**COMP 210, Spring 2001**
**Lecture 10: More Family Trees**

**Reading Assignment:** Sections 14-17 in the text.

**Reminders:**

- Homework assignment due **Wednesday** 2/14/01
- Exam 2/23/2001, in class (DH 1055)

**Review**

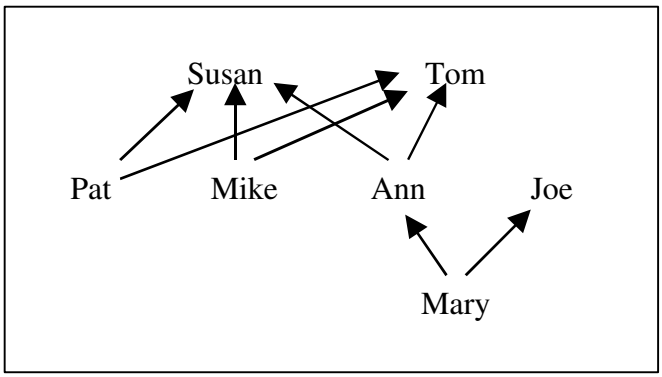
1. Introduced non-list information structures with the example of a child-centric family tree–-that is, a family tree structured from the child's point of vies.
2. Built a program **`in-family?`** that checked a symbol for membership in a family tree. See the posted lecture notes for a correction to 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 (**`FT`**) as:

```
;; an FT is either
;;   - a symbol, or
;;   - (make-FT name father mother)
;; where name is a symbol and father and mother are FT's
(define-struct FT (name father mother))
```

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




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 **`FT`**? 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 **FT**.

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

;; Examples
(define ft1 empty)
(define ft2
  (make-Child 'Mary
              (make-Child 'Ann empty empty 1950 'blue)
              empty
              1975
              'green ))
```

What does the generic template for this richer formulation of **FT** look like?

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

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

```
;; in-family?: FT symbol -> boolean
;; Purpose: returns true if name is in the family tree a-FT
;; Examples: …

;; Template

;; (define  (in-family? a-FT name)
;;   (cond [(empty? a-FT) … ]
;;         [(Child? a-FT) …
;;          (Child-name  a-FT) …
;;             (in-family? (Child-mother a-FT) name) …
;;             (in-family? (Child-father a-FT) name) …
;;          (Child-year a-FT) …
;;          (Child-eyes a-FT) … ]))
(define  (in-family? a-FT name)
  (cond
     [(empty? a-FT)  false]
     [(Child? a-FT)
       (or
         (symbol=? (Child-name a-FT) name)
         (in-family? (Child-mother a-FT) name)
         (in-family? (Child-father a-FT) name))]))
```

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

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

Is this clean code? No, it violates one of the rules of COMP 210–-one discussed in the book that I haven't emphasized in class.

> *A program should only look inside one level of data definition. If you need to look inside more than one level data-definition (nesting* ***`cond`*** *clauses that check the "shape" of the data), use a second function–-a*

*helper function. This rule produces cleaner code comes which, down the road, is easier to understand and easier to modify.*

This version of count-female-ancestors looks inside both **a-FT** and **(Child-mother a-FT)**. Doing so leads to the ugly nested **cond** in the **else** case of the outer **cond.**

Following the "one-level' rule produces a simpler version of **count-female-ancestors**.

```
;; count-mother: FT -> num
;; Pupose: counts the current-mother only in a-FT
(define (count-mother a-FT)
    (cond [(empty? a-FT) 0]
          [else 1]))

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

This program is simpler than our first attempt.

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

```
;; count-mother: FT -> num
;; Pupose: counts current mother only (if blue-eyed)
(define (count-mother a-FT)
  (cond [(empty? a-FT) 0]
        [else (cond [(symbol=? 'blue (Child-eyes a-FT)) 1]
                    [else 0])]))
```

Is this just a matter of esthetics? No. The revised solution encapsulates the counting operation performed in a traversal of an **`FT.`** Hence, any other counting program that requires traversing the tree can be expressed simply by the modifying the function that performs the counting operation.

In fact, we can write an even better program to count female ancestors than our revised version by writing a function that traverses **`FT`**'s and performs any operation that can be computed based on the contents of the "non-recursive" fields of the current node and the values returned by traversing the recursive fields. Such a function takes the specific operations to be performed on **`empty`** and **`make-Child`** objects as *parameters*. We will defer writing this program because the concept of function parameters is not introduced until later in the book.

**Exercise for the bored/ambitious**: read Section 20 in the book and rewrite **`count-female-ancestors`** as an instantiation of a general tree-walking function for the type **`FT`**. Write another function **`oldest`**–using the same tree-walking function--that finds the oldest date of birth in an **`FT`**; if the input is **`empty`**, return the mythical year **`infinity`** defined as **`(/ 1 0.)`**. **Warning:** in DrScheme, you must use the "Intermediate" language level to write these programs; the "Beginner" language will not accept programs that pass functions as arguments.