Swedish Institute of
Computer Science

SICS

SICStus Prolog User's Manual

by

Mats Carlsson and Johan Widén

SICS R88007
Research Report
ISSN 0283-3638

Swedish Institute of Computer Science

SICStus Prolog User's Manual

20 February 1988

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 Widen

This manual corresponds to Sicstus version 0.6.

Introduction

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

For an introduction to programming in Prolog, readers are recommended to consult [Sterling & Shapiro 86]. However, for the benefit of those who do not have access to a copy of this book, and for those who have some prior knowledge of logic programming, a summary of the language is included. See chapter 5 [Prolog Intro], page 71.

This manual describes a Prolog system developed at the Swedish Institute of Computer Science. The system consists of a *WAM* emulator written in C, a library and runtime system written in C and Prolog and an interpreter and a compiler written in Prolog. The Prolog engine is a Warren Abstract Machine (WAM) emulator [Warren 83]. Two modes of compilation are available: in-core i.e. incremental, and file-to-file.

When compiled, a procedure will run about 10 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. It also contains primitives for demand-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 7 [Installation Intro], page 101.

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

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. We shall call *name/arity* a *predicate spec.*

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* can be of one of the forms: *+ArgName* - this argument should be instantiated in calls to the predicate. *-ArgName* - this argument should *not* be instantiated in calls to the predicate. *?ArgName* - this argument may or may not be instantiated in calls to the predicate.

We adopt the following convention for delineating character strings in the text of this manual: when a string is being used as a Prolog atom it is written thus: **user** or **'user'**; but in all other circumstances double quotes are used.

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 SPS respectively.

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 7.1 [Getting Started], page 101):

```
% sicstus
```

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

```
SICStus V0.6(-) Mon Feb 15 11:20:57 MET 1988
Copyright (C) 1988, Swedish Institute of Computer Science.
All rights reserved.
| ?-
```

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 procedure do not have to be immediately consecutive, but remember

that their relative order may be important.

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

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

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

```
| ?- ['myfile.pl'].
| ?- ['/usr/prolog/somefile'].
```

The specified file is then read in. Clauses in the file are stored ready to 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 command.

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 39) 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 procedures in the consulted files will replace any existing clauses for those procedures, 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 command:

```
| ?- [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.

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 (.), and that therefore Prolog will

not execute anything until you have typed the full stop (and then 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 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(_52)

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}
```

as a warning.

The principal use for commands (as opposed to queries) is to allow files to contain directives which call various procedures, but for which you do not want to have the answers printed out. In such cases you only want to call the procedures 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 procedure read that fails to comply with syntax requirements is displayed on the terminal as soon as it is read. A mark indicates the point in the string of symbols where the parser has failed to continue analysis. e.g.

```
member(X, X:L).
```

gives:

```
** atom follows expression **
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).
```

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 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 predicates with no clauses to be reported and the debugging system to be entered at the earliest opportunity (the default state);
- '`fail`', which causes calls to such predicates to fail.

Calls to predicates that have no clauses are not caught.

The built-in predicate

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

unifies *OldState* with the current state and sets the state to *NewState*. 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.

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 procedures.

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 command as soon as possible.
`c`	continue the execution.

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

`e` exit from Prolog, closing all files.

`h`

`?` list available commands.

`t` enable trace. See section 2.3 [Trace], page 17.

1.8 Exiting From The Interpreter

To exit from the interpreter and return to monitor level either type ^D at interpreter top level, or call the built in procedure halt, 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 command 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 procedure 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. All open IO streams are closed, and 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 7.1 [Getting Started], page 101.

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 command, 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. 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 procedures may also be saved with the directive:

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

The file *File* becomes an executable file. See section 7.1 [Getting Started], page 101. After restoring a partial program state, the interpreter will reinitialise itself.

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

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:

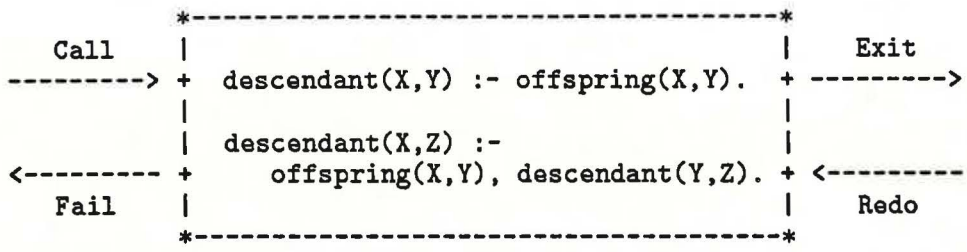
- The *Procedure Box* model of Prolog execution which provides a simple way of visualising control flow, especially during backtracking. Control flow is viewed at the procedure level, rather than at the level of individual clauses.
- The ability to exhaustively trace your program or to selectively set *spy-points*. Spy-points allow you to nominate interesting procedures at which the program is to pause so that you can interact.
- The wide choice of control and information options available during debugging.

Much of the information in this chapter is also in Chapter eight of [Clocksin & Mellish 81] which is recommended as an introduction.

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 procedure 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 behaviour of his program in a consistent way.

Let us look at an example Prolog procedure :



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 procedure 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 procedure. 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 procedure. 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 procedure 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 procedure should really be seen as an invocation_box. That is, there will be a different box for each different invocation of the procedure. Obviously, with something like a recursive procedure, 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.

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 unknown procedures 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 20. 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.

The initial value of *Leashing Mode* is `[call,exit,redo,fail]` (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 procedure 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 procedure 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 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 procedures given by *Spec*. *Spec* is either an atom, a predicate spec, or a list of such specifications. An atom is taken as meaning all the predicates whose name is that atom. If you specify an atom but there is no definition for this predicate (of any arity) then nothing will be done. If you really want to place a spy-point on a currently non-existent procedure, then you must use the full form *atom/arity*; you will get a warning message in this case. If you set some spy-points when the debugger is switched off then it will be automatically switched on.

nospy *+Spec*

This is similar to spy *Spec* except that all the procedures 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 20.

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 is directed. (This allows you to trace programs while they are performing file IO.) The basic format is as follows:

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

The first number is the unique invocation identifier. This is continuously incrementing 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 33) 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 20). 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 procedures 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`, and `'\+'/1`. This means that you will never see messages concerning these predicates during debugging.

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
 d     display            p       print
 w     write
 g     ancestors          g <n>   ancestors n
 n     nodebug            =       debugging
 +     spy this           -       nospy this
 a     abort              b       break
 @     command            u       unify
 <     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 17), 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 procedures, 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 procedure. That is, you will not see anything until control comes back to this procedure (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 procedures 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 trys 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. The system is currently not able to detect this situation, and the behaviour is undefined. It is hoped that this situation will improve in future versions.

`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`. This may be useful if your pretty print routine (portray) is not doing what you want.

`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 49). You can always be sure of jumping to any goal in the ancestor list (by using retry etc). If you supply an integer varn, 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).

`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 4.13 [Debug Pred], page 60.

`+` *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.

`<` 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 a program execution. However this can lead to unexpected behaviour under the following circumstances: a procedure has been successfully executed; it is subsequently redefined by a consult, and is later reactivated by backtracking. When the backtracking occurs, the new clauses for the procedure have physically replaced the old ones, resulting in undefined behaviour. Thus large amounts of (unwanted) activity takes place on backtracking. The problem does not arise if you do the consult when you are at the Call port of the procedure to be redefined.

3. Loading Programs

Programs can be loaded in three different ways: consulted, compiled, or loaded from object files. Loading from objects files 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 [Environ], page 65).

Consulted procedures are equivalent to, but slower than, compiled ones. The two types of procedures can call each other freely. Consulted procedures possess the property of being *dynamic* - other procedures may inspect and modify them, adding or deleting individual clauses.

The SICStus Prolog compiler produces compact and efficient code, running about 10 times faster than interpreter 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. Compiled procedures are not dynamic. Spy-points can be placed on compiled procedures, however.

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 file names 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 procedure defined in the files erases any clauses for that procedure 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`' (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 procedure can call any other.

`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 file '`consulted`' is normally a temporary '`patch`' file containing only the amended procedure(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 file name in the list was described earlier (see section 1.2 [Reading In], page 5).

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

To compile a program to 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 string '`.pl`' to it and try to locate a source file with that name and compile it to an object file. The object file name if formed by appending the string '`.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 file name (specified without the trailing '`.ql`') or a list of file

names. This directive has the same effect as if the source files had been compiled using `compile/1` directly.

3.2 Declarations

When a program is to be loaded, it is sometimes necessary to tell the system to treat some of the procedures specially. This information is supplied by including *declarations* about such procedures in the source file, preceding any clauses for the procedures 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.

The following two declarations are relevant both in Quintus Prolog and in SICStus Prolog:

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

where each *PredSpec* is a predicate spec, causes the specified predicates to become dynamic, which implies that they have to be 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.

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

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 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), or is explicitly abolished. *This declaration is not supported for file-to-file compilation.*

The following two declarations are not normally relevant in any Prologs but SICStus Prolog:

`:- parallel` *PredSpec*, ..., *PredSpec*.

enables the clauses of the specified predicates to be run in parallel. However, the execution of

SICStus Prolog is strictly sequential, and this construct is reserved for future extension which have the ability to run clauses in parallel.

`:- wait` *PredSpec*`, ..., `*PredSpec*`.`

introduces an exception to the rule that goals be run strictly from left to right within a clause. Calls to the specified predicates *suspend* if the first argument of the call is uninstantianted. "As soon as" (see below) that argument becomes instantiated, by some other goal binding it to a non-variable term, the suspended goal is resumed. The user top-level checks whether any goals are still suspended when no more goals suspend. If that is the case, a warning is printed on the terminal, to notify the user that the answer (s)he has got is really a speculative one, since it is only valid if the suspended goal is true:

`** Constraint not yet run:` *Goal*

The behaviour of suspending goals on the first argument cannot be switched off, except by abolishing or redefining the predicate.

