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, Scala 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 yourself, 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 the functions we will discuss with very small code-snippets. Since the compiler is free, you can download them and run every example I give. But if you prefer, you can also easily translate the code-snippets 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 I will develop the good old Emacs-editor is adequate for me and I will run the 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 with 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 response 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 classic example you can try out is

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

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

Note that in this case there is no result: the reason is that `print` does not actually produce a result (there is no `resXX`), 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 point in due course, but if you are curious now, the latter kind of functions always have as return type `Unit`.

If you want to write a stand-alone app, you can implement an object that is an instance of `App`, say

```
object Hello extends App {
    println ("hello world")
}
```

save it in a file, say `hellow-world.scala`, and then run the compiler and runtime environment:

```
$ scalac hello-world.scala
$ scala Hello
hello world
```

As mentioned above, Scala targets the JVM and consequently Scala programs can also be executed by the bog-standard Java runtime. This only requires the inclusion of `scala-library.jar`, which on my computer can be done as follows:

```
$ scalac hello-world.scala
$ java -cp /usr/local/src/scala/lib/scala-library.jar:. Hello
hello world
```

Inductive Datatypes

The elegance and conciseness of Scala programs are often a result of inductive datatypes that can be easily defined. For example in "every-day mathematics" we would define regular expressions simply by giving 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—sequence, alternative and star—and three kinds of leave nodes—null, empty and character). 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 the type for regular expressions. Second, it requires some instances. The cases for $\varnothing$ and ϵ do not have any arguments, while in all 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
```

In order to be an instance of `Rexp`, each case object and case class needs to extend `Rexp`. Given the grammar above, I hope you can see the underlying pattern. If you want to play further with such definitions, feel free to define for example binary trees.

Once you make a definition like the one above, you can represent, for example, the regular expression for $a + b$ in Scala as `ALT(CHAR('a'), CHAR('b'))`. Expressions such as `'a'` stand for ASCII characters. If you want to assign this regular expression to a variable, you can use the keyword `val` and type

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

As you can see, in order to make such assignments, no constructor is required in the class (as in Java). However, if there is the need for some non-standard initialisation, you can of course define such a constructor in Scala. But we omit this here.

Note that Scala in its response says the variable `r` is of type `ALT`, not `Rexp`. This might be a bit unexpected, but can be explained as follows: Scala always tries to find the most general type that is needed for a variable or expression, 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)
```

[2] Happy programming! ;o)

In this case Scala needs to assign a common type to the regular expressions, so that it is compatible with the fact that lists can only contain elements of a single type, in this case the type is `Rexp`.[3]

For types like `List[...]` the general rule is that when a type takes another type as argument, then this is written in angle-brackets. This can also contain nested types as in `List[Set[Rexp]]`, which is a list of sets each of which contains regular expressions.

Functions and Pattern-Matching

I mentioned above that Scala is a very elegant programming language for the code we will write in this module. This elegance mainly stems from the fact that functions can be implemented very easily in Scala. Lets first consider a problem from number theory, called the Collatz-series, which corresponds to a famous unsolved problem in mathematics.[4] Mathematician define this series as:

$$collatz_{n+1} \stackrel{\text{def}}{=} \begin{cases} \frac{1}{2} * collatz_n & \text{if } collatz_n \text{ is even} \\ 3 * collatz_n + 1 & \text{if } collatz_n \text{ is odd} \end{cases}$$

The famous unsolved question is whether this series started with any $n > 0$ as $collaz_0$ will always return to 1. This is obvious when started with 1, and also with 2, but already needs a bit of head-scratching for the case of 3.

If we want to avoid the head-scratching, we could implement this as the following function in Scala:

```
1 def collatz(n: BigInt) : Boolean = {
2   if (n == 1) true else
3   if (n % 2 == 0) collatz(n / 2) else
4   collatz(3 * n + 1)
5 }
```

The keyword for function definitions is `def` followed by the name of the function. After that you have a list of arguments (enclosed in parentheses and separated by commas). Each argument in this list needs its type annotated. In this case we only have one argument, which is of type `BigInt`. This type stands for arbitrary precision integers. After the arguments comes the type of what the function returns—a Boolean in this case for indicating that the function has reached `1`. Finally, after the `=` comes the body of the function implementing what the function is supposed to do. What the `collatz` function does should be pretty self-explanatory: the function first tests whether `n` is equal to 1 in which case it returns `true` and so on.

Notice a quirk in Scala's syntax for `if`s: The condition needs to be enclosed in parentheses and the then-case comes right after the condition—there is no `then` keyword in Scala.

