INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 01: Introduction to Scala

We assume that you have installed Scala and an IDE of your choice. Throughout these exercises, we will use MS Visual Studio. We also assume that you are familiar with a terminal.

Exercise 1:

In this exercise, we will run our first Scala program by copy and pasting some code. Create the file `TP1.scala`, and copy and paste the following code inside that file. You may omit the curly braces as long as you respect the indentation rules (much like Python). If you omit the curly braces, ensure that the last line is a new line without any white space characters (i.e., a carriage return + new line).

```
def square(x: Int): Int = {
    x * x
}

def cube(x: Int): Int = {
    x * x * x
}
```

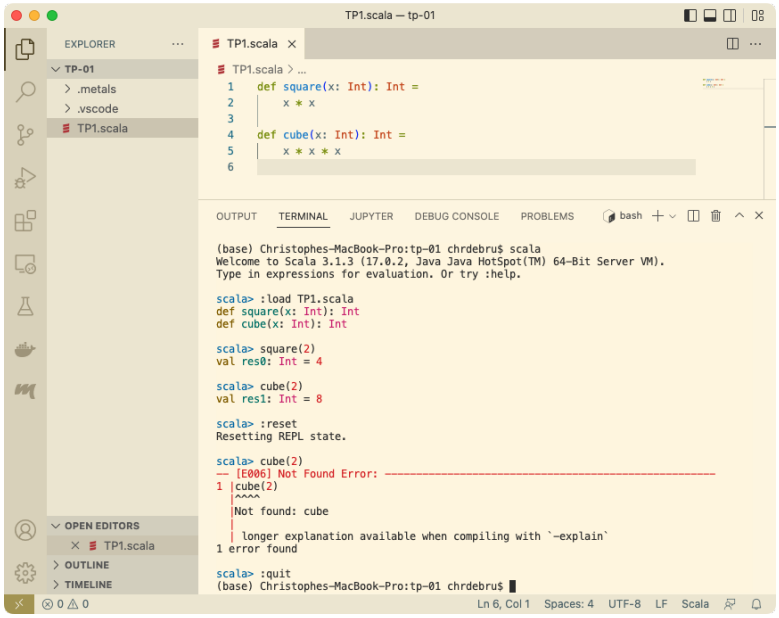
1. Open the Scala REPL (Read-Eval-Print Loop) environment from the command line and load this file by executing: `:load TP1.scala`
2. Try invoking the functions `square` and `cube`.
3. Now reset the REPL environment by invoking `:reset`. Can you still invoke the function of your file? Why is that?
4. Quit the REPL environment by invoking `:quit`.

Solution 1:

In the figure below, we demonstrate how we load a Scala source file and invoking its functions. We also demonstrate resetting and exiting the REPL environment. In this figure, we have omitted the curly braces. Notice the newline on line 6.

Resetting the REPL environment to its initial state means that all of our definitions are removed. The definitions of Scala's standard libraries are not removed. We encourage you to get familiar with resetting the REPL environment as you may inadvertently redefine only some definitions, inadvertently rely on definitions that have been removed in the code, and so forth.

For instance, let's assume we have made two mistakes: 1) the function cube was misspelled as `cute`, and we had implemented `square` as `x * 2` instead of `x * x`. Those two functions are already loaded in REPL. What will happen if we fix those two mistakes and reload our file with `:load TP1.scala`? We will have redefined the function square. We will furthermore define a new function cube. The "old" function cute will remain in the environment, however!



Exercise 2:

In this exercise, you will modify `TP1.scala`. Place the functions cube and square in a module `TP1`. First, load the file and invoke the functions from that module. Now try importing these function definitions from that module into the REPL environment so that you do not need to prefix them with their module. Do not forget to reset your REPL environment if you start from the previous exercise!

Solution 2:

```
object TP1:
    def square(x: Int): Int =
        x * x

    def cube(x: Int): Int =
        x * x * x
```

If you omit curly braces, you have to ensure you respect the indentation. In this example, we must have a new line on line 7. Some systems or environments do not like empty newlines. In Scala, you can explicitly indicate an object's end with an "end marker." Those end makers must appear at the same place where the newline should be.

```
object TP1:
    def square(x: Int): Int =
        x * x

    def cube(x: Int): Int =
        x * x * x
end TP1 // This is the end marker of TP1
```

Now let's use our module. First we will call our functions via our module. Then we will import our modules in the REPL environment.

```
scala> :load TP1.scala
// defined object TP1

scala> TP1.cube(2)
val res0: Int = 8
```

```
scala> TP1.cube(2) + TP1.square(3)
val res1: Int = 17

scala> import TP1._

scala> cube(2) + square(3)
val res2: Int = 17
```

In the next few weeks, we will see more tricks with respect to importing definitions from modules, objects, packages, and so forth.

Exercise 3:

Create the function `roots` that takes as input the coefficients a, b, and c of a quadratic polynomial $ax^2 + bx + c$. `roots` returns a list of roots given by the formula $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. If $b^2 - 4ac < 0$, then there are no roots. If $b^2 - 4ac = 0$, then there is only one root. In all other cases, there are two roots.

Solution 3:

```
def roots(a: Double, b: Double, c: Double): List[Double] = {
    val t = math.pow(b, 2) - (4 * a * c)
    if (t < 0) List[Double]()
    else if (t == 0) List(-b / (2 * a))
    else List((-b - math.sqrt(t)) / (2 * a), (-b + math.sqrt(t)) / (2 * a))
}
```

```
scala> roots(1,0,0)
val res3: List[Double] = List(-0.0)

scala> roots(-1,0,-3)
val res5: List[Double] = List()

scala> roots(2,0,-2)
val res6: List[Double] = List(-1.0, 1.0)
```

Exercise 4:

(Challenge) Define `sumInt1`, a recursive function that takes as argument a natural number (positive integer in Scala) n and returns the sum of all natural numbers lower than or equal to n. Is your definition tail recursive? Justify your answer. If `sumInt1` is tail-recursive, define a version `sumInt2` that is recursive, but not tail recursive. If `sumInt1` is recursive, `sumInt12` should use a local tail-recursive function.

Solution 4:

```
def sumInt1(n: Int): Int =
    if n == 0 then 0
    else n + sumInt1(n - 1)

def sumInt2(n: Int): Int =
    @annotation.tailrec
    def iter(n: Int, acc: Int): Int =
        if n == 0 then acc
        else iter(n - 1, acc + n)
    iter(n, 0)
```

References

[1] Paul Chiusano and Rnar Bjarnason. 2015. Functional Programming in Scala (2nd. ed.). Manning Publications Co., USA.