10/28/16

-Facts
-Rules
-Query

Suppose my file had only

fact

And in Prolog, I type the following. I will get the following output

?- fact
true
?- something
false

Now, suppose I have the following facts

prereq(cs31,cs32).
prereq(cs32,cs33).
prereq(cs32,cs111).
prereq(cs32,cs118).
prereq(cs33,cs118).
prereq(cs32,cs131).
prereq(cs33,cs131).
prereq(cs32,cs132).
prereq(cs131,cs132).

?-prereq(cs31,cs32)
true
?-prereq(cs31,cs33)
false

There is no real functionality with “prereq”, it is simply fact-checking
Note:
1) Facts start with a lowercase letter
2) Variables start with an uppercase letter
3) Each statement ends with a period (syntax)

What happens here?

?- prereq(X, cs131).

```
      X = cs32
      X = cs33

?- prereq(X, cs131), prereq(X, cs33).
      X = cs32

?- prereq(X, cs131, prereq(Y, cs33).
      X = cs32
      Y = cs32
```

We can also define rules which are different from facts.

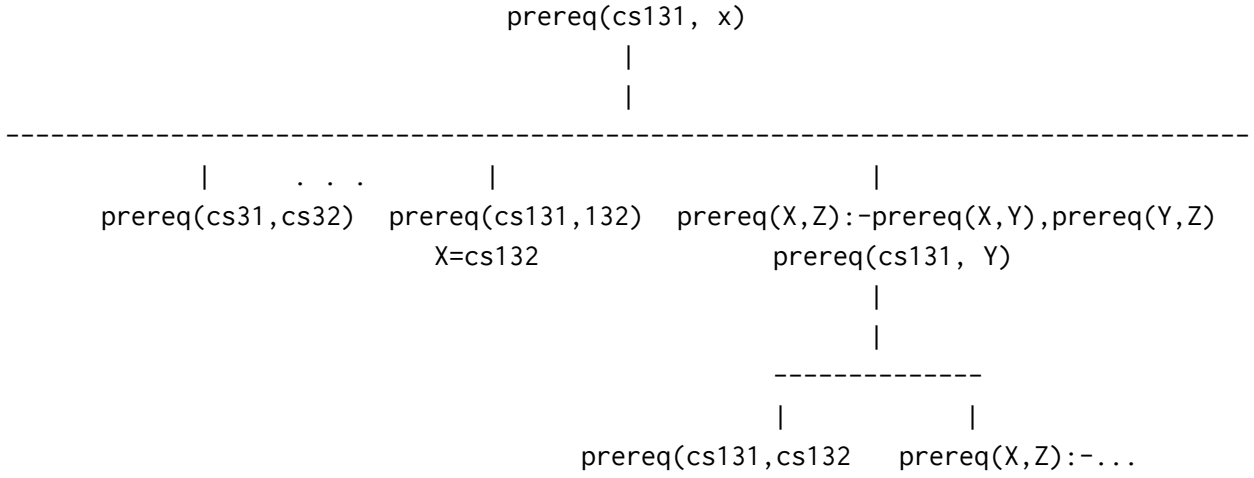
Say, I want the desired functionality such that ancestor(cs31, cs131) returns true.

```
ancestor(X,Z):-prereq(X,Z).
ancestor(X,Z):-prereq(X,Y), ancestor(Y,Z).
```

Can I do this?

prereq(X,Z):-prereq(X,Z), prereq(Y,Z). (stack overflow)

Proof tree:



Math in Prolog:

```
X is 5 + 1
6 > 5
5 < 6
3 =\= 2
```

mylength(L,N).

mylength([1,2,3],3) - true
mylength([1,2], 3) - false

[H | T] (representation of list)

Solution:

```
mylength([], 0).
mylength([H | T], N):-mylength(T,X), N is X + 1.
```

mylength([1,2,3], X)
X = 3

In the reverse direction,

mylength(X, 3)
X = [_1000, _100+, _1008] (some output like this, look up the syntax)

Now, say I want a member function

member(2, [1,2,3]) true
member(4, [1,2,3]) false

```
member(H, [H | _]).
member(X, [_, T]):-member(X, T)
```

member(X, [1,2,3])
X = 1
X = 2
X = 3

Another function:

app([1,2], [3,4], [1,2,3,4]) - true

How do we write this?

```
app([], L, L).
app([H|T], L2, [H|L3]):-app(T,L2, L3)
```

Now,
app([1,2], [3,4], X)
X = [1, 2, 3, 4]

```
app([1,2], X, [1,2,3,4])
      X = [3,4]
```

Let's do reverse

```
rev([1,2,3], [3,2,1])

rev([], []).
rev([H|T], R):-rev(T,RT), append(RT,[H],R)
```

^-- O(n^2)

A second try:

```
acc([H|T], A, R):-acc(T,[H|A], R).
acc([], A, A].
rev2(L,R):-acc(L,[],R)
```

A hard problem: The N-Queens problem (with an 8x8 chessboard)

How do we do this in Prolog?

```
nqueens([]).
nqueens([queen(X,Y)|T]):-nqueens(T),
                          member(X, [1,2,3,4,5,6,7,8]),
                          member(Y, [1,2,3,4,5,6,7,8]),
                          noattack(queen(X,Y),T).

noattack(_, []).
noattack(queen(X,Y), [(queen(X1,Y1)|T]:-
                                        X=\=X1,
                                        Y=\=Y1,
                                        X-X1=\=Y-Y1
                                        X1-X=\=Y-Y1,
                                        noattack(queen(X,Y),T).
```

Now, we can do

```
nqueens([A,B,C,D,E,F,G])
```

And get a list of coordinates

Very slow language (can result in a huge recursion tree), but powerful it what it can solve.

Variations of the Farmer, Wolf, Goat, and Cabbage problem

```
move(Start,Item,End)

move([X,X,G,C], wolf[Y,Y,G,C]):-swap(X,Y).
max([X,W,X,C], goat[Y,W,Y,C]):-swap(X,Y).
move([X,W,G,X], cabbage[Y,W,G,Y]:-swap(X,Y).
move(X,W,G,[]], nothing, [Y,W,G,[]]:-swap(X,Y).

oneEq(X,X,_).
oneEq(X,_X).

safe([F,W,G,C]):-
             oneEq(F,G,W),
             oneEq(F,G,C).
```

Finally,

```
solution(Start[H|T]):-
             move(Start,H,Next),
             safe(Next),
             solution(Next,T).
```

There are some constructs that prune solutions to make a function run faster. f