# Advanced Programming Lab Guideline Solutions

## Lab 1

1. Write and test a function called *square* that takes a parameter of type Double and returns a Double that is the square of the parameter
2. Write and test a function called *distance* that takes parameters x1,y1,x2 and y2, all of type Double as parameters. These represent the coordinates of two points in 2D space (x1,y1) and (x2,y2). Your function should return the distance between these points, using the distance formula:

For example, distance(0,0,1,1) should return 1.414 (approximately)

You can make use of your square function in the distance function. You will also need to use Scala’s *sqrt* function, which is in the *scala.math* package, so you will need the following statement:

import scala.math.sqrt

1. Modify your distance function to use default and named parameters so that it can be called with two parameters only, and the distance from (0,0) to that point will be calculated, e.g. distance(x2=1,y2=1).

### Code Outline

import scala.math.sqrt

def square(value: Double) : Double = {value \* value}

def distance(x1:Double=0, y1:Double=0, x2:Double, y2:Double) : Double = {

sqrt(square(x2-x1) + square(y2-y1))

}

val result1 = distance(0,0,1,1)

println(result1)

val result2 = distance(x2=1,y2=1)

println(result2)

## Lab 2

### List

1. 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*

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

val sumList = myList.sum

println(sumList)

val myList2 = myList.reverse

println(myList2)

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.

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

val myList2 = myList.take(3)

println(myList2)

val myList3 = myList.takeRight(3)

println(myList3)

val myTuple = myList.splitAt(myList.size/2)

println(myTuple.\_1)

println(myTuple.\_2)

1. 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)

val myList2 = myList.takeWhile(p => p < 6)

println(myList2)

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

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

val myList2 = myList.filter(p=>p<6).reverse.splitAt(2)

println(myList2)

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

val myList2 = myList.takeRight(myList.size-3).min

println(myList2)

### String

1. Reverse the string (note that as with the *List* examples, you don’t change the string, you create a new one

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

val myString2 = myString.reverse

println(myString2)

1. 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.

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.

def isPalindrome(stringToCheck: String):Boolean = {

val strippedString = stringToCheck.filter(p => p != ' ')

strippedString.equalsIgnoreCase(strippedString.reverse)

}

println(isPalindrome(myString))

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

val myString2 = myString.toUpperCase().distinct

.count(c => c != ' ')

println(myString2)

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"

val newString = "A man, a plan, a canal, Panama"

def isPalindrome(stringToCheck: String):Boolean = {

val strippedString = stringToCheck.filter(p => p != ' ')

strippedString.equalsIgnoreCase(strippedString.reverse)

}

println(isPalindrome(newString.filter(c => c != ',')))

### Recursion

1. Write a recursive function called *gcd* to find the greatest common divisor of two integers (you can use either of the programming styles illustrated)

def gcd(first:Int, second:Int):Int = first % second match {

case 0 => second

case \_ => gcd(second, first % second)

}

println(gcd(120,50))

println(gcd(34,11))

def gcd2(first:Int, second:Int):Int = second match {

case 0 => first

case \_ => gcd2(second, first % second)

}

println(gcd2(120,50))

println(gcd2(34,11))

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 = pow match {

case 0 => 1

case \_ => value \* power(value, pow - 1)

}

println(power(2,4))

println(power(2,10))

## Lab 3

### Lists and recursion

1. Write a function *sum* that uses recursion to calculate the *sum* of the elements of a list of Ints. You can use either an *if* expression or a *match* expression. What will the stopping condition be in this case - It’s actually slightly simpler than the length example?

val list1 = 1::2::3::4::5::6::7::8::9::10::Nil

def sumList1(ls:List[Int]):Int = ls.tail match {

case Nil => ls.head

case \_ => ls.head + sumList1(ls.tail)

}

println(sumList1(list1))

def sumList2(ls:List[Int]):Int = ls match {

case h :: Nil => h

case \_ :: tail => ls.head + sumList2(tail)

}

println(sumList2(list1))

println(list1.sum)

### Tail Recursion

1. Write and test a tail-recursive version of your *sum* function

def sumListTail(ls:List[Int]):Int = {

def sumList\_nested(ls:List[Int], sum:Int):Int =

ls match {

case Nil => sum

case h :: tail => sumList\_nested(tail, sum + h)

}

sumList\_nested(ls, 0)

}

println(sumListTail(list1))

1. Write and test a tail-recursive version of the recursive *power* function you created in Lab 2.

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

def powerTail\_nested(value:Int, pow:Int,

result:BigInt):BigInt = pow match {

case 0 => result

case p => powerTail\_nested(value, p-1,

result \* value)

}

powerTail\_nested(value, pow, 1)

}

println(powerTail(2, 4))

1. (Challenge) The following function (if and match versions shown) calculates the nth element of the Fibonacci sequence 1,1,2,3,5,8,13,… (each element is the sum of the two previous ones).

