CS 2500, Spring 2013
Problem Set 10

Due date: Tuesday, April 2 @ 11:59pm

Programming Language: Intermediate Student Language with Lambda

You must work on and submit this homework with your assigned partner. You will receive no credit for ignoring your partner and working on the homework alone or not working on the homework at all.

Using Blackboard, submit a single Racket file containing all of the code and documentation for this assignment. Place your names and husky email addresses in a comment at the beginning of your file.

Name your file: hw10-lastname1-lastname2.rkt

You must follow the design recipe in your solutions: graders will look for data definitions, contracts, purpose statements, examples/tests, and properly organized function definitions. For the latter, you must follow templates. You do not need to include the templates with your homework, however, unless the question asks for it.

Problem 1.

Review section 14.4 in the book, including the problems. Now, consider these data definitions:

```
  ; a Value is one of: Number String
  ;
  ; a Prim is one of:
  ; '+ '- '* '/ 'sqrt
  ; 'string-length 'string-append 'number->string
  ;
  ; a PExp (primitive expression) is one of: Value PApp
  ;
  ; a PApp (primitive application) is a (cons Prim [Listof PExp])
```

In all of the following, use loop functions such as `map` and `member?` where possible, and consider using `apply` where appropriate as well. You may find it easier to first implement the functions without these "power tools" and then look for ways to simplify the code by using them.

a) Fill in the `...` in the following function, and make sure all the tests pass:

```
  ; prim?: Any -> Boolean
  ; is the argument a primitive?
  (define (prim? v) ...)

  (check-expect (prim? '+) true)
  (check-expect (prim? 'number->string) true)
  (check-expect (prim? 'foo) false)
```

b) Fill in the `...` in the following function, and make sure all the tests pass:

```
; apply-prim: Prim [Listof Value] -> Value
; apply a primitive to a list of parameter values
; the primitives '+ '- '* '/ and 'string-append accept two or more
; inputs each
; the remaining primitives accept one input only
(define (apply-prim fn vals) ...)

(check-expect (apply-prim '+ '(1 2 3)) 6)
(check-expect (apply-prim '- '(1 2 3)) -4)
(check-expect (apply-prim '* '(1 2 3)) 6)
(check-expect (apply-prim '/ '(1 2)) (/ 1 2))
(check-expect (apply-prim 'sqrt '(4)) 2)
(check-expect (apply-prim 'string-length '("foo")) 3)
(check-expect (apply-prim 'string-append '("a" "b")) "ab")
(check-expect (apply-prim 'number->string '(23)) "23")
```

c) fill in the `...` in the following function, and make sure all the tests pass:

```
; prim-eval: PExp -> Value
; evaluate a primitive expression
(define (prim-eval s) ...)

(check-expect (prim-eval 2) 2)
(check-expect (prim-eval "a") "a")
(check-expect (prim-eval '(+ 3 (* 2 3))) 9)
(check-expect (prim-eval '(string-append "a" (number->string 2)))
              "a2")
(check-expect (prim-eval '(* (string-length "bob") 2)) 6)
```

Congratulations, you have now made a Scheme expression interpreter that can handle numeric and string operations, including nested sub-expressions! However, it does not know about user-defined functions...

Problem 2.

In an arithmetic sequence, a_0, a_1, a_2,..., a_n, a_{n+1},..., each successor term a_{n+1} is the result of adding a fixed constant to a_n. For example, the sequence *8, 13, 18, 23, ...* has a starting point of 3 and the constant is 5. From these two facts, called*starting point* and *summand*, respectively, all other terms in the sequence can be determined.

a) Develop the recursive function `a-fives`, which consumes a natural number and recursively determines the corresponding term in the above series.

```
(a-fives 0) -> 8
(a-fives 1) -> 13
```

b) Develop the function `seq-a-fives`, which consumes a natural number `n` and creates the sequence of the first `n` terms according to `a-fives`. Hint: Use `build-list`.

Problem 3.

Given the following data definitions:

```
;; A Grade is: (make-grade Symbol Number)
(define-struct grade (letter num))

;; The Symbol in a Grade represents

;;     'A  >= 90
;;     'B  >= 80
;;     'C  >= 70
;;     'D  >= 60
;;     'F  < 60

;; A [Listof Grades] ...
(define grades
 (list (make-grade 'D 62) (make-grade 'C 79) (make-grade 'A 93)
       (make-grade 'B 84) (make-grade 'F 57) (make-grade 'F 38)
       (make-grade 'A 90) (make-grade 'A 95) (make-grade 'C 76)
       (make-grade 'A 90) (make-grade 'F 55) (make-grade 'C 74)
       (make-grade 'A 92) (make-grade 'B 86) (make-grade 'F 43)
       (make-grade 'C 73)))
```

Design the requested functions to manipulate `Grades`. You *must* use the given list as one of your tests.

For each you may use a `local` function or an anonymous (`lambda`) function.

Note: if you do not use the DrRacket *loop* function mentioned, you will not receive credit for the sub-problem!

a) Design the function `log->los` that converts a `[listof Grade]` into a `[Listof Symbol]` that contains just the letter grade, using the Racket function map.
b) Using `foldr`, design the function `average-grade` that finds the average (number) Grade in a `[Listof Grade].`
c) Design a function `all-above-79` that returns a list of only the grades that are above 79, using `filter`.
d) Use `andmap` to design the function `all-pass?` that checks to see if all the Grades in a given list are not 'F.
e) Finally design the function `bonus` that adds 5 to all of the Grades in a given list, and updates the letter portion of the Grade if it changes. Use map to design your function... it *must* return a `[Listof Grade]`!