Team 14

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
11/06/24	1.0	Initial Version.	Team 14

Table of Contents

1. Introduction **4**
- 1.1 Purpose 4
- 1.2 Scope 4
- 1.3 Definitions, Acronyms, and Abbreviations 4
- 1.4 References 4
- 1.5 Overview 4

2. Architectural Representation **5**

3. Architectural Goals and Constraints **5**

4. Logical View **6**
- 4.1 Overview 6
- 4.2 Architecturally Significant Design Modules or Packages 6
  - User Interface Package: 6
  - Lexer Package: 6
  - Parser Package: 6
  - Abstract Syntax Tree Package: 7

5. Interface Description **9**

6. Quality **9**

Software Architecture Document

1. Introduction

1.1 Purpose

Within this design document, we will outline the design and structure of the Arithmetic Expression Evaluator. The evaluator shall be able to: receive an expression, parse through it, and return an answer to the user. The purpose and functionality of key modules (e.g.Lexer Package, Parser Package, etc.) shall be given in-depth description and modeling. Developers can use this document to understand the interconnecting components of the software for implementation and refinement, while Stakeholders will get to know how this software meets their requirements.

1.2 Scope

This Software Architecture Document outlines the main architecture of the Arithmetic Expression Evaluator project, which is a C++ program that evaluates mathematical expressions with support for operators and nested parentheses. The document details the architectural decisions that impact parsing, operator precedence, error handling, and command-line interface interactions. It is intended to guide developers, testers, and people who have a stake in the project in understanding the system's structure and design decisions.

1.3 Definitions, Acronyms, and Abbreviations

- Expression: A combination of numbers, operators, and parentheses to perform a calculation (e.g., 3 + (4 * 2)).
- Evaluator: The part of the program that calculates the result of an expression.
- Parsing: The process of breaking down an expression into parts for easier evaluation.
- Operator Precedence: Rules that define the order of operations (e.g., multiplication before addition).
- CLI (Command-Line Interface): A text-based interface where users enter expressions to evaluate.
- Grammar Syntax Terms
  - E -> T + E, T - E, T
  - T -> F * T, F / T, F
  - F -> Integer, (E), -F
- Tokens: The symbols and numbers that represent integers and different operators that form an expression.

1.4 References

Project Plan Document: Title- Arithmetic Expression Evaluator Project Plan, Date: 2024-10-18, Authors- Our group, Source- University of Kansas, EECS 348 Course

Software Requirements Specification: Title: Arithmetic Expression Evaluator SRS, Date: 2024-10-22, Author- Our group, Source- University of Kansas, EECS 348 Course

UPEDU Guide: Title- Guide to Software Architecture and Development, Source: UPEDU reference guide

Standard C++ Documentation, Source- https://cplusplus.com

1.5 Overview

This Software Architecture Document gives an outline of the key parts of the Arithmetic Expression Evaluator. It includes sections on the main goals of the architecture, the core components and how they interact, the program's interface, and considerations for quality and performance. Each section is designed to help readers understand how the evaluator is structured, the design decisions made, and how the program operates as a whole.

2. Architectural Representation

The software architecture of the Arithmetic Expression Evaluator in C++ is designed to be modular and maintainable, represented using a layered approach that delineates clear responsibilities among distinct packages. This system comprises four primary packages: the User Interface Package, the Lexer Package, the Parser Package, and the Abstract Syntax Tree (AST) Package. Each package encompasses specific classes and functions that collectively enable the system to parse and evaluate arithmetic expressions efficiently.

The architecture is represented using class diagrams, sequence diagrams, and component diagrams to illustrate the interactions and dependencies among packages. Each view of the system highlights essential elements, including classes, interfaces, and their relationships. For example:

- User Interface View: Represents how the user interacts with the system, capturing input, displaying results, and facilitating user commands.
- Lexer and Parser View: Showcases the flow of tokenization and parsing logic, detailing how tokens are identified and expressions are broken down into parseable structures.
- AST Representation View: Depicts the hierarchical structure of nodes that form the abstract syntax tree, each subclass representing different types of operations and operands within an expression.

