Logic Programming

Lecture 3: Recursion, lists, and data structures

Outline for today

- Recursion
  - proof search behavior
  - practical concerns
- List processing
- Programming with terms as data structures

Recursion

- So far we've (mostly) used *nonrecursive* rules
- These are limited:
  - not Turing-complete
  - can't define *transitive closure*
    - e.g. `ancestor`

Recursion (1)

- Nothing to it?

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

- Just use recursively defined predicate as a goal.
- Easy, right?

Depth first search, revisited

- Prolog tries rules **depth-first** in program order
  - **no matter what**
  - Even if there is an "obvious" solution using later clauses!

```
p :- p.
p.
```

- will **always** loop on first rule.

Recursion (2)

- Rule order can matter.

```
ancestor2(X,Y) :- parent(X,Z),
                  ancestor2(Z,Y).
ancestor2(X,Y) :- parent(X,Y).
```

This may be less efficient (tries to find **longest** path first)

This may also **loop** unnecessarily (if parent were cyclic).

Heuristic: write base case rules first.

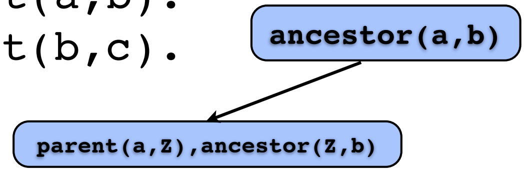
Rule order matters

```
parent(a,b).
parent(b,c).
```

ancestor(a,b)

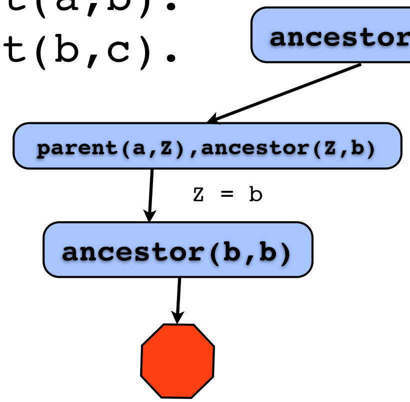
Rule order matters

```
parent(a,b).
parent(b,c).
```



Rule order matters

```
parent(a,b).
parent(b,c).
```



ancestor(a,b)

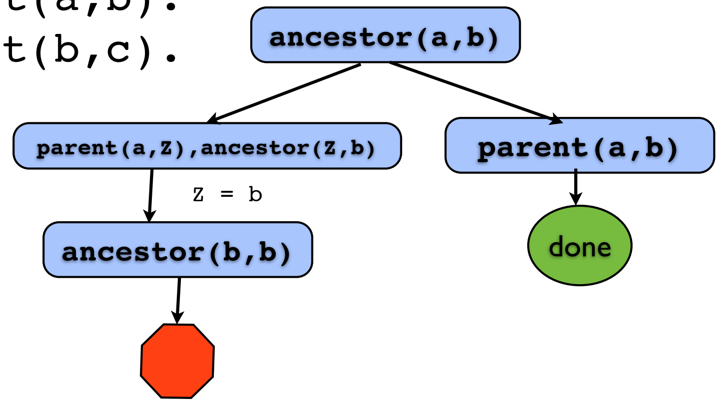
parent(a,Z),ancestor(Z,b)

Z = b

ancestor(b,b)

Rule order matters

```
parent(a,b).
parent(b,c).
```



ancestor(a,b)

parent(a,Z),ancestor(Z,b)

Z = b

ancestor(b,b)

parent(a,b)

done

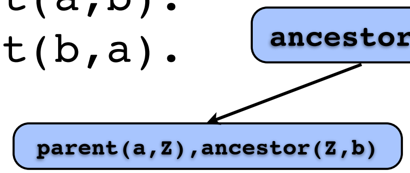
Rule order matters

```
parent(a,b).
parent(b,a).
```

ancestor(a,b)

Rule order matters

```
parent(a,b).
parent(b,a).
```

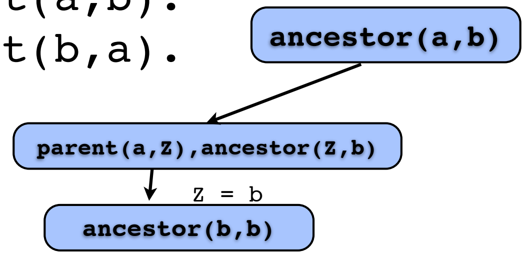


ancestor(a,b)

parent(a,Z),ancestor(Z,b)

