Assignment: 6

Due:	Tuesday, February 23, 2016 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:	`clicker.rkt`, `student.rkt`, `collection.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*, *string-append* or *list-ref* in any of your solutions.
- 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. For this question, place your solution in the file `clicker.rkt`.

   Every lecture contains clicker questions in which students enter an answer of A, B, C, D or E. However, students do not always attend lectures and are hence unable to provide a response. We can use a symbol to represent each possible clicker response or 'none if the student did not enter a response. The student clicker responses for the term can be represented by (*listof* (*anyof* 'A 'B 'C 'D 'E 'none)) and the correct answers to the clicker questions by (*listof* (*anyof* 'A 'B 'C 'D 'E)).

   Write a function *clicker-grade* that consumes, in the following order, a list of student clicker responses (*listof* (*anyof* 'A 'B 'C 'D 'E 'none)) and a list of correct answers (*listof* (*anyof* 'A 'B 'C 'D 'E)) and produces a *Num*, the resulting clicker participation grade. The two consumed lists have equal length and should be consumed in the order given above. Students are rewarded 1 mark for responding to the question and an additional 1 mark if the answer submitted is correct. The resulting grade will be a fractional value between 0 (none answered) and 100 (at least 75% answered correctly).

   You may assume that the two lists have equal length, the total number of clicker questions asked in class is greater than zero and the list length is divisible by 4. The grading details are also posted on the course webpage under "Grading & Marks". We have also posted a solution for the Assignment 1 bonus question that you may use (or use your own) as a helper function.

2. Place your answers for this question in the file `student.rkt`.

   (a) Write the function *tallest* which consumes a list of symbols (representing first names of students) and a list of positive numbers (representing the corresponding height of the students). You can assume that both lists are the same length and both are non-empty. The function should produce the name of the student who is the tallest: in the case of a tie, produce the name of the first student in the list with that maximal height. **For this part, you must use pure structural recursion.**

   (b) Write the function *shortest* which consumes a list of symbols (representing first names of students) and a list of positive numbers (representing the corresponding height of the students). You can assume that both lists are the same length and both are non-empty. The function should produce the name of the student who is the shortest: in the case of a tie, produce the name of the first student in the list with that minimal height. **For this part, you must use accumulative recursion.** Hint: write an accumulative helper function (with more parameters than *shortest*) that keeps track of the information you know so far as you progress down the list. That is, your main function does not need to use accumulative recursion, but your helper function should.

   (c) Write the function *student-al* which consumes a list of distinct symbols (representing first names of students) and a list of positive numbers (representing the corresponding height of the students). You can assume that the lists have the same length. The function *student-al* should produce an association list (i.e., a list of lists) with the keys being the student names (symbols) and the values being the height of the corresponding student (positive numbers). The order of the keys should be the same as the consumed list of names.

   (d) Write the function *basketball* which consumes an association list as created in the previous part (i.e., a list of lists, with the keys being the names as symbols, and the values being the numerical height of the students) and a positive height and produces the list of names (i.e., symbols) for those students that are at least as tall as the given height, in the same order that the names appear in the consumed list.

3. For this question, place your solution in the file `collection.rkt`.

   A disorganized collector, has been purchasing magazines without keeping track of which ones he already owns and which ones he still needs to complete his collection. We will represent each individual *Magazine* by using the following structure:

   > (*define-struct magazine* (*title issue*))
   > ;; A Magazine is a (make-magazine Str Nat)

   You may assume that all magazines use sequential issue numbers starting with 1.

   The collector has made a rudimentary list, in no particular order, of the magazines in his collection as an unsorted (*listof Magazine*).

   An *Index* is an **association list** where **key** is a *Str* representing the title of a *Magazine* and the **value** is a sorted non-empty list of *Nat* (with no duplicates) that represents the magazine issue numbers that the collector owns. The **key**s in the index are sorted by using *string<?*.

   > ;; An Index is one of:
   > ;; * empty
   > ;; * (cons (list Str (listof Nat)) Index) where (listof Nat) is a non-empty list

   (a) Write a predicate *magazine<?* that consumes two *Magazine*s and compares them lexicographically (for sorting); i.e. the function produces *true* if the first *Magazine* is lexicographically strictly less than the second *Magazine* and *false* otherwise. If the two *Magazine*s are equal, the function should produce *false*. The lexicographical order of a *Magazine* is primarily determined by the *title*, breaking ties by the *issue* number. The built-in function *string<?* can be used to determine the lexicographical order of *Str*s used in the *title*.

