CSCI 3155: Lab Assignment 1

Spring 2014: ~~Due Friday, September 4, 2015~~ Saturday, September 5, 2015

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

Form your team of 8-10 persons in your lab section. Then, find a partner from your team for this lab assignment. You will work on this assignment closely with your partner. However, note that **each student needs to submit** and are individually responsible for completing the assignment.

You are welcome to talk about these questions beyond your teams. However, we ask that you code in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus.

> *Both your ideas and also the clarity with which they are expressed matter—both in your English prose and 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 pre-conditions or invariants for your functions (either in comments, as assertions, 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.

Finally, make sure that your file compiles and runs on COG. A program that does not compile will *not* be graded.

Submission Instructions. Upload to the moodle exactly three files named as follows:

- `Lab1-`*YourIdentiKey*`.pdf` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable).
- `Lab1-`*YourIdentiKey*`.scala` with your answers to the coding exercises.
- `Lab1-`*YourIdentiKey*`.jsy` with a challenging test case for your JAVASCRIPTY interpreter.

Replace *YourIdentiKey* with your IdentiKey (e.g., for me, I would submit `Lab1-bec.pdf` and `Lab1-bec.scala`). Don't use your student identification number. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Submit your `Lab1.scala` file to COG for auto-testing. We ask that you submit both to COG and to moodle in case of any issues.

Sign-up for an interview slot for an evaluator. To fairly accommodate everyone, the interview times are strict and **will not be rescheduled**. Missing an interview slot means missing the interview evaluation component of your lab grade. Please take advantage of your interview time to maximize the feedback that you are able receive. Arrive at your interview ready to show your implementation and your written responses. Implementations that do not compile and run will not be evaluated.

Getting Started. Clone the code from the Github repository `https://github.com/bechang/pppl-labs` and follow the instructions in the `README.md`.

A suggested way to get familiar with Scala is to do some small lessons with Scala Koans (`http://www.scalakoans.org/`). For this lab, we suggest that you consider looking at the AboutAsserts, AboutLiteralBooleans, AboutLiteralNumbers, AboutLiteralStrings, AboutMethods, AboutRecursion, AboutTuples, AboutCaseClasses, and AboutPatternMatching koans.

1. **Feedback**.. Complete the survey linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.

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).

   (a) Consider the following Scala code.

   ```
   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?

   (b) 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)
```

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?

3. **Scala Basics: Typing**. 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?

```
1   def g(x: Int) = {
2     val (a, b) = (1, (x, 3))
3     if (x == 0) (b, 1) else (b, a + 2)
4   }
```

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

$e : \tau$ because

$e_1 : \tau_1$ because

…

$e_2 : \tau_2$ because

…

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
```

Yes, the body expression of plus is well-typed with type Int.

`x + y` : `Int` because

`x` : `Int`

`y` : `Int`

4. **Run-Time Library**. Most languages come with a standard library with support for things like data structures, mathematical operators, string processing, etc. Standard library functions may be implemented in the object language (perhaps for portability) or the meta language (perhaps for implementation efficiency).

   For this question, we will implement some library functions in Scala, our meta language, that we can imagine will be part of the run-time for our object language interpreter. In actuality, the main purpose of this exercise is to warm-up with Scala.

   (a) Write a function `abs`

   def `abs(n: Double): Double`

   that returns the absolute value of `n`. This 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.

   (b) Write a function `xor`

   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 practice, do not use the Boolean operators. Instead, only use the **if**-**else** expression and the Boolean literals (i.e., **true** or **false**).

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

5. **Run-Time Library: Recursion**.

   (a) Write a recursive function `repeat`

   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 function `goog.string.repeat` in the Google Closure library.

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

   (b) In this exercise, we will implement the square root function—`Math.sqrt` in the JavaScript standard library. To do so, we will use 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:[1]

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

[1] The following link is a refresher video on this algorithm: `http://www.youtube.com/watch?v=1uN8cBGVpfs`, January 2012

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 $\sqrt{c}$:

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

i. First, implement a function `sqrtStep`

```
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.

ii. 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.
Challenge yourself to implement this function using recursion and no mutable variables (i.e., **var**s)—you will want to use a recursive helper function. It is also quite informative to compare your recursive solution with one using a **while** loop.
Instructor Solution: 7 lines (including 2 lines for closing braces and 1 line for a `require`).

iii. Now, implement a function `sqrtErr`

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

that 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, challenge yourself to implement this function using recursion and compare your recursive solution to one with a **while** loop.
Instructor Solution: 5 lines (including 1 line for a closing brace and 1 line for a `require`).

