C-Prolog User's Manual

Version 1.5
October 24, 1996

Edited by Fernando Pereira[1]
SRI International, Menlo Park, California

from material by

David Warren
SRI International, Menlo Park, California

David Bowen, Lawrence Byrd
Dept. of Artificial Intelligence, University of Edinburgh

Luis Pereira
Dept. de Informatica, Universidade Nova de Lisboa

Abstract

This is a revised 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 DECsystem-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 DECsystem-10/20 Prolog manual.

[1]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 VAX[2] machines under the UNIX[3] and VAX/VMS operating systems, on the Sun workstation under 4.1/2 UNIX, and has been ported with minor changes to other 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 builtin (system defined) procedures. Its design was based on the (Edinburgh) DECsystem-10 Prolog system and the system is closely compatible with DECsystem-10 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 *lconsulting* 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/reconsulting 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.

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

```
cprolog
```

that invokes C-Prolog.

[2]VAX, VMS, PDP and DECsystem-10 are trademarks of Digital Equipment Corporation.

[3]UNIX is a Trademark of Bell Laboratories.

We assume that there are three keys or key combinations that achieve the effects of terminating an input line, marking end of input, and interrupting execution of a program. Because these depend on operating system and individual tastes, they are denoted in this manual by END-OF-LINE, END-OF-INPUT and INTERRUPT respectively (they are carriage return, control-Z and control-C on the computer this is being written on).

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 2.4) but it is unnecessarily painful. We shall assume both upper and lower case throughout this manual.

If you type the ‘cprolog’ command, Prolog will output a banner and prompt you for directives as follows:

```
C-Prolog version 1.5
| ?-
```

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 ‘cprolog’ 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.

```
cprolog prog          (Restore ‘‘prog’’)
```

C-Prolog uses six major internal data areas: the *heap*, the *global stack*, the *local stack*, the *trail*, the *atom area* and the *auxiliary stack*. Although the system is initially configured with reasonable allocations for each of those areas, a particular program might exceed one of the allocations, causing an error message. Ideally, the system should adjust the available storage among areas automatically, but this facility is not implemented yet. Instead, the user may specify when starting Prolog the allocations (in K bytes) for some or all of the areas. This is done by giving command line switches between the program name and the optional file argument. For example,

```
cprolog -h 1000 -g 1000 -l 500 bigprogram
```

specifies heap and global stack allocations of 1000 K and a local stack allocation of 500 K. The full set of switches is:

-h *N*	heap allocation is *N* K bytes
-g *N*	global stack allocation is *N* K bytes
-l *N*	local stack allocation is *N* K bytes
-t *N*	trail allocation is *N* K bytes
-a *N*	atom area allocation is *N* K bytes
-x *N*	auxiliary stack allocation is *N* K bytes

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 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-INPUT 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 period (``.''), and that therefore Prolog will not execute anything for you until you have typed the period (and then END-OF-LINE) 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

```
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 END-OF-LINE terminates the question, while typing ``;'' followed by END-OF-LINE 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 END-OF-LINE 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, such as[4]:

```
:-([ bits, bobs, mainpart, testcases, data, junk ]).
```

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. 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:

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

```
?- save(file).
```

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

1.7. 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:

```
cprolog file
```

After execution of this command, the interpreter will be in *exactly* the same 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 1.5 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.

1.8. 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'', ''t'', ''d'' or ''c'' followed by END-OF-LINE. The ''a'' response will force Prolog to abort back to top level, the ''t'' option will switch on tracing, the ''d'' response will switch on debugging and continue the execution, and the ''c'' response will just continue the execution.

1.9. 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* and except for the effect of **abort**s (see below), it is as if the interpreter was at top level. If break is called within a break, then another recursive break is started (and the message will say (level 2) etc). Breaks can be arbitrarily nested.

Typing the END-OF-INPUT character will close the break 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-INPUT character is needed to close the break, because *all* break levels are discarded and the system returns right back to top level. The ''a'' response to INTERRUPT (described above) can also be used to force an abort.

1.10. 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, 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-INPUT charater at top level also causes C-Prolog to terminate.

2. Prolog Syntax

2.1. Terms

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

The constants include *number*s such as

```
0   -999   -5.23  0.23E-5
```

Constants also include *atom*s 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

```
       s
     /  \
   np      vp
   |     / \
 john    v     np
         |     |
       likes  mary
```

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

```
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.

