Muhammad Chaudhary

1. Introduction

This report describes the design and implementation of an Infix Expression Evaluator. In order to return a numerical value, the code parses and evaluates a string that represents an infix mathematical expression. Along with key features like operator precedence and the use of parentheses to regulate the order of operations, the evaluator supports a wide range of arithmetic, logical, and comparison operators. The implementation uses a stack-based approach. This report will include a UML class diagram that depicts the system's structure, an analysis of the algorithmic efficiency of key functions, and any references used.

2. UML Class Diagram

```
@startuml
class Evaluator {
  - precedence: map<string, int>
  - isBinary: map<string, bool>
  + Evaluator()
  + eval(expression: string): int
  - tokenize(expression: string): string
  - isOperator(c: char): bool
  - performOperation(op: string, b: int, a: int = 0): int
  - validateExpression(expression: string): void
}
@enduml
```

Explanation of the Diagram:

Class Name: **Evaluator**

Attributes:

- precedence: map<string, int>: A private map containing the precedence level for each supported operator (represented as a string). Higher integer values indicate greater precedence.

- isBinary: map<string, bool>: A private map that determines whether a given operator (string) is binary (needs two operands) or unary.

Methods:

+ Evaluator(): The public constructor for the Evaluator class. It creates precedence and isBinary maps containing the supported operators and their properties.

+ eval(expression: string): int: A public method that accepts an infix expression string as input and returns an integer representation of its evaluation. This is the main entry point for using the evaluator.

- tokenize(expression: string): string: A private method for handling the raw input expression string. It removes whitespace and may manage the grouping of multi-character operators into single tokens.

- isOperator(c: char): bool: A private constant method that determines whether a given character is part of a recognized operator.

- **performOperation(op: string, b: int, a: int = 0): int:** A private method that accepts an operator and its operands (up to two) and performs the necessary calculation. It supports binary and unary operations.

- **validateExpression(expression: string): void:** A private method that checks the input expression string for common syntax errors before starting the evaluation process.

Relationships:

The diagram shows no explicit relationships to other classes, implying that the Evaluator class is self-contained in terms of functionality for this basic implementation. In a more complex system, it may interact with other classes to perform tasks such as advanced error reporting or handling various data types.

3. Efficiency of Algorithms

This section analyzes the Big-O notation for the significant functions within the Evaluator class.

Evaluator() (Constructor): The constructor creates two std::map objects with a fixed set of operators. The number of operators remains constant. Therefore, the time complexity of the constructor is O(1) (constant time).

tokenize(const std::string& expression): This function iterates through the input expression string once to remove any whitespace. If the expression's length is n, the function does a fixed amount of work for each character. Therefore, the time complexity is O(n) (linear time).

isOperator(char c) const: This function makes a fixed number of comparisons to determine whether a character is an operator. So the time complexity is O(1) (constant time).

performOperation(const std::string& op, int b, int a = 0): This function performs a series of constant-time comparisons and arithmetic operations based on the input operator. The time complexity is O(1).

validateExpression(const std::string& expression): This function loops through the input expression string once to perform validation checks (parentheses matching, consecutive operators/operands, and so on). Each character performs a consistent amount of work. Thus, the time complexity is O(n) (linear time).

eval(const std::string& expression): This is the main evaluation function. It involves iterating over the tokenized expression. In the worst-case scenario, where the expression is well-formed, each number and operator will be processed and pushed or popped from the stacks only once. The length of the expression determines how many operations are performed. As a result, the eval function has a time complexity of O(n) (linear), where n is the length of the tokenized expression.

4. Possibility of Doing Better:

For the given requirements of evaluating infix expressions with operator precedence and parentheses, the eval function's time complexity of O(n) is generally considered ideal. Any algorithm would have to go through each character in the input expression at least once to determine its meaning and role in the expression.

While the constant factors may differ depending on the specific implementation details (for example, the overhead of stack operations or map lookups), the fundamental linear relationship with input size is difficult to improve for this type of problem.

5. References

Used a lot of google searches and slides from the class.