Prolog Tutorial

Christodoulos Karavasilis

2024

Overview

Prolog is a Turing-complete, *declarative* programming language based on first-order logic. A program in *pure* Prolog is a collection of *Horn* clauses of the form:

$$H \leftarrow B_1, B_2, ..., B_n$$

with H being the head of the rule, and $B_1, .., B_n$ making up the body. Such a rule is interpreted as “If B_1 and B_2 and ... B_n are true, then H is also true”. In this tutorial, we are concerned with a subset of pure Prolog.

Some examples:
Socrates is a man.

```
man(socrates).
```

Note that “man” can be thought of as a unary predicate, and “socrates” as a constant.
Every man is mortal. $(\forall X(man(X) \rightarrow mortal(X)))$

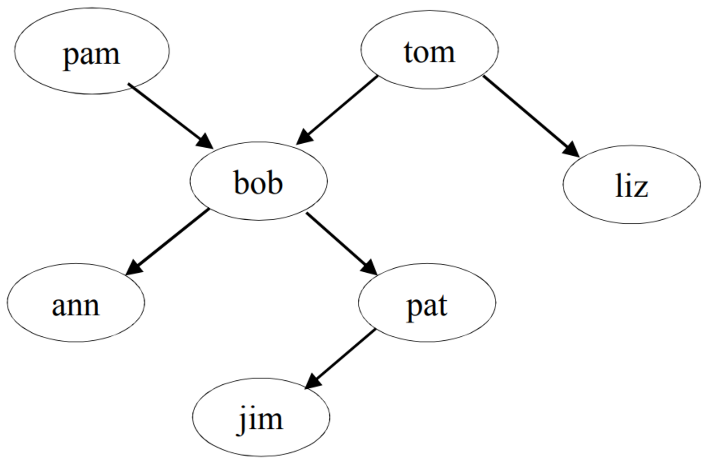
```
mortal(X):-man(X).
```

‘X’ is a variable. The convention in Prolog is that constants start with a lower case, whereas variables start with an upper case. Consider a program with the above two clauses. We can make *queries* to see if something is true.

```
?-man(socrates).
  yes

?-man(zeus).
  no

?-mortal(bob).
  no

?-mortal(X).
  X = socrates
  yes
```

Consider now the parent-child relationships as shown in the figure below. We can encode that information with the following program:

```
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```



Queries:

```
?-parent(bob,pat).
  yes

?-parent(liz, pat).
  no

?-parent(tom, ben).
  no

?-parent(X, liz).
  X = tom
  yes

?-parent(bob, X).
  X = ann -> ;
  X = pat
  yes

?-parent(Y, jim), parent(X, Y).
  X = bob
  Y = pat
  yes

?-parent(X, Y), parent(Y, jim).
  X = bob
  Y = pat
  yes

?-parent(tom, X), parent(X, Y).
  X = bob
  Y = ann -> ;
  X = bob
  Y = pat
  yes

?-parent(X, ann), parent(X, pat).
  X = bob
  yes
```

We can also define a predicate to denote the predecessor relation.

```
predecessor(X,Y):-parent(X,Y).
predecessor(X,Y):-parent(X,Z), predecessor(Z,Y).
```

The first rule can be thought of as the *base case*.

```
?-predecessor(tom,pat)
  yes
```

When trying to answer a query, Prolog parses the program from **top to bottom**, and within the body of a rule, from **left to right**. This is important to remember, in order to choose a correct ordering of our rules. A wrong ordering might even result in an *infinite loop*. For example, consider the following rearrangement of the predecessor rules defined above.

```
predecessor(X,Y):-predecessor(Z,Y), parent(X,Z).
predecessor(X,Y):-parent(X,Y).
```

Queries for the predecessor predicate would never terminate.

The symbol '_' is used to denote an *anonymous variable*. For example we can write:

```
haschild(X):-parent(X,_).
```

as a rule for someone having a child. This can simplify the writing and understanding of programs. Anonymous variables can be used multiple times in a sentence without being bound to the same value.

```
somebody_has_child:-parent(_,_).
```

Structured data. Consider wanting to represent information about a lecture on AI that takes place every Tuesday 10 to 12 by John Doe in the Bahen building, room 123. One way to do this would be the following:

```
course(intro_to_ai,tuesday,10,12,john,doe,bahen,123).
```

Another way to represent it would be:

```
course(intro_to_ai,timeslot(tuesday,10,12),lecturer(john,doe),location(bahen,123)).
```

The second statement represents 'course' as a relation between four items. Using structured data can often improve readability and allows us to write more concise rules that promote modularity.

Lists are a powerful data structure that is often used in Prolog. '[ ]' denotes the empty list. '|' is used to denote the head of the list, followed by the tail of the list, i.e. [Head|Tail]. The following lists are equivalent:

```
[ann, tom, bob, pat]
[ann|[tom, bob, pat]]
[ann|[tom|[bob, pat]]]
[ann|[tom|[bob|[pat]]]]
[ann|[tom|[bob|[pat|[]]]]]
[ann, tom|[bob, pat]]
[ann, tom, bob|[pat]]
[ann, tom, bob, pat|[]]
```

More examples of lists:

```
[a]
[X]
[X, Y]
[X|Y]
[[]]
[2, 3|L]
[p(1, 3)|R]
[[1, 2], [3, 4, 5], [], [6]]
[A, B|[e, f, g]]
[q([2, 7]), r([[4]])]
[[a, b]|[c, d]]
[_|_]
```

Consider the problem of translating word for word. We want to define "translate(Words,Mots)", where Mots is a list of French words that is the translation of the list of English words Words. dict(X,Y) denotes a dictionary that gives us the direct translation of individual words.

