Swedish Institute of Computer Science

SICStus Prolog User's Manual

October 1991

T91:11B

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

Modified for SICStus Prolog by Mats Carlsson and Johan Widén
Extended for the Industrialization of SICStus Prolog by
Johan Andersson, Stefan Andersson,
Kent Boortz, Hans Nilsson, Thomas Sjöland

Swedish Institute of Computer Science
PO Box 1263
S-164 28 KISTA, Sweden

This manual corresponds to SICStus Prolog release 2.1.

The Industrialization of SICStus Prolog was funded by
Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket
under the IT4-program.

Introduction

Prolog is a simple but powerful programming language developed at the University of Marseilles (*Prolog : Manuel de Reference et d'Utilisation* by P. Roussel, Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975), as a practical tool for *programming in logic* (*Logic for Problem Solving* by R.A. Kowalski, DCL Memo 75, Dept of Artificial Intelligence, University of Edinburgh, March, 1974). 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 *The Art of Prolog* by L. Sterling and E. Shapiro, The MIT Press, Cambridge MA, 1986. 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, a summary of the language is included. For a more general introduction to the field of Logic Programming see *Artificial Intelligence: Logic for Problem Solving* by R.A. Kowalski, North Holland, 1979. See Chapter 8 [Prolog Intro], page 107.

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 defined by D.H.D. Warren in *An Abstract Prolog Instruction Set*, Tech. Note 309, SRI International, Menlo Park, CA, 1983. 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 store more economically. However, it is recommended that the new user should gain experience with the interpreter before attempting to use the compiler. The interpreter facilitates the development and testing of Prolog programs as it provides powerful debugging facilities. It is only worthwhile compiling programs which are well-tested and are to be used extensively.

SICStus Prolog follows the mainstream Prolog tradition in terms of syntax and built-in predicates, and is largely compatible with DECsystem-10 Prolog and Quintus Prolog (*Quintus Prolog Reference Manual version 10*, Quintus Computer Systems, Inc, Mountain View, 1987). It also contains primitives for data-driven and object-oriented programming.

Certain aspects of the Prolog system are unavoidably installation dependent. Whenever there are differences, this manual describes the SICS installation which runs under Berkeley UNIX. See Chapter 10 [Installation Intro], page 139.

This manual is based on the *DECsystem-10 Prolog User's Manual* by D.L. Bowen (editor), L. Byrd, F.C.N. Pereira, L.M. Pereira, D.H.D. Warren.

Notational Conventions

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.

When introducing a built-in predicate, we shall present its usage with a *mode spec* which has the form *name(arg, ..., arg)* where each *arg* denotes how that argument should be instantiated in goals, and has one of the following forms:

:ArgName This argument should be instantiated to a term denoting a goal or a clause, or which is going to be passed to a built-in predicate that treats module prefixes specially.

+*ArgName*

This argument should be instantiated to any other term.

-*ArgName* This argument should *not* be instantiated in goals for the predicate.

?ArgName

this argument may or may not be instantiated in goals for the predicate.

Mode specs are not only used in the manual, but are part of the syntax of the language as well. When used in the source code, however, the *ArgName* part must be omitted. That is, *arg* must be either :, `+`, `-`, or ?.

In the context of certain compiler declarations, we shall need the following notation: Predicates in Prolog are distinguished by their name *and* their arity. The notation *name*/*arity* is therefore used when it is necessary to refer to a predicate unambiguously; e.g. `concatenate/3` specifies the predicate which is named "concatenate" and which takes 3 arguments. In general, a *predicate spec* takes the form *name*/*arity*. With the introduction of the module system, this spec is only defined relative to the "current" module. An absolute predicate spec must include a module prefix: *module*:*name*/*arity*.

It is possible to link user-written C code with a subset of SICStus Prolog to create stand-alone applications, called *Runtime Systems*. When introducing a built-in predicate, any limitations on its use in Runtime Systems will be mentioned.

1 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.

1.1 Getting Started

SICStus is normally started from one of the UNIX shells. It is often convenient to run in a GNU Emacs shell window, available by the Emacs command `M-x shell`. To run the Prolog interpreter, perform the shell command (see Section 10.1 [Getting Started], page 139):

```
% prolog
```

The interpreter responds with a message of identification and the prompt '`| ?- `' as soon as it is ready to accept input, thus:

```
SICStus 2.1 #0 : Mon Oct 14 09:27:36 MET 1991
Copyright (C) 1988-1991 Swedish Institute of Computer Science.
All rights reserved.
| ?-
```

When SICStus is initialized it looks for a file '`~/.sicstusrc`' and consults it, if it exists.

At this point the interpreter is expecting input of a directive, i.e. a *query* or *command*. See Section 1.4 [Directives], page 7. You cannot type in clauses immediately (see Section 1.3 [Inserting Clauses], page 7). While typing in a directive, the prompt (on following lines) becomes '`     `'. That is, the '`| ?- `' appears only for the first line of the directive, and subsequent lines are indented.

1.2 Reading in Programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the interpreter. The clauses of a predicate do not have to be immediately consecutive, but remember that their relative order may be important (see Section 8.3 [Procedural], page 115).

To input a program from a file *file*, just type the filename inside list brackets (followed by full-stop and carriage-return), thus:

```
| ?- [file].
```

This instructs the interpreter to read in (*consult*) the program. Note that it may be necessary to surround the whole file specification *file* with single quotes to make it a legal Prolog atom; e.g.

```
| ?- ['myfile.pl'].

| ?- ['/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 interpreter displays on the terminal the time spent for read-in. This indicates the completion of the directive.

Predicates that expect the name of a Prolog source file as an argument use `absolute_file_name/2` (see Section 4.1.4 [Stream Pred], page 42) to look up the file. This predicate will first search for a file with the suffix '`.pl`' added to the name given as an argument. If this fails it will look for a file with no extra suffix added. There is also support for libraries.

In general, this directive can be any list of filenames, such as:

```
| ?- [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.

Note that `consult/1` in SICStus Prolog behaves like `reconsult/1` in DEC-10 Prolog.

1.3 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 directive:

```
| ?- [user].
|
```

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

```
{user consulted, 20 msec 200 bytes}
```

1.4 Directives: Queries and Commands

Directives are either *queries* or *commands*. Both are ways of directing the system to execute some goal or goals.

In the following, suppose that list membership has been defined by:

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

(Notice the use of anonymous variables written '`_`'.)

The full syntax of a query is '`?-`' followed by a sequence of goals. E.g.

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

At interpreter top level (signified by the initial prompt of '`| ?- `'), a query may be abbreviated by omitting the '`?-`' which is already included in the prompt. Thus a query at top level looks like this:

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

Remember that Prolog terms must terminate with a full stop (`.` followed by whitespace), and that therefore Prolog will not execute anything until you have typed the full stop (and then `RET`, carriage-return) 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, then the system answers

```
yes
```

and execution of the query terminates.

If variables are included in the query, then the final value of each variable is displayed (except for anonymous variables). Thus the query

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

would be answered by

```
X = a
```

At this point the interpreter is waiting for input of either just a carriage-return (`RET`) or else a `;` followed by `RET`. Simply typing `RET` terminates the query; the interpreter responds with '`yes`'. However, typing `;` causes the system to *backtrack* (see Section 8.3 [Procedural], page 115) looking for alternative solutions. If no further solutions can be found it outputs '`no`'.

The outcome of some queries is shown below, where a number preceded by `_` is a system-generated name for a variable.

```
| ?- 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

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

```
X = f(_A)

yes
| ?-
```

Commands are like queries except that

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

Commands start with the symbol ':-'. (At top level this is simply written after the prompted '| ?- ' which is then effectively overridden.) Any required output must be programmed explicitly; e.g. the command:

```
:- member(3, [1,2,3]), write(ok).
```

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

{WARNING: goal failed: :- *Goal*}

as a warning.

The principal use for commands (as opposed to queries) is to allow files to contain directives which 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 which consults a whole list of other files, e.g.

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

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

```
| ?- [myprog].
```

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

1.5 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 terminal as soon as it is read, along with its position in the input strean 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).

** in lines 11-12 **
** , or ) expected in arguments **
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 which were successfully read-in, e.g.

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

NOTE: The built in predicates `read/1-2` normaly raise an exception on syntax errors (see Section 4.5 [Exception], page 53). The behavior is controlled by the flag `syntax_errors` (see `prolog_flag/3`).

1.6 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 3.2.7 [Declarations], page 30) 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. The state of the catching facility can be:

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

`error` which causes calls to such predicates to raise an exception (the default state). See Section 4.5 [Exception], page 53.

`fail` 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*. It fails if the arguments are not appropriate. The built-in predicate `debugging/0` prints the value of this state along with its other information. This state is also controlled by the flag `unknown` (see `prolog_flag/3`).

1.7 Program Execution And Interruption

Execution of a program is started by giving the interpreter a directive which contains a call to one of the program's predicates.

Only when execution of one directive is complete does the interpreter become ready for another directive. However, one may interrupt the normal execution of a directive 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 interpreter 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 possible commands are:

`a` abort the current computation.

`b` invoke the Prolog interpreter recursively.

`c` continue the execution.

`d` enable debugging. See Chapter 2 [Debug Intro], page 15.

`e` exit from SICStus, closing all files.

`h`
`?` list available commands.

`t` enable trace. See Section 2.3 [Trace], page 18.

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

1.8 Exiting From The Interpreter

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

1.9 Nested Executions—Break and Abort

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 directives to solve goals etc. This is achieved by issuing the directive (see Section 1.7 [Execution], page 11):

```
| ?- break.
```

This causes a recursive call to the Prolog interpreter, indicated by the message:

```
{ Break level 1 }
```

You can now type queries just as if the interpreter 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 which 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 }
```

Alternatively, the suspended execution can be aborted by calling the built-in predicate `abort/0`.

A suspended execution can be aborted by issuing the directive:

```
| ?- abort.
```

within a break. In this case no ^D is needed to close the break; *all* break levels are discarded and the system returns right back to top-level. IO streams remain open, but the debugger is switched off. `abort/0` may also be called from within a program.

1.10 Saving and Restoring Program States

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

The state of a program may be saved on disk for future execution. To save a program into a file *File*, perform the directive:

```
| ?- save(File).
```

This predicate may be called at any time, for example it may be useful to call it in a break in order to save an intermediate execution state. The file *File* becomes an executable file. See Section 10.1 [Getting Started], page 139.

Once a program has been saved into a file *File*, the following directive will restore the interpreter to the saved state:

```
| ?- restore(File).
```

After execution of this directive, which may be given in the same session or at some future date, the interpreter will be in *exactly* the same state as existed immediately prior to the call to `save/1`. Thus if you saved a program as follows:

```
| ?- save(myprog), write('myprog restored').
```

then on restoring you will get the message '`myprog restored`' printed out.

A partial program state, containing only the user defined predicates may also be saved with the directive:

```
| ?- save_program(File).
```

The file *File* becomes an executable file. See Section 10.1 [Getting Started], page 139. After restoring a partial program state, the interpreter will reinitialize itself.

Note that when a new version of the Prolog system is installed, all program files saved with the old version become obsolete.

The save facility is not available in Runtime Systems and the predicates defined in this chapter are undefined, see Chapter 7 [Runtime Systems Intro], page 95.

2 Debugging

This chapter describes the debugging facilities that are available in the Prolog interpreter. 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 *spy-points*. Spy-points allow you to nominate interesting predicates at which the program is to pause so that you can interact.
- 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 *Programming in Prolog* by W.F. Clocksin and C.S. Mellish (Springer-Verlag, 1981) which is recommended as an introduction.

The debugger is not available in Runtime Systems and the predicates defined in this chapter are undefined, see Chapter 7 [Runtime Systems Intro], page 95.

2.1 The Procedure Box Control Flow Model

During debugging the interpreter 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 interpreter 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 the interpreter, 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
         *--------------------------------------*
```

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 labelled arrows indicate the control flow in and out of this box. There are four 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 any 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 any 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 which called this predicate and keep moving backwards up the code looking for choice points.

In terms of this model, the information we get about the procedure box is only the control flow through these four ports. This means that at this level we are not concerned with which clause matches, and how any subgoals are satisfied, but rather we only wish to know the initial goal and the final outcome. However, it can be seen that whenever we are trying to satisfy subgoals, what we are actually doing is passing through the ports of *their* respective boxes. If we were to follow this, then we would have complete information about the control flow inside the procedure box.

Note that the box we have drawn round the predicate should really be seen as an *invocation box*. That is, there will be a different box for each different invocation of the predicate. Obviously, with something like a recursive predicate, there will be many different *Calls* and *Exits* in the control flow, but these will be for different invocations. Since this might get confusing each invocation box is given a unique integer identifier.

There is actually a fifth kind of port, *Exception*. Control passes out of this port when an exception has been raised by either a call to `raise_exception/1` or an error in a built-in predicate. See Section 4.5 [Exception], page 53. The *Exception* port is used for exiting the box until the exception is caugth by an `on_exception/3`.

2.2 Basic Debugging Predicates

The interpreter provides a range of built-in predicates for control of the debugging facilities. The most basic predicates are as follows:

`debug`
: Switches the debugger on. (It is initially off.) In order for the full range of control flow information to be available it is necessary to have this 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 below), but information prior to this will just be unavailable.

`nodebug`
: Switches the debugger off. If there are any spy-points set then they will be kept but disabled.

`debugging`
: Prints onto the terminal information about the current debugging state. This will show:
  1. Whether undefined predicates are being trapped.
  2. Whether the debugger is swithed on.
  3. What spy-points have been set (see below).
  4. What mode of leashing is in force (see below).
  5. What the interpreter maxdepth is (see below).

2.3 Tracing

The following built-in predicate may be used to commence an exhaustive trace of a program.

`trace` Switches the debugger on, if it is not on already, and ensures that the next time control enters a procedure box, a message will be produced and you will be asked to interact. The effect of trace 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 2.6 [Debug Options], page 21. In particular, you can just type `RET` (carriage-return) 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. You will notice that the interpreter stops at all ports. However, if this is not what you want, the following built-in predicate gives full control over the ports at which you are prompted:

`leash(+`*Mode*`)`

Leashing Mode is set to *Mode*. Leashing Mode determines the ports of procedure 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 the ports of spy-points are always leashed (and cannot be unleashed). *Mode* can be a a subset of the following, specified as a list:

`call` Prompt on Call.

`exit` Prompt on Exit.

`redo` Prompt on Redo.

`fail` Prompt on Fail.

`exception`

Prompt on Exception.

The initial value of *Leashing Mode* is `[call,exit,redo,fail,exception]` (full leashing).

`notrace` Equivalent to `nodebug`.

2.4 Spy-points

For programs of any size, it is clearly impractical to creep through the entire program. *Spy-points* make it possible to stop the program whenever it gets to a particular predicate which is of interest. Once there, one can set further spy-points in order to catch the control flow a bit further on, or one can start creeping.

Setting a spy-point on a predicate indicates that you wish to see all control flow through the various ports of its invocation boxes. When control passes through any port of a procedure box with a spy-point set on it, a message is output and the user is asked to interact. Note that the current mode of leashing does not affect spy-points: user interaction is requested on *every* port.

Spy-points are set and removed by the following built-in predicates which are also standard operators:

`spy` *:Spec* Sets spy-points on all the predicates given by *Spec*, which can have the forms:

Name all predicates of that name no matter what arity,

Name/Arity
the predicate of that name and arity,

Name/Low-High
the predicates of that name with arity in the range *Low-High*,

Module:Spec
specifying a particular module instead of the default type-in module, see Section 5.2 [Module Spec], page 83), and

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

You cannot place a spy-point on an undefined predicate. If you set some spy-points when the debugger is switched off then it will be automatically switched on. Examples:

```
| ?- spy [user:p, m:q/2-3].
| ?- spy m:[p/1, q/1].
```

`nospy` *:Spec*
This is similar to spy *Spec* except that all the predicates given by *Spec* will have previously set spy-points removed from them.

`nospyall` This removes all the spy-points that have been set.

The options available when you arrive at a spy-point are described later. See Section 2.6 [Debug Options], page 21.

2.5 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 terminal regardless of where the current output stream is directed. (This allows you to trace programs while they are performing file IO.) The basic format is as follows:

S `23  6  Call:` *T* `foo(hello,there,_123) ?`

S is a spy-point indicator. It is printed as '`+`', indicating that there is a spy-point on `foo/3`, or ' ', denoting no spy-point.

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 first number is the unique invocation identifier. It is nondecreasing regardless of whether or not you are actually seeing the invocations (provided that the debugger is swithed 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.

The next word specifies the particular port (Call, Exit, Redo or Fail).

The goal is then printed so that you can inspect its current instantiation state. This is done using `print/1` (see Section 4.1.2 [Term IO], page 36) so that all goals output by the tracing mechanism can be pretty printed if the user desires.

The final '`?`' is the prompt indicating that you should type in one of the option codes allowed (see Section 2.6 [Debug Options], page 21). If this particular port is unleashed then you will obviously not get this prompt since you have specified that you do not wish to interact at this point.

Note that not all procedure calls are traced; there are a few basic predicates which have been made invisible since it is more convenient not to trace them. These include debugging directives and basic control structures, including `trace/0`, `debug/0`, `notrace/0`, `nodebug/0`, `spy/1`, `nospy/1`, `nospyall/0`, `leash/1`, `debugging`, `true/0`, `!/0`, `','/2`, `'->'/2`, `;/2`, `'\+'/1`, and `if/3`. This means that you will never see messages concerning these predicates during debugging.

There are two 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 Chapter 6 [Blocking], page 89). 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.

2.6 Options available during Debugging

This section describes the particular options that are available when the system prompts you after printing out a debugging message. All the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the terminal with any blanks being completely ignored up to the next terminator (carriage-return, line-feed, or escape). Some options only actually require the terminator; e.g. the creep option, as we have already seen, only requires `RET`.

The only option which you really have to remember is '`h`' (followed by `RET`). This provides help in the form of the following list of available options.

```
RET   creep             c       creep
 l    leap              s       skip
 r    retry             r <i>   retry i
 f    fail              f <i>   fail i
 d    display           p       print
 w    write
 g    ancestors         g <n>   ancestors n
 &    blocked goals     & <n>   nth blocked goal
 n    nodebug           =       debugging
 +    spy this          -       nospy this
 a    abort             b       break
 @    command           u       unify
 e    pending exception
 <    reset printdepth  < <n>   set printdepth
 ^    reset subterm     ^ <n>   set subterm
 ?    help              h       help
```

`c`
`RET`
: *creep* causes the interpreter to single-step to the very next port and print a message. Then if the port is leashed (see Section 2.3 [Trace], page 18), 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 terminal.

`l`
: *leap* causes the interpreter to resume running your program, only stopping when a spy-point 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 spy-points on an evenly spread set of pertinent predicates, and then follow the control flow through these by leaping from one to the other.

`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 then no message at all will appear until control returns. This includes calls to predicates with spy-points set; they will be masked out during

the skip. There is a way of overriding this : the `t` option after a ^C interrupt will disable the masking. Normally, however, this masking is just what is required!

`r` *retry* can be used at any of the four ports (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 after the retry command, then this is taken as specifying an invocation number and the system tries to get you to the Call port, not of the current box, but 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 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, then 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.

`d` *display goal* displays the current goal using `display/1`. See Write (below).

`p` *print goal* re-prints the current goal using `print/1`. Nested structures will be printed to the specified *printdepth* (below).

`w` *write goal* writes the current goal on the terminal using `write/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. It uses the `ancestors/1` built-in predicate (see Section 4.6 [State Info], page 54). You can always be sure of jumping to any goal in the ancestor list (by using retry etc). If you supply an integer *n*, then only that number of ancestors will be printed. That is to say, the last *n* ancestors will be printed counting back from the current goal. The list is printed using `print/1` and each entry is preceded by the invocation number followed by the depth number (as would be given in a trace message).

`&` *print blocked goals* prints a list of the goals which are currently blocked in the current debugging session together with the variable that each such goal is blocked on (see Chapter 6 [Blocking], page 89). The goals are enumerated from 1 and up. If you supply an integer *n*, then only that goal will be printed. The goals are printed using `print/1`. and 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` options because they always return to the debugger.

`=` *debugging* outputs information concerning the status of the debugging package. See Section 4.12 [Debug Pred], page 68.

`+` *spy this*. Set a spy-point on the current goal.

`-` *nospy this*. Remove spy-point from the current goal.

`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 of the interpreter. (This is the same as the built-in predicate `abort/0`.)

`b` *break* calls the built-in predicate `break/0`, thus putting you at interpreter 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 spy-points 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 your terminal, and a command is then read from the terminal and executed as if you were at top level.

`u` *unify* is available at the Call port and gives you the option of providing a solution to the goal from the terminal 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 your terminal, and the solution is then read from the terminal and unified with the goal.

`e` *pending exception* is available at the Exception port. It displays the raised exception that caused the transfer to the Exception port.

`<` While in the debugger, a *printdepth* is in effect for limiting the subterm nesting level when printing the current goal using `print/1`. When displaying or writing the current goal, all nesting levels are shown. The limit is initially 10. This command, without arguments, resets the limit to 10. With an argument of *n*, the limit is set to *n*.

`^` While at a particular port, a current *subterm* of the current goal is maintained. It is the current subterm which 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 which is 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.

`?`
`h` *help* displays the table of options given above.

2.7 Consulting during Debugging

It is possible, and sometimes useful, to consult a file whilst in the middle of program execution. Predicates, which 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 virtual copy of its definition for backtracking purposes.

3 Loading Programs

Programs can be loaded in three different ways: consulted or compiled from source file, or loaded from object files. The latter is the fastest way of loading programs, but of course requires that the programs have been compiled to object files first. Object files may be handy when developing large applications consisting of many source files, but are not strictly necessary since it is possible to save and restore entire execution states (see Section 4.15 [Misc Pred], page 75).

Consulted, or interpreted, predicates are equivalent to, but slower than, compiled ones. Although they use different representations, the two types of predicates can call each other freely.

The SICStus Prolog compiler produces compact and efficient code, running about 8 times faster than consulted code, and requiring much less runtime storage. Compiled Prolog programs are comparable in efficiency with LISP programs for the same task. However, against this, compilation itself takes about twice as long as consulting and some debugging aids, such as tracing, are not applicable to compiled code. Spy-points can be placed on compiled procedures, however.

The compiler operates in three different modes, controlled by the “Compilation mode” flag (see `prolog_flag/3`). The possible states of the flag are:

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

`fastcode`
: Compilation produces native machine instructions. Currently only available for Sun-3, Sun-4, and 68K-based NeXT computers. Fastcode runs about 3 times faster than compactcode.

  In Runtime Systems, predicates compiled to fastcode cannot be called from a directive in the same file as the predicate is defined in.

`profiledcode`
: Compilation produces byte-coded abstract instructions instrumented to produce execution profiling data. See Section 4.13 [Profiling], page 69.

  Profiling is not available in Runtime Systems.

The compilation mode can be changed by issuing the directive:

```
| ?- prolog_flag(compiling, OldValue, NewValue).
```

A Prolog program consists of a sequence of *sentences* (see Section 8.9.2 [Sentence], page 125). Commands and queries encountered among the sentences are executed immediately as they are encountered, unless they can be interpreted as *declarations* (see Section 3.2.7 [Declarations], page 30), which affect the treatment of forthcoming clauses. Clauses are loaded as they are encountered.

A Prolog program may also contain a list of sentences (including the empty list). This is treated as equivalent to those sentences occurring in place of the list. This feature makes it possible to have `term_expansion/2` (see Section 4.14 [Definite], page 71) "return" a list of sentences, instead of a single sentence.

3.1 Predicates which Load Code

To consult a program, issue the directive:

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

where *Files* is either the name of a file (including the file '`user`') or a list of filenames instructs the interpreter to read-in the program which is in the files. For example:

```
| ?- consult([dbase,'extras.pl',user]).
```

When a directive is read it is immediately executed. Any predicate defined in the files erases any clauses for that predicate already present in the interpreter. If the old clauses were loaded from a different file than the present one, the user will be queried first whether (s)he really wants the new definition. However, for existing predicates which have been declared as `multifile` (see below) new clauses will be added to the predicate, rather than replacing the old clauses. If clauses for some predicate appear in more than one file, the later set will effectively overwrite the earlier set. The division of the program into separate files does not imply any module structure—any predicate can call any other (see Chapter 5 [Module Intro], page 83).

`consult/1`, used in conjunction with `save/1` and `restore/1`, makes it possible to amend a program without having to restart from scratch and consult all the files which make up the program. The consulted file is normally a temporary "patch" file containing only the amended predicate(s). Note that it is possible to call `consult(user)` and then enter a patch directly on the terminal (ending with ^D). This is only recommended for small, tentative patches.

