Let's Write Some Code

Write these predicates:

1. uncle
2. aunt
3. nephew
4. niece
5. grandparent
6. ancestor

Transitive Relations

```
parent(sally,jane).      parent(bob,jane).
parent(sally,john).      parent(bob,john).
parent(mary,sally).      parent(al,sally).
parent(ann,bob).         parent(mike,bob).
parent(jean,al).         parent(joe,al).
parent(ruth,mary).       parent(jim,mary).
parent(esther,ruth).     parent(mick,ruth).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

?- grandparent(Y,jane).
Y = mary ;
Y = al ;
Y = ann ;
Y = mike ;
No
?- ancestor(X,jane).
X = sally ;
X = bob ;
X = mary ;
X = al ;
X = ann ;
X = mike ;
X = jean ;
X = joe ;
X = ruth ;
X = jim ;
X = esther ;
X = mick ;
No
```


Steps to a Recursive Predicate

1. **Predicate Form:**
   - Choose a predicate name appropriate for something that is true or false.
   - Choose mnemonic argument names.
2. **Spec:** Write the specification in this form:
   pred succeeds iff ...
3. **Base Cases:**
   - When is it so easy to tell the predicate is true that you needn't check any further?
   - Write these base case(s).
4. **Recursive Cases:**
   - When it's not trivial, what do you need to know is true before you can be sure the predicate is true?
   - This is the antecedent of your rule.
   - There may be several non-trivial cases, each needing a rule.

Lists in Prolog

Two ways to describe a list:

1. [*elements-with-commas*]

   Egs: `[a, b, c]`
   `[]`
   `[a, [b, c], d, [], e]`
   `[a, X, c, d]`

2. [*first* | *rest*] (*rest* must be a list)

   Egs: `[a | [b, c]]`
   `[a | Rest]`

Question: Why use the second form with |?

Unifying Lists

```
?- [X, Y, Z] = [john, likes, fish].


?- [cat] = [X|Y].


?- [1,2] = [X|Y].


?- [a,b,c] = [X|Y].


?- [a,b|Z]=[X|Y].


?- [X,abc,Y]=[X,abc|Y].


?- [[the|Y]|Z] = [[X,hare] | [is,here]].
```


Let's Write Some List Predicates

1. member(X, List).
2. append(List1, List2, Result).
3. swapFirstTwo(List1, List2).
4. length(List).

List Membership

Definition of `member`...

```
?- member(a,[a,b]).
Yes
?- member(a,[b,c]).
No
?- member(X,[a,b,c]).
X=a ;
X=b ;
X=c ;
No
?- member(a,[c,b,X]).
X=a ;
No
?- member(X,Y).
X=_G72, Y=[_G72|_G73] ;
X=_G74, Y=[_G72,_G74|_G75] ;
X=_G76, Y=[_G72,_G74,_G76|_G77] ;
...
```

Lazy evaluation of potentially infinite data structures

Append - More than "appending"

Definition of `append`

Build a list:

?- append([a],[b],Y).
Y=[a,b]
Yes

Break a list up:

?- append(X,[b],[a,b]).
X=[a]
Yes
?- append([a],Y,[a,b]).
Y=[b]
Yes

Append (cont.)

?- append(X,Y,[a,b]).
X=[],Y=[a,b] ;
X=[a],Y=[b] ;
X=[a,b],Y=[] ;
No

Generate lists:

?- append(X,[b],Z).
X=[],Z=[b] ;
X=[_G98],Z=[_G98,b] ;
X=[_G98,_G102],Z=[_G98,_G102,b] ;
. . .

Computing the Length of a List

Definition of `length`...

```
?- length([a,b,c],L).
L = 3

?- length([],L).
L = 0

?- length(X,3).
X = [_66,_68,_70]

?- length(X,0).
X = []
```

NOTE: Use built-in length function!!

Accessing More Than One Initial Element

Definition of `swap_first_two`...

```
?- swap_first_two([a,b], [b,a]).
Yes
?- swap_first_two([a,b], [b,c]).
No
?- swap_first_two([a,b,c], [b,a,c]).
Yes
?- swap_first_two([a,b,c], [b,a,d]).
No
?- swap_first_two([a,b,c], X).
X = [b,a,c];
No
?- swap_first_two([a,b|Y], X).
Y = _56, X = [b,a|_56];
No
?- swap_first_two([],X).
No
?- swap_first_two([a],X).
No
?- swap_first_two([a,b],X).
X = [b,a];
No
```


Lists of a Specified Length

Definition of `list_of_elem`...

```
?- list_elem(X,b,3).
X = [b,b,b];
ERROR:  Out of global stack
?- list_of_elem(X,Y,2).
X = [_50,_50]
Y = _50;
ERROR:  Out of global stack
```

Lists of a Specified Length

New definition of `list_of_elem`...

```
?- working_list_elem(X,b,3).
X = [b,b,b];
No

?- working_list_elem(X,Y,2).
X = [_50,_50]
Y = _50;
No
```