EECS348: Term Project in C++

Project Title: Arithmetic Expression Evaluator in C++

Project Objective

The aim of this project is to create a C++ program that can parse and evaluate arithmetic expressions containing operators +, -, *, /, %, and ^ as well as numeric constants. The program should be able to handle expressions with parentheses to define precedence and grouping.

This is a software engineering project, and as such the emphasis extends beyond the final product; it encompasses the development process. Your project deliverables will include a meticulously crafted project plan, a requirements document, a design document that seamlessly aligns with the specified requirements, a set of rigorous test cases derived from the requirements and the design, and ultimately, the fully realized product. This holistic approach ensures that each component and stage of the project is interlinked and coherent, where the design directly reflects the stipulated requirements, and the code implementation faithfully mirrors the intricacies of the design. The requirements for these documents will be provided to you separately to ensure clarity and consistency in the project's execution.

Project Overview

In this project, you will build a versatile arithmetic expression evaluator using C++. The program will take an arithmetic expression as input, parse it, and calculate the result according to the order of operations (PEMDAS). This project will help you reinforce your understanding of parsing techniques, data structures, and algorithm design.

Key Features

1. **Expression Parsing:** Your program should be able to parse arithmetic expressions entered by the user, taking into account operator precedence and parentheses.
2. **Operator Support:** Implement support for the following operators:
    - **+** (addition)
    - **-** (subtraction)
    - ***** (multiplication)

- **/** (division)
- **%** (modulo)
- **^** (exponentiation) -- In the future you may be asked to also allow ** as the exponentiation operator to experience change request.

3. **Parenthesis Handling:** Ensure that your program can handle expressions enclosed within parentheses to determine the order of evaluation.
4. **Numeric Constants:** Recognize and calculate numeric constants within the expression.

Project Tasks

1. **Expression Parsing:**
    - Implement a function to tokenize the input expression.
    - Create a data structure, such as a stack or a tree, to represent the expression's structure.
2. **Operator Precedence:**
    - Define the precedence of the operators according to the PEMDAS rules.
    - Implement the logic to evaluate the expression while considering operator precedence.
3. **Parenthesis Handling:**
    - Develop a mechanism to identify and evaluate expressions within parentheses.
4. **Numeric Constants:**
    - Recognize numeric constants in the input. Initially assume the input will be integers only. In the future, there may a request to accommodate floating point input values also so you experience change requests and embracing change.
5. **User Interface:**
    - Create a user-friendly and legible command-line interface that allows users to enter expressions and displays the calculated results.
6. **Error Handling:**

- Implement robust error handling to manage scenarios like division by zero or invalid expressions.

Project Guidelines

- Use object-oriented programming principles to structure your code.
- Include comments and documentation to explain the logic and functionality of your program.
- Develop unit tests to verify the correctness of your expression evaluator.
- Ensure that your program provides clear and informative error messages for invalid input.

Deliverables

1. The most common software engineering artifacts, e.g., a project management plan, a requirements document, a design document, and test cases. These will be described in separate announcements.
2. A well-documented C++ program that can evaluate arithmetic expressions with the specified operators and features.
3. A user manual or README file explaining how to use your program, including examples.

Grading Criteria of Final Product

Your final product will be evaluated based on the following criteria (a total of 120 points):

- Correctness of expression evaluation (e.g., handling of operator precedence and parentheses) [60 points]
- Robustness and error handling [20 points]
- Code quality, including structure and readability [20 points]
- Documentation and user manual quality [20 points]

Note: Feel free to explore additional features or optimizations beyond the specified requirements to enhance your project. Good luck and have fun coding!

Examples of Valid and Invalid Expressions

Valid Expressions

Remember that in a valid expression, operators and operands must be correctly matched, and the expression must adhere to mathematical rules (e.g., no division by zero).

1. **Addition: 3 + 4**
    - Result: **7**
    - Explanation: This expression adds two numeric constants, resulting in a valid calculation.
2. **Subtraction with Parentheses: 8 - (5 - 2)**
    - Result: **5**
    - Explanation: The parentheses ensure that the subtraction inside them is performed first, leading to the correct result.
3. **Multiplication and Division: 10 * 2 / 5**
    - Result: **4**
    - Explanation: The multiplication and division operators are applied from left to right, resulting in the final answer.
4. **Exponentiation: 2 ^ 3**
    - Result: **8**
    - Explanation: The **^** operator calculates 2 raised to the power of 3.
5. **Mixed Operators: 4 * (3 + 2) % 7 - 1**
    - Result: **5**
    - Explanation: This expression combines multiple operators and parentheses to correctly calculate the result step by step.
6. **Complex Addition with Extraneous Parentheses: (((2 + 3))) + (((1 + 2)))**

- Result: **8**
- Explanation: While there are multiple sets of extraneous parentheses, they do not affect the validity of the expression. The addition is performed correctly.

