Project Based Evaluation

Project Report

Semester-IV (Batch-2023)

DSA Calculator

Supervised By:

Dr. Heena Wadhwa

Submitted By:

Maridul Walia 2310991897 G21

Mehak Walia 2310991901 G21

Michael Dhiman 2310991902 G21

Ramya Padala 2310992587 G21

Department of Computer Science and Engineering
Chitkara University Institute of Engineering & Technology,
Chitkara University, Punjab

Abstract

This project introduces a feature-rich, modular Java-based toolkit designed for the manipulation, transformation, evaluation, and visualization of mathematical and symbolic expressions. Serving both educational and practical debugging purposes, the toolkit allows users to convert expressions between infix, prefix, and postfix notations using stack-based evaluators. It provides separate engines for numeric, symbolic, and bitwise evaluations, supporting both literals and variable inputs while delivering step-by-step computational breakdowns. Through persistent logging mechanisms, each session captures detailed computation logs and user inputs in time stamped text files, aiding reproducibility and error tracking.

The system is organized into clearly defined modules—Converter, ExpressionEvaluator, StepByStepEvaluatorNumeric, StepByStepEvaluatorString, BitwiseEvaluator, and ExpressionTreeVisualizer—each handling specific tasks such as conversion, arithmetic or symbolic evaluation, bitwise operations, or ASCII-based tree rendering. These components communicate via a command-line interface managed by Main.java, which guides the user through various options while handling exceptions and input validations gracefully. The ASCII tree visualization module enhances the user's understanding by representing expression structure in a human-readable format, complementing the logged evaluation steps.

This extensible and maintainable toolkit supports the addition of new operators, data types, and interface layers. It ensures usability across different user categories—students, educators, and developers—while maintaining clarity, transparency, and performance. Its combination of theoretical correctness and practical traceability makes it a valuable resource for expression handling in both academic and professional contexts.

Table of Contents

1. Introduction

1.1 Background

Expressions form the core of computational logic and symbolic reasoning in nearly all areas of computer science and software engineering, whether performing arithmetic calculations, evaluating logical conditions, or building complex algorithms, developers routinely interact with expressions in one form or another. These expressions are not limited to basic numerical computations—they extend to logical expressions, symbolic algebra, and even the low-level processing of data using bitwise operators. In environments ranging from simple calculators to sophisticated compilers and interpreters, correct parsing, conversion, and evaluation of expressions are foundational tasks.

Expressions can be represented in multiple notations, including infix, prefix, and postfix. Infix notation, the most familiar to humans (e.g., a + b), is used extensively in everyday arithmetic. Prefix notation (e.g., + a b) places operators before their operands and is particularly suited for certain types of parsing and evaluation strategies. Postfix notation (e.g., a b +), commonly used in stack-based systems and calculators, simplifies the parsing process by removing the need for parentheses and respecting operator precedence without requiring lookahead or backtracking.

While these notations are mathematically equivalent in expressive power, each has unique structural and operational advantages. However, converting between them and understanding the transformation process poses a significant learning curve for students and junior developers. Additionally, manually debugging expressions or verifying the intermediate steps of conversion and evaluation can be cumbersome and error-prone, especially in complex nested cases involving variables, multi-character tokens, or intricate operator precedence rules.

To address these challenges, we introduce a robust, extensible Java-based expression processing toolkit. This software suite automates the parsing, conversion, visualization, and evaluation of expressions across a range of domains. It is designed not only as an educational tool for students learning about data structures and expression parsing but also as a practical utility for professionals seeking to debug or validate expressions in various applications such as interpreters, compilers, or custom scripting languages.

1.2 Objectives

The primary objective of this project is to create a reliable, extensible, and educational tool that facilitates both the understanding and practical application of expression parsing and evaluation. The toolkit seeks to serve both novice and experienced developers by providing clear, traceable, and reusable tools for expression handling.

The key objectives can be summarized as follows:

- **Enable Detailed Traceability:** The system captures and logs each step in the parsing, conversion, and evaluation pipeline. These logs include intermediate forms of expressions, parsing decisions, and stack operations. This level of transparency is particularly useful in educational settings where students need to see how an expression is transformed internally.
- **Support Diverse Expression Types**: Unlike many educational tools that focus solely on numeric arithmetic, this toolkit is designed to handle symbolic and bitwise expressions as well. It can operate with variables (e.g., x, y) and simulate evaluation in symbolic contexts, broadening its applicability to algebraic reasoning, logic design, and computer architecture.
- **Improve Conceptual Clarity through Visualization**: The expression tree visualization feature converts linear expressions into binary trees that visually demonstrate the hierarchical structure of operations. For instance, in the expression a + b * c, the visualization will clearly show that b * c is evaluated before a + ... This graphical representation helps users understand how precedence and associativity rules influence evaluation.
- **Enhance Usability and Auditability**: The command-line interface is user-friendly, with intuitive prompts, error messages, and interactive menus. Every input and output is timestamped and recorded in dedicated text files (input.txt, output.txt), making it easy to revisit previous sessions and trace errors or analyze behavior retrospectively.
- **Ensure Scalability and Modularity:** Built on object-oriented principles, the toolkit is organized into clearly defined classes and interfaces, enabling developers to extend functionality with minimal impact on the system. For example, new operator types can be introduced simply by implementing a corresponding handler class, and evaluation engines can be swapped or enhanced without refactoring core components.
- **Deliver a Portable CLI Utility:** The toolkit is implemented entirely in Java and does not rely on external libraries, ensuring it can run seamlessly on any platform that

supports the Java Runtime Environment (JRE). This makes it ideal for educational deployments, scripting, or integration into broader Java-based systems.

1.3 Significance

The development of this Java-based expression toolkit holds considerable significance across both educational and professional domains. It addresses multiple gaps in current tools and learning methodologies, while promoting deeper conceptual understanding, automation, and operational reliability in expression handling tasks.

Enhancing Educational Outcomes

For students learning programming languages, compilers, or discrete mathematics, mastering expression evaluation and transformation is a fundamental yet often difficult topic. Manual computation of infix-to-postfix conversions or tree-based evaluations can be error-prone and mentally taxing, particularly in the early learning stages. By automating these processes and providing clear, step-by-step logs, the toolkit bridges the gap between abstract concepts and practical execution. The ASCII-art based tree visualizations further reinforce learning by enabling students to see how operators and operands are structured and executed internally.

Additionally, the logging features provide instructors with a powerful way to track student interactions and learning progress. Logs can be reviewed to identify common errors, misunderstood concepts, or incomplete conversions, making this toolkit not only a student learning aid but also a teaching support tool.

Supporting Professional Development and Debugging

In real-world software development, especially in systems dealing with interpreters, compilers, or scripting engines, expression parsing and evaluation are core features. Manually validating correctness in these scenarios can be challenging, particularly when dealing with dynamic expressions or user inputs. The toolkit simplifies these tasks by providing:

- Automated conversion between notation formats,
- Reliable evaluation routines for both numerical and symbolic data, and
- Transparent logging that captures all steps taken by the system.

Such capabilities significantly reduce the time and effort required to debug complex expressions or integrate custom evaluators into larger applications. This can be especially

valuable in fields like embedded systems, computer algebra systems (CAS), and automated testing frameworks.

Promoting Standardization and Reproducibility

By ensuring that all user inputs, operations, and results are logged in a consistent format with timestamps, the toolkit supports reproducibility and accountability—an essential requirement in both academic and professional settings. Whether conducting experiments, grading student assignments, or debugging system behavior in production, having a full record of operations helps validate outcomes and enforce traceability.

Moreover, the consistent notation handling and conversion logic promote better adherence to standards in expression formatting, reducing ambiguity and minimizing interpretation errors when expressions are shared across systems or teams.

Enabling Future Expansion and Integration

The object-oriented design and modular structure of the toolkit make it a strong foundation for future enhancements. Its architecture supports easy integration with graphical user interfaces, web-based platforms, or advanced computational backends. Developers can extend the toolkit to support:

- Additional data types (e.g., floating-point, boolean),
- Custom functions and operator overloading,
- Syntax highlighting and graphical rendering, or
- Export to formats like XML, JSON, or LaTeX for broader interoperability.

This extensibility ensures that the toolkit is not a static utility, but a growing platform capable of evolving with user needs and technological advancements.

