****

## School of Computing, Engineering and Built Environment

## 

## Programming Paradigms - MHI325688

## Coursework Evaluation

## 

## Ewan Allison - S2004763

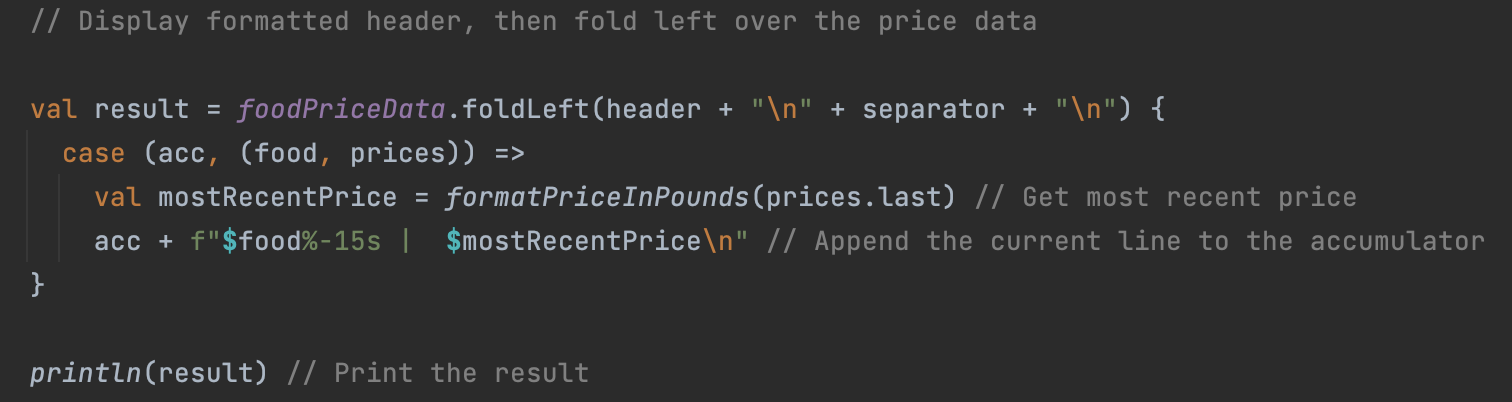
**Introduction**

This coursework was to develop a food prices application using Scala, a language known for supporting functional programming principles. The application uses a provided dataset of food items and their historical prices to provide functionalities such as displaying recent food prices, comparing average prices over 12 months, and enabling users to create a “basket” with various foods and quantities. Developed with a focus on functional thinking and programming style, this application exemplifies how functional concepts can be applied in real-world applications and scenarios.

This evaluation will highlight the functional programming techniques used in the application. Specific code examples will be provided to illustrate these concepts in practice, along with a commentary on their impact on the development process and the quality of the solution. Additionally, a comparison will be drawn between functional and imperative programming styles, and it will be explored how the different approaches tackle the same problem.

