1985

The design and implementation of a PROLOG interpreter /

Andrew Davison
Lehigh University

THE DESIGN AND IMPLEMENTATION OF A PROLOG INTERPRETER.

by

Andrew Davison.

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in the

Department of Computer Science

and Electrical Engineering

Lehigh University

1985

This thesis is accepted and approved in partial fulfillment of the requirement for the degree of Master of Science.

May 9, 1985
(date)

Samuel L. Gulden
Professor in Charge

[illegible]
Head of the Division of
Computer Science

[illegible]
Chairperson of the Department

Table of Contents

Abstract.

The design and implementation of a PROLOG interpreter, written in PASCAL, is explained. The main design issues of PROLOG are how to deal with its pattern matching and backtracking features. Solutions are given to these problems. In particular, the pattern matching of this interpreter is more powerful than many other PROLOG systems. Algorithms are also included which offer a more flexible way of implementing PROLOG's search strategies.

It is hoped that this interpreter will help in the teaching of PROLOG and also compiler design.

1. Introduction.

The purpose of this report is two fold. Firstly, an explanation will be given of how the PROLOG interpreter was implemented. The program was written solely in PASCAL. It shall be assumed that the reader is already familiar with PASCAL and also with basic compiler/interpreter design. If not then the books by Welsh [1], Welsh & McKeag [2] and Gries [3] are recommended. Only PASCAL features which are complicated or dependent on the host machine (in this case the DEC20) will be described. For more information on the DEC20 and its PASCAL, the user manuals for these can be read [4] [5]. Similarly only design features unique to the interpreter will be described. These considerations have dictated the following chapters which explain how such things as pattern matching and backtracking of PROLOG are dealt with. It is also assumed that the reader is familiar with PROLOG. As explained in the following chapters the version of PROLOG implemented is a large subset of the one explained in the first 11 chapters of the book by Clocksin & Mellish [6]. This book will be referred to using the abbreviation C&M.

The second aim of this report is to explain how to use this PROLOG interpreter and to explain what the user should expect. If the reader is only interested in this, then chapters 3,4 and 5 can be ignored. Instead, it is recommended that the reader should first read C&M and then appendix I and appendix II.

2. Syntax.

The syntax of this PROLOG is laid out in extended Backus-Naur form (EBNF) in the third appendix. In general, the syntax is exactly that of the PROLOG in C&M. However, there are 4 restrictions.

2.1 Restrictions.

The first is that infix and postfix form is not allowed. This will only become a nuisance when arithmetic is being carried out. Thus

```
X is 10 + Y - Z
```

must be written as

```
is( X, +(10, -(Y,Z)))
```

The second restriction is in the notation for lists. C&M allows

```
[ x,y,z]
```

and also

```
.(x, .(y, .(z,[])))
```

These two thing are equivalent. The '.' is also used as a period to terminate a clause. So for ease of implementation the '.' functor is not allowed. Thus

```
.( x, .(y,[]))
```

will give a syntax error. The only list form allowed is

```
[x,y]
```

This is perfectly good in nearly all cases. Unfortunately in C&M, the predicates 'functor', 'arg' and '=..' ('univ') can use this first notation. These 3 predicates can manipulate lists by converting

```
                    [x,y]
        to          .(x, .(y,[]))
```

In this interpreter if such a thing is tried then an error will be output.

Another restriction is the use of built in predicate names as ordinary names. Lists of these built in predicates are given in the fourth appendix. For example, asserta(X) is an one argument built in predicate. When the interpreter accepts the word asserta it will label it as a predicate operator. In this interpreter if a predicate name or symbol is going to appear as an ordinary name or symbol, it is necessary to put it into quotes. Thus

```
?- asserta(fact(1)).
```

will call the built in predicate asserta. But

```
?- 'asserta'(12).
```

will call the ordinary fact, or rule, called asserta. All built in predicates, when used, must use their right number of arguments. Thus

```
    ?- asserta(fact(2)).    is correct

but ?- asserta( fact(3),fact(4) ).   is wrong.
```

A small restriction is that the grammar rule notation of chapter 9 of C&M is not implemented. In most PROLOGs, it is a package built on top of the basic PROLOG. That is what could be done here, if required.

2.2 Discussion of EBNF.

The notation includes

'X', (X), {X}, | and /\

'X'	is used to indicate that X is a terminal symbol.
(..)	is used to group strings together.
{..}	is used to mean 0 or more strings.
|	means 'or'.
/\	means the empty string.

```
Thus
         ( fact | question | rule ) '.'
means          fact '.'
         or    question '.'
         or    rule '.'
```

There are some constraints on the syntax which can not be coded in EBNF. One of these is the fact that comments can appear almost anywhere. Thus you can have

```
          hello(mary).  /* comment */
or        /* comment */ hello(mary).
or        hello  /* comment */ (mary).
```

Comments are consumed in the lexical scanning done by procedure 'nextsymbol' and never reach the syntax analysis sections. Comments and spaces are ignored in between words, numbers and symbols but must not appear in the middle of these things. Thus

```
          hel /* comment */ lo(mary).
or        hel lo(mary).
```

will give syntax errors. A symbol may consist of more than 1 character. For example, the question operator is made up of 2 characters.

```
          '?-'
```

The set of predicate symbols is defined in procedure 'initsetofsymbols'. The set for predicate words is also defined there with the 'predicateop' set being the union of these.

The size of identifiers is limited to 9 characters. Thus

```
          artificial(intelligence)
```

will become

```
          artificia(intellige)
```

in the database. This will become apparent when such a fact is listed. Strings and words in quotes are also truncated

```
              "intelligence"  -->  "intellige"
and           'intelligence'  -->  'intellige'
```

The length limit can be altered by changing 'alfalength' in the constant section of the program. Care must also be taken to change the length of any constant strings.

```
e.g           spaces = '          '
```

must be altered to the new alfalength.

Integers are limited to values between 0 and 'maxint' which is 16383. Changes to 'maxint' must also be done to 'lastdigmax' and 'max10' all of which are in the constant section.

Looking at the EBNF there is quite a large amount of repetition. There are rules for

```
structure        and        argstructure
question                    argquestion
fact                        argfact
rule                        argrule
```

'arg' is short for argument. The reason for this is that the syntax for the arguments of clauses is slightly more general than the syntax for the clauses. An argument may be a variable while a clause may not be. For instance

```
                 hello(mary).

and              ?- hello(mary).
```

are correct

but

```
            X(mary).

    and     ?- X(mary).
```

are incorrect.

Arguments , however, can be of this form
e.g

```
            test((X(mary))).

    and     ?- test((?- X(mary))).
```

are correct.

Notice that such an argument structure must be in parentheses to deal with the scope of the symbols. For instance

```
        ?- test( ?-X,Y ).
```

will cause an error because it is unclear if the Y is the 2nd argument of test or the 2nd goal in the question

```
        ?- X,Y
```

Simple variables can be written with or without parentheses.

```
e.g     ?- test(X).

   or   ?- test((X)).
```

are both correct. Parentheses are not ignored when pattern matching takes place. So a fact

```
        test((a))
```

would cause

```
        ?- test(X)
```

to instantiate X to the value (a). In some PROLOGs, variables are allowed to be the functors of facts, rules and questions.

Unfortunately this means more checking at pattern matching time to deal with all possible matches. See chapter 5 for a more detailed discussion of this. In this PROLOG if a question has a variable functor then it can be satisfied by being executed in the 'call' predicate.

Thus

```
?- X(mary).
```

can be coded as

```
?- call( X(mary) ).
```

X must have a value so a question like the one above would still fail, but fail with a semantic error not a syntatical one.

Not included in the EBNF are the symbols which denote end of session and end of file. The end of session symbol is '$'. It causes a jump from 'nextsymbol' to the end of the interpreter. The end of file symbol '^' must be included at the end of every file. It causes the input stream to switch back to the terminal keyboard after a file has finished. This happens in procedure 'nextline'.

The procedures in the interpreter to deal with the EBNF are named using the EBNF names, or minor variations. Thus

line ---> procedure nextline

term ---> procedure term

and structure ---> procedure structure

For ease of programming some EBNF constructs have been merged. Thus

```
fact and rule  ---> procedure factorrule

argfact and argrule ---> procedure argfactorrule
```

3. Data Structures.

There are 3 main data structures in the interpreter

```
            the datatree    (see the type called 'dtree')
            the database    ('database')
    and     the variable list  ('varpter').
```

The 2 minor structures are

```
            the printed variable name list  ('namepter')

    and     the list of predicates to be looked
            at during execution              ('spypter').
```

The first 4 will be discussed here, the last in chapter 7 on diagnostics.

3.1 The datatree.

This is the structure that contains the parse of the input line of PROLOG which can be a question, rule or fact. As expected it is a tree structure. A tree can contain 3 different sorts of data in a node. It can contain information on variables, constants or identifiers. Identifiers are the largest class including atoms, built in predicate words (like 'asserta' and 'listing') and also symbols such as '=','?-' and '=..'.

A variable node consists of a variable name and a pointer ('varval') to the variable list where its value is.

A constant node contains an integer value.

An identifier node consists of its name ('idname'), and the number of arguments associated with it ('noofargs').

e.g	hello(10)	has 1 argument
	bye	has 0
	hello(man(X), but(t))	has 2 arguments

The node also has a number ('numofmatch') indicating when it was matched against another tree. This is only used when the node is in a question tree and the node is matched against a fact or rule tree in the database. There is also a boolean called 'cutflag' which is used by the predicate '!' ('cut'). This flag can stop another match being tried. 'Dbruleused' is a pointer to the database fact or rule tree which this node of the question tree is matched against.

3.2 The database.

This is a doubly linked list which contains the fact or rule trees. Each database node also contains the name of the tree that is hanging from it. This speeds up the search for a tree. Trees are put into the database in alphabetical order.

3.3 The variable list.

This is another doubly linked list which contains values for variables or possibly pointers to other variables. A value may not only be a number or identifier but can be an entire tree. Infact, a value is stored in a 'dtree' that is pointed to by 'stval'. The variable's name is also stored and also a flag ('owncopy') to indicate if 'stval' is pointing to its own value or another variable's value. This flag is used at garbage collection time. 'Uninstval' contains an integer. If a variable is to be printed that hasn't a value it will print an unique number instead.

e.g

```
        ?- test(X).

may produce     X = _21
```

This number is what is stored in 'uninstval'. Finally there are 2 other integer variables, 'creatnum' and 'mtnum'. 'Creatnum' gets a value when a variable in a tree has a variable location allocated to it. 'Mtnum' gets a value when this variable location gets a value or gets a pointer to another variable. They are related by the fact that

$$\text{creatnum} \leqslant \text{mtnum}$$

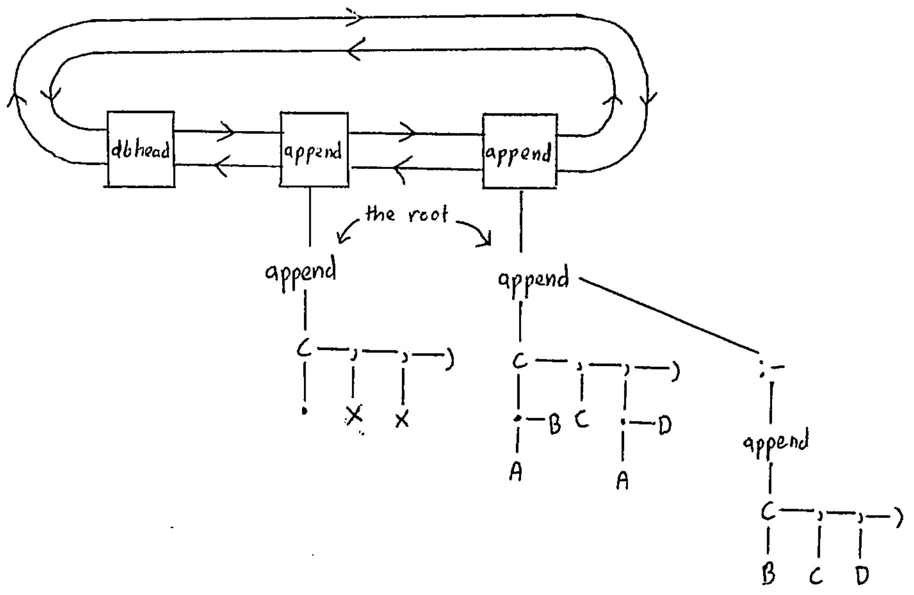
and are used for garbage collection. A global variable 'matchnum' gives them their integer values. 'Matchnum' increases and decreases with the size of the current question tree (see chapter 4). When 'matchnum' becomes less than or equal to a 'mtnum' in a variable location then its 'stval' can be deleted. When 'matchnum' becomes less than 'creatnum' for that location, the location can be deleted and its space in the variable list freed. The example in chapter 4 will make this clear.

3.4 Examples of the main structures.

The data structures created after two rules and 1 question have been typed will be shown. The question tree and variable list will be shown just before question satisfaction begins. Assume that

```
                append([],X,X).
                append([A|B],C,[A|D]) :- append(B,C,D).
and
                ?- append([a],[b,c],Q).
```

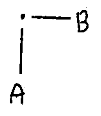
are typed. The first 2 rules are put into the database which will now have 2 trees hanging from it.



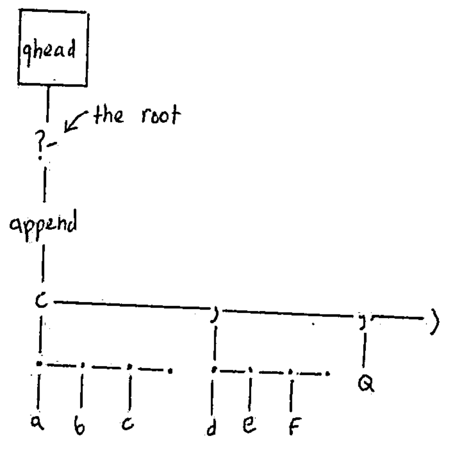
The words and symbols represent the things stored in the 'dtree' records. Not all the values in a node are shown. For instance, each of the 'append' nodes will have 3 as their 'noofarg' values.

If 2 rules have the same functor, their order in the database is the order in which they were typed.

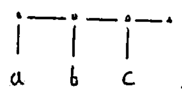
Notice how lists are stored. The empty list [] becomes a '.' node. A list in head and tail notation [A|B] becomes



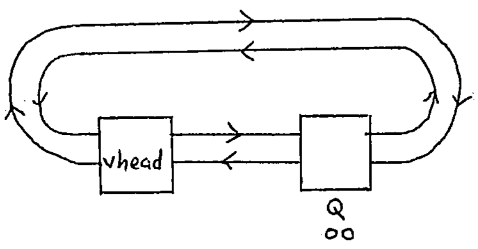
The question when typed will produce a question tree.



