COMS W3137 - Recitation Notes
Week 2: Scala Basics

Daniel Bauer

February 2, 2019

Scala `http://www.scala-lang.org` is a rich multi-paradigm language. The default implementation of Scala runs on the Java Virtual Machine. Scala supports both functional and imperative object oriented programming. The language aims to support both small-scale and large-scale development (the name *Scala* is short for Scalable Language).

1 Running Scala

There are three ways to run Scala code: By using the interactive Read Evaluate Print Loop (REPL), by running a Scala script which is evaluated in the REPL line-by-line, and by compiling scala to byte code and then running the byte code. The last option is identical to how you would compile and run a Java program. The REPL is a nice way to get acquinted with the Scala langauge and quickly try out some of the concepts we discuss in this section.

1.1 Read Evaluate Print Loop

To open the Read Evaluate Print Loop (REPL), run the command `scala` from the command line.

```
$ scala
Welcome to Scala 2.12.8 (OpenJDK 64-Bit Server VM, Java 11.0.2).
Type in expressions for evaluation. Or try :help.

scala>
```

You can now type any expression. Scala wille evaluate the expression, and then print the reuslt. To quit the REPL, type `:quit`.

```
scala> 40 + 2
res0: Int = 42

scala> "Hello"
res1: String = Hello
```

Note the output `res0: Int = 42`. `res0` is a name (specifically a `val`) that you can use to refer back to the value 42.

```
scala> res0
res2: Int = 42
```

Note how Scala creates a new name to refer to the result of each new expression. `res0 : Int` means that the data type of this name is integer (`Int`) and `res1 : String` means that `res1` is of type string. We will say a lot more about data types and names below.
We can also define our own names

```
scala> val answer : Int = 40 + 2
answer: Int = 42

scala> answer + 8
res3: Int = 50
```

To quit the REPL, type `:quit`.

```
scala> :quit
$
```

1.2 Scala Scripts

To run a script, we just place our Scala code in a file and then pass it as a command line parameter to the `scala` program.
Assume that we place the following program in the file `hello_script.scala`.

```
val answer : Int = 40 + 2
println("Hello World")
println(answer)
```

More flexible since don't need to create a class, can simply be expressions without a 'container'

We can now run the program on the console

```
$ scala -nc hello_script.scala
Hello World
42
$
```

The `-nc` flag turns off the offline compiler, which would normally attempt to compile the program into bytecode first. `scala` interprets the program line-by-line, just as if you had typed it into the REPL. Unlike in the REPL, the result of each expression is not printed automatically and we need to use the `println` built-in function instead. This function prints a string, followed by a line break.

Scala scripts can be used as a replacement for other scripting languages, like Bash or Perl.

1.3 Compiling to Bytecode

Typically, to run a full Scala program, you first compile the program to bytecode. This is similar to how you would run a program in Java.
Assume that we place the following program in the file `Hello.scala`.

```
object Hello {
    def main(args : Array[String]) {
        println ("Hello World");
    }
}
```

We use the `scalac` command to compile the program into byte code. This will produce two class files, including `Hello.class`. You can then run this class using the `scala` command.

```
$ ls
Hello$.class
Hello.class
Hello.scala
$ scala Hello
Hello World
$
```

The Scala script from the previous section executes line-by-line. When we compile a program to byte-code, this will no longer work. Just like in Java, Scala programs consist of individual classes and other objects. Therefore, every program you write needs to have at least a class or object definition (we will get back to differences between classes and objects later). Trying to compile `scala-script.scala` results in a number of errors:

```
$ scalac hello_script.scala
hello.scala:1: error: expected class or object definition
val answer : Int = 40 + 2
^
hello.scala:2: error: expected class or object definition
print("Hello World")
^
hello.scala:3: error: expected class or object definition
print(answer)
^
three errors found
```

A program will often consist of multiple .scala files. To compile all scala files in the local directory at once, you can use `scalac *.scala`.

In Java, public classes need to be stored in a file of the same name. Scala has no such restriction. We could have called the `.scala` file in the previous example anything we like. In fact, one file may contain any number of classes and objects. It is strongly recommended, however, that you break down Scala programs into individual units, like you would a Java program. Unless there is a good reason (for example a class belongs logically to another class – a situation commonly encountered when constructing data structures), a file should contain exactly one class (Some exception to the one-class rule will be discussed below). This style benefits readability and performance. If you change the code for one class, only this one class needs to be recompiled. Compilation time can quickly become an issue once your programs become larger. As you develop and debug your programs you will want to recompile frequently. Another way to speed up the compilation process is to use the fast scala compiler by simply using the `fsc` command instead of the `scalac` command. The first time you compile a program with `fsc`, a demon will start and continue running in the background. Every subsequent call to `fsc` will cause the demon to recompile the program, saving some of the startup time that would be required with the `scalac` command.
Finally, for larger projects that include package dependencies, I strongly recommend that you consider using a build tool, such as the Scala Build Tool *sbt*[1].