7. **Mixed Operators with Extraneous Parentheses: ((5 * 2) - ((3 / 1) + ((4 % 3))))**
   - Result: **6**
   - Explanation: This expression combines various operators with multiple sets of extraneous parentheses, but they do not change the order of operations or the final result.
8. **Nested Parentheses with Exponents: (((2 ^ (1 + 1)) + ((3 - 1) ^ 2)) / ((4 / 2) % 3))**
   - Result: **4**
   - Explanation: This expression includes nested parentheses and exponentiation, creating complexity, but it adheres to the correct order of operations.
9. **Combination of Extraneous and Necessary Parentheses: (((((5 - 3))) * (((2 + 1))) + ((2 * 3))))**
   - Result: **12**
   - Explanation: This expression includes both extraneous parentheses and necessary parentheses to clarify the order of operations. It evaluates correctly.
10. **Extraneous Parentheses with Division: ((9 + 6)) / ((3 * 1) / (((2 + 2))) - 1)**
    - Result: **-60**
    - Explanation: Extraneous parentheses are added for clarity, but they do not affect the validity of the expression. The division, multiplication, and subtraction are performed correctly.
11. **Combining Unary Operators with Arithmetic Operations: +(-2) * (-3) – ((-4) / (+5))**
    - Result: **6.8**

- Explanation: This expression combines unary **+** and **-** operators with multiplication, division, and addition.

12. **Unary Negation and Addition in Parentheses: -(+1) + (+2)**
    - Result: **1**
    - Explanation: Unary negation and addition operators are used within parentheses, followed by addition.
13. **Negation and Addition with Negated Parentheses: -(-(-3)) + (-4) + (+5)**
    - Result: **-2**
    - Explanation: This expression demonstrates nested unary negations and additions, with some values negated and others added.
14. **Unary Negation and Exponentiation: +2 ^ (-3)**
    - Result: **0.125**
    - Explanation: The unary **+** and **-** operators are used with exponentiation to calculate a fractional result.
15. **Combining Unary Operators with Parentheses: -(+2) * (+3) - (-4) / (-5)**
    - Result: **-6.8**
    - Explanation: This expression combines unary operators with parentheses and arithmetic operations.

Invalid Expressions

In the following examples, the issues can include unmatched parentheses, division by zero, incorrect operator usage, missing operands, or the use of invalid characters as operators, all of which lead to the expressions being invalid.

1. **Unmatched Parentheses: 2 * (4 + 3 - 1**
    - Explanation: This expression has unmatched opening and closing parentheses, making it invalid.

2. **Operators Without Operands: * 5 + 2**
    - Explanation: The ***** operator lacks operands on the left, making the expression invalid.
3. **Incorrect Operator Usage: 4 / 0**
    - Explanation: Division by zero is undefined in mathematics, so this expression is invalid.
4. **Missing Operator: 5 (2 + 3)**
    - Explanation: The expression lacks an operator between **5** and **(2 + 3)**, making it invalid.
5. **Invalid Characters: 7 & 3**
    - Explanation: The **&** character is not a valid arithmetic operator, so this expression is invalid in the context of arithmetic operations.
6. **Mismatched Parentheses: (((3 + 4) - 2) + (1)**
    - Explanation: The parentheses are not properly matched, with one closing parenthesis missing, making the expression invalid.
7. **Invalid Operator Usage: ((5 + 2) / (3 * 0))**
    - Explanation: This expression attempts to divide by zero, which is mathematically undefined, rendering the expression invalid.
8. **Invalid Operator Sequence: ((2 -) 1 + 3)**
    - Explanation: The expression contains an operator **-** without a valid operand on its left, making it invalid.
9. **Missing Operand: ((4 * 2) + (-))**
    - Explanation: There is a missing operand after the **-** operator, making the expression invalid.
10. **Invalid Characters: ((7 * 3) @ 2)**
    - Explanation: The **@** character is not a valid arithmetic operator in this context, causing the expression to be invalid.