Programming Paradigms:

Coursework 1

Evaluation Report

2024-2025

Department of Computing

Submitted for the Degree of:
BSc Software Development for Business

Name: Vincenzo Russo

Programme : Software Development for Business

Matriculation Number: S2336012

Word Count: 1659

(***excluding contents pages, figures, tables, references and Appendices***)

"Except where explicitly stated, all work in this report, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award"

Signed by Student:____________________ **Date:** 11/12/2024

Table of Contents

Table of Figures

Table of Tables

1. Project Background

This project involved creating a Scala Singleton-object-based console application designed to perform statistical analysis on Formula One data. The console acts a User-Interface (UI) for the end-user, accepting a numerical input for analysis selection. The application reads data from a text file and displays six statistical summaries to the end-user. Data is then parsed into a Map, where the season year ([Int]) serves as the key, and its corresponding value is a List of Tuples containing the driver's name ([String]), points ([Float]), and wins ([Int]). IntelliJ IDEA was chosen as the development environment due to its comprehensive support for JVM languages like Scala, providing advanced tools for efficient code development and debugging (Hunt, 2014, p. 479; JetBrains, 2024).

2. Functional Thinking

Functional thinking focusses on declaring *what* a program should do, rather than *how* it should do it. This contrasts imperative thinking, where an application instead details a list of instructions it should follow. Central concepts that inspired functional ideals in the development process were immutability and pure functions.

2.1 Immutability

In Scala, variable declaration revolves around two dynamic and statically typed identifiers: val and var. The latter is a mutable identifier that allows for value reassignment after initialisation at runtime. In contrast, the former val cannot be changed once assigned without flagging a compilation error; it's an immutable-typed variable, which is good in this context.

Immutability promoted concurrency and thread-safety when programming the application; it allowed for safe sharing of data across different threads at runtime by reusing existing structures. The approach reduced computational overhead by encouraging the use of optimal algorithms and memory management practices, thereby minimising memory misuse.

```scala
// Read data from file via readFile utility function and store in map
private val mapData: Map[Int, List[(String, Float, Int)]] = readFile(dataFilePath) match {
  case Right(data) => data // Return the data if successful
  case Left(error) => // Handle the error
    println(s"Error reading data file: $error")
    sys.exit(1)
}
```

Figure 1: mapData

Figure 1 above exemplifies this, located in the global scope of the application. Here, mapData is defined using val and assigned to an immutable collection structure, Map, where the data is collected from an alternate function. The immediacy of the dataset declared immutable forced subsequent functional thinking throughout development, focussing on transformation rather than mutation. In contrast, an imperative approach would rely on mutable state, increasing the risk of race conditions and inefficiencies due to unnecessary memory allocations. Instead of altering mapData directly, pure functions were employed to operate on its contents, promoting safer, predictable code with a declarative style.

2.2 Pure Functions

Pure functions produce the same output for any given input, i.e., create no side-effects. In the application, pure backend functions were designed to express the intended results declaratively by describing *what* should be computed rather than specifying *how* to compute it. This approach allowed a clear separation of concerns by consistently returning new Maps based on selected analysis criteria.

For example, as seen in Appendix A, Figure 3, the function getAvgPoints, when given the same input data Map, declaratively returns the same output of all drivers' average points across all seasons. This ensures referential transparency; the program's expected behaviour remains unchanged, and execution stays consistent. The function's consistency preserves data integrity, especially in floating-point calculations. Slight variations in input point values could significantly alter the average point calculation – undesirable since accuracy is essential.

3. Functional Techniques

Efforts were made to achieve an application of high standard using available multiple techniques. Not all techniques are discussed; however, the following section will explore notable ones.

3.1 Higher-Order Functions

A higher-order function is a function that either takes one or more functions as arguments, or returns a function as a result. In the application, higher-order functions were used extensively to abstractly derive results, enabling clear and reusable logic without explicitly manipulating the dataset's values. The ability to chain them was also a large component in development, allowing to perform complex transformations in a clean, modular way.

```scala
// Backend function to get selected driver's total points
private def getSelectedPoints(data: Map[Int, List[(String, Float, Int)]], inputName: String): Map[String, Float] = {   vrusso300 *
  val fullName = inputName
  val sumPoints = data.collect { case (_, drivers) =>
    // Filter based on simple function to check if the name matches
    drivers.filter { case (name, _, _) => name.equalsIgnoreCase(fullName) }
      // Sum selected driver
      .map { case (_, points, _) => points }.sum
    // Sum all seasons
  }.sum

  // Return the selected driver's total points as a Map
  Map(fullName -> sumPoints)
}
```

Figure 2: getSelectedPoints

In Figure 2 above, the getSelectedPoints function demonstrates this use of higher-order functions. This backend function takes data as a parameter and applies three succinct higher-order transformations to filter for the driver that matches the end-user's selection. It then sums up all their points per season.