2. Problem Definition and Requirements

2.1 Problem Statement

Despite the foundational importance of expression parsing in computer science, learners and developers often struggle with internalizing its mechanisms. Common pain points include confusion over operator precedence, associativity rules, parenthesis handling, and stack behavior during parsing and evaluation. These issues are exacerbated when expressions involve a mix of operators or nested sub-expressions. Traditional learning methods rely heavily on static examples or manual dry runs, which are prone to oversight and misinterpretation. Furthermore, debugging expression-handling logic in code becomes increasingly complex without visualization or stepwise feedback mechanisms. Therefore, there is a need for an interactive, reliable, and transparent toolkit that supports expression processing at all stages: conversion, evaluation, and structural visualization.

2.2 Software Requirements

- **Operating System:**
  - Minimum: Windows 7, Linux (Ubuntu 16.04+), macOS 10.12
  - Recommended: Windows 10 or newer, Ubuntu 20.04+, macOS 11 or newer
- **Java Development Kit (JDK):**
  - Minimum: JDK 8
  - Recommended: JDK 17 or later
- **Integrated Development Environment (IDE):**
  - Optional: Any text editor or terminal
  - Recommended: IntelliJ IDEA, Eclipse, or NetBeans for better development experience
- **Build Tools**:
  - Optional: Manual compilation using javac
  - Recommended: Maven or Gradle for project management and scalability
- **Java Runtime Environment (JRE):**
  - Minimum: JRE 8
  - Recommended: JRE 17 or newer
- **File and Logging Support:**
  - Uses built-in Java I/O libraries
  - Compatible with Java NIO for enhanced file system performance

- **Visualization Output:**
    - Supports basic command-line interfaces
    - Recommended: CLI with extended ASCII or Unicode support for optimal tree rendering

2.3 Hardware Requirements

- **Processor (CPU):**
    - Minimum: Dual-core 1.6 GHz
    - Recommended: Quad-core 2. GHz or higher
- **Memory (RAM):**
    - Minimum: 2 GB
    - Recommended: 4 GB or more
- **Storage:**
    - Minimum: 100 MB free space for application and basic logs
    - Recommended: 500 MB+ for extended logs, session backups, or additional modules
- **Display:**
    - Minimum: 1024×768 resolution
    - Recommended: 1366×768 or higher for better visibility of tree structures and logs
- **Keyboard:**
    - Standard QWERTY keyboard recommended for CLI input
- **Network (Optional):**
    - Not required for core functionality
    - May be used for future online integration, remote logging, or updates

2.4 Use Cases

- **UC1:** Convert the postfix expression 5 1 2 + 4 * + 3 - into infix format, log each conversion step, and compute the final result using stack tracing.
- **UC2:** Parse and render the expression tree for a+b*(c^d-e)^(f+g*h)-i in an ASCII-art format to help visualize the evaluation order and sub-tree structures.

- **UC3:** Evaluate a complex bitwise expression such as ~a&(b<<2) using symbolic variables, prompting users for inputs and tracing each step of computation.
- **UC4:** Log all user sessions including inputs, transformation steps, errors, and outputs to input.txt and output.txt, with ISO-format timestamps to support replay, comparison, or debugging.

3. Proposed Design

3.1 Architecture Overview

The application is designed using a modular, class-oriented architecture to facilitate maintainability, scalability, and testability. Each module performs a specific responsibility and interacts with others via well-defined interfaces, enabling the easy extension of features like new operators, notations, or evaluation types.

- **Main.java**: Acts as the primary controller. Presents a menu-driven interface that routes the user to various functionalities such as infix-to-postfix conversion, expression evaluation, bitwise operation handling, and expression visualization. It also handles basic input validation and user prompts.
- **Converter.java / StepByStepEvaluatorString.java**: These classes handle the conversion of expressions from one notation to another (e.g., infix to postfix or prefix). Step-by-step evaluators provide verbose logging of the conversion process to help users understand how each token is processed, how stacks are modified, and how operator precedence affects output.
- **ExpressionEvaluator.java / StepByStepEvaluatorNumeric.java**: These modules perform the actual evaluation of arithmetic or algebraic expressions. They support detailed evaluation logs including intermediate stack states, operation sequences, and result tracking. Users are prompted for variable values if symbols are detected.
- **BitwiseEvaluator.java**: Specializes in parsing and evaluating expressions containing bitwise operators like &, |, ^, <<, >>, and ~. It uses Java's native bitwise operations while internally converting expressions into a form that can be parsed using recursive descent or precedence-based parsing.
- **ExpressionTreeVisualizer.java**: Converts valid expressions into binary trees. Traverses the tree to calculate layout coordinates and generates a clean ASCII-art representation that reflects operator precedence and expression structure.

3.2 Module Descriptions

Notation Converters

- Employ stack-based parsing algorithms.
- Respect operator precedence and associativity.
- Detect and handle syntactical errors such as mismatched parentheses or invalid tokens.
- Optionally log steps like push/pop operations for educational output.

Evaluation Engines

- Support arithmetic, algebraic, and potentially symbolic evaluation.
- Dual-stack strategy (operator + operand) ensures clean separation of concerns.
- Includes support for parentheses, multi-digit numbers, and variables.
- Logs each operation applied, including intermediate results and stack content.

Bitwise Module

- Fully supports Java-compatible bitwise operations.
- Recognizes and evaluates unary ~ and binary &, |, ^, <<, >>.
- Handles input validation and provides syntax-specific error messages (e.g., shift without operand).
- Demonstrates evaluation with examples to clarify bitwise behaviors.

Visualizer

- Uses recursive parsing to build expression trees.
- Computes relative positions using tree depth and node spread.
- Outputs a static ASCII visualization that mirrors the logical structure of the expression.
- May be extended to support color-coded or type-labeled nodes in a GUI.

3.3 Data Flow

The end-to-end flow of the application is designed to be robust and resilient to malformed user inputs. Below is the generic pipeline:

1. **User Input**: Collected via console prompts or batch file input.
2. **Tokenization**: Input is split into tokens considering whitespace, multi-character operands, and operators.
3. **Routing to Module**: Based on user choice, input is passed to the relevant converter, evaluator, or visualizer.
4. **Processing**:
    - Converters reorder expressions using precedence rules.
    - Evaluators compute the result step-by-step.
    - Visualizers convert expression to trees.
5. **Logging & Output**:
    - Console output provides immediate feedback.
    - Optional file logging of operations, errors, and final results.

3.4 Functional Requirements

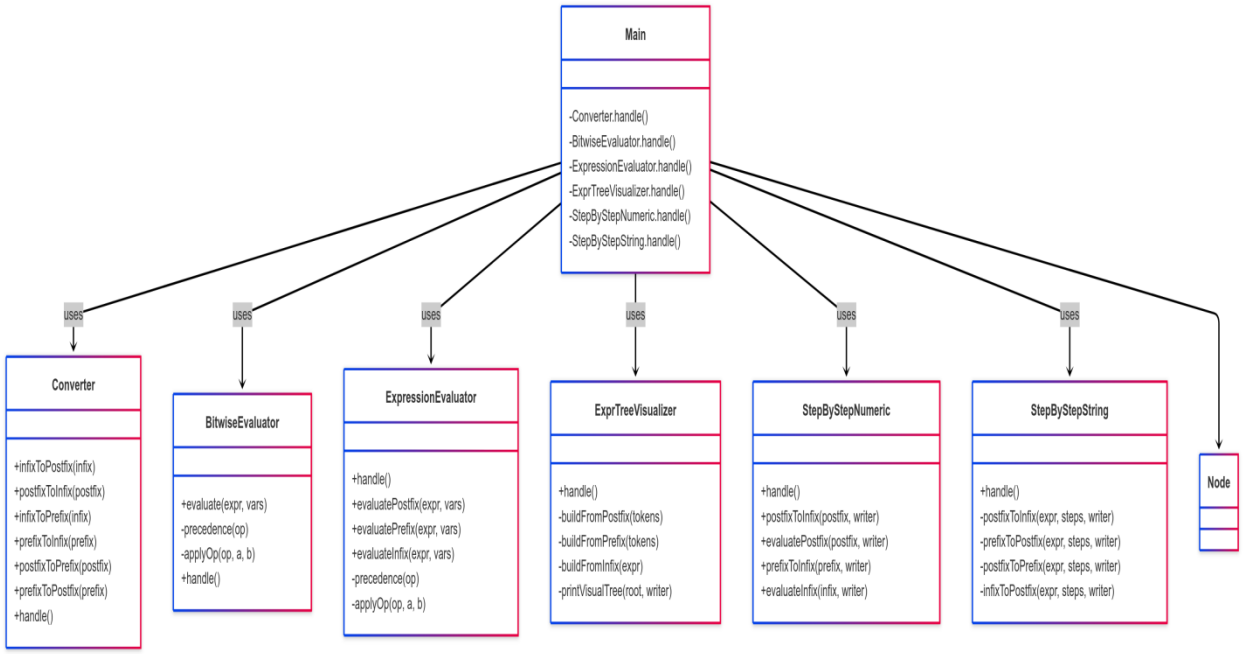
1. Accept and correctly interpret expressions in infix, prefix, and postfix notations.
2. Support both numeric literals (e.g., 4, 5.2) and symbolic variables (e.g., a, x1).
3. Log every session, including user inputs, transformation steps, errors, and final outputs, into input.txt and output.txt, each entry timestamped in ISO 8601 format.
4. Build and display expression trees in ASCII form, clearly showing parent-child operator relationships.
5. Evaluate arithmetic, symbolic, and bitwise expressions, with proper precedence handling and operator associativity.
6. Prompt user-friendly CLI interactions with validation at each input stage.
7. Handle and report syntax errors, undefined variables, or division-by-zero errors gracefully without crashing the session.

