82/11

C-Prolog User's Manual
Version 1.2a

ed Fernando Pereira March 1983

EdCAAD
Edinburgh Computer Aided Architectural Design

UNIVERSITY OF EDINBURGH
DEPARTMENT OF ARCHITECTURE
22 Chambers Street
Edinburgh EH1 1GZ
Tel: 031 667 1011 ext. 4598

C-Prolog User's Manual
Version 1.2a

Edited by Fernando Pereira[*]

SRI International, Menlo Park, California

from material by

David Warren

SRI International, Menlo Park, California

David Bowen, Laurence Byrd

Dept. of Artificial Intelligence, University of Edinburgh

Luis Pereira

Dept. de Informatica, Universidade Nova de Lisboa

ABSTRACT

This is a preliminary edition of the user's manual for C-Prolog, a Prolog interpreter written in C for 32 bit machines. C-Prolog is based on an earlier Prolog interpreter written in IMP for the EMAS operating system by Luis Damas, who borrowed many aspects of the design from the DEC-10/20 Prolog system developed by David Warren, Fernando Pereira, Lawrence Byrd and Luis Pereira. This manual is based on the EMAS Prolog manual, which in turn was based on the DEC-10/20 Prolog manual.

* Formerly at EdCAAD, Dept. of Architecture, University of Edinburgh

1. Using C-Prolog

1.1. Preface

This manual describes C-Prolog, a Prolog interpreter written in C. C-Prolog was developed at EdCAAD, Dept. of Architecture, University of Edinburgh, and is based on a previous interpreter, written in IMP for the EMAS operating system by Luis Damas of the Dept. of Computer Science, University of Edinburgh. C-Prolog was designed for machines with a large, uniform, address space, and assumes a pointer cell 32 bits wide. At the time of writing, it has been tested on VAXes under the Unix† and VAX/VMS operating systems, and has been ported with minor changes to MC68000-based workstations and to the Three Rivers PERQ.

Prolog is a simple but powerful programming language originally developed at the University of Marseilles, as a practical tool for programming in logic. From a user's point of view the major attraction of the language is ease of programming. Clear, readable, concise programs can be written quickly with few errors. Prolog is especially suitable for high-level symbolic programming tasks and has been applied in many areas of Artificial Intelligence research.

The system consists of a Prolog interpreter and a wide range of evaluable predicates (system provided procedures). Its design was based on the (Edinburgh) DEC10 Prolog system and the system is closely compatible with DEC10 Prolog and thus is also reasonably close to PDP-11 Unix and RT-11 Prolog.

This manual is not intended as an introduction to the Prolog language and how to use it. For this purpose you should study:

Programming in Prolog
W. Clocksin & C. Mellish
Springer Verlag 1981

This manual assumes that you are familiar with the principles of the Prolog language, its purpose being to explain how to use C-Prolog, and to describe all the evaluable predicates provided by C-Prolog.

1.2. Using C-Prolog - Overview

C-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 number of files using a text editor. C-Prolog can then be instructed to read-in programs from these files; this is called "consulting" the file. To change parts of a program being run, it is possible to "reconsult" files containing the changed parts. Reconsulting means that definitions for procedures in the file will replace any old definitions for these procedures.

It is recommended that you make use of a number of different files when writing programs. Since you will be editing and consulting/re-consulting individual files it is useful to use files to group together related procedures; keeping collections of procedures that do different things in different files. Thus a Prolog program will consist of a number of files, each file containing a number of related procedures.

† Unix is a Trademark of Bell Laboratories.

When your programs start to grow to a fair size, it is also a good idea to have one file which just contains commands to the interpreter to consult all the other files which form a program. You will then be able to consult your entire program by just consulting this single file.

1.3. Access to C-Prolog

In this manual, we assume that there is a command on your computer

    prolog

that invokes C-Prolog.

Since Prolog makes syntactic use of the difference between upper and lower case it is important that you have your terminal set up so that it accepts lower case in the normal way. This means, for a start, that you must be using an upper and lower case terminal - and not, for example, an upper case only teletype. It is possible to use Prolog using upper case only (see Section 1.14) but it is unnecessarily painful. We shall assume both upper and lower case throughout this manual.

If you type the 'prolog' command, Prolog will output a banner and prompt you for directives as follows:

```
C-Prolog              version n
| ?-
```

There will be a pause between the first line and the prompt while the system loads itself. It is possible to type ahead during this period if you get impatient.

If you give an argument to the 'prolog' command, C-Prolog will interpret it as the name of a file containing a saved state created earlier, and will restore that saved state. Saved states will be explained fully later.

    prolog prog   (Restore "prog")

1.4. Reading-in Programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the interpreter. The clauses of a procedure do not have to be immediately consecutive, but remember that their relative order may be important.

To input a program from a file *file*, give the directive:

    | ?- [*file*].

which will instruct the interpreter to read-in (or *consult*) the program. The file specification *file* must be a Prolog atom. It may be any file name, note that if this file name contains characters which are not normally allowed in an atom then it is necessary to surround the whole file specification with single quotes (since quoted atoms can include any character), for example

    | ?- ['people/greeks'].

The specified file is then read in. Clauses in the file are stored in the database ready to be executed, 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 in reading-in and the number of bytes occupied by the program.

