Programming Paradigms Coursework

Alastair Kilgour – S2221119

Contents

[Testing 2](#_Toc152843740)

[Main Menu 2](#_Toc152843741)

[Average Price Comparison Menu 5](#_Toc152843742)

[Basket Building Menu 8](#_Toc152843743)

[Evaluation 10](#_Toc152843744)

# Testing

## Main Menu

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Test Case** | **Expected Output** | **Actual Output** | **Action Taken** |
| S1 | Start the app to open the console and load data. | 1: Current Prices  2: Highest and Lowest Prices  3: Median Prices  4: Largest Price Rise  5: Compare Average Prices  6: Build Food Basket  0: Quit  Choose an option: | 1: Current Prices  2: Highest and Lowest Prices  3: Median Prices  4: Largest Price Rise  5: Compare Average Prices  6: Build Food Basket  0: Quit  Choose an option: | None – output as expected |
| S2 | Type “0” and click ENTER to quit the application. | Application closes. | Application closes. | None – output as expected |
| S3 | Type “1” and click ENTER to view current prices. | Displaying current prices:  APPLE -> 281p  BUTTER -> 739p  POTATO -> 73p  FLOUR -> 62p  RICE -> 55p  MILK -> 71p  TOMATO -> 294p  OIL -> 2084p  BEEF -> 931p  CHICKEN -> 789p | Displaying current prices:  APPLE -> 281p  BUTTER -> 739p  POTATO -> 73p  FLOUR -> 62p  RICE -> 55p  MILK -> 71p  TOMATO -> 294p  OIL -> 2084p  BEEF -> 931p  CHICKEN -> 789p | None – output as expected |
| S4 | Type “2” and click ENTER to view highest and lowest prices of each product. | APPLE -> Highest: 281p, Lowest: 200p  BUTTER -> Highest: 797p, Lowest: 660p  POTATO -> Highest: 137p, Lowest: 55p  FLOUR -> Highest: 66p, Lowest: 41p  RICE -> Highest: 79p, Lowest: 43p  MILK -> Highest: 117p, Lowest: 71p  TOMATO -> Highest: 316p, Lowest: 159p  OIL -> Highest: 2484p, Lowest: 1823p  BEEF -> Highest: 1299p, Lowest: 927p  CHICKEN -> Highest: 881p, Lowest: 602p | APPLE -> Highest: 281p, Lowest: 200p  BUTTER -> Highest: 797p, Lowest: 660p  POTATO -> Highest: 137p, Lowest: 55p  FLOUR -> Highest: 66p, Lowest: 41p  RICE -> Highest: 79p, Lowest: 43p  MILK -> Highest: 117p, Lowest: 71p  TOMATO -> Highest: 316p, Lowest: 159p  OIL -> Highest: 2484p, Lowest: 1823p  BEEF -> Highest: 1299p, Lowest: 927p  CHICKEN -> Highest: 881p, Lowest: 602p | None – output as expected |
| S5 | Type “3” and click ENTER to view median prices of each product. | APPLE -> 225p  BUTTER -> 737p  POTATO -> 94p  FLOUR -> 58p  RICE -> 63p  MILK -> 92p  TOMATO -> 221p  OIL -> 2159p  BEEF -> 1154p  CHICKEN -> 726p | APPLE -> 225p  BUTTER -> 737p  POTATO -> 94p  FLOUR -> 58p  RICE -> 63p  MILK -> 92p  TOMATO -> 221p  OIL -> 2159p  BEEF -> 1154p  CHICKEN -> 726p | None – output as expected |
| S6 | Type “4” and click ENTER to view the product with the largest price rise in the last 6 months. | Product with highest price rise in 6 months: OIL with a rise of 202p | Product with highest price rise in 6 months: OIL with a rise of 202p | None – output as expected |
| S7 | Type “5” and click ENTER to view menu to select first product to compare averages of. | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  Choose an option: | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  Choose an option: | None – output as expected |
| S8 | Type “6” and click ENTER to view menu to select first product to add to basket. | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  0: Evaluate Basket  Choose an option: | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  0: Evaluate Basket  Choose an option: | None – output as expected |
| E1 | Type “7” and click ENTER to test invalid selection handling. | Invalid selection! | Error thrown: no match case found. | Changed “maxChoice” variable passed to “getMenuResponse” function to “6” from “7” |
| E1.2 | Type “7” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E2 | Type “-1” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E3 | Type “one” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E4 | Provide no input and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |

## Average Price Comparison Menu

All cases assume tester has already navigated to the “Compare Average Prices” function per the steps in test S5.  
**\*** Case assumes tester has selected a valid first option when prompted to.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S9-1 | Type “1” to select “APPLE” and click ENTER. | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  Choose an option: | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  Choose an option: | None – output as expected |
| S9-2 | Type “2” to select “BUTTER” and click ENTER | APPLE has an average price of 235p  BUTTER has an average price of 734p  There is a difference of 499p | APPLE has an average price of 235p  BUTTER has an average price of 734p  There is a difference of 499p | None – output as expected |
| S10-1 | Type “10” to select “CHICKEN” and click ENTER | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  Choose an option: | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  Choose an option: | None – output as expected |
| S10-2 | Type “9” to selected “BEEF” and click ENTER | CHICKEN has an average price of 738p  BEEF has an average price of 1140p  There is a difference of 402p | CHICKEN has an average price of 738p  BEEF has an average price of 1140p  There is a difference of 402p | None – output as expected |
| E5 | Type “0” and click ENTER to test invalid selection handling. | Invalid selection! | Error thrown: index out of bounds. | Added a “minChoice” parameter to “getMenuResponse” function to stop users entering a value below that value. |
| E5.2 | Type “0” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E6 | Type “11” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E7 | Type “one” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E8 | Provide no input and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E9 | **\*** Type “0” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E10 | **\*** Type “11” to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E11 | **\*** Type “one” to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected. |
| E12 | **\*** Provide no input and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected. |

