Homework 2

Functional and Parallel Programming (Term III/2020)

built on 2021/05/30 at 05:42:55 **due:** *Monday June 14th @ 11:59pm*

This assignment contains only a coding portion. The aim of this assignment is to gain more experience programming in the functional style. We're providing stubs for you; please *download the starter code from the course website.*

You will zip all your work in one zip file and upload it to Canvas before the due date.

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!

Read This (0 points)

To familiarize yourself with Scala collections, read the following:

`https://docs.scala-lang.org/overviews/collections/introduction.html`

The document covers many useful data types, including sets, maps, arrays, and vectors. It discusses both mutable and immutable collections. For this assignment (and the next couple of weeks), we will use only *immutable* collections—no mutable collections!

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.

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]
```

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.

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.

Thesaurus & Graph Search (15 points)

You will implement a generic breadth-first search (BFS) and use it to solve a few related problems. This task contains 3 subtasks.

Part I: Breadth-First Search (BFS)

Remember breadth-first search from your data structures & algorithms course? It is probably a good idea to review your notes from years ago. For a quick recap: You're starting at some vertex, known as the *source vertex*, often denoted by *s*. Your goal is to find the shortest-path paths from this source to all vertices of the graph. The algorithm explores the graph level by level. At every point, you maintain what is known as the *frontier*. To go from one level to the next, you expand the frontier by putting into the new frontier the neighbors of all the nodes in the current frontier that have not been explored before. In pseudo-code:

```
def bfs(nbrs: V => Set[V], src: V) =
    distFromSrc = 0
    frontier = { src } // set of vertices on the frontier
    visited = { src } // set of already-visited vertices

    while (frontier.nonEmpty) {
       frontier_ = expand(frontier, visited, nbrs)
       visited_ = visited + frontier
       frontier, visited_, distFromSrc = frontier_, visited_, distFromSrc + 1
    }
```

where the function `expand` generates the next frontier by

1. Expanding each vertex u in the frontier as its neighbors `nbr(u)`;
2. Collecting the expanded vertices and removing those that have already been visited.

A byproduct of this process is a shortest-path tree, which stores how the vertices of the graph can be reached from the source. This is usually kept as a map, where for every vertex u, the map indicates which vertex is used immediately before reaching u on the shortest path from `src` to u.

In this subtask, you'll save your work in `GraphBFS.scala`. You are to write a function

```
def bfs[V](nbrs: V => Set[V], src: V): (Map[V, V], Map[V, Int])
```

which takes as input a neighbor function and a source vertex, and returns a pair `(parent, distance)`, where

- the neighbor function `nbrs` is a function that takes a vertex and yields a set of neighboring vertices of this vertex.

- `parent` is a map from vertex to vertex, mapping each vertex u to the vertex used immediately before reaching u on the shortest path from `src` to u. Some exceptions: if a vertex w is not reachable from `src`, w will not be present in the map. Moreover, the `src` vertex maps to itself.
- `distance` is a map from a vertex to an integer representing the distance from `src`. Distance is measured in terms of the number of edges used. This means, `src` is at distance 0 away from `src` itself.

You may find the following structure useful:

```
def bfs[V](nbrs: V => Set[V], src: V) = {

  def expand(frontier: Set[V], parent: Map[V, V]): (Set[V], Map[V, V]) =
       // derive new frontier and new parent map

  def iterate(frontier: Set[V], parent: Map[V, V], distance: Map[V, Int], d: Int) =
    if (frontier.isEmpty)
      (parent, distance)
    else {
      val (frontier_, parent_) = expand(frontier, parent)
      val distance_ = // derive new distance map

      iterate(frontier_, parent_, distance_, d + 1)
    }

  iterate(Set(src), Map(src -> src), Map(), 0)
}
```

Part II: Thesaurus

You'll now put your implementation to use. You will use it to solve the thesaurus problem: given a pair of words A and B, find the shortest chain of synonyms that connect the pair of words. In this subtask, you will implement 2 components:

- a thesaurus database parser; and
- an adaptor that interfaces with the BFS code above.

For this part, your work will go into `Thesaurus.scala`.

Thesaurus Parser:

The thesaurus database parser will read a file residing on disk and create an internal representation of the thesaurus database that allows for efficient lookup of synonyms. The database format is as follows:

Database Format: Included in the starter package is a thesaurus database extracted from Priceton's WordNet-2.0 data by the OpenOffice team. The first few lines of the file look as follows:

