Natural Language Processing With Prolog in the IBM Watson System

March 31, 2011

By Adam Lally (1) and Paul Fodor (2)

(1) IBM Thomas J. Watson Research Center
(2) Stony Brook University

On February 14-16, 2011, the IBM Watson question answering system won the JeopardyMan vs. Machine Challenge by defeating two former grand champions, Ken Jennings and Brad Rutter. To compete successfully at Jeopardy!, Watson had to answer complex natural language questions over an extremely broad domain of knowledge. Moreover, it had to compute an accurate confidence in its answers and to complete its processing in a very short amount of time.

Coming up…

Monday 11/14: HW10 due (Prolog intro)

Tuesday 11/15: HW11 out (Prolog games)

Thursday 11/18: **Midterm distributed**

Monday 11/21: NO HW DUE

Tuesday 11/22: Midterm due

Thursday 11/24: Thanksgiving (no class)

Tuesday 11/29: HW11 due (Prolog games)

CS 42: Prolog Details

Even More Details:
http://www.learnprolognow.org

But first: Python

Does this work? Could we fix it? Should we?

```
def is_list_sorted(t):
    a = t
    a.sort()
    return a == t
```

Which is Better?

```
class Animal(object):
    def speak(self):
        raise NotImplementedError()

class Dog(Animal):
    def speak(self):
        print "woof!"

class Cat(Animal):
    def speak(self):
        print "meow"

def makeSpeak(a):
    a.speak()

d=Dog()
c=Cat()
makeSpeak(d)   # woof!
makeSpeak(c)   # meow
```

```
class Dog(object):
    def speak(self):
        print "woof!"

class Cat(object):
    def speak(self):
        print "meow"

def makeSpeak(a):
    a.speak()

d=Dog()
c=Cat()
makeSpeak(d)   # woof!
makeSpeak(c)   # meow
```

Prolog: The Beautiful Dream

We feed Prolog a bunch of facts relevant to our problem

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avacado).
pairs(walnut, banana).
pairs(apple, banana).
pairs(banana, ginger).
pairs(banana, cloves).
pairs(banana, strawberry).
pairs(banana, coriander).
pairs(strawberry, honey).
pairs(strawberry, ginger).
pairs(strawberry, tea).
pairs(tea, walnut).
pairs(tea, tomato).
pairs(tea, milk).

pairs(X,X).
pairs(X, coconut).
```

We describe how to *recognize* a solution

```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us.

So buy now! **http://www.swi-prolog.org/**

POETS & POETRY: He was a bank clerk in the Yukon before he published "Songs of a Sourdough" in 1907

The output of the parser includes, among many other things, that "published" is a verb with base form (or lemma) "publish", subject "he", and object "Songs of a Sourdough".

Next, Watson applies numerous detection rules that match patterns in the parse. These rules detect elements such as the focus of the question (the words that refer to the answer, in this case "he"), the lexical answer types (terms in the question or category that indicate what type of entity is being asked for, in this case "poet"), and the relationships between entities in either a question or a potential supporting passage.

We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness. The information in the parse is easily converted into Prolog facts, such as (the numbers representing unique identifiers for parse nodes):

```
lemma(1, "he").
partOfSpeech(1,pronoun).
lemma(2, "publish").
partOfSpeech(2,verb).
lemma(3, "Songs of a Sourdough").
partOfSpeech(3,noun).
subject(2,1).
object(2,3).
```

Such facts were consulted into a Prolog system and several rule sets were executed to detect the focus of the question, the lexical answer type and several relations between the elements of the parse. A simplified rule for detecting the authorOf relation can be written in Prolog as follows:

trace. / notrace. / debug. / nodebug.

spy(female).

