EC8206: FUNCTIONAL PROGRAMMING

GROUP PROJECT REPORT

GROUP MEMBERS:

EG/2020/4162 – RATHNAYAKE R.M.B.T.M.

EG/2020/4035 – KUMARI D.P.S.T.

SEMESTER: 08

DATE : 12 /19 /2025

1 Problem Statement and Industrial Motivation

1.1 Problem Statement

Educational institutions handle large volumes of student academic records, including marks for multiple subjects, class performance summaries, and ranking calculations. These operations require accurate, consistent, and auditable processing. Traditional imperative implementations often rely on mutable state, making the system prone to data inconsistency, race conditions in concurrent environments, and difficulty in tracing transformations applied to data.

The problem addressed in this project is to design and implement a Student Marks Analyzer that processes student records and generates,

- Individual student performance analysis with subject-wise breakdowns
- Class-level statistical summaries (averages, grade distributions)
- Subject-wise analytics across all students
- Ranking systems and identification of at-risk students
- Search and filtering capabilities by student ID or grade.

1.2 Industrial Motivation

In real-world educational platforms, grading and analytics systems must be highly reliable, thread-safe, and easily verifiable. Errors in academic records can result in incorrect student evaluations, unfair rankings, and institutional credibility issues.

Functional Programming (FP) offers strong guarantees that are valuable in such industrial systems:

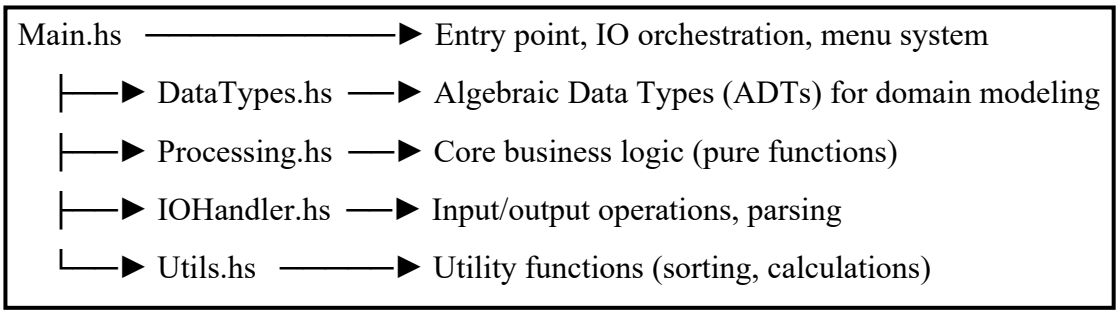
- Immutability (ensures data integrity) prevents accidental data corruption
- Pure functions (predictable, auditable calculations) enable deterministic and auditable computations
- Strong type safety reduces runtime failures
- Concurrency-readiness allows safe parallel data processing

This project models a simplified but realistic academic analytics system aligned with industry requirements for correctness and scalability.

2 Functional Design and Architecture

2.1 System Architecture

The application follows a modular, layered architecture.



2.2 Core Data Types

The system uses Algebraic Data Types (ADT) to represent real-world entities.

```
data Subject = Subject { subjectName :: String, subjectMark :: Double }
data Student = Student { studentId :: String, studentName :: String, subjects :: [Subject]}
data Grade = A | B | C | S | F deriving (Show, Eq, Ord)
data StudentReport = StudentReport { reportStudent :: Student, avgMark :: Double,
    grade :: Grade, rank :: Int, subjectWisePerformance :: [(String, Double, Grade)] }
data Result a = Success a | Error String  -- Functional error handling
```

2.3 Key Functions and Type Signatures

2.3.1 Pure Calculation Functions

```
calculateAverage :: [Double] -> Double
determineGrade :: Double -> Grade
createStudentReport :: Student -> Result StudentReport
processStudents :: [Student] -> Result [StudentReport]
calculateClassStatistics :: [StudentReport] -> Result Statistics
calculateSubjectStatistics :: [Student] -> [SubjectStatistics]
```

2.3.2 Higher-order Functions

```
mergeSort :: (a -> a -> Bool) -> [a] -> [a] -- Generic sorting with custom comparator
filterByGrade :: Grade -> [StudentReport] -> [StudentReport] -- Filter by grade
```

Each function has a clear type signature, which explains what the function expects and what it returns.

3 Functional Programming Concepts Applied

3.1 Pure Functions

Pure functions always give the same output for the same input. As an example, `calculateAverage` only calculates the average using given marks.

3.2 Immutability

In this system, data is never changed after it is created**.** Instead of modifying existing data, new data is created. `addBonusMarks` creates new students rather than modifying existing ones, eliminating state-related bugs.

3.3 Recursion

`findMax` uses recursion instead of loops for declarative iteration.