Rule order matters

```
parent(a,b).
parent(b,a).
```



ancestor(a,b)

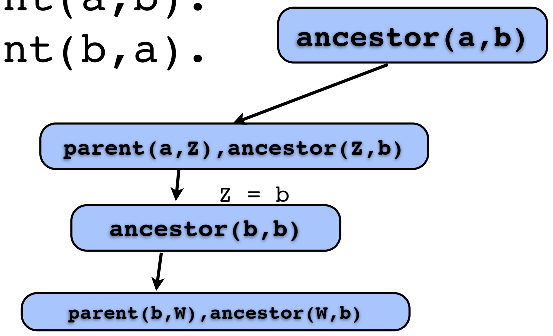
parent(a,Z),ancestor(Z,b)

Z = b

ancestor(b,b)

Rule order matters

```
parent(a,b).
parent(b,a).
```



ancestor(a,b)

parent(a,Z),ancestor(Z,b)

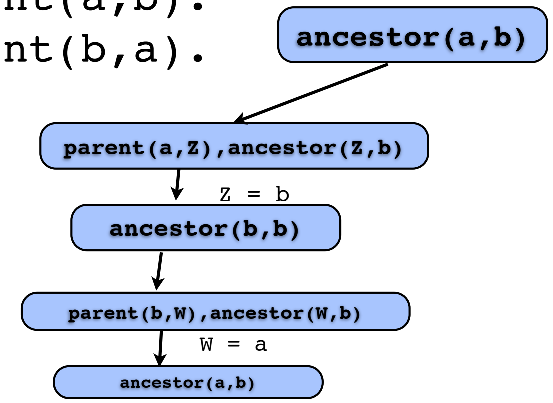
Z = b

ancestor(b,b)

parent(b,W),ancestor(W,b)

Rule order matters

```
parent(a,b).
parent(b,a).
```



ancestor(a,b)

parent(a,Z),ancestor(Z,b)

Z = b

ancestor(b,b)

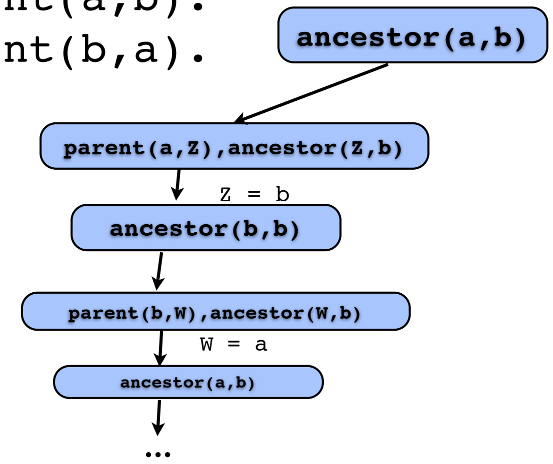
parent(b,W),ancestor(W,b)

W = a

ancestor(a,b)

Rule order matters

```
parent(a,b).
parent(b,a).
```



ancestor(a,b)

parent(a,Z),ancestor(Z,b)

Z = b

ancestor(b,b)

parent(b,W),ancestor(W,b)

W = a

ancestor(a,b)

...

Recursion (3)

- Goal order can matter.

```
ancestor3(X,Y) :- parent(X,Y).
ancestor3(X,Y) :- ancestor3(Z,Y),
                  parent(X,Z).
```

This will list all solutions, then loop.

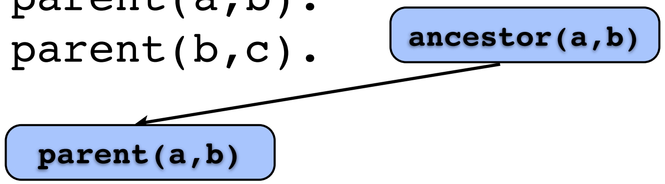
Goal order matters

```
parent(a,b).
parent(b,c).
```

`ancestor(a,b)`

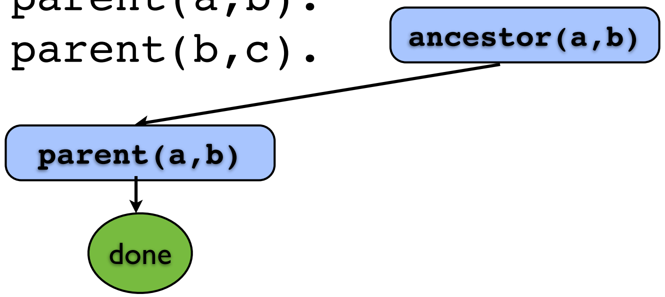
Goal order matters

```
parent(a,b).
parent(b,c).
```



