Department of Artificial Intelligence
University of Edinburgh

DECsystem-10 PROLOG USER'S MANUAL

10 November 1982

D.L. Bowen (editor), L. Byrd, F.C.N. Pereira,
L.M. Pereira, D.H.D. Warren

This manual corresponds to Prolog version 3.47.

Copyright (C) 1982 University of Edinburgh, Dept of Artificial Intelligence

Table of Contents

Introduction 1

Notational Conventions 2

1. HOW TO RUN PROLOG 3

1.1. Getting Started 3
1.2. Reading-in Programs 3
1.3. Inserting Clauses at the Terminal 4
1.4. Directives: Questions and Commands 4
1.5. Syntax Errors 7
1.6. Undefined Predicates 7
1.7. Program Execution And Interruption 8
1.8. Exiting From The Interpreter 9
1.9. Nested Executions - Break and Abort 9
1.10. Saving and Restoring Program States 9
1.11. Initialisation 10
1.12. Logging 11

2. DEBUGGING 13

2.1. The Procedure Box Control Flow Model 13
2.2. Basic Debugging Predicates 15
2.3. Tracing 15
2.4. Spy-points 16
2.5. Format of Debugging messages 17
2.6. Options available during Debugging 18
2.7. Reconsulting during Debugging 21

3. COMPILING 23

3.1. Calling the Compiler 23
3.2. Public Declarations 23
3.3. Mixing Compiled and Interpreted Code 24
3.4. Mode Declarations 25
3.5. Indexing 26
3.6. Tail Recursion Optimisation 26
3.7. Practical Limitations 27

4. BUILT-IN PROCEDURES 29

4.1. Input / Output 30
4.1.1. Reading-in Programs 31
4.1.2. File Handling 32
4.1.2.1. An Example 32
4.1.3. Input and Output of Terms 33
4.1.4. Character Input/Output 34
4.2. Arithmetic 36
4.3. Comparison of Terms 38
4.4. Convenience 40
4.5. Extra Control 41
4.6. Information about the State of the Program 42
4.7. Meta-Logical 44

4.8. Modification of the Program 46
4.9. Internal Database 47
4.10. Sets 49
4.11. Compiled Program 51
4.12. Debugging 52
4.13. Definite Clause Grammars 54
4.14. Environmental 57

I. The Prolog Language 63

I.1. Syntax, Terminology and Informal Semantics 63
I.1.1. Terms 63
I.1.2. Programs 66
I.2. Declarative and Procedural Semantics 69
I.2.1. Occur Check 70
I.3. The Cut Symbol 71
I.4. Operators 72
I.5. Syntax Restrictions 74
I.6. Comments 75
I.7. Full Prolog Syntax 76
I.7.1. Notation 76
I.7.2. Syntax of Sentences as Terms 77
I.7.3. Syntax of Terms as Tokens 78
I.7.4. Syntax of Tokens as Character Strings 79
I.7.5. Notes 81

II. Programming Examples 83

II.1. Simple List Processing 83
II.2. A Small Database 83
II.3. Quick-Sort 84
II.4. Differentiation 84
II.5. Mapping a List of Items into a List of Serial Numbers 85
II.6. Use of Meta-Predicates 85
II.7. Prolog in Prolog 86
II.8. Translating English Sentences into Logic Formulae 87

III. Installation Dependencies 89

III.1. Getting Started 89
III.2. Using a Terminal without Lower-Case 89
III.3. Using a Monitor without Virtual Memory 90
III.4. Using TOPS-20 90

Summary of Evaluable Predicates 91

Standard Operators 94

Index 97

INTRODUCTION

Prolog is a simple but powerful programming language developed at the University of Marseilles [Roussel 75], as a practical tool for programming in logic [Kowalski 74] [van Emden 75] [Colmerauer 75]. 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.

For an introduction to Prolog, readers are recommended to consult [Clocksin & Mellish 81]. However, for the benefit of those who do not have access to a copy of this book, and for those who have some prior knowledge of logic programming, a summary of the language is included in Appendix I of this manual.

This manual describes the Prolog system developed in the Department of Artificial Intelligence at the University of Edinburgh for the DECsystem-10 [Warren 77] [Warren et al. 77]. The system comprises an interpreter and a compiler, both written largely in Prolog itself. At the user level the compiler is viewed as a built-in procedure which may be called from the interpreter.

When compiled, a procedure will run 10 to 20 times faster and use store more economically. However, it is recommended that the new user should gain experience with the interpreter before attempting to use the compiler. The interpreter facilitates the development and testing of Prolog programs as it provides powerful debugging facilities. It is only worthwhile compiling programs which are well-tested and are to be used extensively.

Certain aspects of the Prolog system are unavoidably installation dependent. In particular, the Monitor being used affects a number of the built-in procedures. Whenever there are differences, this manual describes the Edinburgh installation which runs TOPS-10 version 7.01, and users of other installations should refer to Appendix III.

This manual is based on the "User's Guide to DECsystem-10 Prolog" by L.M. Pereira, F.C.N. Pereira and D.H.D. Warren, which it supersedes. Part of Chapter 2 is taken from [Byrd 80]. Useful comments on drafts of this manual were made by Lawrence Byrd, Luis Jenkins, Richard O'Keefe, Fernando Pereira, Robert Rae and Leon Sterling.

The Prolog system is maintained by the Department of Artificial Intelligence on behalf of the Special Interest Group in Artificial Intelligence of the Engineering Board Computing Committee of the Science and Engineering Research Council.

NOTATIONAL CONVENTIONS

In this manual we shall assume the "full character-set" convention which requires the availability of lower-case characters. When lower-case is not available, the "no lower-case" convention must be used. See Appendix III for details.

Notice that metavariables are underlined in this manual. For example, a symbol such as foo or P may be used in the text to refer to some item in the object language, Prolog.

Predicates in Prolog are distinguished by their name AND their arity. The notation name/arity is therefore used when it is necessary to refer to a predicate unambiguously; e.g. concatenate/3 specifies the predicate which is named "concatenate" and which takes 3 arguments. Predicate names are written in bold type.

We adopt the following convention for delineating character strings in the text of this manual: when a string is being used as a Prolog atom it is written in single quotes, e.g. 'user'; but in all other circumstances double quotes are used.

When referring to keyboard characters, printing characters are written as ordinary text strings, e.g. "h", while control characters are written with a preceding "^". Thus ^C is the character you get by holding down the Control key while you type "c". Finally, the special control characters carriage-return, line-feed and escape (or altmode) are often abbreviated to <cr>, <lf> and <esc> respectively.

CHAPTER 1

HOW TO RUN PROLOG

The DECsystem-10 Prolog system offers the user an interactive programming environment with tools for incrementally building programs, debugging programs by following their executions, and modifying parts of programs without having to start again from scratch.

The text of a Prolog program is normally created in a file or a number of files using one of the standard text editors. The Prolog interpreter can then be instructed to read-in programs from these files; this is called consulting the file.

1.1. Getting Started

To run the Prolog interpreter, perform the Monitor command:

```
.r prolog
```

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

```
Edinburgh DEC-10 Prolog version 3.47
University of Edinburgh, September 1982

| ?-
```

At this point the interpreter is expecting input of a directive, i.e. a question or command (see Section 1.4). You cannot type in clauses immediately (but see Section 1.3). This state is called interpreter_top_level. While typing in a directive, the prompt (on following lines) becomes "| ". That is, the "?-" appears only for the first line of the directive, and subsequent lines are indented.

1.2. Reading-in Programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the interpreter. The clauses of a 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, just type the file name inside list brackets (followed by full-stop and carriage-return), thus:

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

This instructs the interpreter to read-in (consult) the program. The file specification file must be a Prolog atom. It may include a device specification and/or an extension, but not a path. Note that it is then necessary to surround the whole file specification with single quotes; e.g.

```
| ?- ['dska:myfile.pl'].
```

The specified file is then read in. Clauses in the file are stored ready to be interpreted, while any directives are obeyed as they are encountered. When the end of the file is found, the interpreter displays on the terminal the time spent for read-in and the number of words occupied by the program. This indicates the completion of the command.

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

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

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

```
| ?- [-tests,-fixes].
```

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 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, i.e. any such previously existing clauses in the database will be deleted. Reconsulting is useful for telling Prolog about corrections to your program.

1.3. Inserting Clauses at the Terminal

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

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

and the new prompt "| " shows that the interpreter is now in a state where it expects input of clauses or directives. To return to interpreter top level, type ^Z. This is equivalent to an end of file for the ersatz file 'user'.

1.4. Directives: Questions and Commands

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

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

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

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

The full syntax of a question is "?-" followed by a sequence of goals. E.g.

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

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

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

Remember that Prolog terms must terminate with a full stop ("."), and that therefore Prolog will not execute anything until you have typed the full stop (and then carriage-return) at the end of the question.

If the goal(s) specified in a question can be satisfied, and if there are no variables as in this example, 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 input of either just a carriage-return (<cr>) or else a ";" followed by <cr>. Simply typing <cr> terminates the question; the interpreter responds with "yes". However, typing ";" 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]).

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

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

X = f(_52)

yes
| ?-
```

Commands are like questions except that

1. Variable bindings are not displayed if and when the command succeeds.

2. You are not given the chance to backtrack through other solutions.

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

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

directs the system to check whether 3 belongs to the list [1,2,3], 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. If no solution can be found, the system gives:

```
?
```

as a warning.

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

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

If a command like this was contained in the file 'myprog' then typing the

following at top-level would be a quick way of loading your entire program:

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

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

1.5. Syntax Errors

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

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

gives:

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

if ':' has not been declared as an infix operator.

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

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

1.6. Undefined Predicates

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

- 'trace', which causes calls to predicates with no clauses to be reported and the debugging system to be entered at the earliest opportunity;

- 'fail', which causes calls to such predicates to fail (the default state).

The evaluable predicate

```
unknown(OldState,NewState)
```

unifies OldState with the current state and sets the state to NewState. It fails if the arguments are not appropriate. The evaluable predicate debugging prints the value of this state along with its other information. Please note that there is a time (not space) overhead of about 70% when running interpreted programs with the facility enabled ('trace' state). It is hoped that this facility will be improved in the future.

1.7. Program Execution And Interruption

Execution of a program is started by giving the interpreter a directive which contains a call to one of the program's 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 typing ^C once if the system is in input wait, or twice if it is running. This ^C_interruption has the effect of suspending the execution, and the following message is displayed:

```
function (H for help):
```

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

```
a   abort the current command as soon as possible
b   break the current execution - see Section 1.9
c   continue the execution
e   exit from Prolog, closing all files
g   switch garbage collection (initially on)
h   list available commands
m   exit to Monitor level (can use Monitor command CONTINUE to proceed)
n   disable trace - see Chapter 2
t   enable trace - see Chapter 2
```

Note that it is not possible to break or abort directly out of compiled code. Each of the options "a", "b" and "t" requests action by the interpreter, so that if you have interrupted compiled code these cause a resumption of execution which only stops when control is returned to the interpreter.

Note also that the abort option ("a") does not get you out of [user] (inserting clauses from the terminal). You have to type ^Z to stop inserting clauses (or you can use the exit ("e") option to get right out of Prolog).

If when trying to interrupt a program with ^C you accidentally get to Monitor level, (perhaps because you typed too many ^Cs), type CONTINUE to return to the ^C interruption.

1.8. Exiting From The Interpreter

To exit from the interpreter and return to monitor level either type ^Z at interpreter top level, or call the built-in procedure halt, or use the "e" (exit) command following a ^C interruption. ^Z is different from the other two methods of exit in that it prints out some statistics. Also it is possible to re-start (using the Monitor command CONTINUE) after a ^Z if you change your mind.

1.9. Nested Executions - Break and Abort

The Prolog system provides a way to suspend the execution of your program and to enter a new incarnation of the top-level where you can issue directives to solve goals etc. This is achieved by calling the evaluable predicate break or by typing "b" after a ^C interruption (Section 1.7).

This causes the interpreter to suspend execution immediately prior to the next call to an INTERPRETED procedure. The message:

```
[ Break  (level 1) ]
```

is then displayed. This signals the start of a break-level, and you can type questions just as if the interpreter was at top level.

The interpreter indicates your current break level (i.e. depth of nested breaks) by printing the break level before the final yes/no response to questions. E.g. at break level 2 this would look like:

```
| ?- true.

