Homework 1

Due date: Monday, September 21, 2015

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 the course mailing list to locate a partner. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment. You are welcome to talk about these exercises in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

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 expressing computation currently unfamilar to you.

Finally, make sure that your file compiles and runs (using Scala 2.11.7). A program that does not compile will *not* be graded.

Submission instructions. Upload to NYU classes exactly two files named as follows:

- `hw01-YourNYULogin.pdf` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable).
- `hw01-YourNYULogin.scala` with your answers to the coding exercises.

Replace `YourNYULogin` with your NYU login ID (e.g., for me, I would submit files named `hw01-tw47.pdf` and `hw01-tw47.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.

Getting started. Download the code pack `hw01.zip` from the assignment section on the NYU classes page. Follow the `README` file included in the package for help getting your development environment set up.

Problem 1 Scala Basics: Binding and Scope (12 Points)

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? **(4 Points)**

(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? **(8 Points)**

Problem 2 Scala Basics: Typing (10 Points)

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 explanation, first, give the types for the names `a` and `b`. Then, explain the body expression using the following format:

$$
\begin{array}{lll}
e : \tau & \text{because} & \\
 & e_1 : \tau_1 & \text{because} \\
 & & \ldots \\
 & e_2 : \tau_2 & \text{because} \\
 & & \ldots
\end{array}
$$

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
              _ + _ : (Int, Int) => Int
```

Problem 3 Run-Time Library (18 Points)

When we talk about language interpreters we distinguish between the *object language* and the *meta language*. The object language is the language for which we write the interpreter, i.e., the interpreter takes a program written in the object language as input. The *meta language* is the language in which we implement the interpreter itself.

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 exercise, 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 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`. **(2 Points)**

(b) Write a function `swap`

```
def swap(p: (Int, Int)): (Int, Int)
```

that takes a pair of values of type `Int` and returns a new pair with the two values from the input pair swapped. **(2 Points)**

(c) 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. Try to make your function tail-recursive. **(4 Points)**

(d) 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 differentiable

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:

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

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). **(2 Points)**

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. Is your implementation tail-recursive?**(4 Points)**

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,

$$|x^2 - c| < \epsilon \ .$$

You can use your absolute value function `abs` implemented in a previous part. A wrapper function `sqrt` is given in the template that simplifies `sqrtErr` calls with a choice of x_0 and ϵ. Again, challenge yourself to implement this function using recursion and compare your recursive solution to one with a **while** loop. **(4 Points)**