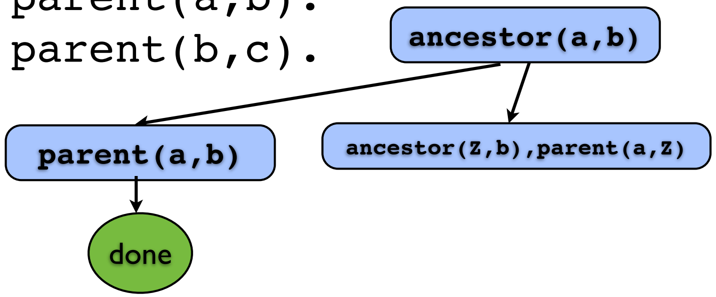
Goal order matters

```
parent(a,b).
parent(b,c).
```



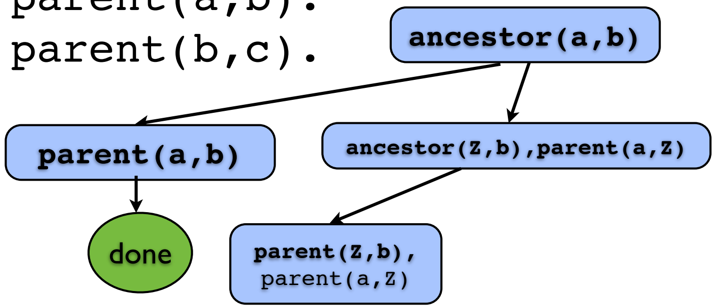
Goal order matters

```
parent(a,b).
parent(b,c).
```



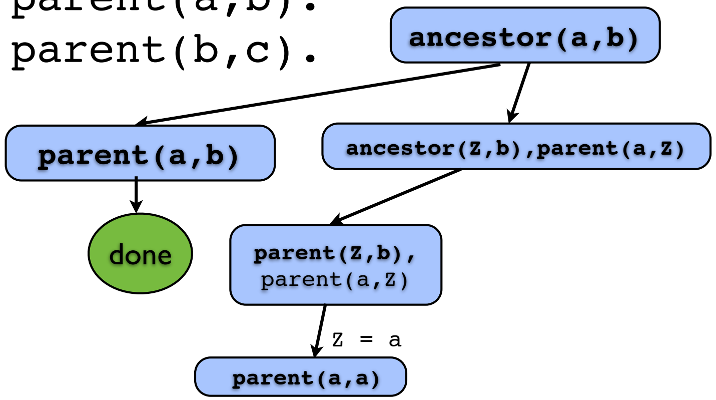
Goal order matters

```
parent(a,b).
parent(b,c).
```



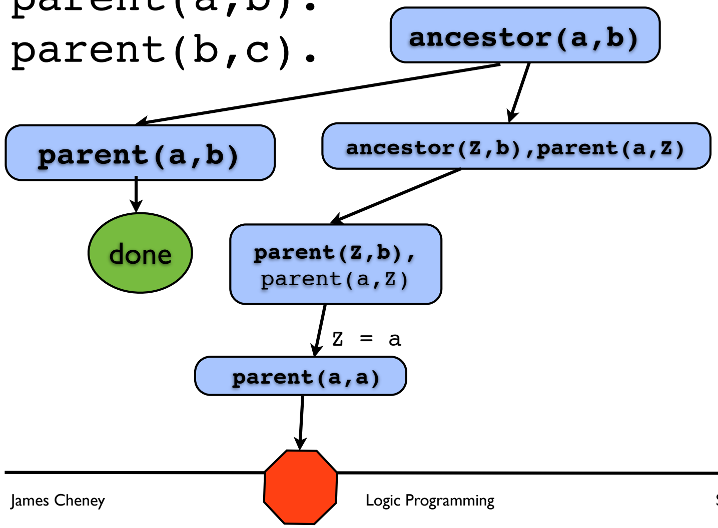
Goal order matters

```
parent(a,b).
parent(b,c).
```



