A Crash-Course in Scala

The Very Basics

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 a compiler. This helps a lot with interactively developing programs. Once you installed Scala, you can start the interpreter by typing on the command line:

```
$ scala
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 9).
Type in expressions for evaluation. Or try :help.

scala>
```

The precise response may vary depending on the version and 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; `res0` is a name that Scala gives automatically to the result. You can reuse this name later on. Another classic example you can try out is

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

Note that in this case there is no result. The reason is that `print` does not actually produce a result (there is no `resX` and no type), 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 later, but if you are curious now, the latter kind of functions always has `Unit` as return type. It is just not printed.

You can try more examples with the Scala interpreter, but try first to guess what the result is (not all answers by Scala are obvious):

```
scala> 2 + 2
scala> 1 / 2
scala> 1.0 / 2
scala> 1 / 2.0
scala> 1 / 0
scala> 1.0 / 0.0
scala> true == false
scala> true && false
scala> 1 > 1.0
scala> "12345".length
```

Stand-Alone Scala Apps

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

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

save it in a file, for example `hello-world.scala`, and then run the compiler and runtime environment:

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

Like Java, 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
```

You might need to adapt the path to where you have installed Scala.

Values

In the lectures I will try to avoid as much as possible the term *variables* familiar from other programming languages. The reason is that Scala has *values*, which can be seen as abbreviations of larger expressions. For example

```
scala> val x = 42
x: Int = 42

scala> val y = 3 + 4
y: Int = 7

scala> val z = x / y
z: Int = 6
```

Why the kerfuffle about values? Well, values are *immutable*. You cannot change their value after you defined them. If you try to reassign `z` above, Scala will yell at you:

```
scala> z = 9
error: reassignment to val
       z = 9
         ^
```

So it would be a bit absurd to call values as variables...you cannot change them. You might think you can re-assign them like

```
scala> val x = 42
scala> val z = x / 7
scala> val x = 70
scala> println(z)
```

but try to guess what Scala will print out for `z`? Will it be `6` or `10`? A final word about values: Try to stick to the convention that names of values should be lower case, like `x`, `y`, `foo41` and so on.

Function Definitions

We do functional programming. So defining functions will be our main occupation. A function `f` taking a single argument of type `Int` can be defined in Scala as follows:

```
def f(x: Int) : String = EXPR
```

This function returns the value resulting from evaluating the expression `EXPR` (whatever is substituted for this). The result will be of type `String`. It is a good habbit to include this information about the return type always. Simple examples of Scala functions are:

```
def incr(x: Int) : Int = x + 1
def double(x: Int) : Int = x + x
def square(x: Int) : Int = x * x
```

The general scheme for a function is

```
def fname(arg1: ty1, arg2: ty2,..., argn: tyn): rty = {
  BODY
}
```

where each argument requires its type and the result type of the function, `rty`, should be given. If the body of the function is more complex, then it can be enclosed in braces, like above. If it it is just a simple expression, like `x + 1`, you can omit the braces. Very often functions are recursive (call themselves) like the venerable factorial function.

```
def fact(n: Int): Int =
  if (n == 0) 1 else n * fact(n - 1)
```

Loops, or better the Absence thereof

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, let us assume you have a list of numbers from 1 to 8 and want to build the list of squares. The list of numbers from 1 to 8 can be constructed in Scala as follows:

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

Generating from this list, the list of squares in a programming language such as 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.

A map essentially takes a function that describes how each element is transformed (for example squared) and a list over which this function should work. There are two forms to express such maps in Scala. The first way is called a *for-comprehension*. Squaring the numbers from 1 to 8 would look as follows:

```
scala> for (n <- (1 to 8).toList) yield n * n
res2: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64)
```

The important keywords are **`for`** and **`yield`**. This for-comprehension roughly states that from the list of numbers we draw `n`s and compute the result of `n * n`. As you can see, we specified the list where each `n` comes from, namely `(1 to 8).toList`, and how each element needs to be transformed. This can also be expressed in a second way in Scala by using directly `map`s as follows:

```
scala> (1 to 8).toList.map(n => n * n)
res3 = List(1, 4, 9, 16, 25, 36, 49, 64)
```

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 of your lessons about the lambda-calculus where this would

have been written as $\lambda n.\, n * n$. It might not be obvious, but for-comprehensions are just syntactic sugar: when compiling, Scala translates for-comprehensions into equivalent maps. This even works when for-comprehensions get more complicated (see below).

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

