built on 2017/09/26 at 23:52:46 ***due:*** *thu oct 5 @ 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: Read Aloud, Upgraded (4 points)

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

When we read aloud the list [1,1,1,1,4,4,4] , we most likely say four 1s and three 4s, instead of uttering each number one by one. This simple observation inspires the function you are about to implement. First, you're to write a function `def readAloud(lst: List[Int]): List[Int]` that takes as input a list of integers (positive and negative) and returns a list of integers constructed using the following "read-aloud" method:

> Consider the first number, say m. See how many times this number is repeated consecutively. If it is repeated k times, this gives rise to two entries in the output list: first the number k, then the number m. (This is similar to how we say "four 2s" when we see [2,2,2,2] .) Then we move on to the next number after this run of m. Repeat the process until every number in the list is considered.

The process is perhaps best understood with a few examples:

- readAloud([]) should return []
- readAloud([1,1,1]) should return [3,1]
- readAloud([−1,2,7]) should return [1,−1,1,2,1,7]
- readAloud([3,3,8,−10,−10,−10]) should return [2,3,1,8,3,−10]
- readAloud([3,3,1,1,3,1,1]) should return [2,3,2,1,1,3,2,1]

You then notice that you can reconstruct the original list from a "read-aloud" list. For the next part of this problem, you'll implement a function `def unreadAloud(rlst: List[Int]): List[Int]` that takes as input a read-aloud list and returns the original list.

Task 2: Turn It Around (4 points)

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

Using the only collection datatype we know so far (the `List`), we can represent a 2-dimensional array as a list of lists, i.e. `List[List[T]]`. In this problem, you will compute what is known as the "transpose" of a given array. If `A[][]` is an m-by-n array, the transpose of `A` is the an array `B[][]`, which has dimension n-by-m, where `B[i][j] = A[j][i]`.

You will implement a function `def transpose(A: List[List[Int]]): List[List[Int]]` that takes in a 2-dimensional array `A` and returns the transpose of `A`.

Your function must not be excessively slow. That is, transposing a 1000-by-1000 array should take less than a second. Do not use the built-in transpose.

Task 3: Filter & Map (4 points)

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

Scala has built-in `map` and `filter` functions that operate on lists. In this problem, you're writing your own version of these two functions. The function signatures of your functions are:

```
def map[B](f: (A) => B, xs: List[A]): List[B]
def filter(p: (A) => Boolean, xs: List[A]): List[A]
```

Map. The `map` function takes a function `f: (A) => B` and a list `xs`, and returns a new list which applies `f` on each element of `xs`. Notice that `f` is a function that expects input of type `A` and returns an output of type `B`. Hence, even though `xs` has type `List[A]`, by applying `f` on every element of `xs`, we obtain a list of type `List[B]` in return.

To understand what your `map` should do, play around with the implementation that ships with Scala. For example, try the following code:

```
val xs = List(3, 2, 5, 7, 1, 9) // xs: List[Int]
val f = (x: Int) => "z" + x.toString // f: Int => String
val ys = xs.map(f)    // ys: List[String]
```

Filter. The `filter` function takes a function `p: (A) => Boolean` and a list `xs: List[A]`. Notice that `p` takes input of type `A` and returns a Boolean. The `filter` function applies `p` to each element of `xs` and generates a new list retaining only those for which `p` returned `True`, preserving the list order.

To understand what your `filter` should do, play around with the implementation that ships with Scala. For example, try the following code:

```
val xs = List(3, 2, 5, 7, 1, 9) // xs: List[Int]
val f = (x: Int) => x%2 == 1 // returns True if x is odd
val ys = xs.filter(f)    // ys: List[Int]
```

Task 4: Friendly Options (6 points)

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

In this task, we're going to explore a few interesting patterns involving `Option`. You will practice these techniques by implementing the following functions:

1. Your input list is made up of ordered pairs `(String, String)`, where the first component represents the key and the second component is the value corresponding to that key. You are to implement a function

   ```
   def lookup(xs: List[(String, String)], key: String): Option[String]
   ```

   that takes as input a list of key-value pairs, as specified above, and a key—and returns the following: If the key is present in this list, returns `Some(v)`, where `v` is the value corresponding to the first key (from left) found. Otherwise, it returns `None`.

   As an example, consider the list `xs = List(("a", "xy"), ("c", "pq"), ("a", "je"))`. Calling `lookup(xs, "a")` should return `Some("xy")`. But calling `lookup(xs, "b")` should return `None`.
2. Consider the following process, which is pretty typical in bussiness applications: You are given as input a `loginName` and you're to find the average score for the division to which the person with this `loginName` belongs.
   a) First, you have the `loginName`
   b) You're going to translate that into a `userId` using a function called `userIdFromLoginName`

c) You're then going to translate that `userId` into which major s/he is using a function called `majorFromUserId`.
d) You're then going to translate the major into which division that major is in by using a function called `divisionFromMajor`.
e) Finally, derive the score for that division using the function `averageScoreFromDivision`. The result of this is what you will return.

