**CS3733-B00: Software Engineering**

MzScheme Programming Exercises : Solutions

---

1. Program *duple*, which consumes a number $n$ and an item $x$ and produces a list consisting of $n$ copies of $x$.

```
;; Examples
> (duple 2 3)
(3 3)
> (duple 4 '(ho ho))
((ho ho) (ho ho) (ho ho) (ho ho))

(define (duple n x)
  (cond
    [(zero? n) empty]
    [else (cons x (duple (- n 1) x))]))
```

2. Program *invert*, which consumes a list of two-element lists and produces a list with each two-element list reversed.

```
;; Examples
> (invert '((a 1) (a 2) (b 1) (b 2)))
((1 a) (2 a) (1 b) (2 b))

(define (invert L)
  (cond
    [(empty? L) empty]
    [(cons? L) (cons (list (second (first L))
                           (first (first L)))
                     (invert (rest L)))]))
```

3. Program *swapper* which consumes two items and a list and returns the same list as the original but with all occurrences of the first item replaced with the second and vice-versa.

```
;; Examples
> (swapper 'a 'd '(a b c d))
(d b c a)
> (swapper 'a 'd '(a d () c d))
(d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
((y) x (z (y)))

(define (swapper s1 s2 L)
  (cond
    [(empty? L) empty]
    [(cons? L) (cons (cond
                       [(list? (first L)) (swapper s1 s2 (first L))]
                       [(symbol=? (first L) s1) s2]
                       [(symbol=? (first L) s2) s1]
                       [else (first L)])
                     (swapper s1 s2 (rest L)))]))
```

4. Program *mergesort*, which consumes two lists of numbers and returns the list sorted in ascending order. Use a helper function *merge* which consumes two lists sorted in ascending order and returns a sorted list of all numbers in the two input lists.

```
;; Examples
> (merge '(1 4) '(1 2 8))
(1 1 2 4 8)
> (merge '(35 62 81 90 91) '(3 83 85 90))
(3 35 62 81 83 85 90 90 91)
> (mergesort '(8 2 5 2 3))
(2 2 3 5 8)

(define (merge L M)
  (cond
    [(empty? L) M]
    [(empty? M) L]
    [(< (first L) (first M)) (cons (first L) (merge (rest L) M))]
    [else (cons (first M) (merge L (rest M)))]))

(define (before L n)
  (cond
    [(zero? n) empty]
    [else (cons (first L) (before (rest L) (- n 1)))]))

(define (after L n)
  (cond
    [(zero? n) L]
    [else (after (rest L) (- n 1))]))

(define (mid-length L) (truncate (/ (length L) 2)))

(define (mergesort L)
  (cond
    [(empty? L) empty]
    [(cons? L) (cond
                 [(empty? (rest L)) (list (first L))]
                 [else (merge (mergesort (before L (mid-length L)))
                              (mergesort (after L (mid-length L))))])]))
```

5. A set of shapes contains circles (defined by center point and radius), rectangles (defined by top-left corner, width, and height), and squares (defined by top-left corner and width).

   Write the following programs over shapes:

   (a) *area*, which consumes a shape and returns its area

   (b) *translate*, which consumes a shape and a number (the x-offset) and returns a shape with the same dimensions as the original, but moved horizontally by the x-offset.

```
;; A point is a structure (make-point x y) where
;; x and y are numbers
(define-struct point (x y))
```

```
(define shape%
  (class* object% () ()
    (public
      ;; translate-point-x : point number → point
      ;; produces a new point that translates the old point by x-offset
      ;; in the horizontal axis
      (translate-point-x
       (lambda (a-point x-offset)
         (make-point (+ (point-x a-point) x-offset)
                     (point-y a-point)))))
    (sequence (super-init))))

;; area : → number
;; returns the area of the given shape

;; translate : number → shape
;; returns a new shape with the same dimensions as the original but moved
;; horizontally by x-offset
(define shapeI (interface () area translate))

;; get-center : → point
;; returns the point at the circle's center
(define circleI (interface (shapeI) get-center))

(define circle%
  (class* shape% (circleI) (center radius)
    ;; center is a point
    ;; radius is a natural number
    (inherit translate-point-x)
    (public
      (get-center (lambda () center))
      (area (lambda () (* 2 pi radius)))
      (translate (lambda (x-offset)
                   (make-object circle%
                     (translate-point-x center x-offset)
                     radius))))
    (sequence (super-init))))

;; get-tl-corner : → point
;; returns the point at the rectangle's top-left corner
(define rectI (interface (shapeI) get-tl-corner))

(define rectangle%
  (class* shape% (rectI) (tl-corner width height)
    ;; tl-corner is a point
    ;; width and height are natural numbers
    (inherit translate-point-x)
    (public
      (get-tl-corner (lambda () tl-corner))
      (area (lambda () (* width height)))
      (translate (lambda (x-offset)
                   (make-object rectangle%
```

```
                 (translate-point-x tl-corner x-offset)
                 width height))))
  (sequence (super-init))))

(define square%
  (class* rectangle% () (tl-corner width)
    ;; tl-corner is a point
    ;; width is a natural number
    (inherit translate-point-x)
    (override
      (translate (lambda (x-offset)
                   (make-object square%
                     (translate-point-x tl-corner x-offset)
                     width))))
    (sequence (super-init tl-corner width width))))

;;TESTS

(define S (make-object square% (make-point 4 5) 10))
(define S1 (send S translate 6))
(= (point-x (send S1 get-tl-corner)) 10)
(= (send S1 area) 100)

(define C (make-object circle% (make-point 100 100) 25))
(define C1 (send C translate −25))
(= (point-x (send C1 get-center)) 75)
```

