**Martin Vladimirov Karastoyanov**

**S2031121**

**Programming Paradigms Coursework**

**Submission 1: Documentation**

**Evaluation**

This evaluation looks at a Scala solution that integrates functional programming principles and has been developed for a food pricing application. The codebase is an example of functional programming, with features like pattern matching, higher-order functions, and immutability. The purpose of this evaluation is to assess how these ideas affected the process of development.

## 1. Functional Thinking and Programming style

### Immutability

The ***ListMap*** data structure is used to store price data. This data structure is inherently immutable, which means that its values cannot be changed after they are created. This is a good choice for this data structure because it ensures that the price data remains consistent and prevents accidental modifications.

|  |
| --- |
| val mapdata = readFile("data.txt") |

It is necessary to take note of the first declaration of ***mapdata*** as ***var*** in the **readFile** function, which is a one-time event that occurs when reading a file and requires to be declared as **var**, however in the main body of the Application, this value is assigned to a ***val*** ListMap with the same name.

|  |
| --- |
| var mapdata: ListMap[String, List[Int]] = ListMap() |

### High-Order Functions

In this solution, the higher-order functions are implemented through ***actionMap***, ***menu***, ***readOption***, and individual task-handling functions. The ***actionMap*** plays a crucial role in associating numeric user options with specific functions. Because of the dynamic function assignment made possible by these higher-order functions, the code is flexible and

expandable.

|  |
| --- |
| val actionMap = Map[Int, () => Boolean](1 -> handleOne, 2 -> handleTwo, 3 -> handleThree  , 4 -> handleFour, 5 -> handleFive, 6 -> handleSix, 7 -> handleSeven) |

### Pure Functions

The code relies heavily on pure functions, as they do not have side effects and produce the same output, contributing to easier reasoning and testing. For example, functions like **getLastValues**, **getMinMaxPrices**, and **getMedianPrices** they perform calculations and produce the same output for the same input, without modifying the external state of the program.

|  |
| --- |
| def getLastValues(): Map[String, Any] = {  mapdata.view.mapValues(values => values.lastOption.getOrElse("N/A")).to(ListMap)  } |

### Function Composition

The code uses function composition to create more complex operations. It creates a chaining of functions, resulting in data transformation in a step-by-step manner. It allows for readability and breaking complex operations into smaller, composable functions.

|  |
| --- |
| def getMinMaxPrices(): Map[String, (Any, Any)] = {  mapdata.view.mapValues(values => {  val highestPrice = values.max  val lowestPrice = values.min  (highestPrice, lowestPrice)  }).to(ListMap)  } |

In this instance, the ***mapValues*** function integrates an anonymous function that calculates the highest and lowest prices for each food item. It then creates a tuple with the highest and lowest prices and creates a new map associating each item with the prices, which is then converted to a ***ListMap***, to preserve the order of the initial ***mapdata***.

### Pattern Matching

The code also uses a pattern matching functionality, which is used for handling different cases, with a clear example in the ***menu*** function.

|  |
| --- |
| def menu(option: Int): Boolean = {  actionMap.get(option) match {  case Some(f) => f()  case None =>  println("Sorry, that command is not recognized\n")  true  }  } |

Using pattern matching increases the code readability and provides a concise syntax for branching, as well as stick to the functional programming and Scala methodology.

## 2. Possible alternative techniques

### 2.1 Immutability

If order is not crucial, a regular ***Map*** could be used, however I wanted to keep the order of the “data.txt” file, making the application easier for testing and keeping the same insertion order throughout the whole Application. Alternatively, a ***List*** or **Vector of tuples** could be used, but these wouldn't provide constant time access to elements.

### 2.2 Pattern Matching

Using a series of ***if-else*** statements could achieve the same result, but pattern matching is more concise and idiomatic in Scala, as well as the functional programming methodology.

### 2.3 High-Order Functions

These could be separate classes or objects, but using higher-order functions keeps related functionality together. The potential drawback lies in consolidating all functions as methods within a single class or another, potentially defaulting to an object-oriented programming (OOP) approach.

