SICStus Prolog Condensed Manual

CIS, UPenn

September 4, 1995

This manual is a condensed version of Sicstus Prolog Manual and corresponds to SICStus Prolog 3 #1. The full manual is available on-line through Emacs Info (choose "Info" from "Help" menu or type "C-h" then "i") or in the directory "/src/sicstus3/doc" on Gradient and Eniac.

Contents

0.1 Introduction . . . 2
0.2 Notational Conventions . . . 2

1 How to run Prolog **3**
1.1 Getting Started . . . 3
1.2 Reading in Programs . . . 3
1.3 Inserting Clauses at the Terminal . . . 4
1.4 Directives: Queries and Commands . . . 4
1.5 Syntax Errors . . . 6
1.6 Undefined Predicates . . . 6
1.7 Program Execution And Interruption . . . 6
1.8 Exiting From The Interpreter . . . 7
1.9 Nested Executions—Break and Abort . . . 7
1.10 Saving and Restoring Program States . . . 7
1.11 Emacs Interface . . . 8

2 Debugging **9**
2.1 The Procedure Box Control Flow Model . . . 9
2.2 Basic Debugging Predicates . . . 10
2.3 Tracing . . . 11
2.4 Spy-points . . . 11
2.5 Format of Debugging messages . . . 12
2.6 Options available during Debugging . . . 12
2.7 Consulting during Debugging . . . 14
2.8 Advanced Debugging Predicates . . . 14

3 Loading Programs **14**
3.1 Predicates which Load Code . . . 14
3.2 Declarations . . . 15
3.3 Pitfalls of File-To-File Compilation . . . 15
3.4 Indexing . . . 15
3.5 Tail Recursion Optimization . . . 15

4 Built-In Predicates **15**
4.1 Input / Output . . . 16
4.2 Arithmetic . . . 19
4.3 Comparison of Terms . . . 22
4.4 Control . . . 23
4.5 Error and Exception Handling . . . 24
4.6 Information about the State of the Program . . . 25
4.7 Meta-Logic . . . 25
4.8 Modification of the Program . . . 27

4.9 Internal Database . . . 28
4.10 All Solutions . . . 29
4.11 Debugging . . . 30
4.12 Execution Profiling . . . 30
4.13 Definite Clause Grammars . . . 30
4.14 Miscellaneous . . . 31

5 The Module System **31**

6 Coroutining Facilities **31**

7 Mixing C and Prolog **31**

8 The Prolog Language **31**
8.1 Syntax, Terminology and Informal Semantics . . . 32
8.2 Declarative Semantics . . . 36
8.3 Procedural Semantics . . . 36
8.4 Occur Check . . . 37
8.5 The Cut Symbol . . . 38
8.6 Operators . . . 38
8.7 Syntax Restrictions . . . 40
8.8 Comments . . . 41
8.9 Full Prolog Syntax . . . 41

9 Programming Examples **41**

10 Summary of Built-In Predicates **41**

11 Standard Operators **51**

0.1 Introduction

This manual describes a Prolog system developed at the Swedish Institute of Computer Science. Parts of the system were developed by the project "Industrialization of SICStus Prolog" in collaboration with Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket. The system consists of a *WAM* emulator written in C, a library and runtime system written in C and Prolog and an interpreter and a compiler written in Prolog. The Prolog engine is a Warren Abstract Machine (WAM) emulator defined by D.H.D. Warren in *An Abstract Prolog Instruction Set*, Tech. Note 309, SRI International, Menlo Park, CA, 1983. Two modes of compilation are available: in-core i.e.incremental, and file-to-file.

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

0.2 Notational Conventions

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

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

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

+ArgName This argument should be instantiated to any other term.

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

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

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

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

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. However, it is often convenient to run SICStus under GNU Emacs instead. A GNU Emacs mode for SICStus is described later (see Section 1.11 [Emacs Interface], p. 8). From a UNIX shell, SICStus is started by performing:

```
% sp [-f] [-i] [-l prolog-file] [-a argument...]  (on Eniac)
```

⇒ Refer to the full manual for more details.

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

```
SICStus 2.1 #0 :  Mon Oct 14 09:27:36 MET 1991
| ?-
```

1.2 Reading in Programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the interpreter. The clauses of a predicate do not have to be immediately consecutive, but remember that their relative order may be important (see Section 8.3 [Procedural], p. 36).

To input a program from a file *file*, just type the filename inside list brackets (followed by `.` and `RET`), thus:

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

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

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

The specified file is then read in. Clauses in the file are stored so that they can later be interpreted, while any directives are obeyed as they are encountered. When the end of the file is found, the interpreter displays on the terminal the time spent for read-in. This indicates the completion of the directive.

Predicates that expect the name of a Prolog source file as an argument use `absolute_file_name/2` (see Section 4.1.4 [Stream Pred], p. 18) to look up the file. This predicate will first search for a file with the suffix '`.pl`' added to the name given as an argument. If this fails it will look for a file with no extra suffix added. There is also support for libraries.

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

```
| ?- [myprog,extras,tests].
```

In this case all three files would be consulted.

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

1.3 Inserting Clauses at the Terminal

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

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

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

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

1.4 Directives: Queries and Commands

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

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

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

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

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

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

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

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

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

If the goal(s) specified in a query can be satisfied, and if there are no variables as in this example, 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 `RET` or else a `;` followed by `RET`. Simply typing `RET` terminates the query; the interpreter responds with '`yes`'. However, typing `;` causes the system to *backtrack* (see Section 8.3 [Procedural], p. 36) looking for alternative solutions. If no further solutions can be found it outputs '`no`'.

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

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

X = tom ;
X = dick ;
X = harry ;
no
| ?- member(X, [a,b,f(Y,c)]), member(X, [f(b,Z),d]).

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

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

X = f(_A)

yes
| ?-
```

Commands are like queries except that

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

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

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

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

{`Warning:` *Goal* `- goal failed`}

as a warning.

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

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

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

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

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

1.5 Syntax Errors

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

```
| member(X, X$L).

* in lines 11-12 **
* , or ) expected in arguments **
member ( X , X
* here **
$ L ) .
```

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

Note that any comments in the faulty line are not displayed with the error message. If you are in doubt about which clause was wrong you can use the `listing/1` predicate to list all the clauses which were successfully read-in, e.g.

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

NOTE: The built in predicates `read/1-2` normaly raise an exception on syntax errors (see Section 4.5 [Exception], p. 24). The behavior is controlled by the flag `syntax_errors` (see `prolog_flag/3`).

1.6 Undefined Predicates

There is a difference between predicates that have no definition and predicates that have no clauses. The latter case is meaningful e.g. for dynamic predicates (see Section 3.2 [Declarations], p. 15) that clauses are being added to or removed from. There are good reasons for treating calls to undefined predicates as errors, as such calls easily arise from typing errors.

The system can optionally catch calls to predicates that have no definition. The state of the catching facility can be:

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

`error` which calls the user defined predicate `unknown_predicate_handler/3` (see Section 4.14 [Misc Pred], p. 31). If this call fails calls an exception is raised (see Section 4.5 [Exception], p. 24). This is the default state.

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

Calls to predicates that have no clauses are not caught.

1.7 Program Execution And Interruption

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

Only when execution of one directive is complete does the interpreter become ready for another directive. However, one may interrupt the normal execution of a directive by typing `^C`. This `^C` interruption has the effect of suspending the execution, and the following message is displayed:

```
Prolog interruption (h or ?  for help) ?
```

At this point the interpreter accepts one-letter commands corresponding to certain actions. To execute an action simply type the corresponding character (lower or upper case) followed by `RET`. The possible commands are:

`a` abort the current computation.

`b` invoke the Prolog interpreter recursively.

`c` continue the execution.

`d` enable debugging. See Section 2 [Debug Intro], p. 9.

`e` exit from SICStus, closing all files.

`h`

`?` list available commands.

`t` enable trace. See Section 2.3 [Trace], p. 11.

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

1.8 Exiting From The Interpreter

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

1.9 Nested Executions—Break and Abort

The Prolog system provides a way to suspend the execution of your program and to enter a new incarnation of the top level where you can issue directives to solve goals etc. This is achieved by issuing the directive (see Section 1.7 [Execution], p. 6):

```
| ?- break.
```

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

```
{Break level 1}
```

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

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

```
{End break}
```

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

A suspended execution can be aborted by issuing the directive:

```
| ?- abort.
```

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

1.10 Saving and Restoring Program States

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

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

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

This predicate may be called at any time, for example it may be useful to call it in a break in order to save an intermediate execution state. The file *File* becomes an executable file. See Section 1.1 [Start], p. 3.

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

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

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

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

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

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

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

The file *File* becomes an executable file. See Section 1.1 [Start], p. 3. After restoring a partial program state, the interpreter will reinitialize itself.

If the environment variable `SP_COMPRESS_SAVE` is set to anything but 0, the saved state will be about twice as small, and the time it takes to create it about twice as long, as otherwise. This environment variable has no effect on `restore/1`: the fact whether or not a saved state is compressed is recorded in the file.

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

The save facility is not available in Runtime Systems and the predicates defined in this chapter are undefined, see Section 7 [Runtime Systems], p. 31.

1.11 Emacs Interface

Assuming the GNU Emacs mode for SICStus has been installed, inserting the following lines in your '`/.emacs`' will make Emacs use this mode automatically when editing files with a '`.pl`' extension:

```
(setq load-path (cons "/usr/local/lib/sicstus2.1" load-path))
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog" "Major mode for editing prolog programs" t)
```

where '`/usr/local/lib/sicstus2.1`' should be replaced by the name of the SICStus source code directory.

The Emacs mode will use the value of the environment variable `EPROLOG` as a shell command to invoke SICStus. This value defaults to `prolog`. The Emacs mode provides the following commands:

`M-x run-prolog` Run an inferior Prolog process, input and output via the buffer `*prolog*`.

`C-c K` The entire buffer is compiled.

`C-c k` The current region is compiled.

`C-c C-k` The predicate around point is compiled. Empty lines are treated as predicate boundaries.

`C-c C` The entire buffer is consulted.

`C-c c` The current region is consulted.

`C-c C-c` The predicate around point is consulted. Empty lines are treated as predicate boundaries.

2 Debugging

This chapter describes the debugging facilities that are available in the Prolog interpreter. The purpose of these facilities is to provide information concerning the control flow of your program. The main features of the debugging package are as follows:

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

The Procedure Box model of execution is also called the Byrd Box model after its inventor, Lawrence Byrd.

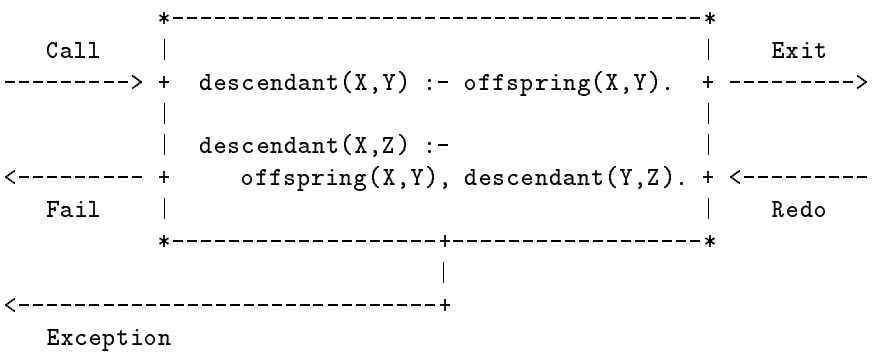
Much of the information in this chapter is also in Chapter eight of *Programming in Prolog* by W.F. Clocksin and C.S. Mellish (Springer-Verlag, 1981) which is recommended as an introduction.

The debugger is not available in Runtime Systems and the predicates defined in this chapter are undefined, see Section 7 [Runtime Systems], p. 31.

2.1 The Procedure Box Control Flow Model

During debugging the interpreter prints out a sequence of goals in various states of instantiation in order to show the state the program has reached in its execution. However, in order to understand what is occurring it is necessary to understand when and why the interpreter prints out goals. As in other programming languages, key points of interest are predicate entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly starts backtracking. The Procedure Box model of Prolog execution views program control flow in terms of movement about the program text. This model provides a basis for the debugging mechanism in the interpreter, and enables the user to view the behavior of the program in a consistent way.

Let us look at an example Prolog predicate :



The first clause states that Y is a descendant of X if Y is an offspring of X, and the second clause states that Z is a descendant of X if Y is an offspring of X and if Z is a descendant of Y. In the diagram a box has been drawn around the whole predicate and labelled arrows indicate the control flow in and out of this box. There are five such arrows which we shall look at in turn.

Call This arrow represents initial invocation of the predicate. When a goal of the form `descendant(X,Y)` is required to be satisfied, control passes through the *Call* port of the descendant box with the intention of matching a component clause and then satisfying any subgoals in the body of that clause. Note that this is independent of whether such a match is possible; i.e.first the box is called,

and then the attempt to match takes place. Textually we can imagine moving to the code for descendant when meeting a call to descendant in some other part of the code.

Exit This arrow represents a successful return from the predicate. This occurs when the initial goal has been unified with one of the component clauses and any subgoals have been satisfied. Control now passes out of the *Exit* port of the descendant box. Textually we stop following the code for descendant and go back to the place we came from.

Redo This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes through the *Redo* port of the descendant box. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary.

Fail This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing. Control now passes out of the *Fail* port of the descendant box and the system continues to backtrack. Textually we move back to the code which called this predicate and keep moving backwards up the code looking for choice points.

Exception This arrow represents an exception which was raised in the initial goal, either by a call to `raise_exception/1` or by an error in a built-in predicate. See Section 4.5 [Exception], p. 24. Control now passes out of the *Exception* port of the descendant box and the systems continues to pass the exception to outer levels. Textually we move back to the code which called this predicate and keep moving backwards up the code looking for a call to `on_exception/3`.

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

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

2.2 Basic Debugging Predicates

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

`debug` Switches the debugger on. (It is initially off.) In order for the full range of control flow information to be available it is necessary to have this on from the start. When it is off the system does not remember invocations that are being executed. (This is because it is expensive and not required for normal running of programs.) You can switch *Debug Mode* on in the middle of execution, either from within your program or after a `^C` (see trace below), but information prior to this will just be unavailable.

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

`debugging` Prints onto the terminal information about the current debugging state. This will show:

1. Whether undefined predicates are being trapped.
2. Whether the debugger is swithed on.
3. What spy-points have been set (see below).
4. What mode of leashing is in force (see below).
5. What the interpreter maxdepth is (see below).

2.3 Tracing

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

`trace` Switches the debugger on, if it is not on already, and ensures that the next time control enters a procedure box, a message will be produced and you will be asked to interact. The effect of trace can also be achieved by typing `t` after a `^C` interruption of a program.

At this point you have a number of options. See Section 2.6 [Debug Options], p. 12. In particular, you can just type `RET` to *creep* (or single-step) into your program. If you continue to creep through your program you will see every entry and exit to/from every invocation box. 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 subset of the following, specified as a list:

`call` Prompt on Call.

`exit` Prompt on Exit.

`redo` Prompt on Redo.

`fail` Prompt on Fail.

`exception` Prompt on Exception.

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

`notrace` Equivalent to `nodebug`.

2.4 Spy-points

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

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

Spy-points are set and removed by the following built-in predicates. The first two are also standard operators:

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

Name all predicates of that name no matter what arity,

Name/Arity the predicate of that name and arity,

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

Module:Spec specifying a particular module instead of the default type-in module, see Section 5 [Module Spec], p. 31), and

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

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

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

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

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

`spypoint_condition(`*:Goal*`,`*?Port*`,`*+Test*`)` Sets a *conditional* spy-point on the predicate for *Goal*. When the debugger reaches the spy-point, three conditions must be met for the spy-point to apply: The actual goal must match *Goal*, the actual port must match *Port*, and *Test* (an arbitrary Prolog goal) must succeed.

The options available when you arrive at a spy-point are described later. See Section 2.6 [Debug Options], p. 12.

2.5 Format of Debugging messages

We shall now look at the exact format of the message output by the system at a port. All trace messages are output to the terminal regardless of where the current output stream is directed. (This allows you to trace programs while they are performing file IO.) The basic format is as follows:

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

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

T is a subterm trace. This is used in conjunction with the '`^`' command (set subterm), described below. If a subterm has been selected, *T* is printed as the sequence of commands used to select the subterm. Normally, however,, *T* is printed as ' ', indicating that no subterm has been selected.

The first number is the unique invocation identifier. It is nondecreasing regardless of whether or not you are actually seeing the invocations (provided that the debugger is swithed on). This number can be used to cross correlate the trace messages for the various ports, since it is unique for every invocation. It will also give an indication of the number of procedure calls made since the start of the execution. The invocation counter starts again for every fresh execution of a command, and it is also reset when retries (see later) are performed.

The number following this is the *current depth*; i.e.the number of direct *ancestors* this goal has.

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

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], p. 16) 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], p. 12). If this particular port is unleashed then you will obviously not get this prompt since you have specified that you do not wish to interact at this point.

Note that not all procedure calls are traced; there are a few basic predicates which have been made invisible since it is more convenient not to trace them. These include debugging directives and basic control structures, including `trace/0`, `debug/0`, `notrace/0`, `nodebug/0`, `spy/1`, `nospy/1`, `nospyall/0`, `leash/1`, `debugging/0`, `true/0`, `!/0`, `','/2`, `'->'/2`, `;/2`, `'\+'/1`, and `if/3`. 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 end of the line (`RET`). Some options only actually require the terminator; e.g.the creep option, as we have already seen, only requires `RET`.

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

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

`c`

`RET` *creep* causes the interpreter to single-step to the very next port and print a message. Then if the port is leashed (see Section 2.3 [Trace], p. 11), the user is prompted for further interaction. Otherwise, it continues creeping. If leashing is off, creep is the same as *leap* (see below) except that a complete trace is printed on the terminal.

`l` *leap* causes the interpreter to resume running your program, only stopping when a spy-point is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing. All you need to do is to set spy-points on an evenly spread set of pertinent predicates, and then follow the control flow through these by leaping from one to the other.

`s` *skip* is only valid for Call and Redo ports. It skips over the entire execution of the predicate. That is, you will not see anything until control comes back to this predicate (at either the Exit port or the Fail port). Skip is particularly useful while creeping since it guarantees that control will be returned after the (possibly complex) execution within the box. If you skip then no message at all will appear until control returns. This includes calls to predicates with spy-points set; they will be masked out during the skip. There is a way of overriding this : the `t` option after a `^C` interrupt will disable the masking. Normally, however, this masking is just what is required!

`r` *retry* can be used at any of the four ports (although at the Call port it has no effect). It transfers control back to the Call port of the box. This allows you to restart an invocation when, for example, you find yourself leaving with some weird result. The state of execution is exactly the same as when you originally called, (unless you use side effects in your program; i.e.asserts etc. will not be undone). When a retry is performed the invocation counter is reset so that counting will continue from the current invocation number regardless of what happened before the retry. This is in accord with the fact that you have, in executional terms, returned to the state before anything else was called.

`f` *fail* can be used at any of the four ports (although at the Fail port it has no effect). It transfers control to the Fail port of the box, forcing the invocation to fail prematurely.

`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], p. 25). You can always be sure of jumping to any goal in the ancestor list (by using retry etc). If you supply an integer *n*, then only that number of ancestors will be printed. That is to say, the last *n* ancestors will be printed counting back from the current goal. The list is printed using `print/1` and each entry is preceded by the invocation number followed by the depth number (as would be given in a trace message).

`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.

`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.

⇒ Refer to the full manual for more details.

2.7 Consulting during Debugging

It is possible, and sometimes useful, to consult a file whilst in the middle of program execution. Predicates, which have been successfully executed and are subsequently redefined by a consult and are later reactivated by backtracking, will not notice the change of their definitions. In other words, it is as if every predicate, when called, creates a virtual copy of its definition for backtracking purposes.

If SICStus is run via the Emacs interface, the commands for loading code (such as `C-c C-c`, `consult-region`) are not directly available when the system prompts you after printing out a debugging message. Press `b` followed by `RET` to get an interpreter top level, ready to accept the Emacs commands. Type `^D` to return to the debugging port.

2.8 Advanced Debugging Predicates

⇒ Refer to the full manual for more details.

3 Loading Programs

Programs can be loaded in three different ways: consulted or compiled from source file, or loaded from object files. The latter is the fastest way of loading programs, but of course requires that the programs have been compiled to object files first. Object files may be handy when developing large applications consisting of many source files, but are not strictly necessary since it is possible to save and restore entire execution states (see Section 4.14 [Misc Pred], p. 31).

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

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

⇒ Refer to the full manual for more details.

3.1 Predicates which Load Code

To consult a program, issue the directive:

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

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

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

When a directive is read it is immediately executed. Any predicate defined in the files erases any clauses for that predicate already present in the interpreter. If the old clauses were loaded from a different file than the present one, the user will be queried first whether (s)he really wants the new definition. However, for existing predicates which have been declared as `multifile` (see below) new clauses will be added to the predicate, rather than replacing the old clauses. If clauses for some predicate appear in

more than one file, the later set will effectively overwrite the earlier set. The division of the program into separate files does not imply any module structure—any predicate can call any other (see Section 5 [Module Intro], p. 31).

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

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

This is a shorthand way of consulting a list of files. (The case where there is just one filename in the list was described earlier (see Section 1.2 [Reading In], p. 3).

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

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

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

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

3.2 Declarations

⇒ Refer to the full manual for more details.

3.3 Pitfalls of File-To-File Compilation

⇒ Refer to the full manual for more details.

3.4 Indexing

⇒ Refer to the full manual for more details.

3.5 Tail Recursion Optimization

⇒ Refer to the full manual for more details.

4 Built-In Predicates

It is not possible to redefine built-in predicates. An attempt to do so will give an error message. See Section 10 [Pred Summary], p. 41.

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

Input / Output
- Reading-in Programs
- Input and Output of Terms
- Character IO
- Stream IO
- Dec-10 Prolog File IO

Arithmetic
Comparison of Terms
Control
Error and Exception Handling
Information about the State of the Program
Meta-Logic

Modification of the Program
Internal Database
All Solutions
Debugging
Profiling
Definite Clause Grammars
Miscellaneous

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

4.1 Input / Output

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

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

4.1.1 Reading-in Programs

When the predicates discussed in this section are invoked, filenames are relative to the current working directory (`cwd`). During the load, the `cwd` is temporarily changed to the directory containing the file being read in. This has the effect that if one of these predicates is invoked recursively, the filename of the recursive load is relative to the directory of the enclosing load. See Section 3 [Load Intro], p. 14 for an introduction to these predicates.

Directives will be executed in order of occurrence. Be aware of the changed `cwd` as it could have an effect on the semantics of directives. Only the first solution of directives is produced, and variable bindings are not displayed. Directives that fail or raise exceptions cause warnings, but do not terminate the load.

⇒ Refer to the full manual for more details.

4.1.2 Input and Output of Terms

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

Some of these predicates support a notation for terms containing multiple occurrences of the same subterm (cycles and DAGs). The notation is `@(`*Template*`,`*Substitution*`)` where *Substitution* is a list of *Var*=*Term* pairs where the *Var* occurs in *Template* or in one of the *Terms*. This notation stands for the instance of *Template* obtained by binding each *Var* to its corresponding *Term*. The purpose of this notation is to provide a finite printed representation of cyclic terms. This notation is not used by default, and `@/2` has no special meaning except in this context.

`read(`*?Term*`)`

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

`write(?Term)`

`write(+Stream,?Term)` The term *Term* is written onto *Stream* according to current operator declarations.

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

`writeq(?Term)`

`writeq(+Stream,?Term)` Similar to `write(Stream,Term)`, but the names of atoms and functors are quoted where necessary to make the result acceptable as input to `read/2`, provided the same operator declarations are in effect.

`print(?Term)`

`print(+Stream,?Term)` *Hookable.* Prints *Term* onto *Stream.* This predicate provides a handle for user defined pretty printing:

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

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

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

`format(+Format,+Arguments)`

`format(+Stream,+Format,+Arguments)` Print *Arguments* onto *Stream* according to format *Format.* *Format* is a list of formatting characters. If *Format* is an atom then `name/2` (see Section 4.7 [Meta Logic], p. 25) will be used to translate it into a list of characters. Thus

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

has the same effect as

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

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

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

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

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

will result in

