THE PROLOG PHENOMENON

Drew McDermott
Yale University

In 1968, Carl Hewitt introduce PLANNER, a "procedural deductive system." [Hewitt 72] It featured some very innovative concepts for the theorem-proving community of that time: the procedural interpretation of deduction, pattern-directed procedure invocation, an indexed data base of assertions and programs, and nondeterminism (backtracking). A simple version was implemented by Sussman, Winograd and Charniak [Sussman 71] and used by Winograd in his exciting work on natural language. [Winograd 72] A very similar language, QA4, was implemented on the West Coast. [Rulifson 72] These languages, known by the ambitious term "AI languages," were widely perceived as providing built-in facilities that everyone would need for the next generation of AI systems, the way LISP had provided facilities for the previous one.

By 1972, however, people were being disillusioned. These languages became known for being inefficient and hard to control. [Sussman 72] Some attempts were made to implement improved systems without the drawbacks [McDermott 73], but they didn't really catch on. By 1975, no one was using the "AI languages" in the United States. LISP remained the language of choice, and people tended to use it to implement tools that departed drastically from the PLANNER tradition. [Bobrow 77]

At about the time AI languages were dying here, several Europeans, notably Alain Colmerauer [Roussel 75] and Robert Kowalski [van Emden 76], rediscovered the procedural interpretation of deduction. This was embodied in a language called PROLOG (for "PROgramming in LOGic") that seemed remarkably like PLANNER. Most Americans probably thought this was just the beginning of a delayed version of events here, and expected disillusionment to set in fairly quickly.

This has not happened. PROLOG has attracted and held a user community that is as fanatically devoted as most Americans are to LISP. (Its size is as yet small, but it is growing.) On a recent trip to Great Britain, I spent much of my time talking about PROLOG with some members of this community. My conclusion is that PROLOG is an interesting and powerful language, that deserves to have, and undoubtedly will have, more use here in the United States. In what follows, I will describe the language and say what I like about it (and what I don't like).

In LISP, you distinguish between a program and a single function. In PROLOG, instead of functions you have <u>relations</u>. A relation is an ordered set of <u>clauses</u>. A clause is of the form

```
pattern :- body
```

meaning, "To execute pattern, do body." The body is a list of <u>literals</u>, of the form <u>predicate</u>(-<u>args</u>-), although infix notations are allowed. (This is the syntax of Edinburgh PROLOG [Warren 77], which I will follow.)

For example, we can say

```
quadrat(A,B,C,Realroots) :-
   discrim(A,B,C,D), quadratl(A,B,D,Realroots).

discrim(A,B,C,D) :-
   mult(B,B,Bsquared), mult(A,C,Pl),
   mult(4,Pl,P2), add(Bsquared,D,P2).

quadratl(A,B,D,[]) :- D<0.

quadratl(A,B,D,[R]) :-
   D=0, add(B,MinusB,0), mult(2,A,TwoA),
   mult(R,TwoA,MinusB).

quadratl(a,B,d,[Rl,R2] :-
   D>0, add(B,MinusB,0), sqrt(D,SqrtD),
   add(MinusB,SqrtB,Numl),
   add(Num2,SqrtD,MinusB), mult(2,A,TwoA),
   mult(TwoA,Rl,Numl), mult(TwoA,mR2,Num2).
```

This program finds the real roots of a quadratic. It does not show PROLOG at its best (or its worst), but it does make it easy to compare with more traditional languages. The first thing to note is that clauses do not contain LISPy deeply-nested function calls, but instead a sequence of relation calls. To add 3 and 5, you write add(3,5,X), which will "set X to 8."

I put quotes around this last phrase to save PROLOG experts from apoplexy. X does not get set to 8 really, since there is no assignment in PROLOG. The interpreter instead tries to make add(3,5,X) true, by instantiating the variable if that's what is takes. If X is unassigned, it gets set to 8. If X is already set to 9, it does not change in value, but instead a failure occurs. (See below). Furthermore, exactly which variable is the output can vary. add(3,X,8) "sets" X to 5 if it is not assigned already. (Most implementations can't handle add(X,Y,8).)

