BY BILL THOMPSON
AND BEV THOMPSON

Implementing a PROLOG Interpreter: Programming Structures

In the last issue we looked at the method PROLOG uses to search its data base in order to satisfy a query. This month we'll take a brief look at some of the programming components involved in actually implementing a PROLOG interpreter.

The first step is to make some decisions about the storage of sentences and queries. Some form of linked allocation method is an obvious choice. This method gives us a great deal of flexibility in designing the data structures, and if we use dynamic storage allocation, we are saved the trouble of having to estimate things like array sizes and rule and query lengths. Since many implementations of PROLOG are designed this way, studying linked allocation should give us some insight into the behavior of commercial PROLOG interpreters.

We need to be able to store four basic types of items: functors, constants, variables, and *cons* items. The first three have already been introduced, the fourth is the glue that holds the rules and queries together. The functors, constants, and variables are stored in records that contain a string field to store the actual data and a tag field to indicate which kind of data item the string represents. The *cons* items require a bit more explanation.

A *cons* item is a bit like the round disks that come with Tinkertoys. You insert sticks into the disks and at the other end of the stick you can attach an object or another disk. Connecting the disks and sticks together allows you to build large, complicated structures out of simple pieces. Similarly, the *cons* items allow us to build complicated data structures in memory by attaching constants, variables, and functors to one another in the proper order.

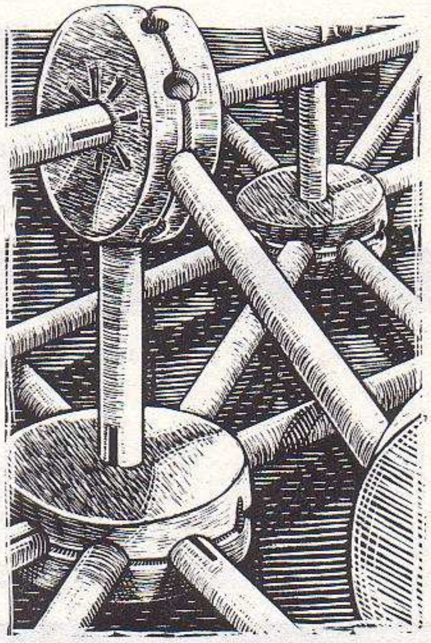


ILLUSTRATION: CHRIS MORTENSEN

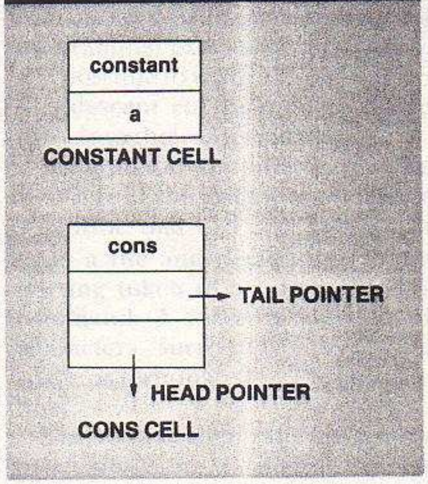
A look at some programming components for a PROLOG interpreter

A *cons* item consists of a tag, which identifies the record as a *cons* item, and two pointers, a head pointer and a tail pointer. The head pointer points to the first item in the list; this could be a data item, such as a constant, or the start of another list. In the latter case, the head pointer points to another *cons* item. The tail pointer points to the rest of the list. The items are linked together in a list by the tail pointers.

As we build lists, *cons* items will proliferate, but this method gives us a great deal of flexibility. We can represent a large number of complicated structures using this method without having to change our basic allocation routines. Figure 1 shows a typical *cons* cell. The basic data types are illustrated in Listing 1.

The next step is to define some memory allocation routines for the various kinds of nodes. We create a routine called *alloc_str* to allocate string storage for functors, constants, and variables. We pass this routine a string and a node type and it creates a node of the proper type to hold the string. The routine returns a pointer to the new node. Listing 2 contains this routine. *get_memory* does the actual work of retrieving free memory. This is a

FIGURE 1. Cell containing constant 'a' and *cons* cell.



low-level routine that varies a great deal among different implementations of programming languages, so we simply ignore its details and assume our programming language gives us the tools to construct the necessary allocation routines.

