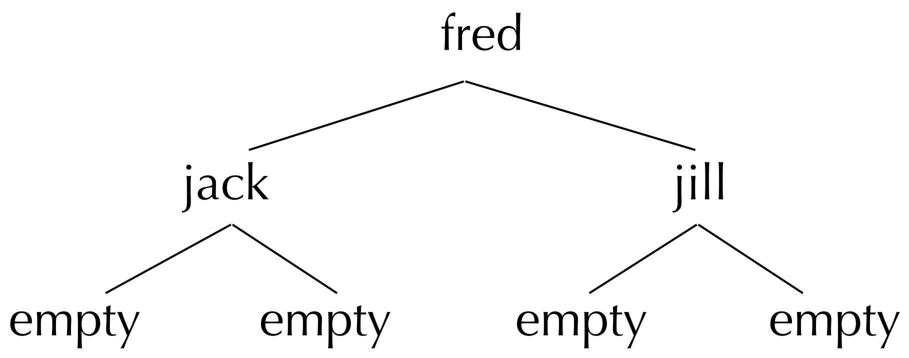
Recursive Programs

- Compound terms can contain other compound terms.
- A compound term can contain the same kind of term, i.e. it can be *recursive*.

  tree(tree(empty, jack, empty), fred, tree(empty, jill, empty))

- "empty" is an arbitrary symbol used to represent the empty tree.
- A structure like this could be used to represent a binary tree that looks like:



Binary Trees

- A binary tree is either empty or it is a structure that contains data and left and right subtrees which are also trees.

- To test if some datum is in the tree:

```
in_tree(X, tree(_, X, _)).
in_tree(X, tree(Left, Y, Right) :-
   X \= Y,
   in_tree(X, Left).
in_tree(X, tree(Left, Y, Right) :-
   X \= Y,
   in_tree(X, Right).
```

The size of a tree

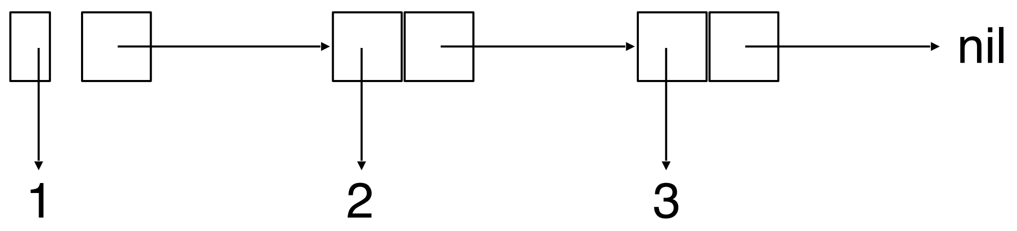
- The size of the empty tree is 0.
- The size of a non-empty tree is the size of the left subtree plus the size of the right subtree plus one for the current node.

```
tree_size(empty, 0).
tree_size(tree(Left, _, Right), N) :-
   tree_size(Left, LeftSize),
   tree_size(Right, RightSize),
   N is LeftSize + RightSize + 1.
```

Lists

- A list may be nil or it may be a term that has a head and a tail. The tail is another list.
- A list of numbers, [1, 2, 3] can be represented as:

```
list(1, list(2, list(3, nil)))
```



- Since lists are used so often, Prolog has a special notation:

```
[1, 2, 3] = list(1, list(2, list(3, nil)))
```

Examples of Lists

```
?- [X, Y, Z] = [1, 2, 3].
X = 1
Y = 2
Z = 3
```

Unify the two terms on either side of the equals sign.

Variables match terms in corresponding positions.

```
?- [X | Y] = [1, 2, 3].
X = 1
Y = [2, 3]
```

The head and tail of a list are separated by using '|' to indicate that the term following the bar should unify with the tail of the list

```
?- [X | Y] = [1].
X = 1
Y = []
```

The empty list is written as '[]'.

The end of a list is *usually* '[]'.

More list examples

```
?- [X, Y | Z] = [fred, jim, jill, mary].

X = fred
Y = jim
Z = [jill, mary]
```

There must be at least two elements in the list on the right

```
?- [X | Y] = [[a, f(e)], [n, b, [2]]].

X = [a, f(e)]
Y = [[n, b, [2]]]
```

The right hand list has two elements:

```
    [a, f(e)]   [n, b, [2]]
```

Y is the tail of the list, [n, b, [2]] is just one element

List Membership

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

Rules about writing recursive programs:

- Only deal with one element at a time.
- Believe that the recursive program you are writing has already been written and works.
- Write definitions, not programs.

Concatenating Lists

```
conc([1, 2, 3], [4, 5], [1, 2, 3,4, 5])
```

Start planning by considering simplest case:

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

Clause for this case:

```
conc([], X, X).
```

Concatenating Lists

Next case:

```
conc([1], [2], [1, 2])
```

Since `conc([], [2], [2])`

```
conc([A | B], C, [A | D]) :- conc(B, C, D).
```

Entire program is:

```
conc([], X, X).
conc([A | B], C, [A | D]) :-
   conc(B, C, D).
```

Reversing Lists

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

Start planning by considering simplest case:

```
rev([], [])
```

Note:

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

and

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

```
rev([], []).
rev([A | B], C) :-
  rev(B, D),
  conc(D, [A], C).
```

An Application of Lists

Find the total cost of a list of items:

```
cost(flange, 3).
cost(nut, 1).
cost(widget, 2).
cost(splice, 2).
```

We want to know the total cost of [flange, nut, widget, splice]

```
total_cost([], 0).
total_cost([A | B], C) :-
       total_cost(B, B_cost),
       cost(A, A_cost),
       C is A_cost +  B_cost.
```

Reference

- Ivan Bratko, *Programming in Prolog for Artificial Intelligence*, 4th Edition, Pearson, 2013.