# Functional Thinking

Functional Programming is an approach that emphasises the use of functions and expressions, treating computation as the evaluation of mathematical functions and avoiding state changes and mutable data. Immutability is an important concept in functional programming because state changes can cause unexpected behaviour. The solution that I created demonstrates adherence to several functional programming principles through the consistent use of immutable data structures, higher-order functions, and pure functions.

Immutable data structures, such as the ‘val’ keyword for constant variables (See Figure 1) and the use of the ‘Map’ collection (See Figure 2), ensures that once data is assigned, it remains unaltered.

A computer screen with text

Description automatically generated

Figure 1: Demonstrating the use of the val constant

A screen shot of a computer program

Description automatically generated

Figure 2: Demonstrating the use of the Map data structure

Higher-order functions are employed throughout the code, examples being use of ‘map’ ‘foreach’, and ‘filter’ on collections like list. For instance, the GetCurrentPrices and GetMinMaxPrices functions utilise the map function to transform the input data immutably (See Figure 3 and 4). Moreover, the GetMedianPrices function employs a pure function to calculate the median prices for each food item based on a sorted list of prices (See Figure 5). This emphasis on pure functions, which produce deterministic outputs based solely on their inputs, contributes to the functional style of the solution. Additionally, the use of recursion in the MainMenu function annotated with ‘@tailrec’ for tail call optimisation (See Figure 6), showcases an attempt to embrace functional programming practices in control flow.

A computer screen shot of text

Description automatically generated

Figure 3: Demonstrating the use of the map higher order function

A computer code with text

Description automatically generated with medium confidence

Figure 4: Demonstrating the use of the map higher order function 2

A computer screen shot of a program code

Description automatically generated

Figure 5: Demonstrating the use of a pure function

A screenshot of a computer program

Description automatically generated

Figure 6: Demonstrating the use of the tailrec identifier

# Programming Style

Functional programming styles tend to lean towards readability of individual function although when you take a step back, keeping track of the flow of operations can be quite tricky. Which is why my attempt tried to implement pattern matching, string interpolation, and lambda expressions where possible.

## Pattern Matching

Pattern matching is primarily used to replace if else statements where there are more than 2 conditions, or where more conditions may be added in the future. It is a powerful feature in programming that allows you to match complex data structures or values against a set of patterns. Providing a concise and readable way to destructure and analyse data, making the code more expressive and often eliminating the need for lengthy if-else sequences (See Figure 7 and 8). Although both the match and if-else sequences are similar in the number of lines, pattern matching requires significantly less characters to create a case which can add up for the developer’s time rather quickly. It even improves compile times as pattern matching is far more efficient, and the compiler can even warn the developer of an incorrect pattern.

|  |  |
| --- | --- |
| A screen shot of a computer program  Description automatically generated | A screen shot of a computer program  Description automatically generated |
| Figure 7: Pattern matching | Figure 8: If-else alternative of Figure 7 |

## String Interpolation

Although string interpolation is not strictly functional and is used in many other languages like C#, it aligns well with functional programming principles by promoting immutability and providing a tool for building strings in a way that is more declarative and less prone to error compared to manual string concatenation.

Scala implements string interpolation incredibly well for the developer improving readability as well as writability (See Figures 9, 10, and 11). My personal preference is for inline string interpolation as it provides the fastest writability as you don’t have to consider the variable as separate from the string whilst it still allowing you to format certain data types like Double whereas it is slightly more difficult to do when using concatenation.



Figure 9: Inline string interpolation



Figure 10: Placeholder string interpolation



Figure 11: Concatenation

## Lambda Expressions

Lambda Expressions or Anonymous Functions or Function Literals are closely associated with functional programming. Lambda expressions are a feature of many programming languages that support functional programming, and they align with several key principles of functional programming. By being anonymous, lambda expressions are able to be used in assignment to variables, passed to other functions, return as values, or return other functions as results. For example, the CalculateBasketTotal function iterates through a map, multiplying each individual item price with the quantity and then eventually summing all results into a total variable (See Figure 12). This could alternatively be extrapolated to a single “Multiply” function that takes in both parameters which would add extra, unnecessary lines of code, while anonymous functions tend to avoid this.

