SICStus Prolog

User's Manual
(Summary)

Prepared by Jimmy H.M. Lee

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

Contents

1 An Introduction to SICStus Prolog **1**
- 1.1 Running SICStus Prolog . . . 2
- 1.2 Loading Programs . . . 2
- 1.3 Interrupting a Running Prolog Program . . . 3
- 1.4 Splitting programs . . . 3
- 1.5 Dynamic Predicates . . . 3

2 The Syntax of SICStus Prolog **4**
- 2.1 Constants . . . 4
- 2.2 Variables . . . 4
- 2.3 Compound Terms . . . 5
- 2.4 Programs . . . 5
- 2.5 Miscellaneous . . . 6

3 Debugging **6**
- 3.1 Leashing . . . 7
- 3.2 Spying on Code . . . 7
- 3.3 Traveling Between Ports . . . 7
- 3.4 Basic Traveling Commands . . . 8

4 Definite Clause Grammars **9**
- 4.1 An Example . . . 10
- 4.2 How Grammar Rules Are Translated into Prolog Clauses . . . 10

5 Built-in Predicates **11**
- 5.1 Arithmetic . . . 11
- 5.2 Term Comparison . . . 12
- 5.3 Input/Output . . . 12
- 5.4 Control . . . 13
  - 5.4.1 Cut . . . 13
  - 5.4.2 Negation As Failure . . . 13
- 5.5 Meta-logical . . . 14
- 5.6 Program Modification . . . 14
- 5.7 All Solution Predicates . . . 15

1 An Introduction to SICStus Prolog

The SICStus Prolog system is a compiler-based Prolog development environment which is compatible with the Edinburgh Prolog. Programs written for an Edinburgh standard system should run in SICStus Prolog with little or no modification.
You should view this document as a reference summary but not an introduction to Prolog. You may acquaint yourselves with the Edinburgh-style Prolog in the following books:

- Clocksin,W.F., Mellish,C.S., Programming in Prolog, Springer-Verlag, 1984 (QA76.73 P76C57).
- Bratko,I., Prolog Programming for Artificial Intelligence, AddisonWesley, 1986 (Q336 B74).
- Sterling,L., Shapiro,E., The Art of Prolog: Advanced Programming Techniques, The MIT Press, 1987 (QA76.73 P76S74).

This note is by no means a replacement for the SICStus Prolog User's Manual (referred to as the user's manual hereafter) either. Readers should consult the user's manual for full details of the SICStus Prolog system.

1.1 Running SICStus Prolog

The SICStus Prolog system can be started by typing at your operating system prompt:

```
% sics
```

The system automatically loads the file of system builtin predicates. It starts up by printing the following banner:

```
SICStus 2.1 #8:  Sun Sep 26 19:53:33 HKT 1993
| ?-
```

The prompt "`| ?-`" is the normal top level Prolog prompt. It indicates that SICStus Prolog is ready to accept queries. There are several other more advanced ways of invoking SICStus Prolog. Please refer to the manual page on your system.
There are two normal ways to exit SICStus Prolog:

- typing a Control-D, or
- entering the goal `halt.` at the top level.

```
| ?- halt.
```

Note that a period followed by a RETURN *must always be typed after a goal.*

1.2 Loading Programs

SICStus Prolog recognizes files with the "`.pl`" suffix as Prolog program files. Suppose your program resides in "`myfile.pl`." You can load your program into Prolog by *consulting* the file "`myfile.pl`" using one of the followings:

- `| ?- [myfile].`
- `| ?- consult(myfile).`
- `| ?- compile(myfile).`

The effect of `compile/1` is very much like that of `consult/1`, except that all new procedures will be stored in compiled rather than interpreted form. If you have several files to load, then you can use, for example:

- `| ?- [myfile1,myfile2,myfile3].`
- `| ?- consult([myfile1,myfile2,myfile3]).`
- `| ?- compile([myfile1,myfile2,myfile3]).`

If you are interested in examining what has been loaded, you can use the `listing/1` predicate to list all the clauses which were successfully read-in, e.g.

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

1.3 Interrupting a Running Prolog Program

SICStus Prolog has a simple break handler which is invoked when Control-C is pressed. If a program is in an infinite loop, or if it is just executing longer than you expect, you can press Control-C, to which Prolog will respond by displaying the message

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

At this point, you can either type "`h`" to see a list of the options available to you, as show below, or you can simply type the letter that corresponds to the option you want to select. If you type "`h`", Prolog displays the following list of options:

```
Prolog interrupt options:
a  abort     -  cause abort
b  break     -  cause break
c  continue  -  do nothing
d  debug     -  start debugging
e  exit      -  cause exit
t  trace     -  start tracing
h  help      -  get this list
Prolog interruption (h for help)?
```

To select an option, type the letter that corresponds to that option and press the RETURN key.