```
ISO8859-1
'hood|1
(noun)|vicinity|locality|neighborhood|neighbourhood|neck of the woods
.
. (lines omitted)
.
abbey|3
(noun)|church|church building
(noun)|convent
(noun)|monastery
.
.
.
```

The first line shows the encoding used, in this case, ISO8859-1, which is the encoding you should use when reading the file. See the documentation for `scala.io.Source` for more information. Subsequent lines are thesaurus entries, which are presented in the following format.

- Each group begins with a stem word, followed by `|`, and the number of synonym groups to follow. For example, `abbey|3` means the stem word is `abbey` and there are 3 groups of synonyms related to this stem, which are listed below.
- When there are *n* synonym groups, the next *n* lines after the stem show words belonging to these groups. The fields are `|` delimited, with the first entry of each line indicating the part of speech.
- Sometimes there are trailing whitespaces before and after a word, potentially an error in the database. Your code should *remove these trailing whitespaces*.

For ease, we'll combine all synonym groups for the same stem into one. Hence, in our synonym graph, we'll have, for example, an edge between `abbey` and each of the following words: `church`, `church building`, `convent`, `monastery`.

You will implement a parser by writing a function `def load(filename: String)` that parses the database completely and returns a representation of the database in a form of your choosing.

Gluing Them Together:

To put everything together, you'll implement a function with the following type signature:

```
def linkage(thesaurusFile: String): String => String => Option[List[String]]
```

That is, this function takes the name/path of a thesaurus database file (e.g., `/home/kanat/foo.txt`). It returns a function that accepts `wordA: String`—and produces another function that accepts `wordB: String` and returns a value of type `Option[List[String]]`.

As the type suggests, when called using `linkage(thFile)(wordA)(wordB)`, it will produce `Some` of a list of synonym words that link `wordA` to `wordB`. The return will be `None` if no such linkages exist. As an example, one possible answer for `linkage(thFile)("clear")("vague")`, on the example thesaurus database, is `Some(List(clear, light, faint, vague))`, indicating that "light" is listed as a synonym of "clear"; "faint", a synonym of "light"; and "vague", a synonym of "faint". And this is the shortest possible chain from "clear" to "vague."

For performance, your function must be *staged* so as to satisfy the following behaviors:

```
val findLinks = linkage(someThesaurusFile)
// the above should parse the given database and store it in the closure of findLinks
val f = findLinks("clear")
// this should perform the search from the word clear; it is this step that
// you call bfs.
f("vague") // compute a path from 'clear' to 'vague', based on the bfs outcome.
```

A correct implementation should return a sequence of length 4 from "clear" to "vague"—and a sequence of length 6 from "logical" to "illogical". And for sure, there's really no good reason to try "earthly" to "poison".

Part III: Maze

As another application of this, consider the problem of finding the shortest route out of a maze. A *maze* is a puzzle consisting of a system of passages and barriers, and the objective is to find a (shortest) path from a starting point (source) to an exit (destination). Here, a maze is given as a list of strings that "graphically" represents the maze. For instance, a maze 6-unit tall and 18-unit wide is shown on below, with the starting point marked with an `s` and the destination marked with an `e`. The letter '`x`' denotes a barrier, and an empty space is a passable area. On the right is a graphical rendition of this maze with a shortest path (there are many shortest paths) drawn in dotted lines.

```
maze = [
  "xxxxxxxxxxxxxxxxxx",
  "x    x      x   ex",
  "x    x   x  x xxxx",
  "x        x  x    x",
  "xs   x   x       x",
  "xxxxxxxxxxxxxxxxxx"
]
```

We will develop a program so that given such a maze as input, it will return a sequence of moves to go from the starting point to the destination. This will be in the form of a string listing the moves, using "`u`' for up, 'd' for down, '`l`" for left, and '`r`' for right. Hence, as an example, the string '`rrrrurr`' denotes the actions of moving right 3 times, then up, then moving right 2 more times. (Think of an old game where the resolution is low and you're pressing the arrow keys to navigate.)

Your task: In `Maze.scala`, implement a function

```
def solveMaze(maze: List[String]): Option[String]
```

that takes in a maze encoded as described above and returns `Some` of a string representing a sequence of moves as already detailed. It will return `None` if there is no way to exit the maze.

You should use your BFS implementation from the previous part.

Important: For a 250x250 maze, your code should finish within 2 seconds.

Compilation: This is a good excuse to start learning a build tool for Scala. Most Scala-native projects use a build tool called `sbt`, which you'll read more about online. Assuming you have some familiarity with a build tool (e.g., make, maven, and gradle), sbt shouldn't be much of a surprise.

To get you started, we're including a Scala "makefile" called `build.sbt`. Run `sbt compile` to compile your project. Use `sbt console` to start Scala REPL in the environment of this project.

IntelliJ has pretty good sbt integration. Start by importing existing source files by pointing IntelliJ to the `build.sbt` file.