PROLOG

reference manual

ARM Evaluation System

Acorn OEM Products

PROLOG

Part No 0448,012
Issue No 1.0
14 August 1986

© Copyright Acorn Computers Limited 1986

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act, or for the purpose of review, or in order for the software herein to be entered into a computer for the sole use of the owner of this book.

Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

- The manual is provided on an 'as is' basis except for warranties described in the software licence agreement if provided.
- The software and this manual are protected by Trade secret and Copyright laws.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

There are no warranties implied or expressed including but not limited to implied warranties or merchantability or fitness for purpose and all such warranties are expressly and specifically disclaimed.

In case of difficulty please contact your supplier. Every step is taken to ensure that the quality of software and documentation is as high as possible. However, it should be noted that software cannot be written to be completely free of errors. To help Acorn rectify future versions, suspected deficiencies in software and documentation, unless notified otherwise, should be notified in writing to the following address:

Customer Services Department,
Acorn Computers Limited,
645 Newmarket Road,
Cambridge
CB5 8PD

All maintenance and service on the product must be carried out by Acorn Computers. Acorn Computers can accept no liability whatsoever for any loss, indirect or consequential damages, even if Acorn has been advised of the possibility of such damage or even if caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Econet® and The Tube® are registered trademarks of Acorn Computers Limited.

ISBN 1 85250 007

Published by:
Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN, UK

Edited by William Clocksin

Based on material from the C-PROLOG User's Manual by Fernando Pereira, which is based on the EMAS PROLOG User's Manual by Luis Dama which in turn is based on the User's Guide to DEC system- 10/ 20 PROLOG by Fernando Pereira, David Warren, David Bowen, Lawrence Byrd, and Luis Pereira.

Produced by Baddeley Associates Limited, Cambridge

Contents

1. About this guide 1
   1.1 Conventions used in this guide 1
2. About PROLOG 2
3. Using ARM PROLOG 3
   3.1 Access to ARM PROLOG 3
   3.2 Reading in programs 4
   3.3 Questions 5
   3.4 Saving a program state 7
   3.5 Restoring a saved program 8
   3.6 Program execution and interruption 8
   3.7 Nested executions: break and abort 9
   3.8 Leaving PROLOG 10
4. Summary of PROLOG syntax 11
      1 Comments 11
      2 Terms 11
      3 Variables 12
      4 Compound terms 12
      5 Operators 15
   4.1 Syntax errors 20
5. Summary of PROLOG semantics 21
   5.1 Declarative and procedural semantics 24
   5.2 Occurs check 26
   5.3 Specifying control information 27
6. ARM PROLOG's built-in predicates 29
   6.1 Input/ output 30
   6.2 Reading in programs 31
      6.2.1 compile(F) 31
      6.2.2 consult(F) 31
      6.2.3 [F1, F2, ..., Fn] 31
   6.3 Opening and closing files 31
      6.3.1 see(F) 31
      6.3.2 seeing(F) 31
      6.3.3 seen 31
      6.3.4 tell(F) 31
      6.3.5 telling(F) 32
      6.3.6 told 32

6.4 Reading and writing PROLOG terms 32
  6.4.1 read(X) 32
  6.4.2 read(X,Y) 32
  6.4.3 write(X) 32
  6.4.4 display(X) 32
  6.4.5 writeq(X) 32
6.5 Getting and putting characters 33
  6.5.1 nl 33
  6.5.2 get0(N) 33
  6.5.3 get(N) 33
  6.5.4 skip(N) 33
  6.5.5 put(N) 33
  6.5.6 tab(N) 33
6.6 Arithmetic 33
6.7 Affecting the flow of the execution 35
  6.7.1 , Q 35
  6.7.2 P ; Q 35
  6.7.3 true 35
  6.7.4 X = Y 35
  6.7.5 X \= Y 35
  6.7.6 ! 36
  6.7.7 \+P 36
  6.7.8 P -> Q ; R 36
  6.7.9 P -> Q 36
  6.7.10 repeat 36
  6.7.11 fail 36
  6.7.12 forall(G,T) 36
6.8 Classifying and operating on PROLOG terms 37
  6.8.1 var(X) 37
  6.8.2 nonvar(X) 37
  6.8.3 atom(X) 37
  6.8.4 integer(X) 37
  6.8.5 atomic(X) 37
  6.8.6 functor(T,F,N) 37
  6.8.7 arg(I,F,X) 37
  6.8.8 X =.. Y 38
  6.8.9 name(X,L) 38
  6.8.10 call(X) 38
  6.8.11 X (where X is a variable) 39
  6.8.12 numbervars(T,M,N) 39

6.9 Processing sets 39
6.9.1 setof(X,P,S) 39
6.9.2 bagof(X,P,B) 40
6.9.3 findall(X,P,L) 41
6.9.4 Y^Q 41
6.10 Comparing terms 41
6.10.1 X == Y 42
6.10.2 X \== Y 42
6.10.3 T1@<T2 42
6.10.4 T1@>T2 42
6.10.5 T1@=<T2 42
6.10.6 T1@>=T2 42
6.10.7 compare(Op, T1, T2) 42
6.10.8 sort(L1, L2) 43
6.10.9 keysort(L1, L2) 43
6.11 Manipulating the PROLOG program database 43
6.11.1 assert(C) 43
6.11.2 assert(C,R) 44
6.11.3 asserta(C) 44
6.11.4 asserta(C,R) 44
6.11.5 assertz(C) 44
6.11.6 assertz(C,R) 44
6.11.7 clause(P,Q) 44
6.11.8 clause(P,Q,R) 44
6.11.9 retract(C) 45
6.11.10 abolish(N,A) 45
6.11.11 listing(N,A) 45
6.12 Manipulating the internal indexed database 45
6.12.1 recorded(K,T,R) 46
6.12.2 recorda(K,T,R) 46
6.12.3 recordz(K,T,R) 46
6.12.4 erase(R) 46
6.13 Interacting with the programming environment 46
6.13.1 writedepth(X,Y) 46
6.13.2 writewidth(X,Y) 47
6.13.3 unknown(X,Y) 47
6.13.4 op(P,T,N) 47
6.13.5 break_handler 47
6.13.6 error_handler(N,X) 48
6.13.7 abort 48

6.13.8 save(F) 48
6.13.9 statistics(X,Y) 48
6.13.10 system(X) 48
6.14 Defining modules 49
6.14.1 module(X) 50
6.14.2 endmodule(X) 50
6.14.3 visa(A,F) 50
6.14.4 sacred 50
6.14.5 omni 51
6.14.6 import(F,M) 51
7. What's particular to ARM PROLOG 53
7.1 The user language 53
7.2 Implementation details 54
7.3 Restrictions on data structures 54
7.3.1 Integers 54
7.3.2 Strings 55
7.3.3 Atoms 55
7.3.4 Variables 55
7.3.5 Compound terms 55
7.4 Errors 55
7.5 Input and output 56
7.6 Operator declarations 56
7.7 Built-in predicates exported to the user 57
7.8 Arithmetic expressions 60

1. About this guide

If PROLOG is new to you, you should first read a standard text such as *Programming in PROLOG* (by W F Clocksin and C S Mellish, published by Springer-Verlag, 1981).