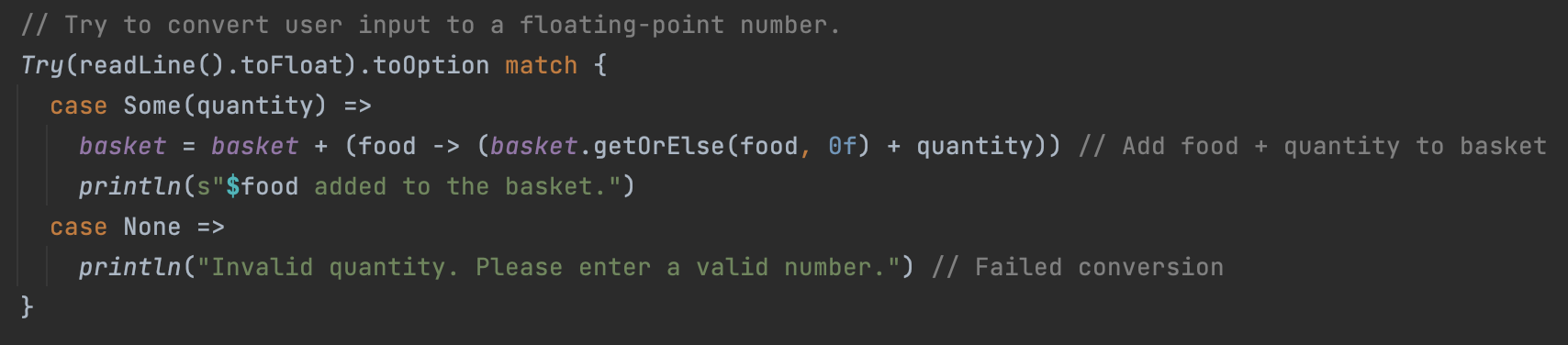
**Functional Thinking**

The developed Scala food prices application utilises several qualities derived from functional thinking and principles. One is focusing on results rather than the steps required, which specifically means taking a declarative programming approach rather than an imperative one. One example of this approach within the food price application is how the retrieval of data is performed using the “foldLeft” method:

****

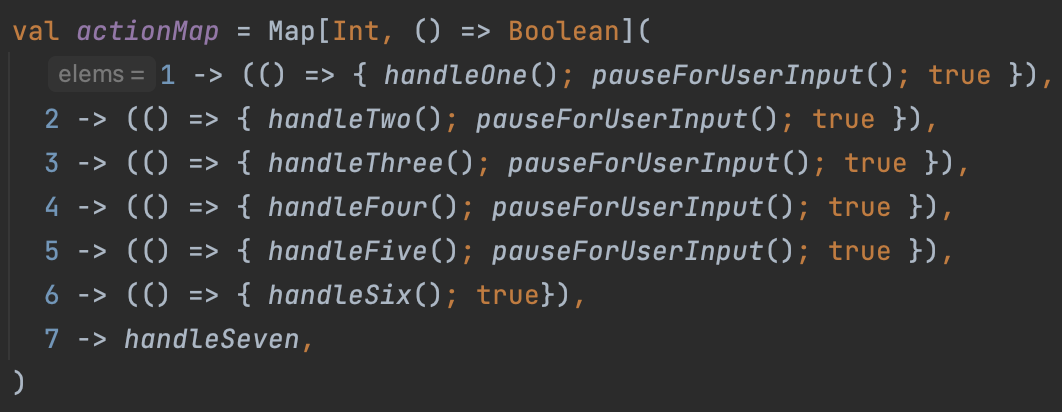
“foldLeft” is a recursive method that traverses elements within a list and helps with preventing stack overflow exceptions, while also avoiding mutable state. This method solely focuses on accumulating the result of retrieving the most recent price. In contrast, a declarative approach such as using a map would focus more on the process of retrieval.

One other principle of functional thinking is offloading mundane details to the programming languages. This can be voiced as harnessing the power of programming language by utilising its in-built functionalities, instead of coding your implementations which may be inefficient. An example of this within the food price application is using a “Try” construct for exception handling when a user attempts to enter a food item quantity.