2 Variables and Basic Data Types

2.1 `var` and `val`

Scala has two types of variables, called `var` and `val` after the key word that is used to declare them.[2] The difference between them is that, while `vars` can be reassigned, `vals` are set to an initial value that can never be changed later in the program. In other words, `vars` behave like Java variables. You can think of `vals` as names for the result of some intermediate computation. `vals` are **immutable** – their value cannot be changed (mutated), while `vars` are **mutable**. Attempting to reassign a `val` results in an error.

[1] http://www.scala-sbt.org
[2] There is a third type, called a `lazy val` which forces lazy evaluation of an expression.

```
scala> var y : Int = 5
y : Int = 5

scala> y = 23
y : Int = 23

scala> val x : Int = 24
x: Int = 24

scala> x = 23
<console>:4: error: reassignment to val
       x = 23
         ^
```

When a variable is declared, the data type of the variable is indicated after the variable name, separated by a `:`. This is different from Java, where the type preceeds the variable name (and in Java ther eis no `var` or `val` keyword). For example, in Java you would declare a variable like this: `int y = 5`.
Unless specified, a new variable is a `var` by default.

```
scala> y : Int = 10
y : Int = 10
scala> y = 5
y : Int = 5
```

Because we will be mostly concerned with functional program and avoid imperative programing, we will mostly use `val`s.

2.2 Basic Data Types

Each variable has a data type. Scala, like Java, is statically typed, which means that all data types are already known at compile time, before the program runs. This helps prevent runtime errors. No further type checking is needed at runtime. Once declared, the type of a variable may never change. This is true even for `var`s: their value may change, but their type is fixed. In Java, data types typically need to be defined explicitly by the programmer. Scala performs *type inference* to determine types automatically at compile time as much as possible. For example, when a variable is initialized with an integer, Scala infers that this variable should be of type Int, even though we did not specifically set the type. When type inference is not successful and Scala cannot determine the type, compilation fails. We will discuss type inference in more detail later.

```
scala> val x = 23
x : Int = 23
```

There are no primitive data types in Scala

Scala Class	Description	Java type	Java Class
Byte	8-bit signed integer	byte	Byte
Short	16-bit signed integer	short	Short
Int	32-bit signed integer	int	Integer
Long	64-bit signed integer	long	Long
Char	16bit Unicode character (unsigned)	char	Character
Float	32-bit single precision float	float	Float
Double	64-bit single precision float	double	Double
String	a sequence of Chars	n/a	String
Boolean	a boolean (either `true` of `false`)	bool	Boolean

Table 1: Basic data types in Java and Scala. **Java type** indicates the corresponding basic data type in Java. **Java class** indicates the corresponding Java wrapper class.

Java has a number of built-in basic data types, such as *int*, *char*,and *bool*. Values of these type are not class instances in Java. But in some situations, class instances are expected, for example when instantiating generic data structures (discussed below). For this purpose, Java provides wrapper classes such as `Integer`, `Character` and `Boolean` in the package `java.lang`. Because Scala runs on the Java Virtual Machine, it supports the same data types, but unlike Java, Scala has no built-in basic data types. Instead all Scala types are classes. All values are instances of some class, similar to the wrapper classes in Java. Table 1 provides an overview of basic data types and the corresponding Java class. These basic Scala classes are in the package `scala`.
All Java wrapper classes can also be used in Scala. In fact, any Java class can be instantiated in Scala, allowing seamless interoperability. You can use existing Java libraries in Scala and use Scala to extend existing Java programs. When possible, it is recommended that you stick with the Scala classes.

```
scala> val z : java.lang.Integer = 7
y: Integer = 7
```

3 Everything is an Expression

An expression is anything that evaluates to a value.

```
scala > 25 + 5 // This is an expression
Res0 : Int = 30
```

A statement is a piece of code that has an effect on the state of the program, but that does not result in a value. For example, in a language like Java variable

declarations, `if` conditions, loops, and output commands are statements. Scala has no true statements. Almost everything is an expression.
Consider, for example, simple output using `println`

```
scala > val z = println("Hello")
Hello
z : Unit = ()
```

Unit = () is pretty much null in Java

`Unit` is a special type that has only one value. It's special purpose is to provide a value when an expression would not return anything otherwise. This is commonly encountered when calling methods on mutable objects, which change the state of the object but do not return a usable value. As mentioned before, in functional programing we wish to avoid this style and we will therefore not encounter many situations in which `Unit` is being returned. Note that Scala's use of `Unit` is somewhat more consistent than what Java provides. In Java, methods that do not return a value can be declared `void` (in place of a return data type). Alternatively, a Java methods that has a return data type, can return `null`. This however is considered bad style and can result in null pointer exceptions at runtime.

