PROLOG From the Bottom Up

By Bill Thompson and Bev Thompson*

The Japanese Fifth Generation Computer Systems project uses a PROLOG derivative as its fundamental programming language in an attempt to undercut the dominance of American and European manufacturers.

SINCE AI is such a new field, very few have had the opportunity of the apprentice to study under a master craftsman. Except for those working in large university and corporate AI labs, most of us who are interested in AI are working in relative isolation, hungrily devouring all of the books, articles, and inexpensive software we can get our hands on.

In this column we would like to become your colleagues in the exchange of ideas, techniques, and information about resources. We hope that as we share our ideas with you, you will do the same with us. If you have a better solution to a problem than the one we present, let us know. Also, if you have a problem that is of general interest, or an unusual technique, by all means send it along.

We are planning to use actual code in presenting new programming techniques. In the column itself we will print chunks of code, and at time we will make programs available on the bulletin boards listed on page 5. Since we're gearing this column to those who, like us, come from the more traditional programming backgrounds, we've decided to work primarily in Pascal. This seems like a good choice not only because it's a good teaching language but also because it's familiar to most C and Modula programmers. As with everything else, our direction will be heavily influenced by your opinions, so be sure to be free with them.

We've always had the idea that one of the most effective ways to learn a new programming language is to write a program that implements part of the language. In our first two columns, we will describe a very basic implementation of PROLOG. Our focus in the first article is not how to program in PROLOG but how PROLOG operates and what features distinguish it from more conventional programming languages. In the next column we will discuss the programming structures that are used to implement those features. This will not only provide you with a valuable look inside the language but will also demonstrate some interesting Pascal programming techniques.

You adventuresome souls who want to get a jump on the next column may want to download the completed VT-PROLOG (very tiny) interpreter written in Turbo Pascal, which is available on the bulletin boards.

A SIMPLE EXAMPLE

We will begin to try to understand how PROLOG operates by examining a simple example. Consider a collection of sentences. The sentences consist of single lowercase letters, called facts, or more complicated structures, called

*Originally published in AI Expert, September 1986. Revised and typeset by Mark Morgan Lloyd, January 2010.

rules. The rules consist of a single lowercase letter and a special symbol (:-), followed by a fact or series of facts separated by commas. Each sentence is terminated by a period. This collection of sentences is called a data base. The following is a sample data base:

1. a :- b,c,d.
2. a :- e,f.
3. c.
4. d.
5. e.
6. f.
7. g :- e,f.

The numbers aren't part of the data base, they are merely for our convenience. Line 1 can be read "*a* is true if *b* is true and *c* is true and *d* is true." Line 3 says "*c* is true." The *a*s, *b*s etc. can stand for objects in the real world, but since PROLOG programs manipulate symbols and not their meaning, we will not confuse the issue by attaching an explicit meaning to each symbol. In line 1, *a* is called the head and *b,c,d* is called the tail of the rule. Line 3 has a head but no tail. The commas in rules stand for the word "and."

This data base might not seem particularly useful, but it is still possible to perform some interesting operations upon it. For instance, we might like to ask some questions about its contents. We can see that *c* is true within the data base, but what about *a* and *e*? Are facts *a* and *e* both true within this data base? A question about the data base is called a query. We write this query as *a,e.* It is easy to examine the data base and see if *a,e* is true, but as the data base becomes larger and the queries become more complicated. a systematic method of searching the data base will be necessary. We can develop this method by noticing a couple of facts. First, an empty query (symbolised by *NIL*) is always true. Second, if the head of the query and the head of the rule match, replacing the head of the query by the tail of the rule doesn't change whether the query is true or false.

The second statement may require a little explanation. To prove the query *a,e.*, we notice that line 1 of the data base says that *a* is true if *b*, *c*, and *d* are true. In other words, *a,e.* is true if *b*, *c*, and *d* are true and *e* is true. That condition can be expressed by the query *b,c,d,e.* If we can find a series of transformations such as this that reduce the query to *NIL*, then the original query was true.

