Kansas Instruments

Arithmetic Expression Evaluator Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
10/31/2024	1.01	Assigning roles	Emma, Kit, Khang, Vrishank
11/3/2024	1.02	Completed section 1.5, section 3, and added “Modularity” to section 1.3 and a reference in 1.4	Emma
11/4/2024	1.03	Completed sections 6 and 7 and added “UI” to section 1.3 and a reference in 1.4	Phoenix
11/7/2024	1.04	Completed sections 2 and 4.1, added “Software system’s architecture” to section 1.3 and a reference to 1.4	Khang
11/8/2024	1.05	Completed section 8 “Quality” and added to definition(Extensibility, Scalability) and references.	Jakob
11/9/2024	1.06	Completed sections 1,1.1,1.2	Kit
11/10/2024	1.07	Completed sections 5 and 5.1, added to 1.3 and updated formatting	Vrishank
11/10/2024	1.08	Completed sections 5.2, 4, and 4.1	Aiden

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document for the Arithmetic Expression Evaluator provides an organized architectural overview of the system. This document includes critical elements like purpose, scope, and definitions necessary to understand the architecture. It describes the different architectural views and decisions that structure the system, ensuring a reliable and extensible evaluation framework. This document aims to facilitate the development and enhancement of the evaluator by providing a consistent architectural reference for developers, testers, and stakeholders.

1.1 Purpose

This document serves as a comprehensive architectural overview for the Arithmetic Expression Evaluator, capturing significant architectural decisions and representing various aspects of the system through views, such as logical, process, development, and physical. Its purpose is to act as a technical guide during the development process, offering a foundation for implementing the design and code. It provides a clear documentation resource for stakeholders and everyone else. This document is overall made for everyone such as group members, quality testers, and academic reviewers who need to understand the architectural decisions and follow the project's progress/planning.

1.2 Scope

The Software Architecture Document applies to the Arithmetic Expression Evaluator project, guiding the structural and functional aspects of its components. It details the organization of modules responsible for parsing, evaluating, error handling, and displaying results. The document covers essential functionalities, such as parsing arithmetic expressions, evaluating them using the order of operations (PEMDAS), handling errors, and providing a command-line interface. It also addresses error management by explaining how the system handles invalid inputs or operations, such as division by zero, and delivers error messages.

1.3 Definitions, Acronyms, and Abbreviations

- **Modularity**: The design principle of breaking a system into distinct, self-contained components or modules, each responsible for a specific functionality, making the system easier to understand, maintain, and modify.
- **UI**: Short for user interface, the frontend visual aspect the user interacts with and reads.
- **Software system's architecture**: The set of principal design decisions made about a system to be developed
- **Extensibility**: The ability of the software to accommodate future changes and enhancements with minimal impact on existing functionality.
- **Scalability**: The capability of the software to handle an increasing volume of work or expand its functionality without compromising performance.
- **AST (Abstract Syntax Tree)**: A tree representation of the syntactic structure of the input expression, where each node represents a construct occurring in the source code (in this case, operators and operands in the arithmetic expression).
- **PEMDAS**: The order of operations for arithmetic expressions: Parentheses, Exponents, Multiplication, Division (from left to right), Addition, and Subtraction (from left to right). This rule is essential for correctly evaluating expressions with multiple operators.
- **Lexer:** A tokenizer that converts a sequence of characters into a sequence of tokens. Tokens are smaller, meaningful units that the system can understand, such as keywords, operators, literals, or symbols.

1.4 References

- https://www.castsoftware.com/glossary/what-is-software-architecture-tools-design-definition-explanation-best#:~:text=Software%20architecture%20is%2C%20simply%2C%20the,used%20to%20design%20the%20software.
- https://www.onlinegdb.com/online_c++_compiler
- Hierarchical Architecture.

- https://syndicode.com/blog/12-software-architecture-quality-attributes/
- https://en.wikipedia.org/wiki/Modular_programming
- https://en.wikipedia.org/wiki/Lexical_analysis

1.5 Overview

The Software Architecture Document provides a guide to the organization and structure of the document, explaining what each part contains and its purpose within the overall architectural description. It helps readers navigate the document by summarizing key sections like the Introduction, which includes the purpose and scope of the architecture; Architectural Representation, which describes how the architecture is structured and the models used; Architectural Goals and Constraints, outlining critical architectural requirements; Use-Case View, describing how major functionalities are implemented; Logical View, detailing the design model decomposition; and further sections on interfaces, size and performance, and quality considerations. This section ensures that readers understand the document layout and can quickly locate detailed descriptions of the system's architecture.

2. Architectural Representation

The Arithmetic Expression Evaluator's architecture can be represented by:

Logical View (Components): The components work in a hierarchical structure, passing the arithmetic expression through in order of the lexer, parser, evaluator, error handler, and then display.

- Lexer - Breaks down the input expression into individual tokens separated by constants, operators, and parentheses.
- Parser - The parser depends on the lexer, building a tree based on the tokens from the lexer. The tree represents the structure and order of operations for the expression.
- Evaluator - The evaluator depends on the parser, taking the parsed expression and performing the operations from the lowest level of the tree up.
- Error handler - The error handler may depend on any of the lexer, parser, or evaluator, processing any errors found and issuing a warning of the type of error.
- Display - The display depends on the evaluator and error handler, either printing the final result from the evaluator or an error message from the error handler to the user.

