COMP 210, Spring 2002
Lecture 10: More on Trees

Start of the second third of COMP 210 –- the first lecture for the second exam.

Reminders:

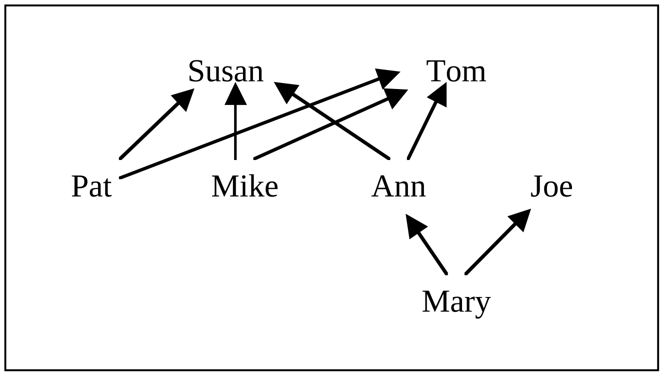
- Homework assignment next **Friday** 2/15/02
- Exam will be 2/13/2002, in class–closed-notes, closed-book
  - ➢ Covers sections 1-12 (natural numbers but not family trees)
- Review will be Monday at 7:30pm, location to be announced

Review

⇒ Started on family trees

*<You are responsible for this lecture on the **second** exam>*

Pre-load Side Board with



```
;; 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)
```

Template 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)) … ]
    ) )
```

In-family?

```
;; in-family?: ftn  symbol → boolean
;; Purpose: determine if the symbol is in the ftn
;;               return true if found and false otherwise
(define  (in-family? a-ftn  kin)
   (cond
      [(symbol?   a-ftn)   (symbol=?  a-ftn  kin)]
      [(ftn?          a-ftn)
         (or
            (symbol=?   (ftn-name  a-ftn)  kin)
            (in-family?  (ftn-mother a-ftn)  kin )
            (in-family?  (ftn-father   a-ftn)  kin )
            ) ]   )  )
```

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

Should we consider writing a helper function to compare the names? After all, the function occurs in two places. The function would look something like

```
(define (compare-names n1 n2)
   (symbol=?  n1  n2))
```

This function looks a little ridiculous. It simply passes **n1** and **n2** on to the built-in function **symbol=?** and returns the result. Why would we build a helper function for that?

Well, with **name** implemented as a symbol, writing **compare-names** will make little sense. If, however, names were, themselves, compound objects where the equality test required use of selector functions, or application of multiple equality tests, then abstracting out this function into a helper like **compare-names** would make sense.

Sometimes, you can see these coming. More often, you will discover the need for a helper function like **compare-names** as you are writing the code that needs help. You should still go ahead, create the helper function, and use it to simplify the code. Using a helper function to replace short but complex sequences of code that are repeated makes the resulting code easier to read. It also centralizes the knowledge and control into the helper function–-in the sense that a later change can be made in one place, rather than in many places. This should, in principle, lead to software that is easier to understand, to modify, and to maintain.

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.]

Another version of in-family?

```
;; in-family?: ftn  symbol → boolean
;; Purpose: determine if the symbol is in the ftn
;;              return true if found and false otherwise
(define  (in-family? a-ftn  kin)
   (cond
      [(symbol?   a-ftn)   (symbol=?  a-ftn  kin)]
      [(ftn?        a-ftn)
         (cond
            [(symbol=?   (ftn-name  a-ftn)  kin)    true]
            [(in-family?  (ftn-mother a-ftn)  kin )  true]
            [(in-family?  (ftn-father   a-ftn)  kin )  truie]
            [else                                    false]
            ) ]   )  )
```

More Complex Information

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 item. If you need to look inside more than one data item, 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."

Not to Belabor the Point

Here is some more hand-evaluation on family trees. You should look through this material. I skipped over this in lecture in the interest of moving quickly.

To finish up with **in-family?** on this version of **family trees**, let's apply the program to some of our example data.

```
(in-family? 'Joe  'Keith)
⇒  (cond
      [(symbol?   'Joe)   (symbol=?  'Joe  'Keith)]
      [(ftn?         'Joe)
         (or
              (symbol=? (ftn-name  'Joe)  'Keith)
              (in-family?  (ftn-mother 'Joe)  'Keith )
              (in-family?  (ftn-father   'Joe)  'Keith )
              ) ]   )  )

⇒  (cond
      [true      (symbol=?  'Joe 'Keith)]
      [(ftn?           'Joe)
         (or
              (symbol=? (ftn-name  'Joe)  'Keith)
              (in-family?  (ftn-mother 'Joe)  'Keith )
              (in-family?  (ftn-father   'Joe)  'Keith )
       ) ]   )  )
⇒      true      (symbol=?  'Joe 'Keith)]
⇒     (symbol=?  'Joe 'Keith)
⇒     false
```

What about a more complex example?

