CS116 Winter 2011 Assignment 2
Due: Tuesday, January 18 at 10:00AM

The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.

Important Information:

1. Read the course Style Guide for information on assignment policies and how to organize and submit your work. In particular, your solutions should be placed in files `a2qY.rkt`, where `Y` is a value from 1 to 5.
2. Be sure to download the interface file to get started.
3. Do not copy the purpose directly from the assignment description – it should be in your own words.
4. You should use abstract list functions (`map, filter, foldr, build-list`) where appropriate. Solutions that use explicit recursion **will not receive any correctness marks,** unless noted explicitly in the question that recursion is allowed.
5. All helper functions and constants should be contained within the main function definition using `local` or `lambda`. When a function is used only once as a helper to an abstract list function, you must use `lambda`, or marks will be deducted.
6. Do not use `reverse`.
7. You may assume that all consumed data will satisfy any stated assumptions, unless indicated otherwise.

Language level: Intermediate Student with `lambda`.
Coverage: Module 2

The following structure (based on a real card game called Lost Cities) is needed for this assignment.

```
(define-struct Lost-Cities-Card (value colour))
;; A Lost-Cities-Card is a structure
;; (make-Lost-Cities-Card v c), where
;; v is the symbol 'multiply or an integer in the
;;   range [2..10]
;; c is a symbol for the colour of the card (one of
;;   'red, 'blue, 'white, 'yellow, 'green).
```

1. Write a Scheme function `count-multiply-cards`, which consumes a list of `Lost-Cities-Card` values, and produces the number of cards with `value 'multiply`, regardless of their `colour`. For example,

```
(count-multiply-cards
  (list (make-Lost-Cities-Card 'multiply 'red)
        (make-Lost-Cities-Card 8 'blue)
        (make-Lost-Cities-Card 'multiply 'blue))) => 2
```

2. In Lost Cities, points are scored based on colour sequences played. All points of a colour are totaled (with `'multiply` cards having value 0, all other cards are their `value` field), and 20 points is deducted to get a subtotal. If there are `m 'multiply` cards in the sequence, then the subtotal is multiplied by (`m`+1), to get a revised total. If there are more than 8 cards (including `'multiply` cards) in the sequence, then a bonus of 20 points is added to the revised total. An empty sequence is worth 0 points. For example,
    - the score of the sequence containing `'multiply` is -40
    - the score of the sequence containing `'multiply`, `'multiply`, 7,10 is -9
    - the score of the sequence containing 2,3,4,5,6,7,8,9,10 is 54.

    Write the function `score-Lost-Cities`, which consumes a list of `Lost-Cities-Card` structures (all the same colour), and produces the score for that list, according to the rules above.

3. A full deck of cards[1] for the game Lost Cities consists of 11 cards for each of the 5 colours: two multiply cards, and 9 cards labeled 2 through 10. Write a function `build-deck-to` which consumes a natural number `top-value` between 0 and 10, and produces all the cards in the Lost Cities deck whose `value` is less than or equal to `top-value`. The following convention should be used:
    - If `top-value=0`, then create a single card of each colour with `value 'multiply`.
    - If `top-value=1`, then create two cards of each colour with `value 'multiply`, grouping each colour together, in the order given in the data definition.
    - If `top-value>1`, then create (`top-value+1`) cards of each colour, with `values 'multiply, 'multiply, 2, …, top-value` (in that order), grouping each colour together.

    You must use `build-list` and other abstract list functions to create the list in your solution.

    For example, `(build-deck-to 3)` produces

```
(list
 (make-Lost-Cities-Card 'multiply 'red)
 (make-Lost-Cities-Card 'multiply 'red)
 (make-Lost-Cities-Card 2 'red)
 (make-Lost-Cities-Card 3 'red)
 (make-Lost-Cities-Card 'multiply 'blue)
 (make-Lost-Cities-Card 'multiply 'blue)
 (make-Lost-Cities-Card 2 'blue)
 (make-Lost-Cities-Card 3 'blue)
```

[1] A real Lost Cities deck has three multiply cards per colour, rather than two. But for our purposes, two is sufficient.

```
    (make-Lost-Cities-Card 'multiply 'white)
    (make-Lost-Cities-Card 'multiply 'white)
    (make-Lost-Cities-Card 2 'white)
    (make-Lost-Cities-Card 3 'white)
    (make-Lost-Cities-Card 'multiply 'yellow)
    (make-Lost-Cities-Card 'multiply 'yellow)
    (make-Lost-Cities-Card 2 'yellow)
    (make-Lost-Cities-Card 3 'yellow)
    (make-Lost-Cities-Card 'multiply 'green)
    (make-Lost-Cities-Card 'multiply 'green)
    (make-Lost-Cities-Card 2 'green)
    (make-Lost-Cities-Card 3 'green)).
```

4. In Lost Cities, players try to build increasing sequences of cards of each colour. A card, `c`, is playable on another card, `top-card`, if the two cards are the same colour, and one of the following conditions are satisfied by their values:
   - the value of `top-card` is `'multiply`, or
   - the value of `top-card` < the value of `c`.

   Note that this means that a card with value `'multiply` can only be played on the other `'multiply` card of the same colour.

   Write a function `playable-cards` that consumes a single `Lost-Cities-Card`, and produces a function. The new function consumes a list of `Lost-Cities-Card`s, and produces only those that are playable on the card consumed by `playable-cards`. For example,

```
((playable-cards (make-Lost-Cities-Card 'multiply 'blue))
 (list (make-Lost-Cities-Card 'multiply 'red)
       (make-Lost-Cities-Card 8 'blue)
       (make-Lost-Cities-Card 'multiply 'blue)
       (make-Lost-Cities-Card 3 'blue)
       (make-Lost-Cities-Card 'multiply 'green)
       (make-Lost-Cities-Card 7 'blue)
       (make-Lost-Cities-Card 6 'red)))
 ⇨  (list (make-Lost-Cities-Card 8 'blue)
          (make-Lost-Cities-Card 'multiply 'blue)
          (make-Lost-Cities-Card 3 'blue)
          (make-Lost-Cities-Card 7 'blue))
```

and

```
((playable-cards (make-Lost-Cities-Card 4 'blue))
 (list (make-Lost-Cities-Card 'multiply 'red)
       (make-Lost-Cities-Card 8 'blue)
       (make-Lost-Cities-Card 'multiply 'blue)
       (make-Lost-Cities-Card 3 'blue)
       (make-Lost-Cities-Card 'multiply 'green)
       (make-Lost-Cities-Card 7 'blue)
```

```
       (make-Lost-Cities-Card 6 'red)))
 ⇨  (list (make-Lost-Cities-Card 8 'blue)
          (make-Lost-Cities-Card 7 'blue)).
```

5. Complete the function `select-min` which consumes a function `f` and three lists `A`,`B`,`C`, all of the same length and containing the same type of values. The function `f` consumes a single value of the same type as the list elements, and produces a number. Consider the values `a`,`b`,`c`, where `a` is from `A`, `b` is from `B`, `c` is from `C`, all located at the same location in their respective lists. The list produced by `select-min` will include
   - `a` if `(f a)` is less or equal to both `(f b)` and `(f c)`,
   - `b` if `(f b)` is less than `(f a)` and less or equal to `(f c)`,
   - `c` otherwise,

   for each such triple.

   For example,

```
(select-min string-length
   (list "hi"    "there"   "friend"  "a-z")
   (list "two"   "times"   "three"   "123")
   (list "about" "nothing" ""        "!!!"))
=> (list "hi"    "there"   ""        "a-z")
```

   For this question, you may use structural recursion (though it can be completed using abstract list functions).