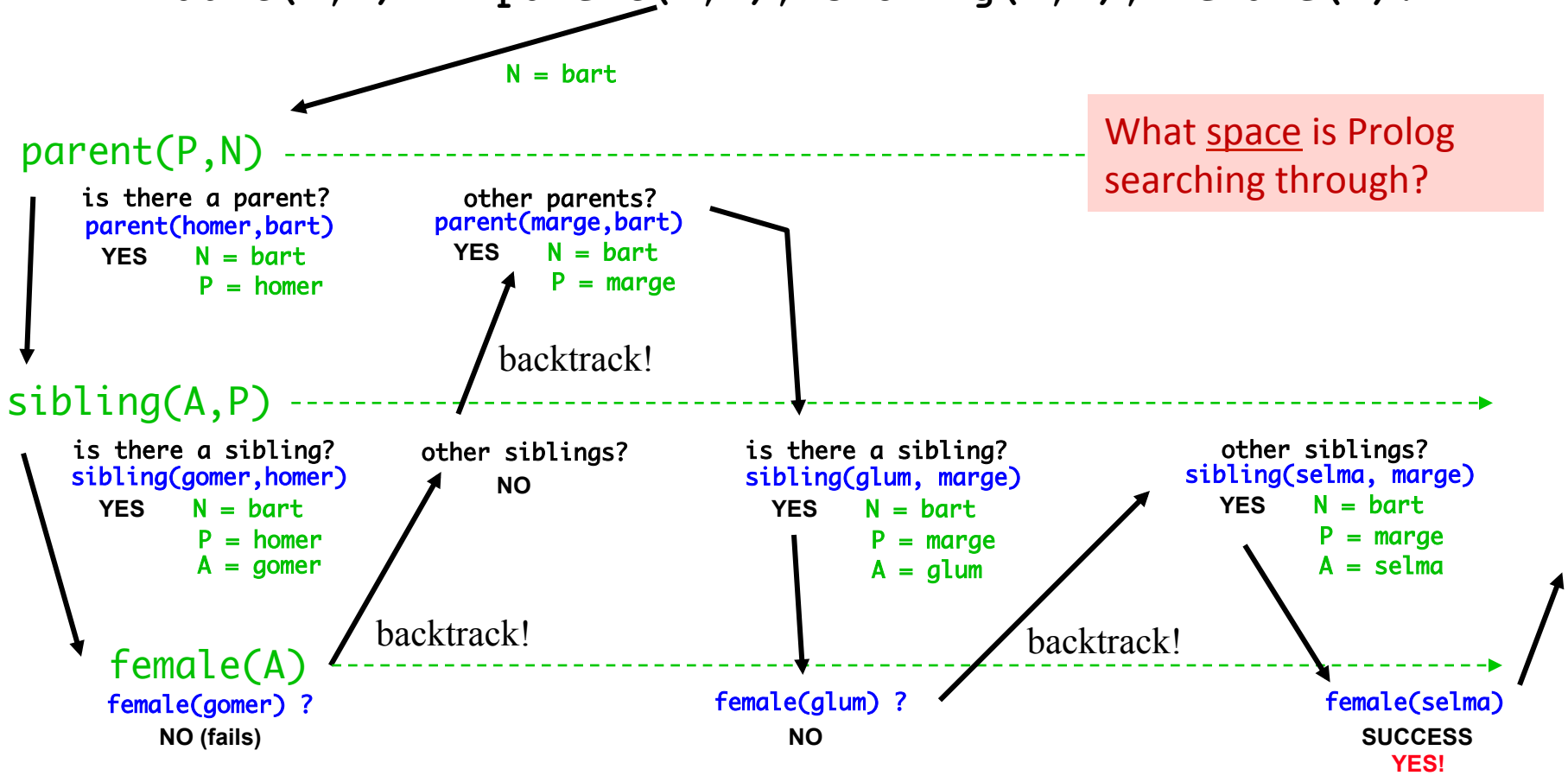
nospyall.

Prolog `is` DFS

Who are Bart's aunts?

aunt(A,bart)

```
aunt(A,N) :- parent(P,N), sibling(A,P), female(A).
```



Prolog Execution

Depth-first search through possible bindings

Given a goal, Prolog tries the each rule of that name...

If a rule has subgoals (a right-hand side),

Subgoals are checked in order, binding variables as needed

Those bindings persist until backtracking undoes them

Watch out for unbound variables… !

Be careful with negation!

*If a goal or subgoal fails, Prolog **backtracks** and tries again with the next available option (another binding or rule).*

trace.

