作业 2 表达式求值问题

解雲暄 3190105871

索引

作业 2 表达式求值问题
索引
1 问题描述
2 工程概述
3 算法流程与数据结构概述
3.1 算法简述
3.2 词法分析
3.3 表达式求值
3.3.1 压栈
3.3.2 出栈和计算
3.4 类和数据结构概述
4 MFC 设计概述
5 测试与分析
5.1 运算测试
5.2 错误提示
5.2.1 词法错误提示
5.2.2 语法错误提示
5.2.3 除以 0 错误
5.2.4 括号不配对
5.3 分析

1 问题描述

利用 MFC 完成一个表达式求值软件。支持：

- 整数、浮点数的四则运算；
- 通过括号调整运算顺序。

本工程额外支持了：

- - 表示减法或负号（+ 不可以表示正号）；
- 三角函数 sin, cos 以及常数 π (用 PI 表示)；
  - sin, cos 和 pi 不区分大小写；
  - 支持 3sin PI 而不必是 3*sin PI 这样符合数学习惯的写法；
- 结果为整数的整数运算得到整数结果，含浮点数的运算或者结果为浮点数的运算得到浮点数结果；
  - 例如，`4/2` 的结果为整数 `2`，而 `5/2` 的结果为实数 `2.5`。
- 对表达式错误（包括 **词法错误、语法错误和其他具体错误**）的检测和提示；
- 按回车进行计算而不是默认的退出程序、输入文本框被更改时清空输出文本框等**友好的应用逻辑**。

本工程中，输入表达式的空格都是可选的；空格和制表符将被直接忽略。例如，`5 6+4` 将被识别为 `56+4` 并得到结果 `60`。

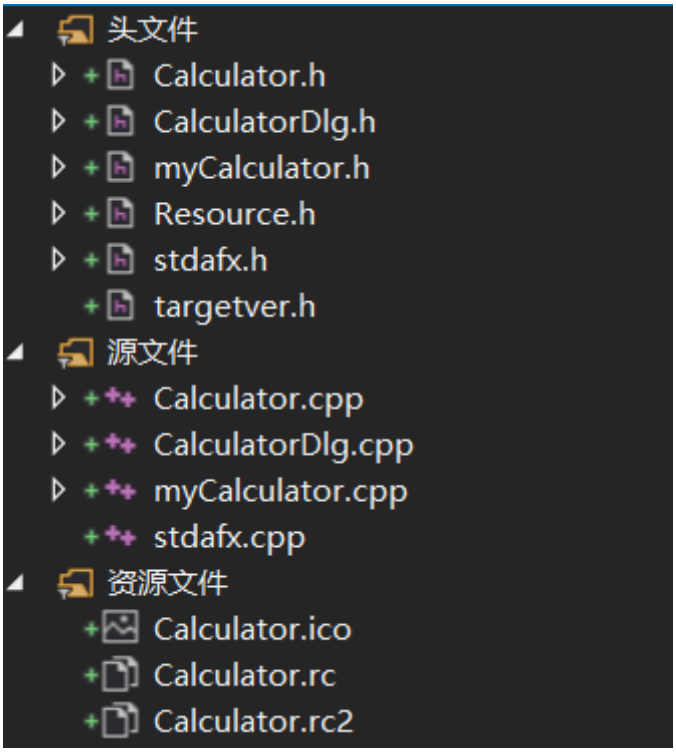
[!] 本工程不支持对超过 C++ 中超过 int 和 double 规模限制的数字的运算；同时也不提供相应的检查和报错。

2 工程概述

如下图所示是工程的主要文件。其中，`myCalculator.h` 和 `myCalculator.cpp` 主要实现了完成计算功能的类 `MyCalculator`。其他文件是 MFC 的一些基本文件；在其中的主要改动包括：

- `Calculator.cpp` 中：
  - 完成了“计算”、“退出”两个按钮的点击事件响应，设置了 **当输入文本框被更改时清空输出文本框** 的逻辑；
  - 实现了函数 `startCalc()`，完成读取输入、启动计算、填写输出的工作；

    - 设置了对 private 变量（两个文本框的内容）`calcInput` 和 `calcOutput` 的 getter 和 setter（setter 中同时写入图形界面）；
    - 重载了 `CCalculatorDlg::PreTranslateMessage` 函数，设置了 **回车进行计算而不是退出程序** 的逻辑。
- `Calulator.h` 中将文本框内容设为了 private，声明了 private 成员变量 `myCalculator`（`MyCalculator` 的实例）。
- `stdafx.h` 中添加了若干需要的头文件；`Calculator.rc` 中设计了 MFC 的界面。



