CS224 – Prolog

Purpose: Prolog is a logic programming language in which you write logical rules and make queries given those rules. Prolog does all the logical inference for you.

Knowledge: This activity will help you become familiar with the following content knowledge:

- How to write Prolog relations.
- How the algorithms in Prolog answer queries

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 45 minutes.

1. On phoenix, copy the files `~jillz/cs224/activity12.pl`

   Examine the contents of the file `activity12.pl` which defines two relations `parent` and `ancestor`. Note that constant values are always lower case and variables upper case. We can try out this in the terminal by entering the command `swipl` and then load the files as follows:

   ```
   [activity12].
   ```

   Note that all prolog commands end in a period.

   Try the following queries and notes the results.

   ```
   ancestor(amy,bob).
   ancestor(bob,amy).
   ancestor(X,bob).
   ```

   From this example you should see that Prolog relations can be facts like that amy is the parent of bob, and that someone is their own ancestor.
   Prolog relations can also be rules like that someone is an ancestor of someone else if they are the parent of someone who is an ancestor.

   Add additional relations that convey the following information:

   (a) bob is the parent of john
   (b) bob is also the parent of jill
   (c) complete a rule that indicates when X is the grandparent of Y.

   Test your changes with the query: `grandparent(X,Y)`. (You can type `;` after the first answer to see further answers).

2. Prolog allows lists to be represented by a term such as `[X|Y]`, where `X` is the head of the list and `Y` is the tail. Take a look at the relation `append2` which appends two lists in the activity file. The relation takes three arguments so that `append2(A,B,C)` is true if the list `A` appended to the list `B` results in the list `C`. Load this file in prolog and then try the following query:

   ```
   append2([1,2],[3,4],A).
   ```

   Because this is a relation rather than a function, we can use variables as the first two arguments in queries. Try finding all possible answers to the following query by using the semicolon.

   ```
   append2(X,Y,[1,2]).
   ```

   Now take a look at the relation `reverse1`. In your own words, explain how this relation works.

3. We will now take a deep dive into how Prolog is doing what it is doing.

   A substitution is a function that binds variables to terms. Prolog reports the substitution used to answer the query.

   Two Prolog terms t1 and t2 unify if there is some substitution σ (their unifier) that makes them identical ($\sigma(t1) = \sigma(t2)$).
   For example, f(X,b) and f(a,Y) unify: a unifier is $\{X \rightarrow a,\ Y \rightarrow b\}$

   Is there a unifier for a(X,X,b) and a(c,X,X)?
   Try each of the following in swipl to see which unify and which do not. From these results describe the rules for unification:

   ```
   me = me.
   me = you.
   me = X.
   f(a,X) = f(Y,b).
   f(X) = g(X).
   f(X) = f(a,b).
   f(a,g(X)) = f(Y,b).
   ```

4. Prolog chooses unifiers that do just enough substitution to unify and no more. This is called the "most general unifier" or MGU. What would be the MGU for `parent(X,Y)` and `parent(fred,Y)`?

   Prolog also uses an algorithm call "resolution". Suppose we have a rule (often called a clause) `p(f(Y)) :- q(Y), r(Y).` We will represent this clause as a list of terms `[p(f(Y)),q(Y),r(Y)]`. We also have a query which is our goal [p(X),s(X)]. We can use resolution which tries unification with the clause on the first item in the goal:

   ```
   function resolution(clause,goals):
       let sub = MGU with the head(clause) and head(goals)
       return sub(tail(clause) concatenated with tail(goals))
   ```

   What would be the result of `resolution([p(f(Y)),q(Y),r(Y)], [p(X),s(X)])` ?

5. Unification and resolution are the building blocks of a Prolog interpreter. Here is the algorithm:

   ```
   function solve(goals)
     if goals is empty then success
     else for each clause c in the program, in order
         if head(c) does not unify with head(goals) then do nothing
         else solve(resolution(c, goals))
   ```

   Let's look at a trace with a short program:

   ```
   1. p(f(Y)) :- q(Y), r(Y).
   2. q(g(Z)).
   3. q(h(Z)).
   4. r(h(a)).
   ```

   A partial trace for the query `p(X)` when trying each of the four clauses:

   ```
   solve([p(X)])
     1. solve([q(Y),r(Y)])
         ....
     2. nothing
     3. nothing
     4. nothing
   ```

   What happens next on the call `solve([q(Y),r(Y)])`?

