Winter 2024
Victoria Sakhnini

Assignment:	6
Due:	Tuesday, March 19, 2024 9:00 pm
Coverage:	End of Module 10
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Simple and Accumulative recursion
Files to submit:	`recursion.rkt`, `inventory.rkt` , `bst-full.rkt`, , , `bonus-a06.rkt`

- Make sure you read the A06 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- Policies from Assignment A05 carry forward.
- Unless otherwise specified, you may only use Racket language features we have covered up to the coverage point above.

Here are the assignment questions that you need to submit:

1. **(30%)**: You **must use** accumulative recursion to implement the following functions:

   (a) `in-range`, which consumes two numbers (`a` and `b`) and a list of numbers. It produces the number of elements in the list that are between `a` and `b` (inclusive). The range is inclusive, so the numbers 3, 3.141 and 4 are all between 4 and 3.

   `in-range` will use exactly one helper function, `in-range/acc`. `in-range/acc` will not use any helper functions. Built-in functions are permitted, of course.

   (b) `spread`, which consumes a non-empty list of numbers and produces the non-negative difference between the maximum element in the list and minimum element of the list. If the maximum element is the same as the minimum element, `spread` produces zero.

   `spread` will use exactly one helper function, `spread/acc`. `spread/acc` will not use any helper functions. Built-in functions are permitted, of course.

   (c) `sel-sort` consumes a list of numbers and sorts them into non-decreasing order using an algorithm known as "selection sort". That is, it does the same thing as insertion sort from the beginning of M08, but uses a different approach.

   The core idea of insertion sort is to take the first number off the list and insert it in the right place in the the recursively sorted rest of the list.

   The core idea of selection sort is to select the right number (the smallest one!) and `cons` it onto the front of the recursively sorted remainder of the list.

   There are four functions involved:

i. `sel-sort` is a wrapper function for `sel-sort/sf`. It also handles an empty list.

ii. `sel-sort/sf` consumes a non-empty list of numbers where the smallest element in the list is the first one ("sf" stands for "smallest first") and produces a sorted list.

    `sel-sort/sf` uses generative recursion: the recursive application of `sel-sort/sf` is applied to a list that is **not** one step closer to the base case using the data definition (even though it is one element shorter).

    Need a hint?

iii. `smallest-first` consumes a non-empty list of numbers and produces that same list but with a smallest element at the beginning of the list. The order of the remaining elements is not specified.

iv. `smallest-first/acc` is a helper function for `smallest-first` that must use accumulative recursion.
    Need a hint?

Hints:

- Start with `smallest-first`. Once that is working, work on `sel-sort` itself.
- `smallest-first` may be hard to test because the order of the list is unspecified. Focus on small tests.

Place all your solutions in the file `recursion.rkt`

2. **(50%)**: Best Electric Vehicles specializes in selling electric vehicles (EVs). The company would like to organize it's inventory (a list of EVs on their lot). The record for each EV contains the following fields:

   - `model`: a string to represent the make and model of the EV (e.g., Hyundai Kona, Audi e-Tron, Chevrolet Bolt, Tesla Model 3, etc.).
   - `year`: a natural number indicating the year of manufacture.
   - `price`: the selling price of the EV.
   - `mileage`: the mileage reading from the odometer of the EV.
   - `mpge`: the advertised miles per gallon equivalent for the EV. (MPGe)[1]. For example, the Hyundai Kona has a MPGe of 132 in the city, whereas some models of the Audi eTron are as low as 62 MPGe in the city.

   (a) Write a structure definition for `ev` and an accompanying data definition for the type `EV`. The capitalizations in the preceding sentence are important! The fields in your structure definition **must match** the names and order specified above.

   (b) Write a function `adjust-prices` that consumes a list of electric vehicles and a number representing a percent change. It produces a list with the same vehicles such that the

[1] Gas Mileage of All-Electric Vehicles

price of each vehicle has been lowered or raised by the amount given. For example, a $30,000 car with a change of 0.10 (10%) should have a new price of $33,000. A change of -0.10 (-10%) results in a new price of $27,000. The order of the list should be preserved.

(c) Write a function `(build-inventory models years prices mileage mpge)`. Each parameter is a list; all the lists are the same length. The ith value in each list is an attribute of the ith car in the inventory – the car's price off the `prices` list, the car's mpge off the `mpge` list, etc. `build-inventory` produces a list of `EV`s with the same information.