3.1 Compound Expressions

Every block of code is a *compound* expression that evalutes to the value of the last expression in the block.

```
scala> val z = {
     |   val x= 4;
     |   val y = 2;
     |   x/y}
z: Int = 2
```

x is used only in the code block, doesn't exist outside code block

In the previous example, the entire block in curly brackets evaluates to 2 because the last expression in the block evaluates to 2. What happens if the block only contains variable declarations?

```
scala> val z = { val x= 4; }
z: Unit = ()
```

3.2 Arithmetic Expressions and Casting

In Scala everything is an expression. Scala can automatically convert values to a more expressiv type (up-casting). When performing arithmetic operations that involve multiple types, the result will be of the more expessive type. Note that `Char` is actually a numeric data type. For example, the value 115 corresponds to the character *s*.

```
scala > val x = 's';
x : Char = s
scala > val y : Int = x;
y: Int = 115
scala > 16 - x
res1: Int = -99
```

Scala does not permit automatic downcasting. The following will result in an exception.

```
scala> val x : Double = 25.5
x: Double = 25.5

scala> val y : Int = x
<console>:12: error: type mismatch;
 found   : Double
 required: Int
       val y : Int = x
```

Instead, to convert the double to an integer, rounding down, we can call the `toInt` method on the `Double` instance. Methods that do not have parameters must be called without () parenthesis in Scala.

```
scala> val y : Int = x.toInt
y: Int = 25
```

4 Control Structures

4.1 Booleans and Comparisons

Comparisons in Scala work the same way as in Java. Comperators like `==` and `>` are used to create expressions that evaluated to an object of type `Boolean`. The Boolean type only has two values: `true` and `false`. A boolean expression is any expression that evaluates to a `Boolean`.

```
scala> true
res1: Boolean = true

scala> val x = 5
x : Int = 5

scala> x < 3+2
res3 : Boolean = false

scala> val y = "Hello"
y : String = Hello

scala> y == "Hello"
res4 : Boolean = true
```

Can compare all data types with ==

4.2 Conditionals

At first glance conditionals work almost as in Java, using the `if` and `else` keywords, followed by a boolean expression in parentheses.

```
val a = 7
if ( a % 2 == 0) {
    println ( " a is even " )
} else if ( a % 3 == 0) {
    println ( " a is a multiple of 3 " )
} else println ( " none of the above " )
```

`a % b` is the *modulo* operator that returns the remainder of dividing integer a by integer b.
Unlike in Java however, conditionals are expressions too. It is therefore incorrect to talk about `if`-*statements* in Scala.

```
val result : String = if ( a % 2 == 0) {
                        "even"
                      } else if (a % 3 == 0) {
                        "multiple of 3"
                      } else {
                        "neither even nor multiple of 3"
                      }
```

val result will equal the last expression in corresponding if block,
in this case, val result = “neither even not multiple of 3”; no need for return statement

The conditional expression evaluates to one of the three strings. Note that the brackets are optional. They are used to define blocks, but each block contains only one expression in this case. In many cases, conditional expressions are very short and the brackets can be ommitted entirely. For example, to compute the absolute value of `a`:

```
val abs_a = if (a > 0) a else -a
```

Treating conditionals as expressions encourages a functional programming style. The focus is on *computing the values* and then continuing the computation with that value, not on making an explicit update to some mutable variable. Think about the following: If we restrict ourselves to using only `vals` then a language in which conditionals are statements would not be very useful.
Note that Java (and other C-like languages) allow you to write similar expressions, using special notation called the *ternary operator*. In Java, the previous example would be

```
int abs_a = a > 0 ? a : -a;   // Java ternary operator example
```

If the condition to the left of the `?` is true, then return the value to the left of the `:` otherwise return the value to the right of the `:`.