[2] yes
| ?-
```

A ^Z character will close the break and resume the execution which was suspended, starting at the procedure call where the suspension took place.

A suspended execution can be aborted by issuing the directive:

```
| ?- abort.
```

within a break. In this case no ^Z is needed to close the break; ALL break levels are discarded and the system returns right back to top-level.

1.10. Saving and Restoring Program States

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

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

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

This predicate may be called at any time, for example it may be useful to call it in a break in order to save an intermediate execution state.

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

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

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

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

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

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

1.11. Initialisation

When Prolog starts, it looks for a file called 'prolog.bin' in the user's default path. If found, this file is presumed to be a saved program state and is restored. The effect is the same as if you had typed

```
| ?- restore('prolog.bin').
```

If no such file is found, a similar search is made for a file called 'prolog.ini'. If there is one, it is consulted, as if you had typed

```
| ?- ['prolog.ini'].
```

The idea of 'prolog.ini' is that you can put regularly used procedures and directives in there so that you don't have to type them every time you run Prolog.

The 'prolog.bin' facility is likely to be useful in conjunction with plsys(run(_,_)) (see page 61) which allows you to run other programs from within Prolog. If you can get the other program to run Prolog again when it finishes, this facility makes it possible to return to the exact state where you were before running the other program. In this case the save/2 evaluable predicate (page 57) should be used to save the state (into 'prolog.bin') since you will need to distinguish returning from the initial save, and returning from a subsequent (automatic) restore when Prolog is re-run.

1.12. Logging

When Prolog is entered, all terminal interaction is automatically written to the file 'prolog.log' in append mode. That is, if 'prolog.log' already exists in the user's logged-in directory, then the new data is appended to it. Otherwise 'prolog.log' is created in the user's logged-in directory. This facility can be switched off by calling the evaluable predicate nolog, and on again by calling log.

CHAPTER 2

DEBUGGING

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

- The Procedure_Box model of Prolog execution which provides a simple way of 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 also in Chapter 8 of [Clocksin & Mellish 81] which is recommended as an introduction.

2.1. The Procedure Box Control Flow Model

During debugging the interpreter prints out a sequence of goals in various states of instantiation in order to show the state the program has reached in its execution. However, in order to understand what is occurring it is necessary to understand when and why the interpreter prints out goals. As in other programming languages, key points of interest are 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   |                                       |    Redo
          *---------------------------------------*
```

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 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; i.e. first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for descendant when meeting a call to descendant in some other part of the code.

Exit
: This arrow represents a successful return from the 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.

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

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

2.2. 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 a ^C (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.

2.3. 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. The effect of trace can also be achieved by typing "t" after a ^C interruption of a program.

At this point you have a number of options which will be detailed in Section 2.6. In particular, you can just type <cr> (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 Redo 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 Leashing_Mode to Mode, where Mode can be one of the following:

```
full  - prompt on Call, Exit, Redo and Fail
tight - prompt on Call, Redo and Fail
half  - prompt on Call and Redo
loose - prompt on Call
off   - never prompt
```

or any other combination of ports as described in Section 4.12.

The initial value of Leashing Mode is 'half'. (You could use a 'prolog.ini' file to change it if you always wanted it set to something else.)

2.4. Spy-points

For programs of any size, it is clearly impractical to creep through the entire program. Spy-points make it possible to stop the program whenever it gets to a particular 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> (e.g. 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 is similar to spy X except that 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 in Section 2.6.

2.5. Format of Debugging messages

We shall now look at the exact format of the message output by the system at a port. All trace messages are output to the terminal regardless of where the current output 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; i.e. the number of direct ancestors this goal has.

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

The goal is then printed so that you can inspect its current instantiation state. This is done using print (see Section 4.1.3) so that all goals output by the tracing mechanism can be pretty printed if the user desires.

The final "?" is the prompt indicating that you should type in one of the option codes allowed (see 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 trace them. These include all primitive I/O evaluable predicates (e.g. get, put, read, write), all basic control structures (e.g. ',', ';', '->') and all debugging control evaluable predicates (e.g. debug, spy, leash, trace). This means that you will never see messages concerning these predicates during debugging.

2.6. Options available during Debugging

This section describes the particular options that are available when the system prompts you after printing out a debugging message. All the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the terminal with any blanks being completely ignored up to the next terminator (carriage-return, line-feed, or escape). Some options only actually require the terminator; e.g. the creep option, as we have already seen, only requires <cr>.

The only option which you really have to remember is "h" (followed by <cr>). This provides help in the form of the following list of available options.

```
   <cr>  creep                        c     creep
   <lf>  leap                         l     leap
   <esc> skip                         s     skip
    x    back to choice point         q     quasi skip
    r    retry                        r <n> retry goal n
    f    fail                         f <n> fail goal n
    ;    redo
    a    abort                        e     exit from Prolog
    h    help                         p     print goal
    w    write goal                   d     display goal
    g    print ancestor goals         g <n> latest n ancestors
    @    accept command               b     break
    [    consult user                 n     nodebug
```

The first three options are the basic control decisions. Since they are the most frequently used, each may be invoked by a single keystroke.

c
<cr> Creep
causes the interpreter to single-step to the very next port and print a message. Then if the port is leashed (see Section 2.3), 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
<lf> 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
<esc> Skip
is only valid for Call and Redo 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 a ^C 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.

x Back to choice point
The X option gives you the ability to quickly fail back to the last real choice point. When you know that something is going to fail back to some earlier choice point, and all you are interested in is where this point is, then it is rather tedious to follow the execution all the way back. This option is only applicable at Fail and Redo ports; it keeps failing until either a Call port or an Exit port is traversed; this will be just after the choice point. You cannot interact during this time but the system prints the direct path back from where you started. This will be a sequence of Fails followed by a sequence of Redos (either sequence may be empty). These are standard debugging messages (as given earlier) except that the first two characters will be "=>". When you are arrive at the Call (or Exit) port the normal debugging action will occur: you will be reprompted if the port is leashed or if a spy-point is set on the procedure.

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

If you supply an integer after the retry command, then this is taken as specifying an invocation number and the system trys to get you to the Call port, not of the current box, but of the invocation box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts ("!") in your program. If this is the case then the system will not find the required invocation as it backtracks, and it will end up going back too far. When it spots this it will

stop. The result of one of these big jumps will therefore be either to get you back to the invocation you wanted to get to, or to the first actually available invocation before this point. A message "[** JUMP **]" is output to indicate what has occurred.

f Fail
is similar to Retry in that it transfers control to the Fail port of the current (or specified) box. This puts your execution in a position where it is about to backtrack out of the current (or specified) invocation, i.e. you have manually failed the initial goal.

; Redo
can be called at an Exit port to force a move to the Redo port.

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.)

e Exit from Prolog
causes an irreversible exit from the Prolog system back to the Monitor. (This is the same as the evaluable predicate halt.)

h Help
displays the table of options given above.

p Print goal
re-prints the current goal using print

w Write goal
writes the current goal on the terminal using write. This may be useful if your pretty print routine (portray) is not doing what you want.

d Display goal
displays the current goal using display. See Write (above).

g Print ancestor goals
provides you with a list of ancestors to the current goal, i.e. all goals that are hierarchically above the current goal in the calling sequence. It uses the ancestors evaluable predicate (page 42). You can always be sure of jumping to any goal in the ancestor list (by using retry etc). If you supply an integer n, then only that number of ancestors will be printed. That is to say, the last n ancestors will be printed counting back from the current goal.

@ Accept command
gives you the ability to call arbitrary Prolog goals. It is effectively a one-off break (see below). The initial message "| :- " will be output on your terminal, and a command is then read from the terminal and executed as if you were at top level.

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 (^Z) 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.

[Consult user
allows you to insert clauses and then return to where you were. It is the same as "@" followed by "[user].".

n nodebug
switches Debug Mode off. Notice that this is the correct way to switch debugging off at a trace point. You cannot use the "@" or "b" options because they always restore Debug Mode upon return.

2.7. 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.

CHAPTER 3

COMPILING

The DECsystem-10 Prolog compiler [Warren 77] produces compact and efficient code, running 10 to 20 times faster than interpreter code, and requiring much less runtime storage. Compiled Prolog programs are comparable in efficiency with LISP programs for the same task, compiled by current DECsystem-10 LISP compilers [Warren et al. 77]. However, against this, compilation itself is several times slower than "consulting" and most of the debugging aids, such as tracing, are not applicable to compiled code.

3.1. Calling the Compiler

To compile a program, use the evaluable predicate:

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

where Files is either the name of a file (including the ersatz file 'user') or a list of file names. The procedures contained in these files will be compiled. For example:

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

Outwardly, the effect of compile is very much like that of reconsult. If clauses for some predicate appear in more than one file, the later set will effectively overwrite the earlier set, The division of the program into separate files does not imply any module structure - any compiled procedure can call any other.

3.2. Public Declarations

To make a compiled procedure accessible from interpreted code (including directives) it is necessary to declare it to be public, using the command to the compiler:

```
:- public Predicates.
```

where Predicates is either a predicate specification of the form Name/Arity, or a conjunction of such specifications. For example:

```
:- public concatenate/3, member/2, ordered/1, go/0.
```

Public declarations may appear anywhere within a compiled file. It is not necessary for the public declaration to precede the corresponding procedure, nor even for them to be in the same file.

3.3. Mixing Compiled and Interpreted Code

For public predicates, a compiled procedure overwrites any previous interpreted version (cf. reconsult). Similarly, a subsequent reconsult of an interpreted version will overwrite the compiled version.

It is possible to have a compiled procedure with the same name and arity as a quite different interpreted procedure. Provided that the compiled procedure is NOT declared to be public, the two procedures will never interfere with one another: compiled code will use the compiled version while interpreted code will use the interpreted version.