## 2. Comparison of functional and imperative style

Functional programming enhances ease of reasoning, conciseness, and readability in code development. By emphasising the creation of pure functions, functional programming promotes a modular approach that isolates each function's behaviour, making it easier to understand and reason about the code. Moreover, functional constructs like higher-order functions and pattern matching contribute to code conciseness and readability, enhancing overall comprehension. Moreover, minimising mutable state helps prevent bugs related to unexpected changes in variables.

Imperative programming, in contrast, involves a more step-by-step approach to code execution. While it allows for explicit control over mutable state, it may lead to increased complexity in reasoning about program behaviour. As in comparison to the functional programming just answering the question of “what", the imperative programming answers to the question "how", that can lead to a bigger code block and the potential for bugs, especially related to mutable state changes is heightened. Imperative programming relies on explicit commands and may require a careful management of side effects to maintain code clarity.

Having explored OOP concepts and methodology throughout my degree, making the shift to functional thinking presents a challenge, even though it promises a new problem-solving approach. While acknowledging the potential of recursion for tackling iteration-related challenges in coursework, I hesitated to use it independently. Instead, I chose the pre-defined functions provided by the Scala library for iteration tasks, as I lacked confidence in implementing recursion on my own and determining whether it is the correct approach.

Given the freedom to choose a programming language, I would opt for Java or C++ for this task, as I feel more at ease with imperative programming styles involving loops, class organisation, and object-oriented principles. However, following this coursework and the Programming Paradigms module, I am inclined to believe that a blend of imperative and functional approaches could prove beneficial. Embracing a "what" rather than a "how" perspective might offer advantages, even if it comes with potential performance costs and a more open-ended code and architectural structure. This realisation suggests that I may find increased confidence in adopting a combined approach for future projects. Considering this, a language like Python could be a more suitable choice for a problem of this nature.

