摘要

本次编程作业主要利用 `expression_evaluator` 类定义了一个简单运算的计算器，实现了加减乘除、负数、科学计数法和幂运算和对浮点数的中缀表达式求值。

1 设计思路及实现原理

1. 准备工作。根据用户的输入习惯，如果表达式中有空格，会先删除空格。其次用户输入负数的时候，必须以括号的形式输入，比如 $(-1)+2$，而不是 $-1+2$，这样一旦检测到表达式中有 $(-$ 就会在负号之前插入 0，比如 $(-1)+2=(0-1)+2$。这一部分通过 `modify()` 函数实现。

2. 定义运算符和优先级。以及各种运算符背后的具体运算，其中^ 代表指数运算，比如 $23=2^3=8$；e 代表科学计数法，比如 $1.2e1=1.2\times10^1=12$。具体的成员函数如下：

```
            bool isoperator(const char &c);
                if(c=='+' || c=='-' || c=='*' || c=='/' || c=='e' || c=='^'){
                    return true;
                }
            int prior(const char &c);
            double calculate(const double &x, const double &y, const char &c) {
                if (c == 'e') {
                    return x * pow(10, y);
                } else if (c == '^') {
                    return pow(x, y);
                }
            }

```

Listing 1: 运算符实现

3. 检查表达式是否合法。在运算之前先检查括号是否匹配和表达式是否合法，其中用栈来检测括号是否匹配，遇到左括号入栈，遇到右括号左括号出栈，最后检查栈是否为空。定义了以下两个公有成员函数

```
            bool isLegal(){
            int l=expressions.size();
            for(int i=0; i<l-1; ++i){
                bool a=!isdigit(expressions[i]) && expressions[i]!='(' && expressions[i]!=')';
                bool b=!isdigit(expressions[i+1])&& expressions[i+1]!='(' && expressions[i+1]!=')';
                bool c= expressions[i]=='.';
                bool d=expressions[i+1]=='.';
                if((i==0 && a ) || (i==l-2 && b ) ||
                (a && b) || (a&& expressions[i+1]==')')||(expressions[i]=='('&&expressions[i+1]==')')||
                (expressions[i]=='('&& b) ||(c && expressions[i+1]==')') ||(c && expressions[i+1]=='(') ||
                (expressions[i]=='(' && d) ||(expressions[i]==')' && d)||
                (expressions[i]=='(' && expressions[i+1]==')')){
                    return false;
                }
            }

```

Listing 2: 运算符实现

这会将括号不匹配、两个运算符直接相连（比如 $2+-1$）、运算符出现在开头结尾（比如 $-2+3$，$2*5/$ ，$3+(+1)$ ，$2-(2*)$ ）、小数点和运算符或者括号直接相连（比如 $2.(3*4)$ ，$3.+2$ ，$(2+3).5$ ）以及两个小数点（比如 $1..2+1$ ，$1.2.1+2$ ）、左右括号直接相连都看作不合法的表达式。但是 0021 这种以 0 开头的数字，是合法的，会舍弃掉多余的 0.

4. 计算。这会通过公有成员函数 `evaluate()` 实现,如果表达式不合法,会输出 `brackets are matched or illegal expression` 或者 `invalid expression` 来提示一个数字出现了多个小数点。一边生成后缀表达式一边计算。具体的，解析连续的数字或小数点，转换为双精度数压入数字栈；根据优先级规则处理操作符，必要时计算栈顶操作符的结果；通过括号分隔优先级不同的子表达式，计算并丢弃括号；遍历。

2 测试函数

1. 设计了如下几个函数分别测试所有公有成员函数，包括表达式是否合法、括号匹配、处理浮点数、处理指数和科学计数法、处理负数和比较复杂的表达式计算。

```
            testModify();
            testIsLegal();
            testIsMatch();
            testEvaluate();
            testExponentiation();
            testNegativeNumbers();
            testFloatingPoint();
            testComprehensive();
            testDivideByZero();
            testEmptyExpression();