When a compiled version of a public procedure is in force, the interpreted procedure is actually replaced by a clause:

```
P :- incore(P).
```

where P is the most general goal for that predicate, and incore is a standard evaluable predicate (analogous to call) through which are accessible all the compiled procedures for public predicates. Note that one can therefore effectively extend or modify a public procedure using consult, asserts etc., but these changes will only be visible to interpreted code.

There are two ways for compiled code to utilise interpreted procedures:

- If there are no compiled clauses for the predicate, the interpreted procedure is invoked automatically.

- call(P) always calls the interpreter. Note that this implies that if you wish to call compiled procedures via call you must declare them to be public.

To clarify all this, here is an example. First we compile the following file:

```
:- public f/1, g/1.
f(a).
g(X) :- f(X).
g(X) :- h(X).
```

There are no clauses for h/1. Next we consult the following:

```
f(b).
h(c).
```

Now, if we call f we get:

```
?- f(X).
X = a ;
X = b ;
no
```

That is, we use both the compiled and the interpreted clauses for f. However, if we call g:

```
?- g(X).
X = a ;
X = c ;
no
```

then g calls only the compiled version of f so that the solution "X = b" is not found. The second clause for g calls h, and since there are no compiled clauses for h this call is passed to the interpreter which does find a solution ("X = c").

3.4. Mode Declarations

When a program is to be compiled, it is often worthwhile to include mode declarations which inform the compiler that certain procedures will only be used in restricted ways, i.e. that some arguments in the call will always be "input", while others will always be "output". Such information enables the compiler to generate more compact code making better use of runtime storage. The saving of runtime storage in particular can often be very substantial. Mode declarations also help other people to understand how your program operates.

A mode declaration is given by a compiler directive of the form

```
:- mode P(M).
```

where P is the name of a procedure, and M specifies the "modes" of its arguments. M consists of a number of "mode items", separated by commas, one for each argument position of the predicate concerned. A mode item is either '+', '-' or '?'. Mode '+' specifies that the corresponding argument in any call to the procedure will always be instantiated, while mode '-' specifies that the argument will always be uninstantiated. Mode '?' indicates that there is no restriction on the form of the argument; a mode declaration such as

```
:- mode concatenate(?,?,?).
```

is equivalent to omitting the declaration altogether.

If, for example, you know that the first two arguments of the procedure concatenate will always be "input", you can give it the mode declaration:

```
:- mode concatenate(+,+,?).
```

If, in addition, you are prepared to guarantee that the third argument will always be "output", you can strengthen the mode declaration to

```
:- mode concatenate(+,+,-).
```

It is permissible to combine a number of mode declarations into the one command, e.g.

```
:- mode concatenate(+,+,-), member(+,?), ordered(+).
```

To have any effect, a mode declaration must appear before the clauses of the procedure it concerns. What happens when a mode declaration is violated by a procedure call depends on the precise form of the goal and clause head. The call will either succeed as if there was no mode declaration, or cause an error message and fail. Because the precise result is liable to change in future releases of the system, and because it is far too complicated to be discussed here, the user should assume that ALL infringed mode declarations will cause an error message and backtracking.

Mode declarations are ignored by the interpreter.

3.5. Indexing

In contrast to the interpreter, the clauses of a compiled procedure are indexed according to the principal functor of the first argument in the head of the clause. This means that the subset of clauses which match a given goal, as far as the first step of unification is concerned, is found very quickly, in practically constant time (i.e. in a time independent of the number of clauses in the procedure). This can be very important where there is a large number of clauses in a procedure. Indexing also improves the Prolog system's ability to detect determinacy - important for conserving working storage.

3.6. Tail Recursion Optimisation

The compiler incorporates "tail recursion optimisation" to improve the speed and space efficiency of determinate procedures.

When execution reaches the last goal in a clause belonging to some procedure, and provided there are no remaining backtrack points in the execution so far of that procedure, all of the procedure's local working storage is reclaimed BEFORE the final call, and any structures it has created become eligible for garbage collection. This means that programs can now recurse to arbitrary depths without necessarily exceeding core limits. For example:

```
cycle(State) :- transform(State,State1), cycle(State1).
```

where transform is a determinate procedure, can continue executing indefinitely, provided each individual structure, State, is not too large. The procedure cycle is equivalent to an iterative loop in a conventional language.

To take advantage of tail recursion optimisation one must ensure that the Prolog system can recognise that the procedure is determinate at the point where the recursive call takes place. That is, the system must be able to detect that there are no other solutions to the current goal to be found by subsequent backtracking. In general this involves reliance on the DEC-10 Prolog compiler's indexing and/or use of cut.

3.7. Practical Limitations

At present, the space occupied by superseded compiled code is not reclaimed.

Making a predicate public leads to the generation of a significant amount of extra code. Moreover, at present, this code is re-generated every time compile is called, (and the space occupied by the old code is not reclaimed). Therefore it is worthwhile, when possible, to include all the files to be compiled in a single call to compile.

The execution of compiled code cannot be "broken" or "aborted" in the same way as interpreted code. A Break or Abort reply to a ^C interruption only takes effect at the next entry to an interpreted procedure.

You may notice that there is a slight pause on starting and finishing a compilation. This is because the compiler resides in a separate overlay, which has to be swapped in and out.

CHAPTER 4

BUILT-IN PROCEDURES

Built-in procedures are also called evaluable_predicates.

It is not possible to redefine built-in procedures. At present an attempt to do so will give no error message, but the clauses for the redefinition will simply be ignored. It is hoped that this situation will be improved in future so that an appropriate error message is given; in the meantime it is best, when in doubt about the choice of a predicate name, to check with the list of the evaluable predicates given on page 91.

The DECsystem-10 Prolog system provides a wide range of built-in procedures to perform the following tasks:

- Input / Output
    - Reading in Programs
    - File Handling
    - Input and Output of Terms
    - Character Input/Output
- Arithmetic
- Comparison of Terms
- Convenience
- Extra Control
- Information about the State of the Program
- Meta-Logical
- Modification of the Program
- Internal Database
- Sets
- Compiled Program
- Debugging
- Definite Clause Grammars
- Environmental

The following descriptions of the evaluable predicates are grouped according to the above categorisation of their tasks.

4.1. Input / Output

A total of fourteen I/O streams may be open at any one time for input and output. An extra stream is 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. Terminal output is only displayed after a newline is written or ttyflush is called. When the program is waiting for input from the terminal, the default prompt is "|: ".

When the current input (or output) stream is closed, the user's terminal becomes the current input (or output) stream. No file other than the ersatz file 'user', can be simultaneously open for input and output.

A file is referred to by its name written as an atom, i.e. it must be surrounded by single quotes if it is not already a legal atom. e.g.

```
myfile
'123'
'DATA.LST'
'DTA1:ABC.PL'
```

It is NOT possible to refer to a file by a name exceeding six letters in length (excluding the device and extension); i.e. there is no automatic truncation of names.

Path specifications are not understood, i.e. you cannot specify a user number with a file name. However, device names are understood, and if you are using TOPS-10 version 7.01 you will find that using logical names is a good fix - see the PATH system command.

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

End of file is signalled by a ^Z (Control and Z, ASCII code 26) character. Any more input requests for a file whose end has been reached causes an error failure. ^Z typed at the terminal causes the equivalent condition for the ersatz file 'user'.

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

[File|Files]
This is a shorthand way of consulting or reconsulting a list of files. (The case where there is just one file name in the list was described in Section 1.2.) 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).
```

4.1.2. 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 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.

rename(F,N) If file F is currently open, it is closed and renamed to N. If N is '[]', the file is deleted.

log Enables the logging of terminal interaction to file 'prolog.log'. It is the default.

nolog Disables the logging of terminal interaction.

4.1.2.1. An Example

Here is an example of a common form of file processing:

```
    process_file(F) :-
            see(F),                 % Open file F
            repeat,
              read(T),              % Read a term
              process_term(T),      % Process it
            T = end_of_file,        % Loop back if not at end of file
            seen.                   % CLOSE the file
```

4.1.3. Input and Output of Terms

read(X) The next term, delimited by a full-stop (i.e. a "." followed by either a space or a control character), is read from the current input stream and unified with X. The syntax of the term must agree 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 term '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 current operator declarations.

display(X) The term X is displayed on the terminal (which is not necessarily the current output stream) 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 output 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 called recursively on the components of Term, unless Term is atomic in which case it is written via write.

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

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

4.1.4. Character Input/Output

In character input, the sequence <cr><lf> (carriage-return,line-feed) is read in as the single character newline, ASCII code 31. This is reversed on output, where putting newline inserts a <cr><lf> sequence in the output stream (but see ttyput below if you don't want this conversion to take place on output).

There are two sets of character I/O predicates. The first set uses the current input and output streams, while the second always uses the terminal.

nl — A new line is started on the current output stream.

get0(N) — N is the ASCII code of the next character from the current input stream.

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

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

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

tab(N) — N spaces are output to the current output stream. N may be an integer expression.

The above predicates are the ones which are the most commonly used, as they can refer either to files or to the user's terminal. In most cases these predicates are sufficient, but there is one limitation: if you are outputting to the terminal via put then nothing is output until such time as you put a newline character. If this line by line output is inadequate, you have to use ttyflush (see below).

The predicates which follow always refer to the terminal. They are convenient for writing interactive programs which also perform file I/O.

ttynl A new line is started on the terminal and the buffer is flushed.

ttyflush Flushes the terminal output buffer. Output to the terminal, using either ttyput or put, normally simply goes into an output buffer until such time as a newline is output. Calling this predicate forces any characters in this buffer to be output immediately.

ttyget0(N) N is the ASCII code of the next character input from the terminal.

ttyget(N) N is the ASCII code of the next non-blank printable character from the terminal.

ttyskip(N) Skips to just past the next ASCII character code N from the terminal. N may be an integer expression.

ttyput(N) The ASCII character code N is output to the terminal. N may be an integer expression.

This predicate can also be used (under TOPS-10 only) for image mode output which is useful for driving graphics terminals: if N>127 or N<0, then the 8 low order bits of N are output to the terminal in image mode, bypassing the log file, and without buffering. Thus the following call will output all characters C without altering or logging them.

```
ttyput(8'300000+C)
```

In particular, this mode of calling prevents the normal substitution of <cr><lf> for the newline character.

4.2. Arithmetic

Arithmetic is performed by built-in procedures which take as arguments integer expressions and evaluate them. An integer expression is a term built from integers and variables using functors which have an arithmetical meaning. At the time of evaluation, each variable in an integer expression must be bound to an integer.

Although Prolog integers must be in the range -2^17 to 2^17-1, the integers in arguments to arithmetic procedures and the intermediate results of the evaluation may range from -2^35 to 2^35-1.

Only certain functors are permitted in an integer expression. These are listed below, together with an indication of their meanings. X and Y are assumed to be integer expressions.

X+Y	integer addition
X-Y	integer subtraction
X*Y	integer multiplication
X/Y	integer division
X mod Y	X modulo Y
-X	unary minus
X/\Y	bitwise conjunction
X\/Y	bitwise disjunction
\(X)	bitwise negation
X<<Y	bitwise left shift of X by Y places
X>>Y	bitwise right shift of X by Y places
!(X)	the number in the range 0 to 2^18-1 which is equal to X modulo 2^18
$(X)	the number in the range -2^17 to 2^17-1 which is equal to X modulo 2^18
[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.

In interpreted code ONLY, variables in an integer expression which is to be evaluated may be bound to other integer expressions rather than just integers, e.g.

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

    ?- evaluate(24*9,Ans).
```

This does NOT work for compiled code (a warning message will be output if it is attempted, and zero will be taken as the value of the variable).