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

```
| ?- [myprogram,extras,testbits].
```

In this case all three files would be consulted. If a filename is preceded by a minus sign, as in:

```
| ?- [-testbits,-moreideas].
```

then that file is reconsulted. The difference between consulting and reconsulting is important, and works as follows: if a file is consulted then all the clauses in the file are simply added to C-Prolog's database. If you consult the same file twice then you will get two copies of all the clauses. However, if a file is reconsulted then the clauses for all the procedures in the file will replace any existing clauses for those procedures, that is any such previously existing clauses in the database get thrown away. Reconsulting is useful for telling Prolog about corrections in your program.

Clauses may also be typed in directly at the terminal. To enter clauses at the terminal, you must give the directive:

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

The interpreter is now in a state where it expects input of clauses or directives. To get back to the top level of the interpreter, type the end-of-file* character.

Typing clauses directly into C-Prolog is only recommended if the clauses will not be needed permanently, and are few in number. For significant bits of program you should use an editor to produce a file containing the text of the program.

1.5. Directives: Questions and Commands

When Prolog is at top level (signified by an initial prompt of "| ?- ", with continuation lines prompted with "| ", that is indented out from the left margin) it reads in terms and treats them as directives to the interpreter to try and satisfy some goals. These directives are called questions. Remember that Prolog terms must terminate with a full-stop ("."), and that therefore Prolog will not execute anything for you until you have typed the full-stop (and then <return>) at the end of the directive.

Suppose list membership has been defined by:

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

(Note the use of anonymous variables written "_").

If the goal(s) specified in a question can be satisfied, and if there are no variables as in this example:

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

then the system answers

```
yes
```

and execution of the question terminates.

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

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

would be answered by

* Control-D, control-Z or some other, depending on your tastes and the operating system.

```
X = a
```

At this point the interpreter is waiting for you to indicate whether that solution is sufficient, or whether you want it to backtrack to see if there are any more solutions. Simply typing <return> terminates the question, while typing ";" followed by <return> causes the system to backtrack looking for alternative solutions. If no further solutions can be found it outputs

```
no
```

The outcome of some questions 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]).
Y = b,
X = f(b,c),
Z = c
                        % Just <return> typed here
yes
| ?- member(X,[f(_),g]).
X = f(_1728)
yes
| ?-
```

When C-Prolog reads terms from a file (or from the terminal following a call to [user]), it treats them all as program clauses. In order to get the interpreter to execute directives from a file they must be preceded by '?-', for questions, or ':-', for commands.

Commands are like questions except that they do not cause answers to be printed out. They always 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; for example, the command

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

directs the system to check whether 3 belongs to the list [1,2,3], and to output "ok" if so. Execution of a command terminates when all the goals in the command have been successfully executed. Other alternative solutions are not sought (one may imagine an implicit "cut" at the end of the command). If no solution can be found, the system gives:

```
?
```

as a warning.

The main use for commands (as opposed to questions) is to allow files to contain directives which call various procedures, but for which you don't want to have the answers printed out. In such cases you only want to call the procedures for effect. 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, mainpart, testcases, data, junk ]).
```

† The extra parentheses, with the ':-' immediately next to them, are currently essential due to a problem with prefix operators (like ':-') and lists. They are not required for commands

If this directive was contained in the file 'program' then typing the following at top level would be a quick way of loading your entire program:

```
| ?- [program].
```

When you are simply interacting with the top level of the Prolog interpreter the distinction between questions and commands is not very important. At the top level you should normally only type questions. In a file, if you wish to execute some goals then you should use a command. That is, to execute a directive in a file it must be preceded by ":-", otherwise it will be treated as a clause.

1.6. Syntax Errors

Syntax errors are detected when reading. Each clause, directive or in general any term read-in by the built-in procedure **read** that fails to comply with syntax requirements is displayed on the terminal as soon as it is read. A mark indicates the point in the string of symbols where the parser has failed to continue its analysis. For example, typing

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

gives:

```
***syntax error***
member(X,X
***here***
 L).
```

Syntax errors do not disrupt the (re)consulting of a file in any way except that the clause or command with the syntax error will be ignored†. All the other clauses in the file will have been read-in properly. If the syntax error occurs at top level then you should just retype the question. Given that Prolog has a very simple syntax it is usually quite straightforward to see what the problems is (look for missing brackets in particular). See Section 1.13 for details of the syntax of Prolog terms. The book "Programming in Prolog" gives further examples.

1.7. Saving A Program

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 a file for future execution. To save a program into a file *file*, perform the command:

?- save(*file*).

Save can be called at top level, from within a break level (q.v.), or from anywhere within a program.

1.8. Restoring A Saved Program

Once a program has been saved into a file *file*, C-Prolog can be restored to this saved state by invoking it as follows:

prolog *file*

After execution of this command, the interpreter will be in EXACTLY the same

---

that do not contain lists. This restriction will be eventually removed.

† After all, it could not be read.

state as existed immediately prior to the call to **save**, except for open files, which are automatically closed by **save**. That is to say execution will start at the goal immediately following the call to **save**, just as if **save** had returned successfully. If you saved the state at top level then you will be back at top level, but if you explicitly called **save** from within your program then the execution of your program will continue.

Saved states can only be restored when C-Prolog is initially run from command level. Version 1.2a provides no way of restoring a saved state from inside C-Prolog.

Note that when a new version of C-Prolog is installed, saved states created with the old version may become unusable. You are thus advised to rely on source files for your programs and not on some gigantic saved state.