A suspended goal is resumed before the next procedure call, where the following built-in predicates do not count as procedures in compiled code:

```
'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
```

Note also that there is an implicit cut in the `\+` and `->` constructs.

Sometimes, it is crucial that the suspended goal be resumed before a call to one of the above built-in predicates. Since most of the above are meta-logical primitives, the semantics of them can depend on whether a variable is currently bound etc. For example, if a unification is followed by a

cut, and the unification may cause suspended a goal to be resumed, and the goal might fail, then the cut should not happen. Unfortunately, with the formulation

```
X=Y, !, p(Z), ...
```

the cut will happen before resuming the suspended goal. Inserting a dummy goal '**true**' forces dummy goals to be invoked. Thus the following formulation achieves the desired timing:

```
X=Y, true, !, p(Z), ...
```

The following two declarations are sometimes relevant in other Prologs, but are ignored by SICStus Prolog. They are however accepted for compatibility reasons:

:- public *PredSpec*, ..., *PredSpec*.

In some Prologs, this declaration is necessary for making compiled predicates visible for the interpreter. In SICStus Prolog, any predicate may call any other, and all are visible.

:- mode *ModeSpec*, ..., *ModeSpec*.

where each *ModeSpec* is a mode spec. 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. SICStus Prolog ignores mode declarations. However, mode declarations may be used as a commenting device, as they express the programmer's intention of data flow in predicates. If you do so, use only the atoms '+', '-', and '?' as arguments in your mode specs, as in

```
:- mode append(+, +, -).
```

3.3 Indexing

In contrast to the interpreter, the clauses of a compiled procedure 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 in the procedure). This can be very important where there is a large number of clauses in a procedure.

Indexing also improves the Prolog system's ability to detect determinacy - important for conserving working storage.

3.4 Tail Recursion Optimisation

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

When execution reaches the last goal in a clause belonging to some procedure, and provided there are no remaining backtrack points in the execution so far of that procedure, all of the procedure'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 procedure, can continue executing indefinitely, provided each individual structure, *State*, is not too large. The procedure cycle is equivalent to an iterative loop in a conventional language.

To take advantage of tail recursion optimisation one must ensure that the Prolog system can recognise that the procedure 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 5.3 [Cut], page 80.

3.5 Practical Limitations

The number of arguments of a procedure may not exceed 256.

The number of variables in a clause may not exceed 256.

Compiled code cannot be traced.

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 8 [Pred Summary], page 103.

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
- Convenience
- Extra Control
- Information about the State of the Program
- Meta-Logical
- Miscellaneous
- Modification of the Program
- Internal Database
- Sets
- Interface to Foreign Language Functions
- Debugging
- Definite Clause Grammars
- Environmental

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

4.1 Input / Output

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

A stream to a file *FileName* can be opened 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, or for finding the currently existing streams.

There are three standard IO streams, referred to as:

`user_input`
: The standard input stream.

`user_output`
: The standard output stream.

`user_error`
: The standard error stream.

The atom `user` denotes `user_input` or `user_output`, depending on context. Terminal output is only guaranteed to be displayed after a newline is written or `ttyflush/0` is called.

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` 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 terminal. The predicates `seeing/1` and `telling/1` are used for retrieving the file name associated with the current input and output streams (respectively).

A file is referred to by its name written as an atom, i.e. it must be surrounded by single quotes if it is not already a legal atom. e.g.

```
myfile
'123'
'data.lst'
'/usr/prolog/abc.pl'
```

All IO errors normally cause an abort, except for the effect of the built-in predicate `nofileerrors/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,

file names are relative to the current working directory. If invoked recursively, i.e. in the scope of another invocation of one of these predicates, file names are relative to the directory of the file being read in.

`consult(`*+Files*`)`

`reconsult(`*+Files*`)`

`[`*File*`|`*Files*`]`
: Consults the source file or list of files specified by *Files*. See chapter 3 [Load Intro], page 25.

`[`*+File*`|`*+Files*`]`
: Shorthand notation for consulting a list of files.

`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. See chapter 3 [Load Intro], page 25.

`fcompile(`*+Files*`)`
: Compiles the source file or list of files specified by *Files*. The suffix '`.pl`' is added to the given file names to yield the real source file names. The compiled code is placed on the object file or list of files formed by adding the suffix '`.ql`' to the given file names. See chapter 3 [Load Intro], page 25.

`load(`*+Files*`)`
: Loads the object file or list of files specified by *Files*. See chapter 3 [Load Intro], page 25.

4.1.2 Input and Output of Terms

`read(`*?Term*`)`
: The next term, delimited by a full-stop (i.e. a . followed by either a space or a control character), is read from the current input stream and unified with *Term*. The syntax of the term must agree with current operator declarations. If a call `read(`*Term*`)` causes the end of the current input stream to be reached, *Term* is unified with the term `end_of_file`. Further calls to `read/1` for the same stream will then cause an error failure.

`write(`*?Term*`)`
: The term *Term* is written to the current output stream according to current operator declarations.

`display(?Term)`

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

`write_canonical(?Term)`

Similar to `write(Term)`. The term will be written according to the standard syntax. The output from `write_canonical/1` can be parsed by *read/1* even if the term contains special characters or if operator declarations have changed.

`writeq(?Term)`

Similar to `write(Term)`, but the names of atoms and functors are quoted where necessary to make the result acceptable as input to `read/1`.

`print(?Term)`

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

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

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 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)`

This writes the clause *Clause* to the current output stream, exactly as `listing/0`, `listing/1` would have written it, including a period at the end

`format(+Format,+Arguments)`

Print *Arguments* onto the current output 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 50) 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` is a Prolog implementation of the C `stdio` function `printf()`.

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 ASCII 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)
```

If N is not supplied the action defaults to

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

`~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.

`~N`D (Print decimal.) The argument is an integer. Identical to `~N`d except that , will separate groups of three digits to the left of the decimal point. Example:

```
format("Hello ~1D world!", [12345])
```

will print as

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

`~N`r (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.

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

```
format("Hello ~16R world!", [15])
```

will print as

```
'Hello F world!'
```

`~N`s (Print string.) The argument is a list of ASCII 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/1` (see section 4.1.2 [Term IO], page 33). 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/1` (see section 4.1.2 [Term IO], page 33). Example:

```
assert((portray([X|Y]) :-
    write('cons('),
    print(X),
    write(','),
    print(Y),
    write(')'))).
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/1` (see section 4.1.2 [Term IO], page 33). 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/1` (see section 4.1.2 [Term IO], page 33). 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 always uses the terminal.

nl A new line is started on the current output stream.

get0(?*N*) *N* is the ASCII code of the next character from the current input stream.

get(?*N*) *N* is the ASCII code of the next non-blank printable character from the current input stream.

skip(+*N*)
Skips to just past the next ASCII character code *N* from the current input stream. *N* may be an arithmetic expression.

put(+*N*) ASCII character code *N* is output to the current output stream. *N* may be an arithmetic expression.

tab(+*N*) *N* spaces are output to the current output stream. *N* may be an arithmetic expression.

The above predicates are the ones which are the most commonly used, as they can refer either to files or to the user's terminal. In most cases these predicates are sufficient, but there is one limitation: if you are outputting to the terminal via **put/1** then nothing is output until such time as you put a newline character. If this line by line output is inadequate, you have to 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 terminal and the buffer is flushed.

ttyflush Flushes the terminal output buffer. Output to the terminal, using either **ttyput/1** or **put/1**, 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. N.B. Under UNIX, it is not necessary to use this predicate, as terminal output is not buffered.

`ttyget0(?N)`
: *N* is the ASCII code of the next character input from the terminal.

`ttyget(?N)`
: *N* is the ASCII code of the next non-blank printable character from the terminal.

`ttyput(+N)`
: The ASCII character code *N* is output to the terminal. *N* may be an arithmetic expression.

`ttyskip(+N)`
: Skips to just past the next ASCII character code *N* from the terminal. *N* may be an arithmetic expression.

`ttytab(+N)`
: *N* spaces are output to the terminal. *N* may be an arithmetic expression.

4.1.4 Stream IO

The following predicates manipulate streams.

`open(+FileName,+Mode,-Stream)`
: Open file '`FileName`' with mode *Mode* and unify the resulting stream 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.

`close(+X)`
: If *X* is a stream then the stream is closed. If *X* is an atom then *X* is assumed to be the name of an open file. The file is closed.

`absolute_file_name(+RelativeName,?AbsoluteName)`
: If *RelativeName* is of the form `library(Name)` then `absolute_file_name/2` will search for *Name* (according to the suffix rules below) in the directories specified by `library_directory/1`.

  If '*RelativeName*`.pl`' is found then *AbsoluteName* will be unified with the full path name of this file. *AbsoluteName* will otherwise be unified with the full path name of *RelativeName*.

`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)`
: *FileName* is unified with the name of *Stream*. *Mode* is unified with the mode of *Stream*. This predicate can be used for enumerating all currently open streams through backtracking.

`set_input(+Stream)`
: Set the current input to *Stream*.

`set_output(+Stream)`
: Set the current output to *Stream*.

`flush_output(+Stream)`
: Flush all internally buffered characters 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 '*Name*`.pl`' or '*Name*' is found.

  Directories to be searched may be added by using `asserta/1` or `assertz/1`:

  ```
  asserta(library_directory((Directory)))
  ```

`open_null_stream(-Stream)`
: Open an output stream to the null device. Everything written to this stream will be thrown away.

