# An Implementation of PROLOG

by

Grant Maxwell Roberts

A thesis
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1977

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below and give address and date.

## 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 PR●LOG. 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(8,4). 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(3), natural language understanding systems( ) and a program for automatic plan generation(9).

The original PR●LOG language and an interpreter for it were developed at the University of Marseille by

Colmeraur and his colleagues. The author has developed an implementation in an attempt to provide a more usable version of PROLOG. The principal goal of this implementation was to reduce the execution time of PROLOG programs. Another important aim was to make the PROLOG system more convenient to use by altering the syntax and providing improved system functions, particularly for error recovery.

This document is intended to serve a dual purpose. It provides a user's manual for the system. It also describes the design and implementation of the language. The user' manual is contained in 2 The Language and 3 Built-in Predicates. Sections 2.1 to 2.3 inclusive describe the basic features of the language. Section 2.4 is a detailed reference for the language syntax. A complete description of the built-in predicates is provided in 3 Built-in Predicates.

The data structures and algorithms of the system are described in 4 The Implementation. The various decisions and tradeoffs made in the design of the language and in its implementation are discussed in 5 Design Decisions. Possible future modifications are discussed in 6 Future Considerations for PROLOG.

## 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! The possibility of different search strategies for PROLOG is discussed in 6 Future Considerations for PROLOG.

An explanation of resolution logic in terms of classical logic is given in ( ). Programming using resolution logic is discussed in (4).

## 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 - basically any sequence of letters and digits. A constant may be an integer or an atom. e.g. ABC and X2.

(b) a variable - an asterisk followed by a sequence of letters and digits. e.g. *X 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)).

The syntax can be described in BNF notation:

```
<term>::= <atom>      |
          <integer>   |
          <variable>  |
          <skeleton>  |
          ( <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>::= * |

<variable> <letter> |

<variable> <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 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.

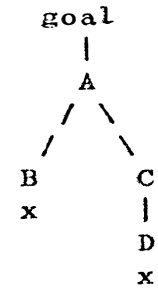
B.

C<-D.

D.

<-A.

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




This is a completed implication tree 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.

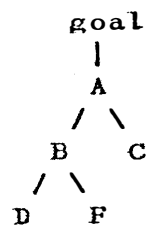
G.

H.

The initial state of the proof is represented as:


The first axiom for A is selected, namely A<-B&C giving:

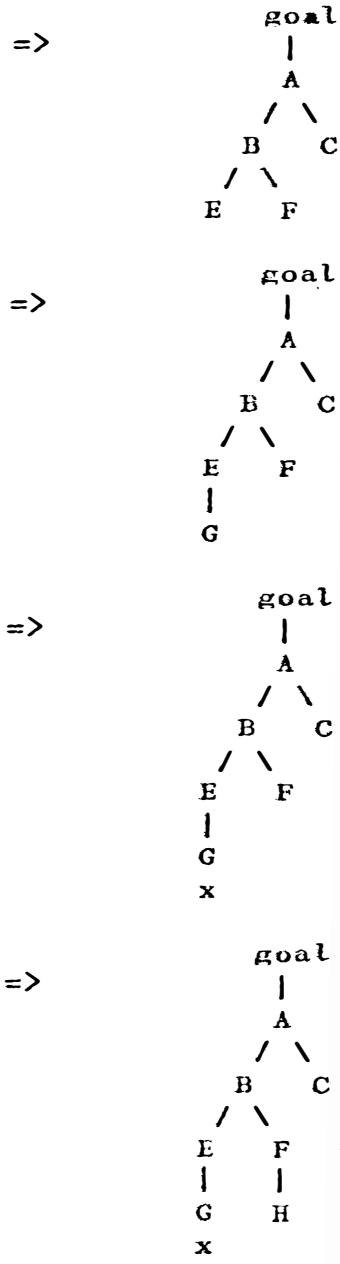
The prover always works in a depth-first left-to-right fashion. Consequently the next literal to be proven is B. The axiom B<-D&F is selected:

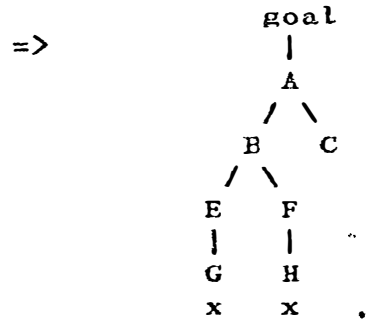


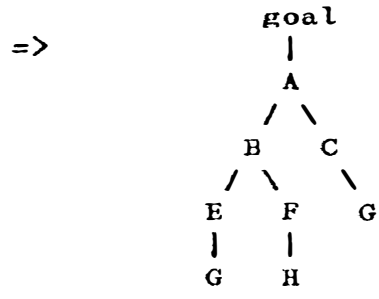

The prover then attempts to prove D. But there are no axioms for D so the prover must backtrack. This involves backing up the proof and trying other alternatives. A choice point in the proof is a point where an axiom was chosen to prove a literal and more axioms remain to be tried. Backtracking involves 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.

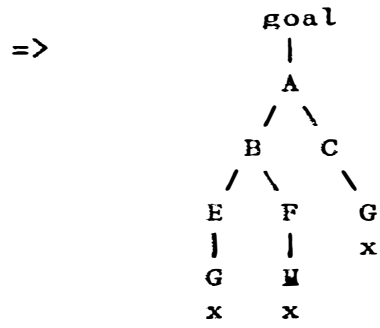
The most recent choice point in the current proof is the point where the axiom B<-D&F was selected. The proof is backed up to this point and the other axiom,

B<-E&F, is selected. The proof continues as shown below:

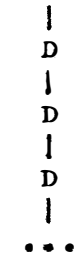





The final proof is represented by a completed implication tree. Of course, if the proof fails then the implication tree is never completed. If, in this example, we omit the axiom C<-G then the proof attempt will fail. Alternatively, if we include another axiom D<-D then the prover will attempt to construct 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)
```

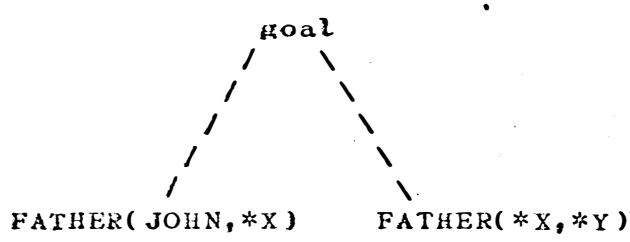
Now we need to find an instance of an axiom which we can use in the proof of SON(*Z,JOHN). The appropriate instance is formed from SON(*X,*Y)<-FATHER(*Y,*X) by substituting *Z for *X and JOHN for *Y to give SON(*Z,JOHN)<-FATHER(JOHN,*Z). The tree now is:

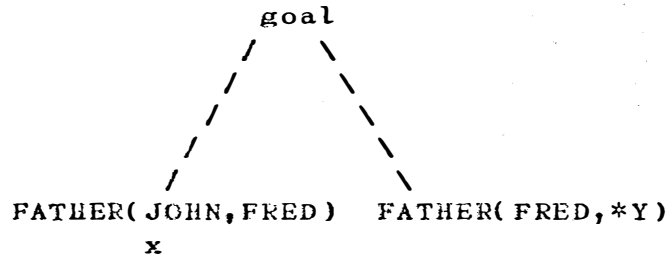
```
    goal
     |
SON(*Z,JOHN)
     |
FATHER(JOHN,*Z)
```

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 <u>unification.</u> Next we want to find an axiom whose head will <u>unify</u> with FATHER(JOHN,*Z). The first axiom for FATHER matches if we substitute FRED for *Z. This gives the completed implication tree:

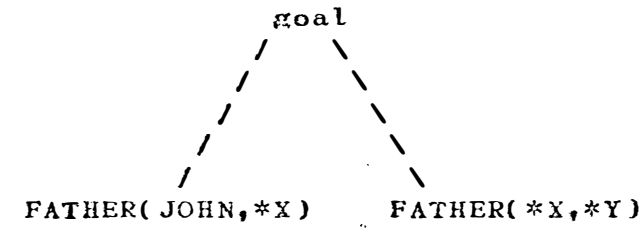
```
    goal
     |
SON(FRED,JOHN)
     |
FATHER(JOHN,FRED)
    x
```

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

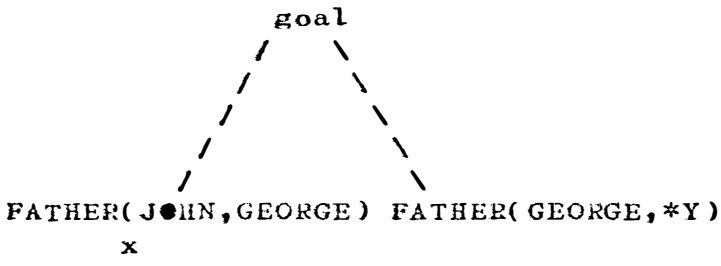


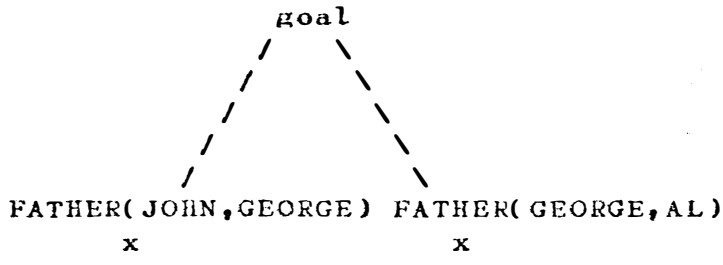

The attempt to solve the subgoal FATHER(FRED,*Y) fails since this term will not unify with any of the axiom heads. Backtracking occurs and the proof is backed up to the point where the FATHER(JOHN,FRED) axiom was activated. This axiom is then deactivated and any substitutions made when (or since) this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