1.9. Program Execution And Interruption

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

Only when execution of one directive is complete does the interpreter become ready for another directive. However, one may interrupt the normal execution of a directive by hitting the interrupt key on your terminal. In response to the prompt

    Action (h for help):

you can type either "a", "d" or "c" followed by <return>. The "a" response will force Prolog to abort back to top level, the "d" response will switch on debugging and continue the execution, and the "c" response will just continue the execution.

1.10. Nested Executions - Break and Abort

C-Prolog 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. When the evaluable predicate **break** is called, the message

    [Break (level 1)]

will be displayed. This signals the start of a *break-level* and except for the effect of **aborts** (see below), it is as if the interpreter was at top level. If break is called within a break-level, then another recursive break-level is started (and the message will say (level 2) etc). Break-levels may be arbitrarily nested.

Typing the end-of-file character will close the break-level and resume the execution which was suspended, starting at the procedure call where the suspension took place.

To abort the current execution,forcing an immediate failure of the directive being executed and a return to the top level of the interpreter, call the evaluable predicate **abort**, either from the program or by executing the directive:

    | ?- abort.

within a break. In this case no end-of-file character is needed to close the break, because ALL break levels are discarded and the system returns right back to top level. The "a" interrupt (described above) can also be used to force an abort.

1.11. Exiting From The Interpreter

To exit from C-Prolog interpreter you should give the directive:

```
| ?- halt.
```

This can be issued either at top level, or within a break-level, or indeed from within your program.

If your program is still executing then you should interrupt it and abort to return to top level so that you can call **halt**.

Typing the end-of-file charater at top level also causes C-Prolog to terminate.

1.12. Debugging facilities

The debugging facilities in version 1.2a of C-Prolog are still under development. The predicates described in Section 2.11 are all available but their behaviour is somewhat unsatisfactory. When better facilities become available this section will be replaced by a supplement which will provide a proper description.

1.13. Prolog Syntax

1.13.1. 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 *numbers* such as

```
0  -999  -5.23  0.23e-5
```

Constants also include *atoms* such as

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

The symbol for an atom can be any sequence of characters, written in single quotes if there is possibility of confusion with other symbols (such as variables or numbers). As in other programming languages, constants are definite elementary objects.

Variables are distinguished by an initial capital letter or by the initial character "_", for example

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

If a variable is only referred to once, it does not need to be named and may be written as an "anonymous" variable, indicated by the underline character "_".

A variable should be thought of as standing for some definite but unidentified object. 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 constant declarations in Pascal.

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

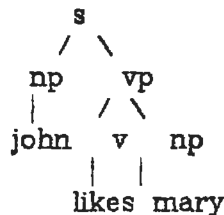
```
point(X,Y,Z)
```

An atom is considered to be a functor of arity 0.

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

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

as optional alternatives to

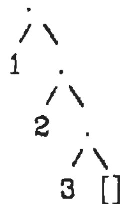
    +(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)

Operators are described fully in the next section.

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,[])))

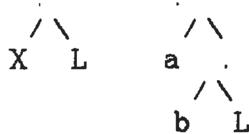
but which is normally written, in a special list notation, as

    [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 list syntax is only syntactic sugar for terms of the form '.'(_,_) and does not provide any additional facilities that were not available in Prolog.

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

"Prolog"

represents exactly the same list as

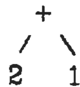
[80,114,111,108,111,103]

1.13.2. Operators

Operators in Prolog are simply a notational convenience. For example, the expression

2 + 1

could also be written +(2,1). It should be noticed that this expression represents the data structure



and not the number 3. The addition would only be performed if the structure was passed as an argument to an appropriate procedure (such as **is** - see Section 2.2).

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

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

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

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

(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

With an operator of type 'xfx', it is a requirement that both of the two subexpressions which are the arguments of the operator must be of LOWER precedence than the operator itself, i.e. their principal functors must be of lower precedence, unless the subexpression is explicitly bracketed (which gives it zero precedence). With an operator of type 'xfy', only the first or left-hand subexpression must be of lower precedence; the second can be of the SAME precedence as the main operator; and vice versa for an operator of type 'yfx'.

For example, if the operators '+' and '-' both have type 'yfx' and are of the same precedence, then the expression

    a-b+c

is valid, and means

    (a-b)+c   i.e. +(-(a,b),c)

Note that the expression would be invalid if the operators had type 'xfx', and would mean

    a-(b+c)   i.e. -(a,+(b,c))

if the types were both 'xfy'.

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)".

In C-Prolog, a functor named *name* is declared as an operator of type *type* and precedence *precedence* by calling the evaluable predicate **op**:

    | ?- 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* in C-Prolog, namely:

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).        /* See note below */
:- op(  900,  fy, [ not, \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                    =:=, =\=, <, >, =<, >= ]).
:- op(  500, yfx, [ +, -, /\, \/ ]).
:- op(  500,  fx, [ +, - ]).
:- op(  400, yfx, [ *, /, //, <<, >> ]).
:- op(  300, xfx, [ mod ]).
:- op(  200, xfy, [ ^ ]).
```

Operator declarations are most usefuly placed in directives at the top of your program files. In this case the directive should be a command as shown above. Another common method of organisation is to have one file just containing commands to declare all the necessary operators. This file is then always consulted first.

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':