8. Allow users to review previous logs or rerun past expressions using saved input records.

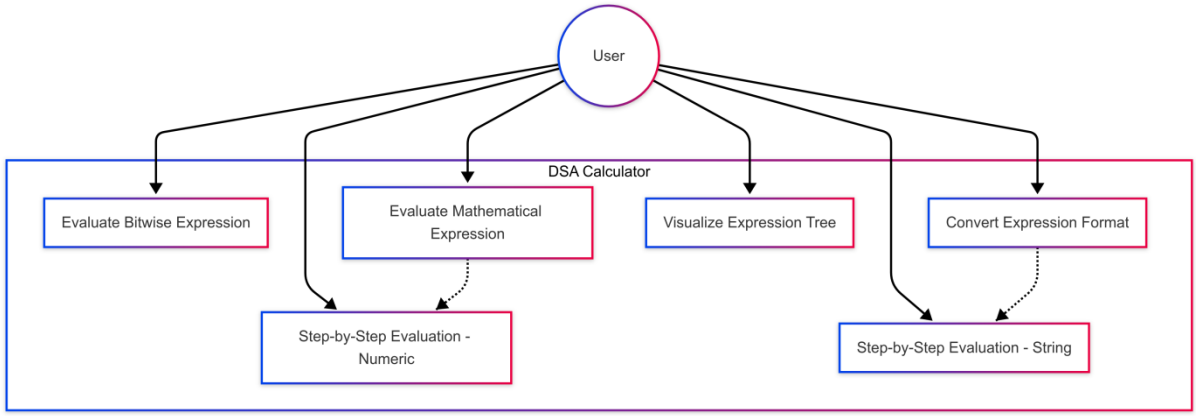
3.5 Non-Functional Requirements

- **Portability:** Must run on any system supporting Java 8 or above without external dependencies.
- **Performance:** Should exhibit linear or near-linear time complexity for tokenization, conversion, and evaluation operations to ensure responsiveness even with nested or lengthy expressions.
- **Usability:** Interface should include clear menu options, progress prompts, input validation messages, and help commands for new users.
- **Maintainability:** Modular architecture with logically divided classes and interfaces must support future extension, including:
    - New operator types (e.g., logical, ternary)
    - GUI-based front-end modules
    - Integration with IDEs or developer tools
- **Reliability:** Fail-safe behavior for invalid input; no operation should cause the program to crash or produce incorrect results silently.
- **Security:** Handle user-supplied input without executing arbitrary code or exposing file-system vulnerabilities (e.g., when logging input).

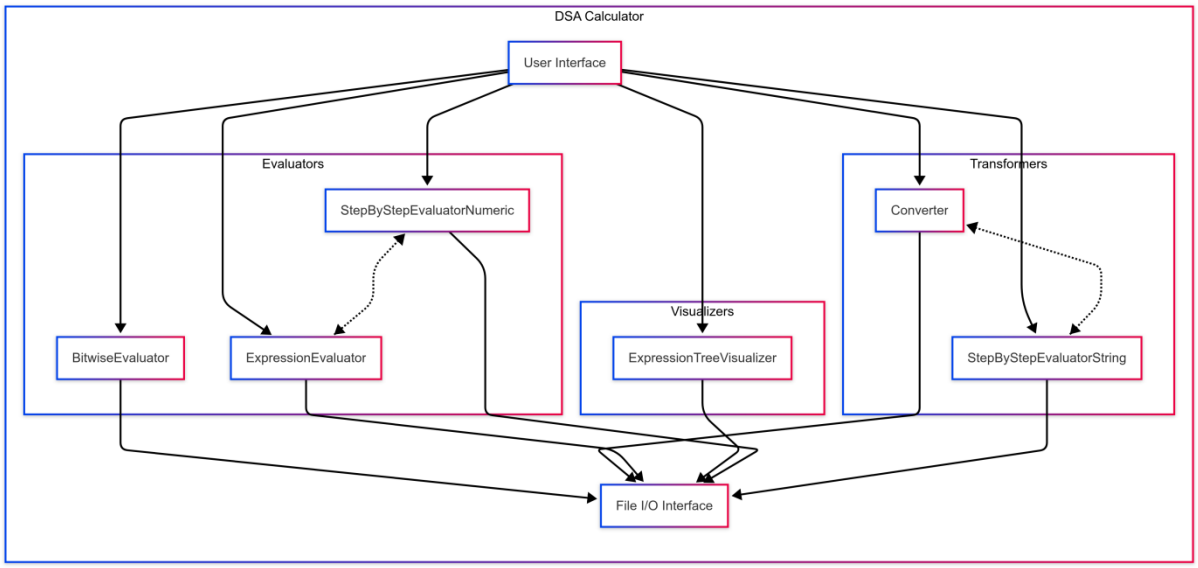
3.6 UML Diagram



3.6.1 Class Diagram



3.6.2 Use Case Diagram



3.6.3 Compoment Diagram

4. Results

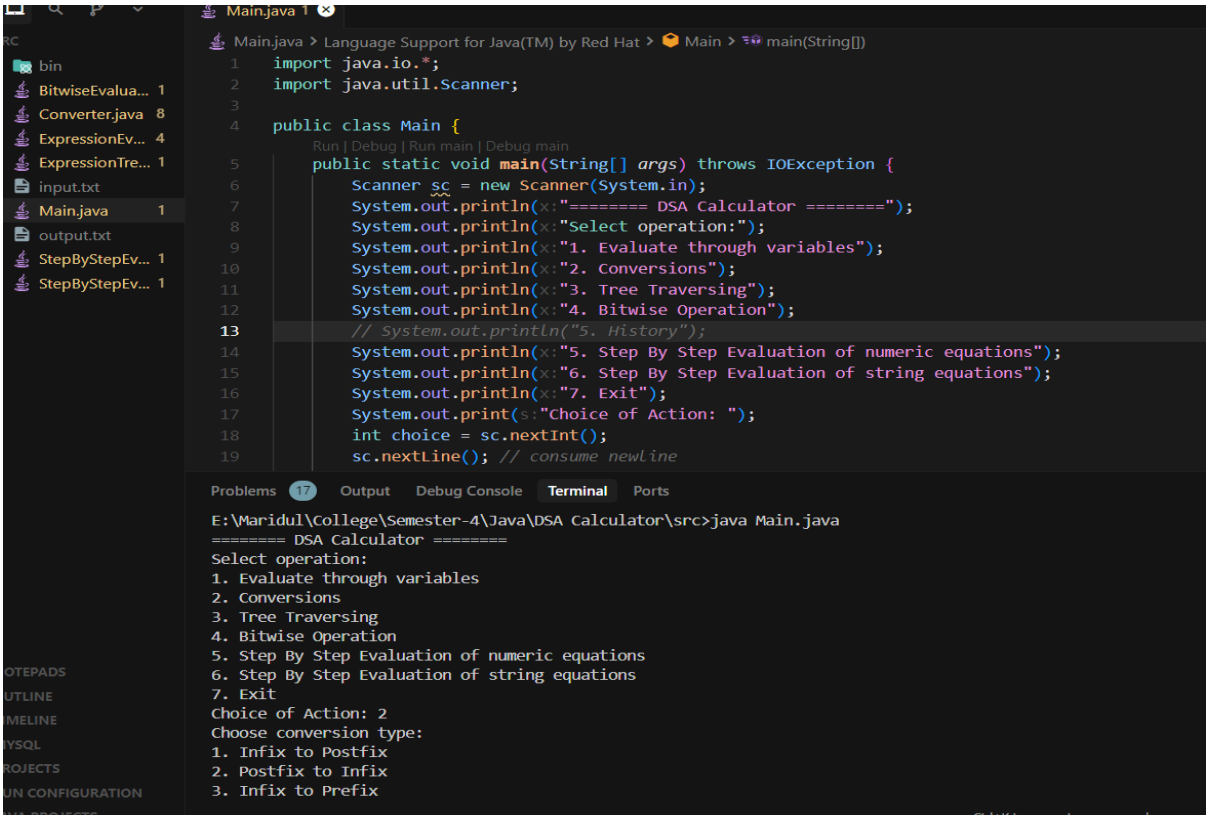
The application has been rigorously tested for its ability to handle various expression formats, convert between notations and evaluate results correctly (both numerically and bitwise), generate clear visualizations, and log every session for audit and debugging purposes.

