Prolog, part 2

CS 440: Programming Languages and Translators
Lecture 25, Mon 4/29

Review

• Prolog is a programming language based on logic.
  • Its core is as a declarative language: Instead of writing a program, you write out the specification for what a legal output looks like.
  • To execute a query (answer the question "Is there a value that satisfies …(property)…?") Prolog does a backtracking search to find a value that satisfies the query.
  • For efficiency, Prolog does have some imperative features (ordering of the database of facts and rules; left-to-right search to satisfy the requirements of a rule). (Plus cuts, later today.)

• To form predicates, Prolog builds up from primitive predicates $P(a_1, a_2, \ldots, a_n)$, (and $e_1 < e_2$, etc.).
  • A fact is a statement that some predicate is true. E.g., food(pizza).
  • A rule has the form $q :\neg p_1, p_2, \ldots, p_n$, where $q$ is the conclusion and the $p$’s are the antecedents.
    • E.g., mammal(X) :- dog(X). % if dog then mammal; i.e., all dogs are mammals.
  
• In a rule, a variable that appears in the conclusion is universally quantified; a variable that appears only in the antecedents is existentially quantified.
  
  • E.g., path(X, Y) :- link(X, Z), path(Z, Y).
  
  • I.e., for all X and Y, X and Y are connected if there exists a Z that X links to and is connected to Y.

• A database is a list of facts and rules. E.g.,
  
  dog(fido).
dog(rin-tin-tin).
kangaroo(skippy).
mammal(X) :- dog(X).
mammal(X) :- kangaroo(X).

• Given a query such as mammal(X). (Are there any mammals?) Prolog tries to prove that the query is true by searching through the database for applicable facts and rules. It uses syntactic unification to give values to variables. E.g., the query dog(X) unifies with dog(fido) if $X = fido$, so Prolog would present $X = fido$ as an answer to the query. If you ask for another proof, Prolog would present $X = rin-tin-tin$ as a second proof. A third request would come up with false.

• Prolog is not purely declarative: It searches the database top-down, so dog(fido) is found before dog(rin-tin-tin). To prove a rule, Prolog tries to prove the antecedents left-to-right (for $q :\neg p_1, p_2$).
  
  • This second feature means there's a difference between the two rules
    
    path(X,Y) :- path(X,Z), link(Y,Z).

connected(X, Y) :- link(X, Z), connected(Z, Y).

• With the first rule, asking connected(chicago, B) (what locations are connected to Chicago?) leads to looking for a Z where connected(chicago, B), which leads to looking for a Z₁ where connected(chicago, B), etc. This halts when the prover runs out of stack space.

• With the second rule, asking connected(chicago, B) leads to asking link(chicago, Z), which presumably leads to checking a list of basic facts and then looking for a connection between Z and B that is shorter than the one from Chicago to B.

More Examples

% DIRECTED GRAPH PATHS (modified from Learn Prolog Now Section 3.4)

link(1,2). link(3,4). link(5,6). link(7,8). link(9,10). link(12,13).
link(13,14). link(15,16). link(17,18). link(19,20). link(4,1).
link(6,3). link(4,7). link(6,11). link(14,9). link(11,15). link(16,12).
link(14,17). link(16,19).

% link(3,8). % add to get multiple paths from 3 to 8
% link(8,3). % add to get cycle from 3 ->* 8 ->* 3

• Note links are one-way: link(1, 2) doesn’t imply link(2, 1).

% path(X,Y,L): Is L a list of the nodes along a path from X to Y?
% path(X,Y,[X,Y]) :- link(X,Y).
path(X,Y,[X,Z|T]) :- link(X,Z), path(Z,Y,[Z|T]).

% Queries
% path(3,8,_): Is there a path from 3 to 8?
% path(3,8,L): What paths are there from 3 to 8?
% path(3,End,_): Where can you get to, starting from 3?
% path(3,_,P): What paths start with 3?
% path(3,3,_): Is there a cycle from 3 to 3?

• For testing, adding in link(3, 8) enables multiple paths from 3 to 8. Adding link(8, 3) enables a cycle from 3.
• If we want to check simply for the existence of a path, we can either re-use the path predicate or write a new one. A cycle is simply a path from a node to itself.

```prolog
% Check for path endpoints but not actual path
path2(X,Y) :- path(X,Y,__). % calculates path but ignores it

path3(X, Y) :- link(X, Y).
path3(X, Y) :- link(X, Z), path3(Z, Y). % doesn't calculate path

% queries
% path3(3,8)  % is there a path from 3 to 8?
% path3(3,End) % Where can you get to, starting from 3?
% path3(3,3)  % Is there a cycle from 3 to 3?

% cycle(X) -- is there a cycle from X to X?
% cycle(X) :- path3(X, X). % will allow repeated true answers
```

**Avoiding Backtracking Using Cuts**

• Notice with the cycle from node 3 (recall we can get one if we `link(8, 3)`), there are an infinite number of answers depending on how many times we go around the loop. E.g., `path3(3, P)` finds a path `{3, ... 3}`, then `{3, ... 3, ... 3}`, etc. That at least comes up with different answers for each request whereas `path2(3, 3)`, `path3(3, 3)`, and `cycle(3)` just keep coming up with true.

• To avoid this, we need to stop the interpreter from repeatedly backtracking to find another way to prove (e.g.) `cycle(3)`. We can do this by introducing the cut operator ! as part of the cycle rule.

```prolog
% If there's a cycle, there's no need to ask for multiple proofs of it.
% Adding a cut will stop those attempts.
% cycle(X) :- path3(X, X), !. % allows only one occurrence of true
```

• As before, to prove `cycle(X)`, the interpreter tries to prove `path3(X, X)`. If successful, the interpreter notes the ! and continues on to conclude that `cycle(X)` is true. If we ask for another proof of `cycle(X)`, the interpreter will need to backtrack on the proof it found, but when it encounters the cut, it will fail and not continue.

• This cut is an example of a **green cut**, one that doesn't change the set of answers.

• Where the cut is placed is important: With `path4` below, where the cut is between the `link` and recursive `path4` terms, the interpreter finds `path4(3, 3, P)`, \( P = [3, 4, 7, 8, 3] \) as expected (among other paths). If we add the reverse links `link(4,1)`, `link(1,2)`, and `link(2,3)`, then the cut doesn't prevent us from finding \( P = [3, 4, 7, 8, 3, 4, 7, 8, 3] \) (two loops around the cycle) and so on.
% This cut is in the wrong place and doesn't stop the infinite number
% of answers
%
path4(X,Y,[X,Y]) :- link(X,Y).
path4(X,Y,[X,Z|T]) :- link(X,Z), !, path4(Z,Y,[Z|T]).

• In path5 below, we'll find exactly one path from 3 to 3 and stop. In general, path5(X, Y, P) will find one
  instance of P and then stop because of the cut at the end of the rule. This is great for cyclic paths, but not great for
  finding multiple non-cyclic paths.

% This cut causes us to find exactly one path from X to Y.
% This is great for cycles but not so great if you're
% looking for multiple noncyclic paths.
%
path5(X,Y,[X,Y]) :- link(X,Y), !.
path5(X,Y,[X,Z|T]) :- link(X,Z), path5(Z,Y,[Z|T]), !.

• This is an example of a red cut, one that does change the set of answers.