This guide includes:

- instructions on starting and running a basic PROLOG session on the ARM evaluation system (starting on page 3)
- two chapters which together give a quick-reference summary of the main points covered in *Programming in PROLOG* (pages 11 and 21)
- a detailed guide to the built-in predicates available in ARM PROLOG (page 29)
- a summary of the ways in which ARM PROLOG differs from other versions of PROLOG (page 53).

1.1 Conventions used in this guide

As you work with the PROLOG interpreter, you type a full stop then press RETURN

to indicate the end of a line. PROLOG then interprets what you have typed. For simplicity, in this guide we do not show the RETURN, and we show the full stop only in:

- syntax definitions
- full examples.

2. About PROLOG

PROLOG is a simple but powerful programming language originally developed at the University of Marseilles as a practical tool for programming in logic. PROLOG is especially suitable for high-level symbolic programming tasks and has been applied in many areas of artificial intelligence research.

The interactive ARM PROLOG system consists of an incremental PROLOG compiler, a run-time system, and a wide range of built-in (system-defined) procedures. The system is closely compatible with DECsystem-10 PROLOG and thus is reasonably compatible with descendants of DECsystem-10 PROLOG, such as C-PROLOG, Quintus PROLOG, and POPLOG.

3. Using ARM PROLOG

The text of a PROLOG program is normally created in a number of files using the TWIN editor. ARM PROLOG can then be instructed to read in programs from these files. ARM PROLOG can read in programs in either of two ways:

- consulting
- compiling.

When a file is consulted, it is read in together with information about the clause needed when debugging the program. During normal use, however, such extra information is not required, and programs are usually compiled. Consulting takes more time than compiling, and consulted programs occupy more store than compiled ones.

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

When your programs start to grow to a fair size, it is also a good idea to have one file which only contains commands to the system 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.

3.1 Access to ARM PROLOG

Because PROLOG makes syntactic use of the difference between upper-case and lower-case letters it is important that you have your keyboard set up so that it accepts lower case in the normal way. Hence, ensure the CAPS LOCK and SHIFT LOCK lamps are not lit.

To enter PROLOG, type:

```
prolog
```

PROLOG will output a banner and prompt you for directives as follows:

```
ARM PROLOG Version 1.23
?-
```

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

Once you are in PROLOG, there are three important keys or sequences to remember:

- RETURN terminates an input line
- CTRL D marks end of input
- ESCAPE interrupts execution of a program.

3.2 Reading in programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the compiler. The clauses of a procedure do not have to be immediately consecutive, but remember that their relative order in the file controls the order in which the system tries them.

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

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

which will instruct the system to consult the program. The file specification `file` must be a PROLOG atom. It may be any ADFS filename specification. PROLOG does not use special extension. Note that if this filename 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:

```
?- ['joe.test'].
```

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.

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

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

In this case all three files would be consulted.

To compile clauses, use the `compile` predicate, which will take either an atom or a list of atoms as an argument:

```
?- compile(prog).
?- compile([myprogram,'jon.extras',testbits]).
```

Compilation is the recommended way to read in clauses, as it is faster and the program will occupy less store.

Clauses may also be typed in directly. To enter a clause, you must give the directive:

```
?- [user].
:
```

The system is now in a state where it expects input of clauses or directives. This is indicated by the `:` prompt. To get back to the top level of the system, type CTRL D.

Typing clauses directly into ARM PROLOG is only recommended if the clauses will not be needed permanently, and are few in number. For larger programs you should use TWIN to produce a file containing the text of the program.

3.3 Questions

When PROLOG is at top level (signified by an initial prompt of `?`, it reads in terms and treats them as directives to the system to try and satisfy some goals. These directives are called questions (or queries). Remember that PROLOG terms must terminate with a full stop (.), and that therefore PROLOG will not execute anything for you until you have typed the full stop, and then pressed RETURN at the end of the directive.

For example, suppose list membership has been defined by the following (where _ is an anonymous variable):

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

If the goals 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
more (y/ n)?
```

At this point the system waits for you to indicate whether that solution is sufficient, or whether you want it to backtrack to see if there are any more solutions. Typing `n` (for no) followed by [RETURN] terminates the question, while typing `y` (for yes) followed by [RETURN] causes the system to backtrack to look for alternative solutions. If no further solutions can be found it outputs:

```
no
```

The outcome of some questions is shown below.

```
?- member(X, [tom,dick,harry]).
X = tom
more (y/n)? y
X = dick
more (y/n)? y
X = harry
more (y/n)? y
no
?- member(X, [a,b,f(Y,c)]), member(X, [f(b,Z),d]).
X = f(b,c)
Y = b
Z = c
more (y/n)? n
yes
?- member(X, [f(_),g]).
X = f(_1728)
more (y/n)? y
X = g
more (y/n)? y
no
?-
```

When PROLOG reads terms from a file (or from the keyboard following a call to `[user[`), it treats them all as program clauses. In order to get the system to execute directives from a file they must be preceded by `?-`.

3.4 Saving a program state

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`, call the goal:

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

The goal `save` can be called at top level, from within a break level (see below), or from anywhere within a program.

Note that save only makes a copy of the current memory contents. Saving a state does not make any changes to source files. If a new version of PROLOG is installed, saved states created with the old version will almost certainly become unusable. You are thus advised to keep up to date source files for your programs at all times, and only use saved states for convenience.

3.5 Restoring a saved program

Once a program has been saved into a file file, PROLOG can be restored to this saved state by calling the goal:

```
?- restore(file).
```

After execution of this goal, 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 succeeded. 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.

There is another version of save:

```
save(file, Restart)
```

will save into the file file as before, and will instantiate the variable Restart to 0. When the file is restored, Restart will be instantiated to 1.

3.6 Program execution and interruption

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

Only when execution of one directive is complete does the system become ready for another directive. However, one may interrupt the normal execution of a directive by pressing ESCAPE causes the goal break_handler to be called at the earliest opportunity.

3.7 Nested executions: break and abort

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. When the built-in predicate `break_handler` is called, the message:

```
(Break) ?-
```

will be displayed. This signals the start of a break-level and except for the effect of `abort` (see below), it is as if the interpreter were at top level. If `break_handler` is called within a break-level, then another recursive break-level is started. Break-levels may be nested arbitrarily deeply, although this practice is not very useful.

Typing CTRL D will close the break-level and resume the execution which was suspended, starting at the procedure call where the suspension took place. Do look at the prompt before you type CTRL D. If the prompt is `?-` you are at the very top level, and CTRL D will take you right out of PROLOG, and your current state will be irrevocably lost. An equivalent way of getting out of break levels is to type:

```
end_of_file.
```

in response to the `?-` prompt. This too will take you out of PROLOG if you are at the top level. It has the advantage over

CTRL D that you can easily and clearly include it in scripts, should you want to call `break_handler` in a script.

To abort the current execution, forcing an immediate failure of the directive being executed and a return to the top level of the system, call the built-in predicate `abort`, either from the program or by executing the directive:

```
?- abort.
```

within a break. In this case no CTRL D is needed to close the break, because all break levels are discarded and the system returns right back to top level.

3.8 Leaving PROLOG

To leave ARM PROLOG, type:

```
?- halt.
```

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

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

Typing CTRL D at top level also causes ARM PROLOG to terminate.

4. Summary of PROLOG syntax

For full details on PROLOG syntax, see *Programming in PROLOG.*

1. Comments

ARM PROLOG supports two kinds of comments. One form is:

```
/ * comment text */
```

These comments can stretch over any number of lines and pages. That is precisely the problem with them. If you forget a closing comment bracket your comment will quietly eat large quantities of your program. So these comments are not favoured for annotations within clauses. Also, the comment brackets may not be nested. The other style of comment is end of line comments:

```
% comment reaching to the end of the line
```

which are opened by a percent sign and closed by a new line.

2. Terms

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

The constants include integers such as:

```
0   -999   8'177   2'101101
```

Constants also include atoms such as:

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

The name of an atom may be any sequence of characters not including the ASCII NUL character. It must be put in single quotes, unless it is:

- a sequence of sign characters
- an identifier starting with a lower case letter
- a singleton special character like `!` or `;`.

If you want a single quote in a name, it must be written twice. Thus the quotes are needed in each of the atoms spelled as:

```
'!!'   '%rem'   'has spaces'   '1234'   'NotAVariable'   'a-b'
'has one quote '' in it'
```

As in other programming languages, constants are definite elementary objects.

3. Variables

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 a writable storage location as in most programming languages; rather it is a local name for some data object.

It is possible to write variable names entirely in upper case, but solid upper case is far less readable than mixed case. Nor is it necessary to write constant symbols entirely in lower case; `nilTree` is a perfectly good atom name.

4. Compound terms

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:

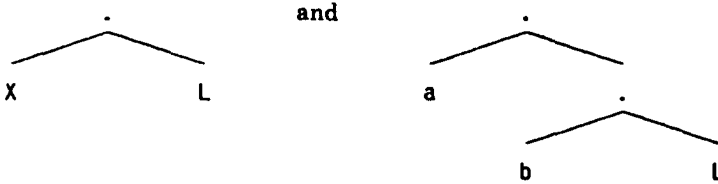
```
point(X,Y,Z)
```

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

You may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the term:

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

would be pictured as the structure:



Sometimes it is convenient to write certain functors as operators. Binary functors may be declared as infix operators, and unary functors may be declared as prefix or postfix operators. Thus it is possible to write:

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

as optional alternatives to:

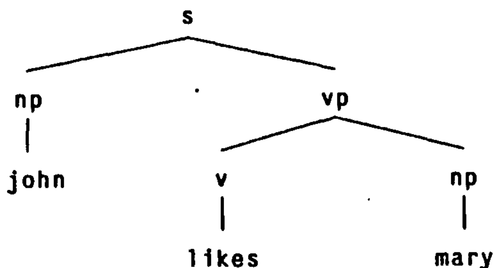
```
+(X,Y)    ;(P,Q)    <(X,Y)    +(X)
```

Operators are described fully in the next section.

Lists form an important class of data structures in PROLOG. They are essentially the same as the lists of Lisp. A list is either:

- the atom `[]` representing the empty list, or
- 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 represented by the compound term having the following structure:



which can be written using the standard syntax, as

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

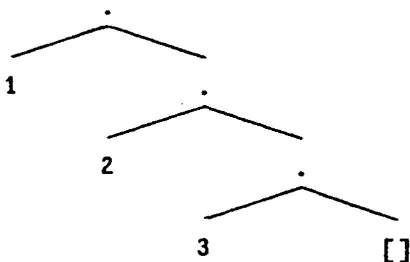
but which is normally written, in a special list notation, as

```
[1,2,3]
```

The special list notation in the case when the tail of a list is a variable is shown by these two examples:

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

which can be represented by the structures:



respectively.

Note that this list syntax is only syntactic sugar for terms of the form `.(_,_)` and does not provide any additional facilities not otherwise available in PROLOG.

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

```
'PROLOG'
```

represents exactly the same list as:

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

As with atoms and single quotes, wpage a double quote in a string you must write it twice. For example:

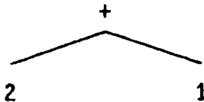
```
This ''string'' has four ''double quotes'' in it'
```

5. 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 compound term having the structure:



and not the number 3. The addition would only be performed if the structure was passed as an argument to an appropriate goal, such as `is`.

The PROLOG syntax caters for operators of three main kinds:

- infix
- prefix
- postfix.

An ifix 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 and 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 sub-expression 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 sub-expressions which are the arguments of the operator must be of lower precedence than the operator itself. That is, their principal functors must be of lower precedence, unless the sub-expression is explicitly bracketed (which gives it zero precedence). wpage of type `xfy`, only the first or left-hand sub-expression must be of lower precedence; the second can be of the same precedence as the main operator; and vice vers for an operator of type `yfx`. Operators of type `yfy` do not exist and would of course be useless.

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

```
a-b+c
```

is syntactically correct, and means:

```
(a-b)+c, that is, +(-(a,b),c)
```

Note that the expression would be syntactically incorrect if the operators had type `xfx`. The expression would mean:

```
a-(b+c), that is, -(a,+(b,c))
```

if the types were both `xfy`.

The possible types for a prefix operator are:

```
fx and fy
```

and for a postfix operator they are:

```
xf and 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 syntactically correct, although:

```
not P
```

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

In ARM PROLOG, an atom of name Name is declared as an operator of type Type and precedence Precedence by calling the built-in predicate `op`:

```
?- op(Precedence, Type, Name).
```

The argument Name can also be a list of names of operators of the same type and precedence. There are two predicates for checking whether an atom has operator properties and finding what they are.

```
?- current_op(Precedence, Type, Name).
```

backtracks through all the current operator declarations matching the goal.