`stream_code(?Stream,?StreamCode)`
: *StreamCode* is a foreign language (C) version of *Stream*. Under UNIX *StreamCode* is a `stdio FILE *`.

`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 type an error message and then abort execution.

Several IO predicates that use the streams `user_input` or `user_output` implicitly (see section 4.1.2 [Term IO], page 33) (see section 4.1.3 [Char IO], page 38) are available in an alternative version where the stream is specified explicitly. The rule is that the stream is the first argument.

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

`get(+Stream,?C)`

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

`nl(`*+Stream*`)`

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

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

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

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

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

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

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

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

4.1.5 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`, otherwise with the current input stream.

`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`, otherwise with the current output stream.

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

4.1.6 An Example

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

```
process_file(F) :-
        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
```

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 the one provided by the C `long int` type, typically `[-2^31, 2^31-1]`. Integers in the range `[-2^25, 2^25-1]` are stored and computed with more efficiently than larger integers.

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, an expression evaluates to a float if any of its arguments is a float, otherwise to an integer.

`X+Y`	This evaluates to the sum of X and Y.
`X-Y`	This evaluates to the difference of X and Y.
`X*Y`	This evaluates to the product of X and Y.
`X/Y`	This evaluates to the quotient of X and Y. The value is always a *float*.
`X//Y`	This evaluates to the *integer* quotient of X and Y.
`X mod Y`	This evaluates to the *integer* remainder after dividing X by Y.
`-X`	This evaluates to the negative of X.
`integer(X)`	This evaluates to the nearest integer between X and 0, if X is a float, otherwise to X itself.
`float(X)`	This evaluates to the floating-point equivalent of X, if X is an integer, otherwise to X itself.
`X/\Y`	This evaluates to the bitwise conjunction of the integers X and Y.
`X\/Y`	This evaluates to the bitwise disjunction of the integers X and Y.
`X\/Y`	This evaluates to the bitwise exclusive or of the integers X and Y.
`\(X)`	This evaluates to the bitwise negation of the integer X.
`X<<Y`	Bitwise left shift of X by Y places.
`X>>Y`	Bitwise right shift of X by Y places.
`[X]`	A list of just one element evaluates to X if X is a number. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its ASCII code; e.g. `"A"` behaves within arithmetic expressions as the integer 65.

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).
```

This works even for compiled code.

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 is 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 arithmetic value of Y.
X `>=` Y	The numeric value of X is greater than or equal to the arithmetic 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 42) 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).
- Numbers, in numeric order. An integer is put before the equivalent floating point number (e.g. 1 is put before 1.0).
- Atoms, in alphabetical (i.e. ASCII) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order). Remember, lists are equivalent to compound terms with principal functor `'.'/2`.

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

```
[ X, -9, 1, 1.0, fie, foe, fum, X = Y, fie(0,2), fie(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 4.4 [Convenience], page 46).

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:

`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, and any identical (i.e. `==`) elements are merged, yielding the list *List2*. (The time taken to do this is at worst order ($N \log N$) 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. (The time taken to do this is at worst order ($N \log N$) where N is the length of *List1*.)

4.4 Convenience

P , *Q* *P* and *Q*.

P ; *Q* *P* or *Q*.

`true`

`otherwise`

Always succeeds.

`fail`

`false` Always fails.

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

`dif(`*X*`,`*Y*`)`

Constrains *X* and *Y* to represent different terms i.e. to be non unifiable. Calls to `dif/2` succeed, fail, or suspend depending on whether *X* and *Y* are sufficiently instantiated. For example:

```
| ?- dif(X,a).

 ** Constraint not yet run: dif(_140,a)
X = _140

| ?- dif(X,a), X=a.

no

| ?- dif([X|a],[b|Y]), X=a.

X = a,
Y = _59
```

`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 *FlagName*, then set the value of *FlagName* to *NewValue*. The possible *FlagNames* and values are:

`character_escapes`

`on` or `off`. Enable or disable character escaping. Presently this has *no effect* in SICStus Prolog.

`debugging`

`trace`: turn on trace mode. `debug`: turn on the debugger. `off`: turn off trace and the debugger.

`fileerrors`
: `on`: equivalent to `fileerrors/0`. `off`: equivalent to `nofileerrors/0`.

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

`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.

`gc_trace`
: `verbose`: turn on verbose tracing of garbage collection. `terse`: turn on terse tracing of garbage collection. `off`: turn off tracing of garbage collection.

`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`.

`unknown`
: `trace`: cause calls to predicates with no definition to be reported and the debugging system to be entered at the earliest opportunity. `fail`: cause calls to such predicates to fail.

`prolog_flag(`*+FlagName*`,` *?OldValue*`)`
: This is a shorthand for

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

4.5 Extra Control

`!`
: See section 5.3 [Cut], page 80.

`\+` *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.
\+(_).
```

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 `\+` of arity 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.
```

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

P `->` *Q* When occurring other than as one of the alternatives of a disjunction, is 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.

`repeat` Generates an infinite sequence of backtracking choices. It behaves as if defined by the clauses:

```
repeat.
repeat :- repeat.
```

`freeze(`*+Goal*`)`

Suspend *Goal* until *Goal* is ground. This can be used e.g. for defining a sound form of negation by:

```
not(Goal) :- freeze((\+ Goal)).
```

(`not/1` is not a built-in predicate.)

`freeze(`*?X*`,`*+Goal*`)`

Suspend *Goal* until `nonvar(`*X*`)`. This is defined as if by:

```
:- wait freeze/2.
freeze(_, Goal) :- Goal.
```

`frozen(`*-X*`,`*?Goal*`)`

If some goal is suspended on the variable *Var*, then that goal is unified with *Goal*. Otherwise, *Goal* is unified with the atom `true`.

`call(+Goal,?Vars)`

The *Goal* is executed as if by `call/1`. If after the execution there are still some subgoals of *Goal* that are suspended on some variables, then *Vars* is unified with the list of such variables. Otherwise, *Vars* is unified with the empty list `[]`.

4.6 Information about the State of the Program

`listing` Lists in the current output stream all the clauses in the current interpreted program. Clauses listed to a file can be consulted back.

`listing(+A)`

If *A* is just an atom, then the interpreted procedures for all predicates of that name are listed as for `listing/0`. The argument *A* may also be a predicate spec in which case only the clauses for the specified predicate are listed. Finally, it is possible for *A* to be a list of specifications of either type, e.g.

```
:- listing([concatenate/3, reverse, go/0]).
```

`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.

Only available when the debugger is switched on.

`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.

Only available when the debugger is switched on.

`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)`

Name is the name of a user defined predicate, and *Head* is the most general form of that predicate. This predicate can be used to enumerate all user defined predicates through backtracking.

`predicate_property(?Head,?Property)`

Term is the most general form of an existing predicate, 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`, `multifile`, `parallel`, and `wait`, for predicates that have been declared to have these properties (see section 3.2 [Declarations], page 27).

A predicate is dynamic iff it is interpreted. This predicate can be used to enumerate all existing predicates and their properties through backtracking.

4.7 Meta-Logical

`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 command

```
:- 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`.

`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*`)`
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 a non-negative integer or an integer and 0. If these conditions are not satisfied, an error message is given. 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.) If the initial conditions are not satisfied or *ArgNo* is out of range, the call merely fails.

?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) =.. [product, 0, N, N-1]

N-1 =.. [-, N, 1]

product =.. [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)`

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

```
name(product,[112,114,111,100,117,99,116])

i.e. name(product,"product")

name(1976,[49,57,55,54])

name('1976',[49,57,55,54])

name((:-),[58,45])
```

If *Const* is uninstantiated, *CharList* must be instantiated to a list of ASCII character codes. If *CharList* can be interpreted as a number, *Const* is unified with that number, otherwise with the atom whose name is *CharList*. 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)`
The same as `name(?Const,?CharList)`, but *Const* is constrained to be an atom.

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

`call(+Term)`

`incore(+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*.

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

+Term (where *Term* is a variable) Exactly the same as `call(Term)`.

4.8 Miscellaneous Predicates

`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) :-
        recorda(.,X,Ref),
        instance(Ref,Y),
        erase(Ref).
```

`numbervars(?Term,+N,?M)`
Unifies each of the variables in term *Term* with a special term, so that `write(Term)` (or `writeq(Term)`) 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`.

`setarg(+ArgNo,+CompoundTerm,?NewArg)`
Replace destructively argument *ArgNo* in *CompoundTerm* with *NewArg*. The assignment is undone on backtracking.

`undo(+Term)`
The goal `call(Term)` is executed on backtracking.

4.9 Modification of the Program

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

`assert(`+*Clause*`)`

The current instance of *Clause* is interpreted as a clause and is added to the current interpreted program (with new private variables replacing any uninstantiated variables). The position of the new clause within the procedure concerned is implementation-defined.

`asserta(`+*Clause*`)`

Like `assert/1`, except that the new clause becomes the *first* clause for the procedure concerned.

`assertz(`+*Clause*`)`

Like `assert/1`, except that the new clause becomes the *last* clause for the procedure concerned.

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

Head must be bound to a non-variable term, and the current interpreted program is searched for a clause whose head matches *Head*. The head and body of those clauses are unified with *Head* and *Body* respectively. If one of the clauses is a unit clause, *Body* will be unified with `true`.

`retract(`+*Clause*`)`

The first clause in the current interpreted program that matches *Clause* is erased. *Clause* must be initially instantiated; it is first translated into a clause, which is then matched to the database. This means that

```
retract((p(X) :- Y))
```

matches only clauses whose body is `call(`*Y*`)`, and not all clauses for `p(`*X*`)`. The predicate may be used in a non-determinate fashion, i.e. it will successively retract clauses matching the argument through backtracking.