```
X,Y    ','(X,Y)
```

represent the same compound term. But note that a comma written as a quoted atom is NOT a standard operator.

Note also that the arguments of a compound term written in standard syntax must be expressions of precedence BELOW 1000. Thus it is necessary to bracket the expression "P:-Q" in

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

The following syntax restrictions serve to remove potential ambiguity associated with prefix operators.

a) In a term written in standard syntax, the principal functor and its following "(" must NOT be separated by any intervening spaces, newlines etc. Thus

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

is invalid syntax.

b) If the argument of a prefix operator starts with a "(", this "(" must be separated from the operator by at least one space or other non-printable character. Thus

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

(where ':-' is the prefix operator) is invalid syntax, and must be written as

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

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

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

1.14. Using a Terminal without Lower-Case

The syntax of Prolog assumes that a full ASCII character set is available. With this "full character set" or 'LC' convention, variables are (normally) distinguished by an initial capital letter, while atoms and other functors must start with a lower-case letter (unless enclosed in single quotes).

When lower-case is not available, the "no lower-case" or 'NOLC' convention has to be adopted. With this convention, variables must be distinguished by an initial underline character "_", and the names of atoms and other functors, which now have to be written in upper-case, are implicitly translated into lower-case (unless enclosed in single quotes). For example:

    _VALUE2

is a variable, while

    VALUE2

is 'NOLC' convention notation for the atom which is identical to:

    value2

written in the 'LC' convention.

The default convention is 'LC'. To switch to the "no lower-case" convention, call the built-in procedure 'NOLC', e.g. by the directive:

    | ?- 'NOLC'.

To switch back to the "full character set" convention, call the built-in procedure 'LC', e.g. by:

    | ?- 'LC'.

Note that the names of these two procedures consist of upper-case letters (so that they can be referred to on all devices), and therefore the names must ALWAYS be enclosed in single quotes.

It is recommended that the 'NOLC' convention only be used in emergencies, since the standard syntax is far easier to use and is also easier for other people to read.

2. Built-in Procedures

Built-in procedures are also referred to as *evaluable predicates*.

This section describes all the built-in predicates available in C-Prolog. These predicates are provided in advance by the system and they cannot be redefined by the user. If you try to add clauses for a built-in predicate (with the exception of **expand_term**) then this will cause an error message, and the built-in predicate will be unaffected. The C-Prolog provides a fairly wide range of built-in predicates to perform the following tasks:

Input/Output
Reading-in programs
Opening and closing files
Reading and writing Prolog terms
Getting and putting characters
Arithmetic
Affecting the flow of the execution
Classifying and operating on Prolog terms (meta-logical facilities)
Sets
Term Comparison
Manipulating the Prolog program database
Manipulating the additional indexed database
Debugging facilities
Environmental facilities

The the built-in predicates will be described according to this classification. Appendix I contains a complete list of the built-in predicates.

2.1. Input / Output

A total of 15 I/O streams may be open at any one time for input and output. This includes a stream that is always available for input and output to the user's terminal. A stream to a file *F* is opened for input by the first **see**(*F*) executed. *F* then becomes the current input stream. Similarly, a stream to file *H* is opened for output by the first **tell**(*H*) executed. *H* then becomes the current output stream. Subsequent calls to **see**(*F*) or to **tell**(*H*) make *F* or *H* the current input or output stream, respectively. Any input or output is always to the current stream.

When no input or output stream has been specified, the standard ersatz file 'user', denoting the user's terminal, is utilised for both. When the terminal is waiting for input on a new line, a prompt will be displayed as follows:

"| " top level continuation line.
"| " during consult(user).
"|: " default for read from user program.

When the current input (or output) stream is closed, the user's terminal becomes the current input (or output) stream.

The only file that can be simultaneously open for input and output is the ersatz file 'user'.

A file is referred to by its name, *written as an atom*, e.g.

myfile
'F123'
data_lst
'tom/greeks'

All I/O errors normally cause an **abort**, except for the effect of the evaluable predicate **nofileerrors** decribed below.

End of file is signalled on the user's terminal by issuing the end-of-file character. Any more input requests for a file whose end has been reached causes an error failure.

2.1.1. Reading in Programs

consult(*F*)

Instructs the interpreter to read-in the program which is in file *F*. When a directive is read it is immediately executed. When a clause is read it is put after any clauses already read by the interpreter for that procedure.

reconsult(*F*)

Like **consult** except that any procedure defined in the "reconsulted" file erases any clauses for that procedure already present in the interpreter. **reconsult** makes it possible to amend a program without having to restart from scratch and consult all the files which make up the program.

[*File*|*Files*]

This is a shorthand way of consulting or reconsulting a list of files. A file name may optionally be preceded by the operator '-' to indicate that the file should be "reconsulted" rather than "consulted". Thus