1.4 Splitting programs

SICStus Prolog provides modules facilities[1] and commands in files to facilitate software engineering practices. For example, you may want to split a large program across several files. Suppose you have a main program file and three auxiliary program files `file1.pl`, `file2.pl`, and `file3.pl`. You can put the following line in your main program file

```
:- [file1,file2,file3].
```

In Prolog, you only need to consult your main program file to load the entire program.

1.5 Dynamic Predicates

All predicates in Prolog fall into one of two categories: *static* or *dynamic*. Dynamic predicates can be modified when a program is running; in contrast, static predicates can be modified only by reloading or by `abolish/[1,2]` (see section 5.6).
If a predicate is first defined by being loaded from a file, it is static by default. Sometimes, however, it is necessary to add (assert), remove (retract), or inspect (using `clause/[2,3]`) clauses for a predicate while a program is running (see section 5.6). In order to do that, you must declare the predicate to be dynamic. You can do that by either making explicit declarations (as described below) or using one of the assertion predicates.
To make a dynamic predicate, you insert in the file containing the predicate a declaration, which is of the form

```
:- dynamic name1/arity1,name2/arity2,...,nameN/arityN.
```

[1] We shall not discuss the module facilities in this note. For details, please refer to the user's manual.

2 The Syntax of SICStus Prolog

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

2.1 Constants

A constant is either a *number* (integer or floating-point) or an *atom*.

Numbers The printed form of an integer consists of a sequence of digits optionally preceded by a minus sign (-). These are normally interpreted as base 10 integers. It is also possible to enter integers in other bases (1 through 36); this is done by preceding the digit string by the base (in decimal) followed by an apostrophe. If a base greater than 10 is used, the characters `A-Z` or `a-z` are used to stand for digits greater than 9. Examples of valid integer representations are:

```
1      -2345      85923      8'777      16'3F4A      -12'2A9
```

Note that `+525` is not a valid integer.
A base of zero will return the ASCII code of the (single) character after the quote; for example, `0'a = 97`.
A floating-point number (float) consists of a sequence of digits with an embedded decimal point, optionally preceded by a minus sign (-), and optionally followed by an exponent consisting of upper- or lowercase "`E`" and a signed base 10 integer. Examples of floats are:

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

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

Atoms An atom is identified by its name, which is a sequence of up to 1023 characters (other than the null characters). Atoms are definite elementary objects, and correspond to proper nouns in natural language. For reference purpose, here is a list of the possible forms which an atom may take:

- Any sequence of alphanumeric characters (including "_"), starting with a lowercase letter. Note that an atom may not begin with an underscore.
- Any sequence from the following set of characters (except "`/*`", which begins a comment):

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

- Any sequence of characters delimited by single quotes, such as:

  ```
  'yes'  '%'
  ```

  If a single quote character is included in the sequence it must be written twice, for example:

  ```
  'can''t' ''''
  ```

- And other obscure types that you don't need to know!

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

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

If a variable is 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.

2.3 Compound Terms

The structured data objects of the language are the compound terms. A *compound* (or structured) term is written as a function symbol (a non-numeric constant) followed by its arguments (separated by commas) enclosed in parentheses. The number of arguments is called the *arity* of the structure. For example, "`f(a,b(X),Y)`" has arity 3, function symbol "`f`" and principal functor "`f/3`". Note that a functor of arity 0 is an atom.

Lists List processing is so common in Prolog programming that there is a special notation for *lists*. The simplest list is the empty list, represented by the atom "`[].`". Any nonempty list is a compound term with "`./2`" as principal functor. The following are all different ways of writing the same list:

```
[a,b,c]   .(a,.(b,.(c,[])))   [a|[b|c|[]]]   [a|[b,c]]
```

Strings For convenience, a further notational variant is allowed for lists of integers that corresponds to ASCII character codes. Lists written in this notation are called *strings*. For example,

```
"Humpty-Dumpty"
```

represents exactly the same list as

```
[72,117,109,112,116,121,45,68,117,109,112,116,121]
```

2.4 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. 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. A sentence with an empty head is called a *directive*, of which the most important kind is called a *query* and is written in the form

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

where P and Q are the goals of the body.
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.