You **must use** accumulative recursion to implement `build-inventory`.

```
(define models (list "Hyundai Kona" "Audi e-Tron" "Chevy Bolt"
        "BMW i4" "Tesla M3" "Nissan Leaf"))
(define years   (list  2022  2021  2020  2022   2022   2013))
(define prices  (list 36000 40000 35000 53000  60000   5000))
(define mileage (list 12000  4000  3050    25      5 230000))
(define mpge    (list   132    63   123    75    133     98))


(check-expect (build-inventory models years prices mileage mpge)
 (list (make-ev "Nissan Leaf"  2013  5000 230000  98)
       (make-ev "Tesla M3"     2022 60000      5 133)
       (make-ev "BMW i4"       2022 53000     25  75)
       (make-ev "Chevy Bolt"   2020 35000   3050 123)
       (make-ev "Audi e-Tron"  2021 40000   4000  63)
       (make-ev "Hyundai Kona" 2022 36000  12000 132)))
```

(d) Write a function `compare-ev` which consumes two `EV`s and produces `'lt` if the first one is "less than" the second, `'eq` if they are equal, and `'gt` if the first one is "greater than" the second.

The comparisons should be based on `year`, `mpge`, and `mileage`. Older cars are "less than" newer cars. If they are the same age, cars with a lower efficiency (`mpge`) are "less than" cars with higher efficency. If they are the same age and efficiency, cars with more miles are "less than" cars with fewer miles.

(e) Write a function `sort-evs` that consumes a list of electric vehicles and produces an inventory list in the following order:

i. year (descending order)
ii. mpge (descending order)
iii. mileage (ascending order)

If two `EV`s compare equal, their order in the resulting list does not matter.[2]

[2] In practise, this means that we're not going to test that situation and you don't need to either.

Hint: You may cut, paste, and adapt your `sel-sort` code from Q1. Alternatively, you may adapt the insertion sort code from lecture. In either case, design recipes for the helper functions are not needed.

Place all your functions in the file `inventory.rkt`.

3. **(20%)**: Binary search trees are most useful for searching when every choice of search direction (left or right) eliminates about half of the remaining nodes. The degree to which this occurs depends on the shape of the tree. Trees that look mostly linear (i.e., trees with minimal branching) do not significantly speed up searching. On the other hand, "bushier" trees tend to speed up searches considerably. There are various ways to characterize trees that tend to speed up searches. We will explore one of these in this problem.

   Recall the structure and data definition of a binary search tree from Module 10:

```
(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; requires: key > every key in left BST
;;           key < every key in right BST

;; A Binary Search Tree (BST) is one of:
;; * empty
;; * Node
```

   (a) A binary tree is called *full* if every node possesses either 0 or 2 children. Full trees by their very nature cannot be strictly linear. Write a function `full?` that consumes a binary search tree and produces `true` iff the given binary search tree is full. The empty tree, paradoxically, is considered full.

   (b) Although the fullness property of a binary tree does provide for maximum branching by preventing any node from having exactly one child, this property alone does not guarantee particularly fast searches. Write a function `single-path?` that consumes a binary search tree and produces `true` iff the given binary search tree looks like a single path. A binary search tree looks like a single path iff all children are left children, or all children are right children. Assume that the given tree has at least two nodes.

   Place your solutions in `bst-full.rkt`.

This concludes the list of questions for which you need to submit solutions.

Remember to always test your code in DrRacket before submitting and to always check your basic test results after submitting.

Bonus (5%): Write a new version of `list-xor`, named `list-xor-dups`, that consumes two lists of integers in non-decreasing order. That means there may be duplicates. Produce a list sorted in increasing order consisting of all the integers that appear on one of the lists, but not both, with duplicates removed. For example,

```
(check-expect (list-xor-dups  (list 4 4 4 5 5 6 6) (list 1 2 3 3 3 4 4 4 7))
              (list 1 2 3 5 6 7))
```

Restrictions:

- No helper functions.
- You may only use **`cond`**, **`and`**, **`or`**, `not`, `<`, `=`, `first`, `rest`, `empty?`, `cons?`, and `cons`.

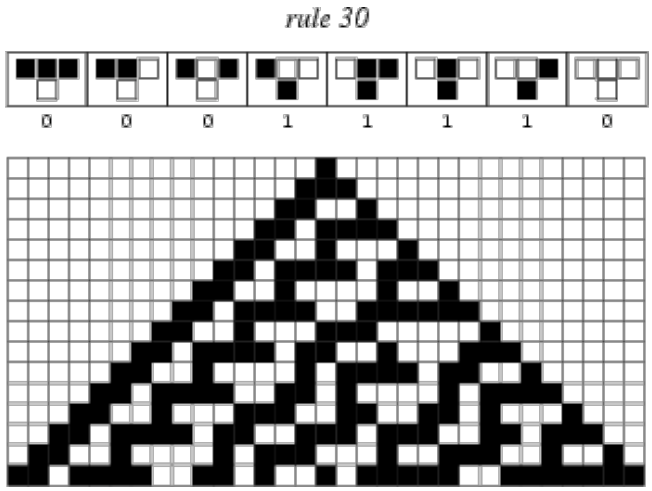
Place your solution in `bonus-a06.rkt`.

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.