****

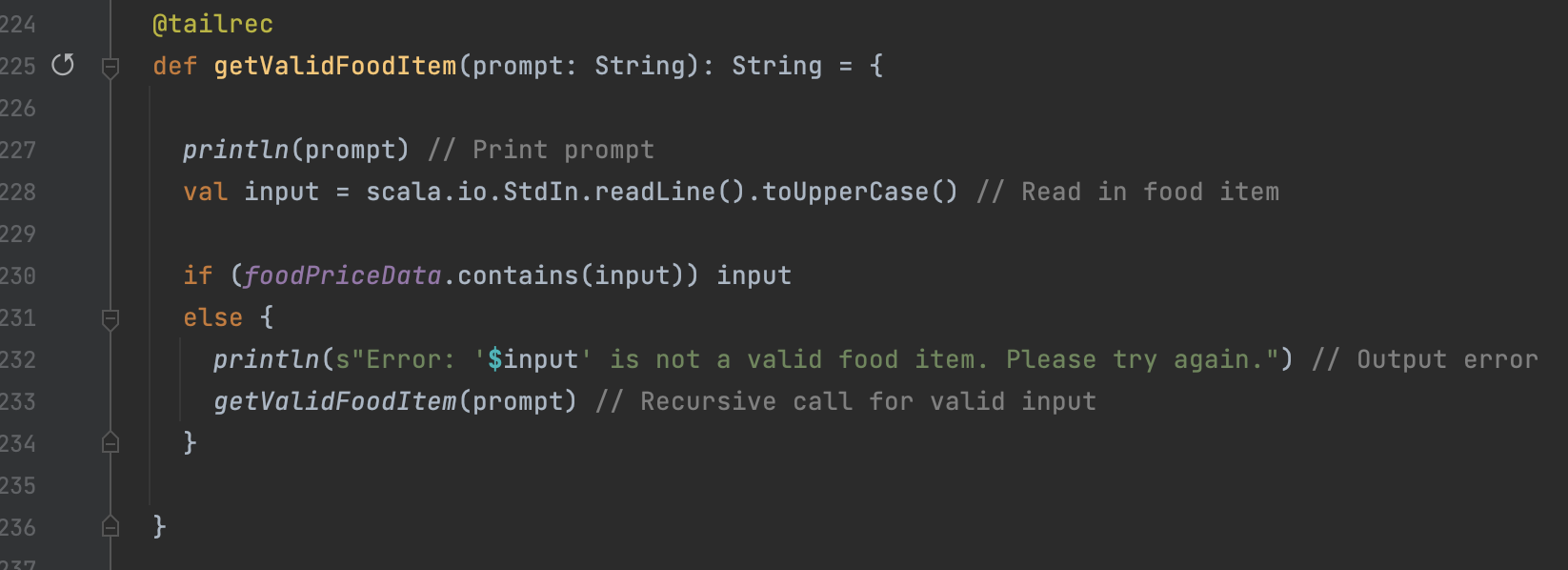
**Functional Programming Style**

Several functional programming techniques were imposed during the development of the food price application. One of which is first-class functions, which are functions that can be declared and invoked, and can be used in every segment of the language as just another data type. This can be observed within “actionMap”, the menu event handling function within the code:



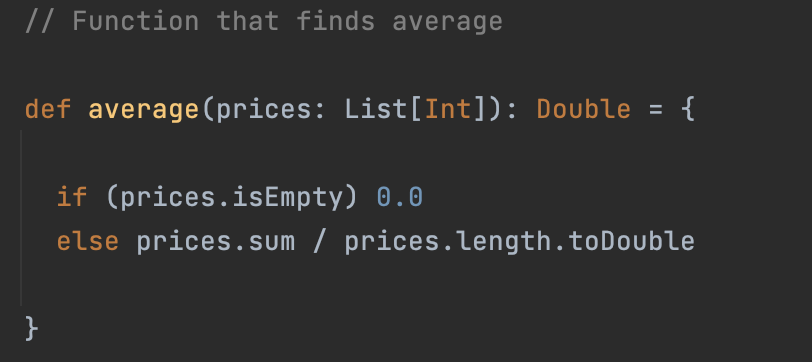
In this example, the map “actionMap” holds function values associated with integer keys. Each function is a first-class citizen that can be passed around and invoked based on the user's input. A “switch/case” statement may have sufficed in this scenario, but using first-class functions better aligns with the functional programming paradigm, and is more robust/scalable.