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*. The predicate definition is retained.

`abolish(`+*Spec*`)`

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

Erase all clauses of the predicate specified by the predicate spec *Spec* or *Name*/*Arity*. *Spec* may also be a list of predicate specs. The predicate definition and all associ-

ated information such as spy-points is also erased. This is only legal for user defined predicates. For user defined predicates that are called by built-in predicates (e.g. `term_expansion/2`), any clauses are erased but the predicate definition is retained.

4.10 Internal Database

The predicates described in this section are primarily concerned with providing efficient means of performing operations on large quantities of data. Most users will not need to know about these predicates.

These predicates make it possible to 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. This special identifier is actually a pointer into the database and needs to be treated with some caution. For safety reasons it is not possible to store the pointers themselves in the database: the term to which the pointer referred might be erased.

Note the difference between this facility and that provided by `assert/1` and related predicates: the latter actually alter the running program. Also the recording predicates use an extra level of indirection, the *Key*, which allows greater flexibility.

`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. The key must be given, and may be an atom, number or compound term. If it is a compound term, only the principal functor is significant.

`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.

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

The term *Term* is recorded in the internal database as the last item for the key *Key*,

where *Ref* is its implementation-defined identifier. The key must be given, and only its principal functor is significant.

`erase(+Ref)`
: The recorded item (or interpreted clause (see section 4.10 [Database], page 54)) 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 all keys for currently recorded terms through backtracking.

Like recorded terms, the clauses of an interpreted program also have a unique implementation-defined identifier. A new set of the predicates (see section 4.9 [Modify Prog], page 53) is given below, each predicate having 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,-Ref)`
: Equivalent to `assert/1` where *Ref* is the implementation-defined identifier of the clause asserted.

`asserta(+Clause,-Ref)`
: Equivalent to `asserta/1` where *Ref* is the implementation-defined identifier of the clause asserted.

`assertz(+Clause,-Ref)`
: Equivalent to `assertz/1` where *Ref* is the implementation-defined identifier of the clause asserted.

`clause(?Head,?Body,?Ref)`
: Equivalent to `clause/2` where *Ref* is the implementation-defined term which uniquely identifies the clause concerned. If *Ref* is not given at the time of the call, *Head* must be instantiated to a non-variable term. Thus this predicate can have two different modes of use, depending on whether the identifier of the clause is known or unknown.

4.11 Sets

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 provable, where that set is non-empty.* The term *Goal* specifies a goal or goals as in `call(`*Goal*`)`. *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 44)f. 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.) For example, the call:

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

might produce two alternative solutions via backtracking:

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

The call:

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

would produce the single result:

```
X = [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 considerable time and space in execution.

X^P The interpreter recognises this as meaning *there exists an X such that P is true*, and treats it as equivalent to `call(P)`. The use of this explicit existential quantifier outside the `setof/3` and `bagof/3` constructs is superfluous.

`findall(`*?Template*`,`*+Goal*`,`*?Bag*`)`
A special case of `bagof/3`, where all free variables in the goal are taken to be existentially quantified.

4.12 Interface to Foreign Language Functions

Functions written in the C language (or any other language that uses the same calling conventions) may be called from Prolog. 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.

`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:

```
move_cursor(5, 23).
```

would translate into the C call `pos_cursor(5,23);`.

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

Load (link) *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. Example:

```
load_foreign_files(['terminal.o'], []).
```

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 C function will be passed a `long`, type converted to `(char *)`.

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

C: *TypeName* `*`

The argument should be instantiated to an integer. The call will otherwise fail. The C function will be passed a `long`, 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 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 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 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 will be converted to a Prolog integer and unified with the Prolog argument.

4.13 Debugging

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

Unifies *OldState* with the current state of the *Action on unknown procedures* flag,

and sets the flag to *NewState*. This flag determines whether or not the system is to catch calls to predicates which are undefined (see section 1.6 [Undefined Predicates], page 10). The possible states of the flag are:

`trace` Causes calls to predicates with no definition 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.

`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. debugging.

`trace` The debugger is switched on with tracing enabled. See section 2.3 [Trace], page 17.

`leash(`*+Mode*`)`
Leashing Mode is set to *Mode*. See section 2.3 [Trace], page 17.

`spy` *+Spec*
Spy-points are placed on all the procedures given by *Spec*. See section 2.4 [Spy-Point], page 18.

`nospy` *+Spec*
Spy-points are removed from all the procedures 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.

4.14 Definite Clause Grammars

Prolog's grammar rules provide a convenient notation for expressing definite clause grammars [Colmerauer 75] [Pereira & Warren 80]. 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 ASCII 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 `;` 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 ASCII 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 overrules 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 by the clause `'C'([X|S], X, S).`

4.15 Environmental

`halt` Causes an irreversible exit from Prolog back to the Monitor.

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

Declares the atom *Name* to be an operator of the stated *Type* and *Precedence* (see section 5.4 [Operators], page 82). *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` Calls the command interpreter recursively. See section 1.9 [Nested], page 12.

`abort` Aborts the current execution. See section 1.9 [Nested], page 12.

`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.

`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).

`save_program(`*+File*`)`

The system saves the currently defined predicates into file *File*. When it is restored, Prolog will reinitialise itself. See section 1.10 [Saving], page 13.

`restore(+File)`

The system is returned to the system state previously saved to file *File*. See section 1.10 [Saving], page 13.

`reinitialise`

This predicate can be used to force the initialisation behaviour to take place at any time.

`maxdepth(+Depth)`

Positive integer *Depth* specifies the maximum depth, i.e. the maximum number of nested interpreted calls, beyond which the interpreter will induce an automatic failure. 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 procedures are not included in the computation of the depth. The interpreter will check for maximum depth only if the debugger is switched on.

`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.

`garbage_collect`

Preform 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:

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

`global_stack`
[*size used, free*]

`local_stack`
[*size used, free*]

`core`

`memory` [*size used*,0]

`heap`

`program` [*size used*,0]

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

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

`trail` [*size used*, *free*]

`choice` [*size used*, *free*]

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 matched to *Old*; the atom bound to New specifies the new prompt. In particular, the goal

```
prompt(X, X)
```

matches the current prompt to *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). Notice 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/1`, 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*`)`

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 the program arguments supplied when the current SICStus process was started. For example, if SICStus were invoked with

```
% sicstus hello world
```

then *Args* will be unified with

```
[hello,world]
```

`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. The control is returned to Prolog upon termination of the shell..

`shell(+Command)`
: Pass *Command* to a new UNIX shell for execution.

`system(+Command)`
: Pass *Command* to a new UNIX `sh` process for execution.

4.16 Compatibility

This section lists predicates which have no effect in SICStus Prolog, but which exist for compatibility with other Prologs.

```
'LC'
'NOLC'
character_count(S,N)

current_functor(X,Y)

current_module(M)

current_module(M,F)

ensure_loaded(F)

gcguide(F,O,N)

help(T)
line_count(S,N)
```

`line_position(S,N)`

`log`

`manual`

`manual(X)`

`module(M)`

`no_style_check(A)`

`nolog`

`plsys(X)` This predicate has one meaningful use: if called as in

```
| ?-  plsys(mktemp(Template, Filename))
```

then *Filename* will be unified with a unique filename constructed from the atom *Template*. This is an interface to `mktemp(3)` in the Unix C library.

`restore(S,X)`

`revive(X,Y)`

`source_file(F)`

`source_file(P,F)`

`stream_position(S,P)`

`stream_position(S,O,N)`

`style_check(A)`

`trimcore`

`use_module(F)`

`use_module(F,I)`

`vms(T)`

5. 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. A much fuller introduction to Prolog may be found in [Sterling & Shapiro 86]. For a more general introduction to the field of Logic Programming see [Kowalski 79].

5.1 Syntax, Terminology and Informal Semantics

5.1.1 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 9, of which base 2 (binary) and base 8 (octal) are probably the most useful. 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 ASCII 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: `+-*/\^<>=`~:.?@#$&`
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 `[ ]` `{ }` 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.

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 characterised 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

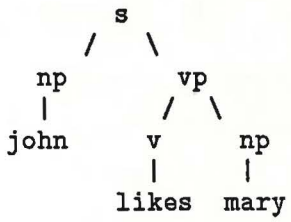
```
point(X, Y, Z)
```

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



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

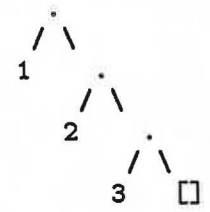
```
+(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)
```

The use of operators is described fully in Section I.4 below.

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



which could be written, using the standard syntax, as

```
.(1,.(2,.(3,[])))
```

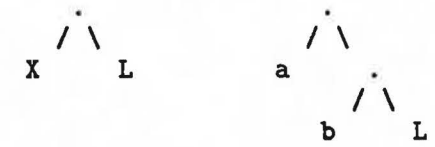
but which is normally written, in a special list notation, as

```
[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



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 ASCII 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]
```

5.1.2 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 is called a *predicate*. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic program consists simply of a sequence of statements called *sentences*, which are analogous to sentences 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

P is true.

and procedurally as

Goal P is 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 *procedure* for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a 3-ary predicate `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 `.`). The rule used to disambiguate terms is that a `.` followed by a layout-character is regarded as a sentence terminator, where a layout-character is defined to be any character less than or equal to ASCII 32 (this includes space, tab, newline and all control characters).

5.2 Declarative and Procedural Semantics

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

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

For example, if a program contains the 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**.

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 his 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.

> To execute a goal, the system searches forwards from the beginning of the program for the first clause whose head matches or unifies with the goal. The unification process [Robinson 1965] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it backtracks, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

For example, if we execute the goal expressed by the 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 goal produced, which is the recursive procedure call

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

Again this goal matches the first clause, instantiating *X1* to [b|*X2*], and yielding the new goal

```
concatenate(X2, Y, [])
```

Now this 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.

5.2.1 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 [Lloyd 85] 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 cyclic terms without looping. Loops in the printer can be interrupted by typing ^C.

5.3 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.

E.g.

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

This procedure 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 procedure can also be used to extract elements from a list, as in

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

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

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

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 procedure.

