1. Exercises: Introduction to Scala

DCC-FCUP, University of Porto
José Proença

Concurrent Programming – Part 2

These exercises are taken mainly from the book "*Learning Concurrent Programming in Scala*", and are designed to test the knowledge of the Scala programming language. You should solve them by sketching out a pseudocode solution, rather than a complete Scala program.

Exercise 1. Set up your computer to compile and run Scala.

1.1. Install SBT – `http://www.scala-sbt.org`.

1.2. Open a Command Promp.

1.3. Create a folder for the project:

```
> mkdir cp2324
> cd cp2324
```

1.4. Create a source code directory for our exercises:

```
> mkdir -p src/main/scala/cp/lablessons/
```

1.5. Create the configuration file `build.sbt` below at the root of `cp2324` (empty lines are mandatory).

```
name := "CP2324"

version := "1.0"

scalaVersion := "2.12.18"
```

1.6. Make your hello-world program. Use your favourite editor `EDIT`.

```
> mkdir src/main/scala/cp/lablessons/lesson1
> EDIT src/main/scala/cp/lablessons/lesson1/HelloWorld.scala
```

```
package cp.lablessons.lesson1

object HelloWorld extends App {
  println("Hello,␣world!")
}
```

1.7. Go back to the terminal and run SBT to start an interactive shell:

```
> sbt
```

1.8. Run your program:

```
sbt> run
```

Note: *You can also compile, ∼compile, console, and many other. For example, you can compile and run java from the command line:*

```
java -cp ~/.sbt/boot/scala-2.12.12/lib/scala-library.jar:target/scala-2.12/classes/
cp.lablessons.lesson1.HelloWorld
```

If multiple Apps exist, run a specific one with the command:

```
sbt> runMain cp.lablessons.lesson1.HelloWorld
```

1.9. Extend the `build.sbt` with dependencies that we may need during the semester.

```
resolvers ++= Seq(
  "Sonatype OSS Snapshots" at
    "https://oss.sonatype.org/content/repositories/snapshots",
  "Sonatype OSS Releases" at
    "https://oss.sonatype.org/content/repositories/releases",
  "Typesafe Repository" at
    "https://repo.typesafe.com/typesafe/releases/"
)

libraryDependencies ++= Seq(
  "commons-io" % "commons-io" % "2.4"
  ,"com.typesafe.akka" %% "akka-actor" % "2.8.5"
  ,"com.typesafe.akka" %% "akka-remote" % "2.8.5"
)
)
```

If you are in the SBT shell and you modify the `build.sbt` you need to reload it:
`sbt> reload`

Exercise 2. Implement a `compose` method with the following signature:

```
def compose[A, B, C](g: B => C, f: A => B): A => C = ???
```

This method must return a function `h`, which is the composition of the functions `f` and `g`.

Exercise 3. Implement a `fuse` method with the following signature:

```
def fuse[A, B](a: Option[A], b: Option[B]): Option[(A, B)] = ???
```

The resulting `Option` object should contain a tuple of values from the Option objects `a` and `b`, given that both `a` and `b` are non-empty. Implement two variations: with for-comprehensions and with pattern matching.

Exercise 4. Implement a `check` method, which takes a set of values of type `T` and a function of type `T => Boolean`:

```
def check[T](xs: Seq[T])(pred: T => Boolean): Boolean = ???
```

The method must return true if and only if the `pred` function returns true for all the values in `xs` without throwing an exception. Use the `check` method as follows:

```
check(0 until 10)(40 / _ > 0)
```

Note: *The `check` method has a curried definition: instead of just one parameter list, it has two of them. Curried definitions allow a nicer syntax when calling the function, but are otherwise semantically equivalent to single-parameter list definitions.*

Exercise 5. Modify the `Pair` class from the theoretical lessons so that it can be used in a pattern match. Implement a method `+(other:Pair): Pair` using pattern matching and test it.

Exercise 6. Implement a `permutations` function, which, given a string, returns a sequence of strings that are lexicographic permutations of the input string:

```scala
def permutations(x: String): Seq[String]
```

Exercise 7. Implement a combinations function that, given a sequence of elements, produces an iterator over all possible combinations of length `n`. A combination is a way of selecting elements from the collection so that every element is selected once, and the order of elements does not matter. For example, given a collection `Seq(1,4,9,16)`, combinations of length 2 are `Seq(1,4)`, `Seq(1,9)`, `Seq(1,16)`, `Seq(4,9)`, `Seq(4,16)`, and `Seq(9,16)`. The combinations function has the following signature:

```scala
def combinations(n: Int, xs: Seq[Int]): Iterator[Seq[Int]]
```

See the Iterator API in the standard library documentation. (Suggestion: implement first a functional variation that returns `Seq[Seq[Int]]`, without the iterator.)

Exercise 8. Implement a method that takes a regular expression, and returns a partial function from a string to lists of matches within that string:

```scala
def matcher(regex: String): PartialFunction[String,List[String]]
```

The partial function should not be defined if there are no matches within the argument strings. Otherwise, it should use the regular expression to output the list of matches.