4.1 Conversion Examples

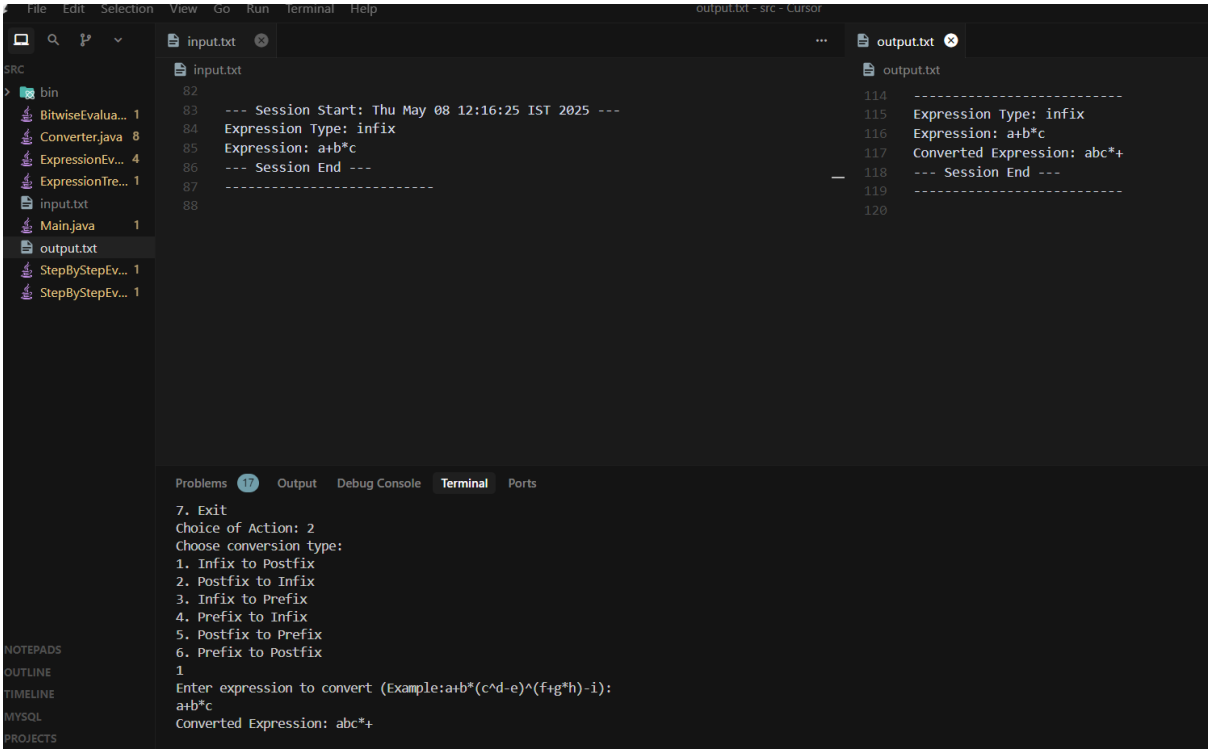
The application successfully handles the conversion between various expression notations. Below are a few tested examples:

- **Infix to Postfix**
    - Input: a+b*c
    - Converted: abc*+
    - Explanation: Multiplication has higher precedence than addition, so 4 * 5 is grouped first.

      Results: -



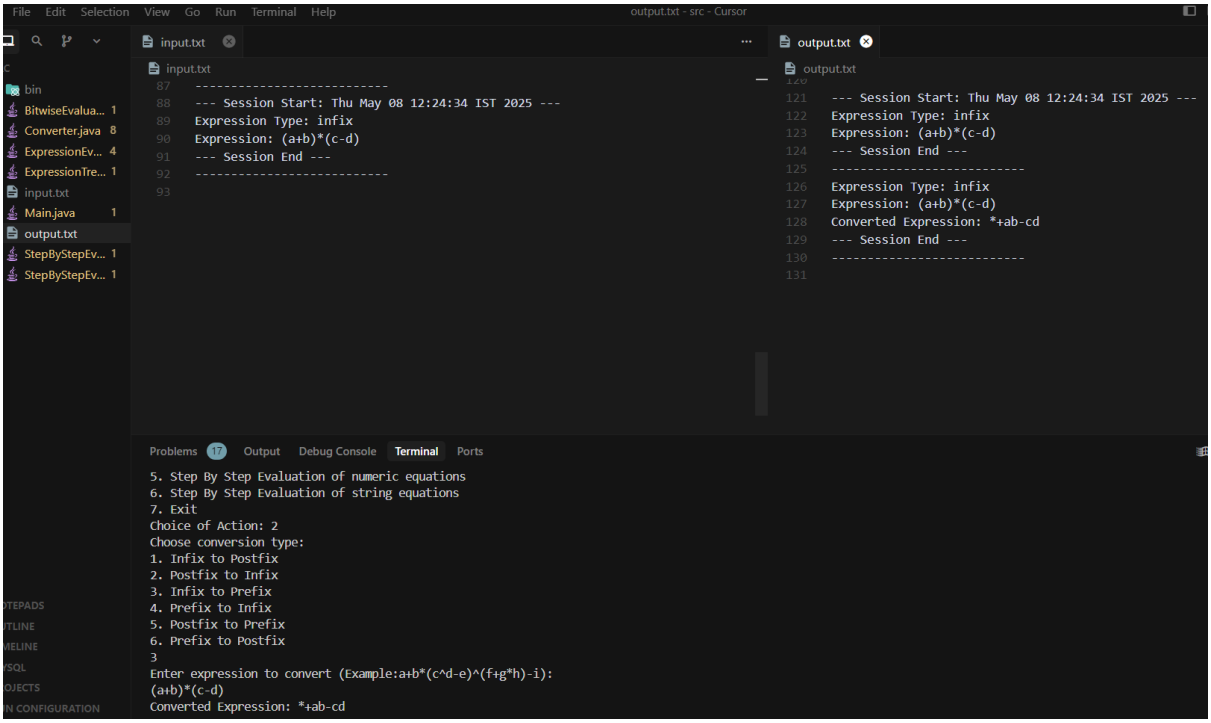
4.1.1 Loading Main class and choosing the options to perform operation



4.1.2 Entering the expression, getting a result and logging the same in files

- **Infix to Prefix**
    - Input: (a+b)*(c-d)
    - Converted: *+ab-cd
    - Explanation: Parentheses ensure a + b and c - d are evaluated first; then the result is multiplied.