   (*magazine<?* (*make-magazine* "Dragon" 27) (*make-magazine* "Dungeon" 6)) ⇒ *true*
   (*magazine<?* (*make-magazine* "Dragon" 22) (*make-magazine* "Dragon" 27)) ⇒ *true*

   (b) Write a function *sort-magazines* that consumes a (*listof Magazine*) and produces a sorted (*listof Magazine*) according to *magazine<?*. You may use any sorting technique discussed in class or on a previous assignment.

   (c) Write a function *need-between* that consumes a sorted (*listof Magazine*), a *Str* representing a magazine *title*, a *Nat* (a low issue number bound) and a *Nat* (a high issue number bound) and produces a sorted (*listof Nat*) corresponding to the magazine *issue*s, with the consumed *title*, between the low and high bounds (inclusive) **not** found in the consumed (*listof Magazine*). You may assume that the low issue bound is less than or equal to the high issue bound. For example:

   (**define** *my-slom* (*list* (*make-magazine* "Dragon" 2) (*make-magazine* "Dragon" 3)))
   (*need-between my-slom* "Dragon" 2 3) ⇒ *empty*
   (*need-between my-slom* "Dungeon" 2 3) ⇒ (*list* 2 3)

(d) After sorting his collection of magazines and removing duplicates, the collector wants to compare his collection with a fellow collector who has completed their collection to determine if he has everything. Write a predicate *magazine-lists-equal?* that consumes two sorted (*listof Magazine*)s and produces *true* if the two lists are the same. You may not use the built-in functions *member?* or *equal?* but may assume that there are no duplicate magazines in each list, individually.

(e) Deciding to buy the collection, the collector must now merge the two collections (each individually sorted) into one. Write a function *merge-collections* that consumes two sorted (*listof Magazine*)s and produces the sorted (*listof Magazine*). The collector only wishes to keep one copy of each distinct magazine so the produced list should not contain any duplicates.

(f) Write a function *create-index* that consumes a sorted (*listof Magazine*) and produces an *Index*, data definition given above. The *Index* is an association list that contains a unique **key** for each magazine title with the corresponding **value** being the sorted non-empty list of magazine issue numbers in ascending order (with no duplicates). Remember that the (*key*, *value*) pairs should also be sorted in ascending lexicographic order by *key*. For example:

```
(define my-slom (list (make-magazine "Dragon" 1) (make-magazine "Dragon" 10)
                      (make-magazine "Omni" 19)))
(create-index my-slom) ⇒ (list (list "Dragon" (list 1 10)) (list "Omni" (list 19)))
```

(g) Write a function predicate *own-magazine?* that consumes an *Index* (as described in the previous part) and a *Magazine*, in that order, and produces *true* if and only if the magazine is in the index.

(h) **5% Bonus**:

Write a function *need-magazines* that consumes an *Index*, a *Str* (a magazine *title*) and a *Nat* (the most recent magazine *issue* number) and produces a *Str* where the string starts with the magazine title, followed immediately by a colon and one space, then one of the following:

- a list of all the *issue* numbers (with the consumed magazine *title*) separated by commas and a single space from 1 to the consumed *issue* number, inclusive, that are not in the *Index*
- "completed" if all issues from 1 to the *issue* number (inclusive) are found in the *Index*
- "need *all*" if there are no issues of this magazine found in the *Index*

For example:

```
(need-magazines (list (list "Dragon" (list 1 2 3))) "Dragon" 3) ⇒ "Dragon: completed"
(need-magazines (list (list "Dragon" (list 1 2 3))) "Dragon" 7) ⇒ "Dragon: 4, 5, 6, 7"
(need-magazines (list (list "Dragon" (list 1 2 3))) "Dungeon" 4) ⇒ "Dungeon: need all"
```

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

$$0.d_1d_2\dots d_t \times \beta^e$$

where β is the base, e is the integer *exponent*, and the *mantissa* is specified by the t digits $d_i \in \{0,\dots,\beta-1\}$. 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.