It is possible to have more than one operator of the same name, so long as they are of different kinds: 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 built-into ARM PROLOG. The following operators have been built-into ARM PROLOG as though the following `op` goals have been executed:

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

Operator declarations are most usefully 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 and ','(X,Y)
```

represent the same compound term. But note that a comma written as a quoted atom is not a standard operator, and that operator declarations for the atom , will have no effect on the parsing of the punctuation mark.

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 black space. Thus `point (X,Y,Z)` is incorrect syntax. Extra space elsewhere does no harm.

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 incorrect syntax, and must be written as:

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

If an operator is written without an argument, as an ordinary atom, the atom is treated as an expression of the same precedence as the operator, and must therefore be bracketed where necessary. Thus the brackets are necessary in `X = (?-)`.

4.1 Syntax errors

Syntax errors are detected when reading. Each clause, directive or in general any term read-in by the built-in predicate `read` that fails to comply with syntax requirements is displayed on the keyboard 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:  incorrect position for culprit
Culprit:  variable
Context:  member(X, X <<here>> L).
Type input again.
```

If you don't want to re-type the input, type the null statement `[].` to return to the prompt.

If the syntax error occurs when reading a file, ARM PROLOG also displays additional information. For example, if the above line appeared in the file named `t`, an attempt to read the clause would result in the message:

```
Syntax Error:  incorrect position for culprit
Culprit:  variable
Context:  member(X , X <<here>> L).
in column 0 of line 2 of file t.
```

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

5. Summary of PROLOG semantics

For full details on PROLOG semantics, see *Programming in PROLOG.*

A fundamental unit of a PROLOG program is the goal. Here are three examples of goals:

```
gives(tom, apple, teacher).

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

X < Y.
```

A goal is a term distinguished by the context in which it appears in the program. The (principal) functor of a goal is called a predicate. It corresponds roughly to a verb in natural language, or to a precedure name in a conventional programming language. Note that predicates with different arities are quite unrelated, even when they have the same name.

A PROLOG program consists 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 term or is empty. The body consists of a sequence of zero or more goals (that is, it too may be empty). The body goals are sometimes called procedure calls.

A clause with empty head is a question, and represents a command to the system to start proving the goals in its body. If P1, P2, *.. Pn stand for n goals, the form of a question is:

```
?- P1, P2, ..., Pn
```

As described before, the `?-` is likely to be already present as a prompt.

From here on, when we use the term clause, we mean 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. 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:

```
Q :-P1, P2, ..., Pn
```

where Q stands for the head and P1, P2, ..., Pn stand for n goals which make up the body. We can read such a clause either declaratively as 'Q is true if P1 and P2 and ... and Pn are true' or procedurally as 'To satisfy goal Q, satisfy goals P1 and P2 and ... and Pn'.

Clauses in general contain variables. Note that variables in difference 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:

clause:	employed(X) :- employs(Y, X).
declarative:	'Any X is employed if any Y employs X'.
procedural:	'To find whether a person X is employed, find whether any Y employs X.'
clause:	derivative(X, X, 1).
declarative:	'For any X, the derivative of X with respect to X is 1.'
procedural:	'The goal of finding a derivative of the expression X with respect to X itself is satisfied by the result 1.'
clause:	?- ungulate(X), aquatic(X).
declarative:	'Is it true, for some X, that X is an ungulate and X is aquatic?'
procedural:	'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 ternary 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 built-in 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 generally be eliminated by defining an extra predicate. For example, 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.

There is one point about disjunction which should be noted: a cut inside a disjunction cuts the whole clause, and not just the disjunction. Thus the definition:

```
(A ; B) :- A.
(A ; B) :- B.
```

though appealing, would be incorrect. Cuts don't notice commas or semicolons. They do notice `\+`, `call`, `forall`, and so on. So you can't always eliminate a disjunction by using a new clause.

5.1 Declarative and procedural semantics

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

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

For example, if a program contains the preceding procedure for `concatenate`, 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 withwpageogram. This sequencing information is, however, very relevant for the procedural semantics that 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 PROLOG programmers direct the system to excute their 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 precedural semantics.

To execute a goal, the system searches forwards from the beginning of the program for the first clause whose head matches according to the unification algorithm. The unification algorithm 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 alX1. 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 bl/ 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.

5.2 Occurs check

PROLOG's unification algorithm does not have what is known as an occurs check. That is, 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 occurs check is not a bug or design oversight, but a conscious design decision. The reason for this decision is that unification with the occurs check is a best linear on the sum of the sizes of the terms being unified, whereas unification without the occurs 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 occurs check is omitted. Thus the absence of a occurs check is essential to make PROLOG into a practical programming lanugage. The inconvenience caused by this restriction seems in practice to be very slight. Usually, the restriction is only felt in toy programs.

5.3 Specifying control information

Besides the sequencing of goals and clauses, PROLOG provides one other 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 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|_] ).
member(X, [_|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:

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

is similar to:

```
procedure S;
if P then Q else R;
end;
```

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 norm method for eliminating a disjunction by defining a extra predicate cannot be applied to a disjunction containing a cut.

There is an explicit notation for if-then-else which you may prefer to use. That is:

```
S :- P -> Q ; R.
```

This control stucture may be embedded in a clause. For example:

```
max_of_3(X, Y, Z, Max) :-
( X > Y -> T = X ; T = Y ),
( T > Z -> Max = T ; Max = Z ).
```

This example is only to show that if-then-else doesn't have to be the only thing in a clause. It is not an example of good style. The equivalent in an Algol-like language would be:

```
if X > Y then T:=X, else T:=Y;
if T > Z then Max:=T else Max:=Z;
end;
```

6. ARM PROLOG's built-in predicates

This section describes all the built-in predicates available in ARM 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 a built-in predicate fails with an error message, and leaves the evaluable predicate unchanged. ARM PROLOG provides a fairly wide range of built-in predicates to perform the following tasks:

(1) input/ output

(2) reading in programs

(3) opening and closing files

(4) reading and writing PROLOG terms

(5) getting and putting characters

(6) arithmetic

(7) affecting the flow of the execution

(8) classifying and operating on PROLOG terms

(9) processing sets

(10) comparing terms

(11) manipulating the PROLOG program database

(12) manipulating the internal indexed database

(13) interacting with the programming environment

(14) defining modules.

The built-in predicates will be described according to this classification. There is a complete list of the built-in predicates, starting on page 57.

6.1 Input/ output

A total of eight input/ output 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. A stream to a file `F` is opened for input by the first `see(F)` executed. File `F` then becomes the current input stream. Similarly, a stream to file `H` is opened for output by the first `tell(H)` executed. File `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 file `user`, denoting the user, is utilised for both input and output. Output to `user` appears on the display screen; input from `user` is typed at the keyboard. When the system is waiting for input on a new line, one of these two prompts will be displayed:

?- system waiting for command

:

`consult(user)` waiting for clause input

When the current input (or output) stream is closed, `user` becomes the current input (or output) stream.

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

A file is referred to by its name, written as an atom. For example:

```
myfile    'F123'    data_list    'people.tom'
```

Note that full Acorn DFS filenames may be used, although it will usually be necessary to enclose them within single quotes to be legal atoms.

All input/ output errors normally cause an abort.

End of file is signalled on the user's keyboard by typing CTRL D. Any more input requests for a file whose end has been reached causes an error failure.