3 算法流程与数据结构概述

3.1 算法简述

该问题的算法部分主要是表达式求值。我们首先引出 **后缀表达式**：

后缀表达式又称逆波兰式（Reverse Polish notation, RPN），定义如下：

1. 如果 E 是一个变量或常量，则 E 的后缀表达式是它本身；
2. 如果 E 是一个 E_1 op E_2 格式的中缀表达式，那么它的后缀表达式为 E_1' E_2' op。其中 E_1' E_2' 是 E_1 E_2 的后缀表达式，op 为任何二元操作符；
3. 形如 (E) 的表达式的后缀表达式为 E 的后缀式。

中缀表达式转为后缀表达式 的算法是，建立一个用于存放运算符的栈，扫描该中缀表达式：

- 如果遇到数字，直接将该数字输出到后缀表达式（以下部分用「输出」表示输出到后缀表达式）；
- 如果遇到左括号，入栈；
- 如果遇到右括号，不断输出栈顶元素，直至遇到左括号（左括号出栈，但不输出）；
- 如果遇到其他运算符，不断去除所有运算优先级大于等于当前运算符的运算符，输出。最后，新的符号入栈；
- 把栈中剩下的符号依次输出，表达式转换结束。

后缀表达式求值 的算法是，维护一个数字栈，每次遇到一个运算符，就取出两个栈顶元素，将运算结果重新压入栈中。最后，栈中唯一一个元素就是该后缀表达式的运算结果。

我们将这两个步骤同步进行进行求值。下面对具体的流程进行描述。

3.2 词法分析

首先我们将输入的表达式转换成单词流。所谓“单词”（token），就是对操作符和操作数的统称，我们用这样一个类进行描述：

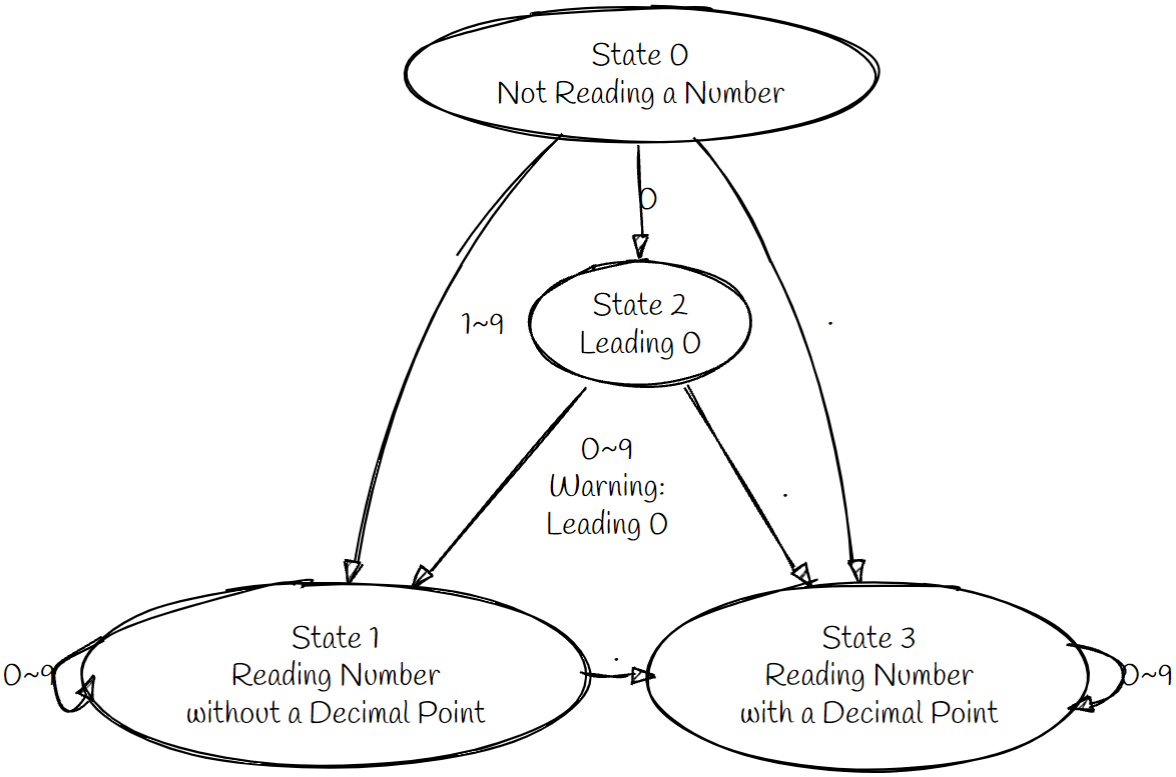
```
class Token {
public:
    enum Type ty;
    int ival;
    double rval;
};
```

