CS 135 Fall 2015
Akinyemi, Heinle, Holtby, Istead, Roegiest, Tompkins

Assignment: 6

Due:	Tuesday, October 27, 2015 9:00pm
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Pure Structural Recursion, Structural Recursion with an Accumulator, and Generative Recursion
Files to submit:	`recursion.rkt`, `estimate.rkt`, `matrix.rkt`, `patient.rkt`, `bonus.rkt`
Warmup exercises:	HtDP 12.2.1, 13.0.3, 13.0.4, 13.0.7, 13.0.8
Practise exercises:	HtDP 12.4.1, 12.4.2, 13.0.5, 13.0.6

- Unless specifically asked in the question, you are not required to provide a data definition or a template in your solutions for any of the data types described in the questions. However, you may find it helpful to write them yourself and use them as a starting point.
- Unless otherwise stated, if *X* is a known type then you may assume that (*listof X*) is also a known type.
- If you create a helper function for the sole purpose of modifying the parameters that were *Str*s and making them (*listof Char*), you must still provide a contract and purpose for the helper function. However, you are not required to provide examples and tests for this function. Your wrapper function, that consumes *Str*s, requires the complete design recipe.
- You may use the abbreviation (*list* ...) or quote notation for lists as appropriate.
- In this assignment you will not be penalized for having an inefficient implementation (e.g. using *append*). Your *Code Complexity/Quality* grade will be determined by how clear your approach to solving the problem is.
- You may reuse the provided examples, but you should ensure you have an appropriate number of examples and tests.
- Your solutions must be entirely your own work.
- For this and all subsequent assignments, you should include the design recipe for all functions, including helper functions (with the exception noted above), as discussed in class.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
- You may **not** use the Racket functions *reverse*, *make-string*, *replicate* or *string-append* in any of your solutions. You may **not** use the built-in Racket function *list-ref*, but you may use your own *my-list-ref* function from question 1.
- You must use the **cond** special form, and are not allowed to use **if** in any of your solutions.
- You may only use the list functions that have been discussed in the notes, unless explicitly allowed in the question.

Here are the assignment questions you need to submit.

1. This question involves recursion. Place your solutions in the file `recursion.rkt`

   (a) Write a function *fibonacci-slow* that consumes a natural number n and produces the n-th Fibonacci number using the well-known formula $F_n = F_{n-1} + F_{n-2}$ with initial values $F_0 = 0$ and $F_1 = 1$.

   For example:
   (fibonacci 0) produces 0
   (fibonacci 1) produces 1
   (fibonacci 2) produces 1
   (fibonacci 3) produces 2
   ...
   (fibonacci 7) produces 13

   The body of your *fibonacci-slow* function **must** include two recursive applications. For example: (*fibonnacci-slow* (− *n* 1)) and (*fibonacci-slow* (− *n* 2)).

   Tip: Avoid trying *fibonacci-slow* on values of n much greater than 35.

   (b) Write a function *fibonacci-fast* that produces identical values to (and has the same contract as) *fibonacci-slow*. *fibonacci-fast* cannot be recursive and must use a helper function *fibonacci-acc* that uses accumulative recursion. The body of your *fibonacci-acc* function **must** have **only one** recursive application.

   As a hint, we have provided a partial solution to *fibonacci-acc*, which has four parameters:

```
(define (fibonacci-acc m n fm-2 fm-1)
  (cond [(= m n) (+ fm-2 fm-1)]
        [else (fibonacci-acc (add1 m) n ... )]))
```

   Tip: Unlike *fibonacci-slow*, you may try *fibonacci-fast* on much larger values.

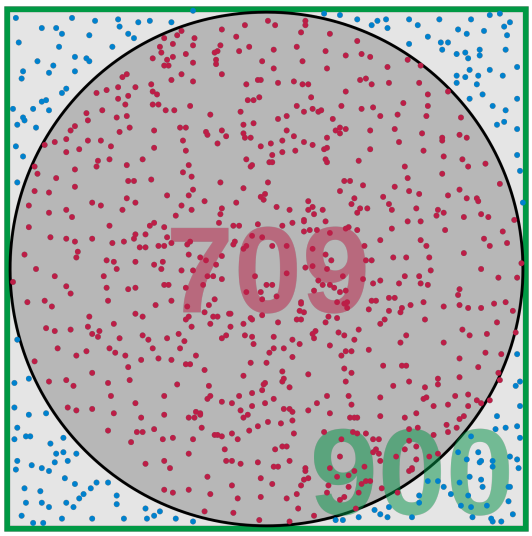
   (c) Write a function, *my-list-ref* that behaves the same as the built-in function *list-ref*. In other words, *my-list-ref* consumes a list and a Natural number n (where n is less than the length of the list) and produces the n-th element of the list (starting at 0). For example, (*my-list-ref* '(a b c) 1) produces 'b. For your **tests** (but not examples), you may use the built-in *list-ref* function. Your function does not have to produce any errors for invalid arguments.