```
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 6 [Example Intro], page 95.

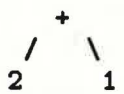
- Write each clause as a self-contained logic rule which just defines the truth of goals which match its head. Then add cuts to remove any fruitless alternative computation paths that may tie up store.
- Cuts are usually placed right after the head, sometimes preceded by simple tests.
- Cuts are hardly ever needed in the last clause of a procedure.

5.4 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



and *not* the number 3. The addition would only be performed if the structure were passed as an argument to an appropriate procedure such as `is/2` (see section 4.2 [Arithmetic], page 42).

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 brackets. 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 brackets, 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 bracketed (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 9 [Standard Operators], page 115).

For example, the standard operators + and - are declared by

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

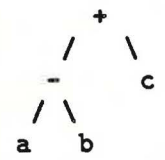
so that

```
a-b+c
```

is valid syntax, and means

```
(a-b)+c
```

i.e.



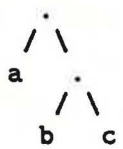
The list functor `'.'` is not a standard operator, but we could declare it thus:

```
:- op(900, xfy, '.').
```

Then

```
a.b.c
```

would represent the structure



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 brackets 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 bracket the expression `P :- Q` in

```
assert((P :- Q))
```

5.5 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:

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.

3. If a prefix operator is written without an argument, as an ordinary atom, the atom is treated as an expression of the same precedence as the prefix operator, and must therefore be bracketed where necessary. Thus the brackets are necessary in

```
X = (?-)
```

5.6 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. The first character after the % may not be a (or a), because the symbols %(and %) are reserved.
2. The symbol /* followed by any sequence of characters (including new lines) up to */.

5.7 Full Prolog Syntax

A Prolog program consists of a sequence of *sentences*. Each sentence is a Prolog *term*. How terms are interpreted as sentences is defined below (see section 5.7.2 [Sentence], page 88). 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 5.7.3 [Term Token], page 89). 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 5.7.4 [Token String], page 90). But we start by describing the notation used in the formal definition of Prolog syntax (see section 5.7.1 [Syntax Notation], page 87).

5.7.1 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 61).

7. In the section describing the syntax of terms and tokens (see section 5.7.3 [Term Token], page 89) particular tokens of the category name are written thus: **name**, while tokens which are individual punctuation characters are written literally.

5.7.2 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 an number or variable }

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 )
                    | 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 }
```

5.7.3 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)
                     |  op(N,fy)
                     |  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
                      { where name is not a prefix operator }

integer         --> natural-number
                  | - natural-number

float           --> unsigned-float
                  | - unsigned-float

functor         --> name
```

5.7.4 Syntax of Tokens as Character Strings

```
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...
                     |  digit ' digit...
                           { where digit > 0 }
                     |  0 ' char

base                --> digit...

unsigned-float      --> simple-float
                     |  simple-float exp exponent

simple-float        --> digit...  .  digit...

exp                 --> e  |  E

exponent            --> digit... | - digit... | + digit...

variable            --> underline ?alpha...

variable            --> capital-letter ?alpha..

string              --> " ?string-item... "

string-item         --> char  { other than " }
                     |  ""

space               --> layout-char...

comment             --> /* ?char... */
                           { where ?char... must not contain */ }
                     |  % rest-of-line

rest-of-line        --> newline
                     |  ?not-end-of-line... newline

not-end-of-line     --> { any character except newline }

newline             --> { LFD }

full-stop           --> . layout-char

char                --> { any ASCII character, i.e. }
                        layout-char
                     |  alpha
                     |  symbol-char
                     |  solo-char
                     |  punctuation-char
                     |  quote-char
```

```
layout-char       --> { any ASCII character code up to 32,
                        includes SPS, RET and LFD }

alpha             --> letter | digit | underline

letter            --> capital-letter | small-letter

capital-letter    --> { any character from the list
                          ABCDEFGHIJKLMNOPQRSTUVWXYZ }

small-letter      --> { any character from the list
                          abcdefghijklmnopqrstuvwxyz }

digit             --> { any character from the list
                          012346789 }

symbol-char       --> { any character from the list
                          +-*/\^<>=`~:.?@#$& }

solo-char         --> { any character from the list
                          ;! }

punctuation-char  --> { any character from the list
                          ()[]{},| }

quote-char        --> { any character from the list
                          '" }

underline         --> { the character _ }
```

5.7.5 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 bracketed 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. Programming Examples

Some simple examples of Prolog programming are given below. They exemplify typical applications of Prolog. We are trying to convey a flavour 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 at proper places (see section 5.3 [Cut], page 80).

6.1 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).
```

6.2 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
```

6.3 Quick-Sort

The goal `qsort(`*L*`, [], `*R*`)` is true if list *R* is a sorted version of list *L*. More generally, the goal `qsort(`*L*`, `*R0*`, `*R*`)` is true if list *R* consists of the members of list L sorted into order, followed by the members of list *R0*. The algorithm used is a variant of Hoare's *Quick Sort*.

```
:- mode qsort(+, +, -).
:- mode partition(+, +, -, -).

qsort([], R, R).
qsort([X|L], R0, R) :-
        partition(L, X, L1, L2),
        qsort(L2, R0, R1),
        qsort(L1, [X|R1], R).

partition([], _, [], []).
partition([X|L], Y, [X|L1], L2) :- X =< Y, !,
        partition(L, Y, L1, L2).
partition([X|L], Y, L1, [X|L2]) :- X > Y,
        partition(L, Y, L1, L2).
```

6.4 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*.

```
:- mode d(+, +, -).
:- op(300, xfy, **).

d(X, X, 1) :- atomic(X), !.
d(C, X, 0) :- atomic(C), !.
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).
```

6.5 Mapping a List of Items into a List of Serial Numbers

The goal `serialise(`*L1*`,` *L2*`)` is true if *L2* is a list of serial numbers corresponding to the members of list *L1*, where the members of *L1* are numbered (from 1 upwards) in order of increasing size. e.g. `?- serialise([1,9,7,7],` *X*`).` gives '*X* = `[1,3,2,2]`'.

```
serialise(Items, SerialNos) :-
        pairlists(Items, SerialNos, Pairs),
        arrange(Pairs, Tree),
        numbered(Tree, 1, N).

pairlists([], [], []).
pairlists([X|L1], [Y|L2], [pair(X,Y)|L3]) :-
        pairlists(L1, L2, L3).

arrange([], void).
arrange([X|L], tree(T1,X,T2)) :-
        split(L, X, L1, L2),
        arrange(L1, T1),
        arrange(L2, T2).

split([], _, [], []).
split([X|L], X, L1, L2) :- !,
        split(L, X, L1, L2).
split([X|L], Y, [X|L1], L2) :- before(X, Y), !,
        split(L, Y, L1, L2).
split([X|L], Y, L1, [X|L2]) :- before(Y, X),
        split(L, Y, L1, L2).

before(pair(X1,Y1), pair(X2,Y2)) :- X1 < X2.

numbered(void, N, N).
numbered(tree(T1,pair(X,N1),T2), N0, N) :-
        numbered(T1, N0, N1),
        N2 is N1+1,
```

```
        numbered(T2, N2, N).
```

6.6 Use of Meta-Predicates

This example illustrates the use of the meta-predicates `var/1`, `arg/3`, and `functor/3`. 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), [U,V,X,DU,V,U,DV], [])
```

```
variables(X, [X|L], L) :- var(X), !.
variables(T, L0, L) :-
        functor(T, _, A),
        variables(0, A, T, L0, L).

variables(A, A, _, L, L) :- !.
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).
```

6.7 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 unary predicate `my_clause`, 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.

6.8 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].
```

7. Installation Dependencies

7.1 Getting Started

To start SICStus issue the shell command:

```
% sicstus options arguments
```

The only option currently available is to raise the size limit of terms which may be stored using `recorda/3`, `asserta/1`, `setof/3`, and related predicates. The default limit is 256. The "size" is approximately defined as the number of distinct variables in a term plus its maximum nesting level. If the limit is exceeded during an execution, Prolog will print the message

```
ERROR: term too large in assert or record
```

and the execution will be aborted.

To raise the limit to 1000, issue the shell command:

```
% sicstus -x 1000 arguments
```

If given, the *arguments* can be retrieved from Prolog by `unix(argv(?Args))`.

To start SICStus from a saved state *file*, issue the shell comand:

```
% file options arguments
```

8. Summary of Built-In Predicates

`abolish(+Preds)`
Make the predicate(s) specified by *Preds* undefined.

`abolish(+Atom,+Arity)`
Make the predicate specified by *Atom/Arity* undefined.

`abort` Abort execution of the current directive.

`absolute_file_name(+RelativeName,?AbsoluteName)`
AbsoluteName is the full path of *RelativeName*.

`ancestors(?Goals)`
The ancestor list of the current clause is *Goals*.

`arg(+ArgNo,+Term,?Arg)`
Argument *ArgNo* of term *Term* is *Arg*.

`assert(+Clause)`
Assert clause *Clause*.

`assert(+Clause,-Ref)`
Assert clause *Clause*, reference *Ref*.

`asserta(+Clause)`
Assert *Clause* as first clause.

`asserta(+Clause,-Ref)`
Assert *Clause* as first clause, reference *Ref*.

