COMP 210, Spring 2002
Lecture 12: Parent-centric Family Trees

Reminders:
- See PowerPoint Slides

Review
1. Spent some time talking about the test
2. Finshed up child-centric family trees. (We have seen 2 versions of this structure; the first was simpler, the second provided some additional functionality.)

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 set of templates 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.

```
;; count-members:   parent -> number
;; Purpose:  tally the number of people in the tree rooted at parent
(define (count-members a-parent)
    (+1 (count-children (parent-children a-parent) ))
    )
```

```
;; count-children:  list-of-children -> number
;; Purpose: compute how many people are in the family trees rooted at
children
(define (count-children a-loc)
   (cond
       [(empty?  a-loc)  0]
       [(cons?     a-loc)
           (+
               (count-members (first a-loc))
               (count-children   (rest  a-loc)))]
     ))
```

The template gives us the code.

Now, write **at-least-two-children**, a program that consumes a parent and returns a list of the names of all parents in the tree with at least two children.

```
;; at-least-two-children:  parent -> list-of-symbol
;; Purpose:  return a list of all people in the tree with at least 2 children
(define (at-least-two-children a-parent)
   (cond
       [(> (num-children (parent-children a-parent)) 2)
        (cons (parent-name a-parent)
              (children-with-two-children (parent-children a-parent)))]
       [else   (children-with-two-children  (parent-children a-parent))]
))
```

```
;; children-with-two-children: list-of-children -> list-of-symbol
;; Purpose:  returns a list of all children with at least 2 children
(define (children-with-two-children a-loc)
   (cond
       [(empty? a-loc)  empty]
       [(cons?    a-loc)
          (append   (at-least-two-children  (first  a-loc))
                     (children-with-two-children (rest a-loc)))] ))

;; num-children:  list-of-children -> num
;; Purpose: counts how many children are in the list
(define (num-children  a-loc)
  (cond
      [(empty?  a-loc)   0]
      [else  (+ 1 (num-children (rest a-loc)))]  ))
```

Append takes two or more lists and returns the list that has the elements of the first, followed by the elements of the second, followed by …

This is just length–-a Scheme built-in function