Topic 8
Sequences as Conventional Interfaces

Section 2.2.3

October 2008

Before we get started…

- Let me go back to a function we were writing at the end of class and let's see if I can explain more clearly why I reacted the way I did to the function we wrote and the comments (parameter names) associated with the function.

Background (1st mistake)

- We never gave a definition of what a tree (of numbers was)
- A tree of numbers is
  - A number
  - A list whose elements are trees of numbers


```
; takes a tree of numbers and returns
; a tree that looks the same but the
; numbers have been scaled by num
(define (map-scale-tree tree num)
   (if (pair? tree)
       (map (lambda (x)
              (scale-tree x num))
            tree)        ; node case
       (* num tree)))    ; leaf case
```


What was I writing?

- Notice that when I was leading the writing of the function, I hadn't bothered to carefully pay attention to the comments (that were on the slide)
- In the variable naming convention I had used I was taking a tree to be an arbitrarily complex LIST whose atomic elements were numbers.
- In that case ALL calls to the function should take a LIST (and not a number).
- This forces a different strategy for writing the function.


```
; takes a list whose atomic elements are
  numbers
; and returns a list that looks the same except
; the original numbers have been scaled by num
(define (map-scale-num-list elist num)
  (map (lambda (x)
         (cond ((pair? x)
                (map-scale-num-list x num))
               (else (* x num))))
       elist))
```

(map-scale-num-list '((2 (1) (4 3))) 5)
→ '((10 (5) (20 15)))

Sequences as conventional interfaces

A process can often be decomposed into a sequence of stages

Examples

- compiling
- finding words that are common to two text files

Common types of stages

- Enumerate – the individual pieces of interest
- Filter – to isolate the pieces you are interested in
- Transduce – change the pieces in some way
- Accumulate – put the changed pieces back together

Sum of squares of odd leaves in a tree

1) Make list of all leaves [enumerate]
2) Extract the odd leaves from list [filter]
3) Make list of the squares [transduce]
4) Sum up the squares [accumulate]

A definition not based on stages

```
; sums the squares of all odd elements in
  a tree
(define (sum-of-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree)
             (square tree)
             0))
        (else (+ (sum-of-odd-squares
                  (car tree))
                 (sum-of-odd-squares
                  (cdr tree))))))
```


Definition based on stages

```
(define (sum-of-odd-squares tree)
  (accumulate +
            0
            (map square
                 (filter odd?
                         (enumerate-tree
                           tree)))))
```

[accumulate] [transduce] [filter] [enumerate]

A general-purpose filter

```
; returns a list containing those
; elements of lst that return non-#f for
; test
(define (filter test lst)
  (cond ((null? lst) empty)
        ((test (car lst))
         (cons (car lst)
               (filter test (cdr lst))))
        (else (filter test (cdr lst)))))

> (filter even? (list 4 5 7 2 6 9 10 1))
  -->
(4 2 6 10)
```

A general-purpose accumulator

```
; put together the elements of lst using
; binary-op
(define (accumulate binary-op
                    init-val
                    lst)
  (if (null? lst)
      init-val
      (binary-op (car lst)
                 (accumulate binary-op
                             init-val
                             (cdr
                                lst)))))
```


Accumulate examples

```
(accumulate + 0 (list 2 4 6 8))
--> 20

(accumulate * 1 (list 2 4 6 8))
--> 384

(accumulate cons () (list 2 4 6 8))
→(2 4 6 8)

(accumulate append () '((a b) (c d) (e f)))
→(a b c d e f)
```


More Accumulate Examples

```
(accumulate (lambda (x y) (if (< x y)
                              y
                              x))
            0
            (list 2 4 6 8 3 5))

--> 8
```


Enumerators are problem dependent

```
; makes a list out of elements between
; numbers lo and hi
(define (enumerate-interval lo hi)
  (if (> lo hi)
      ()
      (cons lo
            (enumerate-interval (+ lo 1)
                                hi))))

(enumerate-interval 3 10) -->
   (3 4 5 6 7 8 9 10)
```


Enumerating leaves

```
; enumerates the leaves of a tree
(define (enumerate-tree tree)
  (cond ((null? tree) ())
        ((not (pair? tree)) (list tree))
        (else
          (append (enumerate-tree (car
                                   tree))
                  (enumerate-tree
                    (cdr tree))))))

if x --> ((1 3) 2 (5 (4 6))), then
(enumerate-tree x) --> (1 3 2 5 4 6)
```


Now we can do it!

```
(define (sum-of-odd-squares tree)
  (accumulate +
              0
              (map square
                (filter odd?
                  (enumerate-tree
                   tree)))))

if x --> ((1 3) 2 (5 (4 6))), then
(sum-of-odd-squares x) --> 35
```

Why decompose into stages?

Because procedures that are decomposed into stages are easier to understand and to write

Data processing example

```
(define (salary-of-highest-paid-programmer
         records)
  (accumulate max
              0
              (map salary
                   (filter programmer?
                           records))))
```


Nested mappings

- maps are like loops, converting one list into another
- loops can be nested; so can maps
- Sometimes we want the results of inner lists appended together instead of returned as a list of lists

Example using nested maps

Given a positive integer n, find all integer triples <i,j,i+j> such that

- 1 <= j
- j < i
- i <= n
- i+j is a perfect square (i.e., the square of another integer)

(square-sum-pairs 8)
→ ((3 1 4) (8 1 9) (7 2 9) (6 3 9) (5 4 9))

- Strategy: find possible perfect squares, then find the ways that they can be written as sums of two integers

Decomposition of problem

1) Enumerate numbers from 1 through n
   (1 2 3 4 5 6 7 8)

2) Make list of squares of these numbers
   (1 4 9 16 25 36 49 64)

3) Save only the squares < 2*n
   [j < i <= n, so i+j <= n + n - 1 = 2n - 1]
   (1 4 9)

Last stage

4) For each square that was saved, make a list of all the triples <i,j,i+j> such that i+j = the square. Append these lists together rather than returning a list of lists of triples.

If n = 8, we want to get the list
((3 1 4) (8 1 9) (7 2 9) (6 3 9) (5 4 9))

Code for first 3 steps

1) Enumerate numbers from 1 through n
2) Make list of squares of these numbers
3) Save only the squares < 2*n

```
(filter
  (lambda (s) (< s (* 2 n)))
  (map square
       (enumerate-interval 1 n)))
```


How do we do the last stage

4) For each square that was saved, make a list of all the triples <i,j,i+j> such that i+j = the square. Append these lists together rather than returning a list of lists of triples.

One possible solution

```
(define (square-sum-pairs n)
  (accumulate
    append
    empty
    (map (lambda (s)
           (map (lambda (j)
                  (list (- s j) j s))
                (enumerate-interval
                  (max 1 (- s n))
                  (* 0.5 ( - s 1)))))
```


(continued)

```
(filter
     (lambda (s) (< s (* 2 n)))
     (map square
          (enumerate-interval 1 n))))))
```


A general-purpose procedure

```
(define (flatmap proc list)
  (accumulate append
              empty
              (map proc list)))
```

Another possible solution

```
(define (square-sum-pairs n)
  (flatmap (lambda (s)
             (map (lambda (j)
                    (list (- s j) j s))
                  (enumerate-interval
                    (max 1 (- s n))
                    (* 0.5 (- s 1)))))
           (filter (lambda (s) (< s (* 2 n)))
              (map square
                (enumerate-interval 1 n)))))
```