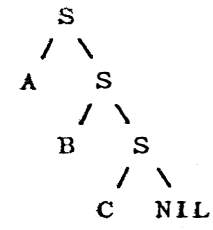

The axiom FATHER(JOHN,GEORGE) is about to be selected for unification with FATHER(JOHN,*X). This unification succeeds giving:




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:

```
        goal
         |
ELEMENT(C,S(A,S(B,S(C,S(D,NIL)))))
         |
ELEMENT(C,S(B,S(C,S(D,NIL))))
         |
ELEMENT(C,S(C,S(D,NIL)))
         x
```

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).
```

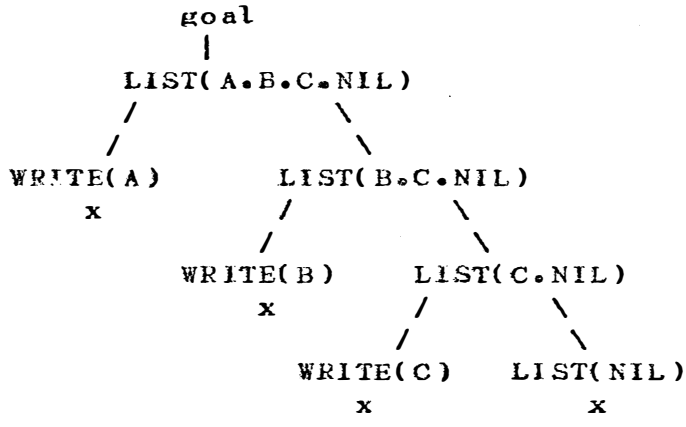
Suppose we want an axiom to write all the elements of a set. The following axioms will suffice:

```
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:

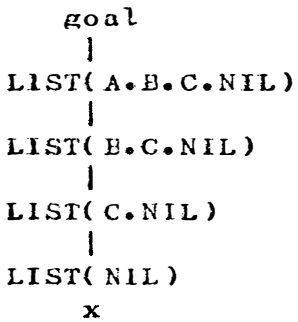
B.

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:




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".

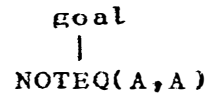
The axioms for NOTEQ remain to be defined. The axioms

are:

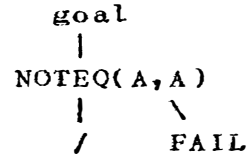
```
NOTEQ(*X,*X)<- / & FAIL.

NOTEQ(*X,*Y).
```

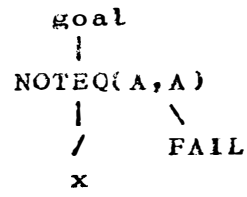
These axioms make use of a special control feature, the slash(/). To illustrate this feature we trace the attempt to prove the goal <-NOTEQ(A,A). Initially, we have:




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 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) Special Symbols - This group consists of all symbols not in any of the three 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 an asterisk(*), followed by the variable name. The variable name is a sequence of letters and digits. Thus *X, *A1B2C3, and *37 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 or digits.

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 two symbols which belong to the list of declared special character pairs. This list is dynamic

in nature and is discussed in 3.6 Database Predicates in conjunction with the OP built-in predicate. The initial list of special pairs consists of a single entry for "<-".

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 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).

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:

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 3.6 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 comment 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 3.7 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.

# 3 Built-in Predicates

## 3.1 Introduction

The implementation provides several built-in predicates. These predicates provide facilities which it is 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. The single exception to this is the OP built-in predicate described in 3.7 Database Predicates.

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

Structural Predicates - ATOM, CONS, INT, SKEL, STRING, VAR.

Input/Output Predicates - NEWLINE, READ, READCH, WRITE, WRITECH

Arithmetic Predicates - DIFF, PROD, QUOT, REM, SUM

Workspace Predicates - CLEAR, COPY, LOAD, PCOPY, SAVE, WSID

Database Predicates - ADDAX, AX, AXN, CONTROL, DELAX, OP

Execution Control Predicates - ANCESTOR, RETRY, /, &, |, FAIL, ERROR, STOP, meta variable

Miscellaneous Predicates - DIGIT, LETTER, 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, and STRING.

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 */
```

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. It is used to decompose a skeleton into a list consisting of the skeleton name following 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 an 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 CONSTANT(*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, such that 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).
```

## 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 identifier. A file identifier may consist of from 1 to 8 characters, the first of which must be a letter and the remainder must be letters or digits. 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).

READ is a predicate with one or two arguments. The second argument is the optional file identifier. 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 or two arguments. The second argument is the optional file identifier. 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.

READCH is a predicate with one or two arguments. The second argument is the optional file identifier. 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.

WRITECH is a predicate with one or two arguments. The second argument is the optional file identifier. 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 symetrical. 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. NEWLINE 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').
```

## 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

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).
```

### 3.5 Workspace Predicates

A set of PROLOG axioms is referred to as a workspace. When the PROLOG system is running, the current set of axioms is referred to as the active workspace. In addition a library of workspaces is maintained. A system of built-in predicates is provided for manipulating these workspaces.

A workspace in the library is identified by a workspace identifier. A workspace identifier is a sequence of from 1 to 8 letters or digits. The first character of the identifier must be a letter. The

active workspace also has a workspace identifier associated with it (refer to the WSID predicate below). If the identifier of the active workspace is CLEAR then a SAVE predicate may not be executed without resetting the workspace identifier.

CLEAR is a predicate with no arguments. It has the effect of clearing the active workspace of all axioms and setting it to the initial state.

LOAD is a predicate with one argument. The argument must be an atom whose value is a valid workspace identifier. The active workspace is loaded from the library workspace with the specified identifier. Any axioms or terms in the original active workspace are lost. The workspace identifier in the new active workspace is set to that of the workspace which was loaded.

SAVE is a predicate with no arguments. It causes the active workspace to be saved in the library member specified by the workspace identifier in the active workspace. The active workspace is left unchanged.

WSID may have zero, one or two arguments. When used with no arguments it causes the workspace identifier associated with the active workspace to be

displayed. When used with one argument, the argument must specify a valid workspace identifier. The workspace identifier in the active workspace is reset to the specified value. When WSID is used with two arguments an attempt is made to unify the current workspace identifier with the second argument. If this unification succeeds, the current workspace identifier is reset to the value specified by the first argument. The first argument must be a valid workspace identifier or an error occurs.

COPY is a predicate with one or two arguments. The first argument always specifies a workspace identifer. Data is copied into the active workspace from the library workspace with the given identifier. If a single argument is specified then all axioms and operator declarations are copied from the library workspace. If a second argument is specified then it must be an identifier (i.e. an atom). All axioms and all operator declarations for the given identifier are to be copied. When an attempt is made to copy the axioms for a given predicate name and number of arguments a check is made to see if any axioms exist for that name and number of arguments in the active

workspace. If any such axioms exist they are deleted before the new axioms are copied in. Similarly if an operator declaration for an identifier is found in the library workspace then all declarations for the operator in the active workspace are deleted and the new declaration is added.

PCOPY is a predicate with one or two arguments. It is similar to COPY but it performs a protected copy. The difference is that PCOPY never deletes axioms or operator declarations from the active workspace. When the situation arises which causes COPY to perform a deletion, PCOPY leaves the active workspace data intact and does not copy the axioms or operator declarations in question.

## 3.6 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 and OP.

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 unifcation 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 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

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 the 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. LOWER is initially set to OFF.

## 3.7 Execution Control Predicates

The execution control predicates provide facilities for testing and controlling the progress of a proof. The ANCESTOR, RETRY, /, &, |, FAIL, ERROR, and STOP 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.

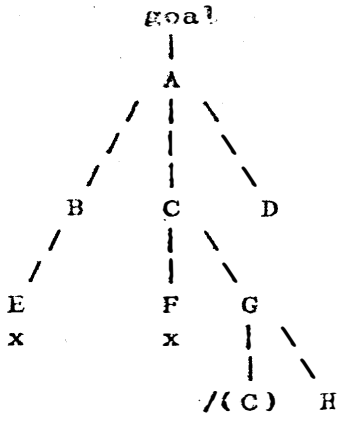
F.

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 3.6 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.

## 3.8 Miscellaneous Predicates

The miscellaneous group includes predicates to test the collating sequence of constants and to test if a symbol is a letter or a digit. 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 test the relation between two constants. Each predicate h a s t o arguments, both of which must constants. The relations which cause each predicate to succeed listed below.

LT - argument 1 is less th n argument

LE - argument 1 is less th n or equ l to argu ent 2

GT argument 1 is great r than argu ent 2

GE - argument 1 is greater than or e u l to argument 2

EQ argument 1 is equal to rgu nt 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,'+012').
```

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 capital letter then the call succeeds. If the argument of DIGIT is an integer from 0 to 9 inclusive then the call succeeds.

Examples: The following predicate calls succeed

```
<-LETTER(Z).

<-DIGIT(0).