We also need some routines to perform the actual construction of lists. The basic list construction routine is called *cons,* which is short for construct. It is passed two pointers. The first points to an item that becomes the head of a list. The second points to the list to which the new item will be attached. *cons* creates a new *cons* node and sets the node's head pointer equal to the first pointer and its tail pointer equal to the second. The *cons* function returns a pointer to the new node. Lists are constructed by the repeated application of *cons.*

Routines to take lists apart are also necessary. *head* returns the head pointer from a *cons* cell and *tail* returns the tail pointer. *cons, head,* and *tail* are illustrated in Listing 3. Figure 2 shows some of the steps involved in creating the list ('a' 'b' 'c').

One other basic routine is also useful. *append_list* attaches one list to the tail of the other and returns a pointer to the newly constructed list. It does this by *cons* ing the head of the first list to a list created by appending the second list to the tail of the first. Listing 4 contains *append_list.* If you are uncomfortable with list processing or recursion in general, it is worthwhile to spend some time studying this routine. It illustrates the kind of subtle but powerful programming techniques used in processing lists. In many programming languages, recursive routines pay a heavy performance price. We will ignore performance constraints for the present. We can always tune the program up later.

LISTING 1. Record description for a node.

```
node_type = (cons_node,func,variable,constant) ;
node_ptr = ^node ;
node = RECORD
         CASE tag : node_type OF
          cons_node : (tail_ptr : node_ptr ;
                       head_ptr : node_ptr) ;
          func,
          constant,
          variable  : (string_data : string80) ;
         END ;
```

LISTING 2. Low-level allocation routine for strings.

```
FUNCTION alloc_str(typ : node_type ; s : string80) : node_ptr ;
 VAR
  pt : node_ptr ;
 BEGIN
  get_memory(pt) ;
  pt^.tag := typ ;
  pt^.string_data := s ;
  alloc_str := pt ;
 END ; (* alloc_str *)
```

LISTING 3. Basic list-processing routines.

```
FUNCTION cons(new_node,list : node_ptr) : node_ptr ;
 VAR
  p : node_ptr ;
 BEGIN
  get_memory(p) ;
  p^.tag := cons_node ;
  p^.head_ptr := new_node ;
  p^.tail_ptr := list ;
  cons := p ;
 END ; (* cons *)
FUNCTION head(list : node_ptr) : node_ptr ;
 BEGIN
  IF list = NIL
   THEN head := NIL
   ELSE head := list^.head_ptr ;
 END ; (* head *)
FUNCTION tail(list : node_ptr) : node_ptr ;
 BEGIN
  IF list = NIL
   THEN tail := NIL
  ELSE IF list^.tag = cons_node
   THEN tail := list^.tail_ptr
  ELSE tail := NIL ;
 END ; (* tail *)
```

COMPILING

Once we have decided on the basic storage methods, the next step is to develop a method of transforming the external form of the rules into their internal form as linked lists. To do this we have to be more precise about what constitutes a legitimate rule or query. The Backus-Naur form (BNF) shown in Listing 5 illustrates the formal syntax of rules and queries for our interpreter. In a BNF description of a grammar, "::-" means "is defined to be," and " | " means "or." In Listing 5, items surrounded with curly braces are descriptive rather than formal definitions. Items surrounded by single quotation marks are literal items and must appear exactly as shown in Listing 5. All other items are nonterminal components of the grammar and must be defined in the grammar. Using the grammar expressed in Listing 5, it is possible to write a simple recursive descent compiler to translate rules into their internal form.

To write the compiler, start at the first line of the grammar and write a routine that can accept a token from a file and decide if it is the starting token of a query, rule, or command. A token is a string of characters surrounded by white space or terminated by the end of a line.

The routine then calls the appropriate procedure to analyze the rest of the sentence. Each routine in the parser accepts tokens until it reads a token it can't recognize. At this point it returns control to the routine which called it. As a side effect of recognizing the components of a rule or query, the compiler builds the linked structure that represents the final form of the rule or query. The VTPROLOG interpreter available from the *AI EXPERT* Bulletin Board Services listed on page 5 and CompuServe account contains a simple compiler that accomplishes this. We won't go into the details of parsing and compiling here, we'll save that for another column.

The compiler constructs a data base, which consists of a linked list of *cons* nodes. Each node's *head_ptr* points to a linked list that represents the rule. Each component of the rule is also a list. The list might contain a single item in the case of a constant or variable, or could be more complicated in the case of a functor and its components. Figure 3 illustrates how a sentence like:

"likes(paul,X) : – likes(X,wine)."

