A Tutorial Introduction to Prolog

Mehmet Gençer

November 18, 2009

1 Introduction: Prolog programs and queries

Prolog is a symbollic programming language based on declarative programming, not procedural programming. What we do when writing Prolog programs is to declare facts and rules. After that we ask questions to Prolog. Prolog tries to answer our questions by searching through possible ways of deducing satisfiability of our question based on given facts and rules (by doing so called backtracking search).

facts *relations*

Prolog programs contain statements about facts, relations, and rules. Every statement ends with a period, ".". A prolog program contains statements, or comment lines which start with %.

If a Prolog statement is an assertion about one thing, we call it a fact, it it is about multiple things, we call it a relation:

```
male(ali).
male(veli).
female(zeynep).
parent(ali,ayse).
parent(ali,ahmet).
parent(zeynep,ayse).
```

Prolog does not care about what "ali" or "ayse" is. It only knows assertions about them. This is why we call Prolog a symbollic programming language.

Note that interpretation of the parenthood relation is on us, i.e. parent(a,b) can mean that "a is parent of b" or "b is parent of a". In this example I mean the former. The only thing Prolog knows is a relation called "parent" exist between two entities.

If you store the above program in a file (e.g. family.pl), you can start the Prolog interpreter to load the file, and ask questions to Prolog. Start an interpreter:

```
\$ gprolog
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?-
```

Note that the interpreter prompts us with "?-", meaning that it is awaiting questions or orders. We tell the interpreter to load our program by entering:

```
[family.pl].
```

in the prompt. Now we can ask query Prolog to answer our questions, as follows:

queries

```
male(ali).
yes
| ?-
```

Prolog answered "yes" as the fact can be proven based on the known facts. We did not tell Prolog, as in a procedural language, how to reach an answer. We rather declare some facts, and let Prolog search for an answer. (However, as we will se below, Prolog has its certain ways -backtracking search- to search for an answer, and we will, and may need to, have some control over it in terms of how we define rules.)

variables

The questions can take the form of a search if we use a variable, so that Prolog will search for values that can be assigned to variable to satisfy what we want:

```
male(X).
```

Note that variables start with a capital letter. When we ask the following question, Prolog looks for values of X which satisfies male(X). It will display the first solution as follows:

```
X = ali ?
```

And will await a keystroke. Pressing ";" will cause Prolog to find more answers, if any. And pressing "enter" will cause it to discontinue its search, and prompt for new questions. Note that Prolog starts checking facts in the order they are given. Thus "ali" is found before "veli" as a solution.

search order

2 Rules and variables

rules We can define rules as follows:

```
father(X,Y):-parent(X,Y), male(X).
```

The above rule states that X is Y's father if X is Y's parent AND X is male. Add this rule to your program, and reload the program. Now you can ask a question like "father(X,Y)." to find all pairs which has a fatherhood relation:

```
father(X,Y).
X = ali,
Y = ayse ;
X = ali,
Y = ahmet.
```

Let us trace how Prolog answers this question, by turning on the tracer while asking the question:

```
trace,father(X,Y).
   Call: (8) father(_G335, _G336) ? creep
   Call: (9) parent(_G335, _G336) ? creep
   Exit: (9) parent(ali, ayse) ? creep
   Call: (9) male(ali) ? creep
   Exit: (9) male(ali) ? creep
   Exit: (8) father(ali, ayse) ? creep
X = ali,
Y = ayse ;
   Redo: (9) parent(_G335, _G336) ? creep
   Exit: (9) parent(ali, ahmet) ? creep
   Call: (9) male(ali) ? creep
   Exit: (9) male(ali) ? creep
   Exit: (8) father(ali, ahmet) ? creep
X = ali,
Y = ahmet ;
   Redo: (9) parent(_G335, _G336) ? creep
   Exit: (9) parent(zeynep, ayse) ? creep
   Call: (9) male(zeynep) ? creep
   Fail: (9) male(zeynep) ? creep
   Fail: (8) father(_G335, _G336) ? creep
false.
```

Note that Prolog first tries to find a pair of entities which satisfies parent(X,Y) relation, and only after that it checks whether X is male. Thus the

way we define rules effect the order of search Prolog conducts. Try defining a new rule:

```
father2(X,Y):-male(X),parent(X,Y).
```

in your program, and trace how Prolog answers "father2(X,Y)".

