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 (not without problems...) 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.). If you are an experienced user of IntelliJ or Eclipse, you are welcome to try their integration, but do not ask us for support. You are guaranteed to experience trouble.
- *You have read the chapters of the book scheduled for this week.* (Chapters 1–3)

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. There is *extremely* many exercises this week, by design. The idea is to give students who have little functional programming experience a good material to catch up in the beginning. 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. We will have much less exercises in the following 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 10 hours, plus 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 solutions of exercises this week.

Exercise 1 [–]. *Learning the toolchain and code layout.* Obtain this week's code from our BitBucket repository (https://bitbucket.org/itu-square/2019-adpro-fall). 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

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

`square` function is not implemented.

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

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: $0, 1, 1, 2, 3, 5, \ldots$. 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.

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]

[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]

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)
```

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]

Exercise 8 [–]. What will be the result of the following match expression?[7] Answer the question without running the code on your computer.

```
List(1,2,3,4,5) match {
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(h, t) => h + sum(t)
  case _ => 101
}
```

Exercise 9 [–]. Implement the function `tail` for removing the first element of a list. Note that the function takes constant time. What are different choices you could make in your implementation if the list is `Nil`?[8]

```
def tail[A] (as: List[A]) : List[A]
```

Exercise 10 [–]. Generalize `tail` to the function `drop`, which removes the first `n` elements from a list. Note that this function takes time proportional only to the number of elements being dropped—we do not need to make a copy of the entire list.[9]

```
def drop[A](l: List[A], n: Int): List[A]
```

Exercise 11 [–]. Implement `dropWhile`, which removes elements from the given list prefix as long as they match a predicate `f`.[10]

```
def dropWhile[A](l: List[A], f: A =>Boolean): List[A]
```

[4] Exercise 2.4 [Chiusano, Bjarnason 2014]
[5] Exercise 2.4 [Chiusano, Bjarnason 2014]
[6] Exercise 2.5 [Chiusano, Bjarnason 2014]
[7] Exercise 3.1 [Chiusano, Bjarnason 2014]
[8] Exercise 3.2 [Chiusano, Bjarnason 2014]
[9] Exercise 3.4 [Chiusano, Bjarnason 2014]
[10] Exercise 3.5 [Chiusano, Bjarnason 2014]

Exercise 12 [–]. Implement a function, `init`, that returns a list consisting of all but the last element of the original list. So, given `List(1,2,3,4)`, `init` will return `List(1,2,3)`.

```
def init[A](l: List[A]): List[A]
```

Is this function constant time, like `tail`? Is it constant space? [11]

Exercise 13 [–]. Compute the length of a list using `foldRight`:[12]

```
def length[A](as: List[A]): Int
```

Exercise 14 [+]. The function `foldRight` presented in the book is not tail-recursive and will result in a `StackOverflowError` for large lists. Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that *is* tail-recursive:

```
def foldLeft[A,B](as: List[A], z: B)(f: (B, A) =>B) : B
```

For comparison consider that:

`foldLeft (List(1,2,3,4),0) (_ + _)` computes $(((0 + 1) + 2) + 3) + 4$ while
`foldRight (List(1,2,3,4),0) (_ + _)` computes $1 + (2 + (3 + (4 + 0)))$.

In this case the result is obviously the same, but not always so. The two functions also have different space usage.[13]

Exercise 15 [–]. Write `product` (computing a product of a list of integers) and a function to compute the `length` of a list using `foldLeft`.[14]

Exercise 16. Write a function that returns the reverse of a list (given `List(1,2,3)`, it returns `List(3,2,1)`). Use one of the fold functions (no recursion) [15]

Exercise 17. Write `foldRight` using `foldLeft` (**Hint:** use reverse). This version of `foldRight` is useful because it is tail-recursive, which means it works even for large lists without overflowing the stack.

Exercise 18 [+]. Write `foldLeft` in terms of `foldRight`. Do not use `reverse` here (reverse is a special case of `foldLeft` so a solution based on `reverse` is cheating a bit).

Hint: to do this you will need to synthesize a function that computes the run of `foldLeft`, and then invoke this function. To implement `foldLeft[A,B]` you will be calling

