIR***'
'`--with_tk=`***DIR***'
'`--with_bdb=`***DIR***'
: Specify the installation path for third-party software. 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 run-time '`.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.

'`--multi-sp-aware`'
: Compile the application with support for using more than one SICStus runtime in the same process. Not compatible with '`--static`' or pre-linked foreign resources. See Section 8.2 [Multiple SICStus Run-Times in C], page 297, for details.

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 run-time (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, 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 run-time 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'
'*.lib'
'*.dylib' These files are assumed to be input-files to the linker and will be passed on unmodified.

'*.c'
'*.cc'
'*.C'
'*.cpp'
'*.c++' These files are assumed to be C/C++ source code and will be compiled by the C/C++-compiler before being passed to the linker.

If an argument is still not recognized, it will be passed unmodified to the linker.

See Also

See Section 6.7.3 [The Application Builder], page 280.

13.4 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 255). `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 *Option*s and *InputFile*s 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.

'`-vv`' Be very verbose. Prints everything printed by '`--verbose`', and switches on verbose flags (if possible) to the compiler and linker.

'`--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.

'`--cflag=`***CFlag***'
: *CFlag* is a comma-separated list of options to send to the C-compiler. Any commas in the list will be replaced by spaces. This option can occur multiple times.

'`-LD`' Do not process the rest of the command-line, but send it directly to the compiler/linker. Syntactic sugar for '`--cflag`'.

'`--sicstus=`***Executable***'
: `spld` relies on using SICStus during some stages of its execution. The default is the SICStus-executable installed with the distribution. *Executable* can be used to override this, in case the user wants to use another SICStus executable.

'--with_jdk=***DIR***'
'--with_tcltk=***DIR***'
'--with_tcl=***DIR***'
'--with_tk=***DIR***'
'--with_bdb=***DIR***'
: Specify the installation path for third-party software. 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.

'--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 pre-linked into a Prolog system. See also the `spld` tool, Section 6.7.3 [The Application Builder], page 280.

'--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 '-R***path***' for each '-L***path***'. The '--no-rpath' option inihibits 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 '--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, the default *namebase* is the resource name followed by '_s'.

'--header=***headername***'
: Specify the name of the generated header file. The default if 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 prototypes corresponding to the `foreign/[2,3]` declarations in the prolog code.

'--multi-sp-aware'
'--exclusive-access'
'--context-hook=***name***'
'--no-context-hook'

Specifies various degrees of support for more than one SICStus in the same process. See Section 8.3 [Foreign Resources and Multiple SICStus Run-Times], page 299, 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 '-L***dir***' to the search path used by the run-time loader (using the SysV ld -R option or similar). The option '--moveable' turns off this behavior. For additional details, see the corresponding option to spld (see Section 6.7.3 [The Application Builder], page 280).

'--structs'

The Prolog source file uses library(structs). This option makes splfr understand foreign type specifications and translate them into C declarations in the generated header file. See See Section 10.22 [lib-structs], page 455.

Files

Arguments to spld not recognized as options are assumed to be input-files and are handled as follows:

'*.pro'
'*.pl' The Prolog file containing the relevant declarations. Exactly one such argument should be given.

'*.so'
'*.sl'
'*.s.o'
'*.o'
'*.obj'
'*.dll'
'*.lib'
'*.dylib' These files are assumed to be input-files to the linker and will be passed on unmodified.

'*.c'
'*.cc'
'*.C'
'*.cpp'
'*.c++' These files are assumed to be C/C++ source code and will be compiled by the C/C++-compiler before being passed to the linker.

See Also

Section 6.2.5 [The Foreign Resource Linker], page 260.

13.5 `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 NT/2000/XP, `splm` must be run by a user with Administrative rights. As of SICStus 3.10, the windowed version of SICStus (`spwin.exe`) has a menu item for license entry, making `splm` unnecessary under Windows.

Files

'`library/license.pl`'

See Also

Section 3.1 [Start], page 21.

13.6 spxref — Cross Referencer

Synopsis

% spxref [-R] [-v] [-c] [-i ifile] [-w wfile] [-x xfile] [-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, it also traces from `user:runtime_entry/1`. If individual module-files are being checked, 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 9.11 [The Cross-Referencer], page 327.

References

[Aggoun & Beldiceanu 90]
A. Aggoun and N. Beldiceanu, ***Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems***, Actes du sminaires Programmation en Logique, Trgastel, France, May 1990.

[Aggoun & Beldiceanu 93]
A. Aggoun and N. Beldiceanu, ***Extending CHIP in order to Solve Complex Scheduling and Placement Problems***, Mathl. Comput. Modelling, vol. 17, no. 7, pp. 57–73, Pergamon Press Ltd., 1993.

[Beldiceanu, Carlsson & Petit 04]
N. Beldiceanu, M. Carlsson, T. Petit, ***Deriving Filtering Algorithms from Constraint Checkers***, In. Conf. on Principles and Practice of Constraint Programming (CP2004), LNCS 3258, Springer-Verlag, 20004.

[Beldiceanu & Contejean 94]
N. Beldiceanu and E. Contejean, ***Introducing Global Constraints in CHIP***, Mathl. Comput. Modelling, vol. 20, no. 12, pp. 97–123, Pergamon Press Ltd., 1994.

[Bryant 86]
R.E. Bryant, ***Graph-Based Algorithms for Boolean Function Manipulation***, IEEE Trans. on Computers, August, 1986.

[Carlsson 90]
M. Carlsson, ***Design and Implementation of an OR-Parallel Prolog Engine***, SICS Dissertation Series 02, 1990.

[Carlsson & Beldiceanu 02]
M. Carlsson, N. Beldiceanu, ***Arc-Consistency for a Chain of Lexicographic Ordering Constraints***, SICS Technical Report T2002-18, 2002.

