Lab Assignment 1

Note: README.pdf is built from this file using the included file generate; if you are having trouble making sense of any LaTeX formatted mathematical notation, take a look at the pdf.

The purpose of this assignment is to warm-up with Scala and to refresh preliminaries from prior courses.

Find a partner. You will work on this assignment in pairs. Feel free to use Piazza to locate a partner.

You are welcome to talk about these questions in larger groups. However, we ask that you write up and code your answers with your lab team. Please acknowledge in your writeup anyone you spoke with about the lab, including your partner(s) and those outside your lab group. Use the functionality of GitHub, especially branching and code review, to demonstrate your collaboration.

You must work in at groups of at least 2 (preferable) but not greater than 3. **Absolutely no work will be accepted from teams consisting of only one student.**

What you need to do

You need to work in the group repo we have assigned to you, by using a branch development process with your group to converge on a solution in your group repo. At grading time, your pull request from this group repo will be considered complete if it contains:

1. A detailed work-in-progress pull request. Use a task list so you can check off items over time as you push up to your repo.
2. Your code submission implemented within *Lab1.scala* — simply modify the given template file in `/src/main/scala/*` — along with adding tests in `src/test/scala/*`. Look for `UnsupportedOperationException` and of course failing tests.
3. Periodically push back to your group repository. This should cause cause an autograder to run via a post-commit hook. Review the status message for each commit to see where you stand. Make sure to remove or comment out any print statements you may have inserted for debug purposes, as this could cause the test cases to fail.
4. Write your answers to the written part of *README.pdf* in the file *Lab1.md*. Answers should be written in Markdown and use code block formatting for displaying any code; indent with four spaces or surround with back ticks to do this.

5. Fill out answers to the questions in a Google Form survey.

Task #1 must be completed by Friday 7p October 4. The remaining tasks must be completed by Friday 7p October 11.

Late submission or reassessment of up to 85% is accepted if Task #1 is completed by Friday 7p October 11 and the remaining tasks no later than Friday October 25. Otherwise, no credit will be given for this lab.

You can look at this README file itself to get more ideas on Markdown usage. You may also want to refer to the Pandoc Markdown dialect documentation. In particular, any answers using graphs, drawings, etc. can be saved as an image file in this directory of your group repo and refered to in *Lab1.md*.

Working with Scala

You may find it helpful to develop and test your code using the scala interpreter. First, compile your code using the sbt command below, or using the scala compiler directly

```
$ scalac Lab1.scala
```

You can start the scala interpreter using the command

```
$ scala
```

and can import the functions in your lab in the following way

```
scala> import Lab1._
```

Note that you will need to run the scala interpreter in the same directory as the compiled version (/bin).

`sbt`

In this class, we use `sbt`. We have included an `sbt` build script, `build.sbt`. You can issue the following commands to compile, run and test your code:

```
$ sbt compile
```

compiles your program

```
$ sbt clean
```

deletes the previous compilation

```
$ sbt test
```

executes the test suite. Note that the test suite is designed to test the function of individual terms, and not necessarily the interaction between. The online autotester will contain more complex test cases.

Note that you will need to run these commands from the Lab1 folder.

In some cases, we have asked you to also write the tests, as indicated by tests with `throw new UnsupportedOperationException`.

GitHub repository

The completed lab must be submitted using your GitHub team repository. Issue a pull request against Lab1. Pull requests will be automatically tested as pushed and will be evaluated by your TAs at the due date. Any pull requests, or pushes onto your pull request after this time, will be considered late.

Evaluation criteria

Both your ideas and also the clarity with which they are expressed matter - in your English prose as well as your code!

We will consider the following criteria in our grading:

- *How well does your submission answer the questions?* For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation.
- *How clear is your submission?* If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that "works" deserves full credit. We must be able to read and understand your intent. Make sure you state any preconditions or invariants for your functions (either in comments, as assertions, as additional test cases, or as `require` clauses as appropriate).

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expression computation currently unfamilar to you. Line counts for each function are given for an instructor solution (that uses good

style and does not try optimize for length). While line counts are not an ideal measure of crisp, concise code, the figures give you a sense of what can be done. Can you beat the numbers while maintaining good style? If so, we might just accept your pull request, merge in your code as a better solution, and give you extra credit!

Finally, make sure that your file compiles and runs (using Scala 2.10.2). A program that does not compile will *not* be graded. This is really easy to do because you're using `sbt`, right?