`assertz(+Clause)`
Assert *Clause* as last clause.

`assertz(+Clause,-Ref)`
Assert *Clause* as last clause, reference *Ref*.

`atom(?X)` *X* is an atom.

`atom_chars(?Atom,?CharList)`
The name of atom *Atom* is string *CharList*.

`atomic(?X)`
X is an atom or number.

`bagof(?Template,+Goal,?Bag)`
The bag of instances of *Template* such that *Goal* is provable is *Bag*.

`break` Break at the next interpreted procedure call.

`'C'(?S1,?Terminal,?S2)`
(grammar rules) *S1* is connected by the terminal *Terminal* to *S2*.

`call(+Term)`
Execute the procedure call *Term*.

`call(+Term,?Vars)`
Execute the procedure call *Term*. Any subgoals are suspended on the variables in *Vars*.

`character_count(X,Y)`
Dummy routine for compatibility.

`clause(+Head,?Body)`
There is an interpreted clause, head *Head*, body *Body*.

`clause(?Head,?Body,?Ref)`
There is an interpreted clause, head *Head*, body *Body*, ref *Ref*.

`close(+File)`
Close stream *File*.

`compare(?Op,?Term1,?Term2)`
Op is the result of comparing terms *Term1* and *Term2*.

`compile(+File)`
Compile in-core the procedures in text file(s) *File*.

`consult(+File)`
Update the program with interpreted clauses from file(s) *File*.

`copy_term(?Term,?CopyOfTerm)`
CopyOfTerm is an independent copy of *Term*.

`current_atom(?Atom)`
One of the currently defined atoms is *Atom*.

`current_functor(X,Y)`
Dummy routine for compatibility.

`current_input(?Stream)`
Stream is the current input stream.

`current_key(?KeyName,?KeyTerm)`
KeyName is the atom or number which is the name of *KeyTerm*.

`current_module(?Module)`
Dummy routine for compatibility.

`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)`
A current predicate is named *Name*, most general goal *Head*.

`current_stream(?FileName,?Mode,?Stream)`
FileName and *Mode* are associated with *Stream*.

`debug` Switch on debugging.

`debugging`
: Output debugging status information.

`depth(?Depth)`
: The current invocation depth is *Depth*.

`dif(?X,?Y)`
: The terms *X* and *Y* are different.

`display(?Term)`
: Display term *Term* on the terminal.

`ensure_loaded(File)`
: Dummy routine for compatibility.

`erase(+Ref)`
: Erase the clause or record, reference *Ref*.

`expand_term(+Term1,?Term2)`
: Term *Term1* is a shorthand which expands to term *Term2*.

`fail`

`false`
: Backtrack immediately.

`fcompile(+File)`
: Compile file-to-file procedures in text file(s) *File*.

`fileerrors`
: Enable reporting of file errors.

`findall(?Template,+Goal,?Bag)`
: The bag of instances of *Template* such that an instance of *Goal* is provable is *Bag*.

`float(?X)`
: *X* is a float.

`flush_output(+Stream)`
: Flush the buffers associated with *Stream*.

`foreign(+CFunctionName, +Predicate)`

`foreign(+CFunctionName, +Language, +Predicate)`
: *User defined*, they tell Prolog how to translate *Predicate* to call of *CFunctionName*.

`foreign_file(+ObjectFile,+Functions)`
: *User defined*, tells Prolog that foreign functions *Functions* are in file *ObjectFile*.

`format(+Format,+Arguments)`
: Write *Arguments* according to *Format* on the current output.

`format(+Stream,+Format,+Arguments)`
: Write *Arguments* according to *Format* on the stream *Stream*.

`freeze(+Goal)`
: Suspend *Goal* until *Goal* is ground.

`freeze(?X,+Goal)`
Suspend *Goal* until `nonvar(X)`.

`frozen(-Var,?Goal)`
The goal *Goal* is suspended on the variable *Var*.

`functor(?Term,?Name,?Arity)`
The principal functor of term *Term* has name *Name*, arity *Arity*.

`garbage_collect`
Perform a garbage collection.

`gc` Enable garbage collection.

`gcguide(X,Y,Z)`
Dummy routine for compatibility.

`get(?C)` The next non-blank character from the current input is *C*.

`get(+Stream,?C)`
The next non-blank character from the stream *Stream* is *C*.

`get0(?C)` The next character fro the current input is *C*.

`get0(+Stream,?C)`
The next character from the stream *Stream* is *C*.

`halt` Halt Prolog, exit to the monitor.

`help` Print a help message.

`if(+P,+Q,+R)`
If *P* then *Q* else *R*, exploring all solutions of *P*.

`incore(+Term)`
Execute the procedure call *Term*.

`instance(+Ref,?Term)`
A most general instance of the record reference *Ref* is *Term*.

`integer(?X)`
X is 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*.

`'LC'` Dummy routine for compatibility.

`leash(+Mode)`
Set leashing mode to *Mode*.

`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(X,Y)`
Dummy routine for compatibility.

`line_position(X,Y)`
Dummy routine for compatibility.

`listing` List the current interpreted program.

`listing(+A)`
List the interpreted procedure(s) specified by *A*.

`load(+File)`
Load compiled object file(s) *File* into Prolog.

`load_foreign_files(+ObjectFiles,+Libraries)`
Load (link) files *ObjectFiles* into Prolog.

`log` Dummy routine for compatibility.

`manual`

`manual(+Topic)`
Dummy routines for compatibility.

`maxdepth(+Depth)`
Limit invocation depth to *Depth*.

`module(?Module)`
Dummy routine for compatibility.

`name(?Const,?CharList)`
The name of atom or number *Const* is string *CharList*.

`nl` Output a new line on the current output stream.

`nl(+Stream)`
Output a new line on stream *Stream*.

`no_style_check(X)`
Dummy routine for compatibility.

`nodebug` Switch off debugging.

`nofileerrors`
Disable reporting of file errors.

`nogc` Disable garbage collection.

`'NOLC'` Dummy routine for compatibility.

`nolog` Dummy routine for compatibility.

`nonvar(?X)`
X is a non-variable.

`nospy +Spec`
Remove spy-points from the procedure(s) specified by *Spec*.

`nospyall` Remove all spy-points.

`notrace` Switch off debugging.

`number(?X)`
: *X* is a number.

`number_chars(?Number,?CharList)`
: The name of number *Number* is string *CharList*.

`numbervars(?Term,+N,?M)`
: Number the variables in term *Term* from *N* to *M*-1.

`op(+Precedence,+Type,+Name)`
: Make atom *Name* an operator of type *Type* precedence *Precedence*.

`open(+FileName,+Mode,-Stream)`
: Open file *FileName* in mode *Mode* as stream *Stream*.

`open_null_stream(-Stream)`
: Open an output stream to the null device.

`otherwise`
: Succeed.

`phrase(+Phrase,?List)`

`phrase(+Phrase,?List,?Remainder)`
: List *List* can be parsed as a phrase of type *Phrase*. The rest of the list is *Remainder* or empty.

`plsys(mktemp(+Template, -Filename))`
: *Filename* is a unique filename constructed from the atom *Template*.

`portray(+Term)`
: *User defined*, tells `print/1` what to do.

`predicate_property(?Head,?Prop)`
: *Head* is the most general goal of a currently defined predicate that has the property *Prop*.

`print(?Term)`
: Portray or else write the term *Term* on the current output.

`print(+Stream,?Term)`
: Portray or else write the term *Term* on the stream *Stream*.

`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)`
: Change the prompt from *Old* to *New*.

`put(+C)` The next character sent to the current output is *C*.

`put(+Stream,+C)`
The next character sent to the stream *Stream* is *C*.

`read(?Term)`
Read term *Term* from the current input.

`read(+Stream,?Term)`
Read term *Term* from the stream *Stream*.

`reconsult(+File)`
Update the program with interpreted clauses from file *File*.

`recorda(+Key,?Term,-Ref)`
Make term *Term* the first record under key *Key*, reference *Ref*.

`recorded(+Key,?Term,?Ref)`
Term *Term* is recorded under key *Key*, reference *Ref*.

`recordz(+Key,?Term,-Ref)`
Make term *Term* the last record under key *Key*, reference *Ref*.

`reinitialise`
Initialisation.

`repeat` Succeed repeatedly.

`restore(+File)`
Restore the state saved in file *File*.

`restore(X,Y)`
Dummy routine for compatibility.

`retract(+Clause)`
Erase the first interpreted clause of form *Clause*.

`retractall(+Head)`
Erase all clauses whose head matches *Head*.

`revive(X,Y)`
Dummy routine for compatibility.

`save(+File)`
Save the current state of Prolog in file *File*.

`save(+File,?Return)`
As `save(File)` but *Return* is 0 first time, 1 after a restore.

`save_program(+File)`
Save the current state of the Prolog data base in file *File*.

`see(+File)`
Make file *File* the current input stream.

`seeing(?File)`
The current input stream is named *File*.

`seen` Close the current input stream.

`set_input(`*+Stream*`)`
Set the current input to *Stream*.

`set_output(`*+Stream*`)`
Set the current output to *Stream*.

`setarg(`*+ArgNo*`,`*+CompoundTerm*`,`*?NewArg*`)`
Replace destructively argument *ArgNo* in *CompoundTerm* with *NewArg* and undo on backtracking.

`setof(`*?Template*`,`*+Goal*`,`*?Set*`)`
The set of instances of *Template* such that *Goal* is provable is *Set*.

`skip(`*+C*`)`
Skip characters from the current input until after character *C*.

`skip(`*+Stream*`,`*+C*`)`
Skip characters from *Stream* until after character *C*.

`sort(`*+List1*`,`*List2*`)`
The list *List1* sorted into order yields *List2*.

