LAB 4: WORKING WITH SCALA COLLECTIONS

In this lab you will gain further practice in functional programming in Scala. You will work with functions, maps, lists and tuples. You can use whichever of the tools introduced in Lab 1 you prefer to attempt the following exercises. You should use your lecture notes (part 4&5) and references listed under ‘Further reading’ to help you.

Task 1. Mapping and folding

The ***map*** method of *List* applies a function to each element of the list in turn, and returns a new list containing the resulting elements. The following example applies a function that calculates the square of each element:

```
List(1,2,3,4,5).map(x=>x * x)
```

and returns

```
List(1, 4, 9, 16, 25)
```

The example can also be written with variations on the syntax such as the following:

```
List(1,2,3,4,5) map {x=>x * x}
```

1. Use *map* to transform the list *List("1","2","3","4","5")* to the list of integers *List(1,2,3,4,5).* Use whichever of the above styles of writing map above that you prefer.

2. Use *map* to transform the list *List("aa","bb","cc","dd","ee")* to the list of strings *List("AA","BB","CC","DD","EE")*

The ***foldLeft*** method of *List* transforms the list to a value by applying a function in turn to each element and the result of applying it to previous elements. The following example calculates the factorial of 6 by multiplying the values in the range 1 to 6. The second parameter of *foldLeft* is a function of two parameters *x* and *y* that represent the total so far (*x*) and the current element (y). The first parameter of *foldLeft*, 1, is the initial value for *x* when applying to the first element:

```
List(1,2,3,4).foldLeft(1)((x,y) => x * y)
```

This can also be written with some variations on the syntax, for example:

```
List(1,2,3,4).foldLeft(1){(x,y) => x * y}
```

The function can also be written in shorthand form, where the two _ represent the parameters:

```
List(1,2,3,4).foldLeft(1)(_ * _)
```

3. Create a list of integers *list*, using whatever syntax you prefer for creating a list, containing the values 1 to 19.

4. Define and test a function *sum* that uses the the *foldLeft* method of *List* to evaluate the sum of the list elements. Use whichever of the above styles of writing *foldLeft* above that you prefer. The signature of the sum function should be:

   ```
   def sum1(list:List[Int]):Int =  …
   ```

   What must the value of the first parameter of *foldLeft* be?

5. Define and test a function *length* that uses the *foldLeft* method to compute the length of the list. Again, the function should have a single parameter of type *List[Int]* and return type *Int*.

6. Define and test a function *average* that uses the *foldLeft* method to compute the average (mean) of the list elements (hint: this will need two calls to *foldLeft*). The parameter of this list will be as before, but what should the return type be?

7. Define and test a function last that uses the *foldLeft* method to find the last element of the list. In this case, the first parameter of *foldLeft* (the initial value) needs to be *list.head*, and the function applied should simply return the current element.

You have already seen recursive functions that perform some of these computations – which do you prefer, folding or recursion?

8. Look up the Scala documentation for the *List* class and compare the entries for *foldLeft* and *foldRight*. Do you think it will make any difference if you had used *foldRight* instead of *foldLeft* in any of these examples (you can try it to check)?

9. *(Challenge)* Define and test functions that take a parameter of type *List[Int]* and compute, using *foldLeft*, each of the following. You will need to think carefully about the type to be returned, and the starting value:

   a. the penultimate element of the list
   b. the list in reverse order
   c. the distinct elements of the list (test this with a list containing the elements (1,2,2,3,3,4,5) – the result should contain no repeated values)

Task 2. Maps

A map in Scala is a collection of key/value pairs. Keys and values can be of any type. For this exercise you may find it useful to refer to the Scaladocs fo the Map classes.

Creating maps

You can create a map like this:

```
val french = Map(1->"un", 2->"deux", 3->"trois")
```

Maps are immutable by default. If you want to create a mutable map you need to specify a mutable type explicitly:

```
var mutableFrench = scala.collection.mutable.Map(1-> "un", 2-
>"deux", 3->"trois")
```

If a map is mutable, you can add a new entry using the + operator or += operator, for example:

```
mutableFrench += 4 -> "quatre"
```

You can concatenate maps using the ++ or ++= operator, for example:

```
val moreFrench = Map(5-> "cinq", 6->"six")
mutableFrench ++= moreFrench
```

1. Create a mutable map *airports* containing the following key/value pairs representing cities and the codes of their airports (as strings):

   Glasgow -> GLA
   Dubai -> DXB
   Berlin -> TXL

2. Create a map *moreAirports* containing a single pair:

   Helsinki -> HEL

3. Create a map *evenMoreAirports* containing the pairs:

   Glasgow -> PIK
   Los Angeles -> LAX

4. Create a new map *newAirports* by concatenating *moreAirports* and *evenMoreAirports* using the ++ operator.

5. Add (concatentate) *newAirports* to *airports* using the ++= operator. Look at the result – what happened to Glasgow?

6. Add the following single entry to *airports* using the += operator.

   Tokyo -> HAN

Extracting data from maps

1. You can get the keys of a map as a list using the *keys* method:

   ```
   val cities = airports.keys.toList
   ```

   Use a different method of *Map* to get a list containing the airport codes in *airports*.

