COMP 210, FALL 2000
Lecture 11: Moving Beyond Lists

Reminders:

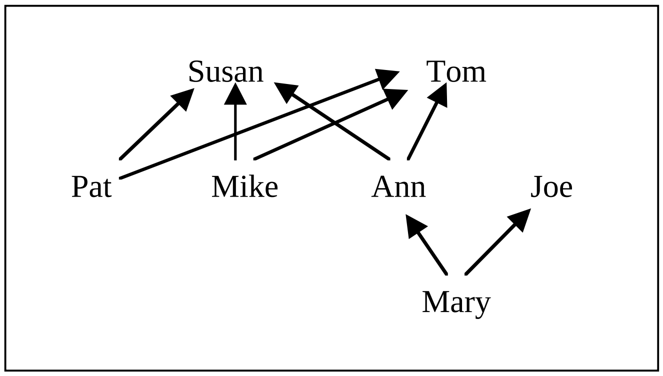
- Homework assignment due **Friday** 2/18/00
- Exam will be 2/16/2000, in class–closed-notes, closed-book

Review

1. Talked about lists with mixed data. More of this on the lab.

Working with Mixed Data

By now you should be comfortable working with lists and with recursion. This gives us the foundation we need to start designing programs that operate over more complex data structures. Today, we'll start by working with family trees.



This family tree depicts three generations of a family. Arrows run from child to parent, so Mary's parents are Ann and Joe, Ann's parents are Susan and Tom, and Pat and Mike are Ann's siblings.

How might we write a data definition that allows us to represent these family trees in Scheme? (Recall, last class we used a list to represent recipes.) This is where I think Computer Science gets fun–-devising new and effective ways to represent complex kinds of information.

```
;; a ftn (for family-tree node) is either
;;        – a symbol, or
;;        – (make-ftn name father mother)
;; where name is a symbol and father &  mother are both ftns
(define-struct ftn (name mother father))

;; Examples
'Mary
(make-ftn  'Ann  'Susan  'Tom)
(make-ftn  'Mary (make-ftn  'Ann  'Susan  'Tom)  'Joe)
(make-ftn   'Pat  'Susan  'Tom)
(make-ftn   'Mike  'Susan  'Tom)
```

Designing Programs for FTNs
What would the template for this **ftn** contain?

```
(define  (f  … a-ftn …)
   (cond
      [(symbol=?   a-ftn)    … ]
      [(ftn?          a-ftn)    …
                  (ftn-name  a-ftn)  …
                  (f  (ftn-mother a-ftn)) …
                  (f  (ftn-father   a-ftn)) … ]
    ) )
```

Let's write a program **in-family?** that consumes an **ftn** and a **symbol** and produces a boolean that indicates whether or not a person with that name is in the family tree.

As a first step, notice that the program will need to compare names. Let's write a helper function **compare-names** that maps two symbols into a boolean.

```
;; compare-names:  symbol symbol -> boolean
;; Purpose:  return true if the symbols are identical
(define (compare-names  n1  n2)
   (symbol=?  n1  n2)
```

Next, we can copy the template over and fill it in.

```
(define  (in-family? a-ftn  name)
   (cond
      [(symbol=?   a-ftn)   (compare-names  a-ftn  name)]
      [(ftn?          a-ftn)
          (or
                  (compare-names (ftn-name  a-ftn)  name)
                  (in-family?  (ftn-mother a-ftn)  name )
                  (in-family?  (ftn-father   a-ftn)  name )
                  ) ]   )  )
```

We can use **or** to check all three possibilities in a single function call, producing the boolean **or** of the answers.

[Actually, the way that you are likely to discover the need for **compare-names** is by developing the code. As you develop **in-family?**, you will discover that it performs the equality test on names in two places. The fact that it occurs multiple times suggests strongly that you break it out into a helper function. In that way, if you need to go back and replace the representation of a name with something more complex–-such as a list of symbols to represent today's elongated, hyphenated names–-all the changes are confined to the function **compare-names**. Without the helper function, these comparisons are spread across the entire program. Finding them, modifying them, and testing them becomes a more significant problem. If all of the tests on a two-digit year had been isolated into a single helper function, or even a couple (for = < & >), the Y2k problem would have been much easier to fix.]

This representation of family trees is quite simple. It only includes people's names and their parent–child relationships. Let's get more realistic. First, we can add more information, such as year of birth (for age) and eye-color. Second, we should be able to account for families where the information about an ancestor is unknown—a common situation in genealogical research.

How would we revise the data definition for **ftn**? These two changes are handled differently. Adding year of birth and eye-color simply adds more fields to the structure. Making allowance for missing parents is a matter of how we build and interpret the data structure; we can use **empty** to represent the missing ancestors and disallow an unencapsulated symbol as a **ftn**.

```
;;  a ftn is either
;;         – empty, or
;;         – (make-ftn name mother father year eyes)
;;  where name is a symbol, mother and father are ftn, year is a number,
;;  and eyes is a symbol
(define-struct  ftn  (name mother father year eyes) )

;; Examples
   empty
   (make-ftn
         'Mary
         (make-ftn  'Ann  empty  empty  1950  'blue)
         empty
         1975
         'green )
```

What does the template for this more complex **ftn** look like?

```
(define  (f   … a-ftn … )
    (cond
         [(empty?  a-ftn)   …  ]
         [(ftn?       a-ftn)   …
                   (ftn-name      a-ftn)  …
                   (f  (ftn-mother   a-ftn) … )  …
                   (f  (ftn-father     a-ftn) … )  …
                   (ftn-year       a-ftn)  …
                   (ftn-eyes       a-ftn)  …
         ]
    ) )
```