WATERLOO PROLOG USER'S MANUAL
Version 1.4

Grant Roberts

Table of Contents

1 Introduction

Research in artificial intelligence has spurred the development of numerous programming languages better oriented to expressing and solving the problems which arise in this field. One of these languages is PROLOG. The acronym PROLOG is derived from PROgramming in LOGic and emphasizes the derivation of the language from predicate logic. The development of PROLOG represents the discovery of a means for using resolution logic as a practical programming language for problem solving.

The semantics of PROLOG are essentially those of first order resolution logic. Consequently the language is both well defined and compact in definition. More important though, the language is a powerful tool for problem solving, as has been demonstrated in the development of several problem solving systems, among them a geometry theorem prover, natural language understanding systems and a program for automatic plan generation.

The Waterloo implementation of PROLOG for the VM/CMS system is intended to provide an efficient and friendly online interpreter which can be used for educational purposes and program development.

This manual provides an elementary introduction to the PROLOG language and the Waterloo PROLOG implementation. Section 2 of the manual describes the language. Those readers who are familiar with a dialect of the PROLOG language may wish to skip subsections 2.1 to 2.3 and read subsection 2.4 The Syntax in Detail. Subsection 2.5 Using Infinite Terms describes a special facility that is provided for manipulating infinite terms. Section 3 is a reference section which defines in detail the builtin functions (effectively a subroutine library) provided in the implementation. Appendix A describes how to use PROLOG under VM/CMS. Appendix B describes the contents of the standard execution file used to invoke PROLOG. Appendix C explains how to invoke PROLOG for use with terminals that do not support lower case letters.

FIX

2 The Language

2.1 Introduction

The semantics of PROLOG is essentially that of resolution logic. But resolution logic itself does not constitute a programming language. Statements in resolution logic are descriptive. They have the form "x is true". In conventional programming languages the statements are imperative. They have the form "perform action x". To derive a programming language from resolution logic we add imperative statements of the form "prove that x is true". A statement of this form is called a goal statement. A PROLOG program consists of a set of goal statements and a set of axioms. The axioms are descriptive, constituting a list of facts. Each goal statement is imperative and requests that axioms be used in an attempt to prove a certain fact.

To the passive language of axioms we have added the notion of goals to yield a language of action, a programming language. This language now allows us to request the construction of a proof. But how will the attempt at a proof proceed? The proof procedure for PROLOG uses resolution in a simple depth first, left to right search strategy. This proof procedure is not complete. Because of the depth first strategy, a proof may not be found even if one exists in the search space. The proof procedure may follow an infinite branch in the search tree and never examine another branch which could yield a satisfactory proof. However, if the proof procedure terminates, we know that it has found the right answer. If it terminates with success then a proof exists. If it terminates with failure then no proof exists in the search space.

This simple search strategy may seem unsatisfactory since it yields an incomplete proof procedure, but it has numerous advantages over more general strategies. It can be implemented in a manner which is more efficient in the use of space than current breadth first search methods. The simplicity of the PROLOG search strategy makes it easy for the programmer to understand and control the search. The strict ordering of the search permits the use of built-in predicates causing side effects(e.g. read and write) with the knowledge that the side effects will occur in a prescribed order. The prospect of output being

created in random order does not seem very pleasant! Thus, it is evident that the simple search strategy possesses several desirable chracteristics. It is also possible to beg the question of search strategy by stating that if anyone wants a general theorem prover then PROLOG is a good language in which to program it!

2.2 Elementary Syntax

This section introduces the syntax of PROLOG axioms and goals. A brief desciption of the basic syntax is provided in preparation for the description of PROLOG execution in 2.3 Execution and Backtracking. A detailed description of all the syntax rules is then provided in 2.4 The Syntax in Detail.

The basic syntactic unit in PROLOG is the term. A term may be:

(a) a constant - a lower case letter followed by any sequence of letters and digits, or any sequence of digits. A constant may be an integer or an atom. e.g. aBc and x2,

(b) a variable - an asterisk or an upper case letter, followed by a sequence of letters and digits. e.g. * and A1.

(c) a skeleton - a skeleton name and a list of one or more argument terms. The argument terms are separated by commas and the list is enclosed in parentheses. e.g. f(x2,Y) and g(B,a,f(3)).

(d) an infinite term reference. The format is ##n## where n consists of one or more digits.

The syntax can be described in BNF notation:

```
<term>::= <atom>          |
          <integer>       |
          <variable>      |
          <skeleton>      |
          <infinite term> |
          ( <term> )
<atom>::= <identifier>
<skeleton>::= <identifier> ( <argument list> )   |
              <term> <infix operator> <term>     |
              <prefix operator> <term>           |
               <term> <suffix operator>
<infix operator>::= <identifier>
<prefix operator>::= <identifier>
```

```
<suffix operator>::= <identifier>
<argument list>::= <term>   |
                   <argument list> , <term>
<variable>::= *     |    <upper case letter> |
              <variable> <letter>             |
              <variable> <digit>
<infinite term>::= ## <digit#> ##
<digit>::= <digit> |
           <digit#> <digit>
```

The rules involving operators describe an alternative notation for skeletons, to be described in 2.4 The Syntax in Detail.

PROLOG axioms and goals are composed of literals. A literal may be a skeleton or a constant. A predicate is the name associated with a literal. If the literal is a skeleton then the predicate is the skeleton name. Otherwise it is the constant associated with the literal.

The general form of a PROLOG axiom is:

<axiom head> <- <axiom body> .

The implication arrow, "<-" is read "is implied by". The axiom head is a single literal. The axiom body is a conjunction of literals. A conjunction of literals may be a single literal or two or more literals separated by the "and" symbol(&). An example of an axiom is:

a <- b & c .

The head is a, the body is b & c and the axiom is read "a is implied by b and c" or "To prove a, first prove b, then prove c". An axiom may have a null body, in which case the implication is omitted and the axiom has the form:

<axiom head> .

An axiom with a null body is called a unit axiom. An example is:

f(m).

This is read "f(m) is true".

The general form of a PROLOG goal is:

<- <goal conjunction>.

The goal conjunction is a single literal or a conjunction of literals. Examples of goals are :

<-p.

<-q(r) & f .

Goal statements may be regarded as abbreviations for axioms of the form:

"goal" <- <goal conjunction>

where "goal" is a distinguished literal which the PROLOG theorem prover attempts to "prove".

From the user point of view the PROLOG system accepts axioms and goals from the terminal. Axioms which are entered are recorded for later use in proofs. An attempt is made to prove a goal statement as soon as it is entered. In the following discussion, goals will always be presented in the form <- <goal conjunction>. When actually using the PROLOG system an abbreviated goal format is available. Refer to Appendix A : Using PROLOG under VM/CMS for further explanation before using PROLOG at a terminal.

In axioms and terms all variables are assumed to be universally quantified. That is, an axiom containing variables is valid for any "values" which the variables may take on. A verbal version of the axiom "father(X,Y) <- son(Y,X)" is "For all values of X and Y, X is the father of Y if Y is the son of X". The substituting of "values" for variables will be discussed further in the next section.

2.3 Execution and Backtracking

PROLOG execution is started by a goal statement. A goal statement is a request for a proof. The execution of a PROLOG program is essentially the actions of an elementary theorem prover attempting a proof.

A series of diagrams may be used to describe the progress of a PROLOG proof. Each diagram, called an implication tree , describes the state of the proof at a given point in time. An implication tree consists of one or more labelled nodes. At the top of the diagram is a node labelled "goal". Each of the other nodes is labelled with a literal and is joined to a parent node immmediately above it. A node is called the child of its parent. A node may be in any one of three states:

(1) open: No attempt has been made to prove the literal labelling the node. The node has no children.

(2) closed: The literal labelling the node has been proven using a unit axiom for the literal. The node is marked with an "X" to distinguish it from an open node. A closed node has no children.

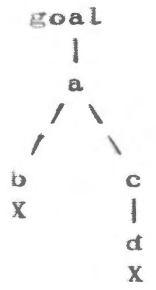
(3) active: The literal labelling the node is

being proven (or has been proven) using a non-unit axiom. The node is labelled with the literal of the axiom head. The children of the node are labelled with the literals of the axiom body. The left-to-right order of the literals in the axiom body is preserved in the diagram. The original goal statement is treated as an axiom of the form "goal <- <goal conjunction>". Thus the children of the goal node are labelled with the literals of the goal conjunction.

Consider the following axioms and goal:

```
a<-b&c.
b.
c<-d.
d.
<-a.
```

The proof of this goal is represented by the following implication tree:



This is a <u>completed implication tree</u> since all nodes are either active or closed. The nodes labelled b and d have been closed using axioms "b." and "d." respectively. The node labelled a is active and has been proven using the axiom "a<-b&c.".

Consider the following example of axioms and a goal statement:

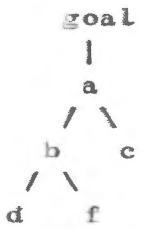
```
a<-b&c.
b<-d&f.
b<-e&f.
c<-g.
e<-g.
f<-h.
g.
```

h.