```
| ?- [file1,-file2,file3].
```

is merely a shorthand for

```
| ?- consult(file1), reconsult(file2), consult(file3).
```

2.1.2. File Handling

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.

close(*F*)

File *F*, currently open for input or output, is closed.

fileerrors

Undoes the effect of **nofileerrors**.

nofileerrors

After a call to this predicate, the I/O error conditions "incorrect file name ...", "can't see file ...", "can't tell file ..." and "end of file ..." cause a call to **fail** instead of the default action, which is to type an error message and then call **abort**.

exists(*F*)

Succeeds if the file F exists.

rename(*F*,*N*)

If File *F* is renamed to *N*. If *N* is '[]', the file is deleted. If *F* was a currently open stream, it is closed first.

2.1.3. Input and Output of Terms

read(*X*)

The next term, delimited by a full stop (i.e. a "." followed by a carriage-return or a space), is read from the current input stream and unified with *X*. The syntax of the term must accord with current operator declarations. If a call **read**(*X*) causes the end of the current input stream to be reached, *X* is unified with the atom 'end_of_file'. Further calls to **read** for the same stream will then cause an error failure.

write(*X*)

The term *X* is written to the current output stream according to operator declarations in force.

display(*X*)

The term *X* is displayed on the terminal in standard parenthesised prefix notation.

writeq(*Term*)

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

print(*Term*)

Print *Term* onto the current output. This predicate provides a handle for user defined pretty printing. If *Term* is a variable then it is written, using **write**(*Term*). If *Term* is non-variable then a call is made to the user defined procedure **portray**(*Term*). If this succeeds then it is assumed that *Term* has been output. Otherwise **print** is equivalent to **write**.

2.1.4. Character Input/Output

nl

A new line is started on the current output stream.

get0(*N*)

N is the ASCII code of the next character from the current input stream.

get(*N*)

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

skip(*N*)

Skips to just past the next ASCII character code *N* from the current input stream. *N* may be an integer expression.

put(*N*)

ASCII character code *N* is output to the current output stream. *N* may be an integer expression.

tab(*N*)

N spaces are output to the current output stream. *N* may be an integer expression.

2.2. Arithmetic

Arithmetic is performed by built-in procedures which take as arguments *arithmetic expressions* and *evaluate* them. An arithmetic expression is a term built from *evaluable functors*, numbers and variables. At the time of evaluation, each variable in an arithmetic expression must be bound to a number or to an arithmetic expression. The result of evaluation will always be converted back to an integer if possible.

Each evaluable functor stands for an arithmetic operation. The adjective "integer" in the descriptions below means that the operation only makes sense for integers, and will fail for floating point numbers.

The evaluable functors are as follows, where *X* and *Y* are arithmetic expressions.

X+*Y*
: addition

X-*Y*
: subtraction

*X***Y*
: multiplication

X/*Y*
: division

X//*Y*
: integer division

X mod *Y*
: *X* (integer) modulo *Y*

-*X*
: unary minus

exp(*X*)
: exponential function

log(*X*)
: natural logarithm

log10(*X*)
: base 10 logarithm

sqrt(*X*)
: square root

sin(*X*)
: sine

cos(*X*)
: cosine

tan(*X*)
: tangent

asin(*X*)
: arc sine

acos(*X*)

arc cosine

atan(*X*)

arc tangent

floor(*X*)

the largest integer not greater than *X*

X^*Y*

X to the power *Y*

X/*Y*

integer bitwise conjunction

X/ *Y*

integer bitwise disjunction

X<<*Y*

integer bitwise left shift of *X* by *Y* places

X>>*Y*

integer bitwise right shift of *X* by *Y* places

X

integer bitwise negation

cputime

CPU time since C-Prolog was started, in seconds.

heapused

Heap space in use, in bytes.

[*X*]

(a list of just one element) evaluates to *X* if *X* is an integer. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its ASCII code; e.g. "A" behaves within arithmetic expressions as the integer 65.

The arithmetic built-in procedures are as follows, where *X* and *Y* stand for arithmetic expressions, and *Z* for some term. Note that this means that **is** only evaluates one of its arguments as an arithmetic expression (the right-hand side one), whereas all the comparison predicates evaluate both their arguments.

Z **is** *X*

Arithmetic expression *X* is evaluated and the result, is unified with *Z*. Fails if *X* is not an arithmetic expression.

X =:= *Y*

The values of *X* and *Y* are equal.

X == *Y*

The values of *X* and *Y* are not equal.

X < *Y*

The value of *X* is less than the value of *Y*.

X > *Y*

The value of *X* is greater than the value of *Y*.

X =< *Y*

The value of X is less than or equal to the value of Y.

$X >= Y$

The value of X is greater than or equal to the value of Y.

2.3. Convenience

P , *Q*

P and *Q*.

P ; *Q*

P or *Q*.

true

Always succeeds.

$X = Y$

Defined as if by the clause " Z=Z. ", that is X and Y are unified.

2.4. Extra Control

!

Cut (discard) all choice points made since the parent goal started execution.

\+ *P*

If the goal *P* has a solution, fail, otherwise succeed. It is defined as if by

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

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.
```

P -> *Q*

When occurring other than as one of the alternatives of a disjunction, is equivalent to

P -> *Q*; fail.

repeat

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

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

fail

Always fails.

2.5. Meta-Logical

var(X)

Tests whether X is currently instantiated to a variable.

nonvar(X)

Tests whether X is currently instantiated to a non-variable term.

atom(X)

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

number(X)

Checks that X is currently instantiated to a number.

integer(X)

Checks that X is currently instantiated to an integer.

atomic(X)

Checks that X is currently instantiated to an atom, number or database reference.

primitive(X)

Checks that X is currently instantiated to a number or database reference.

db_reference(X)

Checks that X is currently instantiated to a database reference.

functor(T,F,N)

The principal functor of term T has name F and arity N, where F is either an atom or, provided N is 0, a number. Initially, either T must be instantiated to a non-variable, or F and N must be instantiated to, respectively, either an atom and a non-negative integer or an integer and 0. If these conditions are not satisfied, an error message is given. In the case where T is initially instantiated to a variable, the result of the call is to instantiate T to the most general term having the principal functor indicated.

arg(I,T,X)

Initially, I must be instantiated to a positive integer and T to a compound term. The result of the call is to unify X with the Ith argument of term T. (The arguments are numbered from 1 upwards.) If the initial conditions are not satisfied or I is out of range, the call merely fails.

