CSE3BDC/CSE5BDC
Lab 04: Programming in Scala

Department of Computer Science and IT, La Trobe University

Submissions close 1pm Monday 4th of May

Scala is a programming language which runs on the Java Virtual Machine. It blends the functional and object-oriented paradigms, so in some ways it is like a cross between Lisp and Java. Scala is used for developing applications with Spark, which is an alternative to MapReduce that you will learn a lot more about in the coming weeks. The goal of this lab is to give you a solid grounding in Scala which will be of enormous benefit when you learn Spark in the next two labs.

In this lab there are a lot of exercises where you just type examples straight into the shell with no modifications. This may seem too easy. But please do not skip any of these, since they are all designed to illustrate a different important idea. Read all the explanations surrounding the examples carefully. We are giving you the full example to speedup your learning. If you had to lookup everything from scratch then we would not be able to fit nearly as much in one lab. At the end of each task there are exercises where you need to apply what you have learnt to write your own program. There are 6 exercises for you to do by yourself, so don't worry, you will get plenty of practice.

At the end of the lab I hope you are used to typing in small scala code examples to test different features out. In the future when you have a question about a particular feature, hopefully you will jump straight into Scala/Spark shell and just experiment with small examples like we are doing in this lab. It really is the best way to learn and often is faster than trying to find examples on the internet that exactly matches your requirements. As usual, there's an accompanying video available in ECHO360, so watch that first for an overview of the lab.

Task A: Working with variables and constants

You know what variables and constants are, but you're not really familiar with Scala. You figure that it's pretty important to be able to use variables in most programming languages so you might as well start here.

1. The Spark shell fully supports Scala since Spark itself is written in Scala. So for this lab we will use the Spark shell to run Scala. However, all the exercises in this lab should run equally well in a Scala shell. To run the Spark shell, open the ClouxLab web console and type spark-shell

2. The Spark shell is wonderful for learning both Scala and Spark. However, it is important for you to keep a record of what you have typed into the shell so that you can refer back to them later. To do so, open the provided file called `lab04.scala` in Jupyter, and enter your

answers to the exercises where instructed. There is space at the bottom of this file for you to store any Scala commands that you tested and felt are worth remembering - keep your exercise answers neat so we can mark them!
For more complex programs that take multiple lines, first write the program in the text file and then copy and paste it into the Spark shell to test. If you make a mistake you can easily correct the text file and then cut and paste again. A useful key binding is Ctrl+L, which clears the screen.

3. There are clearly denoted exercises sprinkled throughout the lab. For these tasks you must store your code inside the `lab04.scala` file where instructed so that we can check it later.

4. For all code examples in this lab please do not cut-and-paste. Please type it in yourself, since some characters do not transfer well when you cut-and-paste.

5. In Scala constants are declared with the `val` keyword. Due to the functional programming aspect of Scala it is preferable to use constants wherever possible. Enter the following commands and see what happens.

   ```
   scala> val my_string = "Hello"
   scala> my_string = "Bye"
   scala> print(my_string)
   ```

   You should notice that it is not possible to reassign the value of `my_string` due to it being a constant.

6. There are situations when we want to use actual variables. This is achieved using the `var` keyword, which enables us to change the value of the variable, unlike `val`.

   ```
   scala> var my_string = "Hello"
   scala> my_string = "Bye"
   scala> print(my_string)
   ```

7. Although you can't see types (for example, `String`) in the code, Scala is still statically typed. This means that you can only store one type of data in a particular variable. For convenience, Scala automatically infers the types of variables instead of requiring the programmer to specify them explicitly like in Java. To illustrate static typing, run the following code and observe what happens.

   ```
   scala> var x = "Hello"
   scala> x = 5
   scala> print(x)
   ```

8. Alright, now let's get a bit tricky and try the following code.

```
scala> var x = 0
scala> x = 9876543210L
scala> print(x)
```

Oh no! The problem here is that when we declare x Scala assumes that it is an Int, because by default 0 has type Int. When we later try to store a Long in x things explode because we can't fit a Long in an Int-type variable - this is very much a square peg in a round hole situation. We can fix this particular problem by explicitly telling Scala the variable type when we perform the declaration.

```
scala> var x:Long = 0
scala> x = 9876543210L
scala> print(x)
```

Exercise 1. Declare a variable, initially setting it's value to your name (as a String). Now try changing the same variable to your age (as an Int). What happens and why?

Task B: Using collections and lambdas

Did you know that the collective noun for llamas is a "herd"?

1. A lambda function is a function that can be defined inline without a name. The syntax for lambda functions in Scala is designed to be very concise. For example, a lambda which describes how to double a number can be written as `x => x * 2`. The part to the left of the "=>" describes the arguments, and the part to the right describes how the return value is calculated.

