## LAB 2: STARTING TO PROGRAM FUNCTIONALLY WITH SCALA

In this lab you will practice working with Scala applying functional thinking. You can use whichever of the tools introduced in Lab 1 you prefer to attempt the following exercises. You should use your lecture notes and references listed under ‘Further reading’ to help you.

### Task 1. Lists and Strings

In these exercises you will focus on results, not steps, and apply the principle:

*“Offload mundane details to programming languages, focus on the unique aspects of your programming problems”*

You will need to know what you can offload, so you will need to refer to the Scala API documentation for the *List* and *String*.

http://www.scala-lang.org/api/2.11.8/#scala.collection.immutable.List

http://www.scala-lang.org/api/2.11.8/#scala.collection.immutable.StringOps

Note that *List* is a class, while The *String* type in Scala has methods that come either from the underlying Java *String* or are added implicitly through [*scala.collection.  
immutable.StringOps*](http://www.scala-lang.org/api/2.11.8/scala/collection/immutable/StringOps.html)*.*

**List**

Create the following List:

val myList = List(1, 3, 1, 7, 9, 5)

1. Find the entry within the *List* class documentation page for the method *head*. What does this do? Call the method to test this.
2. Find and test a method that returns a *List* containing all the remaining elements of *myList*.
3. Perform the following operations on *myList* using a single method call in each case. Note the result and the type of the result for each. Note that the result is a new object, the original *myList* is not changed.

* Sum all the elements in *myList*
* Reverse the order of *myList*

1. Split *myList* into two separate lists – List(1,3,1) and List(7,9,5). As before, look in the documentation for a method that helps you do this.
2. Call the *filter* method of *myList* using:

myList.filter(p => p < 6 )

What is the effect of the lambda expression used as a parameter here? Use the same lambda expression in a call to a different method to get the result List(1,3,1)

1. In some of the exercises you have seen a method of *List* return another *List*. In these cases, you can chain methods together to perform more complex operations.

Your aim is to count the number of ‘1s’ in *myList*. How would you do this in an imperative approach? The functional approach might be to break the task into two operations, the first producing a result that can be fed into the other:

* Extract the ‘1s’
* Count them

This can be done with a chain of calls – test this example:

myList.filter(p => p == 1 ).length

1. Use a chain of at most three calls to produce the following:

(List(5, 1),List(3, 1))

1. Use a chain of two calls to find the lowest value in *myList* excluding the first three elements

**String**

Create the following string:

val myString = "A Santa Lived As a Devil At NASA"

Use the *StringOps* documentation to help with the following:

1. Reverse the string (note that as with the *List* examples, you don’t change the string, you create a new one
2. A palindrome is a word or phrase that is the same read forwards or backwards. Define a function *isPalindrome* that takes a *String* parameter and returns a result indicating whether the *String* is a palindrome. The function body should simply check whether the original string and its reverse are equal. Test your function – is *myString* a palindrome?
3. The phrase in *myString* is usually considered to be a palindrome, if you ignore the spaces and capital letters. Check that the phrase is a palindrome using your *isPalindrome* function and suitable methods of StringOps.

Remove the first word of *myString* and check that it is no longer a palindrome.

Use chains of calls to methods or properties of *StringOps* to do the following:

* Count the number of spaces in *myString*
* Count the number of distinct characters in *myString*, excluding spaces

1. (Challenge) Show that the following phrase is a palindrome using your *isPalindrome* function and suitable methods of *StringOps*.

"A man, a plan, a canal, Panama"

### Task 2. Recursion

In these exercises you will *“Focus on results, not steps (be declarative, not imperative)”* by using recursion in place of an imperative solution.

Example: calculating a factorial

This can be done in Scala like this:

def **factorial**(n: Int): BigInt = {  
 if (n == 0)  
 1  
 else  
 n \* **factorial** (n - 1)  
}

or more succinctly using pattern matching like this:

def **factorial**(n: Int): BigInt = n match {  
 case 0 => 1  
 case \_ => **factorial** (n-1) \* n  
}

The recursive function call (where the function calls itself) is highlighted in each case. Note that the return type is *BigInt*, as factorials can be very large numbers. In each case, the first branch or match is the stopping condition –each call to factorial can either return the value 1 or recursively call factorial again. Look at the example in the notes to see how this gives you the result you want.

1. Write and test the two versions of the factorial function above, e.g. factorial(10) should be 3628800.
2. Write a recursive function called *gcd* to find the greatest common divisor of two integers (you can use either of the programming styles illustrated)

For example, the greatest common divisor of **15** and **12** is **3** – no number greater than 3 divides exactly into both 15 and 12. One approach to showing this is to use the following steps:

*write the two numbers*  **15 12**

*find the remainder on dividing the first by the second*  **15 % 12 = 3**

*write the second number then that remainder* **12 3**

***repeat the operation*** *with these two numbers* **12 % 3 = 0**

**3 0**

When the second number is 0 (stopping condition), the result is the first number, 3 in this case

Your *gcd* function should follow this approach – note where the repetition occurs, this will be where a recursive call is needed.

1. (Challenge) Write and test a recursive version of the *power* function that you tested in Lab 1. The signature of the function should look like this:

def power(value: Int, pow: Int): BigInt = {

Think about what is repeated, and how this can be implemented with recursion – hint: the branch or match will depend on the second, parameter *pow*.

1. Do **Task 2**