Prolog Under the Hood
An Honest Look

by Dennis Merritt

[This article was originally published in PC AI magazine, Sep/Oct 1992. The magazine can be reached at PC AI, 3310 West Bell Rd., Suite 119, Phoenix AZ, USA 85023 Tel: (602) 971-1869, FAX: (602) 971-2321, E-Mail: info@pcai.com, Web: http://www.pcai.com]

Prolog is an exciting and powerful programming language, ideal for most AI applications and for many non-AI applications. After a brief exposure, however, many programmers fail to appreciate its full breadth and never use it again. This article examines the inner workings of Prolog, with an eye on widening the language's appeal.

Introduction

High-level AI tools and most high-level non AI tools (like report writers and code generators) often meet the same fate as Prolog: great first impression, no long-term relationship. On the other hand, lower-level languages such as C, BASIC, COBOL, and even assembler, thrive with an addictive pull that has led many of us through long nights of programming.

Great Expectations

David Liddle's idea on application user interfaces give us a clue as to why lower-level languages draw more people in than higher-level ones (Liddle, 1989). He claims that the most important aspect of a good user interface is how well it leads the user to an accurate conceptual model of how the application works. If the user develops an accurate conceptual model, then the application works as expected. This leads the user to try more things, which also work as expected, leading to an even better understanding, thus drawing the user further and further into the tool.

On the other hand, if the user develops an inaccurate conceptual model, then things do not work as expected. Experiments fail and frustration sets in. Without a way to get a good understanding of how the application works, the user eventually loses interest no matter how slick the interface.

How the application works is not as important for user success as whether it works as expected. The goal of the user interface, then, should be to set the user expectations correctly. In other words, the user interface should teach the user how the application works.

With this in mind, we can see why a language like C is so addictive. As with any language, the syntax is the interface. C's syntax leads the developer to a good conceptual model of how C works. C does its business by generating machine code instructions for a CPU that manipulates memory. The syntax brings this out by including

- the declaration of variables that lead to an understanding of memory,
- sequences of coded instructions that lead to an understanding of the sequential nature of the CPU, and
- constructs (such as pointers) that lead to an understanding of the differences between addresses and values.

The design of low-level programming languages clearly reveals the computer underneath. Assembler does a wonderful job of this. That's why so many programmers are drawn to play with it at some time in their careers.

Low-Level Languages Have Problems, Too

Although low-level languages give a satisfying sense of how a computer works, they are often not very good at expressing the solution to the problem to be solved. The programmer's job is to map the problem's solution into the programming language.

If the problem involves straight-forward 'processing' of 'data', the solution maps relatively easily into the way a computer works, which is it has a processor that manipulates data. But, if the problem is more complex, mapping the solution into the code becomes more difficult.

The Purpose of High-Level Tools...

Higher-level programming tools were invented to bring the language of the tool closer to the language of the problem. Report writers allow you to specify the format of the report in a language closer to the report, rather than having to write the procedural code to generate the report. Inference engines allow you to specify rules of thumb without having to write the procedural code to figure what rules to apply when. Prolog allows you to code in logic--or so it would seem.

...And The Problem with High-Level Tools

Here's the catch. All of these high level programming tools are designed to let you code closer to the problem, and the user interfaces are designed to emphasize how well the tool maps to the problem. In order to be successful with the tool, however, the programmer must have a deep understanding of how it actually works.

But, because the tool's user interface usually emphasizes the tool's high-level aspects, the inner workings are often deliberately hidden.

Keeping the inner workings under wraps leads to frustration with learning a high-level tool. The documentation, the hype, and the syntax of the language are all geared to create in the developer a conceptual model of the higher-level view, and deliberately lead the developer away from a good conceptual model of how the tool actually works.

It's no wonder, then, that many programmers go back to the old way of doing things after a brief exposure to these tools. They promise to let you program at a higher level. Then they don't work as expected.

Prolog

Let's look at how this all plays out in Prolog. Everything about Prolog points to logic programming. Its name is derived from the phrase "PROgramming in LOGic". The academic computer science community considers Prolog an important language because it is about executable logic. Logicians like to teach Prolog for the same reason. You might be reading this article to learn about logic programming.

How Prolog Really Works

Although Prolog's original intent was to allow programmers to specify programs in a syntax close to logic, that is not how Prolog works. In fact, a conceptual understanding of logic is not that useful for understanding Prolog.

Prolog is much better understood as a language with two unusual features--unification and backtracking. (It also makes heavy use of recursion, which is more common than

unification and backtracking but difficult to grasp if you haven't encountered it before.)