Notice how the list [a,b,c] is stored as



The only variable, Q is stored in the variable list.



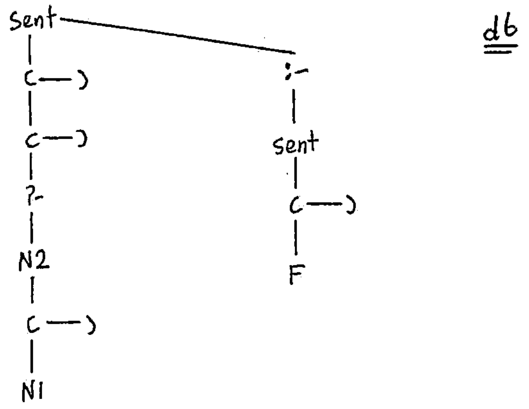
The notation 00 indicates that the 'creatnum' = 0 and the 'mtnum' = 0. Not shown is a pointer from the Q in the question tree to the Q in the variable list.

In future the 'qhead', 'vhead' and 'dbhead' will not be included in a drawing. Where necessary a 'db' or 'q' will be written next to the structure.

e.g

sent((?- N2(N1))) :- sent(F).

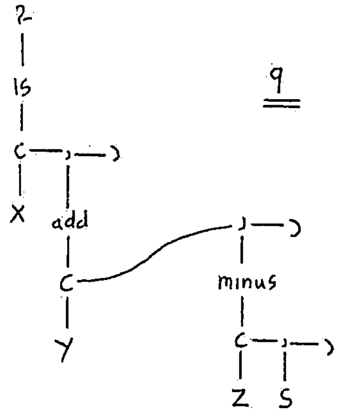
becomes



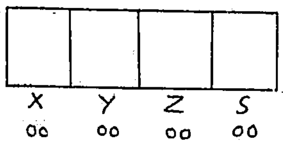
e.g

?- is(X, add(Y, minus(Z,S))).

becomes



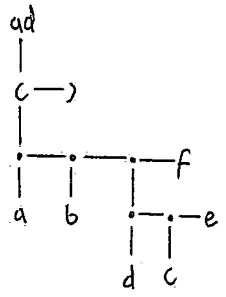
<u>q</u>



e.g

ad([a,b,[d,c|e] |f]).

becomes

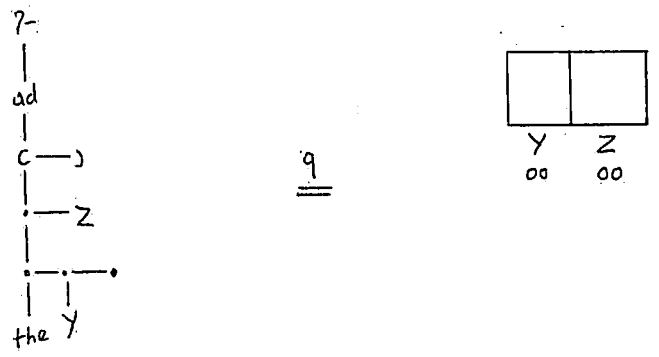


<u>db</u>

e.g

?- ad([[the|Y]|Z]).

becomes



There is a certain amount of redundancy in the parse trees but this means the program code to generate the tree is quite simple.

Procedures which create the database are 'initdb', 'reinitdb', 'finddbplace', 'insertdb', 'addnode' and 'findfreenode'. Procedures which create the question tree are 'initquestion', 'addnode' and 'findfreenode'. The procedures which create the variable list are 'initquestion' and 'findvarplace'. Procedures which delete the question tree and occasionally the data base trees are 'removetree', 'deletedb' and 'deletequestion'.

Related to the 3 structures of the previous sections are 3 procedures 'ptree', 'pbase' and 'pvarpter' which will print out the values of these data structures. They are in the interpreter as diagnostic procedures and are currently not used. During implementation they were used to see if values were being assigned correctly. On the DEC20, pasddt [5] can also be used.

3.5 The printed variable name list.

This list is used to store the variables which have had their values printed at the end of a question. Thus

```
?- test(X,X,Y).
```

will print the X and Y values only once. Without the name list the X value would be printed twice. This structure is used in the group of procedures just before procedure 'printsuccess' which prints out values if a question has succeeded.

4. Backtracking.

The reader should be familiar with the idea of backtracking in PROLOG. Section 2.6 of C&M runs through a small example of this. The aim of this chapter is to show how backtracking is achieved in this interpreter. One example will be given, using the notation introduced in the previous chapter. Then the procedures within the interpreter which do these things will be named. For the moment only a brief outline of pattern matching will be given. A more complete discussion will be appear in chapter 5. In chapter 10, backtracking and the more general subject of PROLOG's search strategies are returned to. Algorithms are presented which build on the ideas of this chapter and offer more flexible solutions than those given here.