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

[4] See for example `http://mathworld.wolfram.com/CollatzProblem.html`.

The real power of Scala comes, however, from the ability to define functions by pattern matching. In the `collatz` function above we need to test each case using a sequence of `if`s. This can be very cumbersome and brittle if there are many cases. If we wanted to define a function over regular expressions in say Java, which does not have pattern-matching, the resulting code would just be awkward.

Mathematicians already use the power of pattern-matching, for example, when they define the function that takes a regular expression and produces another regular expression that can recognise the reversed strings. The resulting recursive function is often defined as follows:

$$\begin{array}{rcl} rev(\varnothing) & \stackrel{\text{def}}{=} & \varnothing \\ rev(\epsilon) & \stackrel{\text{def}}{=} & \epsilon \\ rev(c) & \stackrel{\text{def}}{=} & c \\ rev(r_1 + r_2) & \stackrel{\text{def}}{=} & rev(r_1) + rev(r_2) \\ rev(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & rev(r_2) \cdot rev(r_1) \\ rev(r^*) & \stackrel{\text{def}}{=} & rev(r)^* \end{array}$$

The corresponding Scala code looks very similar to this definition, thanks to pattern-matching.

```
1 def rev(r: Rexp) : Rexp = r match {
2   case NULL => NULL
3   case EMPTY => EMPTY
4   case CHAR(c) => CHAR(c)
5   case ALT(r1, r2) => ALT(rev(r1), rev(r2))
6   case SEQ(r1, r2) => SEQ(rev(r2), rev(r1))
7   case STAR(r) => STAR(rev(r))
8 }
```

The keyword for starting a pattern-match is `match` followed by a list of `case`s. Before the match can be another pattern, but often as in the case above, it is just a variable you want to pattern-match.

In the `rev`-function above, each case follows the structure of how we defined regular expressions as inductive datatype. For example the case in Line 3 you can read as: if the regular expression `r` is of the form `EMPTY` then do whatever follows the `=>` (in this case just return `EMPTY`). Line 5 reads as: if the regular expression `r` is of the form `ALT(r1, r2)`, where the left-branch of the alternative is matched by the variable `r1` and the right-branch by `r2` then do "something". The "something" can now use the variables `r1` and `r2` from the match.

If you want to play with this function, call, it for example, with the regular expression $ab + ac$. This regular expression can recognise the strings ab and ac. The function `rev` produces the result $ba + ca$, which can recognise the reversed strings ba and ca.

In Scala each pattern-match can also be guarded as in

```
case Pattern if Condition => Do_Something
```

This allows us, for example, to re-write the `collatz`-function from above as follows:

```
1 def collatz(n: BigInt) : Boolean = n match {
2   case n if (n == 1) => true
3   case n if (n % 2 == 0) => collatz(n / 2)
4   case _ => collatz(3 * n + 1)
5 }
```

Although in this case the pattern-match does not improve the code in any way. The underscore in the last case indicates that we do not care what the pattern looks like. Thus Line 4 acts like a default case whenever the cases above did not match. Cases are always tried out from top to bottom.

Loops

Coming from Java or C, you might be surprised that Scala does not really have loops. It has instead, what is in functional programming called maps. To illustrate how they work, lets assume you have a list of numbers from 1 to 10 and want to build the list of squares. The list of numbers from 1 to 10 can be constructed in Scala as follows:

```
scala> (1 to 10).toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Generating from this list the list of squares in a non-functional language (e.g. Java), you would assume the list is given as a kind of array. You would then iterate, or loop, an index over this array and replace each entry in the array by the square. Right? In Scala, and in other functional programming languages, you use maps to achieve the same.

Maps essentially take a function that describes how each element is transformed (for example squaring) and a list over which this function should work. There are two forms to express maps in Scala. The first way is in a `for`-construction. Squaring the numbers from 1 to 10 would look in this form as follows:

```
scala> for (n <- (1 to 10).toList) yield n * n
res2: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

The keywords are `for` and `yield`. This `for`-construction roughly says that from the list of numbers we draw `n`s and compute the result of `n * n`. In consequence we specified the list where each `n` comes from, namely `(1 to 10).toList`, and how each element needs to be transformed. This can also be expressed in a second way by using directly `map` as follows:

```
scala> (1 to 10).toList.map(n => n * n)
res3 = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

In this way the expression `n => n * n` stands for the function that calculates the square (this is how the `n`s are transformed). This expression for functions might remind you on your lessons about the lambda-calculus where this would have been written as $\lambda n.\, n * n$.

It might not be obvious, but `for`-constructions are just syntactic sugar: when compiling, Scala translates them into equivalent maps.

The very charming feature of Scala is that such maps or `for`-constructions can be written for any kind of data collection, such as lists, sets, vectors and so on. For example if we instead compute the reminders modulo 3 of this list, we can write

