Done by: Dmitrii Machurak
Date: 10/26/2023

High and Low level design documentation for Algebraic Expression Evaluator

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this high-level design document is to provide an overview of the architecture and key components of the Algebraic Expression Evaluator application.

1.2 Scope

The scope of this document covers the high and low level architectures, major components, and the primary functions of the Algebraic Expression Evaluator. It outlines the system's key features, and system architecture.

2. System Architecture

2.1 High-Level

2.1.1 Overview

The Algebraic Expression Evaluator follows a modular and layered architecture to provide efficient expression evaluation. It consists of several key components that work together to parse and evaluate algebraic expressions.

2.1.2 Components

The main components of the system include:

- Expression Parsing
- Error Handling
- Mathematical Function Handling
- Utility Functions

2.1.3 Data Flow

Data flows through the system as follows:

1. Users input algebraic expressions for evaluation.
2. The Expression Parsing component breaks down the expression and parses individual characters in the string.
3. Mathematical functions are evaluated by the Mathematical Function Handling component.
4. The Error Handling component identifies and reports any errors at any previous step.
5. The results are provided to users through external interfaces

2.2 Low-level

2.2.1 Expression parsing

Expression parsing is at the core of the algebraic expression evaluator and plays a critical role in breaking down input expressions into their constituent parts for further evaluation.

1. **Input Expression Analysis:** The input expression is processed character by character, with each character analyzed to determine its type. The parser distinguishes between:
    - **Numbers:** Numeric values, including integers and decimal numbers.
    - **Operators:** Arithmetic operators such as addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^).
    - **Parentheses:** Both opening and closing parentheses (and curly braces) are identified for grouping and prioritizing operations.
    - **Functions:** Mathematical functions recognized by the parser (e.g., sqrt, sin, cos) for evaluation.
2. **Recursive Subexpression Parsing:** When an opening parenthesis or curly brace is encountered, the parser steps into a subexpression. The subexpression is parsed recursively, ensuring that the correct order of operations is maintained. This recursive approach handles nested subexpressions efficiently.
3. **Operator Precedence:** The parser enforces operator precedence rules, ensuring that operators are evaluated in the correct order. This is critical for accurate arithmetic calculations.
4. **Whitespace Handling:** Whitespace characters are removed from the input expression during the parsing process. This allows users to format expressions for readability without affecting the evaluation.
5. **Syntax Error Detection:** The parser detects and reports syntax errors, such as unbalanced parentheses or operators without operands. In the event of a syntax error, it triggers the Error Handling component to generate informative error messages.
6. **Function and Variable Identification:** The parser identifies and distinguishes between mathematical functions and variables. For functions, it recognizes the function name and identifies the argument that follows.
7. **Order of Processing:** The parser proceeds through the expression in a linear fashion, ensuring that each character is processed and interpreted in the correct sequence.

2.2.2 Expression Evaluation

1. **Postfix Notation:** The parser provides the expression in postfix notation (also known as Reverse Polish Notation or RPN) to the Expression Evaluation component. Postfix notation simplifies the calculation process by explicitly defining the order of operations. In postfix notation, operators follow their operands.
2. **Stack-Based Algorithm:** The Evaluation component employs a stack-based algorithm to perform arithmetic operations in postfix order. The stack data structure is instrumental in this process.

3. **Iterative Process:** The evaluation process is iterative and involves the following steps:
    - The characters of the expression are processed from left to right.
    - When a number is encountered, it is pushed onto the stack.
    - When an operator is encountered, the top elements of the stack are popped (operands), and the operation is performed. The result is then pushed back onto the stack.
    - This process continues until the entire expression is evaluated.
4. **Arithmetic Operations:** Standard arithmetic operations, including addition, subtraction, multiplication, division, and exponentiation, are performed according to the postfix order. The Evaluation component accurately handles these operations.
5. **Result Calculation:** Once the entire expression is processed, the final result is after the end of recursive calls. This result represents the outcome of the algebraic expression evaluation.
6. **Error Handling:** In the event of any errors or issues during the evaluation, the Evaluation component interacts with the Error Handling mechanism to generate informative error messages.