Each Function's Role:

- **.collect:** Initiates the chain of operations. It traverses the data map and applies a partial function to extract relevant driver lists from each season. It processes only the entries matching the specified pattern and discards others.
- **.filter:** Is applied to the list of drivers to filter out entries that match a specific criterion. The filtering logic uses a pattern match where a function compares the driver's name to the specified fullName ignoring case.
- **.map:** After filtering, .map is used to extract the points of each matching driver using pattern matching. This creates a list of points corresponding to the selected driver across multiple seasons.

This approach highlights the power of combining higher-order functions to perform complex data transformations concisely and declaratively, without relying on traditional loops. While a foreach loop could have been used for iteration, the focus was to explore as many different functional looping techniques as possible.

3.2 Tail Recursion

Tail recursion is a special form of recursion where a function calls itself as its last operation before returning a result. This allows the compiler to optimise the call, reusing the same stack-frame instead of creating new ones for each recursion. The application uses tail recursion in a few instances.

In Appendix A, Figure 3, the recursive call calculateAvg(tail, sumFloat(total, points), count + 1) is the last operation performed in each recursive step. Since there's nothing left to process after this last call, the compiler can reuse the same stack frame, making the recursion memory-efficient through tail-call optimisation. The compiler treats this function as tail-recursive through the @tailrec annotation.

As shown in Appendix B, Table 1, after each recursive call, the same stack frame is reused, and the points are updated until the base case is reached. In the base case, where the list is empty (Nil), the total points (2458) are divided by the count of drivers (22), resulting in the average of 111.73. No new stack frame is created during the recursion, unlike traditional recursion, where each call would add a new stack frame – leading to the risk of a stack overflow. This is why tail recursion was chosen over traditional recursion.

3.3 Option, Try, Either and Pattern Matching

Option, Try, Either and Pattern Matching are functional ways to handle the existence/absence of elements, error handling and finding patterns in code respectively. Instead of imperative ways, like traditional if/else statements and throwing stack exceptions, these methods were used throughout the application to handle unexpected behaviours.

This is especially true as seen in Appendix A, Figure 4. The handleInput function uses all three of these to divert the application to the appropriate function based on whether or not end-user input is valid. The function is also curried to allow for partial application. This means the function can be used with different inputs with different variable types, passed when called/needed, disabling the need to rewrite the logic for input handling each time (See Appendix A, Figure 5).

Each Technique's Role:

- **Option and Try**: The function uses Try to attempt to convert the input string to an integer. It's then converted to Option type. If the conversion fails (i.e., the input is a name, and can't be converted to an integer), it returns None, representing the absence of a valid value. This is then handled appropriately by the function.
- **Either:** The function returns an either value, where the Left side contains an error message in case of invalid input, and the Right side contains the valid input (either an integer or a string), depending on whether the input can be successfully processed.
- **Pattern Matching:** The function uses pattern matching to distinguish between the different possible outcomes of Try, Option, and Either. It matches on the Some or None result from Try, and on the Right or Left result from either, allowing the application to follow different execution paths based on the validity of the input.

This approach condenses complex logic into a single function, where otherwise separate functions would be needed to handle different types of input from the command-line menu. The function is reusable based on different contexts, directing the end-user to the function or error based on the call's location. Alternatively, for-comprehensions could've been used, though at time of programming was unfamiliar. They work by combining multiple operations in a sequential style, automatically handling failures and None values.

4. Functional vs. Imperative Approaches

4.1 Functional Approach

Initially, Scala was challenging due to its declarative nature. However, gaining a deeper understanding of concepts like functional loops and immutability in the readFile function led to a more efficient approach. Using foldLeft and immutable variables throughout the function made data processing concise and seamless, eliminating the need for explicit state management. Error handling with Try reduced clutter and enhanced debugging by capturing file-related issues early (See Appendix A, Figure 6). Tail recursion also replaced traditional looping structures like do-while, optimising performance by reusing stack frames. This technique not only streamlined the menu implementation but also proved reusable across other project components (See Appendix A, Figures 3 and 7).

4.2 Alternative Imperative Approach

Coming from an imperative programming background, C# would have been a more familiar choice for developing this application. In such an approach, a dedicated DataManager class would encapsulate file-reading logic using File.ReadAllLines, with conditional if statements to validate and parse each line of the dataset. A menu would be created using a do-while loop, allowing users to interact with the program. However, reliance on mutable variables and nested conditional logic could complicate debugging and reduce readability compared to a functional approach using immutable data structures.

4.3 Conclusion

In retrospect, while Scala required a mindset shift from imperative to functional programming, its capabilities for readability and maintainability proved undeniable. Given the opportunity to choose any language for the application requirements, Scala would now be the preferred choice. Ultimately, the journey from initial challenges to gaining a deeper understanding of its functional paradigm highlighted Scala's value as a powerful tool, encouraging the adoption of a functional approach in future projects.