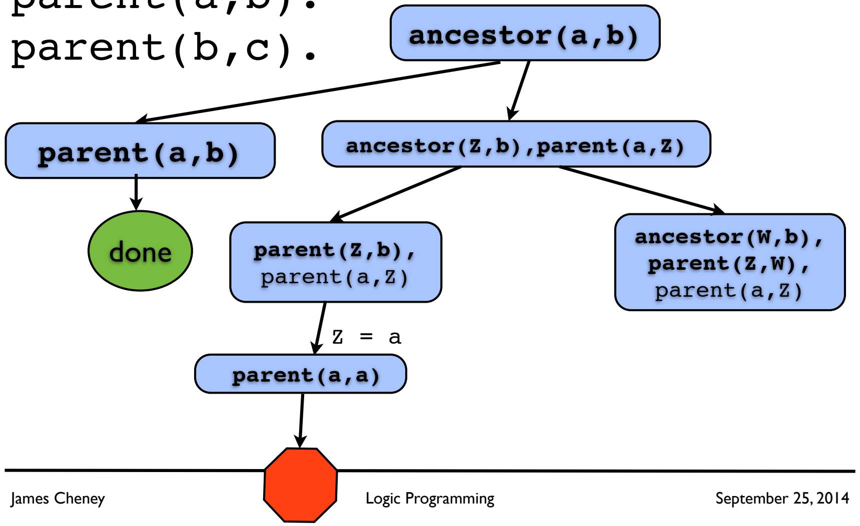
Goal order matters

```
parent(a,b).
parent(b,c).
```



Goal order matters

```
parent(a,b).
parent(b,c).
```



ancestor(a,b)

parent(a,b)

done

ancestor(Z,b),parent(a,Z)

parent(Z,b), parent(a,Z)

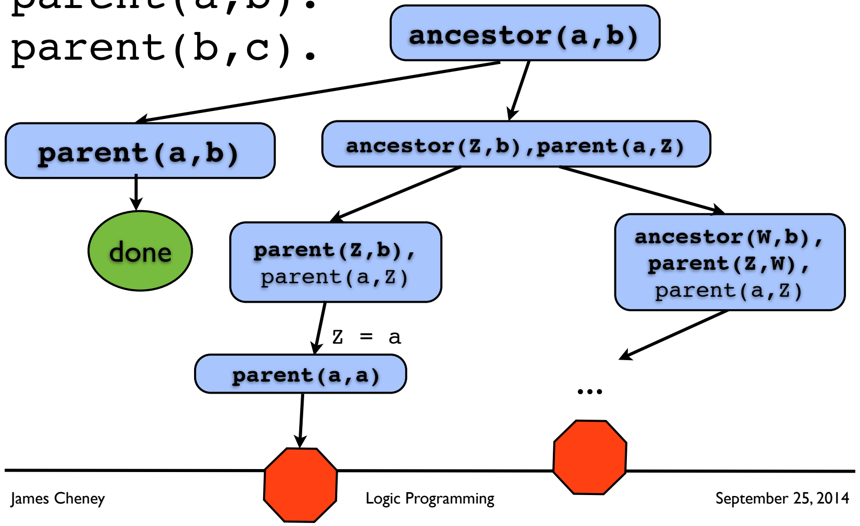
ancestor(W,b), parent(Z,W), parent(a,Z)

Z = a

parent(a,a)

Goal order matters

```
parent(a,b).
parent(b,c).
```



ancestor(a,b)

parent(a,b)

done

ancestor(Z,b),parent(a,Z)

parent(Z,b), parent(a,Z)

ancestor(W,b), parent(Z,W), parent(a,Z)

