Introduction to Scala and Functional Programming

This exercise set assumes that:

- *You have installed SBT*[1] *on your computer.* The Scala Build Tool (SBT) is a modern build tool that ensures that the way you execute and build the system is reproducible on different machines. When using `sbt`, you know you are using the same version of the compiler, library, and the build system, that the teachers use. You also know that your classpath is setup correctly. This minimizes incidental problems and saves time when solving exercises. We distribute an `sbt` configuration with every exercise set and project. There is no need to install Scala, but it does not harm either (occasionally it is useful to be able to start `scala` on its own, without `sbt`).
- The robustness of your setup can be increased by using the provided `Dockerfile` (and customizing and building a docker container). Things will work without it, but this helps in case incompatibilities between OSs arise.
- *You have a working programming editor.* We recommend using `sbt`'s command line interface and a simple code editor (vscode, atom, sublime, emacs, vim, etc.).
- *You have read the chapters of the book scheduled for this week.* (Chapters 1–2)

Exercises marked **[–]** should be easy. Skip them if you already know Scala and functional programming. Exercises marked **[+]** are cognitively more demanding and show the more typical work done in the course later on. If you had studied functional programming before, you just need to recall it, and see how things are written in Scala—solve only some exercises. Skip those that you know how to solve. If you are new, we recommend to work intensively in the early weeks.

Do not use variables, side effects, exceptions or return statements.

If you solve all exercises, and your experience with functional programming is *average*, solving all exercises will take about 6 hours, excluding the time used for reading the book, making coffee, chit-chat and smoking :) When skipping the simplest exercises, you can probably do in half of the time. If you have no experience, you are up for a tough and long ride ....

No Hand-in: There is no hand-in this week. Please rely on automatic tests and compiler errors to see whether you are doing fine. Seek contact with teachers to see whether your code meets the quality criteria (our tests only test functionality). We give feedback in exercise sessions and on the forum. It is fine to post your solutions of exercises in the course forum this week.

Exercise 1 [–]. *Learning the toolchain and code layout.* Obtain this week's code from our BitBucket repository (https://github.itu.dk/wasowski/2020-adpro). The code is in the directory `010-intro/`.

Inspect the file `010-intro/src/main/scala/MyModule.scala`. To compile all files in this week's directory execute `sbt compile` in `010-intro/` (or better start `sbt` in this directory and issue command `compile`—this works much faster; sbt is slow to boot). To execute `MyModule` use the command `run` and select `MyModule`. The other choices are the main functions of later exercises. Ignore them for now.

Now run the `test` command of sbt. You will see some tests failing, because you have not solved any exercises yet. Let's zoom into the tests for `MyModule.scala`. We can execute just these tests using the following command: `testOnly MyModuleSpec`. Now you will see a single test failing, that the `square` function is not implemented.

Complete the implementation of the `square` function in `MyModule.scala` (replace the placeholder `???`).

[1] https://www.scala-sbt.org/1.0/docs/Setup.html

Run the test again to check that you have succeeded. Add a line in the `main` method that prints the result of `square` after the absolute value. Recompile the file (`compile`), run it (use `run` and pick the right module, or use `runMain MyModule`).

Now you know the basic tools for the entire course!

A concise guide to SBT: https://www.scala-sbt.org/1.x/docs/sbt-by-example.html

Exercise 2 [–]. In functional languages it is common to experiment with code in an interactive way (REPL = read-evaluate-print-loop). Start Scala's repl using `sbt console`. Experiment with calling `MyModule.abs` and `sqaure` interactively. Store results in new values (using `val`).

Note: to call the functions from `MyModule`, you will need them to be qualified with the object name, e.g. `MyModule.abs`. In order to avoid this, you can import all functions from `MyModule` using: `import MyModule._`. Imports can be added to `build.sbt` so that you don't have to repeat them every time you start the console.[2]

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 [+]. 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: $0, 1, 1, 2, 3, 5, ...$ 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.

For starters you can write a simple recursive version (top down) based on the definition (skip this if you are confident):

$$F_n = F_{n-2} + F_{n-1}$$

An efficient implementation of Fibonacci numbers is by summation bottom-up, not following the recursive mathematical definition. Make some rudimentary tests of the function interactively in the REPL, then try the course test suite (`sbt test`).

Hint: Put a tilda (`~`) in front of an sbt command—it will run automatically every time you change the source file. It is very practical to run `~test`, `~testOnly`, or `~compile`, when working on exercises below. Every time you save the file, you will have the test results almost instantaneously.

Exercise 4. Implement a function that checks if an `Array[A]` is sorted given a 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 5 [+]. 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)
```

[2] https://www.scala-sbt.org/1.x/docs/Howto-Scala.html#Define+the+initial+commands+evaluated+when+entering+the+Scala+REPL

[3] Exercise 2.2 [Chiusano, Bjarnason 2014]

Use it to obtain a curried version of `isSorted` from Exercise 5.[4]

Exercise 6. Implement `uncurry`, which reverses the transformation of curry:

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

Use `uncurry` to obtain `isSorted` back from the curried version created in the Exercise 5.[5]

Exercise 7 [+]. 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 `Function1.andThen` methods from Scala's standard library.[6]

[4] Exercise 2.4 [Chiusano, Bjarnason 2014]
[5] Exercise 2.4 [Chiusano, Bjarnason 2014]
[6] Exercise 2.5 [Chiusano, Bjarnason 2014]