<-DIGIT('+0001').
```

# 4 The Implementation

## 4.1 Introduction

The essential features of the implementation are:

(a) Data structures for

- -terms
- -substitutions
- -environments
- -axioms
- -symbols

(b) Algorithms for

- -unification
- -interpreting axioms
- -backtracking
- -reading terms
- -writing terms

The system is implemented in OS/370 assembler language and relies heavily on the use of macros to simplify the implementation procedure. A general set of program structure and linkage macros is used(7). In addition several macros were written for this

particular application. Included in this group are macros for building terms and axioms as well as describing table entries.

The /370 word consists of 32 bits organized as 4 bytes of 8 bits each. An address is specified by the rightmost (low order) three bytes in a word. This addressing organization influences the details of many of the data structures in the implementation.

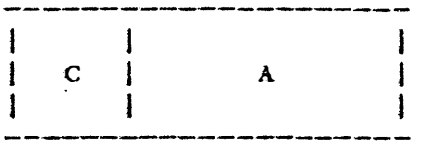
The data structures used rely heavily on those developed in the original PROLOG interpreter. In particular the elegant and efficient structure sharing technique used therein to represent terms and substitutions is used here with minor changes only.

A symbol table is maintained for all identifiers used in the active workspace. Each identifier is assigned a unique string descriptor and is characterized by the address of this descriptor. Thus two identifiers are equal if and only if the corresponding descriptor addresses are equal. The symbol table is discussed further in 4.7 Symbol Table Organization.

## 4.2 Representation of Input Terms and Axioms

An input term is a term which has no substitutions associated with it. An axiom is an input term until it is invoked in a proof, at which point substitutions may occur.

An input term is represented by a term word of the form:




where A is an address or number and C is a one byte type code identifying the kind of term represented. C is called the term identifier or term ID. A is called the term value. The types represented by C are :

(a) an integer: The term word represents an integer. The term value is a twenty-four bit signed twos complement representation of the integer. Thus integers from -8,388,608 to 8,388,607 inclusive can be represented.

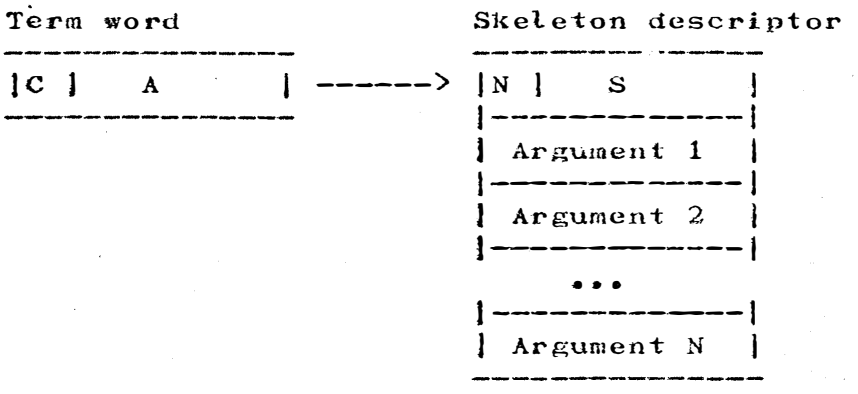
(b) a variable: The term word represents a

variable. Each variable in an input term is represented by a canonical number from 1 to n, where is the number of distinct variables in the term. Thus if the axiom A(*X)<-B(*Y,F(*X)). is read, then *X is associated with 1 and *Y is associated with 2. A variable in an input term is represented by a term word with an ID indicating a variable and a value which is a twenty-four bit displacement. The displacement for a variable is derived directly from the canonical number of the variable in the original term. The displacement, which is equivalent to the canonical number, is used to reduce the calculations required to retrieve the value of an instantiated variable. The use of the displacement will be discussed further in 4.3 Substitution and Constructed Terms.

(c) an atom. The term word represents an atom. The term value specifies the address of a symbol table entry for the atom. The symbol table entry for an identifier is called a

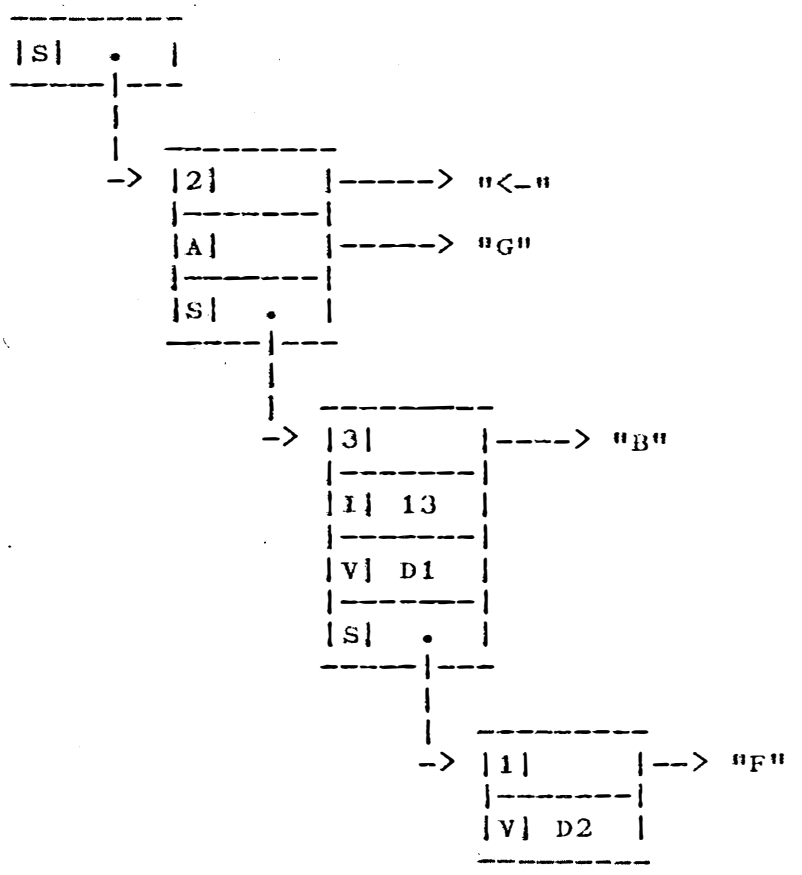
string descriptor.

(d) a skeleton. The term word represents a skeleton by specifying the address of a skeleton descriptor as follows:




A skeleton descriptor occupies two or more words. The first word is called the skeleton key. The skeleton key consists of a one byte count(N) and the address(S) of the string descriptor for the skeleton name. N specifies the number of arguments that the skeleton has. A skeleton can have from 1 to 255 arguments(inclusive). Following the skeleton key are N term words representing the arguments of the skeleton.

The following diagram shows the representation of the axiom G<-B(13,*Y,F(*X)). The term IDs for variables, atoms, integers and skeletons are repesented as V, A, I and S respectively. The displacements for variables with canonical numbers 1 and 2 are represented by D1 and D2 respectively. A pointer to the string descriptor for xxx is represented by --> "xxx".



## 4.3 Substitution and Constructed Terms

When an axiom is activated in a proof, an environment is created for the axiom. The environment contains information for backtracking. It also contains a description of all substitions made in the axiom. This substitution information is recorded in a list called the substitution list or the instantiation list.

A term which has substitutions associated with it is called a constructed term. When an axiom is activated, the axiom (and all of its subterms) become constructed terms. A constructed term is represented as an input term with an associated substitution environment. A constructed term is an instance of the corresponding input term. The value of the constructed term may be determined from the value of the corresponding input term by using the environment to retrieve the substitutions made for the variables in the input term. The internal representation of a constructed term requires two words. The first word is the term word for the input term. The second word specifies the address of the appropriate environment.

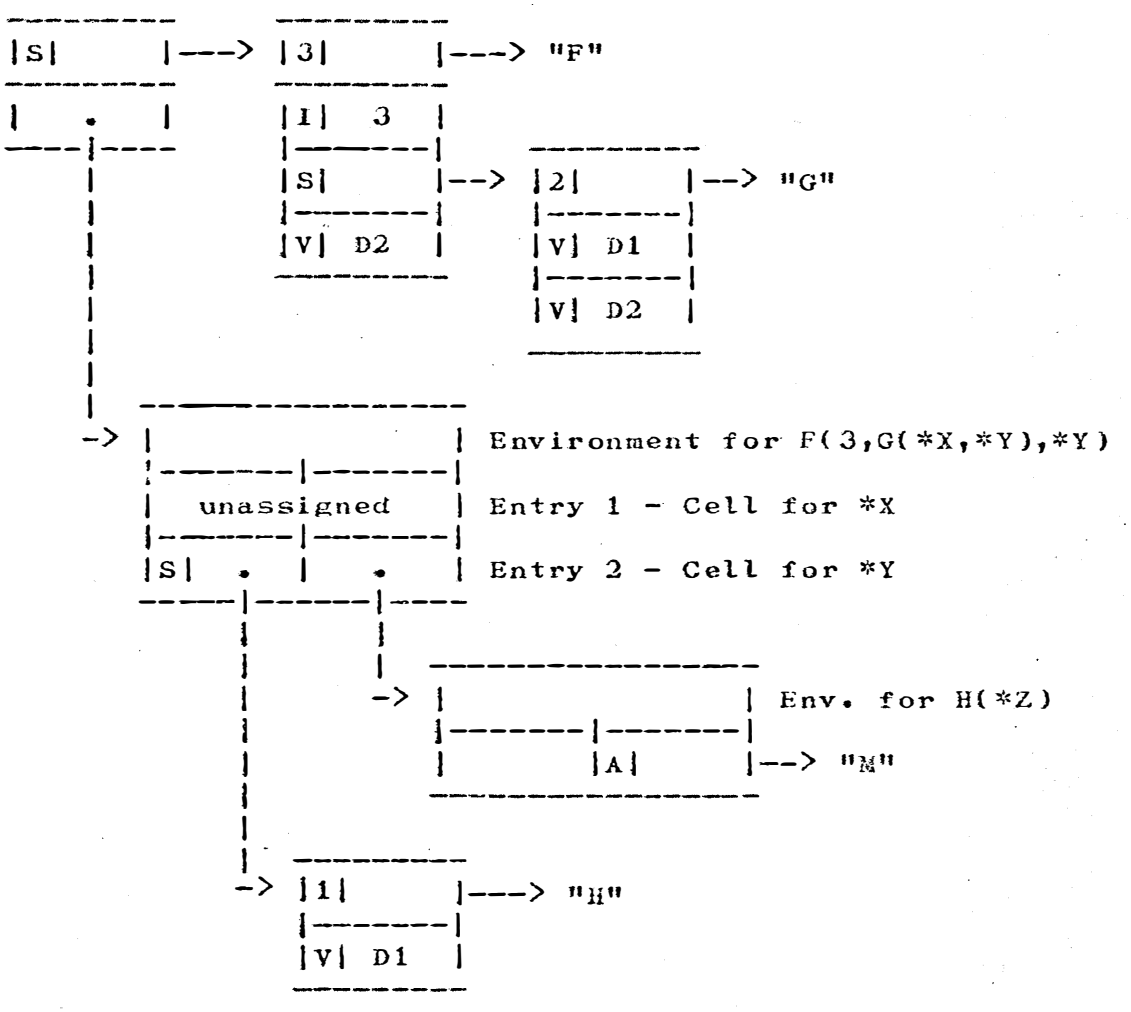
The instantiation list in the environment has an

entry for each variable in the original axiom. The entry for a specific variable is either marked to indicate that the variable is unassigned or else the entry contains a constructed term giving the value of the term substituted for the variable. Thus a variable is instantiated by placing the appropriate constructed term in the instantiation list entry. This instantiation may be undone during backtracking by resetting the entry to "unassigned". Another important feature of this representation for constructed terms is in the area of storage requirements. When an axiom is activated an environment of fixed size is required. This is the only space required (in addition to the space for the input axiom), although the variables in the axiom may be instantiated to terms of arbitrary size and complexity.

The instantiation list is a vector of entries. Each entry (or value cell) corresponds to a variable in the axiom associated with the environment containing the instantiation list. The first instantiation list entry corresponds to the variable with canonical number 1, the second entry to the variable with canonical number 2 and so on. The displacement specified in the

term word for a variable is actually the displacement of the value cell for that variable from the beginning of the environment. Using this displacement, the value cell for a variable can be referenced directly, with no search necessary.

The following diagram shows a constructed term representation for the term F(3,G(*X,H(M)),H(M)). This term has been constructed from the input term F(3,G(*X,*Y),*Y) by substituting H(*Z) for *Y and then substituting M for *Z.




## 4.4 Unification

The unification algorithm used is a simple depth first algorithm. The algorithm attempts to match two constructed terms. The matching process either