A computer screen shot of text

Description automatically generated

Figure 12: Lambda expression

# Imperative vs Declarative

Personally Imperative programming or more specifically Object-Oriented programming in C# is what I prefer to use when tackling a problem. Object-Oriented programming provides a much more robust and expandable program especially for large scale operations where you may have multiple developers working on the same project. People may prefer Scala to C# because C# generates a lot of boiler plate or redundant code that a developer may never touch but in recent years especially with dot NET 8.0 releasing, a lot of the boiler plate that would be generated can be hidden which can help transition Scala developers to C#.

I did enjoy using Scala and it’s a handy language to know especially if you need to interact with Java but don’t want to use it because it can be clunky at times. I think my brain can process OOP thinking much faster as I was able to create the entire application in C# in less than 2 hours (<https://github.com/jgraham8/ScalaCSharpComparison>) while it took me a few days in Scala.

Not being able to use while and for loops and having to use tail recursive functions was pretty detrimental as I struggled to comprehend why someone would choose that way of development over an imperative style (See Figure 13, and 14 for a comparison).

|  |  |
| --- | --- |
| A screenshot of a computer program  Description automatically generated | A screenshot of a computer program  Description automatically generated |
| Figure 13: C# main menu implementation | Figure 14: Scala main menu implementation |

Similarly, not being able to utilise objects was rather frustrating as I couldn’t separate the programs concerns into different classes and essentially chucked everything onto the one document. This also led to code being less readable. In my C# version I implemented Food, Basket, and BasketLine objects to increase reusability and separate logic into their respective classes (See Figure 15 and 16).

|  |  |
| --- | --- |
| A screen shot of a computer program  Description automatically generated | A screen shot of a computer program  Description automatically generated |
| Figure 15: Food Object | Figure 16: Basket & BasketLine Object |