Process View (Connectors): method and function calls will work fine, with each component being called on by the main function when needed. A hierarchical approach can be done with the main function calling on each function sequentially.

Development View (Configuration): The main function will initialize the UI and handle user interactions. The functions provided by components can be accessed through header files..

Rationale: The Main-Program-Subroutine architecture works perfectly due to its strengths and weaknesses. Since the team is already familiar with procedural programming, it allows for easier implementation as there is past experience already there. Additionally, the cons of less maintainability and understandability are mitigated by the project being fairly small-scale. It also means Main-Program-Subroutine architecture will allow for efficient implementation, without need for classes or pipes and filters.

Physical View (Typical run): The main function will take an arithmetic expression input from the user, which is then lexed, parsed, evaluated, and then displayed to the user. Each of the major steps in the process will be passed from component to component in the form of calls to perform each specific task in addition to processing any errors that may come up.

3. Architectural Goals and Constraints

The following requirements and objectives have a significant impact on the architecture:

- Correctness and Reliability: The evaluator must accurately parse and evaluate arithmetic expressions, taking into account operator precedence and grouping using parentheses. This is critical to ensure the correct functionality of the larger compiler product.

- Modularity and Maintainability: The architecture must support a modular design where components (e.g., tokenizer, parser, evaluator) are decoupled and can be modified independently. This modularity will make it easier to adapt the evaluator for future changes, such as handling floating-point numbers or additional operators.
- Efficiency and Performance: Parsing and evaluating expressions should be efficient to provide a smooth user experience. The architecture should utilize optimized algorithms for parsing and precedence handling, with data structures that offer efficient access and manipulation.
- Portability: Since the application is part of a larger compiler project, it must be compatible across various environments where the compiler may be deployed. The use of standard C++ libraries and cross-platform development practices will enhance portability.
- Error Handling and Robustness: The application must handle erroneous input gracefully, providing informative error messages for common issues (e.g., unmatched parentheses, invalid operators). The error-handling mechanisms must be robust to prevent unexpected crashes or undefined behavior.
- User Interface Constraints: The project specifies a command-line interface, which must be user-friendly, clear, and capable of interpreting user input accurately.
- Scalability for Future Enhancements: The design should allow for possible future expansions, such as supporting floating-point arithmetic, additional operators, or integration into more complex compiler workflows.

4. Use-Case View

Primary use cases:

- Evaluate simple arithmetic expression:
    - Trigger: User inputs a simple arithmetic expression **3+4**
    - Flow: The evaluator processes each token, builds a syntax tree, evaluates it according to the order of operations, and outputs the result
    - Outcome: The correct result **7** is displayed on the CLI
- Evaluate Expression with Parentheses:
    - Trigger: User inputs a more complex expression, such as **8-(5-2)**
    - Flow: The lexer breaks the expression into tokens, the parser constructs an Abstract Syntax Tree (AST) that reflects the precedence dictated by parentheses, and the evaluator computes the result
    - Outcome: Displays **5** as the result
- Handle Division by Zero Error:
    - Trigger: User inputs an expression like **4/0**
    - Flow: The expression goes through the lexer and parser as usual. During evaluation, a check identifies division by zero. The evaluator invokes the error handler, which interrupts the operation and displays a meaningful error message

4.1 Use-Case Realizations

A simple expression: **3+4**:

- The software begins by putting the expression through the lexer, where the expression is broken down into 3 parts: 3, +, and 4
- Afterwards, the parser will put this into a tree. Since there's only one operation, it'll be a small tree of 2 levels, with each of the 3 parts being children of the root
- The evaluator will evaluate the tree by taking 3, +, and 4 and performing the operation with the constants
- Finally, the display component will print the output to the screen or terminal

An example with parentheses: 8 - (5 - 2)

- The software first puts the expression through the lexer, where the expression is broken down into parts: 8, -, (, 5, -, 2,),

- The parser then adds the 8, -, and expression in parentheses into a tree, with the expression in parentheses becoming another subtree with 5, -, and 2 being children of the parentheses
- The evaluator starts evaluating from the lowest level, with 5-2, before calculating 8 - (result of 5-2)
- The result is displayed

An example with an error: 4 / 0

- The software will go through each step as usual, putting the expression through the lexer and parser
- Division by zero error would be checked in the evaluation phase, with the evaluator being able to recognize division by zero and passing it on to the error handler
- This will prompt a warning for division by zero error to be displayed to the user

5. Logical View

This section describes the architecturally significant parts of the Arithmetic Expression Evaluator's design model, detailing its decomposition into modules and layers. It discusses the modularity that will be implemented in our program. Each module represents a core functionality and interacts with other modules to achieve the evaluator's objectives, such as parsing, evaluating, error handling, and displaying results.