```
Hello  world!
```

⇒ Refer to the full manual for more details.

4.1.3 Character Input/Output

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

`nl`

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

`get0(`*?N*`)`

`get0(`*+Stream*`,`*?N*`)` *N* is the character code of the next character read from *Stream*. If all characters of *Stream* have been read, *N* is -1, and further calls to `get0/2` for the same stream will then raise an exception, unless the stream is connected to the terminal.

`peek_char(`*?N*`)`

`peek_char(`*+Stream*`,`*?N*`)` *N* is the character code of the next character from *Stream*, or -1, if all characters of *Stream* have been read, *N*. The character is not actually read, it is only looked at and is still available for subsequent input.

`get(`*?N*`)`

`get(`*+Stream*`,`*?N*`)` Same as `get0/2`, except *N* is the character code of the next character that is not a *layout-char* (see Section 8.9 [Token String], p. 41)read from *Stream*.

`skip(`*+N*`)`

`skip(`*+Stream*`,`*+N*`)` Skips just past the next character code *N* from *Stream*. *N* may be an arithmetic expression.

`skip_line`

`skip_line(`*+Stream*`)` Skips just past the next `LFD` from *Stream*.

`put(`*+N*`)`

`put(`*+Stream*`,`*+N*`)` Character code *N* is output onto *Stream*. *N* may be an arithmetic expression.

`tab(`*+N*`)`

`tab(`*+Stream*`,`*+N*`)` *N* spaces are output onto *Stream*. *N* may be an arithmetic expression.

4.1.4 Stream IO

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

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

`read` Open the file for input.

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

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

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

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

`fileerrors` Undoes the effect of `nofileerrors/0`.

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

4.1.5 Socket IO

⇒ Refer to the full manual for more details.

4.1.6 DEC-10 Prolog File IO

The following predicates manipulate files.

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

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

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

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

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

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

4.2 Arithmetic

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

The range of integers is `[-2^2147483616, 2^2147483616)`. Thus for all practical purposes, the range of integers can be considered infinite.

The range of floats is the one provided by the C `double` type, typically `[4.9e-324, 1.8e+308]` (plus or minus).

Only certain functors are permitted in an arithmetic expression. These are listed below, together with an indication of the functions they represent. *X* and *Y* are assumed to be arithmetic expressions. Unless stated otherwise, the arguments of an expression may be any numbers and its value is a float if any of its arguments is a float, otherwise the value is an integer. Any implicit coercions are performed with the `integer/1` and `float/1` functions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`[`X`]` A list of just one number X evaluates to X. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its character code; e.g. `"A"` behaves within arithmetic expressions as the integer 65.

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

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

`gcd(`X`,` Y`)` The value is the greatest common divisor of the two integers X and Y.

`min(`X`,`Y`)` The value is the lesser value of X and Y.

`max(`X`,`Y`)` The value is the greater value of X and Y.

`msb(`X`)` The value is the most significant bit position of the integer X. It is equivalent to, but more efficient than, `integer(log(2,X))`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

yes
```

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

Z `is` X X, which must be an arithmetic expression, is evaluated and the result is unified with Z.

X `=:=` Y The numeric values of X and Y are equal.

X `=\=` Y The numeric values of X and Y are not equal.

$X < Y$ The numeric value of X is less than the numeric value of Y.

$X > Y$ The numeric value of X is greater than the numeric value of Y.

X `=<` Y The numeric value of X is less than or equal to the numeric value of Y.

X `>=` Y The numeric value of X is greater than or equal to the numeric value of Y.

4.3 Comparison of Terms

These built-in predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison (see Section 4.2 [Arithmetic], p. 19) or unification.

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

- Variables, in a standard order (roughly, oldest first—the order is *not* related to the names of variables).
- Floats, in numeric order (e.g.-1.0 is put before 1.0).
- Integers, in numeric order (e.g.-1 is put before 1).
- Atoms, in alphabetical (i.e.character code) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order). Recall that lists are equivalent to compound terms with principal functor `./2`.

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

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

These are the basic predicates for comparison of arbitrary terms:

Term1 `==` *Term2* Tests if the terms currently instantiating *Term1* and *Term2* are literally identical (in particular, variables in equivalent positions in the two terms must be identical). For example, the query

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

fails (answers '`no`') because *X* and *Y* are distinct uninstantiated variables. However, the query

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

succeeds because the first goal unifies the two variables (see Section 4.14 [Misc Pred], p. 31).

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

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

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

Term1 `@=<` *Term2* Term *Term1* is not after term *Term2* in the standard order.

Term1 `@>=` *Term2* Term *Term1* is not before term *Term2* in the standard order.

Some further predicates involving comparison of terms are:

`?=(`*?X*`,`*?Y*`)` Succeeds if *X* and *Y* are either syntactically identical or syntactically non-unifiable.

`compare(`*?Op*`,`*?Term1*`,`*?Term2*`)` The result of comparing terms *Term1* and *Term2* is *Op*, where the possible values for *Op* are:

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

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

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

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

`sort(`*+List1*`,`*?List2*`)` The elements of the list *List1* are sorted into the standard order (see Section 4.3 [Term Compare], p. 22) and any identical elements are merged, yielding the list *List2*. (The time and space complexity of this operation is at worst *O(N lg N)* where *N* is the length of *List1*.)

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

4.4 Control

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

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

`!` See Section 8.5 [Cut], p. 38.

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

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

No cuts are allowed in P.

