Lab 2 : Evaluation of Postfix Expressions

Course: CS20009.04 Data Structure
Name: Yijia Chen
Student Number: 24300240127
Date: October 14, 2025

Introduction

This lab aims to:

- Transform infix expressions to postfix expressions, taking advantage of the LIFO property of a stack.
- Evaluate postfix expressions.

Implementation

Auxiliary Functions

At the beginning of `RPN.cpp`, we implement some auxiliary functions:

- `isOpening`: returns `true` if the character `c` is an opening parenthesis, brace or bracket, and `false` otherwise.
- `isClosing`: returns `true` if the character `c` is a closing parenthesis, brace or bracket, and `false` otherwise.
- `match`: returns `true` if the parenthesis, brace or bracket `a` and `b` match, and `false` otherwise.
- `precedence`: returns the precedence of the operator `op`. In this case, we simply distinguish between `+`, `-`, `*`, `/` and `%`, where the precedence of `*`, `/` and `%` is higher than that of `+` and `-`.

```
bool isOpening(char c) {
    return c == '(' || c == '[' || c == '{';
}

bool isClosing(char c) {
    return c == ')' || c == ']' || c == '}';
}

bool match(char a, char b) {
    return (a == '(' && b == ')')
        || (a == '[' && b == ']')
        || (a == '{' && b == '}');
}

int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/' || op == '%') return 2;
    return 0;
}
```

Infix to Postfix Conversion

The function `infix2postfix` takes a string `infix` as input and returns a string `postfix`.

The stack `s` stores the operators, `+`, `-`, `*`, `/` and `%`, including parentheses, braces and brackets. Its LIFO property ensures that the operators are processed in the correct order.

For each character `c` in the input `infix`, we do the following:

If `c` is a space (e.g. ` `, `\n`, `\t`), we ignore it.

If `c` is part of an operand (i.e. a digit in range `0~9` or a decimal point `.`), we accept those characters as part of the operand by appending them to the string `operandBuffer` repeatedly. Once the current character examined is not part of an operand, which indicates that an entire operand (e.g. `123.45`) has been accepted, we directly push `operandBuffer` to `postfix` and reset `operandBuffer` to an empty string.

Next, we deal with parentheses, braces or brackets.

- If `c` is an opening parenthesis, brace or bracket, we push it to the stack `s`.
- When `c` is a closing parenthesis, we keep popping the stack `s` until we find a matching opening one. The operators between these two parentheses are added to the end of `postfix`.
- If no matching opening parenthesis is found, an error message is printed and the function returns an empty string.

- Note that parentheses are merely used to group operators and adjust the precedence of evaluation, and they do not appear in postfix expressions.

When an operator is encountered, we first check if the stack `s` is empty. If it is, we push the operator to the stack. Otherwise, we compare the precedence of the current operator with the precedence of the top operator in `s`.

- If the precedence of the current operator `c` is lower or equal to that of the top operator, we pop the top operator from the stack and add it to the end of `postfix`. We then push `c` to the stack.
- On the other hand, if the precedence of `c` is higher than that of the top operator, we push `c` to the stack.
- This ensures that the operators are processed in the correct order.

After we have traversed through all characters in `infix`, we need to check whether there are any operands or operators left in `operandBuffer` and `s`, respectively. If so, we add them to the end of `postfix`.

Finally, the desired postfix expression `postfix` of the given infix expression is returned.

Note that we add whitespace characters between operands and operators in `postfix`, for the convenience of the function `evaluatePostfix` that uses `stringstream` to read or convert the operands and operators efficiently.

```
string infix2postfix(const string& infix) {
    stack<char> s;
    string postfix = {};
    string operandBuffer = {};

    for (char c : infix) {
        if (isspace(c)) continue;

        if (isdigit(c) || c == '.') {
            operandBuffer.push_back(c);
        } else {
            if (!operandBuffer.empty()) {
                postfix += operandBuffer + ' ';
                operandBuffer.clear();
            }

            if (isOpening(c)) s.push(c);
            else if (isClosing(c)) {
                while (!s.empty() && !isOpening(s.top())) {
                    postfix += s.top(); postfix += ' ';
                    s.pop();
                }
                if (!s.empty() && match(s.top(), c)) s.pop();
                else {
                    cerr << "Error: Unbalanced brackets.\n";
                    return "";
                }
            }

            else { // operator
                while (!s.empty() && precedence(c) <= precedence(s.top())) {
                    if (isOpening(s.top())) break;
                    postfix += s.top(); postfix += ' ';
                    s.pop();
                }
                s.push(c);
            }
        }
    }
    if (!operandBuffer.empty())
        postfix += operandBuffer + ' ';
    while (!s.empty()) {
        if (isOpening(s.top())) {
            cerr << "Error: Unbalanced brackets.\n";
            return "";
        }
```

```
            postfix += s.top(); postfix += ' ';
            s.pop();
        }
        return postfix;
    }
```

Postfix Expression Evaluation

In `evaluatePostfix`, the stack `s` stores the operands, possibly results of previous calculation that are also operands in the next step.

Similar to `infix2postfix`, we need to pay special attention to operands of type `double`. We use `stringstream` to automatically convert the string `token` to a `double` value.

