3.22

MINISTÉRIO DA HABITAÇÃO E OBRAS PÚBLICAS

Laboratório Nacional de Engenharia Civil

USER's GUIDE TO DECsystem-10 PROLOG

Provisional version

Lisboa, Outubro de 1978

Trabalho integrado no PLANO DE INVESTIMENTOS

RELATÓRIO

MINISTÉRIO DA HABITAÇÃO E OBRAS PÚBLICAS

LABORATÓRIO NACIONAL DE ENGENHARIA CIVIL

DIVISÃO DE INFORMÁTICA

Proc.03/13/5570

USER'S GUIDE TO DECsystem-10 PROLOG
(Provisional version)

Lisboa, Outubro de 1978

ERRATA

1. Errors in the User's Guide

- Prolog's unification doesn't have a "occur check", ie. when unifying a variable against a term the system doesn't 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". Note that the absence of the occur check is not a bug or design oversigth, 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, where as 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 (eg. 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 is in practice very slight, and since Prolog started to be used, seven years ago in Marseille, there is no record of a practical programming task which was hampered in a fundamental way by the absence of a occur check. Usually, the restriction is only felt in toy programs.

- It is not said that 'retract' is nondeterminate, ie. that it will retract sucessive clauses matching the argument, through backtracking. Also, it is not explained that the argument to 'retract' 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).

- It is not said that the interpreter ignores compiler directives in a file being (re)consulted.

- Postfix ';' is not a standard operator. '=..' is a standard infix operator.

```
:-op(700,xfx,=..).
```

- The last clause of example 9.4 (differentiation) should read

```
d(C,X,0) :- atomic(C), C\== X, !.
```

- The example of trace in pg.20 uses the definition of 'concatenate' given in section 9.1.

- The restrictions on the rhs argument of a compiled 'is' (arithmetic evaluation) are not sufficiently stressed:-

> The rhs of a compiled 'is' must be a term formed of evaluable arithmetic functors, arithmetic constants and variables. At run time, all the variables must be bound to integers, otherwise a warning message is issued and the value of the variable is considered to be 0. An arithmetic constant is either an integer or a term [x], where x is an integer. This last form allows one to write "c" for the (constant) character c. The arithmetic evaluable functors are _+_, _-_, _*_, _/_, _mod_, -_, _/_, _\/_, _\, _<<_, _>>_, !_ and $_ .

- The module name given in the ':-module' directive in a compiled module should not coincide with the name of any unary predicate defined in the module or called from the module and defined by an ':-ext' directive. This is a permanent restriction, not a bug.

- The syntax no longer allows [..X] to stand for X. This should be reflected in Section 7.3, where the definition of listexpr should read:-

```
listexer          --> subterm(999)
                    |  subterm(999) , listexer
                    |  subterm(999) :  subterm(999)
                    |  subterm(999) ...  subterm(999)
```

2. Bugs in the System

- It is not checked whether the arguments to '(re)consult' are atoms. If they are not, a "mode error" happens, and the (re)consulting fails.

- If the last argument of 'numbervars' is instantiated, this evalpred may go into a loop.

- 'not' is not traced.

- 'instance' returns '?(X)' where it should return 'X'.

- 'clause' fails with a "mode error" if its second argument is instantiated. To circumvent this, if you expect to use 'clause' with argument instantiated, use instead a predicate 'safeclause', which you should define as

```
safeclause(P,Q) :- clause(P,Q0), Q = Q0.
```

- In certain (unclear...) circunstances, the compiler fails to produce the code for a predicate which has both 'mode' and 'ext', 'program' or 'module' declarations. Errors will occur during the assembly, the loading or the execution of the compiled module. There is no sure cure for this, but it seems to be better if the module is ordered as follows:-

```
:-ext(...).                 /* ext declarations */
...
:-entry(...).               /* entry declarations */
...
:-mode ...(...).            /* mode declarations */
...
:-module ... or             /* module or
:-program ...                  program declaration */

/* clauses */
...
```

- A clause with two sucessive cuts, eg. "a:-b,!,!,c", is not executed properly by the interpreter: a call to '$(_)' is generated. However, as two sucessive cuts are equivalent to a single cut, this is not a restriction.

- Prolog doesn't understand line numbered files. The characters which make up the line numbers are treated as normal input characters.

USER'S GUIDE to DECsystem-10 PROLOG

September 1978

Luis Moniz Pereira

Divisao de Informatica
Laboratorio Nacional
de Engenharia Civil
Lisbon

Fernando C N Pereira & David H D Warren

Department of Artificial Intelligence
University of Edinburgh

NB. This issue of the User's Guide describes the DECsystem-10 Prolog interpreter version 1.32 and compiler version 1.11. Comments and suggestions are welcomed.

LNEC - Proc.03/13/5570

CONTENTS

1.0 INTRODUCTION

Prolog is a simple but powerful programming language developed at the University of Marseille [Roussel 1975], as a practical tool for programming in logic [Kowalski 1974] [van Emden 1975] [Colmerauer 1975]. 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.