6.2 Reading in programs

6.2.1 compile(F)

This predicate instructs the system 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 system for that procedure. Clauses which are compiled cannot be accessed by the built-in predicates `retract`, `listing`, or `clause`.

6.2.2 consult(F)

This is like compile, except that additional information is stored in the database to allow certain debugging operations. Clauses which are consulted may be accessed by the built-in predicates `retract`, `listing`, or `clause`.

6.2.3 [F1, F2, ..., Fn]

This is a shorthand way of consulting a list of files. Thus:

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

is merely a shorthand for:

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

6.3 Opening and closing files

6.3.1 see(F)

This predicate makes file `F` become the current input stream.

6.3.2 seeing(F)

This unifies file `F` with the name of the current input file.

6.3.3 seen

This closes the current input steam.

6.3.4 tell(F)

This makes file `F` become the current output stream.

6.3.5 telling(F)

This unifies file `F` with the name of the current output file.

6.3.6 told

This closes the current output stream.

6.4 Reading and writing PROLOG terms

6.4.1 read(X)

The next term, terminated by a full stop (`.` followed by a 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.

6.4.2 read(X,Y)

The next term is read and unified with X just as if you had called `read(X)`. Y is unified with a list of variables and their names, where each element of the list is Atom=Var, Atom being a PROLOG atom representing the name of the variable, and Var being the PROLOG variable that name signified. An example of such a list might be `['X' = _271, 'Vars' = _275[`. The list is in no particular order.

6.4.3 write(X)

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

6.4.4 display(X)

The term X is written on the current output stream in standard parenthesised prefix notation. This is handy for checking the effect your operator declarations have had.

6.4.5 writeq(X)

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

6.5 Getting and putting characters

6.5.1 nl

A new line is started on the current output stream.

6.5.2 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 (26) is returned and the stream closed.

6.5.3 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.

6.5.4 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.

6.5.5 put(N)

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

6.5.6 tab(N)

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

6.6 Arithmetic

Arithmetic is performed by built-in predicates which take as arguments arithmetic expressions. An arithmetic expression is a termbuilt from built-in functors, numbers and variables. At the time of evaluation, each variable in an arithmetic expression must be bound to an integer or to an arithmetic expression. Each built-in functor stands for an arithmetic operation.

Open-code is generated for arithmetic expressions known at compile-time. Variables in arithmetic expression may be bound to either integers or other arithmetic expressions; in the latter case, the bound expression will be interpreted at run-time. Arithmetic exceptions cause goal failure.

The following expressions can make up arithmetic expressions to be used on the right-hand side of an is. X and Y are expressions.

+X	unary addition
-X	unary subtraction
X + Y	addition
X - Y	subtraction
X * Y	multiplication
X / Y	division
X mod Y	remainder
X /\ Y	bit conjunction
X \/ Y	bit disjunction
X << Y	bit shift X left by Y bits
X >> Y	bit shift X right by Y bits
\X	bit negation
cputime	cpu time since the start of the session, in milliseconds
calls	number of PROLOG procedure calls since the start of the session
an integer	the value of any integer
a list	the first element of the list is evaluated as an expression. Because a quoted string is just a list of integers, this allows a quoted character to be used in place of its ASCII code; for example, A behaves within arithmetic expressions as the integer 65

The arithmetic built-in 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. Gives an error message 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
succ(M,N)	expresses the relation 'M and N are both integers, and N=M+1'. It may be used to add 1 to M or to subtract 1 from N

6.7 Affecting the flow of the execution

6.7.1 , Q

Means P and Q.

6.7.2 P ; Q

Means P or Q.

6.7.3 true

Always succeeds.

6.7.4 X = Y

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

6.7.5 X \ = Y

Succeeds when X and Y do not unify.

6.7.6 !

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

6.7.7 \ +P

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

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

6.7.8 P -> Q ; R

Analogous to 'if P then Q else R' It is defined as if by:

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

6.7.9 P -> Q

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

```
P -> Q ; fail
```

6.7.10 repeat

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

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

6.7.11 fail

Always fails.

6.7.12 forall(G,T)

For each way of proving the generator G, calls the test T once. Logically it expresses bounded quantification: T is true for each G. Procedurally it is a for loop. For example, if `between(Lo,Hi,X)` enumerates integers `X` between `Lo` and `Hi`, we can define primality by `isPrime(N) :- forall(between(2,N,X), N mod X =\ = 0).`

6.8 Classifying and operating on PROLOG terms

6.8.1 var(X)

Tests whether X is currently instantiated to a variable.

6.8.2 nonvar(X)

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

6.8.3 atom(X)

Checks that X is currently instantiated to an atom.

6.8.4 integer(X)

Checks that X is currently instantiated to an integer.

6.8.5 atomic(X)

Checks that X is currently instantiated to an atom or integer.

6.8.6 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, the goal fails. 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.

6.8.7 arg(I,F,X)

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

6.8.8 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. For example:

```
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 alist of determinate length whose head is an atom, or to a list of length 1 whose head is a number.

6.8.9 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. For example:

```
name(product,[112,114,111,100,117,99,116])
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. For example:

```
?- name(X,[104,101,108,108,111])).
X = hello
?- name(X,'hello').
X = hello
```

Beware: there are PROLOG atoms which cannot be constructed this way. One example is `123`. The PROLOG reader accepts that as an atom, but given the goal `?- name(X,'123').`, X will be instantiated to the integer `123`.

6.8.10 call(X)

If X is instantiated to a term which would be acceptable as a 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.

6.8.11 X (where X is a variable)

Exactly the same as `call(X)`.

6.8.12 numbervars(T,M,N)

Unifies each of the variables in term T with a special term (`'$ VAR'(I)` for I from M to N-1), so that `write(T)` prints each of those variables in a readable way. Its real purpose is to make the term T ground. M should be bound to an integer when the call is made. After the call, N-M is the number of variables the term used to have.

6.9 Processing sets

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

6.9.1 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. 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 built-in predicte 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.

6.9.2 bagof(X,P,B)

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.

6.9.3 findall(X,P,L)

This is similar to `bagof`, except that all free variables in the generator P are taken to be existentially quantified whether you say so or not; and that, if no solutions can be found, L will be unified with the empty list. There are rare occasions when it is more convenient than `setof`.

6.9.4 Y^Q

The system recognises this as meaning 'there exists an X such that 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.

