COMP 200: Elements of Computer Science
Fall 2004
Lecture 9: September 14, 2004

Introducing Lists

On the Board

Homework 2: due at start of class today
Reading: Chapter 2

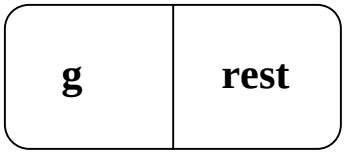
Making Lists

Suppose we don't know how many students are in the class and we need to record and manipulate their grades. A ***list*** is a natural, intuitive way of organizing open-ended collections of data, such as the set of things that a child must take to school or the set of groceries that a shopper needs to buy.

For now, assume that all we need to keep is the scores. Soon, we'll add the complication of associating a name and other information with the scores.

In Scheme, we can represent a list of scores, or a *list-of-numbers* as:

```
;  a list-of-numbers is
;      (make-lon  num  others)
;  where num is a number and others is a list-of-numbers
(define-struct  lon  (num others)
```



The definition of ***list-of-numbers*** refers back to itself. Each ***list-of-numbers*** contains another ***list***, a kind of self-referential structure that we call a structural recursion.

To define such a list, we could write

```
(define example
   (make-lon 1
      (make-lon 2
         (make-lon 3
            (make-lon 4    ) ) ) ) )
```

But, what goes inside that last slot? — the ***others*** portion of the last ***make-lon*** in the definition. If we restrict that slot to holding a ***list-of-numbers***, we have an unending structural recursion. Clearly, we need a degenerate case or a base case. The base Scheme implementation contains an object named

empty to handle just this situation. The object ***empty*** is important enough that it has its own predicate — ***empty?***

(***empty?*** x) returns ***true*** if x is the object ***empty*** and false otherwise.

We need to revise the definition of ***list-of-numbers***

```
;  a list-of-numbers is either
;      empty, or a structure
;      (make-lon num  others)
;  where num is a number and others is a list-of-numbers
(define-struct  lon  (num  others)
```

Now, we can write our example list as

```
(define example
   (make-lon 1
      (make-lon 2
         (make-lon 3
            (make-lon 4   empty ) ) ) ) )
```

Board work: Fill it in on the original copy

Template for list-of-numbers

We need a template for programs written using *list-of-numbers*. The template is somewhat more complicated than others that we have seen, because of the self-reference and because *list-of-numbers* itself is defined as either one of two alternatives — *empty* or a *list-of-numbers*.

```
;  template for list-of-numbers
(define (ALonFn alon)
  (cond
    ( (empty? alon)  … )
    ( else
          …  (lon-num alon) …
          … (ALonFn (lon-others alon))… )
  ) )
```

From the red arrow (self reference)

To write a program that counts the number of entries in a *list-of-numbers*, we start with a contract, purpose, and header.

```
; CountListLength: list-of-numbers → number
; Purpose: return the number of nonempty elements in the input list
(define (CountListLength alon) …)
```

Next, some test data:

Input	Answer
(define example (make-lon 1 (make-lon 2 (make-lon 3 (make-lon 4 empty)))))	10
Empty	0

Now, filling in the template, we get something like

```
; CountListLength: list-of-numbers → number
; Purpose: return the number of nonempty elements in the input list
(define (CountListLength alon)
   (cond
     ((empty?  alon)   0)
     (else                (+ 1  (CountListLength (lon-others alon))))
   ))
```

From the test data

What about a program that sums the elements in a *list-of-numbers*?

.. next lecture..