succeeds or fails. If it fails then the effect of any substitutions made during the unification attempt is removed (i.e. the value cells for any variables which were instantiated are reset to "unassigned").

The unification algorithm does not have an "occurs" check. These means that it will not detect that *X and F(*X) are not unifiable. An attempt to unify these two terms will cause F(*X) to be substituted for *X. Printing the resultant term will generate an "infinite" output of the form F(F(F... This check is omitted for reasons of efficiency and it appears that the occurs check is seldom necessary in PROLOG programming.

The structure sharing technique used to represent constructed terms requires the unification algorithm to "look up" the value of any variable that it encounters. The fundamental step of this lookup process is called dereferencing. Corresponding to a constructed term is a dereferenced value which is derived as follows:

(1) If the term word of the constructed term does not specify a variable then the dereferenced value is the constructed term itself.

(2) If the input term is an uninstantiated

variable (indicated by an unassigned value cell for the variable), then the dereferenced value of the constructed term is the constructed term itself.

(3) If the term word specifies an instantiated variable then the dereferenced value of the term is the dereferenced value of the constructed term in the value cell for the variable.

Note that in (3) above, a search down a chain of references will occur in the case of a variable bound to a variable which is bound to a variable, etc. By dereferencing all terms the unification algorithm attempts to reduce the time required subsequently to retrieve the value of a term.

In the following description of the unification process, it is assumed that all constructed terms are dereferenced before checking them for matching. Also the statement "A is a skeleton" is used as an abbreviation for "The dereferenced value of A is a skeleton". Similarly "A is a variable" is used as an abbreviation for "The dereferenced value of A is an

uninstantiated variable". The various cases which occur in the unification of the two constructed terms A and B are described below:

(1) A and B are both variables. If A and B both refer to the same variable (i.e. the addresses of their value cells are equal), then return "success". If A and B refer to different variables then the variables must become bound to each other. We must decide whether to substitute A for B or B for A. The rule used is: Substitute the variable whose value cell has the lowest address for the variable whose value cell has the highest address. This ordering is selected so that a trace entry is generated as seldom as possible(trace entries are described in 4.7 Backtracking and Trace Entries). Perform the indicated substitution and return "success". The substitution of (say) B for A is performed by placing the constructed term for B in the value cell for A. This is called "assigning B to A".

(2) A is a variable and B is not a variable.

Assign B to A and return "success".

(3) B is a variable and A is not a variable. Assign A to B and return "success".

(4) A is a constant and B is not a variable. If B is a constant equal in value to A then return "success", otherwise return "failure".

(5) A is a skeleton and B is not a variable. If B is not a skeleton then return "failure". If the skeleton name or number of arguments for B is not the same as for A then return "failure". If the name and number of arguments match then call the unification algorithm recursively for each pair of corresponding argument terms. If any of these unifications fail then return "failure". If they all succeed then return "success".

In order to implement the recursion required when matching skeletons, a unification stack is used. Each entry in the stack contains five words, namely:

- the term word for A.
- the environment pointer for A.
- the term word for B.

- the environment pointer for B.
- the index of the current skeleton argument.

## 4.5 Axiom Environments

Corresponding to each activation of an axiom there is an environment. The environment contains an instantiation list as described earlier. The environment also contains other information required for backtracking.

The nature of the backtracking algorithm ensures that the lifetimes of axiom environments are nested. More explicitly, if environment A is created before environment B then environment B will be annihilated before A. Consequently environments may be allocated and freed according to a stack discipline in the environment stack.

Three registers are reserved for pointing to the environment stack. The register names and their respective uses are:

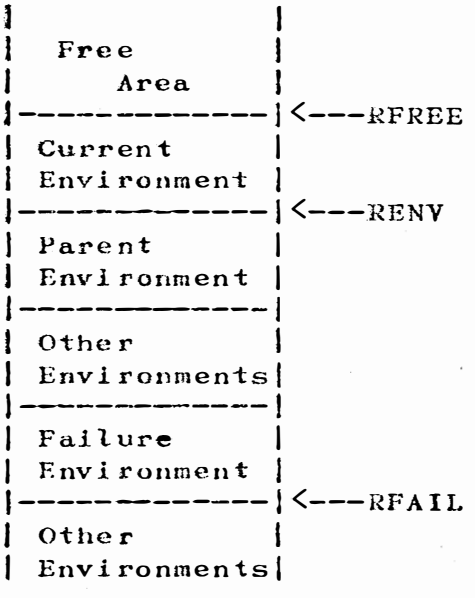
RFREE points to the beginning of the free

area on the environment stack.

RENV points to the current environment.

RFAIL points to the failure environment( the failure environment is explained in 4.7 Backtracking and Trace Entries ).

When an axiom is about to be invoked, RENV points to the parent environment , that is, the environment for the axiom which contains the "call" to the current axiom. To create the axiom environment the free pointer (RFREE) is incremented by the size of the new environment and the environment pointer (RENV) is set to the newly created environment. The following diagram illustrates the use of these registers.



An environment contains the following fields:

- failure code and failure pointer.
- failure environment pointer.
- success code and success pointer.
- success environment pointer.
- term word for argument literal.
- the instantiation list.

The data related to failure is described in 4.7 Backtracking and Trace Entries. The success environment pointer is a pointer to the parent environment which is the environment to which control will be returned when this subgoal succeeds. The argument literal is the term from the parent axiom with which the head of this axiom will be unified. The term word for the argument literal is a word describing the argument literal in the format (term ID,pointer). Since the argument term must be an atom or a skeleton the standard format for the term ID is relaxed. An atom is represented by the standard code. A skeleton may be represented using any other eight bit code.

The success code and success pointer occupy one word. On entry to an axiom this code/pointer is contained in register RRET. The code can have two

interpretations:

(1) The code is zero. This occurs if the parent axiom is interpreted. In this case the pointer points to the term word for the remaining conjunction of the right hand side of the axiom. A success return will go back to the appropriate place in the interpreter to process this conjunction.

(2) The code is nonzero. This occurs if the parent axiom is compiled. In this case the pointer gives the address in the parent compiled routine to which return is to be made. This code pointer pair may be set in the RRET register with a single BAL or BALR instruction.