Remember that with prefix operators such as this one it is necessary to be careful about spaces if the argument starts with a `(`. For example:

```
| ?- \+ (P,Q).
```

is this operator applied to the conjunction of P and Q, but

```
| ?- \+(P,Q).
```

would require a predicate `\+ /2` for its solution. The prefix operator can however be written as a functor of one argument; thus

```
| ?- \+((P,Q)).
```

is also correct.

`+P -> +Q ; +R` Analogous to

if P then Q else R

i.e.defined as if by

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

No cuts are allowed in P.

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

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

The precedence of `->` is less than that of `;` (see Section 8.6 [Operators], p. 38), so the expression is read as

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

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

```
(P -> Q; fail)
```

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

if P then Q else R

but differs from `P -> Q ; R` in that `if(P, Q, R)` explores *all* solutions to the goal P. There is a small time penalty for this—if P is known to have only one solution of interest, the form `P -> Q ; R` should be preferred.

No cuts are allowed in P.

`otherwise`

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

`false`

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

`repeat` Generates an infinite sequence of backtracking choices. In sensible code, `repeat/0` is hardly ever used except in *repeat loops*. A repeat loop has the structure

```
Head :-
                ...
                save(OldState),
                repeat,
                        generate(Datum),
                        action(Datum),
                        test(Datum),
                !,
                restore(OldState),
                ...
```

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

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

`call(`*:Term*`)`

`incore(`*:Term*`)`

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

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

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

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

4.5 Error and Exception Handling

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

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

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

In the Development System, any previously uncaught exception is caught and an appropriate error message is printed before returning to the top level. In Runtime Systems, uncaught exceptions are caught at the innermost call from C instead. The printing of these and most other messages in the Development System is handled by the predicate `print_message/2`. The behavior of this predicate can be overridden by defining `user:portray_message/2`, so as to suppress or alter the format of certain messages. These predicates work as follows:

`print_message(+Severity, +Message)` *Hookable.* Most messages from the system are printed by calling this predicate. Before anything is printed, however, `print_message/2` calls `user:portray_message/2` with the same arguments, so as to give the user a means of intercepting the message before it is actually printed. If `user:portray_message/2` succeeds, nothing is printed, otherwise *Message* is formatted and printed using the default method.

Message is a term that encodes the message to be printed. The format of message terms is subject to change, but can be inspected in the file '`$SP_PATH/Library/msgs.pl`'. *Severity* is an atom denoting the severity of the message, and is one of:

4.6 Information about the State of the Program

`listing`

Lists onto the current output stream all the clauses in the current interpreted program (in the type-in module, see Section 5 [Module Spec], p. 31). Clauses listed onto a file can be consulted back.

`listing(:Spec)` Lists all interpreted predicates covered by *Spec*, which is described under `spy/1` (see Section 2.4 [Spy-Point], p. 11), but which may also contain variables. For example:

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

`current_atom(?Atom)` If *Atom* is instantiated then test if *Atom* is an Atom.

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

`current_predicate(?Name,:Head)`

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

4.7 Meta-Logic

The predicates in this section are meta-logical and perform operations that require reasoning about the current instantiation of terms or decomposing terms into their constituents. Such operations cannot be expressed using predicate definitions with a finite number of clauses.

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

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

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

`nonvar(?X)` Tests whether X is currently instantiated. This is the opposite of `var/1`.

`ground(?X)` Tests whether X is completely instantiated, i.e.free of unbound variables.

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

`float(?X)` Checks that X is currently instantiated to a float.

`integer(?X)` Checks that X is currently instantiated to an integer.

`number(?X)` Checks that X is currently instantiated to a number.

`atomic(?X)` Checks that X is currently instantiated to an atom or number.

`simple(?X)` Checks that X is currently uninstantiated or instantiated to an atom or number.

`compound(?X)` Checks that X is currently instantiated to a term of arity > 0 i.e.a list or a structure.

`callable(`*?X*`)` Checks that *X* is currently instantiated to a term valid as a goal i.e.a compound term or an atom.

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

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

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

+Term `=..` *?List*

?Term `=..` *+List* *List* is a list whose head is the atom corresponding to the principal functor of *Term*, and whose tail is a list of the arguments of *Term*. e.g.

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

If *Term* is uninstantiated, then *List* must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number. Note that this predicate is not strictly necessary, since its functionality can be provided by `arg/3` and `functor/3`, and using the latter two is usually more efficient.

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

`name(`*?Const*`,`*+CharList*`)` If *Const* is an atom or number then *CharList* is a list of the character codes of the characters comprising the name of *Const*. e.g.

```
| ?- name(product, L).
L = [112,114,111,100,117,99,116]
| ?- name(product, "product").
| ?- name(1976, L).
L = [49,57,55,54]
| ?- name('1976', L).
L = [49,57,55,54]
| ?- name((:-), L).
L = [58,45]
```

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

```
| ?- name(X, [58,45]).
X = :-
| ?- name(X, ":-").
X = :-
| ?- name(X, [49,50,51]).
X = 123
```

Note that there atoms are for which `name(`*Const*`,`*CharList*`)` is true, but which will not be constructed if `name/2` is called with *Const* uninstantiated. One such atom is the atom `'1976'`. It is recommended that new programs use `atom_chars/2` or `number_chars/2`, as these predicates do not have this inconsistency.

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

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

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

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

`term_hash(`*+Term*`,`*+Depth*`,`*+Range*`,`*?Hash*`)` If *Term* is instantiated to the given *Depth*, an integer hash value in the range [0,*Range*) corresponding to *Term* is unified with *Hash*. Otherwise, the goal just succeeds.

The *depth* of a term is defined as follows: the (principal functor of) the term itself has depth 1, and an argument of a term with depth *i* has depth *i+1*.

Depth should be an integer >= -1. If *Depth* = -1, *Term* must be ground.

Range should be an integer >= 1.

```
| ?- term_hash([a,b,_], 3, 4, H).
H = 0
| ?- term_hash([a,b,_], 4, 4, H).
true
| ?- term_hash(f(a,f(b,f(_,[]))), 2, 4, H).
H = 3
```

`term_hash/4` is provided primarily as a tool for the construction of sophisticated Prolog clause access schemes. Its intended use is to generate hash values for terms that will be used with first argument clause indexing, yielding compact and efficient multi-argument or deep argument indexing.

`subsumes_chk(`*+General*`, `*+Specific*`)` True when *Specific* is an instance of *General*, i.e.if there is a substitution that leaves *Specific* unchanged and makes *General* identical to *Specific*. It doesn't bind any variables.

```
subsumes_chk(f(X), f(a)).
true
| ?- subsumes_chk(f(a), f(X)).
no
| ?- subsumes_chk(A-A, B-C).
no
| ?- subsumes_chk(A-B, C-C).
true
```

`term_subsumer(`*+Term1*`, `*+Term2*`, `*-General*`)` Binds *General* to the most specific term that generalizes *Term1* and *Term2*. This process is sometimes called *anti-unification*, as it is the dual of unification.

```
| ?- term_subsumer(f(g(1,h(_))), f(g(_,h(1))), T).
T = f(g(_B,h(_A)))
| ?- term_subsumer(f(1+2,2+1), f(3+4,4+3), T).
T = f(_A+_B,_B+_A)
```

4.8 Modification of the Program

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

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

Note that a term *Head* `:-` *Body* must be enclosed in parentheses when it occurs as an argument of a compound term, as '`:-`' is a standard infix operator with precedence greater than 1000 (see Section 8.6 [Operators], p. 38), e.g.:

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

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

`assert(`*:Clause*`)`

`assert(`*:Clause*`,-`*Ref*`)` The current instance of *Clause* is interpreted as a clause and is added to the current interpreted program. The predicate concerned must be currently be dynamic or undefined and the position of the new clause within it is implementation-defined. *Ref* is a unique identifier of the asserted clause. Any uninstantiated variables in the *Clause* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 6 [Blocking], p. 31).

`asserta(`*:Clause*`)`

`asserta(`*:Clause*`,-`*Ref*`)` Like `assert/2`, except that the new clause becomes the *first* clause for the predicate concerned.

`assertz(`*:Clause*`)`

`assertz(`*:Clause*`,-`*Ref*`)` Like `assert/2`, except that the new clause becomes the *last* clause for the predicate concerned.

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

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

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

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

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

`abolish(`*:Spec*`)`

`abolish(`*:Name*`,+`*Arity*`)` Erase all clauses of the predicate specified by *Spec* or *Name*/*Arity*. *Spec* is described under `spy/1` (see Section 2.4 [Spy-Point], p. 11) and *Name* may be prefixed by a module name (see Section 5 [Module Spec], p. 31).

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

4.9 Internal Database

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

These predicates store arbitrary terms in the database without interfering with the clauses which make up the program. The terms which are stored in this way can subsequently be retrieved via the key

on which they were stored. Many terms may be stored on the same key, and they can be individually accessed by pattern matching. Alternatively, access can be achieved via a special identifier which uniquely identifies each recorded term and which is returned when the term is stored.

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

`recorda(`*+Key*`,`*?Term*`,-`*Ref*`)` The term *Term* is recorded in the internal database as the first item for the key *Key*, where *Ref* is its implementation-defined identifier. The key must be given, and only its principal functor is significant. Any uninstantiated variables in the *Term* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 6 [Blocking], p. 31).

`recordz(`*+Key*`,`*?Term*`,-`*Ref*`)` Like `recorda/3`, except that the new term becomes the *last* item for the key *Key*.

`erase(`*+Ref*`)` The recorded item (or dynamic clause (see Section 4.9 [Database], p. 28)) whose implementation-defined identifier is *Ref* is effectively erased from the internal database or interpreted program.

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

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

4.10 All Solutions

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

`setof(`*?Template*`,`*:Goal*`,`*?Set*`)` Read this as "*Set* is the set of all instances of *Template* such that *Goal* is satisfied, where that set is non-empty". The term *Goal* specifies a goal or goals as in `call(`*Goal*`)` (see Section 4.4 [Control], p. 23). *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], p. 22). If there are no instances of *Template* such that *Goal* is satisfied then the predicate fails.

The variables appearing in the term *Template* should not appear anywhere else in the clause except within the term *Goal*. Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case the list *Set* will only provide an imperfect representation of what is in reality an infinite set.

