CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment: 6

Due:	Tuesday, October 31, 2017 9:00pm
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Pure Structural or Accumulative
Files to submit:	`tabular.rkt`, `anagrams.rkt`, `trees.rkt`
Warmup exercises:	HtDP 12.2.1, 13.0.3, 13.0.4, 13.0.7, 13.0.8
Practice exercises:	HtDP 12.4.1, 12.4.2, 13.0.5, 13.0.6

- **Make sure you read the OFFICIAL A06 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 05 carry forward.
- Unless otherwise stated, if *X* is a known type then you may assume that (*listof X*) is also a known type.
- You may only use the functions that have been discussed in the lecture slides, unless explicitly allowed or disallowed in the question. In particular, you may NOT use any of the following Racket functions: *reverse*, *make-string*, *replicate*, *member*, *member?* or *list-ref*. You may define your own versions of these functions as long as they are written using pure structural recursion.
- If your function is a wrapper function, it will require all design recipe components, but the helper function which it calls will require only the purpose, contract, and definition: no examples or tests are needed for the helper function.

1. This question involves manipulating tabular data. Place your solutions in the file `tabular.rkt`

   Tabular data is a common way to represent individual data that has common characteristics. For example, all students in this course have a variety of marks (for each assignment, for each midterm, etc.), but all students have the same number of marks. We can store information for one individual in one row, where each column represents the particular piece of data we wish to store. For example, the table

$$\begin{pmatrix} 8 & 3 & 4 & 9 \\ 3 & 7 & 5 & 6 \\ -1 & 1 & -3 & 0 \end{pmatrix}$$

   can be one way to represent four pieces of information for three different people. We can reference particular elements by their row and column position: for example, the number 1 is in row 2, column 1. That is, rows and columns are indexed starting from 0, as was discussed in the *mult-table* example in Module 06.

   We will represent this tabular data in Racket by way of a list of lists of numbers. For example, the above tabular data can be represented as:

   (*list* (*list* 8 3 4 9) (*list* 3 7 5 6) (*list* −1 1 −3 0)).

   The data definition below will be helpful in your contracts and when writing your functions:

```
;; A Table is a (listof (listof Num))
;; requires: each sub-list has the same length
```

   (a) Write the function *mult-by* which consumes a number and a table, and produces the table resulting from each number in the original table being multiplied by the consumed number.

   (b) Write the function *get-elem* which consumes a row (as a natural number), a column (as a natural number) and a table, and produces the number which is in that row and column in the table. If the table is not large enough in either dimension, this function should produce *false*. There is a built-in function, called *list-ref*, which you are not permitted to use. You may write your own version of *list-ref* if you wish.

   (c) Write the function *col* which consumes a column number and table, and produces a list of all numbers in that column in the table, when read from top to bottom (i.e., starting from row 0). If there is no such column, this function should produce the empty list. For example, (*col* 2 (*list* (*list* 8 3 4 9) (*list* 3 7 5 6) (*list* −1 1 −3 0)) ⇒ (*list* 4 5 −3). You may not use (or write your own version of) *length* for this question: use structural recursion to detect if there is no such column.

   (d) Write the function *sum-tables* which consumes two tables (of the same dimensions) and produces the table which results from adding up the elements pairwise between the two tables. Specifically, the value at row *r*, column *c* in the produced table is the sum of the values at row *r*, column *c* in the consumed tables.

2. This question concerns anagrams. An *anagram* is a word or phrase formed by rearranging the characters of some other word or phrase. Well-known examples of anagrams include:

   - `"listen"` and `"silent"`
   - `"eleven plus two"` and `"twelve plus one"`

   This question will determine if two strings are anagrams of one another using two different techniques. For part (c), you should use **accumulative recursion**; for all other parts you should use **structural recursion**, if recursion is required.