6. A gradebook contains certain information for each student: name, id number, and a list of numeric scores on homework assignments.

   Write the following programs over gradebooks:

   (a) *total-points*, which consumes a student name and a gradebook and returns the total homework points earned by the student so far.

   (b) *add-grade*, which consumes a student name, a homework grade, and a gradebook and adds the grade to the gradebook for the named student.

```
(define student%
  ;; name is a symbol and id is a number
  (class* object% () (name id)
    (private
      (grades empty))
    (public
      ;; get-name : → symbol
      (get-name (lambda () name))
      ;; get-id : → number
      (get-id (lambda () id))
      ;; add-grade : number → void
      (add-grade (lambda (grade)
                   (set! grades (cons grade grades))))
      ;; total-grades : → number
      (total-grades (lambda () (sum grades))))
    (sequence (super-init))))
```

```
;; A gradebook is a list of student

;; These two definitons are to demonstrate exception handling.
;; In MzScheme, an exception can raise any value, so you can design your
;; exceptions to return the information that you need to handle the
;; exception. In this case, we are defining an exception for the case when
;; a student is missing in the database. The value to be raised on the
;; exception is simply a record containing the student's name.

(define-struct MissingStudentExn (name))

;; The next definition defines an exception handler. The handler consumes a
;; value and executes some code to process the exception.

(define GradebookHandler
  (lambda (MSExn)
    (printf "Requested student ~a is not in the gradebook ~n"
            (MissingStudentExn-name MSExn))))

;; sum : list-of-nums → number
;; sums all of the numbers in a list
(define sum
  (lambda (alon)
    (cond [(empty? alon) 0]
          [else (+ (first alon) (sum (rest alon)))])))

;; find-student : symbol gradebook → student;; given the name of a student, return the object corresponding to that
;; student in the gradebook. Raise a MissinStudentExn if no such student
;; exists in the gradebook
;; This demonstrates how to raise exceptions. The operator raise consumes
;; any value and sends that value as an exception to the innermost installed
;; exception handler.

(define find-student
  (lambda (student-name in-gradebook)
    (cond [(empty? in-gradebook) (raise (make-MissingStudentExn student-name))]
          [else (cond [(eq? student-name (send (first in-gradebook) get-name))
                       (first in-gradebook)]
                      [else (find-student student-name (rest in-gradebook))])])))

;; find-and-process-student : symbol gradebook (student;; locates named student in given gradebook then processes student with given
;; function
(define find-and-process-student
  (lambda (student-name in-gradebook proc)
    (with-handlers [(MissingStudentExn? GradebookHandler)]
      (let [(the-student (find-student student-name in-gradebook))]
        (proc the-student)))))

;; total-points : symbol gradebook → number
;; produces total points earned by named student in gradebook
(define total-points
  (lambda (student-name in-gradebook)
    (find-and-process-student
     student-name in-gradebook
     (lambda (s) (send s total-grades)))))
```

```
;; add-grade : symbol number gradebook → void
;; records given grade in gradebook for named student
(define add-grade
  (lambda (student-name new-grade in-gradebook)
    (find-and-process-student
     student-name in-gradebook
     (lambda (s) (send s add-grade new-grade)))))
```