The initial state of the proof is represented as:

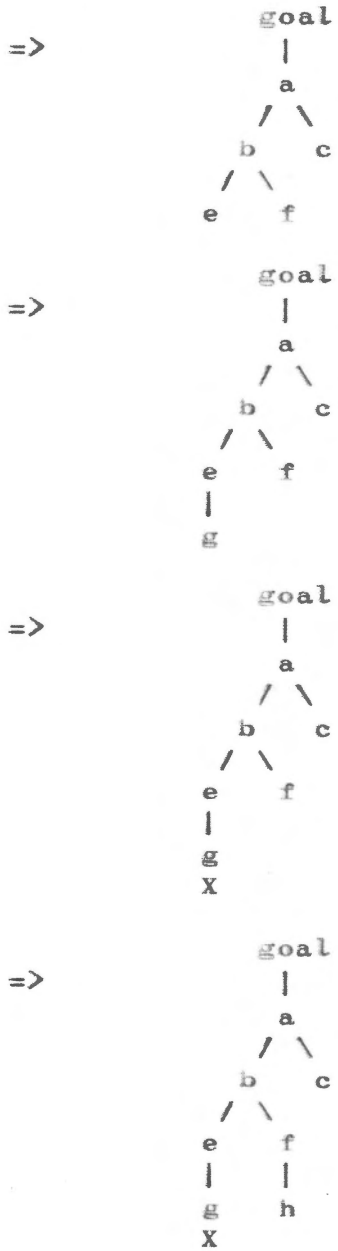
The first axiom for a is selected, namely a<-b&c giving:

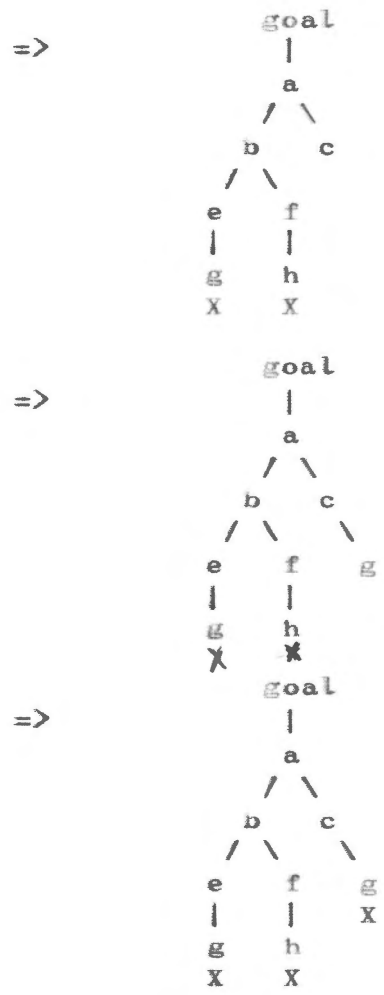
The prover always works in a depth-first left-to-right fashion. Consequently the next literal to be proven is b. The axiom b<-d&f is selected:



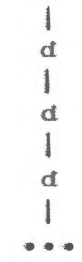
The prover then attempts to prove d. But there are no axioms for d so the prover must backtrack. This involves backing up the proof and trying other alternatives. A <u>choice point</u> in the proof is a point where an axiom was chosen to prove a literal and more axioms remain to be tried. <u>Backtracking</u> involves backing up the proof to the most recent choice point and making a different choice. The order in which the axioms are chosen is not arbitrary. Axioms are always selected in the order in which they appear in the input. In this example b<-d&f will always be examined before b<-e&f.

The most recent choice point in the current proof is the point where the axiom b<-d&f was selected. The proof is backed up to this point and the other axiom, b<-e&f, is selected. The proof continues as shown below:


=>
goal
a
b
c
e
f
=>
goal
a
b
c
e
f
g
=>
goal
a
b
c
e
f
g
X
=>
goal
a
b
c
e
f
g
h
X



The final proof is represented by a completed implication tree. Of course, if the proof fails then the implication tree is never completed. If, in this example, we omit the axiom c<-g then the proof attempt will fail. Alternatively, if we include another axiom d<-d then the prover will attempt to construct an "infinite branch" of the implication tree:



Eventually an error will occur when the proof stack overflows.

In the previous examples, none of the predicates have arguments. For example, the predicate term father(john,fred) has two arguments, john and fred, and can be used to represent the statement "john is the father of fred". PROLOG axioms can also contain variables. For example the axiom son(X,Y)<-father(Y,X) represents the statement "x is the son of y if y is the father of x". Variables in PROLOG are assumed to be universally quantified. That is,an axiom containing a variable is considered to be "true" for any "values" the variable may take. We will make the idea of a variable "taking a value" more precise. In any axiom or goal we can perform a substitution. A substitution replaces all occurrences of a variable by a term. The replacing term may be a constant (such as abc or 32), a skeleton(such as f(a) or g(X,Y)) or another variable. For example, if we substitute a for X in g(X,f(X)) then the resulting term is g(a,f(a)). If we substitute f(Y) for X in h(X,Y) then the result is h(f(Y),Y). When one or more substitutions are applied to a term (or axiom), the result is called an instance of the term (or axiom). For example, son(fred,john)<-father(john,fred) is an instance of son(X,Y)<-father(Y,X) produced by substituting fred for X and john for Y.

To illustrate substitution better, consider the following example:

```
son(X,Y)<-father(Y,X).
father(john,fred).
father(john,george).
father(al,bert).
father(george,al).
```

We wish to solve the goal "<-son(Z,john)". By "solving a goal" we mean finding an instance of the goal which we can prove. In this case we will prove

"son(fred,john)". The proof will be illustrated using implication trees. The initial tree is:

```
        goal
         |
      son(Z,john)
```

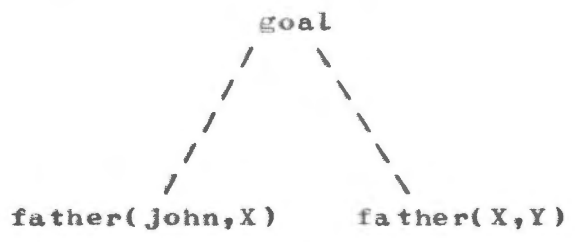
Now we need to find an instance of an axiom which we can use in the proof of son(Z,john). The appropriate instance is formed from son(X,Y)<-father(Y,X) by substituting Z for X and john for Y to give son(Z,john)<-father(john,Z). The tree now is:

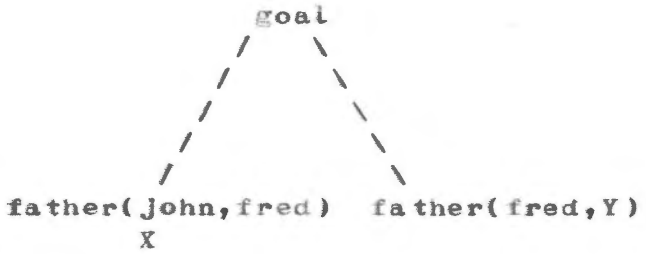
```
        goal
         |
      son(Z,john)
         |
     father(john,Z)
```

Note that we found substitutions that made the head of an axiom the same as the current subterm. The general process of finding substitutions to make two terms the same is called unification. Next we want to find an axiom whose head will unify with father(john,Z). The first axiom for father matches if we substitute fred for Z. This gives the completed implication tree:

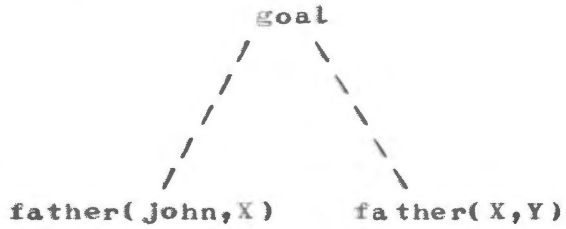
```
        goal
         |
      son(fred,john)
         |
     father(john,fred)
         X
```

As a further example we will attempt to solve the goal <-father(john,X)&father(X,Y). The proof proceeds as follows:

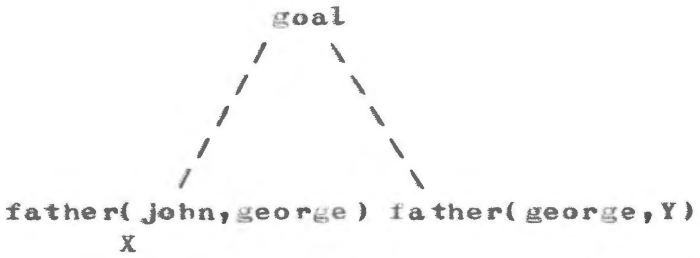




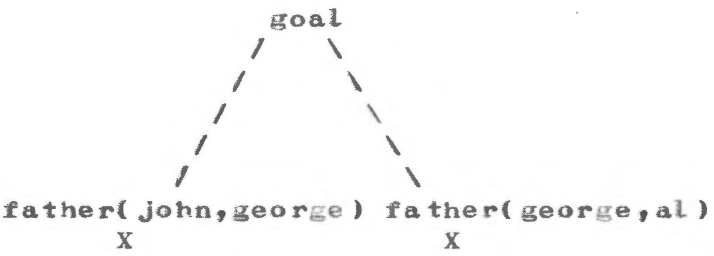
The attempt to solve the subgoal father(fred,Y) fails since this term will not unify with any of the axiom heads. Backtracking occurs and the proof is backed up to the point where the father(john,fred) axiom was activated. This axiom is then deactivated and any substitutions made when (or since) this axiom was selected are "undone". This restores the proof to the point:



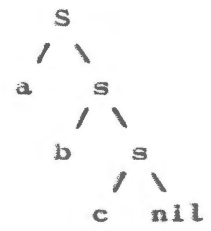
The axiom father(john,george) is about to be selected for unification with father(john,X). This unification succeeds giving:



The axioms for father are then selected in turn for unification with father(george,Y). The unification succeeds for the axiom father(george,al), yielding the completed implication tree:



To illustrate the operation of PROLOG further, the following examples demonstrate the manipulation of more complex data structures. A set of elements (similar to a LISP list) is represented by a term using a constructor s and an end marker nil. For example, the set with elements a,b and c is represented by s(a,s(b,s(c,nil))) or as a diagram:



The empty set is represented by nil. This notation is completely arbitrary and is chosen for this example only.