4.1 An Example.

The following is typed

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).
```

and

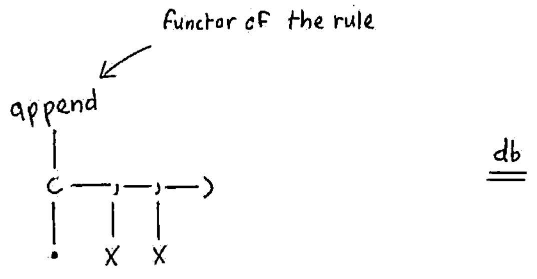
```
?- append(X,Y,[a,b,c]).
```

If this goal is resatisfied until it fails,

it will have given 4 answers.

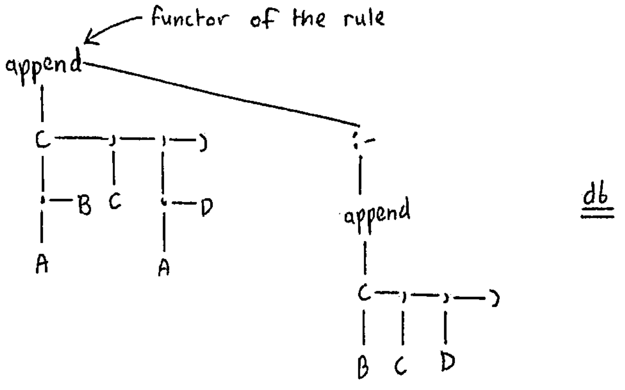
```
X= []        Y= [a,b,c]
X= [a]       Y= [b,c]
X= [a,b]     Y= [c]
X= [a,b,c]   Y= []
```

In the interpreter the two rules will be stored as



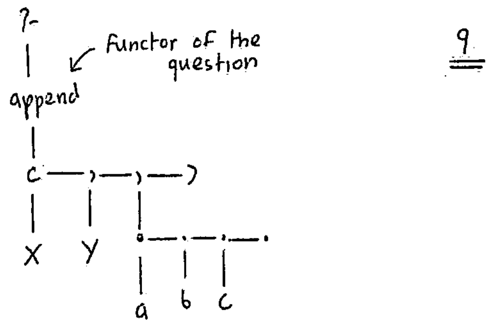
Rule 1.

and

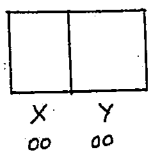


Rule 2.

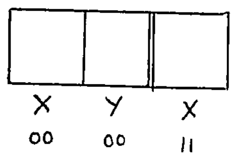
The question will be stored as



The var list will be



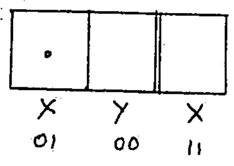
When execution begins the question will try to match against rule 1. Matching starts from the functor of the rule and the functor of the question. The global variable 'matchnum', which has an initial value of 0, is incremented to 1. The variables of the rule are added to the var list. Thus the var list becomes



The 2nd X being from rule 1 and having nothing to do with the 1st X from the question. Since there will now be some confusion over which variable is being referred to a variable may be subscripted with its 'creatnum' and 'mtnum' e.g X(00) and X(11). During pattern matching

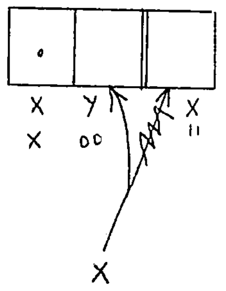
in the question		in the rule
X	matches	'.'

Making the var list

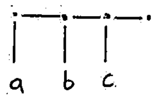


Y	matches	X

Making the var list

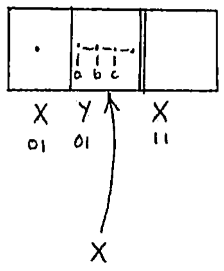


x pointer moved to Y rule

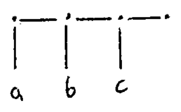


matches X

Making the var list



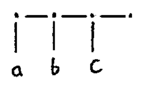
When X (in the question) matches with the '.' (in the rule), X(00) gets a copy of the list. When Y matches with X (in the rule), X's pointer is moved from the X(11) in the var list to the Y in the var list. When the



matches with X (in the rule) which is X(11) it is actually matching with Y. So Y gets a copy of the list. The functor append in the question now has its 'numofmatch' set to 1. Since there is no right hand branch leaving this functor or leaving the matched rule, the question is finished. Thus the variables from the question are printed.

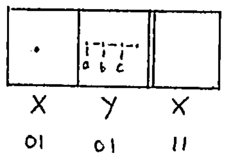
X = X(01) = .   = []

Y =

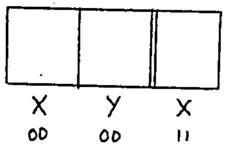


= [a,b,c]

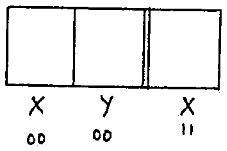
The question could now be deleted (automatically) and another question typed in and that satisfied. Instead the user types a '; <cr>' and the question backtracks for another answer. What now happens is that the question tree is searched from the root until it gets to a functor node whose 'numofmatch' equals the 'matchnum'. In this case it is the very first functor node. Its 'numofmatch' is set to 0 and 'matchnum' is decremented by 1, back to 0. The right hand side of the functor is removed which in this case is empty already. Then the var list is cleaned up. All var list elements which have a 'mtnum' greater than 'matchnum' have their values removed. Then their 'mtnum's are set to the value of their 'creatnum's. Thus with 'matchnum' equal to 0



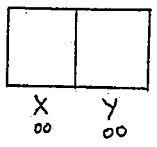
becomes



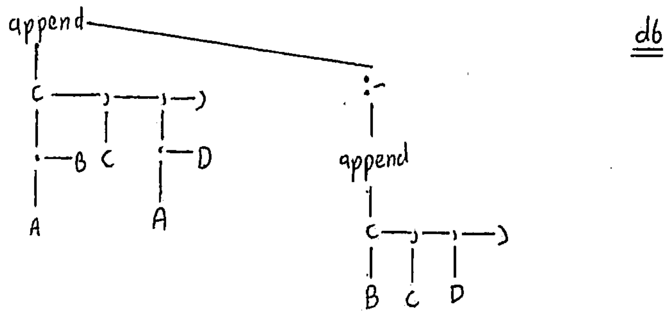
Then those var elements whose 'mtnum's are still greater than 'matchnum' are deleted from the list. Thus with 'matchnum' = 0



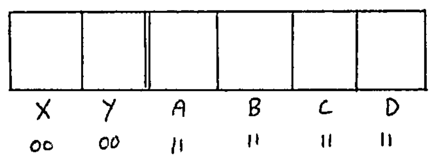
becomes



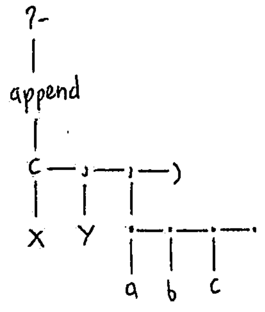
So the question tree and its var list are back to the state they were in initially. There is 1 difference. When the first match took place the functor 'append' in the question tree set its pointer, 'dbruleused', to point at the first 'append' rule in the database. Now when the question is resatisfied another rule is searched for in the datatbase starting from just after this rule. In other words, the first 'append' rule will not be matched with again. Now the 2nd 'append' rule is matched against. This rule is



'Matchnum' is incremented back to 1. A,B,C and D are added to the var list.



This rule is matched against the question tree

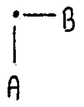


9

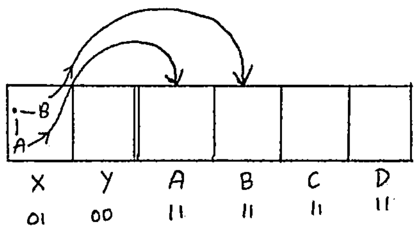
So

in the question   in the rule

X   matches



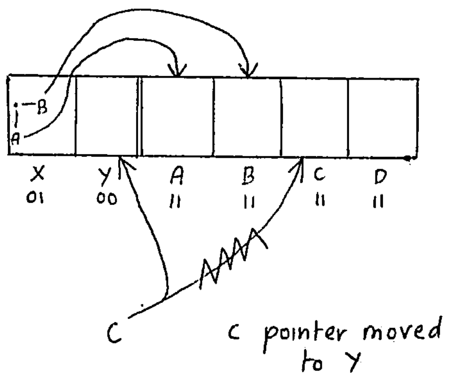
The var list becomes

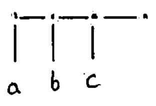


Then

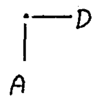
Y matches C

So

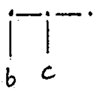




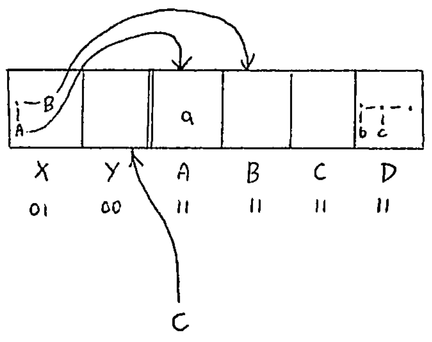
matches



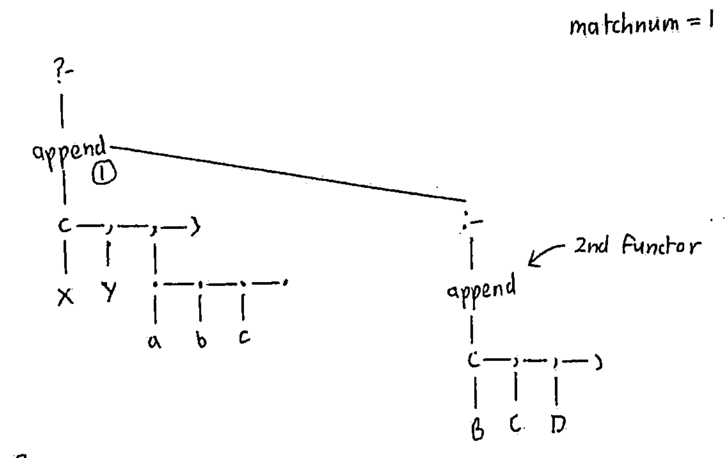
and so

a	matches	A
	matches	D

Giving

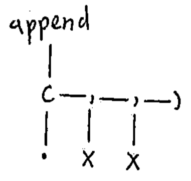


Now, because everything matched, a copy of the right hand side of the database rule is added to the right hand side of the question tree, to give



9

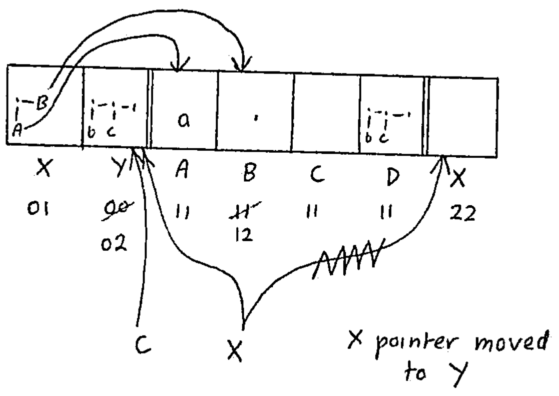
The B,C and D of this copy still point at their values in the var list. The first functor, 'append', gets a 'numofmatch' = 1 and also its 'dbruleused' is set to point to the 2nd 'append' rule and goal satisfaction continues. 'Matchnum' is incremented to 2. Now the new right hand side of the question tree is satisfied. The next functor, also called 'append', is now matched. It is just starting its matching so it will start from the beginning af the database and so match against the first 'append' rule. It will match with db



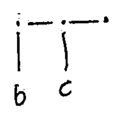
The question's 'dbruleused' is set to point at this db rule. Now

B	matches	'.'
C	matches	X
D	matches	X

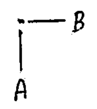
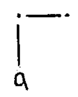
Giving

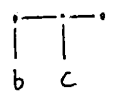


When B matched with '.', B got a copy of the '.' and B's mtnum is set to 2. When C matched with X(22), the X pointer was moved to point at C which meant pointing at Y. When D matched with X(22), X(22) got a copy of D's value, which is

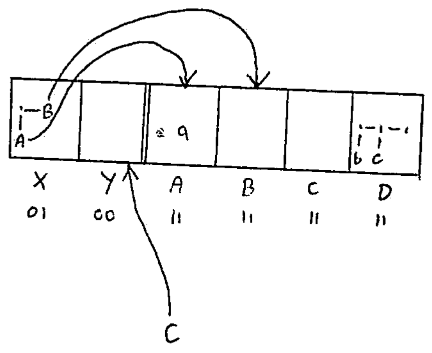


But since X(22) is actually Y then Y got the value with its 'mtnum' being set to 2. Since there is no right hand side of the 1st 'append' rule to copy onto the question tree, the question finishes. The values of X(01) and Y are now printed.

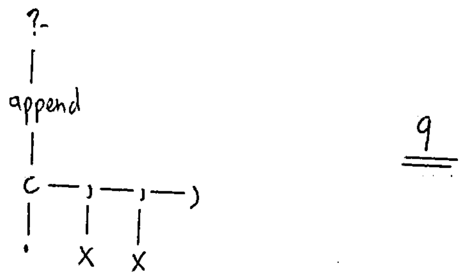
X=  =  = [a]

Y=  = [b,c]

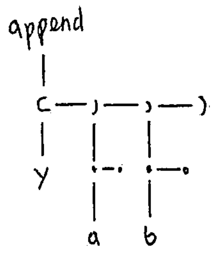
The user then types '; <cr>' so the question will be resatisfied. The question tree is searched from the root again until the functor whose 'numofmatch' equals 'matchnum' (which is 2) is reached. The 'numofmatch' is set to 0, 'matchnum' decremented by 1, the right hand side removed (if possible) and then the var list is reset. Giving



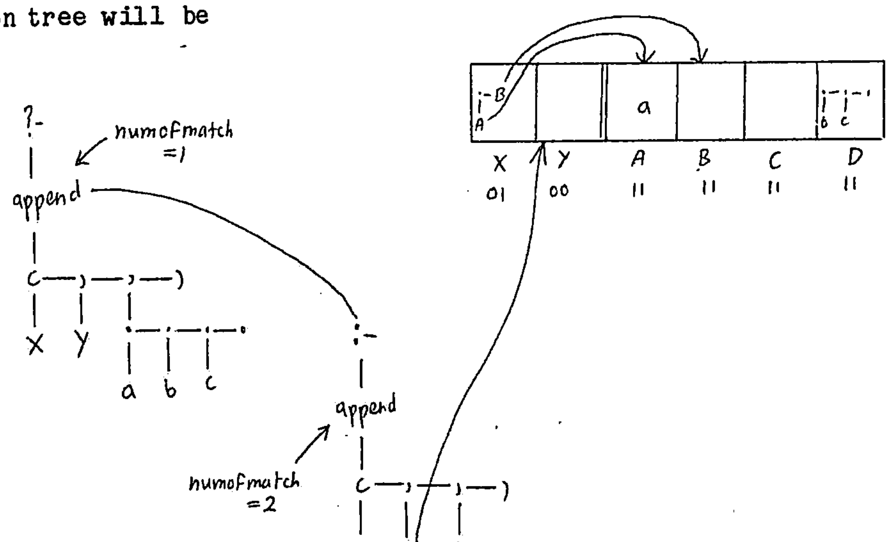
Notice how X and D still have their values but Y and B do not. This was because their 'mtnum's were greater than 'matchnum' (which is 1). The 2nd functor of the question tree can now search for another rule in the database. While doing so 'matchnum' is incremented to 2. The 2nd rule will be found and so carry on to get another answer. Instead let us assume that a 2nd rule is not found. In this case the 'match' flag is set to false. 'Matchnum' is decremented back to 1. Any new variables and values created before the failure are deleted. This is because a failure to match may not occur immediately.



will almost match

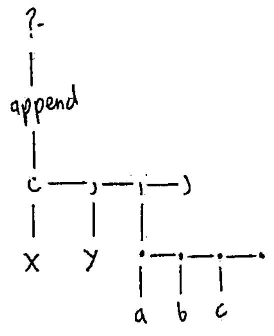


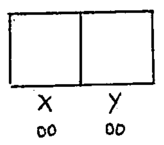
X will get a value and Y will get a space and a value on the var list before failure occurs. When failure occurs the database is searched again from the rule that failed to the end of the database or upto another matching rule. If no rule is found then the question tree will be



with matchnum = 1

At this point the 'match' flag is returned as false. The top level gets this flag back. What it then does is search the tree from the root again but looking for the functor whose 'numofmatch' equals 1 (the current value of 'matchnum'). It will find the first functor of the question tree. 'Numofmatch' will be set to 0, 'matchnum' decremented to 0, the right hand side of the functor deleted, the var list cleaned up and then the goal will try to be resatisfied. At this point the question tree and var list will be





with matchnum = 0

Unfortunately, the 'dbruleused' of this functor is already pointing at the last 'append' rule. The search for a 3rd rule will fail. With this failure and 'matchnum' equal to 0, this question tree can not be satisfied in any way. Thus the interpreter will return an answer of 'no', delete the question tree and the var list and prompt for another line of PROLOG.

4.2 Location of code.

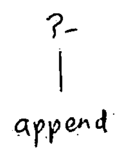
Inside the interpreter the execution of a tree begins with a call to procedure 'answer'. The question tree and var list have already been built using procedures 'initquestion', 'findfreenode', 'addnode' and 'findvarplace'. 'Answer' calls 'dogoal'. If 'dogoal' succeeds then the variables are printed out and then the question may be reanswered in procedure 'reanswer'. Otherwise failure is printed. Procedure 'reanswer' is like procedure 'answer' but has a call to 'redogoal'. Also it loops if another try is required. 'Dogoal' searchs the tree for the first time. It does this by calling 'satisfy'. 'Redogoal' does the same thing but because it is resatisfying and thus searching for a 'numofmatch' that is already in a functor node, 'satisfy' is called with 'nodefound' set to false. The 'numofmatch' being looked for is equal to 'matchnum'. If 'satisfy' fails, 'matchnum' will have been decremented and so 'redogoal' loops and calls 'satisfy' again. Only when 'matchnum' is 0 does 'redogoal' stop looping. At this point failure is final.

So it is clear that backtracking is handled by procedure 'satisfy'. When 'nodefound' is true 'satisfy' is searching a question tree for the first time. When 'nodefound' is false it is searching for the last match to a functor so the match can be undone and another one attempted.

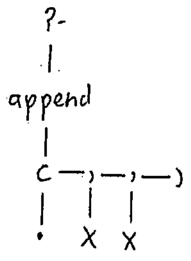
When 'nodefound' is false, 'satisfy' keeps calling itself to get down to the desired branch of the tree. When it reaches the right node, it sets 'nodefound' to true, then does all the cleaning up of the question tree and var list and then calls itself again. Now 'satisfy' will execute the 2nd half of its body where pattern matching takes place. It will either call procedures 'ideval' or 'predeval' depending on whether the functor is an user-defined one or a predicate name.

Other parts of the code of 'satisfy' are to deal with various special cases. If the functor is a symbol such as a comma, ruleop, or questionop it must be ignored. A left parenthesis or nil pointer stops the search down a tree since a preorder search is being used and so in the example below the left tree of 'append' will be looked at before 'append' itself. The test for '(' or nil stops this search.

e.g



or



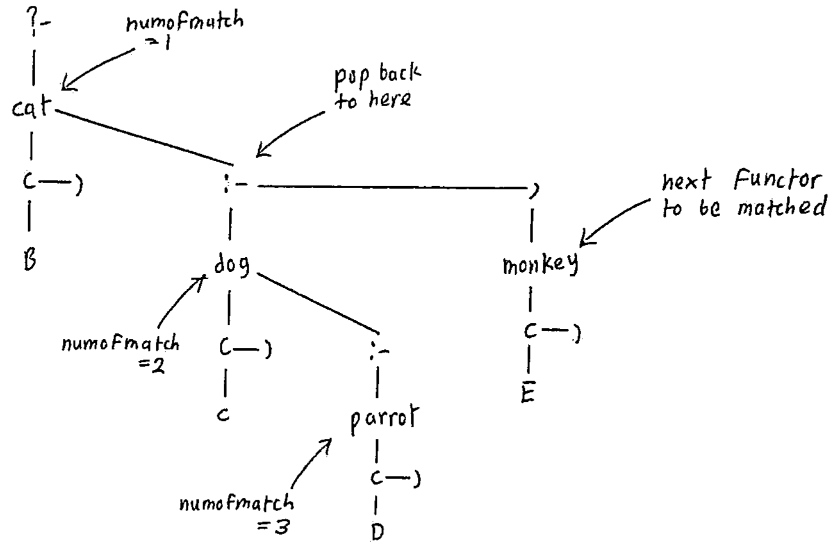
There is code in 'satisfy' so that diagnostics predicates can print out. These code segments are preceded by a test of the flag 'debugon'. There is also a test for the 'cutflag'. It stops pattern matching occuring.

Procedure 'ideval' finds a rule in the database to match with the question. It does this by calling 'findclause'. It then calls procedure 'setdbvar' which adds the variables in the database rule to the var list.

Pattern matching is done through procedure 'ignoreruleop' (see chapter 5). If this is successful and 'match' is true then the right hand side of the rule is added to the question tree. If 'match' is false then 'matchnum' is decremented and the var list is cleaned up. 'Findclause' is called again until success or the database is exhausted.

Procedure 'predeval' is called when a built in predicate is being executed (see chapter 6).

To summerise, backtracking is done by searching the question tree from its root. When the last matched functor node is found then the node will try to be matched against a new rule. During the execution of a question, the question tree will grow and shrink depending on what rules it matches against and what backtracking takes place. This searching from the root is necessary to build up a stack of calls of 'satisfy' so that when a question node matches, the stack can be popped to get the parent node of the current question. Consider



After the 'parrot' subgoal matches, the hardware stack will pop off three 'satisfy' calls to get back to the ':-' node at which point 'satisfy' will call itself twice to get to the 'monkey' node. In most PROLOG implementations this pushing and popping of stack environments is coded explicitely [7].

5. Pattern Matching.

The pattern matching developed in this chapter is more powerful than that found in many PROLOGs. In particular it is possible to have arguments which have variable functors.

Consider

```
?- num(X),call( (X(2)) ).
```

with the clauses

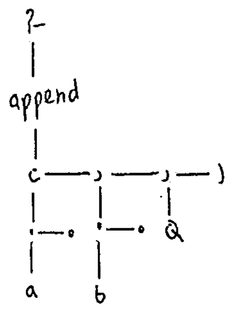
```
num(val).
val(2).
```

The argument of 'call' is X(2). X is a variable functor which, during the execution of the question, is instantiated to the constant 'val'. 'Call', which is a built in predicate, then executes this goal which in its simple form is

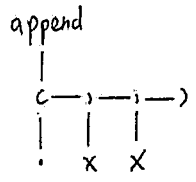
```
val(2)
```

A 'yes' answer is returned since 'val(2)' is in the database.

When backtracking and searching occurs, pattern matching begins by comparing the functors of the question tree and database rules and then moves on to comparing the arguments of each. Thus, for the following,

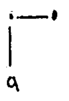


q

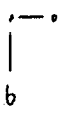


db

pattern matching starts by comparing the functor, 'append', of the question tree with that of the database rule. When these match, pattern matching is done on the arguments of each. So,



matches •



matches X

Q matches X

Arguments can be considerably more complicated than this. Most of the complication arises when 2 arguments are matched which are made up of variables.

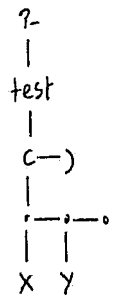
An argument may be an identifier, integer, string, a simple variable or an argument structure. An argument structure must be in parentheses and can be a question, fact or rule which can be made up of variables.

