Administrative Notes

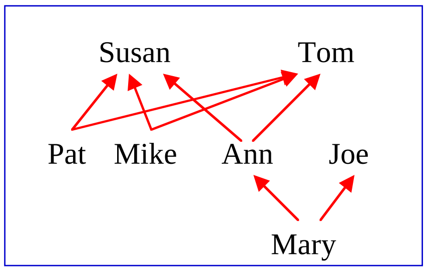
First Exam

- Wednesday, 2/13/2002
  - → In class, in DH 1070
  - → Closed notes, closed book
- Covers Sections 1-12 of the book
  - → Not family trees
  - → Includes natural numbers (lab lecture + today)
- Covers class lectures, lab lectures, homework 1, 2 & 3
- Review session tonight, 7:30 in DH
  - → (room will be posted on door)

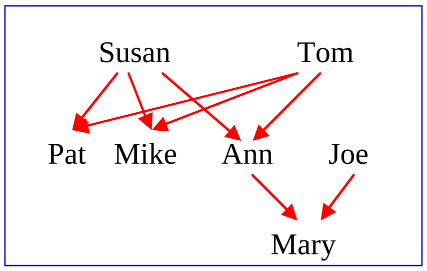
Back to Family Trees

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

Parent-centric Family Trees

What about at-least-two-children

- Consumes a parent
- Returns a list of the names of all parents with ≥ 2 kids
- We'll have 2 programs (from the data-definition & templates)

```
;; at-least-2-children: parent -> list-of-symbol
;; Purpose: build a list of the names of all parents with
;;   two or more children
(define (at-least-2-children  a-parent) ... )

;; children-with-2-children: list-of-parent -> list-of-symbol
;; Purpose: consumes a list of parent & returns a list
;; containing the subset of the input list that have >= 2 kids
(define (children-with-2-children a-loc) ... )
```

Parent-centric Family Trees

Problem-specific knowledge

- At-least-2-children
  - → Need to count immediate descendents
  - → Cons "parent-name" onto list if > 1 descendant
    - Suggests a helper function (> 1 data item in function)
  - → Recur into the next generation
- Children-with-2-children
  - → Test the "first" element
  - → Recur on the rest
  - → Combine the two lists

Parent-centric Family Trees

The first helper function

- Builds on the classic list template

```
;; num-in-list: list-of-parent -> number
;; Purpose:  count the number of parents in the list
(define (num-in-list a-lop)
   (cond
       [(empty?  a-lop)    0]
       [(cons?    a-lop)
           (+        1
                      (num-in-list (rest a-lop)))]
   )
)
```

Parent-centric Family Trees

The second helper function

- Consumes two lists

```
;; combine: list  list ->  list
;; Purpose:  combine the argument lists into one list
(define (combine  list1   list2)
  (cond
     [(empty? list1)        list2]
     [(cons?   list1)
            (cons (first list1)
                     (combine (rest list1) list2) ]
  )
)
```

Now, you develop the rest of the code …

Parent-centric Family Trees

Parent-centric trees don't solve all problems, either

- Number of cousins:
  - → Consume parent and symbol
  - → Return number of cousins that "symbol" has
- Kind-of-cousin:
  - → parent and 2 symbols
  - → Return the relationship (second-cousin, third-cousin, …)
- Lost-parents
  - → Consume parent
  - → Return a list of all people with only one parent

Cannot even ask the question in parent-centric tree

To do real genealogy, need both perspectives