2.5 Miscellaneous

Operators Operators in Prolog are simply a notational convenience. Please refer to user's manual for details.

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

- The character "`%`" followed by any sequence of characters up to the end of the line.
- The symbol "`/*`" followed by any sequence of characters (including newlines) up to the symbol "`*/`."

3 Debugging

During debugging, the debugger stops at various points in the execution of a goal, allowing you see what arguments the goal is called with, what arguments it returns with, and whether it succeeds or fails. As in other programming languages, key points of interest are procedure entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly begins backtracking. The *Procedure Box* model of Prolog execution views program control flow in terms of movement from clause to clause in the procedures of the program. This model provides a basis for the debugging mechanism and enables the user to view program behavior in a consistent way.
Each predicate definition (viewed as a procedure) can be considered as a black box with five ports: *call*, *exit*, *redo*, *fail*, and *exception*. The control flow of a Prolog program can be viewed as motion through one of the five ports:

Call	The entry point into a procedure the first time it is called.
Exit	The port through which execution passes if the procedure succeeds.
Redo	The port through which execution passes when a later goal has failed and the procedure must try and find another solution if possible.
Fail	The port through which execution passes if the procedure fails completely.
Exception	The port through which execution passes if an exception was raised in the initial goal, either by a call to `raise_exception/1` or by an error in a built-in predicate.

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.

3.1 Leashing

At any time, there is a leashing mode in force which determines at which of the five ports of a procedure box (Call, Exit, Redo, Fail, and Exception) the debugger will stop and wait for a command when you creep through your program. By default, the debugger will stop at every port, but sometimes you may wish to reduce the number of times you have to issue commands. To set the leashing mode, that is, to specify ports for leashing, use the "`leash/1`" predicate, the argument of which is a subset of the following, specified as a list: `call`, `exit`, `redo`, `fail`, `exception`. The initial value of the leashing mode is `[call,exit,redo,fail,exception]`.

3.2 Spying on Code

Spypoints allow you to indicate where to stop on a per-predicate or even per-goal basis. For example, you might find that you want to run until some particular predicate is called. In this case, you would set a spypoint on that predicate, using "`spy/1`." Such spypoints are turned off with "`nospy/1`." Spypoints are included in the output of "`debugging/0`." Finally, spypoints can be removed all at once with "`nospyall/0`." The debuggers also have commands for setting spypoints, which are easier to use than these predicates.

3.3 Traveling Between Ports

The built-in predicates "`trace/0`" and "`debug/0`" can be used to start the debugger and place it in the *trace* and *debug* modes respectively. When the trace mode is in force, every time you type a goal at the top level, the debugger starts single-stepping (or creeping as described below) immediately. In contrast, while the debug mode is in force, the debugger does nothing until a call is made to a procedure or goal on which you have placed a spypoint (described below). The predicate "`nodebug/0`" is used to turn off the debugger. Once you are inside the debugger, you can use the commands described in the following subsections to travel between the ports. Typing "`h`" or "`?`" in the debugger gives the following menu:

```
Debugging options:

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

The built-in predicate "`debugging/0`" can be used to find out the current debugging state information.

3.4 Basic Traveling Commands

c,RET *creep* causes the interpreter to single-step to the very next port and print a message. Then if the port is leashed, 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 Control-C interrupt will disable the masking. Normally, however, this masking is just what is required!

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

If you supply an integer after the retry command, then this is taken as specifying an invocation number and the system tries to get you to the Call port, not of the current box, but of the invocation box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts (`!`) in your program. In this case the system fails to the latest surviving Call port before the correct one.

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

If you supply an integer after the command, then this is taken as specifying an invocation number and the system tries to get you to the Fail port of the invocation box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts (`!`) in your program. In this case the system fails to the latest surviving Fail port before the correct one.

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

w *write goal* writes the current goal on the terminal using `write/1`.

g *print ancestor goals* provides you with a list of ancestors to the current goal, i.e. all goals that are hierarchically above the current goal in the calling sequence. It uses the `ancestors/1` built-in predicate. 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.

= *debugging* outputs information concerning the status of the debugging package.

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

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

a *abort* causes an abort of the current execution. All the execution states built so far are destroyed and you are put right back at the top level of the interpreter. (This is the same as the built-in predicate `abort/0`.)

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

4 Definite Clause Grammars

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 operation "`,`" (comma). Definite clause grammars extend context-free grammars in the following ways:

- A non-terminal symbol may be any Prolog term (other than a variable or integer).
- A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list "`[]`." If the terminal symbols are ASCII character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list ("`[]`" or "`""`").
- Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in curly brackets ("{" and "}").
- The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
- Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator "`;`" (semicolon) as in Prolog. (The disjunction operator can also be written as "`|`" (vertical-bar).)
- The cut symbol "`!`" (exclamation point) may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in curly brackets.

4.1 An Example

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value. Create a file, say "`grammar.pl`," containing the following rules:

```
expr(Z) --> term(X), "+", expr(Y), Z is X + Y.
expr(Z) --> term(X), "-", expr(Y), Z is X - Y.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), Z is X * Y.
term(Z) --> number(X), "/", term(Y), Z is X / Y.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), C is -X.
number(X) --> [C], "0"=<C, C=<"9", X is C - "0".
```

In the last rule, `C` is the ASCII code of a decimal digit.
Consult "`grammar.pl`" and this grammar can now be used to parse and evaluate an expression. For example,

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

Z = 14

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

Rest = [],
Z = 13 ?  ;

Rest = [42,53], % Rest = "*5"
Z = 1 ?  ;

Rest = [43,51,42,53], % Rest = "+3*5"
Z = -2 ?  ;

no
```

4.2 How Grammar Rules Are Translated into Prolog Clauses

In general, a grammar rule of the form

$$\mathtt{h}(\vec{V})\ \mathtt{-->}\ \mathtt{b_1}(\vec{V_1}),\ldots,\ \mathtt{b_n}(\vec{V_n}).$$

is translated into the following Prolog clause

$$\begin{array}{l}\mathtt{h}(\vec{V},\mathtt{S_0},\mathtt{S})\ \mathtt{:-}\\ \quad\mathtt{b_1}(\vec{V_1},\mathtt{S_0},\mathtt{S_1}),\\ \quad\mathtt{b_2}(\vec{V_2},\mathtt{S_1},\mathtt{S_2}),\\ \quad\ldots\\ \quad\mathtt{b_n}(\vec{V_n},\mathtt{S_{n-1}},\mathtt{S}).\end{array}$$

where $\vec{V_i}$ is a vector of variables. The translated set of Prolog clauses can then be used as a top-down recursive-decent parser.

5 Built-in Predicates

The SICStus Prolog system provides built-in predicates for a variety of processing needs, such as arithmetic and term manipulation. We briefly discuss *arithmetic*, *term comparison*, *input/output*, *control*, *meta-logical*, *program modification*, and *all solution predicates* in the following subsections. For other facilities, please refer to the user's manual for details.

5.1 Arithmetic

Arithmetic expressions are evaluated by the "`is/2`" predicate, which is in the form "`Term is Expr`." `Term` is either a variable or a number and `Expr` is an arithmetic expression which *must* not contain any uninstantiated variables at the time of invocation. When an "`is/2`" goal is encountered during execution, `Expr` is evaluated and the result is unified with `Term`. The following table lists the valid arithmetic operators that can be used to compose an arithmetic expression.

`X + Y`	Addition
`X - Y`	Subtraction
`X * Y`	Multiplication
`-X`	Unary minus
`X // Y`	Integer division
`X mod Y`	`X` (integer) modulo `Y`
`integer(X)`	The nearest integer between X and 0
`float(X)`	The floating-point equivalent of `X`
`min(X,Y)`	Minimum of `X` and `Y`
`max(X,Y)`	Maximum of `X` and `Y`

The following bit-vector operations apply to integer arguments only.

`X /\ Y`	Bitwise conjunction
`X \/ Y`	Bitwise disjunction
`\(X)`	Bitwise negation
`X << Y`	`X` is shifted left `Y` places
`X >> Y`	`X` is shifted right `Y` places

SICStus also includes an extra set of functions listed below. These may not be supported by other Prologs. The names of these functions suggest their usage. They are:

> `abs(X)`, `min(X,Y)`, `max(X,Y)`, `round(X)`, `truncate(X)`, `floor(X)`, `ceiling(X)`, `sin(X)`, `cos(X)`, `tan(X)`, `cot(X)`, `sinh(X)`, `cosh(X)`, `tanh(X)`, `coth(X)`, `asin(X)`, `acos(X)`, `atan(X)`, `acot(X)`, `asinh(X)`, `acosh(X)`, `atanh(X)`, `acoth(X)`, `sqrt(X)`, `log(X)`, `log(X,Y)` (base `X` logarithm of `Y`), `exp(X)`, `exp(X,Y)` (`X` to the power of `Y`), `inf` (infinity as defined in the IEEE standard), and `nan` (not-a-number as defined in the IEEE standard).

