Administrative Notes

Exam

- Solutions will be posted today or tomorrow
- Look at the solutions

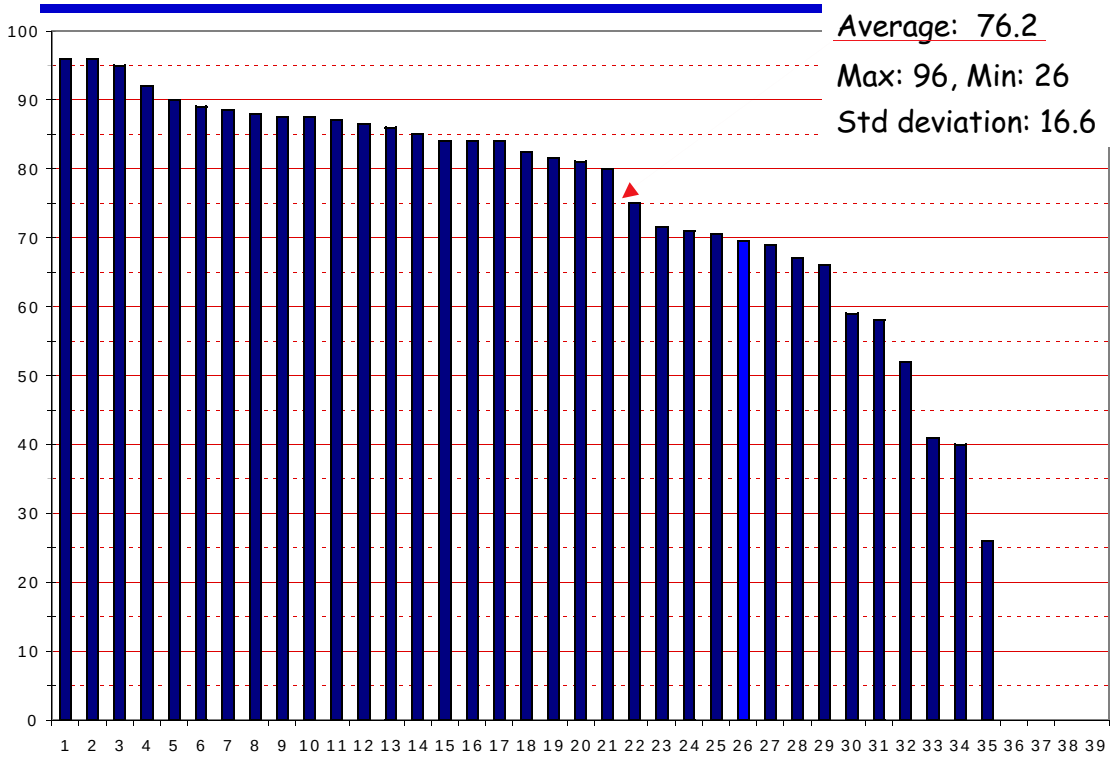
Homework 9 *(Ex. 32.2.1 — 32.2.8 in book)*

- Due Wednesday, April 10, 2002 in class
- Do one sub-problem each day and you will finish early
- Procrastinate and you will not finish

Labs this week as normal

- Challenge lab?

COMP 210, Spring 2002, Second Exam



Finishing up accumulators

The example with reverse was tortured *(my fault)*

- Can we write another classic program with an accumulator?

  → Let's try max, one of our favorite examples

```
;; a non-empty-list-of-number (nelon) is either
;;   — (cons f r) where f is a number and r is empty, or
;;   — (cons f r) where f is a number and r is a nelon
;; We will use Scheme's built-in list constructor to implement nelons
```

```
;; maxacc: nelon -> number
;; Purpose: returns the largest entry in a non-empty list of numbers
(define (maxacc anelon) … )
```


Finishing up accumulators

Max, again

```
;; maxacc: nelon -> number
;; Purpose: returns the largest entry in a non-empty list of numbers
(define (maxacc anelon) … )
```

How do we proceed?

- With an accumulator, can pass along largest element so far
- What does helper do?

```
;; maxh: nelon number -> number
;; Purpose: returns the larger of acc and (max-of-list anelon)
;; acc holds the largest element seen so far
(define (maxh anelon acc) … )
```

Finishing up accumulators

Focusing on maxh

```
;; maxh: nelon number -> number
;; Purpose: returns the larger of acc and (max-of-list anelon)
(define (maxh anelon acc)
  (cond
    [(empty? anelon)              acc]
    [(> (first anelon) acc)       (maxh (rest anelon) (first anelon))]
    [(else                        (maxh (rest anelon) acc)]
  ))
```


Finishing up accumulators

Focusing on maxh

```
;; maxh: nelon number -> number
;; Purpose: returns the larger of acc and (max-of-list anelon)
(define (maxacc anelon acc)
  (cond
    [(empty? anelon)              acc]
    [(> (first anelon) acc)       (maxh (rest anelon) (first anelon))]
    [(else                        (maxh (rest anelon) acc)]
  ))
```

But wait

- Maxh tests (empty? anelon)
- How can a nelon be empty?
- We subtly changed the problem & the contract