The add relation is a primitive of the language. A non-primitive relation call is executed by finding the first clause of the relation that <u>matches</u> the call, and executing its body with the variable bindings from the match. So let's say we called :- quadratl(1,4,16,X). (We distinguish calls from other uses of literals by prefixing them with ":-".) This call would match the first clause, so the body, ":-16<0", is executed. A literal with no variables like this is executed by testing if it's true. It isn't, so a failure occurs. This causes backtracking to the last choice the interpreter made, which in this case was to use clause 1 of the relation. Now clause 2 is tried; :- 16=0 fails, so we try clause 3. :- 16>0 succeeds, so we are on our way. The next call is :- add(4,MinusB,0). This "sets" MinusB to -4, since, 4 + -4 = 0. Similarly, SqrtD becomes 4, Num1 is 0, Num2 is -8, TwoA is 2, R1 is 0, and R2 is -4. (All variables are local to the clause they appear in; there are no free variables in PROLOG programs, although the data base can be used to simulate them; see below.) We are done.

The answer returned is gotten by plugging clause-variable values back into caller variables. When clause 3 was called, X got bound to [R1,R2], so the X returned is [0,-4]. ([e1, e2, ..., eN] is PROLOG's notation for a list of N elements. [e1, e2, ..., eN |] is the list of the given N elements followed by a tail t.)

It is pretty clear that PROLOG is not the world's best language for solving quadratics. What is is really good for are the pattern matching and result building operations we got a taste of in the quadratic example. A better example of this is the way append is defined:

```
append([],L,L).  [If there is no body, :- can be
 omitted.]
append([X:L1], L2, [X|L3]) :- append(L1,L2,L3).
```

Then :- append([a,b],[c,d],A) is evaluated thus:

Clause 2 applies. A is bound to [a|L3]. (Note: variables start with upper-case letters, everything else with lower-case.) The recursive call :- append([b],[c,d],L3) is evaluated. Clause 2 applies again, with L3 bound to [b,L3] (a different L3, of course). The recursive call :- append([],[c,d],L3) is evaluated. Clause 1 applies, giving L3 = [c,d]. Unwinding the results, we have A = [a | [b | [c,d]]] = [a,b,c,d], as desired.

Since all function calling is via pattern matching, it is easy to specify how a result is to be unpacked in the same call that calculates it. E.g.,

```
:- quadrat(X,Y,Z,[R1,R2])
    (where X, Y and Z are already bound)
    --calculate roots and set R1 and R2 to them.
      (If fewer than 2 roots, fail.)

:- append(L1,L2,[_,_,Z|_]) (L1 and L2 already
   bound)
    -- append L1 and L2, and set Z to the third
       element of the result. ("_" is an anonymous
       variable that matches anything.) If fewer
       than 3 elements, fail.
```

Another advantage of this use of pattern matching is that variables can get bound to structures containing variables, which get filled in later. For a simple example, I will use the following relation:

```
member(X, [X|L]).
member(X, [Y|L]) :- member(X,L).
```

Now to find a list containing 3 and abc as members, we write

```
     :- member(3,Y), member(abc, Y)
```

The first call succeeds in clause 1 with Y = [3|L]. Then the second call must use clause 2 to set Y to [3, abc | L]. This corresponds to the answer "Any list starting with 3 and abc has these two things as members." A later user of this result may "set" L to anything he wants, or, more precisely, further instantiate Y's tail.

Note that, as in PLANNER, backtracking is global. If whoever wanted Y doesn't like [3, abc | L] as an answer, backtracking will generate [3 A, abc | L] the next time. If that still isn't good, the next answer will be [3, Z, W, abc | L], and so on ad infinitum. (Z and W are made up by the system; in actual implementations, they are more likely to have names like "_43.") Unfortunately, the second "member" goal never fails, so the system never backtracks to try lists with 3 in other than first position.

This illustrated one of the problems with global backtracking. It is too easy to generate weird loops through large sections of a program that do not converge to any solution, or do so very slowly. For this reason, there are facilities in PROLOG for squelching backtracking. If an exclamation point (or slash in some implementations) is executed, the backtrack history is thrown away up to some point, usually the call to the current relation. So

```
     :- member(3, Y), member(abc, Y), !
```

sets Y to [3, abs | L] and never generates another values; failures will propagate through this clause and keep going. (This was called FINALIZE in PLANNER.) The following construct is very common:

```
     foo(...) :- test, !, arm1.
     foo(...) :- arm2.
```

If the test succeeds, arm1 is evaluated, and failure there or later never propagates to arm2. so this is a kind of if-then-else. In my quadratl example, an experienced programmer would have put ! after D<0 and D=0, and left D>0 out altogether, since now control can reach the third clause only if D is known to be positive already.