Then, we use `switch` statement to deal with different operators. Each `token` read by stream extraction operator `>>` is a whitespace separated sequence of characters, so an operator is the first character `token[0]`. We perform the corresponding operation on the top two operands in `s` and push the result back to `s`, waiting for the next calculation step.

Two points to note:

- The division operation requires that the divisor is not zero. If the divisor is zero, an error message is printed and the function returns zero.
- The modulo operation is only valid when both operands are integers. If either of the operands are not integers, an error message is printed and the function returns zero. It is special that, for the compatibility with other operators `+`, `-`, `*` and `/`, the current two operands `a` and `b` are defined in type `double`. So before the modulo operation, we examine whether `a` and `b` are actually integers by comparing their fractional part with a small number `1e-6`.

Eventually, the result of the last calculation is stored in the top (and as the only element) of `s`, which is the desired value of the postfix expression.

```
double evaluatePostfix(const string& postfix) {
    stack<double> s;
    stringstream ss(postfix);
    string token;

    while (ss >> token) {
        if (isdigit(token[0]) || token[0] == '.')
            s.push(stod(token));

        else { // operator
            if (s.size() < 2) {
                cerr << "Error: Invalid expression.\n";
                return 0;
            }
            double b = s.top(); s.pop();
            double a = s.top(); s.pop();
            switch (token[0]) {
                case '+': s.push(a + b); break;
                case '-': s.push(a - b); break;
                case '*': s.push(a * b); break;
                case '/':
                    if (b == 0) {
                        cerr << "Error: Division by zero.\n";
                        return 0;
                    }
                    s.push(a / b);
                    break;
                case '%':
                    if (abs(a - round(a)) < 1e-6 && abs(b - round(b)) < 1e-6) {
                        s.push(static_cast<int>(a) % static_cast<int>(b));
                    } else {
                        cerr << "Error: Modulus operation is only valid for integers.";
                        return 0;
                    }
                    break;
                default:
                    cerr << "Error: Invalid operator.\n";
                    return 0;
            }
        }
    }
    return s.empty() ? 0 : s.top();
}
```

Header File

All the function prototypes are defined in `RPN.h` .

```
#pragma once
#include <iostream>
#include <sstream>
#include <stack>
#include <string>
#include <cctype>
#include <cmath>
#include <map>

using std::stack; using std::string; using std::cerr;
using std::stringstream; using std::stod;
using std::isspace; using std::isdigit; using std::round;

bool isOpening(char c);
bool isClosing(char c);
bool match(char a, char b);
int precedence(char op);

string infix2postfix(const string& infix);
double evaluatePostfix(const string& postfix);
```

Test Code

```cpp
#include <iostream>
#include <cassert>
#include <cmath>
#include "RPN.h"  // Assuming you saved the main logic in a separate header or cpp file

using namespace std;

// Function to run a single test case and print results
void testInfixToPostfixAndEvaluate(const string& infix) {
    cout << "Infix Expression: " << infix << endl;

    // Convert infix to postfix
    string postfix = infix2postfix(infix);
    cout << "Postfix Expression: " << postfix << endl;

    // Evaluate the postfix expression
    double result = evaluatePostfix(postfix);
    cout << "Evaluated Result: " << result << endl;

    cout << "-----------------------------------" << endl;
}

int main() {
    // Test cases
    try {
        // Test 1: Nested parentheses with multiple operations
        testInfixToPostfixAndEvaluate("3 + (2 * (1 + 2))");

        // Test 2: Multiple operations with different precedence
        testInfixToPostfixAndEvaluate("5 + 3 * 2 - 8 / 4");

        // Test 3: Complex expression with decimals and multiple operators
        testInfixToPostfixAndEvaluate("123.45 * 2.5 + 100 / 4 - 45.67");

        // Test 4: Expression with modulus operator
        testInfixToPostfixAndEvaluate("10 % 3 + 4 * 2");

        // Test 5: Large numbers and division
        testInfixToPostfixAndEvaluate("1000000000 / 2500 + 999999");

        // Test 6: Nested operations and mixing of multiplication, division, and addition
        testInfixToPostfixAndEvaluate("(3 + 5 * (10 / 2)) - 4");

        // Test 7: Complex algebraic expression with parentheses
        testInfixToPostfixAndEvaluate("(2 + 3) * (5 + 8) / (4 - 1)");
```

```
        // Test 8: Division with a result that has many decimal places
        testInfixToPostfixAndEvaluate("50 / 3");

        // Test 9: Combination of addition, subtraction, multiplication, and division
        testInfixToPostfixAndEvaluate("10 + 2 * 5 - 3 / (7 + 1)");

        // Test 10: Division by zero (Edge case)
        cout << "Testing division by zero..." << endl;
        string postfix = infix2postfix("10 / 0");
        if (postfix.empty()) {
            cout << "Handled division by zero correctly!" << endl;
        } else {
            double result = evaluatePostfix(postfix);
            cout << "Evaluated result: " << result << endl;
        }

    } catch (const exception& e) {
        cout << "Test failed with error: " << e.what() << endl;
    }

    return 0;
}
```

Acknowledgement

ChatGPT-5 provided valuable feedback and suggestions on the test code implementation, wording and sentense structure.

Appendix: Lab specification

1. Write code for Infix to postfix conversion by using stack (The code should be able to treat with parenthesis, braces and at least the following operators: +, −, ×, /, mod).
2. Write code for Postfix expression evaluation by using stack.
3. All document for the above codes.