All trigonometric and transcendental functions take float arguments and deliver float values. The trigonometric functions take arguments or deliver values in radians.
The following predicates evaluates each of its arguments as an arithmetic expression and then compares the results. If one argument evaluates to an integer and the other to a float, the integer is coerced to a float before the comparison is made. *Note that two floating-point numbers are equal if and only if they have the same bit pattern.* Because of rounding error, it is not normally useful to compare two floats for equality.

$\texttt{Expr}_1$ `=:=` $\texttt{Expr}_2$	The values of $\texttt{Expr}_1$ and $\texttt{Expr}_2$ are equal
$\texttt{Expr}_1$ `=\=` $\texttt{Expr}_2$	The values of $\texttt{Expr}_1$ and $\texttt{Expr}_2$ are not equal
$\texttt{Expr}_1$ `<` $\texttt{Expr}_2$	The value of $\texttt{Expr}_1$ is less than that of $\texttt{Expr}_2$
$\texttt{Expr}_1$ `>` $\texttt{Expr}_2$	The value of $\texttt{Expr}_1$ is greater than that of $\texttt{Expr}_2$
$\texttt{Expr}_1$ `=<` $\texttt{Expr}_2$	The value of $\texttt{Expr}_1$ is *not* greater than that of $\texttt{Expr}_2$
$\texttt{Expr}_1$ `>=` $\texttt{Expr}_2$	The value of $\texttt{Expr}_1$ is *not* less than that of $\texttt{Expr}_2$

5.2 Term Comparison

Terms are compared according to the standard order defined as:

variables `@<` numbers `@<` atoms `@<` compound terms

Within these categories, ordering is as follows.

- Variables are put in a standard order. (Roughly, the oldest variable is put first; the order is not related to the names of variables.)
- Numbers are put in numeric order. Where a number may be represented by an integer or a floating-point number, as in 2 and 2.0, the integer is considered to be infinitesimally smaller than its floating-point counterpart.
- Atoms are put in alphabetical order according to the character set in use.
- Compound terms are ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order).
- Lists are compared as ordinary compound terms with functor `'.'/2`.

The following are term comparison predicates:

`T1 == T2`	`T1` and `T2` are literally identical (including variable names)
`T1 \== T2`	`T1` and `T2` are *not* literally identical
`T1 @< T2`	`T1` is before term `T2` in the standard order
`T1 @> T2`	`T1` is after term `T2`
`T1 @=< T2`	`T1` is not after term `T2`
`T1 @>= T2`	`T1` is not before term `T2`
`compare(Op,T1,T2)`	`Op` is the result of comparing terms `T1` and `T2`, where the possible values for `Op` are: "`=`," "`<`," and "`>`"

5.3 Input/Output

A stream to a file `F` is opened for input by the first `see(F)` executed, making `F` the current input stream. Similarly, a stream to file `E` is opened for output by the first `tell(E)` executed, making `E` the current output stream. Subsequent calls to `see(F)` or to `tell(E)` make `F` and `E` the current input or output stream respectively. Any input (output) is always to the current input (output) stream. The user terminal, viewed as the file user, is the only file which can be opened simultaneously for input and output. The following built-in predicates are for doing I/O in SICStus Prolog:

`see(F)`	File `F` becomes the current input stream.
`seeing(F)`	`F` is unified with the name of the current input file.
`seen`	Closes the current input stream.
`tell(F)`	File `F` becomes the current output stream.
`telling(F)`	`F` is unified with the name of the current output file.
`told`	Closes the current output stream.
`get0(N)`	`N` is the ASCII code of the next character from the current input stream.
`get(N)`	`N` is the ASCII code of the next non-blank printable character from the current input stream.
`skip(N)`	Skip to just past the next ASCII character code `N` from the current input stream.
`read(X)`	The next term is read from the current input stream and unified with `X`. If there are no more characters left on the current input stream, `X` will be unified with the atom `end_of_file`.
`put(N)`	ASCII character code `N` is output to the current output stream.
`write(X)`	The term `X` is written to the current output stream. The `write/1` predicate uses the operator declarations for printing.
`display(X)`	Same as `write/1` except it does not use the operator declarations for printing.
`nl`	A new line is started on the current output stream.

5.4 Control

Besides the sequencing of goals and clauses, Prolog provides at least two other very important facilities for specifying control information. They are the *cut* and the *is-not-provable* operators, written "`!`" and "`\+`" respectively.

