COMP 210, FALL 2000
Lecture 27: More Programs with Accumulators

Reminders:

1. Missionaries & Cannibals is on the web. Due 12 April 2000. Work one subproblem each day and you'll finish ahead of time. If you start late, it may well eat you alive.
2. Exam 2 was returned on Friday. Statistics in the class notes for Lecture 26.

Review

1. We developed a version of **find-flights** (our program for JetSet Airlines) that used an accumulator to recognize when it was called on a city that had already been processed. This allowed the program to process each city exactly once. This led to correct behavior, a simple termination argument, and a reasonably simple program. (Albeit one that relied on the built-in function **memq**.)

Another Example
Our friends at JetSet Airlines are so impressed with find-flights that they want the class to develop some support programs for their Human Resources department. Each employee is allocated a fixed number of sick days per month. (The number varies with job classification and seniority.) Sick days accumulate month-to-month over a calendar year. On January 1 of each year, the allocation of sick days is reset (the policy known as "use them or lose them"). Since the good folks at JetSet are dedicated Scheme users, they represent each employee's allocation of sick days as a list.

For example, the list (list 2 –1 1) records an employee who was allocated two sick days in January, used one more sick day than allocated in February, and used one less sick day in March than allocated. To manage their assignment of people to tasks, and to predict how many people will be missing from work in December, JetSet needs a program that computes, for a list of sick days, the number of sick days accumulated at each month.

```
;; available-days: list-of-number → list-of-number
;; Purpose: consumes a list of sick days earned per month and produces a list
;;              of sick days available per month
(define (available-days alon) …)

(available-days (list 1)) = (list 1)
(available-days (list 2 –1 1) = (list 2 1 2)
```

You know that this lecture is about accumulators, so let's ignore accumulators for the moment and write a version based on structural recursion. [After all, we're more familiar with structural recursion, and examining the structural recursive version might provide us with some insight into why we would want to use an accumulator.]

```
;; add-to-each: number list-of-number → list-of-number
;; Purpose: add the scalar argument to each element of the list argument
(define (add-to-each n alon)
  (map  (lambda (x) (+ x n))  alon))

;; available-days: list-of-number → list-of-number
;; Purpose: consumes a list of sick days earned per month and produces a list
;;              of sick days available per month
(define (available-days alon)
  (cond  [(empty? alon)  empty]
         [else  (cons  (first alon)
                       (add-to-each  (first alon) (available-days (rest alon))))]))
```

Write out the classic list template and develop from it.

What happens when we invoke this on (list 2 –1 1)

```
(available-days (list 2 –1 1))
= (cons 2 (add-to-each 2 (available-days (list –1 1))))
= (cons 2 (add-to-each 2
                (cons –1 (add-to-each  -1 (available-days (list 1))))))
= (cons 2 (add-to-each 2
                (cons –1 (add-to-each –1
                          (cons 1 (add-to-each 1 (available-days empty)))))))
```

Each call to **add-to-each** passes its list argument to **map**, which traverses the entire list. So, by the time we finish the list, the last element will have been visited by map one time for each other element in the list. We will have added each other element in the list to the final element, individually.

For a list of *n* elements, we will add the first element to *n-1* elements. We will add the second element to *n-2* elements. We will add the third element to *n-3* elements, and so on until we add the $n\text{-}1^{st}$ element to 1 element. The total number of additions that add-to-each performs is

$$n\text{-}1 + n\text{-}2 + n\text{-}3 + \ldots + 2 + 1$$

In the limit, for a list of *n* elements, add-to-each will perform $n(n\text{-}1)/2$ additions. (The sum of 1 to *x* is $x(x+1)/2$. We are computing the sum with an upper limit of *n-1*, so this becomes $(n\text{-}1)(n\text{-}1+1)/2$ which simplifies to $n(n\text{-}1)/2$.) The number of additions grows with the square of the length of the list. As computer scientists, we say that the running time of this algorithm grows quadratically with the size of its input.

In thinking about the problem, we should see that there is a faster way to accomplish the same goal–-use an accumulator. At each point in the list, we can add the previous month's available sick days to the amount accumulated this month. Rather than recompute the previous month's total, we can accumulate it as we traverse the list.

```
;; avail-days-accum: list-of-number number → list-of-number
;; Purpose: uses an accumulator to compute sick days available at the start of
;;              each month
(define (avail-days-accum  alon  accum-days)
  (cond
    [(empty?  alon) empty]
    [else (cons (+ first alon)  accum-days
                (avail-days-accum (rest alon) (+ (first alon) accum-days)))]))
```

As with any function that uses an accumulator, we need to be careful to invoke it the first time with the right value. Thus, to use our initial example, we would invoke it with a sequence such as

```
(avail-days-accum (list 2 –1 1) 0))
```

To ensure that it is invoked correctly, we can wrap it inside a local and hide it inside the implementation of **available-days**.

```
;; available-days: list-of-number → list-of-number
;; Purpose: consumes a list of sick days earned per month and produces a list
;;              of sick days available per month
(define (available-days alon)
   (local [(define (avail-days-accum  alon  accum-days)
             (cond
               [(empty?  alon) empty]
               [else (cons (+ first alon)  accum-days
                           (avail-days-accum (rest alon)
                                             (+ (first alon) accum-days)))]))]
     (avail-days-accum  alon 0)))
```

Another Example

Let's write a program **reverse** that consumes a list (of alpha) and produces a list (of alpha) that has the same elements in the reverse order. That is, the first element of the input becomes the last element of the output. Again, we'll start with a version based on structural recursion.

```
;; reverse: list-of-alpha → list-of-alpha
;; Purpose:  constructs the reverse of a list of items
(define (reverse aloa)
   (cond
     [(empty?  aloa)  empty]
     [(cons? aloa)
       (make-last-item (first aloa) (reverse (rest aloa)))]))
```

```
;; make-last-item:  alpha  list-of-alpha  → list-of-alpha
;; Purpose:  adds an element to the end of a list
(define (make-last-item  an-elt   aloe)
   (cond
      [(empty?  aloe)  (list an-elt)]
      [(cons?  aloe)     (cons (first aloe)
                               (make-last-item an-elt (rest aloe)))]))
```

What happens on a call to reverse?

```
(reverse (list 1 2 3))
= (make-last-item 1 (reverse (list 2 3)))
= (make-last-item 1 (make-last-item 2 (reverse (list 3))))
= (make-last-item 1 (make-last-item 2 (make-last-item 3 (reverse empty))))
= (make-last-item 1 (make-last-item 2 (make-last-item 2 empty)))
… and then it starts returning…
```

Again, to process all of these nested calls to make-last item, we will end up traversing the end of the original list many times. This begins to look like the last example, right down to the fact that it seems to waste a lot of computation.

Can we use an accumulator to simplify the program? Compare the structural version of available-days to the structural version of reverse. Notice that they both pass the balue returned by a recursive call to another recursive procedure. This is precisely what gives rise to the kind of quadratic behavior that we observed when we hand evaluated the examples. It gives rise to a simple rule for when to consider using an accumulator.

Consider using an accumulator if the program processes the return value of a recursive call with another recursive call

Next lecture, we'll look at a process for transforming a program based on structural recursion into one that uses an accumulator, *provided that the original program fits our rule.*