Here Comes The Definition, Again (apologies to A. Lenox)

;;;; 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)

;;;; a list-of-parent is either
;;;; — empty, or
;;;; — (cons f r)
;;;; where f is a parent and r is a list-of-parent
;;;; [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).]
What about a template for these data definitions?

```lisp
;; (define (f a-parent …)
;; (parent-name a-parent) …
;; (parent-year a-parent) …
;; (parent-eyes a-parent) …
;; (g (parent-kids a-parent) …)
;; (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-members` which consumes a parent and returns the number of people in the family tree rooted at the parent.

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

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

The template gives us the code.

**OPTIONAL PROBLEM (10 minutes)**
Write `kids-with-blue-eyes` : parent -> list-of-parent where every parent on the resulting list has blue eyes.
Now, write **at-least-two-kids**, a program that consumes a parent and returns a list of the names of all parents in the tree with at least two kids.

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

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

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

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