CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment: 5

Due:	Tuesday, October 22nd, 9:00 pm
Language level:	Beginning Student with list abbreviations
Files to submit:	`recursion.rkt`, `books.rkt`, `tictactoe.rkt`
Warmup exercises:	HtDP 13.0.3, 13.0.4, 13.0.7, 13.0.8, 17.2.1, 17.3.1, 17.6.4, 17.8.3
Practice exercises:	HtDP 12.4.1, 12.4.2, 13.0.5, 13.0.6, 14.2.3, 14.2.4, 14.2.6,17.1.2, 17.6.2, 17.8.4

- **Make sure you read the OFFICIAL A05 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 04 carry forward.
- This assignment covers material up to the end of Module 08.
- Of the built-in list functions, you may use only `cons`, `first`, `second`, `third`, `rest`, `empty?`, `cons?`, `list`, `member?`, and `length`. You may also use `equal?` to compare two lists.
- If you choose to define a type, provide a complete data definition. You do not need to provide a template, but can have one if you want.
- Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.
- Unless the question specifically says otherwise, you are always permitted to write helper functions. You may use any constants or functions from any part of a question in any other part.

1. **[10% Correctness]** Perform the assignment 5 questions (Modules 7 and 8) using the online evaluation "Stepping Problems" tool linked to the course web page and available at

   https://www.student.cs.uwaterloo.ca/~cs135/assign/stepping/.

   The instructions are the same as those on the previous assignments. If you get stuck, please do not post questions to Piazza asking for the next step, since such questions are considered to be violations of our academic integrity policy. If you are really stuck see an ISA or instructor in person.

2. **[10% Correctness]** For this question, you will be writing recursive functions that perform recursion on more than one parameter at a time.

   (a) Write a function `my-list-ref` that consumes a list of numbers and an index, and produces the element in the list at the consumed index. The index of an element is a natural number representing how many elements are in front of it, meaning the first element is at index 0, and the last element as at index (length - 1) If the index is too large, produce the value `false` instead. For example:

   ```
   (check-expect (my-list-ref '(1 2 3 4) 0) 1)
   (check-expect (my-list-ref '(5 4 3) 2) 3)
   (check-expect (my-list-ref '(2) 20) false)
   ```

   Note: the built-in `list-ref` function does not do the same thing. Rather than producing false if the index is too large, it simply **requires** the index is a valid position in the list.

   (b) Write a function `zip` that consumes two lists with the same length (a list of numbers and a list of strings). The function produces an association list where the keys are the elements of the first list, and the values are the corresponding elements of the second list. For example:

   ```
   (check-expect (zip '(1 2 3 4) '("a" "b" "c" "d"))
                 '((1 "a") (2 "b") (3 "c") (4 "d")))
   (check-expect (zip empty empty) empty)
   ```

   (c) Write a function `count-symbol/2D` that consumes two values, a `(listof (listof Sym))` and a `Sym` and produces the total number of times that Symbol occurs in any of the lists of Symbols.

   Example:

   ```
   (check-expect (count-symbol/2D '((a a b) () (a)) 'a) 3)
   (check-expect (count-symbol/2D empty 'x) 0)
   ```

   Place your solutions in `recursion.rkt`

3. **[25% Correctness]** In this question you will be dealing with books. A Book uses the following data definition

```
;; A Book is a (list Str Str)
```

The first `Str` is the title of the book. The second `Str` is the author of the book (in "Last, First" format).

```
;; useful constants for the examples (don't copy and paste these, they
   are in the starter file)
(define my-bookshelf '(("The Colour of Magic" "Pratchett, Terry")
                       ("Mostly Harmless" "Adams, Douglas")
                       ("Pyramids" "Pratchett, Terry")
                       ("A Brief History of Time" "Hawking, Stephen")))

(define discworld-books '(("The Colour of Magic" "Pratchett, Terry")
                          ("Pyramids" "Pratchett, Terry")))
```

(a) Write the template functions for a `Book` and a `(listof Book)`. Call the templates `book-template` and `listof-book-template`.

(b) Write the function `sort-books` to organize your bookshelf. This function consumes a `(listof Book)`, and produces a list of the same Books in shelf-order. This means that the books are sorted according lexicographically to the author's name (in "Last, First" format). Books by the same author should be sorted lexicographically by their title.

For example:

```
(check-expect (sort-books my-bookshelf)
              '(("Mostly Harmless" "Adams, Douglas")
                ("A Brief History of Time" "Hawking, Stephen")
                ("Pyramids" "Pratchett, Terry")
                ("The Colour of Magic" "Pratchett, Terry")))
```

Use the insertion sort algorithm. You can (and should) base your code on the `insert` and `sort` functions from the slides (Module 08, slides 2 through 9).