每一个 token 都有一个类型，可能是加、减、乘、除、左括号、右括号、sin、cos、π、整数、浮点数。`enum Type` 的定义如下：

```
/* All possible types of tokens. We regard integer and real numbers
 * as 2 different types.
 */
enum Type { PLUS, MINUS, MUL, DIV, LP_, RP, SIN, COS, PI, INUM, RNUM,
NOP = -1 };
/*          0      1     2    3   4   5    6    7   8    9     10
-1*/
```

对于整数（INUM, integer number），我们用 `ival` 字段存储其值；对于浮点数（RNUM, real number），我们用 `rval` 字段存储其值。这两种 token 以及 π 将被压入数字栈 `numStack` 中，其他 token 被压入符号栈 `opStack` 中。这两个栈将在后文详细说明。

词法分析过程会忽略空格、制表符等空白字符；符号的识别是平凡的；值得一提的是对数字的词法识别。我们设计了如下所示的状态机帮助识别：



对应的部分代码如下。这部分代码结合上图很好理解，在此不再赘述。

```
/* In a traversal of input string */

            // reading numbers.
            // state:
            // 0 - Not Reading a Number
            // 1 - Reading Number w/o Decimal Point
```

```
            // 2 - Leading 0
            // 3 - Reading Number w/ Decimal Point
            if ((input[i] >= '0' && input[i] <= '9') || input[i] ==
'.') {
                switch (state) {
                case 0:
                    if (input[i] == '.')
                        state = 3, intP = 0, fraP = 0, base = 0.1;
                    else if (input[i] == '0')
                        state = 2, intP = 0;
                    else
                        state = 1, intP = input[i] - '0';
                    break;
                case 1:
                    if (input[i] == '.')
                        state = 3, fraP = 0, base = 0.1;
                    else
                        intP = intP * 10 + input[i] - '0';
                    break;
                case 2:
                    if (input[i] == '.')
                        state = 3, fraP = 0, base = 0.1;
                    else {
                        state = 1, intP = intP * 10 + input[i] - '0';
                    }
                    break;
                case 3:
                    if (input[i] == '.')
                        return "[Error] Too many decimal points in
number.";
                    else {
                        fraP += base * (input[i] - '0');
                        base *= 0.1;
                    }
                    break;
                } // end of switch(state)
                i++;
            }

/* Another part */

        // The end of reading a number
        if (state != 0 && (input[i] < '0' || input[i] > '9') &&
input[i] != '.') {
            numStack.push(Token());
```

```
            if (state == 3) {
                lastTokenTy = numStack.top().ty = RNUM;
                numStack.top().rval = intP + fraP;
            }
            else {
                lastTokenTy = numStack.top().ty = INUM;
                numStack.top().ival = intP;
            }
            state = 0;
        }
```

词法分析过程会对非法的字符（不可能出现在上述 token 中的字符）和非法字符串（如 csc）等进行检测；出现错误的会在结果栏中提示 `[Error] Lexical Error`。

3.3 表达式求值

实际上，如我们在 3.1 节看到的那样，当我们对操作符进行压栈时，就应当作一些出栈和计算；另外由于出栈和计算在逻辑上是同时进行的，我们将这两个步骤也进行合并。基本的压栈和出栈的逻辑是：

3.3.1 压栈

- 出现 SIN 或 COS 时，检查上一个 token 是不是数字或右括号。如果是的话（如 2sinPI, (1+2)cos PI 等），压入一个 MUL 后再压入当前 token；
- 出现 PI 时，直接将 π 的值作为 RNUM 压入数字栈；
- 出现乘法、除法时，将栈顶所有乘法、除法、SIN、COS 出栈并计算（因为 SIN 和 COS 优先级比乘除法高，而乘除法是左结合的），将结果压入数字栈，将当前 token 压入符号栈；
- 出现加法、减法时，将栈顶所有加法、减法、乘法、除法、SIN、COS 出栈并计算，将结果压入数字栈，将当前 token 压入符号栈；
    - 特别地，如果减号前面没有 token 或是左括号，说明此时减号作为负号使用，额外在栈中压入 INUM，其值为 0。
- 出现左括号时，直接压栈；
- 出现右括号时，将栈顶所有加法、减法、乘法、除法、SIN、COS 出栈并计算，直至遇到一个左括号将其出栈，将结果压入数字栈，右括号不压栈。
    - 特别地，如果没有找到左括号，提示错误 `[Error] Unbalanced parentheses`。