```
e.g                 ?- test( (?- Q,b(C)) ).
                    ?- depth( (X(S) :- b(c)) ).
or                  beg( (X(s,b,c)) ).
```

An argument may also be a list which is of the form

```
                    ?-test([X,Y]).
```

which is stored as



After functors have matched, both question tree and database tree must be descended and the nodes of each compared with one another. When pattern matching gets to a node with a variable then more complicated things must be done. The variable may be instantiated to many different things or uninstantiated. This is also true of the corresponding node in the other tree. Only some of these combinations are allowed and different allowable combinations cause different things to be done.

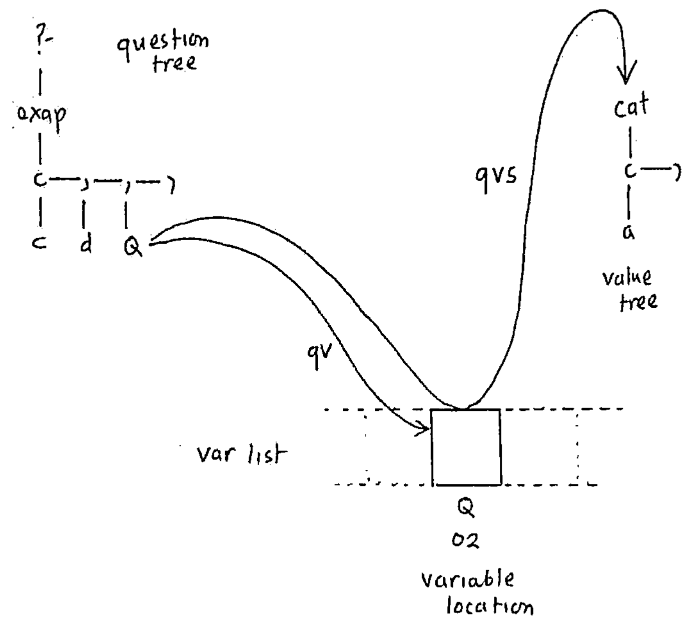
The way these allowable combinations will be discussed is to consider a typical node in the question tree (q) and a typical node in the database tree (db) and then let these nodes be variable,identifier or integer, with 0,1 or 2 branches leaving them. After exhausting all combinations of these possibilities and specifying the actions taken when they occur , it is then easy to show how code has been written for them. The following notation will be used.

var	= variable
uninst	= uninstantiated variable When a variable is uninstantiated it means that it has a variable location in the var list but no value in it
inst	= instantiated variable When a variable is instantiated it means that it has a value in its variable location
ident	= identifier
int	= integer

The following pointers will be abbreviated as

qv	= q's pointer to its variable location
dbv	= db's pointer to its variable location
qvs	= q's pointer to its value where q is a variable
dbvs	= db's pointer to its value where db is a variable

These distinctions are clearer from the diagram

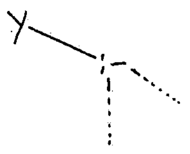


The following procedures will be used

changevars(X,Y)
    Move all pointers, X, so they are pointing
    to the same place that Y is.

evaluate(X,Y,match)
    Carry out pattern matching on the structures
    represented by X and Y. Return success
    or failure in the boolean called 'match'.

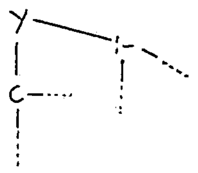
evalleftdb(X,Y,match)
    The structure Y is a node with a right child.
    e.g



'Evalleftdb' carries out pattern matching on the structures represented by X and by Y (with its right child ignored).
'Match' returns success or failure.

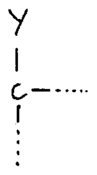
copy(X,Y,nil)
Copy the structure with Y as its root to the location pointed at by X. The nil is for what the copied Y tree should have as its new parent node - nothing.

copy1(X,Y,nil)
Copy only the node Y to the location pointed at by X. Y may have children
e.g



but these are not to be copied over to X.

copyleftdb(X,Y,nil)
Copy only the node Y and its left children to the location pointed at by X. From the example above X would get the tree



fail
Return with 'match' set to false.

5.1 The different kinds of q and db nodes.

For a typical q node and a typical db node consider all the possible structures that they could have hanging from them.

Each q and db structure allowed together will be 1 case. Within each case, the different allowable types of q and db will produce subcases.

For each subcase there will be an action carried out which will affect the var list. These actions will be specified by the abbreviations given in the last section.

Case A.
Q node and db node with no child branches
i.e

q db

A1. Let q,db both = var

Subcase.		Action.
q uninst	db uninst	changevars(dbv,qv)
q inst	db uninst	dbvs := qvs
q uninst	db inst	qvs := dbvs
q inst	db inst	evaluate(qvs,dbvs,match)

A2. Let q,db be either var or ident or both

Subcase.		Action.
q var	db var	--see case A1--
q ident	db var-uninst	copy(dbvs,q,nil)
	-inst	evaluate(q,dbvs,match)
q var-uninst	db ident	copy(qvs,db,nil)

-inst		evaluate(qvs,db,match)
q ident	db ident	does q = db ?

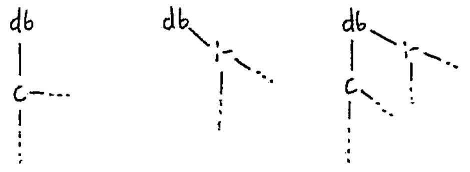
A3. Let q,db be either var or int

Subcase.		Action.
q int	db int	does q = db ?
q int	db var-uninst	copy(dbvs,q,nil)
	-inst	evaluate(q,dbvs,match)
q var-uninst	db int	copy(qvs,db,nil)
-inst		evaluate(qvs,db,match)

Case B.

One of the two nodes q and db do not have children.
Let q be the one without children.
All subcases given will have a symmetrical subcase where db is the childless node.
Only 3 structures are allowed with db as the parent node.
i.e

q and 1 of



B1. Let q,db both = var.

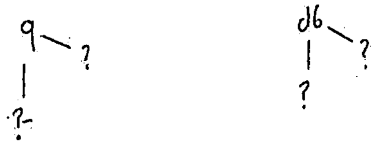
Subcase.		Action.
q uninst	db uninst	copy(qvs,db,nil)
q uninst	db inst	copy(qvs,db,nil)
q inst	db uninst	evaluate(qvs,db,match)
q inst	db inst	evaluate(qvs,db,match)

B2. Let q,db = var,ident or both

Subcase.		Action.
q var	db var	--see case B1--
q ident	db var-uninst	fail
	-inst	fail
q var-uninst	db ident	copy(qvs,db,nil)
-inst		evaluate(qvs,db,match)
q ident	db ident	fail

B3. Let q,db = var,int or both int
All subcases fail.

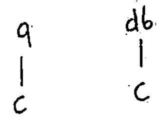
Case C.
Both q and db have 1 or 2 children.
i.e



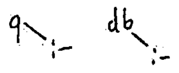
'?' can be '(', ':-' or nothing. '(' must be on the left, ':-' on the right.

There are only 5 cases (out of the possible 9) that can succeed. They are

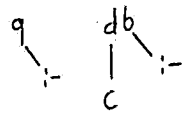
C1



C2

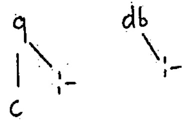


C3

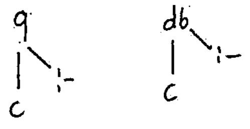


and

C4



C5

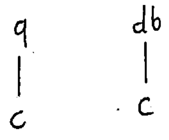


It is clear that case C4 is the same as case C3 but with q and db switched. We shall only consider case C3.

Also case C1, C2 and C5 are similar in that for them to be considered, the children of q and the children of db must be the same. We shall assume such tests will be done in the code and so only consider case C1 here.

For the cases C1 and C3 we must now consider the different types that q and db can be.

Case C1.



C1.1 Let q and db = var

Subcase.		Action.
q uninst	db uninst	changevars(dbv,qv)
q inst	db uninst	dbvs := qvs
q uninst	db inst	qvs := dbvs
q inst	db inst	evaluate(qvs,dbvs,match)

C1.2 Let q and db = var,ident or both

Subcase.		Action.
q var	db var	--see case C1.1--

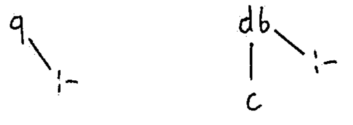
q ident	db var-uninst
	-inst
		copy1(dbvs,q,nil)
		evaluate(q,dbvs,match)

q var-uninst
-inst
	db ident	copy1(qvs,db,nil)
		evaluate(qvs,db,match)

q ident	db ident	does q = db ?

C1.3 Let q and db = var,int or both int
All subcases fail.

Case C3.



C3.1 Let q,db = var

Subcase.		Action.
q uninst	db uninst	copyleftdb(qvs,db,nil)
q uninst	db inst	copyleftdb(qvs,db,nil)
q inst	db uninst	evalleftdb(qvs,db,match)
q inst	db inst	evalleftdb(qvs,db,match)

C3.2 Let q,db = var,ident or both

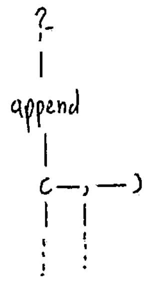
Subcase.		Action.
q var	db var	--see case C3.1--
q var-uninst	db ident	copyleftdb(qvs,db,nil)
-inst		evalleftdb(qvs,db,match)
q ident	db var	fail
q ident	db ident	fail

C3.3 Let q,db = var,int or both int
All subcases fail.

5.2 Location of Code.

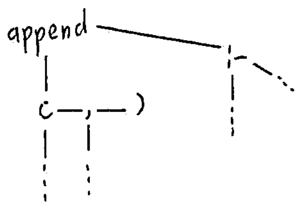
Having concluded an exhaustive analysis, the interpreter code for each case is easily written.

Pattern matching starts from procedure 'ideval' when procedure 'ignoreruleop' is called. 'Ignoreruleop' removes the right hand side of the database rule so that pattern matching will not fail when, for instance



$\underline{\underline{q}}$

is matched with



$\underline{\underline{db}}$

Procedure 'evaluate' separates the cases as have been done here. Some extra tests are done to deal with symbols which are to be ignored, such as '?-', '(', ')' and ','. The cases become the following procedures

procedure 'qdbend'	=	case A
procedure 'qend'	=	case B
procedure 'beval'	=	case C

Within procedure 'qdbend'

procedure 'qdbintend'	=	case A.3
procedure 'qdbidend'	=	case A.2
procedure 'qdbvarend'	=	case A.1

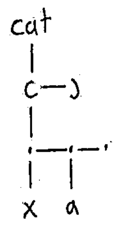
Procedure 'stvcopy' is used to do the job of copy(qvs,db,nil) and also copy(dbvs,q,nil).

In procedure 'beval',

```
    procedure 'bevalsame'    =  case C1,C2 and C5
    procedure 'bevalqruleop' =  case C3 and C4
```

Extra code is needed to deal with the periods inside a list.

e.g cat([x,a])   becomes



Inside procedure 'bevalsame',

```
        procedure 'qdbvarend' = case C1.1
```

Procedure 'stvcopyid' does the job of copy1(qvs,db,nil) and also copy1(dbvs,q,nil).

It should be clear by now how the cases relate to the code. To test for an uninstantiated variable, requires 2 'if' tests. The first to check to see if the node is a variable

```
        if q^.kind = vars then
```

The second to see if it is uninstantiated

```
        if q^.varval^.stval = nil then
```

From this, clearly

```
        qv = q^.varval
```

```
qvs = q^.varval^.stval
 etc
```

The only code which has not been explained is procedure 'changevars' and its functional opposite, procedure 'undochanges'.

'Changevars' moves variable pointers to a new location. Thus if node X has a variable pointer X^.varval (called 'dbvar' in 'changevars') then it will be moved to point at a new variable 'qvar'. It must do this for all nodes with an X variable pointer. That is why 'moveallvars' searches the database tree and the question tree and 'checkvarlist' does the same for the var list. This is a time consuming operation and is to deal with shared variables .

```
e.g          append([],X,X).

with         ?- append(Z,Y,[a,b]).
```

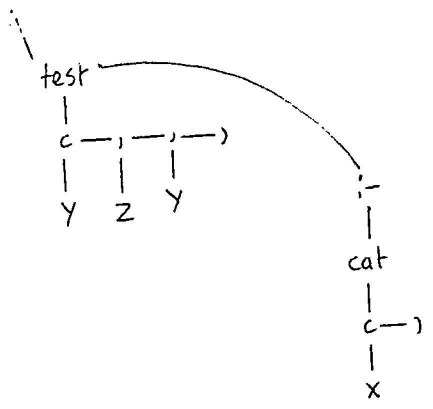
During execution, Y and X will both point to the same variable location which will be empty. Y and X are sharing. When X gets instantiated to [a,b] so will Y and the question will succeed, returning

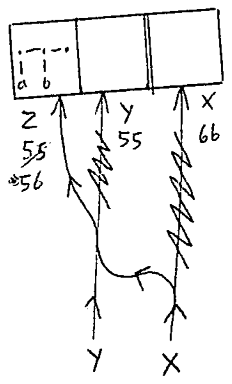
```
        Z = []
        Y = [a,b]
```

All Y variable pointers must point to the same location as all the X variable pointers. If Y is later shared with another variable Q then not only must all the Y pointers be moved to the Q variable location but so must all the X pointers. The time consuming part of this operation is finding all the X and Y pointers in the trees and also in the var list structures. In some PROLOGs, variables like these are set aside from others which do have values [8]. They are then easier to deal with.

Procedure 'undochanges' sets a node's variable pointer back to its own variable in the var list. Some assumptions can be made about undoing changes. Changes are only undone when a goal is being resatisfied. Procedure 'deletevarlist' will remove the pointers to variables created after the present goal. Shared variables which occur earlier than this goal should be left as they are since they do not want to be resatisfied. Only variables created during the match of this goal which are shared need to be looked for.

For example, assume that when matchnum equalled 6 the match between the question 'test(Y,Z,Y)' and the rule 'test(X,X,[a,b]):-cat(X)' took place, to give



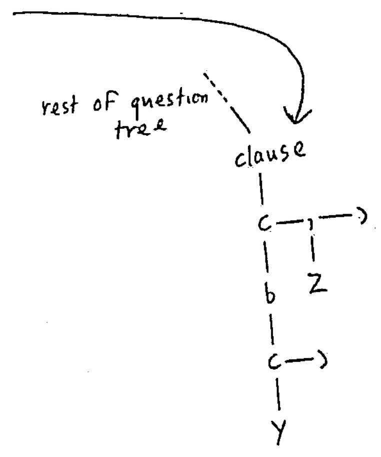


Thus the Y and X pointers are now aimed at the Z variable. When backtracking reaches the 'test' node, Y's variable pointer must be set back to point at its own variable.

6. Built in Predicates.

A complete list of the built in predicates in this interpreter are given in the fourth appendix.

The interpreter code for the predicates is called from procedure 'predeval'. Each procedure that implements a predicate is named by using the predicate name preceded by a 'x'. Thus 'display' is implemented in procedure 'xdisplay(q,match)'. The parameter q is the subtree of the question tree which contains the call to the predicate.Thus when the procedure 'xclause(q,match)' is called, q points to



'Match' is a boolean that returns true or false depending on if the predicate does its job.

A common notation has also been used for the arguments of a predicate.

The first argument is

X1 = q^.left^.left

The 2nd argument is

X2 = q^.left^.right^.left

and so on.

In the above example

$$X1 = \begin{tikzcd} b \arrow[d, dash] \\ c \arrow[r] \arrow[d, dash] & {} \\ Y \end{tikzcd}$$

and X2 = Z

To go from the graphs to pointer values is simple

↓ down = left pointer

→ across = right pointer

Chapter 6 of C&M gives a full discussion of what the built in predicates are meant to do. Of the ones written about in that chapter, this interpreter does not have

reconsult	op
see	seeing
seen	tell

telling told

'Reconsult' is not implemented since it was not considered to be useful. It also did not seem necessary to have the 6 file predicates. The reason for this is that the input and output of the interpreter is a lot simpler than the one discussed in chapter 5 of C&M. The output is only meant to be sent to the screen. Input is usually from the terminal keyboard. While it is possible to read files using 'consult', once the end of file symbol is read ('^'), the input file is once again set to the terminal.

The 'op' predicate is used to set the precedence, position and associativity of an operator. Since all the operators are in prefix form this predicate was not needed.

In the following chapter only the predicates which have complicated code or differ in their operation from those in C&M will be described.

6.1 Call.

In most PROLOG interpreters, when the predicate 'call' is used, it is usually of the form

```
call(s(T))      or      call(X)
```

where X is a variable instantiated to a question like

```
p(Y)
```

where the functor 'p' of the question is a constant.

In this interpreter, the argument of 'call' can be of a more general form. So in

```
call(X)
```

X is a variable which can be instantiated to a question like

```
Z(Y)
```

where the functor 'Z' of the question is an instantiated variable. More simply,

```
call( (Z(Y)) )
```

can be written. Notice in this case, it is necessary to put parentheses around Z(Y) because of the syntax rules.

In other interpreters it is possible to simulate this generality in 'call' by using 'functor' or '=..' ('univ'). In this interpreter, a user can write

```
val(2).
num(val).
?- num(X), call( (X(Y)) ).
```

to obtain

```
X = val
Y = 2
```

In a more simple interpreter, a user must write

```
val(2).

num(val).