3. Architectural Goals and Constraints

The design of this system aims to achieve several architectural goals and adhere to specific constraints:

Goals:

- Modularity and Maintainability: The system architecture is modular, with packages designed to encapsulate specific functionalities (e.g., parsing, tokenization, and evaluation). This modularity supports ease of maintenance and extension, such as adding support for new operators or handling floating-point numbers.
- Scalability: The architecture supports future scalability, enabling the system to incorporate additional features or enhancements with minimal disruption.
- Robust Error Handling: Ensuring the architecture can gracefully handle parsing errors, invalid expressions, and runtime issues such as division by zero is a key focus.
- Efficient Evaluation: The architecture should enable the evaluator to process arithmetic expressions with minimal computational overhead, preserving performance even for complex input.

Constraints:

- Development Tools: The project will be developed using C++ and integrated development environments (IDEs) suitable for C++ development, such as Visual Studio or CLion.
- Design Strategy: The system will employ object-oriented programming (OOP) principles, leveraging classes and functions for cohesive and reusable code.
- Portability: The architecture must ensure that the program can be compiled and run across different platforms that support C++.
- Team Collaboration: The project design aligns with team roles and responsibilities, ensuring that each team member’s work on packages or modules integrates smoothly into the overall architecture.
- Schedule Constraints: The project is bound by academic deadlines, influencing iterative development cycles and code review timelines.

4. Logical View

4.1 Overview

The program is broken up into four primary packages: User Interface, Lexer, Parser, and the Abstract Syntax Tree. The User Interface is meant to accept input of an arithmetic expression from the user and send it to the Parser package. The Parser package, in conjunction with the Lexer package, will identify the syntactic tokens of the expression along with their associated operators and construct an Abstract Syntax Tree out of it according to the rules of operator precedence. The Abstract Syntax Tree package will then evaluate the value of the constructed tree and return said value to the User Interface package for display to the user.

4.2 Architecturally Significant Design Modules or Packages

User Interface Package:

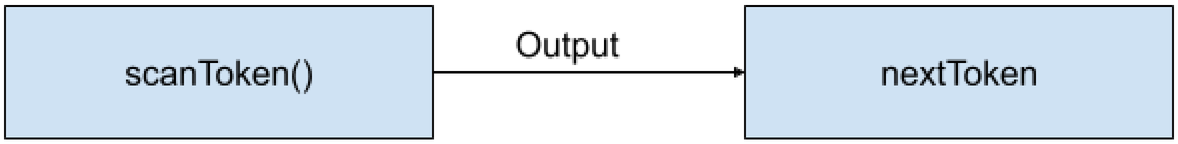
The User Interface Package manages user input and output.

GUI Class

This class serves as the primary interaction point between the user and the arithmetic expression parser. It prompts the user to enter an arithmetic expression, retrieves the input, and calls the Lexer Package to initiate the parsing algorithm. The GUI class will then display the result of the parsing algorithm and the subsequent evaluation.

Lexer Package:

The Lexer Package is a combination of a function and global variable, designed to identify tokens in a string input.



scanToken()