On entry to an axiom, an environment is allocated. The size of the environment depends on the size of the instantiation list. The number of entries in the instantiation list is equal to the number of distinct variables in the original axiom. On entry to an axiom all entries in the instantiation list are marked as "unassigned" by setting the term word in the entry to

0.

## 4.6 The Main Interpreter Routine

The main interpreter routine interprets axioms. It is called from a parent axiom and passed an argument literal as a parameter. The first axiom whose head has the same predicate name and number of arguments is activated. An environment is created and initialized and the unification of the argument literal and the axiom head is attempted. If the unification succeeds then the literals of the axiom body are each "called" in turn. If the original unification fails or any of the "called" literals fail, then the backtracking routine is invoked.

A significant feature of the main interpreter routine is the means of accessing axioms. The string descriptor address is obtained from the argument literal. This string descriptor is the head of a queue of predicate entries. There are two types of predicate entry. One is the system entry which contains information about operator declarations and file

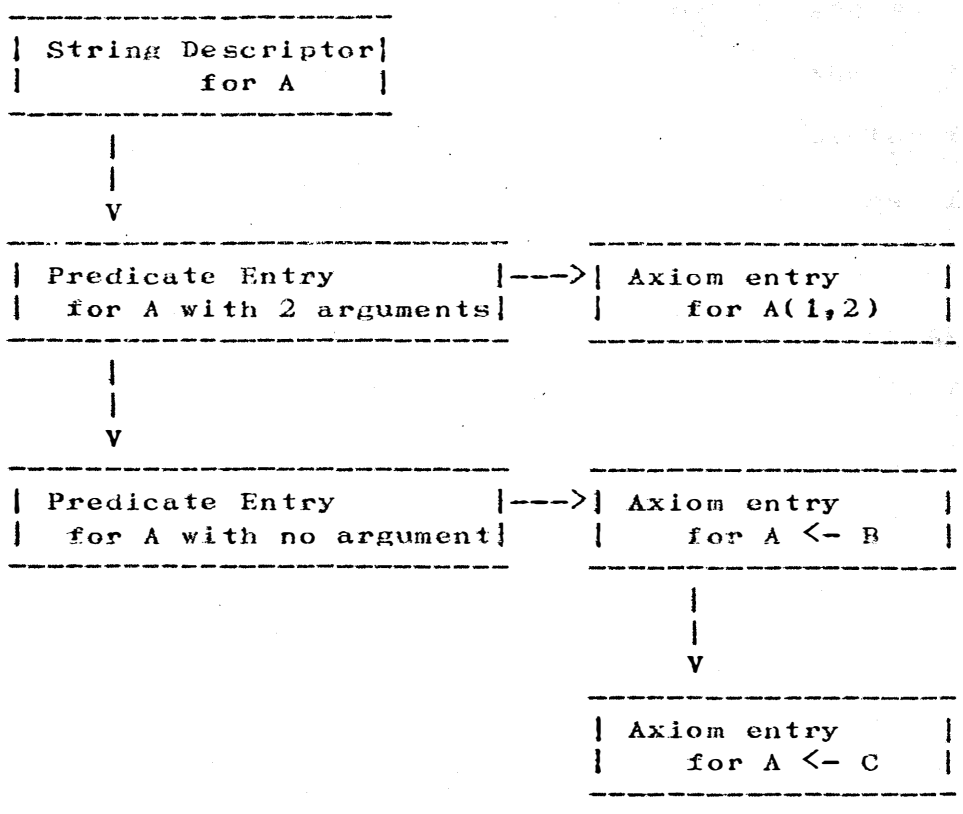
identifiers. The other type of predicate entry contains information about axioms for the given predicate name and a specific number of arguments. The information contained is a pointer to a list of axioms or to a routine for a "compiled" axiom. The majority of the built-in predicates are implemented in assembler code and they are accessed through this routine-type entry.

For a routine entry the predicate entry pointer gives the address of the routine. The routine consists of the routine entry data followed by the actual code. The principal element of the entry data is the environment size. To call a predicate routine, control is passed to a common entry sequence which allocates an environment and saves the important values in the environment. Control is then passed to the routine code.

For a predicate entry of the axiom type the pointer gives the address of the first of a queue of axiom entries. Each axiom entry describes one axiom. The entry also contains a word indicating the size of the instantiation list so that the interpreter routine can allocate an environment of the correct size.

The relationship of string descriptors and

predicate entries is illustrated by the following diagram. The axiom entries are shown for the axioms A(1,2), A <- B and A <- C.




## 4.7 Backtracking and Trace Entries

The backtracking routine is called from the unification routine and from compiled axiom routines

when a failure occurs. The basic functions performed in backtracking are:

(a) Determine the environment to which the proof must "backup" and reset the current environment pointer to this environment. This environment is called the failure environment. Reset the free pointer to free all environments subsequent to the failure environment. Adjust the free pointer to change the size of the failure environment (the next axiom may need an instantiation list of different size).

(b) Remove the effect of all substitutions made since the failure environment was activated.

(c) Reload the previous failure environment pointer from the failure environment.

(d) Load the failure pointer from the failure environment and return either to the interpreter or the compiled axiom routine.

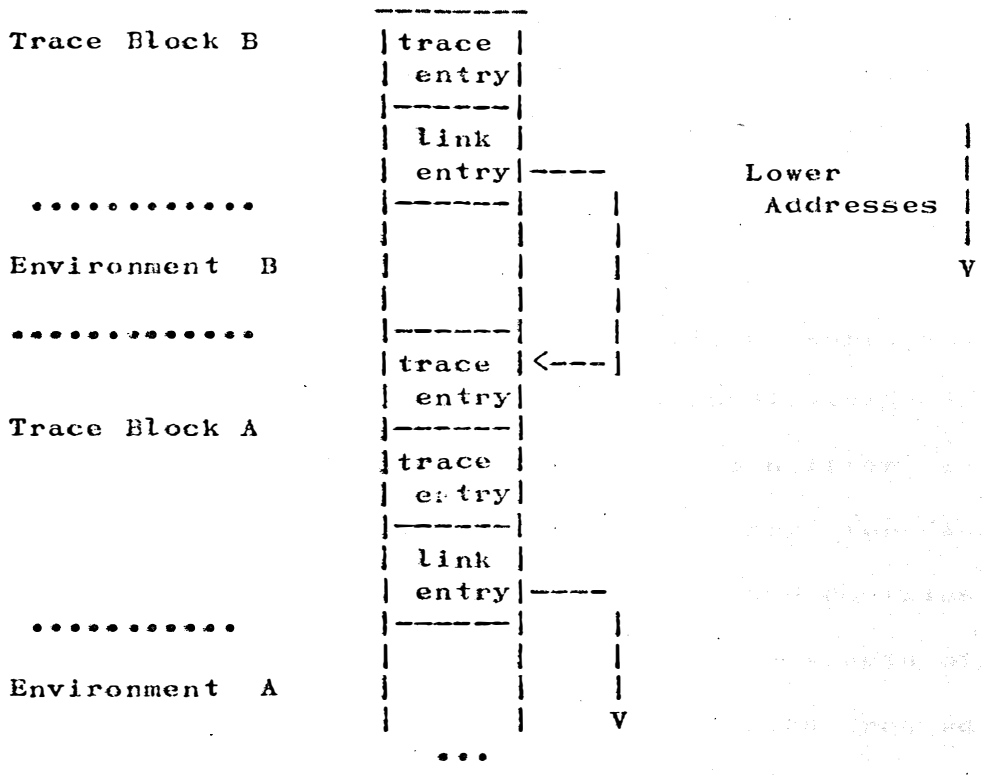
The address of the current failure environment (i.e the address of the environment to which the proof must "backup"), is always contained in the failure

register, RFAIL. On entry to an axiom this pointer is saved in the new environment so that it may be restored on backtracking(step (c) above). The processing of an axiom may or may not reset the failure environment. If the last axiom for a given predicate name and number of arguments is being processed, the failure environment pointer is not changed, since no new alternatives have been introduced. On the other hand if the axiom is not the last then the failure environment pointer is reset to point to the current environment and the failure code/pointer in the current environment is set appropriately.

In order to remove the effects of the appropriate substitutions during backtracking, a record is kept of substitutions. This record takes the form of trace entries, each of which contains the address of the value cell which was set by a substitution. To "undo" the substitution corresponding to a trace entry it is necessary only to set the value cell indicated by the entry to "unassigned". When backtracking is performed, the list of trace entries is processed and all substitutions made since the activation of the failure environment are erased. There is no reason to reset the

value cells in the failure environment and subsequent environments since these environments are freed in the backtracking process. Consequently, trace entries are not generated for assignments into the failure environment and subsequent environments. For this reason, when unifying two variables, the substitution is always performed so that the most recently created value cell is modified. This causes a trace entry to be generated only when necessary.

Each trace entry occupies one word. Trace entries are organized in blocks which are placed in the environment stack. The first entry in a trace block (i.e the entry with the lowest address) is a special entry called a link entry. This entry points to the top of the previous trace block, in order that the backtrack routine can process each trace entry in turn. The following diagram illustrates the structure:




Trace block A contains entries for substitutions made after the activation of environment A and before the activation of environment B. Interlacing trace blocks and environments not only uses one stack instead of two, but it also provides an implicit record of when each trace entry was created, relative to the activation of axioms. The backtracking routine processes only the trace entries whose addresses are greater than the address of the failure environment, since these entries were created after the failure environment. The address of the top of the most recent

trace block is maintained in register RTRACE.

## 4.8 Symbol Table Organization

Each identifier used in the active workspace (either as an atom or as a skeleton identifier) is entered in the symbol table when the identifier is first encountered. The symbol table entry for an identifier is called a string descriptor and contains the value of the identifier, as well as the length of the value. After a string descriptor has been created for a given identifier, all subsequent references to the identifier are made via a pointer to the string descriptor. Consequently the values of two identifiers are equal if and only if the string descriptor pointers are equal.

When an identifier is encountered in the input (or created during execution using the STRING predicate), a search is made to see if a string descriptor exists for the identifier. If the search fails then a new descriptor is created. This search-and-add procedure is performed by the HASH routine.