The DECsystem-10 Prolog system [Warren 1977] [Warren, Pereira, Pereira 1977] comprises an interpreter and a compiler, both written largely in Prolog itself.

The interpreter comprises a superviser ("SVWX") supported by a run-time system (RTS) consisting of unification routines and built-in procedures. It reads in and executes Prolog source programs.

The compiler enables a set of Prolog source modules making up a program to be independently compiled. The compiled modules can then be automatically loaded, together with modules from the RTS, to produce a final executable program. Compiled programs are easily provided with an interpretative interface to the superviser.

The interpreter facilitates the development and testing of Prolog programs. If a procedure is interpreted, modifications to it can be incorporated more quickly. Interpreted programs can have access to compiled procedures.

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. It is only worthwhile compiling programs which are well-tested and are to be used extensively.

NB. In this user's guide we shall assume the "full character-set" convention ('LC') described in Section 3.1, except where otherwise stated. When lower-case is not available, the "no lower-case" ('NOLC') convention must be used. Notice also that metavariables are underlined. For example, a symbol such as foo may be used in the text to refer to some item in the object language, Prolog.

2.0 PROGRAMMING IN LOGIC - THE PROLOG LANGUAGE

This section provides an 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.

2.1 Syntax, Terminology And Informal Semantics

2.1.1 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
```

In DECsystem-10 Prolog, integers are restricted to the range -2^17 to 2^17-1, ie. -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. As an example of the notation used:-

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

all represent the integer fifteen.

Constants also include atoms such as:-

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

The symbol for an atom can be any sequence of characters, which should be written in quotes if there is possibility of confusion with other symbols (such as variables, integers). As in conventional programming languages, constants are definite elementary objects, and correspond to proper nouns in natural language.

Variables are distinguished by an initial capital letter or by the initial character "_", eg.

```
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 a single 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 Algol-68.

The structured data objects of the language are the compound terms. A compound term comprises a <u>functor</u> (called the <u>principal</u> functor of the term) and a sequence of one or more terms called <u>arguments</u>. A functor is characterised by its <u>name</u>, which is an atom, and its <u>arity</u> 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:-

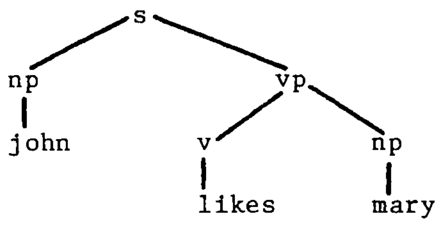
```
point(X,Y,Z)
```

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 <u>operators</u> - 2-ary functors may be declared as <u>infix</u> operators and 1-ary functors as <u>prefix</u> or <u>postfix</u> operators. Thus it is possible to write, eg.

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

as optional alternatives to:-

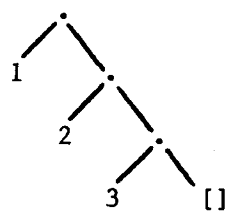
```
+(X,Y)    ;(P,Q)    <(X,Y)    +(X)    ;(P)
```

The use of operators is described fully in Section 2.4 below.

An important class of data structures are the <u>lists</u>. These 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,[])))
```

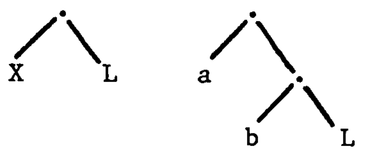
but which is normally written, in a special list notation, as:-

```
[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. These lists may also be written:-

```
[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 <u>strings</u>. An example is:-

```
"DECsystem-10"
```

which represents exactly the same list as:-

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

2.1.2 Programs -

A fundamental unit of a logic program is the <u>goal</u> or <u>procedure call</u>. Examples are:-

```
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 (ie. it too may be empty). If the head is not empty, the sentence is called a clause.

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*."

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."

A sentence with an empty head is called a directive, 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 - ie. 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 of 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 <u>procedure</u> for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a ternary predicate 'concatenate' might 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".

Certain predicates are predefined by <u>built-in procedures</u> supplied by the Prolog system. Such predicates are called <u>evaluable predicates</u>.

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.

2.2 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 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, ie. 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 = []
```

ie. 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.

2.3 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", ie. 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.

Examples:-

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

The only result producing by executing:-

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

is X=a, the other two potential solutions being discarded.

(2)
```
compile(S,R) :- parse(S,T), !, translate(T,R).
```

The procedure 'compile' only calls 'translate' for the first solution produced by 'parse'. Alternative solutions which 'parse' might produce are discarded.

2.4 Operators

The Prolog syntax caters for operators of three main kinds - infix, prefix and postfix. 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, in an otherwise ambiguous subexpression, 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, ie.

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

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

```
        xfx     xfy     yfx
```

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

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

```
        a-b+c
```

is valid, and means:-

```
        (a-b)+c     ie.  +(-(a,b),c)
```

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

```
        a-(b+c)     ie.  -(a,+(b,c))
```

if the types were both 'xfy'.

The possible types for a prefix operator are:-

```
        fx     fy
```

and for a postfix operator they are:-

```
          xf     yf
```

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

```
          not not P
```

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

```
          not P
```

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

In DECsystem-10 Prolog, 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, ie. infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to operators which are provided as _standard_ in DECsystem-10 Prolog, namely:-

```
        :-op( 1200, xfx, [:-,-->]).

