COMP 210, Spring 2001
Lecture 18: Deciding When to Use an Accumulator

Reminders:

1. Exam: next Wednesday evening at 7 P.M.

Review

We wrote a program that reverses a list. Again, the first version was inefficient, because it involved passing the value returned from one recursive procedure to an invocation of another recursive procedure. This led to a COMP 210 rule of thumb:

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

You know by now that we want to keep the interface to reverse intact–-both for the sake of programs that already use reverse and to ensure that we pass the accumulator the correct initial value. Thus, you should be able to guess the basic skeleton for reverse:

```
;; reverse: <T> list of T → list of T
;; Purpose:  constructs the reverse of a list of items
(define (reverse  aloa)
   (local [(define (rev aloa accum)
             (cond
              [(empty? aloa)  …]
              [(cons?  aloa)
                    … (rev  (rest aloa) …
                            (first aloa) … accum …) ]))]
      (rev  aloa  … )  ))
```

Notice that the template for rev differs from our classic list template. We've moved the use of the first element of aloa into the recursive call. (In the classic list template, it occurs before the recursive call.) We've also added **`accum`** to that call as a parameter. Why? We know, from our extensive experience, that **`rev`** will probably combine **`(first aloa)`** with **`accum`** to form the new accumulator for the recursive call on **`rev`**.

We would like to think that we can simply fill in the rest of the body of the program from the template. Unfortunately, it's a little more complex than that. We need to figure out what the accumulator holds. (Otherwise, its hard to figure out how to use it and how to generate a new accumulator from the old one!) Before filling in the template, we need to answer these questions, **AND**, in true COMP 210 fashion, write down a statement that documents the contents of the accumulator. [Without such a statement, it is hard to understand an accumulator-based program.] This definition for the

accumulator gets written (on your homework, on tests, in every program you develop) immediately above the definition of the function that uses the accumulator.

```
; reverse: <T> list of T → list of T
;   constructs the reverse of a list of items
(define (reverse  aloa)
   (local
       [; accum: contains the reversed list of items in aloa
        ; that precede alist
        (define (rev alist accum)
          (cond
             [(empty? alist)  …]
             [(cons?  alist)
                … (rev (rest alist) …
                … (first alist) … accum …)]))]
     (rev aloa … )))
```

The comment should accomplish two things. It should describe the type of the accumulator value (so that we can write code) and it should describe the useful property of the accumulator on which the program relies. In this case, the comment makes it clear that **accum** is a list of items derived from **aloa** (it is a list of alpha), and that, on each call to **rev**, **accum** holds the reverse of that part of **aloa** that has already been processed and discarded (to form **alist**).

This property is an invariant that holds before and after each call to **rev**.

Now, we can fill in the rest of the program.

```
; reverse:  <T> list of T → list of T
; Purpose:  constructs the reverse of a list of items
(define (reverse  aloa)
   (local [; accum: contains the reversed list of items in aloa
           ; that precede alist
       (define (rev alist accum)
         (cond [(empty? alist) accum]
               [(cons? alist)
                (rev (rest alist)
                (cons (first alist) accum))]))]
     (rev aloa empty)))
```

So, what was the process:

1. Write the structural recursion version
2. Write down the template for an accumulator version, preserving the interface and hiding the accumulator version inside a local. [This allows us to initialize the accumulator in a safe and certain fashion.]
3. Decide what the accumulator should hold and write down a comment that documents the accumulator's type and states the invariant on which code relies for correctness.
4. Fill in the details.

Yet Another Example

Consider the program sum that computes the sum of a list of numbers. Can we write an accumulator version? Is there a reason to do so?

```
;; sum: list-of-number → number
;; Purpose: computes the sum of all numbers in the list
(define (sum alon)
  (cond [(empty? alon) 0]
        [(cons? alon) (+ (first alon) (sum (rest alon)))]))
```

Notice that this doesn't have the case of passing the result of one recursion to another recursion. We write down the skeleton:

```
;; sum: list-of-number → number
;; Purpose: computes the sum of all the numbers in a list
(define (sum orig-alon)
  (local
      [; accum:
       (define (sum-accum alon accum)
         (cond [(empty? alon)  … ]
               [(cons? alon)
                … (sum-accum  (rest alon)
                … (first alon) … accum …)]))]
    (sum-accum orig-alon … )))
```

Then, we fill in the rest of the code and comments

```
;; sum: list-of-number → number
;; Purpose: computes the sum of all the numbers in a list
(define (sum orig-alon)
   (local
       [; accum: contains the sum of all numbers in orig-alon
        ;        that precede the alon
         (define (sum-accum alon accum)
           (cond
             [(empty? alon) accum ]
             [(cons? alon)
              (sum-accum (rest alon) (+ (first alon) accum))]))]
     (sum-accum alon 0 )))
```

Is this example any better (or any different) than the original version? Both perform essentially the same amount of work.

Original version:

```
    (sum (list 2 5 3 7))
= (+ 2 (sum (list 5 3 7)))
= (+ 2 (+ 5 (sum (list 3 7))))
= (+ 2 (+ 5 (+ 3 (sum (list 7)))))
= (+ 2 (+ 5 (+ 3 (+ 7 (sum empty))))))
= (+ 2 (+ 5 (+ 3 (+ 7 0))))
= (+ 2 (+ 5 (+ 3 7)))
= (+ 2 (+ 5 10))
= (+ 2 15)
= 17
```

Accumulator version:

```
    (sum (list 2 5 3 7))
= (sum-accum (list 2 5 3 7) 0)
= (sum-accum (list 5 3 7 ) 2)
= (sum-accum (list 3 7) 7)
= (sum-accum (list 7) 10)
= (sum-accum empty 17)
= 17
```

The evaluations have different shapes (when they are written out). The structural version builds up a series of pending additions until it hits empty, then performs all of the additions on the way back from the call. After each recursive call finishes, an addition is performed inside the incarnation of the function that initiated the call. The accumulator version simply and directly returns the result of the recursive call, so it doesn't build up this context of pending computation. The hand evaluation is simpler (and easier to understand). Does it perform any fewer additions? **NO**.

However, this can be more efficient to execute. Imagine a list of 10,000 numbers, or 10 million numbers. The space required to hold this pending

context can grow quite large, to the point where it can exhaust the memory resources available in your machine. The accumulator version avoids stacking up this pending context, so it avoids the issue. Recursion of this form is called *tail-recursion*.