COMP 210, Spring 2001
Lecture 11: Parent-centric Family Trees

Reminders:

- Next homework (#4) due Wednesday
- Exam in class on Friday, February 23, 2001. Covers material in book through Section 15.

Review

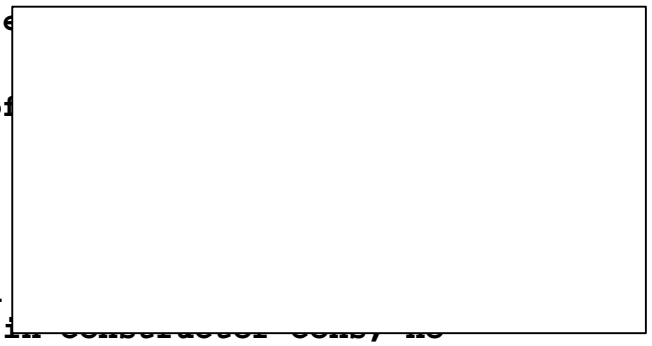
- Finished child-based family trees. (We have seen two versions of this structure; the first was simple but inelegant, while the second provided additional functionality.)

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



We also need a data-definition for **`list-of`**

```
;; a list-of-Person is either
;; - empty, or
;; - (cons f r)
;; where f is a Person and r is a
;; [Since we are using the built-in
;;  define-struct in necessary.]
```

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

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

What are the generic templates for these data definitions?

```
;; (define  (f … a-Person …)
;;    … (Person-name  a-Person) …
;;    … (Person-year   a-Person) …
;;    … (Person-eyes a-Person) …
;;   (g (Person-children a-Person)) … )

;; (define  (g … a-lop …)
;;   (cond
;;     [(empty? a-lop) … ]
;;     [(cons? a-lop)
;;       … (f … (first a-lop) … ) …
;;       … (g … (rest a-lop)) … ]))
```

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-people`** which consumes a **`Person`** and returns the number of people in the family tree rooted at the **`Person`**.

```
;; count-people: Person -> number
;; Purpose: tallies the number of people in the tree a-Person
(define (count-people a-Person)
  (add1 (count-children (Person-children a-Person))))
```

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

The template gives us the code.

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

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

```
;; children-with-two-children: list-of-Person -> list-of-symbol
;; Purpose: returns a list of all people in a-loc 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  (add1 (num-children (rest a-loc)))]))

; append is a Scheme library function with th
;   definition
; append: list list -> list  (more precisely
; Purpose: given l1 = (a1 … am) and l2 =
;          the list (a1 … am b1 … bn)
; (define (append lop1 lop2) …)
; Examples
; …
; Correct Template
; (define (append lop1 lop2 )
;   (cond [(empty? lop1) … ]
;         [(cons? lop1) …(firs
;                        … (appe
;
; (define (append lop1 lop2 )
;   (cond [(empty? lop1) lop2]
;         [(cons? lop1) (cons (first lop1)
;                             (append (rest lop1) lop2))]))
```

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

In class Problem

Write **`has-blue-eyes: Person -> list-of-Person`** where every Person on the resulting list has blue eyes.