```
translate([],[]).
translate([Word|Words],[Mot|Mots]):-dict(Word,Mot), translate(Words,Mots).

dict(the,le).
dict(dog,chien).
dict(chases,chasse).
dict(cat,chat)
```

Query:

```
?-translate([the,dog,chases,the,cat],X).
  X = [le,chien,chasse,le,chat]
  yes
```

The translation can also work the other way around (French to English).

Negation in Prolog is implemented following the *negation as failure* paradigm, a notion closely related to the *closed world assumption*[1]. In other words, if Prolog is unable to prove an assertion, it considers it to be false. One must be careful when using negation, to make sure the resulting behavior is as expected. Negation can be used with the 'not()' or the '\+' operator.
Example:

```
sad(X):-not(happy(X)).
happy(X):-beautiful(X), rich(X).
```

[1]The assumption that every true statement is known to be true.

```
rich(bill).
beautiful(michael).
rich(michael).
beautiful(cinderella).
```

Queries:

```
?-sad(bill).
  Yes
?-sad(cinderella).
  Yes
?- sad(michael).
  No
?-sad(jim).
  Yes
?-sad(Someone).
  No
```

Equality operator. There are a few different notions of equality in logic programming. We focus on the '=' operator which is used or *unification equality*, and succeeds when the two terms can be unified. Examples:

```
?-X = X
  X = X
  Yes

?-X = Y
  Y = X
  Yes

?-X = a
  X = a
  Yes

?-a = a
  Yes

?-a = b
  No
```

Example programs.

1. Let $friends(A, B)$ denote that A and B are friends. How do we express the fact that this is a symmetrical relation? Wrong solution:

   ```
   friends(X,Y):-friends(Y,X).
   ```

   The above would result in an infinite loop. Better solution:

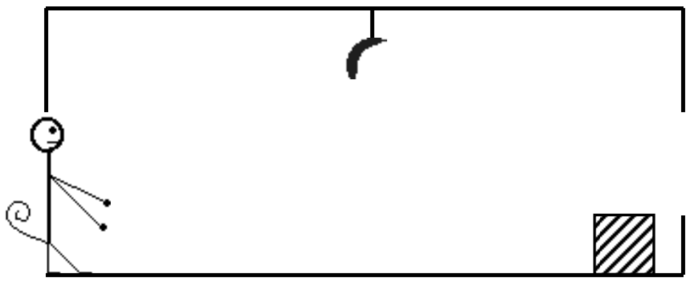
   ```
   friends(bob,tom).
   friends(bob,lilly).

   are_friends(X,Y):-friends(X,Y).
   are_friends(X,Y):-friends(Y,X).
   ```

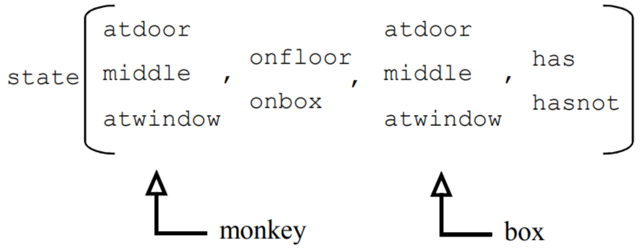
   Defining a new predicate is a common way of avoiding such loops.

2. Let $friend(A, B)$ denote that A and B are friends. A person is popular if they have at least two friends. Define predicate *popular*(X).

```
friends(bob,tom).
friends(bob, lilly ).

are_friends(X,Y):-friends(X,Y).
are_friends(X,Y):-friends(Y,X).

popular(X):-are_friends(X,Y),are_friends(X,Z),not(Y=Z).
```

3. *The Monkey and Banana* problem. A hungry monkey enters a room where there is a banana hanging from the center of the room. To grab the banana, the monkey has to be on the same level and in the middle of the room. There is a box next to a window which the monkey has to use in order to reach the banana. A visual representation of the problem, along with the *state* structure we are using, is shown in the figure below. We want to see if/how the monkey can get the banana.





'%' and '/* */' are used for line-comments and blocks of comments respectively.

```
%If the monkey is on the box in the middle of the room, it can get the banana
move(state(middle, onbox, middle, hasnot),
     grasp,
     state(middle, onbox, middle, has)).

%The monkey can climb on the box, assuming they both are at the same location
move(state(P, onfloor, P, H),
     climb,
     state(P, onbox, P, H)).

%The monkey can push the box from one location to another
move(state(P1, onfloor, P1, H),
     push(P1, P2),
        state(P2, onfloor, P2, H)).

%The monkey can walk to any location
move(state(P1, onfloor, B, H),
        walk(P1, P2),
        state(P2, onfloor, B, H)).

canget(state(_, _, _, has)). %base case
canget(State1):-move(State1, Move, State2),canget(State2).
```

Query:

```
?-canget(state(atdoor, onfloor, atwindow, hasnot)).
  yes
```

What if we want to record the actions required to get to the goal state? We can use a list, and define canget() as follows:

```
canget(state(_, _, _, has), []).
canget(State, [Action|Actions]) :-move(State, Action, NewState),
                                   canget(NewState, Actions).
```

Query:

```
?-canget(state(atdoor, onfloor, atwindow, hasnot), Actions).
  Actions = [walk(atdoor,atwindow),push(atwindow,middle),climb,grasp]
  yes
```