2. You can get the value for a specified key using the *get* method:

   ```
   val gla = airports.get(“Glasgow”)
   ```

   Try this. What type is the result? Try finding the value for the key “London”. Why do you think the *get* method returns the type that it does?

3. Where a method returns an *Option*, the safest way to extract the value you want from the *Option* is to use pattern matching:

   ```
   val gla2 = airports.get("Glasgow") match {
     case Some(ap) => ap
     case None => "not found"
   }
   ```

   Try this, and try also with the key “London”.

   Include use of find – tweak this example to fit:
   find method of map
   give an example, e.g.:
   val airports = Map("GLA" -> "Glasgow", "TXL" -> "Berlin", "PIK" -> "Glasgow", "LAX" -> "Los Angeles")
   val default = ("not found","")
   var value = "Berlin"
   airports.find(_._2==value).getOrElse(default)._1
   value = "Los Angeles"
   airports.find(x =>x._2==value).getOrElse(default)._1
   value = "Glasgow"
   airports.find(_._2==value).getOrElse(default)._1

Iterating

Sometimes it is useful to iterate through a collection without returning a result. For example, you might simply want to print the contents of the collection. (In functional terms, this would be an example of a side-effect as it changes the state of the console display).

You can do this for a list using a *for* expression, for example

```
for (x <- mylist) {
  println(x)
}
```

or the *foreach* method of List

```
mylist foreach {
  x => println(x)
}
```

For a map, you can iterate through the key/value pairs:

```
for ((k, v) <- mymap) {
  println(s"$k - $v")
}

mymap foreach {
  case (k, v) => println(s"$k - $v"))
}
```

1. Using either *for* expressions or *foreach*, print the contents of the airports map as follows (first just the keys, then keys and values together):

   ```
   Code - Helsinki
   Code - Los Angeles
   Code - Tokyo
   Code - Glasgow
   Code - Dubai
   Code – Berlin

   City:Helsinki - Code:HEL
   City:Los Angeles - Code:LAX
   City:Tokyo - Code:HND
   City:Glasgow - Code:PIK
   City:Dubai - Code:DXB
   City:Berlin - Code:TXL
   ```

Task 3. Tuples and zipping

A Scala tuple is a class that can contain a miscellaneous collection of elements. It is “1-indexed”, so the first element has index 1.

You can create a tuple and access its elements like this:

```
val tuple = ("apple", "red")
val fruit = tuple._1
val colour = tuple._2
```

Tuples can contain elements of different types:

```
val getUserInfo = ("Al", 42, 200.0)
```

You can extract several elements at once and assign these to named variables:

```
val(name, age, weight) = getUserInfo
```

1. Try the examples above. Evaluate the variables name, age and weight to ensure they have the values you expect.

Tuples are useful in many situations and can be used along with other collections. For example, you can have a list of tuples:

```
val fruits = List(("apple", "red"), ("banana", "yellow"),
("orange", "orange"))
```

If the tuples have two elements each, you can convert this to a Map:

```
val fruitMap = fruits toMap
```

You can also have a tuple containing lists:

```
val lists = (List(1,2,3), List(4,5,6), List(7,8,9))
```

2. Create a map of fruits/colours using the above code, and print the contents by iterating to give:

   ```
   Fruit:apple – Colour:red
   Fruit:banana – Colour:yellow
   Fruit:orange – Colour:orange
   ```

3. Create the following lists:

   ```
   val cities = List("Glasgow", "Dubai", "Berlin")
   val codes = List("GLA","DXB","TXL")
   ```

4. Evaluate the following expression, which uses the *zip* operator, and describe the result and the effect of *zip*.

   ```
   cities zip codes
   ```

5. Use the *zip* operator and a suitable method call to create a map of cities/codes (similar to the ones in task 2), and print the contents by iterating.

You have now seen how to use *zip* to join two lists to form a map. The next example shows another way to use *zip* to combine lists.

6. Create the following lists that represent golf players and their scores in two rounds of a tournament.

   ```
   val players = List("Stenson", "Mickelson", "Galllacher")
   val round1 = List(70, 68, 70)
   val round2 = List(65, 72, 68)
   ```

7. Check that

   ```
   round1 zip round2
   ```

   creates a list of tuples, each of which contains one player's scores for the two rounds:

   ```
   List((70,65), (68,72), (70,68))
   ```

8. Instead of creating a map of the results, you can create a new list that contains the total score for each player by applying a suitable function to each tuple in the list:

   ```
   val scores2 = round1 zip round2 map{
     case (x, y) => x + y
   }
   ```

   Try this and check that the result is as you expect.

9. Finally, use *zip* again to create a map of players and total scores and print the contents by iterating, to give:

   ```
   Player:Stenson - total score 135
   Player:Mickelson - total score 140
   Player:Galllacher - total score 138
   ```