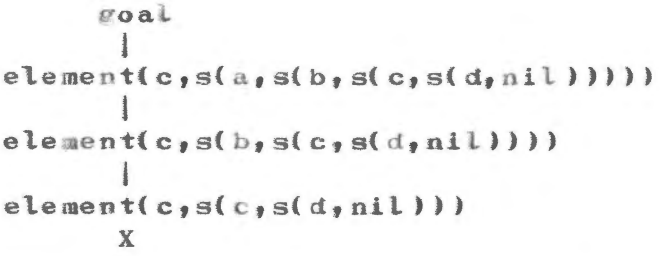
A reasonable definition for the "element" relation is:

```
element(X,s(X,Y)).
element(X,s(Y,Z))<-element(X,Z).
```

Verbally these axioms might be stated as "X is an element of a set if it is the first element in the set or if it is an element of the set of elements following the first element.". The goal

```
<-element(c,s(a,s(b,s(c,s(d,nil)))))
```

yields the following completed implication tree:



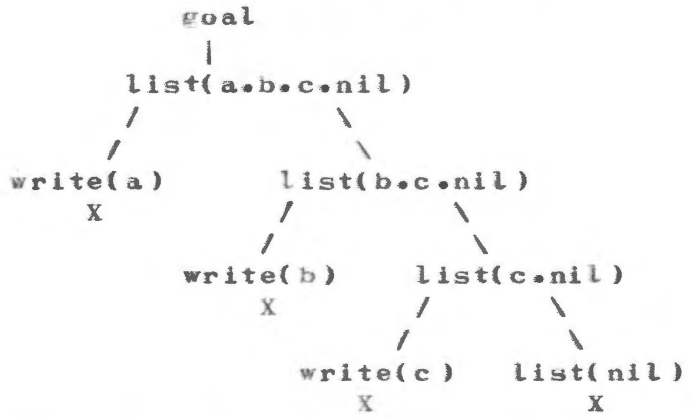
This syntax for representing sets is clearly cumbersome. To simplify this, infix notation may be used(infix, prefix and suffix notation are explained more fully in 2.4 The Syntax in Detail). If we use a "." as the constructor and use infix notation then we can denote the set with elements a,b and c by a.b.c.nil. The axioms for element become:

```
element(X,X.Y).
element(X,Y.Z)<-element(X,Z).
```

Suppose we want an axiom to write all the elements of a set. The following axioms will suffice:

```
list(X.Y)<-write(X)&list(Y).
list(nil).
```

write is a built-in predicate which always succeeds and has the side effect of displaying its argument term on the terminal. The term is written followed by a period (the end of term delimiter). The goal statement <-list(a.b.c.nil) succeeds. The completed implication tree is:



The output on the terminal is:

a.

b.
c.

The following axiom could also be used to list the elements of a set on the terminal:

```
list(X.Y)<-write(X)&FAIL.
list(X.Y)<-list(Y).
list(nil).
```

The goal <-list(a.b.c.nil) will list all elements of the indicated set and then succeed. The completed implication tree is:

```
        goal
         |
    list(a.b.c.nil)
         |
    list(b.c.nil)
         |
    list(c.nil)
         |
    list(nil)
         X
```

Suppose we wish to define axioms for a predicate notel(X,Y) which succeeds if X is not an element of Y. Reasonable axioms for this predicate might be:

```
notel(X,nil).
notel(X,Y.Z) <- noteq(X,Y)&notel(X,Z).
```

Verbally these axioms might be stated:

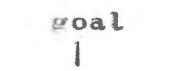
"X is not an element of the empty set".

"X is not an element of the set consisting of Y and some other elements if X is not equal to Y and X is not an element of the set of other elements".

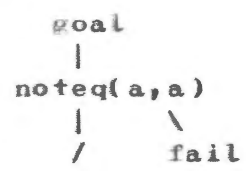
The axioms for noteq remain to be defined. The axioms are:

```
noteq(X,X)<- / & fail.
noteq(X,Y).
```

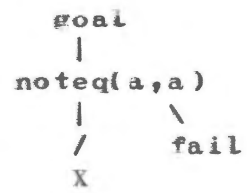
These axioms make use of a special control feature, the slash(/). To illustrate this feature we trace the attempt to prove the goal <-noteq(a,a). Initially, we have:



The first axiom is selected giving:



The slash predicate always succeeds. It is used to prevent certain alternatives from being considered in the proof. In this case it prevents the second axiom for noteq from being considered. The implication tree looks like:



The fail predicate has no axioms and consequently it fails. Since the remaining axiom for noteq is not considered, there are no remaining choice points and the entire proof fails.

Conversely the goal <-noteq(a,b) succeeds. The head of the axiom noteq(X,X) <- / & fail cannot be unified with noteq(a,b) so the next axiom is selected. The unification succeeds and the proof is complete.

The action of the slash predicate is described more precisely: When the slash predicate is executed, it removes all choice points in the proof from the point when the axiom containing the slash was selected to the current point in the proof.

The slash predicate is utilized for two main purposes. The first is to affect the meaning of an axiom, often to handle negation as in noteq above. The second use is to improve the efficiency of a program by preventing spurious choices from being considered. For example, consider the following axiom used to test if

two sets have one or more common elements:

```
intersect(A,B)<-element(X,A) & element(X,B).
```

If a call to the intersect predicate succeeds and then backtracking returns to that point, then the element axioms will cause other choices for X to be tried. Normally the attempt to find a different common element is completely unnecessary since it has already been proven that A and B intersect. This extra search can be eliminated by using the following axiom for intersect:

```
intersect(A,B)<-element(X,A) & element(X,B) & /.
```

2.4 The Syntax in Detail

A PROLOG program consists of a sequence of symbols belonging to a symbol vocabulary. In this implementation the EBCDIC character set is used. Any one byte value is a valid symbol, even though it may not have an explicit EBCDIC graphic code. These symbols are divided into four groups as follows:

(a) Letters - The upper and lower case letters from A to Z.

(b) Digits - The digits from 0 to 9.

(c) Punctuation Symbols - This group consists of the left and right parentheses, the comma, the apostrophe, the quote and the end-of-term symbol(the period).

(d) The Underscore - This symbol can be used in constants and variable names.

(e) Special Symbols - This group consists of all symbols not in any of the four preceding categories.

The fundamental syntactic construct in PROLOG is the term. As stated earlier, a term may be a variable, a constant or a skeleton.

A variable is represented by ~~a variable~~ or asterisk(*) ~~name. The variable name is~~ an upper case letter followed by a sequence of letters and digits. Thus X, A1B2C3, ~~and~~ *aB1 and Abc are all variables. In addition a single asterisk(*) is a variable of a special sort. It is called an anonymous variable and has the special significance that each occurrence is considered to represent a distinct variable.

A constant is a sequence of symbols enclosed in

apostrophes. The sequence represents the value of the constant. Note that if the value contains an apostrophe, then the apostrophe must be duplicated. Examples of constants are:

```
'ABC'
'37+A)'
')'',''
''
```

The value of the third constant shown above consists of the three symbols right parenthesis, apostrophe and comma, in that order. The value of the last constant consists of no symbols. The apostrophes enclosing a constant are not always required. They may be omitted if any of the following conditions are satisfied:

1/ The value of the constant consists entirely of symbols which are letters, digits and underscores, and the initial symbol is not an upper case letter.

2/ The value of the constant consists of one symbol which is not a punctuation symbol.

3/ The value of the constant consists of the single period symbol and the constant is not followed by a blank.

4/ The value of the constant consists of up to eight special characters and there is an operator declaration for the value in the data base. For a further explanation of operator declarations, see 3.5 Database Predicates in conjunction with the OP built-in predicate. Note that underscores are not special symbols.

Integers are constants whose values satisfy certain criteria. A constant is an integer if and only if it satisfies any of the following:

1/ Its value consists of one or more digits.

2/ Its value consists of the symbol "+" followed by one or more digits.

3/ Its value consists of the symbol "-" followed by one or more digits.

Integers may be used as arguments to several built-in predicates which perform the fundamental operations of integer arithmetic. Two integer constants are

The value of the constant consists entirely of special characters (with maximum length eight)

equal(i.e. indistinguishable) if their values are the same after any "+" symbols and leading zeroes have been dropped. Thus 001, '+0001' and 1 are all equal integers. Note that signed integers must be enclosed in apostrophes.
A constant which is not an integer is an atom. ab, 'AB(', '+' and '' are all atoms. A sequence of symbols which satisfy the criteria for an atom is called an identifier.

A skeleton consists of an identifier and one or more argument terms. Both predicates and functions are represented as skeletons. A skeleton has the following format:

<identifier> (<argument list>)

The argument list consists of one or more terms separated by commas. Examples of skeletons are:

```
fact(1)
g(1,X,f(1))
'A/.)'(X,Y)
```

Note that any of the argument terms of a skeleton may in turn be skeletons.

To permit a more convenient representation for skeletons, identifiers can be declared as infix, prefix or suffix. For example, if the identifier likes is declared as infix then the skeleton represented as likes(a,b) can also be represented as a likes b. Similarly, if the identifier ! is declared as suffix then !(a) can be represented as a!.

An identifier used as the skeleton identifier in infix, prefix or suffix form is called an operator. The use of operator notation is provided in addition to the basic notation for skeletons which was first described. The two forms may be mixed freely. For example, if likes is declared as infix then f(a likes b,likes(c,d)) is a perfectly acceptable form. A term is represented in canonical form when it is represented without using infix, prefix, or suffix notation.

In any term, subterms may be parenthesized to indicate the term structure. For example:

a+(b-c) is equivalent to +(a,-(b,c))
but (a+b)-c is equivalent to -(+(a,b),c).

insert

Any term or subterm may be parenthesized. If likes is infix then ((a)) likes (c likes(d)) is a valid term equivalent to likes(a,likes(c,d)).

An identifier can be declared as both prefix and infix simultaneously but an identifier which is declared as suffix can not be declared as infix or prefix. An identifier is declared by adding an operator declaration axiom. The format for ~~the axiom to be added is:~~ an operator declaration is:

op(<identifier>,<type>,<priority>).

<identifier> is the identifier to be declared.
<type> specifies the declaration type and may be any of: prefix, suffix, lr, rl.
<priority> is a positive integer less than or equal to 1000.

