

Programming paradigms Coursework 1

Kieran Harley



December 12, 2023

Glasgow Caledonian University

S2214057

Contents

[Table of Figures 1](#_Toc153299001)

[Test Plan 2](#_Toc153299002)

[Testing Evidence Figures 3](#_Toc153299003)

[Evaluation 11](#_Toc153299004)

[Test Evaluation 11](#_Toc153299005)

[Functional Thinking 11](#_Toc153299006)

[Functional Programming Style 11](#_Toc153299007)

[Comparison of Functional and Imperative Style 12](#_Toc153299008)

[Different Language 12](#_Toc153299009)

# Table of Figures

[Figure 1: Test Figure 1 3](#_Toc153284926)

[Figure 2: Test Figure 2 4](#_Toc153284927)

[Figure 3: Test Figure 3 4](#_Toc153284928)

[Figure 4: Test Figure 4 5](#_Toc153284929)

[Figure 5: Test Figure 5 5](#_Toc153284930)

[Figure 6: Test Figure 6 5](#_Toc153284931)

[Figure 7: Test Figure 7 6](#_Toc153284932)

[Figure 8: Test Figure 8 6](#_Toc153284933)

[Figure 9: Test Figure 9 6](#_Toc153284934)

[Figure 10: Test Figure 10 7](#_Toc153284935)

[Figure 11: Test Figure 11 7](#_Toc153284936)

[Figure 12: Test Figure 12 8](#_Toc153284937)

[Figure 13: Test Figure 13 8](#_Toc153284938)

[Figure 14: Test Figure 14 8](#_Toc153284939)

[Figure 15: Test Figure 15 9](#_Toc153284940)

[Figure 16: Test Figure 16 9](#_Toc153284941)

[Figure 17: Menu system for reference. 10](#_Toc153284942)

# Test Plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test case | Test name | Input | Expected output | Actual Output |
| 1.1 | Access to datafile | val datafile = "src/data.txt" | File is now ready to be read | Works |
| 1.2 | Read datafile | val foodData = Source.fromFile(datafile).mkString | File is read and data can be interprited by application. | Application progresses successfully, and the data is read. |
| 2.1 | Menu | 1 | Application shows current prices | Works, see test figure 1 |
| 2.2 | Menu | 9000 | Application rejects the option but menu re runs. | Works, see test figure 2 |
| 2.3 | Menu | Text | Application rejects the option but menu re runs. | Application breaks, with numberformatexception, see test figure 3 |
| 3.1 | Current prices | Shows current prices when 1 is entered | Application shows current prices. | Works, see test figure 1 |
| 4.1 | Highest / lowest | Shows highest and lowest prices when 2 is entered at menu. | Application shows highest and lowest prices. | Works, see test figure 4 |
| 5.1 | Median | Shows median prices when 3 is entered. | Application shows the median prices of the food items. | Works, see test figure 5. |
| 6.1 | Rising most | Calls risen() when 4 is entered at menu | Application shows which item has risen the most in the last 6 months. | Works, see test figure 6. |
| 7.1 | Compare two averages | Calls averages() when 4 is entered at menu | Application shows a list of averages, and prompts user to select 2 to compare | Works see test figure 7 |
| 7.2 | Compare two averages | APPLE, BEEF | Application takes two inputted values as a request and shows the comparison of the two. | Works see test figure 8 |
| 7.3 | Compare two averages | Apple, Beef | Application rejects two choices as they are not uppercase and app employs recursion | App does not use recursion, but error is handled, and user friendly notice is displayed. See test figure 9 |
| 8.1 | Food basket | Calls basket() when 6 is entered at menu | Application shows the current values, and prompts user to enter an item to add to baket. | Works, see test figure 10. |
| 8.2 | Food basket | Add items to basket | Item and quantity is taken from user and then added to the basket, then user is asked if they want to continue | Works, see test figure 11 |
| 8.3 | Food basket | Enter y to keep adding items to basket | Recursion with same menu as before to add another item | Does not display menu again, but does prompt user again to add item to basket. See test figure 12 |
| 8.4 | Food basket | Enter n to stop adding | Application stops and then basket summary is printed | Works, see test figure 13. |
| 8.5 | Food basket | Enter food item when you should enter amount | Friendly reminder that this is the wrong type of data for this menu. | App breaks with numberformatexception. Can be fixed with exception handling. See test figure 14. |
| 8.6 | Food basket | Enter food item capitalised instead of uppercase. | Friendly reminder that this is the wrong type of input and app employs recursion to allow user to go again. | Works, see test figure 15. |