**Testing**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **DESCRIPTION** | **EXPECTED RESULTS** | **ACTUAL RESULTS** | **PASS / FAIL** |
| T01 | Load Data from a Valid File | The program loads the file into the mapdata variable and displays the menu with no errors. The program then displays the menu. | The program successfully loads the file into the mapdata variable and displays the menu with no errors. The program then displays the menu. | **Pass** |
| T02 | Load Data from an Invalid File | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. | The Try cause catches the error and prints an error message on the console. The program then displays the menu. | **Pass** |
| M1 | Run the application and choose type 1 to choose option 1 from the menu – “Get current price for each food” | Information is printed on the console in the format {food}: {currentPrice} for each food item in mapdata | The console displays message in the format {food}: {currentPrice} for each food item in mapdata, using the last value from the data.txt list for each food | **Pass** |
| M2 | Run the application and choose type 2 to choose option 2 from the menu – “Get highest and lowest prices withing the last 24 months for each food” | Information is printed on the console in the format {food}: Highest Price - {highestPrice}, Lowest Price {lowestPrice} for each food item in mapdata | Information is printed on the console in the format {food}: Highest Price - {highestPrice}, Lowest Price {lowestPrice} for each food item in mapdata, using the values from the last 24 months. | **Pass** |
| M3 | Run the application and choose type 3 to choose option 3 from the menu – “Get median price for each food” | Information is printed on the console in the format {food}: Median Price - {medianPrice} for each food item in mapdata in decimal format | Information is printed on the console in the format {food}: Median Price - {medianPrice} for each food item in mapdata in decimal format in decimal format, using the values from the last 24 months. | **Pass** |
| M4 | Run the application and choose type 4 to choose option 4 from the menu – “Get the food with the biggest rise in price” | The item with the highest number (highest rise in price) is displayed with the value of increase next to it (rounded to 2 decimal points) in a format: The food with the largest average rise over the last 6 months is: {food} with a rise of {priceRiseValue}. In this case the food should be Chicken and the raise should be 29.67 | On the console the following text is printed: “The food with the largest average rise over the last 6 months is: CHICKEN with a rise of 29.67” | **Pass** |
| M5 | Run the application and choose type 5 to choose option 5 from the menu – “Compare 2 foods by average price”. Then type 2 valid products in **upper** case, for example “TOMATO” and “CHICKEN” | Information is printed on the console in the format Average value for {food} -> {averageValue}  Average value for {food} -> {averageValue}  {foodWithHigherValue} has a higher average value for the last 24 months. | On the console the following text is printed: “Average value for TOMATO -> 223.96  Average value for CHICKEN -> 738.33  CHICKEN has a higher average value for the last 24 months.” | **Pass** |
| M6 | Run the application and choose type 5 to choose option 5 from the menu – “Compare 2 foods by average price”. Then type 2 valid products in **lower** case, for example “tomato” and “chicken” | Information is printed on the console in the format Average value for {food} -> {averageValue}  Average value for {food} -> {averageValue}  {foodWithHigherValue} has a higher average value for the last 24 months. | On the console the following text is printed: “Average value for TOMATO -> 223.96  Average value for CHICKEN -> 738.33  CHICKEN has a higher average value for the last 24 months.” | **Pass** |
| M7 | Run the application and choose type 5 to choose option 5 from the menu – “Compare 2 foods by average price”. Then type 2 valid products in **mixed** case, for example “ToMaTo” and “cHiCkEn” | Information is printed on the console in the format Average value for {food} -> {averageValue}  Average value for {food} -> {averageValue}  {foodWithHigherValue} has a higher average value for the last 24 months. | On the console the following text is printed: “Average value for TOMATO -> 223.96  Average value for CHICKEN -> 738.33  CHICKEN has a higher average value for the last 24 months.” | **Pass** |
| M8 | Run the application and choose type 5 to choose option 5 from the menu – “Compare 2 foods by average price”. Then type 2 invalid products, for example “CHICKEN” and “TOFU” | A warning message is printed on the console. And the program does not calculate anything. | “Warning: One or both foods not found.” is displayed on the console. | **Pass** |
| M9 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 3 valid products in **upper** case and a positive number in either decimal or integer format, with a whitespace between the food name and the price, for example “RICE 2.5 BEEF 1 APPLE 1.5” | Information is printed on the console in the format: Total basket value: {basketValue} | On the console the following text is printed: “Total basket value: 1490.0” | **Pass** |
| M10 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 3 valid products in **lower** case and a positive number in either decimal or integer format, with a whitespace between the food name and the price, for example “rice 2.5 beef 1 apple 1.5” | Information is printed on the console in the format: Total basket value: {basketValue} | On the console the following text is printed: “Total basket value: 1490.0” | **Pass** |
| M11 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 3 valid products in **mixed** case and a positive number in either decimal or integer format, with a whitespace between the food name and the price, for example “RiCe 2.5 bEeF 1 aPPle 1.5” | Information is printed on the console in the format: Total basket value: {basketValue} | On the console the following text is printed: “Total basket value: 1490.0” | **Pass** |
| M12 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 3 valid products and a **negative** number in either decimal or integer format, with a whitespace between the food name and the price, for example “rice 2.5 beef -1 apple 2.5” | The application should not proceed with calculations and display a warning message that weight cannot be a negative number | On the console the following text is printed: “Total basket value: -372.0” | **Fail** |