Parting words

In the words of a TA who previously taught this class: strive for cake!

Problems for Lab 2

Scala Basics: Binding and Scope

For each the following uses of names, give the line where that name is bound. Briefly explain your reasoning (in no more than 1-2 sentences).

```
1  val pi = 3.14
2  def circumference(r: Double): Double = {
3    val pi = 3.14159
4    2.0 * pi * r
5  }
6  def area(r: Double): Double =
7    pi * r * r
```

- The use of `pi` at line #4 is bound at which line?
- The use of `pi` at line #7 is bound at which line?

Consider the following Scala code

```
1  val x = 3
2  def f(x: Int): Int =
3    x match {
4      case 0 => 0
5      case x =>
6        val y = x + 1
7        ({
8           val x = y + 1
```

```
9             y
10          } * f(x - 1))
11        }
12      }
13    val y = x + f(x)
```

Questions:

- The use of `x` at line 3 is bound at which line?
- The use of `x` at line 6 is bound at which line?
- The use of `x` at line 10 is bound at which line?
- The use of `x` at line 13 is bound at which line?

Scala Basics: Types

In the following, I have left off the return type of function `g`. The body of `g` is well-typed if we can come up with a valid return type. Is the body of `g` well-typed?

```
def g(x: Int) = {
  val (a, b) = (1, (x, 3))
  if (x == 0) (b, 1) else (b, a + 2)
}
```

If so, give the return type of `g` and explain how you determined this type. First, give the types for the names `a` and `b`. Then, explain the body expression using the following format:

1) $e : \tau$ because
   a) $e_1 : \tau_1$ because
      i) ...
   b) $e_2 : \tau_2$ because
      ii) ...

where e_1 and e_2 are subexpressions of e. Stop when you reach values (or names).

As an example of the suggested format, consider the `plus` function:

```
def plus(x: Int, y: Int) = x + y
```

In this case, we would state, **Yes**, the body expression of `plus` is well-typed with type `Int`:

1) `x + y:  Int` because

   a) `x:  Int`
   b) `y:  Int`

Some library functions

Most languages come with a standard library with support for such things as data structures, mathematical operators, and string processing. For this question, we will implement some library functions.

For each function f that you are asked to write in this question, write one test case as a `def` named `test`f that returns a `Boolean` (with the first letter capitalized) that is a check between an expected result and the computed result.

One can then write specifications in `FlatSpec` to make sure your implementation passes your unit test.

For example,

```
  def plus(x: Int, y: Int): Int = x + y

  class PlusSpec extends FlatSpec {
    behavior of "plus(x, y)"

    it should "return the sum of inputs x and y" in {
            assert(plus(1,1) === 2)
            assert(plus(3,5) === 8)
            assert(plus(1,0) === 1)
     }

    it should "also work if x and/or y is negative" in {
        assert(plus(-1,1) === 0)
        assert(plus(-5,-3) === -8)
        assert(plus(-2,4) === 2)
    }
}
```

`abs` function

Write a function to complete the following definition:

```
  def abs(n: Double): Double
```

that returns the absolute value of `n`. This is a function that takes a value of type `Double` and returns a value of type `Double`. This function corresponds to the JavaScript library function `Math.abs`.

Instructor Solution: 1 line.

`xor` function

Write a function of signature

```
def xor(a: Boolean, b: Boolean): Boolean
```

that returns the exclusive-or of `a` and `b`. The exclusive-or returns `true` if and only if exactly one of `a` or `b` is `true`. For this exercise you may not use the Boolean operators. Instead, only use the `if-else` expression and the Boolean literals (`true`, `false`).

Instructor Solution: 4 lines (including 1 line for a closing brace).

Recursion

Write a recursive function for `repeat` with the signature

```
def repeat(s: String, n: Int): String
```

where `repeat(s, n)` returns a string with `n` copies of `s` concatenated together. For example, `repeat("a",3)` returns `"aaa"`. This function corresponds to the Google Closure library function `goog.string.repeat`.

Instructor Solution: 4 lines (including 1 line for a closing brace).

Square root

In this exercise, we will implement the square root function — `Math.sqrt` in the JavaScript standard library — using Newton's method (also known as Newton-Raphson).

Recall from calculus that a root of a differentiable function can be iteratively approximated by following tangent lines. More precisely, let f be a differentialable function, and let x_0 be an initial guess for a root of f. Then, Newton's method specifies a sequence of approximations $x_0, x_1, \ldots$ with the following recursive equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

You may want to watch a video as a refresher on this algorithm.