```
| ?- [File|Files].
```

This is a shorthand way of consulting a list of files. (The case where there is just one filename in the list was described earlier (see Section 1.2 [Reading In], page 6).

To compile a program in-core, use the built-in predicate:

```
| ?- compile(Files).
```

where *Files* is specified just as for `consult/1`.

The effect of `compile/1` is very much like that of `consult/1`, except all new procedures will be stored in compiled rather than consulted form. However, predicates declared as dynamic (see below) will be stored in consulted form, even though `compile/1` is used.

To compile a program into an object file, use the built-in predicate:

```
| ?- fcompile(Files).
```

where *Files* is specified just as for `consult/1`. For each filename in the list, the compiler will append the suffix '`.pl`' to it and try to locate a source file with that name and compile it to an object file. The object filename if formed by appending the suffix '`.ql`' to the specified name. The internal state of SICStus Prolog is not changed as result of the compilation.

To load a program from a set of object files, use the built-in predicate:

```
| ?- load(Files).
```

where *Files* is either a single object filename (specified without the trailing '`.ql`') or a list of filenames. For each filename in the list, this predicate will first search for a file with the suffix '`.ql`' added to the name given as an argument. If this fails it will look for a file with no extra suffix added. This directive has the same effect as if the source files had been compiled using `compile/1` directly (but see Section 3.3 [Pitfalls], page 31).

Finally, to ensure that some files has been compiled or loaded, use the built-in predicate:

```
| ?- ensure_loaded(Files).
```

where *Files* is either a single filename or a list of filenames, similar to the arguments accepted by the above predicates. The predicate takes the following action for each *File* in the list of filenames:

1. If the *File* is `user`, `compile(user)` is performed;
2. If *File* cannot be found, not even with a '`.pl`' or '`.ql`' extension, an error is signalled;
3. If an object file is found which has not yet been loaded or which has been modified since it was last loaded, the file is loaded;
4. If a source file is found which has not yet been loaded or which has been modified since it was last loaded, the file is compiled;

5. If both a source file and an object file are found, item 3 or 4 applies depending on which file was modified most recently; in case 4 `ensure_loaded/1` does *not* cause object files to become recompiled.
6. Otherwise, no action is taken.

3.2 Declarations

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 which they concern. A declaration is written just as a command, beginning with '`:- `'. A declaration is effective from its occurrence through the end of file.

Although declarations that affects 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.

3.2.1 Multifile Declarations

A declaration

:- multifile *PredSpec*, ..., *PredSpec*.

where each *PredSpec* is a predicate spec, causes the specified predicates to be `multifile`. This means that if more clauses are subsequently loaded from other files for the same predicate, then the new clauses will not replace the old ones, but will be added at the end instead. The old clauses are erased only if the predicate is reloaded from its "home file" (the one containing the multifile declaration), if it is reloaded from a different file declaring the predicate as multifile (in which case the user is queried first), or if it is explicitly abolished.

Furthermore the compilation mode of the "home file" determines the compilation mode of any subsequently loaded clauses. For example, if the "home file" declares the predicate as multifile and dynamic, any subsequent clauses will be stored in consulted form even if loaded by `compile/1`. If the "home file" was compiled to native code, any subsequent clauses will also be compiled to native code even if the compilation mode for the subsequent file was `compactcode`.

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

The `multifile` declaration cannot be used in Runtime Systems.

3.2.2 Dynamic Declarations

A declaration

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

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 consulted 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.

3.2.3 Block Declarations

A declaration

:- `block` *BlockSpec*, ..., *BlockSpec*.

where each *BlockSpec* is a mode spec, introduces an exception to the rule that goals be run strictly from left to right within a clause. See Section 6.1 [Block Decl], page 89 for details.

3.2.4 Meta-Predicate Declarations

A declaration

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

where each *MetaPredSpec* is a mode spec, informs the compiler that certain arguments of the declared predicates are used for passing goals. To ensure the correct semantics in the context of multiple modules, clauses or directives containing goals for the declared predicates may need to have those arguments module-name-expanded. See Section 5.5 [Meta Exp], page 85 for details.

3.2.5 Module Declarations

A declaration

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

where *PublicPredicateList* 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 5.3 [Def Modules], page 84 for details.

3.2.6 Public Declarations

A declaration

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

where each *PredSpec* is a predicate spec, has no effect whatsoever, but is accepted for compatibility reasons. In some Prologs, this declaration is necessary for making compiled predicates visible for the interpreter. In SICStus Prolog, predicate visibility is handled by the module system. See Chapter 5 [Module Intro], page 83.

3.2.7 Mode Declarations

A declaration

```
:- mode ModeSpec, ..., ModeSpec.
```

where each *ModeSpec* is a mode spec, has no effect whatsoever, but is accepted for compatibility reasons. In some Prologs, this declaration helps reduce the size of the compiled code for a predicate, and may speed up its execution. Unfortunately, writing mode declarations can be error-prone, and since errors in mode declaration do not show up while running the predicates interpretively, new bugs may show up when predicates are compiled. However, mode declarations may be used as a commenting device, as they express the programmer's intention of data flow in predicates.

3.3 Pitfalls of File-To-File Compilation

When loading clauses belonging to a multifile predicate from an object file different from the predicate's "home file", the compilation mode used when the new clauses were compiled must match that of the current clauses. Otherwise, the new clauses are ignored and a warning message is issued.

When compiling to an object file, remember that directives occurring in the source file are executed at *run time*, not at compile time. For instance, it does not work to include directives that assert clauses of `term_expansion/2` (q.v.) and rely on the new transformations to be effective for subsequent clauses of the same file or subsequent files of the same compilation. For a definition of `term_expansion/2` to take effect, it should be loaded as a separate file before being used in the compilation of another file.

Operator declarations (q.v.) are an exception to the above rule. If the compiler encounters a command

```
:- op(P, T, N).
```

that command will be executed at compile time as well as at run time.

3.4 Indexing

The clauses of any predicate are *indexed* according to the principal functor of the first argument in the head of the clause. This means that the subset of clauses which match a given goal, as far as the first step of unification is concerned, is found very quickly, in practically constant time (i.e. in a time independent of the number of clauses of the predicate). This can be very important where there is a large number of clauses for a predicate. Indexing also improves the Prolog system's ability to detect determinacy—important for conserving working storage.

3.5 Tail Recursion Optimization

The compiler incorporates *tail recursion optimization* to improve the speed and space efficiency of determinate predicates.

When execution reaches the last goal in a clause belonging to some predicate, and provided there are no remaining backtrack points in the execution so far of that predicate, all of the predicate's local working storage is reclaimed *before* the final call, and any structures it has created become eligible for garbage collection. This means that programs can now recurse to arbitrary depths without necessarily exceeding core limits. For example:

```
cycle(State) :- transform(State, State1), cycle(State1).
```

where `transform/2` is a determinate predicate, can continue executing indefinitely, provided each individual structure, *State*, is not too large. The predicate cycle is equivalent to an iterative loop in a conventional language.

To take advantage of tail recursion optimization one must ensure that the Prolog system can recognize that the predicate is determinate at the point where the recursive call takes place. That is, the system must be able to detect that there are no other solutions to the current goal to be found by subsequent backtracking. In general this involves reliance on the Prolog compiler's indexing and/or use of cut, see Section 8.5 [Cut], page 119.

4 Built-In Predicates

It is not possible to redefine built-in predicates. An attempt to do so will give an error message. See Chapter 11 [Pred Summary], page 141.

SICStus Prolog provides a wide range of built-in predicates to perform the following tasks:

- Input / Output
  - Reading-in Programs
  - Input and Output of Terms
  - Character IO
  - Stream IO
  - Dec-10 Prolog File IO
- Arithmetic
- Comparison of Terms
- Control
- Error and Exception Handling
- Information about the State of the Program
- Meta-Logic
- Modification of the Program
- Internal Database
- All Solutions
- Foreign Language Functions Interface
- Debugging
- Definite Clause Grammars
- Miscellaneous

The following descriptions of the built-in predicates are grouped according to the above categorization of their tasks.

4.1 Input / Output

There are two sets of file manipulation predicates in SICStus Prolog. One set is inherited from DEC-10 Prolog. These predicates always refer to a file by name. The other set of predicates is inherited from Quintus Prolog and refer to files as streams. Streams correpond to the file pointers used at the operating system level.

A stream can be opened and connected to a filename or UNIX file descriptor for input or output by calling the predicate `open/3`. `open/3` will return a reference to a stream. The stream may then be passed as an argument to various IO predicates. The predicate `close/1` is used for closing a stream. The predicate `current_stream/3` is used for retrieving information about a stream, and for finding the currently existing streams.

The possible formats of a stream are:

`'$stream'(`*X*`,`*Y*`)`
: A stream connected to some file. *X* and *Y* are integers.

`user_input`
: The standard input stream, i.e. the terminal, usually.

`user_output`
: The standard output stream, i.e. the terminal, usually.

`user_error`
: The standard error stream, i.e. the terminal, usually.

The DEC-10 Prolog IO predicates manipulate streams implicitly, by maintaining the notion of a *current input stream* and a *current output stream*. The current input and output streams are set to the `user_input` and `user_output` initially and for every new break (see Section 1.9 [Nested], page 12). The predicates `see/1` and `tell/1` can be used for setting the current input and output streams (respectively) to newly opened streams for particular files. The predicates `seen/0` and `told/0` close the current input and output streams (respectively), and reset them to the standard input and output streams. The predicates `seeing/1` and `telling/1` are used for retrieving the filename associated with the current input and output streams (respectively).

The possible formats of a filename are:

`user`
: This "filename" stands for the standard input or output stream, depending on context. Terminal output is only guaranteed to be displayed after a newline is written or `ttyflush/0` is called.

`library(File)`
: where *File* is an atom, denotes a file *File* (with an optional '`.pl`' suffix when consulting or compiling or an optional '`.ql`' suffix in `load/1`) sought in the directory path(s) specified by the user defined predicate `library_directory/1`.

File
: where *File* is any atom other than `user`, denotes a file *File* (with optional suffixes as above) sought in the current working directory.

Filenames which begin with '`~`' or '`$`' are treated specially. For example,

`'~/sample.pl'`
: is equivalent to '`/home/sics/al/sample.pl`', if '`/home/sics/al`' is the user's home directory. (This is also equivalent to '`$HOME/sample.pl`' as explained below.)

`'~clyde/sample.pl'`
: is equivalent to '`/home/sics/clyde/sample.pl`', if '`/home/sics/clyde`' is Clyde's home directory.

`'$UTIL/sample.pl'`

is equivalent to ``'/usr/local/src/utilities/sample.pl''``, if ``'/usr/local/src/utilities''`` is the value of the environment variable `UTIL`, as defined by the shell command `setenv`.

Failure to open a file normally causes raising of an exception. This behavior can be turned off and on by of the built-in predicates `nofileerrors/0` and `fileerrors/0` decribed below.

4.1.1 Reading-in Programs

If the predicates discussed in this section are invoked in the scope of the interactive toplevel, filenames are relative to the current working directory. If invoked recursively, i.e. in the scope of another invocation of one of these predicates, filenames are relative to the directory of the file being read in. See Chapter 3 [Load Intro], page 25 for an introduction to these predicates.

`consult(`*:Files*`)`
`reconsult(`*:Files*`)`
`[]`
`[`*:File*`|+`*Files*`]`

Consults the source file or list of files specified by *File* and *Files*. If *File* and *Files* are prefixed by a module name, the module is created if it doesn't exist already, and the files are consulted into it. (These predicates are not available in Runtime Systems.)

`compile(`*:Files*`)`

Compiles the source file or list of files specified by *Files*. The compiled code is placed in-core, i.e. is added incrementally to the Prolog database. If *Files* are prefixed by a module name, the module is created if it doesn't exist already, and the files are compiled into it. (This predicate is not available in Runtime Systems.)

`fcompile(`*:Files*`)`

Compiles the source file or list of files specified by *Files*. If *Files* are prefixed by a module name, that module name will be used for module name expansion during the compilation (see Section 5.7 [Mod Compile], page 86). The suffix '`.pl`' is added to the given filenames to yield the real source filenames. The compiled code is placed on the object file or list of files formed by adding the suffix '`.ql`' to the given filenames. (This predicate is not available in Runtime Systems.)

`load(`*:Files*`)`

Loads the object file or list of files specified by *Files*. If *Files* are prefixed by a module name, the module is created if it doesn't exist already, and the files are loaded into it. (The predicate `load/1` will not write any progress messages in Runtime Systems.)

`ensure_loaded(`*:Files*`)`

Compiles or loads the file or list of files specified by *Files*, comparing last modified times with the time that the file was last read in. If *Files* are prefixed by a module name, the module is created if it doesn't exist already, and the files are loaded into it. (The predicate `ensure_loaded/1` is equivalent to `load/1` in Runtime Systems.)

`source_file(`*?File*`)`

File is the absolute name of a source file currently in the system. (This predicate is not available in Runtime Systems.)

`source_file(`*:Head*`,`*?File*`)`
`source_file(`*-Head*`,`*?File*`)`

Head is the most general goal for a predicate loaded from *File*. (This predicate is not available in Runtime Systems.)

4.1.2 Input and Output of Terms

Several IO predicates that use the current input or output stream available in an alternative version where the stream is specified explicitly. The rule is that the stream is the first argument, which defaults to the current input or output stream, depending on context.

`read(`*?Term*`)`
`read(+`*Stream*`,`*?Term*`)`

The next term, delimited by a full-stop (i.e. a `.` followed by either a space or a control character), is read from *Stream* and is unified with *Term*. The syntax of the term must agree with current operator declarations. If a call `read(`*Stream*, *Term*`)` causes the end of *Stream* to be reached, *Term* is unified with the term `end_of_file`. Further calls to `read/2` for the same stream will then raise an exception, unless the stream is connected to the terminal.

`write(`*?Term*`)`
`write(+`*Stream*`,`*?Term*`)`

The term *Term* is written onto *Stream* according to current operator declarations.

`display(`*?Term*`)`

The term *Term* is displayed *onto the standard output stream* (which is not necessarily the current output stream) in standard parenthesized prefix notation.

`write_canonical(`*?Term*`)`
`write_canonical(+`*Stream*`,`*?Term*`)`

Similar to `write(`*Stream*`,`*Term*`)`. The term will be written according to the standard syntax. The output from `write_canonical/2` can be parsed by `read/2` even if the term contains special characters or if operator declarations have changed.

`writeq(`*?Term*`)`
`writeq(+`*Stream*`,`*?Term*`)`

Similar to `write(`*Stream*`,`*Term*`)`, but the names of atoms and functors are quoted where necessary to make the result acceptable as input to `read/2`.

`print(?Term)`
`print(+Stream,?Term)`

Print *Term* onto *Stream*. This predicate provides a handle for user defined pretty printing:

- If *Term* is a variable then it is output using `write(Stream,Term)`.
- If *Term* is non-variable then a call is made to the user defined predicate `portray/1`. If this succeeds then it is assumed that *Term* has been output.
- Otherwise `print/2` is called recursively on the components of *Term*, unless *Term* is atomic in which case it is written via `write/2`.

In particular, the debugging package prints the goals in the tracing messages, and the interpreter top level prints the final values of variables. Thus you can vary the forms of these messages if you wish.

Note that on lists (`[_|_]`), `print/2` will first give the whole list to `portray/1`, but if this fails it will only give each of the (top level) elements to `portray/1`. That is, `portray/1` will not be called on all the tails of the list.

`portray(+Term)`

A user defined predicate. This should either print the *Term* and succeed, or do nothing and fail. In the latter case, the default printer (`write/1`) will print the *Term*.

`portray_clause(+Clause)`
`portray_clause(+Stream,+Clause)`

This writes the clause *Clause* onto *Stream* exactly as `listing/0-1` would have written it, including a period at the end.

`format(+Format,+Arguments)`
`format(+Stream,+Format,+Arguments)`

Print *Arguments* onto *Stream* according to format *Format*. *Format* is a list of formatting characters. If *Format* is an atom then `name/2` (see Section 4.7 [Meta Logic], page 56) will be used to translate it into a list of characters. Thus

```
| ?- format("Hello world!", []).
```

has the same effect as

```
| ?- format('Hello world!', []).
```

`format/2` and `format/3` is a Prolog interface to the C `stdio` function `printf()`. It is due to Quintus Prolog.

Arguments is a list of items to be printed. If there is only one item it may be supplied as an atom. If there are no items then an empty list should be supplied.

The default action on a format character is to print it. The character `~` introduces a control sequence. To print a `~` repeat it:

```
| ?- format("Hello ~~world!", []).
```

will result in

```
Hello ~world!
```

A format may be spread over several lines. The control sequence `\c` followed by a `LFD` will translate to the empty string:

```
| ?- format("Hello \c
world!", []).
```

will result in

```
Hello world!
```

The general format of a control sequence is '~*NC*'. The character *C* determines the type of the control sequence. *N* is an optional numeric argument. An alternative form of *N* is '*'. '*' implies that the next argument in *Arguments* should be used as a numeric argument in the control sequence. Example:

```
| ?- format("Hello~4cworld!", [0'x]).
```

and

```
| ?- format("Hello~*cworld!", [4,0'x]).
```

both produce

```
Helloxxxxworld!
```

The following control sequences are available.

'~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 an character code. *N* defaults to one and is interpreted as the number of times to print the character.

'~*N*e'
'~*N*E'
'~*N*f'
'~*N*g'
'~*N*G' (Print float). The argument is a float. The float and *N* will be passed to the C `printf()` function as

```
printf("%.Ne", Arg)
printf("%.NE", Arg)
printf("%.Nf", Arg)
printf("%.Ng", Arg)
printf("%.NG", Arg)
```

respectively.

If *N* is not supplied the action defaults to

```
printf("%e", Arg)
printf("%E", Arg)
printf("%f", Arg)
printf("%g", Arg)
printf("%G", Arg)
```

respectively.

'~*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. Example:

```
| ?- format("Hello ~1d world!", [42]).
| ?- format("Hello ~d world!", [42]).
```

will print as

```
Hello 4.2 world!
Hello 42 world!
```

respectively.

'~ND' (Print decimal.) The argument is an integer. Identical to '~Nd' except that ',' will separate groups of three digits to the left of the decimal point. Example:

```
| ?- format("Hello ~1D world!", [12345]).
```

will print as

```
Hello 1,234.5 world!
```

'~Nr' (Print radix.) The argument is an integer. *N* is interpreted as a radix. *N* should be >= 2 and <= 36. If *N* is missing the radix defaults to 8. The letters 'a-z' will denote digits larger than 9. Example:

```
| ?- format("Hello ~2r world!", [15]).
| ?- format("Hello ~16r world!", [15]).
```

will print as

```
Hello 1111 world!
Hello f world!
```

respectively.

'~NR' (Print radix.) The argument is an integer. Identical to '~Nr' except that the letters 'A-Z' will denote digits larger than 9. Example:

```
| ?- format("Hello ~16R world!", [15]).
```

will print as

```
Hello F world!
```

'~Ns' (Print string.) The argument is a list of character codes. Exactly *N* characters will be printed. *N* defaults to the length of the string. Example:

```
| ?- format("Hello ~4s ~4s!", ["new","world"]).
| ?- format("Hello ~s world!", ["new"]).
```

will print as

```
Hello new  worl!
Hello new world!
```

respectively.

'~i' (Ignore argument.) The argument may be of any type. The argument will be ignored. Example:

```
| ?- format("Hello ~i~s world!", ["old","new"]).
```

will print as

```
Hello new world!
```

'~k' (Print canonical.) The argument may be of any type. The argument will be passed to `write_canonical/2` (see Section 4.1.2 [Term IO], page 36). Example:

```
| ?- format("Hello ~k world!", [[a,b,c]]).
```

will print as

```
Hello .(a,.(b,.(c,[]))) world!
```

'~p' (Print.) The argument may be of any type. The argument will be passed to `print/2` (see Section 4.1.2 [Term IO], page 36). Example:

```
| ?- assert((portray([X|Y]) :- print(cons(X,Y)))).
| ?- format("Hello ~p world!", [[a,b,c]]).
```

will print as

```
Hello cons(a,cons(b,cons(c,[]))) world!
```

'~q' (Print quoted.) The argument may be of any type. The argument will be passed to `writeq/2` (see Section 4.1.2 [Term IO], page 36). Example:

```
| ?- format("Hello ~q world!", [['A','B']]).
```

will print as

```
Hello ['A','B'] world!
```

'~w' (Write.) The argument may be of any type. The argument will be passed to `write/2` (see Section 4.1.2 [Term IO], page 36). Example:

```
| ?- format("Hello ~w world!", [['A','B']]).
```

will print as

```
Hello [A,B] world!
```

'~*N*n' (Print newline.) Print *N* newlines. *N* defaults to 1. Example:

```
| ?- format("Hello ~n world!", []).
```

will print as

```
Hello
 world!
```

4.1.3 Character Input/Output

There are two sets of character IO predicates. The first set uses the current input and output streams, while the second set always uses the standard input and output streams. The first set is available in an alternative version where the stream is specified explicitly. The rule is that the stream is the first argument, which defaults to the current input or output stream, depending on context.

`nl`
`nl(+`*Stream*`)`

A new line is started on *Stream* by printing a line feed (`LFD`). If *Stream* is the terminal, its buffer is flushed.

`get0(`*?N*`)`
`get0(+`*Stream*`,`*?N*`)`

N is the character code of the next character read from *Stream*. If a call `get0(`*Stream*`,` *N*`)` causes the end of *Stream* to be reached, *N* is -1. Further calls to `get0/2` for the same stream will then raise an exception, unless the stream is connected to the terminal.

`get(`*?N*`)`
`get(+`*Stream*`,`*?N*`)`

Same as `get0/2`, except *N* is the character code of the next character that is not a *layout-char* (see Section 8.9.4 [Token String], page 127) read from *Stream*.

`skip(+`*N*`)`
`skip(+`*Stream*`,+`*N*`)`

Skips just past the next character code *N* from *Stream*. *N* may be an arithmetic expression.

`put(+`*N*`)`
`put(+`*Stream*`,+`*N*`)`

character code *N* is output onto *Stream*. *N* may be an arithmetic expression.

`tab(+`*N*`)`
`tab(+`*Stream*`,+`*N*`)`

N spaces are output onto *Stream*. *N* may be an arithmetic expression.

The above predicates are the ones which are the most commonly used, as they can refer to any streams. In most cases these predicates are sufficient, but there is one limitation: if you are writing to the terminal, the output is not guaranteed to be visible until a newline character is written. If this line by line output is inadequate, you must use `ttyflush/0` (see below).

The predicates which follow always refer to the terminal. They are convenient for writing interactive programs which also perform file IO.

`ttynl` A new line is started on the standard output stream and its buffer is flushed.

`ttyflush` Flushes the standard output stream buffer. Output to the terminal normally simply goes into an output buffer until such time as a newline is output. Calling this predicate forces any characters in this buffer to be output immediately.

`ttyget0(`*?N*`)`

N is the character code of the next character input from the standard input stream.

`ttyget(`*?N*`)`

N is the character code of the next non-blank printable character from the standard input stream.

`ttyput(+`*N*`)`

The character code *N* is output to the standard output stream. *N* may be an arithmetic expression.

`ttyskip(+`*N*`)`

Skips to just past the next character code *N* from the standard input stream. *N* may be an arithmetic expression.

`ttytab(+`*N*`)`

N spaces are output to the standard output stream. *N* may be an arithmetic expression.

4.1.4 Stream IO

The following predicates manipulate streams. Character and line counts are maintained per stream. All streams connected to the terminal, however, share the same set of counts. For example, writing to `user_output` will advance the counts for `user_input`, if both are connected to the terminal.

`open(+`*FileName*`,+`*Mode*`,-`*Stream*`)`

If *FileName* is a valid file name, the file is opened in mode *Mode* (invoking the UNIX function `fopen`) and the resulting stream is unified with *Stream*. *Mode* is one of:

`read` Open the file for input.

`write` Open the file for output. The file is created if it does not already exist, the file will otherwise be truncated.

`append` Open the file for output. The file is created if it does not already exist, the file will otherwise be appended to.

If *FileName* is an integer, it is assumed to be a file descriptor passed to Prolog from a foreign language function call. The file descriptor is connected to a Prolog stream (invoking the UNIX function `fdopen`) which is unified with *Stream*.

`close(+`*X*`)`

If *X* is a stream the stream is closed. If *X* is the name of a file opened by `see/1` or `tell/1` the corresponding stream is closed.

`absolute_file_name(+`*RelativeName*`,?`*AbsoluteName*`)`

This predicate is used by all predicates that refer to filenames for resolving these. The argument *RelativeName* is interpreted as a filename according to the filename syntax rules (see Section 4.1 [Input Output], page 33). If the specified file is found (possibly with a '`.pl`' or '`.ql`' extension if consulting or compiling source files or loading object files), *AbsoluteName* is unified with the full path name of this file. If *RelativeName* is `user`, then *AbsoluteName* is also unified with `user`; this "filename" stands for the standard input or output stream, depending on context.

`current_input(?`*Stream*`)`

Unify *Stream* with the current input stream.

`current_output(?`*Stream*`)`

Unify *Stream* with the current output stream.

`current_stream(?`*FileName*`,?`*Mode*`,?`*Stream*`)`

Stream is a stream which was opened in mode *Mode* and which is connected to the absolute file name *Filename* (an atom) or to the file descriptor *Filename* (an integer). This predicate can be used for enumerating all currently open streams through backtracking.

`set_input(+`*Stream*`)`

Set the current input stream to *Stream*.

`set_output(+`*Stream*`)`

Set the current output stream to *Stream*.

`flush_output(+Stream)`

Flush all internally buffered characters for *Stream* to the operating system.

`library_directory(?Directory)`

A user defined predicate. This predicate specifies a set of directories to be searched when a file specification of the form `library(Name)` is used. The directories are searched until a file with the name '*NameSuffix*' or '*Name*' is found (see Section 4.1 [Input Output], page 33), where *Suffix* is '`.ql`' when loading object files and '`.pl`' otherwise.

Directories to be searched may be added by using `asserta/1` or `assertz/1` (see Section 4.8 [Modify Prog], page 58), provided that `library_directory/1` has been declared to be dynamic:

```
| ?- assertz(library_directory((Directory))).
```

`open_null_stream(-Stream)`

Open an output stream to the null device. Everything written to this stream will be thrown away.

`character_count(+Stream,?N)`

N is the number of characters read/written on stream *Stream*.

`line_count(+Stream,?N)`

N is the number of lines read/written on stream *Stream*.

`line_position(+Stream,?N)`

N is the number of characters read/written on the current line of *Stream*.

`stream_code(+Stream,?StreamCode)`
`stream_code(?Stream,+StreamCode)`

StreamCode is the file descriptor (an integer) corresponding to the Prolog stream *Stream*. This predicate is only useful when streams are passed between Prolog and C. A C function wishing to perform I/O on a stream may compute the `FILE *` stream pointer as `(stdin + fd)`, where `fd` is the file descriptor passed from Prolog. Conversely, the file descriptor can be computed as `fileno(s)` from the `FILE *` stream pointer `s`.

Warning: Mixing C I/O and Prolog I/O on the same stream is not recommended practice. The problem is that the character and line counts for a stream are only kept up to date for Prolog I/O (see `character_count/2`, `line_count/2`, and `line_position/2`).

`fileerrors`

Undoes the effect of `nofileerrors/0`.

`nofileerrors`

After a call to this predicate, failure to locate or open a file will cause the operation to fail instead of the default action, which is to raise an exception with an error message.

4.1.5 Socket IO

The following predicates manipulate sockets. They are all rather straight-forward interfaces to the corresponding C-library predicates with the same name (except `current_host/1`). The reader should therefore study the C-manual for a deeper description.

An interrupt-handler could be declared to be evaluated when something is readable on those streams (see socket_interrupt/3 below).

The *Domain*, is either the atom `'AF_INET'` or `'AF_UNIX'`. They correspond directly to the same domains in UNIX.

An *Address* is either `'AF_INET'(Host,Port)` or `'AF_UNIX'(SocketName)`. *Host* is an atomic hostname, *Port* is a portnumber and *SocketName* is an atom denoting a socket. A reader familiar with UNIX's sockets will understand this immediately.

All streams below can be both read from and written on. All IO-predicates operating on streams can be used, for example `read/2`, `write/2`, `format/3`, `current_stream/3`, etc.

`socket(+`*Domain*`,-`*Socket*`)`
: A socket *Socket* in the domain *Domain* is created.

`socket_bind(+`*Socket*`,?`*Address*`)`
: The socket *Socket* is bound to *Address*.

`socket_connect(+`*Socket*`,?`*Address*`,-`*Stream*`)`
: The socket *Socket* is connected to *Address*. *Stream* is a special stream on which items can be both read and written.

`socket_listen(+`*Socket*`,+`*Length*`)`
: The socket *Socket* is defined to have a maximum backlog queue of *Length* pending connections.

`socket_accept(+`*Socket*`,-`*Stream*`)`
: The first connection to socket *Socket* is extracted. The stream *Stream* stream is opened for read and write on this connection.

`socket_select(?`*Socket*`,-`*NewStream*`,+`*TimeOut*`,+`*Streams*`,-`*ReadStreams*`)`
: The list of streams in *Streams* is checked for readable characters. The streams are either on the form `'$stream'/2` or the atom `user_input`. The list *ReadStreams* returns the streams with readable data. If *TimeOut* is instantiated to `off`, the predicate waits until something is available. If *TimeOut* is `S:U` the predicate waits at most `S` seconds and `U` microseconds. Both `S` and `U` must be integers `>=`0. If there is a timeout, *ReadStreams* is `[]`. If *Socket* is instantiated, `socket_select/5` also waits for connections to this socket. If a connection is found, *NewStream* will be a stream for communicating with the connecting process.

`current_host(?`*HostName*`)`
: *HostName* is unified with the name of the machine the process is executing on.

`socket_interrupt(`*?Stream*`,`*?OldHandler*`,`*?NewHandler*`)`

unifies *OldHandler* with the current interrupt handler and installs *NewHandler* as the new interrupt handler for the stream *Stream* associated with a socket. Interrupt handlers are specified as atoms. The atom `off` indicates that the interrupt mechanism is turned off for *Stream*. Any other atom is the name of a predicate invoked when something is readable on *Stream*. The handler predicate has one argument, the stream that is readable. For example,

```
socket_interrupt(Stream, _, int_handler).
```

will enable the interrupt mechanism. Given the predicate

```
int_handler(Stream) :-
        read(Stream, Data),
        write(Data), nl.
```

the term read from *Stream* will be written to the current output. NOTE: there is no guarantee that a complete Prolog term is available yet. If not, `read/2` will suspend as usual. (This predicate is not available in Runtime Systems.)

4.1.6 DEC-10 Prolog File IO

The following predicates manipulate files.

`see(+`*File*`)`

File *File* becomes the current input stream. *File* may be a stream previously opened by `see/1` or a filename. If it is a filename, the following action is taken: If there is a stream opened by `see/1` associated with the same file already, then it becomes the current input stream. Otherwise, the file *File* is opened for input and made the current input stream.

`seeing(`*?FileName*`)`

FileName is unified with the name of the current input file, if it was opened by `see/1`, with the current input stream, if it is not `user_input`, otherwise with `user`.

`seen` Closes the current input stream, and resets it to `user_input`.

`tell(+`*File*`)`

File *File* becomes the current output stream. *File* may be a stream previously opened by `tell/1` or a filename. If it is a filename, the following action is taken: If there is a stream opened by `tell/1` associated with the same file already, then it becomes the current output stream. Otherwise, the file *File* is opened for output and made the current output stream.

`telling(`*?FileName*`)`

FileName is unified with the name of the current output file, if it was opened by `tell/1`, with the current output stream, if it is not `user_output`, otherwise with `user`.

`told` Closes the current output stream, and resets it to `user_output`.

4.1.7 An Example

Here is an example of a common form of file processing:

```
process_file(F) :-
        seeing(OldInput),
        see(F),                 % Open file F
        repeat,
          read(T),              % Read a term
          process_term(T),      % Process it
          T == end_of_file,     % Loop back if not at end of file
        !,
        seen,                   % Close the file
        see(OldInput).
```

The above is an example of a *repeat loop*. Nearly all sensible uses of `repeat/0` follow the above pattern. Note the use of a cut to terminate the loop.

4.2 Arithmetic

Arithmetic is performed by built-in predicates which take as arguments *arithmetic expressions* and evaluate them. An arithmetic expression is a term built from numbers, variables, and functors that represent arithmetic functions. At the time of evaluation, each variable in an arithmetic expression must be bound to a non-variable expression. An expression evaluates to a number, which may be an *integer* or a *float*.

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).

Only certain functors are permitted in an arithmetic expression. These are listed below, together with an indication of the functions they represent. *X* and *Y* are assumed 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.

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

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

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

X`-`*Y*
: The value is the difference of *X* and *Y*.

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

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

X//*Y* The value is the *integer* quotient of *X* and *Y*.

X mod *Y* The value is the *integer* remainder after dividing *X* by *Y*, i.e. `integer(X)-integer(Y)*(X//Y)`.

`integer(X)`
The value is the closest integer between *X* and 0, if *X* is a float, otherwise to *X* itself.

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

X/*Y* The value is the bitwise conjunction of the integers *X* and *Y*.

X\\/*Y* The value is the bitwise disjunction of the integers *X* and *Y*.

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

\\(*X*) The value is the bitwise negation of the integer *X*.

X<<*Y* The value is the integer *X* shifted left by *Y* places.

X>>Y The value is the integer *X* shifted right by *Y* places.

[*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.

SICStus also includes an extra set of functions listed below. These may not be supported by other Prologs. All trigonometric and transcendental functions take float arguments and deliver float values. The trigonometric functions take arguments or deliver values in radians.

`abs(X)` The value is the absolute value of *X*.

`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 most significant bit position of the integer *X*. This is computed as the greatest integer such that `1<<msb(X) =< X` for positive *X* and `1<<msb(X) >= X` for negative *X*.

`round(X)` The value is the float that is the closest integral value to *X*. If *X* is exactly half-way between two integers, it must be rounded to the closest even integral value.

`truncate(X)`
The value is the float that is the closest integer between *X* and 0.

`floor(X)` The value is the float that is the greatest integral value less or equal to *X*.

`ceiling(X)`
The value is the float that is the least integral value greater or equal to *X*.

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

`cos(X)` The value is the cosine of *X*.

`tan(X)` The value is the tangent of *X*.

`cot(`X`)` The value is the cotangent of X.

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

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

`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`)` 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`)` The value is the square root of X.

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

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

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

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

`inf` The value is infinity as defined in the IEEE standard.

`nan` The value is not-a-number as defined in the IEEE standard.

Variables in an arithmetic expression which is to be evaluated may be bound to other arithmetic expressions rather than just numbers, e.g.

```
evaluate(Expression, Answer) :- Answer is Expression.

| ?- evaluate(24*9, Ans).
Ans = 216 ?

yes
```

Arithmetic expressions, as described above, are just data structures. If you want one evaluated you must pass it as an argument to one of the built-in predicates listed below. Note that it only evaluates one of its arguments, whereas all the comparison predicates evaluate both of theirs. In the following, X and Y stand for arithmetic expressions, and Z for some term.

Z `is` X	The arithmetic expression X is evaluated and the result is unified with Z. Fails if X is not an arithmetic expression.
X `=:=` Y	The numeric values of X and Y are equal.
X `=\=` Y	The numeric values of X and Y are not equal.
X `<` Y	The numeric value of X is less than the numeric value of Y.
X `>` Y	The numeric value of X is greater than the numeric value of Y.
X `=<` Y	The numeric value of X is less than or equal to the numeric value of Y.
X `>=` Y	The numeric value of X is greater than or equal to the numeric value of Y.

4.3 Comparison of Terms

These built-in predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison (see Section 4.2 [Arithmetic], page 46) or unification.

The predicates make reference to a *standard total ordering* of terms, which is as follows:

- Variables, in a standard order (roughly, 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 the arguments (in left-to-right order). Recall that lists are equivalent to compound terms with principal functor `./2`.

For example, here is a list of terms in the standard order:

```
[ X, -1.0, -9, 1, fie, foe, X = Y, foe(0,2), fie(1,1,1) ]
```

These are the basic predicates for comparison of arbitrary terms:

Term1 `==` *Term2*

Tests if the terms currently instantiating *Term1* and *Term2* are literally identical (in particular, variables in equivalent positions in the two terms must be identical). 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 Section 4.15 [Misc Pred], page 75).

Term1 `\==` *Term2*

Tests if the terms currently instantiating *Term1* and *Term2* are not literally identical.

Term1 `@<` *Term2*

Term *Term1* is before term *Term2* in the standard order.

Term1 `@>` *Term2*

Term *Term1* is after term *Term2* in the standard order.

Term1 `@=<` *Term2*

Term *Term1* is not after term *Term2* in the standard order.

Term1 `@>=` *Term2*

Term *Term1* is not before term *Term2* in the standard order.

Some further predicates involving comparison of terms are:

`?=(`*?X*`,`*?Y*`)`

Succeeds if *X* and *Y* are either identical or non-unifiable.

`compare(`*?Op*`,`*?Term1*`,`*?Term2*`)`

The result of comparing terms *Term1* and *Term2* is *Op*, where the possible values for *Op* are:

`=` if *Term1* is identical to *Term2*,

`<` if *Term1* is before *Term2* in the standard order,

`>` if *Term1* is after *Term2* in the standard order.

Thus `compare(=,Term1,Term2)` is equivalent to `Term1 == Term2`.

`sort(+`*List1*`,`*?List2*`)`

The elements of the list *List1* are sorted into the standard order (see Section 4.3 [Term Compare], page 49) and any identical elements are merged, yielding the list *List2*. (The time and space complexity of this operation is at worst *O(N lg N)* where *N* is the length of *List1*.)

`keysort(+`*List1*`,?`*List2*`)`

The list *List1* must consist of items of the form *Key-Value*. These items are sorted into order according to the value of *Key*, yielding the list *List2*. No merging takes place. This predicate is *stable*, i.e. if `K-A` occurs before `K-B` in the input, then `K-A` will occur before `K-B` in the output. (The time and space complexity of this operation is at worst *O(N lg N)* where *N* is the length of *List1*.)

4.4 Control

`+`*P* `, +`*Q* Prove *P* and if it succeeds, then prove *Q*.

`+`*P* `; +`*Q* Prove *P* or if it fails (or if the continuation fails), prove *Q* instead of *P*.

`!` See Section 8.5 [Cut], page 119.

`\+ +`*P* If the goal *P* has a solution, fail, otherwise succeed. This is not real negation ("*P* is false"), but a kind of pseudo-negation meaning "*P* is not provable". It is defined as if by

```
\+(P) :- P, !, fail.
\+(_).
```

No cuts are allowed in ***P*****.**

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.

`+`*P* `-> +`*Q* `; +`*R*

Analogous to

if *P* then *Q* else *R*

i.e. defined as if by

```
(P -> Q; R) :- P, !, Q.
(P -> Q; R) :- R.
```

No cuts are allowed in ***P*****.**

Note that this form of if-then-else only explores *the first* solution to the goal *P*.

Note also that the `;` is not read as a disjunction operator in this case; instead, it is part of the if-then-else construction.

The precedence of `->` is less than that of `;` (see Section 8.6 [Operators], page 120), so the expression is read as

`;(->(P,Q),R)`

`+P -> +Q` When occurring as a goal, this construction is read as equivalent to

`(P -> Q; fail)`

`if(+P,+Q,+R)`

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.

No cuts are allowed in *P*.

`otherwise`

`true` These always succeed. Use of `otherwise/0` is discouraged, because it is not as portable as `true/0`, and because the former may suggest a completely different semantics than the latter.

`false`

`fail` These always fail. Use of `false/0` is discouraged, because it is not as portable as `fail/0`, and because the latter has a more procedural flavor to it.

`repeat` 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(OldState),
        repeat,
          generate(Datum),
          action(Datum),
          test(Datum),
        !,
        restore(OldState),
        ...
```

The purpose is to repeatedly perform some *action* on elements which are somehow *generated*, e.g. by reading them from a stream, until some *test* becomes true. Usually, *generate*, *action*, and *test* are all determinate. Repeat loops cannot contribute to the logic of the program. They are only meaningful if the *action* involves side-effects.

The only reason for using repeat loops instead of a more natural tail-recursive formulation is efficiency: when the *test* fails back, the Prolog engine immediately reclaims any working storage consumed since the call to `repeat/0`.

`call(:Term)`

`incore(:Term)`

:Term If *Term* is instantiated to a term which would be acceptable as the body of a clause, then the goal `call(Term)` is executed exactly as if that term appeared textually in its place, except that any cut (`!`) occurring in *Term* only cuts alternatives in the execution of *Term*. Use of `incore/1` is not recommended.

If *Term* is not instantiated as described above, an error message is printed and the call fails.

`time_out(`*:Goal*`,+`*Time*`,`*?Result*`)`

The *Goal* is executed as if by `call/1`. If the execution exceeds *Time* milliseconds the goal will fail and *Result* is unified with the atom `time_out`. If the goal succeds within the specified time *Result* is unified with the atom `success`. *Time* must be a number between (not including) 0 and 2147483647.

N.B. This predicate cannot be used to interrupt waiting input-operations. (This predicate is not available in Runtime Systems.)

4.5 Error and Exception Handling

The two built-in predicates `on_exception/3` and `raise_exception/1` are used to alter the control flow to meet exception and error conditions. The equivalent of a `raise_exception/1` is also executed by the built-in predicates when errors occur.

`on_exception(`*?Pattern*`,`*:ProtectedGoal*`,`*:Handler*`)`
`raise_exception(+`*Exception*`)`

`on_exception/3` calls *ProtectedGoal*. If this succeeds or fails, so does the call to `on_exception/3`. If however, during the execution of *ProtectedGoal*, there is a call to `raise_exception(`*Exception*`)`, then *Exception* is copied and the stack is unwound back to the call to `on_exception/3`, whereupon the copy of *Exception* is unified with *Pattern*. If this unification succeeds, then `on_exception/3` calls the goal *Handler* in order to determine the success or failure of `on_exception/3`. Otherwise, the stack keeps unwinding, looking for an earlier invocation of `on_exception/3`. *Exception* may be any term.

At the toplevel, any previously uncaught exception is caught and an appropriate error message is printed. This printing may be controlled by the user by defining the predicate `print_uncaught_exception/1`.

`print_uncaught_exception(+`*Exception*`)`

A user defined predicate. *Exception* is an exception caught at the toplevel. This predicate may perform user defined printing and succeed, or do nothing and fail. In the latter case, the default printer will print the *Exception*.

In Runtime Systems, uncaught exceptions are caught at the innermost call from C instead.

The built-in predicates may raise exceptions as follows on errors. Usually the goal and the argument number causing the error are contained in the exception.

`instantiation_error(`*Goal*`,`*ArgNo*`)`

> *Goal* was called with insufficiently instantiated variables.

`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(`*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(`*Goal*`,`*ArgNo*`,`*ObjectType*`,`*Culprit*`,`*Reserved*`)`

> Something does not exist as indicated by the arguments. If the unknown-flag (see `prolog_flag/3`) is set to `error`, this error is raised with *ArgNo* set to 0.

`permission_error(`*Goal*`,`*Operation*`,`*ObjectType*`,`*Culprit*`,`*Reserved*`)`

> The *Operation* is not permitted on *Culprit* of the *ObjectType*.

`context_error(`*Goal*`,`*ContextType*`,`*CommandType*`)`

> The *CommandType* is not permitted in *ContextType*.

`syntax_error(`*Goal*`,`*Position*`,`*Message*`,`*Tokens*`,`*AfterError*`)`

> A syntax error was found when reading a term with `read/1-2`. This error is raised only if the `syntax_errors` flag (see `prolog_flag/3`) is set to `error`.

`system_error(`*Message*`)`

> An error occurred while dealing with the operating system.

`internal_error(`*Message*`)`

> This kind of error should not normally occur.

4.6 Information about the State of the Program

`listing`

> Lists onto the current output stream all the clauses in the current interpreted program (in the type-in module, see Section 5.2 [Module Spec], page 83). Clauses listed onto a file can be consulted back.

`listing(`*:Spec*`)`

> Lists all interpreted predicates covered by *Spec*, which is described under `spy/1` (see Section 2.4 [Spy-Point], page 18), but which may also contain variables. For example:

```
| ?- listing([concatenate/3, reverse, m:go/2-3, bar:_]).
```

`ancestors(`*?Goals*`)`

> Unifies *Goals* with a list of ancestor goals for the current clause. The list starts with the parent goal and ends with the most recent ancestor coming from a call in a compiled clause.
>
> The predicate is only available when the debugger is switched on. (This predicate is not available in Runtime Systems.)

`subgoal_of(`*?S*`)`

Equivalent to the sequence of goals:

`| ?- ancestors(`*Goals*`), member(`*S*`, `*Goals*`).`

where the predicate `member/2` (not a built-in predicate) successively matches its first argument with each of the elements of its second argument. See Section 1.4 [Directives], page 7.

The predicate is only available when the debugger is switched on. (This predicate is not available in Runtime Systems.)

`current_atom(`*?Atom*`)`

If *Atom* is instantiated then test if *Atom* is an Atom.

If *Atom* is unbound then generate (through backtracking) all currently known atoms, and return each one as *Atom*.

`current_predicate(`*?Name*`,`*:Head*`)`
`current_predicate(`*?Name*`,`*-Head*`)`

Name is the name of a user defined predicate, and *Head* is the most general goal for that predicate, possibly prefixed by a module name. This predicate can be used to enumerate all user defined predicates through backtracking.

`predicate_property(`*:Head*`,`*?Property*`)`
`predicate_property(`*-Head*`,`*?Property*`)`

Head is the most general goal for an existing predicate, possibly prefixed by a module name, and *Property* is a property of that predicate, where the possible properties are

- one of the atoms `built_in` (for built-in predicates) or `compiled` or `interpreted` (for user defined predicates),
- zero or more of the atoms `dynamic` and `multifile`, for predicates that have been declared to have these properties (see Section 3.2.7 [Declarations], page 30),
- one or more terms `(block `*Term*`)` for predicates that have block declarations (see Section 3.2.7 [Declarations], page 30),
- the atom `exported` or terms `imported_from(`*ModuleFrom*`)` for predicates exported or imported from modules (see Chapter 5 [Module Intro], page 83),
- the term `(meta_predicate `*Term*`)` for predicates that have meta-predicate declarations (see Section 5.6 [Meta Decl], page 86).

This predicate can be used to enumerate all existing predicates and their properties through backtracking.

4.7 Meta-Logic

The predicates in this section are meta-logical and perform 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.

`var(`*?X*`)` Tests whether *X* is currently uninstantiated (`var` is short for variable). An uninstantiated variable is one which has not been bound to anything, except possibly another uninstantiated variable. Note that a structure with some components which are uninstantiated is not itself considered to be uninstantiated. Thus the directive

```
| ?- var(foo(X, Y)).
```

always fails, despite the fact that *X* and *Y* are uninstantiated.

`nonvar(`*?X*`)`

Tests whether *X* is currently instantiated. This is the opposite of `var/1`.

`ground(`*?X*`)`

Tests whether *X* is completely instantiated, i.e. free of unbound variables.

`atom(`*?X*`)` Checks that *X* is currently instantiated to an atom (i.e. a non-variable term of arity 0, other than a number).

`float(`*?X*`)`

Checks that *X* is currently instantiated to a float.

`integer(`*?X*`)`

Checks that *X* is currently instantiated to an integer.

`number(`*?X*`)`

Checks that *X* is currently instantiated to a number.

`atomic(`*?X*`)`

Checks that *X* is currently instantiated to an atom or number.

`functor(+`*Term*`,?`*Name*`,?`*Arity*`)`
`functor(?`*Term*`,+`*Name*`,+`*Arity*`)`

The principal functor of term *Term* has name *Name* and arity *Arity*, where *Name* is either an atom or, provided *Arity* is 0, an integer. Initially, either *Term* must be instantiated, or *Name* and *Arity* must be instantiated to, respectively, either an atom and an integer in [0..256) or an atomic term and 0. In the case where *Term* is initially uninstantiated, the result of the call is to instantiate *Term* to the most general term having the principal functor indicated.

`arg(+`*ArgNo*`,+`*Term*`,?`*Arg*`)`

Initially, *ArgNo* must be instantiated to a positive integer and *Term* to a compound term. The result of the call is to unify *Arg* with the argument *ArgNo* of term *Term*. (The arguments are numbered from 1 upwards.)

`+`*Term* `=..` *?List*
?Term `=..` `+`*List*

List is 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*. E.g.

```
| ?- product(0, n, n-1) =.. L.

L = [product,0,n,n-1]

| ?- n-1 =.. L.

L = [-,n,1]

| ?- product =.. L.

L = [product]
```

If *Term* is uninstantiated, then *List* must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number. Note that 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.

