CSU2510H Exam 2 Rubric – Spring 2011

Problem 1 You've just been hired by Adobe to write their next generatation PDF rendering engine. As your first task, you've been assigned to write an *outline viewer* that displays the overall structure of a document. A document can consist of any number of sections, and a section has a title and any number of subsections, and a subsection has a title and any number of subsubsection, and so on.

18 POINTS

1. Design a representation of documents that can handle documents like this:

```
1. OO = struct + fun
1.1. Rocket
1.2. Moon
2. Design
2.1. Announce
2.1.1. Assign
3. End
```

   Note that the section numbers are *NOT* part of the document!

```
;; A Sec is one of:                [5pt]
;; - (mt-sec%)
;; - (cons-sec% String Sec Sec)
;; A Doc is a Sec.
(define-class mt-sec%)
(define-class cons-sec%
  (fields title sub next))
```

2. Using your design, give the representation of the document in part 1.

```
(cons-sec% "OO = struct + fun"       [5pt]
           (cons-sec% "Rocket"
                      (mt-sec%)
                      (cons-sec% "Moon"
                                 (mt-sec%)
                                 (mt-sec%)))
           (cons-sec% "Design"
                      (cons-sec% "Announce"
                                 (cons-sec% "Assign"
                                            (mt-sec%)
                                            (mt-sec%))
                                 (mt-sec%))
                      (cons-sec% "End"
                                 (mt-sec%)
                                 (mt-sec%))))
```

3. Design a method `outline` that produces a list of strings, which are the sectional headings in order *and with section numbers*.

   In other words, if `d` represents the document in part 1, `outline` should produce the following:

```
> (d . outline)
'("1. OO = struct + fun"
  "1.1. Rocket"
  "1.2. Moon"
  "2. Design"
  "2.1. Announce"
  "2.1.1. Assign"
  "3. End")
```

```
;; Number -> String                [8pt]
;; Purpose: 5 => "5."
(define (sec-str n)
  (string-append (number->string n) "."))

;; Sec implements:
;; Produce outline view of document.
;; outline : -> [Listof String]

;; Produce headings of document with section numbers.
;; Assume: n is the current sectional number.
;; headings : Number -> [Listof String]

;; mt-sec%
(define/public (headings n) empty)
(define/public (outline) empty))

;; con-sec%
(define/public (outline)
  (headings 1))

(define/public (headings n)
  (cons (string-append (sec-str n) " " (field title))
        (append (map (lambda (s) (string-append (sec-str n) s))
```

```
                    ((field sub) . headings 1))
                ((field next) . headings (add1 n))))))

(check-expect ((mt%) . outline) empty)
(check-expect ((cons-sec% "A" (mt%) (mt%)) . outline)
              '("1. A"))

(check-expect ((cons-sec% "A"
                          (mt-sec%)
                          (cons-sec% "B"
                                     (mt-sec%)
                                     (mt-sec%)))
               . outline)
              '("1. A" "2. B"))

(check-expect ((cons-sec% "A"
                          (cons-sec% "B"
                                     (mt-sec%)
                                     (mt-sec%))
                          (mt-sec%))
               . outline)
              '("1. A" "1.1. B"))
```

Problem 2 The Programming Research Lab has been acquiring some new decorative objects, in the form of rectangles and rectangular solids. Unfortunately, they've all come unpainted, and now the graduate students have to paint them. Dan has been trying to calculate how much this will cost, and he's enlisted you to help. 19 POINTS

First, some basic facts. Rectangles have an area which is the width times height, and must be painted on both sides. Rectangular solids have a surface area which is the sum of the areas of all of the sides, or $2 \cdot (w \cdot d + d \cdot h + w \cdot h)$. Red paint costs 2 dollars per square foot, and blue paint costs 5 dollars per square foot. Each decorative object has a color, and the appropriate dimensions, measured as an integer number of feet.

Design a representation for these decorative objects, and implement the `expense` method. You should use *overriding* to avoid repeating code—in particular, you should only have one version of the `expense` method.

```
;; A Rect is (rect% Number Number)           [+1]
;; an implements
;; expense : -> Number                       [+2]
;; how much will it cost to paint this?
;; area : -> Number                          [+2]
;; how much surface area does this have
(define-class rect%
  (fields w h color)
  (define/public (expense)                   [+3]
    (cond [(string=? (field color) "blue") (* 5 (area))]
  [else (* 2 (area))]))
  (define/public (area) (* (w) (h) 2)))      [+1]

(check-expect ((rect% 10 10 "blue") . area) 200)   [+2]
(check-expect ((rect% 10 10 "red") . expense) 400)

;; A 3DRect is (3drect% Number Number Number String) [+1]
;; and extends Rect                                  [+1]
(define-class 3drect%
  (super rect%)                                     [+1]
  (fields d)                                        [+1]
```

```
  ;; overriding
  (define/public (area)                                  [+2]
    (* 2 (+ (* (w) (h))
    (* (h) (d))
    (* (w) (d))))))

(check-expect ((3drect% 3 10 10 "red") . area) 320)   [+2]
(check-expect ((3drect% 3 10 10 "blue") . expense) 1600)
```