Providing multiple definitions for a rule is equivalent to OR operation. Either of the following will satisfy Prolog:

```
somebodysparent(X):-father(X,Y).
somebodysparent(X):-mother(X,Y).
```

The same thing could be expressed concisely using ";" for OR:

```
somebodysparent(X):-father(X,Y); mother(X,Y).
```

In cases like the above, we do not care about the value of Y. Therefore we can use "_" as an anonymous variable

anynymous variables

```
somebodysparent(X):-father(X,_); mother(X,_).
```

Sometimes a proof is based on failure to proove otherwise. For example we can say that someone has no child if he/she is not parent of anyone:

```
hasnochild(X):- \+ parent(X,_).
```

directives The "\+" is a predefined directive is Prolog, which is called negation as *negation as failure* failure.

We can use variables in rules for intermediate value storage. The following program contains statements to find Fibonacci numbers:

```
fibo(0,1).
fibo(1,1).
fibo(N,X):-
  N > 1,
  N1 is N-1,
  N2 is N-2,
  fibo(N1,F1),
  fibo(N2,F2),
  X is F1+F2.
```

variable binding The expression "N1 is N-1" will cause N1 to be bound to the value of N currently being tried by Prolog in its search. You can query the value of a Fibonacci number as follows:

```
?-fibo(5,X).
```

In this specific example there will be only one value for X that satisfy the relation of being fibo() of number 5.

2.1 Examples

1. Grammar rules and vocabulary for a simplified Turkish:

```
subject(ali).
subject(veli).
verb(at).
verb(tut).
noun(top).
sentence(S,N,V):-subject(S),noun(N),verb(V).
```

   we can query the program to generate all possible sentences in this language:

```
?-sentence(C).
```

3 Using lists

lists

Prolog programs can operate on lists. An expression such as "[a,b,c]" will be interpreted as a list. The predefined method, member(), checks list membership:

```
?- member(a,[a,b,c]).
true
```

The special notation "[H|T]" allow us to access head and tail of a list. For example the following rules provide a program to check if a list of numbers is in ascending order:

```
asc([]).
asc([A]).
asc([A|[B|T]]):- A=<B, asc([B|T]).
```

First two rules assert that an empty list and a list with single element are considered in ascending order. The third rule asserts that a list strating with A and B and having a tail list T is inascending order if A is less than or equal to B, and the list starting with B and having tail T can be proven to be in ascending order.

As a different example, following rules provide a means to find a list which is the result of appending first parameter to the second:

```
lappend([],X,X).
lappend([A|X],Y,[A|Z]):-lappend(X,Y,Z).
```

You can query the program as follows:

```
lappend(X,Y,[a,b]).
```

To find all possible X,Y pairs whose appending results in [a,b].

3.1 Examples

1. A modified version of simplified Turkish which allows for imperative sentences:

   ```
   sentence([S,N,V]):-subject(S),noun(N),verb(V).
   sentence([S,V]):-subject(S),verb(V).
   ```

   We can query the program to generate all possible sentences in this language:

   ```
   ?-sentence(S).
   ```

2. Finding unique values in the list:

   ```
   unique([],[]).
   unique([H|T],U):-member(H,T),unique(T,U).
   unique([H|T],[H|U]):- \+member(H,T),unique(T,U).
   ```

3. greatest common divisor:

   ```
   gcd(X,Y,Y):-X == Y.
   gcd(X,Y,Y):- X >= Y, W is X mod Y, W == 0.
   gcd(X,Y,Z):-X >= Y, W is X mod Y, gcd(Y,W,Z).
   gcd(X,Y,Z):-X < Y, gcd(Y,X,Z).
   ```

4. Finding maximum or minimum value:

   ```
   max([X],X).
   max([X|T],X):-max(T,X2),X>=X2.
   max([X|T],X2):-max(T,X2),X<X2.
   ```

5. Sorting: Using the function above:

   ```
   nsort([X],[X]).
   nsort([X|T],[X|T]):-max([X|T],X),nsort(T).
   ```

It is left as an exercise to find which type of sort the above rules simulate.