```
foldRight[A,B=>B] (... , ...) (...)
```

that shall compute a new function, which then needs to be called. To our best knowledge this implementation of `foldLeft` has no practical use, but it is an interesting mind twister. It also demonstrates how to use anonymous functions to synthesize and delay computations. This technique is used for many things. We shall use it to implement lazy streams in several weeks.[16]

Note: *From now on we consider use of recursion a bad-smell if a piece of code could be written*

[11] Exercise 3.6 [Chiusano, Bjarnason 2014]
[12] Exercise 3.9 [Chiusano, Bjarnason 2014]
[13] Exercise 3.10 [Chiusano, Bjarnason 2014]
[14] Exercise 3.11 [Chiusano, Bjarnason 2014]
[15] Exercise 3.12 [Chiusano, Bjarnason 2014]
[16] Exercise 3.13 [Chiusano, Bjarnason 2014]

more or less directly using a fold function, or other higher order functions (map, forall, exists, etc.). Recursion should only be used if we are dealing with a non-standard structure of iteration. A fold should only be used if any of the other simpler HOFs cannot be used. This also means that in the course exam, if you use recursion, where a simpler solution with standard HOFs is possible, you will not receive top points.

Exercise 19 [+]. Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists. Use `append`, which concatenates the two lists (described in the book).[17]

Exercise 20. Write a function filter that removes elements from a list unless they satisfy a given predicate `f`.[18]

```
def filter[A](as: List[A])(f: A =>Boolean) : List[A]
```

Exercise 21 [+]. Write a function `flatMap` that works like map except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
def flatMap[A,B] (as: List[A]) (f: A =>List[B]): List[B]
```

For instance, `flatMap (List(1,2,3)) (i =>List(i,i))` should result in `List(1,1,2,2,3,3)`. `flatMap` will be key in the rest of the course (together with `map`).[19]

Exercise 22. Use `flatMap` to implement `filter`. Both are standard HOFs in Scala's libraries. They were also introduced in the slides and in the book.[20]

Exercise 23 [–]. Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6,7)` become `List(5,7,9)`.Trailing elements of either list are dropped.[21]

Exercise 24 [–]. Generalize the function you just wrote so that it is not specific to integers or addition. It should work with arbitrary binary operations. Name the new function `zipWith`.[22]

Exercise 25 [+]. Implement a function `hasSubsequence` for checking whether a `List` contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)` , `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function however comes most naturally. Note: Any two values `x` and `y` can be compared for equality in Scala using the expression `x ==y`. Here is the suggested type:

```
def hasSubsequence[A](sup: List[A], sub: List[A]): Boolean
```

Recall that an empty sequence is a subsequence of any other sequence.[23]

[17] Exercise 3.15 [Chiusano, Bjarnason 2014]
[18] Exercise 3.19 [Chiusano, Bjarnason 2014]
[19] Exercise 3.20 [Chiusano, Bjarnason 2014]
[20] Exercise 3.21 [Chiusano, Bjarnason 2014]
[21] Exercise 3.22 [Chiusano, Bjarnason 2014]
[22] Exercise 3.23 [Chiusano, Bjarnason 2014]
[23] Exercise 3.24 [Chiusano, Bjarnason 2014]

Exercise 26 [+]. Recall the structure of Pascal's triangle structure (this animation summarizes the key information needed: https://upload.wikimedia.org/wikipedia/commons/0/0d/PascalTriangleAnimated2.gif). Write a recursive function `pascal (n :Int) :List[Int]` that generates the nth row of Pascal's triangle. For example, `pascal(1)` should generate `List(1)`, `pascal(2)` should generate `List(1,1)`, `pascal(3)` should generate `List(1,2,1)` and `pascal(4)` should generate `List(1,3,3,1)`.