Administrative Notes

First Exam

- Wednesday, 2/13/2002
  - → In class, in DH 1070
  - → Closed notes, closed book
- Covers Sectons 1-12 of the book
  - → Not family trees
  - → Includes natural numbers (lab lecture + today)
- Covers class lectures, lab lectures, homework 1, 2 & 3
- Know the pledge !

Natural Numbers

210 so far …

- Simple algebra (area of DH 1070)
- Single structures (planes, brands, points, …)
- Recursive data structures (lists, trees)

Programming with these data structures involves *"structural recursion"*

Natural numbers

- For 210, a natural number is a non-negative integer
- These numbers have a natural recursive structure
- How does this fit into our computational model?
  - → Why bring this up, since it doesn't fit the models?

Natural Numbers

Data analysis for Natural Numbers:

;; a natural number is either

;;   – zero, or

;;   – if N is a natural number, then (add1 N) is a natural number

;; we can use Scheme's built-in implementation of numbers

Structure of the natural numbers

- Recursive, like the definition of a list
- Resembles the sketch of an induction proof

Programming with the Natural Numbers

Structural recursion on the Natural Numbers

- We can build a template like the list template

```
;; template for Natural Numbers
(define  (f  a-natnum ... )
   (cond
      [(=  0    a-natnum)     ...]
      [(>  0    a-natnum)    (f  (sub1  a-natnum) ... ) ]
   ))
```

What's this? Where did it come from?

- Can do structural recursion on natural numbers
  - → Recursion in the data is implicit, not explicit
- Why do this?
  - → To simplify reasoning about the resulting program

Programming with the Natural Numbers

Factorial

- Factorial(n) = n * (n-1) * … * 2 * 1
- Factorial(0) = 1

```
;; Factorial:  NatNum -> NatNum
(define  (Factorial  N)
   (cond
      [(=  0   N)    1]
      [(<  0   N)   (*  N   (Factorial  (sub1  N)) ]
   ))
```

Handles the special case when N = 0

Can write this test as (zero? N)

- Why does this terminate?
  - → Intuition says it halts
  - → Calls to Factorial cannot go on forever

Programming with the Natural Numbers

```
;; Factorial:  NatNum -> NatNum
(define  (Factorial  N)
   (cond
      [(=  0   N)    1]
      [(>  0   N)   (*  N (factorial  (sub1  N)) ]
   ))
```

Sketch of Proof

- Factorial(N) has two cases
  - → N = 0 and it returns 1
  - → N > 0
    - Since N is a natural number, we know it can be derived from 0 by repeated calls to add1
    - So, repeated calls to sub1 must eventually produce 0
    - Code always recurs on (sub1 N) ⇒ recursion halts

Programming with the Natural Numbers

```
;; Factorial:  NatNum -> NatNum
(define  (Factorial  N)
  (cond
     [(=  0    N)     1]
     [(>  0    N)    (*  N (factorial  (sub1  N)) ]
  ))
```

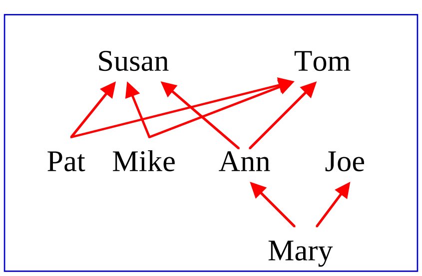
The primary reasons for introducing the Natural Numbers in COMP 210 and for working with them are:

- ***To add formalism to our thinking about structural recursion***
- ***To demonstrate that we don't need a data structure to perform structural recursion; we just need data with a structure – either explicit or implicit***
- ***To add another dimension to our understanding of arithmetic***

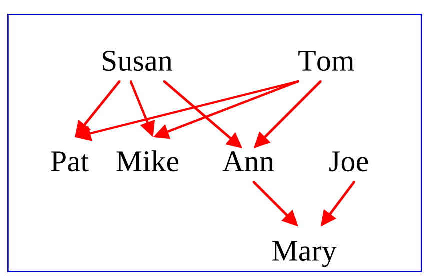
Back to Family Trees

So far, our trees have been rather biased

- Have a *child-centric* view of the world
  - → All links run from parent to child
- Another view is possible – *parent-centric* trees



Child-centric tree



Parent-centric tree

Which one is the right picture?

Parent-centric Family Trees

Data definitions are natural

```
;; a parent is a structure
;;   (make-parent  name year eye-color children)
;; where name & eye-color are symbols,
;;           year is a number, and children is a list of parent

;; a list-of-parent is either
;;   – empty, or
;;   – (cons f r)
;;     where f is a parent and r is a list-of-parent
;; We will use Scheme's built-in list construct
```


Parent-centric Family Trees

```
;; a parent is a structure
;;   (make-parent  name year eye-color children)
;; where name & eye-color are symbols,
;;           year is a number, and children is a list of parent

;; a list-of-parent is either
;;   – empty, or
;;   – (cons f r)
;;     where f is a parent and r is a list-of-parent
;; We will use Scheme's built-in list construct
```

Mutually recursive data structures

- Makes programming a little more complex
- Two data-definitions means two templates, two programs, …

Parent-centric Family Trees

```
(define (f  a-parent ...)
    ... (parent-name a-parent) ...
    ... (parent-year   a-parent) ...
    ... (parent-eye-color a-parent) ...
    ... (g  (parent-children a-parent) ... ) ... )

(define (g a-loc)
   (cond
      [(empty? a-loc)    ... ]
      [(cons?    a-loc)
            ... (f (first a-loc) ...) ...
            ... (g (rest a-loc) ... ) ] ) )
```

Mutually recursive data structures

- Template reflects the data
- Use it in the same basic methodology

Parent-centric Family Trees

Develop count-members:

```
;; count-members: parent -> number
;; Purpose: tally people in tree rooted at parent
(define (count-members  a-parent)
   (+1  (count-children (parent-children a-parent) ) )

;; count-children: list-of-parent -> number
;; Purpose: tally people in all the family trees rooted in the
;;          list-of-parents passed as an argument
(define (count-children a-loc)
   (cond
      [(empty? a-loc)    0 ]
      [(cons?    a-loc)
            (+ (count-members (first a-loc) )
               (count-children (rest a-loc)  ) ] ) )
```

Next class: we will work more with parent trees