        :-op( 1200,  fx, [:-,?-]).

        :-op( 1100, xfy, ';' ).

        :-op( 1100,  xf, ';' ).

        :-op( 1050, xfy,  -> ).

        :-op( 1001,  fx, mode ).

/*      :-op( 1000, xfy, ',' ).     See note below.   */

        :-op(  700, xfx, [=,==,\==,is,<,>,=<,>=,=:=,=\=]).

        :-op(  700, xfy, [:=,+:=]).

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

        :-op(  500,  fx, [+,-]).

        :-op(  400, yfx, [*,/,<<,>>]).

        :-op(  300, xfx, mod ).
```

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', ie.

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

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

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

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

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.
```

is invalid syntax, and must be written as eg.

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

(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 = (?-)
```

3.0 HOW TO USE THE INTERPRETER

To run the interpreter under the TOPS-10 Monitor with virtual memory, perform the Monitor command:-

```
        R PROLOG
```

{If your installation doesn't have Prolog in the system (SYS) area, type instead RUN PROLOG [p,pn] , where [p,pn] is the Prolog area.}

The interpreter responds with a message of identification and the prompt '|' as soon as it is ready to accept input. At this point the interpreter is expecting input of a directive, ie. a question or command. This state is called interpreter top level.

> Note 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.

If your terminal does not provide lower-case characters, you must first of all type the command:-

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

and then observe the "no lower-case" convention described in the following section.

3.1 'LC' And 'NOLC' Conventions

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', eg. by the command:-

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

To switch back to the "full character set" convention, call the built-in procedure 'LC', eg. 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.

3.2 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. The text of a program is normally created separately in a file (or files), using a text editor.

To input a program from a file file, give the special command:-

```
[file].
```

which will instruct the interpreter to read-in the program. If the file has some extension, its name is "PROG.PL" say, it is necessary to give the complete filename between single quotes. When the end of file or the special command ":-end." are found in file, 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.

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].
```

The interpreter is now in a state where it expects input of clauses or directives. To return to interpreter top level, give the special command:-

```
        :-end.
```

or alternatively just type ^Z. Either of these causes an end of file for the ersatz file ´user´, and end of file terminates program read-in.

3.3 Directives

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

Suppose list membership has been defined by:-

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

Note the use of anonymous variables written "_". The command:-

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

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

```
        ?
```

as a warning.

The syntax of a question is the same as a command, except that it is introduced by "?-" instead of ":-". If the specified goal(s) can be satisfied, the final value of each distinct variable is displayed (except for anonymous variables). The system then pauses awaiting input of a single character, followed by <cr> (carriage return), from the user. If the character is ";", the system backtracks to find an alternative solution. If no more solutions can be found it outputs:-

```
        no
```

and execution of the question is complete. If the user does not wish to see any alternative solutions, he must input a printable character other than ";" to stop the execution. If a question can be satisfied but contains no variables, then the system answers

```
        yes
```

and execution of the question terminates.

NB. At interpreter top level, but NOT during program read-in (whether from 'user' or from a file), a question may be abbreviated by omitting the '?-'. Thus, any term typed in which is neither a list nor has ':-' as principal functor, is taken as a question.

The outcome of some questions is shown below, where a number preceded by "_" is a system-generated name for a variable.

```
        ?- member(X,[tom,dick,harry]).
        X = tom;
        X = dick;
        X = harry;
        no

        ?- member(X,[a,b,f(Y,c)]),member(X,[f(b,Z),d]).
        Y = b,
        X = f(b,c),
        Z = c.

        ?- member(X,[f(_),g]).
        X = f(_1728).

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

3.4 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 to 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. eg.

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

gives:-

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

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

3.5 Saving A Program

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

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

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

3.6 Restoring A Saved Program

Once a program has been saved into a file _file_, the following command 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'. For example, upon completion of a 'restore', the character convention in force will be that which was previously saved.

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

3.7 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 (control C) if the system is in input wait, or ^C^C (two control Cs) 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:-

B,	break the current execution;
C,	continue the execution;
E,	exit from Prolog, closing all files;
G,	switch garbage collection (initially on);
H,	list available commands;
M,	break to Monitor level (use Monitor command 'CONT' to proceed);
N,	disable trace;
T,	enable trace.

The use of these commands is explained in the sections that follow.

If when trying to interrupt a program with ^C you accidentally get to Monitor level, (perhaps because you typed too many ^Cs), type 'CONT' to proceed.

3.8 Suspending, Saving, And Restoring An Execution

When the execution of a program is interrupted, all the information necessary for continuing the execution is still retained by the interpreter. This information constitutes the _state_ of the execution.

The state of an execution may be saved and restored in exactly the same way as program states, by means of the 'save' and 'restore' commands. Thus, if you want to save into a file _file_ the state of an execution for future resumption then, after a ^C interruption, type B (break) followed by <cr>. This will cause the interpreter to suspend execution immediately prior to the next call to an INTERPRETED procedure. The message:-