(c) A friend has never heard of your favorite author! You decide to do the right thing and lend them every book that author has ever written, immediately.

Write the function `books-by-author` that consumes a `(listof Book)` and the name of an author, and produces a `(listof Book)` containing just the books written by that author. The order of the books is unchanged.

For example:

```
(check-expect (books-by-author my-bookshelf "Pratchett, Terry")
              discworld-books)
(check-expect (books-by-author my-bookshelf "King, Stephen")
              empty)
```

The following problems make use of an additional data type, the Author Index.

```
;; An AuthorIndex is a (listof (cons Str (listof Str)))
;; requires: The first Str in each inner list is unique
```

In each inner list, the first string represents the name of the author, and the rest of the strings represent the titles of books by that author. In other words, it is similar to an association list where the keys are `Str` and the values are `(listof Str)`. Similar, but not identical! The `AL` in the slides uses `(list key value)` for the key-value pair, while the `AuthorIndex` uses `(cons key value)` instead, since the value is always a list.

Example:

```
(define my-index '(("Pratchett, Terry"
                    "The Colour of Magic" "Pyramids")
                   ("Hawking, Stephen" "A Brief History of Time")
                   ("Adams, Douglas")))
```

In this index, there are two books by Terry Pratchett ("The Colour of Magic" and "Pyramids"), one by Stephen Hawking ("A Brief History of Time") and none at all by Douglas Adams (someone must have borrowed it).

(d) Write the function `book-by-author?` that consumes an `AuthorIndex`, an author, and a book title, and produces `true` if the index contains a book by that name and author, `false` otherwise.

Examples:

```
(check-expect (book-by-author? my-index
              "Pratchett, Terry"
              "The Colour of Magic")
              true)
(check-expect (book-by-author? my-index "King, Stephen" "It") false)
```

(e) Write the function `build-author-index` that consumes a`(listof Book)` and a list of unique authors (Strings). The function produces an `AuthorIndex` where the keys are the authors consumed (in the same order) and the values are the titles of all books by that author in the list of `Book` (also in the same order).

Example:

```
(check-expect (build-author-index my-bookshelf '("Adams, Douglas"
                                                 "Hawking, Stephen"))
              '(("Adams, Douglas" "Mostly Harmless")
                ("Hawking, Stephen" "A Brief History of Time")))
```

Place your solutions in `books.rkt`

4. **[35% Correctness]**

In this question you will be playing a large game of Tic-Tac-Toe. In this game players take turn placing Xs and Os in a 3x3 grid, with the X player making the first move. A player wins if they managed to fill a row, column, or diagonal with their letter. If there are no free spots, the game is a draw.

In this question you will deal with an $N \times N$ grid, where N is an odd natural number.

We can represent such a grid in Racket using lists of lists.

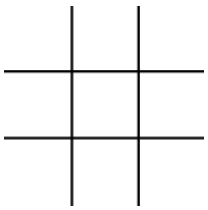
```
;; A Tic-Tac-Toe Grid (T3Grid) is a (listof (listof (anyof 'X 'O '_)))
;; requires: all lists have the same length, and that length is odd
;;           The number of 'X and 'O is equal, or there is one more 'X
```

Here, the symbols `'X` and `'O` (A capital "Oh", not a zero) represent the X and O player, and the symbol `'_` represents a blank square.

Examples

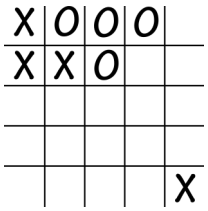
Standard 3x3 Grid, no moves

```
(define grid1
        '((_ _ _)
          (_ _ _)
          (_ _ _)))
```



5x5 grid, 4 X moves, 4 O moves

```
(define grid2
        '((X O O O _)
          (X X O _ _)
          (_ _ _ _ _)
          (_ _ _ _ _)
          (_ _ _ _ X)))
```



Tiny grid, 1 X move, no 0 moves.

```
(define grid3
        '((X)))
```

Because players take turns, the number of Xs will either be equal to the number of Os, or else it will be one greater than the number of Os. As always, you do not need to check this requirement (it is part of the data definition, so it can safely be assumed).

Additionally, for any functions that consume a row number and/or a column number, these numbers are required to be valid. They start counting from 0, so, $0 \leq$ row number, column number $<$ N (where N is the number of rows and columns in the grid).