| M13 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 3 valid products and a positive number in either decimal or integer format, with a whitespace between the food name and the price, except for one of the food items, where a number would not be provided after it, for example “rice 2.5 beef apple 2.5” | The application should not proceed with calculations and display a warning message that each food item should be followed by a number in decimal or integer format and the number should be positive. | The application crashes as the function tries to parse the food “apple” for the weight of the beef, leading to NumberFormatException error. | **Fail** |
| M14 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 3 valid products and a positive number in either decimal or integer format, with a whitespace between the food name and the price, except for the last food item, where a number would not be provided after it, for example “rice 2.5 beef 1 apple ” | The application should not proceed with calculations and display a warning message that each food item should be followed by a number in decimal or integer format and the number should be positive. | On the console the following text is printed: “Total basket value: 1068.5”. The application ignores the last item and calculates the first 2 food items in the basket. | **Fail** |
| M15 | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 2 valid and 1 invalid product and a positive number in either decimal or integer format, with a whitespace between the food name and the price, for example “rice 2.5 beef 1 apple 1.5 tofu 2.5 coffee 2.5” | A warning message for each unrecognised food should be printed in the format of:  “Warning: Unrecognised food '{food}', skipping.”.  Then after iterating through all items a warning messaged with all unrecognised food in the basket should be printed in the format of:  “Warning: Basket contains unrecognised foods: {food1}, {food2}.”.  Then a message is printed on the console in the format: Total basket value: {basketValue} | On the console the following text is printed:  “Warning: Unrecognised food 'COFFEE', skipping.  Warning: Unrecognised food 'TOFU', skipping.  Warning: Basket contains unrecognised foods: COFFEE, TOFU.  Total basket value: 1490.0” | **Pass** |
|  | Run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then press enter without inputting any food items. | The application should display the total value of the basket as 0.0 as there are no items, no warning message or error will be displayed. | On the console the following text is printed:  “Total basket value: 0.0” | **Pass** |
| M17 | Run the application and choose type 7 to choose option 7 from the menu – “Quit” | The message “Selected Quit” should be printed on the console and the Application should exit with code 0. | “Selected Quit” is printed on the console and the Application successfully exits withs code 0. | **Pass** |
| M18 | Load Data from an Invalid File then run the application and choose type 1 to choose option 1 from the menu – “Get current price for each food” | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. As the mapdata is empty an empty string is expected to be printed upon choosing the option. | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. An empty string is printed on the screen as no data is available. | **Pass (Might be fail depending on if the user wants the application to stop after data fail to load)** |
| M19 | Load Data from an Invalid File then run the application and choose type 2 to choose option 2 from the menu – “Get highest and lowest prices withing the last 24 months for each food” | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. As the mapdata is empty an empty string is expected to be printed upon choosing the option. | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. An empty string is printed on the screen as no data is available. | **Pass (Might be fail depending on if the user wants the application to stop after data fail to load)** |
| M20 | Load Data from an Invalid File then run the application and choose type 3 to choose option 3 from the menu – “Get median price for each food” | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. As the mapdata is empty an empty string is expected to be printed upon choosing the option. | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. An empty string is printed on the screen as no data is available. | **Pass (Might be fail depending on if the user wants the application to stop after data fail to load)** |
| M21 | Load Data from an Invalid File then run the application and choose type 4 to choose option 4 from the menu – “Get the food with the biggest rise in price” | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. As the mapdata is empty an empty string is expected to be printed upon choosing the option. | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. The Application crashes upon choosing option 4 as the function maxBy is called to iterate on an empty collection. | **Fail** |
| M22 | Load Data from an Invalid File then run the application and choose type 5 to choose option 5 from the menu – “Compare 2 foods by average price”. Then type 2 products, for example “Tomato” and “Chicken”. | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. As the mapdata is empty, a warning message is printed on the console, that one or both foods have not been found. And the program does not calculate anything. | The program cannot load the file into the mapdata variable and an error message on the console. The program then displays the menu. The console prints the message: “Warning: One or both foods not found.”. | **Pass** |
| M23 | Load Data from an Invalid File then run the application and choose type 6 to choose option 6 from the menu – “Calculate Food Basket”. Then type 2 products and a positive number in either decimal or integer format, with a whitespace between the food name and the price, for example “rice 1 beef 1” | A warning message that both the items have not been recognised should be displayed in the format of:  “Warning: Unrecognised food '{food}', skipping.”.  Then warning messaged with all unrecognised food in the basket should be printed in the format of:  “Warning: Basket contains unrecognised foods: {food1}, {food2}.”.  Then a message is printed on the console in the format: Total basket value: 0.0 (as the basket will always be empty) | On the console the following text is printed:  “Warning: Unrecognised food 'RICE', skipping.  Warning: Unrecognised food 'BEEF', skipping.  Warning: Basket contains unrecognised foods: RICE, BEEF.  Total basket value: 0.0” | **Pass** |
| M24 | Load Data from an Invalid File then run the application and choose type 7 to choose option 7 from the menu – “Quit” | The message “Selected Quit” should be printed on the console and the Application should exit with code 0. | “Selected Quit” is printed on the console and the Application successfully exits withs code 0. | **Pass** |