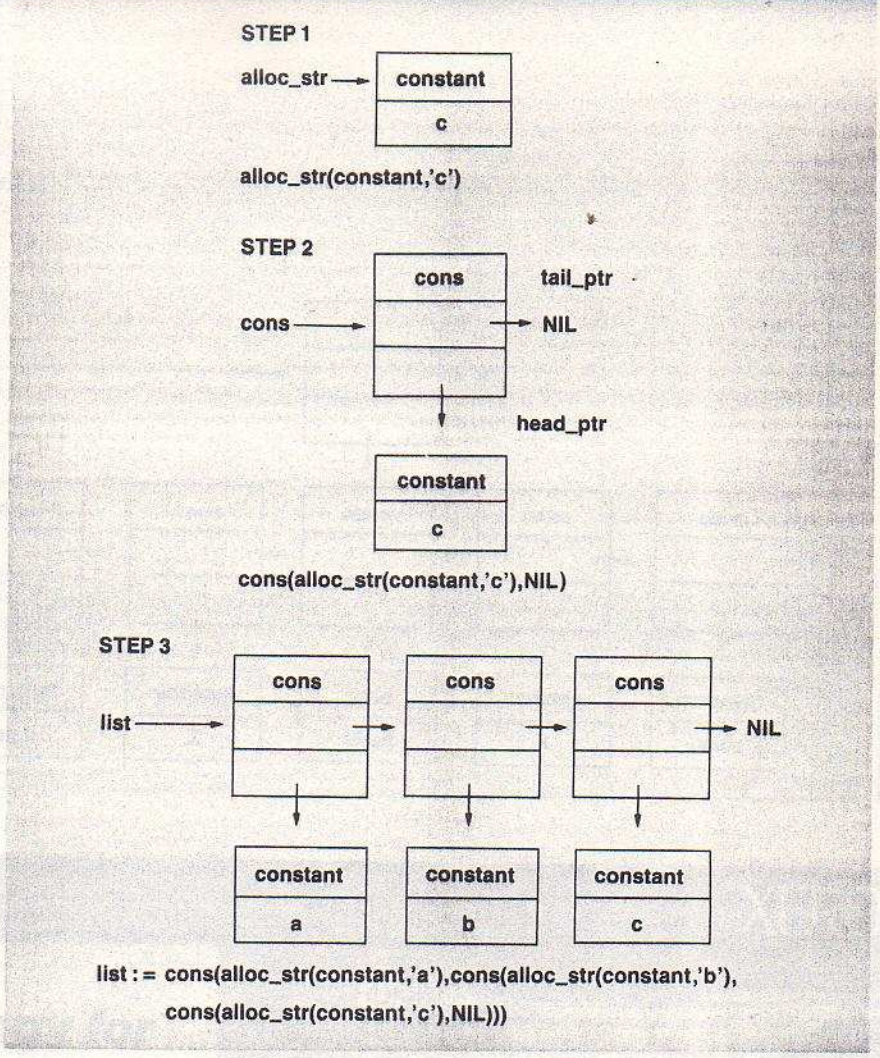
is included as the second sentence in the data base. This looks complicated, but we can take it apart easily by using *head* and *tail* procedures. For example, *head(tail(data_base))* points to the second sentence in the data base.

SOLVING QUERIES

Queries are compiled into a structure exactly like rules since they are syntactically similar. Only minor changes are necessary to the previous column's pseudocode version of *solve.* Listing 6 illustrates the new version of *solve.* We pass *solve* two pointers, one to the current query and one to the current environment, and an integer representing the current recursion level. The environment list is a linked list of *cons* items. Each *cons* node points to a list consisting of a variable and its binding. Figure 4 illustrates a typical environment list.

To perform the copy routines described in the pseudocode version of *solve,* we call *copy_list.* This function accepts a list and an integer representing the recursion level and returns a pointer to a new list. The new list is similar to the old, except that the variables in the new list have had the recursion level ap-

FIGURE 2. Constructing the list ('a,' 'b,' 'c').