## Basket Building Menu

All cases assume tester has already navigated to the “Build Food Basket” function per the steps in test S6.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S11-1 | Type “1” and click ENTER, then type 3 and click ENTER to add 3kg of APPLE to basket. | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  0: Evaluate Basket  Choose an option: | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  0: Evaluate Basket  Choose an option: | None – output as expected |
| S11-2 | Type “10” and click ENTER, then type 1 and click ENTER to add 1kg of CHICKEN to basket. | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  0: Evaluate Basket  Choose an option: | 1: APPLE  2: BUTTER  3: POTATO  4: FLOUR  5: RICE  6: MILK  7: TOMATO  8: OIL  9: BEEF  10: CHICKEN  0: Evaluate Basket  Choose an option: | None – output as expected |
| S11-3 | Type “0” and click ENTER to view details about the basket. | Your basket costs 1632p.  It contains 2 items. | Invalid selection! | Changed “minChoice” variable to 0 from 1 in “getMenuCResponse |
| E13 | Type “11” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E14 | Type “-1” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E15 | Type “APPLE” and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |
| E16 | Provide no input and click ENTER to test invalid selection handling. | Invalid selection! | Invalid selection! | None – output as expected |

# Evaluation

## Functional Thinking and Programming Style

The idea of functional programming is that as much data as possible is kept immutable, i.e. its state cannot be changed. This helps avoid unexpected state changes which could lead to issues in the application later down the line. This is done through the use of functions which can be used to evaluate a single variable, thus reducing the chances of other data being changed in the process. My application utilises various functional programming techniques in its implementation.

The most commonly used technique was to use Scala’s “val” keyword, short for “value”. This marks the variable as immutable, meaning its value cannot be changed after the line where it is declared. This was utilised in places where a value had to be retrieved from a function’s return, but afterwards did not need to be changed.

The second most commonly used technique was tail recursion. This involves calling a function within itself to carry out an operation on a variable multiple times until a condition is met. This is especially useful when carrying out operations on a collection, such as printing all the product keys to the console when creating the basket menu.  
One of the issues associated with tail recursion is that the application will stack instances of functions on top of each other until the exit condition is met. Whilst on small collections, such as those implemented by this application, this is fine, on larger collections it can lead to the device running out of memory.  
An alternative approach to tail recursion would have been to use the “foreach” function available to most collections in Scala. This uses lambda expressions to create what is known as an anonymous function which is then carried out on each item within the collection. This can help avoid issues with the device running out of memory present in tail recursion.

The idea of anonymous functions, or functors, is to pass a function into a higher-order function implemented into the programming language, allowing for operations on immutable data. These are used in a number of places throughout the application. Most notably, they are used in the “breakdownData” function to remove any whitespace from each price before converting them into integers.  
Another place they are used is in the various “match” functions used throughout the program to evaluate a user’s input. For example: if the user enters “1” on the main menu, the application selects the corresponding case and runs the code contained within as if it were a function of its own.

Another functional idea implemented is the idea of “monads”. A monad is the context which contains a function.  
Scala’s built-in “Option” monad was used to determine if the application successfully read in data from the file. The idea is the “getData” function returns an option which either has the value of “Some” or “None”. If it returns “Some”, the data is contained inside the Option context and can be extracted and saved in the “appData” map. On the other hand, if the function returns “None”, the applications knows something went wrong and there is no data available, so displays an error message and closes.

## Functional vs Imperative Styles

Whilst the idea of functional programming is to maintain the state of the program throughout, the idea of imperative is to use statements which actively change the program’s state.

If given a choice of programming languages, I would use the imperative C# over a functional alternative. C# is a class-based object-oriented programming language, meaning it utilises a class system to store the data in an object with predetermined variables. It does, however, utilise a few functional principles to a limited extent, such as anonymous functions.  
When the application is first loaded, instead of loading all the data into a map, it can be loaded into objects with each containing all the information about each product. This means that instead of having to extract and match keys when looking for a specific value, utilising anonymous functions to search for an object with a matching product name would require only a single line of code as opposed to the minimum of three required when using Scala functionally.  
Another advantage would be the use of traditional loops as opposed to tail recursion. This avoids memory issues caused by tail recursion and cuts out the various “if else” statements needed in each recurring function to check if the exit condition is met.  
Yet another advantage is the ability to use exceptions within the code. In functional programming, only raw errors exist. However, exceptions contain both the raw error and further information, such as custom error messages, which can be used to both cut down the amount of code needed, and to provide more information on what went wrong.

One of the biggest advantages of using C# over functional Scala is readability. Using functional Scala, every operation is spread across one or more functions which leads to a vastly increased amount of code and makes it harder to keep a track of the logic behind what is happening. Whilst each individual function may be readable, the application as a whole is not.  
With C#, a single operation can be completed in a single function in fewer lines of code. This makes the logic far simpler to keep track of, and reduces the bloating of code only necessary for the internal logic of the program as opposed to carrying out an operation.