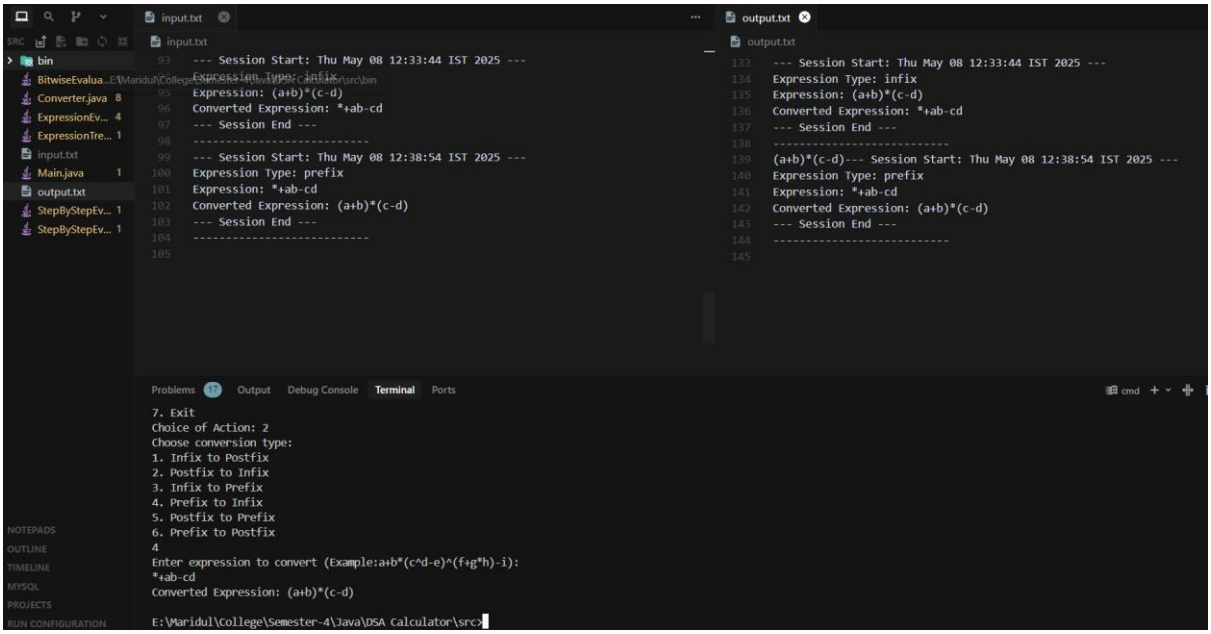
      Results: -



4.1.3 Entering the expression, getting a result and logging the same in files

- **Prefix to Infix**
    - Input: *+ab-cd
    - Converted: ((a+b)*(c-d))

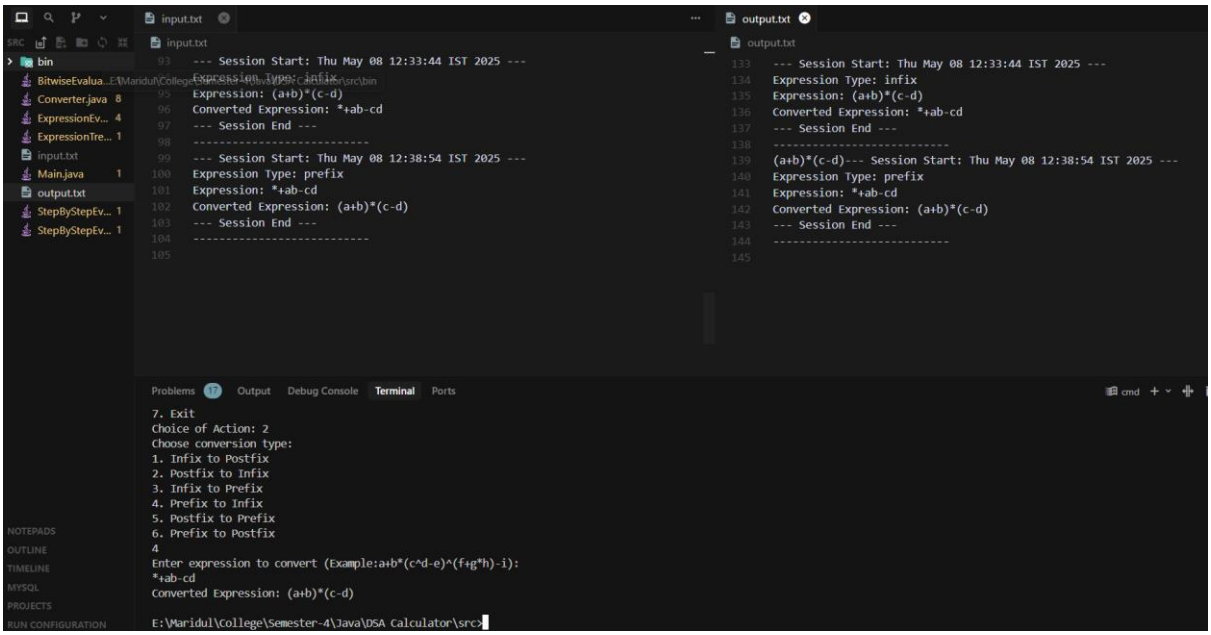
      Results: -



4.1.4 Entering the expression, getting a result and logging the same in files

- **Postfix to Infix**
    - Input: ab+cd-*
    - Converted: ((a+b)*(c-d))

Results: -



4.1.4 Entering the expression, getting a result and logging the same in files

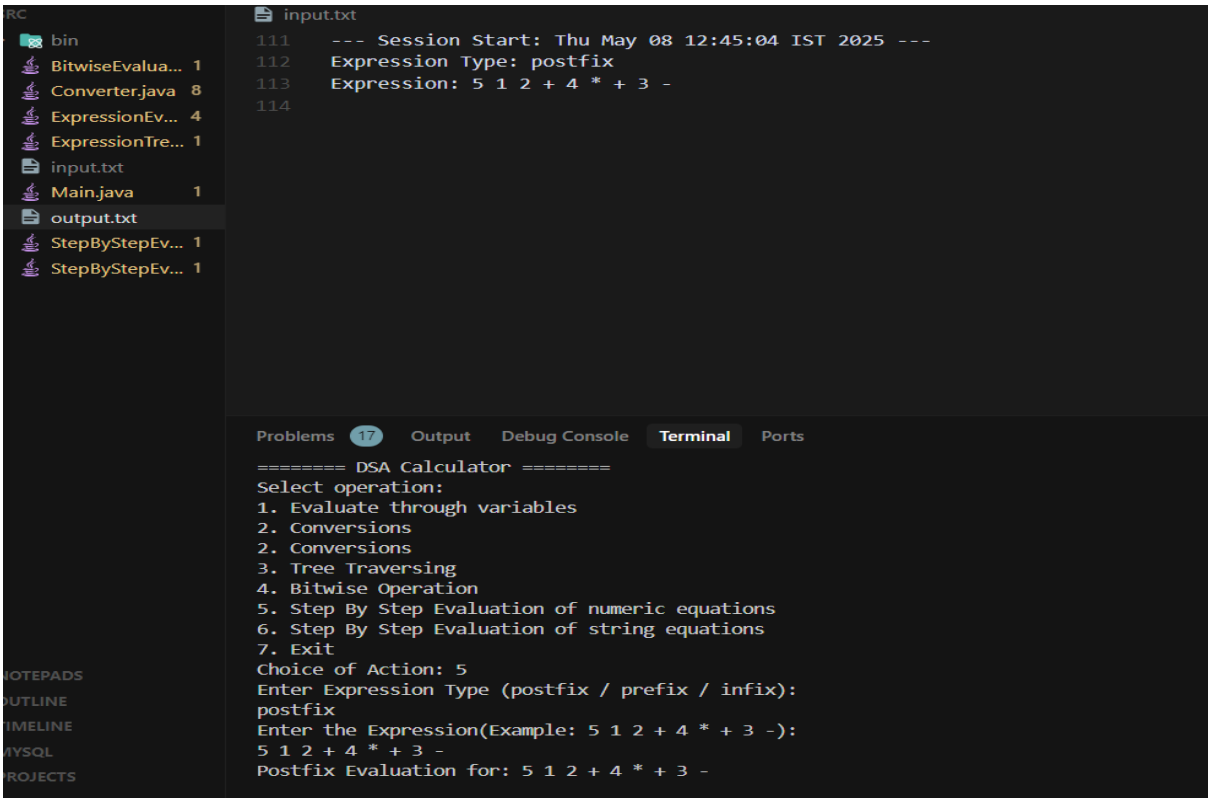
Each conversion is accompanied by step-by-step logs that show how tokens are processed, how operators are pushed/popped from the stack, and the final result of the conversion.