The symbol table search is made using hash chains. HASHHEAD is a vector of link entries. The value of an identifier is hashed (using the HASH macro) to give an integer x. The link entry indicated by HASHHEAD(x) is selected as the head of a chain of string descriptors to be searched.

Each string descriptor contains the following data:

- the value of the identifier.
- the length of the value.
- the attributes of the identifier.
- a link entry.

An identifier can have any of the following attributes:

PREFIX - the identifier is declared as a prefix operator.

SUFFIX - the identifier is declared as a suffix operator.

LR - the identifier is declared as a left-to-right infix operator.

RL - the identifier is declared as a right-to-left infix operator.

SPECIAL - the identifier consists of one or two special characters and need not be

separated from its operands by a blank when used as a prefix, suffix or infix operator.

For identifiers consisting of one special character, the SPECIAL attribute is set when the string descriptor is constructed by the HASH routine. The SPECIAL attribute is also set for an identifier consisting of two special characters when an operator declaration is made for the identifier. The other four attributes are maintained through the addition and deletion of axioms for the OP predicate.

A link entry is a pair of the form (code,pointer). The possible forms of this pair are:

(LAST,-)

(DIRECT,pointer)

(INDIRECT,pointer)

If the code is LAST then this is the last entry in the chain. If the code is DIRECT then the pointer addresses the next string descriptor entry in the chain. If the code is INDIRECT then the pointer addresses a predicate table entry for this string descriptor. The predicate table entry contains a link entry which continues the hash chain. The link entry in

the predicate entry can have the form (LAST,-) or (DIRECT,pointer).

Each string descriptor can have any number of corresponding predicate entries, organized in a queue. Predicate entries may be of three types; namely a system type, an axiom type or a routine type. The axiom and routine types each contain a number of arguments field and a pointer. The number of arguments field indicates that the entry applies to the predicate with the given identifier and the indicated number of arguments. For an axiom type entry the pointer indicates the first entry in an axiom queue. For a routine type entry the pointer indicates the routine entry sequence. Predicate entries of the system type are distinguished by a number of arguments field which is negative. System type entries are used to record operator declarations and file information associated with the identifier.

## 4.9 Storage Management

The initialization routine acquires one large area

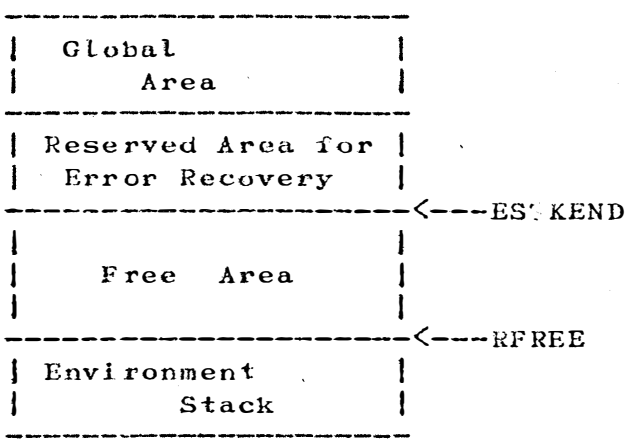
from which all storage requirements are satisfied. The two principal requirements for storage are for the global area and the environment stack. The environment stack starts at the bottom of the acquired region and grows upward, while the global area starts at the top of the region and grows downward. The main elements allocated in the global area are:

(1) string descriptors

(2) axiom entries

(3) axiom routines

The limit pointer for the environment stack is maintained in ESTKEND. ESTKEND is always set to be at least one environment size below the actual limit since the routine entry sequence saves several pointers in the new environment before checking for space. Normally ESTKEND is maintained well below the bottom of the global area to reserve space for the error handling axioms, which will be called if the environment stack reaches ESTKEND. The following diagram shows the various areas.




The top of the environment stack is also used temporarily for write entries(in TMPUT) and for parse stack entries(in TMGET). During unification a unification stack is created which starts at ESTKEND and grows downward. Since string descriptors can be allocated in the global area during parsing, RFREE is temporarily set to ESTKEND when reading terms in order to prevent string descriptors from overlaying the area containing variable names and the input term skeletons.

## 4.10 Axiom Management

As described earlier, axioms are stored as *axiom entries* which are queued from predicate entries. Each

axiom entry contain the following data:

- a link to the next entry or a "last" flag,
- the term word and skeleton descriptors representing the axiom,
- a count of the number of value cells needed for the axiom environment,
- a "free link".

The "free link" is used to maintain a special queue of deleted axioms. When an axiom is deleted the space may not be immediately freed since constructed terms in the proof may reference the axiom or subterms of it. In order to recover space from deleted axioms it is necessary to defer the freeing of the axiom space. An axiom entry is queued on a special deferred free queue when it is deleted. When a proof is completed and the supervisor is about to read another goal or axiom, entries on this queue are then transferred into the "real" free queue.

## 4.11 Reading of Terms

All axioms and terms in the system are read using the same mechanism, a stack driven parser. The parser uses two separate stacks, a token stack and a term stack. The internal representation for the term being parsed is constructed on the term stack. The term stack can consequently contain zero or more skeleton descriptors. The parse stack contains tokens and is used to control the parsing process.

The first step in reading a term is the tokenization of the input. The TKGET routine is called to return a token on the top of the parse stack. The token returned is one of the following types:

<identifier>,

<term>,

<comma>,

<left parenthesis>,

<right parenthesis>,

<end of term>.

The token of type <term> is returned for integers and variables.

The parse stack entry describing the token

contains the following information:

- the token type as described above
- the token priority(initially set to -1 by TKGET and may be reset by TMGET)
- The token value. This is relevent for TERM and ID type tokens only. The value is represented by a term word. An identifier is represented by the term word for the corresponding atom.

The parsing is performed using a shift-reduce algorithm(1). The TMGET routine is the parse driver routine. It calls TKGET to place a token on the stack, and then performs any appropriate reductions. The reductions which TMGET may perform correspond to the following BNF rules.

<atom>::=<identifier>

<left-to-right operator>::=<identifier>

<right-to-left operator>::=<identifier>

<prefix operator>::=<identifier>

<suffix operator>::=<identifier>

<term>::=<atom>

The appropriate reduction (if any) is determined using

a two symbol look ahead and also examining operator declarations. The TMGET routine calls the RESOLVE routine when other reductions may be required. RESOLVE is called if the top of the parse stack is neither a term nor an identifier.

The RESOLVE routine makes reductions corresponding to the following BNF rules.