`name(+`*Const*`,?`*CharList*`)`
`name(?`*Const*`,+`*CharList*`)`

If *Const* is an atom or number then *CharList* is a list of the character codes of the characters comprising the name of *Const*. E.g.

```
| ?- name(product, L).

L = [112,114,111,100,117,99,116]

| ?- name(product, "product").

| ?- name(1976, L).

L = [49,57,55,54]

| ?- name('1976', L).

L = [49,57,55,54]

| ?- name((:-), L).

L = [58,45]
```

If *Const* is uninstantiated, *CharList* must be instantiated to a list of character codes. If *CharList* can be interpreted as a number, *Const* is unified with that number, otherwise with the atom whose name is *CharList*. The length of *CharList* must be less than 512. E.g.

```
| ?- name(X, [58,45]).

X = :-

| ?- name(X, ":-").

X = :-

| ?- name(X, [49,50,51]).

X = 123
```

Note that there atoms are 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_chars/2` or `number_chars/2`, as these predicates do not have this inconsistency.

`atom_chars(+`*Const*`,?`*CharList*`)`
`atom_chars(?`*Const*`,+`*CharList*`)`

The same as `name(`*Const*`,`*CharList*`)`, but *Const* is constrained to be an atom.

`number_chars(+`*Const*`,?`*CharList*`)`
`number_chars(?`*Const*`,+`*CharList*`)`

The same as `name(`*Const*`,`*CharList*`)`, but *Const* is constrained to be a number.

4.8 Modification of the Program

The predicates defined in this section allow modification of the loaded program as it is actually running. Clauses can be added to the program (*asserted*) or removed from the program (*retracted*).

For these predicates, the argument *Head* must be instantiated to an atom or a compound term. The argument *Clause* must be instantiated either to a term *Head* `:-` *Body* or, if the body part is empty, to *Head*. An empty body part is represented as `true`.

Note that a term *Head* `:-` *Body* must be enclosed in parentheses when it occurs as an argument of a compound term, as '`:-`' is a standard infix operator with precedence greater than 1000 (see Section 8.6 [Operators], page 120), e.g.:

```
| ?- assert((Head :- Body)).
```

Like recorded terms, the clauses of dynamic predicates also have a unique implementation-defined identifier. Some of the predicates below have an additional argument which is this identifier. This identifier makes it possible to access clauses directly instead of requiring a normal database (hash-table) lookup. However it should be stressed that use of these predicates requires some extra care.

`assert(`*:Clause*`)`
`assert(`*:Clause*`,`*-Ref*`)`

The current instance of *Clause* is interpreted as a clause and is added to the current interpreted program. The predicate concerned must be currently be dynamic or undefined and the position of the new clause within it is implementation-defined. *Ref* is a unique identifier of the asserted clause. Any uninstantiated variables in the *Clause* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Chapter 6 [Blocking], page 89).

`asserta(`*:Clause*`)`
`asserta(`*:Clause*`,`*-Ref*`)`

Like `assert/2`, except that the new clause becomes the *first* clause for the predicate concerned.

`assertz(`*:Clause*`)`
`assertz(`*:Clause*`,`*-Ref*`)`

Like `assert/2`, except that the new clause becomes the *last* clause for the predicate concerned.

`clause(`*:Head*`,`*?Body*`)`
`clause(`*:Head*`,`*?Body*`,`*?Ref*`)`
`clause(`*?Head*`,`*?Body*`,+`*Ref*`)`

The clause `(`*Head* `:-` *Body*`)` exists in the current interpreted program, and is uniquely identified by *Ref*. The predicate concerned must currently be dynamic. At the time of call, either *Ref* must be instantiated to a valid identifier, or *Head* must be instantiated to an atom or a compound term. Thus `clause/3` can have two different modes of use.

`retract(`*:Clause*`)`

The first clause in the current interpreted program that matches *Clause* is erased. The predicate concerned must currently be dynamic. `retract/1` may be used in a non-determinate fashion, i.e. it will successively retract clauses matching the argument through backtracking. If reactivated by backtracking, invocations of the predicate whose clauses are being retracted will proceed unaffected by the retracts. This is also true for invocations of `clause` for the same predicate. The space occupied by a retracted clause will be recovered when instances of the clause are no longer in use.

`retractall(`*:Head*`)`

Erase all clauses whose head matches *Head*, where *Head* must be instantiated to an atom or a compound term. The predicate concerned must currently be dynamic. The predicate definition is retained.

`abolish(`*:Spec*`)`
`abolish(`*:Name*`,+`*Arity*`)`

Erase all clauses of the predicate specified by *Spec* or *Name*`/`*Arity*. *Spec* is described under `spy/1` (see Section 2.4 [Spy-Point], page 18) and *Name* may be prefixed by a module name (see Section 5.2 [Module Spec], page 83).

The predicate definition and all associated information such as spy-points is also erased. The predicates concerned must all be user defined.

4.9 Internal Database

The predicates described in this section were introduced in early implementations of Prolog to provide efficient means of performing operations on large quantities of data. The introduction of indexed dynamic predicates have rendered these predicates obsolete, and the sole purpose of providing them is to support existing code. There is no reason whatsoever to use them in new code.

These predicates store arbitrary terms in the database without interfering with the clauses which make up the program. The terms which are stored in this way can subsequently be retrieved via the key on which they were stored. Many terms may be stored on the same key, and they can be individually accessed by pattern matching. Alternatively, access can be achieved via a special identifier which uniquely identifies each recorded term and which is returned when the term is stored.

`recorded(`*?Key*`,`*?Term*`,`*?Ref*`)`

The internal database is searched for terms recorded under the key *Key*. These terms are successively unified with *Term* in the order they occur in the database. At the same time, *Ref* is unified with the implementation-defined identifier uniquely identifying the recorded item. If the key is instantiated to a compound term, only its principal functor is significant. If the key is uninstantiated, all terms in the database are successively unified with *Term* in the order they occur.

`recorda(+`*Key*`,`*?Term*`,`*-Ref*`)`

The term *Term* is recorded in the internal database as the first item for the key *Key*, where *Ref* is its implementation-defined identifier. The key must be given, and only its principal functor is significant. Any uninstantiated variables in the *Term* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Chapter 6 [Blocking], page 89).

`recordz(+`*Key*`,`*?Term*`,`*-Ref*`)`

Like `recorda/3`, except that the new term becomes the *last* item for the key *Key*.

`erase(+`*Ref*`)`

The recorded item (or dynamic clause (see Section 4.9 [Database], page 60)) whose implementation-defined identifier is *Ref* is effectively erased from the internal database or interpreted program.

`instance(+`*Ref*`,`*?Term*`)`

A (most general) instance of the recorded term or clause whose implementation-defined identifier is *Ref* is unified with *Term*. *Ref* must be instantiated to a legal identifier.

`current_key(`*?KeyName*`,`*?KeyTerm*`)`

KeyTerm is the most general form of the key for a currently recorded term, and *KeyName* is the name of that key. This predicate can be used to enumerate in undefined order all keys for currently recorded terms through backtracking.

4.10 All Solutions

When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following built-in predicates are provided to automate this process.

`setof(`*?Template*`,`*:Goal*`,`*?Set*`)`

Read this as "*Set* is the set of all instances of *Template* such that *Goal* is satisfied, where that set is non-empty". The term *Goal* specifies a goal or goals as in `call(`*Goal*`)` (see Section 4.4 [Control], page 51). *Set* is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 4.3 [Term Compare], page 49). If there are no instances of *Template* such that *Goal* is satisfied then the predicate fails.

The variables appearing in the term *Template* should not appear anywhere else in the clause except within the term *Goal*. 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 the list *Set* will only provide an imperfect representation of what is in reality an infinite set.

If there are uninstantiated variables in *Goal* which do not also appear in *Template*, then a call to this built-in predicate may backtrack, generating alternative values for *Set* corresponding to different instantiations of the free variables of *Goal*. (It is to cater for such usage that the set *Set* is constrained to be non-empty.) Two instantiations are different iff no renaming of variables can make them literally identical. For example, given the clauses:

```
likes(bill, cider).
likes(dick, beer).
likes(harry, beer).
likes(jan, cider).
likes(tom, beer).
likes(tom, cider).
```

the query

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

might produce two alternative solutions via backtracking:

```
Y = beer,   S = [dick, harry, tom]
Y = cider,  S = [bill, jan, tom]
```

The query:

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

would then produce:

```
SS = [ (beer,[dick,harry,tom]), (cider,[bill,jan,tom]) ]
```

Variables occurring in *Goal* will not be treated as free if they are explicitly bound within *Goal* 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:

```
S = [bill, dick, harry, jan, tom]
```

in contrast to the earlier example.

`bagof(`*?Template*`,`*:Goal*`,`*?Bag*`)`

This is exactly the same as `setof/3` except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. The effect of this relaxation is to save a call to `sort/2`, which is invoked by `setof/3` to return an ordered list.

?X`^`*:P* The interpreter recognizes this as meaning "there exists an *X* such that *P* is true", and treats it as equivalent to *P* (see Section 4.4 [Control], page 51). The use of this explicit existential quantifier outside the `setof/3` and `bagof/3` constructs is superfluous and discouraged.

`findall(`*?Template*`,`*:Goal*`,`*?Bag*`)`

Bag is a list of instances of *Template* in all proofs of *Goal* found by Prolog. The order of the list corresponds to the order in which the proofs are found. The list may be empty and all variables are taken as being existentially quantified. This means that each invocation of `findall/3` succeeds *exactly once*, and that no variables in *Goal* get bound. Avoiding the management of universally quantified variables can save considerable time and space.

`findall(`*?Template*`,`*:Goal*`,`*?Bag*`,`*?Remainder*`)`

Same as `findall/3`, except *Bag* is the list of solution instances appended to *Remainder*, which is typically unbound.

4.11 Foreign Language Functions Interface

Functions written in the C language (or any other language that uses the same calling conventions) may be called from Prolog using this interface. The foreign language functions interface is due to Quintus Prolog. Only atomic arguments and function values are allowed by this interface.

The interface can be used in two ways. With *dynamic linking*, foreign language modules may be linked in as needed. However: once a module has been linked in to the Prolog load image it is not possible to unlink the module. With *static linking*, foreign language modules are statically linked to the SICStus emulator, and only the binding of predicate names to functions needs to be done at runtime. In Runtime Systems, static linking is the only option available.

4.11.1 Interface Predicates

To perform either dynamic or static linking, the following predicates are used:

`foreign_file(+`*ObjectFile*`,+`*Functions*`)`

A user defined predicate. Specifies that a set of C language functions, to be called from Prolog, are to be found in *ObjectFile*. *Functions* is a list of functions exported by *ObjectFile*. Only functions that are to be called from Prolog should be listed. For example:

```
foreign_file('terminal.o', [scroll,pos_cursor,ask]).
```

specifies that functions `scroll()`, `pos_cursor()` and `ask()` are to be found in object file '`terminal.o`'.

`foreign(+`*CFunctionName*`, +`*Predicate*`)`
`foreign(+`*CFunctionName*`, +`*Language*`, +`*Predicate*`)`

User defined predicates. They specify the Prolog interface to a C function. *Language* is at present constrained to the atom `c`. *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 into the corresponding C arguments.

```
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);`.

`load_foreign_files(`*:ObjectFiles*`,+`*Libraries*`)`

Links *ObjectFiles* into the Prolog load image. *ObjectFiles* is a list of C object files. *Libraries* is a list of libraries. The C library `'-lc'` will always be used and need not be specified.

ObjectFiles may be prefixed by a module name (see Section 5.2 [Module Spec], page 83), in which case the calls to `foreign_files/2` and `foreign/2-3` are made in that module and the predicates defined are placed in it.

By default, dynamic linking is performed. If, however, the SICStus emulator has been statically linked with foreign language modules, static linking is performed and *Libraries* is simply ignored. Example:

```
| ?- load_foreign_files(['terminal.o'], []).
```

`prepare_foreign_files(+`*ObjectFiles*`)`

Generates the interface code necessary for *static* linking for the functions declared for the C object files specified as *ObjectFiles*. This must be a list of *all* the object files that are going to be used by `load_foreign_files/2`.

The interface code is written to '`flinkage.c`' (in the current working directory). This predicate is not available in Runtime Systems and is not needed at all for *dynamic* linking.

The third argument of the predicate `foreign/3` specifies how to translate between Prolog arguments and C arguments.

`Prolog: +integer`

`C: long` The argument should be instantiated to an integer or a float. The call will otherwise fail.

`Prolog: +float`

`C: double` The argument should be instantiated to an integer or a float. The call will otherwise fail.

`Prolog: +atom`

`C: unsigned long`

The argument should be instantiated to an atom. The call will otherwise fail. Each atom in SICStus is associated with a unique integer. This integer is passed as an unsigned long to the C function. Note that the mapping between atoms and integers depends on the execution history.

`Prolog: +string`

`C: char *` The argument should be instantiated to an atom. The call will otherwise fail. The C function will be passed the address of a text string containing the printed representation of the atom. The C function should *not* overwrite the string.

`Prolog: +string(`*N*`)`

`C: char *` The argument should be instantiated to an atom. The call will otherwise fail. The printable representation of the string will be copied into a newly allocated buffer. The string will be truncated if it is longer than *N* characters. The string will be blank padded on the right if it is shorter than *N* characters. The C function will be passed the address of the buffer. The C function may overwrite the buffer.

`Prolog: +address`

`C: char *` The argument should be instantiated to an integer; the call will otherwise fail. The argument should be either 0 or a pointer *P* previously passed from C to Prolog. The value passed will be `NULL` or *P*, respectively, type converted to `(char *)`.

`Prolog: +address(`*TypeName*`)`

`C:` *TypeName* `*`

The argument should be instantiated to an integer. The call will otherwise fail. The argument should be either 0 or a pointer *P* previously passed from C to Prolog. The value passed will be `NULL` or *P*, respectively, type converted to `(`*TypeName* `*)`.

`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. The Prolog integer will be unified with the Prolog argument.

`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. The Prolog float will be unified with the Prolog argument.

`Prolog: -atom`

`C: unsigned long *`

The C function is passed a reference to an uninitialized `long`. The value returned should have been obtained earlier from a `+atom` type argument. Prolog will attempt to associate an atom with the returned value. The atom will be unified with the Prolog argument.

`Prolog: -string`

`C: char **`

The C function is passed the address of an uninitialized `char *`. The returned string will be converted to a Prolog atom. The atom will be unified with the Prolog argument. C may reuse or destroy the string buffer during later calls.

`Prolog: -string(`*N*`)`

`C: char *` The C function is passed a reference to a character buffer large enough to store an N character string. The returned string will be stripped of trailing blanks and converted to a Prolog atom. The atom will be unified with the Prolog argument.

`Prolog: -address`

`C: char **`

The C function is passed the address of an uninitialized `char *`. The returned value, which must be `NULL` or a value created by `malloc()`, will be converted to a Prolog integer and unified with the Prolog argument.

`Prolog: -address(`*TypeName*`)`

`C:` *TypeName* `**`

The C function is passed the address of an uninitialized *TypeName* `*`. The returned value, which must be `NULL` or a value created by `malloc()`, will be converted to a Prolog integer and unified with the Prolog argument.

`Prolog: [-integer]`

`C: long` *F*`()`

The C function should return a `long`. The value returned will be converted to a Prolog integer. The Prolog integer will be unified with the Prolog argument.

`Prolog: [-float]`

`C: double` *F*`()`

The C function should return a `double`. The value returned will be converted to a Prolog float. The Prolog float will be unified with the Prolog argument.

`Prolog: [-atom]`

`C: unsigned long` *F*`()`

The C function should return an `unsigned long`. The value returned should have been obtained earlier from a `+atom` type argument. Prolog will attempt to associate an atom with the returned value. The atom will be unified with the Prolog argument.

`Prolog: [-string]`

C: char *F()

The C function should return a char *. The returned string will be converted to a Prolog atom. The atom will be unified with the Prolog argument. C may reuse or destroy the string buffer during later calls.

Prolog: [-string(N)]

C: char *F()

The C function should return a char *. The first N characters of the string will be copied and the copied string will be stripped of trailing blanks. The stripped string will be converted to a Prolog atom. The atom will be unified with the Prolog argument. C may reuse or destroy the string buffer during later calls.

Prolog: [-address]

C: char *F()

The C function should return a char *. The returned value, which must be NULL or a value created by malloc(), will be converted to a Prolog integer and unified with the Prolog argument.

Prolog: [-address(*TypeName*)]

C: *TypeName* *F()

The C function should return a *TypeName* *. The returned value, which must be NULL or a value created by malloc(), will be converted to a Prolog integer and unified with the Prolog argument.

Consider, for example, a function which returns the square root of its argument after checking that the argument is valid, defined in the file 'math.c':

```
#include <math.h>
#include <stdio.h>

double sqrt_check(d)
     double d;
{
  if (d < 0.0)
    d = 0.0,
    fprintf(stderr, "can't take square root of a negative number\n");

  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:

```
foreign_file('math.o', [sqrt_check]).

foreign(sqrt_check, c, sqrt(+float, [-float])).
```

```
:- load_foreign_files(['math.o'], ['-lm']).
```

A simple session using this function could be:

```
| ?- [math].
{consulting /home/sics/al/math.pl...}
{math consulted, 160 msec 597 bytes}

yes
| ?- sqrt(5.0, X).

X = 2.23606797749979 ?

yes
| ?- sqrt(-5.0, X).
can't take square root of a negative number

X = 0.0 ?

yes
```

4.11.2 Static Linking

Using static linking, the foreign language modules is statically linked with the emulator code and with *interface code*. This interface code is created by first loading into SICStus Prolog *all* `foreign_file/2` and `foreign/2-3` declarations that are going to be used by `load_foreign_files/2`, and then calling `prepare_foreign_files/1` as described above.

Once the interface code has been generated, the foreign language modules can be statically linked with the emulator. The whole procedure is best illustrated by an example. We first exctract the declarations into a file '`math2.pl`':

```
foreign_file('math.o', [sqrt_check]).

foreign(sqrt_check, c, sqrt(+float, [-float])).
```

and use them as follows:

```
% prolog
SICStus 2.1 #0 : Mon Oct 14 09:27:36 MET 1991
Copyright (C) 1988-1991 Swedish Institute of Computer Science.
All rights reserved.

| ?- [math2], prepare_foreign_files(['math.o']).

{consulting /home/sics/al/math.pl...}
```

```
{math consulted, 20 msec 597 bytes}
{flinkage.c generated, 20 msec}

yes
| ?- ^D
{ End of SICStus execution, user time 0.100 }
% cc -c flinkage.c
% setenv SP_PATH /usr/local/lib/sicstus2.1
% cc $SP_PATH/Emulator/sp.o flinkage.o math.o -lm -o sp
% ./sp -f -b $SP_PATH/Library/boot.ql
booting SICStus...please wait
SICStus 2.1 #0 : Mon Oct 14 09:27:36 MET 1991
Copyright (C) 1988-1991 Swedish Institute of Computer Science.
All rights reserved.

| ?- [math].

{consulting /home/sics/al/math.pl...}
{math consulted, 20 msec 597 bytes}

yes
| ?- sqrt(5.0, X).

X = 2.23606797749979 ?

yes
```

At this time, `save_program/1` can be called to create an executable saved state for quick start-up. See Chapter 10 [Installation Intro], page 139.

In general, to statically link the user code with the emulator, create the interface code ('`flinkage.o`') and issue a Shell command

```
% cc $SP_PATH/Emulator/sp.o flinkage.o OBJECTFILES LIBRARIES -lm LDFLAGS -o
sp
```

where the environment variable `SP_PATH` should be defined as the name of the SICStus source code directory ('`/usr/local/lib/sicstus2.1`' in the example), and `LDFLAGS` should have the same value as in `$SP_PATH/Makefile`.

4.12 Debugging

Debugging predicates are not available in Runtime Systems.

`unknown(`*?OldState*`,`*?NewState*`)`

Unifies *OldState* with the current state of the "Action on unknown predicates" flag, and sets the flag to *NewState*. This flag determines whether or not the system is to catch calls to undefined predicates (see Section 1.6 [Undefined Predicates], page 10). The possible states of the flag are:

`trace` Causes calls to undefined predicates to be reported and the debugging system to be entered at the earliest opportunity (the default state).

`fail` Causes calls to such predicates to fail.

`error` Causes calls to such predicates to raise an exception. See Section 4.5 [Exception], page 53.

`debug` The debugger is switched on with tracing disabled. See Section 2.2 [Basic], page 17.

`nodebug`

`notrace` The debugger is switched off. See Section 2.2 [Basic], page 17.

`trace` The debugger is switched on with tracing enabled. See Section 2.3 [Trace], page 18.

`leash(+`*Mode*`)`

Leashing Mode is set to *Mode*. See Section 2.3 [Trace], page 18.

`spy` *:Spec* Spy-points are placed on all the predicates given by *Spec*. See Section 2.4 [Spy-Point], page 18.

`nospy` *:Spec*

Spy-points are removed from all the predicates given by *Spec*. See Section 2.4 [Spy-Point], page 18.

`nospyall` This removes all the spy-points that have been set.

`debugging`

Displays information about the debugger. See Section 2.2 [Basic], page 17.

4.13 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.

The profiling package was written by M.M. Gorlick and C.F. Kesselman at the Aerospace Corporation (*Timing Prolog Programs Without Clocks*, Proc. Symposium on Logic Programming, pp. 426–432, IEEE Computer Society, 1987).

Profiling is not available in Runtime Systems.

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 directive:

```
| ?- 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 compilation mode 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.

`profile_data(`*:Files*`,`*?Selection*`,`*?Resolution*`,`*-Data*`)`

This unifies *Data* with profiling data collected from the predicates defined in *Files*. *Files* should be either a single filename or a list of, possibly module prefixed, filenames.

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.*

`backtracks`

All instances of entering a clause by backtracking are counted.

`choice_points`

All instances of creating a choicepoint are counted. This occurs, roughly, when the implementation determines that there are more than one possibly matching clauses for a procedure call, and when a disjunction is entered.

`shallow_fails`

All instances of backtracking "early" in a clause or disjunct when there are outstanding alternatives for the current procedure call are counted.

`deep_fails`

All instances of backtracking "late" in a clause or disjunct, or when there are no outstanding alternatives for the current procedure call, are counted. The reason for distinguishing shallow and deep failures is that the former are considerably cheaper to execute than the latter.

`execution_time`

The execution time for the selected predicates, clauses, or disjuncts is estimated in artificial units.

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 *Module:PredName*`-`*Count*, where *Count* is a sum of the corresponding counts per clause.

`clause` *Data* is a list of *Module:ClauseName*`-`*Count*, where *Count* includes counts for any disjunctions occurring inside that clause. Note, however, that the selections `calls` and `backtracks` do *not* include counts for disjunctions.

`all` *Data* is a list of *Module:InternalName*`-`*Count*. This is the finest resolution level, counting individual clauses and disjuncts.

Above, *PredName* is a predicate spec, *ClauseName* is a compound term *PredName*`/`*ClauseNumber*, and *InternalName* is either
ClauseName—corresponding to a clause, or
`(`*ClauseName*`-`*DisjNo*`)/`*Arity*`/`*AltNo*—corresponding to a disjunct.

`profile_reset(`*:Files*`)`

Zeroes all counters for predicates defined in *Files*, which should be either a single filename or a list of filenames, similar to the argument accepted by `profile_data/4`.

4.14 Definite Clause Grammars

Prolog's grammar rules provide a convenient notation for expressing definite clause grammars, see *Les Grammaires de Metamorphos* by A. Colmerauer, Technical Report, Groupe d'Intelligence Artificielle, Marseille-Luminy, November, 1975, and *Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks* by F.C.N. Pereira and D.H.D. Warren, in *Artificial Intelligence* 13:231-278, 1980.

Definite clause grammars are an extension of the well-known context-free grammars. 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 '`,`'.

Definite clause grammars extend context-free grammars in the following ways:

1. A non-terminal symbol may be any Prolog term (other than a variable or number).
2. 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 '`""`'.
3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in '`{}`' brackets.
4. 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).

5. Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator ';' or '|' as in Prolog.
6. 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 '{}' brackets.

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
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 some digit.

The query

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

will compute Z=14. The two extra arguments are explained below.

Now, in fact, grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. We now show how to translate grammar rules into ordinary clauses by making explicit the extra arguments.

A rule such as

```
p(X) --> q(X).
```

translates into

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

If there is more than one non-terminal on the right-hand side, as in

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

then corresponding input and output arguments are identified, as in

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

Terminals are translated using the built-in predicate `'C'(`*S1*`,` *X*`,` *S2*`)`, read as "point *S1* is connected by terminal *X* to point *S2*", and defined by the single clause

```
'C'([X|S], X, S).
```

(This predicate is not normally useful in itself; it has been given the name upper-case `c` simply to avoid using up a more useful name.) Then, for instance

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

is translated by

```
p(X, S0, S) :-
        'C'(S0, go, S1),
        'C'(S1, to, S2),
        q(X, S2, S3),
        'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves, e.g.