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

Z is X — Integer expression X is evaluated and the result, reduced modulo 2^18 to a number in the range -2^17 to 2^17-1, is unified with Z. Fails if X is not an integer 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.

4.3. Comparison of Terms

These evaluable predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should NOT be used when what you really want is arithmetic comparison (Section 4.2) or unification.

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

- variables, in a standard order (roughly, oldest first - the order is NOT related to the names of variables);
- integers, 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 40).

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.)

4.4. Convenience

P , Q   P and Q.

P ; Q   P or Q.

true   Always succeeds.

fail   Always fails.

X = Y   Defined as if by the clause " Z=Z. "; i.e. X and Y are unified.

length(L,N) L must be instantiated to a list of determinate length. This length is unified with N.

4.5. Extra Control

! See Section I.3.

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

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

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

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

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

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

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

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

is also correct.

P -> Q ; R Analogous to

```
"if P then Q else R"
```

i.e. defined as if by

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

Not yet available for compiled code.

P -> Q When occurring other than as one of the alternatives of a disjunction, is equivalent to

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

Not yet available for compiled code.

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

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

4.6. Information about the State of the Program

listing
: Lists in the current output stream all the clauses in the current interpreted program. Clauses listed to a file can be consulted back.

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

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

numbervars(X,N,M)
: Unifies each of the variables in term X with a special term, so that write(X) (or writeq(X)) prints those variables as ("A" + (i mod 26))(i/26) where i ranges from N to M-1. N must be instantiated to an integer. If it is 0 you get the variable names A, B, .., Z, A1, B1, etc. This predicate is used by listing.

ancestors(L)
: Unifies L with a list of ancestor goals for the current clause. The list starts with the parent goal and ends with the most recent ancestor coming from a call in a compiled clause. The list is printed using print and each entry is preceded by the invocation number in parentheses followed by the depth number (as would be given in a trace message). If the invocation does not have a number (this will occur if Debug Mode was not switched on until further into the execution) then this is marked by "-".

  Not available for compiled code.

subgoal_of(S)
: Equivalent to the sequence of goals:

```
ancestors(L), member(S,L)
```

  where the predicate member (not an evaluable predicate) successively matches its first argument with each of the elements of its second argument. (See page 4 for a definition of member.)

  Not available for compiled code.

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 currently exists an interpreted procedure.

4.7. Meta-Logical

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

```
:- var(foo(X,Y)).
```

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

nonvar(X) Tests whether X is currently instantiated. This is the opposite of var.

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

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

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

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, an integer. Initially, either T must be instantiated, 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 uninstantiated, the result of the call is to instantiate T to the most general term having the principal functor indicated.

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

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

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

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

product =.. [product]
```

If X is uninstantiated, 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 an integer.

name(X,L) If X is an atom or integer 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((:-),[58,45])
```

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

```
| ?- name(X,[58,45]).

X = :-

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

X = :-
```

call(X) If X is instantiated to a term which would be acceptable as the body of a clause, then the goal call(X) is executed exactly as if that term appeared textually in its place, except that any cut ("!") occurring in X only cuts alternatives in the execution of X.

If X is not instantiated as described above, an error message is printed and call fails. If X contains calls of compiled procedures then those procedures must be declared to be public (see page 23).

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

4.8. 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").

assert(C) The current instance of C is interpreted as a clause and is added to the current interpreted 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.

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

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

clause(P,Q) P must be bound to a non-variable term, and the current interpreted 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'.

retract(C) The first clause in the current interpreted program that matches C is erased. C must be initially instantiated; it is first translated into a clause, which is then matched to the database. This means that

```
retract((p(X):-Y))
```

matches only clauses whose body is call(Y), and not all clauses for p(X). The predicate may be used in a non-determinate fashion, i.e. it will successively retract clauses matching the argument through backtracking.

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

abolish(Name,Arity)
Effectively removes the procedure, either interpreted or public compiled, for the predicate specified by Name and Arity.

4.9. Internal Database

The predicates described in this section are primarily concerned with providing efficient means of performing operations on large quantities of data. Most users will not need to know about these predicates.

These predicates make it possible to store arbitrary terms in the database without interfering with the clauses which make up the program. The terms which are stored in this way can subsequently be retrieved via the key on which they were stored. Many terms may be stored on the same key, and they can be individually accessed by pattern matching.

Alternatively, access can be achieved via a special identifier which uniquely identifies each recorded term and which is returned when the term is stored. This special identifier is actually a pointer into the database and needs to be treated with some caution. For safety reasons it is not possible to store the pointers themselves in the database: the term to which the pointer referred might be erased.

Note the difference between this facility and that provided by assert and related predicates: the latter actually alter the running program. Also the recording predicates use an extra level of indirection, the Key, which allows greater flexibility.

recorded(Key,Term,Ref)
: The internal database is searched for terms recorded under the key Key. These terms are successively unified with Term in the order they occur in the database. At the same time, Ref is unified with the implementation-defined identifier uniquely identifying the recorded item. The key must be given, and may be an atom, integer 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 implementation-defined identifier. 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 implementation-defined identifier. The key must be given, and only its principal functor is significant.

erase(Ref)
: The recorded item (or interpreted clause - see assert/2 etc. below) whose implementation-defined identifier is Ref is effectively erased from the internal database or interpreted program.

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

Like recorded terms, the clauses of an interpreted program also have a unique implementation-defined identifier. A new set of the predicates described in Section 4.8 is given below, each predicate having an additional argument which is this identifier. This identifier makes it possible to access clauses directly instead of requiring a normal database (hash-table) lookup. However it should be stressed that use of these predicates requires some extra care.

assert(Clause,Ref)
: Equivalent to assert/1 where Ref is the implementation-defined identifier of the clause asserted.

asserta(Clause,Ref)
: Equivalent to asserta/1 where Ref is the implementation-defined identifier of the clause asserted.

assertz(Clause,Ref)
: Equivalent to assertz/1 where Ref is the implementation-defined identifier of the clause asserted.

clause(Head,Body,Ref)
: Equivalent to clause/2 where Ref is the implementation-defined term which uniquely identifies 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 identifier of the clause is known or unknown.

4.10. Sets

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

setof(X,P,S)

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

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

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

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

might produce two alternative solutions via backtracking:

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

The call:

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

would then produce:

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

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

```
Y^Q
```

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

For example:

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

would produce the single result:

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

in contrast to the earlier example.

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

X^P
The interpreter recognises this as meaning "there exists an X such that P is true", and treats it as equivalent to call(P). The use of this explicit existential quantifier outside the setof and bagof constructs is superfluous, and therefore is not recognised by the compiler.

4.11. Compiled Program

compile(Files)

Compiles the file or list of files specified by Files. (See Chapter 3.)

incore(Goal)

The term Goal is interpreted as a call to a current public compiled procedure.

revive(Name,Arity)

If the predicate specified by Name and Arity is public, the current interpreted procedure (if any) for that predicate is replaced by the most recent compiled version. Otherwise the call has no effect.

4.12. Debugging

unknown(OldState,NewState)
Unifies OldState with the current state of the "Action on unknown procedures" flag, and sets the flag to NewState. This flag determines whether or not the system is to catch calls to predicates which have no clauses (see Section 1.6). The possible states of the flag are:

- 'trace', which causes calls to predicates with no clauses to be reported and the debugging system to be entered at the earliest opportunity;
- 'fail', which causes calls to such predicates to fail (the default state).

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.

trace
Debug Mode is switched on, and an immediate Creep decision is made, so that tracing will start with the very next port through which control passes. Since this is a once-off decision, a call to trace is necessary whenever tracing is required right from the start of an execution. (The assumed decision is otherwise Leap.)

leash(Mode)
Leashing Mode is set to Mode. Leashing Mode determines the ports of procedure boxes at which you are to be prompted when you Creep through your program. At unleashed ports a tracing message is still output, but program execution does not stop to allow user interaction. Note that the ports of spy-points are always leashed (and cannot be unleashed). Mode can be one of the following:

```
full  - prompt on Call, Exit, Redo and Fail
tight - prompt on Call, Redo and Fail
half  - prompt on Call and Redo
loose - prompt on Call
off   - never prompt
```

or Mode can be an integer between 0 and 15 which will set any arbitrary combination not covered above: treat the integer as a binary number 2'CERF, where C, E, R and F give the yes/no (1/0) decisions for the Call, Exit, Redo and Fail ports respectively.

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

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

debugging
: Outputs information concerning the status of the debugging package. This will show:

1. whether unknown procedures are being trapped
2. whether Debug Mode is on, and if it is -
    a. what spy-points have been set
    b. what mode of leashing is in force.

4.13. Definite Clause Grammars

Prolog's grammar__rules provide a convenient notation for expressing definite clause__grammars [Colmerauer 75] [Pereira & Warren 80]. Definite clause grammars are an extension of the well-known context-free grammars. A grammar rule in Prolog takes the general form

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

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

Definite clause grammars extend context-free grammars in the following ways:

1. A non-terminal symbol may be any Prolog term (other than a variable or integer).

2. A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list []. If the terminal symbols are ASCII character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, [] or "".

3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in {} brackets.

4. The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).

5. Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator ';' as in Prolog.

6. The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in {} brackets.

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

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

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

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

In the last rule, C is the ASCII code of some digit.

The question

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

will compute Z=14. The two extra arguments are explained below.

Now, in fact, grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. We now show how to translate grammar rules into ordinary clauses by making explicit the extra arguments.

A rule such as

```
p(X) --> q(X).
```

translates into

```
p(X,S0,S) :- q(X,S0,S).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X,Y) --> q(X), r(X,Y), s(Y).
```

then corresponding input and output arguments are identified, as in

```
p(X,Y,S0,S) :- q(X,S0,S1), r(X,Y,S1,S2), r(Y,S2,S).
```

Terminals are translated using the evaluable predicate 'C'(S1,X,S2), read as "point S1 is connected by terminal X to point S2", and defined by the single clause

```
'C'([X|S],X,S).
```

(This predicate is not normally useful in itself; it has been given the name upper-case "c" simply to avoid using up a more useful name.) Then, for instance

```
p(X) --> [go,to], q(X), [stop].
```

is translated by

```
p(X,S0,S) :-
    c(S0,go,S1), c(S1,to,S2), q(X,S2,S3), c(S3,stop,S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves, e.g.

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X,S0,S) :- c(S0,X,S1), integer(X), X>0, q(X,S1,S).
```

Similarly, a cut is translated literally.