2.2.3 Error Handling

1. **Syntax Error Detection:** The system is equipped to detect various types of syntax or semantic errors that may occur in the input expression. Syntax errors include, but are not limited to:
    - Unbalanced parentheses: The system checks for matching opening and closing parentheses to ensure expressions are properly grouped.
    - Operators without operands: The parser identifies cases where operators are used without the required number of operands.

    And semantic errors include logical and arithmetical errors.
2. **Undefined Function Detection:** Similar to variable references, when a mathematical function is called in the expression, the system verifies whether the function is predefined and supported. If an undefined function is detected, the system generates an error message specifying that the function is not recognized.
3. **Informative Error Messages:** Error messages are designed to be clear and informative. They indicate the type of error, its location within the expression, and, in some cases, suggestions for correction. For example, if an undefined variable is detected, the error message might include the variable name for reference.
4. **Error Reporting Mechanism:** The error handling component interacts with the rest of the system to generate and report errors. When an error is detected, it communicates with other components, such as the expression parser and evaluation component, to halt the process and generate the appropriate error message.

5. **User-Friendly Experience:** The goal of error handling is to provide a user-friendly experience. When users encounter errors, they receive informative feedback that guides them in identifying and rectifying issues in their input expressions.

2.2.4 Data Flow

Data flows from parsing module to the evaluation module recursively until the expression ends. at every stage of the flow, the data is being checked for errors. See figure 1.

1. **Expression Parsing:** The Expression Parsing component processes the input expression character by character, distinguishing between numbers, operators, parentheses, and functions.
2. **Mathematical Function Handling:** Mathematical functions are recognized, and their arguments are evaluated.
3. **Error Handling:** The Error Handling component detects and reports errors, providing informative error messages to users.
4. **Expression Evaluation:** The evaluated result is generated and presented to the user.

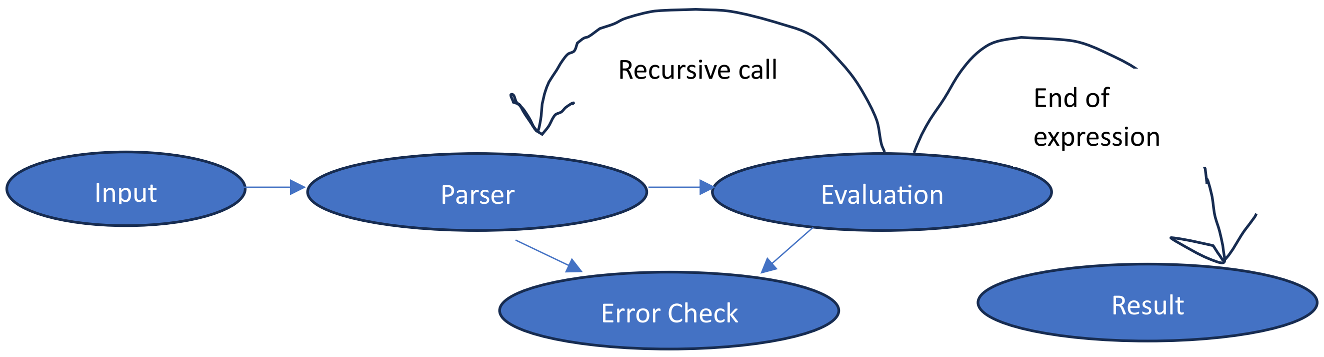

Figure 1 Data Flow

3. External Interfaces

3.1 Java API Interface

3.2 Console Interface (Optional)

- An optional command-line interface may be used for manual testing and debugging purposes.

4. Dependencies

4.1 Java Runtime Environment

4.2 External Libraries

- The system uses standard Java libraries for mathematical operations.