?- num(X), functor(F,X,1), call(F).
```

to obtain

```
X = val
F = val(2)
```

In both cases, 'call' has executed the goal 'val' to find a value of 2. An alternative approach is to write

```
?- num(X), =..(F,[X,Y]), call(F).
```

to get

```
X = val
Y = 2
F = val(2)
```

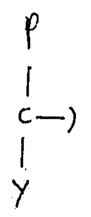
In other words, in this interpreter it is posssible to execute a question which has an uninstantiated variable as its functor. To deal with this complication, procedure 'xcall' calls a procedure 'getcallarg' which makes a simplified copy of the 'call' argument and puts it into the variable 'newq'. 'Newq' will be the argument with the functor variable replaced by its value . Thus in the case of

```
call( (Z(Y)) )
```

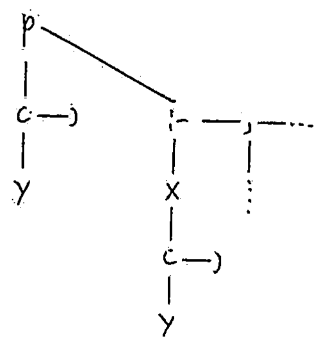
where Z is instantiated to 'p' then 'newq' will contain

```
p(Y)
```

The argument is then satisfied as if it were a question using procedure 'satisfy'. As described earlier 'satisfy' will add subtrees to the original question tree. Continuing the example, at the end of 'satisfy' the question tree 'p(Y)' which originally looked like



may become



The key feature of this new tree is that the 'p' node now has a right hand subtree. So in 'xcall' after 'satisfy' has finished, procedure 'savechanges' is called. 'Savechanges' adds this new right hand subtree to the original 'call' argument. What this subtree is actually appended to is the functor constant in the 'call' argument. This must first be found because as described above it may be in a variable. When the functor is found it may already have a right hand subtree. This is possible if the 'call' argument has been satisfied before and this time 'xcall' is being executed because the argument is being resatisfied. If the functor already has a right hand subtree, it is deleted and the new right hand subtree is appended.

e.g

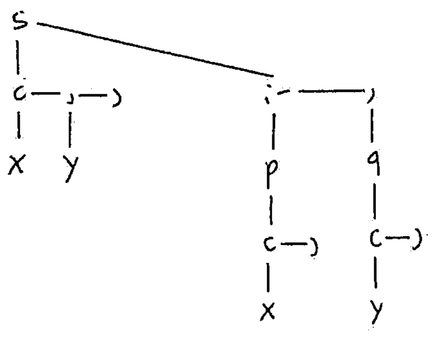
call(P)   is being satisfied

where P is instantiated to Z(X,Y)

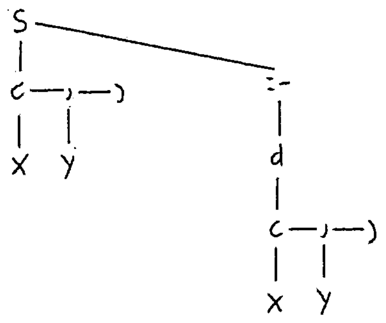
and   Z is instantiated to s

'Newq' will obtain s(X,Y) in 'getcallarg'

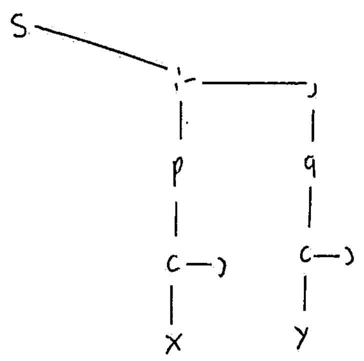
Since s(X,Y) has been satisfied before 'newq' will actually be a tree like



After 'satisfy' has worked on 'newq' let us assume it becomes the tree

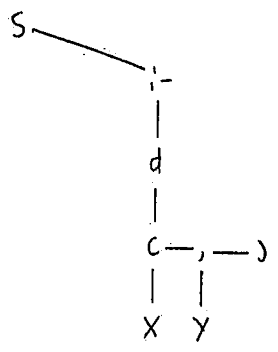


In 'savechanges' the constant functor of 'P' is obtained. Thus 's' is returned. Actually, the tree 's' contains



which is 's' with its old right hand subtree. This subtree is deleted, leaving the tree

The new right hand subtree is appended giving



This is now what is stored in 'Z' when 'savechanges' and then 'xcall' finish.

6.2 Not & ';'(disjunction).

Both these predicates have been implemented using PROLOG rather than PASCAL. When they are called an error is printed telling the user to consult the file 'pred.in'. This file contains their definitions, which are

```
nott(X) :- call(X), ! ,fail.
nott(X).
```

and

```
or(A,B) :- call(A).
or(A,B) :- call(B).
```

Notice that names other than 'not' and ';' are used since these are user defined predicates. Other built in predicates could be implemented this way but their PASCAL versions are as simple. For example, 'nonvar' is defined in PROLOG as

```
nonvar(X) :- var(X), ! ,fail.
nonvar(X).
```

Its PASCAL version is in procedure 'xnonvar' and is also only 2 lines long.

See chapter 10 for further discussion of PROLOG defined predicates.

6.3 Clause.

A variable functor can be assigned to the first argument of 'clause'. To simplify this argument, the procedure 'simplifyhead' is called. Thus in

```
            clause( (Z(X)), Y)
            where Z is instantiated to 'p'
            then X1 = (Z(X))
 After a call to 'simplifyhead'
                  newx1 = p(X)
```

It should be clear that X1 contains the first argument of 'clause' which is the head of the rule (or fact) and X2 will contain its right hand side or possibly the value 'true'. The restriction is that X2 will only return the first subgoal of any right hand side. Thus if there is a rule

```
            t(X) :- b(X), c(X).
 then
            ?- clause( t(X), W).
will give
            W = b(X)
```

6.4 Consult.

The code for this predicate is in procedure 'xconsult'. In actuality most of the work is done in 'xsee'. This sets the input file to the filename argument of 'consult'. A limitation of 'consult' is that a filename, being an identifier, should only be 9 characters long. Also since DEC20 filenames must have a file ending with '.<something>' then the filename must be in single quotes or the '.' will be taken as a period in PROLOG syntax and an error will occur. Thus the user must type something like

```
?- consult('pred.in').
```

There is a way to get around these restrictions. If the user types a file name unknown to the system, the user will be queried for another filename. The query comes from the operating system and so the single quotes and 9 character restrictions do not apply. Thus

```
    ?- consult('vbbk.pq').
? File not found - vbbk.pq
Try another file spec: very_long_file.in
```

If the file is found PROLOG is reentered and continues as normal.

All files must have the end of file symbol '^' at the end of them. If this is not found an end of file error will occur and the interpreter will finish. The '^' is used in 'nextline' to reset the input file.

This 'consult' predicate is more general than the one described in C&M since questions can be included in the files. When these questions are executed, input for them, from the user, will still be taken from the keyboard and not be looked for in the file being consulted. Another useful feature is that when a file is consulted, it will also be printed on the terminal screen. This means that if there are any errors in the file, the user will see where they occur.

The form

```
?- [ file1, -file2, 'fred.1' ].
```

is not allowed, althrough excluding the files to be reconsulted of the form -file, a similar notation can be achieved.

e.g

```
?- [ file1, 'fred.1' ].
```

could be written as

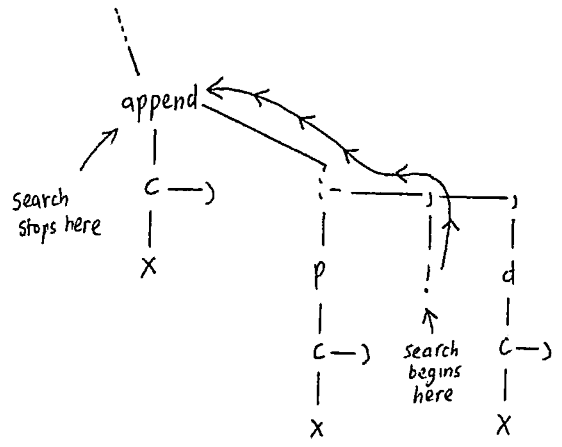
```
?- consult_list( [ file1, 'fred.1' ] ).
```

with the rules

```
consult_list([]).
consult_list([X|Y]) :- consult(X),
                       consult_list(Y).
```

6.5 Cut (!).

The code for this predicate sets a flag called 'cutflag' which is in each node of the question tree that is of type identifier. When a cut symbol is found a search back up the question tree is carried out. The search stops when a questionop is found or the identifier before a ruleop symbol is found. Thus for a question such as



A search is started at the cut and stops at 'append'. Starting at the 'append' node the 'cutflag's in the identifiers are set to true. This is done until the cut symbol is reached. The 'cutflag' is used in 'satisfy' to stop a goal being resatisfied.

6.6 Read.

This predicate, in C&M, reads the next term from the current input stream. A term is a line of PROLOG ending in a full stop. The 'read' in this interpreter reads only the next symbol which may be an identifier, a number or a symbol. A full stop is still required to complete the line. Thus for

```
?- read(duck).
```

the user types duck. <cr>

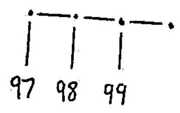
PROLOG responds with

```
yes
```

6.7 Name.

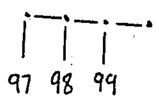
The code for 'name' is long but not particularly complex. The predicate converts an atom into a list or a list into an atom. Thus 'xname' consists of 'if' tests which decide if 'atomtolist' or 'listtoatom' should be called. Procedure 'atomtolist' will convert something like

abc to



where 97 is the ascii code for 'a' and so on.

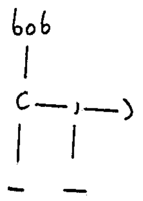
Similarly, 'listtoatom' will convert something like



to abc

6.8 Functor.

'Functor' can be used in 2 ways as described in C&M. The procedure 'xfunctor' first distinguishs between these 2 cases and then manipulates the tree using procedure 'build'. 'Build' takes a functor (bob, for instance) and an integer representing the arity (2, for instance) and creates a tree. For 'bob' and 2 the tree



will be created. '_' is the anonymous variable.

6.9 Arg.

The only difference between the interpreter's 'arg' and the 'arg' in C&M is that in this interpreter

```
?- arg( 2,f(a),X ).
```

will give

```
X = a
```

instead of failing.

If the 1st argument is greater than the arity of the predicate then the variable gets the last argument of the predicate.

6.10 Univ (=..).

In C&M, 'univ' can manipulate lists

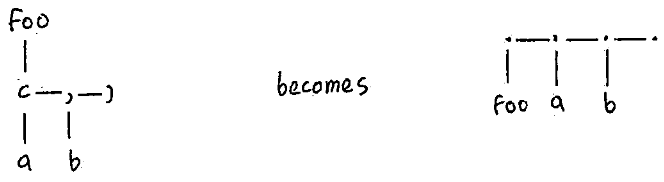
e.g

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

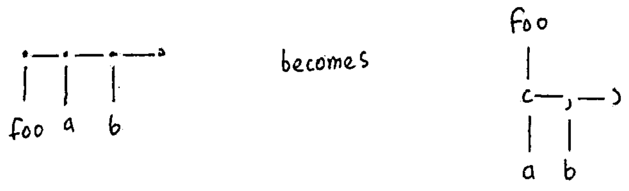
will give

```
L = [ '.', a , [b,c,d] ]
```

This form is not allowed in this interpreter since the dot '.' is not allowed as a functor name for lists. 'Xuniv' uses 2 procedures - 'createlist' and 'createstr'. 'Createlist' converts a structure to a list.



'Createstr' converts a list to a structure.



6.11 Is.

The 2nd argument of 'is' must be an operator. Thus

```
?- is(X,2).
```

is not allowed. In C&M the 2nd argument must be an arithmetic expression. Unfortunately, a definition of this is not given. In some PROLOGs the above form is allowed. A user could get round the above restriction by typing

```
?- is(X,+(0,2)).
```

In general, assignments of this form seem to go against the spirit of PROLOG. Assigments, at least simple ones, should be done by pattern matching.

6.12 Strict (==).

The code for '==' is in procedure 'xstrict'. It uses a flag 'strictflag'. When 'xstrict' is first called, 'strictflag' is set to false. 'Xstrict' then calls 'xeqop' which calls 'evaluate'. Inside 'evaluate', if two different variables are being matched, eventually 'qdbvarend' will be called and 'strictflag' will be set to true. All the procedures will finish and control return to 'xstrict' which will fail.

7. Diagnostics.

7.1 Diagnostics for PROLOG.

The code for these starts towards the end of the interpreter with procedure 'xtrace' and finishes with procedure 'xnospy'. These procedures use 2 global boolean variables - 'debugon' and 'trace'. 'Debugon' is true whenever diagnostic output can be generated during the execution of a question. 'Debugon' will be true if 'trace' is on or/and a spypoint has been set. 'Trace' is true only if the predicate 'trace' has been switched on. The distinction has been made so that a quick test can be done in 'satisfy' to see if procedure 'debug' has to be called.

To represent the spypoints, a list data structure called 'spypter' is used. The procedures 'addtospy', 'insertspy', 'removespy' and 'ridspy' are used to manipulate it. They are called from the procedures for the spypoint predicates which are 'xdebugging', 'xnodebug', 'xspy' and 'xnospy'. The 'spypter' data structure is referenced using the variable 'spyhead'.

Each node of 'spypter' contains a functor name and a number representing the arity of the predicate. For instance, the list might contain 2 nodes with the same functor name but with different aritys.