   (d) Write a function, *string-of-char*, that consumes a number n and a character c and produces a string of n cs. For example: (string-of-char 5 #\z) produces zzzzz.

   (e) Write a function, *replace-vowels*, that consumes a string and replaces each vowel with a number of xs: the first vowel is replaced with 1 x, the second with 2 xs and the n-th with n xs. You may use *append* for this question. A vowel is one of (a, e, i, o, u). Recursion should occur on a list of characters, not on the string. For example: (replace-vowels "hello world") produces "hxllxx wxxxrld".

2. A Monte Carlo simulation repeats a randomized experiment to obtain a statistical result. These simulations can be used to estimate both complex calculations and simpler values.

   For example, to estimate π, we can generate n uniformly-random points within the unit square and count the number of those points within the unit circle.



   In the above example (borrowed from `wikipedia`), 709 of the 900 randomly generated points were inside of the unit circle. This results in an estimate of π to be ≈ 3.151. The accuracy improves with more random points.

   To generate each point, x and y co-ordinates are generated in the range $[-1,1]$ and each point is considered to be inside of the unit circle if $(x^2+y^2 \leq 1)$.

   A Monte Carlo approximation for π with n points where *inside* points are within the unit circle is $4 * inside/n$.

   Write a function, *estimate-pi*, that consumes a positive integer n and uses Monte Carlo simulation to estimate π with n uniformly-random points. Do not first create a list of n points and then produce the result. Instead, use *accumulative recursion* to generate one point at a time, maintaining a running count for *inside*.

   We have provided the function *random-number* in `estimate.rkt` that you must use in your implementation. Do not change this function.

   Testing code that uses random number generators is beyond the scope of this course. For this question only, do not include examples or tests for any function using *random-number* or it's result.

   Place your solution in the file `estimate.rkt`, below the definition of *random-number*.

3. Matrices are useful tools in mathematics and computer sciences. For the sake of simplicity, consider a matrix to be a 2D grid of elements/numbers where each number is indexed by row and column.

   For example, the following 3x3 matrix A that has 3 rows and 3 columns with element a_{ij} at row i and column j:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

   Using this notation, row 0 is

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \end{pmatrix}$$

   and column 1 is

$$\begin{pmatrix} a_{01} \\ a_{11} \\ a_{21} \end{pmatrix}$$

   In Racket, we can model a Matrix as a list of rows, i.e., a list of lists, and each row list has the same length (it contains the same quantity of numbers).

   For example, the matrix:

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

   can be represented in Racket as (**define** *M* (*list* (*list* 1 2 3) (*list* 4 5 6) (*list* 7 8 9))).

   (a) Write a function, *matrix-row*, that consumes a matrix (list of lists) and a row number and produces that row of the matrix. For example,
   (*matrix-row M* 0) produces (*list* 1 2 3).

   (b) Write a function, *matrix-col*, that consumes a matrix and a column number and produces that column of the matrix. For example,
   (*matrix-col M* 1) produces (*list* 2 5 8).

   (c) Write a function, *get-element*, that consumes a matrix, a row number and a column number, and produces the element at that row and column position. For example,
   (*get-element M* 1 2) produces 6.

   (d) Matrices A and B can be added by the element-wise addition of their elements. For example,

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

   Using element notation, $c_{ij} = a_{ij} + b_{ij}$ for all indices i, j.

   Write a function, *matrix-add*, that consumes two matrices with identical dimensions and produces the result of adding them.

(e) Consider the matrix multiplication $C = AB$. Using element notation, the formula for multiplication is given by

$$c_{uv} = \sum_{i=0}^{n} a_{ui} b_{iv}$$

Write a function, *matrix-multiply*, that consumes two matrices and produces the result of multiplying them. The dimensions of the two matrices do not need to be equal. If matrix A is an $m \times n$ matrix, then matrix B must be an $n \times p$ matrix, it will produce a matrix C with dimensions $m \times p$.

Implementation Hint: The element notation formula for matrix multiplication can be written as the dot product between two lists. What are those lists? Doing some examples by hand may help solve this riddle. You may use the implementation of *dot-product* found in Module 6.

Testing hint: As with number multiplication, there is a zero and a one for matrix multiplication that give the expected result. Find these matrices and use them for testing.

Place your solutions in the file `matrix.rkt`

4. A patient arriving into an emergency room is assessed by a triage nurse. The nurse creates a patient profile that contains their name (string), systolic blood pressure (positive integer), blood oxygenation level (positive number) and temperature (in F, positive number).

   A *Patient-Profile* is a (*list Str Int Num Num*) that corresponds to the above characteristics.