Prolog's interface, however (along with much of the Prolog literature), deliberately leads the developer into a conceptual model of logic--and away from Prolog's true inner workings. The syntax of the language is derived from Horn clauses (an area of logic), and early teaching examples emphasize the Prolog-logic connection.

Questions and Answers

Consider this classic introductory Prolog program (the % sign indicates that the remainder of a line is a comment):

```
human(socrates).                % facts about who is human
human(aristotle).
human(plato).
god(zeus).                      % and who is a god
god(apollo).
mortal(X) :- human(X).  % a logical assertion that X is mortal if X is                                          %human
```

We can pose queries to (ie., ask logic-based questions of) this Prolog program. Here are a few: (Each question-mark is a prompt. Unprompted lines are Prolog responses.)

```
?-mortal(plato).                % is Plato mortal?
yes
?-mortal(apollo).               % is apollo mortal?
no
?-mortal(X).                    % for which X is X mortal?
X = socrates ->;
X = aristotle ->;
X = plato ->;
no
```

This is all fine, but the first glimmer that something is amiss comes from the last question mortal(X), which is equivalent to asking "for which X is X a mortal?" We get the three answers we expect, based on an initial conceptual model of Prolog as logic. Then the system says "no." Why "no?" The answer-"no" is short for "there are no more answers"--suggests that something procedural might be going on here, even though the example looks like an exercise in logic.

Toward Real Applications

Most new Prolog users run into difficulty the first time they try to do something real, something a little more programmatic. Suppose, for example, that after having seen the preceding set of queries top management says, "I want a formatted report listing all the mortals."

There isn't a clue in the language that the following program does the job:

```
mortal_report :-
        write('Report of all known mortals'), nl, nl,
        mortal(X),
        write(X), nl,
        fail.
mortal_report.
```

(Each nl generates a new line.) Given the input mortal_report, the program generates the requested information:

```
?- mortal_report.
Report of all known mortals
socrates
aristotle
plato
yes
```

How does it work? Why does it work? Does any aspect of the language lead a new programmer to put this sequence of instructions together? A conceptual model of executing logic certainly doesn't!

Backtracking and Unification

Consider the two key Prolog concepts, backtracking and unification. Let's see how a conceptual model based on them, rather than on logic, gives a more satisfying view of both the queries and the report program.

First, we need to understand that when we pose a query to Prolog at the ?- prompt we are asking it to see if the pattern of the query matches any patterns in the program. The way it matches patterns is called unification. The way it searches for patterns is called backtracking.

Let's look at the mortal program again. The query ?- human(socrates). will cause Prolog to search for that pattern. Prolog treats each query as a goal. Finding its goal, Prolog responds yes. Similarly ?- human(zeus). will cause 'no' to appear.

If we give it the query ?- human(X). Prolog will search for patterns that can match the query pattern if X takes on certain values. In our example this will work for X = socrates. So Prolog responds X = socrates ->. The arrow is the clue that backtracking is part of Prolog. Unlike in logic, Prolog did not find all the values of X for which the query pattern is true. It just found the first one. The -> indicates there may be more. Hitting ; tells Prolog to go look for more. In response, Prolog unbinds the variable X from the value socrates, continues searching, and responds X = aristotle ->. Again, the ; tells Prolog to look for more solutions, and we get X = plato ->. Asking for still more produces 'no' because there are no more solutions.

Backtracking is the repeated searching for additional solutions, so named because Prolog goes back and tries again to find a solution. Unification is the binding of X with each name in succession.

In the more complex example of the goal ?- mortal(X). Prolog matches the patterns with the rule mortal(X) :- human(X) (which means "X is a mortal if X is human"). Prolog then reduces the search for solutions to mortal(X) to a search for solutions to human(X), and the queries work as before.

Prolog's unification pattern-matching algorithm is much more powerful than this simple example can show, and it can be used to express elegant solutions to many complex

problems. Backtracking search quickly converges on solutions without the need for the developer to code the flow of control structures.

In other words, an awful lot goes on behind the scenes when Prolog responds to the simple query pattern human(X). Figure 1 schematically shows the full procedurality. First the goal, human(X) is called, and Prolog tries to match the pattern with the known facts. That is the significance of the upper left diamond. If Prolog succeeds in finding a match it exits, binding X to the value. If Prolog fails it prints no.