*List*s form an important class of data structures in Prolog. They are essentially the same as the lists of LISP: a list either is the atom

[]

representing the empty list, or is a compound term with functor '.' and two arguments which are respectively the head and tail of the list. Thus a list of the first three natural numbers is the structure

```
  .
 / \
1   .
   / \
  2   .
     / \
    3   []
```

which could be written, using the standard syntax, as

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

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

[1,2,3]

The special list notation in the case when the tail of a list is a variable is exemplified by

[X|L]   [a,b|L]

representing

```
  .          .
 / \        / \
X   L      a   .
              / \
             b   L
```

respectively.

Note that this 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 *string*s. For example,

"Prolog"

represents exactly the same list as

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

2.2. 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 structure

```
  +
 / \
2   1
```

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 5.2).

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.

- In a term written in standard syntax, the principal functor and its following ‘‘(’’ must *not* be separated by any blankspace. Thus

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

  is invalid syntax.
- 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.
  ```
- 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 = (?-)
  ```

2.3. Syntax Errors

Syntax errors are detected when reading. Each clause, directive or in general any term read-in by the evaluable predicate **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[5] 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). The book *Programming in Prolog* gives further examples.

2.4. 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 evaluable predicate ‘NOLC’:

```
| ?- 'NOLC'.
```

To switch back to the full character set convention, call the evaluable predicate ‘LC’:

```
| ?- '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.

3. The Meaning of Prolog Programs

3.1. Programs

A fundamental unit of a logic program is the *literal*, for instance

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

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

A logic *program* consists simply of a sequence of statements called *clauses*, which are analogous to sentences of natural language. A clause comprises a *head* and a *body*. The head either consists of a single literal or is empty. The body consists of a sequence of zero or more literal (that is, it too may be empty). The body literals are called *goals* or *procedure calls*.

A clause with empty head is a *negative clause* or *query*, and represents a command to the interpreter to start proving the goals in its body. The form of a query is

[5] After all, it could not be read.

:- *P,Q,R*.

but, as described in Section 1.5, the ‘‘:-’’ can be omitted when typing a command to the interpreter.

From here on, when we use the term ‘‘clause’’ we mean a *positive* clause, that is one with nonempty head.

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 literal. We interpret this declaratively as

‘‘*P* is true.’’

and procedurally as

‘‘*P* can be satisfied.’’

If the body of a clause is nonempty, the clause is called a *nonunit clause*, and is written in the form

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

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

‘‘*P* is true if *Q* and *R* and *S* are true.’’

or procedurally as

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

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

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

> ‘‘Any X is employed if any Y employs X.’’
>
> ‘‘To find whether a person X is employed,
> find whether any Y employs X.’’

derivative(X,X,1).

> ‘‘For any X, the derivative of X with respect to X is 1.’’
>
> ‘‘The goal of finding a derivative for the expression X with
> respect to X itself is satisfied by the result 1.’’

:- ungulate(X), aquatic(X).

> ‘‘Is it true, for any X, that X is an ungulate and X is
> aquatic?’’
>
> ‘‘Find an X which is both an ungulate and aquatic.’’

In a program, the *procedure* for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a 3-ary predicate **concatenate** might consist of the two clauses

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

where **concatenate**(L1,L2,L3) means ‘‘the list L1 concatenated with the list L2 is the list L3’’.

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form <name>/<arity> is used, for example

