Fall 2024
Abouelkhair, Becker, Jung, Petrick, Reetz, Roh

Assignment: 6

Examples Due:	Thursday, October 31, 2024 9:00 pm
Remainder Due:	Tuesday, November 5, 2024 9:00 pm
Coverage:	Slide 26 of Module 09
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Simple and Accumulative recursion
Files to submit:	`examples-a06.rkt`, `morelistfun.rkt`, `matrix.rkt`, `santa.rkt`

General assignment policies:

- General policies from previous assignments carry forward. **Policies** from the course web site apply.

A06-specific policies and advice:

- Consult the **official A06 Post and FAQ on Ed** for the answers to frequently asked questions.
- For all functions named in the assignment, examples must be submitted by the (earlier) deadline 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.

Here are the assignment questions that you need to submit:

1. **(15%):** Implement the following functions that perform recursion on multiple parameters:

   (a) `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 exceeds the length of the list, produce the value `false` instead. For example:

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

   You may **not** use `length` for this part.

   Note: the built-in `list-ref` function (that you cannot use in this assignment) does not do the same thing. Rather than producing false if the index exceeds the length of the list, it simply **requires** the index to be a valid position in the list.

(b) `zip` consumes two lists with the same length. 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 (list 1 2 3 4) (list "a" "b" "c" "d"))
              (list (list 1 "a") (list 2 "b") (list 3 "c") (list 4
                 "d")))
(check-expect (zip empty empty) empty)
```

(c) `list-xor` consumes two lists of numbers that are sorted in increasing order, `lon1` and `lon2`, and produces a sorted list that contains only the items that are in a or b, but no elements that are contained in both `lon1` and `lon2`. For example, `(check-expect (list-xor (list 1 3 5) (list 2 3 4)) (list 1 2 4 5))`.

Note: You may assume there are no duplicates in either list for this question.

Place your solutions in `morelistfun.rkt`.

2. **(35%):** In Module 08 (slide 30) we introduced a matrix as a list of lists where each nested list represents a row of the matrix and all rows have the same length. For simplicity, consider a matrix to be a 2D grid of items (typically numbers) where each item is indexed by row and column.

   For example, the following 3x3 matrix A that has 3 rows and 3 columns with item a_{ij} at row i and column j:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

   Using this notation, row 0 is

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \end{pmatrix}$$

   and column 1 is

$$\begin{pmatrix} a_{01} \\ a_{11} \\ a_{21} \end{pmatrix}$$

   Note that the number of rows in a matrix may not be the same as the number of columns. A matrix with no elements is represented by empty.

   For example, the matrix:

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

   is represented as `(define M (list (list 1 2 3 4) (list 5 6 7 8)))`.

   (a) Write a data definition for a `Matrix`.

(b) Write a function `matrix-item` that consumes a `Matrix`, a row number and a column number, and produces the item at that row and column position. For example, `(matrix-item M 1 2)` produces `7`.

(c) Write a function `matrix-col` that consumes a `Matrix` and a column number, and produces that column of the matrix. For example, `(matrix-col M 2)` produces `(list 3 7)`.

(d) The transpose of a matrix A is another matrix A^T where each row of A becomes a column of A^T. For example, For example, the matrix:

$$M^T = \begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{pmatrix}$$

Write a function `matrix-transpose` that consumes a `Matrix` and produces the transpose of the one consumed.

(e) Matrix multiplication is a useful tool in mathematics and computer science; for example, image rotation in computer graphics.

The multiplication of two matrices A and B results in a matrix C and is denoted as $C = AB$. The items in C are defined by the formula

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

Note: this is the dot product of a row from A with a column from B. The dimensions of the two matrices do not need to be equal. If matrix A is an $m \times n$ matrix, then matrix B must be an $n \times p$ matrix, it will produce a matrix C with dimensions $m \times p$.

Write a function, `matrix-multiply`, that consumes two matrices and produces the result of multiplying them.

Hint: For testing, you may want to consider multiplying by a matrix with 1s down the diagonal and 0 everywhere else. This is known as the identity matrix and the product with another matrix will simply be the other matrix.

Place your solutions in `matrix.rkt`.

3. **(50%):** Santa Claus runs a worldwide surveillance network to spy on children. He knows when they are sleeping, he knows when they are awake, and he keeps track of all the naughty and nice actions in their lives to determine whether they deserve presents.

   Santa has watched children for a long time and has assigned a niceness score (see data definition below) to all the naughty and nice actions children make. The niceness score and description of the action are stored in an `Action` structure defined below.

As the current children are observed and their actions recorded, Santa creates an association list dictionary `ActionList` using a child's name as the key and the list of their actions as the associated value. Note: we will assume the child's name is unique (even though this is not realistic). Also, all strings are case-sensitive, so, `"Mark"` and `"mark"` represent two different children.

The `Action` structure and `ActionList` are defined as follows:

```
;; A Name is a Str

;; A Desc is a Str
;; Note: This could be a description of an Action or a Wish (part c).