One problem still remains with this method. Some transformations may lead to dead ends. For example, *a,e.* => *b,c,d,e.* will fail because no statement in the data base begins with *b*, so no further transformation is possible. In a case like this we back up to the query before the transformation and try the next rule. We only say that the query was false when all possible transformations of the original query have been exhausted. The process of backing up and trying a new rule is called backtracking.

We can express this a little more concisely with the pseudocode procedure shown in Listing 1. This pseudocode looks a great deal like Pascal, but we don't want to worry about things like data types for rules and queries yet so we will ignore them for the present.

LISTING 1: A procedure to solve queries.

```
PROCEDURE solve(query) ;

 VAR
  i : integer ;

 BEGIN
  IF query = NIL
   THEN write('yes')
   ELSE
    FOR i := 1 TO max_rule_number DO
     IF head(rule[i]) = head(query)
      THEN solve( append( tail(rule[i]), tail(query) ) ) ;
END ; (* solve *)
```

In Listing 1, *rule[i]* is simply one line from the data base. *Head* is a function that returns the first item of either a sentence or a query. *Tail* returns everything in a sentence or query after the head. If there is nothing following the head, *tail* returns *NIL*. *Append* is a procedure that merges pieces of rules and queries to produce a new query. For example, *head(rule[1])* = *a*, *tail(rule[1])* = *b,c,d*, and *append(tail(rule[1]),tail(query))* = *b,c,d,e.* Appending *NIL* to a query returns the original query: *append(f,NIL)* = *f*. The *solve* procedure is recursive; it calls itself with the transformed query. The recursion is terminated when either a query has been reduced to *NIL* or the search of the data base for a particular query is exhausted.

To get a feeling for this process, let's examine the solution to the query *a,e.* in some detail. Listing 2 shows the calls to solve and the transformation of the queries. The indentation level represents the level of recursion.

LISTING 2: Calls to procedure *solve* to answer query *?- a,e.*

```
call to procedure   matching rule

solve(a,e)          1.  a :- b,c,d

  solve(b,c,d,e)    no matching rule

solve(a,e)          2.  a :- e,f

  solve(e,f,e)      5.  e

    solve(f,e)      6.  f

      solve(e)      5.  e

        solve(NIL)  write('yes'), recursion terminated

      solve(e)      no matching rule (back out of recursion, continue search)

    solve(f,e)      no match

  solve(e,f,e)      no match

solve(a,e)          no match, finished
```

UNIFICATION

One reason for collecting facts in a data base is to represent relationships among the data items and to use those relationships to answer questions about the data. To accomplish this, we will first introduce a more convenient notation to represent rules and queries and then modify the *solve* procedure to handle the new notation.

Let's consider a relatively simple data base, which contains some information about a few people's personal preferences:

1. likes(joan,pool).
2. likes(alice,candy).
3. female(joan).
4. female(alice).
5. male(paul).
6. likes(paul,X) :- likes(X,pool),female(X).

Again, the numbers aren't part of the data base. Readers familar with PROLOG may recognise this notation. It is a convenient way to represent relationships. Line 1 may be read "Joan likes pool"[1]. "Likes" is called a functor and names a relationship between the first component, "Joan", and the second "pool". Line 3 says "Joan is a female".

Terms line "Joan", "female", etc. are loaded with millions of associations in our minds, but here we will only be manipulating symbols. Thus the knowledge that Joan is a common female name in North America has no meaning unless it is defined in the data base.

Line 6 is slightly different from the others: it contains a variable. In PROLOG's notation, constants begin with lowercase letters and variables begin with uppercase letters or an underscore. A variable is a symbol whose value will be determined in the context of some query. Line 6 can be read "Paul likes X if X likes pool and X is female". Or, more simply, "Paul likes any female who likes pool".

With this notation we form queries such as "Does Paul like Joan?" or "Who does Paul like?" The former is expressed as *?- likes (paul,joan)* while the latter is expressed as *?- likes(paul,A).* *?-* indicates that what follows is to be treated as a query. The first query will cause *solve* to respond "yes" or "no". In the second case, *solve* should not only tell us that Paul likes someone but who that someone is. The process of finding a value for a variable is called binding the variable, and the value of the variable is often called its binding. The set of all the current variables and their bindings is called the environment.