```

3 结果分析

最终测试的结果如下：

```
    g++ -o main main.cpp
    Running main:
    ./main
    Test modify : ((-3)+4)*5 5
    ----------------------------------------
    Test isLegal for valid expression 3 + 4 Pass
    Test isLegal for invalid expression +3 + 4 Fail
    Test isLegal for invalid expression 3 + + 4 Fail
    Test isLegal for invalid expression 3 + (4 * 2 Pass
    Test isLegal for invalid expression 3 + 4) * 2 Pass
    Test isLegal for invalid expression 3 + 4 * 2 +: Fail
    Test isLegal for invalid expression /3 + 4 * 2 + Fail
    Test isLegal for invalid expression 1..2+2 Fail
    Test isLegal for invalid expression (2+3).5 Fail
    Test isLegal for invalid expression 1.(2-1) Fail
    Test isLegal for invalid expression 3+()+2 Fail
    ----------------------------------------
    Test isMatch for matching parentheses 3 + (4 * 2): Pass
    Test isMatch for unbalanced parentheses 3 + (4 * 2: Fail
    Test isMatch for matching parentheses 3 + 4 * (2 + 5): Pass
    Test isMatch for unbalanced parentheses 3 + 4 * 2): Fail
    ----------------------------------------
    Test evaluate (3 + 4) = 7
    Test evaluate (5 * 6) = 30
    Test evaluate (10 / 2) = 5
    Test evaluate (2 * (3 + 4)) = 14
    Test evaluate (2 + 3 * 5) = 17
    Test evaluate (3 + 4) * 5 = 35
    Test evaluate invalid expression 10 ^ 3 = 1000
    Test evaluate 3 + (4 * 2) = 11
```

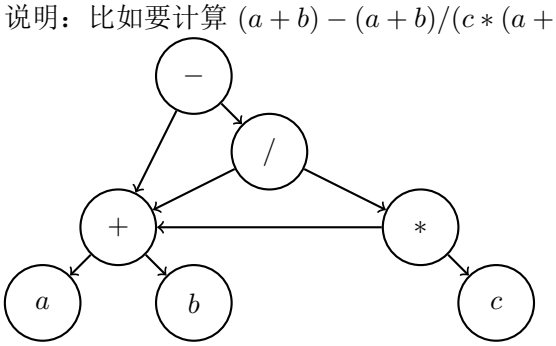
```
Test evaluate 2.1e1+3^2-(-2)*3 = 36
----------------------------------------
Test exponentiation 2 e 3 = 2000
Test exponentiation 5 e 2 = 500
Test exponentiation 1 e 4 = 10000
Test exponentiation 3 e 0 = 3
----------------------------------------
Test negative numbers ((-3) + 4) * 5 =5
Test negative numbers (-3) + 4=1
Test negative numbers 3 + (-4) =: -1
----------------------------------------
Test floating point numbers 3.5 + 4.2 =7.7
Test floating point numbers 5.5 * 2 = 11
Test floating point numbers 10.4 / 2 = 5.2
----------------------------------------
Test Comprehensive expression: 3 + 5 * 2 - 1= 12
Test Comprehensive expression: (3+5)*(2-1)= 8
Test Comprehensive expression: 2+3*(6/3)e1= 62
Test Comprehensive expression: 4e2^0.5= 20
Test Comprehensive expression: 0.08e(-1 )^(1/3) = 0.2
Test Comprehensive expression: 0.21.1+2 = invalid expression
-1
Test Comprehensive expression: 0022+0.11e2-(-3)^2 = 24
----------------------------------------
Test divide by zero 10 / 0: can not be devided by zero
-1
----------------------------------------
Test empty expression: no expression!
-1
```

4 设计的不足以及可能的改进方案

1. 在我的设计中 $-1+2$ 的这种情况将被判断为不合法的输入，如果要进行负数运算，必须要输入 $(-1)+2$，然而在实际中第一种形式的表达式也往往不会引起误解并且被广泛应用。

2. 在我的设计中，尽管实现了对科学计数法的运算，但本质上是将 e 看作了一种特殊的运算符，比如 $2e3 = 2 \times 10^3 = 2000$。但是在实际应用中，科学计数法往往要求形如 a^b 的形式，这里要求 $1 \leq a < 2,\ b \in \mathbb{N}$，但是在我写的运算器中，形如 $0.3e2.5$ 这样的运算也不会报错，这与实际的使用习惯不符。

3. 最后是算法的效率问题。如果一个表达式是 $(1+2)*(1+2)+(1+2)$，这将计算三次 $1+2$，如果表达式非常长，这种重复计算相同的表达式显然是低效率的，为此一个可能的我认为的改进思路是使用有向图。下面举例说明：比如要计算 $(a+b)-(a+b)/(c*(a+b))$，可以建立如下的有向图



然后利用逆拓扑排序，依次运算，但是这种方式可能存在的问题是 $/$ $-$ 运算都是有左右之分的，但是有向无权图的节点连接的其他节点地位是相等的，因此这种方式更适合运算只有 $+$ $*$ 的情况。