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X, S0, S) :-
        'C'(S0, X, S1),
        integer(X),
        X>0,
        q(X, S1, S).
```

Similarly, a cut is translated literally.

Terminals on the left-hand side of a rule translate into an explicit list in the output argument of the main non-terminal, e.g.

```
is(N), [not] --> [aint].
```

becomes

```
is(N, S0, [not|S]) :- 'C'(S0, aint, S).
```

Disjunction has a fairly obvious translation, e.g.

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

translates to

```
args(X, Y, S0, S) :-
        (   dir(X, S0, S1),
            'C'(S1, to, S2),
            indir(Y, S2, S)
        ;   indir(Y, S0, S1),
            dir(X, S1, S)
        ).
```

The built-in predicates which are concerned with grammars are as follows.

`expand_term(+`*Term1*`,?`*Term2*`)`

When a program is read in, some of the terms read are transformed before being stored as clauses. If *Term1* is a term that can be transformed, *Term2* is the result. Otherwise *Term2* is just *Term1* unchanged. This transformation takes place automatically when grammar rules are read in, but sometimes it is useful to be able to perform it explicitly. Grammar rule expansion is not the only transformation available; the user may define clauses for the predicate `term_expansion/2` to perform other transformations. `term_expansion(`*Term1*`,`*Term2*`)` is called first, and only if it fails is the standard expansion used.

`term_expansion(+`*Term1*`,?`*Term2*`)`

A user defined predicate, which overrides the default grammar rule expansion of clauses to be consulted or compiled.

`phrase(`*:Phrase*`,?`*List*`)`
`phrase(`*:Phrase*`,?`*List*`,?`*Remainder*`)`

The list *List* is a phrase of type *Phrase* (according to the current grammar rules), where *Phrase* is either a non-terminal or more generally a grammar rule body. *Remainder* is what remains of the list after a phrase has been found. If called with 2 arguments, the remainder has to be the empty list.

`'C'(`*?S1*`,?`*Terminal*`,?`*S2*`)`

Not normally of direct use to the user, this built-in predicate is used in the expansion of grammar rules (see above). It is defined as if by the clause `'C'([X|S], X, S).`

4.15 Miscellaneous

?X `=` *?Y* Defined as if by the clause `Z=Z.`; i.e. *X* and *Y* are unified.

`length(`*?List*`,?`*Length*`)`

If *List* is instantiated to a list of determinate length, then *Length* will be unified with this length.

If *List* is of indeterminate length and *Length* is instantiated to an integer, then *List* will be unified with a list of length *Length*. The list elements are unique variables.

If *Length* is unbound then *Length* will be unified with all possible lengths of *List*.

`prolog_flag(+`*FlagName*`,?`*OldValue*`,?`*NewValue*`)`

Unify *OldValue* with the value of the flag *FlagName*, then set the value of *FlagName* to *NewValue*. The possible flag names and values are:

`compiling`

Governs the mode in which `compile/1` and `fcompile/1` operate (see Chapter 3 [Load Intro], page 25).

`compactcode`

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

`fastcode` Compilation produces native machine instructions. Only available for Sun-3, Sun-4, and NeXT computers.

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

`debugcode`
Replace compiling with consulting.

`debugging`
Corresponds to the predicates `debug/0`, `nodebug/0`, `trace/0`, `notrace/0` (see Section 4.12 [Debug Pred], page 68).

`trace` Turn on trace mode.

`debug` Turn on the debugger.

`off` Turn off trace mode and the debugger (the default).

(This flag is not available in Runtime Systems.)

`fileerrors`
`on` or `off`. Turn raising of exception on file errors on or off. Equivalent to `fileerrors/0` and `fileerrors/0`, respectively (see Section 4.1.4 [Stream Pred], page 42). Initially `on`.

`gc` `on` or `off`. Turn garbage collection on or off. Initially `on`.

`gc_margin`
Margin: Number of kilobytes. If less than *Margin* kilobytes are reclaimed in a garbage collection then the size of the garbage collected area should be increased. Also, no garbage collection is attempted unless the garbage collected area has at least *Margin* kilobytes. Initially 500.

`gc_trace` Governs 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).

`redefine_warnings`
`on` or `off`. Enable or disable warning messages when a predicate is being redefined from a different file than its previous definition. Initially `on`. (This warning is always disabled in Runtime Systems.)

`single_var_warnings`
`on` or `off`. Enable or disable warning messages when a clause containing non-anonymous variables occurring once only is compiled or consulted. Initially `on`.

`unknown` Corresponds to the predicate `unknown/2` (see Section 4.12 [Debug Pred], page 68).

`trace` Cause calls to undefined predicates to be reported and the debugging system to be entered at the earliest opportunity. (This setting is not possible in Runtime Systems.)

`fail` Cause calls to such predicates to fail.

`error` Causes calls to such predicates to raise an exception (the default). See Section 4.5 [Exception], page 53.

`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.

`fail` The read quietly fails.

`error` An exception is raised. See Section 4.5 [Exception], page 53 (the default).

`prolog_flag(+`*FlagName*`,?`*OldValue*`)`

This is a shorthand for

`prolog_flag(`*FlagName*`,`*OldValue*`,`*OldValue*`)`

`copy_term(?`*Term*`,?`*CopyOfTerm*`)`

CopyOfTerm is an independent copy of *Term*, with new variables substituted for all variables in *Term*. It is defined as if by

```
copy_term(X, Y) :-
        assert('copy of'(X)),
        retract('copy of'(Y)).
```

`numbervars(?`*Term*`,+`*N*`,?`*M*`)`

Unifies each of the variables in term *Term* with a special term, so that `write(`*Term*`)` (or `writeq(`*Term*`)`) (see Section 4.1.2 [Term IO], page 36) prints those variables as (`A` + (i mod 26))(i/26) where i ranges from N to M-1. N must be instantiated to an integer. If it is 0 you get the variable names A, B, ..., Z, A1, B1, etc. This predicate is used by `listing/0, listing/1` (see Section 4.6 [State Info], page 54).

`setarg(+`*ArgNo*`,+`*CompoundTerm*`,?`*NewArg*`)`

Replace destructively argument *ArgNo* in *CompoundTerm* by *NewArg*. The assignment is undone on backtracking. **This operation is only safe if there is no further use of the "old" value of the replaced argument**. The use of this predicate is discouraged, as the idea of destructive replacement is alien to logic programming.

`undo(:`*Term*`)`

The goal `call(`*Term*`)` (see Section 4.4 [Control], page 51) is executed on backtracking.

`halt` Causes an irreversible exit from Prolog back to the shell. (This predicate will only terminate the innermost call from C in Runtime Systems.)

`op(+`*Precedence*`,+`*Type*`,+`*Name*`)`

Declares the atom *Name* to be an operator of the stated *Type* and *Precedence* (see Section 8.6 [Operators], page 120). *Name* may also be a list of atoms in which case all of them are declared to be operators. If *Precedence* is 0 then the operator properties of *Name* (if any) are cancelled.

`current_op(?`*Precedence*`,?`*Type*`,?`*Op*`)`

The atom *Op* is currently an operator of type *Type* and precedence *Precedence*. Neither *Op* nor the other arguments need be instantiated at the time of the call; i.e. this predicate can be used to generate as well as to test.

`break` Invokes the Prolog interpreter recursively. See Section 1.9 [Nested], page 12. (This predicate is not available in Runtime Systems.)

`query_expansion(+`*RawQuery*`,?`*Query*`)`
A *user defined predicate*, which may be used to transform queries entered at the terminal in response to the '`| ?- `' prompt. The Prolog interpreter will call this for every top-level query *RawQuery*. If it succeeds, *Query* will be executed instead of *RawQuery*, but the variable bindings will be printed as usual upon completion. This feature is useful e.g. to implement a simple command interpreter.

`abort` Aborts the current execution. See Section 1.9 [Nested], page 12. (This predicate will only terminate the innermost call from C in Runtime Systems.)

`save(+`*File*`)`
The system saves the current state of the system into file *File*. When it is restored, Prolog will resume execution that called `save/1`. See Section 1.10 [Saving], page 13. (This predicate is not available in Runtime Systems.)

`save(+`*File*`,?`*Return*`)`
Saves the current system state in *File* just as `save(`*File*`)`, but in addition unifies *Return* to 0 or 1 depending on whether the return from the call occurs in the original incarnation of the state or through a call `restore(`*File*`)` (respectively). (This predicate is not available in Runtime Systems.)

`save_program(+`*File*`)`
The system saves the currently defined predicates into file *File*. When it is restored, Prolog will reinitialize itself. See Section 1.10 [Saving], page 13. (This predicate is not available in Runtime Systems.)

`restore(+`*File*`)`
The system is returned to the system state previously saved to file *File*. See Section 1.10 [Saving], page 13. (This predicate is not available in Runtime Systems.)

`reinitialise`
This predicate can be used to force the initialization behavior to take place at any time. When SICStus is initialized it looks for a file '`~/.sicstusrc`' and consults it, if it exists. (This predicate will only terminate the innermost call from C in Runtime Systems.)

`maxdepth(+`*Depth*`)`
The positive integer *Depth* specifies the maximum depth, i.e. the maximum number of nested interpreted calls, beyond which the interpreter will trap to the debugger. The top level has zero depth. This is useful for guarding against loops in an untested program, or for curtailing infinite execution branches. Note that calls to compiled predicates are not included in the computation of the depth. The interpreter will check for maximum depth only if the debugger is switched on. (This predicate is not available in Runtime Systems.)

`depth(?`*Depth*`)`
Unifies *Depth* with the current depth, i.e. the number of currently active interpreted procedure calls. Depth information is only available when the debugger is switched on. (This predicate is not available in Runtime Systems.)

`garbage_collect`
: Perform a garbage collection of the global stack immediately.

`gc`
: Enables garbage collection of the global stack (the default).

`nogc`
: Disables garbage collection of the global stack.

`statistics`
: Display on the terminal statistics relating to memory usage, run time, garbage collection of the global stack and stack shifts.

`statistics(`*?Key*`,`*?Value*`)`
: This allows a program to gather various execution statistics. For each of the possible keys *Key*, *Value* is unified with a list of values, as follows:

  `global_stack`
  : `[`*size used*`,`*free*`]`
  This refers to the global stack, where compound terms are stored.

  `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 UNIX process.

  `heap`
  `program`
  : `[`*size used*`,0]`
  These refer to the amount of memory allocated for compiled and interpreted clauses, symbol tables, and the like.

  `runtime`
  : `[`*since start of Prolog*`,`*since previous statistics*`]`

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

  `stack_shifts`
  : `[`*no. of local shifts*`,`*no. of trail shifts*`,`*time spent*`]`

  Times are in milliseconds, sizes of areas in bytes.

`prompt(`*?Old*`,`*?New*`)`
: The sequence of characters (prompt) which indicates that the system is waiting for user input is represented as an atom, and unified with *Old*; the atom bound to New specifies the new prompt. In particular, the goal `prompt(X, X)` unifies the current prompt with *X*, without changing it. Note that this predicate only affects the prompt given when a

user's program is trying to read from the terminal (e.g. by calling `read/1`). Note also that the prompt is reset to the default '`|: `' on return to top-level.

`version`
Displays the introductory messages for all the component parts of the current system.

Prolog will display its own introductory message when initially run but not normally at any time after this. If this message is required at some other time it can be obtained using this predicate which displays a list of introductory messages; initially this list comprises only one message (Prolog's), but you can add more messages using `version/1`.

`version(+`*Message*`)`
This takes a message, in the form of an atom, as its argument and appends it to the end of the message list which is output by `version/0`.

The idea of this message list is that, as systems are constructed on top of other systems, each can add its own identification to the message list. Thus `version/0` should always indicate which modules make up a particular package. It is not possible to remove messages from the list.

`help`
Displays basic information, or a user defined help message. It first calls `user_help/0`, and only if that call fails is a default help message printed on the current output stream.

`user_help`
A user defined predicate. This may be defined by the user to print a help message on the current output stream.

`unix(+`*Term*`)`
`plsys(+`*Term*`)`
Allows certain interactions with the operating system. Under UNIX the possible forms of *Term* are as follows:

`argv(`*?Args*`)`
Args is unified with a list of ATOMS of the program arguments supplied when the current SICStus process was started. For example, if SICStus were invoked with

```
% prolog hello world 2001
```

then *Args* will be unified with `[hello,world,'2001']`.

`cd(+`*Path*`)`
Change the current working directory to *Path*.

`cd`
Change the current working directory to the home directory.

`shell`
Start a new interactive UNIX shell named in the Unix environment variable `$SHELL`. The control is returned to Prolog upon termination of the shell.

`shell(+`*Command*`)`
Pass *Command* to a new UNIX shell named in the Unix environment variable `$SHELL` for execution.

`shell(+`*Command*`,-`*Status*`)`
Pass *Command* to a new UNIX shell named in the Unix environment variable `$SHELL` for execution. Unify *Status* with the returned status of *Command*.

`system(+`*Command*`)`

Pass *Command* to a new UNIX `sh` process for execution.

`system(+`*Command*`,-`*Status*`)`

Pass *Command* to a new UNIX `sh` process for execution. Unify *Status* with the returned status of *Command*.

`exit(+`*Status*`)`

Terminate the Prolog process with the status *Status*.

`mktemp(+`*Template*`,-`*FileName*`)`

Interface to the C-function `mktemp(3)`. A unique file name is created and unified with *FileName*. *Template* should contain a file name with six trailing `X`s. The file name is that template with the six `X`s replaced with a letter and the process id.

`access(+`*Path*`,+`*Mode*`)`

Tests if *Mode* is the accessability of *Path* as in the C-function `access(2)`.

`chmod(+`*Path*`,-`*Old*`,+`*New*`)`

Old is the current access mode of *Path*; *New* is the new access mode.

`umask(-`*Old*`,+`*New*`)`

Old is the current file mode creation mask; *New* is the new file mode creation mask.

5 The Module System

By making use of the module systems facilities, programs can be divided into different modules. Each module has its own independent predicate name space. This is an important feature for the development of larger programs. The module system of Sicstus Prolog is procedure based. This means that only the predicates are local to a module, whereas terms are global. The module system is flat, not hierarchical, so all modules are visible to one another. It is modeled after the Quintus Prolog module system.

5.1 Basic Concepts

Each predicate in the Prolog system, whether built-in or user defined, belongs to a module. A predicate is generally only visible in the module where it is defined. However a predicate may be *imported* by another module. It is thereby made visible in that module too. Predicates declared as public in a module declaration (see below) are *exported*. Normally only exported predicates may be imported.

There are two predefined modules, the `prolog` module and the `user` module. All the built-in predicates reside in the `prolog` module. The exported built-in predicates are automatically imported into each new module as it is created.

The `user` module is the default module where all user defined predicates go if there is no module specification. Thus the system may be used without the user being aware of the module system at all.

5.2 Module Specification

The module of a predicate call may be specified by prefixing the goal with the module name and the colon operator:

```
| ?- foo:bar(X).
```

When there is no explicit module specification, all goals appearing in files refer to the *source module*. Goals issued at the top level refer to the *type-in* module. The type-in module is by default the `user` module but may be changed by the directive:

```
| ?- module(ModuleName).
```

The source module of a file is either the module specified in a module declaration or, for non module files, the type-in module.

5.3 Defining Modules

A module may be defined by putting a module declaration first in a source file.

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

When the file is loaded, all predicates in the file go to the stated module and the predicates of the `PublicPredicateList` are exported. When a module declaration is processed, all existing predicates in the module are erased before the new ones are loaded. A file which contains a module declaration is henceforth called a *module file*.

A module can also be defined dynamically by asserting or loading predicates to it:

```
| ?- assert(m:p(x)).
```

creates the module `m`, if it does not already exists, and asserts `p(x)` to it.

```
| ?- compile(m:f).
```

creates the module `m` and loads `f` into `m`.

Dynamically created modules have no public predicates.

5.4 Importation

When a module file is loaded by one of the predicates `consult/1`, `compile/1`, `ensure_loaded/1`, or `load/1` (See Chapter 3 [Load Intro], page 25), all the exported predicates of the module file are imported by the receiving module. In addition the built-in predicates `use_module/1-3` are available (see Section 5.9 [Mod Builtin], page 87).

Clashes with already existing predicates, local or imported from other modules, are handled in two different ways: If the receiving module is the `user` module, the user is asked for redefinition of the predicate. For other receiving modules, a warning is issued and the importation is canceled. The binding of an imported predicate remains, even if the origin is reloaded or deleted. However, `abolish/1` breaks up the importation binding. When a module file is reloaded, a check is made that the predicates imported by other modules are still in the public list. If that is not the case, a warning is issued. Note that an imported predicate may be re-exported.

There is no error checking of module handling in Runtime Systems.

5.5 Meta Expansion

Some predicates take goals as arguments (i.e. meta predicates). These arguments must include a module specification stating which module the goal refers. Some other predicates also need module information i.e. `compile/1`. The property of needing module information is declared with a *meta-declaration* (see Section 5.6 [Meta Decl], page 86). Goals for these predicates are module-name-expanded to ensure the module information. Goals issued at toplevel and goals appearing as directives are expanded prior to execution while goals in the bodies of clauses are expanded at compile time. The expansion is made by preceding the relevant argument with '*Module*:'. If the goal is prefixed by '*Module*:', *Module* is used for the expansion, otherwise the source/type-in module is used. An argument is not expanded if:

1. It already has a module prefix.
2. It is a variable which appears in a meta expandable position in the head of the clause.

Some examples:

```
| ?- [user].
| :- meta_predicate p(:), q(:).
| r(X) :- p(X).
| q(X) :- p(X).
| user consulted, 40 msec 1088 bytes

yes
| ?- listing.

r(A) :-
        p(user:A).

q(A) :-
        p(A).

yes
```

Here, `p/1` and `q/1` are declared as meta predicates while `r/1` is not. Thus the clause `r(X) :- p(X)` will be transformed to `r(X) :- p(`M`:X)`, by item 2 above, where M is the type-in module, whereas `q(X) :- p(X)` will not.

```
| ?- m:assert(f(1)).
```

Here, `assert/1` is called in the module `m`. However that does not ensure that `f(1)` is asserted into `m`. The fact that `assert/1` is a meta predicate makes the system module-name-expand the goal, transforming it to `m:assert(m:f(1))` before execution. That way `assert/1` is supplied the right module info.

5.6 Meta-Predicate Declarations

The fact that a predicate needs meta expansion is declared in the following way:

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

which means that the first argument of `p/2` shall be expanded. Instead of the '`:`', an integer also means expansion while anything else, like '`+`', '`-`' or '`?`' for example, means no expansion. A number of built-in predicates have predefined meta-predicate declarations, as indicated by the mode specs in this manual.

5.7 Considerations for file-to-file compilation

Since the module name expansion takes place at compile time, the module into which the file is to be loaded must be known when compiling to object files. This is no problem for module files because the module name is picked from the module declaration. When non-module files are compiled, the file name may be prefixed with the module name that is to be used for expansion.

```
| ?- fcompile(Module:Files).
```

If an object file is loaded into a different module from which it was compiled for, a warning is issued. There is a limitation on the use of meta predicates for object file compilation. The only meta-predicate declarations visible to the compiled file are the ones in the same file and the built-in ones. Meta predicates which are used in the file but defined in other files are not expanded.

5.8 Modules and the Emacs interface

When code is loaded through the Emacs interface, the module into which the loading goes is the same as the module into which the file was originally loaded. If the file has not been loaded before, it goes into the type-in module. If the type-in module is not the user module, the user is asked if the loading should take place. When a piece of a module file which contains the module declaration is loaded, the behavior is the same as when doing a compile/consult except that no importation takes place. Note that when a module declaration is processed, all previous definitions in the module are erased. It will normally not be meaningful to include the module declaration except when the whole file is loaded.

5.9 Built-in Predicates for Modules

`use_module(`*:Files*`)`

Loads *Files* like `ensure_loaded/1` and imports all public predicates. (In Runtime Systems, `use_module/1` is equivalent to `load/1`.)

`use_module(`*:File*`, +`*PredicateList*`)`

Loads *File* like `ensure_loaded/1` and imports the predicates in *PredicateList*. If any of these are not exported, a warning is issued. (In Runtime Systems, `use_module/2` always loads '`.ql`' files and prints no progress messages.)

`use_module(`*?Module*`, `*:File*`, +`*PredicateList*`)`

If *Module* is uninstantiated or not previously loaded into the system, this works exactly like `use_module/2`. If *Module* is already in the system then only importation takes place and *File* is not used. (In Runtime Systems, `use_module/3` is equivalent to `use_module/2`.)

`current_module(`*?Module*`)`

Module is a module in the system. It can be used to backtrack through all modules present in the system.

`current_module(`*?Module*`, `*?File*`)`

Module is the module defined in *File*. (This predicate is not available in Runtime Systems.)

`module(+`*Module*`)`

The type-in module is set to *Module*. See Section 5.1 [Basic Concepts], page 83. (This predicate is not available in Runtime Systems.)

6 Coroutining Facilities

The computation rule in traditional Prolog systems is 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 is *blocked* if certain arguments are uninstantiated and its predicate definition is annotated with a matching *block declaration*. Goals of some built-in predicates, documented in this chapter, may also be blocked if their arguments are not instantiated enough.

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

6.1 Block Declarations

The declaration

`:- block` *BlockSpec*`, ...,` *BlockSpec*`.`

where each *BlockSpec* is a mode spec, specifies conditions for blocking goals of the predicate referred to by the mode spec (`f/3` say). 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` 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.

Block declarations generalise the "wait declarations" of earlier versions of SICStus Prolog. A declaration '`:- wait f/3`' in the old syntax corresponds to '`:- block f(-,?,?)`' in the current syntax. See Section 9.6 [Use Of Term Exp], page 134 for a simple way to extend the system to accept the old syntax.

6.2 Built-in Coroutine Predicates

The coroutining facility can also be accessed by a number of built-in predicates. This makes it possible to use coroutines in a more dynamic way, without having to rely on block declarations:

`when(+`*Condition*`,:`*Goal*`)`

Block *Goal* until the *Condition* is true, where *Condition* is a Prolog goal with the restricted syntax:

`nonvar(`*X*`)`

`ground(`*X*`)`

`?=(`*X*`,`*Y*`)`

Condition`,`*Condition*

Condition`;`*Condition*

For example.

```
| ?- when(((nonvar(X);?=(X,Y)),ground(T)), process(X,Y,T)).
```

`freeze(`*?X*`,:`*Goal*`)`

Block *Goal* until `nonvar(`*X*`)` (see Section 4.7 [Meta Logic], page 56) holds. This is defined as if by:

```
freeze(X, Goal) :- when(nonvar(X), Goal).
```

or

```
:- block freeze(-, ?).
freeze(_, Goal) :- Goal.
```

`frozen(-`*Var*`,?`*Goal*`)`

If some goal is blocked on the variable *Var*, then that goal is unified with *Goal*. Otherwise, *Goal* is unified with the atom `true`. Fails if *Var* is not currently a variable.

`dif(`*?X*`,`*?Y*`)`

Constrains *X* and *Y* to represent different terms i.e. to be non-unifiable. 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).
```

`call_residue(`*:Goal*`,`*?Residue*`)`

The *Goal* is executed as if by `call/1`. If after the execution there are still some subgoals of *Goal* that are blocked on some variables, then *Residue* is unified with a list of such *Variable-Goal* pairs. Otherwise, *Residue* is unified with the empty list `[]`.

6.3 Boolean Constraint Solver

The other major coroutining facility is a solver for constraints over the Boolean domain, i.e. the values 0 and 1. This domain is particularly useful for modeling digital circuits, and the constraint solver can be used for verification, design, optimization etc. of such circuits.

The solver contains predicates for checking the consistency and entailment of a constraint w.r.t. previous constraints, and for computing particular solutions to the set of previous constraints.

The underlying representation of Boolean functions is based on Boolean Decision Diagrams (*Graph-Based Lagorithms for Boolean Function Manipulation* by R.E. Bryant, IEEE Trans. on Computers, August, 1986). This representation is very efficient, and allows many combinatorial problems to be solved with good performance.

Boolean expressions are composed from the following operands: the constants 0 and 1 (`FALSE` and *TRUE*), logical variables, and symbolic constants, and from the following connectives. *P* and *Q* are Boolean expressions, *X* is a logical variable, *Is* is a list of integers or integer ranges, and *Es* is a list of Boolean expressions:

`~` *P*
: True if *P* is false.

