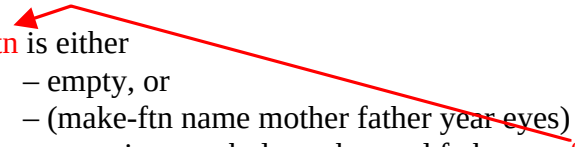
COMP 210, FALL 2000
Lecture 13: Family Trees, From Yet Another Angle

Reminders:
- 1/2 Homework assignment, available today, due Wednesday
- Exam will be handed back on Monday

Review
1. Spent some time talking about the test
2. Developed a new notion of family tree that included **empty** and more fields; then we developed **in-family?** and **count-female-ancestors** for these **ftn**s.

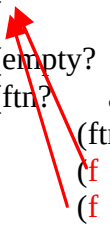
Defining a Family Tree, Take 2



```
;;  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)  …
          ]
    ) )
```

Here is a mildly corrected version of **count-female-ancestors** from the end of last class

```
;; 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."

Parent-centric Family Trees

So far, our family trees are only of interest to children. All edges run from child to parent. (In fact, this is natural. Children are the ones who get to study family trees. Parents usually know more details about their descendants than anyone else wants to know. The difference between a parent's ancestors and a child's ancestors is fairly obvious to the child's parents!)

Assume we wanted to reverse the edges in our family tree and create an information structure that would allow us to ask questions about a person's descendants. What sort of data-definition would we write?

```
;; a parent is a structure
;;       (make-parent   name year eyes  children)
;; where name and eyes are symbols, year is a number, and
;;          children is a list-of-children
(define-struct parent (name year eyes children)
```

Notice that the number of children is indeterminate. With the child tree, the set of parents was fixed and small, so a structure made sense. Here, we use a list.

We also need a data-definition for list-of-children

```
;; a list-of-children is either
;;         –- empty, or
;;         –- (cons f  r)
;;  where f is a parent and r is a list-of-children
;;  [Since we used cons, we don't need the define-struct …]
```

These data-definitions refer to each other. We say that they are mutually dependent or mutually recursive. [The definition of list-of-children is **also** self-referential (recursive).]

```
;; example data
(make-parent  'Tom 1930 'blue
          (cons   (make-parent  'Ann  1952  'green
                        (cons (make-parent 'Mary 1975 'green empty)   empty))
                  (cons  (make-parent 'Mike 1955  'blue  empty)
                         empty)) )
```

What about a template for these data definitions?

```
;; (define  (f    a-parent …)
;;         (parent-name  a-parent) …
;;         (parent-year   a-parent) …
;;         (parent-eyes   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)) … ] ))
```

The template for a mutually recursive data definition contains one template for each constituent data definition. To reflect the recursion in the data definition, we have added the calls to f and g. When the template uses a selector function that refers to an instance of the other data-definition, we have included the appropriate call to the template for that data-definition. In this way, the template reflects the coupling of the data-definitions.

Let's develop the program **count-members** which consumes a parent and returns the number of people in the family tree rooted at the parent.

<and, we ran out of time, probably because I wrote too slowly at the board…>