6. The complete trace for `p(X)` looks like:

```
solve([p(X)])
  1. solve([q(Y),r(Y)])
     1. nothing
     2. solve([r(g(Z))])
        1. nothing
        2. nothing
        3. nothing
        4. nothing
     3. solve([r(h(Z))])
        1. nothing
        2. nothing
        3. nothing
        4. solve([])
     4. nothing
  2. nothing
  3. nothing
  4. nothing
```

What happens at the `solve([])` step? We are missing something though. We want to return the substitutions for this query.

These modifications will collect the substitutions:

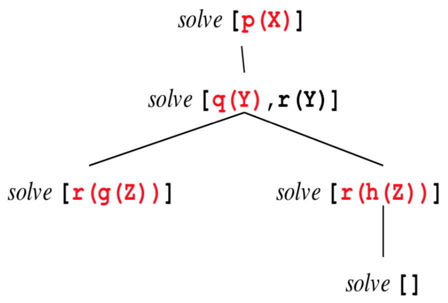
```
function resolution(clause,goals,query):
    let sub = MGU with the head(clause) and head(goals)
    return sub(tail(clause) concatenated with tail(goals),sub(query))

function solve(goals,query)
  if goals is empty then success(query)
  else for each clause c in the program, in order
      if head(c) does not unify with head(goals) then do nothing
      else solve(resolution(c, goals,query))
```

Perform the complete trace for `solve([p(X)],p(X))` to see that it will return `p(f(h(a))`

7. Another very useful way of viewing the Prolog algorithm is through a proof tree. Given our program, we can view the algorithm as this tree:

```
1. p(f(Y)) :- q(Y), r(Y).
2. q(g(Z)).
3. q(h(Z)).
4. r(h(a)).
```

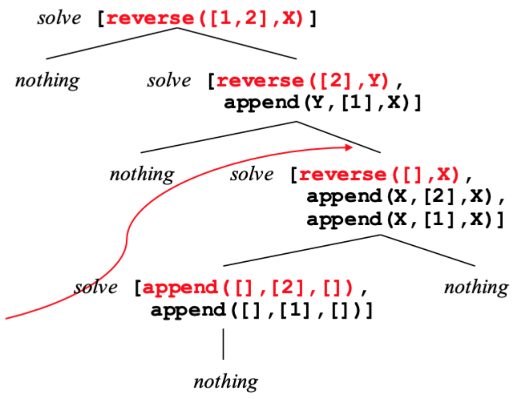


If you look through the earlier trace you will see that Prolog performs a depth first search in the tree.

Consider the program below:

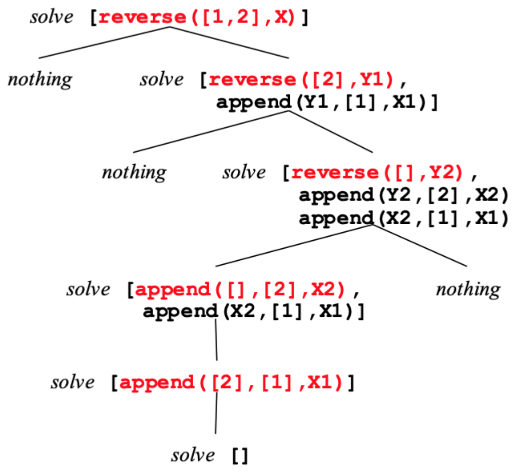
```
reverse([],[]).
reverse([Head|Tail],X) :- reverse(Tail,Y), append(Y,[Head],X).
```

What is the problem with the proof tree and the marked step and how can this problem be fixed?



solve [reverse([1,2],X)]
nothing
solve [reverse([2],Y), append(Y,[1],X)]
nothing
solve [reverse([],X), append(X,[2],X), append(X,[1],X)]
solve [append([],[2],[]), append([],[1],[])]
nothing
nothing

To fix this, Prolog needs to add fresh variables at each clause:


solve [reverse([1,2],X)]
nothing
solve [reverse([2],Y1),
append(Y1,[1],X1)]
nothing
solve [reverse([],Y2),
append(Y2,[2],X2)
append(X2,[1],X1)
solve [append([],[2],X2),
append(X2,[1],X1)]
nothing
solve [append([2],[1],X1)]
solve []