[Carreiro & Gelernter 89a]
N. Carreiro and D. Gelernter, ***Linda in Context***, Comm. of the ACM, 32(4) 1989.

[Carreiro & Gelernter 89b]
N. Carreiro and D. Gelernter, ***How to Write Parallel Programs: A Guide to the Perplexed***, ACM Computing Surveys, September 1989.

[Clocksin & Mellish 81]
W.F. Clocksin and C.S. Mellish, ***Programming in Prolog***, Springer-Verlag, 1981.

[Diaz & Codognet 93]
D. Diaz and P. Codognet, ***A Minimal Extension of the WAM for clp(FD)***, Proceedings of the International Conference on Logic Programming, MIT Press, 1993.

[Fruehwirth 98]
Th. Fruehwirth, ***Theory and Practice of Constraint Handling Rules***, Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3), pp 95-138, October 1998.

[Gorlick & Kesselman 87]
M.M. Gorlick and C.F. Kesselman, ***Timing Prolog Programs Without Clocks***, Proc. Symposium on Logic Programming, pp. 426–432, IEEE Computer Society, 1987.

[Hanak et al. 04]
D. Hank, T. Szeredi, P. Szeredi: ***FDBG, the CLPFD Debugger Library of SICStus Prolog***. Proc. International Workshop on Logic Programming Environments (WLPE'04), 2004.

[Heintze et al. 87]
N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, R. Yap, ***The CLP(R) Programmers Manual***, Monash University, Clayton, Victoria, Australia, Department of Computer Science, 1987.

[Ja ar & Michaylov 87]
J. Jaffar, S. Michaylov, ***Methodology and Implementation of a CLP System***, in J.L. Lassez (ed.), Logic Programming—Proceedings of the 4th International Conference—Volume 1, MIT Press, Cambridge, MA, 1987.

[Kowalski 74]
R.A. Kowalski, ***Logic for Problem Solving***, DCL Memo 75, Dept of Artificial Intelligence, University of Edinburgh, March, 1974.

[Kowalski 79]
R.A. Kowalski, ***Arti cial Intelligence: Logic for Problem Solving***. North Holland, 1979.

[Mehlhorn 00]
K. Mehlhorn and Sven Thiel, ***Faster algorithms for bound-consistency of the sortedness and the alldi erent constraint***, In Sixth Int. Conf. on Principles and Practice of Constraint Programming (CP2000), LNCS 1894, Springer-Verlag, 2000.

[O'Keefe 90]
R.A. O'Keefe, ***The Craft of Prolog***, MIT Press, 1990.

[Ousterhout 94]
John K. Ousterhout, ***Tcl and the Tk Toolkit***. Addison-Wesley, 1994.

[Regin 94] J.-C. Regin, ***A ltering algorithm for constraints of di erence in CSPs***, Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94), pp. 362–367, 1994

[Regin 96] J.-C. Regin, ***Generalized Arc Consistency for Global Cardinality Constraint***, Proc. of the Fourteenth National Conference on Artificial Intelligence (AAAI-96), 1996.

[Regin 99] J.-C. Regin, ***Arc Consistency for Global Cardinality with Costs***, Proc. Principles and Practive of Constraint Programming (CP'99), LNCS 1713, pp. 390-404, 1999.

[Schrijvers & Demoen 04]
T. Schrijvers and B. Demoen, ***The K.U.Leuven CHR System: Implementation and Application***, First Workshop on Constraint Handling Rules: Selected Contributions (T. Fruehwirth and M. Meister, eds.), pp. 1–5, 2004.

[Sellmann 02]
M. Sellmann, ***An Arc Consistency Algorithm for the Minimum Weight All Different Constraint***, Proc. Principles and Practive of Constraint Programming (CP'2002), 2002.

[Robinson 65]
J.A. Robinson, ***A Machine-Oriented Logic Based on the Resolution Principle***, Journal of the ACM 12:23-44, January 1965.

[Roussel 75]
P. Roussel, ***Prolog : Manuel de Reference et d'Utilisation***, Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[Saraswat 90]
V. Saraswat, ***Concurrent Constraint Programming Languages***. PhD thesis, Carnegie-Mellon University, 1990.

[Sterling & Shapiro 86]
L. Sterling and E. Shapiro, ***The Art of Prolog***. The MIT Press, Cambridge MA, 1986.

[Van Hentenryck 89]
P. Van Hentenryck, ***Constraint Satisfaction in Logic Programming***, Logic Programming Series, The MIT Press, 1989.

[Van Hentenryck et al. 92]
P. Van Hentenryck, V. Saraswat and Y. Deville, ***Constraint processing in cc(FD)***, unpublished manuscript, 1992.

[Van Hentenryck & Deville 91]
P. Van Hentenryck and Y. Deville, ***The Cardinality Operator: a new logical connective and its application to constraint logic programming***, In: Eighth International Conference on Logic Programming, 1991.

[Van Hentenryck et al. 95]
P. Van Hentenryck, V. Saraswat and Y. Deville, ***Design, implementation and evaluation of the constraint language cc(FD)***. In A. Podelski, ed., Constraints: Basics and Trends, LNCS 910. Springer-Verlag, 1995.

[Warren 83]
D.H.D. Warren, ***An Abstract Prolog Instruction Set***, Technical Note 309, SRI International, 1983.

Predicate Index

{

\tt\char 92

\tt\char 92+ /1, not provable

\tt\char 92= /2 (built-in)

\tt\char 92==/2 (built-in)

A

B

D

E

H

I

K

L

R

S

T

Keystroke Index

G

H

J

L

M

N

O

P

Q

R

S

T

U

W

Y

Z

Book Index

A

B

C

D

F

J

K

L

M

N

O

Q

R

S

T

U

V