Write a tail-recursive version of this function.

def fibTail(n:Int) = {

def fibTail\_nested(n:Int,result:Int):Int = n match {

case 0 => 0

case 1 => 1

case \_ => fibTail\_nested(n-1, result) +

fibTail\_nested(n-2, result)

}

fibTail\_nested(n, 1)

}

println(fibTail(6))

println(fibTail(10))

def fibTail2(n: Int): Int = {

def fib\_tail(n: Int, a: Int, b: Int): Int = n match {

case 0 => a

case \_ => fib\_tail(n-1, b, a+b)

}

return fib\_tail(n, 0 , 1)

}

println(fibTail2(6))

println(fibTail2(10))

## Lab 4

### Mapping and Folding

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.

List("1","2","3","4","5") map (x => x.toInt)

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

List("aa","bb","cc","dd","ee") map (x => x.toUpperCase)

1. Define and test a function *sum* that uses the the *foldLeft* method of *List* to evaluate the sum of the list elements.

val list1 = List.range(1,20)

def sum(ls:List[Int]):Int = {

ls.foldLeft(0) { \_ + \_}

}

println(list1)

println(sum(list1))

1. 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*.

def length(ls:List[Int]):Int = {

ls.foldLeft(0) ((x,y) => x + 1)

}

println(length(list1))

1. 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*).

def average(ls:List[Int]):Float = {

sum(ls)/length(ls)

}

println(average(list1))

1. 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.

def lastElement(ls:List[Int]):Int = {

ls.foldLeft(ls.head) { (x, y) =>

println("x: "+ x + " y: " + y);y}

}

println(lastElement(list1))

1. *(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: the penultimate element of the list.

def penultimateElement(ls:List[Int]):Int = {

ls.foldLeft(ls.head,ls.tail.head) {(x,y) =>

println("x: "+ x + " y: " + y);

(x.\_2,y)

}.\_1

}

println(penultimateElement(list1))

### Maps

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

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

Helsinki -> HEL

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

Glasgow -> PIK

Los Angeles -> LAX

1. Create a new map *newAirports* by concatenating *moreAirports* and *evenMoreAirports* using the ++ operator.
2. Add (concatentate) *newAirports* to *airports* using the ++= operator. Look at the result – what happened to Glasgow?
3. Add the following single entry to *airports* using the += operator.

Tokyo -> HAN

var airports = Map("Glasgow" -> "GLA", "Dubai" -> "DXB",

"Berlin" -> "TXL")

val moreAirports = Map("Helsinki" -> "HEL")

val evenMoreAirports= Map("Glasgow"->"PIK",

"Los Angeles" -> "LAX")

var newAirports = moreAirports ++ evenMoreAirports

airports ++= newAirports

airports += "Tokyo" -> "HAN"

println(airports)

#### Extracting data from maps

##### Example 1

val cities = airports.keys.toList

println(cities)

##### Example 2

val codes = airports.values.toList

println(codes)

##### Example 3

val gla = airports.get("Glasgow")

println(gla)

val ldn = airports.get("London")

println(ldn)

##### Example 4

val gla2 = airports.get("Glasgow") match {

case Some(ap) => ap

case None => "Not Found"

}

println(gla2)

val ldn2 = airports.get("London") match {

case Some(ap) => ap

case None => "Not Found"

}

println(ldn2)

##### Example 5

val default="Not Found"

var value="CDG"

println(airports.find(\_.\_2==value).getOrElse(default))

##### Example 6

value="PIK"

println(airports.find(\_.\_2==value))

println(airports.find(\_.\_2==value).get)

println(airports.find(\_.\_2==value).get.\_1)

value = "LAX"

println(airports.find(x =>x.\_2==value).getOrElse(default))

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):

airports foreach {

case (k, v) => println(s"Code - $k")

}

airports foreach {

case (k, v) => println(s"City:$k - Code: $v")

}

### Tuples and zipping

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

cities zip codes

1. 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.

val cities = List("Glasgow", "Dubai", "Berlin")

val codes = List("GLA","DXB","TXL")

val citiesList = cities zip codes

println(citiesList)

val citiesMap = citiesList toMap

citiesMap foreach {

case (k, v) => println(s"City:$k - Code: $v")

}

1. Create the lists that represent golf players and their scores in two rounds of a tournament. Check that round1 zip round2 creates a list of tuples, each of which contains one player’s scores for the two rounds:

val players = List("Stenson", "Mickelson", "Galllacher")

val round1 = List(70, 68, 70)

val round2 = List(65, 72, 68)

println(round1 zip round2)

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

val scores2 = round1 zip round2 map{

case (x, y) => x + y

}

val playersList = players zip scores2

val playersMap = playersList toMap

playersMap foreach {

case (k, v) => println(s"Player:$k - total score: $v")

}

## Lab 5

### Partial function application and currying

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. 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.

def isDivisible(x:Int,y:Int):Boolean = {

x % y == 0

}

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

println(List.range(1,11).filter(isEven(\_)))