X =.. Y

Y is a list whose head is the atom corresponding to the principal functor of X and whose tail is the argument list of that functor in X. E.g.

```
product(0,N,N-1) =.. [product,0,N,N-1]

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

product =.. [product]
```

If X is instantiated to a variable, then Y 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.

name(X,L)

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

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

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

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

name(hello,[104,101,108,108,111])

name([],"[]")

If *X* is instantiated to a variable, *L* must be instantiated to a list of ASCII character codes. E.g.

```
| ?- name(X,[104,101,108,108,111]).

X = hello

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

X = hello
```

call(*X*)

If *X* is instantiated to a term which would be acceptable as body of a clause, the goal **call**(*X*) is executed exactly as if that term appeared textually in place of the **call**(*X*), except that any cut ("!") occurring in *X* will remove only those choice points in *X*. If *X* is not instantiated as described above, an error message is printed and **call** fails.

X

(where *X* is a variable) Exactly the same as **call**(*X*).

2.6. Sets

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

setof(*X*,*P*,*S*)

Read this as "*S* is the set of all instances of *X* such that *P* is provable, where that set is non-empty". The term *P* specifies a goal or goals as in **call**(*P*). *S* is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 4.3). If there are no instances of *X* such that *P* is satisfied then the predicate fails.

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

If there are uninstantiated variables in *P* which do not also appear in *X*, then a call to this evaluable predicate may backtrack, generating alternative values for *S* corresponding to different instantiations of the free variables of *P*. (It is to cater for such usage that the set S is constrained to be non-empty.) For example, the call:

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

might produce two alternative solutions via backtracking:

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

The call:

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

would then produce:

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

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

Y^Q

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

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

would produce the single result:

X = [bill, dick, harry, jan, tom]

in contrast to the earlier example.

bagof(*X,P,Bag*)

This is exactly the same as **setof** except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. The effect of this relaxation is to save considerable time and space in execution.

X^P

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

2.7. Comparison of Terms

These evaluable 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 (Section 4.2) 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);
- Database references, roughly in order of age;
- numbers, from -"infinity" to +"infinity";
- atoms, in alphabetical (i.e. ASCII) order;
- complex terms, ordered first by arity, then by the name of principal functor, then by the arguments (in left-to-right order).

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

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

These are the basic predicates for comparison of arbitrary terms:

X == Y

Tests if the terms currently instantiating *X* and *Y* are literally identical (in particular, variables in equivalent positions in the two terms must be identical). For example, the question

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

fails (answers "no") because *X* and *Y* are distinct uninstantiated variables. However, the question

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

succeeds because the first goal unifies the two variables (see page 42).

X \== Y

Tests if the terms currently instantiating *X* and *Y* are not literally identical.

T1 @< *T2*

Term *T1* is before term *T2* in the standard order.

T1 @> *T2*

Term *T1* is after term *T2* in the standard order.

T1 @=< *T2*

Term *T1* is not after term *T2* in the standard order.

T1 @>= *T2*

Term *T1* is not before term *T2* in the standard order.

Some further predicates involving comparison of terms are:

compare(*Op*, *T1*, *T2*)

The result of comparing terms *T1* and *T2* is *Op*, where the possible values for *Op* are:

'=' if *T1* is identical to *T2*,

'<' if *T1* is before *T2* in the standard order,

'>' if *T1* is after *T2* in the standard order.

Thus **compare**(=, *T1*, *T2*) is equivalent to *T1* == *T2*.

sort(*L1*, *L2*)

The elements of the list *L1* are sorted into the standard order, and any identical (i.e. '==') elements are merged, yielding the list *L2*. (The time taken to do this is at worst order (N log N) where N is the length of *L1*.)

keysort(*L1*, *L2*)

The list *L1* must consist of items of the form *Key-Value*. These items are sorted into order according to the value of *Key*, yielding the list *L2*. No merging takes place. (The time taken to do this is at worst order (N log N) where N is the length of L1.)

2.8. Modification of the Program

The predicates defined in this section allow modification of the program as it is actually running. Clauses can be added to the program ("asserted") or removed from the program ("retracted"). Some of the predicates make use of an implementation-defined identifier or *database reference* which uniquely

identifies every clause in the interpreted program. This identifier makes it possible to access clauses directly, instead of requiring a search through the program every time. However these facilities are intended for more complex use of the database and are not required (and undoubtedly should be avoided) by novice users.

assert(*C*)

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

assert(*Clause*,*Ref*)

Equivalent to **assert(_)** where *Ref* is the database reference of the clause asserted.

asserta(*C*)

Like **assert(_)**, except that the new clause becomes the **first** clause for the procedure concerned.

asserta(*Clause*,*Ref*)

Equivalent to **asserta(_)** where *Ref* is the database reference of the clause asserted.

assertz(*C*)

Like **assert(_)**, except that the new clause becomes the **last** clause for the procedure concerned.

assertz(*Clause*,*Ref*)

Equivalent to **assertz(_)** where *Ref* is the database reference of the clause asserted.

clause(*P*,*Q*)

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

clause(*Head*,*Body*,*Ref*)

Equivalent to **clause(_)** where *Ref* is the database reference of the clause concerned. If *Ref* is not given at the time of the call, *Head* must be instantiated to a non-variable term. Thus this predicate can have two different modes of use, depending on whether the database reference of the clause is known or unknown.

retract(*C*)

The first clause in the program that matches *C* is erased. *C* must be initially instantiated to a non-variable. The predicate may be used in a nondeterminate fashion, i.e. it will successively retract clauses matching the argument through backtracking.

abolish(*N*,*A*)

Completely remove all clauses for the procedure with name *N* (which should be an atom), and arity *A* (which should be an integer).