The square root of a real number c for $c > 0$, written $\sqrt{c}$, is a positive x such that $x^2 = c$. Thus, to compute the square root of a number c, we want to find the positive root of the function:

$$f(x) = x^2 - c$$

Thus, the following recursive equation defines a sequence of approximations for $\sqrt{c}$:

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n}$$

`sqrtStep` function

First, implement a function `sqrtStep` of signature

```
def sqrtStep(c: Double, xn: Double): Double
```

that takes one step of approximation in computing $\sqrt{c}$ (i.e., computes x_{n+1} from x_n).

Instructor Solution: 1 line.

`sqrtN` function

Next, implement a function `sqrtN`:

```
def sqrtN(c: Double, x0: Double, n: Int): Double
```

that computes the nth approximation x_n from an initial guess x_0. You will want to call `sqrtStep` implemented in the previous part.

You must implement this function using recursion and no mutable variables (that is `var`) — you will want to use a recursive helper function. In your writeup, compare your recursive solution with one using a `while` loop, showing the code of the latter.

Instructor Solution: 8 lines (including 2 lines for closing braces and 1 line for a `require`.

`sqrtErr` function

Now, implement a function `sqrtErr`:

```
def sqrtErr(c: Double, x0: Double, epsilon: Double): Double
```

This function is very similar to `sqrtN` but instead computes approximations x_n until the approximation error is within ε (`epsilon`), that is, $|x_n^2 - c| < \varepsilon$. You can use your absolute value function `abs` implemented in a previous part. A wrapper function `sqrt` is given in the template that simply calls `sqrtErr` with a choice of `x0` and `epsilon`.

Again, you must implement this function using recursion. Then in your writeup, compare your recursive solution to one written using a `while` loop.

Instructor Solution: 8 lines (including 2 lines for closing braces and 1 line for a `require`.

Data Structures

In this question, we will review implementing operations on binary search trees. Balanced binary search trees are common in standard libraries to implement collections, such as sets or maps. For example, the Google Closure library has `goog.structs.AvlTree`. For simplicity, we will *not* worry about balancing in this question.

Trees are important structures in developing interpreters, so this question is also critical practice in implementing tree manipulations.

A binary search tree is a binary tree that satisfies an ordering invariant. Let n be any node in a binary search tree whose data value is d, left child is l, and right child is r. The ordering invariant is that all of the data values in the subtree rooted at l must be $< d$, and all of the data values in the subtree rooted at r must be $\geq d$.

We will represent a binary search tree using the following Scala classes and objects:

```
sealed abstract class SearchTree
case object Empty extends SearchTree
case class Node(l: SearchTree, d: Int, r: SearchTree)
  extends SearchTree
```

So as we can see from above, a `SearchTree` is either `Empty` or a `Node` with left child `l`, data value `d`, and right child `r`.

For this question, we will implement the following four functions.

The function `repOk`

```
def repOk(t: SearchTree): Boolean
```

checks that an instance of `SearchTree` is a valid binary search tree. In other words, it checks using a traversal of the tree the ordering invariant. This function is useful for testing your implementation. A skeleton of this function has been provided for you in the template.

Instructor Solution: 7 lines (including 2 lines for closing braces).

The function `insert`

```
def insert(t: SearchTree, n: Int): SearchTree}
```

inserts an integer into the binary search tree. Observe that the return type of `insert` is a `SearchTree`. This choice suggests a functional style where we construct and return a new output tree that is the input tree `t` with the additional integer `n` as opposed to destructively updating the input tree. In your writeup, ensure you address why this functional approach is required when using case classes.

Instructor Solution: 4 lines (including 1 line for a closing brace).

The function `deleteMin`

```
def deleteMin(t: SearchTree): (SearchTree, Int)
```

deletes the smallest data element in the search tree (i.e., the leftmost node). It returns both the updated tree and the data value of the deleted node. This function is intended as a helper function for the `delete` function. Most of this function is provided in the starting code we give you.

Instructor Solution: 9 lines (including 2 lines for closing braces).

The function `delete`

```
def delete(t: SearchTree, n: Int): SearchTree
```

removes the first node with data value equal to `n`. This function is trickier than `insert` because what should be done depends on whether the node to be deleted has children or not. Your solution needs to use pattern matching to organize the cases.

Instructor Solution: 10 lines (including 1 line for a closing brace).