4.2 Evaluation Examples

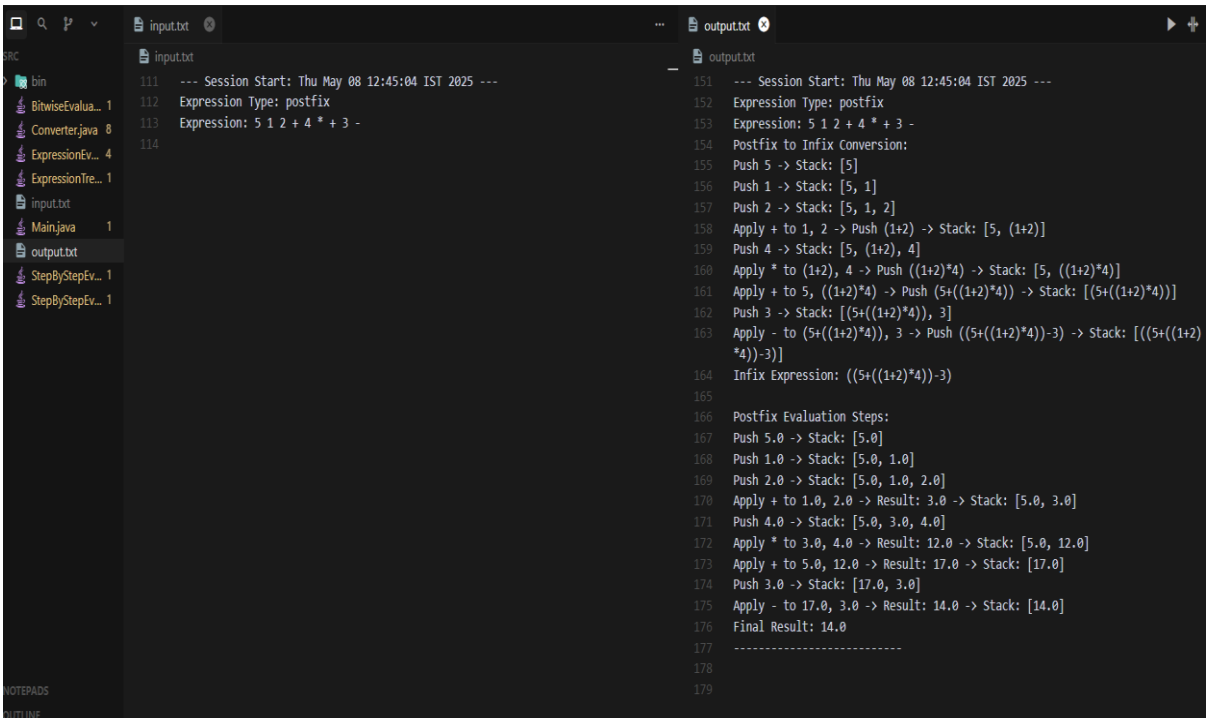
The application supports arithmetic and bitwise evaluation using postfix or infix input expressions. Step-by-step stack logs are generated to help understand the process.

- **Numeric Evaluation (Postfix)**
    - Input: 5 1 2 + 4 * + 3 -
    - Steps:
        1. Push 5
        2. Push 1, Push 2
        3. Evaluate 1 + 2 = 3, push 3
        4. Push 4
        5. Evaluate 3 * 4 = 12, push 12
        6. Evaluate 5 + 12 = 17, push 17
        7. Push 3
        8. Evaluate 17 - 3 = 14
    - Output: 14.0

Results: -



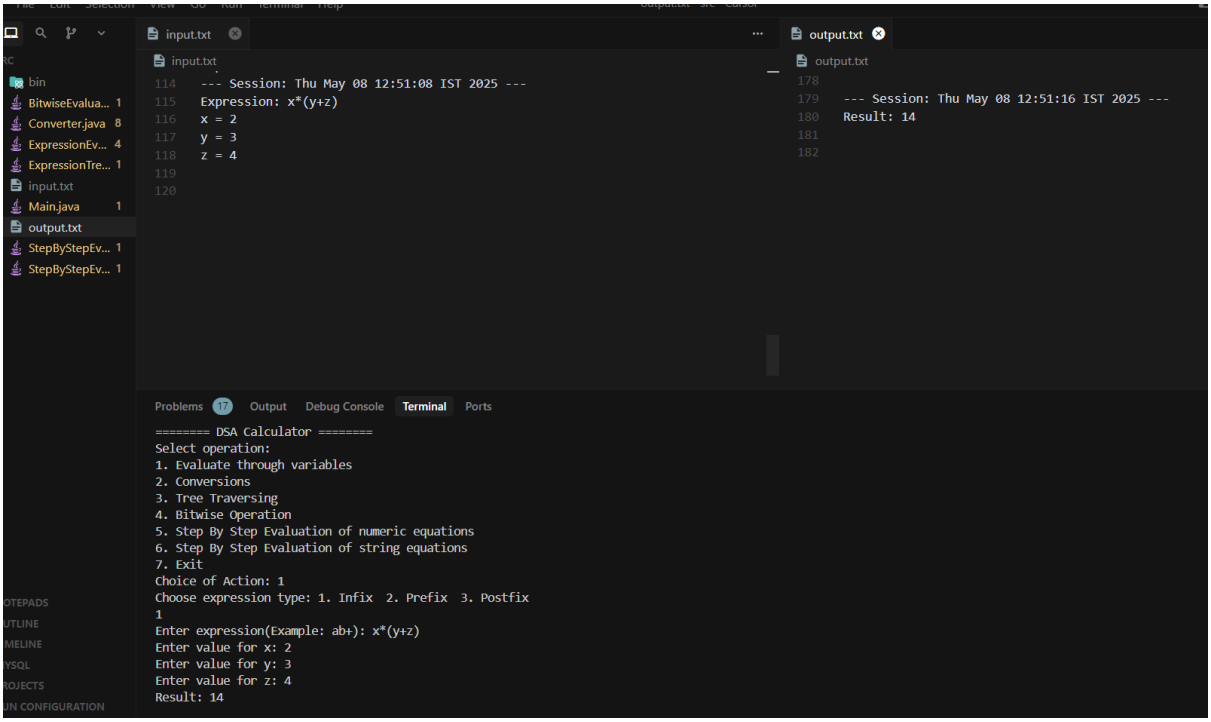
4.2.1 Entering the expression



4.2.2 Logging of Steps in the output.txt

- **Variable-Based Evaluation (Infix with prompt)**
    - Input: x*(y+z)
    - Prompt: User is asked for values of x, y, and z
    - Given: x = 2, y = 3, z = 4
    - Output: 2 * (3 + 4) = 14