P `*` *Q*
: True if *P* and *Q* are both true.

P `+` *Q*
: True if at least one of *P* and *Q* is true.

P `#` *Q*
: True if exactly one of *P* and *Q* is true.

X `^` *P*
: True if there exists an *X* such that *P* is true. Same as *P*`[`*X*`/0] + `*P*`[`*X*`/1]`.

P `=:=` *Q*
: Same as `~`*P* `#` *Q*.

P `=\=` *Q*
: Same as *P* `#` *Q*.

P `=<` *Q*
: Same as `~`*P* `+` *Q*.

P `>=` *Q*
: Same as *P* `+` `~`*Q*.

P `<` *Q*
: Same as `~`*P* `*` *Q*.

P `>` *Q*
: Same as *P* `*` `~`*Q*.

`card(`*Is*`,`*Es*`)`
: True if the number of true expressions in *Es* is a member of the set denoted by *Is*.

Symbolic constants (Prolog atoms) denote parametric values and can be viewed as all-quantified variables whose quantifiers are placed outside the entire expression. They are useful for forcing certain variables of an equation to be treated as input parameters.

Built-in predicates:

`bool:sat(+`*Expression*`)`

: *Expression* is a Boolean expression. This checks the consistency of the expression w.r.t. the accumulated constraints, and, if the check succeeds, *tells* the constraint that the expression be true.

  If a variable *X*, occurring in the expression, is subsequently unified with some term *T*, this is treated as a shorthand for the constraint

```
?- bool:sat(X=:=T).
```

`bool:taut(+`*Expression*`, ?`*Truth*`)`

: *Expression* is a Boolean expression. This *asks* whether the expression is now entailed by the accumulated constraints (*Truth*=1), or whether its negation is entailed by the accumulated constraints (*Truth*=0). Otherwise, it fails.

`bool:labeling(+`*Variables*`)`

: *Variables* is a list of variables. The variables are instantiated to a list of 0s and 1s, in a way that satisfies any accumulated constraints. Enumerates all solutions by backtracking, but creates choicepoints only if necessary.

Some examples:

```
| ?- bool:sat(X + Y).

bool:sat(Y=\=_A*X#X) ?
```

illustrates three facts. First, any accumulated constraints affecting the top-level variables are displayed as floundered goals, since the query is not true for all `X` and `Y`. Secondly, accumulated constraints are displayed as `bool:sat(`*V*`=:=`*Expr*`)` or `bool:sat(`*V*`=\=`*Expr*`)` where *V* is a variable and *Expr* is a "polynomial", i.e. an exclusive or of conjunctions of variables and constants. Thirdly, `_A` had to be introduced as an artificial variable, since `Y` cannot be expressed as a function of `X`. That is, `X + Y` is true iff there exists an `_A` such that `Y=\=_A*X#X`. Let's check it!

```
| ?- bool:taut(_A ^ (Y=\=_A*X#X) =:= X + Y, T).

T = 1 ?
```

verifies the above answer. Notice that the formula in this query is a tautology, and so it is entailed by an empty set of constraints.

```
| ?- bool:taut(A =< C, T).

no
| ?- bool:sat(A =< B), bool:sat(B =< C), bool:taut(A =< C, T).

T = 1,
bool:sat(B=:=_B*_A*C#_B*C#_A*C),
bool:sat(A=:=_A*C) ?

| ?- bool:taut(a, T).

T = 0 ?

yes
| ?- bool:taut(~a, T).

T = 0 ?
```

illustrates the entailment predicate. In the first query, the expression "A implies C" is neither known to be true nor false, so the query fails. In the second query, the system is told that "A implies B" and "B implies C", so "A implies C" is entailed. The expressions in the third and fourth queries are to be read "for each a, a is true" and "for each a, a is false", respectively, and so `T = 0` in both cases since both are unsatisfiable. This illustrates the fact that the implicit universal quantifiers introduced by symbolic constants are placed in front of the entire expression.

```
| ?- [user].
| adder(X, Y, Sum, Cin, Cout) :-
      bool:sat(Sum =:= card([1,3],[X,Y,Cin])),
      bool:sat(Cout =:= card([2-3],[X,Y,Cin])).
| user consulted, 40 msec 576 bytes

yes
| ?- adder(x, y, Sum, cin, Cout).

bool:sat(Sum=:=cin#x#y),
bool:sat(Cout=:=x*cin#x*y#y*cin) ?

yes
| ?- adder(x, y, Sum, 0, Cout).

bool:sat(Sum=:=x#y),
bool:sat(Cout=:=x*y) ?

yes
| ?- adder(X, Y, 0, Cin, 1), bool:labeling([X,Y,Cin]).

Cin = 0,
X = 1,
Y = 1 ? ;

Cin = 1,
X = 0,
Y = 1 ? ;
```

```
Cin = 1,
X = 1,
Y = 0 ? ;
```

illustrates the use of cardinality constraints and models a one-bit adder circuit. The first query illustrates how representing the input signals by symbolic constants forces the output signals to be displayed as functions of the inputs and not vice versa. The second query computes the simplified functions obtained by setting carry-in to 0. The third query asks for particular input values satisfying sum and carry-out being 0 and 1, respectively.

7 Generating Runtime Systems

It is possible to mix C and Prolog code to create stand-alone applications, *Runtime Systems*. The *Runtime Kernel* (described in this chapter) gives you functions to control the Prolog execution and to convert between C and Prolog data. You may also write deterministic Prolog predicates in C.

To build a Runtime System you need:

'`Runtime/runtime.o`'
: The *Runtime Kernel* contains the SICStus Prolog engine together with control and interface functions.

'`Runtime/runtime.h`'
: The macros and declarations for the interface functions.

'`Runtime/flinkage.c`'
: Default interface code for the foreign language functions interface (see Section 4.11.2 [Foreign], page 67).

Your program code
: Your C code, object files and Prolog code.

SICStus
: You must compile your Prolog program with the predicate `fcompile/1` in your ordinary SICStus system. This will produce the Prolog object files that you load into the Runtime System at runtime.

To run a Runtime System on a target machine you need:

'`Runtime/*.ql`'
: The *Runtime Library* is the built-in predicates written in Prolog. This code is loaded into the program at runtime by the function `SP_initialize()`. These files must be located in a directory '`$SP_PATH/Runtime/`' on the target machine. The environment variable `SP_PATH` should contain an appropriate pathname.

Your binary program
: This is your C program linked with the Runtime Kernel.

Your Prolog object code
: This is your Prolog program compiled with `fcompile/1`. This code is loaded at runtime from C with the function `SP_load()`.

7.1 Calling Prolog from C

In Runtime Systems there is no automatic conversion of arguments in a call to Prolog. You don't write any declarations in the Prolog code. The arguments to a call of a predicate from C have the type `SP_term *`. Conversion of the C data to and from this type is done explicitly in the C code. This is a small example of C code to call a predicate `foo/1` which is known to return an atom:

```
#include "Runtime/runtime.h"

main()
{
  SP_term arg1;
  SP_pred_ref pred;

  if (SP_initialize())
  {
    SP_put_variable(&arg1);
    pred = SP_predicate("current_host",1,"");
    if (pred && SP_query(pred,&arg1))
    {
      char *text;
      SP_get_atom(arg1, &text);
      printf("Proved to be: %s\n",text);
    }
  }
}
```

7.2 Control of Execution

The Runtime Kernel is available in the usual development system as well as in Runtime Systems. The Runtime Kernel functions for calling Prolog from C and their support functions can be called from the main program of a Runtime System, from predicates defined with `SP_install_c_predicate()` (see Section 7.6 [Calling C], page 104), and asynchronously via `SP_event()` (see below). They *cannot* be called via predicates defined with the foreign language function interface (see Section 4.11.2 [Foreign], page 67).

7.2.1 Preparing a Call to Prolog from C

You must call `SP_initialize()` before any other calls in a Runtime System. The function `SP_initialize()` will allocate data areas used by Prolog and load the Runtime Library.

This function will look for the environment variable `SP_PATH` to find the directory '`Runtime/`' that contains the Runtime Library. If `SP_PATH` is undefined it will assume the path is '`./`' and look for '`./Runtime/`':

```
int SP_initialize()
```

You can load your Prolog code compiled to Prolog object files into the system with the call `SP_load()`. This is the C equivalent of the Prolog predicate `load/1`:

```
int SP_load(ql_file)
     char *ql_file;
```

Before calling a predicate you have 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 defined. This definition could be used in more than one call to the same predicate. The module specification is optional. If `NULL` (the null pointer) or `""` (the empty string) is given then the default `user` module is assumed:

```
SP_pred_ref SP_predicate(name_string, arity, module_string)
     char *name_string;
     long int arity;
     char *module_string;
```

7.2.2 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, and try to prove it. The call will return 1 if the goal succeeded, 0 if it failed, and -1 if it raised an uncaught exception. Note that the arguments to the predicate are given by reference:

```
int SP_query(predicate, arg1, arg2, ...)
     SP_pred_ref predicate;
     SP_term *arg1, *arg2, ...;
```

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 finitely fail. This will reclaim the storage used after the call:

```
int SP_query_cut_fail(predicate, arg1, arg2, ...)
     SP_pred_ref predicate;
     SP_term *arg1, *arg2, ...;
```

7.2.3 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, or `NULL`, if given an invalid predicate reference. Note that if a new query is opened while another is already open, the new query must be terminated before performing any action on the old one. That is, queries must be strictly nested:

```
SP_qid SP_open_query(predicate, arg1, arg2, ...)
     SP_pred_ref predicate;
     SP_term *arg1, *arg2, ...;
```

The function `SP_next_solution()` will cause the Prolog engine to find solutions of the open query. The `SP_term *` variables that you sent with the call to `SP_open_query()` will be rebound to new values. `SP_next_solution()` will return 1 for success and 0 for failure or invalid usage, and -1 for uncaught exception. The C variable `errno` will be set to zero for failure and to a non-zero value for invalid usage:

```
int SP_next_solution(query)
     SP_qid query;
```

You can terminate a query in two ways. The function `SP_cut_query()` will only take away the choices created since the corresponding `SP_open_query()`. The data created in the call are still valid and could be passed to another call. The function `SP_close_query()` will restore the state to what it was before the call to `SP_open_query()`. If the call created data that you want to keep, it must be converted to C data before calling `SP_close_query()`. Both functions return 1 for success and 0 for invalid usage:

```
int SP_cut_query(query)
     SP_qid query;

int SP_close_query(query)
     SP_qid query;
```

7.2.4 Calling Prolog Asynchronously

A Prolog execution may be interrupted by UNIX signals. If you wish to call Prolog back from a signal handler you can not use `SP_query()` etc. directly. The call to Prolog has to be delayed until a time when the Prolog execution can accept an interrupt. The function `SP_event()` serves this purpose, and installs the function `func` to be called from Prolog when the execution can accept a callback. Returns 0 if installation failed. No queue of functions is maintained; that is, from the point when `SP_event()` was called until `func` is invoked, further calls to `SP_event()` will merely fail.

Depending on the value returned from `func`, the interrupted Prolog execution will just continue (value=1), backtrack (value=0), or process an exception (value=-1). These values are the ones returned from the functions for calling Prolog from C:

```
int SP_event(func)
     int (*func)();
```

A UNIX signal handler having called `SP_event()` should call `SP_continue()` as its last action, to ensure that the interrupt is processed as early as possible:

```
void SP_continue()
```

The following piece of C code illustrates these facilities. The function `signal_init()` installs the function `signal_handler()` as the primary UNIX signal handler for the signals `SIGUSR1` and `SIGUSR2`. That function invokes the predicate `prolog_handler/1` as the actual signal handler, passing the signal number as an argument to the predicate.

```
SP_pred_ref event_pred;
int signal_no;

static int signal_event()
{
  SP_term x;

  SP_put_integer(&x, signal_no);
  return SP_query(event_pred, &x);
}

static void signal_handler(sig)
int sig;
{
  SP_event(signal_event);
  signal_no = sig;
  SP_continue();
}
```

```
signal_init()
{
  event_pred = SP_predicate("prolog_handler",1,"");

  signal(SIGUSR1, signal_handler);
  signal(SIGUSR2, signal_handler);
}
```

7.3 Support Functions

There is a set of functions to convert data between the basic C types and the Prolog data type term. There is also a set of routines to test if a term can be converted to a specific C type and there is a function to unify two terms.

7.3.1 Creating Prolog Terms

`void SP_put_variable(SP_term *t)`
: Creates a new variable.

`void SP_put_integer(SP_term *t, long l)`
: Creates a Prolog integer from a C long integer.

`void SP_put_float(SP_term *t, double d)`
: Creates a Prolog float from a C double.

`void SP_put_atom(SP_term *t, char *name)`
: Creates an atom from a C string.

`void SP_put_compound(SP_term *t, char *name, int arity)`
: Creates a compound term with all the arguments unbound.

`void SP_put_arg(int i, SP_term t, SP_term arg)`
: Fills in the i'th argument of a compound term.

7.3.2 Memory Allocation

It is possible that the C functions to create Prolog data will overflow the Prolog data areas. If you create more than a few small terms in C you should call a special allocation function `SP_get_memory()` prior to any allocation, to ensure that sufficient memory is available. The arguments are, except for the expected amount of memory, the addresses of the pointers to previously created structures. This will allow `SP_get_memory()` to update these pointers if the terms pointed out are moved by the storage handler. The argument list is terminated by `NULL` (a null pointer).

`int SP_get_memory(int amount, SP_term *t1, ..., SP_term *tn, NULL)`

Allocates *amount* words of memory on the Prolog heap, updating the terms *t1*, ..., *tn* if necessary.

You must calculate the amount of memory needed yourself. Calculations are made in heap words. The *Prolog heap* is where most Prolog data are kept.

variable Variables take one word. A variable is created with `SP_put_variable()` and may be created by a call to `SP_unify()` or `SP_compare()`.

integer Small integers, from -2**25 to 2**25-1, will not occupy any memory but larger integers (up to 32 bits) take three words.

float A floating point number takes four words.

atom Atoms do not occupy any heap memory.

list Lists take two words.

structure A structure takes one word plus one word for each argument.

calls to Prolog

When a call to a Prolog predicate *Name/Arity* is made, at least 1154+2**Arity* heap words must be available.

7.3.3 Getting C Data from Prolog Terms

These functions will take a term and convert it to C data. They return zero if the conversion fails, and a nonzero value otherwise.

`int SP_get_integer(SP_term t, long *l)`

Gets the C integer from a Prolog term.

`int SP_get_float(SP_term t, double *d)`

Gets the C float from a Prolog term.

`int SP_get_atom(SP_term t, char **name)`

Gets a pointer to the string that is the name of this atom. This string must *not* be modified. The actual string pointer is the same for all occurrences of this atom, so to check whether two atoms are identical it suffices to compare the string pointers.

`int SP_get_compound(SP_term t, char **name, int *arity)`

Gets the C name and arity from the principal functor of a Prolog compound term. The string must *not* be modified. The actual string pointer is the same for all occurrences of this functor, so to check whether two functors are identical it suffices to compare the string pointers and arities.

`int SP_get_arg(int i, SP_term t, SP_term *arg)`

Gets the i'th argument of a Prolog compound term.

`int SP_term_type(SP_term t)`

Returns one of `SP_TYPE_VARIABLE`, `SP_TYPE_INTEGER`, `SP_TYPE_FLOAT`, `SP_TYPE_ATOM`, or `SP_TYPE_COMPOUND`, depending on the type of the term *t*.

7.3.4 Other Functions

`int SP_unify(SP_term x, SP_term y)`

Unifies two terms, returning zero on failure and nonzero on success.

`int SP_compare(SP_term x, SP_term y)`

Returns `-1` if `x @< y`, `0` if `x == y` and `1` if `x @> y`

7.4 Life Time of Terms

It is important to understand the lifetime of Prolog data in Runtime Systems. Since we use pointers from C into the Prolog data areas, there are occasions when the data pointed at are changed or even made invalid. The following rules should be followed.

- All arguments to a Prolog query must be terms passed by reference. A garbage collection done by the Prolog engine can move terms in memory and the interface has to update your pointers. This is an example of an invalid program:

```
{
  SP_pred_ref pred = SP_predicate("foo",1,"");
  SP_term arg1, p;

  SP_put_variable(&arg1);
  p = arg1;                      /* Copy the pointer */

  if (pred && SP_query(pred,&arg1))
  {
    char *text;
    SP_get_atom(p, text);        /* WRONG!!! p might be invalid */
    printf("%s\n",text);
  }
}
```

- Always terminate a call initiated with `SP_open_query()`. A call that failed is already terminated but you could call `SP_cut_query()` or `SP_close_query()` anyway. A call that didn't fail must be terminated. Undefined behavior will result if you don't.

7.5 An Example

This is an example of how to create a Runtime System. First a small Prolog program 'train.pl' that will give you a route from one train station to another. Compile this program in SICStus with fcompile(train):

```
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).
```

Now a C program 'train.c' that will call the Prolog code and write out all the routes found between two stations:

```
#include "Runtime/runtime.h"

main()
{
  SP_pred_ref pred;
  SP_qid goal;
  SP_term from,to,way;

  if (SP_initialize()
      && SP_load("train.ql")
      && (pred = SP_predicate("connected",4,"")))
  {
    SP_put_atom(&from, "Stockholm");
    SP_put_atom(&to, "Orebro");
    SP_put_variable(&way);

    if (goal = SP_open_query(pred,&from,&to,&way,&way))
    {
      while (SP_next_solution(goal)) write_way(way);
      SP_cut_query(goal);
    }
```

```
  }
}

write_way(way)
     SP_term way;
{
  SP_term from;

  printf("\n");
  while (SP_term_type(way)==SP_TYPE_COMPOUND)
  {
    char *text;

    SP_get_arg(1, way, &from);
    SP_get_arg(2, way, &way);
    SP_get_atom(from, &text);
    printf("Path: %s\n",text);
  }
}
```

Finally, link the program with the Prolog engine and C interface routines and run it:

```
% cc -o train train.c Runtime/flinkage.c Runtime/runtime.o -lm; train

Path: Stockholm
Path: Katrineholm
Path: Hallsberg
Path: Kumla
Path: Orebro

Path: Stockholm
Path: Vasteras
Path: Orebro
```

7.6 Calling C from Prolog

There are two ways of calling predicates written in C, with the foreign language functions interface (see Section 4.11.2 [Foreign], page 67) or with the Runtime Kernel interface.

Using the Runtime Kernel interface, you can write the predicates together with your other C sources and call a function `SP_install_c_predicate()` at runtime to make them visible to the Prolog engine:

```
SP_pred_ref SP_install_c_predicate(pname,arity,pmodule,procedure)
     char *pname;
     int arity;
     char *pmodule;
     int (*procedure)();
```

where the `procedure` should return 1 for success, 0 for failure, and -1 for an uncaught exception. Its arguments are all of type `SP_term`.

It is possible to call Prolog from your predicates written in C. There is no restriction on the level of recursive calls to the interface.

In this example, the predicate `foo/1` (written in Prolog) and the predicate `bar/1` (written in C) are mutually recursive. The main program performs the necessary initialization and issues the initial call to `foo/1`:

```
foo(0).
foo(N) :-
  N1 is N - 1,
  write(N),
  bar(N1).
```

```
#include "Runtime/runtime.h"

SP_pred_ref pred;

int bar(x)
     SP_term x;
{
  return SP_query(pred,&x);
}

main()
{
  if (SP_initialize())
  {
    SP_term r;

    SP_install_c_predicate("bar",1,"",bar);

    SP_load("foo.ql");
    pred = SP_predicate("foo",1,"");

    SP_put_integer(&r,5);

    SP_query(pred,&r);
  }
}
```

7.7 Linking and Error Handling

To create a Runtime System you have to link the Runtime Kernel (‘`Runtime/runtime.o`’) with your own code. Unfortunately, the Runtime Kernel contains symbols which may clash with symbols used in your code. At present, the only way to solve this problem is to rename the symbols in the user code.

The ordinary foreign language function interface can be used in Runtime Systems with static linking (for a more detailed description see Section 4.11.2 [Foreign], page 67). To use it, interface code must first be created by loading into SICStus Prolog *all* `foreign_file/2` and `foreign/2-3` declarations that are going to be used by `load_foreign_files/2`, and then calling the predicate `prepare_foreign_files/1`. A file '`flinkage.c`' is generated with interface code for the relevant predicates. This code then replaces the default '`Runtime/flinkage.c`'.

There is no safe error handling in a Runtime System. All interface functions assume that the arguments are valid.

The Prolog engine does not perform any signal handling in Runtime Systems, except as installed by the `SP_event()` mechanism.

8 The Prolog Language

This chapter provides a brief introduction to the syntax and semantics of a certain subset of logic (*definite clauses*, also known as *Horn clauses*), and indicates how this subset forms the basis of Prolog.

8.1 Syntax, Terminology and Informal Semantics

8.1.1 Terms

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

The constants include *integers* such as

```
0   1   999   -512
```

Besides the usual decimal, or base 10, notation, integers may also be written in any base from 2 to 36, of which base 2 (binary), 8 (octal), and 16 (hex) are probably the most useful. Letters `A` through ! `Z` (upper or lower case) are used for bases greater than 10. E.g.

```
15   2'1111   8'17
```

all represent the integer fifteen.

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

```
0'A
```

is equivalent to `65` (the numerical value of the character code for `A`).

Constants also include *floats* such as

```
1.0   -3.141   4.5E7   -0.12e+8   12.0e-9
```

Note that there must be a decimal point in floats written with an exponent, and that there must be at least one digit before and after the decimal point.

Constants also include *atoms* such as

```
a   void   =   :=   'Algol-68'   []
```

Constants are definite elementary objects, and correspond to proper nouns in natural language. For reference purposes, here is a list of the possible forms which an atom may take:

1. Any sequence of alphanumeric characters (including `_`), starting with a lower case letter.
2. Any sequence from the following set of characters:
   `+-*/\^<>=`~:.?@#$&`
   This set can in fact be larger; see Section 8.9.4 [Token String], page 127 for a precise definition.
3. Any sequence of characters delimited by single quotes. If the single quote character is included in the sequence it must be written twice, e.g. `'can''t'`.
4. Any of: `! ; [] {}`
   Note that the bracket pairs are special: `[]` and `{}` are atoms but `[`, `]`, `{`, and `}` are not. However, when they are used as functors (see below) the form `{`*X*`}` is allowed as an alternative to `{}(`*X*`)`. The form `[`*X*`]` is the normal notation for lists, as an alternative to `.(X,[])`.

Variables may be written as any sequence of alphanumeric characters (including `_`) starting with either a capital letter or `_`; e.g.

```
X   Value   A   A1   _3   _RESULT
```

If a variable is only referred to once in a clause, it does not need to be named and may be written as an *anonymous* variable, indicated by the underline character `_`. A clause may contain several anonymous variables; they are all read and treated as distinct 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 writeable storage location as in most programming languages; rather it is a local name for some data object, cf. the variable of pure LISP and identity declarations in Algol68.

The structured data objects of the language are the 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 functor is named `point` of arity 3, with arguments `X`, `Y` and `Z`, is written

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

Note that an atom is considered to be a functor of arity 0.

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 term

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

would be pictured as the structure

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

Sometimes it is convenient to write certain functors as operators—2-ary functors may be declared as infix operators and 1-ary functors as prefix or postfix operators. Thus it is possible to write, e.g.

```
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 below (see Section 8.6 [Operators], page 120).

Lists form an important class of data structures in Prolog. They are essentially the same as the lists of LISP: a list either is the atom [] representing the empty list, or is a compound term with functor . and two arguments which are respectively the head and tail of the 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

```
.(1,.(2,.(3,[])))
```

but which is normally written, in a special list notation, as

```
[1,2,3]
```

The special list notation in the case when the tail of a list is a variable is exemplified by

```
[X|L]     [a,b|L]
```

representing

```
     .               .
    / \             / \
   X   L           a   .
                      / \
                     b   L
```

respectively.

Note that this notation does not add any new power to the language; it simply makes it more readable. E.g. the above examples could equally be written

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

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

```
"SICStus"
```

which represents exactly the same list as

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

8.1.2 Programs

A fundamental unit of a logic program is the *goal* or procedure call. E.g.

```
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 identifies what *predicate* the goal is for. 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 of 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 (i.e. it too 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

P.

where P is the head goal. We interpret this declaratively as

Goals matching P are true.

and procedurally as

Goals matching P are satisfied.