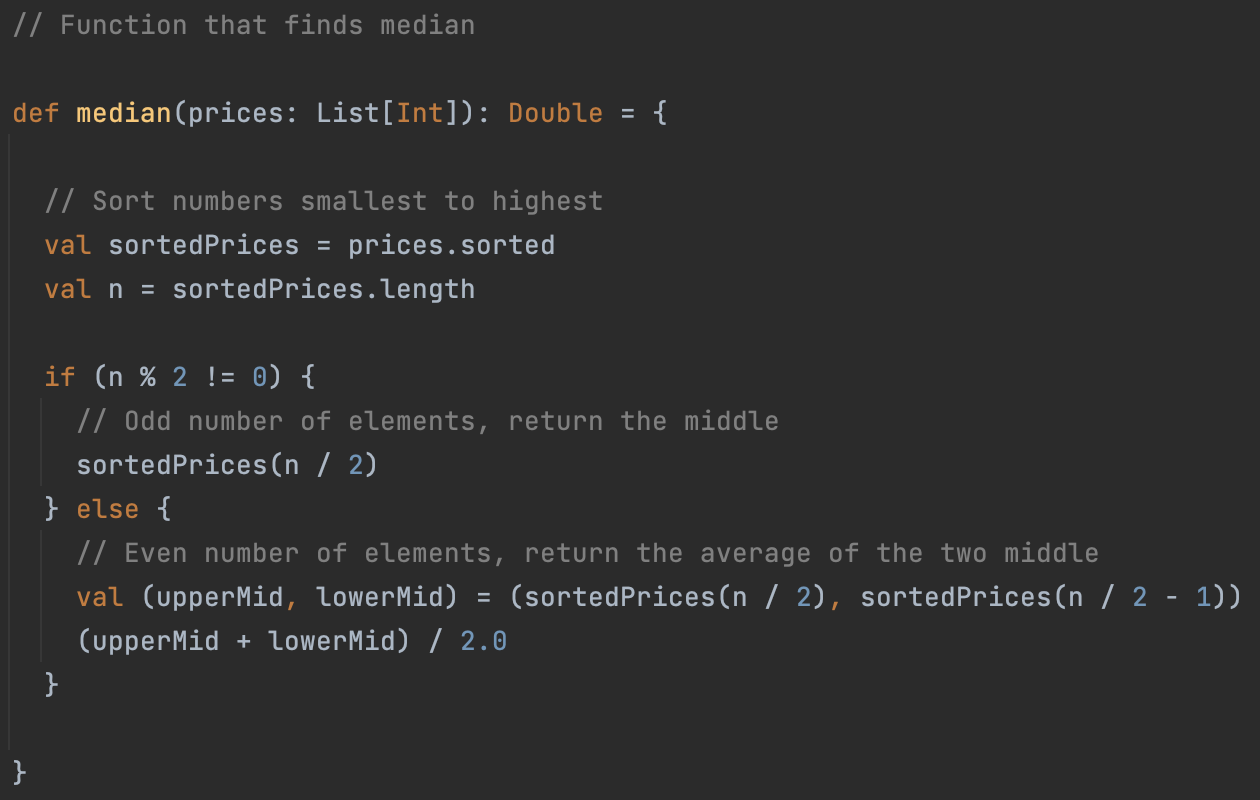
Traditional programming often facilitates different types of loops such as the “for” or “do-while” loops. However functional programming substitutes this for recursion, which can be defined as a function calling itself until a condition is met. This functional method of looping means there mutable state such as a counting variable, however, the downside to this method is that if the depth of recursion becomes too large - the application can stack overflow and crash. A way to avoid this is to use tail recursion. This is implemented when retrieving a valid food item from the user:



Tail recursion is achieved by placing the recursive self-call at the end of the function and, in Scala, annotating the code with the line “@tailrec” which lets the language know to optimise it. Under the hood, the compiler treats this function in a “loop-like” manner allowing it to reuse the same stack-frame for each call, so the issue of stack overflow is negated. Though an iterative use of while loops may have been sufficient for this scenario, tail recursion is a better-suited choice because Scala offers in-built optimizations that better align with functional paradigms.