The declaration types of suffix and prefix have an obvious interpretation. The types rl and lr are used to declare operators as infix right-to-left and left-to-right respectively. For example, if "." is declared as rl then

a.b.nil is equivalent to a.(b.nil)
and to .(a,.(b,nil))

If "+" is declared as lr then

a+b+c is equivalent to (a+b)+c
and to +(+(a,b),c)

The priority specified in the declarations gives the position of the declarations in a priority hierarchy. The larger the numeric priority the stronger the "binding" of the operator. The following examples illustrate the function of the priority. For these examples assume that the following declarations are in effect:

```
op(¬,prefix,40).
op(!,suffix,70).
op(.,rl,50).
op(+,lr,60).
op(-,lr,60).
```

Then:

¬a! is equivalent to ¬(a!)
a+b-c.d+e.f is equivalent to ((a+b)-c).((d+e).f)

¬a+b! is equivalent to ¬(a+(b!))

The problem of resolving the case where two identifiers have equal priorities but different declaration types has not yet been discussed. For instance if the declarations in effect are:

```
op(+,lr,60).
op(-,rl,60).
```

then how is a+b-c to be interpreted ?
The rule for resolving such conflicts is:

> If the rightmost operator is declared rl and the leftmost operator is prefix(or rl) then treat the rightmost binding as the strongest.
> Otherwise treat the leftmost binding as strongest.

The example a+b-c is equivalent to (a+b)-c. This detail is confusing, and it is recommended that the user not declare operators with the same priorities and different types and hence avoid the condition completely. The above description is included solely for the sake of completeness.

The initial state of the PROLOG system includes several operator declarations, namely:

```
op(<-,rl,10).
op(<-,prefix,10).
op(|,rl,20).
op(&,rl,30).
op(¬,prefix,40).
op(.,rl,100).
```

Operator declarations can be added and deleted by adding and deleting axioms for the op predicate as described in 3.5 Database Predicates.

An input term must be delimited by an end-of-term character. The period is used. To distinguish between the use of the period as an operator and its use as the end of term character, the following rules are used. A period that is not enclosed in apostrophes, double quotes or comment delimiters is treated as an end of term delimiter if:

(a) it is followed immediately by one or more

blanks or

(b) it is the last character of an input line. (By line we mean either an input line from the terminal or an input record from a file).

Blanks may be freely used in the input term, subject to the following conditions:

(a) Blanks may not be used internal to an unquoted identifier or constant (e.g. ab is different from a b since ab is a single identifier and a b represents two identifers, namely a followed by b).

(b) Blanks may be used in a quoted constant or identifier but they are included in the value of the constant(e.g. 'A B' is not the same constant as 'AB').

(c) One or more blanks must be used to separate the following:

(1) two quoted identifiers or constants(e.g. 'A''B' represents a constant with value A'B whereas 'A' 'B' represents two constants with values A and B respectively).

(2) two unquoted identifiers or constants where neither consists solely of special characters(e.g. a; is equivalent to a ; but a12 is not equivalent to a 12).

(d) Blanks must not be used after a period except where the period is an end-of-term delimiter.

Whenever one or more blanks may be used, a comment may be inserted. A <u>comment</u> has the form:

/*<comment characters>*/

<comment characters> may be any sequence of characters not including an asterisk followed immediately by a slash. Note that this format for a comment implies that if / is declared as a prefix or infix operator and is used followed by a variable then a blank must appear between the / and the * of the variable. To help detect errors caused by an improperly closed comment a warning message is issued if a /* is encountered in a comment.

Axiom and goal statements are special cases of terms. They are read and parsed using the operator

declarations. Thus the axiom a<-b&c could also have been entered as <-(a,&(b,c)). A goal statement is a term of the form:

<-(<goal conjunction>).

An axiom is a term of the form:

<-(<head>,<goal conjunction>).
or <head>.

<head> can be an atom or a skeleton.
e.g. a
a(1,X)
'B:'(*)
<goal conjunction> can have the form
<goal literal>
or the form
&(<goal literal>,<goal conjunction>)
<goal literal> can be an atom,skeleton or a variable. A variable goal literal is called a meta variable and is described in 3.6 Execution Control Predicates.

A list of terms is formed with the list constructor "." and the end-of-list marker nil. For example the list with elements a, b and c is represented as a.b.c.nil or in canonical form as .(a,.(b,.(c,nil))). The empty list is represented as nil. A string is a list of characters, or more precisely, a list of constants each with a single character value. An abbreviated format is provided to represent strings. The format is:

"<characters>"

For example:
"abc" is equivalent to a.b.c.nil.
"( )" is equivalent to '('.')'.nil.
An empty list may also be specified:
"" is equivalent to nil.
Note that "ab" is equivalent to .(a,.(b,nil)) only if the period is declared as infix right-to-left.

2.5 Using Infinite Terms

After you have become familiar with PROLOG, you may ask what happens when a PROLOG program attempts a unification such as f(X) with X. One possible approach would be to have the unification fail, rather than construct a 'loop'. It turns out that allowing this sort of loop provides a useful capability for manipulating 'infinite' terms. The term resulting from unifying f(X) and X is f(f(f(...))).

This PROLOG implementation allows representations of infinite terms to be read, written, unified or even added to the database. A special notation is introduced to simplify the representation. The notation is simple but not necessarily easy to read. The best notation for understanding these is a two dimension graph style notation. Unfortunately, such a notation is not well suited to conventional input and output devices.

The notation chosen for this implementation, extends the standard term notation, by allowing an 'infinite term specification' or 'loop specification' to be used in place of any subterm. The loop specification has the format ##n## where n is the length of the loop. The term resulting from the unification of X and f(X) would be represented as f(##1##). Similarly if we unify X and f(g(1,X,h)) then the result would be represented as f(g(1,##2##,h)). Any output term, containing a loop, will be written using this notation. Similarly, any term being read in may contain subterms using this notation. If an input term contains an invalid infinite term reference, for example too large a loop length, then an error message will be printed and the term rejected. For example the term 'f(g(##3##))' is invalid.

When a term is written, the loop length specified is not necessarily minimal. For example, if the result of unifying X and f(f(X)) is printed, the format will be f(f(##2##)), even though the format f(##1##) would be more compact.

Axioms containing infinite term references may be added to the data base. However, infinite conjunctions on the right hand side of an axiom are not permitted. For example, a(##1##)<-b is valid but a(X) <- b & ##1##

is invalid.

3 Built-in Predicates

3.1 Introduction

The implementation provides several built-in predicates. These predicates provide facilities which are either impossible or inconvenient for the programmer to implement directly in PROLOG. Many built-in predicates have side effects, particularly those associated with input and output. The built-in predicates can succeed or fail, exactly as other predicates do. They can also terminate with an error message if the arguments are inappropriate.

In general, it is not possible to add axioms for built-in predicates, since they have a fixed definition. The op and trace predicates represent one type of exception to this, in that axioms for these predicates may be added or deleted but the presence of trace or op axioms in the database have side effects. The other type of built-in predicate which allow addition and deletion include error and attn predicates. Error and attn are special interfaces provided to invoke user axioms when exceptional conditions occur. ~~built-in predicate described in 3.5 Database Predicates.~~

The built-in predicates are divided into six groups. The groups and their members are:

Structural Predicates - atom, cons, int, skel, string, var, argument, functor

Input/Output predicates - fileclose, newline, read, readch, readempty, tab, write, writech, writeq

Arithmetic Predicates - diff, prod, quot, rem, sum, random

Database Predicates - addax, ax, axn, control, delax, op, freeax_

Execution Control Predicates - ancestor, retry, /, &, |, fail, repeat, error, attn, stop, meta variable, trace, systrace_

Miscellaneous Predicates - digit, letter, upshift, system, eq, ge, gt, le, lt, ne

The predicates of each of the above groups are described in the following sections.

3.2 Structural Predicates

These predicates provide for altering and testing the structure of terms. The predicates are atom, int, var, skel, cons, string, argument, and functor

atom, int, var, and skel each have a single argument. If the argument is of the type specified by the predicate name, namely an atom, integer, variable or skeleton ~~respectively~~, then the predicate succeeds. Otherwise, the predicate fails. In no case is any substitution performed or are any error messages produced.

Example:

```
test(X)<-int(X)&testint(X).
test(X)<-atom(X)&testatom(X).
/* use testint to process an integer and testatom
    to process an atom */
```

See cons predicate

Suppose we wish to define an axiom which is passed a skeleton and prints the skeleton name. In order to do this we need the cons predicate. cons is used to decompose a skeleton into a list consisting of the skeleton name followed by its arguments. For example the call <-cons(X,a(b)) will cause X to be unified with a.b.nil. cons may also be used to construct a skeleton term from a list consisting of the skeleton name followed by its arguments. For example, the call <-cons(f.X.3.nil,Y) unifies Y with f(X,3). cons treats a constant as a skeleton of 0 arguments, as shown in the examples below. If the second argument is not a variable then a list consisting of the skeleton name followed by its arguments is unified with the first argument. If the second argument is a variable then a skeleton is constructed from the first argument and unified with the second argument. In this case the first argument must be a list whose first element is a constant and whose remaining elements are to be the arguments. If the first element of the list is an integer then there must be no more elements in the list, since an integer is not a valid skeleton name.

Examples:

The following calls succeed.

```
<-cons(atom.nil,atom).
<-cons(10.nil,10).
<-cons(a.b(c).d.X.nil,a(b(c),d,X)).
```

The following axiom accepts a skeleton as a first argument and returns in the second argument a skeleton like the first but with an initial argument of 99 added.

```
expand(Sk1,Sk2)<- cons(N.Args,Sk1) &
                  cons(N.99.Args,Sk2).
```

Suppose we wish to determine if a constant contains the letter a in its value. If the first argument of the string predicate is a constant then the second argument is unified with the list of characters in the value of the constant. The following axioms define a predicate constanta(X) which succeeds if X is a constant containing an a.

```
constanta(Con) <- string(Con,List) &
                  lista(List).
lista(a.Rest).
lista(First.Rest) <- lista(Rest).
```

The string predicate may also be used to compose a constant from the list of symbols in its value. There are two possible formats for a call to string:

(a) the first argument is a constant, the constant is decomposed to create a list whose elements are the symbols in the constant's value. This list is unified with the second argument.

