CS 2500, Spring 2013
Problem Set 9

Due date: Wednesday, March 27 @ 11:59pm

Programming Language: Intermediate Student with Lambda

You must work on and submit this homework with your assigned partner. New partners have been assigned in Blackboard. 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: hw9-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.

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 `[NEListof 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 `[NEListof 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) Write a function `find-string` that takes in a `[Listof 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.
Revise your Frogger project once more. Use `local` and "loops" (abstractions such as `map, foldr, filter`, *etc.*) wherever your functions may benefit from them, especially for the lists of objects in your project.

You should notice that the length of your program decreases considerably.
If you have a new partner you actually have two different code bases from which you can start. You are free to pull code from both.

Problem 4.

a) Develop data definitions for binary trees of `Symbols` and binary trees of `Numbers`. The numbers and symbols should occur at the leaf positions only.

Create two instances of each, and *abstract* over the data definitions.
Design the function `height`, which consumes any binary tree of either type and computes its height. That is, the maximum number of `nodes` from the root of the given tree to any leaf (including the leaf, but not including the root node in the count). Here are some tests to further explain:

```
(check-expect (height 5) 0)
(check-expect (height (make-node 'yes (make-node 'no
                                         'maybe))) 2)
```

b) A leafy binary tree is a binary tree with the symbol `'leaf` at its leaves.

Design a function that consumes a natural number `n` and creates (a list of) all leafy binary trees of height `n`.
Hint: Design a function that consumes a natural number `n` and creates (a list of) all leafy binary trees of height *equal or less than* `n`.
For instance, `(all-bt 2)` should produce

```
(list
    (make-node (make-node 'leaf 'leaf)
               (make-node 'leaf 'leaf))
    (make-node (make-node 'leaf 'leaf) 'leaf)
    (make-node 'leaf (make-node 'leaf 'leaf)))
```