COMP 210, Spring 2000
First Exam, Solution Key

Problem 1

```
;; Part a

;; area-of-circle: number -> number
;; Purpose: takes the input number as the radius of a circle
;;          and produces the area of that circle
(define (area-of-circle R)
  (* pi R R))

;; Part b

;; area-of-rectangle: number number-> number
;; Purpose: takes a pair of input numbers and interprets them
;;          as the perpendicular sides of a rectangle.  Given
;;          those "side lengths", it computes the rectangle's area.
(define (area-of-rectangle s1 s2)
  (* s1 s2))

;; Parts c and d

;; type the expressions into DrScheme and use the stepper to execute
;; them
```

```
;; Data definitions for the rest of the test

;; an order is
;;  (make-order name TM AT SB)
;; where name is a symbol & TM, AT, & SB are all numbers
(define-struct order (name TM AT SB))

;; example orders
;; Todd ordered 3 Thin Mints & 2 Animal Treasures
;; (make-order 'Todd 3 2 0)
;; Tim ordered 1 of each
;; (make-order 'Tim 1 1 1)
;; Keith is on a diet
;; (make-order 'Keith 0 0 0)


;; a list-of-order is either
;;   - empty, or
;;   - (cons f r)
;; where f is an order and r is a list-of-order
;; [We will use the Scheme built-in lists, so no
;;  define-struct is needed.]

;; example list-of-order
;; The whole 2nd floor crew
;; (cons (make-order 'Todd 3 2 0)
;;       (cons (make-order 'Tim 1 1 1)
;;             (cons (make-order 'Keith 0 0 0) empty) ) )
```

Problem 2

```
;; Part a – Template for order

;; (define ( f ... an-order ...)
;;    (  ... (order-name  an-order) ...
;;       ... (order-TM    an-order) ...
;;       ... (order-AT    an-order) ...
;;       ... (order-SB    an-order) ... ))

;; Part b

;; order-boxes : order -> number
;; Purpose: consumes an order and produces the number of
;;          boxes required to satisfy the order
(define (order-boxes an-order)
  (+  (order-TM an-order)
      (order-AT an-order)
      (order-SB an-order)
      ))

;; Part c

;; I worked this one two ways, with inexact numbers ($3.50) and with
;; rational numbers (7/2) ... either one is acceptable.
;; [Note: I renamed that latter version to allow them to co-exist.]

;; order-price : order -> number
;; Purpose: consumes an order and produces the total price of the order,
;;          based on a price of $3.50 for Thin Mints, $3.75 for Animal
;;          Treasures, & $3.00 for Shortbreads
(define (order-price an-order)
  (+  (* (order-TM an-order) 3.50)
      (* (order-AT an-order) 3.75)
      (* (order-SB an-order) 3.00)
      ))

;; this version uses rational number, which may be more
;; comfortable than the inexact numbers, which appear with
;; the prefix #i...
;;
;; rational-order-price : order -> number
;; Purpose: consumes an order and produces the total price of the order,
;;          based on a price of 7/2 for Thin Mints, 15/4 for Animal
;;          Treasures, & 3 for Shortbreads

(define (rational-order-price an-order)
  (+  (* (order-TM an-order) 7/2 )  ;; could be 7/2
      (* (order-AT an-order) 15/4)  ;; could be 15/4
      (* (order-SB an-order) 3   )  ;; could be 3
      ))
```

Problem 3

```
;; Part a – Template for list-of-order
;; (define ( f  a-loo ...)
;;    (cond
;;       [(empty? a-loo)   ...]
;;       [(cons?  a-loo)
;;            ... (first a-loo) ...
;;            ... (f (rest a-loo)) ...]
;;    ) )
;; Part b
;; boxes-for-scout: list-of-order -> number
;; Purpose: consumes a list-of-order and produces the total
;;          number of boxes (of all kind) ordered
(define (boxes-for-scout a-loo)
  (cond
    [(empty? a-loo)      0]
    [(cons? a-loo)
       (+ (order-boxes (first a-loo))
          (boxes-for-scout (rest a-loo)))]
  ) )
;; Part c
;; big-order : list-of-order -> list-of-order
;; Purpose: consumes a list of order and produces a
;;          list containing the subset of those orders
;;          that purchase 6 or more boxes
(define (big-order a-loo)
  (cond
    [(empty? a-loo)      empty]
    [(cons? a-loo)
       (cond
         [(<= 6 (order-boxes (first a-loo)))
          (cons (first a-loo) (big-order (rest a-loo)))]
         [else (big-order (rest a-loo))]
       )]
    ))
```