Backtracking can be exploited to do certain kinds of loop. A file transducer might be written thus:

```
     filecopy(F1,F2) :-
        open(F1), open(F2),
        repeat,
           read(F1,X),
           (X=eof, !, close(F2);
           write(F2,X), fail).
```

repeat always succeeds, and can be backtracked to an infinite number of times. (p;q) tries p and, if a backtracking failure propagates out of p, then tries q. (This was called THOR in Micro-PLANNER). So in filecopy, if X doesn't equal eof, it is written to F2, and the fail cause a backtrack to the repeat. Note that the read and write are not undone on failure. When X does = eof, the ! throws away the backtrack history, including the repeat, and the output file is closed.

Here's another example: This program asserts (adds to the data base) every instance of p that can be deduced, and then succeeds:

```
assertall(P) :-
   P, assert(P), fail.

assertall(_). --Without this, assertall would
                always eventually fail.
```

There is an efficient indexed data base which holds all relations. Notice that a relation that consists only of a long list of body-less clauses like on(A, B). , on(C, table).,... is still considered a relation; assert adds a clause. Like I/O operations, asserts are not undone on failure. (This differs from PLANNER). Note that P, a literal, can be carried around as the value of a variable before being used in the body of a clause.

Another handy way to use literals as data is the "=.." feature. e =..31 succeeds if e is a literal and 1 is a list whose head is the predicate and whose tail is a list of the arguments of e. So the LISP convention that programs are data is honored in PROLOG. Here are some examples:

```
:- add(3,5,X) =.. L
   -- sets L to [add, 3, 5, X]

:- append([add,3,5], [X], L), E =.. L, E
   -- sets E to add(3,5,X), then sets X to 8.

:- (P :- Q) =.. [:-, fallible(X), human(X)]
   -- sets P to fallible(X) and Q to human(X)
```

As the last example shows, P:-Q, P+Q, and P>Q are all represented internally in a very similar way, and the system is not persnickety about the syntax of first-order logic.

Now I will comment about what I like about PROLOG:

First, it has a very nice way of doing pattern matching, result building, and list structure operations. These are combined in a rather different way from LISP, a way that has a lot of appeal. The pattern matcher is much more powerful than those PLANNER, QA4 or Conniver. The program for append would not work in any of these language.

Second, you get records for free. Say you want a record type "goalnode" to implement a story-understanding program. (Cf. [Charniak 80].) These have three slots: the character, this goal, and a list of the methods he might try to achieve his goal. We let each goalnode be a term goalnode(char, goal, [-plans-]). Pattern matching works exactly the same on terms like this as on lists. For example, to add a new goalnode to a list, merging plan lists if necessary, we could use

```
gnadd(GN, [], [GN]).

gnadd(goalnode(C, G, Pl),
     [goalnode(C, G, P2) | L],
     [goalnode(C, G, P) | L])
  :- !, append(Pl, P2, P).

gnadd(goalnode(Cl, Gl, Pl), [GN | Ll], [GN | L2])
  :- gnadd(goalnode(Cl, Gl, Pl), Ll, L2).
```

(The "!: in clause 2 is to prevent failures from causing a later call to clause 3.)

Actually, it is misleading to say that pattern matching "works the same" on terms as on lists, since internally lists are terms, of the form cons(el,cons(e2 , ...)).

Third, the PROLOG language has been implemented extremely efficiently, notably by David Warren and colleagues at Edinburgh. It is implemented as though it were just another lexically scoped nondeterministic programming language. For one thing, there are working PROLOG compilers; no American AI language ever had one, unless you count SAIL. [Feldman 72] For another, some implementations do "tail-recursive" function calling a la SCHEME. [Steele 78] The main difference between PROLOG and conventional languages is the way arguments and answers are passed up and down. These are handled by having the caller's variables be bound to structures containing the value cell s of the callee's variables. When the later are set or undone on backtracking, the values are automatically transmitted properly. The resulting representation of lists and other data structures is not as compact as LISP's, but just about as fast for most purposes. (To compensate for the space required to build a list like [X | L], where the "car" and "cdr" are waiting to be filled when X and L are known, the system does not have to use any space to remember to do a cons when X and L are known. This allows tail recursion in more circumstances than in a LISP-like language.)

Fourth, PROLOG supplies certain AI-oriented features, such as pattern matching and an assertional data base, so that the user doesn't have to implement them. It is not dogmatic about how they are to be used, so if you see a way to use them it is usually not hard (using things like =..) to massage your program and data into a form where they can be brought to bear.