If the body of a clause is non-empty, the clause is called a *non-unit clause*, and is written in the form

P :- *Q*, *R*, *S*.

where P is the head goal and Q, R and S are the goals which make up the body. We can read such a clause either declaratively as

> P is true if Q and R and S are true.

or procedurally as

> To satisfy goal P, satisfy goals Q, R and S.

A sentence with an empty head is called a *directive* (see Section 1.4 [Directives], page 7), of which the most important kind is called a *query* and is written in the form

```
?- P, Q.
```

where P and Q are the goals of the body. Such a query is read declaratively as

> Are P and Q true?

and procedurally as

> Satisfy goals P and Q.

Sentences generally contain variables. Note that variables in different sentences are completely independent, even if they have the same name—i.e. the *lexical scope* of a variable is limited to a single sentence. Each distinct variable in a sentence should be interpreted as standing for an arbitrary entity, or value. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

1. `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."
2. `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."

3. `?- ungulate(`X`), aquatic(`X`).`

   "Is it true, for any X, that X is an ungulate and X is aquatic?"

   "Find an X which is both an ungulate and aquatic."

In any program, the *predicate* for a particular (principal) functor is the sequence of clauses in the program whose head goals have that principal functor. For example, the predicate for a 3-ary functor `concatenate/3` might well consist of the two clauses

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

where `concatenate(`*L1*`,`*L2*`,`*L3*`)` means "the list *L1* concatenated with the list *L2* is the list *L3*". Note that for predicates with clauses corresponding to a base case and a recursive case, the preferred style is to write the base case clause first.

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; e.g. `concatenate/3`.

Certain predicates are predefined by built-in predicates supplied by the Prolog system. Such predicates are called *built-in predicates*.

As we have seen, the goals in the body of a sentence are linked by the operator '`,`' which can be interpreted as conjunction ("and"). 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

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

which can be read as

> 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.

Such uses of disjunction can always be eliminated by defining an extra predicate—for instance the previous example is equivalent to

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

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

—and so disjunction will not be mentioned further in the following, more formal, description of the semantics of clauses.

The token '|', when used outside a list, is an alias for ';'. The aliasing is performed when terms are read in, so that

```
a :- b | c.
```

is read as if it were

```
a :- b ; c.
```

Note the double use of the '.' character. On the one hand it is used as a sentence terminator, while on the other it may be used in a string of symbols which make up an atom (e.g. the list functor `./2`). The rule used to disambiguate terms is that a '.' followed by a *layout-char* is regarded as a sentence terminator (see Section 8.9.4 [Token String], page 127).

8.2 Declarative 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 preceding procedure for `concatenate/3`, then the declarative semantics tells us that

```
?- concatenate([a], [b], [a,b]).
```

is true, because this goal is the head of a certain instance of the first clause for `concatenate/3`, namely,

```
concatenate([a], [b], [a,b]) :- concatenate([], [b], [b]).
```

and we know that the only goal in the body of this clause instance is true, since it is an instance of the unit clause which is the second clause for `concatenate/3`.

8.3 Procedural Semantics

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* which 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 the program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics. We first illustrate the semantics by the simple query

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

We find that it matches the head of the first clause for `concatenate/3`, with *X* instantiated to `[a|`*X1*`]`. The new variable *X1* is constrained by the new query produced, which contains a single recursive procedure call:

```
?- concatenate(X1, Y, [b]).
```

Again this goal matches the first clause, instantiating *X1* to `[b|`*X2*`]`, and yielding the new query:

```
?- concatenate(X2, Y, [])
```

Now the single goal will only match the second clause, instantiating both *X2* and *Y* to `[]`. Since there are no further goals to be executed, we have a solution

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

i.e. a true instance of the original goal is

```
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, against the second clause for concatenate, goals already solved once using the first 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 *unification* algorithm (see *A Machine-Oriented Logic Based on the Resolution Principle* by J.A. Robinson, *Journal of the ACM* 12:23-44, January 1965) 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 simple too: “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, then 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 *floundered*. The interpreter top-level checks for this condition. If detected, the outstanding blocked subgoals are printed on the terminal 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.

In compiled code, the computation rule is not completely obeyed, as calls to certain built-in predicates compile to instructions. Such calls are executed even if a unification just prior to the call causes a blocked goal to become unblocked. The following built-in predicates do not compile to procedure calls in compiled code. Note also that there is an implicit cut in the `\+` and `->` constructs:

```
'C'/3
arg/3
atom/1
atomic/1
compare/3
float/1
functor/3
is/2
integer/1
nonvar/1
number/1
var/1
'=='/2 '\=='/2 '@<'/2 '@>='/2 '@>'/2 '@=<'/2
'=:='/2 '=\='/2 '<'/2 '>='/2 '>'/2 '=<'/2
'=..'/2 '='/2 ','/2 !/0
```

Sometimes, it is crucial that the blocked goal be executed before a call to one of the above built-in predicates. Since most of the above are meta-logical primitives, their semantics can depend on whether a variable is currently bound etc. Consider, for example, the clauses and query

```
:- block test(-).
test(2).

data(1).
data(2).

?- test(X), data(X), !, ...
```

thus the first match for `data(X)` causes the blocked goal `test(X)` to be unblocked, but since the cut is selected before `test(X)`, the system is committed to the first match for `data(X)`, and the query fails. However, inserting a dummy goal `true` enables the unblocked goal to be selected before the cut:

```
?- test(X), data(X), true, !, ...
```

As `test(1)` fails, the system backtracks to the second clause for `data(X)`, and the query succeeds with the answer

```
X = 2
```

See Chapter 6 [Blocking], page 89 for a description of the available facilities involving blocked goals.

8.4 Occur Check

It is possible, and sometimes useful, to write programs which unify a variable to a term in which that variable occurs, thus creating a cyclic term. The usual mathematical theory behind Logic Programming forbids the creation of cyclic terms, dictating that an *occur check* should be done each time a variable is unified with a term. Unfortunately, an occur check would so expensive as to render Prolog impractical as a programming language. Thus cyclic terms may be created and may cause loops trying to print them.

SICStus Prolog mitigates the problem by its ability to unify and compare (see Section 4.3 [Term Compare], page 49) cyclic terms without looping. Loops in the printer can be interrupted by typing ^C.

8.5 The Cut Symbol

Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the *cut* symbol, 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 symbol 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. that goal which 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 which were executed during the execution of those preceding goals.

For example:

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

This predicate can be used to test whether a given term is in a list. E.g.

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

returns the answer '`yes`'. The predicate 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|_]) :- !.
```

In this case, the above call would extract only the first element of the list (`d`). On backtracking, the cut would immediately fail the whole predicate.

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

This is equivalent to

```
x := if p then q else r;
```

in an Algol-like language.

It should be noticed that a cut discards all the alternatives since 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. We would like to advise all users to follow these general rules. Also see Chapter 9 [Example Intro], page 131.

- Write each clause as a self-contained logic rule which just defines the truth of goals which match its head. Then add cuts to remove any fruitless alternative computation paths that may tie up store.
- Cuts are usually placed right after the head, sometimes preceded by simple tests.
- Cuts are hardly ever needed in the last clause of a predicate.

8.6 Operators

Operators in Prolog are simply a *notational convenience*. For example, the expression `2+1` could also be written `+(2,1)`. This expression 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 predicate such as `is/2` (see Section 4.2 [Arithmetic], page 46).

The Prolog syntax caters for operators of three main 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 is written after its single argument.

Each operator has a precedence, which is a number from 1 to 1200. The precedence is used to disambiguate expressions where the structure of the term denoted is not made explicit through the use of parentheses. The general rule is that it is the operator with the *highest* precedence that 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, i.e.

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

If there are two operators in the subexpression 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 a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself, i.e. their principal functors must be of lower precedence, unless the subexpression is explicitly parenthesized (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.

A functor named name is declared as an operator of type *Type* and precedence precedence by the command

```
:- op(precedence, type, name).
```

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, i.e. infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to operators which are provided as standard. Declarations of all the standard operators can be found elsewhere (see Chapter 12 [Standard Operators], page 153).

For example, the standard operators `+` and `-` are declared by

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

so that

```
a-b+c
```

is valid syntax, and means

```
(a-b)+c
```

i.e.

```
       +
     /   \
    -     c
   / \
  a   b
```

The list functor . is not a standard operator, but if we declare it thus:

```
:- op(900, xfy, .).
```

then `a.b.c` would represent the structure

```
    .
   / \
  a   .
     / \
    b   c
```

Contrasting this with the diagram above for `a-b+c` shows the difference betweeen `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.

Note that the arguments of a compound term written in standard syntax must be expressions of precedence *below* 1000. Thus it is necessary to parenthesize the expression `P :- Q` in

```
| ?- assert((P :- Q)).
```

8.7 Syntax Restrictions

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

1. In a term written in standard syntax, the principal functor and its following `(` must *not* be separated by any intervening spaces, newlines etc. Thus

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

   is invalid syntax.
2. If the argument of a prefix operator starts with a `(`, this `(` must be separated from the operator by at least one space or other non-printable 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 below 1000, this interpretation would fail as soon as the ';' (precedence 1100) was encountered.

In contrast, the term:

```
:- (p;q),r.
```

is valid syntax and represents the following structure.

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

8.8 Comments

Comments have no effect on the execution of a program, but they are very useful for making programs more readily comprehensible. Two forms of comment are allowed in Prolog:

1. The character `%` followed by any sequence of characters up to end of line.
2. The symbol `/*` followed by any sequence of characters (including new lines) up to `*/`.

8.9 Full Prolog Syntax

A Prolog program consists of a sequence of *sentences* or lists of sentences. Each sentence is a Prolog *term*. How terms are interpreted as sentences is defined below (see Section 8.9.2 [Sentence], page 125). Note that a term representing a sentence may be written in any of its equivalent syntactic forms. For example, the 2-ary functor ':-' 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 brackets and commas.

We define below how lists of tokens are interpreted as terms (see Section 8.9.3 [Term Token], page 126). Each list of tokens which is read in (for interpretation as a term or sentence) has to be terminated by a full-stop token. Two tokens must be separated by a space token if they could otherwise be interpreted as a single token. Both space tokens and comment tokens 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).

We define below defines how tokens are represented as strings of characters (see Section 8.9.4 [Token String], page 127). But we start by describing the notation used in the formal definition of Prolog syntax (see Section 8.9.1 [Syntax Notation], page 125).

8.9.1 Notation

1. Syntactic categories (or *non-terminals*) are written thus: *item*. Depending on the section, a category may represent a class of either terms, token lists, or character strings.
2. 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*, *F3*, etc.
3. Certain definitions and restrictions are given in ordinary English, enclosed in `{ }` brackets.
4. A category written as *C...* denotes a sequence of one or more *C*s.
5. A category written as *?C* denotes an optional *C*. Therefore *?C...* denotes a sequence of zero or more *C*s.
6. A few syntactic categories have names with arguments, and rules in which they appear may contain meta-variables looking thus: *X*. The meaning of such rules should be clear from analogy with the definite clause grammars (see Section 4.14 [Definite], page 71).
7. In the section describing the syntax of terms and tokens (see Section 8.9.3 [Term Token], page 126) particular tokens of the category name are written thus: *name*, while tokens which are individual punctuation characters are written literally.

8.9.2 Syntax of Sentences as Terms

sentence `-->` *clause* `|` *directive* `|` *grammar-rule*

clause `-->` *non-unit-clause* `|` *unit-clause*

directive `-->` *command* `|` *query*

non-unit-clause `-->` *head* `:-` *goals*

unit-clause `-->` *head*
`{` where *head* is not otherwise a *sentence* `}`

command `-->` `:-` *goals*

query `-->` `?-` *goals*

```
head            --> term
                        { where term is not a number or variable }

goals           --> goals , goals
                  | goals -> goals ; goals
                  | goals -> goals
                  | \+ goals
                  | goals ; goals
                  | goal

goal            --> term
                        { where term is not a number
                and is not otherwise a goals }

grammar-rule    --> gr-head --> gr-body

gr-head         --> non-terminal
                  | non-terminal , terminals

gr-body         --> gr-body , gr-body
                  | gr-body -> gr-body ; gr-body
                  | gr-body -> gr-body
                  | \+ gr-body
                  | gr-body ; gr-body
                  | non-terminal
                  | terminals
                  | gr-condition

non-terminal    --> term
                        { where term is not a number or variable
                and is not otherwise a gr-body }

terminals       --> list | string

gr-condition    --> { goals }
```

8.9.3 Syntax of Terms as Tokens

```
term-read-in    --> subterm(1200) full-stop

subterm(N)      --> term(M)
                        { where M is less than or equal to N }

term(N)         --> op(N,fx) subterm(N-1)
                        { except the case - number }
                        { if subterm starts with a (,
                op must be followed by a space }
                  | op(N,fy) subterm(N)
                        { if subterm starts with a (,
                          op must be followed by a space }
                  | subterm(N-1) op(N,xfx) subterm(N-1)
                  | subterm(N-1) op(N,xfy) subterm(N)
                  | subterm(N) op(N,yfx) subterm(N-1)
                  | subterm(N-1) op(N,xf)
```

```
                    |   subterm(N) op(N,yf)

term(1000)         --> subterm(999) , subterm(1000)

term(0)            --> functor ( arguments )
                          { provided there is no space between
                            the functor and the ( }
                    |   ( subterm(1200) )
                    |   { subterm(1200) }
                    |   list
                    |   string
                    |   constant
                    |   variable

op(N,T)            --> name
                          { where name has been declared as an
                            operator of type T and precedence N }

arguments          --> subterm(999)
                    |   subterm(999) , arguments

list               --> []
                    |   [ listexpr ]

listexpr           --> subterm(999)
                    |   subterm(999) , listexpr
                    |   subterm(999) | subterm(999)

constant           --> atom | number

number             --> integer | float

atom               --> name

integer            --> natural-number
                    |   - natural-number

float              --> unsigned-float
                    |   - unsigned-float

functor            --> name
```

8.9.4 Syntax of Tokens as Character Strings

By default, SICStus uses the ISO 8859/1 character set standard, but will alternatively support the EUC (Extended UNIX Code) standard. This is governed by the value of the environment variable `LC_CTYPE` (see Chapter 10 [Installation Intro], page 139).

The character categories used below are defined as follows in the two standards:

layout-char

In ISO 8859/1, these are character codes 0..32 and 127..159. In EUC, these are character codes 0..32 and 127. The common subset includes characters such as `TAB`, `LFD`, and `SPC`.

small-letter

In ISO 8859/1, these are character codes 97..122, 223..246, and 248..255. In EUC, these are character codes 97..122 and 128..255. The common subset are the letters `a` through `z`.

capital-letter

In ISO 8859/1, these are character codes 65..90, 192..214, and 216..222. In EUC, these are character codes 65..90. The common subset are the letters `A` through `Z`.

digit In both standards, these are character codes 48..57, i.e. the digits `0` through `9`.

symbol-char

In ISO 8859/1, these are character codes 35, 36, 38, 42, 43, 45..47, 58, 60..64, 92, 94, 96, 126, 160..191, 215, and 247. In EUC, these are character codes 35, 36, 38, 42, 43, 45..47, 58, 60..64, 92, 94, 96, and 126. The common subset is `+-*/\^<>=`~:.?@#$&`.

solo-char In both standards, these are character codes 33 and 59 i.e. the characters `!` and `;`.

punctuation-char

In both standards, these are character codes 37, 40, 41, 44, 91, 93, and 123..125, i.e. the characters `%(),[]{|}`.

quote-char

In both standards, these are character codes 34 and 39 i.e. the characters `"` and `'`.

underline In both standards, this is character code 95 i.e. the character `_`.

```
token            --> name
                   | natural-number
                   | unsigned-float
                   | variable
                   | string
                   | punctuation-char
                   | space
                   | comment
                   | full-stop

name             --> quoted-name
                   | word
                   | symbol
                   | solo-char
                   | [ ?layout-char... ]
                   | { ?layout-char... }

quoted-name      --> ' ?quoted-item... '

quoted-item      --> char  { other than ' }
                   | ''

word             --> small-letter ?alpha...
```

```
symbol          --> symbol-char...
                       { except in the case of a full-stop
                         or where the first 2 chars are /* }

natural-number  --> digit...
                  |  base ' alpha...
                     { where each alpha must be less than the base,
                        treating a,b,... and A,B,... as 10,11,... }
                  |  0 ' char
                     { yielding the character code for char }

base            --> digit...   { in the range [2..36] }

unsigned-float  --> simple-float
                  |  simple-float  exp  exponent

simple-float    --> digit... . digit...

exp             --> e  |  E

exponent        --> digit... | - digit... | + digit...

variable        --> underline  ?alpha...
                  |  capital-letter  ?alpha...

string          --> " ?string-item... "

string-item     --> char   { other than " }
                  |  ""

space           --> layout-char...

comment         --> /* ?char... */
                       { where ?char... must not contain */ }
                  |  % ?not-end-of-line...  newline

not-end-of-line --> { any character except newline }

newline         --> { LFD }

full-stop       --> . layout-char

char            --> { any character, i.e. }
                     layout-char
                  |  alpha
                  |  symbol-char
                  |  solo-char
                  |  punctuation-char
                  |  quote-char

alpha           --> capital-letter | small-letter | digit | underline
```

8.9.5 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 which has the 1-ary functor `-` as its principal functor.
5. The character `"` within a string must be written duplicated. Similarly for the character `'` within a quoted atom.
6. 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.
7. 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 Programming Examples

Some simple examples of Prolog programming are given below. They exemplify typical applications of Prolog. We are trying to convey a flavor of Prolog programming style as well, by following the simple rules:

- Base case before recursive cases.
- Input arguments before output arguments.
- Use cuts sparingly, and *only* at proper places (see Section 8.5 [Cut], page 119). 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.
- Use disjunctions sparingly, *always* put parentheses around them, *never* put parentheses around the individual disjuncts, *never* put the '`;`' at the end of a line.

The code herein was derived in part from shared code written by by R.A. O'Keefe.

9.1 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).
```

9.2 A Small Database

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
```

9.3 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` (see Section 4.3 [Term Compare], page 49) 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).
```

The goal `put_assoc(`*Key*`, `*OldAssoc*`, `*Val*`, `*NewAssoc*`)` is true when *OldAssoc* and *NewAssoc* define the same mapping for all keys other than *Key*, and `get_assoc(`*Key*`, `*NewAssoc*`, `*Val*`)` is true.

```
put_assoc(Key, t, Val, Tree) :- !, Tree = t(Key,Val,t,t).
put_assoc(Key, t(K,V,L,R), Val, New) :-
        compare(Rel, Key, K),
        put_assoc(Rel, Key, K, V, L, R, Val, New).

put_assoc(=, Key, _, _, L, R, Val, t(Key,Val,L,R)).
put_assoc(<, Key, K, V, L, R, Val, t(K,V,Tree,R)) :-
        put_assoc(Key, L, Val, Tree).
put_assoc(>, Key, K, V, L, R, Val, t(K,V,L,Tree)) :-
        put_assoc(Key, R, Val, Tree).
```

9.4 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*.

```
:- op(300, xfy, **).

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).
```

9.5 Use of Meta-Logical Predicates

This example illustrates the use of the meta-logical predicates `var/1`, `arg/3`, and `functor/3` (see Section 4.7 [Meta Logic], page 56). The procedure call `variables(`*Term*`, `*L*`, [])` instantiates variable *L* to a list of all the variable occurrences in the term *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) :-
        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).
```

9.6 Use of term_expansion/2

This example illustrates the use of `term_expansion/2` to augment the built-in predicate `expand_term/2` which works as a filter on the input to compile and consult. The code below will allow the declaration '`:- wait f/3`' as an alias for '`:- block f(-,?,?)`'. Wait declarations were used in previous versions of SICStus Prolog.

```
:- op(1150, fx, [wait]).

term_expansion((:- wait F/N), (:- block Head)) :-
        functor(Head, F, N),
        wb_args(N, Head).

wb_args(1, Head) :- !, arg(1, Head, -).
wb_args(N, Head) :-
%       N>1,
        arg(N, Head, ?),
        N1 is N-1,
        wb_args(N1, Head).
```

9.7 Use of Boolean Constraints

Verification of digital circuits is a major use of Boolean constraints. The predicate `fault/3` below describes a 1-bit adder consisting of five gates, with at most one faulty gate. If one of the variables `Fi` is equal to 1, the corresponding gate is faulty, and its output signal is undefined (i.e., the constraint representing the gate is relaxed).

Assuming that we have found some incorrect output from a circuit, we are interesting in finding the faulty gate. Two instances of incorrect output are listed in `fault_ex/2`:

```
fault([F1,F2,F3,F4,F5], [X,Y,Cin], [Sum,Cout]) :-
        bool:sat(
                card([0-1],[F1,F2,F3,F4,F5]) *
                (F1 + (U1 =:= X * Cin)) *
```

```
                        (F2 + (U2 =:= Y * U3)) *
                        (F3 + (Cout =:= U1 + U2)) *
                        (F4 + (U3 =:= X # Cin)) *
                        (F5 + (Sum =:= Y # U3))
                    ).

fault_ex(1, Faults) :- fault(Faults, [1,1,0], [1,0]).
fault_ex(2, Faults) :- fault(Faults, [1,0,1], [0,0]).
```

To find the faulty gates, we run the query

```
| ?- fault_ex(I,L), bool:labeling(L).

I = 1,
L = [0,0,0,1,0] ? ;

I = 2,
L = [1,0,0,0,0] ? ;

I = 2,
L = [0,0,1,0,0] ? ;

no
```

Thus for input data [1,1,0], gate 4 must be faulty. For input data [1,0,1], either gate 1 or gate 3 must be faulty.

To get a symbolic representation of the outputs interms of the input, we run the query

```
| ?- fault([0,0,0,0,0], [x,y,cin], [Sum,Cout]).

bool:sat(Sum=:=cin#x#y),
bool:sat(Cout=:=x*cin#x*y#y*cin)
```

which shows that the sum and carry out signals indeed compute the intended functions if no gate is faulty.

9.8 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)`.