The space occupied by retracted or abolished clauses will be recovered when instances of the clause are no longer in use.

See also **erase** (Section 2.10) which allows a clause to be directly erased via its database reference†.

---

† This is a lower level facility, required only for complicated database manipulations.

2.9. Information about the State of the Program

listing

Lists in the current output stream all the clauses in the program.

listing(*A*)

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

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

current_atom(*Atom*)

Generates (through backtracking) all currently known atoms, and returns each one as *Atom*.

current_functor(*Name,Functor*)

Generates (through backtracking) all currently known functors, and for each one returns its name and most general term as *Name* and *Functor* respectively. If *Name* is given, only functors with that name are generated.

current_predicate(*Name,Functor*)

Similar to **current_functor**, but it only generates functors corresponding to predicates for which there exists a procedure.

2.10. Internal Database

This section describes predicates for manipulating an internal indexed database that is kept separate from the normal program database. They are intended for more sophisticated database applications and are not really necessary for novice users. For normal tasks you should be able to program quite satisfactorily just using **assert** and **retract**.

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 database reference of the recorded item. The key must be given, and may be an atom or complex term. If it is a complex term, only the principal functor is significant.

recorda(*Key,Term,Ref*)

The term *Term* is recorded in the internal database as the first item for the key *Key*, where *Ref* is its database reference. The key must be given, and only its principal functor is significant.

recordz(*Key,Term,Ref*)

The term *Term* is recorded in the internal database as the last item for the key *Key*, where *Ref* is its database reference. The key must be given, and only its principal functor is significant.

erase(*Ref*)

The recorded item *or* clause whose database reference is *Ref* is effectively erased from the internal database or program. An erased item will no longer be accessible through the predicates that search through the database, but will still be accessible through its database reference, if this is available in the execution state after the call to **erase**. Only when all instances of the item's database reference have been forgotten through

database erasures and/or backtracking will the item be actually removed from the database.

erased(*R*)

Suceeds if *R* is a database reference to a database item that has been **erased**, otherwise fails.

instance(*Ref*, *Term*)

A (most general) instance of the recorded term whose database reference is *Ref* is unified with *Term*. *Ref* must be instantiated to a database reference. Note that **instance** will even pick database items that have been **erased**.

2.11. Debugging

The debugging package in version 1.2a of C-Prolog is preliminary and likely to be improved. The appearance of the debugging aids is thus likely to change; however, the predicates described here will not change - rather they will gradually be made more effective.

debug

Debug mode is switched on. Information will now be retained for debugging purposes and executions will require more space.

nodebug

Debug mode is switched off. Information is no longer retained for debugging, and spy points are removed.

trace

Debug mode is switched on, and the interpreter starts tracing from the next call to a user goal. If **trace** was given in a command on its own, the goal(s) traced will be those of the next command. Since this is a once-off decision, a call to trace is necessary whenever tracing is required right from the start of an execution, otherwise tracing will only happen at spy points.

spy *Spec*

Spy-points will be placed on all the procedures given by *Spec*. All control flow through the ports of these procedures will henceforth be traced. If debug mode was previously off, then it will be switched on. *Spec* can either be a predicate specification of the form *Name/Arity* or *Name*, or a list of such specifications. When the *Name* is given without the *Arity* this refers to all predicates of that name which currently have definitions. If there are none, then nothing will be done. Spy-points can be placed on particular undefined procedures only by using the full form, *Name/Arity*.

nospy *Spec*

Spy-points are removed from all the procedures given by *Spec* (as for **spy**).

debugging

Outputs information concerning the status of the debugging package. This will show whether debug mode is on, and if it is

what spy-points have been set

what mode of leashing is in force.

2.12. Environmental

'NOLC'

Establishes the "no lower-case" convention described in Section 1.14.

'LC'

Establishes the "full character set" convention described in Section 1.14. It is the default setting.

op(*priority*, *type*, *name*)

Treat name *name* as an operator of the stated *type* and *priority* (refer to Section 1.13.2). *name* may also be a list of names in which case all are to be treated as operators of the stated *type* and *priority*.

break

Causes the current execution to be interrupted at the next procedure call. Then the message "[Break (level 1)]" is displayed. The interpreter is then ready to accept input as though it was at top level. If another call of **break** is encountered, it moves up to level 2, and so on. To close the break and resume the execution which was suspended, type the end-of-file character. Execution will be resumed at the procedure call where it had been suspended. Alternatively, the suspended execution can be aborted by calling the evaluable predicate **abort**. Refer to Section 1.10.

abort

Aborts the current execution taking you back to top level. Refer to Section 1.10.

save(*F*)

The system saves the current state of the system into file *F*. Refer to Section 1.7.

prompt(*Old*, *New*)

The sequence of characters (prompt) which indicates that the system is waiting for user input is represented as an atom, and matched to *Old*; the atom bound to *New* specifies the new prompt. In particular, the goal

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

matches the current prompt to X, without changing it. Note that this only affects the prompt issued for read's in the user's program; it will not change the propmts used by the system at top level etc.

system(*String*)

Calls the operating system with string *String* as argument. For example

```
system("ls")
```

will produce a directory listing on Unix.

sh

Suspends C-Prolog and enters a recursive command interpreter. On Unix, the shell used will be that specified in the environment variable SHELL.

