Programming Paradigms

Coursework Project

Evaluation

Submitted by:

Abdelrahman Mohamed – S2168397

Submitted to:

Dr. James Paterson

Evaluation

In this report, I will evaluate my work on the Course work project implemented in Scala. The project involves the development of different functions to have the operations working successfully, and uses different data structures to represent the data that we have such as maps, lists, and higher-order functions.

Functional Thinking

In terms of functional thinking, the solution consistently demonstrates the application of functional principles. The algorithms and data structures used in the solution are designed to be immutable and stateless, which allows them to be easily composed and reused in different contexts. The solution also makes heavy use of higher-order functions, as well as functions like `map`, `fold`, and `mapValues`, which are key characteristics of functional programming.

Throughout the application, composition of higher-order functions has been used to implement the functions properly in a well-structured architecture, so the code is more concise and easier to read by abstracting out common patterns. For example, when the user invokes a menu operation, a handler function "Predicate" is called, which itself calls another higher-order function that takes a supplier function as the argument. [Screenshot 1]

```
// handlers for menu options
Abdelrahman
def handleOne(): Boolean = {
  menuShowCurrentStocks(currentPrice)
  true
}

Abdelrahman
def handleTwo(): Boolean = {
  menuShowHighLow(HighLow)
  true
}

Abdelrahman
def handleThree(): Boolean = {
  menuShowMedian(Median)
  true
}
```

Screenshot 1

Evaluation

The `MenuOpertaion` functions were structured so that they are allowed to interact with the user and handle the interactions. Each one of them is built to call the operation function and output and loop through the result to output it to the user in a clearly formatted way. For the functions that needed the user’s input, they were responsible for checking on the input to make sure it was available on the data before passing it into the operation function, leveraging the `pattern matching` and `Option` method. [Screenshot 2]

```
Abdelrahman
def menuShowMedian(f: () => Map[String, Double]) = {
  f() foreach { case (stock, median) => println(s"$stock: $median") }
}

Abdelrahman
def menuShowRise(f: () => String) = {
  var result = f()
  println(s"The stock $result had the largest rise over the last week.")
}

Abdelrahman
def menuCompareAverage(f:(String,String) =>  (Double,Double)) = {
  var flag = false
  print("First Stock: ")
  val FirstStock = readLine()
  print("Second Stock: ")
  val SecondStock = readLine()
  mapdata.get(FirstStock) match { //Check if the stock symbols is correct
    case Some(f) =>
      mapdata.get(SecondStock) match {
        case Some(f) => flag = true
        case None => println("The Second stock is not recognized")
      }
    case None => println("The first stock is not recognized")
  }
  if (flag) {
    val(average1,average2) = Average(FirstStock,SecondStock);
```

Screenshot 2

Each operation function was done differently using functional methods based on the function’s operations and what it’s supposed to do. While implementing all of them, functional principles were heavily used and thought of, thus, all of them are based on the commonly used functional methods of transforming and reducing the *`mapdata`* variable that holds the data into a simpler component representing the operation needed immediately without using mutable variables or explicit looping.

Functional Programming Style

In terms of functional programming style, the application uses a number of techniques that are commonly used in functional programming. The first operation makes use of the `for-comprehension` syntax along with the `yield` method, which allows us to write complex transformations and operations in a more concise and readable way and helps to ensure the integrity and immutability of the data. Thus, it becomes easy to transform the complex *`mapdata`* map into a simple list representing the most recent prices without the need for mutable variables. [screenshot 3] Alternatively, another technique that could be used here was the mapValues method that applies (_.last) on each element, but the `for-comprehension` method was used because it offers clearer code.

Abdelrahman *

```
def currentPrice(): Map[String, Int] = {
  for ((stock, values) <- mapdata)
    yield (stock, values.last)
}
```

Screenshot 3

The following operations use the `map` method to apply a function to each stock in the *`mapdata`* variable. This is useful for transforming the elements in the data structure in some way, such as by applying a mathematical operation to each element or by extracting specific information from each element. For example, in the second operation, it was used to get the minimum and maximum value for each stock and transform the result into a simple map without the need to have additional variables. In the third operation, it is used to apply `sort, split`, and mathematical operations on the list in a very simple and clean way. In the fourth operation, It is used to access the elements of the list and do some calculations, then we use the `maxBy` method with the "`. Notation`" to get our needed result. [Screenshot 4]

```scala
Abdelrahman
def HighLow(): Map[String, List[Int]] = {
  mapdata.map { case (stock, values) =>
    stock -> List(values.min, values.max)
  }
}

Abdelrahman
def Median(): Map[String, Double] = {
  mapdata.map { case (stock, values) =>
    val sortedValues = values.sorted
    val (up, down) = sortedValues.splitAt(sortedValues.size / 2)
    val median = (up.last + down.head) / 2.0
    stock -> median
  }
}

Abdelrahman
def Rise(): String = {
  var Rise = mapdata.map { case (stock, values) =>
    var rise = values.last - values(values.size - 7)
    stock -> rise
  }
  Rise.maxBy(_._2)._1
}
```

Screenshot 4

The fourth operation leverage the `mapValues` method to partially apply the needed calculations on the `mapdata` in order to get the average price of each stock, by transforming the values of a map without having to explicitly loop through the keys, which can be a more efficient and elegant approach. Then a call to the variable with the names of the two stocks inputted by the user is done that returns a tuple of the needed answer. Alternatively, this can be done by applying the `mapValues` method twice for each stock and saving them in variables, but the concept of partial functions allows us to have this simple version. [Screenshot 5]

```
Abdelrahman
def Average(FirstStock: String, SecondStock: String):(Double,Double) = {
  val average = mapdata.mapValues(values => values.sum.toDouble / values.length)
  (average(FirstStock),average(SecondStock))
}
```

Screenshot 5

Functional vs. Imperative

Overall, I believe that the use of functional programming techniques in this solution has contributed to the ease of development and the quality of the solution. The use of immutability and statelessness makes it easy to reason about the behavior of the algorithms and data structures, and the use of higher-order functions allows you to write concise and elegant code. This is because functional programs are typically composed of small, modular functions that each have a well-defined purpose and input-output relationship like what we have in the application.

In contrast, if this project was built imperatively with Python for example, it would be much more complex, longer, and probably repeated for all operations with many different control structures that are difficult to follow. Most of them will be built by looping through the *mapdata* variable and accessing the needed elements by index and do the needed operations imperatively, by specifying a step-by-step execution method.