```
scala> (1 to 8).toList.map(n => n % 3)
res4 = List(1, 2, 0, 1, 2, 0, 1, 2)
```

If we, however, transform the numbers 1 to 8 not into a list, but into a set, and then compute the reminders modulo 3 we obtain

```
scala> (1 to 8).toSet[Int].map(n => n % 3)
res5 = Set(2, 1, 0)
```

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

```
scala> for (n <- (1 to 8).toSet[Int]) yield n % 3
res5 = Set(2, 1, 0)
```

For-comprehensions can also be nested and the selection of elements can be guarded. For example if we want to pair up the numbers 1 to 4 with the letters a to c, we can write

```
scala> for (n <- (1 to 4).toList;
            m <- ('a' to 'c').toList) yield (n, m)
res6 = List((1,a), (1,b), (1,c), (2,a), (2,b), (2,c),
            (3,a), (3,b), (3,c), (4,a), (4,b), (4,c))
```

Or if we want to find all pairs of numbers between 1 and 3 where the sum is an even number, we can write

```
scala> for (n <- (1 to 3).toList;
            m <- (1 to 3).toList;
            if (n + m) % 2 == 0) yield (n, m)
res7 = List((1,1), (1,3), (2,2), (3,1), (3,3))
```

The `if`-condition in the for-comprehension filters out all pairs where the sum is not even.

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-comprehension needs to be modified. The reason is that `print`,

you guessed it, does not produce any result, but only produces what is in the functional-programming-lingo called a side-effect. Printing out the list of numbers from 1 to 5 would look as follows

```
scala> for (n <- (1 to 5).toList) print(n)
12345
```

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 variable assignment (`val square_n = ...`) and also what is called in Scala a *string interpolation*, written `s"..."`. The latter is for printing out an equation. It 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-effects is in Scala called `foreach`. So you could also write

```
scala> (1 to 5).toList.foreach(n => print(n))
12345
```

or even just

```
scala> (1 to 5).toList.foreach(print)
12345
```

Again I hope this reminds you a bit of your lambda-calculus lessons, where an explanation is given why both forms produce the same result.

If you want to find out more about maps and functions with side-effects, you can ponder about the response Scala gives if you replace `foreach` by `map` in the expression above. Scala will still allow `map` with side-effect functions, but then reacts with a slightly 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 complicated ones like in Scala, can take a module on their own.[1]

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 as seen earlier need to be given in angle-brackets, for example `List[Int]` or `Set[List[String]]` or `Map[Int, Int]`.

There are a few special type-constructors that fall outside this pattern. One is for tuples, where the type is written with parentheses. For example

```
(Int, Int, String)
```

is for a triple (a tuple with three components—two integers and a string). Tuples are helpful if you want to define functions 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 return type needs to be of type `(Int, Int)`. Incidentally, this is also the input type of this function. Notice this function takes *two* arguments, namely `m` and `n`, both of which are integers. They are "packaged" in a pair. Consequently the complete type of `quo_rem` is

```
(Int, Int) => (Int, Int)
```

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

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

It takes an integer as input argument and returns a string. Unlike other functional programming languages, there is in Scala no easy way to find out the types of existing functions, except by 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

[1] 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.

as second, and the result of the function is a boolean. Though silly, a function of this type would be

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

which checks whether the integer `n` corresponds to the name `s` given by the function `mk_string`. Notice the unusual way of specifying the arguments of this function: the arguments are given one after the other, instead of being in a pair (what would be the type of this function then?). This way of specifying the arguments can be useful, for example in situations like this

```
scala> List("one", "two", "three", "many").map(chk_string(2))
res4 = List(false, true, false, false)

scala> List("one", "two", "three", "many").map(chk_string(3))
res5 = List(false, false, false, true)
```

In each case we can give to `map` a specialised version of `chk_string`—once specialised to 2 and once to 3. This kind of "specialising" a function is called *partial application*—we have not yet given to this function all arguments it needs, but only some of them.

Coming back to the type `Int => String => Boolean`. The rule about such function types is that the right-most type specifies what the function returns (a boolean in this case). The types before that specify how many arguments the function expects and what their type is (in this case two arguments, one of type `Int` and another of type `String`). Given this rule, what kind of function has type `(Int => String) => Boolean`? Well, it returns a boolean. More interestingly, though, it only takes a single argument (because of the parentheses). The single argument happens to be another function (taking an integer as input and returning a string). Remember that `mk_string` is just such a function. So how can we use it? For this define the somewhat silly function `apply_3`:

```
def apply_3(f: Int => String): Bool = f(3) == "many"

scala> apply_3(mk_string)
res6 = true
```

You might ask: Apart from silly functions like above, what is the point of having functions as input arguments to other functions? In Java there is indeed no need of this kind of feature: at least in the past it did not allow such constructions. I think, the point of Java 8 is to lift this restriction. But in all functional programming languages, including Scala, it is really essential to allow functions as input argument. Above you already seen `map` and `foreach` which need this. Consider the functions `print` and `println`, which both print out strings, but the latter adds a line break. You can call `foreach` with either of them and thus changing how, for example, five numbers are printed.

```
scala> (1 to 5).toList.foreach(print)
12345
scala> (1 to 5).toList.foreach(println)
1
2
3
4
5
```

This is actually one of the main design principles in functional programming. You have generic functions like `map` and `foreach` that can traverse data containers, like lists or sets. They then take a function to specify what should be done with each element during the traversal. This requires that the generic traversal functions can cope with any kind of function (not just functions that, for example, take as input an integer and produce a string like above). This means we cannot fix the type of the generic traversal functions, but have to keep them *polymorphic*.[2]

There is one more type constructor that is rather special. It is called `Unit`. Recall that `Boolean` has two values, namely `true` and `false`. This can be used, for example, to test something and decide whether the test succeeds or not. In contrast the type `Unit` has only a single value, written `()`. This seems like a completely useless type and return value for a function, but is actually quite useful. It indicates when the function does not return any result. The purpose of these functions is to cause something being written on the screen or written into a file, for example. This is what is called they cause some effect on the side, namely a new content displayed on the screen or some new data in a file. Scala uses the `Unit` type to indicate that a function does not have a result, but potentially causes some side-effect. Typical examples are the printing functions, like `print`.

Cool Stuff

The first wow-moment I had with Scala was when I came across the following code-snippet for reading a web-page.

```
import io.Source
val url = """http://www.inf.kcl.ac.uk/staff/urbanc/"""
Source.fromURL(url)("ISO-8859-1").take(10000).mkString
```

These three lines return a string containing the HTML-code of my webpage. It actually already does something more sophisticated, namely only returns the first 10000 characters of a webpage in case it is too large. Why is that code-snippet of any interest? Well, try implementing reading-from-a-webpage in Java. I also like the possibility of triple-quoting strings, which I have only seen

[2] Another interestic topic about types, but we omit it here for the sake of brevity.

in Scala so far. The idea behind this is that in such a string all characters are interpreted literally—there are no escaped characters, like \n for newlines.

My second wow-moment I had with a feature of Scala that other functional programming languages do not have. This feature is about implicit type conversions. If you have regular expressions and want to use them for language processing you often want to recognise keywords in a language, for example `for`, `if`, `yield` and so on. But the basic regular expression `CHAR` can only recognise a single character. In order to recognise a whole string, like `for`, you have to put many of those together using `SEQ`:

```
SEQ(CHAR('f'), SEQ(CHAR('o'), CHAR('r')))
```

This gets quickly unreadable when the strings and regular expressions get more complicated. In other functional programming languages, you can explicitly write a conversion function that takes a string, say `"for"`, and generates the regular expression above. But then your code is littered with such conversion functions.

In Scala you can do better by "hiding" the conversion functions. The keyword for doing this is `implicit` and it needs a built-in library called

```
scala.language.implicitConversions
```

Consider the code

```
import scala.language.implicitConversions

def charlist2rexp(s: List[Char]) : Rexp = s match {
  case Nil => EMPTY
  case c::Nil => CHAR(c)
  case c::s => SEQ(CHAR(c), charlist2rexp(s))
}

implicit def string2rexp(s: String) : Rexp =
  charlist2rexp(s.toList)
```

where the first seven lines implement a function that given a list of characters generates the corresponding regular expression. In Lines 9 and 10, this function is used for transforming a string into a regular expression. Since the `string2rexp`-function is declared as `implicit`, the effect will be that whenever Scala expects a regular expression, but I only give it a string, it will automatically insert a call to the `string2rexp`-function. I can now write for example

