

## Functional Programming – Mini Project Report

Project: Student Marks Analyzer Group Members: [Member 1], [Member 2], [Member 3], [Member 4]

---

### 1. Problem Statement and Industrial Motivation

In many educational institutions, analyzing student performance is essential for academic planning, identifying high-achieving students, and providing remedial support to underperforming students. Traditionally, marks processing is done manually or with spreadsheets, which is error-prone and not easily scalable.

This project implements a Student Marks Analyzer in Haskell to process student marks from a CSV file and produce:

- Ranked results with grades
- Class statistics (highest, lowest, mean scores)
- ASCII-based visual representation of scores

Industrial relevance:

- Demonstrates reliable, auditable data pipelines.
- Shows pure functions, immutability, and predictable computation.
- Supports parallel computation for scalability.

---

### 1. Functional Design

#### 2.1 Data Types (DataTypes.hs)

```
data Student = Student { name :: String, marks :: [Int] }
data Result = Result { studentName :: String, total :: Int, average :: Float,
grade :: String, finalScore :: Float }
data Statistics = Statistics { highest :: Float, lowest :: Float, mean ::
Float }
```

#### 2.2 IO Handling (IOHandler.hs)

```
parseStudent :: String -> Student
parseStudent line = let parts = splitByComma line
                    nm = head parts
                    mk = map read (tail parts)
                    in Student nm mk

readStudents :: FilePath -> IO [Student]
readStudents path = fmap (map parseStudent . lines) (readFile path)
```

#### 2.3 Processing Pipeline (Processing.hs)

```

computeResult :: Student -> Result
computeResult s = let ms = marks s
                    avg = averageOf ms
                    final = case ms of (m1:m2:m3:_) -> fromIntegral m1*0.6 +
fromIntegral m2*0.2 + fromIntegral m3*0.2
                                _ -> avg
                    total = sum ms
                    in Result (name s) total avg (gradeFromAverage final) final

rankResults :: [Result] -> [Result]
rankResults = reverse . sortOn finalScore

```

## 2.4 Utilities (Utils.hs)

```

averageOf :: [Int] -> Float
averageOf xs = fromIntegral (sum xs) / fromIntegral (length xs)

gradeFromAverage :: Float -> String
gradeFromAverage x
  | x >= 85  = "A"
  | x >= 70  = "B"
  | x >= 50  = "C"
  | x >= 35  = "D"
  | otherwise = "F"

```

## 2.5 Main Module (Main.hs)

```

main :: IO ()
main = do
  putStrLn "Enter CSV file path (e.g., students.csv):"
  path <- getLine
  students <- readStudents path

  let results = parMap rseq computeResult students
  let ranked = rankResults results
  let stats = computeStatistics ranked

  putStrLn "\n--- Class Statistics ---"
  print stats

  putStrLn "\n--- Ranked Results Table ---"
  putStrLn (asciiTable ranked)

```

---

## 1. Functional Programming Concepts Applied

Concept	Usage in Project
Pure Functions	computeResult, averageOf, gradeFromAverage
Higher-Order Functions	map, parMap, sortOn
Algebraic Data Types	Student, Result, Statistics
Immutability	All transformations produce new values; no mutable state
Recursion	splitByComma and list processing
Parallelism	Control.Parallel.Strategies for parMap

---

## 1. Sample Input and Output

students.csv:

```
Alice,85,90,76
Bob,60,65,70
Charlie,50,55,45
Diana,95,95,95
```

GHCi Run:

```
Enter CSV file path (e.g., students.csv):
students.csv

--- Class Statistics ---
Statistics {highest = 95.0, lowest = 48.3, mean = 72.6}

--- Ranked Results Table ---
Student  | Score | Grade
-----
Diana    | 95    | A
Alice    | 85    | A
Bob      | 65    | B
Charlie  | 50    | C
```

---

## 1. Discussion

2. Correctness & Reliability: Pure functions make each step predictable and testable.
3. Modularity: Each module has a single responsibility.
4. Parallelism: parMap allows concurrent processing of student results.
5. Maintainability: New rules, statistics, or scoring can be added in pure functions.
6. FP Benefits: Eliminates side effects except controlled IO, encourages reasoning about behavior, scales naturally.

Possible Extensions: - Weighted marks for additional assignments - Graphical visualization instead of ASCII - Export JSON or CSV output

---

1. References / Tools Used
2. Haskell (GHC 9.8.2 / GHCi)
3. Control.Parallel.Strategies
4. CSV file input as real-world dataset
5. Functional design patterns from FP literature