```
<term>::= ( <term> )     |
          <skeleton>

<skeleton>::= <identifier> ( <argument list> )   |
              <term> <infix operator> <term>     |
              <prefix operator> <term>           |
              <term> <suffix operator>

<argument list>::= <term>     |
                   <argument list> , <term>
```

The RESOLVE routine determines the applicable reduction (if any) by examining the type and priority of the top three stack entries. When a reduction produces a skeleton, RESOLVE builds an appropriate skeleton descriptor on the term stack. The term word for the descriptor is placed in the stack entry produced by the reduction.

## 4.12 Writing of Terms

All terms and axioms are written using the same mechanism. TMPUT is a stack-driven routine which writes out terms using the current operator declarations. Terms are written in a minimally parenthesized form. Consider as an example the term .(A,(+(-(b,!(C)),D)) with the following operator declarations:

```
OP(.,RL,100).

OP(-,RL,150).

OP(+,RL,200).

OP(!,SUFFIX,250).
```

We begin by processing the outer skeleton. This will be written in the format "_._" where _ denotes a subterm whose format has not yet been determined. The first argument of the outer skeleton is processed and the output format is now "A._". We now process the "+" subterm. This subterm will be written in infix notation. To decide whether to parenthesize the subterm we examine the left priority context and the right priority context. In this case the left priority context is 100, the priority of the infix ".". In general the left priority context of a subterm is

defined as follows:

(1) If the subterm appears immediately to the right of an infix operator then the left priority context is the priority of that infix operator.

(2) If the subterm appears immediately to the right of a prefix operator then the left priority context is the priority of the prefix operator.

(3) Otherwise the left priority context is 0.

Similarly we define the right priority context of a subterm as follows:

(1) If the subterm appears immediately to the left of an infix operator then the right priority context is the priority of the infix operator.

(2) If the subterm appears immediately to the left of a suffix operator the the right priority context is the priority of the suffix operator.

(3) Otherwise the right priority context is 0.

In this example, the right priority context of the "+" subterm is 0. Since the priority of the infix "+" is greater than both the right and left priority contexts, this subterm need not be parenthesized. Thus the output will have the format "A._+_" and we next process the "-" subterm.

This subterm has a left priority context of 100 and a right priority context of 200. Since the infix "-" has a priority of 150, which is less than 200, we must parenthesize this subterm. The output now has format "A.(_-_)+_" and we next process the first argument of the "-" which is an atom. This gives an output format of "A.(B-_)+_". We next process the "!(C)" subterm. The "!" is a suffix operator. To determine whether to parenthesize a term in suffix format, it is necessary to examine only the left priority context. In this case the left priority context is 150 and "!" has priority 250, so no parentheses are required. The output format is now "A.(B-_!)+_". When the two remaining subterms are processed we have the final output: "A.(B-C!)+D".

This example demonstrates the basis of the "minimal parenthesis" algorithm developed for this

implementation. Prefix subterms were not discussed but they are treated analogously to suffix subterms: A subterm in prefix format needs to be parenthesized if the right priority context is greater than the priority of the prefix operator.

This basic method is refined in order to handle left-to-right and right-to-left infix operators. Firstly all the operator priorities are multiplied by four so that any two distinct priorities differ by at least four. For infix operators two priorities are created: a right priority and a left priority. Intuitively, the left priority is the priority visible from the left and the right priority is the priority visible from the right. Thus, in comparing the priorities of two infix operators to decide on parenthesizing, the left priority of the right one is compared with the right priority of the left. For both right-to-left and left-to-right operators the left priority is the same as the priority. For a right-to-left operator the right priority is one less than the priority. For a left-to-right operator the right priority is one more than the priority. This refinement for infix operators extends the basic method

to handle left-to-right and right-to-left operators correctly, when two adjacent infix operators have equal priority.

The term writing routine is effectively a stack-driven tree traversal program. The stack consists of zero or more write entries. Each write entry represents a subterm of the original term (or equivalently a node in the tree representation of the term). Each entry contains the following fields:

- the substitution environment of the subterm
- a pointer to the current argument in the argument list of the subterm skeleton descriptor
- a count of the number of arguments which remain to be processed in this subterm
- the left priority context
- the right priority context
- a flag indicating whether or not the subterm is parenthesized
- flags indicating if the subterm is being written in infix, prefix, suffix or basic skeleton notation

# 5 Design Decisions

## 5.1 Introduction

This section describes many of the more significant design decisions made in implementing the system. An attempt is made to outline the motivations for the various decisions and the alternatives that were considered. The features affected by the design decisions fall into two groups. Language features are readily visible to the user of the system. Internal features are not readily visible but have implications regarding efficiency and ease of implementation. The major features of both types are discussed in the following sections.

## 5.2 Infix, Prefix and Suffix Operators

It is clear from experience with PROLOG that the ability to declare operators as infix or prefix is very useful. This feature is retained in an essentially unaltered form. A single operator may have both a

prefix and an infix declaration. This dual declaration is allowed because of its obvious application for operators such as "+" and "-". The "<-" operator is also used in both prefix and infix forms to represent goals and axioms respectively.

In some situations suffix operators allow a more natural notation. Consequently suffix declarations are also supported. In order to prevent ambigous representations and to simplify the parsing of terms the restriction is imposed that a suffix operator may not simultaneously be declared as infix or prefix.

To provide a more flexible user interface, operator declarations may be accessed, added or deleted by manipulating the OP built-in predicate.

## 5.3 The Representation of Terms

A flexible form of input for terms is provided. A term may span several input lines or several terms may be input on one line. Since infix operators are allowed it is necessary to indicate the end of an input term by using some sort of delimiter. The end-of-term

delimiter is chosen to be ".". The character "." as the last character of an input line is treated as an end-of-term. The character "." followed by a blank marks the end of a term, wherever it occurs in a line. This imposes restrictions on the use of "." as an operator: it may not be the last character on a line and may not be followed by a blank. Alternatively a special symbol could have been reserved as the end-of-term delimiter and disallowed as an operator. The obvious choice for such a special character was ";", following its use in other languages, but this would have disallowed the use of ";" as an operator.

A variable is represented by an asterisk followed by the variable name. The use of anonymous variables (i.e. variables with no name) is introduced to abbreviate the notation in certain cases. The variable name may be any of a sequence of letters and digits. When a term is written the variables in the term are given names according to their canonical numbers (i.e. *1,*2 etc.). This gives the motivation for allowing variable names consisting solely of digits.

An atom or identifier is represented by a sequence of characters enclosed in apostrophes. In certain cases

the enclosing apostrophes may be omitted. The use of apostrophes allows the use of punctuation characters and special characters in atoms and identifiers. Obviously, the apostrophes are made optional to yield a briefer and more readable syntax.

Since a large number of printers and interactive terminals are unable to deal with lower case letters, lower case letters are not allowed in unquoted identifiers. For this same reason the default is for all input from the terminal to be translated to upper case. To maintain flexibility, a facility is provided to avoid the translation (i.e. the CONTROL predicate with the key LOWER).

## 5.4 The Representation of Axioms and Goals

In the original Marseille version of PROLOG, an axiom is represented as a sequence of signed literals(e.g. +P-Q-R). There are several disadvantages to this notation. Firstly the "+" and "-" signs give a deceptive indication of generality, but add nothing to the power of the language. The inference rules used by

the PROLOG proof procedure can be explained using the simpler and more natural rules of implication rather than the more general rules of resolution.

In this implementation an axiom is represented as an implication(e.g. P<-Q&R). This representation has the advantage that an axiom can now be interpreted in an obvious way as a term where "<-" and "&" are declared as infix operators of the appropriate priorities. Representing an axiom as a term of a certain form yields several benefits:

(a) Axioms and terms can now be read and written using a common mechanism.

(b) Axioms can be easily manipulated without resorting to a special list format.

(c) Representations for alternation and negation can now be easily included. If the identifier '¬' is declared as a prefix operator of appropriate priority then negated literals can be used.

e.g. A<-B&¬C.

¬C<-E&F.

(d) Infix, prefix and suffix forms of predicates can be used since axioms are just special

cases of terms.

e.g. If LIKES is declared as infix then

A LIKES B is a valid axiom.

When an axiom is invoked in a proof, the right hand side or body of the axiom becomes a subgoal in the proof. Because of the similarity of function between a goal and an axiom body, a similar format is chosen for both.

Examples of goals are:

<-A.

<-F(*X)&G.

The operator "<-" is declared as prefix so that a goal is also a term. This notation has the further advantage that goals and axioms are now easily distinguished without depending on context. Consequently they may be freely interspersed in the input.

## 5.5 Built-in Predicates

This implementation includes some of the built-in predicates of the original PROLOG in a modified form.

Other of the original predicates are omitted entirely and some new built-in predicates are included. These changes were made in an attempt to achieve the following goals:

(a) a more powerful and uniform system for manipulating axioms and operator declarations. An attempt was made to make the language more "self-conscious", that is, to provide the ability to access and change all aspects of the PROLOG environment from PROLOG programs.

(b) improved access to external files. Any number of files may be accessed by name. The original PROLOG system provided a single fixed input file.

(c) more "meta" facilities. In particular the ability to determine the type of a term (i.e. variable, skeleton, atom or integer).

(d) more general facilities for manipulating PROLOG workspaces. The original PROLOG provided a simple SAVE function only. The extensive workspace facilities described are based on similar funtions provided by the APL

language.

(e) improved facilities for testing and error recovery. The ERROR predicate is provided to allow user controlled display of information at the point of an error. This feature is similar to the ON ERROR facility of the PL/I language.

## 5.6 Predicates, Skeletons and Their Arity

A skeleton is determined by a skeleton identifier and a number of arguments. The question arises: Should a fixed number of arguments be associated with each identifier? More specifically: Is it appropriate to use the skeletons F(1,2) and F(1,2,3) in the same proof? Placing restrictions on the number of arguments would have the advantage of detecting certain user errors (such as typing F(1.2,3.4) instead of F(1.2,3,4)). However the restrictions would preclude the natural use of "+" and "-" as both unary and binary operators. Also, it is difficult to determine the "correct" number of arguments for an identifier without introducing some

form of declarations. Primarily for these reasons it was decided not to associate a specific number of arguments with a skeleton identifier. A similar discussion applies to the use of a predicate identifier with various numbers of arguments. The simplicity and usefulness of "optional arguments" is demonstrated by many of the built-in predicates provided in this implementation. The reduced facility for error detection is alleviated by the provision of the NOAX option in the CONTROL built-in predicate.

## 5.7 Internal Features

The design of internal features is influenced by two considerations: efficiency and ease of implementation. The efficient and elegant structure sharing method for representing terms, used in the Marseille implementation, is employed with no major changes. This representation also allows the use of a stack for axiom environments.

An area, called the global area, is also required for permanent data items, such as axioms and string

descriptors. The global area and the environment stack are allocated at opposite ends of a common area and grow towards each other.

Certain substitutions must be recorded as they are made so that they may be "undone" by backtracking. To this end it is necessary to save the address of each value cell in which a substitution was made. Thus we need to accumulate a list of value cell addresses or trace entries. When backtracking is performed it is necessary to determine all trace entries which have been created since the creation of a given axiom environment. To provide a record of the time of creation of trace entries versus axiom environments and to simplify storage management, it was decided to place the trace entries on the environment stack.

An early decision was made to support a mixture of compiled and interpreted axioms. It was decided that this facility merged the best features of convenient program development and efficient execution. The provision of this mixed feature influenced the format chosen for axiom environments.

For an axiom interpreter it is necessary to record in the environment, the next axiom alternative

available(in case of failure) and the remaining goal conjunction of the current axiom (in case of success). Each of these two items can be recorded in a three byte address. For a compiled routine it is necessary to record the success return address and the failure return address. These too can each be recorded in a three byte address. To maintain reasonable space efficiency it was decided to use the same two words in the environment to record the two items for the interpreter and the two items for compiled code. The high order byte of each word would contain a code indicating the type (compiled or interpreted). A code is used in both words to provide increased flexibility: when the interpreter processes the last literal of the axiom body, it flags the success pointer with the "compiled" code and sets the success pointer to a routine which immediately "succeeds".