`source_file(X)`

`source_file(X,Y)`
Dummy routines for compatibility.

`spy` *+Spec*
Set spy-points on the procedure(s) specified by *Spec*.

`statistics`
Output 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*.

`stream_position(X,Y)`

`stream_position(X,Y,Z)`
Dummy routines for compatibility.

`style_check(X)`
Dummy routine for compatibility.

`subgoal_of(`*?Goal*`)`
An ancestor goal of the current clause is *Goal*.

`tab(`*+N*`)` Send *N* spaces to the current output.

`tab(`*+Stream*`,`*+N*`)`
Send *N* spaces to the stream *Stream*.

`tell(+File)`
: Make 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.

`told` Close the current output stream.

`trace` Switch on debugging and start tracing immediately.

`trimcore` Dummy routine for compatibility.

`true` Succeed.

`ttyflush` Transmit all outstanding terminal output.

`ttyget(?C)`
: The next non-blank character input from the terminal is *C*.

`ttyget0(?C)`
: The next character input from the terminal is *C*.

`ttynl` Output a new line on the terminal.

`ttyput(+C)`
: The next character output to the terminal is *C*.

`ttyskip(+C)`
: Skip over terminal input until after character *C*.

`ttytab(+N)`
: Output *N* spaces to the terminal.

`undo(+Term)`
: The goal `call(Term)` is executed on backtracking.

`unix(+Term)`
: Interact with the operating system.

`unknown(?OldState,?NewState)`
: Change action on unknown procedures from *OldState* to *NewState*.

`use_module(X)`

`use_module(X,Y)`
: Dummy routines for compatibility.

`user_help`
: *User defined*, tells `help/0` what to do.

`var(X)` *X* is a variable.

`version` Displays introductory and/or system identification messages.

`version(+Message)`
: Adds the atom *Message* to the list of introductory messages.

`vms(X)` Dummy routine for compatibility.

`write(?`*Term*`)`
Write the term *Term* on the current output.

`write(+`*Stream*`,?`*Term*`)`
Write the term *Term* on the stream *Stream*.

`write_canonical(?`*Term*`)`
Write *Term* on the current output so it may be read back.

`write_canonical(+`*Stream*`,?`*Term*`)`
Write *Term* on the stream *Stream* so it may be read back.

`writeq(?`*Term*`)`
Write the term *Term* on the current output, quoting names where necessary.

`writeq(+`*Stream*`,?`*Term*`)`
Write the term *Term* on the stream *Stream*, quoting names where necessary.

`!` Cut any choices taken in the current procedure.

`(+`P`,+`Q`)` P and Q.

`(+`P`;+`Q`)` P or 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.

`\+ +`P Goal P is not provable.

`?`X`^+`P There exists an X such that P is provable.

`+`X `=:= +`Y
As numeric values, X is equal to Y.

`+`X `=\= +`Y
As numeric values, X is not equal to Y.

`+`X`<+`Y As numeric values, X is less than Y.

`+`X`=<+`Y As numeric values, X is less than or equal to Y.

`+`X`>+`Y As numeric values, X is greater than Y.

`+`X`>=+`Y As numeric values, X is greater than or equal to Y.

`?`X`=?`Y Terms X and Y are equal (i.e. unified).

`?`*Term* `=..` `?`*List*
The functor and arguments of term *Term* comprise the list *List*.

`?`*Term1* `==` `?`*Term2*
Terms *Term1* and *Term2* are strictly identical.

`?`*Term1* `\==` `?`*Term2*
Terms *Term1* and *Term2* are not strictly identical.

?Term1 **@<** *?Term2*

Term *Term1* precedes term *Term2*.

?Term1 **@=<** *?Term2*

Term *Term1* precedes or is identical to term *Term2*.

?Term1 **@>** *?Term2*

Term *Term1* follows term *Term2*.

?Term1 **@>=** *?Term2*

Term *Term1* follows or is identical to term *Term2*.

[*+File*|*+Files*]

Perform consult(s) on the listed files.

9. Standard Operators

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic,
                    multifile, parallel, wait ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).      /* See note below */
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                  =:=, =\=, <, >, =<, >= ]).
:- 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.

References

[Clocksin & Mellish 81]
Clocksin W.F. and Mellish C.S.
Programming in Prolog.
Springer-Verlag, 1981.

[Colmerauer 75]
Colmerauer A.
Les Grammaires de Metamorphose.
Technical Report, Groupe d'Intelligence Artificielle, Marseille- Luminy, November, 1975.
Appears as "Metamorphosis Grammars" in "Natural Language Communication with Computers", Springer Verlag, 1978.

[Kowalski 74]
Kowalski R.A.
Logic for Problem Solving.
DCL Memo 75, Dept of Artificial Intelligence, University of Edinburgh, March, 1974.

[Kowalski 79]
Kowalski R.A.
Artificial Intelligence: Logic for Problem Solving.
North Holland, 1979.

[Pereira & Warren 80]
Pereira F.C.N. and Warren D.H.D.
Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks.
Artificial Intelligence 13:231-278, 1980.
Also available as Research Paper 116, Dept of Artificial Intelligence, University of Edinburgh.

[Roussel 75]
Roussel P.
Prolog : Manuel de Reference et d'Utilisation
Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[Sterling & Shapiro 86]
Sterling L. and Shapiro E.
The Art of Prolog
The MIT Press, Cambridge MA, 1986.

[van Emden 75]
van Emden M.H.

Programming with Resolution Logic.
Technical Report CS-75-30, Dept of Computer Science, University of Waterloo, Canada, November, 1975.

[Warren 83]
Warren D.H.D.
An Abstract Prolog Instruction Set.
Tech. Note 309, SRI International
Menlo Park, CA, 1983.

Predicate Index

A

B

C

D

E

F

G

H

T

U

V

W

Concept Index

P

Q

R

S

T

U

W

Table of Contents

SICS Research Reports

Box 1263
S-164 28 Kista
Sweden

R 86001 Yoeli, M. and B. Pehrson, *Behavior-Preserving Reductions of Communicating System Nets*, 1986

R 86002 Hausman, B., *A Simulator of the OR-Parallel Token Machine*, 1986

R 86003 Ciepielewski, A. and B. Hausman, *Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs*, 1986

R 86004 Karjoth, G., P. Sjödin and S. Weckner, *A Sophisticated Environment for Protocol Simulation and Testing*, 1987

R 86005 Hallnäs, L., *On the Interpretation of Inductive Definitions*, 1986 (Not available, see R 86005B)

R 86005B Hallnäs, L., *Partial Inductive Definitions*, 1987 (Revised version of R 86005)

R 86006 Mohamed Ali, K., A., *OR-Parallel Execution of Prolog on a Multi-Sequential Machine*, 1986 (Not available, see R 86006B)

R 86006B Mohamed Ali, K., A., *OR-Parallel Execution of Prolog on a Multi-Sequential Machine*, 1986 (Revised version of R 86006)

R 86007 Wærn, A., *Process Models of Logic Programs: a Comparison*, 1987

R 86008 Mathieu, P., *On the Learning of Functional Dependencies in Deductive Databases*, 1986 (an extended abstract)

R 86009 Appleby, K., S. Haridi and D. Sahlin, *Garbage Collection for Prolog Based on WAM*, 1987

R 86010 Hallnäs, L., *Generalized Horn Clauses*, 1986 (No longer distributed; see R 88005)

R 86011 Carlsson, M., *On Compiling Indexing and Cut for the WAM*, 1987

R 86012 Carlsson, M., *An implementation of "dif" and "freeze" in the WAM*, 1986

R 86013 Elshiewy, N., *Time, Clocks and Committed Choice Parallelism for Logic Programming of Real Time Computations*, 1987

R 87001 Mohamed Ali, K., A., *A Method for Implementing Cut in Parallel Execution of Prolog*, 1987

R 87002 Rayner, M. and S. Janson, *Epistemic Reasoning, Logic Programming, and the Interpretation of Questions*, 1987

R 87003 Gunningberg, P., *Innovative Communication Processors: A Survey*, 1987

R 87004 Gadener, C., M. Lidén and J. Riboe, *ORPWAM An Implementation Study, Part 1*, 1987

R 87005 Mohamed Ali, K., A. and S. Haridi, *Global Garbage Collection for Distributed Heap Storage Systems*, 1987

R 87006 Hausman, B., A. Ciepielewski and S. Haridi, *OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors*, 1987

R 87007 Holmgren, F. and A. Wærn, *A Scheme for Compiling GHC to Prolog Using Freeze*, 1987

R 87008 Sahlin, D., *Making Garbage Collection Independent of the Amount of Garbage*, 1987 (Appendix to SICS Research Report R 86009)

R 87009 Sjödin, P., *Optimizing Protocol Implementations for Performance - A Case Study*, 1987

R 87010 Franzén, T., *Algorithmic Aspects of Intuitionistic Propositional Logic*, 1987 (Not available, see R 87010B)

R 87010B Franzén, T., *Algorithmic Aspects of Intuitionistic Propositional Logic*, 1988 (Revised version of R 87010)

R 88001 Rayner, M., Å. Hugosson and G. Hagert, *Using a Logic Grammar to Learn a Lexicon*, 1988

R 88002 Franzén, T., *Logic Programming and the Intuitionistic Sequent Calculus*, 1988

R 88003 Rayner, M. and Å. Hugosson, *Reasoning about Procedural Programs in a Chess Ending*, 1988

R 88004 Wærn, A., *An Implementation Technique for the Abstract Interpretation of Prolog*, 1988.

