Exercise 6: Introduction to Scala

NTNU

TDT4165 høst 2018

Scala is both a functional and object-oriented programming language. This exercise will introduce you to Scala and familiarize you with concepts utilized in the upcoming Scala project. While you may be tempted to approach this exercise in an object-oriented way, **we want you to solve the tasks functionally, like you would do Haskell**. Many things that are done in Haskell are doable in Scala, and we will actively compare them so you will have more ease with the transition.

The easiest way to run Scala is to install **sbt** at http://www.scala-sbt.org/download.html. Scala builds on top of Java, so you will need to install a Java environment as well.

1 Hello World

(a) Create a new folder with a file called `Main.scala` with the following content

```
object Hello extends App {
    println("Hello World")
}
```

Navigate to the folder you just created and run your code using `sbt run` in the terminal.

To get used to Scala we start looking at arrays and variables:

```
object Hello extends App {
    val x = Array(1, 2, 3)
    println(s"There are ${x.length} elements")
    for (i <- x) println(i)
}
```

In this example, we bound an array to a `val` x. Note that we cannot change an object that was assigned to a `val`, while a `var` may change its binding.

(b) Generate an array containing the values 1 up to (and including) 50 using a for loop.

(c) Unlike Haskell where only immutable data structures are allowed, in Scala, we may use both. Array is one such mutable data structure. Append to the previous array the values 51 to 100 (inclusive) using a `Range`. (Hint: Use 51 to 100 (evaluates to a `Range`), the `map` method, and a lambda (anonymous function))

(d) Create a function that sums the elements in an array of integers using a for loop.

(e) Create a function that sums the elements in an array of integers using recursion.

(f) Create a function to compute the nth Fibonacci number using recursion, without using memoization (or other optimizations). Use `BigInt` instead of `Int`. What is the difference?

2 Lazy Evaluation

As shown in exercise 2, Haskell is lazy: an expression is evaluated only if it is used. Scala is strict by default, but can be lazy if explicitly specified.

(a) One way to specify laziness in Scala is by using anonymous functions. Create a function `nonlazy_exec` that takes in 2 arguments: a lambda **function** of the signature $() \implies$ `BigInt`, and a `Boolean`. The `nonlazy_exec` function shall: first assign the lambda's return value to a val (when we call the lambda, the return value shall be the val). Then if the Boolean evaluates to true, print the val. Otherwise exit the function without printing. The order of the above operations are to be followed exactly.

Your function must be callable like so:

```
nonlazy(() => nth_fibonacci(30), true)
```

(b) Create the function `lazy_exec` with the same operations as in **(a)**, but assign the lambda's value to a `lazy val` instead of `val`.

(c) Play around with the functions' calls and explain the differences between them. When do you think it is helpful to use lazy evaluation?

3 Concurrency in Scala

One of the most important goals in the Scala project is to learn concurrency programming. Here, it is done by using threads. You can read more about how to program threads in Scala at [here]. The file "Learn Concurrent Programming in Scala.pdf" uploaded on Blackboard will also be helpful for this task.

(a) Create a function that takes as argument a function and returns a Thread initialized with the input function. Make sure that the returned thread is not started.

(b) Create a recursive function that creates `n` lambdas. Each lambda prints its corresponding nth Fibonacci number. (This means you return an Array of functions, the return type being `Array[() => Unit]`)

(c) Use the result from your lambda generator in **b)** and thread generator from **a)** and `map` each lambda to a thread. (Hint: Use the `map` method on the `Array` with the argument being the function you made in **a)**.

(d) `map` each thread from **c)** to start. What does this do and what kind of Array does this return?

(e) The code snippet below is not thread-safe. Why is this so and how would you change it so that it is thread-safe? Hint: atomicity.

```
private var counter: Int = 0

def increaseCounter(): Int = {
    counter += 1
    counter
}
```

(f) One problem you will often meet in concurrency programming is deadlock. What is deadlock, and what can be done to prevent it? Write in Scala an example of a deadlock using lazy val.