concatenate/3.

Certain predicates are predefined by procedures supplied by the Prolog system. Such predicates are called *evaluable predicates*.

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

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

which can be read as

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

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

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

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

3.2. Declarative and Procedural Semantics

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

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

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

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

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

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

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

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

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

For example, if we execute the goal in the query

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

we find that it matches the head of the first clause for **concatenate**, with X instantiated to [a|X1]. The new variable X1 is constrained by the new goal produced, which is the recursive procedure call

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

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

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

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

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

representing a true instance of the original goal

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

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

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

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

in that order, by rematching, against the second clause for **concatenate**, goals already solved once using the first clause.

3.3. Occurs Check

Prolog's unification does not have an *occurs check*, i.e. when unifying a variable against a term the system does not check whether the variable occurs in the term. When the variable occurs in the term, unification should fail, but the absence of the check means that the unification succeeds, producing a *circular term*. Trying to print a circular term, or trying to unify two circular terms, will cause an infinite loop and possibly make Prolog run out of stack space.

The absence of the occur check is not a bug or design oversight, but a conscious design decision. The reason for this decision is that unification with the occur check is at best linear on the sum of the sizes of the terms being unified, whereas unification without the occur check is linear on the size of the smallest of the terms being unified. In any practical programming language, basic operations are supposed to take constant time. Unification against a variable should be thought of as the basic operation of Prolog, but this can take constant time only if the occur check is omitted. Thus the absence of a occur check is essential to make Prolog into a practical programming language. The inconvenience caused by this restriction seems in practice to be very slight. Usually, the restriction is only felt in toy programs.

3.4. The Cut Symbol

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

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

For example, the procedure

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

can be used to test whether a given term is in a list, and the goal

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

will succeed. The procedure can also be used to extract elements from a list, as in

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

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

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

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

A procedure of the form

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

is similar to

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

in an Algol-like language.

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

4. Debugging Facilities

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

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

Much of the information in this chapter is similar but not identical to that of Chapter 8 of *Programming in Prolog*.

4.1. The Procedure Box Control Flow Model

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

Let us look at an example Prolog procedure:

```
         *--------------------------------------*
   Call  |                                      |  Exit
---------> +  descendant(X,Y) :- offspring(X,Y).  + --------->
         |                                      |
         | descendant(X,Z) :-                   |
<--------- +    offspring(X,Y), descendant(Y,Z). + <---------
   Fail  |                                      |  BackTo
         *--------------------------------------*
```

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

Call

This arrow represents initial invocation of the procedure. When a **descendant** goal of the form **descendant**(X,Y) is required to be satisfied, control passes through the Call *port* of the **descendant** box with the intention of matching a component clause and then satisfying any subgoals in the body of that clause. Notice that this is independent of whether such a match is possible, that is first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for **descendant** when meeting a call to **descendant** in some other part of the code.

Exit

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

BackTo

This arrow indicates that a subsequent goal has failed and that the system has come back to this goal in an attempt to match the goal against another clause. Control passes through the BackTo port of the **descendant** box in an attempt to rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary. Note that this is somewhat less informative than the Redo port described in *Programming in Prolog*, because it does not show the path in the program from a failed goal back to the first goal where alternative clauses exist, but only this goal.

Fail

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

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

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

4.2. Basic Debugging Predicates

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

debug

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

nodebug

switches debug mode off. If there are any spy points set then they will be removed.

debugging

prints onto the terminal information about the current debugging state. It shows whether debug mode is on or off and gives various other information to be described later.

4.3. Tracing

The following evaluable predicate may be used to commence an exhaustive trace of a program.

trace

Switches debug mode on, if it is not on already, and ensures that the next time control enters a procedure box, a message will be produced and you will be asked to interact.

When stopped at a goal being traced, you have a number of options which will be detailed later. In particular, you can just type END-OF-LINE (carriage-return) to creep (or single-step) into your program. If you continue to creep through your program you will see every entry and exit to/from every invocation box. However, you will notice that you are only given the opportunity to interact on Call and BackTo ports, i.e. a single creep decision may take you through several Exit ports or several Fail ports. Normally this is desirable, as it would be tedious to go through all those ports step by step. However, if it is not what you want, the following evaluable predicate gives full control over the ports at which you are prompted.

leash(*Mode*)

Sets the *leashing mode* to *Mode*, where *Mode* can be one of the following

full	prompt on Call, Exit, BackTo and Fail
tight	prompt on Call, BackTo and Fail
half	prompt on Call and BackTo
loose	prompt on Call
off	never prompt

or any other combination of ports as described later. The initial *leashing mode* is ‘half’.

4.4. Spy Points

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

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

Spy points are set and removed by the following evaluable predicates which are also standard operators:

spy *X*

Sets spy points on all the procedures given by *X*. *X* is either a single predicate specification, or a list of such specifications. A predicate specification is either of the form <atom>/<arity>, which means the procedure with the name <atom> and an arity of <arity> (for example member/2, foo/0, hello/27), or it is of the form <atom>, which means all the procedures with the name <atom> that currently have clauses in the data-base (e.g. member, foo, hello). This latter form may refer to multiple procedures which have the same name but different arities. If you use the form <atom> but there are no clauses for this predicate (of any arity) then nothing will be done. If you really want to place a spy point on a currently non-existent procedure, then you must use the full form <atom>/<arity>; you will get a warning message in this case. If you set some spy points when debug mode is off then it will be automatically switched on.

nospy *X*

This reverses the effect of spy *X*: all the procedures given by *X* will have previously set spy points removed from them.

The options available when you arrive at a spy-point are described below.

4.5. Format of Debugging Messages

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

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

The ‘‘**’’ indicates that this is a spy-point. If this port is not for a procedure with a spy-point set, then there will be two spaces there instead. If this port is the requested return from a Skip then the second character becomes ‘‘>’’. This gives four possible combinations:

‘‘**’’	This is a spy-point.
‘‘*>’’	This is a spy-point, and you also did a Skip last time you were in this box.
‘‘ >’’	This is not a spy-point, but you did a Skip last time you were in this box.
‘‘ ’’	This is not a spy-point.

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

The number following this is the current depth, that is, the number of direct ancestors this goal has.

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

The goal is then printed so that you can inspect its current instantiation state.

The final ‘‘?’’ is the prompt indicating that you should type in one of the option codes allowed (see next section). If this particular port is unleashed then you will obviously not get this prompt since you have specified that you do not wish to interact at this point.