5.4.1 Cut

The cut is inserted in the program just like a goal, but is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned. The effect of the cut is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the parent goal, 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.

5.4.2 Negation As Failure

SICStus Prolog provides an "is-not-provable" operator `\+` P. If the goal P has a solution, then `\+` P fails, otherwise succeeds. 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.

5.5 Meta-logical

Meta-logical predicates are those predicates which allow you to examine the current instantiation state of a simple or compound term, or the components of a compound term. We describe a few commonly used predicates in the following table. For a full list of available meta-logical predicates, please refer to the user's manual.

`var(X)`	`X` is a currently uninstantiated variable.
`nonvar(X)`	`X` is a non-variable.
`integer(X)`	`X` is an integer.
`atom(X)`	`X` is an atom.
`float(X)`	`X` is a floating-point number.
`number(X)`	`X` is an integer or floating-point number.
`atomic(X)`	`X` is an atom or a number.
`simple(X)`	`X` is an atom, number, or variable.
`compound(X)`	`X` is a compound term (arity > 0).
`callable(X)`	`X` is a term that call/1 would take as an argument; atom or compound term.
`ground(X)`	`X` contains no uninstantiated variables.
`functor(Term,Symbol,Arity)`	`Symbol` is the principal function symbol of the term `Term` and `Arity` is the arity (number of arguments). This may be used to find the functor of a given term, or to construct a term having a given functor.
`arg(Index,Term,Arg)`	`Arg` is the `Index`th argument of `Term`. `Term` must not be a variable, but any other kind of term is acceptable. `Index` must be instantiated to an integer between 1 and the number of arguments of `Term` inclusively; otherwise, `arg/3` quietly fails.
`Term =..[Atom|Args]`	`Symbol` is the principal function symbol of the term `Term` and `Args` is a list of the remaining arguments of `Term`. This may be used to compose or decompose a term.
`name(Atom,Chars)`	`Chars` is a list of characters that comprises the symbol `Atom`.
`call(X)`	If `X` is instantiated to an atom or compound term, then the goal `call(X)` is executed exactly as if that term appeared textually in its place, except that any `cut (!)` occurring in `X` only cuts alternatives in the execution of `X`.

5.6 Program Modification

An interesting question in Prolog is what happens if a procedure modifies itself, by asserting or retracting a clause, and then fails. On backtracking, does the current execution of the procedure use new clauses which are added to the bottom of the procedure?

SICStus Prolog uses the *logical* view in updating dynamic predicates. This means that the definition of an interpreted procedure that is to be visible to a call is effectively frozen when the call is made. A procedure always contains, as far as a call to it is concerned, exactly the clauses it contained when the call was made. A useful way to think of this is to consider that a call to a dynamic procedure makes a virtual copy of the procedure and then runs the copy rather than the original procedure. Any changes to the procedure made by the call are immediately reflected in the Prolog database, but not in the copy of the procedure being run. Thus, changes to a running procedure will not be visible on backtracking. A subsequent call, however, makes and runs a copy of the modified Prolog database. Any changes to the procedure that were made by an earlier call will now be visible to the new call.

Only predicates defined to be *dynamic* can be modified using the predicates during execution. The following predicates are used to add and delete clauses to and from the database in a various ways.

`assert(C)`	clause `C` is asserted in an arbitrary position in its predicate
`asserta(C)`	clause `C` is asserted before existing clauses
`assertz(C)`	clause `C` is asserted after existing clauses
`retract(C)`	erase the first dynamic clause that matches `C`
`retractall(H)`	erase every clause whose head matches `H`
`abolish(F)`	abolish the predicate(s) specified by `F`
`abolish(F,N)`	abolish the predicate named F of arity `N`
`clause(P,Q)`	there is a clause for a dynamic predicate with head `P` and body `Q`

5.7 All Solution Predicates

When there are many solutions to a goal, and a list of all those solutions is desired, the following built-in predicate are useful:

`setof(Template,Generator,Set)`	`Set` is the set of all instances of `Template` such that `Generator` is provable, where that set is non-empty. `Set` is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms.
`bagof(Template,Generator,Set)`	Exactly the same as `setof/3` except that the list returned is not ordered and may contain duplicates.
`findall(Template,Generator,Set)`	A special case of `bagof/3`, where all free variables are taken to be existentially quantified.

If there are no solutions, `setof/3` and `bagof/3` fail but `findall/3` succeeds with `Set=[]`.