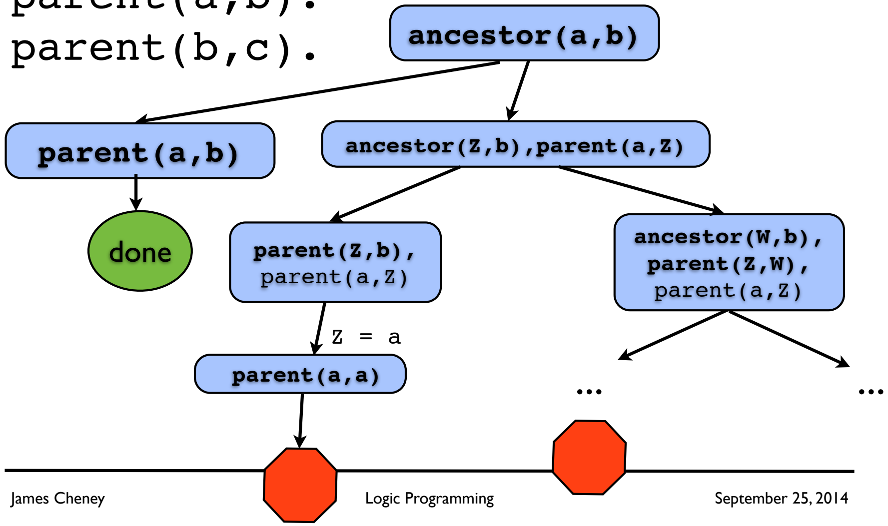
Z = a

parent(a,a)

...

Goal order matters

```
parent(a,b).
parent(b,c).
```



ancestor(a,b)

parent(a,b)

done

ancestor(Z,b),parent(a,Z)

parent(Z,b), parent(a,Z)

ancestor(W,b), parent(Z,W), parent(a,Z)

Z = a

parent(a,a)

...

...

Recursion (4)

- Goal order can matter.

```
ancestor4(X,Y) :- ancestor4(Z,Y),
                  parent(X,Z).
ancestor4(X,Y) :- parent(X,Y).
```

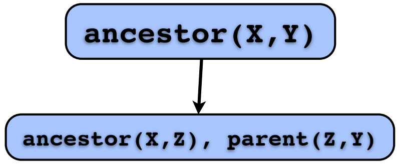
This will **always** loop!

Heuristic: try non-recursive goals first.

Goal order matters

ancestor(X,Y)

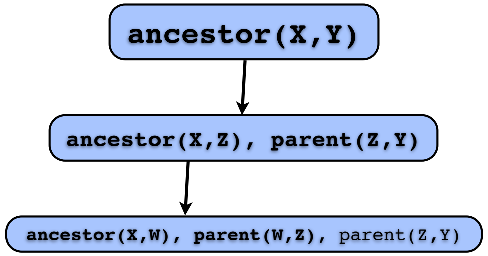
Goal order matters



ancestor(X,Y)

ancestor(X,Z), parent(Z,Y)

Goal order matters

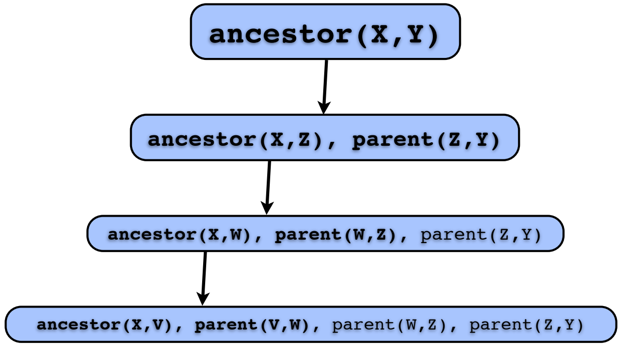


ancestor(X,Y)

ancestor(X,Z), parent(Z,Y)

ancestor(X,W), parent(W,Z), parent(Z,Y)

Goal order matters



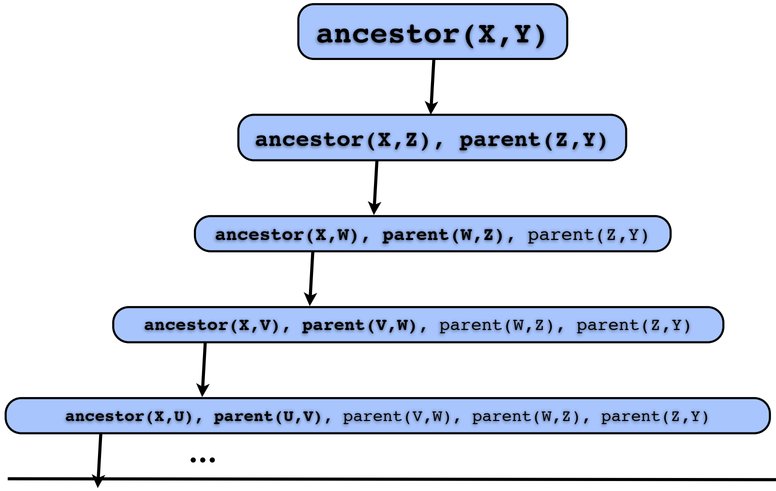
ancestor(X,Y)

ancestor(X,Z), parent(Z,Y)

ancestor(X,W), parent(W,Z), parent(Z,Y)

ancestor(X,V), parent(V,W), parent(W,Z), parent(Z,Y)

Goal order matters



`ancestor(X,Y)`

`ancestor(X,Z), parent(Z,Y)`

`ancestor(X,W), parent(W,Z), parent(Z,Y)`

`ancestor(X,V), parent(V,W), parent(W,Z), parent(Z,Y)`

`ancestor(X,U), parent(U,V), parent(V,W), parent(W,Z), parent(Z,Y)`

...

