LAB 5: MORE SCALA FUNCTIONAL PROGRAMMING TECHNIQUES AND CREATING A SCALA APPLICATION

In this lab you will gain further practice in functional programming in Scala and learn how to create a simple menu-driven application in Scala. You will work with partial function application and currying, and see examples of the use of *map*, *flatMap* and for comprehensions.

For this lab you should create an IntelliJ Scala project *lab5project*. Tasks 1 and 2 can be done with Scala worksheets, while Task 3 will require you to create a Scala application in your project.

Task 1. Partial function application and currying

Example 1 – Partially applied function

1. Define and test a function *isDivisible* which has two integer parameters x and y and returns true if x is divisible by y, false otherwise.

2. Define a variable *isEven* and assign to it the result of partially applying *isDivisible* as follows:

   ```
   val isEven = isDivisible(_: Int, 2)
   ```

   What is the type of *isEven*?

3. Test that you can apply *isEven* to a single parameter and check whether it is even, for example

   ```
   isEven(10)
   ```

4. Use *isEven* as a predicate in the filter method of List to transform List(1,2,3,4,5,6,7,8,9,10) to a list containing even numbers only.

Example 2 – Another partially applied function

In lab 4 you worked with a map containing airport cities and codes. In this example you will create functions to iterate through and print the contents of a similar map.

1. Create the following map:

   ```
   var airports = Map("Glasgow" -> "GLA", "Dubai" -> "DXB",
   "Berlin" -> "TXL")
   ```

2. Define and test the following function to print the keys and values in a map:

```
def printMap(mymap: Map[String,String]) = {
  for ((k, v) <- mymap) {
    println(s"$k - $v")
  }
}
```

Note that this function will work only for a map where keys and values are strings. A more generic version of the function that could be used with other types for keys and values could be defined:

```
def printMap[A,B](mymap: Map[A,B]) = {
  for ((k, v) <- mymap) {
    println(s"$k - $v")
  }
}
```

However, for this exercise it will be simpler to use the first version and work with strings only, although if you prefer you can use the generic approach.

---

3. Modify *printMap* as follows so that it becomes a higher-order function, where the second parameter is a function that determines how to print each key-value pair.

```
def printMap(mymap: Map[String,String],f:(String,String)=>
Unit) = {
  for ((k, v) <- mymap) {
    f(k,v)
  }
}
```

Don't try to test the modified *printMap* yet. In the following steps you will define functions that can be used for the second parameter of *printMap*, and then you will test it.

4. Define the following function that can be used as a parameter – this function prints a key and value in the same format as the first version of *printMap*.

```
def printKeyValue(k:String, v:String) = {
    println(s"$k - $v")
}
```

5. Define a similar function *printValueOnly* that prints out a value only in the following format:

```
Value – GLA
```

6. Test the higher-order *printMap* function using the airports map and each of the functions *printKeyValue* and *printValueOnly*

7. Define a variable *printKeysValues* and assign to it the result of partially applying *printMap*, specifying the function parameter only, as *printKeyValue*. The type of *printKeysValues* should be Map[String,String] => Unit.

   Test as follows:

   ```
   printKeysValues(airports)
   ```

   Note that if you were frequently needing to print out the keys and values of maps in an application, your code could be made simpler and more readable using this technique.

Example 3 – Curried function

1. Create a curried version of *printMap*, called *printMap_curried*. This will involve simply modifying the parameter list. You should be able to test your curried function by calling it as follows:

   ```
   printMap_curried(airports)(printValueOnly)
   ```

2. Test also by calling the curried function with a lambda expression as the second parameter list, and enclosing this in {} instead of () as follows:

   ```
   printMap_curried(airports){
     (k,v)=> println(s"Key -  $k")
   }
   ```

3. How would this compare with calling *printMap* with a lambda? Which would be clearer and more readable?

Task 2. Map, flatMap and for comprehensions

Example 1 – Comparing map and flatMap

In this task you will extract data from Scala maps using *map* and *flatMap*, and you will see how the same results can be achieved, more conveniently, using for comprehensions.

1. Create the following map as your data for this exercise (or continue from Task 1):

   ```
   var airports = Map(
     "Glasgow" -> "GLA",
     "Dubai" -> "DXB",
     "Berlin" -> "TXL")
   ```

2. Use the *get* method of Map to find the value (airport code) for a given key (city)?

   ```
   airports.get("Glasgow")
   ```

3. Try doing the same for a key that doesn't exist in the map:

   ```
   airports.get("Edinburgh")
   ```

   What values did you get? What is the type of the result returned by get?

