Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

Assignment: 6

Assignment:	6
Due:	Tuesday, November 1, 2022 9:00 pm
Coverage:	End of Module 10
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Simple and Accumulative recursion; Generative as guided
Files to submit:	`recursion.rkt`, `inventory.rkt`, `sets.rkt`

- Make sure you read the A06 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- Your solutions for assignments will be graded on both correctness and on readability. This assignment will be auto-graded and hand-marked.
- Policies from Assignment A05 carry forward.
- Submit examples for all questions by **Friday, October 28 at 8:00AM** (before the due date of assignment A06). See the Example Submissions and Basic Test Results post on Piazza for examples of correct and incorrect example submissions. Examples are expected for `smallest-first` in Q1c, but none of the other named helper functions in Q1.

Here are the assignment questions that you need to submit.

1. 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. 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.

[1] Gas Mileage of All-Electric Vehicles

(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 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 i^{th} value in each list is an attribute of the i^{th} 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`, `mgpe`, 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 EVs compare equal, their order in the resulting list does not matter.[2]

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. Consider the data definition:

```
;; A (setof X) is a (listof X)
;;   Requires:  There are no duplicates
;;              The list is sorted in ascending order
```

Note that the second requirement (sorted) means that two `X`'s can be ordered. We can use this data definition for numbers and strings, but for symbols we would need to do something else. Note: For this question you are banned from using `member?`. Any helpers that mimic the behaviour of `member?` will also result in point deductions.

(a) Write a function definition for `(union s1 s2)` where `s1` and `s2` are each a `(setof Nat)`. It produces $s1 \cup s2$, a `(setof Nat)` with all the elements that occur in either `s1` or `s2` with no duplicates.

(b) Write a function definition for `(intersection s1 s2)` where `s1` and `s2` are each a `(setof Nat)`. It produces $s1 \cap s2$, a `(setof Nat)` with all the elements that occur in both `s1` and `s2` with no duplicates.

Place your solutions to this problem in file `sets.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.

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

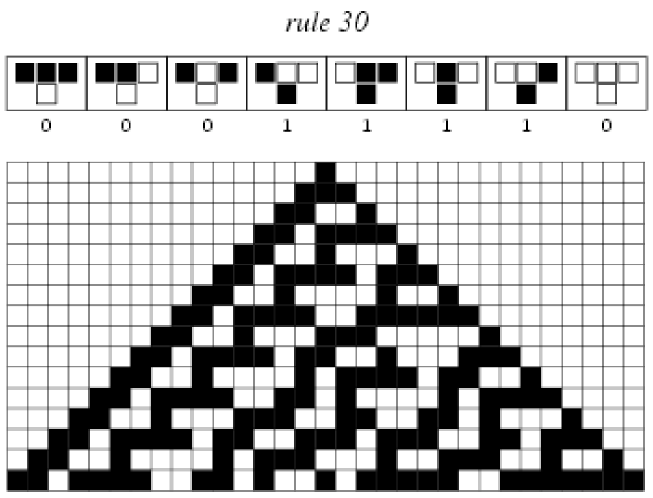
A cellular automaton is a way of describing the evolution of a system of cells (each of which can be in one of a small number of states). This line of research goes back to John von Neumann, a mathematician who had a considerable influence on computer science just after the Second World War. Stephen 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

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

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 DrRacket 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.