3.3.2 出栈和计算

下面的计算都会对运算数是 INUM 还是 RNUM 进行讨论；但是 SIN 和 COS 固定返回实数 RNUM ； 整数除法依据结果进行返回。

- 如果栈顶为 SIN 或 COS 而数字栈为空，或者栈顶为加减乘除而数字栈元素不足两个，提示语法错误 [Error] Syntax Error。（运算结束后，数字栈内的元素不为 1 个时也会提示该错误，例如 1PI。）
- 对于加减乘除这些二元运算，对数字栈顶两个元素出栈进行相应计算后将结果压栈。
    - 对于除法，如果除数和被除数均为整数且 **能够整除**，结果才返回整数；否则返回浮点数。
    - 特别地，对于除数为 0 的情况，提示错误 [Error] Devide by 0 （考虑浮点误差）。
- 对于 SIN 和 COS，直接对数字栈顶的元素出栈进行运算再将结果压栈即可。

3.4 类和数据结构概述

MyCalculator 类实现了计算功能，类定义如下：

```
class MyCalculator {
private:
    std::stack<Token> opStack, numStack;
    std::string input;
    enum Type lastTokenTy;
    std::string lexicalAnalysis();
    std::string pushOp(char op);
    std::string popOp();

public:
    MyCalculator() = default;
    std::string calculate(std::string &input);
};
```

- 可以看到，这个类封装良好，除了构造函数外唯一外界可用的函数即为 calculate 函数。这个函数实现了计算的启动，返回错误信息或计算结果。
- opStack 和 numStack 是两个 token 栈，其含义已经在前面详细讲解。input 是输入字符串的一份拷贝。lastTokenTy 是词法分析中上一个 token 的类型。
- lexicalAnalysis 函数实现了词法分析，pushOp 和 popOp 分别实现了压栈和出栈计算的功能，具体逻辑已经在 3.2 和 3.3 节展示。

4 MFC 设计概述

有关 MFC 的文件及具体更改已经在第 2 节中阐述，此处简述一些 MFC 设计过程中的具体考量及其实现。

- MFC 的界面设计在 `Calculator.rc` 中进行。为了保证良好的工程习惯，我们将需要交互的控件进行了正确的命名：

```
#define IDC_BUTTON_CALC                 1000
#define IDC_BUTTON_EXIT                 1001
#define IDC_EDIT_INPUT                  1002
#define IDC_EDIT_OUTPUT                 1003
```

- 我们将输出框设为 Read Only，这是符合工程逻辑的。
- 我们通过重载 `CCalculatorDlg::PreTranslateMessage` 函数，实现了回车进行计算的逻辑：

```
BOOL CCalculatorDlg::PreTranslateMessage(MSG* pMsg)
{
    if (pMsg->message == WM_KEYDOWN && pMsg->wParam == VK_RETURN)
    {
        startCalc();
        return TRUE;
    }
    return __super::PreTranslateMessage(pMsg);
}
```

- 设置了更改输入时清空输出的逻辑：

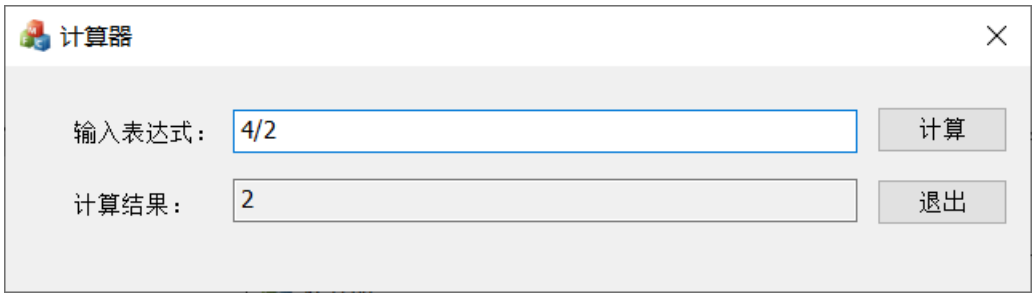
```
void CCalculatorDlg::OnEnChangeEditInput()
{
    // when the input is changed, clear the output textbox
    CString output;
    output.Format(_T(""));
    setOutput(output);
}
```