(b) the first argument is a variable, the second argument must be a list of zero or more elements, each element is a constant with a value consisting of a single symbol. The first argument is unified with the constant whose value consists of the symbols in the list.

If the arguments are other than as prescribed, an error message is generated. Examples:

The following calls succeed.

```
<-string('ABC',"ABC").
<-string('',nil).
<-string(abc,a.b.c.nil).
<-string(0012,1.2.nil).
```

The following predicate accepts a constant as a first argument and produces the second argument by prefixing the first with a q.

```
append(In,Out) <- string(In,S) &
                  string(Out,q.S).
```

The argument predicate can be used to select the argument of a skeleton corresponding to an appropriate index. For example, the goal:

```
<-argument(f(1,8,27,64),3,Cube).
```

will succeed and unify Cube with 27. Similarly the goal:

```
<-argument(f(1,8,27,64),X,64).
```

will unify X with 4, namely the index of the argument which unifies with 64. The argument predicate is always called with three parameters. The first argument must be a skeleton or atom or else an error will result. If the first argument is an atom it is treated as a skeleton with zero arguments and the predicate simply fails. The second argument may be an integer or a free variable. An attempt is made to unify the second argument with each successive index from 1 to the arity of the skeleton. For each index the third parameter is unified with the corresponding skeleton argument. The argument predicate behaves as though it were defined by the following axioms:

```
argument(Skeleton,Index,Arg) <-
                    cons(Name.Arglist,Skeleton) &
                    atom(Name) &
                    list_index(Arg,Arglist,Index).

list_index(Arg,Arg.List,1).
list_index(Arg,*.List,N) <- list_index(Arg,List,M) &
                            sum(M,1,N).
```

3.3 Input/Output Predicates

Input/Output predicates are provided to allow a PROLOG program access to external data. A file is identified by a constant whose value is the file

explain functor (insert #1)

identifier. A file identifier may consist of from 1 to 8 characters, of which the first must be a letter and the remainder must be digits or letters. The file identifier is converted to uppercase by all input/output predicates, since many file systems do not allow lower case file names. The input/output predicates each have an optional file identifier argument. If this argument is omitted the main input/output stream is assumed (i.e. the terminal for an interactive session). The file identifier is optional for all input/output predicates except the fileclose predicate, for which it is manditory. Several of the input/output predicates may also have an optional record number argument. This record number may be a positive integer and is used to position to the appropriate record in the file before performing the indicated operation.

read is a predicate with one, two or three arguments. The second argument is the optional file identifier. The third argument is an optional record number. It must be a positive integer, indicating where in the file the read is to start. The first record in the file has a record number of 1. A term is read from the indicated file and unified with the first argument. The term must be delimited with the end of term character. If the end of the input file has been reached the predicate fails. If backtracking returns to the read then a read of the next term will be attempted. If the term read cannot be unified with the first argument or the format of the term is invalid then backtracking will cause a read of the next term to be attempted.

write is a predicate with one, two or three arguments. The second argument is the optional file identifier. The third argument is an optional record number. The term specified by the first argument is written on the indicated file. The term is delimited by the end of term character. The term is written using prefix, infix and suffix notation where appropriate, as indicated by the operator declarations at the time of writing.

writeq is a predicate with one, two or three arguments. It functions in a manner very similar to write. The only difference occurs in the format of the written output. writeq encloses identifiers in quotes(i.e apostrophes) as required, to ensure that the

written term can be read back in by the read predicate. Thus any identifiers containing blanks, punctuation symbols, etc. will be written enclosed in apostrophes.

readch is a predicate with one, two or three arguments. The second argument is the optional file identifier. The third argument is an optional record number. A single character is read from the given file. The constant whose value is the single character is unified with the first argument. If the end of an input line (or record) has been reached then the first character of the next line (or record) is read. If the end of the input file has been reached then the predicate fails. If backtracking subsequently returns to this point or if the unification of the first argument and the character fails, then the next character in the input file is read and the unification reattempted.

The readempty predicate is provided for use in conjunction with the readch predicate. It allows record boundaries to be detected when reading a character at a time. readempty is a predicate with one optional argument - the file identifier. The readempty predicate succeeds if the input buffer is empty(i.e. the next readch will cause a new physical record to be read).

writech is a predicate with one, two or three arguments. The second argument is the optional file identifier. The third argument is an optional record number. The first argument specifies a term which is formatted using the operator declarations (as for write) and placed in the output buffer for the given file. If the buffer is filled then it is written to the given file (and emptied). If the buffer is partially filled then it is not written out. Note that the readch and writech predicates are not symmetrical. The writech predicate can be used to write a single character but it is considerably more general than readch.

newline is a predicate with one optional argument. The argument is the file identifier. It writes the current output buffer to the given file and empties the buffer. newline is used in conjunction with writech. For example, the goal statement:

```
<-writech('on ') & writech(one) &
  writech(' line.') & newline.
```

causes the following to be written on the terminal:

on one line.

Note that this output is identical to that produced by the call

```
<-write('on one line').
```

or by the call

```
<-writech('on o') & write('ne line').
```

fileclose is a predicate with one argument - a file identifier. fileclose may be used to logically close a file so that it may be reread from the beginning. Note that when a file is used for input after output, the file is automatically closed so that the first input will be from the beginning of the file. In a similar manner, output after input will cause an automatic close. Output to an existing file will be appended to the end of the file.

tab is a predicate with one or two arguments. The second argument is the optional file identifier. The first argument must be a non-negative integer. It specifies the number of blanks to be written on the output file.

3.4 Arithmetic Predicates

There are several predicates which are included to provide the basic operations of integer arithmetic. Each predicate has three arguments. The first two are the input parameters and the last is the result parameter. The first two arguments must be integers. The appropriate integer function of the first arguments is unified with the third argument.

The arithmetic predicates are:

- diff - difference (subtraction)
- prod - product
- quot - quotient
- rem - remainder
- sum - sum
- random - random generator

TAKE OUT

The following axioms define a predicate which calculates the factorial function of its first argument.

```
fact(0,1).
fact(X,Y)<- diff(X,1,X1) &
            fact(X1,Y1) &
```

```
                    prod(X,Y1,Y).
```

The following calls succeed:

```
        <-diff(3,2,1).
        <-prod(10,20,200).
        <-quot(205,10,20).
        <-rem(205,10,5).
        <-sum(1,20,21).
```

explain random (insert #2)

3.5 Database Predicates

The database built-in predicates provide the facility for updating the database (i.e. the set of axioms in the active workspace). The predicates provided are addax, ax, axn, control, delax, op and freeax_.

The addax predicate is used to add an axiom to the database. It has one or two arguments. The first argument must be a valid axiom. It may be:

(a) a unit axiom. In this case it is a skeleton or an atom.

(b) a non-unit axiom. In this case it is of the form <head><-<body>. <head> must be a skeleton or atom.

The axiom specified by the first argument is added to the database. If a single argument is specified then the axiom is added after all other axioms with the same predicate name and number of arguments. If the second argument is specified it must be an integer or a variable. We first explain the case of a call with two arguments where the second is an integer. This integer specifies where this axiom is to be added, as an index in the list of all axioms for the same predicate name and number of arguments. Consider the following list of axioms:

```
a(1).
a(2)<-b.
a(X)<-c(X).
a(4).
```

If the predicate call <-addax(a(m)). or <-addax(a(m),5). or <-addax(a(m),100). were issued then the new list would be:

```
a(1).
a(2)<-b.
a(X)<-c(X).
a(4).
```

a(m).

If the call <-addax(a(q),1). were then issued the list would become:

a(q).
a(1).
a(2)<-b.
a(X)<-c(X).
a(4).
a(m).

The index specified gives the index in the list where the axiom is to be added. If the index is 1 or less then the axiom is added before the first axiom in the list. Similarly if the index is greater than the index of the last axiom then the new axiom is added at the end of the list.

If addax is called with a second argument of a variable, the axiom specified by the first argument is added at the end of the list and its index is then unified with the second argument.

The delax predicate is used to delete an axiom from the database. It may be called with one or two arguments. The first argument is a term representing an axiom. The first argument may be:

(a) a unit axiom. In this case it is a skeleton or atom.

(b) a non-unit axiom. In this case it is of the form <head><-<body>. <head> must be a skeleton or atom.

Thus the first argument specifies the name and number of arguments for the axiom to be deleted. If only one argument is specified then an attempt is made to unify the argument with each of the relevant axioms in the database. The axioms are selected in the order in which they appear in the database. If no axiom is found which is unifiable with the first argument then the predicate fails. If the unification succeeds for an axiom then the axiom is deleted and the predicate succeeds. If backtracking subsequently returns to this point then the predicate will fail, thus preventing accidental deletion of further axioms.

If two arguments are specified then the second argument is considered to be the axiom index. It may be a variable or an integer. The attempts to unify the first argument with the database axioms proceeds as in the case of one argument. If the unification succeeds for a given axiom then an attempt is made to unify the

axiom index with the second argument. If the attempt fails then the search through the axioms is resumed. If the attempt succeeds then the axiom is deleted and the predicate succeeds. If backtracking subsequently returns to this point then the predicate will fail.

The ax and axn predicates are used to retrieve axioms from the database. The axn predicate retrieves axioms using the predicate name and number of arguments. The ax predicate retrieves axioms using a model axiom head.