```
e.g        sum(X,Y,Z) -->    sum, 3
```

```
sum(X,Y)    -->    sum, 2
```

Each node is in alphabetical order and if 2 nodes have the same name then their order in 'spypter' is the order in which they were typed . If a question of the form

```
?- spy(sort).
```

is asked then the spypter node is

```
sort, 0
```

The syntax, spy[sort(2),append] is not allowed. Something like it could be added using

```
?- spy_list([sort(2),append]).
```

with the clauses

```
spy_list([]).
spy_list([X|Y]) :- spy(X),
                   spy_list(Y).
```

Similar restrictions apply to 'nospy'. Once again the user could define

```
nospy_list([]).
nospy_list([X|Y]) :- nospy(X),
                     nospy_list(Y).
```

The PASCAL procedures 'xtrace' through to 'xnospy' only deal with setting up the boolean flags and the 'spypter' data structure. The structures are used by calls to 'debug' when 'debugon' is true. 'Debug' is called in 'satisfy' from 4 different places when a question is being treated differently. They are the 4 places that control can flow through a question as outlined in section 8.3 of C&M, called CALL, EXIT, FAIL and REDO.

Since a question in this interpreter is in the form of a tree and because of the way that a goal is resatisfied by searching from the root of the tree down to that goal, REDO is dealt with in a different way than that given in C&M. For example, in C&M, if 'trace' is on and the question

```
?- des(X),fail.
```

is typed with the following clauses in the database

```
des(X) :- b(X),c(X).
b(2).
c(2).
```

Then when the goal 'fail' is reached, the following will be printed

```
CALL : fail
FAIL : fail
```

C&M's diagnostics will then print

```
REDO : des(2)
```

since 'des' is the last goal in the question that succeeded. Then

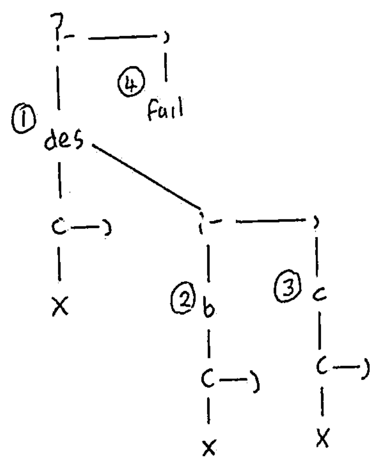
```
REDO : c(2)
```

will be printed since this was the last subgoal in the 'des' goal that succeeded. Then

```
FAIL : c(X)
```

will be printed since this subgoal fails.

In this interpreter, the difference in REDO can be seen in terms of the question tree which will be



The numbers indicate the order of the matches. At this point

```
CALL : fail
FAIL : fail
```

is printed. Now the first part of the question will be resatisfied. This is done by searching for a goal whose 'matchnum' equals 3. When this is found, the diagnostic is printed

```
REDO : c(2)
```

then

```
FAIL : c(X)
```

Thus the difference between C&M's REDO diagnostic and this interpreter's REDO diagnostic is that when backtracking takes place the parent goals reentered (in order to resatisfy their children) are not printed. In practice, this means that the number of REDOs printed is less than in C&M. The same number of CALLs, EXITs and FAILs are printed and usually the user is only looking for these three and in particular FAIL.

7.2 Interpreter diagnostics.

These are diagnostics which help in the debugging of the interpreter itself. They were used during development and have been left in so they can be used in future.

7.2.1 Boolean diagnostics.

These are

```
printeval
helpfulprinting
```

and

```
timer
```

They are set at the very start of the interpreter. 'Printeval' when set to true will cause 'writeln' statements to print things of the type

```
Swop q & db for qend
```

and

```
          evaluate
```

Generally, these statements consist of the name of the procedure that is currently being executed. 'Printeval' switchs on the 'writeln' statements for the pattern matching procedures. 'Helpfulprinting' is the boolean that switchs on 'writeln' statements in all the other large procedures in the interpreter.

'Timer' is slightly different in that it causes the run time for the satisfaction of a question to be printed. The timer starts after the <cr> at the end of a question and finishes when an answer is printed. If a question is resatisfied the timer starts from when the '; <cr>' is typed.

7.2.2 Data structure diagnostics.

These diagnostics are procedures which can be used to print out values in the data structures of the interpreter. They are called 'palfa', 'ptree', 'pbase' and 'pvarpter'. Calls to these procedures can be inserted inside new procedures that are being tested.

e.g

```
          ptree(q)
```

where q is the current node of the question tree. Or

```
          pvarpter(q^.varval)
```

Care must be taken that invalid pointers are not passed to these procedure. Thus if q is nil then

```
          ptree(q^.left)
```

will cause an execution error in PASCAL.

On the DEC20 a similar effect can be achieved by using the debug system [5] and getting the values of the data structure by typing a line of the form

```
          variable =
```

e.g

```
          q =
          q^.varval =
```

8. DEC20 dependencies.

The PASCAL used in this interpreter is the standard type. No DEC20 extensions have been used such as 'others' or 'loop'. This rule has been broken for two cases. The 'timer' diagnostic and also for input of data.

8.1 Timing questions.

The run time of questions is calculated using the DEC20 built in variable, 'runtime'. This variable is used in 4 places - in procedures 'printsuccess', 'printfailure', 'reanswer' and 'answer'.

8.2 Input.

Three input files are used in the interpreter at different times - 'inp', 'tty' and 'input'.

'Inp' is defined in 'initlisting' to be an interactive file. The file 'inp' is read using procedure 'readone'. Such a file type was created so that characters could be read from the DEC20 keyboard without having to wait for a carriage return to release the input buffer. 'Readone' is used by the input predicate procedures 'xget0', 'xget' and 'xskip'. It is also used in 'setdiagnostics'.

'Tty' is the standard input file from the keyboard. It is used in procedure 'readx'. The first character is read from 'tty' in 'initlisting'. 'Readx' is also used in 'nextterminalch' and also function 'tryagain'.

'Input' is the variable that is assigned the names of the files that are being consulted. It is used in 'nextfilech'. 'Input' gets a file name in 'xsee'. It is reset at the end of 'nextline' when the end of file is read.

By having 2 different next character procedures, one for the keyboard and one for files, it is possible to have questions inside consulted files. When a question requires the user to type something, input will be taken from the keyboard and the question will be able to continue.

9. Comparisons with UNH PROLOG.

For convenience, in this chapter, this author's PROLOG interpreter will be called Lehigh PROLOG.

UNH PROLOG is a PROLOG interpreter originally from the University of New Hampshire. It is written in C to run on UNIX. The version that was used by the author is located in the CAE lab in the Civil Engineering department in Fritz lab at Lehigh University. This version had been modified at Syracuse to run on the Data General MV/10000 under AOS/VS. Unfortunately, the modifications had not been totally successful and some advanced features do not work or do not work completely. For example, the ability to include C modules with the PROLOG code does not work.

The syntax used is like that used in the Edinburgh DEC10 PROLOG and thus is very similar to the syntax in C&M and so to that of Lehigh PROLOG. Unlike the Edinburgh PROLOG it does not compile any of its code. It also does not do any garbage collection unlike Lehigh PROLOG. This did not cause any problems in the tests that were carried out.

9.1 Timing comparisons.

In timing comparisons with Lehigh PROLOG it was found that small and medium size programs ran at about the same rate. On large programs, the differences in speeds became increasingly noticable. Both interpreters slowed down but Lehigh PROLOG was almost 15 times slower. Two of the large size programs used were an elisa program (to be found in 'elis.in') and an ATN program ('atn.in').

If these test files are looked at, it can be seen that 'atn.in' and 'elis.in' are not all that long. The reason why they are large programs is that when they execute they both create very large question trees. Time is then spent backtracking along and searching these trees. 'Atn.in' is also large in that it passes sizeable amounts of data about in its variables. This shows a few general rules to be kept in mind when writing a program for Lehigh PROLOG.

1. Try to keep questions short

e.g write

```
?- recognise(S).

?- parse(T).
```

instead of

```
?- recognise(S),parse(T).
```

2. If data has to be passed between questions, save the data in the database using 'asserta' and 'assertz', instead of passing it

through arguments.

3. Where possible try to restrict recursion. Each recursive call of a procedure adds a new subtree to the question tree.

4. Try to instantiate variables as soon as possible. Dont use a lot of uninstantiated variables that are sharing with each other.

These rules will keep the question tree and var list size down and so speed up execution.

9.2 Ease of use.

The UNH PROLOG starts very simply. The user must type

```
x prolog
```

The interpreter then starts with the prompt

```
?-
```

It expects the user to type a question and so prompts with a question symbol. In order to type in facts or rules, the user must first type

```
?- [user]. <cr>
```

to get the prompt

```
|
```

This is slightly inconvenient.

If the user wants to consult a file, he can use the list form.

```
?- ['atn.in']. <cr>
```

The system also allows file names with out the '.' notation which is useful. In that case no quotes are needed.

e.g

```
?- [vix]. <cr>
```

When a file is consulted, it is not listed on the screen while being read in. This means if an error occurs, a message appears out of nowhere. Also a file may not contain questions, unlike Lehigh PROLOG. This will cause difficulties if a user has a lot of files. Each time he consults one he must type in the questions to use them. He must also remember what form a question takes. By being able to have questions in the files where the clauses are defined, the user is spared a lot of typing and having to remember what questions to add. It is also a good aid to documentation.

It is possible to leave UNH PROLOG and edit a file and then return and reconsult it. Unfortunately, sometimes the system will not allow this and give an out of space error. Also if the user leaves PROLOG completely then edits a file and returns, the number of keystrokes are not that much greater.

UNH PROLOG does not allow variable functors. Thus a user can not write

```
            call( (X(Y)) )
```

See chapter 6, section 1 for more details on the extensions to 'call' in Lehigh PROLOG.

UNH PROLOG allows prefix, infix and postfix notations but is limited in that if a predicate is defined as infix or postfix then it can not be written in its default prefix form. Thus

```
        2 < 5 .      is correct
 but
        <(2,5).      gives an error.
```

There also seems to be some restrictions on using reserved words and symbols as ordinary words and symbols. For instance

```
            ';'(2,3).
```

is allowed but not

```
            ':-'(2,3).
```

UNH PROLOG allows real and negative numbers. It also has a lot of extra predicates such as

```
        sin(X), tan(X),  etc
```

One of the most useful is

```
            statistics
```

which gives, amongst other things, the amount of CPU time used by a question.

One inconvenience, which isn't the fault of the UNH PROLOG system, is that the high speed printer connected to the MV/10000 can not print all the PROLOG character set.

Overall, UNH PROLOG is a very interesting system althrough some things on it could be improved.

10. Improvements.

Since research is still continuing into the syntax and semantics of PROLOG as well as the implementation of it, this chapter has the potential to be infinitely long. Instead, discussion will be limited to some of the ideas being considered at present. An excellent overview of these can be found in [9].

10.1 Remaining predicates.

Firstly, the predicates 'op', 'reconsult, 'seeing', 'see', 'seen', 'tell', 'telling' and 'told' could be implemented. In practise this user has only found 'op' to be needed in some cases, where code has been written using a lot of arithmetic predicates.

```
is(X, +(5, *(Y,2)))
```

is a lot less convenient to write than

```
X is 5 + Y * 2
```

10.2 Language Modules.

Having, just recently, had access to two commercial PROLOG systems this user has seen some of the other versions of PROLOG available. The two versions seen were

VMS PROLOG-1

and

UNH PROLOG

Both of these are descendents of the Edinburgh DEC10 PROLOG and vary only in minor ways from the PROLOG described in C&M. Most of the variations are additions to the built in predicates available. Both versions also offer the incorporation of modules of code from other programming languages into PROLOG. VMS PROLOG-1 allows the addition of assembly language and FORTRAN subroutines. Unfortunately, there are restrictions on the types and number of parameters that can be passed. UNH PROLOG which is an interpreter written in C, allows the addition of C modules. There are obviously problems with this since it did not work on the Data General MV/10000 that this PROLOG was on.

The incorporation of other language modules does not seem to follow the spirit of PROLOG which is based on 1st order predicate logic. If the other modules were in the form of abstract data structures and operations, this would make the interface between PROLOG and the other language much more independent of the types of logic and control that the other language uses. Work has been done on implementing abstract data structures in PROLOG [10] so that different abstract objects, written in PROLOG, can communicate with each other by message passing alone. This type of system offers increased modularity and extensibility for PROLOG as well as the possibility of parallel processing. From that stage it is simple to incorporate other objects written in different languages. Such languages have been developed, such as SMALLTALK [11].

10.3 Extra predicates.

Some of the built in predicates in the other PROLOGs are worth implementing in this interpreter. The debugging features used in UNH PROLOG are enhanced to include 'creep', 'leap', 'skip', 'break', 'abort' and 'halt' of section 8.4 of C&M. Also included is 'ancestors(L)' which places the ancestor goals of the current clause into the list L.

Two other interesting predicates implement the idea of sets in PROLOG. These are

```
setof(X,P,S)
```

and

```
bagof(X,P,B)
```

'Setof' places all the instances of X, such that P is provable, into S.

e.g

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

might produce

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

The set S must be non-empty and will be ordered.

The predicate 'bagof' does much the same thing but the list produced will not be ordered and may contain duplicates.

Both PROLOGs examined contain a lot more arithmetic predicates. Connected with this is the fact that both PROLOGs allow real numbers of the form

```
real ::= integer 'E' exponent |
         integer '.' integer  |
         integer '.' integer 'E' exponent


exponent ::=  integer |
              '+' integer |
              '-' integer
```

Negative numbers are also allowed.

Some of the extra predicates are

abs(X)	exp(X)	log(X)
log10(X)	floor(X)	ceil(X)
rand	sin(X)	cos(X)
tan(X)	asin(X)	acos(X)
atan(X)		

Bit operations can also be done

X >> Y	shift X right Y places
X << Y	shift X left Y places
X /\ Y	bitwise conjunction
X \/ Y	bitwise disjunction

These additions to the interpreter would not involve a great deal of work. The syntax analyser would have to be altered to accept real and negative numbers. The code for the predicates would be similar to that in 'calcarith'. The fact that most of these operations are defined in PASCAL simplifies their PROLOG implementation a great deal.

Another useful feature of the UNH PROLOG, if it worked, is the automatic creation of a prolog.log file during the use of the interpreter. The log file contains a copy of everything that has happened during the current job. This feature can be mimicked on this interpreter by using the DEC20 'photo' command just before typing