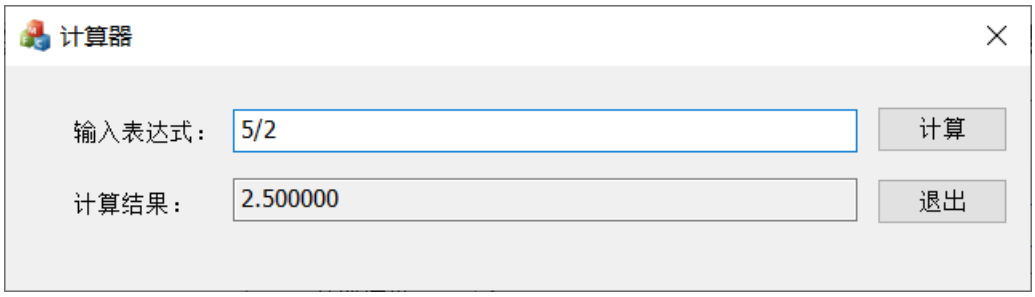
- 将输入输出文本框的变量以及 `MyCalculator` 的实例设为私有；等。更多细节不再赘述。

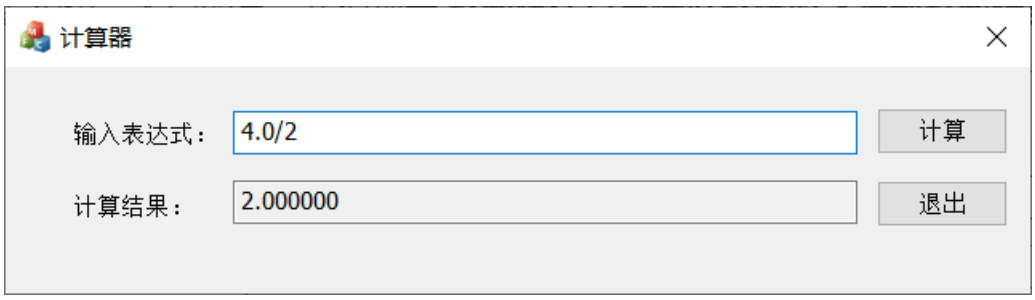
5 测试与分析

5.1 运算测试

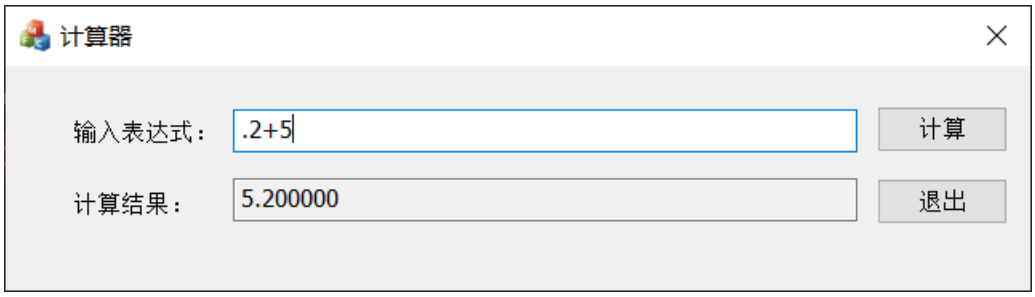
- 与 C++ 等编程语言相同，加法、减法、乘法如果有浮点数参与，则结果为浮点数；否则为整数。
- 但与 C++ 等编程语言不同的是，我们规定 **整数除法只有恰好整除时才得到整数结果**。例如：