Terminals on the left-hand side of a rule translate into an explicit list in the output argument of the main non-terminal, e.g.

```
is(N), [not] --> [aint].
```

becomes

```
is(N,S0,[not|S]) :- c(S0,aint,S).
```

Disjunction has a fairly obvious translation, e.g.

```
args(X,Y) --> dir(X), [to], indir(Y);  indir(Y), dir(X).
```

translates to

```
args(X,Y,S0,S) :-
    dir(X,S0,S1), c(S1,to,S2), indir(Y,S2,S);
    indir(Y,S0,S1), dir(X,S1,S).
```

The built-in procedures which are concerned with grammars are as follows.

expand_term(T1,T2)
: When a program is read in, some of the terms read are transformed before being stored as clauses. If T1 is a term that can be transformed, T2 is the result. Otherwise T2 is just T1 unchanged. This transformation takes place automatically when grammar rules are read in, but sometimes it is useful to be able to perform it explicitly.

phrase(P,L)
: The list L is a phrase of type P (according to the current grammar rules), where P is either a non-terminal or more generally a grammar rule body. P must be non-variable.

'C'(S1,terminal,S2)
: Not normally of direct use to the user, this evaluable predicate is used in the expansion of grammar rules (see above). It is defined by the clause:

```
'C'([X|S],X,S).
```

Note that "%(" and "%)" are allowed as alternatives to "{" and "}" respectively for the benefit of users whose keyboards do not have curly brackets.

4.14. Environmental

halt
: Causes an irreversible exit from Prolog back to the Monitor.

'NOLC'
: Establishes the "no lower-case" convention described in Section III.2.

'LC'
: Establishes the "full character set" convention described in Section III.2. It is the default setting.

op(Priority,Type,Name)
: Declares the atom Name to be an operator of the stated Type and Priority (refer to Section I.4). Name may also be a list of atoms in which case all of them are declared to be operators. If Priority is 0 then the operator properties of Name (if any) are cancelled.

current_op(Precedence,Type,Op)
: The atom Op is currently an operator of type Type and precedence Precedence. Neither Op nor the other arguments need be instantiated at the time of the call; i.e. this predicate can be used to generate as well as to test.

break
: Causes the current execution to be interrupted at the next interpreted 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 ^Z. 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. Refer to Section 1.9.

save(F)
: The system saves the current state of the system into file F. Refer to Section 1.10.

save(F,Return)
: Saves the current system state in F just as save(F), but in addition unifies Return to 0 or 1 depending on whether the return from the call occurs in the original incarnation of the state or through a call restore(F) (respectively).

restore(F)
: The system is returned to the system state previously saved to file F. Refer to Section 1.10.

reinitialise
: This predicate can be used to force the initialisation behaviour described in Section 1.11 to take place at any time. That is, if there is a 'prolog.bin' in the user's directory it is restored. Otherwise, if there is a 'prolog.ini' it is consulted.

maxdepth(D)
: Positive integer D specifies the maximum depth, i.e. the maximum number of nested interpreted calls, beyond which the interpreter will induce an automatic failure. Top level has zero depth. This is useful for guarding against loops in an untested program, or for curtailing infinite execution branches. Note that calls to compiled procedures are not included in the computation of the depth.

depth(D)
: Unifies D with the current depth, i.e. the number of currently active interpreted procedure calls.

gcguide(Function,Old,New)
: Function can be 'margin', which is the number of free pages (1 page = 512 words) below which garbage collection is always tried, or 'cost', which is the percentage of runtime the user accepts to be consumed on garbage collections. The 'margin' value is initially 50 pages, the 'cost' 10 percent. When the procedure is called, the current value corresponding to Function is unified with Old, and New is stored as the new value. If New is not an integer or it is out of range, the call fails.

gc
: Enables garbage collection of the global stack (the default).

nogc
: Disables garbage collection of the global stack.

trimcore
: Reduces free space on the stacks and trail as much as possible and releases store no longer needed. The interpreter automatically calls trimcore after each directive at top-level, after an abort and after a (re)consult.

statistics Display on the terminal statistics relating to memory usage, run time, garbage collection of the global stack and stack shifts.

statistics(K,V)
This allows a program to gather various execution statistics. For each of the possible keys K, V is unified with a list of values, as follows:

Key	Values		
core	low-segment	high-segment	
heap	size	free	
global_stack	"	"	
local_stack	"	"	
trail	"	"	
runtime	since start of Prolog	since previous statistics	
garbage_collection	no. of GCs	words freed	time spent
stack_shifts	no. of local shifts	no. of trail shifts	time spent

Times are in milliseconds, sizes of areas in words. If a time exceeds 129.071 sec., it will be returned as a term of the form

```
xwd(T1,T2)
```

representing

```
T1*2^18 + T2 mod 2^18
```

Note that if such a term occurs in an interpreted arithmetic expression, it will be evaluated correctly.

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 predicate only affects the prompt given when a user's program is trying to read from the terminal (e.g. by calling read). Note also that the prompt is reset to the default '|: ' on return to top-level.

version
Displays the introductory messages for all the component parts of the current system.

Prolog will display its own introductory message when initially run but not normally at any time after this. If this message is required at some other time it can be obtained using this predicate which displays a list of introductory messages; initially this list comprises only one message (Prolog's), but you can add more messages using version/1.

version(Mesg)
This takes a message, in the form of an atom, as its argument and appends it to the end of the message list which is output by version/0.

The idea of this message list is that, as systems are constructed on top of other systems, each can add its own identification to the message list. Thus version/0 should always indicate which modules make up a particular package. It is not possible to remove messages from the list.

plsys(Term) Allows certain interactions with the operating system. On TOPS-10 systems the possible forms of Term are as follows:

core_image
Prepares a core image of the current Prolog state which can be saved with the "save" Monitor command and later run with the "run" Monitor command. For example,

```
| ?- plsys(core_image),
|    display('My Program'), ttynl,
|    reinitialise.
```

At this point the following message is output:

```
[ core image ready - save it with a 'SAVE' command ]
```

and Prolog exits to the Monitor. Now you should type something like:

```
.save myprog
```

This saves your program image in the file MYPROG.EXE. Subsequently, the Monitor command

```
.run myprog
```

causes your saved image to start up just like a normal entry of Prolog except that your own message is displayed on the terminal.

run(Program,Offset)
Runs the program in file Program, starting at offset Offset. Program should be an atom and Offset an integer, e.g.

```
plsys(run('sys:direct',0))
```

tmpcor(IO,TmpFile,TmpData)
Will read/write tmpcor files. IO can be one of {see,tell} but currently only tell (writing tmpcor files) is implemented. TmpFile is an atom with a 3 character name and TmpData a list of ASCII character codes (written easily as a double quoted string "<characters>"). If IO = tell then the ASCII characters are written to the appropriate tmpcor file. If the tmpcor file cannot be set up then a disk file jjjnnn.TMP is used in the usual way (jjj is your job number, and nnn = TmpFile). This evaluable predicate is intended for use with plsys(run(_,_)) since many programs can be given startup instructions via tmpcor files when started at their CCL entry point - i.e. offset 1. e.g.

```
plsys(tmpcor(tell,'edt',"foobaz.pl$")),
plsys(run(teco,1)).
```

APPENDIX I

THE PROLOG LANGUAGE

This appendix provides a brief introduction to the syntax and semantics of a certain subset of logic ("definite clauses", also known as "Horn clauses"), and indicates how this subset forms the basis of Prolog. A much fuller introduction to Prolog may be found in [Clocksin & Mellish 81]. For a more general introduction to the field of Logic Programming see [Kowalski 79].

I.1. Syntax, Terminology and Informal Semantics

I.1.1. Terms

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

The constants include integers such as

```
0   1   999   -512
```

Integers are restricted to the range -2^17 to 2^17-1, i.e. -131072 to 131071. Besides the usual decimal, or base 10, notation, integers may also be written in any base from 2 to 9, of which base 2 (binary) and base 8 (octal) are probably the most useful. E.g.

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

all represent the integer fifteen.

Constants also include atoms such as

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

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

1. Any sequence of alphanumeric characters (including "_"), starting with a lower case letter.

2. Any sequence from the following set of characters: `` +-*/\^<>=`~:.?@#$& ``

3. Any sequence of characters delimited by single quotes. If the single quote character is included in the sequence it must be written twice, e.g. 'can''t'.

4. Any of: ! ; [] {}
   Note that the bracket pairs are special: '[]' is an atom but '[' is not. However, when they are used as functors (see below) the forms [X] and {X} are allowed as alternatives to '[]'(X) and '{}'(X) respectively. Note also that '%(' and '%)' are allowed as synonyms

for '{' and '}' respectively.

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

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

If a variable is only referred to once, 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. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writeable storage location as in most programming languages; rather it is a local name for some data object, cf. the variable of pure LISP and identity declarations in Algol68.

The structured data objects of the language are the compound terms. A compound term comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is 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

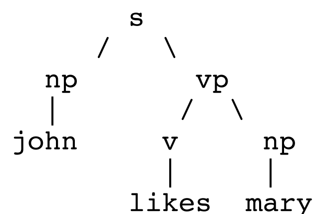
```
point(X,Y,Z)
```

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

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

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

would be pictured as the structure



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

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

as optional alternatives to

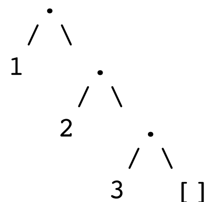
```
+(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)
```

The use of operators is described fully in Section I.4 below.

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

```
[]
```

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



which could be written, using the standard syntax, as

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

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

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

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

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

representing

respectively.

Note that this notation does not add any new power to the language; it simply makes it more readable. E.g. the above examples could equally be written

```
.(X,L)    .(a,.(b,L))
```

The atom ',..' may be used instead of '|' in the list notation above, e.g.

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

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. E.g.

    "DECsystem-10"

which represents exactly the same list as

    [68,69,67,115,121,115,116,101,109,45,49,48]

I.1.2. Programs

A fundamental unit of a logic program is the goal or procedure_call. E.g.

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

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The (principal) functor of a goal 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 sentences, which are analogous to sentences of natural language. A sentence comprises a head and a body. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (i.e. it too may be empty). If the head is not empty, the sentence is called a clause.

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

    P.

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

    "P is true."

and procedurally as

    "Goal P is satisfied."

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

    P :- Q, R, S.

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

    "P is true if Q and R and S are true."

or procedurally as

    "To satisfy goal P, satisfy goals Q, R and S.

A sentence with an empty head is called a directive (see also Section 1.4), of which the most important kind is called a question and is written in the form

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

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

"Are P and Q true?"

and procedurally as

"Satisfy goals P and Q."

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

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

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

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

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

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

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

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

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

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

In any program, the 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 well 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; e.g. concatenate/3.

Certain predicates are predefined by built-in_procedures supplied by the Prolog system. Such predicates are called evaluable_predicates.

