A Crash-Course on Scala

Scala is a programming language that combines functional and object-oriented programming-styles, and has received in the last five years or so 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 Scala programs can run under MacOSX, Linux and Windows.[1] Unlike Java, however, Scala often allows programmers to write very concise and elegant code. Some therefore say Scala is the much better Java. A number of companies, The Guardian, Twitter, Coursera, LinkedIn to name a few, either use Scala exclusively in production code, or at least to some substantial degree. If you want to try out Scala yourself, the Scala compiler can be downloaded from

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

Why do I use Scala in the AFL module? Actually, you can do *any* part of the 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. If I had to do this in Java, for example, I would first have to run through heaps of boilerplate code. Since the Scala compiler is free, you can download the code-snippets and run every example I give. But if you prefer, you can also easily translate them into any other functional language, for example Haskell, Standard ML, F$^{\#}$, Ocaml and so on.

Developing programs in Scala can be done with the Eclipse IDE and also with IntelliJ IDE, 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, you can start the interpreter by typing on the command line:

```
$ 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

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

```
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 `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 later, 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 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, say `hello-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 define regular expressions simply by giving the grammar

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

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] and then compare it with how it can be defined in Scala.

Implementing the regular expressions from above in Scala is actually very simple: It first requires an *abstract class*, say, `Rexp`. This will act as the type for regular expressions. Second, it requires a case for each clause in the grammar. 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 Scala 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 of inductive datatypes, feel free to define for example binary trees.

Once you make a definition like the one above in Scala, you can represent, for example, the regular expression for $a + b$ as `ALT(CHAR('a'), CHAR('b'))`. Expressions such as `'a'` stand for ASCII characters, though in the output syntax the quotes are omitted. 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 too. But we omit such "tricks" 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 our definition the type `Rexp` is more general than **`ALT`**, since it is the abstract class. But 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! ☺

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 first common type is `Rexp`.[3]

For compound types like `List[...]`, the general rule is that when a type takes another type as argument, then this argument type 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 in addition to inductive datatypes, also functions can be implemented very easily in Scala. To show you this, 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:

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

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 in Scala for arbitrary precision integers (in case you want to try out the function on really big numbers). 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.

[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`.

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.

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 Java, for example, which does not have pattern-matching, the resulting code would be just awkward.

Mathematicians already use the power of pattern-matching 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:

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

This function is defined by recursion analysing each pattern of what the regular expression could look like. 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 keyword can be another pattern, but often as in the case above, it is just a variable you want to pattern-match (the `r` after `=` in Line 1).

Each case in this definition 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 $ba + ca$, 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. In cases like `rev` it is really crucial. 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, or the Absence of

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 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 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 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 10).toList, and how each element needs to be transformed. This can also be expressed in a second way in Scala by using directly maps 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 ns 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 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 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 10).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;
            c <- ('a' to 'c').toList) yield (n, c)
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 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) 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 variable assignment (`val square_n = ...`) and what is called a *string interpolation*, written `s"..."`, in order to print out an equation. The 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-effects 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
```

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.[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 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)` for a triple consisting of 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 type needs to be `(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 argument and produces a string. A function of this type is for instance

```
def mk_string(n: Int) : String = n match {
  case 0 => "zero"
  case 1 => "one"
```

[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.

```
    case 2 => "two"
    case _ => "many"
  }
```

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 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`.

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 is their type (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).

Now you might ask, what is the point of having function as arguments to other functions? In Java there is no need of this kind of feature. But in all functional programming languages, including Scala, it is really essential. 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*.[6]

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 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).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 a "webpage" 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 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 language, you can explicitly

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

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 function.

In Scala you can do better by "hiding" the conversion functions. The keyword for doing this is **implicit**. Consider the code

```
1  import scala.language.implicitConversions
2
3  def charlist2rexp(s: List[Char]) : Rexp = s match {
4    case Nil => EMPTY
5    case c::Nil => CHAR(c)
6    case c::s => SEQ(CHAR(c), charlist2rexp(s))
7  }
8
9  implicit def string2rexp(s: String) : Rexp =
10   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 = ALT(SEQ(CHAR(a),CHAR(b)),SEQ(CHAR(a),CHAR(c)))
```

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 $ab + ac$ even simpler as

```
scala> "ab" | "ac"
res10: ALT = 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 variable 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

```
1 def Alts(rs: List[Rexp]) : Rexp = rs match {
2   case Nil => NULL
3   case r::Nil => r
4   case r::rs => ALT(r, Alts(rs))
5 }
6
7 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 you to it to find out how much this simplifies the regular expression in comparison 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

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.

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.

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.