A Crash-Course on Scala

Scala is programming language that combines functional and object-oriented programming-styles, and has received in the last five years quite a bit of attention. One reason for this attention is that, like the Java programming language, it compiles to the Java Virtual Machine (JVM) and therefore can run under MacOSX, Linux and Windows.[1] Unlike Java, however, Scala often allows programmers to write concise and elegant code; some therefore say Scala is the much better Java. If you want to try it out, the Scala compiler can be downloaded from

```
http://www.scala-lang.org
```

Why do I use Scala in the AFL course? Actually, you can do any part of the programming coursework in any programming language you like. I use Scala for showing you code during the lectures because its functional programming-style allows me to implement some of the functions we will discuss with very small and elegant code. Since the compiler is free, you can download it and run every example I give. But if you prefer, you can also translate the examples into any other functional language, for example Haskell, ML, F# and so on.

Writing programs in Scala can be done with the Eclipse IDE and also with IntelliJ, but for the small programs we will look at the Emacs-editor id good for me and I will run programs on the command line. One advantage of Scala over Java is that it includes an interpreter (a REPL, or Read-Eval-Print-Loop) with which you can run and test small code-snippets without the need of the compiler. This helps a lot for interactively developing programs. Once you installed Scala correctly, you can start the interpreter by typing

```
$ scala
Welcome to Scala version 2.11.2 (Java HotSpot(TM) 64-Bit Server VM).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

The precise output may vary due to the platform where you installed Scala. At the scala prompt you can type things like `2 + 3` Ret and the output will be

```
scala> 2 + 3
res0: Int = 5
```

indicating that the result of the addition is of type `Int` and the actual result is `5`. Another example you can type in is

```
scala> print ("hello world")
hello world
```

[1] There are also experimental backends for Android and JavaScript.

which prints out a string. Note that in this case there is no result: the reason is that `print` does not actually produce a result (there is no `res_`), rather it is a function that causes the side-effect of printing out a string. Once you are more familiar with the functional programming-style, you will know what the difference is between a function that returns a result, like addition, and a function that causes a side-effect, like `print`. We shall come back to this later, but if you are curious, the latter kind of functions always have as return type `Unit`.

Inductive Datatypes

The elegance and conciseness of Scala programs stems often from the fact that inductive datatypes can be easily defined. For example in "every-day Mathematics" we would define regular expressions simply by the grammar

$$
\begin{array}{rcll}
r & ::= & \varnothing & \text{null} \\
 & | & \epsilon & \text{empty string} \\
 & | & c & \text{single character} \\
 & | & r_1 \cdot r_2 & \text{sequence} \\
 & | & r_1 + r_2 & \text{alternative / choice} \\
 & | & r^* & \text{star (zero or more)}
\end{array}
$$

This grammar specifies what regular expressions are (essentially a kind of tree-structure with three kinds of inner nodes and three kinds of leave nodes). If you are familiar with Java, it might be an instructive exercise to define this kind of inductive datatypes in Java.[2]

Implementing the regular expressions from above in Scala is very simple: It first requires an abstract class, say, `Rexp`. This will act as type for regular expressions. Second, it requires some instances. The cases for $\varnothing$ and ϵ do not have any arguments, while in the other cases we do have arguments. For example the character regular expression needs to take as an argument the character it is supposed to recognise. In Scala, the cases without arguments are called case objects, while the ones with arguments are case classes. The corresponding code is as follows:

```
abstract class Rexp
case object NULL extends Rexp
case object EMPTY extends Rexp
case class CHAR (c: Char) extends Rexp
case class SEQ (r1: Rexp, r2: Rexp) extends Rexp
case class ALT (r1: Rexp, r2: Rexp) extends Rexp
case class STAR (r: Rexp) extends Rexp
```

Given the grammar above, I hope you can see the underlying pattern. In order to be an instance of `Rexp`, each case object or case class needs to extend `Rexp`.

[2] Happy programming! ;o)

If you want to play with such definitions, feel free to define for example binary trees.

Once you make a definition like the one above, you can represent, say, the regular expression for $a + b$ as `ALT(CHAR('a'), CHAR('b'))`. If you want to assign this regular expression to a variable, you can just type

```
scala> val r = ALT(CHAR('a'), CHAR('b'))
r: ALT = ALT(CHAR(a),CHAR(b))
```

In order to make such assignments there is no constructor need in the class (like in Java). However, if there is the need, you can of course define such a constructor in Scala.

Note that Scala says the variable `r` is of type `ALT`, not `Rexp`. Scala always tries to find the most general type that is needed for a variable, but does not "over-generalise". In this case there is no need to give `r` the more general type of `Rexp`. This is different if you want to form a list of regular expressions, for example

```
scala> val ls = List(ALT(CHAR('a'), CHAR('b')), NULL)
ls: List[Rexp] = List(ALT(CHAR(a),CHAR(b)), NULL)
```

In this case Scala needs to assign a type to the regular expressions, so that it is compatible with the fact that list can only contain elements of a single type, in this case this is `Rexp`.[3] Note that if a type takes another type as argument, this is written for example as `List[Rexp]`.

Functions and Pattern-Matching

Types

Cool Stuff

[3] If you type in this example, you will notice that the type contains some further information, but lets ignore this for the moment.