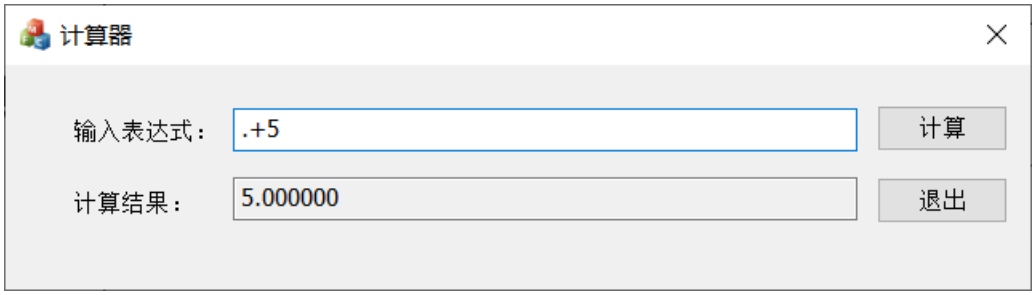




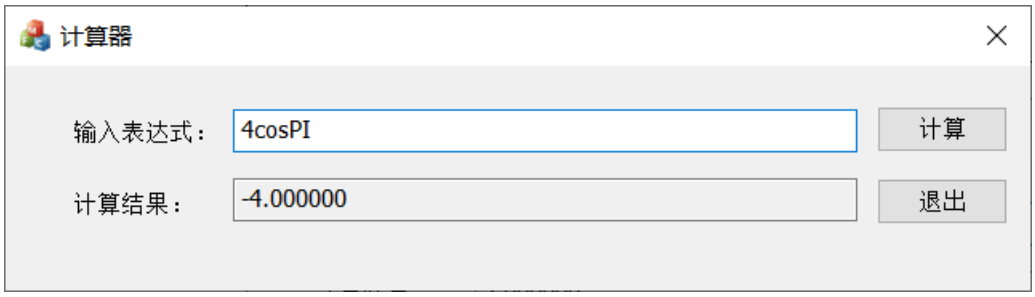


- 词法上，我们支持 `0.2` 等没有小数部分的数省略整数部分；实际上两部分都可以省略；但是有小数点就标明是一个实数，进行实数运算：

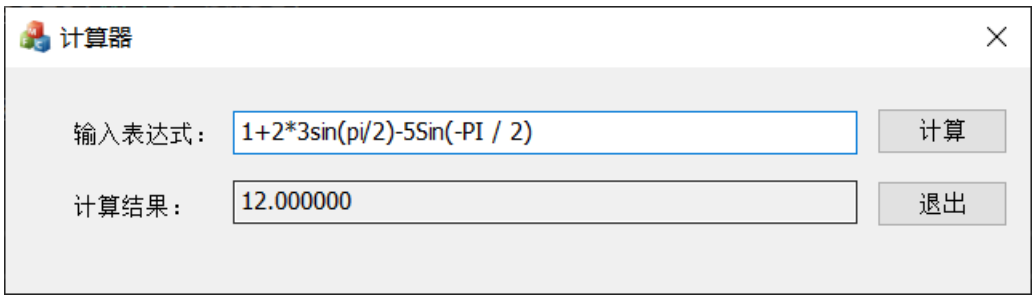




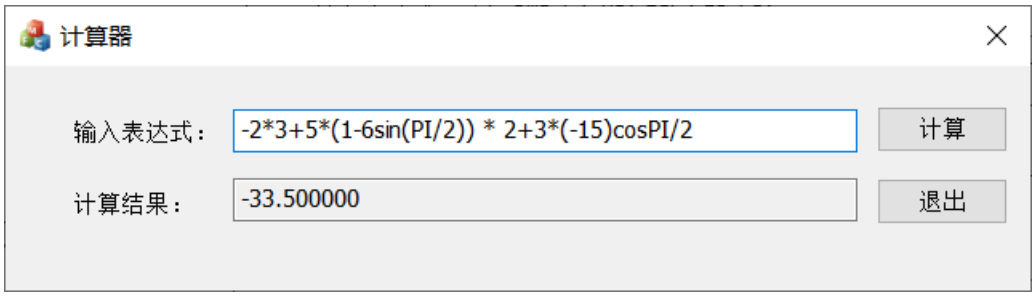
- 另外，我们规定 SIN 和 COS 的返回值一定是实数：



- sin, cos 和 PI 均不区分大小写，表达式里的空格也都会被忽略：



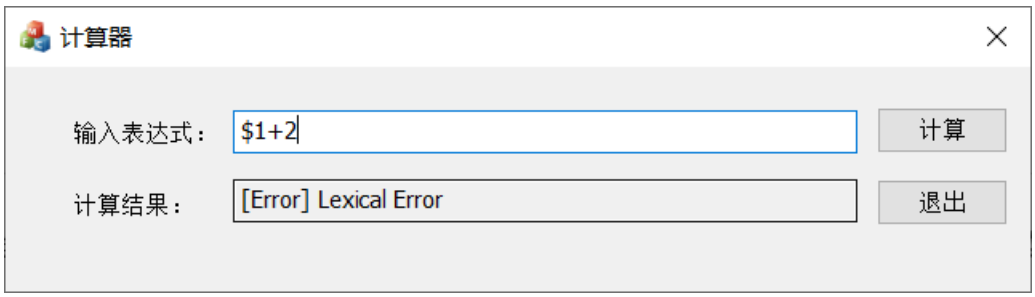
- 上例和本例测试了一些更复杂的运算：

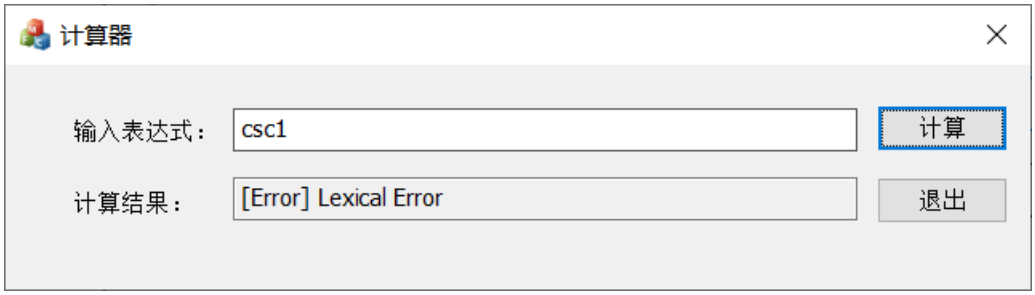


5.2 错误提示

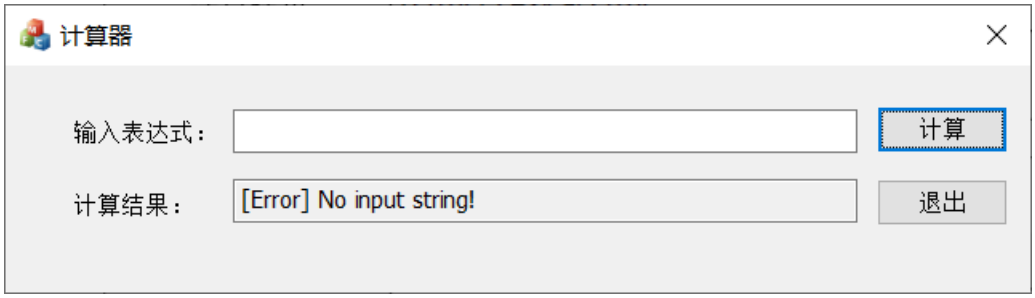
5.2.1 词法错误提示

如 3.2 节所述，词法分析过程会对非法的字符和非法字符串等进行检测：