Recursion and terms

- Terms can be **arbitrarily nested**
- **Example:** unary natural numbers

```
nat(z).
nat(s(N)) :- nat(N).
```

- To do interesting things we need recursion

Addition and subtraction

- **Example:** addition

```
add(z,N,N).
add(s(N),M,s(P)) :- add(N,M,P).
```

- Can run in reverse to find all M,N with M+N=P
- Can use to define `leq`

```
leq(M,N) :- add(M,_,N).
```

Multiplication

- Can multiply two numbers:

```
multiply(z,N,z).
multiply(s(N),M,P) :-
  multiply(N,M,Q), add(M,Q,P).
 square(M) :- multiply(N,N,M).
```

List processing

- Recall built-in **list syntax**

```
list([]).
list([X|L]) :- list(L).
```

- **Example:** list append

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Append in action

- Forward direction

```
?- append([1,2],[3,4],X).
```

- Backward direction

```
?- append(X,Y,[1,2,3,4]).
```

Mode annotations

- Notation `append(+,+,-)`
  - "if you call append with first two arguments ground then it will make the third argument ground"
- Similarly, `append(-,-,+)`
  - "if you call append with last argument ground then it will make the first two arguments ground"
- Not "code", but often used in documentation
  - "?" annotation means either + or –

List processing (2)

- When is something a member of a list?

```
mem(X,[X|_]).
mem(X,[_|L]) :- mem(X,L).
```

- Typical modes

```
mem(+,+)
mem(-,+)
```

List processing(3)

- Removing an element of a list

```
remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]) :- remove(X,L,M).
```

- Typical mode

```
remove(+,+,-)
```

List processing (4)

- Zip, or "pairing" corresponding elements of two lists

```
zip([],[],[]).
zip([X|L],[Y|M],[(X,Y)|N]) :- zip(L,M,N).
```

- Typical modes:

```
zip(+,+,-).
zip(-,-,+).  % "unzip"
```

List flattening

- Write **flatten** predicate `flatten/2`
  - Given a list of (lists of ..) lists
  - Produces a list containing all elements in order

List flattening

- Write **flatten** predicate `flatten/2`
  - Given a list of (lists of ..) lists
  - Produces a list containing all elements in order

```
flatten([],[]).
flatten([X|L],M) :- flatten(X,Y1),
                    flatten(L,Y2),
                    append(Y1,Y2,M).
flatten(X,[X]) :- ???.
```

List flattening

- Write **flatten** predicate `flatten/2`
    - Given a list of (lists of ..) lists
    - Produces a list containing all elements in order

```
flatten([],[]).
flatten([X|L],M) :- fla
                    fla
                    append
flatten(X,[X]) :- ???.
```

We can't fill this in yet!
(more next week)

Records/structs

- We can use terms to define data structures

```
pb([entry(james,'123-4567'),...])
```

- and operations on them

```
pb_lookup(pb(B),P,N) :-
   member(entry(P,N),B).
pb_insert(pb(B),P,N,pb([entry(P,N)|B])).
pb_remove(pb(B),P,pb(B2)) :-
   remove(entry(P,_),B,B2).
```

Trees

- We can define (binary) trees with data:

```
tree(leaf).
tree(node(X,T,U)) :- tree(T), tree(U).
```

Tree membership

- Define **membership in tree**

```
mem_tree(X,node(X,T,U)).
mem_tree(X,node(Y,T,U)) :-
  mem_tree(X,T) ;
  mem_tree(X,U).
```

Preorder traversal

- Define **preorder**

```
preorder(leaf,[]).
preorder(node(X,T,U),[X|N]) :-
    preorder(T,L),
    preorder(U,M),
    append(L,M,N).
```

- What happens if we run this in reverse?

Next time

- Nonlogical features
  - Expression evaluation
  - I/O
  - "Cut" (pruning proof search)
- Further reading:
  - Learn Prolog Now, ch. 3-4