The axn predicate has either of the two following formats:

axn(<name>,<nargs>,<axiom>)
axn(<name>,<nargs>,<axiom>,<index>)

The predicate call axn(c,2,A) will cause A to be unified with the first axiom for predicate c with 2 arguments. If there are no axioms for c with two arguments then ~~this call would~~ will fail. If the call succeeds and backtracking subsequently returns to this point then an attempt will be made to unify A with the next axiom for c with two arguments, and so on. The predicate call axn(c,2,A,I) functions identically except that when the call succeeds, I is unified with the index of the axiom unified with A. Similarly the call axn(c,2,A,3) will retrieve the third axiom for c with two arguments, if one exists. The predicate call axn(c,N,A) will unify N with 0 and unify A with the first axiom for c with 0 arguments. If this unification fails or backtracking returns to this point then the next axiom for c with 0 arguments is selected. When all axioms for c with 0 arguments are exhausted then N is unified with 1 and the axioms for c with 1 argument are retrieved in turn. This process can continue until all the axioms for c have been examined. The fourth index argument may be included and it functions analogously to the previous case. For example the goal statement:

<-axn(f,*,A)&write(A)&fail.

lists all axioms for predicate f.

The goal statement:

```
<-axn(f,N,*,1)&write(N)&fail.
```

writes out the different number of arguments for which f has an axiom.

The call axn(Name,N,A) can be used to examine the axioms for each predicate name in turn. First a predicate name is selected from the database and unified with the first argument. Then each of the axioms for this predicate are examined in turn as in the previous examples. After the last axiom for the given name is examined then the first argument will be unified with another name in the database and the search will continue. The order in which the predicate names are examined is not readily predictable since it depends on the hashing algorithm of this implementation. Consequently this order should be considered to be arbitrary. The following goal statement will cause all axioms in the database to be listed:

```
<-axn(*,*,A)&write(A)&fail.
```

The ax predicate functions in a manner very similar to the axn predicate. Again there are two basic formats:

```
ax(<head>,<axiom>).
ax(<head>,<axiom>,<index>).
```

<axiom> and <index> are treated exactly as for the axn predicate. <head> is a model axiom head and may be a skeleton, an atom or a variable. If <head> is not a variable then it specifies a predicate name and number of arguments implicitly. The axioms for this name and number of arguments are examined as for axn. If <head> is a variable then all axioms in the database are examined in turn as for axn(*,*,A). If an axiom unifies with the specified axiom then a model of the axiom head is unified with the first argument. By a model we mean a skeleton with anonymous variables for all arguments. The model idea is introduced so that a theorem prover written in PROLOG may use ax to retrieve the axioms relevant to a predicate term without actually unifying the axiom head and the predicate term.

The op predicate is used to manipulate operator declarations. Its use was introduced in 2.4 The Syntax

in Detail. Adding a unit axiom for the op predicate (with 3 arguments) is equivalent to adding an operator declaration. Similarly, deleting a unit op axiom deletes the operator declaration represented. Thus one can delete an operator declaration with a call of the form:

delax(op(<operator>,<type>,<priority>)).

where:

<operator> is an atom identifying the operator.
<type> is an atom specifying the declaration type and may be any one of lr,rl,prefix or suffix.
<priority> may be an integer or a variable.

If a matching declaration is found it is deleted.

A call to the op predicate may be used to retrieve an operator declaration. For example, the call op(.,rl,P) succeeds if "." is declared as rl. In this case P would be unified with the priority. The call op(.,T,P) succeeds if there is an operator declaration for "." The following goal statement will list all prefix operators:

<-op(Op,prefix,*)&write(Op)&fail.

In this case backtracking to the op predicate call causes each prefix declaration to be retrieved in turn. Note that the order in which the declarations are retrieved is pseudo-random and not the order in which the original declarations were added. However, if an operator is declared as both prefix and infix, the prefix declaration is always retrieved first. The following goal statement will list all operator declarations:

<-op(Op,T,P)&write(op(Op,T,P))&fail.

The control predicate is used to provide some special global variable facilities. The control predicate has two arguments, a key and a result. For example, the call <-control(top,X) retrieves the result corresponding to key top and unifies this result with X. The key and result pairs are manipulated in a fashion similar to operator declarations. To add a key-result pair, an axiom for control is added. Adding the axiom control(top,3) records result 3 for the key top. Only one pair can be recorded for any key value. If a pair exists with the same key as one being added, then the previous pair is replaced. The key must be an atom. The result associated with the key must be an atom or an integer. A key-result pair may be deleted by

deleting the appropriate axiom for the control predicate. For example <-delax(control(top,*)) will delete the key-result pair with key top. A subsequent call of the form <-control(top,*) would fail since no pair exists. The call <-delax(control(top,99)) would succeed only if the key-result pair of top-99 is currently recorded. The key-result pairs recorded in the data base may be queried in a manner similar to that used for operator declarations. For example:

<-control(K,R)&write(K.R)&fail.
   lists all key-result pairs in the data base.
<-control(K,99)&write(K)&fail.
   lists all keys with a result of 99.
<-control(i,R)&SUM(R,1,R2)&addax(control(i,R2)).
   increments the result integer corresponding to key i.

The control built-in predicate is also used with certain special keys to control system options. If the key verbose has an associated result of on then the system lists any goal statements which succeed. The goal statement <-<goal conjunction> is written in the form <goal conjunction><-, displaying any instantiations made for variables in the proof. The goal statement <-sum(2,2,*) causes sum(2,2,4)<- to be written on the terminal. If the key verbose does not have result on, then a successful goal statement is not listed.

If the key noax has an associated result of on then the system indicates each call to a predicate for which there are no axioms (and no compiled routines). For each such call a message of the form "noax - xxxxx nn" is displayed. xxxxx is replaced by the predicate name and nn is replaced by the number of arguments. With this feature, the goal <-sum(1,2.3)|prodq(3,4,12) causes the following messages to be displayed:

```
noax - sum 2
noax - prodq 3
?
```

This feature is initially enabled and may be disabled by deleting the control(noax,on) axiom or adding control(noax,off). To enhance the usability of this feature, the fail predicate (with no arguments) is included as a built-in predicate which always fails. Thus spurious messages of the form noax - fail 0 are avoided.

The key lower is used to control the translation of input from the main input stream. If lower is set to on then lower case letters from the terminal are input as lower case. If lower is not set to on then lower case letters from the terminal are translated to upper case as they are input. The initial setting of lower is determined based on the mode of operation when PROLOG is initiated. If single uppercase letters are assumed to be variables(the default), then lower is initially set to on. If variables can be designated only by using an asterisk, then lower is initially set to off.

The key goalinput can be used to control the input format of goals versus axioms. If goalinput is set to on then input terms are assumed to specify goals. The period character is declared as a prefix operator with the single initial axiom "(.(X))<-addax(X)". Thus with goalinput set to on, axioms may be added to the data base by entering them with a period prefix. With goalinput set off, all input terms are assumed to be axioms unless they have a unary '<-' as the main skeleton. goalinput is initially set to on. To make the input of axioms a bit more flexible when goalinput is on, the following axiom is provided in the initial data base:

(X<-Y)<-addax(X<-Y).

This odd looking axiom makes the initial period optional for non-unit axioms. period(or more formally "unless the input term is an instance of a skeleton for '.' with one argument"). Terms preceded by the period are assumed to be axioms.

The freeax_ predicate (yes it does end in an underscore!) is normally of use only in very specialized instances, usually when writing second level interpreters in PROLOG. When using a second level interpreter which 'never finishes', certain anomalies occur in the recovery of space from deleted axioms. When an axiom is deleted in a proof, the space for the axiom is placed on a 'deferred free list'. The space is not freed directly since the axiom may still be used in the proof. Space on this deferred free list is freed when the proof is completed. Thus in a second level interpreter which is continually adding and deleting axioms, a large deferred free list may be built up and the interpreter can run out of space. To provide for this situation, the freeax_ predicate is provided. Invoking the freeax_ goal causes all space on the

mention the random seed. (Insert 3)

deferred free list to be freed. It is the responsibility of the programmer to ensure that the current proof does not contain any references to freed axioms. Otherwise, disastrous results are likely!

3.6 Execution Control Predicates

The execution control predicates provide facilities for testing and controlling the progress of a proof. The ancestor, retry, /, &, |, repeat, fail, error, attn, stop and trace predicates are included and the meta variable facility is also provided.

The parent of a given literal in a proof is the literal which invoked the axiom containing the given literal. In the implication tree describing the proof, the parent literal labels the node above that labelled with the literal. The ancestors of a literal include its parent and its parent's ancestors. The ancestor predicate is used to examine the ancestors of the literal which invoked the predicate. When ancestor is used with one argument, the argument is unified with the most recent ancestor for which this is possible. If the argument cannot be unified with any ancestor, the predicate fails. If the predicate succeeds and subsequently backtracking returns to this point in the proof, the argument is unified with the next most recent ancestor and so on. The following axiom will list all of the ancestors of the ancestor literal and then fail.

listanc<-ancestor(A)&write(A)&fail. Note that the first ancestor listed will be listanc.

When the ancestor predicate is used with two arguments the first argument functions in the same way as the single argument above. The second argument is the ancestor index. For a given literal the ancestor index of its parent is 1, the ancestor index of its parent's parent is 2, etc. The first argument is unified with each ancestor in turn as above. If this unification is successful then the second argument is unified with the current ancestor index. The following axiom will list the five most recent ancestors of the ancestor literal:

listanc2<-ancestor(A,N)&write(A)&eq(N,5).

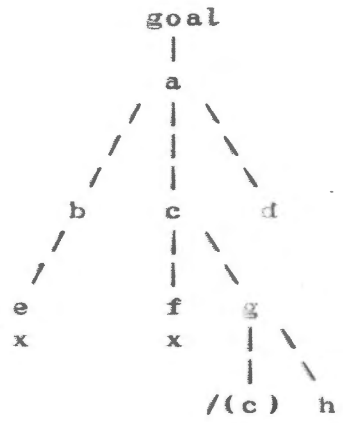
The retry predicate is provided to facilitate

recovery from an error situation. After a correction has been made, the proof may be restarted from some point before the error. retry has one or two arguments which control a search through the ancestors exactly as for ancestor. The difference is the action taken upon success. If an appropriate ancestor is found, the proof is backed up to the point where the subproof for the ancestor literal began and the proof is restarted from that point. retry restores the proof to the state it had at a particular point in the past. Consequently retry is only useful when some change has been made to the axioms.