6.10 Comparing terms

These built-in predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should not be used when what you really want is arithmetic comparison 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
- integers, from $-2^{28}$to + 2^{28-1}
- atoms, in alphabetical (ASCII) order
- compound 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)]
```

The rest of this section lists the basic predicates for comparison of arbitrary terms.

6.10.1 X == Y

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

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

fails (answers no) ause X and Y are distinct uninstantiated variables. However, the question:

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

succeeds ause the first goal unifies the two variables.

6.10.2 X \ == Y

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

6.10.3 T1@<T2

Term T1 is before term T2 in the standard order.

6.10.4 T1@>T2

Term T1 is after term T2 in the standard order.

6.10.5 T1@=<T2

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

6.10.6 T1@>=T2

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

6.10.7 compare(Op, T1, T2)

The result of comparing terms 1 and T2 is Op, where Op will be unified with:

=	when T1 is identical to T2
<	when T1 is before T2 in the standard order
>	when T1 is after T2 in the standard order

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

6.10.8 sort(L1, L2)

The elements of the list L1 are sorted into the standard order, and any identical (that is, ==) 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.)

6.10.9 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.)

6.11 Manipulating the PROLOG program database

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 cause in the compiled 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 ordinary users.

Note that `retract`, `abolish` and `clause` may only access clauses that have been consulted or asserted. It is not possible to `retract`, `abolish` or use `clause` on a clause which has been read in by `compile`.

6.11.1 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 is as though the goal `assertz(C)` had been called. C must be instantiated to a non-variable.

6.11.2 assert(C,R)

Similar to `assert(C)`, but also unifies R with the database reference of the clause asserted.

6.11.3 asserta(C)

Add clause C to the program. The new clause omes the first clause for the procedure concerned.

6.11.4 asserta(C,R)

Similar to `asserta(C)`, but also unifies R with the database reference of the clause asserted.

6.11.5 assertz(C)

Add clause C to the program. The new clause omes the last clause for the procedure concerned.

6.11.6 assertz(C,R)

Similar to `assetz(C)`, but also unifies R with the database reference of the clause asserted.

6.11.7 clause(P,Q)

P must be bound to a non-variable term, and the program is searched for the first clause whose head matches P. The head and body of that clause is unified with P and Q respectively. If the clause is a unit clause, Q will be unified with `true`. This predicate may be used in a non-determinate fashion: that is, it will successively match clauses matching the argument through backtracking.

6.11.8 clause(P,Q,R)

Similar to `clause(P,Q)` but also unifies R with the database reference of the clause concerned. If R is not given at the time of the call, P 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.

6.11.9 retract(C)

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

6.11.10 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. In particular, since `retract` matches against the clause it retracts, that is taken to be a use of that clause, and the space will not be reclaimed until the `retract` goal is failed or exited determinately.

See also `erase`, which allows a clause to be directly erased via its database reference. This is a lower-level facility, required only for complicated database manipulations.

6.11.11 listing(N,A)

List the clauses for the predicate having name N, arity A on the current output stream. Only clauses that have been asserted or that have been read in by `consult` are accessible. Clauses that have been read in by `consult` will be listed with the variable names preserved. Clauses that have been asserted will have variable names replaced by internal names.

6.12 Manipulating the internal indexed 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`, but a program such as an expert system shell (which is obliged to maintain a database where it has no control over the names used) can use the internal database to prevent the user's knowledge base clashing with the program's own clauses.

6.12.1 recorded(K,T,R)

The internal database is searched for terms recorded under the key K. These terms are successively unified with term T in the order they occur in the database. At the same time, R is unified with the database reference of the recorded item. The key must be given, and may be an atom or compound term. If it is a compound term, only the principal functor is significant.

6.12.2 recorda(K,T,R)

The term T is recorded in the internal database as the first item for the key K, where R is its database reference. The key must be given, and only its principal functor is significant.

6.12.3 recordz(K,T,R)

The term T is recorded in the internal database as the last item for the key K, where R is its database reference. The key must be given, and only its principal functor is significant.

6.12.4 erase(R)

The recorded item or clause whose database reference is R is, in effect, 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 or backtracking will the item be actually removed from the database.

6.13 Interacting with the programming environment

6.13.1 writedepth(X,Y)

When terms are written (using `write`, `writeq`, `display` and so on), it is possible to specify a depth beyond which a nested term will be written in an abbreviated form. `writedepth(X,Y)` unifies X to the current depth limit, and unifies Y to a desired new depth limit. Terms nested beyond the depth limit are written as ....

6.13.2 writewidth(X,Y)

It is possible to limit the length of a line of input to a given number of characters. `writewidth(X,Y)` unifies X to the current line width limit, and unifies Y to a desired new line width limit.

6.13.3 unknown(X,Y)

It is possible to specify what action is to be taken by the system when an attempt is made to call a predicate for which no clauses exist. `unknown(X,Y)` unifies X to the current action, and unifies Y to a desired new action. Possible actions are: `error` (call the error handler), `break` (call the break handler) or `fail`.

6.13.4 op(P,T,N)

Treat atom N as an operator of the stated type T and priority P. Argument N may also be a list of names in which case all are to be treated as operators of the stated type and priority.

6.13.5 break_handler

When [ESCAPE]

is pressed, `break_handler` is called at the earliest opportunity. The definition of `break_handler` can be supplied by the user, but the default definition is provided which behaves as follows. The current execution is be suspended at the next procedure call. Then the message `(break) ?-` is displayed. The system is then ready to accept input as thought it was at top level. If another call of `break_handler` is encountered, another break state is entered recursively, and so on. To close the break and resume the execution which was suspended, type [CTRL] D. Execution will be resumed at the procedure call where it had been suspended. Alternatively, the suspended execution can be aborted by calling the built-in predicate `abort`.

6.13.6 error_handler(N,X)

When an execution error occurs, `error_handler` is called at the earliest opportunity with an error code N (a small integer) and a culprit X (any term). The definition of `error_handler` can be supplied by the user, but the default definition is provided which behaves as follows. An error message corresponding to N is printed, followed by the culprit. Then, `abort` is called. User- defined error handlers should always call `abort`, as the state of the system may be unrecovered.

6.13.7 abort

Aborts the current execution taking you back to top level.

6.13.8 save(F)

The system saves the current state of the system into file F.

6.13.9 statistics(X,Y)

Instantiates X to the number of PROLOG goals called since the beginning of the session. Instantiates Y to the number of milliseconds elapsed since the beginning of the session.

6.13.10 system(X)

Given string or atom X representing an ADFS command, execute the command. The number of characters in the command is limited to 255. Goal `system` succeeds regardless of the outcome of the command. The goal fails if the command is too long. Here is an example of obtaining a catalogue listing of ADFS device `:4`, and transferring a file into the current directory:

```
?- system('cat :4').
$                    (02)
Drive:4             Option 00 (Off )
Dir. TestDir        Lib. Library

fred     WR (02)
yes

?- system('copy :4.test @').
yes
```

6.14 Defining modules

A module consists of a collection of clauses that define procedures and declare operators. The clauses are enclosed in the brackets `module` and `endmodule`. Procedure definitions and operator declarations within the brackets are said to be local to the module. A module constitutes a wall around its local procedures whose transparency is under the control of the programmer. Procedures local to a module are said to be at the same scope level as the module.

Nesting of modules is permitted, but nesting itself does not confer any locality implications. If module A is nested within module B, then A and B can be rewritten non-nested with identical meaning. Nesting is permitted as a syntactic convenience ause the user's program may use `consult` or `compile` to cause other programs to be compiled from within its module, and the compiled programs may contain other modules.

Six primitives are defined here:

- `module`
- `endmodule`
- `import`
- `visa`
- `sacred`
- `omni`.

The purpose of `module` and `endmodule` is to denote the scope of a module. The purpose of `import` is to control visibility by creating procedures that are visible within the current module. The purpose of `visa`, `sacred`, and `omni` is to confer attributes on to procedures defined within the current module.

6.14.1 module(X)

Denotes the entry into a scope level of lexical modularity having name X. Module X omes the current module. Modules can be nested, but procedures local to a module nested within the current module are not considered as being defined within the current module. It is illegal to enter a module of name N within the scope of a module of name N, and an attempt to do this will result in an error `N is already current.`

6.14.2 endmodule(X)

Denotes the end of a scope level of lexical modularity. The name X must be the same as the name of the current module, or an error `scope violation` occurs. After `endmodule` is evaluated, the previous level omes current. It is permitted to enter the same module more than once, provided that the previous entries are closed by `endmodule.`

6.14.3 visa(A,F)

A is an attribute, or a list of attributes, or the empty list (legal attributes are the atoms `sacred` and `omni`). F is a functor denotation or a list of functor denotations (the functor having name X and arity N is denoted by the structure X/n). Let X/n be the functor denotation which specifies a procedure P defined within the current module. The `visa` primitive specifies that it is permissible for another module to import P, subject to the attributes in A. P is not made visible, but simply has permission to be visible. An attempt to grant `visa` to an `omni` procedure will result in an error `P is already visible.` If no attributes (other than default permission) need to be conferred, then A is the empty list.

6.14.4 sacred

If the `sacred` attribute is conferred to procedure P, then any subsequent attempts to modify the contents of P outside the current module will fail and be met with an error `attempt to modify a sacred procedure.` This is intended to provide security when it is desired to maintain the integrity of procedures that are imported and hence subject to attempts of attack. This mechanism is used to write-protect built-in predicates.

6.14.5 omni

If the `omni` attribute is conferred on to procedure P, then P is visible to all modules. This mechanism is used to provide a way of making all the built-in predicates visible to all modules without overhead.

6.14.6 import(F,M)

F is a functor denotation or a list of functor denotations (the functor having name X and arity N is denoted by the structure X/n). Let X/n be the functor denotation which specifies a procedure P defined within the module M. The `import` primitive specifies that P is now visible within the current module. It is necessary for P to have been granted `visa` from module M, otherwise an error `P does not exist` occurs. If P is declared sacred within module M, then P will be sacred within the current module. At attempt to import an omni procedure will result in an error `P is already visible`.

The semantics of the metacall `call(X)` relate to modules in an awkward way if the `call(X)` is used in a module different than the current module. In this case, the procedure to be called will be one visible in the current module, and not one visible in the module where the `call(X)` is found. To solve the awkward case, a new metacall is required. The new metacall is defined such that `call(X,M)` calls procedure X as defined in module M.

For example, a module is defined that makes available a quicksort procedure. The quicksort procedure is declared as sacred, with permission to be visible outside the module. The split procedure is local to the module, and can never be visible outside the module. The concatenate procedure is required, so it is imported from a module called `lists`.

```
?- module(sort).

?- import(concatenate/ 3,lists).

?- visa(sacred,quicksort/ 2)

quicksort([],[]).
quicksort([H|T) ,S) :-
                          split(H<T<A<B),
                          quicksort(B<B1),
                          concatenate(A1,[H|B1],S).

split(H,[A|X],[A|Y],Z) :- A < H, split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :- H < A, split(H,X,Y,Z).
split(_,[],[],[]).