Prolog: The Reality

We feed Prolog a bunch of facts relevant to our problem

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
%% etc.
```

```
pairs(X,X).
pairs(X, coconut).

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

We describe how to *recognize* a solution

```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us?

Prolog: The Reality (2)

We feed Prolog a bunch of facts relevant to our problem

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

pairs(X,X).
pairs(X, coconut).
```

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
%% etc.
```

We describe how to *recognize* a solution

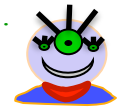
```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us?

when in Prolog…

Equals Aren't!

This is ***bound*** to cause problems…



	positive	negative
Binding Try to make the two sides equal by defining variables	= Success makes them equal	\= Succeeds if they can't be made equal
Structure Are the two sides **already** identical?	== yes	\== no
Math! Tries to make left the same as the **value of** right.	**`is`** the right-hand side MUST have values, not unbound variables	

Prolog knows arithmetic, not algebra!

"Quiz"

Name: _____

For each line, determine if Prolog will find it **`true`** or **`false`**. What bindings will Prolog create?

Binding

`=`

tries to ***assign*** any two structures to one another…

```
[F|R] = [42].          <--
[F|R] = [].            <--
[Root,L,R] = [42,[],[]]  <--
[Root,L,R] = [42,[]]     <--
X = 5+2.               <--
X \= Y.                <--
X=3, Y=2, X \= Y.      <--
```

Structure

`==`

compares any two structures to one another…

```
1+2 == 1+2.            <--
1+2 == 2+1.            <--
[1,X] == [1,X].        <--
[1,2] == [1,X].        <--
X \== Y.               <--
X=3, Y=3, X \== Y.     <--
```

Math

`is`

computes the right-hand side and binds to the left-hand side

```
X is 5+2.              <--
X is Y+3.              <--
Y = 6, X is Y*7.       <--
1+2 is 2+1.            <--
X = 3, Y < X           <--
```

Not all equals are created equal!

For each line, determine if Prolog will find it **`true`** or **`false`**. What bindings will Prolog create?

Binding

`=`

tries to ***assign*** any two structures to one another…

```
[F|R] = [42].          <--  true, binds F to 42 and R to []
[F|R] = [].            <--  false, [] does not have a first!
[Root,L,R] = [42,[],[]]  <--  true, binds Root=42, L=R=[]
[Root,L,R] = [42,[]]     <--  false, different # of elems.
X = 5+2.               <--  true, but X is 5+2, not 7
X \= Y.                <--  false, X CAN unify with Y.
X=3, Y=2, X \= Y.      <--  true, they can not be unified
```

Structure

`==`

compares any two structures to one another…

```
1+2 == 1+2.            <--  true, the same structure
1+2 == 2+1.            <--  false, NOT the same structure!
[1,X] == [1,X].        <--  true
[1,2] == [1,X].        <--  false, no bindings are made
X \== Y.               <--  true, they have different names!
X=3, Y=3, X \== Y.     <--  false, it uses the values!
```

Math

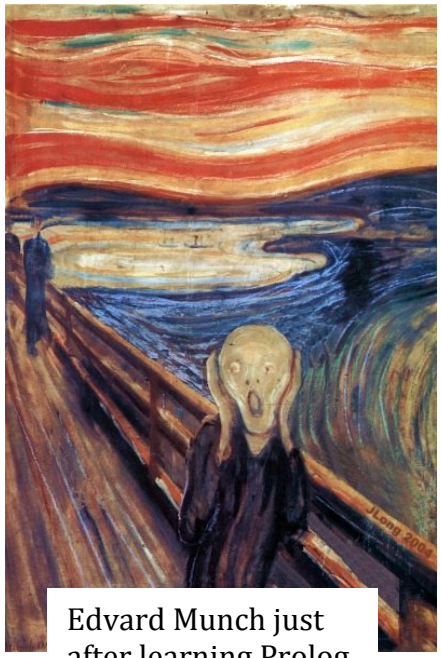
`is`

computes the right-hand side and binds to the left-hand side

```
X is 5+2.              <--  true, now X is 7
X is Y+3.              <--  error: Y is unbound
Y = 6, X is Y*7.       <--  true, X is bound to 42
1+2 is 2+1.            <--  false: only evals the RHS
X = 3, Y < X           <--  error: Y is unbound
```