5.1 Overview

The design of the Arithmetic Expression Evaluator follows a modular and layered approach. Each core function of the evaluator is encapsulated within specific components. This design allows the system to maintain a separation of the different components. Each component has distinct responsibilities and interacts with other components in a controlled manner. The primary modules are:

1. **Lexer Module**: Responsible for breaking down the input expression into tokens. Each token represents a small part of the expression, such as constants, operators, and parentheses. The lexer identifies these tokens and passes them to the parser for further processing.
2. **Parser Module**: Takes the tokens provided by the lexer and arranges them into a structured format, typically an abstract syntax tree (AST). This structure represents the order of operations and hierarchy in the arithmetic expression, laying the foundation for accurate evaluation.
3. **Evaluator Module**: Traverses the AST generated by the parser to calculate the result of the expression. It applies mathematical rules and order of operations (PEMDAS) to evaluate sub-expressions before combining them to generate the final result.
4. **Error Handler Module**: Monitors the process for potential issues, such as division by zero or unmatched parentheses. When errors are detected, it interrupts the evaluation and passes meaningful error messages to the display module, ensuring robust error handling.
5. **Display Module**: Communicates the final result or error messages to the user. This module handles formatting the output for the command-line interface, allowing the user to understand the outcome of their input expression or the reason for any errors.

Each module is designed with a clear input-output relationship with other modules, which simplifies debugging, testing, and future enhancements. The layered architecture also facilitates potential modifications, such as introducing additional operators or more complex error handling, without disrupting the overall system structure.

5.2 Architecturally Significant Design Modules or Packages

Lexer Package:

- Description: Responsible for tokenizing the user’s input into identifiable symbols like operators, constants, and parentheses.

- Lexer Class: Handles the breakdown of raw input strings into tokens, classifying each as an operand, operator, or grouping symbol.

Parser Package:

- Description: Constructs an Abstract Syntax Tree (AST) based on the tokens from the lexer, representing the structure of the expression and operator precedence
- Classes:
    - Parser: Transforms tokens into an AST that respects PEMDAS. Manages errors like unbalanced parentheses
    - ASTNode: Represents each node in the AST, containing references to operands, operators, and their relationships

Error Handler Package:

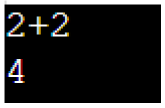
- Description: Detects and manages errors such as syntax errors, division by error, or unmatched parentheses
- ErrorHandler Classes: Analyzes error states from the lexer, parser, or evaluator, issuing warnings to the user.

Display Package:

- Description: Formats and outputs either the computed result or any error messages to the CLI
- Display Class: Provides a simple interface to output results or errors

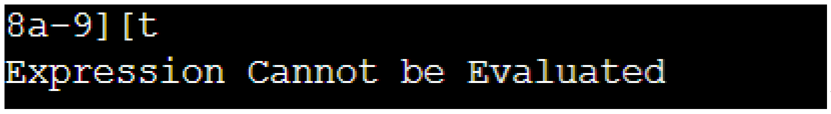
6. Interface Description

The interface will be a standard console input. The user may put in any expression and we evaluate if it is nonsense or not. We then will display a new line for the user for their output, then prompt the user for more expressions or to quit. Here are some examples of the input output interface:



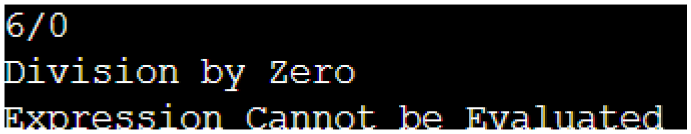
```
2+2
4
```

Standard input



```
8a-9][t
Expression Cannot be Evaluated
```

Nonsense input



```
6/0
Division by Zero
Expression Cannot be Evaluated
```

Mathematical Nonsense input

An improved UI with a more "professional" look would be a "nice-to-have" and is a secondary goal if time allows it. Mock output was generated using an online C++ interpreter (onlinegdb.com/online_c++_compiler)

7. Size and Performance

With the scale and purpose of the nature of the software, performance is not a large constraint we must deal with as most reasonable implementations should be of an adequate speed. But a goal is for a calculation to not take more than two seconds, and numeric data should be stored in either doubles or long doubles. These are high cost data types but being we will have a low number of these variables as they are frequently overwritten it will not consume high amounts of memory.

8. Quality

The architecture of this arithmetic expression parser emphasizes modularity to support a wide range of quality attributes. The design prioritizes extensibility and maintainability through modular classes and clear separation of characters, allowing new operators or expression types to be integrated without major changes. Additionally, C++ implementation ensures that the program is compatible across different operating systems, enhancing its portability and making it suitable for diverse environments.

The architecture promotes reliability and safety by incorporating strict error-handling and input validation to manage edge cases like division by zero or unmatched parentheses, preventing the parser from entering undefined states. Thorough testing, and error messages contribute to its reliability and usability, ensuring that users and developers can interact with the system. This careful validation approach also supports security by controlling accepted inputs.

Lastly, the parser is optimized for performance efficiency and scalability with data structures such as stacks and trees that follow precedence rules, allowing it to handle complex expressions with minimal overhead. Designed for usability, the command-line interface provides helpful feedback on errors and results, making the parser accessible to a range of users. Together, these qualities create a reliable, and adaptable system.