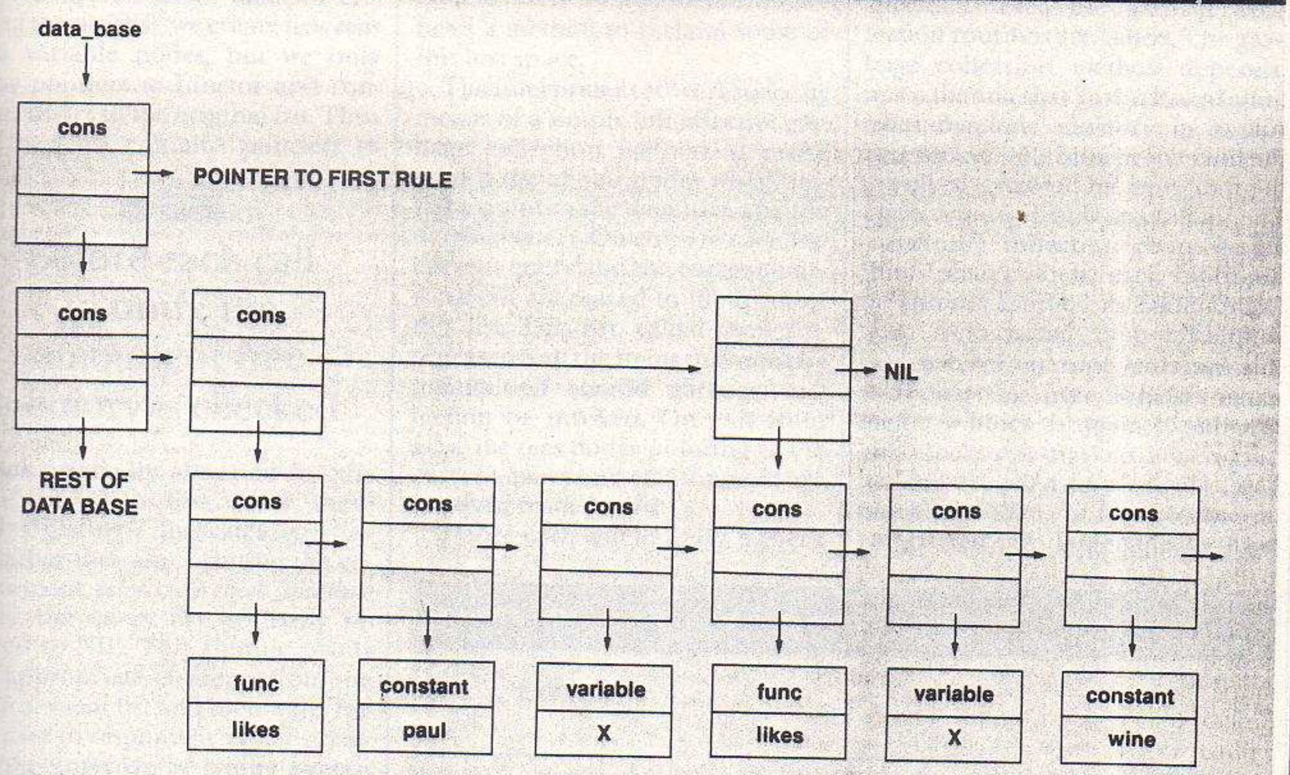
LISTING 4. The *append* routine.

```
FUNCTION append_list(list1,list2 : node_ptr) : node_ptr ;
 BEGIN
  IF list1 = NIL
   THEN append_list := list2
   ELSE append_list := cons(head(list1),append_list(tail(list1),list2));
 END ; (* append_list *)
```

LISTING 5. Grammar of rules and queries.

```
sentence ::- rule | query | command
rule ::- head '.' | head ':-' tail '.'
query ::- '?-' tail '.'
command ::- '@' file_name '.'
head ::- goal
tail ::- goal | goal ',' tail
goal ::- constant | variable | structure
constant ::- {quoted string} | {token beginning with 'a' .. 'z'}
variable ::- {token beginning with 'A' .. 'Z' or '_' }
structure ::- functor '(' component_list ')'
functor ::- {token beginning with 'a' .. 'z'}
component_list ::- goal | goal ',' components_list
file_name ::- {legitimate DOS file name, must be surrounded with
              single quotes if it contains a '.',':' or '\'}
```

FIGURE 3. The structure of *"likes(paul,X):– likes(X,wine)."*



pended. The new list also illustrates an interesting property of linked lists. They can share data. In creating the new list, we create new *cons* and variable nodes, but we only copy pointers to functor and constant nodes in the original list. Thus the new list contains pointers to nodes originally allocated for the first list. Both lists share these nodes. Listing 7 illustrates *copy_list.*