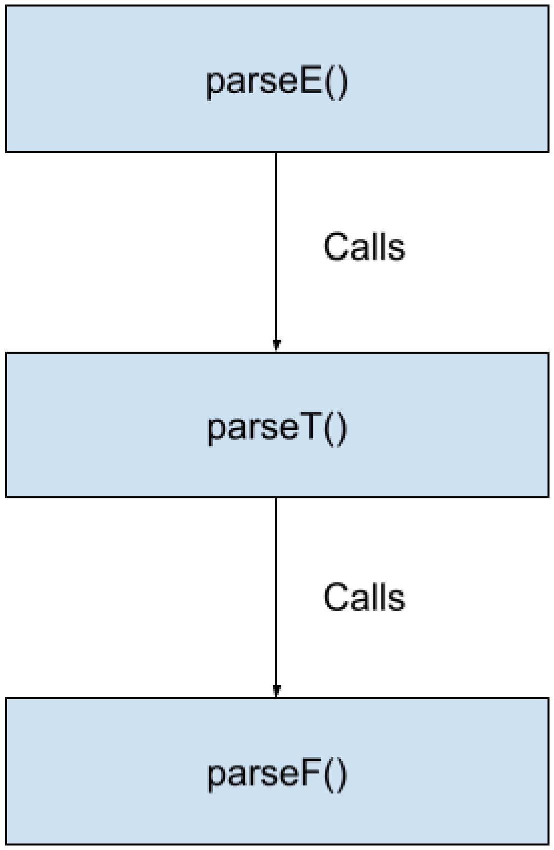
scanToken() scans current input and sets nextToken to point to the newest scanned token.

nextToken

A pointer variable intended to point at the next token to be parsed and evaluated by the Parser Package and Abstract Syntax Tree Package.

Parser Package:

The Parser Package is a package containing a hierarchy of three functions designed to parse through expressions, terms, and factors. These functions exist in a hierarchy because of the recursive descent parsing methods in use by the program.



parseE()

parseE() will scan tokens related to infix operators of addition and subtraction. From those scans, parseE() will generate a new Abstract Syntax Tree node of that operator and return a pointer to the new subtree.

parseT()

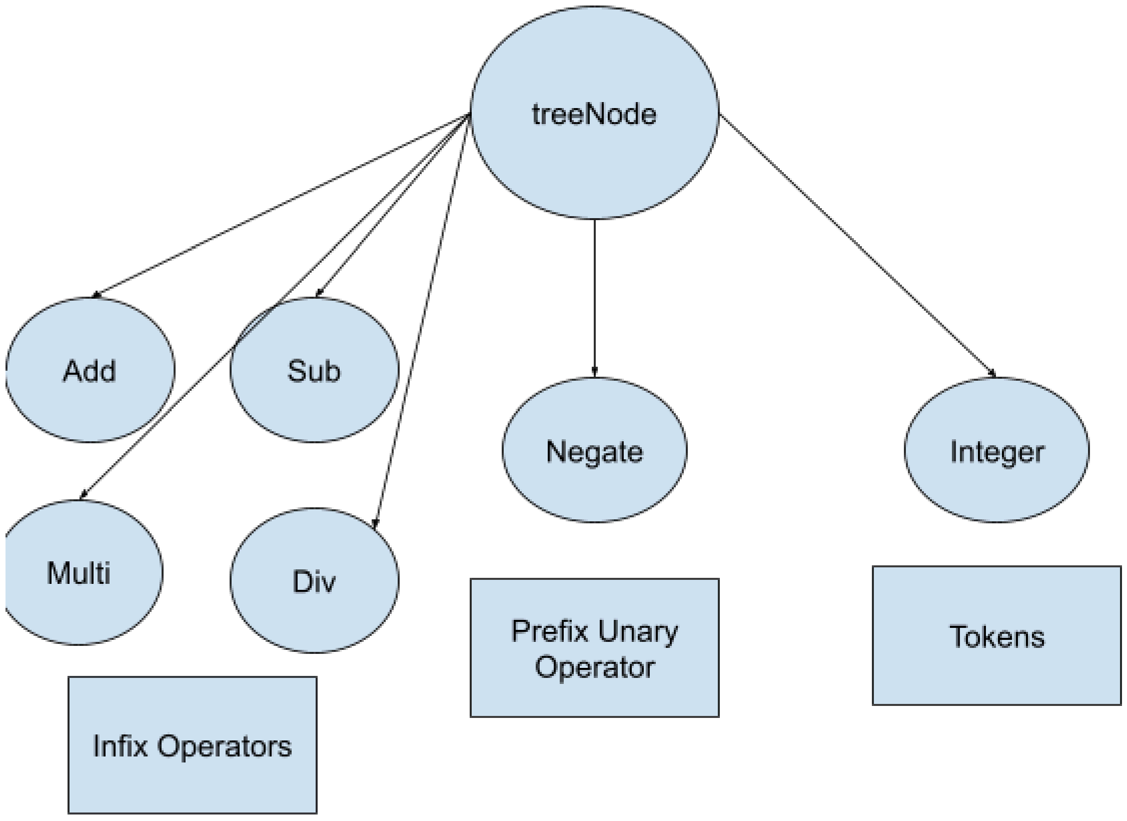
parseT() will scan tokens related to infix operators such as multiplication and division. From those scans, parseT() will generate a new Abstract Syntax Tree node of that operator and return a pointer to the new subtree.