3.4 Higher-Order Functions

Higher-order functions are functions that take other functions as parameters. `mergeSort` accepts comparison functions, enabling `sortByAverage` and `sortByName` reuse.

3.5 Algebraic Data Types

ADT usage ensures only valid data is represented, all cases are handled by the compiler. For example, the `Grade` type ensures only valid grades are used.

3.6 Pattern Matching

Pattern matching is used to handle different cases clearly. `processStudents` uses `case` for clean error handling without nested conditionals.

3.7 Function Composition

`getTopStudents n = take n . sortByAverage` builds pipelines declaratively.

(Sort students → select top students)

4 Implementation

4.1 ETL Pipeline

(1) Extract CSV → Read student data from CSV or user input.

(2) Transform Students → Calculate averages, grades, and statistics using pure functions.

(3) Load outputs → Display formatted results.

Pure/impure separation: business logic in Processing.hs/Utils.hs, IO in Main.hs/IOHandler.hs. This enables easy testing, clear side-effect boundaries, and parallel execution potential.

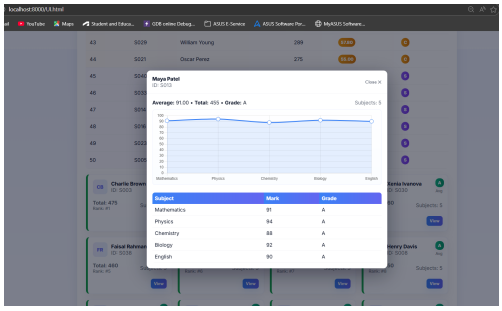
4.2 Error Handling

Errors are handled using the `Result` type. Errors are clearly represented, No silent failures and handled both success and error cases

5 Expected outputs and Possible extensions

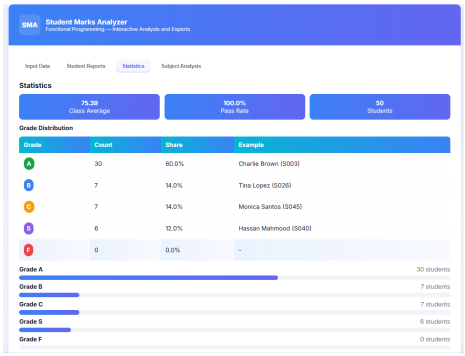
5.1 Expected outputs

5.1.1 Individual Student Report



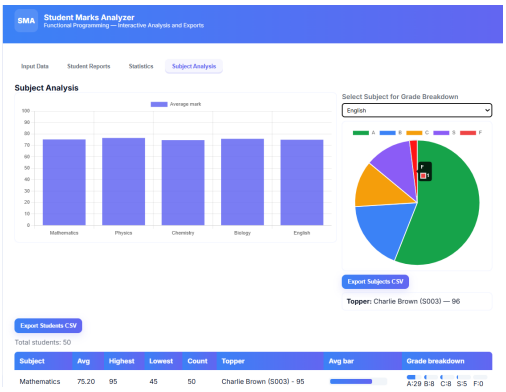
```
Student ID: S001
Name: Kamal Perera
Average Mark: 78.5
Grade: A
Rank: 3
Subjects:
  Mathematics - 85 (A)
  Physics     - 72 (B)
  Chemistry   - 79 (A)
```

5.1.2 Class Statistics



```
Class Average: 65.3
Highest Average: 88.2
Lowest Average: 42.1
Grade Distribution:
A: 5
B: 8
C: 10
S: 4
F: 2
```

5.1.3 Subject-wise Analysis




```
Mathematics Average: 70.2 Physics Average: 66.8 Chemistry Average: 69.1
```

5.2 Possible extensions

- Performance: Parallel processing via Control.Parallel, lazy streaming for millions of records.
- Features: Multi-classes tracking, PDF/Excel export for each student report, ML grade prediction.
- Database: Persistent storage, connection pooling, transactions.

6 Why Functional Programming improves this System

- Reliability: Pure functions guarantee consistent calculations, immutable data prevents corruption, type safety catches errors pre-runtime, explicit Result handling makes failures visible.
- Concurrency Potential: Pure design enables parallel map/reduce for large datasets, concurrent subject statistics, and thread-safe operations via immutability.
- Maintainability: Single-responsibility functions, declarative intent, composable features, referential transparency enables straightforward reasoning.

7 References

[1] B. Bawantha, *"4162-4035-Student-Marks-Analyzer-FP,"* GitHub, Aug. 2025. [Online]. Available: https://github.com/bawantha395/4162-4035-Student-Marks-Analyzer-FP

[2] Haskell Community, *"Haskell Documentation,"* Haskell.org, 2025. [Online]. Available: https://www.haskell.org/documentation/