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

Contents

1 Problem Statement and Industrial Motivation

Educational institutions process thousands of student records annually, requiring automated systems for grade calculation and performance tracking. Traditional imperative approaches suffer from data integrity issues (mutable state), concurrency challenges (thread-unsafe operations), and audit trail complexity (side effects obscuring transformations).

This project implements a Student Marks Analyzer that processes student academic records and generates comprehensive reports including

- 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.

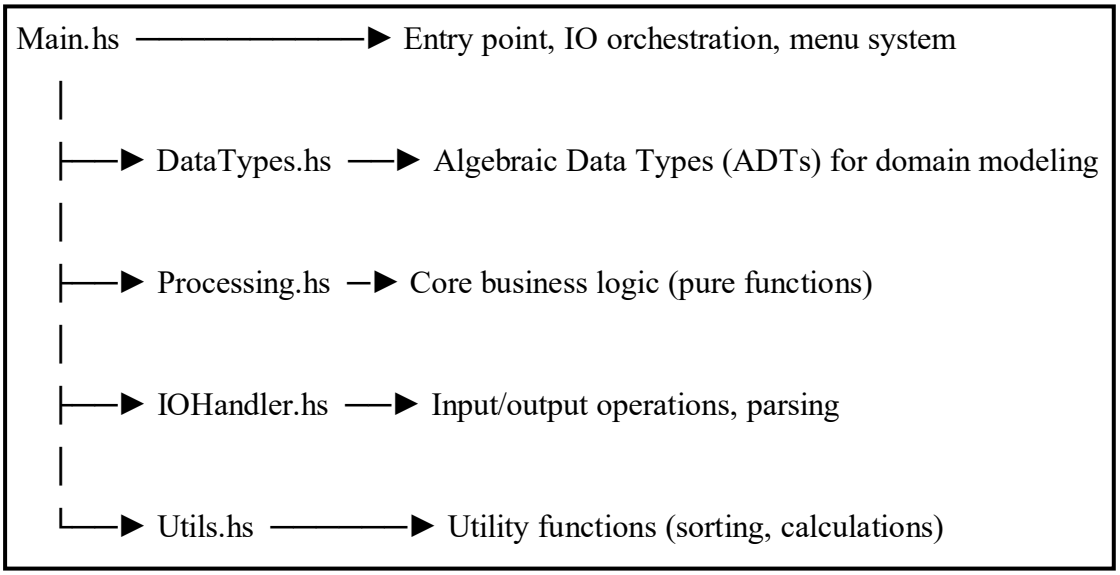
Why Functional Programming?

FP addresses these challenges through: immutability (ensures data integrity), pure functions (predictable, auditable calculations), type safety (compile-time error detection), composability (reusable components), and concurrency-readiness (thread-safe parallelization). The application consists of three core services:

2 Functional Design and Architecture

2.1 System Architecture

The application follows a modular, layered architecture.



2.2 Core Data Types

Algebraic Data Types model the domain precisely.

```
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

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

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

3 Functional Programming Concepts Applied

3.1 Pure Functions

`calculateAverage` has no side effects—same input always produces same output, enabling reliable testing.

3.2 Immutability

`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

`mergeSort` accepts comparison functions, enabling `sortByAverage` and `sortByName` reuse.

3.5 ADTs

`Grade` ensures compiler-checked exhaustive pattern matching; impossible states are unrepresentable.

3.6 Pattern Matching

`processStudents` uses `case` for clean error handling without nested conditionals.

3.7 Function Composition

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

4 Implementation Highlights

4.1 ETL Pipeline

(1) Extract CSV → Students
(2) Transform Students → Reports/Statistics
(3) Load formatted outputs

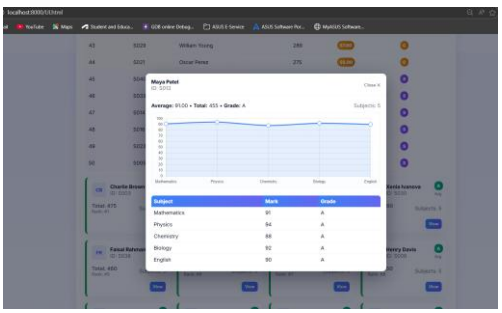
Pure/impure separation: business logic in Processing.hs/Utils.hs, IO in Main.hs/IOHandler.hs. This enables easy testing, clear side-effect boundaries, and parallel execution potential.

4.2 Error Handling

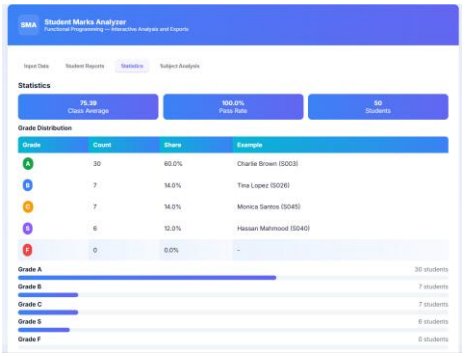
Result type makes errors explicit in signatures—callers must handle both Success and Error cases, preventing silent failures.

5 Sample Outputs

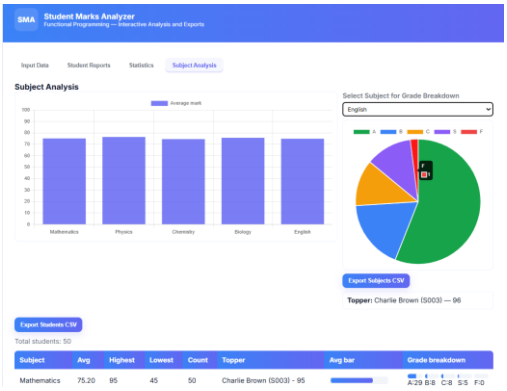
5.1 Individual Student Report



5.2 Class Statistics



5.3 Subject-wise Analysis



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.
- Maintainability: Single-responsibility functions; declarative intent; composable features; referential transparency enables straightforward reasoning.
- Testability: Pure functions require no mocking; deterministic behavior ensures reproducible tests; type signatures document contracts.
- Concurrency Potential: Pure design enables parallel map/reduce for large datasets, concurrent subject statistics, and thread-safe operations via immutability.

7 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.

8 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/