```
% break:
```

will then be displayed. This signals the start of a _break_, and except for the effect of 'abort's (see below), it is as if the interpreter was at top level.

You can now save the state with the directive:-

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

A ":-end." command or a ^Z character will close the break and resume the execution which was suspended. Execution will be resumed at the procedure call where it had been suspended.

A suspended execution can be aborted by issuing the command:-

```
:-abort.
```

within the break; see Section 3.11 below.

A break may also be produced by calling the built-in procedure 'break' anywhere within a program. Breaks may be nested within other breaks.

3.9 Tracing

The Prolog interpreter provides a tracing facility. When tracing is enabled, each procedure call in an interpreted clause is displayed on the terminal with the current values of its arguments. This information is preceded by a number with a "-" sign, which indicates the <u>invocation level</u> of the goal being displayed. (The invocation level of each of the goals of a clause is one greater than that of the goal which activated the clause. The first level is zero.) Also, each time execution of a goal is successfully completed, it is displayed again with the new current values of all its arguments. Again the level of the goal is indicated, this time preceded by a "+" sign.

For example, the question:-

```
?-concatenate([a,b],[c,d],L).
```

when traced produces:-

```
-0 concatenate([a,b],[c,d],_202)
-1 concatenate([b],[c,d],_328)
-2 concatenate([],[c,d],_338)
  +2 concatenate([],[c,d],[c,d])
  +1 concatenate([b],[c,d],[b,c,d])
  +0 concatenate([a,b],[c,d],[a,b,c,d])
L = [a,b,c,d]
```

Tracing has two modes, 'leashed' and 'unleashed'. When tracing in 'leashed' mode, a '?' prompt is issued immediately after each traced call to an INTERPRETED procedure. A one-letter response followed by <cr> must then be given, choosing amongst various tracing and execution options. Possible responses are:-

A,	abort the execution;
B,	suspend the execution, entering a break;
C,	continue tracing;
F,	fail this goal;
H,	list available options;
N,	switch off tracing;
S,	skip tracing of this goal;
U,	switch to 'unleashed' tracing.

Tracing is enabled by the command:-

```
:-trace.
```

and 'leashed' mode is enabled by:-

```
:-leash.
```

Tracing is disabled by the command:-

```
:-notrace.
```

and 'leashed' mode is disabled by:-

```
:-unleash.
```

The defaults are tracing off, and 'leashed' mode for tracing.

Tracing may also be enabled or disabled by typing T (trace) or N (notrace) respectively, and <cr>, following a ^C interruption.

Certain goals are never traced, namely:-
primitive I/O (eg. 'get'),
'write',
trace control (eg. 'notrace'),
and also any goal which is not logically atomic (eg. conjunctions, disjunctions, 'true').

3.10 Logging

When Prolog is entered, all terminal interaction is automatically written to the file PROLOG.LOG in append mode (ie., if PROLOG.LOG already exists, the new data is appended to it). This facility can be switched off by calling the evaluable predicate 'nolog', and on again by calling 'log'.

3.11 Aborting An Execution

To abort the current execution, ie. to force an immediate failure of the directive currently being executed to interpreter top level, call the evaluable predicate 'abort', either from the program or by executing the command:-

```
:-abort.
```

within a break. In this case no ":-end." or ^Z is needed to close the break.

3.12 Exiting From The Interpreter

To exit from the interpreter and return to monitor level either call the built-in procedure 'halt', type ":-end." or ^Z at interpreter top level, or use the E (exit) command following a ^C interruption.

3.13 Special Commands

The special command:-

```
:-end.
```

terminates a break, a program file, or exits to the Monitor from interpreter top level. It is equivalent to ^Z or end of file.

Programs from one or more files may be read-in by giving, as a special command, simply a list of the names of the files. (The case where there is just one name in the list has already been described above). A file name may optionally be preceded by the operator '-' to indicate that the file should be "reconsulted" rather than "consulted". (The built-in procedures 'consult' and 'reconsult' for reading-in programs are described fully in Section 5.1). A special command such as:-

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

is thus merely a shorthand for:-