```
ex prolog.pas
```

10.4 PROLOG defined predicates.

Some PROLOG interpreters have been written in PROLOG. Even at the simpliest level many built in predicates can be defined in terms of other predicates. Some of these definitions are given in C&M. For instance, the following predicates can be defined by other predicates

Predicate	written using
Listing	Clause
Call	Consult
Consult,reconsult	Basic file predicates

2 out of functor,arg,=..	In terms of the 3rd
Skip	Get
Get	Get0
Tab, nl	Put
Nonvar	Var
Atomic	Atom,integer
Repeat	Standard logic
';', not	Call
'\='	'='
'\=='	'=='
'*'	'+'
'/', mod	'-', '+'
'=<'	'=', '<'
'>'	not, '<'
'>='	'>', '='

In this interpreter it was decided only to define 'not' and ';' in terms of other predicates. They are

```
nott(X) :- call(X), ! , fail.
nott(X).
```

and

```
or(X,Y) :- call(X).
or(X,Y) :- call(Y).
```

10.5 Trees and stacks.

Most of the interpreters or compilers for PROLOG use a stack to store goals [7]. This makes backtracking very quick and easy since the current goal is unstacked leaving the previous goal ready to be resatisfied.

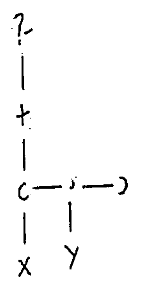
The trouble with a stack implementation is that it limits the types of search and backtracking strategies that can be tested. For instance, it is very inconvenient to unstack a goal that is not on the top of the stack.

A tree representation for a question means that many different types of control strategies can be tried. There is also a vast amount of literature on efficient tree search algorithms and representations for trees [12].

Up until now the PROLOG questions have been represented using the notation below

```
?- t(X,Y).
```

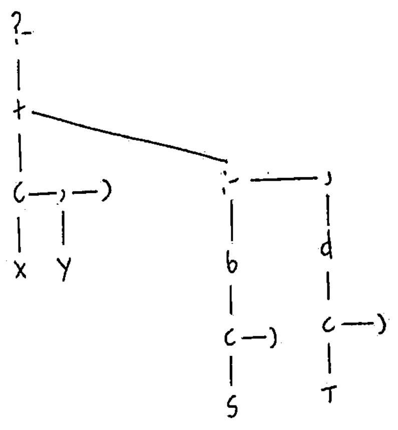
becomes



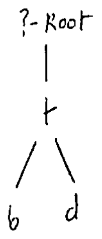
When a goal has been satisfied, by matching

t(S,T) :- b(S),d(T).

for instance, the new subgoals have been appended to the tree



't' can be thought of as being the main goal, with 'b' and 'd' being its subgoals. Thus the question tree becomes

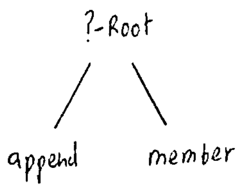


The arguments of the goals can be ignored in this representation.

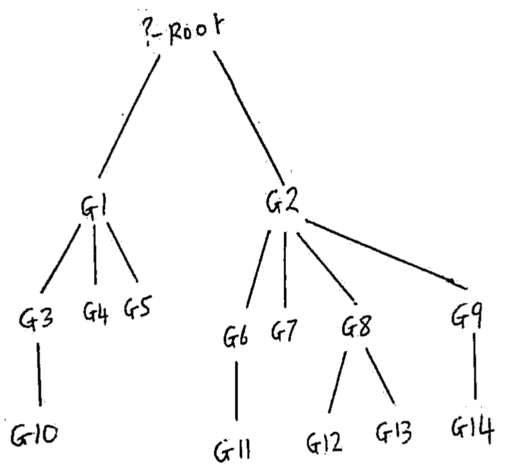
A question such as

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

would become



During the satisfaction of a question, the tree would grow and shrink and may look like



The numbering of the goals is not relevant.

Any particular goal G may be in 1 of 3 states - 'fresh', 'matched' or 'failed'.

A 'fresh' goal is one that has not yet been matched with any clauses in the database.

A 'matched' goal is one that has been matched with a database clause.

A 'failed' goal is a goal that has failed to match against any database clauses.

Given below is pseudo-code for searching a tree and also for backtracking along a tree.

```
procedure search(var finished : boolean);
begin
  finished := false;
  repeat
    PICK a 'fresh' goal from the tree (a leafnode)
    policy : depth-first;
    if found one then
      FIND a clause that matches the goal;
      if successful then
        make it a 'matched' goal;
        add subgoals to the tree below the current
        goal - all 'fresh';
      else
        mark goal as 'failed';
      end
    end
  until all goals are 'matched' or
        got a 'failed' goal;
  if all goals are 'matched' then
     print success;
     if question is not to be resatisfied then
       finished := true;
     end
  end
end; (* search *)
```

```
procedure backtrack(var finished : boolean);
begin
  finished := false;
  repeat
    PICK a 'failed' goal from the tree (a leafnode)
    policy : the latest;
    if found one then
      if parent has all 'failed' children then
        delete children;
        let parent be next 'matched' goal;
        else
          PICK closest 'matched' goal to chosen
          'failed' goal
          policy : the 'matched' goal that is the
                   'failed' goal's sibling or a
                   sibling's descendent;
        end
      else
        PICK a 'matched' goal
        policy : the latest;
      end;
      if found one then
        FIND a clause that may match the goal;
        if successful then
          make the goal 'fresh';
          any goals that are 'failed' are set to 'fresh';
        else
          mark goal as 'failed';
        end
      end
    until empty tree or
          no 'failed' goals;
    if empty tree then
      print failure;
      finished := true;
    end
  end; (* backtrack *)
```

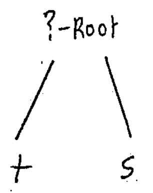
Both these pieces of code would be used in a procedure called 'answer' which would try to find an answer for the current question tree.

```
procedure answer;
begin
  repeat
    search(finished);
    if not finished then
      backtrack(finished);
  until finished;
end; (* answer *)
```

Thus a question

?- t(B),s(D).

would become



where '?-Root' is assumed to start as being a 'matched' goal in the algorithms.

An empty tree is just

?-Root

The goals 't' and 's' are initially 'fresh'.

There are 2 key procedures (functions) used in 'search' and 'backtrack'. They are 'find' and 'pick'.

'Find' choses a clause to match the current goal. In an actual interpreter different policies can be tested inside this function. 'Find' represents the 1st type of nondeterminism discussed by Kowalski [13].

e.g

for a question

```
?- append(X,Y,[a,b]).
```

and 2 clauses

```
1.  append([],X,X).
2.  append([A|B],C,[A|D]) :- append(B,C,D).
```

'find' would decide whether to use clause 1 or 2 to match the question.

'Pick' is the function that decides which goal to use next. Once again, 'pick' can be implemented in many different ways to test different policies. It represents the 2nd sort of nondeterminism mentioned in Kowalski [13].

e.g

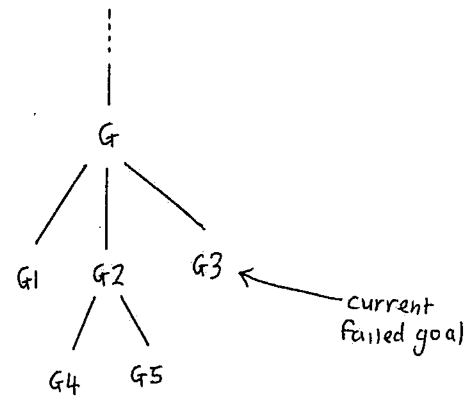
for a question

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

'pick' would decide whether to satisfy (or resatisfy)

```
          append(X,Y,[a,b])
or
          member(X,[a])
```

One assumpion of 'pick' is that it limits its chose to those goals that are leaf nodes. This means that only goals with no children (no subgoals) are picked out. Some of the possible policies for 'pick' have been included in the algorithms. The most complicated policy is in 'backtrack' for choosing a 'matched' goal that is closest to the current 'failed' goal. Graphically, this may be seen as



G1, G4 and G5 are all 'matched' goals. Thus when backtracking takes place this policy will decide between G1, G4 and G5 for the next goal to be resatisfied.

By varying the policies for 'pick' and 'find', all types of searching and backtracking stratergies can be investigated. Thus a possible improvement for the interpreter is to replace the current procedures which handle searching and backtracking with procedures for the algorithms given here. The procedures which would be replaced in the current interpreter would be 'answer', 'reanswer', 'dogoal', 'redogoal' and 'satisfy'. Then the interpreter will become quite an useful tool for testing different strategies like those outlined by Pereira [14]. This is only possible because of the tree structures that are used to represent questions and clauses.

Also implicit in this new notation is the fact that the actual structure of a goal does not need to be stored in the question tree.

e.g

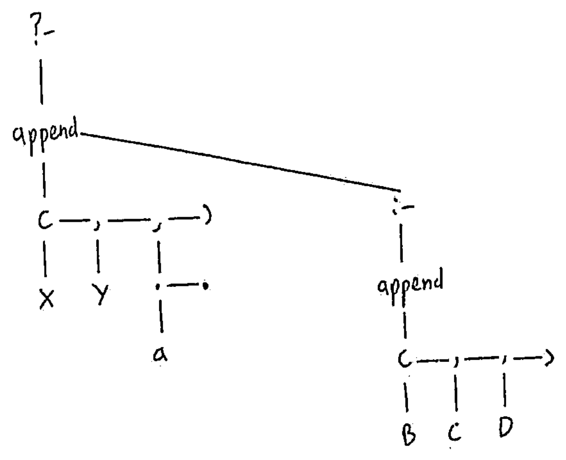
with clauses

```
1. append([],X,X).
2. append([A|B],C,[A|D]) :- append(B,C,D).
```

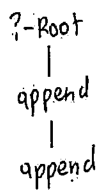
and a question

```
?- append(X,Y,[a]).
```

the final question tree may become



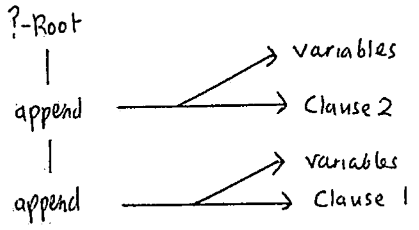
which becomes



The arguments and rule operator can all be ignored. During execution what happened was that the original question tree matched with the 2nd clause. The right hand side of the 2nd clause was added to the question tree. Then this right hand side was matched with the 1st clause. This gave the answer

X = [a]
Y = []

All this can be represented by pointers, like so



Each goal would use one pointer to access its matched clause in the database and use it as a 'skeleton' on which to hang its variables (stored on the other pointer). There is now the possibility that different parts of a question tree may have pointers to the same clause. This does not matter since the clauses are only being used as templates for the variables which are stored in a separate place.

In the current interpreter, pointers are already used to access a goal's variables. Only slight modifications would be needed to access these variables from a goal head instead of the variable nodes. Similarly, a pointer is already used to point at the clause that a goal uses. The pointer is stored in 'dbruleused'. At the moment, it is only used to indicate which clause is matching with a goal.

This idea is already used in most PROLOG interpreters and is called structure sharing [15].

One of the limitations of the algorithms given earlier is that only one goal will be considered at a time. For certain problems, parallel processing of goals would greatly speed up the solution. In that case, problems with dependencies between data would arise. The problem of deadlock would have to be considered. There may be two goals each waiting for each other to finish before they could continue. Some of these problems have been overcome in IC-prolog [16] which allows a certain amount of parallelism. The rule

```
sameleaves(X,Y) :- profile_of(W,X),
                   profile_of(W,Y).
```

can be speeded up by rewriting it as

```
sameleaves(X,Y) :- profile_of(W,X)//
                   profile_of(W,Y).
```

This causes the two profile_of subgoals to be evaluated in parallel. Notice how both subgoals use the value for W. Thus the two subgoals although running independently are constrained by a common variable W which either might change. It is possible to restrict the parallel evaluation so that only 1 subgoal is allowed to give a value to the shared variable W. This is done by annotating one of the occurences of W with a '^'.

e.g

```
        sameleaves(X,Y) :- profile_of(W,X)//
                           profile_of(W^,Y).
```

The '^' expresses the control condition that W must be unbounded on entry to the procedure. Now only this subgoal consumes values. A similar effect can be achieved by annotating the W in the first subgoal with '?'.

e.g

