Winter 2024
Victoria Sakhnini

Assignment: 5

Due:	Tuesday, February 27, 2024 9:00 pm
Coverage:	Slide 44 of Module 07
Language level:	Beginning Student with List Abbreviations
Files to submit:	`cs135search.rkt`, `tictactoe.rkt`

- **Make sure you read the OFFICIAL A05 post on Piazza** for the answers to frequently asked questions.
- Unless otherwise specified, you may only use Racket language features we have covered up to the coverage point above
- It is possible that your functions will not be very *efficient*, and may be slow on long lists. There is no need to test your functions with excessively long lists.
- It's simpler to write many of the required functions on this assignment by first writing and thoroughly testing a number of helper functions. To encourage this approach, you are required provide **appropriate** tests for each helper function, but you do not need to provide a purpose and contract. You should know the purpose and contract (and if you don't you should rethink what you are doing) but you do not need to explicitly write them out.
- Policies from Assignment A04 carry forward.

Here are the assignment questions you need to solve and submit:

1. **(20%):** Complete all the required stepping problems in **Module 7: Nested Lists** at

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

   You should refer to the instructions from A01 Question 1 for the stepper question instructions.

2. **(40%)**: When using Google to find documents on the web, there are a variety of options available. For example, if you searched for two search terms, say *Association* and *List*, you would get web pages that contain both the words *Association* and *List*. If you searched for *Association* and *-List* you would get web pages that contain *Association* but do not contain *List*.

   When looking for all the occurrences of a word in a collection of documents (such as web pages) rather than searching all the documents sequentially, a more efficient approach would be to use an inverted list (also called an inverted index). For this question, we will represent an inverted list as an association list where the *key* is the search term and the *value* is a list of all the documents that the search term occurs in. The list of documents that contain the key is called a `doc-list (DL)`.

```
;; A doc-list (DL) is one of:
;; * empty
;; * (cons  Str DL)
;; Requires: each doc (i.e. Str) only occurs once in the doc-list
;;           the doc-list is in lexicographic order
```

Lexicographic order is the order determined by the Racket function `string<?`, i.e. if `(string<? "a.txt" "b.txt")` is true then `"a.txt"` should appear in the `doc-list` before `"b.txt"`.

An `Inverted List` is defined as follows.

```
;; An Inverted List (IL) is one of:
;; * empty
;; * (cons  (list Str DL) IL)
;; Requires: each key (i.e. Str) only occurs once in the IL.
;;           the keys occur in lexicographic order in the IL.
```

For example, the following three documents were used to build an `Inverted List`.

	Document Name	Contents
1	a.txt	the cat sleeps
2	b.txt	the dog barks
3	c.txt	suddenly the dog chases the cat

The `Inverted List` would look like the following.

```
(list (list "barks"    (list "b.txt"))
      (list "cat"      (list "a.txt" "c.txt"))
      (list "chases"   (list "c.txt"))
      (list "dog"      (list "b.txt" "c.txt"))
      (list "sleeps"   (list "a.txt"))
      (list "suddenly" (list "c.txt"))
      (list "the"      (list "a.txt" "b.txt" "c.txt")))
```

E.g. `"barks"` only appears in the document `"b.txt"` so the `doc-list (DL)` for `"barks"` is `(list "b.txt")`

You may use the following string comparison predicates `string<?`, `string<=?`, `string=?`, `string>?` and `string>=?` for this question. You may not use `member?`.

Hint: You can take advantage of the fact that the lists are sorted in ascending order to make the search easier. Also be aware that the `doc-list`s can be different lengths.

(a) Create a function `both` which consumes two `DL`s and produces a doc-list (DL) that occur in both `DL`s. For example,
`(both (list "b.txt") (list "b.txt" "c.txt"))` should produce `(list "b.txt")`.

(b) Create a function `exclude` which consumes two `DL`s and produces a doc-list (DL) that occur in the first `DL` but not the second one. For example,
`(exclude (list "b.txt" "c.txt") (list "b.txt"))` should produce `(list "c.txt")`.

(c) Create a function `(keys-retrieve doc an-il)` which consumes a `Str` and an `IL` and produces a `(listof Str)` with lexicographic ordering. The values in the produced list are the keys from `an-il` whose `doc-list`s contain `doc`. If `doc` is not contained in the `doc-list` associated with any keys in `an-il`, then `keys-retrieve` returns `empty`.

For example, if we passed "a.txt" and the `IL` shown at the top of this page to `keys-retrieve`, it would return `(list "cat" "sleeps" "the")`.

(d) Create a function `search` which consumes a `Sym`, two `Str`s and an `IL`. It produces a doc-list (DL). The arguments for `search` will always be in one of two possible formats:

- `(search 'both str1 str2 an-il)` which, given two keys `str1` and `str2` from `an-il`, produces a doc-list (DL) containing the documents that are present in both of the keys' associated `doc-list`s.
  For example, if we pass `'both`, `"barks"`, `"dog"` and the `IL` at the top of the page into `search`, then it would return `(list "b.txt")`.
- `(search 'exclude str1 str2 an-il)` which, given two keys `str1` and `str2` from `an-il`, produces a doc-list (DL) containing the documents that are present in the `doc-list` associated with the key `str1`, but not the key `str2`.

Place your solutions in the file `cs135search.rkt`

3. **(40%)**: 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.

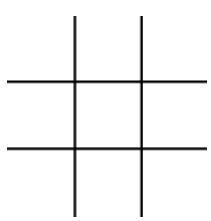
```
;; A Tic-Tac-Toe Grid (T3Grid) is a (listof (listof (anyof 'X 'O '_)))
;; Requires: all lists have the same length, and that length is odd
;;           The number of 'X and 'O is equal, or there is one more 'X
```

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
       (list (list '_ '_ '_)
             (list '_ '_ '_)
             (list '_ '_ '_)))
```



5x5 grid, 4 X moves, 4 O moves

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

X	O	O	O	
X	X	O		
				X

Tiny 1x1 grid, 1 X move, no O moves.

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

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, $0 \leq \text{row number}, \text{column number} < N$ (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) (list '_ '_ '_))
(check-expect (get-column grid2 1) (list 'O 'X '_ '_ '_))
(check-expect (get-column grid3 0) (list '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.

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? (list (list 'X 'X '_)
                               (list 'O 'X 'O)
                               (list 'O '_ '_))
                         0 2 'X) true)
```

Place your solutions in `tictactoe.rkt`

This concludes the list of questions for you to submit solutions, but see the following pages as well, especially the bonus. Don't forget to always check the basic test results after making a submission.

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).