6. Insertion sort:

```
insertToList(X,[Y|T],[X,Y|T]):- Y>=X.
insertToList(X,[Y|T],[Y,X|T]):- X>Y.
insertionSort([],[]).
insertionSort([X],[X]).
insertionSort([H|T],S):-
  insertionSort(T,ST),
  insertToList(H,ST,S).
```

7. Sorting: Quicksort:

```
quicksort([],[]).
quicksort([X|T],S):-
 partition(T,X,Littles,Bigs),
 quicksort(Littles,L),
 quicksort(Bigs,B),
 append(L,[X|B],S).
partition([],X,[],[]).
partition([H|T],X,[H|L],B):-H=<X,partition(T,X,L,B).
partition([H|T],X,L,[H|B]):-H>X,partition(T,X,L,B).
```

4 Boolean Logic with Prolog

Prolog is quite amenable for logic programming (hence the name PROgramming LOGic). Prolog facts and relations, like logic predicates, are binary valued (true or false). Thus we can define a rule for conjunction as follows:

```
and(A,B):-A,B.
```

However if we query Prolog for what values of A and B the result is true we get the following error:

```
?-and(A,B).
ERROR: and/2: Arguments are not sufficiently instantiated
```

This is since Prolog have no means of knowing that it should try only true and false values for A and B variables. Thus we have to limit their values first:

```
bind(true).
bind(fail).
```

Now we can ask a question as follows:

```
?-bind(A),bind(B),and(A, B).
```

Disjunction can be defined similarly:

```
or(A,B) :-A.
or(A,B) :-B.
```

However we have yet another problem. See the results for the query:

```
?- bind(A),bind(B),or(A,B).
A = true,
B = true ;
A = true,
B = fail ;
A = true,
B = true ;
A = fail,
B = true ;
```

We have one of the cases duplicated. If you trace the program, you will see that Prolog tries the alternative rule for or during its search. To provent this

cut behavior we must use the cut operator, !, to prevent Prolog from looking further. Cut operator prevents rest of the search if the expression before it is successful:

```
or(A,B) :- A,!.
or(A,B) :- B.
```

Prolog will try satisfying A first, and if it succeeds it will next encounter ! operator which prevents it from looking further. Otherwise (true,true) values for A and B will succeed in alternative definitions of 'or' and cause the duplicate results.

Now we will try to produce a truth table for a Boolean expression. To do this we will need a means to write out the table to screen. We can use the

write 'write' predicate to do this:

```
write('Hello').
```

strings Write predicate writes its argument (in this case a Prolog string constant, in single quotes) and returns true. We now use this facility to produce a truth table, using the following definitions:

```
do(A,B,_) :- write(A), write('  '), write(B), write('  '), fail.
do(_,_,Expr) :- Expr, !, write(true), nl.
do(_,_,_) :- write(fail), nl.
table(A,B,Expr) :- bind(A), bind(B), do(A,B,Expr), fail.
```

Not that the first rule for 'do' always fail! Its sole purpose is to have the program write the values of A and B to the screen. Since it fails, Prolog will next try the alternative rules. The second rule for 'do' will write 'true' is expression returns true for the values of A and B, and furthermore uses cut operator, !, to prevent other rules for do being tried. Only if it fails, tha last rule will be tried. Both these rules use 'nl' predicate to write a newline to the screen output. A call to 'table' will produce an output as follows:

```
?- table(A,B,and(A,or(A,B))).
true  true  true
true  fail  true
fail  true  fail
fail  fail  fail
```

Generation of truth tables using more than two variables is left as an exercise.

5 Controlling search

Since Prolog makes a depth-first backtracking search, it suffers from weaknesses of depth-first searches. Let us consider the problem of finding a route for travelling between two cities. We will denote that two cities are connected with a road of certain length as in the following database:

```
road(a,b,10).
road(b,c,5).
road(a,c,10).
road(b,d,15).
road(c,d,10).
road(d,e,15).
```

The graph is shown in Figure 1. The following rules provide definitions to find a route between two places:

```
go(X,Y,[]):-road(X,Y,_);road(Y,X,_).
go(X,Y,[H|T]):-go(X,H,[]),go(H,Y,T).
```