Without editing any of your previous code for this problem, add a class year (freshman, etc) for each student and write a program *grade-warning* which consumes a gradebook and a class year and produces a list of names of all students in that year who have fewer than 50 homework points.

```
(define student-year%
  (class* student% () (name id year)
    (public
      ;; in-year : symbol → boolean
      (in-year? (lambda (a-year) (eq? year a-year))))
    (sequence (super-init name id))))

;; grade-warning : gradebook num → list of symbol
;; produces a list of all students in given year whose total points are below 50
(define grade-warning
  (lambda (gradebook year)
    (cond [(empty? gradebook) empty]
          [(cons? gradebook)
           (cond [(and (send (first gradebook) in-year? year)
                       (< (send (first gradebook) total-grades) 50))
                  (cons (send (first gradebook) get-name)
                        (grade-warning (rest gradebook) year))]
                 [else (grade-warning (rest gradebook) year)])])))

;; for those of you interested in learning advanced features, here's
;; another way to write grade-warning. map and filter are built in.
(define grade-warning-advanced
  (lambda (gradebook year)
    (map (lambda (student) (send student get-name))
         (filter (lambda (student)
                   (and (send student in-year? year)
                        (< (send student total-grades) 50)))
                 gradebook))))

;; TESTS

(define Curly (make-object student% 'Curly 1000))
(define Sue (make-object student% 'Sue 1001))
(define Nod (make-object student% 'Nod 1002))
(define GB (list Curly Sue Nod))
(= (total-points 'Sue GB) 0)
(add-grade 'Sue 95 GB)
(= (total-points 'Sue GB) 95)
(add-grade 'Sue 80 GB)
(= (total-points 'Sue GB) 175)
(add-grade 'Mike 90 GB) ;; should raise exception
```

```
(total-points 'Larry GB) ;; should raise exception

(define Mickey (make-object student-year% 'Mickey 1003 'freshman))
(define Daisy (make-object student-year% 'Daisy 1004 'sophomore))
(define Snoopy (make-object student-year% 'Snoopy 1005 'freshman))

(define GB2 (list Mickey Daisy Snoopy))
(equal? (grade-warning GB2 'freshman) (list 'Mickey 'Snoopy))
(add-grade 'Mickey 65 GB2)
(equal? (grade-warning GB2 'freshman) (list 'Snoopy))
(= (total-points 'Mickey GB2) 65)
```

7. A binary tree is a tree in which each node contains a number and accessors to (up to) two other nodes (left and right). A binary search tree is a binary tree with the following property: at each node of the tree, the number at that node must be larger than or equal to the numbers in all nodes accessible down the left branch and smaller than or equal to all nodes accessible down the right branch.

   Write the following programs over binary (search) trees:

   (a) *bst?*, which consumes a binary tree and returns a boolean indicating whether the tree is a binary search tree.

   (b) *bst-order*, which consumes a binary search tree and returns an ordered list of all numbers stored in the tree.

   (c) *bst-add*, which consumes a binary search tree and a number and returns a binary search tree that contains the new number and all of the numbers that were in the original tree.

```
;; A btnode is a either
;; - 'none or
;; - (make-btnode num left right) where
;; num is a number and left and right are btnode

(define-struct btnode (num left right))


;; bst-order : btnode → list-of-nums
;; returns list of numbers in order in binary search tree
(define bst-order
  (lambda (a-node)
    (cond [(eq? a-node 'none) empty]
          [else (append (bst-order (btnode-left a-node))
                        (list (btnode-num a-node))
                        (bst-order (btnode-right a-node)))])))


;; bst-add : btnode num → bsnode
;; returns new bst containing num and all nums from original bst
(define bst-add
  (lambda (a-node newnum)
    (cond [(eq? a-node 'none)
           (make-btnode newnum 'none 'none)]
          [else (cond [(> newnum (btnode-num a-node))
                       (make-btnode (btnode-num a-node)
                                    (btnode-left a-node)
                                    (bst-add (btnode-right a-node) newnum))]
                      [(<= newnum (btnode-num a-node))
```

```
                    (make-btnode (btnode-num a-node)
                                 (bst-add (btnode-left a-node) newnum)
                                 (btnode-right a-node))])]))))

;; let and letrec (in bst?) define local variables. let is for non-recursive
;; definitions and letrec is for recursively-defined variables

;; bst? : btnode → boolean
;; returns true iff binary tree rooted at node is a binary search tree
(define bst?
  (lambda (a-node)
    (let ([node-order (bst-order a-node)])
      (letrec ([ascending?
                (lambda (alon)
                  (cond [(empty? alon) true]
                        [(empty? (rest alon)) true]
                        [else (and (<= (first alon) (first (rest alon)))
                                   (ascending? (rest alon)))]))])
        (ascending? node-order)))))

;; tests

(define BST (make-btnode 5
                         (make-btnode 2
                                      (make-btnode 1 'none 'none)
                                      'none)
                         (make-btnode 8 'none 'none)))

(define BT (make-btnode 5
                        (make-btnode 2
                                     (make-btnode 9 'none 'none)
                                     'none)
                        (make-btnode 8 'none 'none)))

(bst? BST)
(not (bst? BT))
(bst-order BST)
(bst-order (bst-add BST 4))
```

8. A file system consists of files organized into potentially nested subdirectories. For this problem, we are interested in storing the name, size, and contents of each file, and the name and contents (files and subdirectories) of each directory.