Finishing up accumulators

Maxh operates on a list

```
;; maxh: alon number -> number
;; Purpose: returns the larger of acc and (max-of-list alon)
(define (maxh alon acc)
  (cond
    [(empty? alon)                 acc]
    [(cons? alon)
     (cond
       [(> (first alon) acc)       (maxh (rest alon) (first alon))]
       [else                       (maxh (rest alon) acc)]    )]
```

Now, …

- maxacc takes a nelon & uses (first anelon) as initial accum'r
- maxh takes a list & returns a number
  - → Uses (empty? alon) test to return accumulator value

Finishing up accumulators

Putting it together

```
;; maxacc: nelon -> number
;; Purpose: returns the largest entry in a non-empty list of numbers
(define (maxacc anelon)
  (cond
    [(empty? (rest anelon))  (first anelon)]
    [(cons?   (rest anelon))
     (local
       [ ;; maxh: alon number -> number
         ;; Purpose: returns the larger of acc and (max-of-list anelon)
         (define (maxh alon acc)
           (cond
             [(empty? alon)                 acc]
             [(> (first alon) acc)          (maxh (rest alon) (first alon))]
             [(else                         (maxh (rest alon) acc)] ))]
       (maxh (rest anelon) (first anelon)) )]
  ) )
```

Finishing up accumulators

An aside

- We can think of this example as a template for accumulator programs over lists

```
;; f :  list of alpha -> beta
(define (f alist)
   (local [;; acc holds …
           ;; g : alist  -> beta
           ;; Purpose: g does something good
           (define (g alist acc)
             (cond
               [(empty? alist)            …]
               [(cons? alist)
                     … (g   (rest alist)
                            … (first alist)
                            … acc )       ])) ]
        (g  alist … )]     ) )
```

This being 210, you need a comment that explains the accumulator's contents

Need to figure out what the accumulator holds, and how to use it in g

Finishing up accumulators

What's the point?

- Old version of max worked
  - → Used local to make it run in linear time (rather than 2^N)

```
;; maxclassic: nelon -> number
;; Purpose: rehash max, again
(define (maxclassic anelon)
 (cond
   [(empty? (rest anelon)) (first anelon)]
   [(cons?  (rest anelon))
    (local [(define maxrest (maxclassic (rest anelon)))]
      (cond
        [(> (first anelon) maxrest) (first anelon)]
        [else maxrest]
        ))]
  ))
```

Finishing up accumulators

What's the point?

- Old version of max worked
  - → Used local to make it run in linear time (rather than 2^N)
- Does maxacc differ from maxclassic in any useful way
  - → Consider their behavior on (list 1 2 3 4)

This is a point I tried to make with reverse last class

Using the stepper made it particularly hard to see the point

Finishing up accumulators

Consider the evaluation of each function

(maxclassic (list 1 2 3 4))

⇒ defines maxrest0 as (maxclassic (list 2 3 4)
  ⇒ defines maxrest1 as (maxclassic (list 3 4))
    ⇒ defines maxrest2 as (maxclassic (list 4))
      ⇒This returns 4
    ⇒ evaluates the cond and returns 4
  ⇒ evaluates the cond and returns 4
⇒ evaluates the cond and returns 4

Finishing up accumulators

Consider the evaluation of each function

(maxacc (list 1 2 3 4))

⇒ finds (rest anelon) is non-empty & enters local
⇒ evaluates (maxh (list 2 3 4) 1)
  ⇒ evaluates (maxh (list 3 4) 2)
    ⇒ evaluates (maxh (list 4) 3)
      ⇒ evaluates (maxh empty 4) & returns 4
    ⇒ returns 4
  ⇒ returns 4
⇒ returns 4

What's the difference?

Finishing up accumulators

Consider the evaluation of each function

(maxclassic (list 1 2 3 4))

⇒ defines maxrest0 as (maxclassic (list 2 3 4)
  ⇒ defines maxrest1 as (maxclassic (list 3 4))
    ⇒ defines maxrest2 as (maxclassic (list 4))
      ⇒This returns 4
    ⇒ evaluates the cond and returns 4
  ⇒ evaluates the cond and returns 4
⇒ evaluates the cond and returns 4

This context involves further evaluation

Scheme has lots of pending context after the recursive call

Finishing up accumulators

Consider the evaluation of each function

(maxacc (list 1 2 3 4))

⇒ finds (rest anelon) is non-empty & enters local
⇒ evaluates (maxacc (list 2 3 4) 1)
  ⇒ evaluates (maxacc (list 3 4) 2)
    ⇒ evaluates (maxacc (list 4) 3)
      ⇒ evaluates (maxacc empty 4) & returns 4
    ⇒ returns 4
  ⇒ returns 4
⇒ returns 4

This context just returns the value

Scheme has (almost) no context after the call

Finishing up accumulators

Does this matter?

- In large evaluations, that extra context adds up
- Takes space (in DrScheme) and time
- Can become a source of inefficiency

Tail recursion

- A tail-recursion returns the value of a self-recursive call
  - → No further computation
- This is a particularly efficient form of recursion
  - → Most translators (like DrScheme) optimize for this case

Finishing up accumulators

Another use for accumulators

- We can use an accumulator to transform a program into tail-recursive form
- This is an efficiency hack
  - → But can be an important one