?- endmodule(sort).
```

7. What's particular to ARM PROLOG

This section describes all the features of ARM PROLOG that differ from other versions of PROLOG.

7.1 The user language

The dialect of PROLOG available to the user is compatible with the Edinburgh Syntax as used in *Programming in PROLOG* and other texts. ARM PROLOG conforms with the major implementations of PROLOG in supporting overloaded operator declarations with precedences in the range 1..1200, and curly brackets. Programs may contain grammar rules.

Some relevant deviations from normal practice are as follows:

- input is in upper-and-lower-case, so the following syntactic devices are considered obsolete and have been discarded: `,..`, `LC`, `NOLC` and the infix form of the functor .
- the character set consists of 256 characters, represented as integer codes in the range 0 to 255. Character codes in the range 0 to 127 are the ASCII characters, except that character code 8'140 appears as instead of `. Character shapes for codes in the range 128 to 255 are not defined, but those character codes retreated by ARM PROLOG as symbol (sign) characters
- some commonly used predicates defined in various PROLOG libraries have been built-in
- a normal PROLOG top level is provided. As clauses are compiled, the source clause is also retained in the database together with the names of variables used within the clause. This may be used, for example, to list clauses (using `listing`) exactly as they were typed in. However, the user has the option to compile clauses without retaining the source clause in the database. The advantage is faster compilation and less memory usage, but such clauses may not be accessed by `clause`, `listing`, or `retract`

- debugging facilities are not built-in. ause clauses can be compiled, some of the information needed by a built-in debugger can be lost, and to increase performance, clauses may be executed in ways not obvious to the user. However, it is possible to use an interpretive debugger (available from the PROLOG library) in the usual way, which imposes a conventional execution strategy.

7.2 Implementation details

ARM PROLOG includes the following features:

- large address space (2^{28}bytes)
- incremental garbage collection is performed during backtracking, and when clauses are retracted
- all clauses are compiled to compact byte-code representation, with automatic vaiable mode analysis and peephole optimisations
- some built-in predicates are open-coded
- tail recursion optimisation, and early detection of determinacy
- automatic clause indexing on first argument of goals
- arbitrarily long bit-strings that are tested for equality during unification (for compact strings, big numbers).

7.3 Restrictions on data structures

Data structures used in ARM PROLOG programs have the following restrictions:

7.3.1 Integers

Integers are in the range -134217728 to 134217727. Overflow is not checked. An integer to the base N (where 0< N < 9) may be represented as standard, but base 0 is reserved for representing character codes (for example, `0'A == 8'101`).

7.3.2 Strings

Strings are represented as lists of character codes as in the Edinburgh standard, and the double-quote notation can be used. There is an internal data structure used for packed byte strings of arbitrary length, but this is not made available to the user.

7.3.3 Atoms

The names of atoms are internally represented as packed byte strings of arbitrary length (limit 16777215 characters). External syntax as for the Edinburgh standard.

7.3.4 Variables

A maximum of 255 differently-named variables may appear in a clause.

7.3.5 Compound terms

The maximum arity of a functor is 16777215, but in practice is subject to memory limitations. Only terms having arity less than 256 can be compiled.

7.4 Errors

The error-handling mechanism is accessible to the user as a PROLOG procedure `error_handler`. The default action for errors is to print a message and a culprit. Additionally, syntax errors cause extra information to be printed. If the current file is not `user`, then the filename and line number and column number in the file is printed. After a syntax error, recovery is attempted by skipping to what is likely to be the end of the clause, and reading again.

Pressing the interrupt key ESCAPE causes the procedure `break_handler` to be entered at the earliest opportunity. The default break handler provides a top level with the prompt preceded by the word (`Break`). To exit from the break, type the end-of-file character CTRL `D`. The break handler can also be modified by the user.

7.5 Input and output

Output is flushed only:

- when a newline is printed, or
- when input from the user is requested.

The built-in procedure `ttyflush` is provided for unusual applications. Eight files can be open at any given time, including `user` input and `user` output.

7.6 Operator declarations

These are the same as provided by the major PROLOG implementations. In ARM PROLOG, operator declarations are subject to module scoping, and may be imported in the same way as procedures. The following operators have been built-into ARM PROLOG as though the following `op` goals have been executed:

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

7.7 Built-in predicates exported to the user

The following table lists all of ARM PROLOG's built-in predicates. In the middle column:

- A means a predicate new in ARM PROLOG
- L means a PROLOG library predicate which has been built into ARM PROLOG
- O means that the compiler open-codes the predicate (emits a specific machine instruction).

The O entry should not normally concern the user.

abolish(F,N)	Retract all clauses for procedure F, arity N.
abort	Abort execution of current directive.
arg(N,T,A)	O The Nth argument of term T is A.
assert(C)	Assert clause C.
asserta(C)	Assert C as first clause.
assertz(C)	Assert C as last clause.
atom(T)	O T is an atom.
atomic(T)	O T is an atom or an integer.
bagof(X,P,B)	The bag of Xs such that P is provable is B.
break_handler	A Call the break handler.
call(P)	O Call procedure P.
clause(P,Q)	There is a clause, head P, body Q.
clause(P,Q,R)	There is a clause, head P, body Q, ref R.
compare(C,X,Y)	C is the result of comparing terms X and Y.
compile(F)	Extend the program with clauses from file F. Do not retain source.
consult(F)	Extend the program with clauses from file F.
display(T)	Write term T in prefix form.
endmodule(A)	A End of module A.
erase(R)	Erase the clause with ref R.
error_handler(N,T)	A Call the error handler with error number N, culprit T.
fail	O Backtrack immediately.
findall(T,G,L)	L Like bagof(T,G,L), but free variables are existentially quantified.
findall(T,G,S,L)	Like finadall(T,G,L1), append(L1,S,L), but more efficient.
forall(P,Q)	L For all P provable, prove Q.
functor(T,F,N)	O The principal functor of term T is F with arity N.

get(C)	The next non-blank input character is C.
get0(C)	The next input character is C.
halt	Abort and exit PROLOG.
import(P,M)	Import procedure(s) P from module M.
integer(T)	O T is an integer.
Y is X	O Y is the value of arithmetic expression X.
keysort(L,S)	The list L sorted by key yields S.
length(L,M)	L The length of list L is M.
listing(F,A)	List the clauses of predicate with functor F, arity A.
name(A,L)	The name of atom or string or number A is list L.
nl	Output a newline.
nonvar(T)	O Term T is a non-variable.
not(P)	Like \ +(P), but a warning is printed if P is not ground.
numbervars(T,M,N)	Number the variables in term T from M to N-1.
op(P,T,A)	Make atom A an operator of type T, precedence P.
put(C)	Ouput the character C.
read(T)	Read term T.
read(T,L)	A Read term T, with variable name list L.
recorda(K,T,R)	Record T first under key K, with ref R.
recorded(K,T,R)	T is recorded under key K, with ref R.
recordz(K,T,R)	Record T last under key K, with ref R.
repeat	Succeed.
retract(T)	Erase the first clause matching T.
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 name F.
seen	Close the current input stream; revert to { \ tt user} .
setof(X,P,S)	The set of Xs such that P is provable is S.
simpleterm(T)	L Term T is not a compound term.
skip(C)	Skip input characters until after character C.
sort(L,S)	The list L sorted into canonical order is S.
statistics(X,Y)	A X is the number of PROLOG calls since the start of the session. Y is the number of milliseconds elapsed since the start of the session.
succ(M,N)	L Integer N is the successor of integer M.
system(L)	A Execute string or atom L as an ADFS command.
tab(N)	Output N spaces.
tell(F)	Make the current output stream F.
telling(F)	The current output stream is named F.
told	Close the current output stream revert to { \ tt user} .

true	Succeed once.
ttyflush	Write out the terminal output buffer.
unknown(O,N)	Set the 'unknown procedure' action from O to N.
var(T)	O T is a variable.
visa(L)	A Confer permissions on exportable procedures.
write(T)	Write T.
writedepth(O,N)	Set the write depth from O to N.
writeq(T)	Write T, quoting if necessary.
writewidth(O,N)	A Set the output linewidth from O to N.
!	O Cut ny choices taken in the current procedure.
\ +(P)	P is not provable.
P -> ! ; R	O If P is provable, the prove Q, else prove R.
P -> Q	O Like P -> Q ; fail.
X < Y	O As integer expressions, X is less than Y.
X =< Y	O As integer expressions, X is less than or equal to Y.
X > Y	O As integer expressions, X is greater than Y.
X >= Y	O As integer expressions, X is greater than or equal to Y.
X =:= Y	O As integer expressions, X is equal to Y.
X =\ = Y	O As integer expressions, X is not equal to Y.
X = Y	O X unifies with Y.
X \ + Y	X does not unify with Y.
T =.. L	The functor and arguments of T are elements of 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 in the canonical ordering.
X @=< Y	Term X precedes or is identical with Y.
X @> Y	Term X follows term Y.
X @>= Y	Term X follows or is identical with Y.

7.8 Arithmetic expressions

Open-code is generated for arithmetic expressions known at compile-time. Variables in arithmetic expression may be bound to either integers or other arithmetic expressions; in the latter case, the bound expression will be interpreted at run-time. Arithmetic exceptions cause goals failure.

The following expressions can make up arithmetic expressions to be used on the right-hand side of an is. X and Y are expressions.

Expression	Meaning
+X	Unary addition.
-X	Unary subtraction.
X + Y	Addition.
X - Y	Subtraction.
X * Y	Multiplication.
X / Y	Division.
X mod Y	Remainder.
X /\ Y	Bit conjunction.
X \/ Y	Bit disjunction.
X << Y	Bit shift X left by Y bits.
X >> Y	Bit shift X right by Y bits.
\X	Bit negation.
cputime	CPU time since the start of the session, in milliseconds.
calls	Number of PROLOG procedure calls since the start of the session.
an integer	The value of any integer.
a list	The first element of the list is evaluated as an expression.

Acorn
The choice of experience.