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

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

the query

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

might produce two alternative solutions via backtracking:

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

The query:

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

would then produce:

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

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

Y^Q

meaning "there exists a *Y* such that *Q* is true", where *Y* is some Prolog variable.

For example:

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

would produce the single result:

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

in contrast to the earlier example.

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

?X^:P The interpreter recognizes this as meaning "there exists an *X* such that *P* is true", and treats it as equivalent to *P* (see Section 4.4 [Control], p. 23). The use of this explicit existential quantifier outside the `setof/3` and `bagof/3` constructs is superfluous and discouraged.

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

`findall(`*?Template*`,`*:Goal*`,`*?Bag*`,`*?Remainder*`)` Same as `findall/3`, except *Bag* is the list of solution instances appended to *Remainder*, which is typically unbound.

4.11 Debugging

⇒ Refer to the full manual for more details.

4.12 Execution Profiling

⇒ Refer to the full manual for more details.

4.13 Definite Clause Grammars

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

Definite clause grammars are an extension of the well-known context-free grammars. A grammar rule in Prolog takes the general form

head `-->` *body*`.`

meaning "a possible form for *head* is *body*". Both *body* and *head* are sequences of one or more items linked by the standard Prolog conjunction operator '`,`'.

⇒ Refer to the full manual for more details.

4.14 Miscellaneous

⇒ Refer to the full manual for more details.

5 The Module System

By making use of the module systems facilities, programs can be divided into different modules. Each module has its own independent predicate name space. This is an important feature for the development of larger programs. The module system of SICStus Prolog is procedure based. This means that only the predicates are local to a module, whereas terms are global. The module system is flat, not hierarchical, so all modules are visible to one another. It is non-strict, i.e.the normal visibility rules can be overridden by special syntax. No overhead is incurred on compiled calls to predicates in other modules. It is modeled after and compatible with the Quintus Prolog module system. Finally, using the module system is optional, and SICStus Prolog may be used without the user being aware of the module system at all.

Modules in SICStus Prolog can also be used for object-oriented programming. See the description of the SICStus Objects package in the Library Manual for details.

⇒ Refer to the full manual for more details.

6 Coroutining Facilities

The computation rule in traditional Prolog systems is simple: "pick the leftmost goal of the current query". However, SICStus Prolog and other modern implementations have a somewhat more complex computation rule "pick the leftmost *unblocked* goal of the current query". A goal is *blocked* if certain arguments are uninstantiated and its predicate definition is annotated with a matching *block declaration*. Goals of some built-in predicates, documented in this chapter, may also be blocked if their arguments are not instantiated enough.

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

⇒ Refer to the full manual for more details.

7 Mixing C and Prolog

SICStus provides a bi-directional, procedural interface for program parts written in C and Prolog. The interface defines a number of functions and hooks for various operations. In addition there is also the declarative interface for calling C functions from Prolog, accessed by the predicate `load_foreign_files/2`, which may be used in conjunction with, or instead of, the procedural one.

The C support routines are available in the Development System as well as in Runtime Systems. When using the SICStus C Interface in the Development System, an initial function has to be interfaced using `load_foreign_files/2`.

⇒ Refer to the full manual for more details.

8 The Prolog Language

This chapter provides a brief introduction to the syntax and semantics of a certain subset of logic (*definite clauses*, also known as *Horn clauses*), and indicates how this subset forms the basis of Prolog.

8.1 Syntax, Terminology and Informal Semantics

8.1.1 Terms

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

The constants include *integers* such as

```
0 1 999 -512
```

Besides the usual decimal, or base 10, notation, integers may also be written in any base from 2 to 36, of which base 2 (binary), 8 (octal), and 16 (hex) are probably the most useful. Letters `A` through `Z` (upper or lower case) are used for bases greater than 10. E.g.

```
15 2'1111 8'17
```

all represent the integer fifteen.

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

```
0'A
```

is equivalent to `65` (the numerical value of the character code for `A`).

Constants also include *floats* such as

```
1.0 -3.141 4.5E7 -0.12e+8 12.0e-9
```

Note that there must be a decimal point in floats written with an exponent, and that there must be at least one digit before and after the decimal point.

Constants also include *atoms* such as

```
a void = := 'Algol-68' []
```

Constants are definite elementary objects, and correspond to proper nouns in natural language. For reference purposes, here is a list of the possible forms which an atom may take:

1. Any sequence of alphanumeric characters (including _), starting with a lower case letter.
2. Any sequence from the following set of characters: `+-*/\^<>=` `:.?@#$&` This set can in fact be larger; see Section 8.9 [Token String], p. 41 for a precise definition.
3. Any sequence of characters delimited by single quotes. If the single quote character is included in the sequence it must be written twice, e.g. `'can''t'`.
4. Any of: `!` `;` `[]` {} Note that the bracket pairs are special: `[]` and {} are atoms but `[`, `]`, {, and } are not. However, when they are used as functors (see below) the form $\{X\}$ is allowed as an alternative to {}(*X*). The form `[`*X*`]` is the normal notation for lists, as an alternative to `.(X,[])`.

Variables may be written as any sequence of alphanumeric characters (including _) starting with either a capital letter or _; e.g.

```
X Value A A1 _3 _RESULT
```

If a variable is only referred to once in a clause, it does not need to be named and may be written as an *anonymous* variable, indicated by the underline character _. A clause may contain several anonymous variables; they are all read and treated as distinct variables.

A variable should be thought of as standing for some definite but unidentified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writeable storage location as in most programming languages; rather it is a local name for some data object, cf. the variable of pure LISP and identity declarations in Algol68.

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

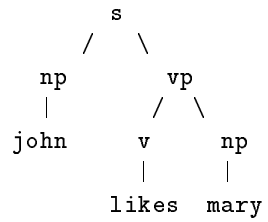
```
point(X, Y, Z)
```

Note that an atom is considered to be a functor of arity 0.

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

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

would be pictured as the structure



Sometimes it is convenient to write certain functors as operators—2-ary functors may be declared as infix operators and 1-ary functors as prefix or postfix operators. Thus it is possible to write, e.g.

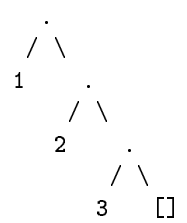
```
X+Y (P;Q) X<Y +X P;
```

as optional alternatives to

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

The use of operators is described fully below (see Section 8.6 [Operators], p. 38).

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,[])))
```

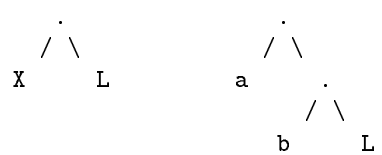
but which is normally written, in a special list notation, as

```
[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 character codes. Lists written in this notation are called *strings*. E.g.

```
"SICStus"
```

which represents exactly the same list as

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

8.1.2 Programs

A fundamental unit of a logic program is the *goal* or procedure call. e.g.

```
gives(tom, apple, teacher) reverse([1,2,3], L) X<Y
```

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

A logic program consists simply of a sequence of statements called *sentences*, which are analogous to sentences of natural language. A sentence comprises a *head* and a *body*. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (i.e.it too may be empty). If the head is not empty, the sentence is called a *clause*.

If the body of a clause is empty, the clause is called a *unit clause*, and is written in the form

P.

where *P* is the head goal. We interpret this declaratively as

> Goals matching *P* are true.

and procedurally as

> Goals matching *P* are satisfied.

If the body of a clause is non-empty, the clause is called a *non-unit clause*, and is written in the form

P `:-` *Q*`,` *R*`,` *S*.

where *P* is the head goal and *Q*, *R* and *S* are the goals which make up the body. We can read such a clause either declaratively as

> *P* is true if *Q* and *R* and *S* are true.

or procedurally as

> To satisfy goal *P*, satisfy goals *Q*, *R* and *S*.

A sentence with an empty head is called a *directive* (see Section 1.4 [Directives], p. 4), of which the most important kind is called a *query* and is written in the form

`?-` *P*`,` *Q*.

where *P* and *Q* are the goals of the body. Such a query is read declaratively as

> Are *P* and *Q* true?

and procedurally as

Satisfy goals P and Q.

Sentences generally contain variables. Note that variables in different sentences are completely independent, even if they have the same name—i.e.the *lexical scope* of a variable is limited to a single sentence. Each distinct variable in a sentence should be interpreted as standing for an arbitrary entity, or value. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

1. `employed(`X`) :- employs(`Y`,`X`).`

   "Any X is employed if any Y employs X."

   "To find whether a person X is employed, find whether any Y employs X."

2. `derivative(`X`,`X`,1).`

   "For any X, the derivative of X with respect to X is 1."

   "The goal of finding a derivative for the expression X with respect to X itself is satisfied by the result 1."

3. `?- ungulate(`X`), aquatic(`X`).`

   "Is it true, for any X, that X is an ungulate and X is aquatic?"

   "Find an X which is both an ungulate and aquatic."

In any program, the *predicate* for a particular (principal) functor is the sequence of clauses in the program whose head goals have that principal functor. For example, the predicate for a 3-ary functor `concatenate/3` might well consist of the two clauses

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

where `concatenate(`*L1*`,`*L2*`,`*L3*`)` means "the list *L1* concatenated with the list *L2* is the list *L3*". Note that for predicates with clauses corresponding to a base case and a recursive case, the preferred style is to write the base case clause first.

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form *name*`/`*arity* is used; e.g. `concatenate/3`.

Certain predicates are predefined by built-in predicates supplied by the Prolog system. Such predicates are called *built-in predicates*.

As we have seen, the goals in the body of a sentence are linked by the operator '`,`' which can be interpreted as conjunction ("and"). It is sometimes convenient to use an additional operator '`;`', standing for disjunction ("or"). (The precedence of '`;`' is such that it dominates '`,`' but is dominated by '`:-`'.) An example is the clause

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

which can be read as

> For any X, Y and Z, X has Z as a grandfather if either the mother of X is Y or the father of X is Y, and the father of Y is Z.

