实验报告

数据结构 *Data Structures*

报告标题: **中缀表达式求值**
学号: 19240212
姓名: 华博文
日期: 2025 年 10 月 14 日

一 实验环境

1 操作系统

Ubuntu 24.04.3 LTS x86_64，Linux 6.14.0-28-generic，AMD Ryzen 7 7700 (16) @ 5.3GHz。

2 编程工具

NeoVim v0.11.1，gcc 13.3.0。

3 其他工具

LaTeX (TeXstudio)。

二 实验内容及其完成情况

1 实验目的

1. **理解中缀表达式结构与求值规则：**本实验旨在深入理解中缀表达式的语法结构及其求值规则。中缀表达式是最常见的数学表达式形式，运算符位于两个操作数之间。通过本实验，将掌握运算符优先级、结合性以及括号的处理方法，为后续复杂表达式的计算打下基础。
2. **掌握栈在表达式求值中的应用：**中缀表达式求值通常采用双栈法：一个操作数栈、一个运算符栈。实验过程中，将学习如何利用栈实现表达式的分步计算，巩固栈的基本操作（入栈、出栈、取栈顶等），并体会栈在算法设计中的重要作用。
3. **提升复杂逻辑的实现能力：**实验涉及运算符优先级比较、括号匹配、多位数及小数解析等多种边界情况。通过实现完整的中缀表达式求值算法，将提升处理复杂逻辑和异常情况的能力，增强代码健壮性和容错性。

2 实验内容

实现一个程序，能够对用户输入的中缀表达式进行求值。采用双栈法（操作数栈与运算符栈），支持基本四则运算和括号，保证程序健壮性，能正确处理非法输入。功能要求如下：

1. 支持运算符：+、-、*、/；
2. 支持括号：(、)；

3. 支持多位数及小数（如 42.57）；
4. 对非法表达式（括号不匹配、运算符错误、除零等）能给出明确提示；
5. 采用 C++ 语言实现。

3 实验过程

3.1 表达式分词 tokenize_expression

表达式分词是整个求值流程的第一步。tokenize_expression 函数负责将用户输入的字符串拆分为有类型的 Token，包括数字、运算符和括号。分词时，程序会逐字符扫描输入，遇到数字或小数点时，会连续读取形成完整的多位数或小数，并通过 std::stod 转换为 **double** 类型。例如，输入 42.57 + 3 会被分解为 Token(NUMBER, 42.57)、Token(OPERATOR, '+') 和 Token(NUMBER, 3)。

分词过程中还会检测数字格式错误（如多重小数点），并对非法字符进行处理。如果发现数字中有多重小数点，或出现非法字符，程序会抛出异常并提示错误。例如：

```
if (expr[i] == '.') {
    if (dot) throw std::runtime_error(ERR_MULTIPLE_DOT);
    dot = true;
}
...
else {
    throw std::runtime_error(ERR_ILLEGAL_CHAR + expr[i]);
}
```

此外，括号和运算符也会被正确识别为独立的 Token，便于后续处理：

```
if (std::string("+-*/").find(expr[i]) != std::string::npos) {
    token_list.push_back(Token(OPERATOR, expr[i]));
    ++i;
} else if (expr[i] == '('  expr[i] == ')') {
    token_list.push_back(Token(PARENTHESIS, expr[i]));
    ++i;
}
```

3.2 运算符优先级与运算 get_precedence、apply_operator

运算符优先级决定了计算的顺序。get_precedence 函数用于返回运算符的优先级，乘除高于加减：

```
int get_precedence(char op) {
    if (op == '+'  op == '-') return 1;
    if (op == '*'  op == '/') return 2;
```

```
    return 0;
}
```

apply_operator 函数负责实际的四则运算操作。它根据传入的运算符类型，对两个操作数进行加、减、乘、除运算。特别地，除法运算时会检测除数是否为零，若为零则抛出 ERR_DIV_ZERO 异常，保证程序健壮性：

```
double apply_operator(double a, double b, char op) {
    switch (op) {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            if (b == 0) throw std::runtime_error(ERR_DIV_ZERO);
            return a / b;
        default:
            throw std::runtime_error(ERR_UNKNOWN_OP);
    }
}
```

3.3 表达式求值 evaluate_expression

evaluate_expression 是核心计算函数，实现了双栈法。遍历分词得到的 Token 列表时，遇到数字直接压入操作数栈 value_stack，遇到运算符则根据优先级与栈顶运算符比较，若优先级较低或相等，则弹出栈顶运算符并进行计算，将结果压入操作数栈。例如，处理 3 + 4 * 2 时，乘法优先级高于加法，先计算 4 * 2，再与 3 相加。