Notice that not all procedure calls are traced; there are a few basic procedures which have been made invisible since it is more convenient not to see them. These include all primitive I/O evaluable predicates (for example **get**, **put**, **read**, **write**), all basic control structures (that is, ‘,’, ‘;’, ‘->’) and all debugging control evaluable predicates (for instance, **debug**, **spy**, **leash**, **trace**). This means that you will never see messages concerning these predicates during debugging.

4.6. Options Available during Debugging

This section describes the particular options that are available when the system prompts you after printing out a debugging message. All the options are one letter mnemonics. They are read from the terminal with any blanks being completely ignored up to the next end of line. The *creep* option needs only the new line.

The only option which you really have to remember is ‘‘h’’. This provides help in the form of the following list of available options:

END-OF-LINE	creep	a	abort
c	creep	f	fail
l	leap	b	break
s	skip	h	help
r	retry	r <n>	retry goal <n>
q	quasi-skip	n	nodebug
g	goal stack	[	read clauses from terminal
e	exit Prolog		

The first three options are the basic control decisions:

c, END-OF-LINE: 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 procedures, and then follow the control flow through these by leaping from one to the other.

s: Skip

Skip is only valid for Call and BackTo ports. It skips over the entire execution of the procedure. That is, you will not see anything until control comes back to this procedure (at either the Exit port or the Fail port). Skip is particularly useful while creeping since it guarantees that control will be returned after the (possibly complex) execution within the box. If you skip then no message at all will appear until control returns. This includes calls to procedures with spy points set; they will be masked out during the skip. There are two ways of overriding this : there is a Quasi-skip which does not ignore spy points, and the ‘‘t’’ option after an interrupt will disable the masking. Normally, however, this masking is just what is required!

q: Quasi-skip

Is like Skip except that it does not mask out spy points. If there is a spy-point within the execution of the goal then control returns at this point and any action can be performed there. The initial skip still guarantees an eventual return of control, though, when the internal execution is finished.

f: Fail

Transfers to the Fail port of the current box. This puts your execution in a position where it is about to backtrack out of the current invocation, that is, you have manually failed the initial goal.

r: Retry

Transfers to the Call port of the current goal, or if given a numeric argument *n*, to the Call port of the invocation numbered *n*. IF the invocation being retried has been deleted by the cut control primitive, the most recent active invocation before it will be retried. If *n* is out of range, the current invocation is retried. This option is extremely useful to go back to an earlier state of computation if some point of interest was overshot in a debugging session. Note however that side effects, such as database modification, are not undone.

g: Goal Stack

Shows the goal that called the current one, and the one that called it, and so on, that is, the **stack** of pending goals. Goals are printed one per line, from the most recent to the least recent. Each goal is labeled by its recursion level *l*, its invocation number *i* and the ordinal position *p* of the clause currently used to solve the 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 evaluable predicate **abort**.)

[: Read clauses from terminal

starts reading clauses typed by the user as if doing a *consult(user)*. An END-OF-INPUT returns to the debugger.

e: Exit from Prolog

causes an irreversible exit from the Prolog system. (This is the same as the evaluable predicate **halt**.)

h: Help

Displays the table of options given above.

b: Break

Calls the evaluable predicate **break**, thus putting you at interpreter top level with the execution so far sitting underneath you. When you end the break with the END-OF-INPUT character, you will be reprompted at the port at which you broke. The new execution is completely separate from the suspended one; the invocation numbers will start again from 1 during the break. Debug mode is not switched off as you call the break, but if you do switch it off then it will be re-switched on when you finish the break and go back to the old execution. However, any changes to the leashing or to spy points will remain in effect.

n: Nodebug

Turns off debug mode. Notice that this is the correct way to switch debugging off at a trace point. You cannot use the ‘‘b’’ option because it always restores debug mode upon return.

4.7. Reconsulting during Debugging

It is possible, and sometimes useful, to reconsult a file whilst in the middle of a program execution. However this can lead to unexpected behaviour under the following circumstances: a procedure has been successfully executed; it is subsequently re-defined by a reconsult, and is later re-activated by backtracking. When the backtracking occurs, all the new clauses for the procedure appear to the interpreter to be potential alternative solutions, even though identical clauses may already have been used. Thus large amounts of (unwanted) activity takes place on backtracking. The problem does not arise if you do the reconsult when you are at the Call port of the procedure to be re-defined.

5. Evaluable Predicates

This section describes all the evaluable predicates available in C-Prolog. These predicates are provided in advance by the system and they cannot be redefined by the user. Any attempt to add clauses or delete clauses to an evaluable predicate fails with an error message, and leaves the evaluable predicate unchanged. The C-Prolog provides a fairly wide range of evaluable 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 evaluable predicates will be described according to this classification. Appendix I contains a complete list of the evaluable predicates.

5.1. Input and 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:

``| ?- ''	interpreter waiting for command
``| ''	interpreter wating for command continuation line
``| ''	**consult**(user) wating for continuation line
``|:''	default for waiting for other user input

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 typing the END-OF-INPUT character. Any more input requests for a file whose end has been reached causes an error failure.

5.1.1. 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).
```