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

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

which can be read as

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

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

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

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

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

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

is read as if it were

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

Note the double use of the "." character. On the one hand it is used as a sentence terminator, while on the other it may be used in a string of symbols which make up an atom (e.g. the list functor '.'). The rule used to disambiguate terms is that a "." followed by a layout-character is regarded as a sentence terminator, where a layout-character is defined to be any character less than or equal to ASCII 32 (this includes space, tab, newline and all control characters).

I.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 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.

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the procedural_semantics which Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute 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 [Robinson 1965] 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, i.e. 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 expressed by the question

```
?- 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 = []
```

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

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

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

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

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

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

I.2.1. Occur Check

Prolog's unification does not have an "occur 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 either cause a loop or the fatal error "? pushdown list overflow".

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 (e.g. what would a user of Fortran feel if an assignment could take an unbounded amount of 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.

I.3. The Cut Symbol

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

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

E.g.

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

This procedure can be used to test whether a given term is in a list. e.g.

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

returns the answer "yes". The 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.

```
    (2)         x :- p, !, q.
                x :- r.
```

This is equivalent to

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

in an Algol-like language.

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

I.4. Operators

Operators in Prolog are simply a NOTATIONAL CONVENIENCE. For example, the expression

```
2 + 1
```

could also be written +(2,1). This expression represents the data structure

```
   +
  / \
 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 4.2).

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 it is the operator with the HIGHEST precedence that is the principal functor. Thus if '+' has a higher precedence than '/', then

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

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

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

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

```
xfx     xfy     yfx
```

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

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

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

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

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

It is possible to have more than one operator of the same name, so long as they are of different kinds, i.e. infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to operators which are provided as standard. Declarations of all the standard operators can be found on page 94.

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

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

so that

```
a-b+c
```

is valid syntax, and means

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

i.e.

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

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

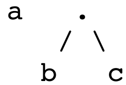
```
:-op(900, xfy, '.').
```

Then

```
a.b.c
```

would represent the structure

```
    .
   / \
```



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

The possible types for a prefix operator are

```
fx      fy
```

and for a postfix operator they are

```
xf      yf
```

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

```
not not P
```

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

```
not P
```

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

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

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

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

I.5. Syntax Restrictions

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

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

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

   is invalid syntax.

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

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

(where ':-' is the prefix operator) is invalid syntax. The system would try to interpret it as the structure:

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

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

In contrast, the term:

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

is valid syntax and represents the following structure.

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

3. 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 = (?-)
```

I.6. Comments

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

1. The character "%" followed by any sequence of characters up to end of line. The first character after the "%" may not be a "(" or a ")", because the symbols "%(" and "%)" are reserved.

2. The symbol "/*" followed by any sequence of characters (including new lines) up to "*/".

I.7. Full Prolog Syntax

A Prolog program consists of a sequence of sentences. Each sentence is a Prolog term. How terms are interpreted as sentences is defined in Section I.7.2 below. Note that a term representing a sentence may be written in any of its equivalent syntactic forms. For example, the 2-ary functor ':-' could be written in standard prefix notation instead of as the usual infix operator.

Terms are written as sequences of tokens. Tokens are sequences of characters which are treated as separate symbols. Tokens include the symbols for variables, constants and functors, as well as punctuation characters such as brackets and commas.

Section I.7.3 below defines how lists of tokens are interpreted as terms. Each list of tokens which is read in (for interpretation as a term or sentence) has to be terminated by a full-stop token. Two tokens must be separated by a space token if they could otherwise be interpreted as a single token. Both space tokens and comment tokens are ignored when interpreting the token list as a term. A comment may appear at any point in a token list (separated from other tokens by spaces where necessary).

Section I.7.4 below defines how tokens are represented as strings of characters. But first Section I.7.1 describes the notation used in the formal definition of Prolog syntax.

I.7.1. Notation

1. Syntactic categories (or "non-terminals") are always underlined. Depending on the section, a category may represent a class of either terms, token lists, or character strings.

2. A syntactic rule takes the general form

   ```
   C --> F1 | F2 | F3
   ```

   which states that an entity of category C may take any of the alternative forms F1, F2, F3, etc.

3. Certain definitions and restrictions are given in ordinary English, enclosed in { } brackets.

4. A category written as C... denotes a sequence of one or more Cs.

5. A category written as ?C denotes an optional C. Therefore ?C... denotes a sequence of zero or more Cs.

6. A few syntactic categories have names with arguments, and rules in which they appear may contain meta-variables in the form of underlined capital letters. The meaning of such rules should be

clear from analogy with the definite clause grammars described in Chapter 4.13.

7. In Section I.7.3, particular tokens of the category name are written as quoted atoms, while tokens which are individual punctuation characters are written literally. To avoid confusion, the punctuation character '|' is written underlined.

I.7.2. Syntax of Sentences as Terms

```
sentence          --> clause | directive | grammar-rule

clause            --> non-unit-clause | unit-clause

directive         --> command | question | file-list

non-unit-clause   --> ( head :- goals )

unit-clause       --> head
                         { where head is not otherwise a sentence }

command           --> ( :- goals )

question          --> ( ?- goals )

file-list         --> list

head              --> term
                         { where term is not an integer or variable }

goals             --> ( goals , goals )
                    |  ( goals ; goals )
                    |  goal

goal              --> term
                         { where term is not an integer
                           and is not otherwise a goals }

grammar-rule      --> ( gr-head --> gr-body )

gr-head           --> non-terminal
                    |  ( non-terminal , terminals )

gr-body           --> ( gr-body , gr-body )
                    |  ( gr-body ; gr-body )
                    |  non-terminal
                    |  terminals
                    |  gr-condition

non-terminal      --> term
                         { where term is not an integer or variable
                           and is not otherwise a gr-body }
```

```
terminals          --> list | string

gr-condition       --> { goals }
```

I.7.3. Syntax of Terms as Tokens

```
term-read-in       --> subterm(1200) full-stop

subterm(N)         --> term(M)
                          { where M is less than or equal to N }

term(N)            --> op(N,fx)
                     |  op(N,fy)
                     |  op(N,fx) subterm(N-1)
                          { except the case '-' number }
                          { if subterm starts with a '(',
                            op must be followed by a space }
                     |  op(N,fy) subterm(N)
                          { if subterm starts with a '(',
                            op must be followed by a space }
                     |  subterm(N-1) op(N,xfx) subterm(N-1)
                     |  subterm(N-1) op(N,xfy) subterm(N)
                     |  subterm(N) op(N,yfx) subterm(N-1)
                     |  subterm(N-1) op(N,xf)
                     |  subterm(N) op(N,yf)

term(1000)         --> subterm(999) , subterm(1000)

term(0)            --> functor ( arguments )
                          { provided there is no space between
                            the functor and the '(' }
                     |  ( subterm(1200) )
                     |  { subterm(1200) }
                     |  list
                     |  string
                     |  constant
                     |  variable

op(N,T)            --> functor
                          { where functor has been declared as an
                            operator of type T and precedence N }

arguments          --> subterm(999)
                     |  subterm(999) , arguments

list               --> '[]'
                     |  [ listexpr ]

listexpr           --> subterm(999)
                     |  subterm(999) , listexpr
```

```
                    |  subterm(999) | subterm(999)
                    |  '..' subterm(999)

constant           --> atom | integer

atom               --> name
                          { where name is not a prefix operator }

integer            --> number
                    |  '-' number

functor            --> name
```

I.7.4. Syntax of Tokens as Character Strings

```
token              --> name
                    |  number
                    |  variable
                    |  string
                    |  punctuation-char
                    |  decorated-bracket
                    |  space
                    |  comment
                    |  full-stop

name               --> quoted-name
                    |  word
                    |  symbol
                    |  solo-char
                    |  []
                    |  {}

quoted-name        --> ' quoted-item... '

quoted-item        --> char  { other than ' }
                    |  ''

word               --> capital-letter ?alpha...
                          { in the 'NOLC' convention only }

word               --> small-letter ?alpha...

symbol             --> symbol-char...
                          { except in the case of a full-stop
                            or where the first 2 chars are /* }

number             --> digit...
                    |  digit ' digit...

variable           --> underline ?alpha...
```

```
variable          --> capital-letter ?alpha..
                         { in the 'LC' convention only }

string            --> " ?string-item... "

string-item       --> char  { other than " }
                   |  ""

decorated-bracket --> %(
                   |  %)

space             --> layout-char...

comment           --> /* ?char... */
                         { where ?char... must not contain */ }
                   |  % rest-of-line

rest-of-line      --> newline
                   |  not-parenthesis ?not-end-of-line... newline

not-parenthesis   --> { any character not in the list
                          ()newline }

not-end-of-line   --> { any character except newline {

newline           --> { ASCII code 31 }

full-stop         --> . layout-char

char              --> { any ASCII character, i.e. }
                      layout-char
                   |  alpha
                   |  symbol-char
                   |  solo-char
                   |  punctuation-char
                   |  quote-char

layout-char       --> { any ASCII character code up to 32,
                        includes <blank>, <cr> and <lf> }

alpha             --> letter | digit | underline

letter            --> capital-letter | small-letter

capital-letter    --> { any character from the list
                          ABCDEFGHIJKLMNOPQRSTUVWXYZ }

small-letter      --> { any character from the list
                          abcdefghijklmnopqrstuvwxyz }

digit             --> { any character from the list
                          012346789 }

symbol-char       --> { any character from the list
                          +-*/\^<>=`~:.?@#$& }
```

```
solo-char          --> { any character from the list
                              ;!% }

punctuation-char   --> { any character from the list
                              ()[]{},| }

quote-char         --> { any character from the list
                              '" }

underline          --> { the character _ }
```

I.7.5. Notes

1. The expression of precedence 1000 (i.e. belonging to syntactic category term(1000)) which is written

   ```
   X,Y
   ```

   denotes the term ','(X,Y) in standard syntax.

2. The bracketed expression (belonging to syntactic category term(0))

   ```
   (X)
   ```

   denotes simply the term X.

3. The curly-bracketed expression (belonging to syntactic category term(0))

   ```
   {X}
   ```

   denotes the term '{}'(X) in standard syntax.

4. The decorated brackets "%(" and "%)" are alternatives for the curly brackets "{" and "}" respectively. e.g.

   ```
   {X} = %(X%)
   ```

5. The character sequence ",.." is allowed as an alternative to "|" in lists, e.g.

   ```
   [X,..L] = [X|L]
   ```

6. Note that, for example, "-3" denotes an integer whereas "-(3)" denotes a compound term which has the 1-ary functor '-' as its principal functor.

7. The character " within a string must be written duplicated. Similarly for the character ' within a quoted atom.

APPENDIX II

PROGRAMMING EXAMPLES

Some simple examples of Prolog programming are given below. For clarity, the example programs are marked with a vertical bar in the left margin.

II.1. Simple List Processing

The goal concatenate(L1,L2,L3) is true if list L3 consists of the elements of list L1 concatenated with the elements of list L2. The goal member(X,L) is true if X is one of the elements of list L. The goal reverse(L1,L2) is true if list L2 consists of the elements of list L1 in reverse order.