4. Now you will search in the *airports* data for the airport codes for all cities in a list. Create the search list as follows:

   ```
   var searchlist = List("Glasgow", "Edinburgh", "Berlin")
   ```

   Note that the search list includes Edinburgh, which is not included in the data. How do you think you would approach this problem using an imperative approach? In Scala, it's a one-liner. Enter the following code:

   ```
   val codes = searchlist.map(x => airports.get(x))
   ```

   Describe the contents of the variable *codes*. This code takes each element of *searchlist* and maps it to the value matching the key equal to that element in the *airports* data.

5. Modify the expression for *codes* to use *flatMap* instead of *map*, and describe the difference this makes to the result.

Example 2 – *for* expression

1. The following expression is equivalent to the expression you used in example 1. Test this expression to check that it gives the same result as in step 5 of example 1.

```
val codes_for = for{
  x <- searchlist
  y <- airports.get(x)
} yield(y)
```

   This example uses a for comprehension. For comprehensions can be thought of as "syntactic sugar" for *map/flatMap*, and are often easier to write and understand. *x* and *y* are known as generators. Note that the y generator includes a reference to *x*, and this is equivalent to the *flatMap* call above. The yield expression allows you to define what result is returned from the generators.

Example 3 – Chaining functions

In example 1 you obtained a list of airport codes for the airports data based on a list of search values. Now you will apply a further transformation to the result extracted from the data by converting all the codes to lower case

1. Enter the following code, which applies the function *toLowerCase* to each element in the result of the search.

```
val codes_lower = searchlist.map(x =>
    airports.get(x)).map(y=>y.toLowerCase)
```

   What happens? Why can you not apply *toLowerCase*?

2. Modify the expression for *codes_lower* to use *flatMap* instead of *map*, and describe the difference this makes. Why does the use of *flatMap* allow these transformations to be chained?

3. Write an equivalent for expression to achieve the same result. This will involve changing only the yield expression in example 2.

Example 4 – A more complex for comprehension

The final example is a bit more complicated, to demonstrate more of the power of for comprehensions. You will define the search using the keys of a map rather than a list, and you will combine the values in the search map with the values extracted from the data.

4. Create the following map:

```
var searchmap = Map(
  "Glasgow" -> "Scotland",
```

```
  "Edinburgh" -> "Scotland",
  "Berlin" -> "Germany")
```

The aim of this exercise is to find the airports, if any, matching the keys in these map, and for each result found, combine the code with the matching country name in the search map, to get the result:

```
List(GLA - Scotland, TXL - Germany)
```

5. Enter the following code and check that it gives the required result. Study the code (and the comments) to make sure you understand how it works. Why do you need the final call to *.toList*?

```
val codes_countries_for =  for{
  x <- searchmap.keys    // the keys in the searchmap
  y <- airports.get(x)   // get codes for which a value
exists
  z <- searchmap.get(x)  // country for each key in
searchmap
} yield(y + " - " + z)

codes_countries_for.toList
```

6. Note that this could also be written using *map/flatMap* calls, but this one is definitely easier with a for comprehension.

For comprehensions are very powerful and there are many possible variations to solve a wide range of problems. There are more examples in your lecture notes, and you can find some more useful examples at:

https://gist.github.com/loicdescotte/4044169
http://eddmann.com/posts/using-for-comprehensions-in-scala/

Task 3. Creating a (functional) menu-driven application

Scala can be used to build pretty much any type of application, for example GUI applications (using Java UI libraries) or web applications (using a framework such as Play). In this task, though, you will create a simple text-based menu-driven console application. You may have created applications like this using other languages, but this example will have a functional twist or two.

The application will read in some data from a text file which contains Bundesliga football teams and their current points total. This data will be stored in a Map. Here is an excerpt from the file:

```
Bayern Munich, 24
RB Leipzig, 24
Hoffenheim, 20
```

The application will then display a menu with options to extract and process information from the data. Actually there will only two options initially, to list the teams and their points in order of the number of points, and to quit the application. An example session (with some output omitted for brevity) is shown below:

```
Please select one of the following:
  1 - show points for all teams
  2 - quit
1
Bayern Munich: 24
RB Leipzig: 24
Hertha BSC: 20
…
Please select one of the following:
  1 - show points for all teams
  2 - quit
2
selected quit

Process finished with exit code 0
```

Most of the code for the application will be provided for you - you will create the application and add this code to it. This will provide an example which will help you with your coursework.