What's wrong here?

```
sibs(X, Y) :-  X \== Y,
               parent(P,X),
               parent(P,Y).
```



Edvard Munch just after learning Prolog

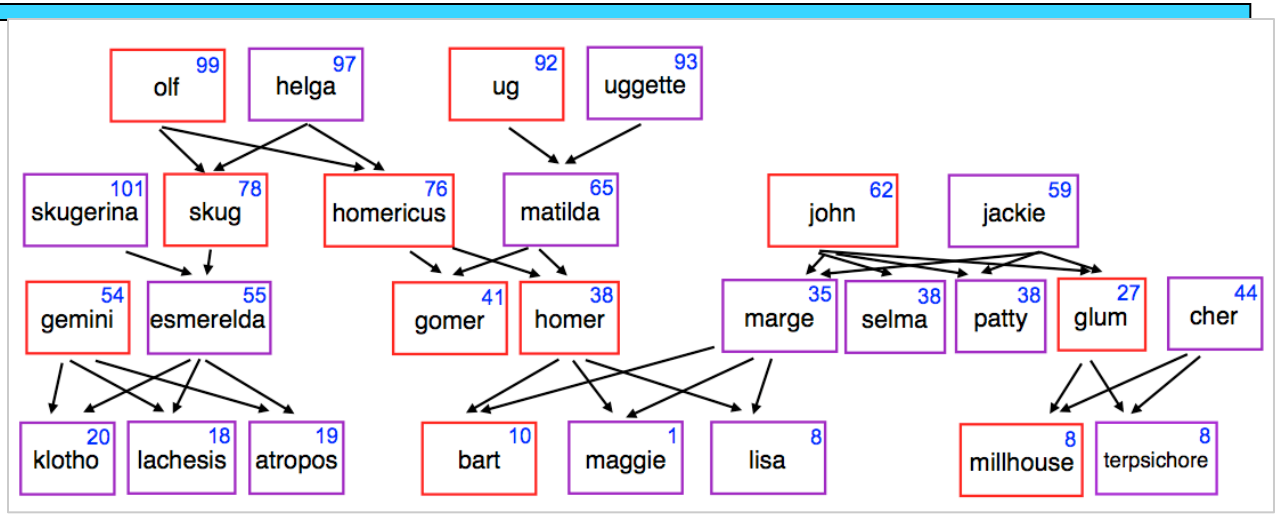
This works only ***when X and Y are bound***!

In Prolog, the programmer needs to keep track of ***bound vs. unbound*** variables!

Watch out! `oldest`

We want `oldest(X)` to answer X = skugerina:

101
skugerina




```
oldest(X) :- AX > AY, age(X,AX), age(Y,AY).
```

What's wrong with each of these alternative definitions?

```
oldest(X) :- age(X,AX), age(Y,AY), AX > AY.
```

Negation in Prolog

```
notoldest(X) :- age(X,AX), age(Y,AY), AX < AY.

oldest(X) :- \+ notoldest(X).
```

Will this work? A. Yes B. No C. only if X is bound

Negation in Prolog

```
notoldest(X) :- age(X,AX), age(Y,AY), AX < AY.

oldest(X) :- person(X), \+ notoldest(X).
```

Will this work? A. Yes B. No C. only if X is bound

Take-home message #2:

Negation does not work unless all of the variables are **BOUND** to values.

Put negative predicates ***last***.

Stay positive!

Math in Prolog?

Racket: ***no side effects*** (assignments changing variables) -- new data is created & returned.

Python: you can ***choose*** to use side effects, or return new data as needed

Prolog: *everything* is done via side effects, there are ***no return values***!

```
fac(X) :- X * fac(X-1).
```

This is wrong in so many ways! What are they?

Math in Prolog?

Racket: ***no side effects*** (assignments changing variables) -- new data is created & returned.