Before each call to unify, the amount of free memory is checked

Rather than just printing the environment, *solve* calls *check_continue* when the query list is finally reduced to NIL. This routine prints the appropriate elements from the environment list and then waits for the user to respond by either pressing the Enter key or typing a semicolon (;). The semicolon is interpreted as a request to continue searching for more solutions. Pressing the Enter key is interpreted to mean that the user is satisfied with the present solution and the search should be abandoned.

unify is implemented as a Boolean function which performs the matching described by Table 1. When *unify* binds a variable, it *cons*es a new list on the front of the current environment list. By putting new bindings on the front of the list, the proper environment is always available for backtracking. Listing 8 illustrates how new lists are attached to the front of the environment list.

make_binding illustrates one other interesting point. Variables that begin with an underscore are not bound to anything. They are called anonymous variables and are useful in forming queries in which we aren't interested in the particular values used to resolve a query.

GARBAGE COLLECTION

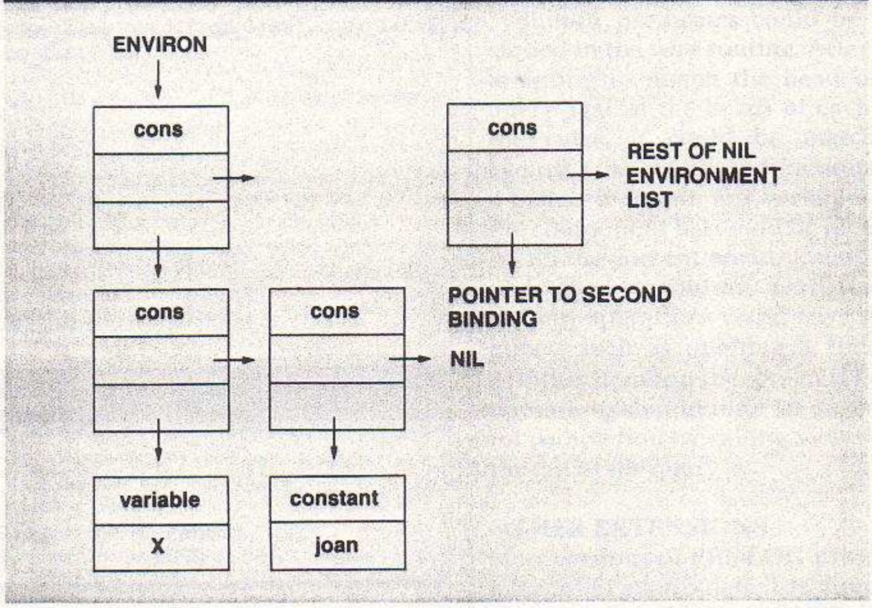
You may have noticed that some of our list operations result in nodes that are inaccessible. Repeated use of *copy_list* in the procedure *solve* results in a series of lists, portions of which we have no access to. This is not a problem if we have infinite memory available to us. Unfortunately, since real computers have severely limited memories, we need a method to reclaim some of this lost space.

The interpreter recovers space by means of a simple but effective garbage collection method. It maintains a list of *cons* nodes which initially point to the data base and the original query. On entry to *solve,* the current query and the current environment are *cons*ed to the front of this list. This list, called *saved_list,* represents all the items that must be maintained should garbage collection be invoked. On exit from *solve,* the *cons* nodes pointing to the current query and environment are removed from the list.

Before each call to unify, a check is made on the amount of free memory. If this amount falls below a specified level, the garbage collection routines are called. The garbage collection method depends upon the fact that Turbo Pascal allocates dynamic memory in eight-byte blocks. The interpreter considers all of dynamic memory to be a collection of eight-byte blocks.

Garbage collection proceeds in three phases. First, each block in memory is marked as being available. Next, *saved_list* is traversed and each cell on *saved_list* is marked as being in use. Finally, each memory block is again examined and blocks not marked as being in use are attached to a special free-block list. Adjacent free blocks are compacted into larger blocks. The

FIGURE 4. Typical environment showing *X* bound to *joan.*



LISTING 6. The *solve* procedure—final version.

```
PROCEDURE solve(list,env : node_ptr ; level : integer) ;
 VAR
  new_env,p : node_ptr ;
 BEGIN
  IF list = NIL
   THEN check_continue
   ELSE
    BEGIN
     p := data_base ;
     WHILE (p <> NIL) AND (NOT solved) DO
      BEGIN
       IF unify(copy_list(head(head(p)),level + 1),head(list),
                 env,new_env)
       THEN solve(append_list(copy_list(tail(head(p)),level + 1),
                 tail(list)),new_env,level + 1) ;
       p := tail(p) ;
      END ;
    END ;
 END ; (* solve *)
```

next time *get_memory* is called, the free list is first examined for a suitable block, and if one is found, that block is reused.