```
scala> ALT("ab", "ac")
res9 = ALT(SEQ(CHAR(a),CHAR(b)),SEQ(CHAR(a),CHAR(c)))
```

Recall that `ALT` expects two regular expressions as arguments, but I only supply two strings. The implicit conversion function will transform the string into a regular expression.

Using implicit definitions, Scala allows me to introduce some further syntactic sugar for regular expressions:

```
implicit def RexpOps(r: Rexp) = new {
  def | (s: Rexp) = ALT(r, s)
  def ~ (s: Rexp) = SEQ(r, s)
  def % = STAR(r)
}

implicit def stringOps(s: String) = new {
  def | (r: Rexp) = ALT(s, r)
  def | (r: String) = ALT(s, r)
  def ~ (r: Rexp) = SEQ(s, r)
  def ~ (r: String) = SEQ(s, r)
  def % = STAR(s)
}
```

This might seem a bit overly complicated, but its effect is that I can now write regular expressions such as $ab + ac$ simply as

```
scala> "ab" | "ac"
res10 = ALT(SEQ(CHAR(a),CHAR(b)),SEQ(CHAR(a),CHAR(c)))
```

I leave you to figure out what the other syntactic sugar in the code above stands for.

One more useful feature of Scala is the ability to define functions with varying argument lists. This is a feature that is already present in old languages, like C, but seems to have been forgotten in the meantime—Java does not have it. In the context of regular expressions this feature comes in handy: Say you are fed up with writing many alternatives as

```
ALT(..., ALT(..., ALT(..., ...)))
```

To make it difficult, you do not know how deep such alternatives are nested. So you need something flexible that can take as many alternatives as needed. In Scala one can achieve this by adding a * to the type of an argument. Consider the code

```
def Alts(rs: List[Rexp]) : Rexp = rs match {
  case Nil => NULL
  case r::Nil => r
  case r::rs => ALT(r, Alts(rs))
}

def ALTS(rs: Rexp*) = Alts(rs.toList)
```

The function in Lines 1 to 5 takes a list of regular expressions and converts it into an appropriate alternative regular expression. In Line 7 there is a wrapper for this function which uses the feature of varying argument lists. The effect of this code is that I can write the regular expression for keywords as

```
ALTS("for", "def", "yield", "implicit", "if", "match", "case")
```

Again I leave it to you to find out how much this simplifies the regular expression in comparison with if I had to write this by hand using only the "plain" regular expressions from the inductive datatype.

More Info

There is much more to Scala than I can possibly describe in this document. Fortunately there are a number of free books about Scala and of course lots of help online. For example

- `http://www.scala-lang.org/docu/files/ScalaByExample.pdf`
- `http://www.scala-lang.org/docu/files/ScalaTutorial.pdf`
- `https://www.youtube.com/user/ShadowofCatron`
- `http://docs.scala-lang.org/tutorials`
- `https://www.scala-exercises.org`

There is also a course at Coursera on Functional Programming Principles in Scala by Martin Odersky, the main developer of the Scala language. And a document that explains Scala for Java programmers

- `http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html`

While I am quite enthusiastic about Scala, I am also happy to admit that it has more than its fair share of faults. The problem seen earlier of having to give an explicit type to `toSet`, but not `toList` is one of them. There are also many "deep" ideas about types in Scala, which even to me as seasoned functional programmer are puzzling. Whilst implicits are great, they can also be a source of great headaches, for example consider the code:

```
scala>  List (1, 2, 3) contains "your mom"
res1: Boolean = false
```

Rather than returning `false`, this code should throw a typing-error. There are also many limitations Scala inherited from the JVM that can be really annoying. For example a fixed stack size. One can work around this particular limitation, but why does one have to? More such 'puzzles' can be found at

`http://scalapuzzlers.com` and `http://latkin.org/blog/2017/05/02/when-the-scala-compiler-doesnt-help/`

Even if Scala has been a success in several high-profile companies, there is also a company (Yammer) that first used Scala in their production code, but then moved away from it. Allegedly they did not like the steep learning curve of Scala and also that new versions of Scala often introduced incompatibilities in old code. In the past two months there have also been two forks of the Scala compiler. It needs to be seen what the future brings for Scala.

So all in all, Scala might not be a great teaching language, but I hope this is mitigated by the fact that I never require you to write any Scala code. You only need to be able to read it. In the coursework you can use any programming language you like. If you want to use Scala for this, then be my guest; if you do not want, stick with the language you are most familiar with.