Python: you can ***choose*** to use side effects, or return new data as needed

Prolog: *everything* is done via side effects, there are ***no return values***!

```
fac(0, 1).
fac(N, Ans) :- M is N-1, fac(M, Ans2), Ans is Ans2 * N.
```

Note: Order matters!

HW 10 (part 2): Old friends

Lists, Trees, and Graphs, Prologified

Test

```
length([a,b,c,d],4).

true
```

```
length([a,b,c],4).

false
```

Generate

```
length([1,2,3],N).

N = 3 ;

false
```

```
length(L,0).

L = [] ;

false
```

what if BOTH are variable?

Expressing `if` in prolog

"The length of L is N"

if

L unifies with []

N unifies with 0

and

```
length(L,N) :-  L = [], N = 0.
```

Even simpler version of this:

```
length([],0).
```

Prolog allows pattern-matching within the left-hand side of its rules!

Nonempty matching: *The cons bar!*

```
length([],0).
```

This matches only the empty list.

This matches any NON-empty list –
and names the first F and the rest R!

```
length([F|R],N)  :-
```

This is read "F cons R" . It only binds to nonempty lists - and you can use F and R!

Expressing `if` in prolog

"F cons R"

F = first of L

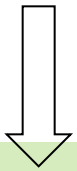
R = rest of L

L is nonempty!

L unifies with [F|R]

"The length of L is N"

if



```
length(L,N)   :-  L = [F|R],
                  length(R,M),
                  N is M+1.
```

and

and

Arithmetic at the <u>end</u>, when M has a value!

Even simpler version of this:

```
length([F|R],N) :- length(R,M),
                   N is M+1.
```

pattern matching is cool!

Prolog's key difference...

Remember, prolog ***does not have return values***!

`member( e, L )` → In Racket/Python, this returns a boolean

`member( E, L )` → In prolog, this can succed or fail.

It can ***bind*** values to E or L !

true when **`E`** is a member of **`L`** and false otherwise

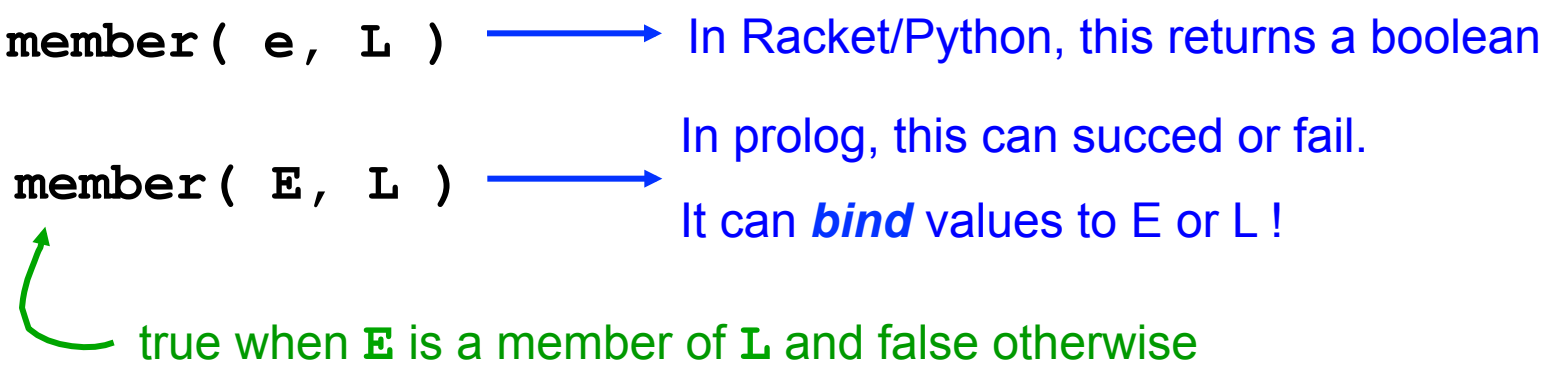
Writing the **`member`** predicate:

```
member( E,          )

member( E,          )                    .
```

Prolog's key difference...

Remember, prolog ***does not have return values***!



Writing the **member** predicate:

```
member( E, [E|R] ).

member( E, [F|R] ) :- mem(E,R).
```

_ *"Don't cares"* _

How does prolog say "I might think about caring in the indefinite future?"