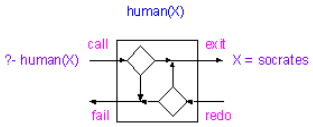

Figure 1: A schematic diagram showing how Prolog responds to a query

If the user asks Prolog to try again (by typing ;), the goal human(X) is reentered but conceptually from the other side. This is backtracking. The lower right diamond is another decision point. It indicates that if another fact is found that matches the pattern, rebind X to it and exit again. If not, the attempted pattern match fails.

Built-In Predicates

Prolog programmers have to do many of the things other programmers do, such as deal with input and output. To accomplish this, Prolog has built-in predicates. Syntactically they look just like the goals, but they behave differently. They only do the I/O when called, not when backtracked into. Further they always succeed when called and always fail when backtracked into. Figure 2 shows the flow of control through a built-in predicate.

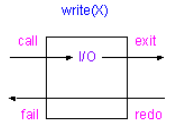

Figure 2: Flow of control through the built-in predicate write/1.

Given the underlying procedural nature of Prolog, it is natural for the programmer to want to exercise control. For our example, one possibility is to loop though all mortals to print a report. A number of predicates provide straightforward control over backtracking. The one in the example is fail. It always fails when called. Figure 3 shows the flow of control though fail.

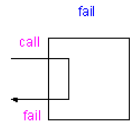

Figure 3: Flow of control through fail

The Report Program Revisited

Let's look at a simpler report program, human_report. This program differs from mortal_report in that it has no nls and no deductions about mortals:

```
human_report :-
        write(heading),
        human(X),
        write(X),
        fail.
```

Figure 4 shows what happens. The first goal writes the heading. The second goal causes a backtracking search for values. The third goal prints the value of X that results from a successful search. The fourth goal always fails, initiating backtracking. Backtracking passes through the third goal, and reenters the second goal, continuing the search for mortals. Another is found and the loop continues to the right. It continues in this way until all mortals have been reported.

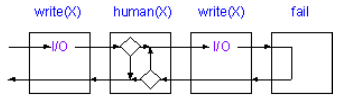

Figure 4: Flow of control through the human_report program

Less Glamor, More Beauty

When you look at Prolog in this procedural way, it seems less glamorous than when you look at it as pure logic programming. However, this view leads to a deeper understanding of the beauty and power of Prolog.

How many applications can benefit from automatic pattern-matching and backtracking search? Just about anything you'd want to do in AI and more.

For example, consider the problem of building a forward-chaining (data-driven) inference engine. This is useful for building several types of complex expert systems, such as configuration and scheduling systems. The inference engine goes though a cycle of looking for rules whose conditions are met, and then taking the appropriate actions.

In C, or some other low-level language, a developer would have to build

- pattern-matching for the conditions
- execution logic for the actions
- search mechanisms to find the matching rules
- a looping procedure.

Prolog handles these easily with unification, backtracking, and recursion. These few lines of code implement a simple forward-chaining inference engine:

```
forward_chain :-
        rule(conditions(X), actions(Y)),
        call(X),
```

```
        call(Y),
        forward_chain.
```

The search for rules that match proceeds via unification and backtracking, making the most complex part of the inference engine very simple. Recursion handles the looping.

Just about anything you want to do in AI requires pattern-matching and search, and that is what Prolog does best. Natural language parsing, game playing, expert system building, frame and object implementation, simulations--all are easy in Prolog.

Further, many conventional applications (like pricing, order processing, inventory checking, accounting transactions, and tax computations) also involve pattern-matching and search. These too are pratical and enjoyable to implement in Prolog.

Once a developer grasps a good conceptual model of how Prolog really works, all of the language's possibilities become clear.

Conclusion

In the AI industry we have created many neat programming tools, none of which have taken off as we hoped. Quite possibly it is because the languages try too hard to be something they are not.

Prolog, billed as "logic programming", is not really. You may be disappointed if that's what you expected to find. On the other hand, having backtracking, unification, and recursion inside one computer language leads to something very powerful and special.

Maybe if it was clearer exactly how tools such as Prolog worked, many programmers would be addictively drawn to them, as they now are to lower-level languages. The challenge to our industry is to design user interfaces that, instead of shielding a practical view of the tool, bring such a view to light. Then we will begin to draw more people in. For now, those interested in Prolog need to find a good book and perform a lot of experiments.

Until we start developing more intuitive interfaces, the tools of the trade will remain in the hands of those who have the perseverance to dig out an understanding of what's hidden beneath the surface.

In the case of Prolog, there is a special reward for that perseverance: It really is sort of like programming in logic.

Reference

Liddle, D. (1989) What Makes A Desktop Different. Paper presented at Agenda '90 Carlsbad, CA.

Dennis Merritt is the author of many Prolog books and articles and president of Amzi! inc.