Such uses of disjunction can always be eliminated by defining an extra predicate—for instance the previous example is equivalent to

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

—and so disjunction will not be mentioned further in the following, more formal, description of the semantics of clauses.

The token '`|`', when used outside a list, is an alias for '`;`'. The aliasing is performed when terms are read in, so that

```
a :- b | c.
```

is read as if it were

```
a :- b ; c.
```

Note the double use of the '.' character. On the one hand it is used as a sentence terminator, while on the other it may be used in a string of symbols which make up an atom (e.g.the list functor `./2`). The rule used to disambiguate terms is that a '.' followed by *layout-text* is regarded as a sentence terminator (see Section 8.9 [Token String], p. 41).

8.2 Declarative Semantics

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

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

For example, if a program contains the preceding procedure for `concatenate/3`, then the declarative semantics tells us that

```
?- concatenate([a], [b], [a,b]).
```

is true, because this goal is the head of a certain instance of the first clause for `concatenate/3`, namely,

```
concatenate([a], [b], [a,b]) :- concatenate([], [b], [b]).
```

and we know that the only goal in the body of this clause instance is true, since it is an instance of the unit clause which is the second clause for `concatenate/3`.

8.3 Procedural Semantics

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the *procedural semantics* which Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute the program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics. We first illustrate the semantics by the simple query

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

We find that it matches the head of the first clause for `concatenate/3`, with X instantiated to `[a|`$X1$`]`. The new variable $X1$ is constrained by the new query produced, which contains a single recursive procedure call:

```
?- concatenate(X1, Y, [b]).
```

Again this goal matches the first clause, instantiating $X1$ to `[b|`$X2$`]`, and yielding the new query:

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

Now the single goal will only match the second clause, instantiating both $X2$ and Y to `[]`. Since there are no further goals to be executed, we have a solution

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

i.e.a true instance of the original goal is

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

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

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

in that order, by re-matching, against the second clause for concatenate, goals already solved once using the first clause.

Thus, in the procedural semantics, the set of clauses

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

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

?- *G1*, ..., *Gn*.

each *Gi* is regarded as a *procedure call*. To execute a query, the system selects by its *computation rule* a goal, *Gj* say, and searches by its *search rule* a clause whose head matches *Gj*. Matching is done by the *unification* algorithm (see *A Machine-Oriented Logic Based on the Resolution Principle* by J.A. Robinson, *Journal of the ACM* 12:23-44, January 1965) which computes the most general unifier, *mgu*, of *Gj* and *H*. The *mgu* is unique if it exists. If a match is found, the current query is *reduced* to a new query

?- (*G1*, ..., *Gj-1*, *B1*, ..., *Bm*, *Gj+1*, ..., *Gn*)*mgu*.

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

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

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

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

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

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

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

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

See Section 6 [Blocking], p. 31 for a description of the available facilities involving blocked goals.

8.4 Occur Check

It is possible, and sometimes useful, to write programs which unify a variable to a term in which that variable occurs, thus creating a cyclic term. The usual mathematical theory behind Logic Programming forbids the creation of cyclic terms, dictating that an *occur check* should be done each time a variable is unified with a term. Unfortunately, an occur check would so expensive as to render Prolog impractical as a programming language. Thus cyclic terms may be created and may cause loops trying to print them.

SICStus Prolog mitigates the problem by its ability to unify, compare (see Section 4.3 [Term Compare], p. 22), assert, and copy cyclic terms without looping. Loops in the printer can be interrupted by typing `^C`.

8.5 The Cut Symbol

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

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

For example:

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

This predicate can be used to test whether a given term is in a list. E.g.

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

returns the answer '`yes`'. The predicate can also be used to extract elements from a list, as in

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

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

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

In this case, the above call would extract only the first element of the list (`d`). On backtracking, the cut would immediately fail the whole predicate.

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

This is equivalent to

```
x := if p then q else r;
```

in an Algol-like language.

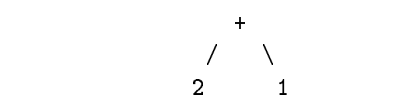
It should be noticed that a cut discards all the alternatives since the parent goal, even when the cut appears within a disjunction. This means that the normal method for eliminating a disjunction by defining an extra predicate cannot be applied to a disjunction containing a cut.

A proper use of the cut is usually a major difficulty for new Prolog programmers. The usual mistakes are to over-use cut, and to let cuts destroy the logic. We would like to advise all users to follow these general rules. Also see Section 9 [Example Intro], p. 41.

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

8.6 Operators

Operators in Prolog are simply a *notational convenience*. For example, the expression `2+1` could also be written `+(2,1)`. This expression represents the data structure



and *not* the number 3. The addition would only be performed if the structure were passed as an argument to an appropriate predicate such as `is/2` (see Section 4.2 [Arithmetic], p. 19).

The Prolog syntax caters for operators of three main kinds—*infix*, *prefix* and *postfix*. An infix operator appears between its two arguments, while a prefix operator precedes its single argument and a postfix operator is written after its single argument.

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

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

are equivalent and denote the term `+(a,/(b,c))`. Note that the infix form of the term `/(+(a,b),c)` must be written with explicit parentheses, i.e.

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

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

```
xfx xfy yfx
```

Operators of type `xfx` are not associative: it is a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself, i.e.their principal functors must be of lower precedence, unless the subexpression is explicitly parenthesized (which gives it zero precedence).

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

A functor named name is declared as an operator of type *Type* and precedence precedence by the command

```
:- op(precedence, type, name).
```

The argument name can also be a list of names of operators of the same type and precedence.

It is possible to have more than one operator of the same name, so long as they are of different kinds, i.e.infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to operators which are provided as standard. Declarations of all the standard operators can be found elsewhere (see Section 11 [Standard Operators], p. 51).

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

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

so that

```
a-b+c
```

is valid syntax, and means

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

i.e.

```
    +
   / \
  -   c
 / \
a   b
```

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

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

then `a.b.c` would represent the structure

```
  .
 / \
a   .
   / \
  b   c
```

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

The possible types for a prefix operator are

```
fx fy
```

and for a postfix operator they are

```
xf yf
```

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

```
not not P
```

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

```
not P
```

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

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

Note that the arguments of a compound term written in standard syntax must be expressions of precedence *below* 1000. Thus it is necessary to parenthesize the expression `P :- Q` in

```
| ?- assert((P :- Q)).
```

8.7 Syntax Restrictions

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

1. In a term written in standard syntax, the principal functor and its following `(` must *not* be separated by any intervening *layout-text*. 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 *layout-char*. Thus
   ```
   :-(p;q),r.
   ```
   (where '`:-`' is the prefix operator) is invalid syntax. The system would try to interpret it as the structure:

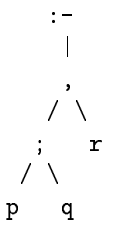
   ```
      ,
     / \
   :-   r
    |
    ;
   / \
  p   q
   ```

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

   In contrast, the term:

```
    :- (p;q),r.
```

is valid syntax and represents the following structure.



8.8 Comments

Comments have no effect on the execution of a program, but they are very useful for making programs more readily comprehensible. Two forms of comment are allowed in Prolog:

1. The character `%` followed by any sequence of characters up to end of line.
2. The symbol `/*` followed by any sequence of characters (including new lines) up to `*/`.

8.9 Full Prolog Syntax

⇒ Refer to the full manual for more details.

9 Programming Examples

⇒ Refer to the full manual for more details.

10 Summary of Built-In Predicates

Refer to Section 4.2 (Arithmetic) for arithmetic functors.

Note that some predicates are not available or works differently in Runtime Systems.

`!` Commits to any choices taken in the current predicate.

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

`(`*Head* `-->` *Body*`)` Not a built-in predicate; reserved syntax for grammar rules.

`(`*+P* `->` *+Q* `;` *+R*`)` If *P* then *Q* else *R*, using first solution of *P* only.

`(`*+P* `->` *+Q*`)` If *P* then *Q* else false, using first solution of *P* only.

`[]`

`[`*:File*`|`*+Files*`]` Updates the program with interpreted clauses from *File* and *Files* (not available in Runtime Systems).

`(:-` *Command*`)` Not a built-in predicate; reserved syntax for commands and declarations.

`(?-` *Query*`)` Not a built-in predicate; reserved syntax for queries.

`(`*Head* `:-` *Body*`)` Not a built-in predicate; reserved syntax for clauses.

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

?X `=` *?Y* The terms *X* and *Y* are unified.

+Term `=..` *?List*

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

+X `=:=` *+Y* *X* is numerically equal to *Y*.

?Term1 `==` *?Term2* The terms *Term1* and *Term2* are strictly identical.

+X `=\=` *+Y* *X* is not numerically equal to *Y*.

+X `=<` *+Y* *X* is less than or equal to *Y*.

+X `>` *+Y* *X* is greater than *Y*.

+X `>=` *+Y* *X* is greater than or equal to *Y*.

?X `^` *:P* Executes the procedure call *P*.

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

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

+X `<` *+Y* *X* is less than *Y*.

?Term1 `@=<` *?Term2* The term *Term1* precedes or is identical to the term *Term2* in the standard order.

?Term1 `@>` *?Term2* The term *Term1* follows the term *Term2* in the standard order.

?Term1 `@>=` *?Term2* The term *Term1* follows or is identical to the term *Term2* in the standard order.

?Term1 `@<` *?Term2* The term *Term1* precedes the term *Term2* in the standard order.

`?=(`*?X*`,`*?Y*`)` *X* and *Y* are either syntactically identical or syntactically non-unifiable.

`abolish(`*:Preds*`)` Makes the predicate(s) specified by *Preds* undefined.

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

`abort` Aborts execution of the current directive (returns to C in recursive calls to Prolog from C).

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

`arg(`*+ArgNo*`,`*+Term*`,`*?Arg*`)` The argument *ArgNo* of the term *Term* is *Arg*.

`assert(`*:Clause*`)`

`assert(`*:Clause*`,-`*Ref*`)` Asserts clause *Clause* with unique identifier *Ref*.