```
:-consult(file1),reconsult(file2),consult(file3).
```

4.0 HOW TO USE THE COMPILER

The DECsystem-10 Prolog compiler [Warren 1977] produces compact and efficient code, running 10 to 20 times faster than interpreter code, and requiring much less runtime storage (stacks). Compiled Prolog programs are comparable in efficiency with Lisp programs for the same task, compiled by current DECsystem-10 Lisp compilers [Warren & al. 1977].

Although compiled code is as _safe_ as interpreted programs (ie., no unpredictable errors or results are possible - barring system bugs!), it is not advisable to compile untested programs, because compilation takes more time (both yours and the machine's), and because currently there are no debugging aids for compiled code.

Note that a compiled procedure can NOT be augmented or modified at runtime with interpreted clauses.

When you are going to the trouble of compiling a program, it is often worthwhile including optional _mode declarations_ to inform the compiler that certain procedures will only be used in restricted ways, ie. 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. Full details of mode declarations are given in Section 4.2.

4.1 Basic Use

You should have library access to the Prolog area. To get it (under the TOPS-10 Monitor), either incant:-

```
.R SETSRC
*LIB:[p,pn]
*^C
.
```

or include in your SWITCH.INI file the line:-

```
LOGIN /LIB:[p,pn]
```

where [_p_,_pn_] is the Prolog area.

A Prolog program to be compiled may be split into several files, which are called _modules_. The compiler creates a file containing information common to all modules of a program, which is called the _functors file_. The user has to give the program some name _name_; the functors file is then called _name_.FNS.

One should include in the main module of the program the compiler directive:-

```
:-program(modules,interactives).
```

where modules is a list of the other modules in the program, and interactives is a list of the predicates defined in this module which are to be made "interactive", ie. accessible to the interpreter. For example, if there are no other modules and the predicates 'tom(_)', 'dick(_,_)' and 'harry', defined in the main module, are to be made interactive, then the appropriate directive is:-

```
:-program([],[tom(1),dick(2),harry(0)]).
```

If there are other modules besides the main one, their names must be listed as the first argument of the 'program' directive, and in addition a compiler directive of the following type must be included in each auxiliary module:-

```
:-module(module-name,interactives).
```

where module-name is the name of the module (only the first 6 characters are significant), and interactives is again the list of predicates defined within the module which are to be made interactive. For example:-

```
:-module(extras,[rag(3),tag(2),bobtail(1)]).
```

All interactive predicates may be directly invoked from interpreted clauses (including directives entered on-line). A compiled clause may directly invoke any of the predicates defined in the same module, and any evaluable predicate. Any interactive predicate (which includes also evaluable predicates and predicates defined by interpreted clauses) may be invoked anywhere by use of the built-in predicate 'call(_)', eg.

```
call(bobtail(X))
```

Non-interactive compiled predicates are generally only accessible in the module in which they are defined, but see Section 4.3 below.

Use of a variable, such as 'P', as a goal is equivalent to the goal:-

```
call(P)
```

At the time of execution, P must be instantiated to a goal of which the predicate is interactive, (or, more generally, to a conjunction, disjunction etc. of such goals).

To compile a program, to be called name, comprising say modules main, module1 and module2, you should type the following (where the ":" and "program:" characters are prompts from the Prolog compiler):-

```
R PLC
program:name
:main
:module1
:module2
:]]
```

{If your installation doesn't have the Prolog compiler in the system (SYS) area, the first line must be .RUN PLC[p,pn], where [p,pn] is the Prolog area. Also note that if the Monitor at your installation doesn't have virtual memory, you should give a core argument when calling the compiler, as described for the interpreter in Section 3.0.}.

Now, the the following message may occur:-

```
No new functors.  Do you still want name.REL?
```

If there is no such file in your area, or if for any reason you think the existing file is not up-to-date, answer "Y", else answer "N", both to be followed by <cr>.

A module name can have the form either file.ext or else just file. In the latter case the extension 'PL' is implied.

During compilation, some files with extension '.TPL' are created. At the end of the compilation, all files with that extension are deleted (so it is better if you do not use that extension for your own purposes!).

To get a core image of the compiled program together with the interpreter (which is like an augmented interpreter), compile the program as described before and then execute the Monitor commands:-

```
LOAD name,main,module1,module2
SAVE name
```

To run your core image, just do:-

```
RUN name
```

{An example of the load sequence is:-

```
LOAD MYJOB,MASTER,SLAVE1,SLAVE2
SAVE MYJOB
```

}.

If you want to recompile some modules of your program, run PLC giving only the names of those modules. When you come to load the core image, the names of all modules must be listed however. Note that a module which has syntax errors is analysed but no object file is produced for it.

If you only want to compile a subset of the modules of a program, and to leave compilation of the others til later, end the compilation with a single "]" instead of "]]". Otherwise time will be wasted in compiling an incomplete version of the functors file. DO, however, make sure that an up-to-date version of the functors file is ultimately compiled, by ending the final compilation before loading with "]]". (NB. If you forget this, unpredictable errors will occur during loading or running the program!)

4.2 Mode Declarations

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

```
:-mode predicate(modes).
```

where predicate is the name of a procedure, and modes specifies the "modes" of its arguments. modes 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 to a NON-variable, while mode '-' specifies that the argument will always be instantiated to a VARIABLE. 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(+,+,-).
```

To have any effect, a mode declaration must appear before the clauses of the procedure it concerns. If, at runtime, a procedure call violates the restriction imposed by a mode declaration, EITHER this will cause an error message to be given and the call to fail, OR the call will proceed normally - which alternative occurs is implementation defined. Mode declarations are ignored by the interpreter.