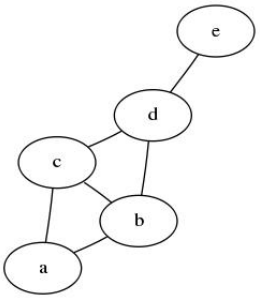

Figure 1: An example graph for route finding problem

The first rule states that if two cities are connected (in either direction since our roads are both way), the route is an empty set, i.e. no intermediate cities should be visited. The second rule states the case where there are intermediate locations to reach from one city to another. When we query the program to find aroute between a and d:

```
?- go(a,d,R).
R = [b] ;
R = [b, c] ;
R = [b, c, d, e] ;
R = [b, c, d, e, d, e] ;
R = [b, c, d, e, d, e, d, e] ;
R = [b, c, d, e, d, e, d, e, d|...]
...
```

we see that the program finds cyclic paths. These are valid but undesired answers. Besides it will not report useful alternatives since it is stuck in, caused by the fact that prolog conducts a depth-first search.

5.1 Searching state spaces

We can use a generalization of state space search by using states, moves, and history of states visited. The goal is to find a sequence of moves so that the problem can be transformed from an initial state to a (the?) goal state.

A naive search can be implemented as follows:

```
solve(State,_,[]):-goal(State).
solve(State,History,[Move|Moves]):-
  canmove(State,Move,State1),
  solve(State1,[State1|History],Moves).
```

provided that definitions of goal() and canmove() are supplied for the particular problem. For example we can apply this approach to route finding problem with the following knowledge base and definitions:

```
road(a,b,10).
road(b,c,5).
road(a,c,10).
road(b,d,15).
road(c,d,10).
road(d,e,15).
canmove(X,[X,Y],Y):-road(X,Y,_);road(Y,X,_).
goal(e).
```

If we now ask for a solution:

```
?-solve(a,[a],Moves).
Moves = [[a, b], [b, c], [c, d], [d, e]] ;
Moves = [[a, b], [b, c], [c, d], [d, e], [e, d], [d, e]] ;
...
```

we can see that the approach does not avoid repeated states.

5.2 Depth-first graph search

To avoid the problem of repeated states we need to make a search which is cautious of cycles, thus avoids revisiting the same note when searching for a solution.The result is a depth-first search avoiding cycles:

```
solve(State,History,[]):-goal(State).
solve(State,History,[Move|Moves]):-
  canmove(State,Move,State1),
  not(member(State1,History)),
  solve(State1,[State1|History],Moves).
```

The main difference of this program is that it checks for an intermediate node not being on the path already visited (i.e. finding only valid moves, given intermediaries), *before* it proceeds to complete the path.

5.3 Breadth-first search

The depth-first search does not yield the shortest path (i.e. number of minimum intermedite stops) to destination in our route finding problem. Furthermore it exploits the fact that Prolog essentially uses a backtracking search, thus avoiding the need to maintain a fringe during the search.

For best-first search, however, one needs to maintain a fringe, and do so in a particular order. As a result one needs to generate all successors of a state. We will use the findall primitive in Prolog to create the successors, as follows:

```
successors([State,History,Moves],Succ):-
   findall(
      [State1,[State1|History],NewMoves],
      (canmove(State,Move,State1),append(Moves,[Move],NewMoves)),
      Succ
   ).
```

Here the built-in append procedure finds the list that is the result of appending its first two arguments.

The findall primitive is among several in Prolog, which are called second-order programming primitives, since in addition to first-order logic, they allow operation over a set of values.

With this definition to create successors, best-first search becomes straightforward:

```
solveBFS([[State,History,Moves]|T],Moves):-goal(State).
solveBFS([[State,History,Moves]|T],OtherMoves):-
  successors([State,History,Moves],Succ),
  append(T,Succ,NewFringe),
  solveBFS(NewFringe,OtherMoves).
```

The above program checks if the first state in the fringe is the solution. If it isn't, it is dropped from the fringe and its successors are added to the fringe, then the solveBFS is called recursively for the new fringe.