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

II.2. A Small Database

The goal descendant(X,Y) is true if Y is a descendant of X.

```
| descendant(X,Y) :- offspring(X,Y).
| descendant(X,Z) :- offspring(X,Y), descendant(Y,Z).
|
| offspring(abraham,ishmael).
| offspring(abraham,isaac).
| offspring(isaac,esau).
| offspring(isaac,jacob).
```

If for example the question

```
?- descendant(abraham,X).
```

is executed, Prolog's backtracking results in different descendants of Abraham being returned as successive instances of the variable X, i.e.

```
X = ishmael
X = isaac
X = esau
X = jacob
```

II.3. Quick-Sort

The goal qsort(L,[],R) is true if list R is a sorted version of list L. More generally, qsort(L,R0,R) is true if list R consists of the members of list L sorted into order, followed by the members of list R0. The algorithm used is a variant of Hoare's "Quick Sort".

```
:-mode qsort(+,+,-).
:-mode partition(+,+,-,-).

qsort([X|L],R0,R) :-
   partition(L,X,L1,L2),
   qsort(L2,R0,R1),
   qsort(L1,[X|R1],R).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :- X =< Y, !,
   partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :- X > Y, !,
   partition(L,Y,L1,L2).
partition([],_,[],[]).
```

II.4. Differentiation

The goal d(E1,X,E2) is true if expression E2 is a possible form for the derivative of expression E1 with respect to X.

```
:-mode d(+,+,-).
:-op(300,xfy,^).
                                                % Basic rules
d(U+V,X,DU+DV) :- !, d(U,X,DU), d(V,X,DV).
d(U-V,X,DU-DV) :- !, d(U,X,DU), d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
d(U^N,X,N*U^N1*DU) :- !, integer(N), N1 is N-1, d(U,X,DU).
d(-U,X,-DU) :- !, d(U,X,DU).
                                                % Terminating rules
d(X,X,1) :- !.
d(C,X,0) :- atomic(C), !.

d(sin(X),X,cos(X)) :- !.
d(cos(X),X,-sin(X)) :- !.
d(exp(X),X,exp(X)) :- !.
d(log(X),X,1/X) :- !.
                                                % Chain rule
d(F_G,X,DF*DG) :- F_G=..[F,G], !,
                  d(F_G,G,DF), d(G,X,DG).
```

II.5. Mapping a List of Items into a List of Serial Numbers

The goal serialise(L1,L2) is true if L2 is a list of serial numbers corresponding to the members of list L1, where the members of L1 are numbered (from 1 upwards) in order of increasing size. e.g. ?-serialise([1,9,7,7],X). gives X = [1,3,2,2].

```
serialise(Items,SerialNos) :-
   pairlists(Items,SerialNos,Pairs),
   arrange(Pairs,Tree),
   numbered(Tree,1,N).

pairlists([X|L1],[Y|L2],[pair(X,Y)|L3]) :- pairlists(L1,L2,L3).
pairlists([],[],[]).

arrange([X|L],tree(T1,X,T2)) :-
   split(L,X,L1,L2),
   arrange(L1,T1),
   arrange(L2,T2).
arrange([],void).

split([X|L],X,L1,L2) :- !, split(L,X,L1,L2).
split([X|L],Y,[X|L1],L2) :- before(X,Y), !, split(L,Y,L1,L2).
split([X|L],Y,L1,[X|L2]) :- before(Y,X), !, split(L,Y,L1,L2).
split([],_,[],[]).

before(pair(X1,Y1),pair(X2,Y2)) :- X1 < X2.

numbered(tree(T1,pair(X,N1),T2),N0,N) :-
   numbered(T1,N0,N1),
   N2 is N1+1,
   numbered(T2,N2,N).
numbered(void,N,N).
```

II.6. Use of Meta-Predicates

This example illustrates the use of the meta-predicates var and =... The procedure call variables(Term,L,[]) instantiates variable L to a list of all the variable occurrences in the term Term. e.g. variables(d(U*V,X,DU*V+U*DV), [U,V,X,DU,V,U,DV], []).

```
variables(X,[X|L],L) :- var(X), !.
variables(T,L0,L) :- T =.. [F|A], variables1(A,L0,L).

variables1([T|A],L0,L) :- variables(T,L0,L1), variables1(A,L1,L).
variables1([],L,L).
```

II.7. Prolog in Prolog

This example shows how simple it is to write a Prolog interpreter in Prolog, and illustrates the use of a variable goal. In this mini-interpreter, goals and clauses are represented as ordinary Prolog data structures (i.e. terms). Terms representing clauses are specified using the unary predicate my_clause, e.g.

```
my_clause( (grandparent(X,Z):-parent(X,Y),parent(Y,Z)) ).
```

A unit clause will be represented by a term such as

```
my_clause( (parent(john,mary) :- true) )
```

The mini-interpreter consists of four clauses:

```
| execute(true) :- !.
| execute((P,Q)) :- !, execute(P), execute(Q).
| execute(P) :- my_clause((P:-Q)), execute(Q).
| execute(P) :- P.
```

The last clause enables the mini-interpreter to cope with calls to ordinary Prolog predicates, e.g. evaluable predicates.

II.8. Translating English Sentences into Logic Formulae

The following example of a definite clause grammar defines in a formal way the traditional mapping of simple English sentences into formulae of classical logic. By way of illustration, if the sentence

    Every man that lives loves a woman.

is parsed as a sentence by the call

```
| ?- phrase(sentence(P), [every,man,that,lives,loves,a,woman]).
```

then P will get instantiated to

```
all(X):(man(X)&lives(X) => exists(Y):(woman(Y)&loves(X,Y)))
```

where ':', '&' and '=' are infix operators defined by

```
:-op(900,xfx,=>).
:-op(800,xfy,&).
:-op(300,xfx,:).
```

The grammar follows:

```
| sentence(P) --> noun_phrase(X,P1,P), verb_phrase(X,P1).
|
| noun_phrase(X,P1,P) -->
|    determiner(X,P2,P1,P), noun(X,P3), rel_clause(X,P3,P2).
| noun_phrase(X,P,P) --> name(X).
|
| verb_phrase(X,P) --> trans_verb(X,Y,P1), noun_phrase(Y,P1,P).
| verb_phrase(X,P) --> intrans_verb(X,P).
|
| rel_clause(X,P1,P1&P2) --> [that], verb_phrase(X,P2).
| rel_clause(_,P,P) --> [].
|
| determiner(X,P1,P2, all(X):(P1=>P2) ) --> [every].
| determiner(X,P1,P2, exists(X):(P1&P2) ) --> [a].
|
| noun(X, man(X) ) --> [man].
| noun(X, woman(X) ) --> [woman].
|
| name(john) --> [john].
|
| trans_verb(X,Y, loves(X,Y) ) --> [loves].
| intrans_verb(X, lives(X) ) --> [lives].
```

APPENDIX III

INSTALLATION DEPENDENCIES

III.1. Getting Started

If the Monitor command

```
.r prolog
```

does not work, then it is likely that Prolog is not in your SYS: area. In this case you will need to type

```
.run prolog[project-no,programmer-no]
```

where the project and programmer numbers identify the area where the PROLOG.EXE and PLCOMP.EXE files are to be found.

III.2. Using a Terminal without Lower-Case

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

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

```
_VALUE2
```

is a variable, while

```
VALUE2
```

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

```
value2
```

written in the 'LC' convention.

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

```
:-'NOLC'.
```

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

```
:-'LC'.
```

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

III.3. Using a Monitor without Virtual Memory

If the TOPS-10 Monitor at your installation does not have the virtual memory option, you should specify in the "r" command the amount of core your program will need for stacks. Thus, you should call Prolog with:

```
.r prolog nK
```

where n-2 should be the stack requirements. A value of 10 for n is ample for most uses. If, however, a running program requires more than the allocated amount, the error message

```
! stack space full
```

is issued and the execution aborted. If your program was not in an infinite recursion due to a programming error, you should try to run Prolog with a higher value of n.

Note that, under such a Monitor, the trimcore predicate will not recover store which is no longer needed.

III.4. Using TOPS-20

The system interaction predicate plsys/1 does not work under TOPS-20 with the arguments run(_,_) and tmpcor(_,_,_). It can still be used, however, for saving a runnable image of a program via

```
plsys(core_image).
```

SUMMARY OF EVALUABLE PREDICATES

```
abolish(F,N)       Abolish the interpreted procedure named F arity N.
abort              Abort execution of the current directive.
ancestors(L)       The ancestor list of the current clause is L.
arg(N,T,A)         The Nth argument of term T is A.
assert(C)          Assert clause C.
assert(C,R)        Assert clause C, reference R.
asserta(C)         Assert C as first clause.
asserta(C,R)       Assert C as first clause, reference R.
assertz(C)         Assert C as last clause.
assertz(C,R)       Assert C as last clause, reference R.
atom(T)            Term T is an atom.
atomic(T)          Term T is an atom or integer.
bagof(X,P,B)       The bag of instances of X such that P is provable is B.
break              Break at the next interpreted procedure call.
'C'(S1,T,S2)       (grammar rules) S1 is connected by the terminal T to S2.
call(P)            Execute the interpreted procedure call P.
clause(P,Q)        There is an interpreted clause, head P, body Q.
clause(P,Q,R)      There is an interpreted 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.
compile(F)         Compile the procedures in text file F.
consult(F)         Extend the interpreted 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, most general term T.
current_predicate(A,P) A current predicate is named A, most general goal P.
current_op(P,T,A)  Atom A is an operator type T precedence P.
debug              Switch on debugging.
debugging          Output debugging status information.
depth(D)           The current invocation depth is D.
display(T)         Display term T on the terminal.
erase(R)           Erase the clause or record, reference R.
expand_term(T,X)   Term T is a shorthand which expands to term X.
fail               Backtrack immediately.
fileerrors         Enable reporting of file errors.
functor(T,F,N)     The principal functor of term T has name F, arity N.
gc                 Enable garbage collection.
gcguide(F,O,N)     Change garbage collection parameter F from O to 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.
incore(P)          Execute the compiled procedure call P.
instance(R,T)      A most general instance of the record reference R is T.,
integer(T)         Term T is an integer.
Y is X             Y is the value of integer expression X.
keysort(L,S)       The list L sorted by key yields S.
leash(M)           Set leashing mode to M.
length(L,N)        The length of list L is N.
listing            List the current interpreted program.
listing(P)         List the interpreted procedure(s) specified by P.
log                Enable logging.
maxdepth(D)        Limit invocation depth to D.
name(A,L)          The name of atom or integer A is string L.
```