4.3 Linking Compiled Modules Together

The predicates defined in a module can be made _directly_ accessible to other modules (i.e. without going through the interpreter) if some extra directives are given to the compiler.

To make the predicate _predicate_ of _n_ arguments accessible, the directives:-

```
:-ext(predicate,n,name).
:-entry(predicate,n).
```

should be given in the module where the predicate is defined; and the directive:-

```
:-ext(predicate,n,name).
```

must be given in each module which refers to the predicate. The _name_ must be a unique external name of up to six characters, not containing '$' (and, for the time being, not beginning with 'P'). Upper and lower case letters are treated as equivalent.

NB. The 'ext' directive MUST precede any other reference in the module to the corresponding predicate, INCLUDING references in 'entry', 'module' or 'program' directives. It is therefore best to place all 'ext' directives at the very top of the module.

4.4 Running A Compiled Program Stand-alone

This method should only be used when the program doesn't need the facilities of the interpreter, such as the 'assert' predicate or tracing. The program will have a high-segment containing only the code for the facilities it uses.

No 'program' or 'module' directives should be given.

The starting point (predicate) of the program should have no arguments and should have the external name 'start' given by an 'ext' directive. For example:-

```
:-ext(do,0,start).
:-entry(do,0).

do :- hello,blabla(X),eat(X,Y),write(Y),bye.
                   .
                   .
                   .
```

To compile and load the program, do as described in Section 4.1, except that the switch "/n" should be appended to the program name in the compilation command, eg.:-

```
R PLC
program:MYJOB/N
: etc.
```

5.0 BUILT-IN PROCEDURES

Built-in procedures are also referred to as _evaluable predicates_.

5.1 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 utilized for input and/or output. Terminal output is only displayed after a newline is written or 'ttyflush' is called. When the terminal is waiting for input on a new line, the prompt '|' is displayed.

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

No file except the ersatz file 'user' can be simultaneously open for input and output.

A file is referred to by its name, _written as an atom_, eg.

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

Note that reference to directories other than the user's is not possible at present.

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 issuing a ^Z (decimal 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'.

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. The consulted file may define its own character convention ('LC' or 'NOLC') without affecting the convention prevailing outside.

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', facilitates the amendment of a program without having to consult again from scratch 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 ":-end." or ^Z). This is only recommended for small, tentative patches.

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.

read(_X_)

The next term, delimited by a "fullstop" (ie. a '.' followed by <cr> or a space), is read from the current input stream and unified with _X_. The syntax of the term must accord with current operator declarations. If a call 'read(_X_)' causes the end of the current input stream to be reached, _X_ is unified with the term ":-end". 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.

nl

A new line is started on the current output stream.

display(_X_)

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

ttynl

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

ttyflush

Flushes the terminal output buffer.

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.

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.

putatom(_X_)

The name of atom _X_ is output to the current output stream.

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, closes it and renames it to _N_. If _N_ is '[]', deletes the file.

log

Enables the logging of terminal interaction to file PROLOG.LOG. It is the default.

nolog

Disables the logging of terminal interaction.

5.2 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 _evaluable functors_, integers and variables. At the time of evaluation, each variable in an integer expression must be bound to an integer, or, for the interpreter ONLY, to an integer expression. 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.

Each evaluable functor stands for an arithmetic operation. The evaluable functors are as follows, where X and Y are 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]	evaluates to X if X is an integer (therefore eg. "A" behaves within arithmetic expressions as an integer constant which is the ASCII code for letter A)

The arithmetic built-in procedures are as follows, where 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.

5.3 Convenience

P,Q

P and Q.

P;Q

P or Q.

true

Always succeeds.

X=Y

Defined as if by clause " Z=Z. ".

length(L,N)

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

5.4 Extra Control

!

See Section 2.3.

not(P)

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

```
        not(P) :- P, !, fail.
        not(_).
```

Not yet available for compiled code.

P -> Q; R

Analogous to

"if P then Q else R"

ie. 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 bactracking choices. It behaves (but doesn't use store!) as if defined by the clauses:-

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

fail

Always fails.

abort

Aborts the current execution. Refer to Section 3.11.

5.5 Meta-Logical

var(X)

Tests whether X is currently instantiated to a variable.

nonvar(X)

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

atom(X)

Checks that X is currently instantiated to an atom (ie. 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.

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).

