Fun with Prolog

Dennis C. Merritt, "Exploring Prolog: Adventures, Objects, Animals, and Taxes", PC AI Magazine, September/October 1993.

(This article is an extract from a longer article that appeared in PCAI magazine, Sep/Oct 1993 issue, and is reprinted with their permission 602/971-1869.)

While I have since written a number of "useful" Prolog programs, I was first drawn to Prolog while I was in the middle of writing, for fun, an adventure game in 'C' on my first "personal" computer. I had started my 'C' program by building the basic tools needed for the game, which included a dynamic database to record the changing state of the game, and the ability to search for symbolic patterns in the state that indicated some action needed to be taken. The action was usually represented by a message to the user and a change of state of the game.

(SIDE BOX - Adventure Games, are games in which the developer creates a simulation of an environment, real or fanciful, that the user explores, usually with some objective in mind. It is generally full of puzzles that need to be solved in order to achieve the objective.)

As I was writing the utility functions for my game, I happened to go to a Boston Computer Society meeting where the speaker was discussing Prolog. I learned that all of the tools I was building were already integral components of the Prolog language.

Prolog has built-in dynamic memory allocation for storing the state of the game that was better than mine because it had an extremely flexible way of representing the data. It has a built-in pattern-matching capability (unification) that was more general and flexible than the pattern-matcher I was implementing, and it had a built-in search mechanism (backtracking). Further, the dynamic memory allocation didn't just store facts, but stored rules as well, so the "data" could embody its own intelligence.

Because of all of this procedural power built into the language, the code the programmer writes looks much more declarative. An application, such as my adventure game, winds up being reduced to an elegant set of logical declarations describing what the program does. (Developers often claim up to a 10-fold reduction in code size going to Prolog. See for example the PCAI article on KnowledgeWare's use of Prolog in the May/June 1993 issue.) For me, this was truly a fun way to program, capturing the essence of the joy of programming -- building logical structures that perform interesting tasks.

While in retrospect, the power of Prolog for writing adventure games might be obvious, think of how many other applications contain elements of remembering program state, searching for pattern matches in that state and taking conditional actions based on them. Further, how many of us think of our programs in logical terms, declaratively at first, and then spend the time translating that into procedural code?

Because of the ubiquitousness of these conditions in applications, Prolog has the potential for making almost any application more fun to write than it would be in a conventional procedural language. The following brief description of an adventure game (fun) and taxes (also fun in Prolog), illustrate some of these points.

Adventure Game

This is a skeleton adventure game in which the object is to go from the house, across the yard, to the duck pen to fetch a duck egg, being careful not to let the fox get any of the ducks. A typical interaction with the program looks like this:

```
>> goto(yard).
 You are in the yard
>> goto(duck_pen).
 You can't get there from here.
>> open(gate).
>> goto(duck_pen).
 You are in the duck_pen
 The ducks have run into the yard.
>> take(egg).
 You now have the egg.
>> goto(yard).
 You are in the yard
>> goto(house).
 You are in the house
 The fox has taken a duck.
 Thanks for getting the egg.
```

Prolog's dynamic database is ideally suited to represent the state of this game. There are Prolog facts in the program that describe where things are and the relationships between them. These are manipulated by other parts of the program, although the term program isn't often used with Prolog. More often a collection of Prolog predicates is referred to as a database, but in reality it is somewhere in between the two.

Some examples of Prolog facts are:

```
location(egg, duck_pen).
location(ducks, duck_pen).
location(fox, woods).
location(you, house).

connect(duck_pen, yard).
connect(yard, house).
connect(yard, woods).
```

One of the major advantages of Prolog is code can be tested almost immediately after being written. For example, after loading the above facts, the following queries could be presented to a Prolog interpreter.

```
        ?- location(fox, X).
        X = woods
        ?- connect(yard, X).
        X = house
        X = woods
```

To implement just the code we've written so far in some other language would require defining the data structures for the state, deciding whether to store them in pre-allocated or dynamically allocated memory, and writing a procedure that walks the state information and uses some pattern-matching code. While these types of programming challenges are fun as well, the resulting code would be visually dominated by the search and match routines with the result that the logic of the application would be hidden, much as the logic of a simple mathematical equation is hidden when it is coded in assembler.

