Taus

Arithmetic Expression Evaluator
Software Requirements Specifications

Version 1.0

Revision History

Date	Version	Description	Author
11/10/24	1.0	Filled out the form	Jonathan Gott, Charlie Doherty, Will Calhoun, Josh Dwoskin, Caleb Hite, Calder Kamman

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the Arithmetic Expression Evaluator project. It captures and conveys the significant architectural decisions made for the system, utilizing multiple architectural views to represent different aspects of the software structure.

1.2 Scope

The architecture outlined in this document applies to the development and integration of the Arithmetic Expression Evaluator component, which will be part of a larger compiler project for language L. It focuses on parsing and evaluating arithmetic expressions while ensuring modular design to support future enhancements.

1.3 Definitions, Acronyms, and Abbreviations

- CLI: Command-Line Interface
- PEMDAS: Parentheses, Exponents, Multiplication and Division, Addition and Subtraction
- SRS: Software Requirements Specification
- UML: Unified Modeling Language

1.4 References

- EECS 348 Term Project Specification (2024)
- Project Plan for Arithmetic Expression Evaluator, Version 1.2, September 25, 2024
- Software Requirements Specification (SRS) for Arithmetic Expression Evaluator, Version 1.0, October 20, 2024

1.5 Overview

This Software Architecture Document details the architecture for the Arithmetic Expression Evaluator, covering its design goals, significant architectural decisions, and various views such as logical, use-case, and interface representations.

2. Architectural Representation

The software architecture for this project will be represented using UML diagrams, including class diagrams to show the relationships between components, and sequence diagrams to illustrate the flow of operations within the evaluator. The architecture emphasizes modularity to facilitate changes and integration into larger systems.

3. Architectural Goals and Constraints

Key architectural goals include modularity, extendibility, and ease of integration with other compiler components.
Constraints include the limited development timeline, initial focus on integer-only inputs, and adherence to C++ standards.

4. Use-Case View

4.1 Use-Case Realizations

The primary use case is "Evaluate Expression":
- The user inputs an arithmetic expression via the CLI.
- The evaluator takes the input, computes the result according to operator precedence, and returns the outcome.
- Error cases, such as invalid syntax or division by zero, are handled with an informative error messages displayed.

5. Logical View

5.1 Overview

The system's logical view includes the main components such as the parser, evaluator, and error handler. These components interact to interpret and process arithmetic expressions in a modular manner, with each component responsible for a specific part of the computation process.

5.2 Architecturally Significant Design Packages

Key packages include:
- **Parser Package**: Tokenizes input and prepares expressions for evaluation.
- **Evaluator Package**: Handles arithmetic operations and applies operator precedence.
- **Error Handler Package**: Manages error detection and reporting for invalid expressions.

6. Interface Description

The CLI interface accepts user input in the form of arithmetic expressions and outputs either the evaluated result or error messages. Interfaces for internal components, such as function calls between the parser and evaluator, are designed to ensure clear data flow and error handling.

7. Size and Performance

The evaluator is designed to be lightweight, suitable for systems with minimal resources. Typical expressions should be processed in under a second, and memory usage is optimized to handle deeply nested expressions efficiently.

8. Quality

The architecture supports quality attributes like modularity, reliability, and maintainability. Error handling ensures robustness, and the modular design allows for future extensions, such as supporting floating-point operations.