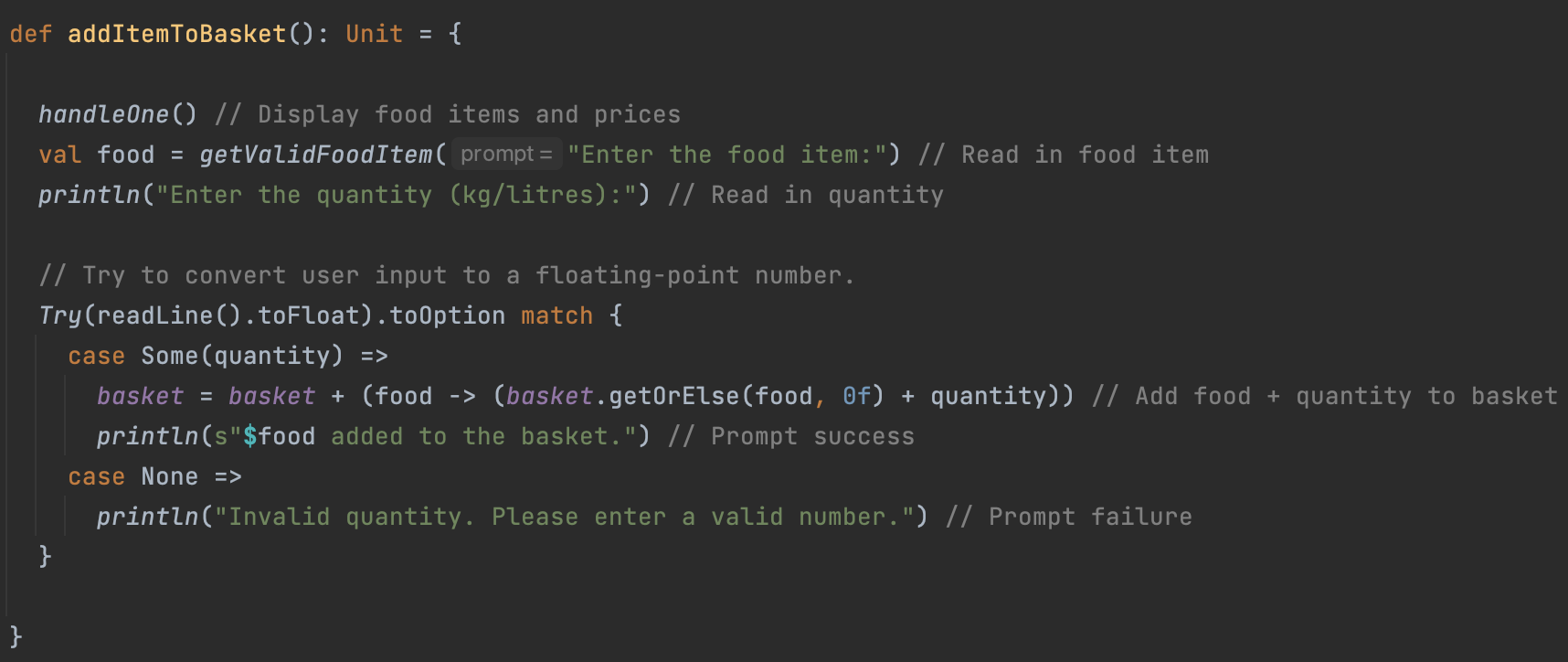
Several pure functions are utilised within the food price application. These are functions that output a value based on its parameters only, without any side effects. They are desirable within functional programming as they are easy to test, make the code concise, and have no dependencies. Two examples of pure functions in the food price application are found when finding the mean and median values, these functions are invoked multiple times across the application:





**Comparison of Functional and Imperative Style**

The use of functional thinking and functional programming styles to develop the food price application improved the ease of development and quality of the solution. This is due to Scala encouraging a declarative-styled approach that focuses on what the application should accomplish instead of how it will achieve it. This is highlighted in the use of the foldLeft method which simplified the process of data calculation while also reducing potential errors. Emphasis placed on immutability also made the application more reliable. Functions with no side effects made debugging errors more straightforward while the code itself was more predictable. This is shown within the application handling of the food basket feature which took an accumulative approach that was clear and controlled.

****

If the development of this application was in JavaScript, using an imperative object-orientated approach, the experience would have greatly differed. Though JavaScript offers some in-built functionality such as maps, filters, and arrow functions which promote a functional style of coding, it doesn't emphasise immutability. This factor presents more potential chances for runtime errors, which contrasts Scala's focus on reliability and robustness. JavaScript, being the more popular language, does have a larger surrounding community when compared to Scala. This offers higher community support and ready-made solutions for common programming issues, which may have eased and accelerated development.

A key personal takeaway from this assignment is that choice in programming paradigms and languages should align with the overall requirements and context of the solution. Due to the pure data-driven nature of the food price application, a functional approach to its programming was most beneficial and best-suited.