Introduction to Scala and Functional Programming

This exercise set assumes that you have installed Scala on your computer, you have a working programming editor, and you have read the chapters of the book scheduled for this week. Exercises marked **[–]** are meant to be very easy, and should likely be skipped by students that already know Scala and functional programming. Exercises marked **[+]** are cognitively more demanding and show the more typical work done in the course later on.

Exercise 1 [–]. (10 minutes) Download `MyModule.scala` from the course BitBucket repository of examples (https://bitbucket.org/andrzej_wasowski/2016-adpro-code-examples). Compile it using the command line compiler (`scalac MyModule.scala`). Run it using the command line interpreter (`scala MyModule`). Inspect the byte code file using `scalap` and `javap`.

Add a function that computes a square of an integer number to this module, and test it in the `main` method. Recompile the file, run it in the interpreter, and inspect it using `scalap` and `javap`.

Finally compile it using `fsc` (fast Scala compiler). The `fsc` is a drop in replacement for `scalac` that uses a service running in the background to compile faster.

Exercise 2 [–]. (10 minutes) In functional languages it is common to experiment with code in an interactive way (REPL = read-evaluate-print-loop). Start Scala's repl using `scala` without any parameters. Load our module using `:load MyModule.scala`. Then experiment with calling `abs` and `sqaure` interactively. Store results in new values (using `val`).

From this point onwards the exercises proceed in file `Exercises.scala` (from the top of the file). The file contains simple instructions in the top.

Exercise 3. (20 minutes) Implement a function `power` that computes x^n for any `Double` number x and any integer (`Int`) exponent n. Start by establishing the type of the function. Then use the following recursive definition to implement the function:

$$x^n = \begin{cases} x^{n/2}x^{n/2} & \text{if } n \text{ is even and positive} \\ xx^{n-1} & \text{if } n \text{ is odd and positive} \\ 1 & \text{if } n = 0 \\ \frac{1}{x^{-n}} & \text{if } n \text{ is negative} \end{cases} \tag{1}$$

The type of the function shall be: `def power (x: Double, n: Int) : Double`.

The function should be *recursive*. The use of `return` statements and of variables (`var`) is forbidden (in this exercise and in the rest of the course). Briefly test your function in the REPL.

Classify recursive calls: which are in the *tail position* and which are not? What input parameters could make the stack depth large? Is non-tail recursion problematic in this function? [1]

Exercise 4 [+]. (20 minutes) Write a recursive function to get the nth Fibonacci number. The type of the function should be: `def fib (n: Int) : Int`

The first two Fibonacci numbers are 0 and 1. The nth number is always the sum of the previous two–the prefix of the sequence is as follows: $0, 1, 1, 2, 3, 5, \ldots$. Make sure that your definition is tail-recursive (so all calls are in tail positions). Use the `@annotation.tailrec` annotation, to make the compiler check this for you.

[1] Source: Exercise 2.10 [Horstmann 2012]

Remember that an efficient implementation of Fibonacci numbers is by summation bottom-up, not following the recursive mathematical definition. If you are lost with the idea, it might be good to write a for loop first on paper, before attempting a referentially transparent implementation.

Make some rudimentary tests of the function interactively in the REPL. Then record them as assertions in the code. [2]

Exercise 5. (15 minutes) Now consider a very similar exercise that appears to be a bit more realistic. Implement a function that computes a total sum of expenses stored in an `Array[Expense]` (an array containing objects of type `Expense`). First, study the implementation of a simple `class Expense` in `Exercises.scala`. Then implement a function of type:

```
def total (expenses: Array[Expense]) : Int
```

Since we are dealing with more complex objects now it quickly becomes impractical to test in the REPL. Better create test cases in the Scala file and test them using the compiled object.

Do not use the standard Scala method `sum`. Make sure that all recursive calls in your implementation are tail recursive. Use `@annotation.tailrec` again to enforce this discipline during compilation.

Exercise 6. Implement `isSorted`, which checks whether an `Array[A]` is sorted according to a given comparison function:

```
def isSorted[A] (as: Array[A], ordered: (A,A)=>Boolean) :Boolean
```

Ensure that your implementation is tail recursive, and use an appropriate annotation.[3]

Exercise 7. (5 minutes) Recall function `power` from exercise 3. Provide a new implementation of that function that is curried. Use the name `power1` to avoid name and type clashes with `power` in the same module.

Exercise 8 [+]. (10 minutes) Implement a currying function: a function that converts a function `f` of two argument that takes a pair, into a function of one argument that partially applies `f`:

```
def curry[A,B,C] (f: (A,B)=>C) : A =>(B =>C)
```

Use `curry` to automatically obtain `power1` from `power` (cf. exercises 3 and 7).[4]

Exercise 9. (5 minutes) Implement `uncurry`, which reverses the transformation of curry:

```
def unurry[A,B,C] (f: A =>B =>C) : (A,B) =>C
```

Use it to obtain `power` from `power1` automatically.[5]

Exercise 10 [+]. (5 minutes) Implement the higher-order function that composes two functions:

```
def compose[A,B,C] (f: B =>C, g: A =>B) : A =>C
```

Do not use the `Function1.compose` and `Function1reconfigurator.andThen` methods from Scala's standard library. [6]

[2] Exercise 2.1 [Chiusano, Bjarnason 2015]
[3] Exercise 2.2 [Chiusano, Bjarnason 2015]
[4] Exercise 2.3 [Chiusano, Bjarnason 2015]
[5] Exercise 2.4 [Chiusano, Bjarnason 2015]
[6] Exercise 2.5 [Chiusano, Bjarnason 2015]