Administrative Notes

Homework 9

- Eight days left
- Eight subparts
- If you haven't started, you need to start now

A Final Word on maxacc

We've seen so many versions of max

- Exponential version
- Linear version that introduced local
- Accumulator version
- Version on binary trees from the test

Variants of max

The linear version based on local

```
;; max1: nelon -> number
(define (max1 anelon)
  (cond
    [(empty? (rest anelon)) (first anelon)]
    [(cons?  (rest anelon))
     (local [(define maxrest (max1 (rest anelon)))
             (define thisone (first anelon))]
       (cond
         [(<= thisone maxrest) maxrest]
         [else thisone]
         ))]
    ))
```


Variants of max

The accumulator version

```
;; max2: list -> number
(define (max2 anelon)
  (local [(define (maxacc anelon acc)
            (cond
              [(empty? anelon) acc]
              [(cons? anelon)
               (cond
                 [(<= acc (first anelon))
                  (maxacc (rest anelon) (first anelon))]
                 [else (maxacc (rest anelon) acc)] )]
              ))]
    (maxacc anelon -1)))  ;; empty list returns -1
```

Variants of max

Version derived from the test

```
;; max3: list -> number
(define (max3 anelon)
  (cond
    [(empty? anelon) -1]
    [(cons?  anelon) (max (first anelon) (max3 (rest anelon)))]
    ))
```

```
(define (max n1 n2)
  (cond
     [(> n1 n2)  n2]
     [else       n1]
   ))
```

- *Uses max to encapsulate the comparison & decision*
- *Avoids the creation of maxrest by using a function argument*

This looks so much simpler

- *Why not write it this way? (are accumulators a waste of effort?)*

Variants of max

We can time these versions of max (list of 1 to 10,000)

- Max-local: range 1,183 milliseconds to 1,984 milliseconds
- Max-acc: range 150 to 167 milliseconds
- Max-max: range from 350 to 884 milliseconds

The lessons:

- Differences in execution time are noticeable
  - → Order of magnitude between best & worst "linear" max
- Variations are due to DrScheme's memory state
- Comparing best time against best time
  - → Max-acc is 1/2 max-max and 1/8 max-local !

Accumulators on trees

What about largest from the test?

- Found the largest number in a binary tree

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


;; template …
(define (f  abnt …)
   (cond
     [(empty? abnt)   …]
     [(bnt?     abnt)    … (bnt-num abnt) …
                         … (f (bnt-left abnt) … ) …
                         … (f (bnt-right abnt) … ) … ]
   ))
```


Accumulators on trees

Filling in the template for largest

```
;; largest: bnt -> number
;; Purpose: return the largest number in the bnt, or zero if
;;        the bnt is empty
(define (largest abnt)
   (cond
     [(empty? abnt)                          0]
     [(bnt?     abnt)    (max (bnt-num abnt)
                             (largest (bnt-left abnt))
                             (largest (bnt-right abnt)))]
   ))
```

This is the answer I was expecting

It works.

Accumulators on trees

Can we make it faster using an accumulator?

- What does an accumulator on a tree do?
- What does the code look like?
- Does it help?

Start with structural version

- See last slide
- Write down the accumulator template

Accumulators on trees

Accumulator template for bnt

```
(define (largest abnt)
  (local [;; acc holds …
          (define (lhelper atree acc)
            (cond [(empty? atree)  …]
                  [else  … (lhelper … (bnt-left atree) …
                                      … (bnt-num atree) … acc …)
                           (lhelper … (bnt-right atree) …
                                      … (bnt-num atree) … acc …) … ]
            )) ]
    (lhelper abnt … )  ))
```

What happened in the else clause of lhelper ?

- *Need two calls for two subtrees*
- *Need some way to combine the results*

Accumulators on trees

Filling in the template

```
(define (largest abnt)
  (local [;; acc holds largest number seen in nodes visited so far
          (define (lhelper atree acc)
            (cond [(empty? atree)  …]
                  [else  … (lhelper … (bnt-left atree) …
                                      … (bnt-num atree) … acc …)
                           (lhelper … (bnt-right atree) …
                                      … (bnt-num atree) … acc …) … ]
            )) ]
        (lhelper abnt … )  ))
```


Accumulators on trees

Filling in the template

```
(define (largest abnt)
  (local [;; acc holds largest number seen in nodes visited so far
          (define (lhelper atree acc)
            (cond [(empty? atree)    acc ]
                  [else  … (lhelper … (bnt-left atree) …
                                      … (bnt-num atree) … acc …)
                           (lhelper … (bnt-right atree) …
                                      … (bnt-num atree) … acc …) … ]
            )) ]
        (lhelper abnt … )  ))
```

Accumulators on trees

Filling in the template

```
(define (largest abnt)
  (local [;; acc holds largest number seen in nodes visited so far
          (define (lhelper atree acc)
            (cond [(empty? atree)    acc ]
                  [else (max (lhelper (bnt-left atree)
                                      (max (bnt-num atree) acc))
                             (lhelper (bnt-right atree)
                                      (max (bnt-num atree) acc)) ) ]
            )) ]
         (lhelper abnt -1 )  ))
```

This works. Is it what we want?

It leaves behind the kind of left context (the outer max) that we tried to avoid by introducing accumulators (not tail-recursive!)

Accumulators on trees

How do we avoid the dreaded pending context?

- We can thread the tree
  - → Work 2nd recursive call into computation of accumulator for the 1st recursive call
  - → Complex notion
  - → Replace (max (bnt-num atree) acc) with

    (lhelper (bnt-right atree) (max (bnt-num atree) acc))

Accumulators on trees

Filling in the template

```
(define (largest abnt)
  (local [;; acc holds largest number seen in nodes visited so far
          (define (lhelper atree acc)
            (cond [(empty? atree)    acc ]
                  [else (lhelper (bnt-left atree)
                                 (lhelper (bnt-right atree)
                                          (max (bnt-num atree) acc) ) ) ]
            )) ]
    (lhelper abnt -1 )  ))
```

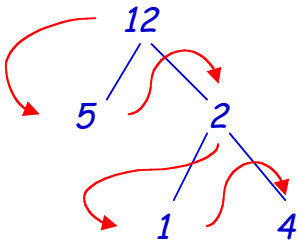
This works. It has no pending left context!

What did it do?

Accumulators on trees

Threading the tree

```
(define TestBnt
  (make-bnt 12
            (make-bnt 5 empty empty)
            (make-bnt 2
                      (make-bnt 1 empty empty)
                      (make-bnt 4 empty empty)))
```



Is this any faster than the version from the template?

- *Need some large test trees to find out*
- *Need a program for generating them*
- *Maybe next class …*