```
scala> (1 to 10).toList.map(n => n % 3)
res4 = List(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

If we, however, transform the numbers 1 to 10 not into a list, but into a set, and then compute the reminders modulo 3 we obtain

```
scala> (1 to 10).toSet[Int].map(n => n % 3)
res5 = Set(2, 1, 0)
```

This is the correct result for sets, as there are only 3 equivalence classes of integers modulo 3. Note that we need to "help" Scala in this example to transform the numbers into a set of integers by explicitly annotating the type `Int`. Since maps and `for`-construction are just syntactic variants of each other, the latter can also be written as

```
scala> for (n <- (1 to 10).toSet[Int]) yield n % 3
res5 = Set(2, 1, 0)
```

While hopefully this all looks reasonable, there is one complication: In the examples above we always wanted to transform one list into another list (e.g. list of squares), or one set into another set (set of numbers into set of reminders modulo 3). What happens if we just want to print out a list of integers? Then actually the `for`-construction needs to be modified. The reason is that `print`, you guessed it, does not produce any result, but only produces, in the functional-programming-lingo, a side-effect. Printing out the list of numbers from 1 to 5 would look as follows

```
scala> for (n <- (1 to 5).toList) println(n)
1
2
3
4
5
```

where you need to omit the keyword `yield`. You can also do more elaborate calculations such as

```
scala> for (n <- (1 to 5).toList) {
  val square_n = n * n
  println(s"$n * $n = $square_n")
}
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
```

In this code I use a string interpolation, written `s"..."` in order to print out an equation. This string interpolation allows me to refer to the integer values `n` and `square_n` inside a string. This is very convenient for printing out "things".

The corresponding map construction for functions with side-effexts is in Scala called `foreach`. So you could also write

```
scala> (1 to 5).toList.foreach(n => println(n))
1
2
3
4
5
```

or even just

```
scala> (1 to 5).toList.foreach(println)
1
2
3
4
5
```

If you want to find out more maps and functions with side-efffects, you can ponder about the response Scala gives if you replace `foreach` by `map` in the expression above. Scala will still allow `map`, but then reacts with an interesting result.

Types

In most functional programming languages types play an important role. Scala is such a language. You have already seen built-in types, like `Int`, `Boolean`, `String` and `BigInt`, but also user-defined ones, like `Rexp`. Unfortunately, types can be a thorny subject, especially in Scala. For example, why do we need to give the type to `toSet[Int]` but not to `toList`? The reason is the power of Scala,

which sometimes means it cannot infer all necessary typing information. At the beginning while getting familiar with Scala, I recommend a "play-it-by-ear-approach" to types. Fully understanding type-systems, especially compicated ones like in Scala, can take a module on their own.[5]

In Scala, types are needed whenever you define an inductive datatype and also whenever you define functions (their arguments and their results need a type). Base types are types that do not take any (type)arguments, for example `Int` and `String`. Compound types take one or more arguments, which need to be given in angle-brackets, for example `List[Int]` or `Set[List[String]]` or `Map[Int, Int]`.

There are e few special type-constructors. One is for tuples, which is written with parentheses. For example `(Int, Int, String)` for a triple consisting of two integers and a string. Tuples are helpful if you want to define a function with multiple results, say the function returning the quotient and reminder of two numbers. For this you might define:

```
def quo_rem(m: Int, n: Int) : (Int, Int) = (m / n, m % n)
```

Since this function returns a pair of integers, its type needs to be `(Int, Int)`.

Another special type-constructor is for functions, written `=>`. For example, the type `Int => String` stands for a function that takes an integer as argument and produces a string. A function of this type is for instance

```
1 def mk_string(n: Int) : String = n match {
2   case 0 => "zero"
3   case 1 => "one"
4   case 2 => "two"
5   case _ => "many"
6 }
```

Unlike other functional programming languages, there is no easy way to find out the types of existing functions except for looking into the documentation

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

The function arrow can also be iterated, as in `Int => String => Boolean`. This is the type for a function taking an integer as first argument and a string as second, and the result of the function is a boolean. Though silly, a function of this type is

```
1 def chk_string(n: Int, s: String) : Boolean =
2   mk_string(n) == s
```

[5]Still, such a study can be a rewarding training: If you are in the business of designing new programming languages, you will not be able to turn a blind eye to types. They essentially help programmers to avoid common programming errors and help with maintaining code.

Cool Stuff

More Info