X \== Y

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

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

arg(I,T,X)

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

X=..Y

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

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

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

product =.. [product]
```

If X is instantiated to a variable, then Y must be instantiated to a list of determinate length whose head is atomic (ie. an atom or 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. eg.:-

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

ie. name(product,"product")

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

name(:-,[58,45])

name([],"[]")
```

If X is instantiated to a variable, L must be instantiated to a list of ASCII character codes. eg.:-

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

X = :-

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

X = :-
```

call(X)

If X is instantiated to a term which would be acceptable as body of a clause, the goal 'call(X)' is executed exactly as if that term appeared textually in place of the 'call(X)'. In particular, any cut ('!') occurring in X is interpreted as if it occurred in the body of the clause containing 'call(X)', unless that clause is a compiled clause, in which case only the alternatives in the execution of X are cut. If X is not instantiated as described above, an error message is printed and 'call' fails.

X

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

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 to a non-variable.

asserta(_C_)

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

assertz(_C_)

Like 'assert(_C_)', 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 clauses 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 to a non-variable, and becomes unified with the value of the erased clause. The space occupied by the erased clause will be recovered when instances of the clause are no longer in use.

retractall(_P_)

All clauses in the current interpreted program whose head matches _P_ are 'retract'ed. _P_ must be bound to a non-variable term.

listing(_A_)

Lists in the current output stream all the interpreted clauses for predicates with name _A_, where _A_ is bound to an atom.

listing

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

> _Note_: If a clause contains any atom or functor whose name has to be written in quotes, the listing of that clause will be still readable, but syntactically incorrect. Otherwise, clauses listed to a file by 'listing(_A_)' or 'listing' can be consulted back.

numbervars(_X_,_N_,_M_)

Unifies each of the variables in term X with a special term, so that 'write(X)' prints each of those variables as "_I", where the Is are consecutive integers from N to M-1. N must be instantiated to an integer.

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. Not available for compiled code.

subgoal_of(S)

The goal 'subgoal_of(S)' is equivalent to the sequence of goals:-

ancestors(L), in(S,L)

where the predicate 'in' successively matches its first argument with each of the elements of its second argument. Not available for compiled code.

5.6 Internal Database

These predicates remain in the system purely for compatibility reasons, and will be removed at some future date.

record(X)

The current instance of X is "recorded" in the internal database at some implementation-defined position in the sequence of terms which constitutes the internal database (with new private variables replacing any uninstantiated variables). X must be instantiated to a non-variable.

recorda(X)

Like 'record(X)', except that the new term is "recorded" at the "top" of the internal database.

recordz(X)

Like 'record(X)', except that the new term is "recorded" at the "bottom" of the internal database.

?(X)

The internal database is searched for previously "recorded" terms which match the current instance of X (which must not be a variable). These terms are successively unified with X in the order in which they are recorded in the internal database.

recorded(X,P)

The database is searched for previously "recorded" terms that match the current instance of X (which must not be a variable). These terms are successively unified with X in the order in which they are recorded in the internal database. P is unified with a "pointer" which identifies the "recorded" term matching X. (A "pointer" is a term whose internal structure is implementation-defined).

instance(P,X)

X is unified with the database term identified by "pointer" P.

erase(P)

The database term identified by "pointer" P is erased from the internal database. The space occupied by the erased term will be recovered when instances of the term are no longer in use.

eraseall(X)

All the database terms matching the current instance of X are "erased", in the sense of 'erase(_)'.

5.7 Environmental

'NOLC'

Establishes the "no lower-case" convention described in Section 3.1.

'LC'

Establishes the "full character set" convention described in Section 3.1. It is the default setting.

trace

Enable trace. Consult Section 3.9 .

notrace

Disable trace. It is the default setting.

leash

Enable 'leashed' mode for tracing. It is the default setting.

unleash

Disable 'leashed' mode for tracing.

op(priority,type,name)

Treat name name as an operator of the stated type and priority (refer to Section 2.4). name may also be a list of names in which case all are to be treated as operators of the stated type and priority.

break

Causes the current execution to be interrupted at the next interpreted procedure call. Then the message " % break: " is displayed. The interpreter is then ready to accept input as though it was at top level. To close the break and resume the execution which was suspended, the command " :-end. " or ^Z must be typed. 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 3.8.

save(F)

The system saves the current state of the system into file F. Refer to Sections 3.5 and 3.8 .

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.

restore(F)

The system is returned to the system state previously saved to file F. Refer to Sections 3.6 and 3.8.

maxdepth(D)

Positive integer D specifies the maximum depth, ie. invocation level, beyond which the system 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.

depth(D)

Integer D will give indication of current level of invocation.

gcguide(N)

N must be instantiated to an integer from 0 to 512, indicating the desirable threshold of global stack pages below which garbage collection should be avoided if possible. The default is N=6.

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, in virtual memory Monitors only, releases core no longer needed, thereby reducing the size of the low segment. 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 core 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:-

```
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.

6.0 DEFINITE CLAUSE GRAMMARS

Prolog's grammar rules provide a convenient notation for expressing definite clause grammars [Colmerauer 1975] [Pereira & Warren 1978]. Definite clause grammars are an extension of the well-known context-free grammars.

A grammar rule takes the general form:-

LHS --> RHS.

meaning "a possible form for LHS is RHS". Both RHS and LHS 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), s(Y,S2,S).
```

Terminals are translated using the 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).
```

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, eg.

```
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, eg.

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

becomes:-

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

Disjunction has a fairly obvious translation, eg.