```
(in-family?
        (make-ftn  'Mary (make-ftn  'Ann  'Susan  'Tom)  'Joe)
        ‘Keith)

⇒ (cond
       [(symbol?  (make-ftn  'Mary (make-ftn  'Ann  'Susan  'Tom)
                                 'Joe))
        (symbol=?  a-ftn  ‘Keith)]
       [(ftn?  (make-ftn  'Mary (make-ftn  'Ann  'Susan  'Tom) 'Joe)
               (or (symbol=? (ftn-name
                                 (make-ftn 'Mary
                                    (make-ftn  'Ann  'Susan  'Tom)
                                    'Joe))
                                 ‘Keith)
                   (in-family?  (ftn-mother
                                  (make-ftn 'Mary
                                    (make-ftn  'Ann  'Susan  'Tom)
                                     'Joe))
                                   ‘Keith)
                   (in-family?  (ftn-father
                                  (make-ftn 'Mary
                                    (make-ftn  'Ann  'Susan  'Tom)
                                     'Joe))
                                   ‘Keith)) ]   )

⇒ (cond
        [false                   (symbol=?  a-ftn  ‘Keith)]
        [(ftn?  (make-ftn  'Mary (make-ftn  'Ann  'Susan  'Tom) 'Joe)
                (or (symbol=? ‘Mary ‘Keith)
                    (in-family? (make-ftn  'Ann  'Susan  'Tom)  ‘Keith)
                    (in-family?  'Joe ‘Keith)) ]   )

⇒ [true  (or (symbol=? ‘Mary ‘Keith)
              (in-family? (make-ftn  'Ann  'Susan  'Tom)  ‘Keith)
              (in-family?  'Joe ‘Keith)) ]
```

In the printed trace, the words false, true and each or are in bold.

```
⇒ (or (symbol=? ‘Mary ‘Keith)
        (in-family? (make-ftn  'Ann  'Susan  'Tom)  ‘Keith)
        (in-family?  'Joe ‘Keith))
⇒ (or  false
        (cond
          [(symbol?   (make-ftn  'Ann  'Susan  'Tom))
            (symbol=?  (make-ftn  'Ann  'Susan  'Tom) ‘Keith)]
          [(ftn? (make-ftn  'Ann  'Susan  'Tom))
            (or
              (symbol=? (ftn-name  (make-ftn  'Ann  'Susan  'Tom))
                     ‘Keith)
             (in-family? (ftn-mother (make-ftn  'Ann  'Susan
                                                       'Tom))
                     ‘Keith)
             (in-family? (ftn-father   (make-ftn  'Ann  'Susan
                                                       'Tom))
                     ‘Keith)

                )])
          (in-family?  'Joe ‘Keith)
        )


⇒ (or  false
        (cond
          [false (symbol=?  (make-ftn  'Ann  'Susan  'Tom) ‘Keith)]
          [(ftn? (make-ftn  'Ann  'Susan  'Tom))
            (or
              (symbol=? (ftn-name  (make-ftn  'Ann  'Susan  'Tom))
                     ‘Keith)
             (in-family? (ftn-mother (make-ftn  'Ann  'Susan
                                                       'Tom))
                     ‘Keith)
             (in-family? (ftn-father   (make-ftn  'Ann  'Susan
                                                       'Tom))
                     ‘Keith)

                )])
          (in-family?  'Joe ‘Keith)
        )
```

```
⇒ (or  false
       (cond
          [true
             (or
                (symbol=? (ftn-name  (make-ftn  'Ann  'Susan  'Tom))
                          'Keith)
               (in-family? (ftn-mother (make-ftn  'Ann  'Susan
                                                  'Tom))
                          'Keith)
               (in-family? (ftn-father  (make-ftn  'Ann  'Susan
                                                   'Tom))
                          'Keith)

                 )])
         (in-family?  'Joe 'Keith)
      )
⇒ (or  false
      (or  (symbol=? (ftn-name  (make-ftn  'Ann  'Susan  'Tom))
                          'Keith)
           (in-family? (ftn-mother (make-ftn  'Ann  'Susan  'Tom))
                          'Keith)
           (in-family? (ftn-father  (make-ftn  'Ann  'Susan  'Tom))
                          'Keith)

                 )
         (in-family?  'Joe 'Keith)
      )
⇒ (or  false
      (or  (symbol=?  'Ann 'Keith)
           (in-family? 'Susan 'Keith)
           (in-family?  'Tom  'Keith) )
      (in-family?  'Joe 'Keith)
      )
⇒ (or  false
      (or false false false)
      false)
⇒ false
```

These all evaluate to false. It just takes a while, expanding each call to **in-family?** and working it through. We've done enough to make the point; the rest would be painful!