COMP 210, Spring 2001
Lecture 17: Programming with Accumulators

Reminders:

1. Exam 2 will be give a week from Wednesday at 7 P.M. in DH 1055.

Assume that we are given a list of numbers

```
(x1 … xn)
```

and that we need to construct the list of partial sums

```
(s1 … sn)
```

where

```
si  = x1 + x2 + … + xi
```

Let **`partial-sums`** be the name of this function. Then:

```
  (partial-sums (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)))))))
= (cons 2
      (add-to-each 2
          (cons -1
              (add-to-each -1 (cons 1 (add-to-each 1 empty)))))))

= (cons 2 (add-to-each 2 (cons -1 (add-to-each -1 (cons 1 empty))))))
= (cons 2 (add-to-each 2 (cons -1 (cons 0 empty))))
= (cons 2 (cons 1 (cons 2 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-1*st element to 1 element. The total number of additions that add-to-each performs is

$$n\text{-}1 + n\text{-}2 + n\text{-}3 + \ldots + 2 + 1 \;=\; n*(n\text{-}1)/2$$

(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-1)*(n-1+1)/2* which simplifies to *n*(n-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–write a helper function that takes an extra argument that accumulates the partial sum for elements already visited. This argument is called an accumulator. At each point in the list, we can add the partial sum for an element to its successor to produce the partial sums for all elements. Rather than recomputing the partial sum for each element of the list, we can accumulate it as we traverse the list.

```
; partial-sums-accum: list-of-number number → list-of-number
;   uses an accumulator to compute the list of partial sums for alon
(define (partial-sums-accum alon accum)
   (cond
      [(empty? alon) empty]
      [else
        (local [(define new-accum (+ (first alon) accum))]
          (cons new-accum
            (partial-sums-accum (rest alon) new-accum)))]))
```

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**.

```
; partial-sums: list-of-number → list-of-number
;   consumes a list of numbers and produces a corresponding list of partial
;   sums
(define (available-days alon)
  (local
      [(define (partial-sums-help alon accum)
          (cond
           [(empty? alon) empty]
           [else
             (local
                 [(define new-accum (+ (first alon) accum))]
               (cons new-accum
                  (partial-sums-help (rest alon) new-accum)))]))]
    (partial-sums-help alon 0)))
```

Another Example

Let's write a program **reverse** that consumes a list (of T) and produces a list (of T) 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: <T> (list-of T) → (list-of T)
;   constructs the reverse of a list of items
(define (reverse aloi)
  (cond [(empty? aloi) empty]
        [(cons? aloi) (make-last-item (first aloi) (reverse (rest aloi)))]))

(define (make-last-item i aloi)  (append aloi (list i)))
```

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)))
= …
```

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 value 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.*