PROJECT 1A: INFIX EXPRESSION PARSER REPORT

Course: Data Structures and Algorithms

Project: Infix Expression Evaluator Using Stacks

Date: 7/25/2025

INTRODUCTION

This project implements an infix expression parser and evaluator. The parser supports a comprehensive set of operators including arithmetic, comparison, logical, and unary operators with proper precedence handling. The implementation follows object-oriented design principles and provide robust error handling with detailed error messages.

The primary goal was to create an efficient parser that can evaluate mathematical expressions in infix notation while maintaining O(n) time complexity and providing meaningful error feedback to users.

ASSUMPTIONS

OPERATOR BEHAVIOR:

- Power operator (^) is implemented as repeated multiplication and assumes non-negative integer exponents only

INPUT CONSTRAINTS:

- Operands are assumed to be non-negative integers only (no support for negative number literals like -5, though unary minus can be applied)

- Expression length is assumed to fit within reasonable memory constraints
  (no specific limit enforced)
- Whitespace handling allows any amount of whitespace between tokens

ERROR HANDLING

- Division by zero is detected and reported with character position information
- First error reporting - only the first syntax error encountered is reported,
  then evaluation stops
- Character position in error messages refers to the position in the original
  string including whitespace

UML CLASS DIAGRAM

Evaluator
- expression: std::string - pos: size_t
+ eval(expr: const std::string&): int - getPrecedence(op: const std::string&): int - isOperator(op: const std::string&): bool - isUnaryOperator(op: const std::string&): bool - performOperation(a: int, b: int, op: const std::string&): int - performUnaryOperation(a: int, op: const std::string&): int - skipWhitespace(): void - parseNumber(): int - parseOperator(): std::string - validateExpression(): void

EFFICIENCY ANALYSIS

TIME COMPLEXITY ANALYSIS

PRIMARY FUNCTIONS

eval() - O(n)

- Justification: Single pass through the expression string where n is the length of the input
- Breakdown:
  * Validation: O(n) - single scan through expression
  * Main parsing loop: O(n) - each character processed once
  * Operator processing: O(1) amortized per operator due to stack operations
  * Overall: O(n) where n is the expression length

validateExpression() - O(n)

- Justification: Single pass-through expression checking syntax rules
- Operations: Character classification and state tracking are O(1) per character

parseNumber() - O(d)

- Justification: Where d is the number of digits in a number
- Note: $d \leq n$, so this contributes to the overall O(n) complexity

parseOperator() - O(1)

- Justification: Checks at most 2 characters for multi-character operators
- Constant time regardless of expression length

HELPER FUNCTIONS:

getPrecedence() - O(1)

- Justification: Simple string comparison with fixed number of operators

performOperation() - O(e) for exponentiation, O(1) for others

- Justification: Power operation requires O(e) where e is the exponent value
- Note: All other operations are constant time arithmetic

performUnaryOperation() - O(1)

- Justification: All unary operations are simple arithmetic

SPACE COMPLEXITY ANALYSIS:

Overall Space Complexity: O(n)

- Operand Stack: O(n/2) in worst case (alternating numbers and operators)
- Operator Stack: O(n/2) in worst case (deeply nested parentheses)
- Input Storage: O(n) for the expression string
- Total: O(n) where n is the expression length

Conclusion

The chosen stack-based approach provides optimal balance of efficiency, simplicity, and maintainability.