Of course, with the introduction of functors and variables, our previous version of *solve* is inadequate. The problem lies in the process of matching the head of a rule against the head of a query. Consider the query *?- likes(paul,joan).* The head of the query is *likes(paul,joan).* We take the functor and its components paul and joan to be a single entity. We can probably agree that it should not match the head of rules 1 through 5. Rule 1 tells us about what joan likes, so the functors match but the first components don't. Similarly, rule 2 tells us what alice likes so it doesn't match the query. Rules 3, 4, and 5 aren't about likes at all so they don't match the query. Rule 6 is different, however. The functors match and the first components match, but what about joan and X? In a case such as this, we say that the head of rule 6 and the query match, with the variable X from the rule bound to joan.

Now we can continue as before. We append the tail of the query *NIL* with the tail of the rule *likes(X,pool),female(X).* and try to solve this transformed query along with the added information that X is bound to joan. The new query could be interpreted as *likes(joan,pool),female(joan).*

Suppose instead we had posed the query *?- likes(paul,A).* Rules 1 through 5 still don't provide us with a match but the head of rule 6 will match, provided we can bind *X* to *A*. When one variable is bound to another,

[1] PROLOG notation has a convention that constants such as somebody's name start with a lowercase letter. For the purpose of explanation quoted examples relax this, hence "Joan likes pool" treats "Joan" as somebody's name as is conventional in written English.

we say that the variables share a binding. When one is matched to a constant, the other automatically takes on the same value.

We still have one complication to consider. What would happen if we were to add a new rule to the data base, for example, *likes(joan,X) :- likes(X,candy)*? Are the *X*s in this rule the same as in rule 6? Or suppose we has posed the query as *?- likes(paul,X)*. Is the *X* in the query the same one that appears in the rules? Questions like this are really an informal way of asking what the scope of a variable is. A variable's scope is the range over which its bindings are valid. Readers who are familiar with block-structured languages like Pascal or C understand the concept of scope. In a block-structured language a variable's scope is the block in which it is defined. That makes it possible to define a local variable, like *X*, in a procedure and also to define another local variable with the same name in another procedure. Since these variables have different scopes, they are different variables.

In PROLOG, a variable's scope is the rule or query that contains it. This means that in *likes(paul,X) :- likes(X,pool),female(X).*, each instance of *X* represents the same variable. That variable is entirely different from the *X* defined in *?- likes(paul,X)*. In order to avoid confusion among variables with the same name but entirely different meanings, we will have *solve* make a copy of the rule being examined. The copy will be exactly the same as the original rule in the data base but all variables in the rule will be tagged by appending the recursion level to them. Since a rule either fails or causes a recursive call to solve, this will mean that when seeking the resolution of a query, each time a rule is encountered its variables will have unique names. Listing 3 contains the new version of *solve*.

LISTING 3: *Solve* procedure using unification.

```
PROCEDURE solve(query,env,level) ;

 VAR
  i : integer ;
  new_env : same as query and env ;

 BEGIN
  IF query = NIL
   THEN print_env(env)
   ELSE
    FOR i := 1 TO max_rule_number DO
     IF unify(copy(head(rule[i]),level+1),head(query),env,new_env)
      THEN solve(append(copy(tail(rule[i]),level+1),tail(query)),new_env,level+1) ;
END ; (* solve *)
```

Solve is called with the current query as before. Two additional parameters are included with each call: *env*, the current environment, and *level*, an integer that contains the current recursion level. *Env* is a list of variables and their bindings. *Print_env* is a routine that prints the current environment. If a query is resolved (if some set of transformations reduce it to *NIL*), the current environment can be printed to show what set of bindings led to the resolution of the query. *Copy* is a function that is passed an object to copy and returns an integer. It then returns a copy of that object with the integer appended to each of the variables in the object.