Problem 3 Tobin-Hochstadt and Van Horn have both been traveling a lot recently, so they've decided to develop software to manage their trips. Since they're busy with writing the exam, they've asked you to help them out. 22 POINTS

Additionally, since they want their program to be as verbose as possible, they've asked you to write it in Java. In each of the following implementations, you must support the `howLong` method.

1. Initially, they've settled on the following kinds of data to represent trips. A `Trip` is either a `PlaneFlight`, with an airline and a distance (in miles), or a `Connection`, composed of two trips.

   Write a data definition for trips, and implement the `howLong` method, which computes the total time of the trip (in minutes), assuming that airplanes fly at 10 miles per minute.

```
interface Trip {                          //[+1]
    // how many minutes will this trip take?
    public Integer howLong();             //[+1]
}

class PlaneFlight implements Trip {         [+1]
    Integer miles;                          [+2]
    String airline;
    PlaneFlight(Integer m, String s)        [+1]
         { miles = m; airline = s; }
    public Integer howLong()                [+1]
      { return miles/10; }
}

class Connection implements Trip {    [+1]
    Trip first;                       [+2]
    Trip second;
    Connection(Trip t1, Trip t2) {      [+1]
        first = t1;
        second = t2;
    }
    public Integer howLong() {                      [+2]
```

```
        return first.howLong() + second.howLong();
    }
}
```

2. Van Horn has discovered that getting around New England is easier if you take the train, even though it takes 2 minutes to go 1 mile. Add a new representation for this new kind of trip, which should have a distance and an indication of whether the trip is on the Acela, in which case the train goes twice as fast (1 minute per mile).

```
class TrainTrip implements Trip {  [+1]
    Integer miles;                 [+1]
    Boolean acela;
    TrainTrip(Integer m, Boolean a) { miles = m; acela = a; } [+1]
    public Integer howLong() {
if (acela) {
    return miles;           [+1]
}
else {
    return miles * 2;       [+1]
}
    }
}

// Another possibility:

class RegTrainTrip implements Trip {
  Integer miles;
  RegTrainTrip(Integer miles) { this.miles = miles; }
  public Integer howLong() {
      return this.miles * 2;
  }
}

class AcelaTrainTrip implements Trip {
  Integer miles;
  AcelaTrainTrip(Integer miles) { this.miles = miles; }
  public Integer howLong() {
      return this.miles;
  }
}
```

3. Tobin-Hochstadt also likes to take the train, but unfortunately, he's been taking the commuter rail instead, which goes as fast as regular trains but is always half an hour late. Add to your data definition of trains a representation for commuter rail trips.

```
class CommuterRail extends TrainTrip {   [+1]
    CommuterRail(Integer m) { super(m,false); } [+1]
    public Integer howLong() {                [+2]
return miles * 2 + 30; // or super() + 30
    }
}

// or

class CommuterRail implements Trip {  [+1]
    Integer miles;
    CommuterRail(Integer m) { miles = m; }   [+1]
    public Integer howLong() {
return miles * 2 + 30;            [+2]
    }
}

// or

class CommuterRail extends RegTrainTrip {
   public Integer howLong() {
       return super() + 30;
   }
}
```

Problem 4 Here are data, class, and interface definitions for Complex Numbers: 20 POINTS

```
;; A Complex is (cplx% Number Number)
;; dist : -> Number
;; How far is this point from the origin.
;; =? : Complex -> Boolean
;; Determine if this Complex is the same as the given one

(define-class cplx%
  (fields real imag)
  (define/public (dist)
    (sqrt (+ (sqr (real)) (sqr (imag)))))
  (define/public (=? that)
    (and (= (real) (that . real))
         (= (imag) (that . img)))))
```

1. Design a `UPair` data definition for an unordered pair of `Complex`, with an `=?` method. Because these are unordered pairs,