parseF()

parseF() will scan tokens related to integers, negations, and parentheses. From those scans, parseF() will generate a new Abstract Syntax Tree node of that type and return a pointer to the new subtree.

Abstract Syntax Tree Package:

The Abstract Syntax Tree package consists of a class hierarchy meant to represent the abstract syntax tree that will be recursively parsed through for evaluation. Each subclass has a print method built in *to print out the subtree it makes up. Each subclass has an evaluation method meant to return the value of the subtree it makes up through recursive calls.*



treeNode

A superclass designed to be a generic node template.

Add

A subclass of treeNode that stores two pointer variables: left and right. The variables point to the left and right treeNode of the Add node respectively. Add's evaluation method returns the sum of the values of its left and right treeNode.

Sub

A subclass of treeNode that stores two pointer variables: left and right. The variables point to the left and right treeNode of the Sub node respectively. Sub's evaluation method returns the difference of the values of its left and right treeNode.

Multi

A subclass of treeNode that stores two pointer variables: left and right. The variables point to the left and right treeNode of the Multi node respectively. Sub's evaluation method returns the product of the values of its left and right treeNode.

Div

A subclass of treeNode that stores two pointer variables: left and right. The variables point to the left and right treeNode of the Multi node respectively. Sub's evaluation method returns the quotient of the values of its left and right treeNode.

Negate

A subclass of treeNode that stores one pointer variable. That variable points to the treeNode representing the argument within the negation's scope. Sub's evaluation method returns the quotient of the values of its left and right treeNode.

Integer

A subclass of treeNode that stores one integer variable. That variable stores an integer value. Integer's evaluation method simply returns this integer value.

5. Interface Description

The interface will feature a simple command-line interface where users can enter their mathematical expressions to be evaluated. The program will display a prompt and subsequently accept the user's valid input. Valid inputs include integers, operators (+, -, *, /), and parentheses that determine precedence. Any invalid inputs will display an error message and re-prompt the user for another expression. Once a valid input is achieved, the result of the arithmetic expression will be returned to the user. This interface design provides seamless user interactions and ensures reliable output.

6. Quality

The modular design of the system allows for easy addition of new features and functionalities. New operators and functions can be integrated without significant changes to the existing codebase or significant cost. Robust error handling is in place to manage parsing errors, invalid expression, and runtime issues. This ensures that the software can handle unexpected inputs and provide helpful feedback to the user allowing for user-friendly experience. The software is developed in C++, allowing for a portable program that can be compiled and run on all platforms that support C++. This cross-platform compatibility allows for broad usability that doesn't require modifications for each specific platform. The use of object-oriented programming and separation of functionalities across different packages (User Interface, Lexer, Parser, AST) makes the codebase easy to understand, maintain, and extend. This is essential for future development and debugging of our software. Detailed documentation further supports the maintainability of the software The command-line interface is designed to be user-friendly allowing users to easily input expressions and receive feedback accordingly with clear and unambiguous error messages that guide users in correcting inputs. The architecture also supports future scalability, enabling the system to incorporate additional features or enhancements. The scalability allows for the system to grow and adapt to new user requirements over time. All these quality attributes aim to deliver an efficient and user-friendly tool that meets the needs of the users and stakeholders.