The slash predicate with no arguments was described in 2.3 PROLOG Execution and Backtracking. The slash predicate is also provided in a more general form with either one or two arguments. The arguments control a search through the ancestors exactly as for ancestor and retry. If this search fails then the predicate fails. If the search succeeds then certain available choices are eliminated from an existing portion of the proof. All choice points are removed in the part of the proof from the point of selection of the given ancestor literal to the current point in the proof. Thus a call of the form /(*) has exactly the same effect as the simple nullary / call. Consider the following example:

```
a<-b&c&d.
b<-e.
c<-f&g.
e.
f.
g<-/(c)&h.
<-a.
...
```

The implication tree has the following form when the unary slash is called:



All choice points from the selection of c<-f&g onward are eliminated. Thus if h fails an alternate proof for e will be attempted (and the subproof of c will be deleted).

The meta variable facility allows a variable to be used in place of a literal in a goal or in the body of an axiom. When the variable is encountered in a proof it must be bound to a literal. The proof proceeds as if this literal occurred instead of the variable. For example, the following axiom defines a predicate exec which reads a term and "executes" it.

```
exec<-read(X)&X.
```

Axioms are included for the &(*,*) and the |(*,*) predicates. The axioms for | are:

```
|(X,Y)<-X.
|(X,Y)<-Y.
```

These axioms allow alternatives to be specified in an axiom body or goal with the desired effect. The axiom for & is:

```
&(X,Y)<-&(X,Y).
```

This axiom may look a bit ridiculous but it is useful, particularly when using the meta variable facility. For instance, if as input to the exec axiom above, a&b is specified, then this axiom for & would be invoked and a and then b would be called.

The fail predicate (with no arguments) is provided as a built-in predicate which always fails. This predicate is provided even though providing no axioms for fail would yield a predicate which always fails.

The reasons for providing such a predicate are:

(a) The fail predicate gives a standard name for a predicate which always fails. This imposes a programming standard which may improve program readability. This standard predicate could also make it easier for a compiler to perform certain optimizations.

(b) The provision of the built-in fail predicate makes the NOax feature of the control feature more useful. Refer to the description of the control predicate in 3.5 Database Predicates for further details.

The stop predicate is used to leave the PROLOG system. The execution of the stop predicate terminates the PROLOG session and returns to the operating system. All axioms and operator declarations in the current workspace are lost.

The repeat predicate can be used with zero to four arguments to perform looping in a proof. Repeat with no arguments succeeds initially and always succeeds on backtracking. Thus it can be used to loop indefinitely. The loop can be terminated only through use of the / or retry predicates. Repeat with one argument provides a similar looping facility but also maintains a loop counter. The argument is first unified with 1 and then to 2 on backtracking, etc. Again the loop can be terminated through / or retry. The second, third and fourth arguments of repeat can be used to specify an initial value, a stopping value and a step value, respectively. If any of these arguments are specified then they must be integers. The second parameter specifies the first value to be used for the counter. If this parameter is omitted, then the starting value is assumed to be 1, as described above. The third parameter is the stopping value. When the loop counter exceeds this value, the repeat predicate fails. The fourth parameter specifies the increment or step value. If it is omitted, then 1 is assumed. A negative step value may be specified, in which case the loop continues until the counter is less than the stopping value. Note that comparison to the stopping value is made on initial entry to repeat, so that the predicate may fail the first time if the stopping value is less than the initial value.

The error built-in predicate differs from the other predicates in the system in that it is not a

built-in predicate definition but a special interface which can be used to call a user-defined predicate. The error predicate (with no arguments) is called when certain non-disastrous errors occur in a proof. A message describing the error is always printed before invoking 'error'. The user may provide any axioms desired to list ancestors, allow axioms to be corrected, or to simply give up. A useful set of axioms for error are included in the standard set of axioms loaded with the PROLOG system. These axioms are defined in Appendix B : The PROLOG EXEC file

The attn built-in predicate is analogous to the error predicate. The attn predicate is called when the attention or break key is pressed on the terminal. User axioms may be added for attn to provide whatever exception handling is desired. The standard set of axioms in the PROLOG EXEC file invoke the axioms for error when attn is called(i.e the axiom for attn is 'attn <- error').

The trace built-in predicate provides special features for debugging PROLOG programs. It allows execution tracing to be enabled or disabled on a predicate by predicate basis. The trace predicate functions in a manner similar to the op predicate, in that axioms for trace can be added, tested or deleted and the presence of trace axioms have side effects. If the data base contains an axiom of the form 'trace(P)' where P is the name of a predicate, then tracing is enabled for all attempts to prove goals with predicate name P.

The actual tracing functions to be performed can be defined by user axioms. The standard PROLOG EXEC file includes axioms which write out the 'position' and the 'goal' for a traced predicate. Four 'positions' are defined, namely 'call' 'exit' 'redo' and 'fail'. The 'call' position occurs when the goal is initially attempted, before any unification has taken place. The 'exit' position occurs after the goal has been successfully proven. At the exit point, the goal has been unified with an axiom head and the axiom body has been executed. The 'redo' position occurs when backtracking returns to the goal, before the attempt has been made to reprove the goal. The 'fail' position occurs after a final unsuccessful attempt has been made to prove the goal.

Enabling tracing for the goal 'Goal' causes

execution to proceed as though t(Goal) became the goal, where t has the following axioms:

```
t(Goal) <- systrace_(call,Goal) &
           Goal &
           ( systrace_(exit,Goal) |
                systrace_(redo,Goal) ).
t(Goal) <- systrace_(fail,Goal).
```

These axioms will cause the systrace_ axiom to be invoked at each of the four positions in proving the goal. Note that for this to work correctly, the systrace_ goal must succeed for positions call and exit and must fail for positions redo and fail. The normal axioms for systrace_ (which are included in the standard PROLOG EXEC file) are as follows:

```
systrace_(Position,Goal) <-
                    systrace(Position,Goal) &
                    fail.
systrace_(call,Goal).
systrace_(exit,Goal).
```

The user may then define the systrace predicate to write anything desired or even prompt for user direction. The PROLOG EXEC file contains the single axiom:

```
systrace(Position,Goal) <- writech(Position) &
                           tab(1) &
                           write(Goal).
```

This axiom may be deleted or preceded by another user axiom to modify the output format. For example, if the following axiom is added prior to the axiom above, then the 'call' position will be traced by a user trace routine:

```
systrace(call,Goal) <- / & trace_call(Goal).
```

In general, the user may find it useful to provide different axioms for systrace, but those for systrace_ should be left unchanged.

3.7 Miscellaneous Predicates

The miscellaneous group includes predicates to test the collating sequence of constants, to test if a symbol is a letter or a digit, and to convert a character to or from uppercase. In addition a system predicate is provided to execute operating system commands. A collating sequence is defined for the values of constants as follows:

(a) Any atom is less than any integer.
(b) Integers are related by the conventional ordering for integers.
(c) Atoms are ordered by the lexical ordering imposed when the ordering of the symbols is as defined by the standard EBCDIC orderings.

Six built-in predicates are provided to test the relation between two constants. Each predicate has two arguments, both of which must be constants. The relations which cause each predicate to succeed are listed below.

lt - argument 1 is less than argument 2
le - argument 1 is less than or equal to argument 2
gt - argument 1 is greater than argument 2
ge - argument 1 is greater than or equal to argument 2
eq - argument 1 is equal to argument 2
ne - argument 1 is not equal to argument 2

Examples: The following predicate calls succeed.

```
<-lt(a,37).
<-gt(3,'-2').
<-ge(a3,a).
<-ne(abc,c).
<-eq('abc',abc).
<-eq(12,'+0012').
```

The predicates letter and digit each have one argument. The argument must be a constant. The predicates test if the value of the constant is a single symbol belonging to the given class. If the argument of letter is a constant consisting of a single letter then the call succeeds. If the argument of digit is an integer from 0 to 9 inclusive then the call succeeds. The upshift predicate has two arguments, of which at least one must be a constant. If the first

argument is a single lowercase letter, then the second argument is unified with the uppercase constant for the same letter. If the first argument is a skeleton or a constant other than a lowercase letter, then the predicate fails. In the remaining case(where the first argument is a free variable), the predicate will succeed only if the second argument is an uppercase letter. In this case the first argument will be unified with the lowercase constant for the same letter.
Examples: The following predicate calls succeed

```
<-letter(z).
<-digit(0).
<-digit('+0001').
<-upshift(*,'A').
<-upshift('a',*).
<-upshift('z','Z').
```

The system predicate allows CMS commands to be executed from the PROLOG environment. It may be invoked with one or two arguments. The first argument specifies the command to be executed. The second argument is optional. If present, it is unified with the integer return code from the CMS command. If the second argument is not present then the return code is ignored. The command to be executed is specified as a list of one or more constants. Each constant corresponds to one token in the CMS command. Tokens may be no more than eight characters long. In order to invoke CP commands, simply use an initial token of cp. Note that all lower case letters in tokens are automatically shifted to uppercase. The left parenthesis preceding the options is a separate token. The following are valid calls to system:

```
<-system(print.prolog.exec.nil).
<-system(l.'*'.prolog.nil,Returncode).
<-system(cp.q.users.nil).
<-system(t.prolog.maclib.'('.member.xxxxxx.nil).
```

Appendix A : Running PROLOG under VM/CMS

The Waterloo PROLOG system is invoked from the VM/CMS environment by typing the command PROLOG. This command invokes the PROLOG EXEC file and subsequently the PROLOG module. The EXEC file supports several options as well as defining numerous utility predicates. The EXEC file is described in detail in Appendix B.

