Introduction to Scala and Functional Programming

It is recommended to read Chapters 1–2 in the book before solving the exercises.

No Hand-in: There is no hand-in this week. Aim at all tests passing.

Do not use variables, side effects, exceptions or return statements.
Exercises are labeled Easy (**E**), Medium (**M**), and Hard (**H**).

Exercise 1. [E] The code is in the directory `010-intro/`.
Inspect `README.md` and `010-intro/Exercises.scala`.

1. Follow the `README.md` to compile all files in the directory.
2. Execute `MyModule` using the `run` command (as described in `README.md`). Convince yourself that the output is as expected, and that the assertion in the main function should not crash.
3. Run all tests (all should fail). Identify the message that the test for Exercise 1 has failed.
4. Complete the implementation of the `square` function in the `MyModule` object (replace the placeholder `???`).
5. Run the test again to check that you have succeeded.
6. Add a line in the `main` method that prints the result of `square` after the absolute value. Recompile the file and run it to convince yourself that your function has been called.

Why am I solving this exercise? To learn the basics of running the scala compiler tool chain and code layout.

Exercise 2. [E] In functional languages it is common to experiment with code in an interactive way in a REPL (read-evaluate-print-loop). Start Scala's repl using `scala-cli repl .`. This starts scala with your project loaded and the classpath configured. Experiment with calling `adpro.intro.MyModule.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 package and/or object name, e.g. `adpro.intro.MyModule.abs`. In order to avoid this, you can import all functions from `MyModule` using: `import adpro.intro.MyModule.*`.

Exercise 3. [H] The first two Fibonacci numbers are $F_1 = 0$ and $F_2 = 1$. The nth number is always the sum of the previous two–the prefix of the sequence is as follows: $0, 1, 1, 2, 3, 5$, ...

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

Recall that an efficient implementation of Fibonacci numbers is by summation bottom-up (from 0 and 1), not by following the recursive mathematical definition (which gives an exponential algorithm).

Implement this efficient solution and use `@annotation.tailrec` to make the compiler check for you that it is tail recursive. Make some rudimentary tests of the function interactively in the REPL, besides using the provided test suite (`scala-cli test`).

Note: In this course, we do not overemphasize tail recursion. We prefer simplicity over optimization, so do not insist on tail recursion unless explicitly asked. The tail-recursive transformation of a recursive function is a case of premature optimization, if there is no risk of exhausting the stack space.

Exercise 4. [M] Implement a higher order function that checks if an `Array[A]` is sorted given a comparison function as an argument:

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

Ensure that your implementation is tail recursive, and use an appropriate annotation.[1]

Example 1: `isSorted[Int] (Array (1,2,3), (x,y) =>x <=y)` should be true
Example 2: `isSorted[Int] (Array (2,2,2), (x,y) =>x ==y)` should be true
Example 3: `isSorted[Int] (Array (2,2,2), (x,y) =>x < y)` should be false

Do not use pattern matching. We did not introduce the syntax for this yet. The point of the exercise is to do with recursion, what you normally do with a for-loop over array indices in an imperative programming style.

Exercise 5. [H] Implement a currying function: a function that converts a binary function `f` (function taking two arguments) into a function that takes one argument and after getting the argument returns a function awaiting the other argument. Once both arguments are given, the transformed function should behave the same as the original `f`:[2]

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

Use it to obtain a curried version of `isSorted` from Exercise 4, so a function of the following type:

```
isSortedCurried:Array[A] =>((A,A) =>Boolean) =>Boolean .
```

Why is currying useful? It allows to apply function arguments partially. For instance the first argument can configure the function for the project, the second can be an actual runtime argument. We need to only apply the first argument once to obtain a specialized function, configured correctly and consistently for the entire project. Very often currying and uncurrying (see below) is just use to make type checking work, in situations when your function is curried (uncurried) but needs to be passed to a higher order function that requires an uncurried (curried) argument.

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

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

Use `uncurry` to obtain `isSortedCurriedUncurried` (equivalent to `isSorted`) from the curried version created in the Exercise 5.[3]

Exercise 7. [M] 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 (the point is to implement the corresponding functionality yourself).[4]

[1] Exercise 2.2 [Pilquist, Chiusano, Bjarnason, 2023]
[2] Exercise 2.3 [Pilquist, Chiusano, Bjarnason, 2023]
[3] Exercise 2.4 [Pilquist, Chiusano, Bjarnason, 2023]
[4] Exercise 2.5 [Pilquist, Chiusano, Bjarnason, 2023]