Given a basic state defined as above, the commands that manipulate the state can be added. The fundamental operation in an adventure game is moving about, so one of the first commands to implement is a goto command that moves the player from one place to another.

```
goto(X) :-
  location(you, L),
  connect(L, X),
  retract( location(you, L) ),
  assert( location(you, X) ),
  write($ You are in the $), write(X), nl.
goto(X) :-
  write($ You can't get there from here. $), nl.
```

When given the goal, goto(X), where X might be one of the places in the game, Prolog tries the first clause which first finds the location of the player, determines if its connected to the destination, and, if so, updates the facts and writes a message. If there is no direct connection, the first clause fails and the second one executes instead, printing up its disappointing message.

(The assert and retract statements are built into Prolog (as are the write and nl (newline) predicates). They are used to dynamically modify the database, so that, in this case, location(you..) now reflects your new location.)

Other commands are implemented in a similar manner.

The fun part of any adventure game is the various side effects, or demons. Here is the main one for this game, the fox will get a duck if the ducks are in the yard and you are in the house.

```
fox :-
  location(ducks, yard),
  location(you, house),
  write($ The fox has taken a duck. $), nl.
fox.
```

The main control loop is handled with a predicate that uses recursion to execute over and over again, until the end state is reached. The first go tests the end condition, and the second reads and executes a command, tries the demons and then does it all over again.

```
go :- done.
go :-
  write('>> '),
  read(X),
  call(X),
  fox,
  go.

done :-
  location(you, house),
  you_have(egg),
  write($ Thanks for getting the egg. $), nl.
```

To run this program, simply consult it into a Prolog interpreter and type 'go'.

This skeletal prototype can easily be enhanced to include more flowing English for descriptions of what's happening and for accepting commands. In a Prolog that supports graphics, the I/O could manipulate graphic images instead of words.

But, no matter how its enhanced, the code for the game will continue to look like a concise logical specification of the game. This is in contrast to a more conventional procedural application that scatters the basic logic of the application amongst the mechanics of how it gets executed.

Taxes

I don't know about you, but I've never really enjoyed figuring out my taxes, that is, until the year I decided to write my own tax program in Prolog. The challenge shifted from getting all the right numbers in the right boxes, to putting together a logical structure that would do it for me.

To illustrate the idea, consider the following simplified tax form, 1040F (F for fantasy, if it were only so simple and cheap.)

```
line 1  wages                      |   |
line 2  tax - enter 5% of line 1   |   |
line 3  withheld                   |   |
line 4  refund (line 3 - line 2)   |   |
```

Each line of the tax form can be represented as a clause of the predicate line. The line predicate has four arguments: the form name, the line number, a description, and a value. The clauses for each line refer to the database of raw data and to each other as necessary. Given this, here is a Prolog program to compute taxes for form 1040F. (Remember, terms beginning with upper case are variables.)

```
tax:-
  line('1040F', 4, refund, X),
  write('They owe you: '), write(X), nl.

line('1040F', 1, wages, X) :- wages(X).
line('1040F', 2, tax, X) :-
  line('1040F', 1, _, WAGES),
  X is 0.05 * WAGES.
line('1040F', 3, withheld, X) :- withheld(X).
line('1040F', 4, refund, X) :-
  line('1040F', 2, _, TAX),
  line('1040F', 3, _, WITHHELD),
  X is WITHHELD - TAX.

wages(30000).

withheld(2000).
```

In this program, tax is the top level predicate which is called in a query to start the program. In true business like fashion, it immediately asks for the bottom line. The rule for line 4 calls rules for line 3 and 2, which call other line rules etc. This program accesses data from the predicates wages and withheld.

Backtracking search causes Prolog to automatically use all the necessary data and subsidiary lines to compute the bottom line. The program is very declarative in its nature, mimicking almost precisely the actual tax form. All of the procedurality required to perform the computation is handled automatically by Prolog.

Here's what it looks like when you run this simple program:

```
?- tax.
They owe you: 500
yes
?-
```

Conclusion

In these two short examples we've seen how the unique features of Prolog, dynamic memory allocation, unification and backtracking, make it a language whose code appears much cleaner, simpler, and "logical." The net result is a language that is, quite simply, a lot of fun to use.

Dennis Merritt is the author of Adventure in Prolog (Springer-Verlag), The Active Prolog Tutor (Amziod), and is a principle of Amziod in Stow Massachusetts, a vendor of Prolog products. He can be reached at e-mail amzi@world.std.com.