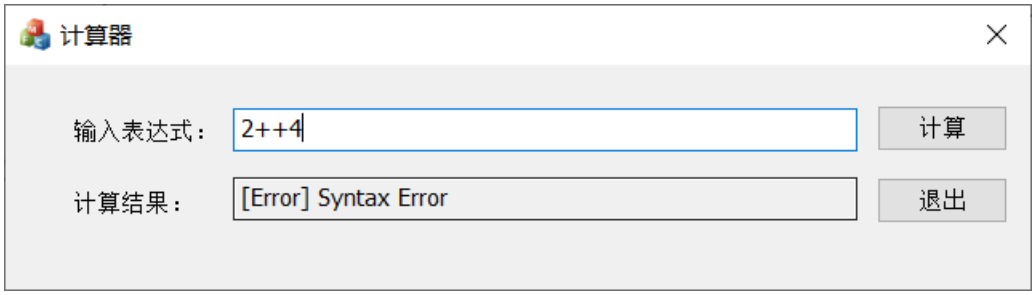


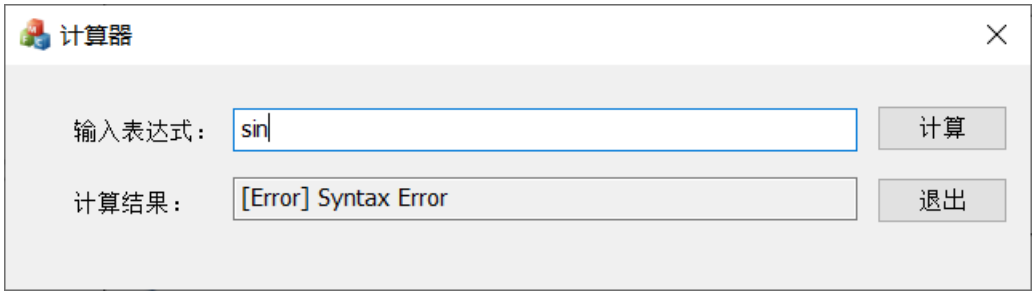


另外，对于空输入，也会提示 `[Error] No input string!`：

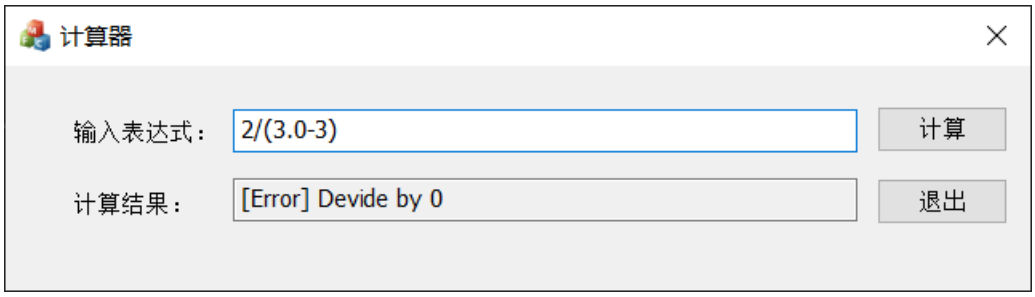


5.2.2 语法错误提示

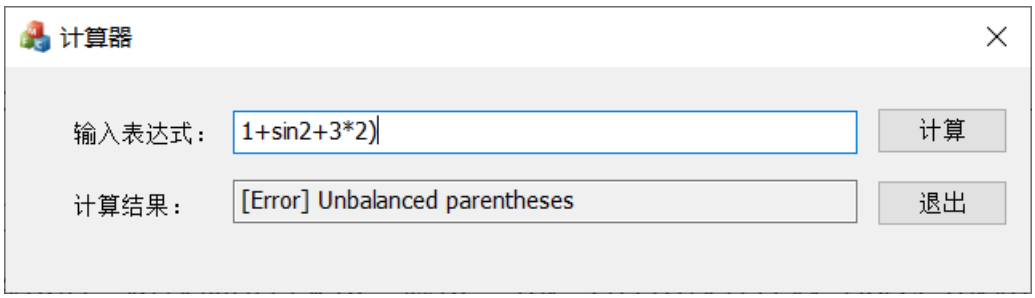




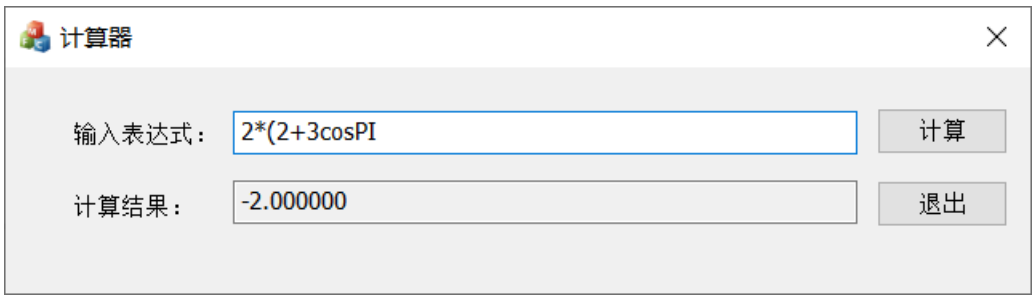
5.2.3 除以 0 错误



5.2.4 括号不配对



而对于缺少右括号这样的情况，程序可以算出正确结果。因为缺少的右括号理应在程序最右边，那么在左括号的维持下，这样的运算顺序和有右括号的情况恰好相同：



5.3 分析

总体而言，本工程实现了一个功能相对完整、交互相对友好的带图形界面的计算器。本工程的亮点在于：

- **良好的工程结构。**`MyCalculator` 类是一个包装良好的类，这个类与 MFC 界面的唯一交互就是 `calculate` 函数；这个类也可以方便地移植到其他 C++ 工程中。另外本工程对成员变量和函数进行了细致的封装，防止了其他用户使用时产生不必要的疑惑，也防止了被恶意用户攻击。
- 支持了四则运算、括号、负号和 sin, cos，**可以基本完成常用的计算。**
- 迎合了一般使用者的习惯，**提供良好的用户体验。**例如：
  - 不同于编程语言而更接近实际的整数/实数判别方式，在保证结果准确的情况下尽可能保持整数运算；
  - 支持 3sin PI，(15+2)cosPI，.3 这样的常用省略写法；

    - 忽略空格、大小写，尊重用户的输入习惯；
    - 回车进行计算，重新输入表达式时清空输出；
    - 考虑到多种可能的错误，出现时给出相应信息而不会崩溃；等。

同时，本工程也有一定缺点，例如：

- 当表达式超出框体长度时，不移动光标不能看到前面的内容。其他计算器同样存在这个问题；但如果能够动态调整窗体大小，显示完整算式则更好；
- 虽然理论上没有表达式长度限制，但是由于设计使用 `int` 和 `double` 进行数据的保存，可能会由于一些超出数据范围的结果或中间结果而计算错误，同时 `double` 精度不满足的情况下也会出现一些差错。