9.9 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]).
```

then 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(300, xfx, :).
```

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].
```

10 Installation Dependencies

10.1 Getting Started

To start SICStus issue the shell command:

```
% prolog [-f] [-i] [-l prolog-file] arguments
```

where the *arguments* can be retrieved from SICStus by `unix(argv(`*Args*`))`, which will unify *Args* with *arguments* represented as a list of atoms. None of the *arguments* must begin with a '`-`' sign.

The flags have the following meaning:

`-f` Fast start. Don't read the '`~/.sicstusrc`' file on startup and on `reinitialise/1`. If the flag is omitted, SICStus will consult this file on startup and on `reinitialise/1`, if it exists.

`-i` Forced interactive. Prompt for user input, even if the standard input does not appear to be a terminal.

`-l` *prolog-file*
Consult the file *prolog-file* on startup and on `reinitialise/1`.

To start SICStus from a saved state *file*, issue the shell command:

```
% file [-f] [-i] [-l prolog-file] arguments
```

or the shell command:

```
% prolog -r file [-f] [-i] [-l prolog-file] arguments
```

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

```
(setq load-path (cons "/usr/local/lib/sicstus2.1" load-path))
(autoload 'run-prolog "prolog"
                  "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog"
                  "Major mode for editing prolog programs" t)
```

where '/usr/local/lib/sicstus2.1' should be replaced by the name of the SICStus source code directory.

The Emacs mode will use the value of the environment variable `EPROLOG` as a shell command to invoke SICStus. This value defaults to `prolog`. The Emacs mode provides the following commands:

`M-x run-prolog`
: Run an inferior Prolog process, input and output via the buffer `*prolog*`.

`C-c K` The entire buffer is compiled.

`C-c k` The current region is compiled.

`C-c C-k` The predicate around point is compiled. Empty lines are treated as predicate boundaries.

`C-c C` The entire buffer is consulted.

`C-c c` The current region is consulted.

`C-c C-c` The predicate around point is consulted. Empty lines are treated as predicate boundaries.

The following environment variable can be set before starting SICStus. Some of these override the default sizes of certain areas. The sizes are given in cells:

`LC_CTYPE` This selects the appropriate character set standard: The supported values are `ja_JP.euc` or `ja_JP.EUC` (for EUC) and `iso_8859_1` (for ISO 8859/1). The latter is the default. In fact, any value other than `ja_JP.euc` or `ja_JP.EUC` will cause ISO 8859/1 to be selected.

`TMPDIR` If set, indicates the pathname where temporary files should be created. Defaults to '/usr/tmp'.

`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.

11 Summary of Built-In Predicates

Note that some predicates are not available or works differently in Runtime Systems.

`!` Commit to any choices taken in the current predicate.

`(+P,+Q)` *P* and *Q*.

`(+P -> +Q ; +R)`
If *P* then *Q* else *R*, using first solution of *P* only.

`(+P -> +Q)`
If *P* then *Q* else fail, using first solution of *P* only.

`[]`
`[:File|+Files]`
Updates the program with interpreted clauses from *File* and *Files*.

`(+P;+Q)` *P* or *Q*.

`?X = ?Y` The terms *X* and *Y* are unified.

`+Term =.. ?List`
`?Term =.. +List`
The functor and arguments of the term *Term* comprise the list *List*.

`+X =:= +Y`
X is numerically equal to *Y*.

`?Term1 == ?Term2`
The terms *Term1* and *Term2* are strictly identical.

`+X =\= +Y`
X is not numerically equal to *Y*.

`+X =< +Y` *X* is less than or equal to *Y*.

`+X > +Y` *X* is greater than *Y*.

`+X >= +Y` *X* is greater than or equal to *Y*.

`?X ^ :P` Executes the procedure call *P*.

`\+ +P` Goal *P* is not provable.

`?Term1 \== ?Term2`
The terms *Term1* and *Term2* are not strictly identical.

`+X < +Y` *X* is less than *Y*.

`?Term1 @=< ?Term2`
The term *Term1* precedes or is identical to the term *Term2* in the standard order.

`?Term1 @> ?Term2`
The term *Term1* follows the term *Term2* in the standard order.

`?Term1 @>= ?Term2`
The term *Term1* follows or is identical to the term *Term2* in the standard order.

?Term1 `@<` *?Term2*
: The term *Term1* precedes the term *Term2* in the standard order.

`?=(`*?X*`,`*?Y*`)`
: *SICStus specific.* *X* and *Y* are either identical or non-unifiable.

`abolish(`*:Preds*`)`
: Makes the predicate(s) specified by *Preds* undefined.

`abolish(`*:Atom*`,+`*Arity*`)`
: Makes the predicate specified by *Atom*`/`*Arity* undefined.

`abort`
: Aborts execution of the current directive (terminates the innermost call from C in Runtime Systems).

`absolute_file_name(+`*RelativeName*`,`*?AbsoluteName*`)`
: *AbsoluteName* is the full pathname of *RelativeName*.

`ancestors(`*?Goals*`)`
: The ancestor list of the current clause is *Goals* (not available in Runtime Systems)

`arg(+`*ArgNo*`,+`*Term*`,`*?Arg*`)`
: Argument *ArgNo* of the term *Term* is *Arg*.

`assert(`*:Clause*`)`
`assert(`*:Clause*`,`*-Ref*`)`
: Asserts clause *Clause* with unique identifier *Ref*.

`asserta(`*:Clause*`)`
`asserta(`*:Clause*`,`*-Ref*`)`
: Asserts *Clause* as first clause with unique identifier *Ref*.

`assertz(`*:Clause*`)`
`assertz(`*:Clause*`,`*-Ref*`)`
: Asserts *Clause* as last clause with unique identifier *Ref*.

`atom(`*?X*`)`
: *X* is currently instantiated to an atom.

`atom_chars(+`*Atom*`,`*?CharList*`)`
`atom_chars(`*?Atom*`,+`*CharList*`)`
: The name of the atom *Atom* is the list of characters *CharList*.

`atomic(`*?X*`)`
: *X* is currently instantiated to an atom or a number.

`bagof(`*?Template*`,`*:Goal*`,`*?Bag*`)`
: *Bag* is the bag of instances of *Template* such that *Goal* is satisfied (not just provable).

`bool:labeling(+`*Variables*`)`
: *SICStus specific.* *Variables* are instantiated to 0s and 1s, satisfying any constraints on them. Enumerates all solutions by backtracking.

`bool:sat(+`*Expression*`)`
: *SICStus specific.* *Expression* is consistent with accumulated Boolean constraints, and is added to those.

`bool:taut(+`*Expression*`,?`*Truth*`)`
: *SICStus specific. Expression* is entailed by accumulated Boolean constraints and *Truth* is 1, or its negation is entailed by accumulated Boolean constraints and *Truth* is 0.

`break` Invokes the Prolog interpreter (not available in Runtime Systems)

`'C'(?`*S1*`,?`*Terminal*`,?`*S2*`)`
: *Grammar rules. S1* is connected by the terminal *Terminal* to *S2*.

`call(:`*Term*`)`
: Executes the procedure call *Term*.

`call_residue(:`*Goal*`,?`*VarsGoals*`)`
: *SICStus specific.* Executes the procedure call *Goal*. Any floundered goals and the variables they are blocked on occur as *Var-Goal* pairs in *VarsGoals*.

`character_count(+`*Stream*`,?`*Count*`)`
: *Count* characters have been read from or written to the stream *Stream*.

`clause(:`*Head*`,?`*Body*`)`
`clause(:`*Head*`,?`*Body*`,?`*Ref*`)`
`clause(?`*Head*`,?`*Body*`,+`*Ref*`)`
: There is an interpreted clause whose head is *Head*, whose body is *Body*, and whose unique identifier is *Ref*.

`close(+`*Stream*`)`
: Closes stream *Stream*.

`compare(?`*Op*`,?`*Term1*`,?`*Term2*`)`
: *Op* is the result of comparing the terms *Term1* and *Term2*.

`compile(:`*Files*`)`
: Compiles in-core the clauses in text file(s) *Files* (not available in Runtime Systems).

`consult(:`*Files*`)`
: Updates the program with interpreted clauses from file(s) *Files* (not available in Runtime Systems).

`copy_term(?`*Term*`,?`*CopyOfTerm*`)`
: *CopyOfTerm* is an independent copy of *Term*.

`current_atom(?`*Atom*`)`
: One of the currently defined atoms is *Atom*.

`current_host(?`*HostName*`)`
: *SICStus specific.* Unifies *HostName* with the name of the host SICStus is running on.

`current_input(?`*Stream*`)`
: *Stream* is the current input stream.

`current_key(?`*KeyName*`,?`*KeyTerm*`)`
: There is a recorded item in the internal database whose key is *KeyTerm*, the name of which is *KeyName*.

`current_module(?`*Module*`)`
: *Module* is a module currently in the system.

`current_module(`*?Module*`,`*?File*`)`
: *Module* is a module currently in the system, loaded from *File* (not available in Runtime Systems).

`current_op(`*?Precedence*`,`*?Type*`,`*?Op*`)`
: Atom *Op* is an operator type *Type* precedence *Precedence*.

`current_output(`*?Stream*`)`
: *Stream* is the current output stream.

`current_predicate(`*?Name*`,`*:Head*`)`
`current_predicate(`*?Name*`,`*-Head*`)`
: A user defined predicate is named *Name*, most general goal *Head*.

`current_stream(`*?FileName*`,`*?Mode*`,`*?Stream*`)`
: There is a stream *Stream* associated with the file *FileName* and opened in mode *Mode*.

`debug` Switches on debugging (not available in Runtime Systems).

`debugging`
: Displays debugging status information (not available in Runtime Systems).

`depth(`*?Depth*`)`
: The current invocation depth is *Depth* (not available in Runtime Systems).

`dif(`*?X*`,`*?Y*`)`
: *SICStus specific.* The terms *X* and *Y* are different.

`display(`*?Term*`)`
: Displays the term *Term* on the standard output stream.

`ensure_loaded(`*:Files*`)`
: Compiles or loads the file(s) *Files* if need be. (In Runtime Systems it is identical to `load/1`.)

`erase(+`*Ref*`)`
: Erases the clause or record whose unique identifier is *Ref*.

`expand_term(+`*Term1*`,`*?Term2*`)`
: The term *Term1* is a shorthand which expands to the term *Term2*.

`fail`
`false` Backtracks immediately.

`fcompile(`*:Files*`)`
: *SICStus specific.* Compiles file-to-file the clauses in text file(s) *Files* (not available in Runtime Systems).

`fileerrors`
: Enables reporting of file errors.

`findall(`*?Template*`,`*:Goal*`,`*?Bag*`)`
`findall(`*?Template*`,`*:Goal*`,`*?Bag*`,`*?Remainder*`)`
: A prefix of *Bag* is the list of instances of *Template* such that *Goal* is provable. The rest of *Bag* is *Remainder* or the empty list.

`float(`*?X*`)`
: *X* is currently instantiated to a float.

`flush_output(+`*Stream*`)`
: Flushs the buffers associated with *Stream*.

`foreign(+`*CFunctionName*`,+`*Predicate*`)`
`foreign(+`*CFunctionName*`,+`*Language*`,+`*Predicate*`)`
: *User defined*, they tell Prolog how to define *Predicate* to invoke *CFunctionName*.

`foreign_file(+`*ObjectFile*`,+`*Functions*`)`
: *User defined*, tells Prolog that foreign functions *Functions* are in file *ObjectFile*.

`format(+`*Format*`,+`*Arguments*`)`
`format(+`*Stream*`,+`*Format*`,+`*Arguments*`)`
: Writes *Arguments* according to *Format* on the stream *Stream* or on the current output stream.

`freeze(`*?Var*`,`*:Goal*`)`
: *SICStus specific.* Blocks *Goal* until `nonvar(`*Var*`)` holds.

`frozen(`*-Var*`,`*?Goal*`)`
: *SICStus specific.* The goal *Goal* is blocked on the variable *Var*.

`functor(+`*Term*`,`*?Name*`,`*?Arity*`)`
`functor(`*?Term*`,+`*Name*`,+`*Arity*`)`
: The principal functor of the term *Term* has name *Name* and arity *Arity*.

`garbage_collect`
: Performs a garbage collection.

`gc`
: Enables garbage collection.

`get(`*?C*`)`
`get(+`*Stream*`,`*?C*`)`
: The next printing character from the stream *Stream* or from the current input stream is *C*.

`get0(`*?C*`)`
`get0(+`*Stream*`,`*?C*`)`
: The next character from the stream *Stream* or from the current input stream is *C*.

`ground(`*?X*`)`
: *SICStus specific.* *X* is currently free of unbound variables.

`halt`
: Halts Prolog (terminates the innermost call from C in Runtime Systems).

`help`
: Prints a help message.

`if(+`*P*`,+`*Q*`,+`*R*`)`
: *SICStus specific.* If *P* then *Q* else *R*, exploring all solutions of *P*.

`incore(+`*Term*`)`
: Executes the procedure call *Term*.

`instance(+`*Ref*`,`*?Term*`)`
: *Term* is a most general instance of the record or clause uniquely identified by *Ref*.

`integer(`*?X*`)`
: *X* is currently instantiated to an integer.

?Y `is` `+`*X*
: *Y* is the value of the arithmetic expression *X*.

`keysort(+`*List1*`,`*?List2*`)`
: The list *List1* sorted by key yields *List2*.

`leash(+`*Mode*`)`
: Sets leashing mode to *Mode* (not available in Runtime Systems).

`length(`*?List*`,`*?Length*`)`
: The length of list *List* is *Length*.

`library_directory(`*?Directory*`)`
: *User defined*, *Directory* is a directory in the search path.

`line_count(+`*Stream*`,`*?N*`)`
: *N* is the number of lines read/written on stream *Stream*.

`line_position(+`*Stream*`,`*?N*`)`
: *N* is the number of characters read/written on the current line of *Stream*.

`listing`
`listing(`*:Specs*`)`
: Lists the interpreted predicate(s) specified by *Specs* or all interpreted predicates in the type-in module.

`load(`*:Files*`)`
: *SICStus specific.* Loads fcompiled file(s) *Files* into Prolog. (It will not write any progress messages in Runtime Systems.)

`load_foreign_files(`*:ObjectFiles*`,+`*Libraries*`)`
: Links object files *ObjectFiles* into Prolog.

`maxdepth(+`*Depth*`)`
: Limits the interpreter's invocation depth to *Depth* (not available in Runtime Systems).

`module(+`*Module*`)`
: Sets the type-in module to *Module*; See Section 5.1 [Basic Concepts], page 83 (not available in Runtime Systems).

`name(+`*Const*`,`*?CharList*`)`
`name(`*?Const*`,+`*CharList*`)`
: The name of atom or number *Const* is the string *CharList*.

`nl`
`nl(+`*Stream*`)`
: Outputs a new line on stream *Stream* or on the current output stream.

`nodebug`
: Switches off debugging.

`nofileerrors`
: Disables reporting of file errors.

`nogc`
: Disables garbage collection.

`nonvar(`*?X*`)`
: *X* is a non-variable.

`nospy` *:Spec*
: Removes spy-points from the predicate(s) specified by *Spec*.

`nospyall` Removes all spy-points (not available in Runtime Systems).

`notrace` Switches off debugging (not available in Runtime Systems).

`number(`*?X*`)`
: *X* is currently instantiated to a number.

`number_chars(+`*Number*`,?`*CharList*`)`
`number_chars(?`*Number*`,+`*CharList*`)`
: The name of the number *Number* is the list of characters *CharList*.

`numbervars(?`*Term*`,+`*N*`,?`*M*`)`
: Number the variables in the term *Term* from *N* to *M*-1.

`on_exception(?`*Pattern*`,:`*ProtectedGoal*`,:`*Handler*`)`
: *SICStus specific.* Executes the procedure call *ProtectedGoal*. If during the execution `raise_exception(`*Exception*`)` is called, and *Exception* matches *Pattern*, the execution of *ProtectedGoal* abort, *Pattern* is unified with a copy of *Exception* and *Handler* is called.

`op(+`*Precedence*`,+`*Type*`,+`*Name*`)`
: Makes atom(s) *Name* an operator of type *Type* precedence *Precedence*.

`open(+`*FileName*`,+`*Mode*`,-`*Stream*`)`
: Opens file *FileName* in mode *Mode* as stream *Stream*.

`open_null_stream(-`*Stream*`)`
: Opens an output stream to the null device.

`otherwise`
: Succeeds.

`phrase(:`*Phrase*`,?`*List*`)`
`phrase(:`*Phrase*`,?`*List*`,?`*Remainder*`)`
: *Grammar rules.* The list *List* can be parsed as a phrase of type *Phrase*. The rest of the list is *Remainder* or empty.

`plsys(+`*Term*`)`
: Invokes operating system services.

`portray(+`*Term*`)`
: *User defined*, tells `print/1` what to do.

`portray_clause(+`*Clause*`)`
`portray_clause(+`*Stream*`,+`*Clause*`)`
: Pretty prints *Clause* on the stream *Stream* or on the current output stream.

`predicate_property(:`*Head*`,?`*Prop*`)`
`predicate_property(-`*Head*`,?`*Prop*`)`
: *Head* is the most general goal of a currently defined predicate that has the property *Prop*.

`prepare_foreign_files(+`*ObjectFiles*`)`
: *SICStus specific.* Generates interface code for *ObjectFiles*. (not available in Runtime Systems).

`print(`*?Term*`)`
`print(+`*Stream*`,`*?Term*`)`
: Portrays or else writes the term *Term* on the stream *Stream* or on the current output stream.

`print_uncaught_exception(+`*Exception*`)`
: *User defined*, customized printing of exceptions caught at the toplevel.

`profile_data(`*:Files*`,`*?Selection*`,`*?Resolution*`,`*-Data*`)`
: *Data* is the profiling data collected from the instrumented predicates defined in the file(s) *Files* with selection and resolution *Selection* and *Resolution* respectively. (not available in Runtime Systems).

`profile_reset(`*:Files*`)`
: The profiling counters for the instrumented predicates in the file(s) *Files* are zeroed (not available in Runtime Systems).

`prolog_flag(+`*FlagName*`,`*?Value*`)`
: *Value* is the current value of *FlagName*.

`prolog_flag(+`*FlagName*`,`*?OldValue*`,`*?NewValue*`)`
: *OldValue* and *NewValue* are the old and new values of *FlagName*.

`prompt(`*?Old*`,`*?New*`)`
: Changes the prompt from *Old* to *New*.

`put(+`*C*`)`
`put(+`*Stream*`,+`*C*`)`
: The next character sent to the stream *Stream* or to the current output stream is *C*.

`query_expansion(+`*RawQuery*`,`*?Query*`)`
: *SICStus specific, user defined*, transforms the interpreter top-level query *RawQuery* into *Query* to be executed.

`raise_exception(+`*Exception*`)`
: *SICStus specific.* Causes abortion of a part of the execution tree scoped by the closest enclosing `on_exception/3` invocation with its first argument matching *Exception*.

`read(`*?Term*`)`
`read(+`*Stream*`,`*?Term*`)`
: Reads the term *Term* from the stream *Stream* or from the current input stream.

`reconsult(`*:Files*`)`
: Updates the program with interpreted clauses from file(s) *Files* (not available in Runtime Systems).

`recorda(+`*Key*`,`*?Term*`,`*-Ref*`)`
: Makes the term *Term* the first record under key *Key* with unique identifier *Ref*.

`recorded(`*?Key*`,`*?Term*`,`*?Ref*`)`
: The term *Term* is currently recorded under key *Key* with unique identifier *Ref*.

`recordz(+`*Key*`,`*?Term*`,`*-Ref*`)`
: Makes the term *Term* the last record under key *Key* with unique identifier *Ref*.

`reinitialise`
: Initializes Prolog, reconsulting '`~/.sicstusrc`' if it exists (terminates the innermost call from C in Runtime Systems).

`repeat`
: Succeeds repeatedly.

`restore(+`*File*`)`
: Restores the state saved in file *File* (not available in Runtime Systems).

`retract(`*:Clause*`)`
: Erases repeatedly the next interpreted clause of form *Clause*.

`retractall(`*:Head*`)`
: Erases all clauses whose head matches *Head*.

`save(+`*File*`)`
`save(+`*File*`,`*?Return*`)`
: Saves the current state of Prolog in file *File*; *Return* is 0 after a save and 1 after a restore (not available in Runtime Systems).

`save_program(+`*File*`)`
: Saves the current state of the Prolog data base in file *File* (not available in Runtime Systems).

`see(+`*File*`)`
: Makes file *File* the current input stream.

`seeing(`*?File*`)`
: The current input stream is named *File*.

`seen`
: Closes the current input stream.

`set_input(+`*Stream*`)`
: Sets the current input stream to *Stream*.

`set_output(+`*Stream*`)`
: Sets the current output stream to *Stream*.

`setarg(+`*ArgNo*`,+`*CompoundTerm*`,`*?NewArg*`)`
: *SICStus specific.* Replaces destructively argument *ArgNo* in *CompoundTerm* with *NewArg* and undoes it on backtracking.

`setof(`*?Template*`,`*:Goal*`,`*?Set*`)`
: *Set* is the set of instances of *Template* such that *Goal* is satisfied (not just provable).

`skip(+`*C*`)`
`skip(+`*Stream*`,+`*C*`)`
: Skips characters from *Stream* or from the current input stream until after character *C*.

`socket(+`*Domain*`,`*-Socket*`)`
: *SICStus specific.* Opens *Socket* in the domain *Domain*.

`socket_accept(+`*Socket*`,`*-Stream*`)`
: *SICStus specific.* Accept a connection to *Socket*, *Stream* is opened for read-write.

`socket_bind(+`*Socket*`,?`*Address*`)`
: *SICStus specific.* The socket *Socket* is bound to *Address*.

`socket_connect(+`*Socket*`,?`*Address*`,-`*Stream*`)`
: *SICStus specific.* Connects *Socket* to *Address* and opens the bidirectional stream *Stream*.

`socket_interrupt(?`*Stream*`,?`*OldHandler*`,?`*NewHandler*`)`
: *SICStus specific.* Sets/reads the interrupt handler connected to the socket stream *Stream*.

`socket_listen(+`*Socket*`,+`*Length*`)`
: *SICStus specific.* Sets the queue-length for *Socket* to *Length*.

`socket_select(?`*Socket*`,-`*NewStream*`,+`*TimeOut*`,+`*Streams*`,-`*ReadStreams*`)`
: *SICStus specific.* Returns a list *ReadStreams* containing streams with pending characters. Only the streams in the list *Streams* are checked. If *Socket* is instantiated, also checks that socket for connections and open *NewStream* as in `socket_accept/2`. *TimeOut* specifies a timeout on the form `off` or `Sec:MicroSec`.

`sort(+`*List1*`,?`*List2*`)`
: The list *List1* sorted into order yields *List2*.

`source_file(?`*File*`)`
`source_file(:`*Pred*`,?`*File*`)`
`source_file(-`*Pred*`,?`*File*`)`
: The predicate *Pred* is defined in the file *File*. (not available in Runtime Systems.)

`spy` *:Spec* Sets spy-points on the predicate(s) specified by *Spec*.

`statistics`
: Outputs various execution statistics.

`statistics(?`*Key*`,?`*Value*`)`
: The execution statistic key *Key* has value *Value*.

`stream_code(?`*Stream*`,?`*StreamCode*`)`
: *StreamCode* is a foreign language (C) version of *Stream*.

`subgoal_of(?`*Goal*`)`
: An ancestor goal of the current clause is *Goal* (not available in Runtime Systems).

`tab(+`*N*`)`
`tab(+`*Stream*`,+`*N*`)`
: Outputs *N* spaces to the stream *Stream* or to the current output stream.

`tell(+`*File*`)`
: Makes file *File* the current output stream.

`telling(?`*File*`)`
: The current output stream is named *File*.

`term_expansion(+`*Term1*`,?`*Term2*`)`
: *User defined*, tells `expand_term/2` what to do.

`time_out(`*:Goal*`,+`*Time*`,`*?Result*`)`
: *SICStus specific.* Limits the time spent on proving the goal *Goal* (not available in Runtime Systems).

`told` Closes the current output stream.

`trace` Switches on debugging and start tracing immediately (not available in Runtime Systems).

`true` Succeeds.

`ttyflush` Flushes the standard output stream buffer.

`ttyget(`*?C*`)`
: The next printing character input from the standard input stream is *C*.

`ttyget0(`*?C*`)`
: The next character input from the standard input stream is *C*.

`ttynl` Outputs a new line on the standard output stream.

`ttyput(+`*C*`)`
: The next character output to the standard output stream is *C*.

`ttyskip(+`*C*`)`
: Skips characters from the standard input stream until after character *C*.

`ttytab(+`*N*`)`
: Outputs *N* spaces to the standard output stream.

`undo(`*:Term*`)`
: *SICStus specific.* The goal `call(`*Term*`)` is executed on backtracking.

`unix(+`*Term*`)`
: Invokes operating system services.

`unknown(`*?OldState*`,`*?NewState*`)`
: Changes action on undefined predicates from *OldState* to *NewState* (not available in Runtime Systems).

`use_module(`*:Files*`)`
: Loads the module file(s) *Files* if necessary and import all public predicates. (In Runtime Systems it is equivalent to `load/1`.)

`use_module(`*:File*`,+`*PredicateList*`)`
: Loads the module file *File* if necessary and imports the predicates in *PredicateList*. (In Runtime Systems it always loads the '`.ql`' file *File*.)

`use_module(+`*Module*`,`*?Files*`,+`*PredicateList*`)`
`use_module(`*?Module*`,`*:Files*`,+`*PredicateList*`)`
: Imports *PredicateList* from *Module* if it is instantiated and the module is currently in the system, otherwise behaves like `use_module/2`. (In Runtime Systems it is equivalent to `use_module/2`.)

`user_help`
: *User defined*, tells `help/0` what to do.

`var(`*?X*`)` *X* is currently uninstantiated.

`version` Displays introductory and/or system identification messages.

`version(+`*Message*`)`

Adds the atom *Message* to the list of introductory messages.

`when(+`*Condition*`,:`*Goal*`)`

SICStus specific. Blocks *Goal* until the *Condition* is true.

`write(?`*Term*`)`
`write(+`*Stream*`,?`*Term*`)`

Writes the term *Term* on the stream *Stream* or on the current output stream.

`write_canonical(?`*Term*`)`
`write_canonical(+`*Stream*`,?`*Term*`)`

Writes *Term* on the stream *Stream* or on the current output stream so that it may be read back.

`writeq(?`*Term*`)`
`writeq(+`*Stream*`,?`*Term*`)`

Writes the term *Term* on the stream *Stream* or on the current output stream, quoting names where necessary.

12 Standard Operators

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic,
                    multifile, block, meta_predicate ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).      /* See note below */
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                 =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, #, /\, \/ ]).
:- op(  500,  fx, [ +, - ]).
:- op(  400, yfx, [ *, /, //, <<, >> ]).
:- op(  300, xfx, [ mod, ~ ]).
:- op(  200, xfy, [ ^ ]).
```

Note that a comma written literally as a punctuation character can be used as though it were an infix operator of precedence 1000 and type `xfy`, i.e.

```
X,Y    ','(X,Y)
```

represent the same compound term. But note that a comma written as a quoted atom is *not* a standard operator.

Index of Built-Ins

(Index is nonexistent)

Concept Index

(Index is nonexistent)

Table of Contents