built on 2016/10/06 at 17:25:05 ***due:*** *thu oct 20 @ 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: CSV Parser (10 points)

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

You will write a CSV parser in this problem. The CSV format is short for Comma-separated values. It is roughly speaking the idea that in every line that represents your data, there is a comma separating adjacent fields. As it turns out, things are as simple. What happens if the data contains a comma inside it? The full specification of the CSV format can be found in this document (`https://www.ietf.org/rfc/rfc4180.txt`)—don't worry, it's one of the shortest RFCs.

Suppose a data file has been read into a string. Your function will take this string and return a list of lists. Each entry in the outer list corresponds to a line; therefore, each inner list contains the different fields on that line. The signature is as follows:

```
def parseCSV(s: String): List[List[String]]
```

To help you develop/debug the code, you should know that

```
val s = scala.io.Source.fromFile(fileName, "US-ASCII").mkString
```

will give you a string `s` containing all the bytes (ASCII encoded) of the file called `fileName`.

Task 2: Basic Equation Solver (10 points)

You're building a numerical equation solver for this assignment. 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 the 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 the above functions are provided):

- `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.

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. For the assignment, you can adapt your own code from lecture.

Compliation: This is a good excuse to start compiling things with `scalac *.scala`. If you are up for it, we're including a Scala "makefile" called `build.sbt` so that we can simply run `sbt compile`. Using `sbt` will require a bit of self-studying (to understand the special nature of it).