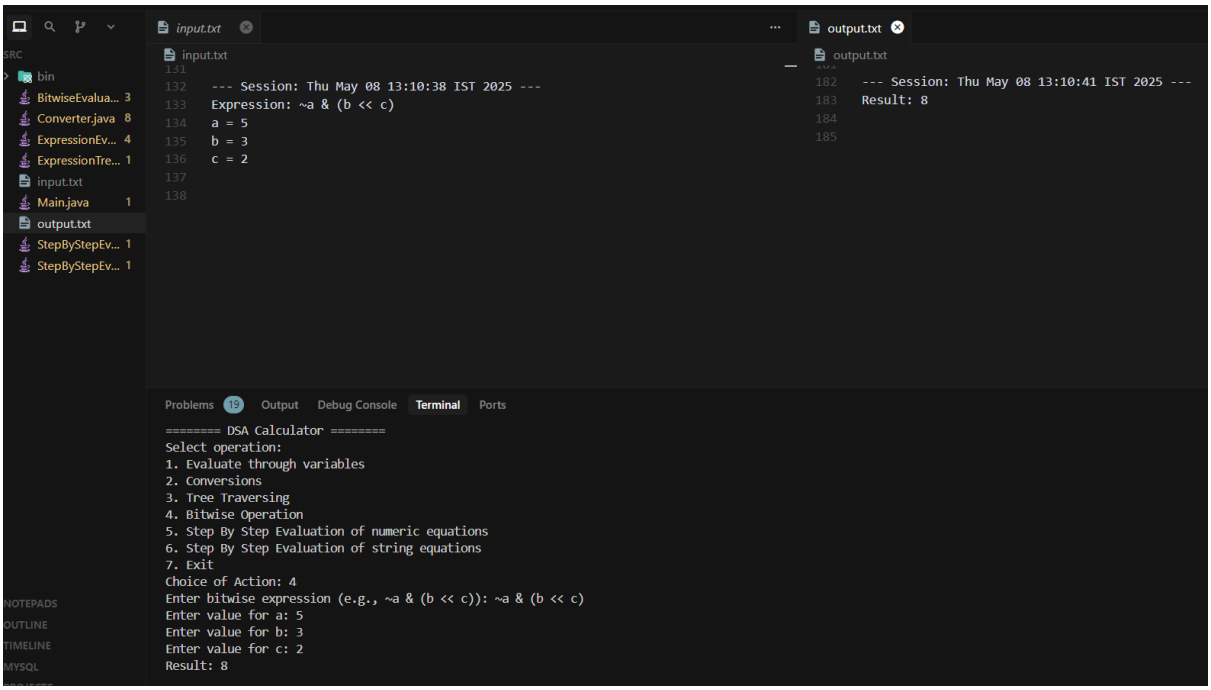
      Results: -



4.2.3 Entering the expression, getting a result and logging the same in files

- **Bitwise Evaluation**
    - Input: ~a&(b<<c)
    - Prompt: User is asked for values of a, b, and c
    - Given: a = 5, b = 3, c = 2
    - Steps:
        1. ~5 = -6 (bitwise NOT)
        2. 3 << 2 = 12 (left shift)
        3. -6 & 12 = 8 (bitwise AND)
    - Output: 8

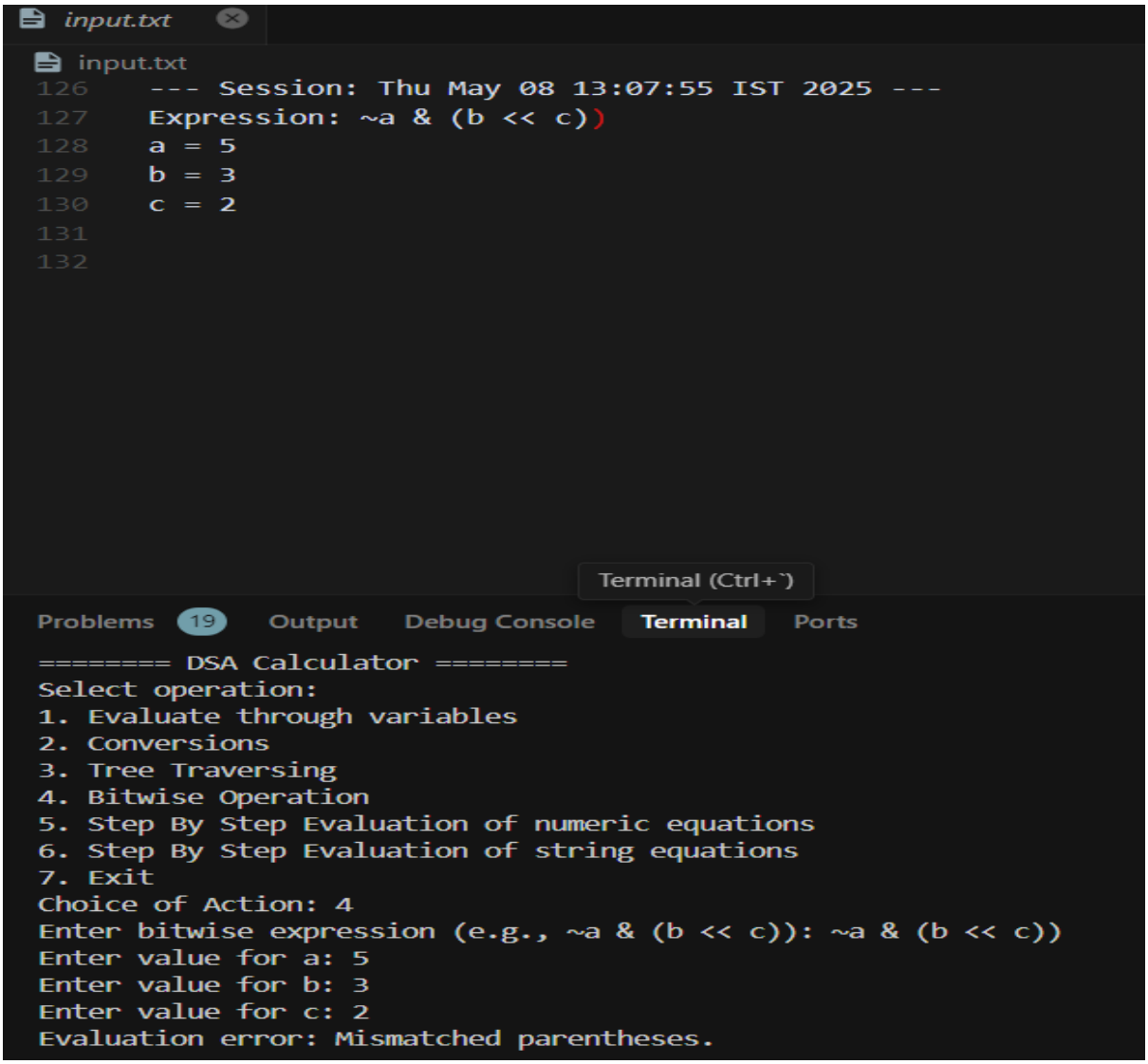
      Result: -



4.2.4 Entering the expression, getting a result and logging the same in files

- **Error Detection**
    - Input: ~a & (b << c))
    - Output: Error - Evaluation error: Mismatched parentheses.

      Results: -



4.2.5 Entering the expression, getting a error, displayed in input.txt

4.3 Visualization Samples

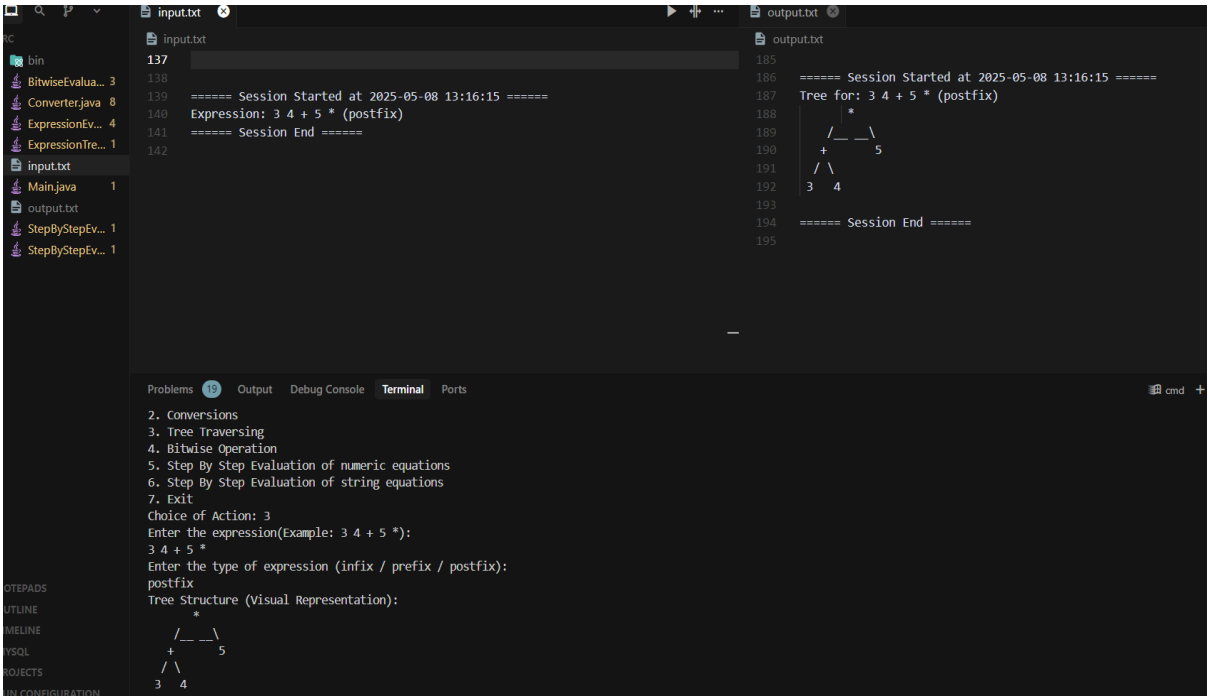
Expressions can be transformed into expression trees and visualized using ASCII graphics, aiding in understanding operator hierarchy and structure.