```
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).
```

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

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 _C_s.

(5) A category written as _?C_ denotes an optional _C_. Therefore _?C..._ denotes a sequence of zero or more _C_s.

(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 Section 6.

(7) In Section 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.

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 }
```

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
```

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                --> space-char...

comment              --> /* ?char... */
                           { where ?char... must not contain */ }

full-stop            --> . space-char

char                 --> { any ASCII character, ie. }
                         space-char
                      |  alpha
                      |  symbol-char
                      |  solo-char
                      |  punctuation-char
                      |  quote-char

space-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 _ }
```

7.5 Notes

(1) The expression of precedence 1000 (ie. 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. eg.

    {X} = %(X%)

(5) The character '|' is allowed as an alternative to ',..' in lists, eg.

    [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.

8.0 RESERVED NAMES

Note, in addition to the list of reserved predicates which follows, that names containing the character "$" are reserved for system use and should not be used.

```
abort ancestors(_) arg(_,_,_) assert(_) asserta(_) assertz(_) atom(_)
atomic(_)
break
c(_,_,_) call(_) clause(_,_) close(_) compactcode consult(_)
core_image
depth(_) display(_)
end entry(_) entry(_,_) erase(_) eraseall(_) ext(_) ext(_,_,_)
fail fastcode fileerrors functor(_,_,_)
gc gcguide(_) get(_) get0(_)
halt instance(_,_) integer(_) is(_,_)
'LC' leash length(_) listing listing(_) log
maxdepth(_) mode(_) module(_,_)
name(_,_) nl 'NOLC' nofileerrors nogc nolog nonvar(_) not(_) notrace
numbervars(_,_,_)
op(_,_,_)
program(_,_) put(_) putatom(_)
read(_) reconsult(_) record(_) recorda(_) recorded(_,_) recordz(_)
repeat rename(_,_) restore(_) retract(_) retractall(_)
save(_) see(_) seeing(_) seen skip(_) statistics statistics(_,_)
subgoal_of(_)
tab(_) tell(_) telling(_) told trace trimcore true ttyflush ttyget(_)
ttyget0(_) ttynl ttyput(_) ttyskip(_)
unleash
var(_) version(_,_,_,_)
write(_)
!
','(_,_)
;(_,_)
=(_,_)
<(_,_)
>(_,_)
>=(_,_)
=<(_,_)
=..(_,_)
==(_,_)
\==(_,_)
=:=(_,_)
=\=(_,_)
->(_,_)
```

9.0 EXAMPLES

Some simple examples of Prolog programming are given below. To clearly differentiate the examples themselves, they are marked with a vertical bar in the left margin.

9.1 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).
```

9.2 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, ie.

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

9.3 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([],_,[],[]).
```

9.4 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,^).
|
| 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).
| d(exp(U),X,exp(U)*DU) :-!, d(U,X,DU).
| d(log(U),X,DU/U) :-!, d(U,X,DU).
| d(X,X,1) :-!.
| d(C,X,0) :- atomic(C), C \== 0, !.
```

9.5 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.
eg. ?-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).
```

9.6 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.
eg. 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).
```

9.7 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 (ie. terms). Terms representing clauses are specified using the unary predicate ´clause´, eg.

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

A unit clause will be represented by a term such as:-

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

The mini-interpreter consists of four clauses:-

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

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

9.8 Translating Enlish 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(P)´, 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].
```

10.0 REFERENCES

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

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

Kowalski R A [1974]
"Logic for Problem Solving".
DCL Memo 75, Dept of AI, Edinburgh. Mar 74.

Pereira F C N & Warren D H D [1978]
"Definite Clause Grammars Compared with Augmented Transition Networks".
Forthcoming report, Dept. of AI, Edinburgh.

Robinson J A [1965]
"A Machine-Oriented Logic Based on the Resolution Principle".
JACM vol 12, pp.23-44. 1965.

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

Warren D H D [1977]
"Implementing Prolog - Compiling Predicate Logic Programs".
Research reports 39 & 40, Dept. of AI, Edinburgh. 1977.

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

Lisboa e Laboratório Nacional de Engenharia Civil em Agosto de 1978

VISTO

[illegible] O ENGENHEIRO DIRECTOR

[illegible]

J. Ferry Borges

Luis Moniz Pereira

Luís Moniz Pereira
ESTAGIÁRIO PARA ESPECIALISTA

Fernando Pereira

Fernando Carlos das Neves Pereira
DEPT. OF ARTIFICIAL INTELLIGENCE
UNIVERSIDADE OF EDINBURGH

O CHEFE DA DIVISÃO DE INFORMÁTICA

[illegible]

Carlos Morais
ESPECIALISTA

DHDWarren

David H.D. Warren
DEPT. OF ARTIFICIAL INTELLIGENCE
UNIVERSIDADE OF EDINBURGH