1. Download the file *lab5.zip* and extract its contents, a set of .txt files. Open *data.txt* in an editor and observe the contents.

2. Create a new Scala object called MyApp in the src folder in your project. Remember that to do this you right-click on the folder and select New > Scala class from the menu. In the Create New Scala Class dialog, enter the name and select Object for Kind.

3. Add the following imports at the top of MyApp.scala:

```
import scala.io.Source
import scala.io.StdIn.readInt
import scala.io.StdIn.readLine
import scala.collection.immutable.ListMap
```

4. Make object MyApp extend App.

Reading the data from file

1. Copy the file *data.txt* into the root folder of your project.

2. Add the contents of the file *applogic_file.txt* into MyApp, inside the object MyApp. Note that this code calls a function called *fileRead* and assigns the result to a variable *mapdata*. This variable is the data for the application, and is of type Map[String, Int]. The code also prints the data so that you can check that it has been read correctly.

3. Add the contents of the file *fileread.txt* into MyApp immediately after the existing content. This defines the *fileRead* function. Review the comments in the code to understand how it works.

4. Run the application (right-click on the object in the Project View and select Run). Check that the data is as you expect. Don't worry about the order that the teams appear in the data, a Map in Scala is not sorted (if you want to have the items sorted by key you can use a SortedMap instead).

5. Once you are confident that the data is being read correctly you can remove or comment out the line that prints the data.

Creating the menu

1. Add the contents of the file *applogic_menu.txt* into MyApp, immediately after the code from *applogic_file.txt*. This code defines an action map, of menu options and the functions they map to, and calls functions to read user input and invoke corresponding functions. Review the comments in the code. The code will not compile at this point as there are references to functions that have not been defined yet.

2. Add the contents of the file *menufunctions.txt* after the existing code. This includes functions to:
    - Show a menu (with two options) to the user and read the user's selected option
    - Invoke a function specified by the user input and the corresponding value in the action map
    - Implement the actions referred to in the action map, to handle each of the menu options. Note that the handler for option 2 simply prints a message and returns false which will cause the application to terminate.

Review the comments in the code. Note that the function *mnuShowPoints* is a higher-order function, whose parameter is the operation that produces the data to be displayed. This code will still not compile as some functions have not been defined yet.

Creating the functionality for the menu options and running the application

1. Add the contents of the file *menuactions.txt* after the existing code. This defines a function to implement the "show points for all teams" menu option. If an option requires further user input then this function would get that input. This particular option doesn't need any further user input, though, so just does the following:
    - Applies the function that is passed to it as a parameter to get the data to display
    - Iterates through the data and prints to the output

2. Add the contents of the file *operations.txt* after the existing code. This defines a function to get the data for the "show points for all teams" menu option (it returns the full set of data in the original map, except sorted in descending order of points).

3. The code should now be complete. Run the application, and select option 1. You should see the teams and their points, with the team with the highest number of points first.

4. Select option 2. The application should terminate.

Adding a menu option

Add a menu option to show the points for a team chosen by the user.

You will need to modify the action map and functions that implement the menu, with a new option that calls

```
mnuShowPointsForTeam(currentPointsForTeam)
```

You will also need to add and complete the following functions:

```
def mnuShowPointsForTeam(f: (String) => (String, Int)) = {
// needs to print a prompt and use readline to get user
input
// should apply function f to user input string and print
result
// note that the result of f is a tuple
}

def currentPointsForTeam(team: String): (String, Int) = {
// should retrieve points value from mapdata for key team
and
// return a tuple of team and points
// remember that get returns an option so you will have to
// pattern match to get the value
```

```
    }
```

Test your modified application. An example session would look like this:

```
Please select one of the following:
  1 - show points for all teams
  2 - show points for selected team
  3 - quit
2
Team>Hertha BSC
Hertha BSC: 20
Please select one of the following:
  1 - show points for all teams
  2 - show points for selected team
  3 - quit
2
Team>Liverpool
Liverpool: 0
Please select one of the following:
  1 - show points for all teams
  2 - show points for selected team
  3 - quit
```

Note that in this implementation if a team name is entered that is not included in the data then the points is shown as 0.

Final note

Note that since Scala supports object-oriented programming, we could have used classes to organise the code, for example by making the menu action functions methods of a controller class. However, we are focusing on the functional programming techniques here so we simply put all functions within the MyApp object.