```
        sameleaves(X,Y) :- profile_of(W?,X)//
                           profile_of(W,Y).
```

The '?' annotation means that the W must be bound to a non-variable apon entry to the procedure.

These control annotations are doing explicitly what could be written into a procedure for 'pick'.

Also included in IC-prolog is stream IO. All these new constructs can be used to illustrate many different control alternatives and can even be used to model data-flow languages [17].

11. Conclusions.

One of the main strengths of this interpreter is also one of its weaknesses - it is written in PASCAL. PASCAL is a powerful language which meant that this interpreter was written using relatively little code and yet still retained a fair amount of clarity. Also since PASCAL was used, the interpreter can be modified with ease. Unfortunately, because PASCAL is so high-level the actual running time of PROLOG programs on the interpreter is quite slow. Even so for small to medium size programs the speed is fairly acceptable and compares favourable with UNH PROLOG.

Since this interpreter offers more powerful pattern matching than most other PROLOGs, programs can be written which are much conciser than those written in many other PROLOGs.

If some of the modifications discussed in chapter 10 are implemented then the interpreter will become an useful research tool. In particular the algorithms for searching and backtracking would enhance the interpreter a great deal.

Since the interpreter was written in standard PASCAL, it should be very simple to move it to another machine - it is almost 100% machine independent.

The greatest use of the interpreter will be as a teaching tool.

Combined with C&M it can be used to teach PROLOG. Also because of the use of standard recursive descent design and a LL(1) grammar, the interpreter can be used to teach compiler design.

It should be clear, from using this interpreter to write some PROLOG, just how powerful the language is. There are many things that can be improved in PROLOG -some of which were described in the previous chapter. Nevertheless, since PROLOG is grounded so firmly in logic, it seems clear that it is the direction in which programming languages should go [18].

PROLOG offers top down inference which unites problem solving and computer programming. By also offering non-determinism, parallelism and pattern matching it provides all the tools needed for applications in artificial intelligence [19].

By being a language which specifies a problem by what is to be done rather than how a thing is to be done, computer programs become a lot simplier to read. More over, since a program is rather like a specification of what it is supposed to achieve, it should be relatively easy, just by looking at it (or, perhaps by some automatic means) to check that it really does do what is required [20].

PROLOG shows that programming in logic is a practical possibility and is as quick and convenient as more conventional languages such as FORTRAN, PASCAL or even LISP [8].

References.

1. Welsh, J, & Elder, J., Introduction to Pascal, Prentice Hall, London, 1979.

2. Welsh, J, & McKeag, M., Structured System Programming, Prentice Hall, London, 1980.

3. Gries, D., Compiler Construction for Digital Computers, John Wiley & Sons, New York, New York, 1971.

4. LUCC, DEC System-20 User's guide, LUCC, Lehigh University, 1979.

5. LUCC, Pascal-20 Introductory User's guide, LUCC, Lehigh University, 1983.

6. Clocksin, W.F, & Mellish, C.S., Programming in Prolog, Springer-Verlag, Berlin, 1981.

7. Bruynooghe, M., ``The memory management of PROLOG implementations,'' From : K.L. Clark (Ed.), Logic Programming, Vol. 16, 1982, pp. 83-98.

8. Warren, D.H.D, Pereira, L.M, & Pereira, F., ``PROLOG - the language and its implementation compared with LISP,'' Proc. Symp. on AI and Programming Languages, SIGPLAN notices, Vol. 12, no.8, 1977, pp. 109-115.

9. Clark, K.L, & Tarnlund, S.A., Logic Programming, Academic Press, London, 1982a.

10. Kahn, K.M., ``Intermission-Actors in PROLOG,'' From : K.L. Clark (Ed.), Logic Programming, Vol. 16, 1982, pp. 213-228.

11. The Xerox Learning Research Group, ``The Smalltalk-80 System,'' Byte, Vol. 6, no.8, 1981, pp. 36-48.

12. Tarjan, R.E., Data Structures and Network Algorithms, SIAM, Philadelphia, 1983.

13. Kowalski, R., Logic for problem solving, Elsevier North Holland, New York, New York, 1979.

14. Pereira, L.M, & Porto, A., ``Selective Backtracking,'' From : K.L. Clark (Ed.), Logic Programming, Vol. 16, 1982, pp. 107-114.

15. Mellish, C.S., ``An alternative to structure sharing in the implementation of a PROLOG interpreter,'' From : K.L. Clark (Ed.), Logic Programming, Vol. 16, 1982, pp. 99-106.

16. Clark, K.L, McCabe, F.G, & Gregory, S., ``IC-PROLOG language features,'' From : K.L. Clark (Ed.), Logic Programming, Vol. 16, 1982, pp. 253-266.

17. Ackerman, W.B., ``Data flow languages,'' Computer, Vol. 15, no.2, 1982, pp. 15-25.

18. Cohen, P.R, & Feigenbaum, E.A., The Handbook of AI, Vol 3, William Kaufmann, Los Altos, California, 1982.

19. Clark, K.L, & McCabe, F.G., Micro-PROLOG : Programming in Logic, Prentice Hall, Englewood Cliffs, N.J, 1981.

20. Manna, Z., Lectures on the logic of Computer Programming, SIAM, Philadelphia, 1980.

I. Using PROLOG.

I.1 Getting started.

A description of how to use the interpreter will be based on the assumption that the user is running it on the DEC20.

The user must first have a copy of the interpreter. At present (May 1985) it is called prolog.pas. A copy can be obtained from

Professor S.Gulden
Department of Computer Science
and Electrical Engineering
Lehigh University

Also required is a PASCAL compiler/interpreter for PROLOG to run on.

If all these requirements are met then the user can start PROLOG by typing

```
ex prolog.pas
```

Alternatively, the user can run the prolog.exe file in this author's directory by typing

```
<davison>prolog
```

No matter which method is used once one of these commands has been typed the system will load and link the program and then print

```
output :
```

The user should type <cr>. The program will then begin

```
LEHIGH-PROLOG 1985
```

```
Printeval ? n
Helpfulprinting ? N
Timer ? Y
```

Three questions will be asked. The user should reply by typing a single letter answer for each one which should be Y(y) or N(n). No <cr> is needed after the letter. The prompt will then appear

```
|
```

The user can then consult a file

```
| ?- consult('file1.in'). <cr>
```

or type a clause

```
      | datum(2). <cr>
or    | ?- hello. <cr>
```

When a question has been satisfied the interpreter will wait for the user to decide what to do.

e.g

```
| ?- beef_stew(X). <cr>

  X = 2 _
        ↑
      cursor
```

The user can type <cr> to finish the question, or '; <cr>' to resatisfy the question.

When built in predicates such as get(X) are being used which require input, the prompt will not appear, and the cursor will remain at the left hand side of the screen waiting for input.

e.g

```
        | ?- get(X). <cr>

        ↑
        cursor
```

After typing input for these kinds of predicates, no <cr> is needed.

To leave PROLOG, the user can type

```
        | $ <cr>
```

or

```
        | <ctrl>c
```

PROLOG files must finish with an end of file symbol ^. if they do not then the interpreter will give an error and stop execution. File names are also limited to 9 characters and must be in quotes. See chapter 6 for the section on 'consult' for more details.

I.2 Common mistakes.

1. Always finish a PROLOG clause with a period '.'. If none is supplied the interpreter will consider the next line to be a continuation of the previous one. Such an error explains why no response occurs when the user types

```
        | ?- consult('dumbo.in') <cr>
        |
```

The interpreter is still waiting for a '.'.

2. Missing quotes can cause large segments of code to be ignored. Thus

```
| ?- consult('mickey.in). <cr>
| ?- consult( oops). <cr>
|
```

will cause nothing to happen. All the input, from the first quote, will be consumed as the name of the file in the first 'consult'. By the third line the interpreter is still waiting for the closing quote to the file name.

3. Misspelt names can cause strange failures of questions.

e.g

```
| bingbong(1).
| bingbong(2).
| ringading(X) :- binbong(X).
| ?- ringading(Q).
  no
```

The rule for 'ringading' misspells 'bingbong' as 'binbong'. Since there are no 'binbong' facts, the 'ringading' question fails.

4. Reserved words and symbols can cause errors if they are used as ordinary words and symbols. To get round this, the words or

symbols must be put in single quotes

e.g

```
        | is(2,4).
```

will cause an error since IS is a built in predicate. Instead write

```
        | 'is'(2,4).
```

II. Error messages.

There are 3 types of error message that can be issued from the interpreter

syntax error messages

built in predicate
error messages

and failure messages

II.1 Syntax error messages.

The syntax error messages are in the form of numbers with an arrow that appears under the part of a fact, rule or question that is syntatically wrong.

e.g

```
| num 23).
Clause ignored

       ^75
```

or

```
| ?- num 23).
Question ignored

        ^75
```

The meaning of the syntax error numbers follow

Number	Meaning
1	Integer is too large
2	Only part of '=..' typed
3	Only part of '\=' typed

4	Only part of '?-' typed
5	Only part of ':-' typed
10	Expected an identifier e.g hello
11	Expected a variable e.g X1
12	Expected an integer e.g 17
13	String e.g "...."
14	Questionop '?-'
15	Ruleop ':-'
16	Leftparent '('
17	Rightparent ')'
18	Comma ','
19	Leftbracket '['
20	End-of-file symbol '^'
21	Rightbracket ']'
22	Headop '|'
23	Period '.'
75	Current symbol in wrong place

II.2 Built in predicate error messages.

The 2nd type of error message is the built in predicate error message. These messages are in the form of sentences preceded by the word 'Error'.

Given below is a list of sentences printed and the procedures from where they originate. It is obvious from the message or messages printed which predicate has failed. The sentences are in alphabetical order.

Sentence	From procedure
';' predicate not built in, CONSULT PRED.IN	xdisjunc
1st CLAUSE argument must be a predicate	xclause
1st NAME argument illegal	xname
2nd argument of IS must be an operator	xis
2nd argument of NAME must be a list	xname
2nd NAME argument illegal	xname
2nd NAME argument must be a list	xname
Arguments must be integers	xnonarith
Argument must be numerical in TAB	xtab
Argument of PUT must be numerical	xput
Arithmetic arguments must be integers	getvalue
Assert fails	xassert
Atom not allowed as argument of GET	xget
Atom not allowed as argument of GETO	xget0
Atom not allowed as argument of SKIP	xskip

CALL fails	xcall
CONSULT fails	xsee
Input must be alphanumeric in READ	xread
List must be numeric in NAME	listtoatom
List not allowed	copiedid
List not allowed as 1st argument of '=..'	xuniv
List not allowed as 1st argument of FUNCTOR	xfunctor
List not allowed as 1st argument of NAME	xname
List not allowed as 1st NAME argument	xname
LISTING fails	xlisting
NAME argument illegal	xname
NOSPY argument must be a predicate	xnospy
NOSPY predicate argument must be numeric	xnospy
NOT predicate not built in, CONSULT PRED.IN	xnot
Number not allowed	copiedid
Overflow in "+"	calcarith
Overflow in "*"	calcarith
Predicate not allowed in CONSULT	xsee
Predicate not allowed in NAME	atomtolist

Relational arguments must be integers	xcompare
RETRACT fails	xretract
SPY argument must be a predicate	xspy
SPY predicate argument must be numeric	xspy
Underflow in "-"	calcarith
Uninstantiated list in NAME	listtoatom
Uninstantiated var	copiedid

II.3 Failure messages.

There are only 2 messages of this type. They are

```
****LINE INCOMPLETE***
printed in procedure 'nextterminalch'
```

and

```
****FILE INCOMPLETE***
printed in procedure 'nextfilech'
```

They both cause the interpreter to cease execution.

III. Extended BNF.

See chapter 2 for a discussion of some of these definitions

```
line ::=  ( fact | question | rule ) '.'


term ::=  constant | variable | structure |
          list | string | predicate |
          '(' argpattern ')'


structure ::=  atom '(' term { ',' term } ')'  |
               atom
                ↑
              not a p_atom


question ::=  '?-' ( structure | predicate )
                { ',' ( structure | predicate ) }


fact ::=  structure


rule ::= structure ':-' ( structure | predicate )
                    { ',' ( structure | predicate ) }


comment ::= '/*' { all_char } '*/'


argpattern ::=  argfact | argquestion | argrule


argstructure
     ::= (atom | varident) '(' term { ',' term } ')' |
         ( atom | varident )
              ↑
         including p_atoms


argquestion ::= '?-' argstructure { ',' argstructure }


argfact ::=  argstructure
```

```
argrule  ::=  argstructure ':-' argstructure
                               { ',' argstructure }


constant  ::=  atom | integer


variable  ::=  ( '_' | upper_case_char ) { char }


list ::= '[' term { ',' term } (('|' term)|/\) ']' |
         '[' ']'


string  ::=  '"' { all_char } '"'


predicate  ::=  p_atom '(' term { ',' term } ')' |
                p_atom


atom  ::=  lower_case_ char { char } |
           ''' { all_char } '''


p_atom  ::=  built in predicate words & symbols
             see 4th appendix

integer  ::= digit { digit }


char  ::=  upper_case_char |
           lower_case_char |
           digit |
           '_'


upper_case_char  ::= 'A' | 'B' | .... | 'Z'


lower_case_char  ::= 'a' | 'b' | .... | 'z'


digit  ::=  '0' | '1' | .... | '9'
```

```
all_char  ::=  DEC20 character set
```

IV. Built in predicate words and symbols.

All the words and symbols used here are reserved. If a user wishes to use the word asserta, for instance, as a name for an ordinary fact, he must write

```
'asserta'(12).
```

and not

```
asserta(12).
```

which will give a syntax error.

Similarly, a user can write

```
'\='(jim).
```

but not

```
\=(jim).
```

When a predicate word or symbol is used, it must have the right number of arguments or an error will be output.

e.g

```
?- asserta( fact(2) ).                is correct
```

but

```
?- asserta( fact(1), fact(2) ).     is wrong.
```

So for each word and symbol listed, its number of arguments (arity) will also be given.

IV.1 Predicate words.

Predicate	Arity	Predicate	Arity
arg	3	asserta	1
assertz	1	atom	1
atomic	1	call	1
clause	2	consult	1
debugging	0	display	1
fail	0	functor	3
get	1	get0	1
integer	1	is	2
listing	1	mod	2
name	2	nl	0
nodebug	0	nonvar	1
nospy	1	not	1
notrace	0	put	1
read	1	repeat	0
retract	1	skip	1
spy	1	tab	1
trace	0	true	0
var	1	write	1

IV.2 Predicate symbols.

Symbol	Arity	Symbol	Arity	Symbol	Arity
=<	2	==	2	=..	2
=	2	>=	2	>	2
\==	2	\=	2	/	2
<	2	+	2	-	2
*	2	!	0	;	2

IV.3 Other symbols.

If other symbols are to be used as atoms then it is still necessary to put them into quotes. This is purely to make the syntax of this PROLOG simple. Thus

```
%(1).
```

will give a syntax error but

```
'%'(1).
```

is correct.

' can not be used as an atom.

V. Files.

What will be given here is a quick summary of the files in this author's directory at the current time (May 1985). If a user wishes access to any of these he should get in touch with

Professor S.Gulden
Department of Computer Science
and Electrical Engineering
Lehigh University

All files with the '.in' postfix are PROLOG files.

File	Purpose
arit.in	Tests of arithmetic predicates
ass.in	Tests of asserta, assertz
atn.in	An ATN program
call.in	Tests of call
chart.in	An active chart parser
clas.in	Tests of var, nonvar, atom, integer, atomic
clau.in	Tests of clause, listing, retract
comp.in	Tests of <, >, >=, =<
cut.in	Tests of ! (cut)
diag.in	Tests of predicate diagnostics
elis.in	An elisa program
eq.in	Tests of =, \=, ==, \==
exap.in	Tests of pattern matching and backtracking without built in predicates

fini.in	A finite state automata parser
func.in	Tests of functor, arg, =.. (univ)
more.in	More tests of PROLOG but using built in predicates
name.in	Tests of name
nott.in	Tests of user defined 'not'
or.in	Tests of user defined 'or'
par.in	A program that executes sentences input in list form
pred.in	The file for user defined PROLOG predicates
prolog.pas .qas .rel .exe	The PROLOG interpreter
read.in	A program that reads in sentences
rept.in	Tests of repeat
rev.in	A program to create and reverse lists
rewr.in	Tests of get0, get, skip, put, nl, display, write
talk.in	A program to read in sentences and execute them

Vita.

My name is Andrew Davison. I was born in Macclesfield, England on July 23rd 1962. My parents are Stanley and Mary Davison.

As an undergraduate, I attended the University of Manchester Institute of Science and Technology (UMIST). I was there from October 1980 until June 1983 and I obtained a 1st class B.Sc Honours degree in Computation.

I have been at Lehigh University since September 1983 and will complete my M.S in Computer Science in June 1985.