built on 2016/10/27 at 13:16:55 ***due:*** *thu nov 3 @ 11:59pm*

This is a programming assignment. We're providing stubs for you; please **download starter code from the course website.** When you're done, make a single zip file and upload it to Canvas.

IMPORTANT: Your implementation must strictly follow the functional style. As a rule of thumb, you cannot use features other than what we have done in class so far. In particular, this means, no loops, no mutable variables (cannot use `var`).

- You can define as many helper functions as necessary. Be mindful of what you should expose to outside your function.
- You are going to be graded on style as well as on correctness.
- Test your code!

Task 1: Using Higher-Order Library Functions (6 points)

For this task, save your code in `UseLib.scala`

You will implement the following functions:

(1) Write a function `def onlyBeginsWithLower(xs: List[String]): List[String]` that takes a string list and returns a string list that has only the strings from the input that start with a lower-case letter. Assume all strings have at least 1 character. You should use the `filter` method.

(2) Write a function `def longestString(xs: List[String]): Option[String]` that takes a string list and returns the longest string in the list. If the list is empty, return None. Use `maxBy`.

(3) Write a function `def longestLowercase(xs: List[String]): Option[String]` that takes a string list and returns the longest string in the list that begins with a lower-case letter. If there are no such strings, return None. Assume all strings have at least 1 character

Task 2: Currying (6 points)

For this task, save your code in `Currying.scala`

Implement the following:

(1) Write a function `firstAnswer` of type

```
(A => Option[B]) => List[A] => B
```

(notice the arguments are curried). The first argument should be applied to elements of the second argument in order until the first time it returns `Some(v)` for some `v` and then `v` is the result of the call to `firstAnswer`. If the first argument returns `None` for all list elements, then `firstAnswer` should raise the exception `NoAnswer`.

(2) Write a function `allAnswers` of type

```
(A => Option[List[B]]) => List[A] => Option[List[B]]
```

(notice the arguments are curried). The first argument should be applied to elements of the second argument.

- If it returns `None` for any element, then the result for `allAnswers` is `None`.
- Else, the calls to the first argument will have produced

  ```
  [Some(lst1), Some(lst2), ..., Some(lstn)]
  ```

  and the result of `allAnswers` is `Some(lst)`, where `lst` is `lst1, lst2, ..., lstn` concatenated together (in that order).

Task 3: Knight's Cycle (8 points)

For this task, save your code in `Knight.scala`

The Knight piece in the game of Chess can move, in one step, from (x, y) to the following coordinates $(x \pm 1, y \pm 2)$ and $(x \pm 2, y \pm 1)$, as long as it is still on the board (see Wikipedia for more details).

A curious property is that for some n-by-n board, it is possible for the knight to start at $(1, 1)$ and visit each and every square of the board exactly once before returning to $(1, 1)$.

In this problem, you will write two functions:

- `def findAllCycles(n: Int): List[List[(Int,Int)]]` that takes in the board size $n \leq 10$ and produces a list of lists of int pairs. Each inner list represents a cycle starting with $(1, 1)$ and ending with $(1, 1)$. The outer list will be empty if there isn't a cycle for this board size.
- `def findOneCycle(n: Int): Option[List[(Int,Int)]]` that takes in the board size $n \leq 10$ and produces one list of coordinates representing the cycle. It will return None if there isn't a cycle for this board size.

Extra-Credit: Optimize your code so that `findOneCycle` is fast(er) than `findAllCycles`—and that it is fast enough for n up to 14.