CS116 Winter 2011 Assignment 4
Due: Tuesday, February 8 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 `a4qY.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 must use generative recursion to solve all but the last problem. Otherwise, you **will not receive any correctness marks**.
5. Structural recursion, accumulative recursion and abstract list functions may also be used.
6. Use `local` for all named helper functions and constants.
7. When a short and simple function is used only once as a helper to an abstract list function, you must use `lambda`.
8. You may assume that all consumed data will satisfy any stated assumptions, unless indicated otherwise.
9. Section 4.4.7 under "*How to Design Programs* Languages" of the Help Desk in DrRacket lists built-in string functions that you may find useful.

Language level: Intermediate Student with `lambda`.
Coverage: Module 4

1. A grid of numbers with n rows and m columns can be represented in Scheme by a list of length n where each element of the list is a list of length m. For example, the 2 by 3 grid to the right can be represented by `(list (list 1 2 3) (list 4 5 6))`.

1	2	3
4	5	6

   Use generative recursion to write a Scheme function `make-grid`, which consumes a list of n^2 numbers and produces an n by n grid of these numbers. You may assume that the length of the consumed list is the square of a positive integer. The order of the numbers in the grid read left-to-right and top-down should match the order of the numbers in the consumed list. For example,
   `(make-grid (list 1 2 3 4 5 6 7 8 9))` should produce
   `(list (list 1 2 3) (list 4 5 6) (list 7 8 9))` and
   `(make-grid (list 17))` should produce `(list (list 17))`.

   This question is based on Exercise 27.2.4 in the HtDP text.

2. DNA is often modelled by strings of the characters A, C, G and T. They are very long and so often need to be compressed. A simple way to do this is to replace all substrings of identical consecutive characters with the character followed by the length of the substring. These substrings must be as long as possible. For example, the run-length encoding of the string "AAGCCCCTTAAAAAAAAAA" is the string "A2G1C4T2A10". This is the unique run-length encoding – something like "A2G1C4T2A4A6" is not valid.

   Use generative recursion to write a function called `rle-encode` that consumes a DNA string and produces its run-length encoding. Here are two examples:

   ```
   (rle-encode "AAGCCCTTAAAAAAAAAA") => "A2G1C3T2A10"
   (rle-encode "") => ""
   ```

   The consumed strings will only consist of upper-case A, C, G and T.

3. Use generative recursion to write a function called `rle-decode` that consumes the run-length encoding of a DNA string (see Question 2) and produces the DNA string.

   Here are two examples:

   ```
   (rle-decode "A2G1C3T2A10") => "AAGCCCTTAAAAAAAAAA"
   (rle-decode "") => ""
   ```

   You may find the built-in functions `string->number`, `number->string` and `make-string` helpful.

   The consumed strings will only consist of upper-case A, C, G and T and positive integers.

4. Silly strings can be constructed as follows.

   - An empty string is a silly string.
   - If `ss` is a silly string, then `(string-append "a" ss)` is a silly string.
   - If `ss` is a silly string, then `(string-append "b" ss "cc")` is a silly string.
   - If `ss` is a silly string, then `(string-append ss "ddd")` is a silly string.

   For example, `"a"`, `"bcc"`, `"aabccddd"` and `"aaabaadddcc"`, are silly strings. The strings `"abc"`, `"bcca"`, `"dddd"` and `"aabccxddd"` are not silly strings.

   Use generative recursion to write a function `silly-string?` that consumes a string `s` and returns `true` if `s` is a silly string and `false` otherwise.

   You must work directly with strings and may not use lists in your solution.

5. This question requires you to fully understand a sorting algorithm studied in lecture. You may but are not required to use generative recursion in your solution.

   Write a function `num-comps` that consumes a list of numbers and produces the number of comparisons performed by insertion-sort as given on Slides 5 and 6 of Module 4. The number of comparisons is the number of times that the `<` (less-than) function is applied. Here are two examples:

   ```
   (num-comps (list 1 2 3 4 5)) => 4
   (num-comps (list 5 4 3 2 1)) => 10
   ```

   Hint: There are many ways to solve this problem. Here are two different approaches:
   - *For each number, count how many times it will be swapped with a number following it.*
   - *Modify insertion sort itself to produce both a sorted list and the number of comparisons it performs.*