5.1.2. 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.

5.1.3. 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**.

5.1.4. 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. If the current input stream reaches its end of file, the ASCII character code for control-Z is returned and the stream closed.

get(*N*)

N is the ASCII code of the next non-blank printable character from the current input stream. It has the same behaviour as **get0** on end of file.

skip(*N*)

Skips to just past the next ASCII character code *N* from the current input stream. *N* may be an integer expression. Skipping past the end of file causes an error.

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.

5.2. Arithmetic

Arithmetic is performed by evaluable predicates 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.

Because arithmetic expressions are compound terms, they use up storage that is only recovered on backtracking. The evaluable predicate **expanded_exprs** can be used to avoid this overhead by preexpanding arithmetic expressions into calls to arithmetic evaluation predicates. However, this makes program read-in slower and clauses bigger.

In general, an arithmetic operation that combines integers and floating point numbers will return a floating point number. However, if the result is integral, it is converted back to integer representation, and the same applies to numbers read in by the reader. Thus, the goal

```
| ?- p(2.0).
```

will succeed with the clause

```
p(2).
```

and the result of the query

```
| ?- X is 2*1.5.
```

is

```
X = 5
```

Numbers may be integers

-33 0 9999

or floating point numbers

1.333 -2.6E+7 0.555E-2

Note that the decimal ‘‘.’’ cannot be confused with the end of clause because the latter must be followed by blank space (space, tab or END-OF-LINE). However, if an operator ‘‘.’’ is declared as infix, it will only be possible to apply it to two numbers if they are separated by blank space from the operator.

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*ˆ*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 evaluable predicates 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*.

expanded_exprs(*Old*,*New*)

Unifies *Old* to the current value of the expression expansion flag, and sets the value of the flag to *New*. The possible values of the flag are 'off' (the default) for not expanding arithmetic expressions into procedure calls, and 'on' to do the expansion.

5.3. 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.

5.4. 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.

5.5. 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 *I*th 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*).

5.6. 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 5.3). 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.

5.7. 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 5.2) 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.)

5.8. 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*)

Similar to **assert**(*Clause*), but also unifies *Ref* with the database reference of the clause asserted.

asserta(*C*)

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

asserta(*Clause*,*Ref*)

Similar to **asserta**(*Clause*), but also unifies *Ref* with the database reference of the clause asserted.

assertz(*C*)

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

assertz(*Clause*,*Ref*)

Similar to **assertz**(*Clause*), but also unifies *Ref* with 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*)

Similar to **clause**(*Head*,*Body*) but also unifies *Ref* with 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 non-determinate 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 5.10) which allows a clause to be directly erased via its database reference[6].

5.9. 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.

5.10. 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.

[6]This is a lower level facility, required only for complicated database manipulations.

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 **erase**d, 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 **erase**d.

5.11. Debugging

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**).

leash(*Mode*)

Sets the leashing mode to *Mode*, where *Mode* can be one of the following

full	prompt on Call, Exit, BackTo and Fail
tight	prompt on Call, BackTo and Fail
half	prompt on Call and BackTo
loose	prompt on Call
off	never prompt
N	*N* is an integer. If the binary notation of *N* is 2'*cebf*, the digits *c*, *e*, *b* and *f* correspond to the Call, Exit BackTo and Fail respectively, and are 1 (0) if the corresponding port is leashed (unleashed).

The initial *leashing mode* is 'half'.

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.

5.12. Environmental

'NOLC'

Establishes the no lower-case convention described in Section 2.4.

'LC'

Establishes the full character set convention described in Section 2.4. It is the default setting.

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

Treat name *name* as an operator of the stated *type* and *priority* (refer to Section 2.2). *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 suspended 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-INPUT 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 1.9.

abort

Aborts the current execution taking you back to top level. Refer to Section 1.9.

save(*F*)

The system saves the current state of the system into file *F*. Refer to Section 1.6.

save(*F*,*When*)

Saves the current state of the system into file *F*. **When** is unified with 0 or 1 depending on whether the system is returning from the **save** goal in the original Prolog session or after the saved state in *F* has been restored by invoking Prlog with file *F* as argument.

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 reads 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.

statistics

Shows the current allocations and amounts used for each of the six working areas of C-Prolog, and also the runtime since C-Prolog started. For example:

```
| ?- statistics.
atom space: 64K (15596 bytes used)
aux. stack: 8K (0 bytes used)
trail: 64K (48 bytes used)
heap: 256K (30664 bytes used)
global stack: 256K (0 bytes used)
local stack: 256K (300 bytes used)
Runtime:     1.42 sec.
| ?-
```

5.13. Preprocessing

expand_term(*T1*,*T2*)

Each top level term *T1* read when consulting a file is rewritten into *T2* before being asserted or executed. The default transformations provided by this predicate are the ones for grammar rules and for inline expansion of arithmetic expressions. The user may define further transformations as clauses for the predicate **term_expansion/**2, which has similar arguments. User defined transformations are applied *before* system-defined ones.

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 *N*th 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 *X*s 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.
expanded_exprs(*O*,*N*)	Expression expansion if *N*=on.
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*.

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 *X*s 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*.
statistics	Display execution statistics.
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*].