COMP 210, FALL 2000
Lecture 29: One Last Look at Accumulators

Reminders:

1. Missionaries & Cannibals alert: Due 12 April 2000.

Review

We looked at a couple of simple examples of accumulator-based computations–-both **reverse** and **sum**. Both are simple programs, but they let us formulate a template for accumulator-based computations and a process for building them. (See the lecture notes).

To finish up class, we looked at the rewrite sequences for **sum** and **sum-accum**. We noted that **sum** builds up a left context of computation that must be completed after the recursive calls return, while **sum-accum** does not. I asserted, without proof, that sometimes the **sum-accum** behavior is preferable. (With enough pending left context, the computation can exhaust the available memory and cause DrScheme to fail. In some implementations, there will be a speed advantage to the accumulator version, as well–-although that is *highly* implementation specific.)

Finally, I want to reiterate that I do not want you to convert every structural recursion to an accumulator computation. Instead, you should keep in mind that accumulators can be used (1) to remember what a function has done in previous parts of the current computation, (2) to speed up some kinds of functions, and (3) to avoid building up a large amount of pending context. This last issue arises in some large computations, where memory becomes a fixed constraint on the size of computation that can be performed.

When it arises (you'll notice it when DrScheme runs out of memory), you can rewrite the program into a form where it uses an accumulator the need to represent all of that pending left context.

Accumulators on Trees

Consider a binary tree of numbers.

```
;; a bnt (binary-number-tree) is either
;;   – empty, or
;;   – (make-bnt num left right)
;; where num is a number and left & right are bnts
(define-struct bnt (num left right))

;; Examples
empty
(make-bnt 1 (make-bnt 2 (make-bnt 3 empty empty) (make-bnt 4 empty empty))
                (make-bnt 5 empty empty))
```

What if we want to sum all of the numbers in a bnt?

```
;; sumtree: bnt → number
;; Purpose: sum all the numbers in a binary number tree
(define (sum abnt) …)
```

According to our methodology & the process developed last class, we start with the structural solution:

```
;; sumtree: bnt → number
;; Purpose: sum all the numbers in a binary number tree
(define (sum abnt)
  (cond
     [(empty? abt)  0]
     [else (+  (bnt-num   abt)
               (sumtree (bnt-left abt))
               (sumtree (bnt-right abt)))] ) )
```

What's this? A second call to st-acc? Needed because we have 2 subtrees !?!

Next, we write down the accumulator template for sum:

```
;; sumtree: bnt → number
;; Purpose: sum all the numbers in a binary number tree
(define (sumtree abt)
    (local [ ;; accum: …
             (define (st-acc  atree accum)
                  (cond  [(empty? atree)  …]
                         [else    (st-acc  … (bnt-left atree) …
                                             (bnt-num atree) .. accum …)
                                  (st-acc  … (bnt-right atree) …
                                             (bnt-num atree) … accum …)] )) ]
           (st-acc   abt  …) ))
```

What happened in the else clause of the cond? We have two recursive calls to **st-acc**. Clearly, the program must process both the left and the right subtree of any node if it is to sum the entire tree. However, the code that is written doesn't make a lot of sense.

Maybe the accumulator invariant will make things clear. What does the accumulator hold? If we follow all precedent, it should hold the sum of all the numbers that st-acc has already seen—all the bnt numbers in nodes of the tree that st-acc has already visited.

Write in the accumulator comment & invariant on the board –-
;; accum: sum of numbers from already visited nodes in abt

Refers to abt not atree

This makes it easy to fill in the empty case; it should simply return accum. The initial accum value is 0. Does it help us with the problem of the multiple subtrees? Not really.

We could simply add the results returned by the two recursive calls, producing code that looks like:

```
                    [else    (+  (st-acc  (bnt-left atree)    (+ (bnt-num atree) accum ))
                                 (st-acc  (bnt-right atree)  (+ (bnt-num atree) accum))]
```

This adds in (bnt-num atree) twice, but that's easy to fix.

```
                    [else    (+  (st-acc  (bnt-left atree)     accum )
                                 (st-acc  (bnt-right atree)  (+ (bnt-num atree) accum))]
```

Is this what we want? When we evaluate it on an example, it leaves behind exactly the kind of left-context that accumulators can avoid. The addition is pending left context. (When we're evaluating the first recursive call, the second one is pending right context, as well.) This isn't the nice clean behavior that we hoped to see.

Can we fix it? The solution is to ***thread*** the tree with the accumulator, by working the second recursive call into the computation of the accumulator for the first recursive call. This produces a version of the else clause that looks like

```
          [else (st-acc (bnt-left atree)  (st-acc (bnt-right atree) (+ (bnt-num atree) accum)))]
```

To recap, the full program is

```
          ;; sumtree: bnt → number
          ;; Purpose: sum all the numbers in a binary number tree
          (define (sumtree abt)
              (local [ ;; accum: …
                        (define (st-acc  atree accum)
                             (cond   [(empty? atree)  accum]
                                     [else (st-acc (bnt-left atree)
                                            (st-acc (bnt-right atree) (+ (bnt-num atree) accum)))]
                             ))]
                    (st-acc   abt  0) ))
```

How does this work? Consider the example.

Work it out graphically. The trace is

```
(St-acc
  ( make-bnt 1 (make-bnt 2 (make-bnt 3 () ()) (make-bnt 4 () ())) (make-bnt 5 () ()))
  0)

 ➔ (St-acc (make-bnt 5 () ()) 1);
    ➔ (St-acc () 6)                                  5+1
    ➔ (St-acc () 6)
 ➔ (St-acc (make-bnt 2 (make-bnt 3 () ()) (make-bnt 4 () ())) 6)
     ➔ (St-acc (make-bnt 4 () ()) 8)                       + 2
        ➔ (St-acc () 12)
        ➔ (St-acc () 12)
  ➔ (St-acc (make-bnt 3 () ()) 12)                          + 4
        ➔ (St-acc () 15)
        ➔ (St-acc () 15)

                                                           + 3
```

You should work a couple of examples to convince yourself that it works correctly and to draw out the threading…

On these examples, the technique looks like a lot of intellectual activity for very small improvements in either conceptual complexity or actual behavior. If, however, you tried to write a program that returned the number of occurrences of a given name in the Houston phonebook, the difference between the straight-forward formulation of the accumulator computation and the threaded version would be significant.