6 Uniform cost search

Despite the improvement, the breadth-first search does not necessarily yield the shortest distance road, it just minimizes the number of intermediate locations. Let us first modify our program by making it report total distance in addition to the route. First modification goes to the canmove() definition so that the distance information will be kept as a part of the move:

```
canmove(X,[X,Y,D],Y):-road(X,Y,D);road(Y,X,D).
```

After this we can add some definitions to produce a version of our best-first search that reports the distance, built on top of the previous version:

```
solveBFS_ReportDistance(Fringe,Moves,Distance):-
  solveBFS(Fringe,Moves),
  distance(Moves,Distance).
distance([],0).
distance([[X,Y,D]],D).
distance([[X,Y,D]|T],Distance):-distance(T,D2),Distance is D+D2.
```

Further improvement of the program to find the shortest path first by applying uniform-cost search requires an additional rearrangement of candidates so that the shorter paths will be explored first. Since we have already intevened with the search order to implement breadth-first search, uniform cost search is rather straightforward. However we first need to create rules to sort the fringe with respect to path distances. For this purpose we will use an adaptation of the insertion sort we have seen before:

```
distanceSort([],[]).
distanceSort([S],[S]).
distanceInsert([S1,H1,M1],[[S2,H2,M2]|T],[[S1,H1,M1],[S2,H2,M2]|T]):-
   distance(M1,D1),
   distance(M2,D2),
   D2>=D1.
distanceInsert([S1,H1,M1],[[S2,H2,M2]|T],[[S2,H2,M2],[S1,H1,M1]|T]):-
   distance(M1,D1),
   distance(M2,D2),
   D2<D1.
distanceSort([H|T],S):-
  distanceSort(T,ST),
  distanceInsert(H,ST,S).
```

With this sorting rules in hand, uniform cost is similar to the best first search except with the addition of fringe sorting:

```
solveUCS([[State,History,Moves]|T],Moves):-goal(State).
solveUCS([[State,History,Moves]|T],OtherMoves):-
  successors([State,History,Moves],Succ),
  append(T,Succ,NewFringe),
  distanceSort(NewFringe,SortedNewFringe),
  solveUCS(SortedNewFringe,OtherMoves).
solveUCS_ReportDistance(Fringe,Moves,Distance):-
  solveUCS(Fringe,Moves),
  distance(Moves,Distance).
```

Now we can query our program for solutions:

```
solveUCS_ReportDistance([[a,[a],[]]],Moves,Distance).
```

7 Constraint satisfaction problems

Constraint satisfaction problems(CSPs) differ from problems like route finding we have examined before since we do not speak of 'optimality' in CSP. For example, consider the problem of coloring the map of Turkey. We want to give different colors to neighboring cities, and the map publisher says they can use only 4 colors. It does not matter which color we use for any city except that we respect the given constraints.

As an example we take the problem of coloring cities in the Aegian region. The data for the map consists of which cities are next to one another:

```
nextTo(izmir,balikesir).
nextTo(izmir,manisa).
nextTo(izmir,aydin).
nextTo(manisa,balikesir).
nextTo(manisa,usak).
nextTo(manisa,aydin).
nextTo(manisa,denizli).
nextTo(aydin,denizli).
nextTo(usak,denizli).
color(red).
color(green).
color(blue).
```

Let us first solve the relaxed problem of finding any coloring combinations:

```
instance([],[],Colors).
instance([[City,Color]|T],[City|Cities],Colors):-
   member(Color,Colors),
   instance(T,Cities,Colors).
```

Now a query like follows will return all possible colorings:

```
?-instance(I,[izmir,balikesir],[red,green]).
I = [[izmir, red], [balikesir, red]] ;
I = [[izmir, red], [balikesir, green]] ;
I = [[izmir, green], [balikesir, red]] ;
I = [[izmir, green], [balikesir, green]] ;
```

We can now put constraints to select a valid instance considering neighbourhoods. A sequence of colorings is invalid if a city's neighbour is colored with the same color:

```
validColoring(Coloring,Cities,Colors):-
  instance(Coloring,Cities,Colors),
  not(invalid(Coloring)).
neighbour(X,N):-nextTo(X,N);nextTo(N,X).
invalid([]):-fail.
invalid([[City,Color]|OtherColorings]):-
  invalid(OtherColorings);
  neighbour(City,OtherCity),
  member([OtherCity,Color],OtherColorings).
```

