



COMP 200: Elements of Computer Science Fall 2004

Lecture 10: September 14, 2004

Introducing Lists

Discuss the exam

On the Board

Homework 2: due at start of class today

Reading: Chapter 2

Making Lists

Suppose we don't know how many students we have to record and manipulate their grades. A more general problem is organizing open-ended collections of data, such as the set of children a child must take to school or the set of grades a student has.

For now, assume that all we need to keep track of is a list of scores. For complication of associating a name and other information with the scores.

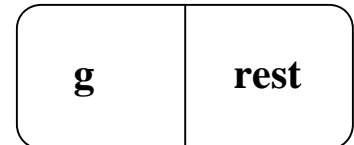
The first half of these notes is a complete overlap with Lecture 9. I repeated some of this material to talk about termination and to get CountListLength back in the front of everyone's mind.

We developed a couple of list-based programs from the template and looked at how following the methodology lets us avoid making the mistake of division by zero in computing the average value in a list.

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  ) ) ) ) )
```

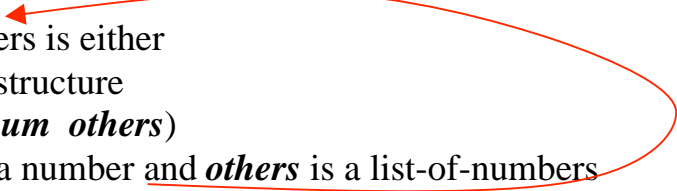
But, what goes inside that last slot? — the *others* portion of the last *make-lon* in the definition. If we restrict that slot to holding a *list-of-numbers*, we have an unending structural recursion. Clearly, we need a degenerate case or a base case. The base Scheme implementation contains an object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

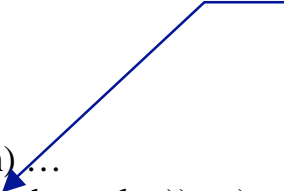


Template for list-of-numbers

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
<i>Empty</i>	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*?

```

; TotalPoints; list-of-numbers → number
; Purpose: sums the numbers in a list-of-numbers
(define (Total alon) ...)

```

Test data ,...

Input	Answer
(define example (make-lon 1 (make-lon 2 (make-lon 3 (make-lon 4 (make-lon 5 (... (make-lon 10 empty) ...)))	55
<i>Empty</i>	0

And the code

```
; Total; list-of-numbers → number
; Purpose: sums a list-of-numbers
(define (Total alon)
  (cond
    ((empty? alon) 0)
    (else
     (+ (lon-num alon)
        (Total (lon-others alon)))))
  ))
```

Finally, we would like to compute an average for the numbers in the list. It is tempting to apply our knowledge of the program directly and write

```
; ListAverage: list-of-numbers → number
; Purpose: compute the average of a list-of-numbers
(define (ListAverage alon)
  (/ (Total alon) (CountListLength alon)))
```

What if CountListLength returns 0? The average is not defined unless the list has at least one entry. Following the methodology leads us to the template, which has two cases for any program that uses a list-of-numbers. That, in turn, leads to a program such as the following:

```
; ListAverage: list-of-numbers → number or false
; Purpose: compute the average of a list-of-numbers
(define (ListAverage alon)
  (cond
    ((empty? alon) false)
    (else
     (/ (Total alon) (CountListLength alon))))
  ))
```