Problem 4

```
;; subtotal : list-of-order symbol -> number
;; Purpose: consumes a list of order and a symbol that specifies
;;          one kind of cookie (i.e. 'ThinMints, 'AnimalTreasures,
;;          or 'Shortbreads).  It produces a number that is the
;;          total boxes of that kind ordered in the list
(define (subtotal a-loo flag)
  (cond
    [(empty? a-loo)   0]
    [(cons?  a-loo)
       (cond
         [(symbol=? 'ThinMints flag)
          (+ (order-TM (first a-loo)) (subtotal (rest a-loo) flag))]
         [(symbol=? 'AnimalTreasures flag)
          (+ (order-AT (first a-loo)) (subtotal (rest a-loo) flag))]
         [(symbol=? 'Shortbreads flag)
          (+ (order-SB (first a-loo)) (subtotal (rest a-loo) flag))]
       )]
    ))

;; Of course, you could also pull out the inner "cond" into a helper
;; function, following the rule discussed in class on Monday 2/14/00
;; This one is actually cleaner and more readable ...

;; alt-subtotal : list-of-order symbol -> number
;; Purpose: consumes a list of order and a symbol that specifies
;;          one kind of cookie (i.e. 'ThinMints, 'AnimalTreasures,
;;          or 'Shortbreads).  It produces a number that is the
;;          total boxes of that kind ordered in the list
(define (alt-subtotal a-loo flag)
  (cond
    [(empty? a-loo)   0]
    [(cons?  a-loo)
       (+ (boxes-by-kind (first a-loo) flag)
          (alt-subtotal (rest a-loo) flag)) ]
  ))

;; boxes-by-kind: order symbol -> number
;; Purpose: takes an order and a symbol representing a kind of
;;          cookie (i.e 'ThinMints, 'AnimalTreasures, or
;;          'Shortbreads) and produces the number of boxes of
;; that kind
(define (boxes-by-kind an-order flag)
  (cond
     [(symbol=? 'ThinMints flag) (order-TM an-order)]
     [(symbol=? 'AnimalTreasures flag) (order-AT an-order)]
     [(symbol=? 'Shortbreads flag) (order-SB an-order)]
  ))
```

Extra Credit

```
;; Sometimes, the template you need to use is the empty template.
;; In this case, you had all the pieces needed to put this function
;; together--you know the name field and can use subtotal to fill
;; in the rest of them.

;; summarize : list-of-order -> order
;; Purpose: consumes a list of order and produces a single order
;;          that summarizes the entire list.
(define (summarize a-loo)
  (make-order 'summary
              (subtotal a-loo 'ThinMints)
              (subtotal a-loo 'AnimalTreasures)
              (subtotal a-loo 'Shortbreads))
  )

;; Some of you relied on the template for list-of-order and came up
;; with this alternative

;; alt-summarize : list-of-order -> order
;; Purpose: consumes a list of order and produces a single order
;;          that summarizes the entire list.
(define (alt-summarize a-loo)
  (cond
    [(empty? a-loo)  (make-order 'summary 0 0 0)]
    [(cons?  a-loo) (make-order 'summary
                         (subtotal a-loo 'ThinMints)
                         (subtotal a-loo 'AnimalTreasures)
                         (subtotal a-loo 'Shortbreads))
     ]))

;; The problem with this particular solution is that it breaks out
;; the structure of the data-definition (by splitting the analysis
;; into a case for empty? and another for cons?) but doesn't follow
;; the recursion by invoking itself.  As of this point in the course,
;; you don't really have the tools to build this one directly.  Thus,
;; this solution represented a good attempt.  I gave it 4 out of 5
points.
```