Realistically I’d never use purely imperative or declarative as I use LINQ fairly often to manipulate collections when creating C# applications, but I’d rather stay away from immutability present in functional programming as I prefer being able to manipulate existing variables and it can be frustrating having to create recursive functions to deal with data manipulation.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Description** | **Input Data** | **Expected Result** | **Actual Result** | **Pass / Fail** |
| APP001 | Open application with data file present |  | 1. Get current price for each food  2. Get highest and lowest prices for each food  3. Get median price for each food  4. Get the symbol for the food which has risen most over the last 6 months  5. Compare average values of two foods  6. Calculate basket total  7. Quit  Enter your choice: | 1. Get current price for each food  2. Get highest and lowest prices for each food  3. Get median price for each food  4. Get the symbol for the food which has risen most over the last 6 months  5. Compare average values of two foods  6. Calculate basket total  7. Quit  Enter your choice: | **PASS** |
| APP002 | Open application without data file present |  | File Manipulation Exception Occurred: java.io.FileNotFoundException: data.txt (The system cannot find the file specified)  Exiting Application...  Process finished with exit code 0 | File Manipulation Exception Occurred: java.io.FileNotFoundException: data.txt (The system cannot find the file specified)  Exiting Application...  Process finished with exit code 0 | **PASS** |
| MEN001 | Use invalid input in main menu | -1,  8,  “one”,  “%”,  “asfasf” | [Result from test APP001] +  Incorrect Selection +  [Reprinting result from test APP001] | A screenshot of a computer program  Description automatically generated | **PASS** |
| MEN002 | Exit Application | 7 | [Result from test APP001] +  Exiting Application...  Process finished with exit code 0 | A screenshot of a computer program  Description automatically generated | **PASS** |
| CUR001 | Display current price of each food | 1 | [Result from test APP001] +  Current Prices  APPLE = £2.81  BUTTER = £7.39  POTATO = £0.73  FLOUR = £0.62  RICE = £0.55  MILK = £0.71  TOMATO = £2.94  OIL = £20.84  BEEF = £9.31  CHICKEN = £7.89 | A screenshot of a computer  Description automatically generated | **PASS** |
| MiM001 | Get highest and lowest prices for each food | 2 | [Result from test APP001] +  Min Max Prices  APPLE: Max = £2.81 | Min = £2.00  BUTTER: Max = £7.97 | Min = £6.60  POTATO: Max = £1.37 | Min = £0.55  FLOUR: Max = £0.66 | Min = £0.41  RICE: Max = £0.79 | Min = £0.43  MILK: Max = £1.17 | Min = £0.71  TOMATO: Max = £3.16 | Min = £1.59  OIL: Max = £24.84 | Min = £18.23  BEEF: Max = £12.99 | Min = £9.27  CHICKEN: Max = £8.81 | Min = £6.02 | A screenshot of a computer  Description automatically generated | **PASS** |
| MED001 | Get median price for each food | 3 | [Result from test APP001] +  Median Prices  APPLE = £2.25  BUTTER = £7.37  POTATO = £0.94  FLOUR = £0.58  RICE = £0.63  MILK = £0.92  TOMATO = £2.21  OIL = £21.59  BEEF = £11.54  CHICKEN = £7.26 |  | **PASS** |
| RIS001 | Get the symbol for the food which has risen most over the last 6 months | 4 | [Result from test APP001] +  Rising Food  OIL: rose by £2.02 | A screenshot of a computer program  Description automatically generated | **PASS** |
| AVG001 | Compare average values of two foods | 5,  1,  10 | [Result from test APP001] +  Compare Average  1. APPLE  2. BUTTER  3. POTATO  4. FLOUR  5. RICE  6. MILK  7. TOMATO  8. OIL  9. BEEF  10. CHICKEN  Enter your choice: 1  Enter your choice: 10  APPLE: Avg = £2.35 | CHICKEN Avg = £7.38 | There is a difference of £5.03 | A screenshot of a computer program  Description automatically generated | **PASS** |
| AVG002 | Compare average values of two foods | 5,  8,  9 | [Result from test APP001] +  Compare Average  1. APPLE  2. BUTTER  3. POTATO  4. FLOUR  5. RICE  6. MILK  7. TOMATO  8. OIL  9. BEEF  10. CHICKEN  Enter your choice: 8  Enter your choice: 9  OIL: Avg = £21.31 | BEEF Avg = £11.40 | There is a difference of £9.91 | A screenshot of a computer  Description automatically generated | **PASS** |
| AVG003 | Test exceptional menu input | 5,  -1,  11,  “one”,  “%”,  “asfasf” | Selection must be between 1 – 10 /  Selection must be an integer | A screenshot of a computer program  Description automatically generated | **PASS** |
| BAS001 | Calculate basket total | 6,  1,  13,  11 | [Result from test APP001] +  Basket Menu  1. APPLE £2.81  2. BUTTER £7.39  3. POTATO £0.73  4. FLOUR £0.62  5. RICE £0.55  6. MILK £0.71  7. TOMATO £2.94  8. OIL £20.84  9. BEEF £9.31  10. CHICKEN £7.89  11. Checkout  Enter your choice: 1  Enter the Amount (KG/L): 13  Enter your choice: 11  Total is £36.53 | A screenshot of a computer screen  Description automatically generated | **PASS** |
| BAS002 | Calculate basket total | 6,  1,  0.5,  11 | [Result from test APP001] +  Basket Menu  1. APPLE £2.81  2. BUTTER £7.39  3. POTATO £0.73  4. FLOUR £0.62  5. RICE £0.55  6. MILK £0.71  7. TOMATO £2.94  8. OIL £20.84  9. BEEF £9.31  10. CHICKEN £7.89  11. Checkout  Enter your choice: 1  Enter the Amount (KG/L): 0.5  Enter your choice: 11  Total is £1.41 | A screenshot of a computer screen  Description automatically generated | **PASS** |
| BAS003 | Attempt to check out an empty basket | 6,  11 | Basket is empty, add items to basket to checkout | A screenshot of a computer  Description automatically generated | **PASS** |
| BAS004 | Attempt to input invalid data | 6,  -1,  12,  1,  -1,  0.00,  “asfasf” | Selection must be between 1 – 11 /  Quantity must be greater than 0.00 | A screen shot of a computer  Description automatically generated | **PASS** |