built on 2017/10/10 at 23:14:03 ***due:*** *sat oct 21 @ 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 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!

Task 2: Thesaurus & Graph Search (20 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.

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:

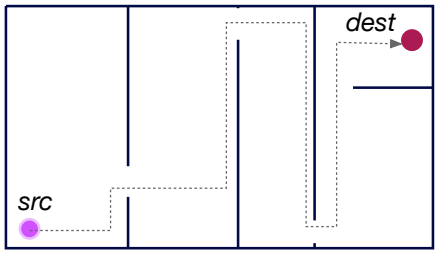
```
val findLinks = linkage(someThesaurusFile)
// the above should parse the given database and store it in the closure of findLinks
val f = findLinks("clear")
// this should perform the search from the word clear; it is this step that
// you call bfs.
f("vague") // compute a path from 'clear' to 'vague',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

Task 3: Basic Equation Solver (20 points)

You're building a numerical equation solver. The interface is primitive, providing

```
def solve(expString: String, varName: String, guess: Double): Double
```

where, for example, `solve("x^2 - 4.0", "x", 1.0)` will solve for x in $x^2-4=0$ with a starting guess of $x=1.0$, and will eventually return `2.0` (or `-2.0` depending on which answer your Newton's method likes more). For full credit, your submission will provide a functioning solver in `solver.Main.solve` with the above signature that implements the Newton's method. Additionally, it will provide functions `differentiate` and `eval`, inside `solver.Process`, with the signature

```
def differentiate(e: Expression, varName: String): Expression
def eval(e: Expression, varAssn: Map[String, Double]): Double
```

You are to ***assume that the input expression always has at least a root.***

Parser: To help you get started, we're supplying a parser to turn a string expression into a recursive data type `Expression` for convenient manipulation and consumption. Details can be found inside `Parser.scala`. For this assignment, however, just about the only thing you need to know is that if there's a string `s` that looks like an expression (`s = "2^x + x + 4*3"`), you can turn it into an Expression value by calling `solver.Parser(s)` or simply `Parser(s)` if you're inside the solver package or have imported it. The following is an example code listing:

```
// assuming we're inside the solver package
val s = "2^x + x + 4*3"
val e: Expression = Parser(s)
```

Supporting Utility: There's a simple routine for pretty-printing, practically converting an Expression value into a string representation for debugging, etc. This lives inside `Process.scala`.

Also housed in the file are functions to be implemented (you're welcome to ditch this skeleton as long as both `differentiate` and `eval` are implemented and match the required type signature):

- `def eval(e: Expression, varAssn: Map[String, Double]): Double` evaluates a given expression `e: Expression` using the variable settings in `varAssn: Map[String, Double]`, returning the evaluation result as a `Double`.
  Example: `eval(e, Map("x"-> 4.0))` evaluates the expression with the variable x set to 4.0.
- `def differentiate(e: Expression, varName: String): Expression` symbolically differentiates an expression `e: Expression` with respect to variable `varName: String`, returning an `Expression` value.
  (*Hint:* The `Expression` type is a sum type. Ponder what it can be.)
- `def simplify(e: Expression): Expression` forms a new expression that simplifies the given expression `e: Expression`. The goal of this function is to produce an expression that is easier to evaluate and/or differentiate. If there's a canonical form you'd like to follow, use this function to put an expression in that form. The idea is to use `simplify` on the differentiated expression so that it is a nicer expression.

Symbolic Differentiation: Your `differentiate` function attempts to compute the (partial) derivative of `e` with respect to a given variable name `varName`. For example, if `varName` is x, then `differentiate` returns an `Expression` that is equivalent to $\frac{de}{dx}$. Do not worry: You really don't have to remember much calculus. Here are some formulas for computing derivatives that you will use:

$$\frac{d}{dx}\text{constant} = 0$$

$$\frac{d}{dx}\left(f(x)+g(x)\right) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$

$$\frac{d}{dx}\left(f(x)\times g(x)\right) = \frac{df(x)}{dx}\cdot g(x) + f(x)\cdot\frac{dg(x)}{dx}$$

$$\frac{d}{dx}\left([f(x)]^h\right) = h[f(x)]^{h-1}\frac{df(x)}{dx} \qquad \text{where } h \text{ contains no variable.}$$

$$\frac{d}{dx}c^{f(x)} = [\ln(c)]\cdot c^{f(x)} \qquad \text{where } c \text{ is a nonzero constant.}$$

You don't need to support expressions beyond what can be differentiated using these formulas. Keep in mind that some expressions have to be "massaged" before they fit one of these forms.

Extra-Credit: Parsing Your Own Expression

After you're done with the functions above, your next task to write your own parser! Yes, that function that takes, e.g., `x^3.5 + 2*(5+x)` and returns a nice `Expression`, similar to what `solver.Parser` does. More precisely, you'll write the function

```
def parse(input: String): Option[Expression]
```

which lives inside `MyParser.scala`.

Inside the same file, there is a tokenizer function that you may find helpful.

Notes

You should have an implementation of the Newton's method already. You can adapt your own code from lecture.

Compliation: 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.