In this connection, I was most impressed by the results of an informal experiment at Edinburgh last year. Two groups of students were taught AI programming, one in POP-2 (a LISP-like language), and one in PROLOG. After two months, the POP-2 group was writing pattern matchers and the PROLOG group was writing natural-language question answerers. Once they had learned PROLOG, the low-level stuff was already there.

Now the things I don't like about PROLOG:

Most published descriptions are wretchedly misleading. (1) The notion that programming in PROLOG is programming in logic is ridiculous. (As many exponents of programming in logic are eager to admit.) Some simple clauses can be thought of as first-order implications ("P:-Q" means "Q implies P"), but not most. So the claim that PROLOG programs are likely to have fewer errors than comparable LISP programs seems dubious. (2) The published articles often mention unification pattern matching. But for efficiency most implementations do not do full unification. That is, the matcher will allow f(x,x) to match f(y, g(y)), and bind x to g(x), without noticing the circularity. (3) It is often claimed that a PROLOG program can be used in more than one way, and simple ones can. For example, add can be used for subtraction. append can be used to:

```
append two list   E.g.: :- append([a,b],[c,d],X)
check an append   E.g.: :- append([a,b],[pc,d],
                                  [a,b,c,d])
decompose a list  E.g.: :- append(X,Y,[a,b,c,d])
etc.
```

Everyone quickly learns how seldom a program works this way. It does tend to be true that a program used to generate a result can be used to check one as well, but any other use will often introduce gross inefficiencies or infinite loops. In practice, this just means programmers have to devote about as much time to thinking about the different tasks a relation might do as they would in writing a set of functions for these tasks in any other language. (Plus, in PROLOG you have the problem of how to keep straight two separate versions of a relation, for different constellations of inputs. [McGabe 79]) (4) Finally, it is often claimed that PROLOG programs do not use side effects. It is true that you can't set a local variable of a clause. But the use of assert (and retract, the data base eraser) enables you to simulate all the global variables you need, and PROLOG programmers do this all the time. For example, here is how you find a list of all known objects with a given property:

```
findall(X,P,L) :- P, assert(note(X)), fail.
findall(_,_,L) :- reap(L).

reap([X|L]) :- retract(note(X)), !, reap(L).
```

This asserts note(X) for each X satisfying P, then erases and collects all the notes. (retract erases <u>one</u> thing matching its argument; if there aren't <u>any</u>, it fails. This version of findall is too simple, since if findall is called in the computation of P, the notes will get reaped by the wrong incarnation.) As in APL, things that are pretty in PROLOG tend to be very pretty and things that are ugly tend to be hideous.

PROLOG may be seen as an effort to simplify a theorem prover down to a point where it is as efficient as a programming language. I approve of this, but I think its inventors may have gone a little too far. They concentrated on implementing one basic idea with more and more efficiency, and have turned their backs on other ideas that may in the long run pay off. In other words, PROLOG may become the FORTRAN of logic-based programming languages.

For example, to test if some element of a list satisfies P, you just do

```
    :- member(X,L), P(X),.
```

To test if <u>every</u> element of a list satisfies P, you must do this:

```
    :- testevery(P,L) .

where
    Testevery(P, []).
    testevery(P, [Y|L]) :- P(Y), testevery(P,L).
```

The first case is just the Skolemized representation of the goal "Exist (X) member(X,L) <u>and</u> P(X)." But the second has nothing to do with <u>the</u> Skolemized version of "For all (X) member(X,L) <u>implies</u> P(X)." This is an awkward asymmetry, which <u>only</u> gets worse with more complex problems.

Let's look a little more closely at this example, in particular at the problem of proving that every element of [1, 1, 3] satisfies P. A theorem prover would do this by trying to prove

member(k, [1,2,3]) <u>implies</u> P(k)

where k is a Skolem constant occurring nowhere else. It would assert

member(k, [1,2,3])
and <u>not</u> P(k)

and look for a contradiction. From member(k, [1,2,3]), it would infer k=1 <u>or</u> k=2 <u>or</u> k=3. We get a contradiction if all elements of a disjunction led to a contradiction, so try

k=1 and <u>not</u> P(k)

By equality substitution, we get <u>not</u> P(1). So if P(1) is provable, we have a contradiction. Similarly with the other two.