Appendices

Appendix A – Large Code Screenshots

```scala
// Backend function to get average points
private def getAvgPoints(data: Map[Int, List[(String, Float, Int)]]): Map[Int, Float] = {  vrusso300

  // Calculate the average points for each season in a tail-recursive func
  @tailrec
  def calculateAvg(drivers: List[(String, Float, Int)], total: Float, count: Int): Float = drivers match {
    // Base case, if the list is empty, return the average
    case Nil => if (count == 0) 0 else total / count
    // Recursive case, sum the points via external sumFloat func and increment the count
    case (_, points, _) :: tail => calculateAvg(tail, sumFloat(total, points), count + 1)
  }

  // Map the seasons to the average points
  val result: Map[Int, Float] = data.map { case (season, drivers) =>
    // Calculate the average points for each season
    val avgPoints = calculateAvg(drivers, 0, 0)
    // Round the points up to 2 decimal places using big decimal, then convert to float
    val roundedAverage = BigDecimal(avgPoints).setScale(2, BigDecimal.RoundingMode.HALF_UP).toFloat
    season -> roundedAverage
  }

  // Return the average points for seasons
  result
}
```

Figure 3: getAvgPoints Function

```scala
// Curried Function to validate and process the input based on a list of expected values
private def handleInput(validOptions: List[Any])(input: String)(contextMessage: String): Either[String, Either[Int, String]] =  vrusso300 *
  Try(input.toInt).toOption match {
    case Some(option) if validOptions.contains(option) =>
      // If parse is successful and the option is in the list, return the option
      Right(Left(option))

    // If parse fails (i.e., is a name and cant be an integer), check if its in the passed list
    case None if validOptions.map(_.toString.toLowerCase).contains(input.toLowerCase) =>
      Right(Right(input))
    case _ =>
      // Return the error message if the input is invalid
      Left(s"Invalid input '$input'. $contextMessage")
  }
```

Figure 4: handleInput Function

```scala
// Curried functions to validate the user input
private def validateName: String => Either[String, Either[Int, String]] = handleInput(nameList)(_)(errName)   new *
private def validateSeason: String => Either[String, Either[Int, String]] = handleInput(seasonList)(_)(errSeason)   new *
private def validateMenuInput: String => Either[String, Either[Int, String]] = handleInput(expectedOptions)(_)(errMenu)   new *
```

Figure 5: Partially Applied Curried Functions

```scala
private def readFile(fileName: String): Either[String, Map[Int, List[(String, Float, Int)]]] = {  vrusso300
  // Safely manage file reading
  Using(Source.fromFile(fileName)) { bufferedSource =>
    val lines = bufferedSource.getLines().toList

    val data = lines.headOption match {
      case None => return Left("File is empty")
      case Some(_) =>
        // Use foldLeft to accumulate the result in an immutable map
        lines.foldLeft(Map[Int, List[(String, Float, Int)]]()) {
          case (mapData, line: String) =>
            val splitLine = line.split(",", 2).map(_.trim)
            val season = splitLine(0).toInt
            val allDrivers = splitLine(1).split(",").map(_.trim)

            // Process each driver entry and create a list of tuples
            val seasonData = allDrivers.map { driverEntry =>
              val driverDetails = driverEntry.split(":").map(_.trim)
              val name = driverDetails(0)
              val stats = driverDetails(1).split(" ").map(_.trim)
              val points = stats(0).toFloat
              val wins = stats(1).toInt
              (name, points, wins)
            }.toList

            // Update the map with the new season data, appending to the existing list
            mapData + (season -> (mapData.getOrElse(season, List()) ++ seasonData))
        }
    }
    Right(data)
  } match {
    case Success(data) => println("Data loaded successfully"); data
    case Failure(exception) => Left(exception.getMessage)
  }
}
```

Figure 6: readFile Function

```
private def menuLoop(): Unit = {  vrusso300
  // Tailrec loop
  @tailrec
  def loop(): Unit = {
    val input = displayMenuAndReadOption()
    validateMenuInput(input) match {
      case Right(Left(option)) =>
        if (processMenuOption(option)) loop()
      case Left(error) =>
        println(error)
        loop()
    }
  }
  // Start the recursive loop
  loop()
}
```

Figure 7: menuLoop Function

Appendix B – Tables

For example, consider the data for year 2023:

Call #	Current Drivers	Total Points	Count	Stack Frame Status
1	[(“Max Verstappen”, 575, 19), (“Sergio Perez”, 285, 2), …]	0.0	0	Stack frame created
2	[(“Sergio Perez”, 285, 2), (“Lewis Hamilton”, 234, 0), …]	575.0	1	Same frame reused (points updated)
…	…	…	…	…
23	[(“Nyck de Vries”, 0, 0), Nil]	2458.0	22	Base case (result then calculated)

Table 1: Tail Recursion Example

References

Hunt, J. (2014). *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Cham: Springer

JetBrains. (2024). *Advanced configuration*. Available at: https://www.jetbrains.com/help/idea/tuning-the-ide.html (Accessed: 7 December 2024).