Example: Postfix Expression 3 4 + 5 *

- **Tree Structure**:

```
      *

   / -- -- \

  +          5

 / \

3   4
```

Results: -



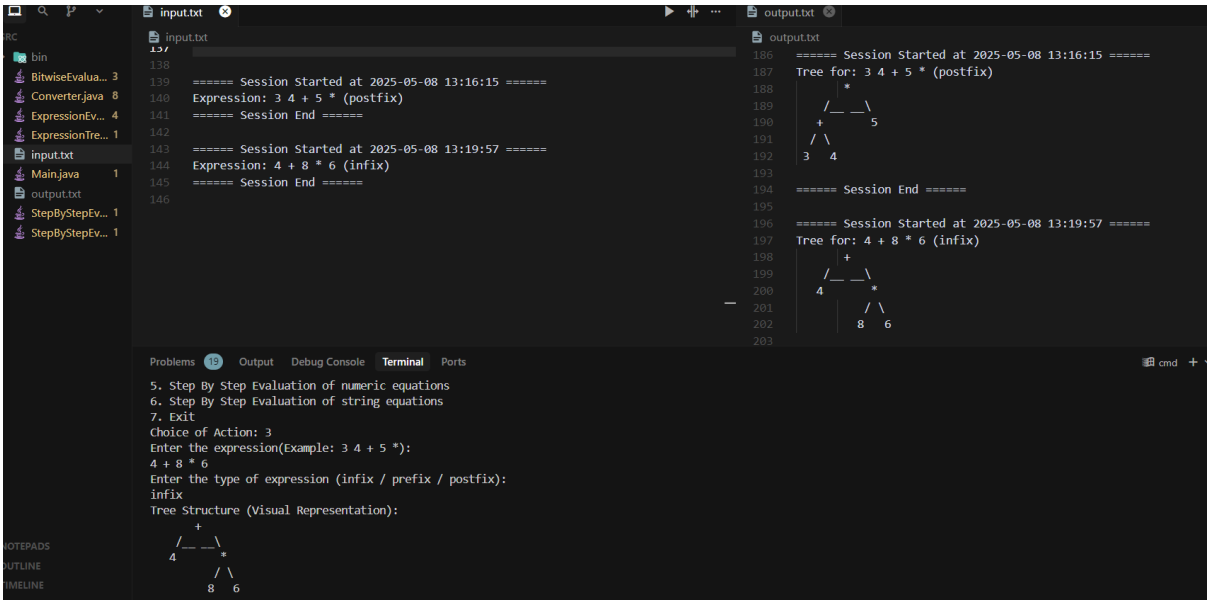
4.3.1 Entering the expression, getting a result and logging the same in files

Example: Infix 4 + 8 * 6

- **Tree Structure**:

```
  +

 / -- -- \

4        *

         / \

        8   6
```

Results: -



4.3.2 Entering the expression, getting a result and logging the same in files

Features:

- Operators serve as internal nodes.
- Operands are leaf nodes.
- ASCII layout reflects depth and branching.
- Could be extended to draw with indentation or levels.

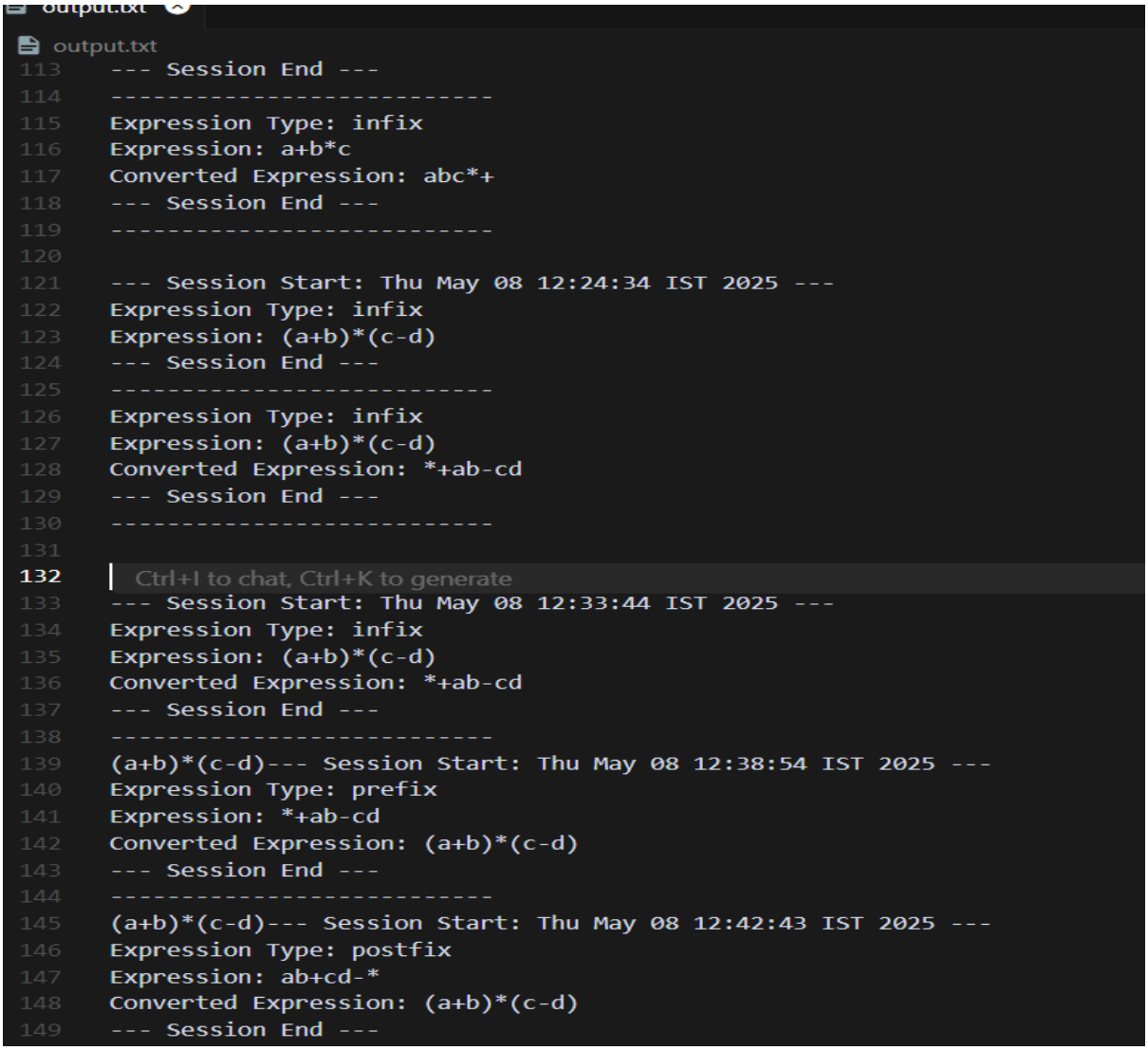
4.4 Logging Verification

The system maintains an output.txt file (or similar, depending on configuration) to log each session in a structured and traceable manner. This ensures that both debugging and evaluation history can be audited post-execution.

Log Format:

- Begins with ISO-formatted timestamp (e.g., 2025-05-08 13:19:57)
- Logs the selected operation (conversion, evaluation, visualization)
- Includes:
    - Input expression
    - Token stream
    - Step-by-step actions (stack pushes/pops, intermediate results)
    - Final output or error messages
- Ends with a session delimiter: --- Session End ---

Example Log Snippet:



This structured logging supports:

- **Traceability**: Easy backtracking of steps.
- **Reproducibility**: Logs can be replayed or used to reconstruct the session.
- **Debugging**: Quick identification of faulty logic or malformed input.

5. References

- GeeksforGeeks - Data Structures and Algorithms in Java
    - https://www.geeksforgeeks.org/data-structures/
- Java Documentation - Oracle
    - https://docs.oracle.com/en/java/javase/
- TutorialsPoint - Java Data Structures
    - https://www.tutorialspoint.com/data_structures_algorithms/index.htm
- Programiz - Data Structures in Java
    - https://www.programiz.com/java-programming/data-structures
- Baeldung - Java Collections Framework
    - https://www.baeldung.com/java-collections
- Stack Overflow
    - https://stackoverflow.com/