`asserta(`*:Clause*`)`

`asserta(`*:Clause*`,-`*Ref*`)` Asserts *Clause* as first clause with unique identifier *Ref*.

`assertz(`*:Clause*`)`

`assertz(`*:Clause*`,-`*Ref*`)` Asserts *Clause* as last clause with unique identifier *Ref*.

`at_end_of_line`

`at_end_of_line(`*Stream*`)` The end of stream or end of line has been reached for *Stream* or from the current input stream.

`at_end_of_stream`

`at_end_of_stream(`*Stream*`)` The end of stream has been reached for *Stream* or from the current input stream.

`atom(`*?X*`)` *X* is currently instantiated to an atom.

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

`atom_chars(`*?Atom*`,`*+CharList*`)` The name of the atom *Atom* is the list of characters *CharList*.

`atomic(`*?X*`)` *X* is currently instantiated to an atom or a number.

`bagof(`*?Template*`,`*:Goal*`,`*?Bag*`)` *Bag* is the bag of instances of *Template* such that *Goal* is satisfied (not just provable).

`block` *Specs* Not a built-in predicate; block declaration.

`bb_delete(`*+Key*`,`*?Term*`)` Delete from the blackboard *Term* stored under *Key*.

`bb_get(`*+Key*`,`*?Term*`)` Get from the blackboard *Term* stored under *Key*.

`bb_put(`*+Key*`,`*+Term*`)` Store the term *Term* under *Key* on the blackboard.

`bb_update(`*:Key*`, `*?OldTerm*`, `*?NewTerm*`)` Replace the term *OldTerm* by the term *NewTerm* under *Key* on the blackboard.

`break` Invokes a recursive top-level. (not available in Runtime Systems).

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

`call(`*:Term*`)`

`(`*Module*`::`*Term*`)` Executes the procedure call *Term* in *Module*.

`call_residue(`*:Goal*`,`*?Residue*`)` Executes the procedure call *Goal*. Any floundered goals and the variables they are blocked on occur as *VarSet-Goal* pairs in *Residue*.

`callable(`*?X*`)` *X* is currently instantiated to a compound term or an atom.

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

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

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

`clause(`*?Head*`,`*?Body*`,`*+Ref*`)` There is an interpreted clause whose head is *Head*, whose body is *Body*, and whose unique identifier is *Ref*.

`close(`*+Stream*`)` Closes stream *Stream*.

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

`compile(`*:Files*`)` Compiles in-core the clauses in text file(s) *Files* (not available in Runtime Systems).

`compound(`*?X*`)` *X* is currently instantiated to a term of arity > 0.

`consult(`*:Files*`)` Updates the program with interpreted clauses from file(s) *Files* (not available in Runtime Systems).

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

`create_mutable(`*+Datum*`,-`*Mutable*`)` *Mutable* is a new mutable term with current value *Datum*.

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

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

`current_key(`*?KeyName*`,`*?KeyTerm*`)` There is a recorded item in the internal database whose key is *KeyTerm*, the name of which is *KeyName*.

`current_module(`*?Module*`)` *Module* is a module currently in the system.

`current_module(`*?Module*`,`*?File*`)` *Module* is a module currently in the system, loaded from *File* (not available in Runtime Systems).

`current_op(?Precedence,?Type,?Op)` Atom *Op* is an operator type *Type* precedence *Precedence*.

`current_output(?Stream)` *Stream* is the current output stream.

`current_predicate(?Name,:Head)`

`current_predicate(?Name,-Head)` A user defined predicate is named *Name*, most general goal *Head*.

`current_stream(?AbsFileName,?Mode,?Stream)` There is a stream *Stream* associated with the file *AbsFileName* and opened in mode *Mode*.

`debug` Switches on debugging (not available in Runtime Systems).

`debugging` Displays debugging status information (not available in Runtime Systems).

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

`display(?Term)` Displays the term *Term* on the standard output stream.

`dynamic` *Specs* Not a built-in predicate; dynamic declaration.

`ensure_loaded(:Files)` Compiles or loads the file(s) *Files* if need be. (In Runtime Systems it is identical to `load/1`.)

`erase(+Ref)` Erases the clause or record whose unique identifier is *Ref*.

`error_exception(+Exception)`

`user:error_exception(+Exception)` *A hook predicate*, *Exception* is an exception that traps to the debugger if it is switched on.

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

`fail`

`false` False.

`fcompile(:Files)` Compiles file-to-file the clauses in text file(s) *Files* (not available in Runtime Systems).

`file_search_path(Alias,?Expansion)`

`user:file_search_path(Alias,?Expansion)` *A hook predicate*, telling how to expand *Alias*(*File*) file names.

`fileerrors` Enables reporting of file errors.

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

`findall(?Template,:Goal,?Bag,?Remainder)` A prefix of *Bag* is the list of instances of *Template* such that *Goal* is provable. The rest of *Bag* is *Remainder* or the empty list.

`float(?X)` *X* is currently instantiated to a float.

`flush_output`

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

`foreign(+CFunctionName,+Predicate)`

`foreign(+CFunctionName,+Language,+Predicate)` *Hook predicates*, tell Prolog how to define *Predicate* to invoke *CFunctionName*.

`foreign_file(+ObjectFile,+Functions)` *A hook predicate*, tells Prolog that foreign functions *Functions* are in file *ObjectFile*.

`format(+Format,+Arguments)`

`format(`*+Stream*`,`*+Format*`,`*+Arguments*`)` Writes *Arguments* according to *Format* on the stream *Stream* or on the current output stream.

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

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

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

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

`garbage_collect` Performs a garbage collection.

`gc` Enables garbage collection.

`get(`*?C*`)`

`get(`*+Stream*`,`*?C*`)` The next printing character from the stream *Stream* or from the current input stream is *C*.

`get0(`*?C*`)`

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

`get_mutable(`*?Datum*`,`*+Mutable*`)` The current value of the mutable term *Mutable* is *Datum*.

`goal_expansion(`*+Goal*`,`*+Module*`,`*-NewGoal*`)`

`user:goal_expansion(`*+Goal*`,`*+Module*`,`*-NewGoal*`)` *A hook predicate.* Defines an alternative goal to be called in place of a call to an unknown predicate.

`ground(`*?X*`)` *X* is currently free of unbound variables.

`halt` Halts Prolog. (returns to C in recursive calls to Prolog from C).

`halt(`*Code*`)` Halts Prolog immediately, returning *Code*.

`help` *Hookable*, prints a help message (not available in Runtime Systems).

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

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

`initialization` Executes a list of user defined goals (not available in Runtime Systems).

`initialization(`*:Goal*`)` Adds *Goal* to the list of goals executed by `initialization/0` (not available in Runtime Systems).

`instance(`*+Ref*`,`*?Term*`)` *Term* is a most general instance of the record or clause uniquely identified by *Ref*.

`integer(`*?X*`)` *X* is currently instantiated to an integer.

?Y `is` *+X* *Y* is the value of the arithmetic expression *X*.

`keysort(`*+List1*`,`*?List2*`)` The list *List1* sorted by key yields *List2*.

`leash(`*+Mode*`)` Sets leashing mode to *Mode* (not available in Runtime Systems).

`length(`*?List*`,`*?Length*`)` The length of list *List* is *Length*.

`library_directory(`*?Directory*`)`

`user:library_directory(`*?Directory*`)` *A hook predicate*, telling how to expand `library(`*File*`)` file names.

`line_count(`*+Stream*`,`*?N*`)` *N* is the number of lines read/written on stream *Stream*.

`line_position(+Stream,?N)` *N* is the number of characters read/written on the current line of *Stream*.

`listing`

`listing(:Specs)` Lists the interpreted predicate(s) specified by *Specs* or all interpreted predicates in the type-in module. Any variables in the listed clauses are internally bound to ground terms before printing. If this causes any blocked goals to be executed, the behavior is undefined.

`load(:Files)` Loads fcompiled file(s) *Files* into Prolog (will not write any progress or error messages in Runtime Systems.)

`load_foreign_files(:ObjectFiles,+Libraries)` *Hookable*, links object files *ObjectFiles* into Prolog.

`meta_predicate` *Specs* Not a built-in predicate; meta_predicate declaration.

`mode` *Specs* Not a built-in predicate; mode declaration.

`module(+Module)` Sets the type-in module to *Module*; See Section 5 [Basic Concepts], p. 31 (not available in Runtime Systems).

`multifile` *Specs* Not a built-in predicate; multifile declaration.

`muse_flag(?FlagName,?Value)` *Value* is the current value of the Muse flag *FlagName*.

`muse_flag(+FlagName,?OldValue,?NewValue)` *OldValue* and *NewValue* are the old and new values of the Muse flag *FlagName*.

`muse_flags` Display all Muse flag settings.

`muse_sync` Synchronize for being leftmost in Muse.

`muse_trace(:Goal)`

`muse_trace(:Goal,+FilePrefix)` Generate a Muse trace file for *Goal*.

`name(+Const,?CharList)`

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

`nl`

`nl(+Stream)` Outputs a new line on stream *Stream* or on the current output stream.

`nodebug` Switches off debugging (not available in Runtime Systems).

`nofileerrors` Disables reporting of file errors.

`nogc` Disables garbage collection.

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

`nospy` *:Spec* Removes spy-points from the predicate(s) specified by *Spec* (not available in Runtime Systems).

`nospyall` Removes all spy-points (not available in Runtime Systems).

`notrace` Switches off debugging (not available in Runtime Systems).

`number(?X)` *X* is currently instantiated to a number.

`number_chars(+Number,?CharList)`

`number_chars(?Number,+CharList)` The name of the number *Number* is the list of characters *CharList*.

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

`on_exception(?Pattern,:ProtectedGoal,:Handler)` Executes the procedure call *ProtectedGoal*. If during the execution `raise_exception(`*Exception*`)` is called, and *Exception* matches *Pattern*, the execution of *ProtectedGoal* abort, *Pattern* is unified with a copy of *Exception* and *Handler* is called.

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

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

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

`otherwise` True.

`parallel`

`parallel` *Specs* Not a built-in predicate; parallel declaration.