## Testing Evidence Figures

A screenshot of a computer

Description automatically generated

Figure : Test Figure 1

A screen shot of a computer

Description automatically generated

Figure : Test Figure 2

A screenshot of a computer program

Description automatically generated

Figure : Test Figure 3

A screen shot of a computer

Description automatically generated

Figure : Test Figure 4

A screen shot of a computer

Description automatically generated

Figure : Test Figure 5

A screen shot of a computer

Description automatically generated

Figure : Test Figure 6

A screenshot of a computer

Description automatically generated

Figure : Test Figure 7

A screen shot of a computer

Description automatically generated

Figure : Test Figure 8

A screenshot of a computer

Description automatically generated

Figure : Test Figure 9

A screen shot of a computer

Description automatically generated

Figure : Test Figure 10

A screenshot of a computer

Description automatically generated

Figure : Test Figure 11

A screenshot of a computer

Description automatically generated

Figure : Test Figure 12

A screen shot of a computer

Description automatically generated

Figure : Test Figure 13

A screenshot of a computer screen

Description automatically generated

Figure : Test Figure 14

A screen shot of a computer program

Description automatically generated

Figure : Test Figure 15

A screen shot of a computer

Description automatically generated

Figure : Test Figure 16

A screen shot of a computer

Description automatically generated

Figure : Menu system for reference

# Evaluation

## Test Evaluation

Unit testing was chosen to record the testing process of this application, as it provides an overview of expected outcomes against the input from the user and then compares them to the actual. This method allows extreme, normal and exceptional use cases to be trialed, to cover all bases, ensuring the application is robust and works as expected.

After analysing the test cases, while there are some things that can be amended, the conclusion is that this application’s development has been a net success. The menu system is simple and clear, and there has been effective implementation of programming methodology to use the information stored within the data.txt file. In extension, the application can perform well against the requirements of producing a list of the current prices, can source the highest and lowest prices of each food item, the median price of each food item, check which food item has risen the most in the last six months, compare the averages of two selected food items, and finally allow the user to add items to a basket and then produce values by how much of the item is ordered (by float) and then print the items that were added to the basket.

## Functional Thinking

Looking at the code, there are many examples of functional principals being incorporated in this application. Firstly, there is an immediate use of recursion to allow the programme to run. This is implemented by a do while loop. This if effectively implemented to provide a menu system the user can choose from. The do while loop is an appropriate choice of functional programming, as this will ensure the menu always runs at least once, and while not receiving an appropriate input, employs recursion to rerun the menu until an acceptable input is taken.

Another example of functional programming is, pattern-matching. This is used within the foreach loops within the application, by way of head and tail (and last) to ensure there is a matching entity within the data.txt file that can be extracted to the application to report a food item and corresponding value to the user. Furthermore, immutable data structures such as Map and ListMap complement pattern matching concepts by maintaining the order of element, this is prevalent in the highlow() method, though this method is commonly recruited throughout the application.

The use of flatMap in the averages() method uses functional composition to combine simpler functions to create more complex variations. This method makes use of this by using the Option monad to represent potential missing values. By using flatMap this lets the application iterate over the data in order to retrieve the name of the food item and the relevant price, before then iterating again after receiving input to compare two averages with each other.

## Functional Programming Style

In this application the use of foreach loops where chosen over the use of a foldleft technique. This decision was made as it could be implemented in an immutable way, ensuring that the collected data from the text file was not modified anyway, instead applying a function to the elements within the file. The foreach loop can also print to console against an input, and therefore provided a slight advantage over foldleft. Foldleft is another good way of interpreting data, however having more confidence using a foreach loop further contributed to the decision to use this method over folds as well as these higher order functions allowing for a more declarative style.

One thing that should have been better implemented was the use of exception handling. In this application, exception handling only exists in a very primitive way, by use of for and do-while loops, instead of explicitly handling common exceptions such as input mismatch and number format exceptions. This would only contribute to a more robust application capable of handling any foreseeable objection without allowing the application to break. Aligning with functional principals, exception handling could have been implemented with ‘either monads’ to let the system proceed if true or handle exception if false.

## Comparison of Functional and Imperative Style

While an imperative style of programming has it’s benefits, the functional style of programming was far superior for this project. This is for many reasons, firstly, a functional programming style allows for greater modularity. This application requires a menu system, with that, the ability to break the code into many functions, allows for more clarity at development time, and made it easier to debug. Having many functions also encourages the use of recursion, as methods can easily be called. Functional further eased the development of this application by way of data structures. The use of these structures complemented the way the code interpreted the data.txt file by using high order functions.

## Different Language

Scala showed that multi-paradigm languages can be very useful when developing these types of applications. With that in mind, if there was a choice at picking another language, it would be wise to pick another with multi-paradigm. After consideration, Python would be a strong suggestion as this would allow for a similar blend of functional and imperative programming styles, while also allowing the developer to code in, what is considered to be, a more high-level language. Python is also renowned for having a great amount of libraries that can be made use of to further assist with the development process.