   (a) Emergency rooms serve patients by priority. That is, patients with serious injuries or illness are treated before those with paper cuts.

   Write a function, *patient-severity*, that consumes a *Patient-Profile* and produces the severity score S defined by

$$S = 0.5 \times BPSys + Oxy + TempF$$

where:

BPSys	=	$(70 - systolic)$ if systolic is < 70
	=	$(systolic - 150)$ if systolic is > 150
	=	0 if $70 <= systolic <= 150$
Oxy	=	$2^{(90 - oxygenation)}$ if $oxygenation < 90$
	=	0 otherwise
TempF	=	$(95 - temperature)$ if $temperature < 95$
	=	$(temperature - 104)$ if $temperature > 104$
	=	0 if $95 <= temperature <= 104$

For example:

(*list* "Alice" 65 87 100) has a severity of 10.5 and
(*list* "Brian" 140 97 105) has a severity of 1

(b) Triage nurses maintain a *Priority-List* of patients sorted so that doctors can serve the most severe patients first. A *Priority-List* is a (*listof* (*list Num Patient-Profile*)) where the *Num* is the calculated *patient-severity* for the corresponding *Patient-Profile*. Furthermore, a *Priority-List* is sorted so that the most severe patients (**largest** severity) are at the beginning of the list. If two patients have the same severity, it does not matter which order they appear in the list.

Write a full data definition and template function for a *Priority-List*.

(c) Write a function, *insert-priority-list*, that consumes a *Priority-List* and a *Patient-Profile* and produces a new *Priority-List* that has the patient inserted into a correct location.

(d) Write a function, *sort-priority-list* that consumes an unsorted list of *Patient-Profile*s (not a *Priority-List*, which is a sorted list of pairs) and produces a *Priority-List*.

(e) Write a function, *merge-priority-list* that consumes two *Priority-List*s and produces a *Priority-List* containing all of the pairs from the two consumed *Priority-List*s.

(f) Occasionally, triage nurses overlook symptoms and assess a severity that is lower than it should be. This is an unacceptable risk to the patients and hospital, so the list of patients is frequently double-checked.

Write a function, *fix-priority-list*, that consumes a *Priority-List* that may have faulty pairs (i.e., when the severity does not match the corresponding *Patient-Profile*). For example, (*list* 5 (*list* "Alice" 65 87 100)) has an incorrect severity. *fix-priority-list* produces a new, properly sorted *Priority-List* with no faulty pairs.

Place your solutions in `patient.rkt`

5. **5% Bonus**: An emergency room is staffed by n doctors, each with a doctor ID from 0 to $(n-1)$. The doctors serve patients in order of severity until all patients have left the hospital. It takes 10 minutes for every severity point to help a patient, e.g., a patient with a severity score of 3 takes 30 minutes to help. Non-integer severity scores are rounded up (e.g., 3.5 becomes 4 and takes 40 minutes to help). As soon as a doctor is free, they serve the next most severe patient immediately. If two doctors are free at the same time, the doctor with the lowest ID serves the most severe patient.

   Write a function, *doctor-allocation* that consumes two parameters: the number of doctors n (at least one) and a *Priority-List* and produces a list of size n with one entry for each doctor (i.e., the first entry in the list corresponds to doctor 0, the second entry for doctor 1, and so on). Each entry is a list containing the total number of minutes worked, followed by the list of patients seen by that doctor (in any order). In other words,
   doctor-allocation produces a (*listof* (*list Nat* (*listof Patient-Profile*)))

   Place your solution in `bonus.rkt`

This concludes the list of questions for which you need to submit solutions. As always, check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

The IEEE-754 standard, most recently updated in 2008, outlines how computers store floating-point numbers. A floating-point number is a number of the form

$$0.d_1 d_2 \ldots d_t \times \beta^e$$

where β is the base, e is the integer *exponent*, and the *mantissa* is specified by the t digits $d_i \in \{0, \ldots, \beta - 1\}$. For example, the number 3.14 can be written

$$0.314 \times 10^1$$

In this example, the base is 10, the exponent is 1, and the digits are 3, 1 and 4. A computer uses the binary number system; the binary equivalent to the above example is approximately

$$0.11001001_2 \times 2^{10_2}$$

where 10_2 is the binary integer representing the decimal number 2. Thus, computers represent such numbers by storing the binary digits of the *mantissa* ("11001001" in the example), and the *exponent* ("10" in the example). Putting those together, a ten-bit floating-point representation for 3.14 would be "1100100110".

However, computers use more binary digits for each number, typically 32 bits, or 64 bits. The IEEE-754 standard outlines how these bits are used to specify the mantissa and exponent. The specification includes special bit-patterns that represent Inf, $-$Inf, and NaN.