`peek_char(?N)`

`peek_char(+Stream,?N)` *N* is the character code of the next character peeked at from *Stream* or from the current input stream.

`phrase(:Phrase,?List)`

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

`portray(+Term)`

`user:portray(+Term)` *A hook predicate*, tells `print/1` what to do.

`portray_clause(+Clause)`

`portray_clause(+Stream,+Clause)` Pretty prints *Clause* on the stream *Stream* or on the current output stream.

`portray_message(+Severity,+Message)`

`user:portray_message(+Severity,+Message)` *A hook predicate*, tells `print_message/2` what to do.

`predicate_property(:Head,?Prop)`

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

`prepare_foreign_files(+ObjectFiles)` Generates interface code for *ObjectFiles* (not available in Runtime Systems).

`print(?Term)`

`print(+Stream,?Term)` *Hookable*, portrays or else writes the term *Term* on the stream *Stream* or on the current output stream.

`print_message(+Severity,+Message)` *Hookable*, portrays or else writes *Message* of a given *Severity* on the standard error stream.

`profile_data(:Files,?Selection,?Resolution,-Data)` *Data* is the profiling data collected from the instrumented predicates defined in the file(s) *Files* with selection and resolution *Selection* and *Resolution* respectively (not available in Runtime Systems nor in Muse).

`profile_reset(:Files)` The profiling counters for the instrumented predicates in the file(s) *Files* are zeroed (not available in Runtime Systems nor in Muse).

`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*.

`prolog_load_context(?Key,?Value)` *Value* is the value of the compilation/loading context variable identified by *Key*.

`prompt(?Old,?New)` Changes the prompt from *Old* to *New*.

`public` *Specs* Not a built-in predicate; public declaration.

`put(`*+C*`)`

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

`raise_exception(`*+Exception*`)` Causes abortion of a part of the execution tree scoped by the closest enclosing `on_exception/3` invocation with its first argument matching *Exception*.

`read(`*?Term*`)`

`read(`*+Stream*`,`*?Term*`)` Reads the term *Term* from the stream *Stream* or from the current input stream.

`read_term(`*?Term*`,`*+Options*`)`

`read_term(`*+Stream*`,`*?Term*`,`*+Options*`)` Reads the term *Term* from the stream *Stream* or from the current input stream with extra *Options*.

`reconsult(`*:Files*`)` Updates the program with interpreted clauses from file(s) *Files* (not available in Runtime Systems).

`recorda(`*+Key*`,`*?Term*`,-`*Ref*`)` Makes the term *Term* the first record under key *Key* with unique identifier *Ref*.

`recorded(`*?Key*`,`*?Term*`,`*?Ref*`)` The term *Term* is currently recorded under key *Key* with unique identifier *Ref*.

`recordz(`*+Key*`,`*?Term*`,-`*Ref*`)` Makes the term *Term* the last record under key *Key* with unique identifier *Ref*.

`reinitialise` Initializes Prolog, reconsulting ' `/.sicstusrc`' if it exists (returns to C in recursive calls to Prolog from C).

`repeat` Succeeds repeatedly.

`require(`*:PredSpecs*`)` Tries to locate and load library files that export the specified predicates. Creates index files if necessary (not available in Runtime Systems).

`restore(`*+File*`)` Restores the state saved in file *File* (not available in Runtime Systems).

`retract(`*:Clause*`)` Erases repeatedly the next interpreted clause of form *Clause*.

`retractall(`*:Head*`)` Erases all clauses whose head matches *Head*.

`save(`*+File*`)`

`save(`*+File*`,`*?Return*`)` Saves the current state of Prolog in file *File*; *Return* is 0 after a save and 1 after a restore (not available in Runtime Systems).

`save_program(`*+File*`)` Saves the current state of the Prolog data base in file *File* (not available in Runtime Systems).

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

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

`seek(`*+Stream*`,`*+Offset*`,`*+Method*`,-`*NewLocation*`)` Sets the stream *Stream* to the byte offset *Offset* relative to *Method*, and *NewLocation* is the new byte offset from the beginning of the file after the operation.

`seen` Closes the current input stream.

`sequential`

`sequential` *Specs* Not a built-in predicate; sequential declaration.

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

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

`set_stream_position(+Stream,+Position)` *Position* is a term representing a new position of *Stream*, which is then set to the new position.

`setof(?Template,:Goal,?Set)` *Set* is the set of instances of *Template* such that *Goal* is satisfied (not just provable).

`simple(?X)` *X* is currently uninstantiated or atomic.

`skip(+C)`

`skip(+Stream,+C)` Skips characters from *Stream* or from the current input stream until after character *C*.

`skip_line`

`skip_line(+Stream)` Skips characters from *Stream* or from the current input stream until the next `LFD`.

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

`source_file(?File)`

`source_file(:Pred,?File)`

`source_file(-Pred,?File)` The predicate *Pred* is defined in the file *File* (not available in Runtime Systems.)

`spy` *:Spec* Sets spy-points on the predicate(s) specified by *Spec* (not available in Runtime Systems).

`spypoint_condition(:Goal,?Port,+Test)` Sets conditional spy-point on the predicate for *Goal* (not available in Runtime Systems).

`statistics` Outputs various execution statistics.

`statistics(?Key,?Value)` The execution statistics key *Key* has value *Value*.

`stream_code(+Stream,?StreamCode)`

`stream_code(?Stream,+StreamCode)` *StreamCode* is an integer representing a pointer to the internal representation of *Stream*.

`stream_interrupt(?Stream,?OldHandler,?NewHandler)` Sets/reads the interrupt handler connected to the stream *Stream* (not available in Muse).

`stream_position(+Stream,?Position)` *Position* is a term representing the current position of *Stream*.

`stream_select(+Streams,+TimeOut,-ReadStreams)` Returns a list *ReadStreams* containing streams with pending characters. Only the streams in the list *Streams* are checked. *TimeOut* specifies a timeout on the form `off` or `Sec:MicroSec` (not available in Muse).

`tab(+N)`

`tab(+Stream,+N)` Outputs *N* spaces to the stream *Stream* or to the current output stream.

`tell(+File)` Makes file *File* the current output stream.

`telling(?File)` The current output stream is named *File*.

`term_expansion(+Term1,?Term2)`

`user:term_expansion(+Term1,?Term2)` *A hook predicate*, tells `expand_term/2` what to do.

`told` Closes the current output stream.

`trace` Switches on debugging and start tracing immediately (not available in Runtime Systems).

`true` Succeeds.

`ttyflush` Flushes the standard output stream buffer.

`ttyget(`*?C*`)` The next printing character input from the standard input stream is *C*.

`ttyget0(`*?C*`)` The next character input from the standard input stream is *C*.

`ttynl` Outputs a new line on the standard output stream.

`ttyput(`*+C*`)` The next character output to the standard output stream is *C*.

`ttyskip(`*+C*`)` Skips characters from the standard input stream until after character *C*.

`ttytab(`*+N*`)` Outputs *N* spaces to the standard output stream.

`undo(`*:Term*`)` The goal `call(`*Term*`)` is executed on backtracking (not available in Muse).

`unknown(`*?OldState*`,`*?NewState*`)` Changes action on undefined predicates from *OldState* to *NewState* (not available in Runtime Systems).

`unknown_predicate_handler(`*+Goal*`,`*+Module*`,-`*NewGoal*`)`

`user:unknown_predicate_handler(`*+Goal*`,`*+Module*`,-`*NewGoal*`)` *A hook predicate.* Defines an alternative goal to be called in place of a call to an unknown predicate.

`update_mutable(`*+Datum*`,`*+Mutable*`)` Updates the current value of the mutable term *Mutable* to become *Datum*.

`use_module(`*:Files*`)` Loads the module file(s) *Files* if necessary and import all public predicates. (In Runtime Systems, it can only load '`.ql`' files.)

`use_module(`*:File*`,`*+PredicateList*`)` Loads the module file *File* if necessary and imports the predicates in *PredicateList*. (In Runtime Systems, it can only load '`.ql`' files.)

`use_module(`*+Module*`,`*?File*`,`*+PredicateList*`)`

`use_module(`*?Module*`,`*:File*`,`*+PredicateList*`)` Imports *PredicateList* from *Module* if it is instantiated and the module is currently in the system, otherwise behaves like `use_module/2` plus unifies *Module* to the module defined by the file.

`user_help`

`user:user_help` *A hook predicate*, tells `help/0` what to do.

`var(`*?X*`)` *X* is currently uninstantiated.

`version` Displays introductory and/or system identification messages (not available in Runtime Systems).

`version(`*+Message*`)` Adds the atom *Message* to the list of introductory messages (not available in Runtime Systems).

`when(`*+Condition*`,`*:Goal*`)` Blocks *Goal* until the *Condition* is true.

`write(`*+Term*`)`

`write(`*+Stream*`,`*+Term*`)` Writes the term *Term* on the stream *Stream* or on the current output stream.

`write_canonical(`*+Term*`)`

`write_canonical(`*+Stream*`,`*+Term*`)` Writes *Term* on the stream *Stream* or on the current output stream so that it may be read back.

`write_term(`*+Term*`,`*+Options*`)`

`write_term(`*+Stream*`,`*+Term*`,`*+Options*`)` Writes the term *Term* on the stream *Stream* or on the current output stream with extra *Options*.

`writeq(`*+Term*`)`

`writeq(`*+Stream*`,`*+Term*`)` Writes the term *Term* on the stream *Stream* or on the current output stream, quoting names where necessary.

11 Standard Operators

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic,
                    multifile, block, meta_predicate,
                    parallel, sequential ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).      /* See note below */
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                 =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, #, /\, \ ]).
:- op(  500,  fx, [ +, - ]).
:- op(  400, yfx, [ *, /, //, <<, >> ]).
:- op(  300, xfx, [ mod ]).
:- op(  200, xfy, [ ^ ]).
```

Note that a comma written literally as a punctuation character can be used as though it were an infix operator of precedence 1000 and type `xfy`, i.e.

```
X,Y ','(X,Y)
```

represent the same compound term. But note that a comma written as a quoted atom is *not* a standard operator.