nl	Output a new line.
nodebug	Switch off debugging.
nofileerrors	Disable reporting of file errors.
nogc	Disable garbage collection.
nolog	Disable logging.
nonvar(T)	Term T is a non-variable.
nospy P	Remove spy-points from the procedure(s) specified by P.
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.
phrase(P,L)	List L can be parsed as a phrase of type P.
plsys(T)	Allows certain interactions with the operating system.
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 interpreted program with procedures from file F.
recorda(K,T,R)	Make term T the first record under key K, reference R.
recorded(K,T,R)	Term T is recorded under key K, reference R.
recordz(K,T,R)	Make term T the last record under key K, reference R.
reinitialise	Initialisation - looks for 'prolog.bin' or 'prolog.ini'.
rename(F,G)	Rename file F to G.
repeat	Succeed repeatedly.
restore(S)	Restore the state saved in file S.
retract(C)	Erase the first interpreted clause of form C.
revive(F,N)	Revive the latest compiled procedure named F arity N.
save(F)	Save the current state of Prolog in file F.
save(F,R)	As save(F) but R is 0 first time, 1 after a 'restore'.
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,S)	The set of instances of X such that P is provable is S.
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) specified by P.
statistics	Output various execution statistics.
statistics(K,V)	The execution statistic key K has value V.
subgoal_of(G)	An ancestor goal of the current clause is G.
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 immediately.
trimcore	Reduce free stack space to a minimum.
true	Succeed.
ttyflush	Transmit all outstanding terminal output.
ttyget(C)	The next non-blank character input from the terminal is C.
ttyget0(C)	The next character input from the terminal is C.
ttynl	Output a new line on the terminal.
ttyput(C)	The next character output to the terminal is C.
ttyskip(C)	Skip over terminal input until after character C.
ttytab(N)	Output N spaces to the terminal.
unknown(O,N)	Change action on unknown procedures from O to N.
var(T)	Term T is a variable.
version	Displays introductory and/or system identification messages.
version(A)	Adds the atom A to the list of introductory messages.

```
write(T)            Write the term T.
writeq(T)           Write the term T, quoting names where 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^P                 There exists an X such that P is provable.
X<Y                 As integer values, X is less than Y.
X=<Y                As integer values, X is less than or equal to Y.
X>Y                 As integer values, X is greater than Y.
X>=Y                As integer values, X is greater than or equal to Y.
X=Y                 Terms X and Y are equal (i.e. unified).
T=..L               The functor and arguments 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 to term Y.
X@>Y                Term X follows term Y.
X@>=Y               Term X follows or is identical to term Y.
[F|R]               Perform the consult/reconsult(s) on the listed files..
```

STANDARD OPERATORS

```
:-op( 1200, xfx, [ :-, --> ]).
:-op( 1200,  fx, [ :-, ?- ]).
:-op( 1150,  fx, [ mode, public ]).
:-op( 1100, xfy, [ ; ]).
:-op( 1050, xfy, [ -> ]).
:-op( 1000, xfy, [ ',' ]).       /* See note below */
:-op(  900,  fy, [ \+, spy, nospy ]).
:-op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                               =:=, =\=, <, >, =<, >= ]).
:-op(  500, yfx, [ +, -, /\, \/ ]).
:-op(  500,  fx, [ +, - ]).
:-op(  400, yfx, [ *, /, <<, >> ]).
:-op(  300, xfx, [ mod ]).
:-op(  200, xfy, [ ^ ]).
```

Note that a comma written literally as a punctuation character can be used as though it were an infix operator of precedence 1000 and type 'xfy', i.e.

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

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

REFERENCES

[Byrd 80] Byrd L.
Understanding the control flow of PROLOG programs.
Research Paper 151, Dept of Artificial Intelligence, University of Edinburgh, 1980.
This paper was presented at the Logic Programming Workshop in Debrecen, Hungary, 1980.

[Clocksin & Mellish 81]
Clocksin W.F. and Mellish C.S.
Programming in Prolog.
Springer-Verlag, 1981.

[Colmerauer 75]
Colmerauer A.
Les Grammaires de Metamorphose.
Technical Report, Groupe d'Intelligence Artificielle,Marseille-Luminy, November, 1975.
Appears as "Metamorphosis Grammars" in "Natural Language Communication with Computers", Springer Verlag, 1978.

[Kowalski 74] Kowalski R.A.
Logic for Problem Solving.
DCL Memo 75, Dept of Artificial Intelligence, University of Edinburgh, March, 1974.

[Kowalski 79] Kowalski R.A.
Artificial Intelligence: Logic for Problem Solving.
North Holland, 1979.

[Pereira & Warren 80]
Pereira F.C.N. and Warren D.H.D.
Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks.
Artificial Intelligence 13:231-278, 1980.
Also available as Research Paper 116, Dept of Artificial Intelligence, University of Edinburgh.

[Roussel 75] Roussel P.
Prolog : Manuel de Reference et d'Utilisation
Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[van Emden 75] van Emden M.H.
Programming with Resolution Logic.
Technical Report CS-75-30, Dept of Computer Science, University of Waterloo, Canada, November, 1975.

[Warren 77] Warren D.H.D.
Implementing Prolog - Compiling Predicate Logic Programs.
Research Reports 39 & 40, Dept of Artificial Intelligence, University of Edinburgh, 1977.

[Warren et al. 77]
Warren D.H.D., Pereira L.M. and Pereira F.C.N.
Prolog - the Language and its Implementation Compared with Lisp.
In Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages. SIGART/SIGPLAN Notices, Rochester, N.Y., August, 1977.

INDEX

'->' - evaluable predicate 41
'<' - evaluable predicate 37
'=' - evaluable predicate 40
'=..' - evaluable predicate 44
'=:=' - evaluable predicate 37
'=<' - evaluable predicate 37
'==' - evaluable predicate 38
'=\=' - evaluable predicate 37
'>' - evaluable predicate 37
'>=' - evaluable predicate 37
'@<' - evaluable predicate 38
'@=<' - evaluable predicate 38
'@>' - evaluable predicate 38
'@>=' - evaluable predicate 38
'\+' - evaluable predicate 41
'\==' - evaluable predicate 38
'^' - evaluable predicate 50

^ 2
^ - evaluable predicate 50
^C interruption 8

Abolish - evaluable predicate 46
Abort - evaluable predicate 9, 57
Ancestors - evaluable predicate 42
Anonymous variables 4, 64
Arg - evaluable predicate 44
Arguments 64
Arity of functor 64
Assert - evaluable predicate 46, 48
Asserta - evaluable predicate 46, 48
Assertz - evaluable predicate 46, 48
Associativity 72
Atom 63
Atom - evaluable predicate 44
Atomic - evaluable predicate 44

Backtracking 69
Bagof - evaluable predicate 50
Body of clause 66
Break - evaluable predicate 9, 57

C - evaluable predicate 55, 56
Call - evaluable predicate 45
Call port of a procedure box 14
Clause 66
Clause - evaluable predicate 46, 48
Clause instance 69
Clauses - declarative and procedural interpretations 66
Close - evaluable predicate 32
Commands 6
Compare - evaluable predicate 39

Compile - evaluable predicate 23, 51
Compiled procedures - access from interpreted code 23
Compound term 63
Constant 63
Consult - evaluable predicate 31
Consulting 3
Creep - debugging option 15, 18
Current_atom - evaluable predicate 42
Current_functor - evaluable predicate 42
Current_op - evaluable predicate 57
Current_predicate - evaluable predicate 43
Cut 71

Debug - evaluable predicate 15, 52
Debug Mode 15
Debugging - evaluable predicate 15, 53
Definite clause grammars 54
Depth - evaluable predicate 58
Directives 4, 66
Display - evaluable predicate 33

Erase - evaluable predicate 47
Evaluable predicates 29, 68, 91
Exit port of a procedure box 14
Expand_term - evaluable predicate 56

Fail - evaluable predicate 40
Fail port of a procedure box 14
Fileerrors - evaluable predicate 32
Functor 64
Functor - evaluable predicate 44

Gc - evaluable predicate 58
Gcguide - evaluable predicate 58
Get - evaluable predicate 34
Get0 - evaluable predicate 34
Goal 66
Grammar rules 54

Halt - evaluable predicate 57
Head of clause 66

Incore - evaluable predicate 51
Indexing 26
Instance - evaluable predicate 47
Instantiated 44
Integer 63
Integer - evaluable predicate 44
Integer expressions 36
Is - evaluable predicate 37

Keysort - evaluable predicate 39

Layout-character 68
LC - evaluable predicate 57, 89

Leap - debugging option 18
Leash - evaluable predicate 16, 52
Leashing Mode 16
Length - evaluable predicate 40
Listing - evaluable predicate 42
Lists 65
Log - evaluable predicate 32

Maxdepth - evaluable predicate 58
Mode declarations 25

Name - evaluable predicate 45
Name of functor 64
Newline character 34
Nl - evaluable predicate 34
Nodebug - evaluable predicate 15, 52
Nofileerrors - evaluable predicate 32
Nogc - evaluable predicate 58
NOLC - evaluable predicate 57, 89
Nolog - evaluable predicate 32
Non-unit clause 66
Nonvar - evaluable predicate 44
Nospy - evaluable predicate 16, 53
Notation 2
Numbervars - evaluable predicate 42

Op - evaluable predicate 57
Operator precedence 72
Operator type 72
Operators 64, 72
Operators - standard 73, 94

Phrase - evaluable predicate 56
Plsys - evaluable predicate 61
Port of a procedure box 14
Predicate 66
Principal functor 64
Print - evaluable predicate 33
Procedure 67
Procedure box model of control flow 13
Procedure call 66
Program 66
Program state 9
Prompt - evaluable predicate 59
Put - evaluable predicate 34

Questions 4, 66

Read - evaluable predicate 33
Reconsult - evaluable predicate 31
Reconsulting 4
Recorda - evaluable predicate 47
Recorded - evaluable predicate 47
Recordz - evaluable predicate 47
Redo port of a procedure box 14

Reinitialise - evaluable predicate 58
Rename - evaluable predicate 32
Repeat - evaluable predicate 41
Restore - evaluable predicate 9, 57
Retract - evaluable predicate 46
Revive - evaluable predicate 51
Running a saved image directly from the Monitor 61
Running other programs from Prolog 61

Save - evaluable predicate 9, 57
See - evaluable predicate 32
Seeing - evaluable predicate 32
Seen - evaluable predicate 32
Sentences 66
Setof - evaluable predicate 49
Skip - debugging option 18
Skip - evaluable predicate 34
Sort - evaluable predicate 39
Spy - evaluable predicate 16, 53
Spy-points 16
Statistics - evaluable predicate 59
Strings 66
Subgoal_of - evaluable predicate 42
Syntax of Prolog 63, 74, 76

Tab - evaluable predicate 34
Tell - evaluable predicate 32
Telling - evaluable predicate 32
Term 63
Told - evaluable predicate 32
Top level 3
Trace - evaluable predicate 15, 52
Trimcore - evaluable predicate 58
True - evaluable predicate 40
Ttyflush - evaluable predicate 35
Ttyget - evaluable predicate 35
Ttyget0 - evaluable predicate 35
Ttynl - evaluable predicate 35
Ttyput - evaluable predicate 35
Ttyskip - evaluable predicate 35

Unification process 69
Uninstantiated 44
Unit clause 66
Unknown - evaluable predicate 7, 52

Var - evaluable predicate 44
Variable 63
Version - evaluable predicate 60

Write - evaluable predicate 33
Writeq - evaluable predicate 33