R 88005 Hallnäs, Lars and Peter Schroeder-Heister, *A Proof-Theoretic Approach to Logic Programming. I. Generalized Horn Clauses*, 1988

R 88006 Nordmark, Erik and Per Gunningberg, *SPIMS: A tool for protocol implementation performance measurements*, **to be published**

R 88007 Carlsson, Mats and Johan Widén, *SICStus Prolog User's Manual*, 1988

SICStus Prolog User's Manual

Errata

April 29, 1988

page 0 Quintus and Quintus Prolog are trademarks of Quintus Computer Systems, Inc.
UNIX is a trademark of Bell Laboratories.
DEC is a trademark of Digital Equipment Corporation.

page 20 Replace "`< <n>  set subterm`" by "`^ <n>  set subterm`".

page 34 The predicate `format/2` is due to Quintus Prolog [1].

page 45 The predicate `keysort/2` 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.

page 46 The predicate `dif/2` is due to Prolog II [2].

page 57 The foreign language function interface is due to Quintus Prolog [1].

page 101 The following environment variables can be set before starting SICStus to override the default sizes of certain areas. The sizes are given in cells:

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.

XREGBANKSIZE Governs the initial size of the temporary register bank. Same as the `-x` option.

References

[1] *Quintus Prolog Reference Manual version 10.* Quintus Computer Systems, Inc, 1987.

[2] A. Colmerauer. *Prolog II: Manuel de Reference et Modele Theorique.* Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1982.

SICStus Prolog User's Manual

Errata

September 29, 1988

page 0 Quintus and Quintus Prolog are trademarks of Quintus Computer Systems, Inc.
UNIX is a trademark of Bell Laboratories.
DEC is a trademark of Digital Equipment Corporation.

page 5 and 66 When SICStus is initialised it looks for a file `prolog.ini` in your home directory. If one is found, it is consulted.

page 11 There is one more `^C` option: `b` – call the command interpreter recursively.

page 20 Replace "`< <n> set subterm`" by "`^ <n> set subterm`".

page 21 The `retry <n>` keeps backtracking until it finds an invocation box whose invocation number is less than or equal to *n*.

page 23 Section 2.7 should read: "It is possible, and sometimes useful, to consult a file whilst in the middle of program execution. Procedures, 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 procedure, when called, creates a virtual copy of its definition for backtracking purposes."

page 34 The predicate `format/2` is due to Quintus Prolog [1].

page 43 The bitwise exclusive or operator is written `^`.

page 45 The predicate `keysort/2` 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.

page 46 The predicate `dif/2` is due to Prolog II [2].

page 47 No cuts are allowed in `P` in the predicates

```
\+ P
P -> Q
P -> Q ; R
if(P, Q, R)
```

page 50 The arity of compound terms created by `functor/3` cannot be greater than 255.

page 51 The length of the string of characters comprising the name of an atom cannot be greater than 512.

page 53 For the predicates in section 4.9, 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.

The definition of `retract/1` should read: "The first clause in the current interpreted program that matches *Clause* is erased. The predicate 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 procedure whose clauses are being retracted will proceed unaffected by the retracts. This is also true for invocations of `clause/2` for the same procedure. The space occupied by a retracted clause will be recovered when instances of the clause are no longer in use."

page 56 The predicates `bagof/3` and `setof/3` generate alternative solutions corresponding to different instantiations of the free variables, where two instantiations are different iff no renaming of variables can make them literally identical.

page 57 The foreign language function interface is due to Quintus Prolog [1].

page 5 and 66 When SICStus is initialised it looks for a file `prolog.ini` in your home directory. If one is found, it is consulted.

page 71 Based integers may be written in any base from 2 to 16.

page 86 The character `%` followed by any sequence of characters up to end of line is a comment.

page 91 The syntax for `natural-number` and `base` is:

```
natural-number  --> digit...
                 |  base ' alpha...
                    { where each alpha must be less than
                      the base }
                 |  0 ' char
                    { yielding the ASCII code for 'char' }

base            --> digit...  { in the range [2..16] }
```

page 101 The following environment variables can be set before starting SICStus to override the default sizes of certain areas. The sizes are given in cells:

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.

XREGBANKSIZE Governs the initial size of the temporary register bank. Same as the `-x` option.

References

[1] *Quintus Prolog Reference Manual version 10.* Quintus Computer Systems, Inc, 1987.

[2] A. Colmerauer. *Prolog II: Manuel de Reference et Modele Theorique.* Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1982.

SICStus Prolog User's Manual

Errata

December 22, 1988

page 70 The predicate `unix/1` has been extended to accept the following arguments in addition to the ones listed in the manual:

`shell(+`*Command*`,?`*Status*`)` *Command* is passed to a new UNIX shell for execution, and *Status* is unified with the value returned by the shell.

`system(+`*Command*`,?`*Status*`)` *Command* is passed to a new UNIX `sh` process for execution, and *Status* is unified with the value returned by the process.

`exit(+`*Status*`)` The SICStus process is exited, returning the integer value *Status*.

`mktemp(+`*Template*`,?`*Filename*`)` *Filename* is unified with a unique filename constructed from the atom *Template*. This is an interface to the UNIX C library function `mktemp(3)`.

`access(+`*Path*`,+`*Mode*`)` The path name *Path* and the integer *Mode* are passed to the UNIX C library function `access(2)`. The call succeeds if access is granted.

`chmod(+`*Path*`,?`*Old*`,?`*New*`)` The path name *Path* and the integer *New* are passed to the UNIX C library function `chmod(2)`. *Old* is unified with the old file mode. The call succeeds if access is granted.

`umask(?`*Old*`,?`*New*`)` The integer *New* are passed to the UNIX C library function `umask(2)`. *Old* is unified with the old file mode creation mask.

The predicate `plsys/1` has been made synonymous with `unix/1`.

page 73 For integers written in hexadecimal notation, the letters *A..F* denote the numbers 10..15.

SICStus Prolog User's Manual

Errata

September 4, 1989

page 70 The predicate `unix/1` has been extended to accept the following arguments in addition to the ones listed in the manual:

`shell(`*+Command*`,` *?Status*`)` *Command* is passed to a new UNIX shell for execution, and *Status* is unified with the value returned by the shell.

`system(`*+Command*`,` *?Status*`)` *Command* is passed to a new UNIX `sh` process for execution, and *Status* is unified with the value returned by the process.

`exit(`*+Status*`)` The SICStus process is exited, returning the integer value *Status*.

`mktemp(`*+Template*`,` *?Filename*`)` *Filename* is unified with a unique filename constructed from the atom *Template*. This is an interface to the UNIX C library function `mktemp(3)`.

`access(`*+Path*`,` *+Mode*`)` The path name *Path* and the integer *Mode* are passed to the UNIX C library function `access(2)`. The call succeeds if access is granted.

`chmod(`*+Path*`,` *?Old*`,` *?New*`)` The path name *Path* and the integer *New* are passed to the UNIX C library function `chmod(2)`. *Old* is unified with the old file mode. The call succeeds if access is granted.

`umask(`*?Old*`,` *?New*`)` The integer *New* are passed to the UNIX C library function `umask(2)`. *Old* is unified with the old file mode creation mask.

page 71 The predicate `plsys/1` has been made synonymous with `unix/1`.

The predicates `source_file(`*?File*`)` and `source_file(`*?Pred*`,` *?File*`)` exist and hold if the predicate *Pred* is defined in the file *File*.

page 73 For integers written in hexadecimal notation, the letters *A..F* denote the numbers 10..15.

SICStus Prolog User's Manual

Errata

October 23, 1989

page 36 The following control sequences have been added for compatibility:

- ~ `N` Print a newline unless at the beginning of a line.
- ~ *N*| Set a tab stop at position *N*, where *N* defaults to the current position.
- ~ *N*+ Set a tab stop at *N* positions past the current position, where *N* defaults to 8.
- ~ *N*t Dummy, for compatibility.

page 48 The predicate `call/2` has been renamed to `call_residue/2` for compatibility reasons.

page 66 There is another `prolog_flag` flag name:

`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.

page 70 The predicate `unix/1` has been extended to accept the following arguments in addition to the ones listed in the manual:

`shell(`*+Command, ?Status*`)` *Command* is passed to a new UNIX shell for execution, and *Status* is unified with the value returned by the shell.

`system(`*+Command, ?Status*`)` *Command* is passed to a new UNIX `sh` process for execution, and *Status* is unified with the value returned by the process.

`exit(`*+Status*`)` The SICStus process is exited, returning the integer value *Status*.

`mktemp(`*+Template, ?Filename*`)` *Filename* is unified with a unique filename constructed from the atom *Template*. This is an interface to the UNIX C library function `mktemp(3)`.

`access(`*+Path, +Mode*`)` The path name *Path* and the integer *Mode* are passed to the UNIX C library function `access(2)`. The call succeeds if access is granted.

`chmod(`*+Path, ?Old, ?New*`)` The path name *Path* and the integer *New* are passed to the UNIX C library function `chmod(2)`. *Old* is unified with the old file mode. The call succeeds if access is granted.

`umask(`*?Old, ?New*`)` The integer *New* are passed to the UNIX C library function `umask(2)`. *Old* is unified with the old file mode creation mask.

page 71 The predicate `plsys/1` has been made synonymous with `unix/1`.

The following new predicates exist:

`source_file(`*?File*`)`

`source_file(`*?Pred*`, `*?File*`)` The predicate *Pred* is defined in the file *File*.

`character_count(`*?Stream*`, `*?Count*`)` *Count* characters have been read from or written to the stream *Stream*.

`line_count(`*?Stream*`, `*?Count*`)` *Count* lines have been read from or written to the stream *Stream*.

`line_position(`*?Stream*`, `*?Count*`)` *Count* characters have been read from or written to the current line of the stream *Stream*.

page 73 For integers written in hexadecimal notation, the letters $A..F$ denote the numbers 10..15.