You're going to implement this logic in a function that has the following signature:

```
def resolve(userIdFromLoginName: String => Option[String],
            majorFromUserId: String => Option[String],
            divisionFromMajor: String => Option[String],
            averageScoreFromDivision: String => Option[Double],
            loginName: String): Double
```

Notice that each of the functions that you need to derive the final answer returns an `Option`, allowing for the possibility of it failing. If any of it fails, your function will return `0.0`.

You can implement this logic using a series of `if-else`. But this will be painful! Instead, learn about `map` and `flatMap`. These two methods operate on an `Option` type.

Hint: Once you master this technique, your answer to this question should be at most 5 lines long.

Task 5: Miscellaneous Date Routines (8 points)

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

We often have to work with date and time. This problem will get you a front-row seat working with date functions. You should know the following before you start:

- In all functions below, a *date* is a 3-tuple `(Int, Int, Int)`, where the first part is the day, the second part is the month, and the third part is the year (in Christian era). For your convenience, we have defined a type alias `Date` for you in the stub file.
- A date is *reasonable* if it has a positive year, a month between 1 and 12, and a day obviously no greater than 31 and also in the range for that specific month. Your solutions need to work correctly on reasonable dates.
- A *day of year* is a number from 1 to 365 (or 366 for a leap year) where, for example, 34 represents February 3. Leap years should be handled properly.

You will implement the following functions and write tests for them (not to be handed in):

(1) a function `def isOlder(x: Date, y: Date): Boolean` that takes two dates and evaluates to true or false. It evaluates to true if the first argument is a date that comes before the second argument. (If the two dates are the same, the result is false.)

(2) a function `def numberInMonth(xs: List[Date], month: Int): Int` that takes a list of dates and a month (i.e., an int) and returns how many dates in the list are in the given month.

(3) a function `def numberInMonths(xs: List[Date], months: List[Int]): Int` that takes a list of dates and a list of months and returns the number of dates in the list of dates that are in any of the months in the list of months. Assume the list of months has no number repeated. (*Hint:* Don't repeat yourself.)

(4) a function `def datesInMonth(xs: List[Date], month: Int): List[Date]` that takes a list of dates and a month and returns a list holding the dates from the argument list of dates that are in the month. The returned list should contain dates in the order they were originally given.

(5) a function `def datesInMonths(xs: List[Date], months: List[Int]): List[Date]` that takes a list of dates and a list of months and returns a list holding the dates from the argument list of dates that are in any of the months in the list of months. You should assume that the list of months has no number repeated. The returned list should contain dates in the order they were originally given.

(6) a function `def dateToString(d: Date): String` that takes a date and returns a string of the form August-10-2017. Use the operator + for concatenating strings.
To produce the month part, do not use a bunch of conditionals. Instead, use a list holding 12 strings and pick out the right element. For consistency, use hyphens exactly as in the example and use English month names: January, February, March, April, May, June, July, August, September, October, November, December.

(7) a function `def whatMonth(n: Int, yr: Int): Int` that takes a day of year (i.e., an int between 1 and 365 or 366) and a year, and returns what month that day is in (1 for January, 2 for February, etc.).

(8) a function `def oldest(dates: List[Date]): Option[Date]` that takes a list of dates and evaluates to a `Date` option. It evaluates to `None` if the list has no dates else `Some(d)` where the date `d` is the oldest date in the list.

(9) a function `def isReasonableDate(d: Date): Boolean` that takes a date and determines if it describes a real date in the common era. A *real date* has a positive year (year 0 did not exist), a month between 1 and 12, and a day appropriate for the month. You should properly handle leap years. Leap years are either divisible by 400 or divisible by 4 but not divisible by 100.

Task 6: Zombies, Revisited (4 points)

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

We want you to implement the `merge` routine for merge sort. But to spice things up a little, we'll bring back your fond memories from Data Structures.

In a remote village known as Salaya, zombies and humans have lived happily together for many decades. In fact, no one can quite tell zombies and humans apart. However, when these "people" line up in a single row, all sorts of trouble ensue, including this weird phenomenon: human beings will line themselves up from tall to short, but zombines act erratically.

In particular, if `line` is an array of heights of the population of this village, we would expect that `line`$[i] \geq$ `line`$[j]$ for $i \leq j$. But this simply isn't true in many cases especially with zombies around. Hence, one nobleman—or is he a zombie?—came to you for help: he wants to know how many pairs of his people violate this social norm.

Your Task: Write a function `def countBad(hs: List[Int]): Int` that takes a list of n numbers and returns the number of pairs $0 \leq i < j < n$ such that `hs`$[i] <$ `hs`$[j]$ (i.e., the number of pairs that violate the social norm).

For example:

- countBad([35, 22, 10]) == 0
- countBad([3,1,4,2]) == 3
- countBad([5,4,11,7]) == 4
- countBad([1, 7, 22, 13, 25, 4, 10, 34, 16, 28, 19, 31]) == 49

Performance Expectations: We expect your code to run in at most $O(n \log n)$ time, where n is the length of the input array.

Here are some tips to get started:

- Eventually you'll want to implement a merge-sort-like algorithm. To begin, ask yourself how can you split a front-access list in half (by size).
- How do you merge? Remember in a cons-list (a front-access list), the only thing you can access cheaply is the head.