Administrative Notes

Next Exam

- Two potential dates
  - → Monday, March 18, 2002 or Wednesday, March 20, 2002
- Two hour exam (*in the evening*)

Sections in the book

- Family trees were 14-16
- Multiple complex arguments was 17

Happy "Thinking Day"

- Shared birthday of Lord & Lady Baden Powell
- Day for reflection on the true meaning of "scouting"

Programs with Multiple Complex Arguments

So far, three cases

- Two arguments, one is not inspected
  - → Use template for the inspected argument

  Example: **append**
- Two arguments, with simplifying property
  - → Lists of same length
  - → Trees of identical shape

  Example: **make-points**
  - → Use one argument to control the flow of the program
- Two arguments, no simplifying assumptions
  - → Build a table of the cases
  - → Develop tests for each case

  Example: **merge**
  - → Use a **cond** with a clause for each case
  - → Lots of opportunities to recur

Programs with Multiple Complex Arguments

Sorting with **merge**

(list c1 c2 c3 c4 c5 c6 c7 c8) ⇒

(list c1) (list c2) (list c3) (list c4) (list c5) (list c6) (list c7) (list c8)
⇓ merge ⇓ merge ⇓ merge ⇓ merge
(list ci cj) (list ck cl) (list cm cn) (list co cp)
⇓ merge ⇓ merge
(list ci cj ck cl) (list cm cn co cp)
⇓ merge
(list ci cj ck cl cm cn co cp)

Question becomes, can we generate singleton lists from a list?

- We do not yet have the tools to do this
- Next section of 210 examines a paradigm that can do this

Moving on …

COMP 210 has mid-term grades due next week …

- Need software to help compute grades
- Assume that I have a list of scores

```
;; a list-of-numbers is either
;;  – empty, or
;;  – (cons f r), where f is a number and r is a list-of-numbers
;; We will use Scheme's built-in list constructor for list-of-numbers
```

- Develop **best-score: list-of-number→number**

Best-score

Example

(list 72 84 99 53 88 75 104 62)

Work from standard list template

```
(define (f a-los …)
   (cond
      [(empty? a-los)                    … ]
      [(cons?   a-los)
            … (first a-los) …
            … (f  (rest a-los) … ) … ]
    ))
```


Best-score

Filling in the template

```
;; best-score:  list-of-number -> number
;; Purpose: return the best score in the list
(define (best-score a-los …)
   (cond
      [(empty? a-los)          ]
      [(cons?   a-los)
            … (first a-los) …
            … (best-score  (rest a-los) … ) … ]
    ))
```

What goes here?

Deep philosophical question

- What is (best-score empty) ?
  - → Since it's a test, we have a lower bound of zero
  - → Can return zero

Best-score

Filling in the template

```
;; best-score:  list-of-score -> number
(define (best-score a-los)
   (max-of-list  a-los  0))

;; bigger: number number -> number
(define (bigger n1 n2)
  (cond [(<= n1  n2)          n2]
        [else                 n1] ))

;; max-of-list:  list number -> number
(define (max-of-list a-list lb)
   (cond  [(empty? a-list)                lb ]
          [(cons?   a-list) (bigger (first a-los) (max-of-list (rest a-los) lb) ]
    ))
```

Helper functions to make it clean

The Real Problem

The lower bound let us sidestep the issue

What if we do not have a lower bound?

- (max empty) must return a number
- There is no good answer for this one
  - → - ∞ is smaller than any other number
  - → How can a program that uses max tell if the list actually contained - ∞ or not?

We need another answer to this quandry

Non-empty lists

A non-empty list lets us finesse the problem in a more rigorous way

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

;; template for nelon
(define  (f  a-nelon  … )
   (cond
      [(empty?  (rest a-nelon))          … (first a-nelon)… ]
      [(cons?    (rest a-nelon))
            … (first a-nelon) …
            … (f  (rest a-nelon) … ) … ]
   ))
```


Non-empty lists

Finding the maximum of a nelon is easier

```
;; max-of-nelon:  nelon -> number
(define  (max-of-nelon  a-nelon)
   (cond
      [(empty?  (rest a-nelon))          (first a-nelon) ]
      [(cons?    (rest a-nelon))
       (cond
            [( >= (first a-nelon)(max-of-nelon (rest a-nelon))) (first a-nelon)]
            [else (max-of-nelon (rest a-nelon))]
       ) ]
   ))
```

We restricted the domain of the inputs to avoid the tricky case – an old and time-honored trick!

Non-empty lists

What's wrong with max-of-nelon?

```
;; max-of-nelon:  nelon -> number
(define  (max-of-nelon  a-nelon)
   (cond
      [(empty?  (rest a-nelon))        (first a-nelon) ]
      [(cons?    (rest a-nelon))
       (cond
           [( >= (first a-nelon) (max-of-nelon (rest a-nelon)) ) (first a-nelon)]
           [else (max-of-nelon (rest a-nelon)) ]
       ) ]
   ))
```

We wrote this expression twice

Evaluating (max-of-nelon (rest a-nelon)) twice is wasteful

⇒ Efficiency is not our objective, but this can be ridiculous

Non-empty lists

How bad can it get?

- Let's try it
- (max (list 1 2 3 4 5 6)) 1
  - → Recurs twice on (list 2 3 4 5 6) 2
  - → Each of those recurs twice on (list 3 4 5 6) 4
  - → Each of those recurs twice on (list 4 5 6) 8
  - → Each of those recurs twice on (list 5 6) 16
  - → Each of those recurs twice on (list 6) 32
  - → Phew! This is getting ridiculous ⇒ 63
- It's a little better if the list is not in order, but …
  - → List of length n calls max $2^n - 1$ times
  - → This is too much
  - → List of length 7 would take 127 calls, 8 would take 255, …

What's the answer?

Need a new (for COMP 210) idea

- Save the value of max-of-list
- Makes it recur only once
- (max (list 1 2 3 4 5 6)) 1
  - → Recurs once on (list 2 3 4 5 6) 1
  - → Recurs once on (list 3 4 5 6) 1
  - → Recurs once on (list 4 5 6) 1
  - → Recurs once on (list 5 6) 1
  - → Recurs once on (list 6) 1
  - → And is done ⇒ 6
- Reduces work to n calls for list of length n
  - → Exponential savings in work are always worth pursuing

Next class

We will introduce a new piece of Scheme syntax

- It will let us save results of temporary computations
- It will improve the power and efficiency of our programs
- It will introduce a critical concept in Computer Science
  - → Lexical scoping

READ INTERMEZZO THREE FOR MONDAY