4.3 Loops

Scala, like Java, has two types of loops, `while` loops and `for` loops. While loops tend to encourage an imperative (i.e. not functional) programming style. The loop is executed as long as the condition is true. But to ever make the condition false, we would have to update the variables involved in the condition, which is exactly what we are trying to avoid in the functional paradigm.
The syntax for while loops is the same as in Java.
As an example, we will use the *Collatz Conjecture*. Consider the following steps. Start with any integer number $n > 1$. If n is even, divide n by two. If n is odd, multiply n by three and add one. The Collatz Conjecture states that after a finite number of steps, n becomes 1. An obvious question to ask is how many steps it takes to reach 1, starting at a given integer n. For example, starting at 23, we would go through the following sequence of numbers: 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

```
var n : Int = 23
var count : Int = 0

println (n)
while (n != 1) {
    count += 1
    if (n % 2 == 0)
        n = n / 2
    else
        n = 3 * n + 1
}
println("Needed " +count.toString + " steps.")
```

The sequence above has 15 steps (exlusive the initial number). A while construction always evaluates to `Unit`, another sign that such structures are to be avoided. Section 5 below shows how to compute the number of steps in a Collatz sequence using recursion.

`for`-loops simplify the task of iterating through the elements of an `Iterable`. A typical type of iterable are sequence data structures, such as lists.

5 Functions

Functions are, as the name suggests, at the core of functional programming. In Scala, functions are first-class citizens. They can be defined anywhere and need not be attached to a class. Functions are in fact objects that can be passed around like other objects. This makes it possible to define *higher-order functions*, which take other functions as arguments. We will discuss these techniques in more detail in the coming weeks.
In Java, methods musts be attached to a class (we will use the term *method* to refer to a function attached to a class instance and the term *function* to refer to functions that do not belong to an instance). A method that does not directly operate on the data fields of a class instance can be declared `static`, but it still belongs to a class. As of Java 8, Java does support anonymous first-order functions (*lambdas*), but it has no support for regular, named first-order functions. Scala also supports *lambdas*. We will discuss how to use these later.
In Scala, a function or method is defined using the `def` keyword. The following example function returns the larger of two values.

```
scala> def max(x : Int, y : Int) : Int = if (x > y) x else y
max: (x: Int, y: Int)Int
```

Each function has zero or more parameters and exactly one return value. The type of the return value, `: Int` in the example, is specified after the parameter list, before the `=`. Each parameter has its own type definition, which is written in the same way as for regular variables. Also note the `=` symbol in front of the function body. The function body can be any expression. In the previous example, the expression is just a simple *if*, so the entire function can be written in one line. More frequently, the body will be a compound expression. Unlike a Java method, Scala functions often do not have explicit `return` statements. The return value of the function is simply whatever the body evaluates to.

5.1 Recursion

As mentioned before, while loops not support functional programming. Instead, in functional programing we typically replace iteration with recursion.
In its simplest form, a recursive function calls itself. The behavior of the recursive function depends on its arguments. Every recursive function needs a base-case, in which no further recursive calls are issued. Each recursive call needs to make progress towards the base case.
For example, we can rewrite the *Collatz* computation from the previous section using a recursive function. The function computes the number of steps the program takes, starting from n, before it reaches 1.

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

In converting iteration to recursion, it is often helpful to start with the base case. The base case can often be derived from the condition that would have terminated the `while` loop (the while loop continues as long as $n > 1$, the recursion terminates once $n == 1$). The arguments passed to each recursive call correspond to the update made to the loop variable inside the loop. The trickiest part ist often how to return the result of a computation as the program returns from the recursive calls. In the `collatz` function, we are interested in counting the number of recursive calls. If the function is called with n=1, no recursive calls are issued. In all other cases, we first issue the recursive call for the next number in the collatz sequence, computing the steps it takes to reach 1 from this point, and then add 1.

Tail recursion is a special type of recursion in which the recursive call is the last expression evaluated before the function returns. Most functional programming languages, including Scala, support tail recursion optimizations. The compiler automatically detects tail recursion and transforms it into a loop. The `Collatz` function in the previous example is *not* tail recursive. After the recursive call, the function still performs an addition. We can rewrite the program as follows:

```
def collatzRec(n : Int, count : Int) : Int = {
     if (n==1)
        count
     else
        if (n%2==0)
            collatzRec(n/2, count + 1)
        else
            collatzRec(3*n+1, count + 1)
}

def collatz(n : Int) : Int = {
    collatzRec(n,0)
}
```

Unfortunately not all instances of recursion can be rewritten as tail recursion.

Exercise: Write a recursive scala function that converts decimal integers into a binary representation. The result should be a String. The function should have the signature `decToBin(n : Int) : String`. Calling `decToBin(9)` should return a string `"1001"`. Try to write the function as a one-liner. Then try to rewrite your function so that is uses only tail recursion.