CS 2500, Spring 2014
Problem Set 10

Due date: Tuesday, March 25 @ 7pm

Programming Language: Intermediate Student with Lambda

Finger Exercises HtDP/2e: 262, 263, 264, 265, 267, 268

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.

a) Complete the following parametric data definition for a non-empty list:

```
; an [NEListof X] is one of...
```

b) Design the function `all-int-squares` which takes in a non-negative integer *n* and returns a `[NEList-of Number]` with the squares of all integers from 0 to *n*, inclusive (i.e. including both 0 and *n*).

c) Write down the ~~parametric~~ data definition for a `UOP` (unary operator) which is any function that takes in a Number and returns a Number. Then abstract `all-int-squares` to `all-int-results` which takes in a non-negative integer *n* and a `UOP` *o* and returns a `[NEList-of Number]` with the results of applying *o* to all integers from 0 to *n*, inclusive. Redesign your `all-int-squares` to use `all-int-results`.

d) Design `all-int-doubles` which uses `all-int-results` and a helper `UOP`, defined with `local` inside `all-int-doubles`, that multiplies its input by two.

Problem 2.

a) Design a function `find-string` that takes in a `[List-of String]` and a `String` and that returns a Boolean, true if and only if the given string was in the list.

b) Abstract `find-string` to `generic-find-string` so that the string comparison operation it uses is a parameter. Then use this abstraction to define `find-string-case-sensitive`, which should operate the same way as the original `find-string`, and `find-string-case-insensitive`, which has the same contract as `find-string` but which ignores the case of alphabetic characters when comparing

strings (i.e. the character `a` is considered the same as `A` and so on; non-alphabetic characters must still match exactly).

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 ISL 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]`!

Problem 4.
Here is a data definition:

```
(define-struct child (father mother name date hair-color))
;; A FTN (Family-tree-node) is one of:
;; - empty
;; - (make-child FTN FTN Symbol Number Symbol)
```

Use the above data definition to solve the following problems:

a) Develop the function count-older that consumes a family-tree-node and a year and returns the number of people in the tree that were born that year or earlier.
b) Develop the function search-tree-older that consumes a family-tree-node and a number and produces a list of all of the people in the tree born in that year or earlier.
c) Develop the function red-haired-ancestors? that determines whether a family-tree-node contains an ancestor with red hair on the father's side *and* an ancestor with red hair on the mother's side.
d) Develop the function update-father that consumes two family-tree-nodes and produces a family-tree-node that updates the father of the first tree to be the second tree.