A typical PROLOG session involves executing goals and adding and deleting axioms. The format of entry for axioms versus goals can be controlled. The standard system starts in what is called "goalinput" mode. In this mode, any term that is input is assumed to be a goal, unless it is in the format ".(...)", in which case the term is assumed to be an axiom to add to the database. In fact "." is treated as a predicate with the single axiom "(.(Goal)) <- addax(Goal).". Consequently in goalinput mode all input terms are treated as goals. If goalinput mode is turned off then goals must be entered in the format "<-(Goal)". This mode of operation is more verbose if numerous goals are being entered, so the goalinput mode is normally preferred. The description of the control axiom in 3.5 Database Predicates outlines how to turn the goalinput mode on and off. Note that in this manual, goals are always described in the "<-(Goal)" syntax for clarity of explanation. When axioms are stored in a file, they are normally stored without the "." prefix. The consult predicate (described in detail in Appendix B) can be used to read the axioms from a file and add them to the data base. In addition, the consult predicate will treat any terms in the file in the format "<-(Goal)" as goals and execute them. Note that when entering axioms or goals in a file or from the terminal, they must always be terminated with a dot which is either the last character in the record or is followed by a blank.

The ted predicate can be used to update files of axioms when errors are detected. The axioms may then be reloaded using the reconsult predicate.

To exit the PROLOG environment completely, use the stop predicate. Simply type 'stop.'. The attention or break key on the terminal may be used to interrupt a PROLOG program. If attention or break is signalled

during a proof or in response to a READ from a PROLOG axiom, then three exclamations points are written on the terminal and the attn axiom is called. Axioms for attn may be defined by the user. A standard set of axioms for attn is described in Appendix B : The PROLOG EXEC file.

The PROLOG system also includes facilities for tracing program execution, in order to aid in debugging. This facility is enabled for predicate P by adding an axiom 'trace(P)'. Note that trace has a single argument which must be an atom. When trace is enabled for a predicate, in conjunction with the standard EXEC file, the progress of proving a goal for the predicate will display the goal at each of four positions. These positions, as well as the functioning of trace are described in detail in 3.6 Execution Control Predicates in conjunction with the trace predicate.

The PROLOG input/output predicates provide facilities for reading and writing CMS files. All files which are accessed are assumed to have a filetype of PROLOG and to have fixed records with a length of 80 characters. When reading from files, a blank mode letter is used so that the normal CMS search order is invoked. When updating files a mode letter of 'A' is used. Consequently, PROLOG programs can update files on the A-disk only.

Appendix B : The PROLOG EXEC file

The Waterloo PROLOG system is invoked from the VM/CMS environment by typing the command PROLOG. This command causes the PROLOG EXEC file to be executed and the PROLOG MODULE to be invoked from the EXEC file. The EXEC file allows several options when invoking PROLOG. The first option controls the size of the data area acquired by PROLOG for execution. This area will be used to contain all axioms as well as the execution stack. The size of the area is specified as the number of 1024 byte blocks of memory to be used. If no size is specified, then 100 is assumed. The format for specifying the size is:

PROLOG nnn

where nnn is the desired size.

In addition, a list of file names separated by blanks may be specified as an operand. Axioms and goals will be read from each of these files in turn, using the consult verb defined below. For example the command PROLOG DATA1 DATA2 will invoke PROLOG and use consult to load axioms from file DATA1 and then DATA2. The workspace size operand may also be used in conjunction with the files list by specifying the size parameter first,as in "PROLOG 2000 DATA1 DATA2".

The PROLOG EXEC file also contains a set of axioms that are added to the initial PROLOG database. These axioms provide various utility functions, including the consult function described above. Each predicate defined in the EXEC file is described below.

The consult predicate has a single operand which is the name of a file of goals and axioms. Each term in the file is read. A term of the form "<-(...)" is executed as a goal. All other terms are added to the database as axioms. When the end of the file is reached, the file is closed.

The reconsult predicate functions similarly to the consult predicate except that it deletes any existing axioms for the predicates which are read in, before adding the axioms in the file. Axioms for op and control are treated differently, in that the existing axioms for these predicates are not deleted.

The list predicate is used to list axiom definitions from the database. If it is used with no arguments or with a single variable as an argument then all axioms in the database are listed. The list predicate may also be used with a predicate name as an argument to list the axioms for that predicate. For example, "list(compute)" will list all axioms for predicate compute, with any number of arguments.

The delaxall predicate is used to delete all axioms for a given predicate. delaxall is invoked with the predicate name as an argument. For example "delaxall(compute)" will delete all axioms for predicate compute.

The error predicate is "built-in" in the sense that it is executed whenever certain errors occur. The PROLOG system provides this error recovery interface to allow the user to investigate the state of the proof and take appropriate recovery action. The set of predicates included for error handling in the exec file provide the following functions:

- when the error predicate is invoked, the ancestors of the error predicate are listed, in ascending order. The number of ancestors listed is controlled by adding the axiom "control(errordepth,X)" where X is a positive integer. The EXEC file sets the initial errordepth to 5.

- after the ancestors are listed, the user is prompted to enter a command. The command entered may be any valid goal. The goal is executed and the success or failure of the goal is indicated by the '?' or '<-' responses. The user is then prompted for another command. The most common commands used at this point are "quit" to terminate the proof, "addax" or "delax" to correct the database and "retry(X)" to retry goal X after a correction has been made.

The error axioms also check for the condition of an error within an error and do not print the ancestors in this case.

The quit predicate is used primarily in error recovery. It terminates the proof and returns to PROLOG command level.

The attn predicate is a built-in "hook" analogous to the "error" predicate. The attn predicate is executed when the attention(or break) key is pressed during a PROLOG proof. The axiom for attn defined in the exec file simply invokes the error predicate to provide exactly the same facilities as error for recovery.

The ted predicate can be used to invoke the CMS transient editor from the PROLOG environment. For example "ted(blocks)" will edit the file blocks and then return to PROLOG.

The ¬ predicate definition is included to handle negation. The goal ¬(Pred) will succeed if and only if the goal "Pred" fails. Note that when ¬ succeeds it doesn't bind any variables.

The systrace predicate is invoked as part of the tracing facility. It is described in more detail in 3.6 Execution Control Predicates in conjunction with the trace predicate.

The axioms used to define these predicates in the exec file are listed below. All internal predicates end in an underscore, to avoid conflicts with user axioms.

```
consult(File) <- read(A,File) & consult_(A)  & fail.
consult(File) <- fileclose(File).
consult_(<-Goal) <- / & Goal & /.
consult_(Axiom) <- addax(Axiom).
reconsult(*) <- delaxall(reconsulted_) &
                        addax(reconsulted_(1)) & fail.
reconsult(File) <- read(A,File) &
                        reconsult_(A)  & fail.
reconsult(File) <- fileclose(File) &
                        delaxall(reconsulted_).
reconsult_(<-Goal) <- / & Goal.
reconsult_(op(X,Y,Z)) <- / & addax(op(X,Y,Z)).
reconsult_(control(X,Y)) <- / & addax(control(X,Y)).
reconsult_(Axiom) <- reconsult_name_(Axiom,Name) &
                     reconsult_start_(Name) &
                     addax(Axiom).
reconsult_name_(Head<-Body,Name) <- / &
                                cons(Name.*,Head).
reconsult_name_(Head,Name) <- cons(Name.*,Head).
reconsult_start_(Name) <- reconsulted_(Name) & /.
reconsult_start_(Name) <- delaxall(Name) &
```

```
                              addax(reconsulted_(Name),1).
list<-list(*).
list(control) <- control(Id,Value)&
      writeq(control(Id,Value))&
      fail.
list(op) <- op(Operator,Type,Priority)&
      writeq(op(Operator,Type,Priority))&
      fail.
list(Name) <- axn(Name,*,Axiom) &
       writeq(Axiom) &
       fail.
list(*).
control(errordepth,5).
error<-ancestor(error,N)&gt(N,1)&/&error_cmd_.
error <- control(errordepth,Depth) &
                          error_list_(Depth)&fail.
error <- error_cmd_.
error_cmd_ <- repeat & writech('ENTER COMMAND:') &
                        newline & error_exec_.
error_exec_ <- read(C) &
           ( (C & error_succeed_(C)) |
              (writech(?) & newline) ) & / & fail.
error_succeed_(C)<-control(verbose,on)&writech(C)&fail.
error_succeed_(*)<-writech('<-')&newline.
error_list_(Depth) <- sum(Depth,2,Depth2) &
                      ancestor(A,Index) &
                      gt(Index,2) & writeq(A) &
                      eq(Index,Depth2) & /.
attn <- error.
quit<-/(goal)&fail.
ted(File)<-system(ted.File.prolog.nil).
¬Pred <- Pred & / & fail.
¬Pred.
systrace_(Type,Goal) <- systrace(Type,Goal)&fail.
systrace_(call,*).
systrace_(exit,*).
systrace(Type,Goal) <- writech(Type) & tab(1) & write(Goal).
delaxall(Name) <- atom(Name) & axn(Name,*,Axiom) &
                      delax(Axiom) & fail.
delaxall(Name).
```

Appendix C : Using PROLOG with uppercase input

The syntax of PROLOG as described in this manual involves both upper and lower case letters. If PROLOG is being used with terminals which do not support lower case, a slightly modified "uppercase only" syntax may be invoked. In this mode of operation, the following changes are made to the standard syntax:

- all variables must begin with an asterisk.
- all symbols which begin with a letter are assumed to be identifiers(i.e either predicate names or constants).
- all input characters are automatically shifted to upper case.

To invoke PROLOG in the uppercase only mode, the PROLOG module must be invoked with the parameter 'L'.(e.g. PROLOG L). If a workspace size is to be specified, the parameter should be 'Lnnn' where nnn is the size(e.g. PROLOG L256). The standard PROLOG EXEC file does not support this mode of operation. To create an appropriate EXEC file, the standard EXEC can be copied, all the axioms converted to uppercase syntax and the lines invoking PROLOG changed to add the 'L' prefix to the first parameter.