With these definitions Prolog will report all possible valid colorings:

```
?-validColoring(C,[izmir,balikesir,aydin,usak,denizli,manisa],[red,blue,green]
C = [[izmir, red], [balikesir, blue], [aydin, blue], [usak, blue], [denizli, r
C = [[izmir, red], [balikesir, green], [aydin, green], [usak, green], [denizli
C = [[izmir, blue], [balikesir, red], [aydin, red], [usak, red], [denizli, blu
C = [[izmir, blue], [balikesir, green], [aydin, green], [usak, green], [denizl
C = [[izmir, green], [balikesir, red], [aydin, red], [usak, red], [denizli, gr
C = [[izmir, green], [balikesir, blue], [aydin, blue], [usak, blue], [denizli,
```

As different from route finding problem, all solutions are as good as any other, apart from your personal taste for colors.

7.1 Constraint Logic Programming over finite domains

Many CSPs can be solved using the standard capabilities of Prolog as we have seen in the above example. Nevertheless, many Prolog system provide additional facilities to solve CSPs, especially those involving numerical problems. These group of facilities are usually labeled as Constraint Logic Programming (CLP). Examples in this section are only tested for GNU Prolog.

Let's take, for example, the solution to the following equations:

$$X = Y + 1$$

$$Y \geq 3, Y \leq 5$$

It first appears as we can express this in Prolog syntax we have learned so far as follows:

```
?- X=Y+1, Y<5, Y>3.
ERROR: </2: Arguments are not sufficiently instantiated
```

Prolog cannot satisfy the criteria. Unless X or Y are chosen from a set of candidate values arising from backtracking process of Prolog, they are not even known to be numbers!

If we restrict the variables in our problem to integers, i.e. if the values are selected from a finite domain(FD), we can use a common set of facilities available in many Prolog systems. GNU-Prolog provides CLP facilities for FD problems. Following is how we solve the equations in the above example using GNU PRolog FD problem solver:

```
?- X+1 #=#Y, Y#>3, Y#<5.
 X = 3
 Y = 4
```

The sign # is used to indicate whether a variable on either size of operator is a constrained variable to be handled by the FD problem solver instead of backtracking.

There happened to be a single solution for this example problem. If we change the interval constraint for Y as follows:

```
?- X+1 #=#Y, Y#>3, Y#=<5.
X = _#3(3..4)
Y = _#22(4..5)
```

GNU Prolog reports intervals for both variables. However this information is not what we need. In order to have GNU Prolog report specific instances of X and Y that satisfy the criteria, we need to use 'fd_labeling(list of variables)' predicate as follows:

```
?-X+1 #=#Y, Y#>3, Y#=<5, fd_labeling([X,Y]).
  X = 3
  Y = 4 ? ;
  X = 4
  Y = 5
```

A predicate, fd_domain(Varlist,rangestart,rangeend) is available in GNU Prolog to express a range for constrained variables. For example following queries for pairs of integers between 1 and 100, whose addition and multiplication are the same:

```
?- X*Y#=#X+Y, fd_domain([X,Y],1,100), fd_labeling([X,Y]).
```

7.1.1 Example: eight queens problem

Let us now try to approach the classical AI toy problem of placing 8 queens on the chess board of size 8x8. We know that each queen will be placed in a row from 1 to 8, hence we can represent a placement with a list of 8 integers where each value indicates the row on which the queen in the correcponding column is placed.

In order to check if a placement is safe so that no queens attack one another, we must check that they are not on the same row, and not on each other's diagonal:

```
noattack(Q,[],_).
noattack(Q1,[Q2|Rest],Diff):-
  Q1#\=#Q2,
  Diff#\=#Q1-Q2,
  Diff#\=#Q2-Q1,
  Diff2 is Diff-1,
  noattack(Q1,Rest,Diff2).
safe([]).
safe([Q|T]):-noattack(Q,T,1),safe(T).
```

Using the above definitions we can now proceed for a solution:

```
eightqueens(Solution):-
  Solution = [_,_,_,_,_,_,_,_],
  fd_domain(Solution,1,8),
  safe(Solution),
  fd_labeling(Solution).
```

Note that the fd_labeling() comes as the last predicate in the program. The following query reports 152 solutions to the problem:

```
?-eightqueens(S)
```

Solution of the problem for the general case of N queens is left as an exercise to the student.