ByteMath

# Arithmetic Expression Evaluator
# Software Architecture Document

Version 1.0

## Revision History

| Date | Version | Description | Author |
|---|---|---|---|
| <dd/mmm/yy> | <x.x> | <details> | <name> |
| 10/11/24 | 1.0 | Added information for Section 5.0-5.2 | Aidan Prather |
| 10/11/24 | 1.0 | Added information for Section 8 | Ellie Thach |
| | | | |

# Table of Contents

# Software Architecture Document

## 1. Introduction

This Software Architecture Document provides a detailed guide to the design and structure of the Arithmetic Expression Evaluator, a C++ component within a larger compiler project for language L. It covers the architectural choices and rationale behind the system's design, including parsing, operator handling, and error management. This document is intended for developers, stakeholders, and maintainers to understand and implement the system effectively.

### 1.1 Purpose

The purpose of this document is to outline the architecture of the Arithmetic Expression Evaluator, capturing key decisions and structures. It will serve as a reference to ensure consistent implementation, integration, and maintenance of the evaluator, addressing both current development and future enhancements.

### 1.2 Scope

This document applies specifically to the Arithmetic Expression Evaluator, detailing its architectural structure, components, and interactions. It focuses on the evaluator's role in parsing and calculating arithmetic expressions and excludes other parts of the compiler.

### 1.3 Definitions, Acronyms, and Abbreviations

- **Parser**: Analyzes expressions and converts them into structured formats.
- **Evaluator**: Computes the result of a parsed expression.
- **PEMDAS**: Operator precedence rules (Parentheses, Exponents, Multiplication/Division,

  Addition/Subtraction).
- **AST**: Abstract Syntax Tree, representing the hierarchical structure of expressions.
- **CLI**: Command Line Interface for user input and interaction.

### 1.4 References

- **EECS348 Course Syllabus** – Professor Hossein Saiedian, Fall 2024.
- **Project Requirements Document** – Outlines functional and non-functional requirements for the evaluator.

### 1.5 Overview

- **Architectural Representation**: Details how the architecture is represented and broken down.
- **Logical View**: Shows core components, including the Parser and Evaluator, and their interactions.
- **Development View**: Organizes source files and modules.
- **Process View**: Describes execution flows, focusing on parsing and evaluation steps.
- **Use Case View**: Links architecture with key use cases, such as handling arithmetic expressions and errors.

## 2. Architectural Representation

The software will use an object-oriented architecture with a class for each key component. Currently, this means there will be:

- A UserInput class that gets the user input and converts it into an understandable form for the Parser
- A Parser class that populates the ExpressionTree class based on the PEMDAS order of operations
- An ExpressionTree class that will store the Integers and Operators
- An Integer class that will store the numbers in the user input
- An Operator class that defines the behavior for all required operators to be used when combining Integers in the ExpressionTree as dictated by the Evaluator class
- An Evaluator class which will control the resolution of the ExpressionTree and return the evaluated expression to the user/caller/Output class

The views are:

- Logical view: The program will combine the classes described above to display the evaluated expression to the user. The UML and class diagrams represent this view.
- Process view: The program will be organized with a package for each core class and a main program that integrates them by calling various routines from each class in an ordered manner.
- Development view: The software will be organized into packages based on the classes included.
- Physical view: All the components of the software should be run on the same machine, and they should be portable as part of a larger expression evaluator package.

## 3. Architectural Goals and Constraints

- **Accuracy and Precision**: The calculator should handle integers and, in the future, floating-point arithmetic with high precision.

- **Responsiveness:** The calculator should be fast enough to process arithmetic expressions in real-time with minimal latency.

- **Efficiency:** Efficient parsing algorithms and memory usage should be considered to handle larger expressions effectively.

- **User Experience**: Clear error messages and feedback for invalid input.

## 4. Logical View

**Important Modules:** The operations for the Arithmetic Expression Evaluator will be operational with the use of the following modules:

- **User Interface**
- **Input**
- **Parser**
- **Data Storage**
- **Evaluator**

### 4.1 Overview

The program will initialize with the User Interface module, prompting the user to input a mathematical expression. Once an input is given, the user-defined equation will be handed off to the Parser module, further interacting with the Data Storage module. Once all data has been handled, control will be handed off to the Evaluator, which will then begin making a mathematical sense of the given data. Once the Evaluator has solved the equation, it will then return the result back to the User Interface to display to the user.

