Kansas Instruments

Compiler Expression Parser
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
11/07/2024	1.0	Started and finished software architecture design	All

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document provides a comprehensive architectural overview of the Arithmetic Expression Evaluator system, behavior, and interactions of system components. This document conveys critical architectural decisions and structural insights, facilitating understanding and development.

1.1 Purpose

The purpose of this document is to provide a detailed description of the system's architecture, using multiple architectural perspectives to capture and communicate the design of the Arithmetic Expression Evaluator project. It aims to clarify the functional components, their interactions, and their organization to support design, development, and testing. This document is essential for developers to understand the architectural design for implementation. Testers to design appropriate test cases aligned with the system's architecture. The structure includes the introduction, architectural representation, logical views, data, and dynamic process descriptions, providing a holistic understanding of the system.

1.2 Scope

This document applies specifically to the Arithmetic Expression Evaluator project, a key part of a larger compiler for language L in EECS348. The architecture affects the system's internal operation, focusing on parsing, evaluating arithmetic expressions, and handling operator precedence, with a detailed focus on code quality, error handling, and correct functionality.

1.3 Definitions, Acronyms, and Abbreviations

SRS: Software Requirements Specification
CLI: Command Line Interface
AST: Abstract Syntax Tree
Expression Parsing: The process of analyzing a sequence of symbols to evaluate its meaning.
Tokenization: The process of breaking a string of text into meaningful components, like numbers and operators.

1.4 References

EECS348 Term Project Outline: Provided by Professor Hossein Saiedian.
C++ Standard Documentation: Reference for syntax and library usage for C++ 17.
"The C++ Programming Language" by Bjarne Stroustrup - Addison-Wesley Publishing, a foundational resource for understanding C++ standards.

1.5 Overview

Section 1: Introduction to the purpose, scope, and organization of the document.
Section 2: Architectural representation, which explains the architectural views used.
Section 3: Dynamic behavior and detailed workflow to handle expression parsing, tokenization, evaluation, and error handling.
Section 4: Guidelines for implementing operator overloading, error handling, and modular design principles

2. Architectural Representation

The software architecture for the Arithmetic Expression Evaluator system represents the overall structure of the system's components and how they interact to evaluate arithmetic expressions. Describing the major packages (e.g., Parser, Tokenizer, Evaluator), their responsibilities, and relationships. Each class has distinct roles in processing the expressions. This view aims to encapsulate specific model elements, aiding stakeholders in understanding different aspects of the system. Together, these views offer a comprehensive perspective of the system's structure and operations.

3. Architectural Goals and Constraints

The architectural goals and constraints include:

- ***Safety***: Managing and handling unexpected or invalid input without crashing.
- ***Security***: Ensuring users are unable to tamper with and manipulate code to expose any vulnerabilities.
- ***Portability***: Allowing for easy portability from one environment to another.
- ***Reuse***: Allowing the possibility of reusing the implemented code for future projects.
- ***Team Structure***: Managing proper work flow, coding requirements, and test cases through collaboration across all team roles.
- ***Design and Implementation Strategy***: Referencing the project plan and software requirements in order to successfully design and implement the architecture.
- ***Development Tools***: Using GitHub, Git, VS Code, and others for development and testing.