   Write the following programs over filesystems:

   (a) *total-size*, which consumes a filesystem and returns the total size of all files in the filesystem.

   (b) *contains-file?*, which consumes a filesystem and a file name and returns a boolean indicating whether the filesystem contains the given file.

   Without data descriptions, this is hard to get right. I show the functional solution here to demonstrate how cleanly this follows from the data descriptions. To turn this into a more OO solution, you need a class for files and a class for directories. I would leave the lists of files and dirs as Scheme lists, rather than objects implementing lists because the objects are too heavyweight for this situation. The functions processing lists of files and directories would remain the same, minus using send instead of function calls for file-func and dir-func (in the templates). The file-func and dir-func functions would becomes methods in the file and dir classes, respectively.

```
;; A file is a structure (make-file name size contents)
;; where name is a symbol, size is a number and contents is any value

(define-struct file (name size contents))

;; A list-of-files is either
;; - empty, or
;; - (cons file L) where L is a list-of-files

;; A directory is a structure (make-dir name dirs files)
;; where name is a symbol, dirs is a list-of-dirs and
;; files is a list-of-files

(define-struct dir (name dirs files))

;; A list-of-dirs is either
;; - empty, or
;; - (cons dir L) where dir is a directory and L is a list-of-dirs
```

Here are the templates/skeletons that arise from these definitions. There is one skeleton function per data description, with calls between skeletons whenever there is a reference between data descriptions

```
(define (file-func a-file)
  ... (file-name a-file)
  ... (file-size a-file)
  ... (file-contents a-file) ...)

(define (lofiles-func a-lof)
  (cond [(empty? a-lof) ...]
        [(cons? a-lof)
         ... (file-func (first a-lof))
         ... (lofiles-func (rest a-lof))]))

(define (dir-func a-dir)
  ... (dir-name a-dir)
  ... (lodirs-func (dir-dirs a-dir))
  ... (lofiles-func (dir-files a-dir)))

(define (lodirs-func a-lod)
  (cond [(empty? a-lod) ...]
        [(cons? a-lod)
         ... (dir-func (first a-lod))
         ... (lodirs-func (rest a-lod))]))


;; lofiles-size : list-of-files → number
;; returns total size of all files in list of files
(define (lofiles-size a-lof)
  (cond [(empty? a-lof) 0]
        [(cons? a-lof)
         (+ (file-size (first a-lof))
            (lofiles-size (rest a-lof)))]))
```

```
;; lodirs-size : list-of-dirs → number
;; returns total size of all files in list of directories
(define (lodirs-size a-lod)
  (cond [(empty? a-lod) 0]
        [(cons? a-lod)
         (+ (total-size (first a-lod))
            (lodirs-size (rest a-lod)))]))

;; total-size : directory → number
;; returns total size of all files (nested) in a directory
(define (total-size a-dir)
  (+ (lodirs-size (dir-dirs a-dir))
     (lofiles-size (dir-files a-dir))))

;; file-eq? : file symbol → boolean
;; returns true if filename same as name of file
(define (file-eq? a-file filename)
  (eq? (file-name a-file) filename))

;; lofiles-contains-file? : list-of-files symbol → boolean
;; returns true if list of files contains file with given name
(define (lofiles-contains-file? a-lof filename)
  (cond [(empty? a-lof) false]
        [(cons? a-lof)
         (or (file-eq? (first a-lof) filename)
             (lofiles-contains-file? (rest a-lof) filename))]))

;; contains-file? : directory symbol → boolean
;; returns true if directory contains file with given name
(define (contains-file? a-dir filename)
  (or (lodirs-contains-file? (dir-dirs a-dir) filename)
      (lofiles-contains-file? (dir-files a-dir) filename)))

;; lodirs-containts-file? : list-of-dirs symbol → boolean
;; return true if list of directories contains given filename
(define (lodirs-contains-file? a-lod filename)
  (cond [(empty? a-lod) false]
        [(cons? a-lod)
         (or (contains-file? (first a-lod) filename)
             (lodirs-contains-file? (rest a-lod) filename))]))

;; TESTS

(define MyDir
  (make-dir 'Home (list (make-dir 'Classes
                                  (list (make-dir
                                         'CS3733
                                         empty
                                         (list (make-file 'hwk 25 'assignments))))
                                  empty)
                        (make-dir 'Papers
                                  empty
```

```
                              (list (make-file 'paper1 350 'technique1)
                                    (make-file 'paper2 750 'newtechnique))))
                (list (make-file 'mail 1000 (list 'a 'b 'c)))))

(contains-file? MyDir 'paper2)
(not (contains-file? MyDir 'addresses))
(= (total-size MyDir) 2125)
```