BUILT-IN PREDICATES

The entire interpreter described is available in source code form from the *AI EXPERT* BBS and CompuServe account. It is written in Turbo Pascal for the IBM PC and compatibles. It should not be difficult to convert to another version of Pascal or another language. If you wish to convert the program to another language, the target language should support recursion and some form of dynamic memory allocation. C or Modula-2 would be an ideal target language.

This interpreter illustrates the pattern-matching capabilities of PROLOG, but most implementations of PROLOG are considerably more sophisticated than this. Most commercial PROLOGs allow users to define rules that display messages on the screen, read the keyboard, and control the search mechanism. Also, our simple interpreter contains no arithmetic capability or ability to make numerical or string comparisons. Commercial interpreters provide these capabilities through built-in predicates. A built-in predicate is a goal whose definition is provided by the interpreter rather than the programmer.

Built-in predicates could be included in the *solve* routine. After attempting to match the head of a query against the heads of each of the rules, it could be matched against a list of built-in functions. If a match is found, any variables in the component list could be looked up on the current environment list and the operation performed. Built-in operations could have side effects such as opening a file or printing items on the screen. These operations should indicate successful completion by calling *solve* with the tail of the query.

LISTING 7. Routine to copy lists.

```
FUNCTION copy_list(list : node_ptr ; copy_level : integer) : node_ptr ;
 VAR
  temp_list,p : node_ptr ;
  level_str : string[6] ;
 PROCEDURE list_copy(from_list : node_ptr ; VAR to_list : node_ptr) ;
  BEGIN
   IF from_list <> NIL
    THEN
     CASE from_list^.tag OF
      variable : to_list := alloc_str(variable,
                                      concat(from_list^.string_data,
                                             level_str)) ;
      func,
      constant  : to_list := from_list ;
      cons_node : BEGIN
                   list_copy(tail(from_list),to_list) ;
                   to_list := cons(copy_list(head(from_list),copy_level),
                                   to_list) ;
                  END ;
     END ;
  END ; (* list_copy *)
 BEGIN
  str(copy_level,level_str) ;
  level_str := concat('#',level_str) ;
  temp_list := NIL ;
  list_copy(list,temp_list) ;
  copy_list := temp_list ;
 END ; (* copy_list *)
```

TABLE 1. Unification of items in rules and queries.

Item in the rule	Item within the query: Constant (C2)	Variable (V2)	Functor (F2)
Constant (C1)	Succeed if C1 = C2	Succeed bind V2 to C1	Fail
Variable (V1)	Succeed bind V1 to C2	Succeed bind V1 to V2	Succeed bind V1 to F2
Functor (F1)	Fail	Succeed bind V2 to F1	Succeed if expressions have same functor and arity and each pair of components can be unified

LISTING 8. Procedure to bind variables.

```
PROCEDURE make_binding(l1,l2 : node_ptr) ;
 BEGIN
  IF copy(string_val(head(l1)),1,1) <> '_'
   THEN new_environ := cons(cons(head(l1),l2),environ)
   ELSE new_environ := environ ;
  unify := true ;
 END ; (* make_binding *)
```

OTHER EXTENSIONS

Most versions of PROLOG provide a means to manipulate lists directly from rules. A list is an ordered sequence of elements. The ability to manipulate lists in PROLOG allows us to write concise programs to perform tasks like parsing and sorting. List manipulation in PROLOG will be the subject of a later article.

We hope that these two articles have given you some idea of the issues involved in implementing a PROLOG interpreter. The full VT-PROLOG interpreter is available for downloading from the *AI EXPERT* BBS. AI

Bill Thompson and Bev Thompson are writers and consultants located in Nassau, N.Y. They specialize in implementing AI techniques on microcomputers and are the authors of MicroExpert, an expert system shell. They have worked extensively on knowledge-based designs.