

010 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.

Expected average time to solve this set: 3 hours (An over-qualified student should solve them in 1 hour, and under-qualified student can solve them in 9 hours — but please don't wait until you used 9 hours with asking for help; please ask for hints on the course forums as soon as you get stuck. There is nothing wrong with that. You can be stuck for 90 hours.)

Hand-in: File `Exercises.scala` with solved Exercises 4–8 (or as much as you managed). Make sure that the file compiles (so comment out partial solutions). We don't accept Word files, PDF files, or zipped files. Such files will be rejected. Only a plain `.scala` files that compile are accepted. Also make sure that you have filled in your ITU account names on top of the file.

Exercise 1 [–]. Download `MyModule.scala` from the course BitBucket repository of examples (<https://bitbucket.org/itu-square/2018-adpro-spring>). 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 `jvarkit`.

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 `jvarkit`.

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 [–]. 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 `square` 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`, similarly to Java: `import MyModule._`

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 *n*th Fibonacci number. The type of the function should be: `def fib (n: Int) : Int`

The first two Fibonacci numbers are 0 and 1. The *n*th 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.

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.¹

Exercise 4. 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 5. 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.²

Exercise 6[+]. 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 it to obtain a curried version of `isSorted` from Exercise 5.³

Exercise 7. 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 6.⁴

Exercise 8[+]. 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.⁵

¹Exercise 2.1 [Chiusano, Bjarnason 2014]

²Exercise 2.2 [Chiusano, Bjarnason 2014]

³Exercise 2.4 [Chiusano, Bjarnason 2014]

⁴Exercise 2.4 [Chiusano, Bjarnason 2014]

⁵Exercise 2.5 [Chiusano, Bjarnason 2014]