For this question (and *only* this question) you may use the `list-ref` function.

(a) Write the function `whose-turn` that consumes a `T3Grid` and determines whose turn it is. X goes first, so if the number of Xs and Os is equal, X goes next (produce `'X`). If the number of Xs is 1 greater than the number of Os, O goes next (produce `'O`).

Examples: In `grid1` and `grid2` X goes next. In `grid3` O goes next (the game is over since the grid is full, but the function is still defined since that is a valid `T3Grid`).

(b) Write the function `grid-ref` that consumes a `T3Grid` and a row and column number, and produces the symbol located at that location. Row and column numbers start counting from 0.

Examples:

```
(check-expect (grid-ref grid2 1 2) 'O)
(check-expect (grid-ref grid2 0 0) 'X)
```

(c) Part of figuring out if a player has won is seeing if they have filled in a row or a column. Finding a row is easy, since each row already in list form. Columns, on the other hand, are spread across multiple lists. So, it would be nice to be able to convert them into a list.

Write the function `get-column` that consumes a `T3Grid`, and a column number, and produces a list of the symbols in that column.

Examples:

```
(check-expect (get-column grid1 0) '(_ _ _))
(check-expect (get-column grid2 1) '(O X _ _ _))
(check-expect (get-column grid3 0) '(X))
```

(d) Write the function `will-win?` that consumes a `T3Grid`, a row number, a column number, and a player (either `'X` or `'O`). The function produces true if that player would win by placing a marker at the given location, and false otherwise. Note that if the given location is not blank, then the player will not win (you cannot win by making an illegal move).

To keep things simple, you do not need to worry about checking the diagonals, only rows and columns. (However, you are allowed to check the diagonals if you want to!)

You may assume that nobody has won yet in the consumed board.

Examples:

```
(check-expect (will-win? grid1 0 0 'X) false)
(check-expect (will-win? '((X X _)
                           (O X O)
                           (O _ _))
                         0 2 'X) true)
```

Place your solutions in `tictactoe.rkt`

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

There is a strong connection between recursion and induction. Mathematical induction is the proof technique often used to prove the correctness of programs that use recursion; the structure of the induction parallels the structure of the function. As an example, consider the following function, which computes the sum of the first n natural numbers.

```
(define (sum-first n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum-first (sub1 n)))]))
```

To prove this program correct, we need to show that, for all natural numbers n, the result of evaluating *(sum-first n)* is $\sum_{i=0}^{n} i$. We prove this by induction on n.

Base case: $n = 0$. When $n = 0$, we can use the semantics of Racket to evaluate *(sum-first 0)* as follows:

```
(sum-first 0) ; =>
(cond [(zero? 0) 0][else ...]) ; =>
(cond [true 0][else ...]) ; =>
0
```

Since $0 = \sum_{i=0}^{0} i$, we have proved the base case.

Inductive step: Given $n > 0$, we assume that the program is correct for the input $n-1$, that is, *(sum-first (sub1 n))* evaluates to $\sum_{i=0}^{n-1} i$. The evaluation of *(sum-first n)* proceeds as follows:

```
(sum-first n) ; =>
(cond [(zero? n) 0][else ...]) ;(we know n > 0) =>
(cond [false 0][else ...]) ; =>
(cond [else (+ n (sum-first (sub1 n)))]) ; =>
(+ n (sum-first (sub1 n)))
```

Now we use the inductive hypothesis to assert that *(sum-first (sub1 n))* evaluates to $s = \sum_{i=0}^{n-1} i$. Then *(+ n s)* evaluates to $n + \sum_{i=0}^{n-1} i$, or $\sum_{i=0}^{n} i$, as required. This completes the proof by induction.

Use a similar proof to show that, for all natural numbers n, *(sum-first n)* evaluates to $(n^2 + n)/2$.

Note: Summing the first n natural numbers in imperative languages such as C++ or Java would be done using a `for` or `while` loop. But proving such a loop correct, even such a simple loop, is considerably more complicated, because typically some variable is accumulating the sum, and its value keeps changing. Thus the induction needs to be done over time, or number of statements

executed, or number of iterations of the loop, and it is messier because the semantic model in these languages is so far-removed from the language itself. Special temporal logics have been developed to deal with the problem of proving larger imperative programs correct.

The general problem of being confident, whether through a mathematical proof or some other formal process, that the specification of a program matches its implementation is of great importance in *safety-critical* software, where the consequences of a mismatch might be quite severe (for instance, when it occurs with software to control an airplane, or a nuclear power plant).