括号的处理则更为复杂，遇到左括号直接压入运算符栈，遇到右括号则不断弹出运算符并计算，直到遇到匹配的左括号为止。如果括号不匹配，程序会抛出 ERR_PAREN_MISMATCH 错误。如下代码片段展示了括号处理逻辑：

```
if (token.type == PARENTHESIS) {
    if (token.op == '(') {
        op_stack.push('(');
    } else if (token.op == ')') {
        bool found_left = false;
        while (!op_stack.empty()) {
            if (op_stack.top() == '(') {
                op_stack.pop();
                found_left = true;
                break;
```

```
            } else {
                if (value_stack.size() < 2) throw
                 ↪  std::runtime_error(ERR_MISSING_OPERAND);
                double b = value_stack.top();
                value_stack.pop();
                double a = value_stack.top();
                value_stack.pop();
                char op = op_stack.top();
                op_stack.pop();
                value_stack.push(apply_operator(a, b, op));
            }
        }
        if (!found_left) throw
         ↪  std::runtime_error(ERR_PAREN_MISMATCH);
    }
}
```

表达式处理完毕后，程序会依次弹出剩余运算符并计算，最终操作数栈应只剩一个结果。若栈中多余操作数或操作数不足，分别抛出 ERR_EXTRA_OPERAND 或 ERR_MISSING_OPERAND 错误，确保表达式结构正确：

```
while (!op_stack.empty()) {
    if (op_stack.top() == '('  op_stack.top() == ')') throw
     ↪  std::runtime_error(ERR_PAREN_MISMATCH);
    if (value_stack.size() < 2) throw
     ↪  std::runtime_error(ERR_MISSING_OPERAND);
    double b = value_stack.top();
    value_stack.pop();
    double a = value_stack.top();
    value_stack.pop();
    char op = op_stack.top();
    op_stack.pop();
    value_stack.push(apply_operator(a, b, op));
}
if (value_stack.size() != 1) throw
 ↪  std::runtime_error(ERR_EXTRA_OPERAND);
```

3.4 主程序与异常处理

主函数负责与用户交互，循环读取输入表达式，调用分词和求值函数，并输出结果或错误提示。所有异常均在主函数中捕获，并统一输出为 [错误] ... 格式，提升用户体验。如下代码片段展示了主循环的异常处理：

```
try {
    auto token_list = tokenize_expression(input_line);
    double result = evaluate_expression(token_list);
    std::cout << " = " << result << std::endl;
} catch (const std::exception &e) {
    std::cout << "[错误] " << e.what() << std::endl;
}
```

用户可多次输入表达式，按 Ctrl+C 退出程序。整个流程实现了对中缀表达式的高效、健壮求值，支持多位数、小数、括号嵌套及异常处理，充分体现了栈结构在表达式计算中的应用价值。

4 实验结果与分析

本实验实现的中缀表达式求值程序在功能和健壮性方面表现良好。经过多组测试，程序能够正确处理各种合法表达式，包括带括号的嵌套运算、多位数和小数。例如，输入 `3 + 4 * (2 - 1)`，程序会先分词为数字和运算符，然后根据优先级和括号规则，先计算括号内的 `2 - 1`，再进行乘法和加法，最终输出 `= 7`。对于小数运算如 `42.57 / 2`，分词和计算均能准确完成，输出 `= 21.285`。

在异常处理方面，程序对非法输入有明确提示。例如，输入 `3 + (4 * 2` 时，由于括号不匹配，程序会抛出异常并输出 `[错误] 括号不匹配`。输入 `5 / 0` 时，检测到除零操作，输出 `[错误] 除零错误`。对于包含非法字符的表达式如 `2 + a`，分词阶段会捕获并提示 `[错误] 非法字符: a`。这些错误处理均通过如下代码实现：

```
catch (const std::exception &e) {
    std::cout << "[错误] " << e.what() << std::endl;
}
```

此外，程序对表达式结构的异常也有检测。例如，输入 `1 +` 或 `1 2 +`，会分别提示“表达式缺少操作数”或“表达式多余操作数”，保证结果的唯一性和正确性。整体来看，程序不仅能正确计算结果，还能对各种边界情况和错误输入做出合理响应，体现了良好的健壮性和用户体验。

三 实验总结

通过本次实验，对中缀表达式的结构和求值规则有了深入理解，尤其是运算符优先级、结合性以及括号的处理。实验过程中，双栈法的应用使表达式求值过程变得清晰和高效，进一步巩固了对栈这一数据结构的掌握。分词、优先级判断、括号匹配和异常处理等环节的实现，锻炼了复杂逻辑的编程能力。

代码实现过程中，采用了现代 C++ 风格，结构清晰，命名规范，错误信息集中管理，便于维护和扩展。异常处理机制保证了程序的健壮性，即使面对各种非法输入也能给出明确提示，避免程序崩溃。主循环设计使用户可以多次输入表达式，交互体验良好。

总的来说，本实验不仅完成了中缀表达式求值的功能目标，还提升了对数据结构和算法设计的理解与应用能力。通过实际编程和测试，体会到细致的边界处理和错误检测对于高质量程序的重要性，为后续学习更复杂的表达式处理和数据结构应用打下了坚实基础。