2.13. Pre-Processing

expand_term(*T1*, *T2*)

When a program is read in, some of the terms read are transformed before being stored as clauses. If *T1* is a term that can be transformed, *T2* is the result. Otherwise *T2* is just *T1* unchanged. The only transformation available in version 1.2a translates grammar rules into clauses. This means that grammar rules are automatically accepted, and read-in properly, by **consult** and **reconsult**. The user may define his own term rewritting for read-in by adding clauses to **expand_term** with **asserta**.

Appendix I - Summary of Evaluable Predicates

abolish(F,N)	Abolish the procedure named F arity N.
abort	Abort execution of the current directive.
arg(N,T,A)	The Nth argument of term T is A.
assert(C)	Assert clause C.
assert(C,R)	Assert clause C, ref. R.
asserta(C)	Assert C as first clause.
asserta(C,R)	Assert C as first clause, ref. R.
assertz(C)	Assert C as last clause.
assertz(C,R)	Assert C as last clause, ref. R.
atom(T)	Term T is an atom.
atomic(T)	Term T is an atom or integer.
bagof(X,P,B)	The bag of Xs such that P is provable is B.
break	Break at the next procedure call.
call(P)	Execute the procedure call P.
clause(P,Q)	There is a clause, head P, body Q.
clause(P,Q,R)	There is an clause, head P, body Q, ref R.
close(F)	Close file F.
compare(C,X,Y)	C is the result of comparing terms X and Y.
consult(F)	Extend the program with clauses from file F.
current_atom(A)	One of the currently defined atoms is A.
current_functor(A,T)	A current functor is named A, m.g. term T.
current_predicate(A,P)	A current predicate is named A, m.g. goal P.
db_reference(T)	T is a database reference.
debug	Switch on debugging.
debugging	Output debugging status information.
display(T)	Display term T on the terminal.
erase(R)	Erase the clause or record, ref. R.
erased(R)	The object with ref. R has been erased.
expand_term(T,X)	Term T is a shorthand which expands to term X.
exists(F)	The file F exists.
fail	Backtrack immediately.
fileerrors	Enable reporting of file errors.
functor(T,F,N)	The top functor of term T has name F, arity N.
get(C)	The next non-blank character input is C.
get0(C)	The next character input is C.
halt	Halt Prolog, exit to the monitor.
instance(R,T)	A m.g. instance of the record ref. R is T.
integer(T)	Term T is an integer.
Y is X	Y is the value of arithmetic expression X.
keysort(L,S)	The list L sorted by key yields S.
leash(M)	Set leashing mode to M.
listing	List the current program.
listing(P)	List the procedure(s) P.
name(A,L)	The name of atom or number A is string L.
nl	Output a new line.
nodebug	Switch off debugging.
nofileerrors	Disable reporting of file errors.
nonvar(T)	Term T is a non-variable.
nospy P	Remove spy-points from the procedure(s) P.
number(T)	Term T is a number.
op(P,T,A)	Make atom A an operator of type T precedence P.
primitive(T)	T is a number or a database reference
print(T)	Portray or else write the term T.

Predicate	Description
prompt(A,B)	Change the prompt from A to B.
put(C)	The next character output is C.
read(T)	Read term T.
reconsult(F)	Update the program with procedures from file F.
recorda(K,T,R)	Make term T the first record under key K, ref. R.
recorded(K,T,R)	Term T is recorded under key K, ref. R.
recordz(K,T,R)	Make term T the last record under key K, ref. R.
rename(F,G)	Rename file F to G.
repeat	Succeed repeatedly.
retract(C)	Erase the first clause of form C.
save(F)	Save the current state of Prolog in file F.
see(F)	Make file F the current input stream.
seeing(F)	The current input stream is named F.
seen	Close the current input stream.
setof(X,P,B)	The set of Xs such that P is provable is B.
sh	Start a recursive shell
skip(C)	Skip input characters until after character C.
sort(L,S)	The list L sorted into order yields S.
spy P	Set spy-points on the procedure(s) P.
system(S)	Execute command S.
tab(N)	Output N spaces.
tell(F)	Make file F the current output stream.
telling(F)	The current output stream is named F.
told	Close the current output stream.
trace	Switch on debugging and start tracing.
true	Succeed.
var(T)	Term T is a variable.
write(T)	Write the term T.
writeq(T)	Write the term T, quoting names if necessary.
'LC'	The following Prolog text uses lower case.
'NOLC'	The following Prolog text uses upper case only.
!	Cut any choices taken in the current procedure.
\+ P	Goal P is not provable.
X<Y	As numbers, X is less than Y.
X=<Y	As numbers, X is less than or equal to Y.
X>Y	As numbers, X is greater than Y.
X>=Y	As numbers, X is greater than or equal to Y.
X=Y	Terms X and Y are equal (i.e. unified).
T=..L	The functor and args. of term T comprise the list L.
X==Y	Terms X and Y are strictly identical.
X\==Y	Terms X and Y are not strictly identical.
X@<Y	Term X precedes term Y.
X@=<Y	Term X precedes or is identical Y.
X@>Y	Term X follows term Y.
X@>=Y	Term X follows or is identical to term Y.
[F|R]	Perform the (re)consult(s) specified by [F|R].