4. Use-Case View

N/A

4.1 Use-Case Realizations

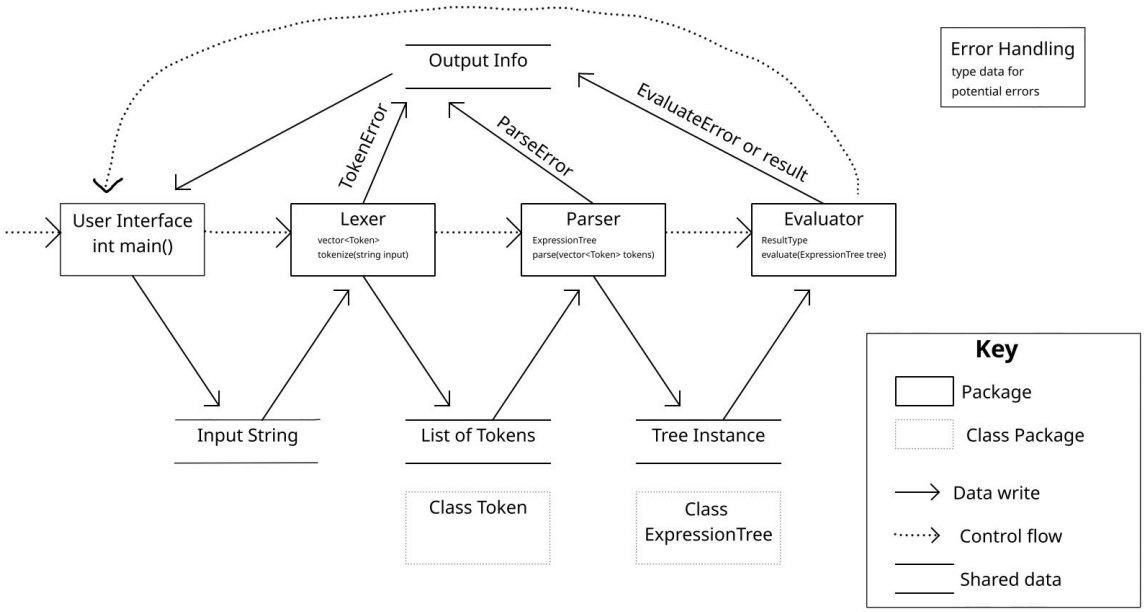
N/A

5. Logical View

5.1 Overview

This subsection describes the overall decomposition of the design model in terms of its package hierarchy and layers.

5.2 Architecturally Significant Design Modules or Packages



User Interface: The primary executable package of the project will be responsible for taking user input via a welcoming CLI, and passing it along to the relevant modules. It must be able to anticipate any result (e.g. an error) and properly communicate it to the user through the CLI.

Lexer: Responsible for transforming an input string into a list of distinct tokens. Functionality should be extensible with the recognition of new tokens.

> **Class Token:** A class storing necessary information for the parser and evaluator to process each token. May be extended with subclasses as needed (e.g. Operator, Value).

Parser: Takes a list of tokens from the lexer and transforms it into a tree, where each node is an operator and each leaf is a value. Should be arranged based on the correct order of operations.

> **Class ExpressionTree:** A class which stores Tokens as an AST.

Evaluator: A routine to traverse the expression tree from the parser and output the most simplified result.

Error Handling: A package containing type information to ensure UI and consuming packages can provide detailed reports when something goes wrong.

Testing (not pictured): A secondary executable package or set of executable packages responsible for ensuring that all software components pictured work as expected, and continue to work following updates.

6. Interface Description

User Input:

- User will be prompted to input an arithmetic expression to be evaluated in the command line interface.
- Valid user input involves any of the previously defined operators: exponentiation, multiplication, division, modulo, addition, and subtraction as well as matching parentheses and numeric constants.
- Example of Valid Input: (5 + 2) * 3 % 2
- Example of Invalid Input: ((3 / 2) - 50 ** 2

Resulting Output:

- Output will be displayed in the command line interface.
- Errors will prompt the user to input another arithmetic expression until valid input is entered.
- Example of Output: 20

7. Size and Performance

N/A

8. Quality

Reusability: This project prioritizes reusability by using functions that will be called over and over again so that we don't have to write the same code multiple times, which will also make the program run much more efficiently.

Maintainability: By writing proper comments throughout the code describing what each section properly does, it will be easy for others to understand what is happening within the program and be maintained in the future.

Extensibility: By properly utilizing compartmentalized functions, we hope to achieve a modular approach that would be easy to expand upon in the future with more complicated and in-depth features.

Portability: By writing the code in a common language, C++, which means that the code will be able to be easily compiled and run on any major computer of any major operating system.

Reliability: By giving good feedback on crashes and invalid inputs, it will provide enough information that it will easy to identify when the project fails and how it could be improved to be made more stable.