;; A NiceScore n is an Int
;; Requires: -100 <= n <= 100 and n not= 0

(define-struct action (niceness desc))
;; An Action is a (make-action NiceScore Desc)

;; An ActionList is a (listof (list Name (listof Action)))
;; Requires: The list is sorted alphabetically by child name.
;;           Each list of Actions is non-empty.
;; Note: the order of Actions for the same child is arbitrary.
```

For example:

```
(define action1 (make-action 3 "Prepared assignment question"))
(define action2 (make-action -7 "Questions are too hard"))
(define actlst (list (list "Zaphod" (list action1 action2))))
```

Note: an additional larger example is provided in the file `santa-ex.rkt` that you can use to help test your implementations.

(a) Write a function `extreme-actions` that consumes a child's name and an `ActionList` and produces either:

- `empty` if the child's name does not appear in the `ActionList`.
- a list of two strings where the first string is the description of the child's action with the lowest niceness score and the second string is the description of the child's action with the highest niceness score.

If two `Actions` have the same niceness score, choose the first one in the list. If the child has only one recorded `Action`, the extremes will be the same.

For example:

```
(check-expect
  (extreme-actions "Zaphod" actlst)
  (list "Questions are too hard" "Prepared assignment question"))
```

Restriction: You must use accumulative recursion to find the lowest and highest naughty/nice actions. Your implementation should only make 1 pass through the `(listof Action)` to determine both values at the same time; i.e., you should **not** do one pass through the list to find the lowest and then a second pass to the find the highest.

(b) At various times, Santa gets reports from his agents around the world and needs to update his action list. An `ActionUpdate` is a list containing fixed-length lists of a child's name and a new `Action`. The updates are sent in frequently, so, in an update, a child will only have one new action.

```
;; An ActionUpdate is a (listof (list Name Action))
;; Requires: The list is sorted alphabetically by child name.
;;           A name will only appear once in the list.
```

Write a function `merge-actions` that consumes an `ActionList` and an `ActionUpdate` and produces a new `ActionList` with any new children and actions included. Since a `(listof Action)` in the `ActionList` is unsorted, place the new `Action` from the `ActionUpdate` at the front of the corresponding child's `(listof Action)`.

For example:

```
(define action3 (make-action 42 "Told a good joke about recursion."))
(define newactlst (list (list "Zaphod" (list action3 action1
   action2))))
(check-expect (merge-actions actlst (list (list "Zaphod" action3)))
   newactlst)
```

Note: since both lists are already sorted, you should perform a merge on the two lists. Inserting each new action into Santa's action list one at a time (similar to insertion sort) would be very inefficient.

(c) Throughout the year (some children are very eager), children send Santa wish lists of the gifts they would like to receive. Santa goes through each list and stores each gift request in a `Wish` structure that stores the score Santa determines is needed to obtain the gift and a description of the gift. A child's `Wish`es are then sorted in non-decreasing score order and stored in `Wishlist`; i.e., gift with the lowest score is first. The children's `Wishlist`s along with their names are then stored in a `ChildrenList`. The structure and data definitions are as follows:

```
(define-struct wish (score gift))
;; A Wish is a (make-wish NiceScore Desc)
;; Requires: score is further restricted to be > 0
```

```
;; A WishList is a (listof Wish)
;; Requires: Wishes are sorted in non-decreasing order by score.

;; A ChildrenList is a (listof (list Name Wishlist))
;; Requires: The list is sorted alphabetically by child name.
```

For example:

```
(define wish1 (make-wish 32 "Amigurumi Bee Plushie"))
(define wish2 (make-wish 99 "Wayne Gretzky Rookie Card"))
(define chldlst (list (list "Zaphod" (list wish1 wish2))))
```

It's now time for Santa to decide which gifts each child will receive. Santa's gift choices are stored in a list for each child in a `GiftList`:

```
;; A GiftList is a (listof (list Name (listof Desc)))
;; Requires: The GiftList is sorted alphabetically by child name.
;;           The gifts (listof Desc) are sorted in non-increasing
;;           order of score; i.e., gift with highest score is first.
```

Note: Santa wants his list of gifts to be sorted in non-increasing order of the niceness score (highest score first) because when he puts the presents under the tree, he wants the best gift to placed the furthest back so it will be the last one the child opens.

In this question, you are asked to write two functions. First, write a function `choose-gifts` that consumes an overall niceness score that is an `Int` and a `(listof Wish)` and produces a `(listof Desc)`. This function determines the gifts a child will receive based on their overall niceness score, and the niceness scores Santa assigned to the gifts in their `(listof Wish)` according to the following rules, in the following order:

- If $N < 0$ then Santa gives them `"coal"`.
- If $N = 0$ then Santa gives them `"socks"`.
- If $N > 0$ but the child did not send a wish list to Santa, Santa gives them `"socks"`.
- If $N > 0$ but N is less than the score of the first `Wish` on the child's `WishList`, Santa gives them `"socks"`.
- Otherwise, if $N > 0$ then they receive all of the gifts with niceness less than or equal to N from their `Wishlist`.

A set of tests is provided in `santa-ex.rkt` to help you verify the correctness of your implementation.

The second function to write is `assign-gifts` that consumes an `ActionList` and a `ChildrenList` and produces a `GiftList`. A child's overall niceness score (used to determine which gifts they are given) is the sum of their niceness scores in their `(listof Action)`s from the `ActionList`. If the child is not listed in the `ActionList`

but is listed in the `ChildrenList`, Santa assigns them a niceness score of `0`. Use the `choose-gifts` function from above to determine the gifts given.

For example:

```
(check-expect (assign-gifts actlst chldlst)
              (list (list "Zaphod" (list "coal"))))
(check-expect (assign-gifts newactlst chldlst)
              (list (list "Zaphod" (list "Amigurumi Bee Plushie"))))
```

Note: the ordering of gifts in the `GiftList` for a child is the reverse ordering compared with their `WishList`. **You may not use `append`** to add items to the end of the list (this would be rather inefficient). **You may not use `reverse`** to fix the ordering of your list. *Hint*: Use accumulative recursion to make the list in reverse order.

Also Note: Similar to part b), since both lists are already sorted, you should only do one pass through each list and process both lists as the same time.

Place your solutions in `santa.rkt`.

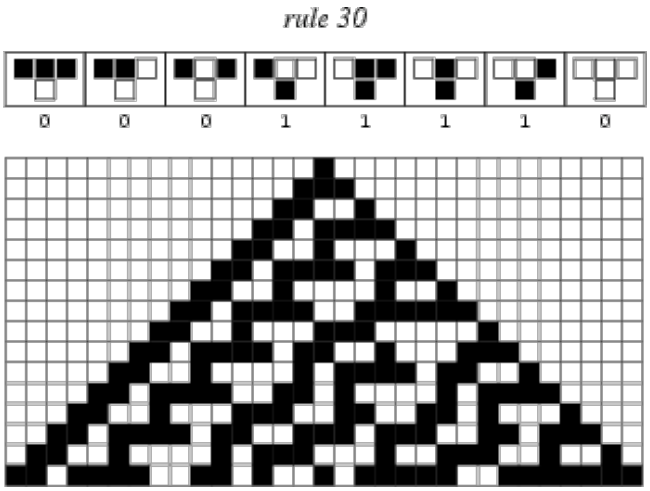
This concludes the list of questions for you to submit solutions. Don't forget to always check the basic test results after making a submission.

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

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". The following images shows the evolution of Rule 30 through 16 clock ticks, starting at the top of the 16x32 grid.



This illustration starts the automaton with a single black cell. At the next tick, it stays black because "010" (the cell and it's neighbour on each side translated from white-black-white to binary) is 2 and the $(2+1)^{nd}$ bit from the left in the rule is 1 (black). The cell's neighbour on the right is translated from black-white-white to "100" or 4 in binary. That bit in Rule 30 is also black. Similarly, the left neighbour translates to "001" which also outputs black. All the other cells translate to white-white-white and thus remain white.

To code such an automaton in Racket, start with a function named `applyRule` that consumes a list of cells (a list of 0s and 1s) and a representation of the rule – either an integer between 0 and 255 or a list of 1s and 0s. `applyRule` produces another list of cells.

After `applyRule` is working, use the `2htdp/image` package referenced in the A03 Enhancements to visualize the results.

For further information, start with `http://mathworld.wolfram.com/CellularAutomaton.html`, the origin of the illustration above.