**Assignment:** 6  
**Examples Due:** Friday, November 3, 2023 8:00 am  
**Remainder Due:** Tuesday, November 7, 2023 9:00 pm  
**Coverage:** End of Module 10  
**Language level:** Beginning Student with List Abbreviations  
**Allowed recursion:** Simple and Accumulative recursion  
**Files to submit:** examples-a06.rkt, morelistfun.rkt, signmag.rkt, bst-full.rkt, santa-sus.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.
- Submit examples for questions 1, 2(c), 2(d), 3(a), and 4 by the examples due date specified above.

Here are the assignment questions that you need to submit:

1. 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. At a very low level in most computers, there is no such thing as a negative number. There are a few ways that computers pretend to handle negative numbers; one is called the sign-magnitude representation of numbers. We will implement the sign-magnitude representation using Racket.

(a) Write the structure definition for a SignMag that complies with the following data definition:

   A SignMag is a (make-signmag Sym Num)
   requires: sign is one of 'positive, 'negative, 'zero
             mag > 0
             if sign is 'zero, then mag = 1

   The SignMag representation of a positive number p has sign as 'positive and mag as p.
   The SignMag representation of a negative number n has sign as 'negative and mag as −n.
   The SignMag representation of the number 0 has sign as 'zero and mag as 1. SignMag
   values with sign as 'zero and mag as anything other than 1 are not allowed.

(b) Write a template, signmag-template, for your structure definition.

(c) Write a function num->signmag that consumes a number and produces its equivalent
    SignMag value.

(d) Write a function called signmag->num that consumes a SignMag structure and produces
    its equivalent number.

Place your solutions in signmag.rkt.
3. 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:

```scheme
(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. Create a constant `bad-full-tree` that encodes a full binary search tree on the numbers 1,2,3,4,5,6,7,8,9, with the maximum possible height. (The height of a tree is defined as the maximum number of edges traversed from the root to a leaf.)

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

4. 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 tallies up the naughty and nice actions in their lives to determine whether they deserve presents. Each naughty or nice action taken by every child is recorded in its own Action structure:

```scheme
(define-struct action (name niceness desc))
;; An Action is a (make-action Str Int Str)
;; ;; requires: -100 <= niceness <= 100
;; ;; niceness cannot be zero
```

The first field represents a child’s name (which we will assume is unique, even though this is totally unrealistic). The second field represents a niceness score (which can only be between
-100 and 100). A positive score means the action is nice and a negative score means the action is naughty. Every score is either positive or negative; Action items with a zero score are not recorded. The third field is a textual description of the action. For example:

(make-action "Byron" 3 "Prepared tutorial questions")

means that Byron prepared tutorial questions, which is worth a niceness score of 3. Note that all strings are case-sensitive, so "Byron" and "byron" represent two different children.

Santa stores all the children’s actions in a list. There is an example of a (small) list of actions in q4provided.rkt.

Children also submit to Santa lists of Wishes:

(define-struct wish (score gift))

;; A Wish is a (make-wish Nat Str)
;; requires: score > 0

The first field represents a minimum niceness score required to receive the gift. The second field is a description of the gift the child wishes to receive. For example:

(make-wish 32 "Amigurumi Bee Plushie")

indicates that a gift of an Amigurumi Bee Plushie requires a niceness score of 32.

Each child stores their wishes in a WishList:

;; A WishList is a (listof Wish)
;; requires: wishes are sorted in non-increasing order by niceness
;; score.

Santa maintains the children’s WishLists in an association list where the keys are child names:

;; A ChildrenList is either
;; * empty
;; * (cons (list Str WishList) ChildrenList)

There is an example of a ChildrenList in q4provided.rkt.

Place your solutions in santa-sus.rkt.

(a) Write a function extreme-actions that consumes a child’s name and a list of Actions and produces a list of strings. If the child’s name does not appear in the list of Actions, produce empty. Otherwise, produce a list of two strings. The first element in the produced list is the description of the action the child performed with the smallest niceness score, and the second element is the description of the action the child performed with
the largest niceness score. If two actions have the same niceness score, choose the first one in the list. Note that if the child has only one recorded Action, the extremes will be the same.

Here are some examples:

(check-expect
  (extreme-actions "Zahra" instr-actlst)
  (list "Was disobedient at the grocery store" "Cleaned her room without being nagged"))
(check-expect
  (extreme-actions "Patrick" instr-actlst)
  (list "Joined a bicycle gang" "Joined a bicycle gang").

Restriction: You must only go through the list once. Note that functions like length and member? go through the list, so using these in a recursive function violates this restriction.

Hint: Use accumulative recursion.

(b) Write a function gifts-received which consumes a child’s name, that child’s total niceness score, a ChildrenList, and a negative integer representing Santa’s naughtiness threshold. The function produces either a symbol or a list of strings according to the following rules:

- If a child’s name is not in the ChildrenList, produce empty regardless of their total niceness score.
- If a child has a high enough total niceness score to qualify for any presents in the ChildrenList, The child receives all the presents with a niceness score that is less than or equal to the child’s total niceness score. These presents must appear in the produced list in non-decreasing order based on their score.
- If the child has a non-negative total niceness score, the child receives get a “Playing Card Deck” so they can play Dou Dizhu with their friends. This will always appear at the beginning of the produced list of gifts.
- If a child has been naughty and has a negative total niceness score, but the total niceness score is greater than or equal to Santa’s naughtiness threshold, they receive a lump of coal, indicated by 'coal.
- If a child has been very naughty and has a negative total niceness score below Santa’s naughtiness threshold, the child receives a visit from the Krampus, indicated by 'krampus.

Here are some examples:

(check-expect (gifts-received "Byron" 20 instr-childrenlist -10)
(list "Playing Card Deck"
  "Colourful Pencils"
  "Hot Wheels").
(check-expect (gifts-received "Armin" 10 instr-childrenlist -11)
  (list "Playing Card Deck"))
(check-expect (gifts-received "Patrick" -10 instr-childrenlist -12)
  'coal)
(check-expect (gifts-received "Zahra" -15 instr-childrenlist -13)
  'krampus)

**Restriction:** Do not use `reverse` (or make your own version of it) or sort the produced list. Instead, think about your alternatives for solving this problem. At least one of them is (relatively) short and sweet.

(c) Santa is finding the number of Actions to be overwhelming. He would like to store them in a Binary Search Tree indexed by child name.

The Binary Search Tree is made of **ActionNodes**, defined as follows:

```
(define-struct actionnode (name score actions left right))
;; An ActionNode is a (make-actionnode Str Int (listof Action)
;; ActionSearchTree
;; ActionSearchTree)
;; requires: name > every name in left ActionNode
;; name < every name in right ActionNode
```

The **score** is the *total niceness score* of the child, calculated by adding the niceness scores of all **Action** events for that child in the **ActionNode**. Those **Action** events are stored in the **actions** field, as an unordered list. New **Action** events are added to the front of the **actions** list.

An **ActionSearchTree** (ActionST) is defined as follows:

```
;; An ActionSearchTree (ActionST) is one of:
;; * empty
;; * an ActionNode
```

Write a function `add-action` which consumes an **Action** and an **ActionSearchTree**. It produces an **ActionST** consisting of the original tree with the new **Action** added, possibly creating an **ActionNode** or updating the **ActionNode**’s score.

(d) Santa now needs to generate a list of children and presents. Write a function `gift-list` which consumes an **ActionSearchTree**, a **ChildrenList**, and a negative integer representing his naughtiness threshold, and produces a list of pairs. The first element of each pair is the child’s name, and the second is the gift(s) the child receives (which might be a list of strings representing the received gift(s), or one of the symbols 'coal or 'krampus). The result should be in increasing order of child name, with all names in the **ActionSearchTree** represented. (If a name is in the **ChildrenList** but not in the **ActionSearchTree** it is ignored.)

There is an example in q4provided.rkt.
You may use the built-in function `append` in your solution.

**Hint:** How can you recursively visit every node in an ActionST? How can you do so in increasing order?

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](http://mathworld.wolfram.com/CellularAutomaton.html), the origin of the illustration above.