CS 135 Fall 2012
Ali,Fraser,Goldberg,Hoey,
Kaplan,Kogan,Petrick,Tompkins

Assignment: 6

Due:	Tuesday, October 23, 2012 9:00pm
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Pure structural recursion
Files to submit:	`clicker.rkt`, `collection.rkt`, `recursion.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

You can re-use the provided examples, but you should ensure you have an appropriate number of examples and a sufficient number of tests. All sorted lists are to be sorted in non-decreasing (ascending) order. Functions that produce ordered output should do so intrinsically, and must not sort or reverse an intermediate result. You may not use the Scheme functions *reverse* or *remove* or any functions not discussed in class. When in doubt, ask a question in Piazza. 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* (*union* 'A 'B 'C 'D 'E 'none)) and the correct answers to the clicker questions by (*listof* (*union* 'A 'B 'C 'D 'E)).

   Write a function *clicker-grade* that consumes, in the following order, a list of student clicker responses (*listof* (*union* 'A 'B 'C 'D 'E 'none)) and a list of correct answers (*listof* (*union* '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 divisible by 4. The grading details are also posted on the course webpage under "Grading & Marks". You may use the model solution (or your own) from Assignment 2 as a helper function.

2. 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 = (make-magazine String 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 *String* representing the title of a *Magazine* and the **value** is a sorted (*ne-listof 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 String (ne-listof Nat)) Index)
```

(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 *String*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*). 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 *String* representing a magazine *title*, a *Nat* (a low issue number bound) and *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*. The *Index* is an association list that contains a unique **key** for each magazine title with the corresponding **value** being the sorted nonempty 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 *String* (a magazine *title*) and a *Nat* (the most recent magazine *issue* number) and produces a *String* 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*"

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

Below are three functions each with 3 parameters. For each function create a list of *Symbol*s representing the type of recursion used in the function followed by a description of how the function makes use of each argument (in the order listed in the function definition).

Represent the type of recursion used by one of the following *Symbol*s:

- 'pure for pure structural recursion
- 'acc for structural recursion with an accumulator
- 'gen for generative recursion

Represent each argument by one of:

- 'onestep if the argument is used for getting one step closer to a base case
- 'accumulator if the argument is used as an accumulator
- 'generated if the argument is a generated value
- 'alongride if the argument is "going along for the ride"

Create a list of lists containing your answers for all three functions (in order of Function A B C) and store the result as a constant called *my-a6q3-soln* in the `recursion.rkt` file. The constant definition must have the following format:

```
(define my-a6q3-soln (list (list 'recursionA 'argA1 'argA2 'argA3)
                           (list 'recursionB 'argB1 'argB2 'argB3)
                           (list 'recursionC 'argC1 'argC2 'argC3)))
```

where each 'recursion and each 'arg symbol should be replaced with a symbol from above.

```
;; Function A
(define (sum-from-to x y z)
  (cond
    [(= y z) x]
    [else (sum-from-to (+ x y) (add1 y) z)]))

;; Function B
(define (sum-by-mn l m n)
  (cond
    [(empty? l) 0]
    [else (+ (* m n) (first l) (sum-by-mn (rest l) m n))]))

;; Function C
(define (sum-race-up x y z)
  (cond
    [(= z x) x]
    [else (+ x (sum-race-up (* x 2) (remainder y x) z))]))
```

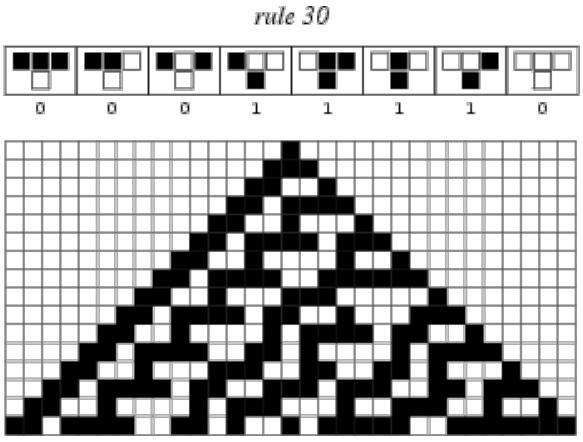
This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the public test results after making a submission.

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

A cellular automaton is a way of describing the evolution of a system of cells (each of which can be in one of a small number of states). This line of research goes back to John von Neumann, a mathematician who had a considerable influence on computer science just after the Second World War. Stephen Wolfram, the inventor of the Mathematica math software system, has a nice way of describing simple cellular automata. Wolfram believes that more complex cellular automata can serve as a basis for a new theory of real-world physics (as described in his book "A New Kind of Science", which is available online). But you don't have to accept that rather controversial proposition to have fun with the simpler type of automata.

The cells in Wolfram's automata are in a one-dimensional line. Each cell is in one of two states: white or black. You can think of the evolution of the system as taking place at clock ticks. At one tick, each cell simultaneously changes state depending on its state and those of its neighbours to the left and right. Thus the next state of a cell is a function of the current state of three cells. There are thus 8 (2^3) possibilities for the input to this function, and each input produces one of two values; thus there are 2^8 or 256 different automata possible.

If white is represented by 0, and black by 1, then each automaton can be represented by an 8-bit binary number, or an integer between 0 and 255. Wolfram calls these "rules". Rule 0, for instance, states that no matter what the states of the three cells are, the next state of the middle cell is white, or 0. But Rule 1 says that in the case where all three cells are white (the input is 000, which is zero in binary), the next state of the middle cell is black (because the zeroth digit of 1, or the digit corresponding to the number of 2^0s in 1, is 1, meaning black). In the other seven cases, the next state is white.



This is all made clearer by the pictures at the following URL, from which the picture at the right is taken:

`http://mathworld.wolfram.com/CellularAutomaton.html`

Some of these rules, such as rule 30, generate unpredictable and apparently chaotic behaviour (starting with as little as one black cell with an infinite number of white cells to left and right); in fact, this is used as a random number generator in Mathematica.

You can use DrScheme to investigate cellular automata and draw or animate their evolution, using the `io.ss`, `draw.ss`, `image.ss`, or `world.ss` teachpacks. Write a function that takes a rule

number and a configuration of cells (a fixed-length list of states, of a size suitable for display) and computes the next configuration.