Backtracking could have been handled in any of several ways. The current environment could always contain a pointer to the previous environment on the stack. Then backtracking could trace back through the environments on the stack until an environment with a remaining alternative was found. It is more efficient

to have the current environment contain a pointer to the most recent environment with a remaining alternative, so that the appropriate environment can be located in a single step. To facilitate this, a register pointing to the current failure environment is maintained. This register also allows a more efficient handling of trace entries. It is necessary to trace only those assignments into value cells below the current failure environment since backtracking will erase the failure

An atom is represented by the address of a symbol table entry. The alternative of representing each atom by a separate string was rejected. This alternative would necessitate time-consuming comparison of strings during unification. Hashing seemed to be the only reasonable access method for a symbol table. Since the number of atoms in use in different PROLOG workspaces varies a great deal, the method of hashing into a symbol table of fixed size was rejected. The use of hash chains was deemed the best choice. To facilitate parsing of input and the implementation of the STRING built-in predicate, the maximum length for an identifier was set at 256 characters.

The memory architecture of the /370 was a strong influence in determining the representation for terms. The storing of a twenty-four bit address in a thirty-two bit word allows efficient representation of a term using an eight bit code and a twenty-four bit address or number. The selection of the codes for the four types of terms was also based on efficiency considerations. The efficiency of the dereferencing of terms is very important and depends on the rapid recognition of variables. Consequently the type code to represent a variable was chosen so that the word representing a variable would be negative and the word representing any other type of term would be positive.

Since skeletons vary in size, it seemed necessary to represent a skeleton term by a pointer to some sort of skeleton descriptor. This descriptor would need to contain the following information:

- the skeleton name
- the number of arguments
- the arguments

The arguments can each be represented by a single term word. The skeleton name can be represented by a twenty-four bit pointer to a symbol table entry. It is

desirable to make the representation for a skeleton an integral number of words in size. Consequently the choices open are to restrict the maximum number of arguments to 255 and record the value in eight bits or to reserve an extra word for the number of arguments. The limit of 255 arguments seemed reasonable, so that alternative was chosen.

In a PROLOG workspace, various pieces of information are associated with identifiers. The information which can be stored for an identifier can include all or any of the following:

- operator declarations
- axioms
- routines (for built-in predicates and compiled axioms)
- file information
- information for the CONTROL built-in predicate

It must be possible to access this information from the string descriptor for the identifier. Consequently it was decided to provide for the chaining of predicate entries from string descriptors. Instead of reserving space in each string descriptor for a pointer to a

predicate entry chain, the technique of indirect pointers described in 4.8 Symbol Table Organization was used.

It was anticipated that compiled code routines would need to access predicate entries directly to obtain the addresses of other routines. Accordingly, the predicate entries are organized in the predicate table. The entries may be accessed using an offset from the table base. In normal execution a fixed register is reserved for the predicate table base. It is assumed that the maximum predicate table size will be restricted to 4096 bytes, in keeping with the limit on base-displacement addressing on the /370. This limit is not yet imposed since the compiler has not been implemented.

It was decided to provide the implementation with a facility for programmable error recovery. This required the reservation of space on the stack for the execution of user written error axioms in the event of a stack overflow. Consequently a reserved space system is implemented to provide space for stack growth if an overflow is detected.

# 6 Future Considerations for PROLOG

## 6.1 Introduction

This implementation includes numerous significant changes and extensions to the original PROLOG language. PROLOG is a relatively new language and consequently there is a wealth of further extensions which may be considered. Some of the possible extensions are small in scope and can be implemented with relatively little effort. Others represent major changes to the power of the language and consequently major efforts in implementation. The following sections outline some possible extensions.

## 6.2 More Built-in Predicates

The PROLOG language can always be extended by adding more built-in predicates. There is a tradeoff in adding these, since every addition makes the language more difficult to learn and increases the size and complexity of the system. With these considerations in

mind, the following predicates are suggested.

PROVABLE( *X )

This predicate succeeds if some instance of *X is provable from the axioms. This predicate could be implemented by proving *X in the usual manner and then erasing th proof of *X and any substitutions made during the proof. This predicate can be defined in the existing implementation using the meta variable facility and the slash. The provision of this and some of the following built-in predicates would standardize the predicate names used for several common functions. This standardization would also allow a compiler to recognize certain standard predicates with defined characteristics and to optimize accordingly.

UNPROVABLE( *X )

This predicate succeeds if no instance of *X is provable from the axioms. Note that no instantiation is performed. This predicate can be defined in the existing implementation using the meta variable facility and the slash. PROVABLE

could be defined by:

PROVABLE(*X) <- UNPROVABLE(UNPROVABLE(*X)).

UNIFY(*X,*Y)

This predicate succeeds if *X and *Y are unifiable. If the predicate succeeds then *X and *Y are unified. This predicate could be defined in the existing system using the axiom UNIFY(*X,*X).

UNIFIABLE(*X,*Y)

This predicate succeeds if *X and *Y are unifiable. No unification is performed.

DUPLICATE(*X,*Y)

This predicate succeeds if *Y can be unified with a copy of *X. More specifically, a copy of the term bound to *X is created with the variables renamed. DUPLICATE is closely related to the above predicates.

PROVABLE(*X) can be defined by:

PROVABLE(*X) <-DUPLICATE(*X,*Y)&*Y.

UNIFIABLE(*X,*Y) can be defined by:

UNIFIABLE(*X,*Y) <- DUPLICATE(*X,*Z) &

DUPLICATE(*Y,*Z).

INSTANCE(*X)

This predicate is proposed as a means of instantiating all variables in the term *X. The features appropriate for a predicate of this sort are not readily apparent. One suggestion is that in this form ,the variables are unified with the integers 1,2,3, etc., to provide a "most specific instance" of the term. A more general form might be INSTANCE(*X,*Y) where the term *Y is used as a model for the instantiation of the variables in the term *X. For instance, if *Y is bound to V(*) then the variables in *X would be instantiated to V(1), V(2), etc. The instance predicate would be useful in a compiler for compiling PROLOG axioms directly into code. It could also be used as a general "meta" facility for terms. For example, it could be used to replace the MKGROUND predicate in WARPLAN(9).

CONDENSE

This predicate could be implemented as a predicate

with a pragmatic significance, but no semantic significance. Specifically, it would recover space on the environment stack by causing a portion of the proof to be condensed. The desirability of such a feature is clearly dependent on the implementation.

COUNT

This predicate is proposed as a means of providing loop control. It is suggested that four forms of this predicate be provided.

COUNT - this predicate succeeds when first invoked and when backtracked to. Thus it can be used to perform looping. The loop can only be terminated through the use of RETRY or /.

COUNT(*X) - same as for COUNT but when first invoked *X is instantiated to 1 and backtracking causes *X to be instantiated to 2 then 3 etc.

COUNT(*X,*N) - same as for COUNT(*X) but when first invoked, *X is instantiated to *N (*N must be bound to an integer). Backtracking causes *X to be incremented as before.

COUNT(*X,*N,*M) - same as for COUNT(*X,*N) but backtracking will only succeed while *X is less than or equal to *M. *M must be bound to an integer.

SUBTERM(*SKEL,*INDEX,*RESULT)

This predicate can be used to select the argument of a skeleton with an appropriate index. For example the call <-SUBTERM(F(1,8,27,64),3,*CUBE) will succeed and will set *CUBE to 27. Similarly the call <-SUBTERM(F(1,8,27,64),*I,64) will set *I to 4 (the index of the first argument which is unifiable with 64).

CODEAX(*X)

This predicate succeeds if there is an axiom in

coded form (i.e. compiled) with the same predicate name and number of arguments as the term *X. This is intended for user written theorem provers, to allow them full access to the PROLOG axioms and coded axioms.

CODEAXN(*X,*N)

This predicate succeeds if there is an axiom in coded form (i.e compiled) with the predicate name given by atom *X and with the number of arguments given by integer *N. This predicate is included for uniformity with the AX and AXN predicates.

Built-in predicates might also be useful in the following areas

- adding and deleting compiled code axioms
- more powerful library predicates
- interfacing to subroutines in other languages
- providing special tracing and debugging features
- more general file capabilities

## 6.3 More Realistic Data Base Facilities

Resolution logic has been shown to combine simplicity and power of expression when used as a data base definition/query language. In the practical sense though, this and previous PROLOG implementations have not provided a realistic means for manipulating a data base of significant size. To extend the implementation to include this facility, several features need to be considered. First it will be necessary to develop a technique for storing axioms on an external storage medium. In this implementation, the actual maximum size for the axioms and work areas of the active workspace is 16 megabytes, though the practical maximum is considerably lower. If some axioms are to be stored "internally" and others "externally" then a criterion must be established to determine the storage mode for a given axiom. This could either be determined automatically by the system or specified by the user. For instance a MODE built-in predicate could be provided to allow the user to specify an "internal" or "external" mode for any predicate name. Another feature which is desirable from an efficiency point of

view is the provision of "unordered" axioms. For example, consider a data base consisting entirely of axioms of the form NAME(xxxx) where xxxx is an atom. To determine if an atom "is a NAME" we do not want to search through all the "NAMES". Clearly a hashing technique is desirable. This sort of technique is easiest if we do not need to remember the original order of the NAME axioms. It might be desirable to have a MODE predicate which allows the user to define an "ordered" or "unordered" mode for any predicate name.

Numerous other questions need to be resolved in order to provide an efficient and elegant data base system within PROLOG. 6.4 Real Arithmetic

Arithmetic values in this implementation are restricted to integers. In many cases real arithmetic would also be useful. To include real arithmetic, several decisions must be made. The syntactic representation for real constants must be selected. The unification technique for reals must also be determined. The problem in this area is the means for

comparing reals, since strict equality is probably inadequate due to roundoff errors. The built-in predicates would also have to be modified and extended in order to provide the basic arithmetic operations. A facility for formatting output may also be required.

Support for real arithmetic does not appear to be easy to provide. It may be that the added complexity does not warrant the effort required for implementation.

## 6.5 A More Sophisticated Proof Procedure

The power of PROLOG could be extended by "improving" the proof procedure. The danger is that more elaborate proof procedures incur greater overhead and require more complex data structures. Such changes might erode the very advantages of PROLOG as an efficient (and restrictive) theorem prover.

Numerous avenues remain to be explored in this area. Extensions to provide "bottom up" and "breadth first" facilities need to be investigated further.

## 6.6 Higher Order Facilities

This implementation can be used to "mimic" second order features by using certain built-in predicates. The provision of any true second order facilities needs to be investigated.

## 7 Conclusions

The preceding sections have described the main features of this implementation of PROLOG. The implementation has been completed as described except for the built-in workspace predicates described in 3.5 Workspace Predicates. This implementation was developed on an IBM 370/158 using VM/CMS. Some sample programs were used to compare the efficiency of the new implementation and the original implementation from Marseille. The programs chosen involve the WARPLAN system for plan generation(9). The first set of times compares the time required to load the axioms for the WARPLAN system. The next two sets of times give the times required to solve the problems <-PLANS(ON(A,B),START) and <-PLANS(ON(A,D),START) respectively, using the axioms for the blocks world as described in (9). The comparison is based on seconds of virtual central processor time on a 370/158.

| | Marseille Interpreter | New Interpreter |
|---|---|---|
| Load WARPLAN | 58.2 | .40 |
| Problem 1 | 2.72 | .16 |
| Problem 2 | 3.97 | .24 |

These times give an approximate measure of relative performance. They show an improvement factor of over 15 for execution and over 100 for the loading of axioms. No comparison of the space efficiency of the two implementations has been attempted. No sigrificant differences are anticipated in this area.

Other more subjective evaluations remain to be made. These evaluations will be made by the final users of the system.

It is hoped that this implementation will stimulate the development of the PROLOG language and will provide a base for future enhancements.

## 8 Bibliography


1. Alfred V. Aho and Jeffrey D. Ullman: The Theory of Parsing, Translation and Compiling, Volume I, Prentice-Hall, 1972.

2. G. Battani and H. Meloni: Rapport de D.E.A. d'Informatique Appliquee, Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille, 1973.

3. H. Coelho and L. M. Pereira: GEOM: A PROLOG Geometry Theorem Prover, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1976.

4. M. H. van Emden: Programming with Resolution Logic, Research Report CS75-30, Dept. of Computer Science, Univ. of Waterloo, 1975.

5. H. B. Enderton: A Mathematical Introduction to Logic, Academic Press, 1973.

6. Nils J. Nilsson: Problem Solving Methods in Artificial Intelligence, McGraw-Hill, 1971.

7. Grant M. Roberts: A Macro Reference Manual, Computing Centre, University of Waterloo, 1975.

8. P. Roussel: PROLOG Manuel d'Utilization, Groupe

d'Intelligence Artificielle, Marseilles-Luminy, 1975

9. D. H. Warren: WARPLAN: A System for Generating Plans, Dept. of Computational Logic Memo 76, Edinburgh, 1974.