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

val airports = Map("Glasgow" -> "GLA", "Dubai" -> "DXB",

"Berlin" -> "TXL")

def printMap(myMap:Map[String, String]) = {

for ((k,v) <- myMap) {

println(s"$k - $v")

}

}

printMap(airports)

1. 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.
2. 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 printMap2(myMap:Map[String, String],

f:(String,String) => Unit) = {

for ((k,v) <- myMap) {

f(k,v)

}

}

def printKeyValue(k:String, v:String) = {

println(s"$k : $v")

}

printMap2(airports, printKeyValue(\_,\_))

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

Value – GLA

def printValueOnly(k:String,v:String) = {

println(s"value - $v")

}

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

printMap2(airports, printKeyValue(\_,\_))

printMap2(airports, printValueOnly(\_,\_))

1. 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.

val printKeysValues = printMap2(\_: Map[String,String],printKeyValue(\_,\_))

printKeysValues(airports)

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:
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:

def printMap\_curried(myMap:Map[String, String])(f:(String,String) => Unit) = {

for ((k,v) <- myMap) {

f(k,v)

}

}

printMap\_curried(airports)(printValueOnly)

printMap\_curried(airports) {

(k,v) => println(s"Key - $k")

}

### Map, flatMap and for comprehensions

1. Create the following map as your data for this exercise (or continue from Task 1):
2. Use the *get* method of Map to find the value (airport code) for a given key (city)?

val airports = Map("Glasgow" -> "GLA", "Dubai" -> "DXB",

"Berlin" -> "TXL")

airports.get("Glasgow")

airports.get("Edinburgh")

1. 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")

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

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

val codes2 = searchList.flatMap(x => airports.get(x))

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)

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

val codes\_lower = searchList.flatMap(

x => airports.get(x)).map(y=>y.toLowerCase)

val codes\_for2 = for {

x <- searchList

y <- airports.get(x)

} yield(y.toLowerCase)

1. 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)

1. 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   
 y <- airports.get(x)   
 z <- searchmap.get(x)   
} yield(y + " - " + z)  
  
codes\_countries\_for.toList

### Creating a (functional) menu-driven application

**object MyApp extends App {**

// APPLICATION LOGIC

// read data from file

**val mapdata = readFile("data.txt")**

// print data to check it's been read in correctly

**println(mapdata)**

// define menu options as a Map of actions

**val actionMap = Map[Int, () => Boolean](1 -> handleOne,**

**2 -> handleTwo,**

**3 -> handleThree)**

// loop to read input and invoke menu option

**var opt = 0**

**do {**

**opt = readOption**

**} while (menu(opt))**

// UTILITY FUNCTIONS

// reads data file - comma separated file

**def readFile(filename: String): SortedMap[String, Int] = {**

// create buffer to build up map as we read each line

**var mapBuffer: SortedMap[String, Int] = SortedMap()**

**try {**

**for (line <- Source.fromFile(filename).getLines()) {**

**val splitline = line.split(",").map(\_.trim).toList**

**mapBuffer = mapBuffer ++**

**SortedMap(splitline.head ->**

**splitline.tail.head.toInt)**

**}**

**} catch {**

**case ex: Exception => println("Exception happened.")**

**}**

**mapBuffer**

**}**

// FUNCTIONS FOR MENU

// shows menu and reads input

**def readOption: Int = {**

**println(**

**"""|Please select one of the following:**

**| 1 - show points for all teams**

**| 2 - show points for chosen team**

**| 3 - quit""".stripMargin)**

**readInt()**

**}**

// invokes selected menu option

**def menu(option: Int): Boolean = {**

**actionMap.get(option) match {**

**case Some(f) => f()**

**case None =>**

**println("Sorry, that command is not recognized")**

**true**

**}**

**}**

// handlers for menu options

**def handleOne(): Boolean = {**

**mnuShowPoints(currentPoints)**

**true**

**}**

**def handleTwo(): Boolean = {**

**mnuShowPointsForTeam(currentPointsForTeam)**

**true**

**}**

// FUNCTIONS THAT INVOKE ACTION AND INTERACT WITH USER

**def mnuShowPoints(f:() => Map[String, Int]) = {**

**f() foreach {case (x,y) => println(s"$x: $y")}**

**}**

**def mnuShowPointsForTeam(f:(String) => (String, Int)) = {**

**val team = readLine("Enter Team")**

**val result = currentPointsForTeam(team)**

**println(result)**

**println(result.\_1)**

**println(result.\_2)**

**val points = result.\_2**

**println(s"$team: $points")**

**}**

// OPERATION FUNCTIONS

**def currentPoints():Map[String,Int] = {**

**ListMap(mapdata.toSeq.sortWith(\_.\_2 > \_.\_2):\_\*)**

**}**

**def currentPointsForTeam(team: String): (String,Int) = {**

**println(s"Team is $team")**

**mapdata.get(team) match {**

**case Some(x) => (team,x)**

**case None => println("team not recognized");(team, 0)**

**}**

**}**

**}**