2. The main use for lambda functions is as arguments to other functions. As an example, consider the case where we wish to double the elements in a list of numbers.

   ```
   scala> val numbers = List(2, 13, 7, 22)
   scala> val doubled = numbers.map(x => x * 2)
   scala> print(doubled)
   ```

   The map method creates a new list by applying the provided function to each item in the original list. In this case the provided function is the lambda `x => x * 2`, which describes how to double an element of the list.

3. Scala also has a shorthand for writing simple lambdas like `x => x * 2` using the underscore ("_") character which allows you to skip naming the parameter `x`. Try the following—you should get the same result as before.

   ```
   scala> numbers.map(_ * 2)
   ```

4. Lambdas can have multiple arguments. Consider the reduce method, which takes all of the items in a collection and combines them to produce a single result. We can use this method to sum up the numbers from 1 to 10 as follows.

   ```
   scala> (1 to 10).reduce((x, y) => x + y)
   ```

   The `reduce` function takes two values and then combines them into one value using the function supplied. This is applied recursively until the entire collection is reduced to just one value. The `(1 to 10)` is actually a function call that returns a collection called a range containing the integers 1 to 10. Try typing just `1 to 10` into the shell and see what happens.

5. It is still possible to use the underscore notation when there are multiple arguments.

   ```
   scala> (1 to 10).reduce(_ + _)
   ```

6. Here is another very useful function, called filter. In this example we are only keeping the numbers above 5.

   ```
   scala> val numbers = List(2, -1, 3, 2, 100, 13, 7, 22)
   scala> numbers.filter(_ > 5)
   ```

Task C: Working with tuples

Tuples. Tuh-ples. Hehe.

1. A tuple is like a record with unnamed fields, where elements are identified by their position in the tuple. For example, we can store a person's name, age and vegetarian status as a three-element tuple.

   ```
   scala> val person = ("Joe Citizen", 23, true)
   ```

2. Now we know that creating tuples is fairly straightforward, but how can we retrieve the values? The syntax is a little bit strange, but the first field is obtained using `tuple._1`, the second with `tuple._2` and so forth.

   ```
   scala> val tuple = ("1", 2, '3')
   scala> println(tuple._1)
   scala> println(tuple._3)
   ```

   Exercise 2. Time for a little exercise. There's a method called "zip" which combines two lists into one list where each element is a pair of values in the form of a tuple. Here's an example.

```
scala> print(List(1, 2, 3).zip(List(4, 5, 6)))
List((1,4), (2,5), (3,6))
```

Using this method and other information from the lab write code which calculates the dot product of two vectors. In case you've forgotten,

$$u \cdot v = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$$

Use the following template to test your code out.

```
scala> val u = List(2, 5, 3)
scala> val v = List(4, 3, 7)
scala> val dot = ...TODO...
scala> print(dot)
44
```

Task D: Working with Arrays and Lists

In this task we will go over some basic operations with Arrays and Lists.

1. Accessing an element of an array in Scala is done by using function call syntax. Hence we use the `()` instead of the usual `[]` to index into an array. Arrays in Scala are mutable, which means you can change the values inside an array. Try the following:

```
scala> val numbers = Array(2, 13, 7, 22)
scala> val x = numbers(0)
scala> val y = numbers(2)
scala> numbers(2) = 10
scala> for(i <- numbers) println(i)
```

2. Unlike arrays, lists in Scala are immutable by default. This means a new list is created every time you add something into a list. The following shows how you can add an element to the head of a list and also how to concatenate two lists together. Although adding something to the head of list creates a new list. This operation can be done in constant time, since it just creates a new head element and then link the old list to it.

```
scala> var letters = List("A", "B", "C", "D")
scala> letters = "E" :: letters
scala> val animals = List("Cat", "Dog", "Shark", "Elephant")
scala> letters = letters ::: animals
```

Task E: Creating named functions

Even though you dove head-first into the world of Scala's lambdas, collections and tuples you can still recall something about putting chunks of code in bizarre constructs called "functions". You wonder to yourself how you might do the same in Scala...

1. Let's first consider procedures, which are functions which do not return anything. In Scala procedures are implemented as follows

   ```
   def greet(name:String) {
     println("Hello " + name + "!")
   }
   ```

   Functions in Scala are called in a similar way to C and Java. Enter the `greet` procedure into a Scala shell and call it with your name as its parameter.

2. Now let's use the `greet` function to print a list of names as follows.

   ```
   val names = List("Peter", "John", "Mary", "Henry")
   names.foreach(greet(_))
   ```

3. In practice most of the functions we write do involve returning a value, particularly in a functional programming language like Scala. Here's an example of a function which multiplies two integers together

   ```
   def multiply(x:Int, y:Int) : Int = {
     x * y
   }
   ```

   Note that we must specify the return type of the function, it cannot be inferred. Furthermore, it is not necessary to use the `return` keyword as Scala will automatically assume that you are returning the last thing in the function body. However, you can use the `return` keyword if you wish (try it and see).