   ```
   ((upair% (cplx% 1 1) (cplx% 2 2)) . =?
    (upair% (cplx% 2 2) (cplx% 1 1)))
   ```

   should produce `#t`.

```
;; Part 1
;; A UPair is (upair% Posn Posn) [+1]
(define-class upair%
  (fields left right)            [+1]
  ;; =? : UPair -> Boolean       [+1]
  ;; compare this with p         [+1]
  (define/public (=? p)          [+3]
    (or (and ((left)  . =? (p . left))
     ((right) . =? (p . right)))
(and ((left)  . =? (p . right))
     ((right) . =? (p . left))))))
```

```
(define p1 (cplx% 1 2))
(define p2 (cplx% 2 1))
(define u1 (upair% p1 p2))
(define u2 (upair% p2 p1))
(define u3 (upair% p2 p2))
(check-expect (u1 . =? u2) true)              [+2]
(check-expect (u1 . =? u3) false)
```

2. Revise your data, class, and interface definitions so that your unordered pairs can store other kinds of data, not just `Complexes`. Make your solution as general as possible, but no more.

```
;; A [UPair X] is (upair% X X)                [+2]
;; where X implements                         [+2]
;; =? : X -> Boolean

;; for not changing the code                  [+1]

(define-class upair%
  (fields left right)
  ;; =? : UPair -> Boolean
  ;; compare this with p
  (define/public (=? p)
    (or (and ((left)  . =? (p . left))
     ((right) . =? (p . right)))
(and ((left)  . =? (p . right))
     ((right) . =? (p . left))))))

(define p1 (cplx% 1 2))
(define p2 (cplx% 2 1))
(define u1 (upair% p1 p2))
(define u2 (upair% p2 p1))
(define u3 (upair% p2 p2))
(check-expect (u1 . =? u2) true)
(check-expect (u1 . =? u3) false)
```

3. Since `UPair` is unordered, switching the two components of a pair should produce a pair that's equal to the original one. Define a predicate that expresses this property, and use it to state two test cases.

```
;; Part 3

;; swap-prop : [UPair X] -> Boolean                [+1]
;; check if this pair is equal to itself when swapped
(define (swap-prop up)                             [+3]
  (up . =? (upair% (up . right) (up . left))))

(check-expect (swap-prop (upair% (cplx% 1 2) (cplx% 2 1))) true) [+2]
(check-expect (swap-prop (upair% (cplx% 1 1) (cplx% 1 1))) true)
```

Problem 5 Here's an implementation of lists that supports visitors: 11 POINTS

```
;; A [Listof X] is one of
;; - (mt%)
;; - (cons% X [Listof X])
;;
;; and implements:
;;
;; Accept the visitor and visit this list's data.
;; visit : [IListVisitor X Y] -> Y

(define-class mt%
  (define/public (visit v)
    (v . mt)))

(define-class cons%
  (fields first rest)
  (define/public (visit v)
    (v . cons (field first)
              ((field rest) . visit v))))
```

The `[IListVisitor X Y]` interface represents a computation over a `[Listof X]` that produces a `Y`:

```
;; An [IListVisitor X Y] implements:

;; Visit an empty list
;; mt : -> Y

;; Visit a cons list
;; cons : X Y -> Y
```

1. Design a class that implements `[IListVisitor Number Number]` that sums a list of numbers.

```
;; [5pt]
;; A Sum is a (sum%), implements [IListVisitor Number Number]
(define-class sum%
  ;; Sum an empty list
  ;; -> Number
  (define/public (mt) 0)
  ;; Sum a cons list
  ;; Number Number -> Number
  (define/public (cons first rest)
    (+ first rest)))

(check-expect ((mt%) . visit (sum%)) 0)
(check-expect ((cons% 5 (cons% 7 (mt%))) . visit (sum%)) 12)
```

2. Recall our previous discussion of functions-as-objects, which relied on an interface for objects representing functions:

```
;; An [IFun X Y] implements:

;; Apply this function to given argument.
;; apply : X -> Y
```

Design a class `map%` that implements `[IListVisitor [Listof X] [Listof Y]]`.

For example, here is a use of the `map%` visitor that maps the representation of an add1 function over a list of numbers.

```
;; An Add1 is a (add1%), implements [IFun Number Number].
(define-class add1%
  (define/public (apply x) (add1 x)))

> ((cons% 1 (cons% 2 (cons% 3 (mt%)))) . visit (map% (add1%)))
(cons% 2 (cons% 3 (cons% 4 (mt%))))


;; [6pt]
;; A Map is a (map% [IFun X Y])                    ;[1pt]
;; implements [IListVisitor X [Listof Y]]
(define-class map%
  (fields f)                                       ;[1pt]
  ;; Map an empty list.
  ;; -> [Listof Y]
  (define/public (visit-mt) (mt%))                 ;[1pt]
  ;; Map a cons list.
  ;; X [Listof Y] -> [Listof Y]
  (define/public (visit-cons first rest)           ;[1pt]
    (cons% (f . apply first) rest)))               ;[2pt]
```