### 4.2 Architecturally Significant Design Modules or Packages

- **User Interface**
  - The User Interface module is the front-end aspect of the Arithmetic Expression Evaluator, allowing the user to interact with a terminal to input data into the program. A simple UserInterface class will handle the general display of information, allowing the UserInput class to display any necessary information to receive an expression input.
- **Input**
  - The Input module oversees handling the user-defined input. By using the UserInput class, the user will enter an expression to be solved. Once entered, UserInput will process the data into a workable form, then runs an ErrorHandler class to confirm that no invalid information was given in the input. If error is found, it will be displayed according to the UserInterface. Once the input is cleaned and error-free, the information will be handed off to the Parser module for further processing. If input is received from an external source/program, it will be directly fed into the UserInput class and handled similarly.
  - **UserInput:** Contains the logic to handle intaking and converting the input into a form usable by the Parser.
  - ErrorHandler: Error checking will be included to confirm the given input is a workable expression, considering both invalid mathematic input as well as generic input errors.
- **Parser**
  - Once data is given to the Parser, the module will be responsible for segmenting the input into data that can be saved within the Data Storage module and later referenced in by the Evaluator. The Parser calls the ExpressionTree class to handle the mathematical data of the expression, ordering the data respectively to the order of operations.
- **Data Storage**
  - Data Storage will include any important variables, structures, or likewise information that would need to be stored or referenced temporarily to perform other operations. The ExpressionTree and Integer classes will be directly changed by the output of the Parser module, having the results generated from the given user input as the basis for the data. An Operator class will also be included within Data Storage.
  - **ExpressionTree**: Stores integers and operators in PEMDAS order to be used by the Evaluator module.
  - **Operator**: Contains the default behaviors for mathematical operators.
- **Evaluator**
  - Once all data has been processed and ready for evaluation, the Evaluator module completes any final computations. Referencing the data within the ExpressionTree structure created by the Parser module, an Evaluator class operates over the structure to yield a final result of the given input. Once a result is returned, it will be returned to the User Interface module to display the answer back to the user. Additionally, the results will be stored within Data Storage in case an external program would need to reference a specific result for any operations.

## 5. Interface Description

- **Screen Layout:** The terminal will display a prompt for the user to enter an arithmetic expression. Upon execution, the result is shown, followed by the prompt again. This process repeats until the user exits the program.

- **Prompts and Result Display:** The prompt will always display `>`, indicating that the user can input their arithmetic expression. After the expression is entered, the result of the calculation is displayed on the next line under the prompt, preceded by the text "**Result:**". The user can type **`exit`** to quit the application at any point.

- **Arithmetic Expressions**: The user can input simple arithmetic expressions that may include:

a. **Operators**: `+`, `-`, `*`, `/`, `%`
b. **Parentheses** for grouping expressions: `(`,`)`
c. **Integers**: Whole numbers (positive and negative)
d. **Spaces**: Spaces should be ignored between numbers and operators.

- **Errors in Input**:
    a. Invalid arithmetic operators: `++`, `--`, etc.
    b. Mismatched parentheses: `5 + (3 * 2`.
    c. Division by zero: `10 / 0`.
    d. Non-numeric characters that are not part of operators.

If the user types **`exit`**, the program will display **end of program** message and terminate

## 6. Quality

**Extensibility**: By using compartmentalized functions, we are building a modular system that can be easily expanded with more complex features in the future. This approach will keep functionality well-organized, support easier updates, testing, maintenance, and ensuring our system can adapt as needed.

**Reliability**: Providing informative and clear feedback on invalid inputs and crashes ensures that issues can be quickly identified and diagnosed. By conducting targeted troubleshooting and highlighting areas that need enhancement, this approach contributes to a more reliable system overall.

**Portability**: By implementing the code in C++, a widely supported language, we can ensure compatibility across a broad range of platforms. This allows the code to be easily compiled and executed on any major operating system, promoting portability and accessibility.

**Maintainability**: Including clear and descriptive comments throughout the code will help other team members understand the function and purpose of each portion, making it easier to follow. Proper documentation ensures that updates, troubleshooting, and further development can be managed well and be able to maintain the program well.