Why aren't disjunction splitting and equality substitution in PROLOG? Probably because no one knows how to implement them efficiently enough to avoid slowing down the awesome interpreters now available. Perhaps the instincts of those responsible for today's implementations are correct, but perhaps someone should be trying to find speedy implementations of features like these. They would certainly improve the clarity of many programs.

Americans and Europeans have reacted differently to the problems of PLANNER-like languages. Americans tended to see the PLANNER interpreter as a problem solver, with PLANNER programs as data. It didn't work very well, so the Americans tried implementing different interpreters. The results were systems like NOAH [Sacerdoti 77] and AMORD. [de Kleer 77] They were implemented in LISP, and so were just as slow as PLANNER. but they have served as vehicles for studying control regimes that are fundamentally different from PLANNER's backtracking.

The Europeans went in a different direction. What they liked best about logic was its variable-binding machinery. Their attitude towards backtracking has been simply that it is a programmer's duty to remember that his programs will be executed backward as well as forward, that his programs must correct bad guesses as well as exploit good ones. If the backwards execution blows up, he must debug his program, not rewrite the interpreter, just as with more prosaic kinds of infinite loops. Once this burden was shifted away from the language implementer and onto the programmer, the logical next step was to freeze the interpreter design and make it as efficient as possible. The result is a programming language, not a problem solver or theorem prover; it doesn't compete with NOAH, but with LISP. And it's my impression that it competes pretty well.

The effect is to reverse the usual images of American and European computer scientists. In this case, the Americans have pursued impractical theoretical studies, while the Europeans have bummed the hell our of a hack.

Acknowledgements

I thank Lawrence Bird, Bob Kowalski, Frank McCabe, Fernando Pereira and David Warren for talking to me about PROLOG. I thank Ernie Davis, Bob Kowalski and Chris Riesbeck for comments on this paper. All opinions are mine alone.

References

[Bobrow 77] D.G. Bobrow and Terry Winograd. An overview of KRL, a knowledge representation language. Cognitive Science 1(1):3, 1977.

[Charniak 80] Eugene Charniak, Christopher Riesbeck and Drew McDermott. Artificial Intelligence Programming. Erlbaum Associates, 1980.

[de Kleer 77] Johan de Kleer, Jon Doyle, Guy L Steele, Jr., and Gerald J. Sussman. AMORD: explicit control of reasoning. in Proc. Symposium on AI and Programming Languages, pages 116. SIGPLAN/SIGART, 1977. (This was SIGART Newsletter 64.).

[Feldman 72] J.A. Feldman, J.R. Low, D.C. Swinehart, R.H. Taylor. Recent developments in SAIL, an ALGOL based language for artificial intelligence. Memo 176, Stanford AI Laboratory, 1972.

[Hewitt 72] Carl Hewitt. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a Robot. Technical Report 258, MIT AI Laboratory, 1972.

[McCabe 79] Frank McCabe and Keith Clark. The control facilities of IC PROLOG. Technical Report, Imperial College Department of Computing and Control, 1979.

[McDermott 73] Drew McDermott and Gerald J. Sussman. The Conniver reference manual. Memo 259a, MIT AI Laboratory, 1973.

[Roussel 75] P. Roussel. Prolog: Manual de reference et d'utilisation. Technical Report, Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[Rulifson 72] J.F. Rulifson, J.A. Derksen and R.J. Waldinger. QA4, a procedural calculus for intuitive reasoning. Technical Note 73, SRI AI Center, 1972.

[Sacerdoti 77] Earl Sacerdoti. A Structure for Plans and Behavior. American Elsevier Publishing Company, Inc., 1977.

[Steele 78] Guy L. Steele, Jr., and Gerald J. Sussman. The revised report on SCHEME, a dialect of LISP. Memo 453, MIT AI Laboratory, 1978.

[Sussman] Gerald J. Sussman, Terry Winograd and Eugene Charniak. MicroPLANNER reference manual. Memo 203a, MIT AI Laboratory, 1971.

[Sussman 72] Gerald J. Sussman and Drew McDermott. From PLANNER to Conniver, a genetic approach. In Proc. FJCC 41, pages 1171, AFIP, 1972.

[van Emden 76] M.H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. JACM 23(4):733, 1976.

[Warren 77] David H.D. Warren, Luis Pereira and Fernando Pereira. PROLOG: the language and its implementation compared with LISP. in Proc. Symposium on AI and Programming Languages. SIGPLAN/SIGART, 1977. (This was SIGART Newsletter 64).

[Winograd 72] Terry Winograd. Understanding Natural language. Academic Press, 1972.