   (a) Write the function *sort-chars* which consumes a list of characters and produces a sorted list (in increasing order) containing exactly those characters. There may be duplicate characters in the list. For example, (*sort-chars* (*list* #\o #\r #\d #\e #\r)) ⇒ (*list* #\d #\e #\o #\r #\r). Using the function *string->list* for your examples and testing would be a good idea.

   (b) Using part (a), write the function *anagrams/sort?* which consumes two strings and produces *true* if the two strings are anagrams of each other, and *false* otherwise.

   (c) Write the function *freq-count* which consumes a list (of any type) and produces a list of pairs, where the first element of each pair is an element from the list and the second element of each pair is the number of times that element appeared in the consumed list. You will need to use a helper function which uses **accumulative recursion**. For example, (*freq-count* '(red 7 9 (7 9) red 9)) ⇒ '(((7 9) 1) (9 2) (7 1) (red 2)). Note that the elements in the produced list may be in any order. Since *freq-count* is a **wrapper function**, read the first page about the design recipe components required for this part of the question.

   (d) Write the function *freq-equiv?* which consumes two lists containing elements of type (*list Any Nat*) and determines if they are rearrangements of each other. For example:
   (*freq-equiv?* '((red 5) (blue 9) ("string" 0)) '((blue 9) (red 5) ("string" 0))) ⇒ *true*
   (*freq-equiv?* '((red 5) (blue 6)) '((blue 5) (red 6))) ⇒ *false*
   (*freq-equiv?* '((blue 5)) '((red 4) (blue 5))) ⇒ *false*
   You can assume that there is at most once occurrence of each "key" (first element of the pair) in each list. This restriction will allow you to use structural recursion. (*Edit Oct 21, 9:44am:* added uniqueness clarification.)

   (e) Write the function *anagrams/count?* which consumes two strings and produces *true* if the two strings are anagrams of each other, and *false* otherwise. You must use parts (c) and (d) to solve this problem: that is, you should count the frequency of each character in the given strings, and then determine if those counts are equivalent.

   Place your solutions in the file `anagrams.rkt`.

3. Using the definition of a binary tree (BT) given on slide 08-32, write functions which operate on binary trees.

For the purposes of explanations below, we will use:

```
(define exampleBT
  (make-node 1 "a"
    (make-node 7 "b" empty empty)
    (make-node 3 "c" (make-node 7 "d" empty empty) empty)))
```

(a) Write the function *height* which consumes a binary tree and produces the height of the tree. The height is specified as:

- the empty tree has height 0;
- the tree composed of just the root has height 1;
- otherwise, the height of a tree is one more than the height of its tallest subtree.

The height can be thought of as the number of nodes along the longest path from the root to a leaf node. You may find the Racket function *max* helpful to avoid recursing on the same subtree more than once (which would lead to an exponential blowup similar to the function *max-list* outlined in slide 07-6). For example (*height exampleBT*) ⇒ 3.

(b) Write the function *find-in-tree* which consumes a binary tree and a list of symbols, where those symbols are either 'L (for left) or 'R (for right). The function *find-in-tree* should either return the key which is in the node rooted at the tree after following those movements, starting at the root of the tree, or, if that path goes beyond a leaf in the tree, the function should produce *false*. For example:
(*find-in-tree exampleBT* '(R L)) ⇒ 7
(*find-in-tree exampleBT empty*) ⇒ 1
(*find-in-tree exampleBT* '(L L)) ⇒ *false*

(c) Write the function *prune* which consumes a binary tree and a number, and produces the binary tree where all subtrees rooted at the consumed number (as a key) have been removed from the consumed tree. Be aware that since the consumed value is a binary tree, there may be many subtrees with the consumed number as the key, and all such subtrees should be removed. For example:
(*prune exampleBT* 1) ⇒ *empty*
(*prune exampleBT* 7) ⇒ (*make-node* 1 "a" *empty* (*make-node* 3 "c" *empty empty*))
(*Edit Oct 21, 9:40am:* updated second example output to be correct.)

Place your solutions in `trees.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.