6. **Data Structures Review: Binary Search Trees**.

In this question, we will review implementing operations on binary search trees from Data Structures. Balanced binary search trees are common in standard libraries to implement

collections, such as sets or maps. For example, the Google Closure library for JavaScript 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 trees containing integer data using the following Scala **case class**es and **case object**s:

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

A `IntTree` 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.

(a) The function `repOk`

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

checks that an instance of `IntTree` is 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).

(b) The function `insert`

```
def insert(t: IntTree, n: Int): IntTree
```

inserts an integer into the binary search tree. Observe that the return type of `insert` is a `IntTree`. 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.

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

(c) The function `deleteMin`

```
def deleteMin(t: IntTree): (IntTree, 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 template.

Instructor Solution: 9 lines (including 2 lines for closing braces and 1 line for a `require`).

(d) The function `delete`

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

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. We advise that you take advantage of pattern matching to organize the cases.

Instructor Solution: 10 lines (including 2 lines for closing braces).

7. **JavaScripty Interpreter: Numbers**.

   JavaScript is a complex language and thus difficult to build an interpreter for it all at once. In this course, we will make some simplifications. We consider subsets of JavaScript and incrementally examine more and more complex subsets during the course of the semester. For clarity, let us call the language that we implement in this course JAVASCRIPTY.

   For the moment, let us define JAVASCRIPTY to be a proper subset of JavaScript. That is, we may choose to omit complex behavior in JavaScript, but we want any programs that we admit in JAVASCRIPTY to behave in the same way as in JavaScript.

   In actuality, there is not one language called JavaScript but a set of closely related languages that may have slightly different semantics. In deciding how a JAVASCRIPTY program should behave, we will consult a reference implementation that we fix to be Google's V8 JavaScript Engine. We will run V8 via Node.js, and thus, we will often need to write little test JavaScript programs and run it through Node.js to see how the test should behave.

   In this lab, we consider an arithmetic sub-language of JavaScript (i.e., an extremely basic calculator). The first thing we have to consider is how to represent a JAVASCRIPTY *program as data* in Scala, that is, we need to be able to represent a program in our object/source language JAVASCRIPTY as data in our meta/implementation language Scala.

   To a JAVASCRIPTY programmer, a JAVASCRIPTY program is a text file—a string of characters. Such a representation is quite cumbersome to work with as a language implementer. Instead, language implementations typically work with trees called *abstract syntax trees* (ASTs). What strings are considered JAVASCRIPTY programs is called the *concrete syntax* of JAVASCRIPTY, while the trees (or *terms*) that are JAVASCRIPTY programs is called the *abstract synatx* of JAVASCRIPTY. The process of converting a program in concrete syntax (i.e., as a string) to a program in abstract syntax (i.e., as a tree) is called *parsing*.

   For this lab, a parser is provided for you that reads in a JAVASCRIPTY program-as-a-string and converts into an abstract syntax tree. We will represent abstract syntax trees in Scala using **case class**es and **case object**s. The correspondence between the concrete syntax and the abstract syntax representation is shown in Figure 1.

   (a)

   Interpreter 1. Implement the `eval` function

   def `eval(e: Expr): Double`

   that evaluates a JAVASCRIPTY expression `e` to the Scala double-precision floating point number corresponding to the *value* of `e`.

   Consider a JAVASCRIPTY program *e*; imagine *e* stands for the concrete syntax or text of the JAVASCRIPTY program. This text is parsed into a JAVASCRIPTY AST `e`, that is, a Scala value of

```
sealed abstract class Expr
case class N(n: Double) extends Expr
  N(n)   n
case class Unary(uop: Uop, e1: Expr) extends Expr
  Unary(uop, e1)   uop e1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
  Binary(bop, e1)   e1 bop e2

sealed abstract class Uop
case object Neg extends Uop
  Neg   -

sealed abstract class Bop
case object Plus extends Bop
  Plus   +
case object Minus extends Bop
  Minus   -
case object Times extends Bop
  Times   *
case object Div extends Bop
  Div   /
```

Figure 1: Representing in Scala the abstract syntax of JAVASCRIPTY. After each **case class** or **case object**, we show the correspondence between the representation and the concrete syntax.

type `Expr`. Then, the result of `eval` is a Scala number of type `Double` and should match the interpretation of e as a JavaScript expression. These distinctions can be subtle but learning to distinguish between them will go a long way in making sense of programming languages.

At this point, you have implemented your first language interpreter!