COMP 210, FALL 2000
Lecture 12: More Family Trees

Reminders:

- Homework assignment due **Friday** 9/30/00
- Exam will be next class, 9/27/2000, in class (DH 1055)

Review

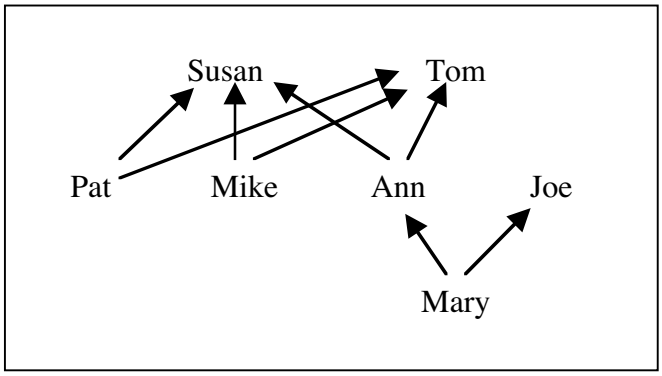
1. Introduced non-list information structures with the example of a child-centric family tree–-that is, a family tree structured from the child's point of vies.
2. Built a program **in-family?** that checked a symbol for membership in a family tree. See the posted lecture notes for a correction to what I said about the need for a helper function in **in-family?**

Back to Family Trees

As you recall, we had defined a family-tree node (**ftn**) as:

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

From this point, we went on to build the program **in-family?** that consumed a **ftn** and a symbol and returned a boolean that indicated whether or not the symbol was found in the argument **ftn**.



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)  …
         ]
    ) )
```

What does the program **in-family?** look like on this new version of **ftn**?

```
;;  in-family? :  ftn  symbol -> boolean
;; Purpose:  returns true if symbol is in the family tree
(define  (in-family? a-ftn name )
    (cond
        [(empty?  a-ftn)   false  ]
        [(ftn?      a-ftn)
             (or
                (symbol=? (ftn-name  a-ftn)  name )
                (in-family?  (ftn-mother   a-ftn)  name)
                (in-family?  (ftn-father     a-ftn)  name) )
         ]
    ) )
```

Done without a helper function because the actual function is trivial.

Let's develop the program **count-female-anscestors**: ftn -> number. It should return the number of female ancestors in the **ftn**; a person does not count as their own ancestor.

```
;; count-female-ancestors: ftn -> num
;; Purpose: consumes a ftn and returns the number of female ancestors
(define (count-female-ancestors a-ftn)
   (cond
      [(empty? a-ftn)     0]
      [else
         (cond
          [(empty?  (ftn-mother a-ftn) (count-female-ancestors (ftn-
father a-ftn))]
           [else (+ 1
                    (count-female-ancestors (ftn-mother a-ftn))
                    (count-female-ancestors (ftn-father   a-ftn)) )]  )]
   ))
```

Is this ok? No, it violates one of the rules of COMP 210–-one discussed in the book that I haven't hit on heavily in class.

> A program should only look inside one data definition. If you need to look inside more than one data-definition, use a second function–-a helper function. The code comes out cleaner; down the road, it is easier to understand and easier to modify.

This version of count-female-ancestors looks inside both **a-ftn** and **(ftn-mother a-ftn)**. Doing so leads to all that mess in the **else** case of the outer **cond.**

Following the rule produces a somewhat simpler version of count-female ancestors.

```
;; count-mother: ftn -> num
;; Pupose: determine how many ancestors to add for current mother
(define (count-mother a-ftn)
   (cond
      [(emtpy?     a-ftn)     0]
      [else                   1]
    ))

;; count-female-ancestors: ftn -> num
;; Purpose: consumes a ftn and returns the number of female ancestors
(define (count-female-ancestors a-ftn)
   (cond
      [(empty? a-ftn)     0]
      [else
         (+ 1 (count-mother (ftn-mother a-ftn)
               (count-female-ancestors (ftn-mother a-ftn))
               (count-female-ancestors (ftn-father   a-ftn)) )]
   ))
```

This is much cleaner.

What if we wanted to only count blue-eyed female ancestors? What must we change? Only the helper function!

```
;; count-if-blue-eyes:  ftn -> num
;; Purpose:  returns 1 if the ftn has blue eyes, 0 otherwise
(define (count-if-blue-eyes a-ftn)
   (cond
      [(symbol=?  'blue  (ftn-eyes a-ftn))   1]
      [else                                  0]
    ))

;; count-mother: ftn -> num
;; Pupose: determine how many ancestors to add for current mother
(define (count-mother a-ftn)
   (cond
     [(emtpy?    a-ftn)    0]
     [else                 (count-if-blue-eyes a-ftn)]
    ))
```

Is this just a matter of esthetics? To some extent, it is. This is where the art comes into programming. The decomposition of the problem into two functions produces a clean, crisp, understandable separation of concerns. The program count-female-ancestors processes the item passed to it. The program count-mother processes the item passed to it. To accomplish its job, count-female-ancestors uses both a recursive call on itself and the call to count-mother. Notice that count-mother is the only place where a number other than zero gets added into the count. The decomposition rule had the effect of separating out the search criterion from the mechanism that guides the search. The result is a cleaner, more readable, more "elegant."