4. When you call a function that takes 2 arguments (e.g. multiply(.., ..)) with one argument specified and the other not specified (e.g. multiply(_ :Int, 3)) scala returns a new function that takes just one argument (the unspecified argument). Let's see how this works in practice. Try the code below and see what you get.

   ```
   scala> val numbers = 1 to 10
   scala> val largerNumbers = numbers.map(multiply(_, 3))
   ```

   Exercise 3. Write a function called `extract` `that` takes two arguments. The first argument is a string called `word` and the second is a integer called `n`. The function returns the *nth* character of word if n is less than the length of the word. If n is greater or equal to the length of the word then the function returns the character '-'. Hint: if statements in Scala are constructed just like Java if statements. Test it using the code below.

```
scala> val names = List("Peter", "John", "Mary", "Henry")
scala> val result = names.map(extract(_,4))
scala> print(result)
List(r, -, -, y)
```

Task F: Using pattern matching

If Java's switch statement went to the gym more often and consumed lots of protein powder it might be half as powerful as Scala's case statement.

1. Scala's case statement can perform all of the tasks you expect from Java's switch statement. The following code demonstrates how to branch based on an integer value.

```
scala> val x = 2
scala> x match {
     | case 1 => print("one")
     | case 2 => print("two")
     | case 3 => print("three")
     | }
```

2. The real power of Scala's pattern matching comes from it's ability to work with any data type including tuples.

```
scala> val thing:Any = ("Jim", 21)
scala> thing match {
     | case (name, age:Int) if age < 18 => print(name + " is young")
     | case (name, 18) => print(name + " is eighteen")
     | case (name, age) => print(name + " is an adult")
     | case _ => print("This isn't a person!")
     | }
```

Try running the pattern matching on different values. Is it possible for multiple cases to be run for the same value?

3. Pattern matching can be useful when dealing with collections, as we can easily classify elements as we iterate over the collection. For example, try the following code and convince yourself that it behaves as expected.

```
scala> val numbers = List(1, 4, 3, 7, 5, 9)
scala> val mapped = numbers.map(_ match {
     | case 5 => 0
     | case x if x < 5 => -1
     | case x if x > 5 => 1
     | })
scala> print(mapped)
```

Task G: Putting it all together

You think you know a bit about Scala, but you're not sure. Better do some more exercises to prove to yourself that you are truly a Scala-master!

Exercise 4. Write a Scala function called `find_max` which takes a single argument of type `Seq[Int]` and prints out the maximum value in the sequence. Using `Seq` allows the function to take many different types of collections because `Seq` is a base type that `Lists`, `Vectors`, `Arrays`, `Ranges` and so forth all inherit from, and the `[Int]` signifies that only sequences of integers are allowed in this function. Use the following template to test your program. Hint: the function body just needs to be 1 line of code. The `reduce` and `Math.max` functions might be useful.

```
scala> val numberList = List(4, 7, 2, 1)
scala> find_max(numberList)
7
scala> val numberArray = Array(4, 7, 2, 1)
scala> find_max(numberArray)
7
```

Exercise 5. Write a Scala function called `matchedNumbers` which accepts two sequences of integers as parameters. The function should return a list containing items that appear at the same position in both sequences. Use the following template to test your code. Hint: you'll need to use the `zip`, `filter`, and `map` functions.

```
scala> val list1 = List(1, 2, 3, 10)
scala> val list2 = List(3, 2, 1, 10)
scala> val ma = matchedNumbers(list1, list2)
(2, 10)
```

Task H: Bonus exercise

Despite the fact that the exercises in this section are not required to receive full marks for the lab, you are so keen to learn more that you give them a go anyway! Or not, that's on you.

Exercise 6. Write a Scala function called `eligibility` which determines whether someone is allowed entry to the Boy Scouts. Valid members of the Boy Scouts must be male and under 13 years old[1]. People's details are stored as tuples of the form (<name>, <age>, [“male”|“female”]). Your function should accept one such tuple as its argument and return a `Boolean` signifying whether the person should be granted entry. If the person is denied entry the function should print a message explaining why (eg. “Mary is not male”). Use the following template to test your program.

```
scala> val people = List(("Harry", 15, "male"), ("Peter", 10, "male"),
                         ("Michele", 20, "female"), ("Bruce", 12, "male"),
                         ("Mary", 2, "female"), ("Max", 22, "male"))
scala> val allowedEntry = people.filter(eligibility(_))
Harry is too old
Michele is too old and not male
Mary is not male
Max is too old
scala> print(allowedEntry)
List((Peter,10,male), (Bruce,12,male))
```

[1]These criteria are invented for the exercise, and don't reflect real Boy Scouts policies.