Prolog let's you say "I don't care!"

```
mem( E, [E|R] ).
mem( E, [F|R] ) :- mem(E, R).
```

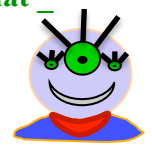
We never used R …
it's a ***singleton***.

We never used F either!

The underscore _ is a place holder:

```
mem( E, [E|_] ).
mem( E, [_|R] ) :- mem(E, R).
```

Let me underscore that _ has no value!



Lists and trees, prolog style…

(length and append are built-in, but we'll write them here, too)

length(L, N)

length([], 0).

length([F|R], N) :- length(R,M), N is M+1.

---

append(L, M, Both)

base case **append(**

rec case **append(**

BST = [42, [10,[],[]], [60,[],[]]], for example

Short. Sweet!

nnodes(BST, N)

base case **nnodes(**

rec case **nnodes([Root,L,R], N) :-**

Careful… Bound vs. Unbound

```
len([], 0).

len([_|R], N) :- len(R,N2), N2 is N-1.
```

When will Prolog not like this definition?

When N is initially unbound.

len([1,2,3], 3). %works

len([1,2,3], N). % error when evauating N-1.

Quiz

lastof(E, L) true if Eis the last element of the list L

```
lastof( E,  [E]  ) .
lastof( E,[_|R])  :- lastof( E,  R ).
```

---

reverse(L, Rev) true if Rev is the reverse of the list L

```
reverse( [], [] ).
reverse( [F|R], Y) :-  reverse(R,Z), append(Z, [F], Y).
```

---

treefind(E, BST) true if E is a node found in BST (a bin. search tree)

```
treefind( E, [E, _, _] ).
treefind( E, [Root, Left, _] ) :- E < Root, treefind(E, Left).
treefind( E, [Root, _, Right] ) :- E > Root, treefind(E, Right).
```

Extra! kid(Par, Graph, K) true if K is a child of Par in graph Graph (list of edges)

```
kid( a, [ [a,b], [b,c], [a,c] ], K ).
K = b  ;
K = c  ;
No
```

```
kid(Par,Graph,K)  :-  member([Par,K], Graph).
```

No ***kid***ding!

HW 10 (part 1): Prolog as family…

grandparent(GP, GK)

Lines of code?

cousins(C1, C2)

hasYS(X) true iff X has a younger sister…

hasFirstGC(Parent) true iff Parent has child who is the oldest grandchild of one of Parent's parents!

HW 10(part 2): Prolog & friends…

removeOne(E, L, NewL) **NewL is L with an E removed.**

< 15 lines
TOTAL

count(E, L, N) **There are exactly N Es in L.**

find(Pattern, Target, Index) **The list Pattern is found in the list Target at location Index.**

depth(BST, Depth) **The binary tree BST has a depth (or height) of Depth.**

insert(E, BST, NewBST) **NewBST is BST with E appropriately inserted.**

I think we've been down this PATH before?!!

path(A, B, Graph, Path) **Path is a path from A to B in Graph, a list of [src,dst] edges.**

Prolog: Order matters!

```
count( E, [], 0 ).
count( E, [E|R], N ) :- count( E, R, M ), N is M+1.
count( E, [F|R], N ) :- E \== F, count( E, R, N ).
```

What will the above predicate do on the following query:

```
?- count( E, [spam, oh, spam], N ).
```

A. Fail completely (i.e., answer "false")

B. Work correctly

C. Bind E and N to some correct values, but fail to bind to other correct values

D. Bind E or N to at least one incorrect value

E. Error

Prolog: Order matters!

```
count( E, [], 0 ).
count( E, [E|R], N ) :- count( E, R, M ), N is M+1.
count( E, [F|R], N ) :- E \== F, count( E, R, N ).
```

What will the above predicate do on the following query:

```
?- count( E, [spam, oh, spam], N ).
```

A. Bind E and N to some correct values, but fail to bind to other correct values

E = spam, N = 2 ;

E = oh, N = 1 ;

E = spam, N = 1