Unify is a Boolean function that matches the heads of sentences against the head of the query. If it returns a value of true (indicating that a match has been made) *new_env* contains a copy of the old environment plus any new bindings that *unify* may have made. (The routine is named unify rather than something like *match* because the process of matching and binding is normally called unification.) *Unify* isn't particularly difficult to design. We'll leave aside most of its details, but Table 1 indicates how rules and queries should be unified.

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

Unify will have to be recursive because of the requirement to unify each of the components of a complex expression. Arity, mentioned in the table, is the number of components in the expression.

This description of the interpreter is somewhat concise, so let's look at a few steps in the resolution of the query *?- likes(paul,A).* to get a better understanding of this process. Table 2 shows the steps involved in resolving this query. One thing to note in this table is that when *unify* encounters a variable it looks up that variable's binding in the current environment. If it finds a binding, it attempts to unify the binding with the parameter being matched. Thus, in attempting to unify joan with *X#0*, it will look up the binding of *X#0* and then try to unify joan and *A*. Since *A* isn't bound, these two items are unified by binding *A* to joan.

TABLE 2: Solving the query *?- likes(paul,A).*

QUERY	ENVIRONMENT	LEVEL	MATCHING RULE
likes(paul,A)	NIL	0	0 likes(paul,X) :- likes(X,pool),female(X)
likes(X#0,pool),female(X#0)	(X#0 A)	1	1 likes(joan,pool)
female(X#0)	(A joan) (X#0 A)	2	3 female(joan)
NIL			

Although this description has been greatly simplified and does not consider such important concepts as how arithmetic and negation are handled, we hope it gives you an understanding of how simple relationships can be placed in a data base and the retrieved. If you download the program, experiment with setting up a simple relationship data base such as the one we've discussed, This will help you become more accustomed to looking at heads and tails and understanding the process of instantiation that makes PROLOG such a powerful symbol manipulator.

In the next issue we'll jump right in and see how to implement VT-PROLOG.

RESOURCES

When this article was originally published in 1986 the field of artificial intelligence was going through a "golden age" with extensive and well-funded research engendered, in part, by the Japanese Fifth Generation Computer Systems project.

The literature of PROLOG was growing rapidly. The basic form of the VT-PROLOG interpreter came from an article "Describing PROLOG by its Interpretation and Compilation" by Jaques Cohen, December 1985, *Communications of the ACM.* In the same issue was the article "PROLOG in 10 Figures" by Alain Colmerauer, one of the original developers of PROLOG. That article described PROLOG by means of a series of figures, both eye catching and informative. "Learning about PROLOG", a useful introductory article on PROLOG by Ramachandran, Bharath and Margaret Sklar, appeared in the July 1985 issue of COMPUTER LANGUAGE, pp. 49-54. The August 1985 issue of BYTE was dedicated to declarative languages, of which PROLOG was the most important example. PD PROLOG was a public domain interpreter available on a variety of bulletin board systems and was released by Automata Design Associates, 1370 Arran Way, Dresher, Pa. 19025.

By 2010 there is still significant interest in PROLOG, particularly with the addition of constraint logic programming capabilities to several implementations. Needless to say, the rise of the Internet with searching and indexing services has obviously made it far easier to find useful information and implementations, but it is hoped that the system described in this paper is still of some interest.

VT-PROLOG as currently released exists in these versions:

1.1	Original source as released in 1986, written by Bill Thompson for Turbo Pascal v3.
1.2	Modified in 2010 by Mark Morgan Lloyd to compile with both v3 and v5.5.
1.3	Modified to compile with both Turbo Pascal v5.5 and Free Pascal v2.4 (*forthcoming*).

Despite being more than 20 years old it still serves as an interesting testbed for experimentation, and there are still areas where an individual or small team could make useful contributions.

For example, the majority of PROLOG implementations are inherently single-threaded, i.e. irrespective of the number of CPUs ("cores") in a computer only one is actively engaged in solving a query. Since the major Pascal implementations support multi-threaded applications with ease, could this be used to provide a transparent performance boost to PROLOG?

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