


数据结构课程设计项目说明文档

——算数表达式求解

作者姓名：陈垚昕
学号：
指导教师：张颖
学院/专业：软件学院/软件工程

数据结构课程设计项目说明文档

——算数表达式求解

- 1. 分析
 - 1.1 背景分析
 - 1.2 功能分析
- 2. 设计
 - 2.1 基础数据结构设计与基础信息
 - 链表结点结构设计
 - 链表栈类设计
 - 2.2 计算器类设计
 - 2.3 符号优先级设定
- 3. 核心功能实现
 - 3.1 核心运算操作
 - 流程图
 - 3.2 输入中缀表达式并进行合法性预先检查
 - 3.3 中缀表达式符号，运算数入栈
 - 处理数字（浮点数）
 - 处理运算符
 - 3.4 处理双目运算符与括号情况
 - 直接入栈情况
 - 遇到右括号)的情况
 - 栈顶元素优先级大于当前运算符优先级
 - 流程图
 - 3.5 根据栈的结果进行最终计算
 - 3.6 系统类运行
 - 流程图
- 4. 测试
 - 4.1 常规结果测试
 - Task 1 范例
 - Task 2 一般四则运算
 - Task 3 含单对括号的运算
 - Task 4 含多重括号的运算
 - Task 5 含单目运算符的运算
 - Task 6 浮点数运算功能
 - Task 7 浮点数的取余运算功能
 - 4.2 边界测试
 - Task 1 单个数的输出
 - Task 2 括号在输入表达式的首位
 - Task 3 括号中只有一个数字而非运算式
 - 4.3 错误结果测试
 - Task 1 括号数目不匹配
 - Task 2 出现除以0的操作
 - Task 3 输入表达式中含有非法符号
 - Task 4 输入表达式中不含有=
 - Task 5 输入表达式中=未在末尾出现
 - Task 6 输入表达式中未按照合法的形式安排符号与计算式

1. 分析

1.1 背景分析

从键盘上输入中缀算数表达式，包括括号，计算出表达式的值。

1.2 功能分析

程序对所有输入的表达式作简单的判断，如表达式有错，能给出适当的提示。支持包括加减，乘除取余，乘方和括号等操作符，其中优先级是等于<括号<加减<乘除取余<乘方，且能处理单目运算符：+或-。

在满足题目要求的前提下，该计算器同时支持浮点数的运算（包括取余操作）

2.设计

2.1 基础数据结构设计与基础信息

计算器问题的求解关键是把中缀表达式转化为后缀表达式。中缀表达式虽然是常见的表达式形式，但难以被计算机识别。而后缀表达式则是对计算机易于处理的表达式。故本程序的核心思想是把中缀表达式转化为后缀表达式，将中缀表达式的运算符与操作数按照某种规则存放在相应的栈中，并伴随优先级处理。将中缀表达式处理完成后，通过两个栈中存放的内容与顺序，即可以后缀表达式的处理方式，从运算数栈取出两个数，从运算符栈取出一个操作符，运算后将结果存入运算数栈的操作。循环的结果是最终运算数栈中唯一的一个数据。

本程序使用模板类链表栈形式存储数据，并通过计算器类**Calculator**进行运算操作。

链表结点结构设计

```
template <class T>
struct LinkNode {
    //数据成员
    T data;
    LinkNode<T>* nextNode;

    //构造函数
    LinkNode() :nextNode(nullptr) {}
    LinkNode(const T& value) :data(value), nextNode(nullptr) {}
};
```

链表栈类设计

```
template <class T>
class LinkStack {
public:
    //构造与析构函数
    LinkStack() = default;
    ~LinkStack() { clear(); };

    //内容获取
    bool empty()const { return stackSize == 0; }
    LinkNode<T>* top() { return _pTop; }

    //链表单元操作
    void push(const T& x);
    void pop();

    //链表操作
    void clear();
```

```
private:
    LinkNode<T>* _pTop;
    size_t stackSize;
};
```

2.2 计算器类设计

本课设通过 **Calculator** 类实现计算器功能，实现良好的封装。成员包括一个用于存储运算数的double类型链表栈 **operandStack**，一个用于存储运算符的char类型链表栈**operatorStack**。该类运行时，先输入一个中缀表达式，并将中缀表达式转化为后缀表达式的存储栈形式，最终通过计算求出最终结果。过程中包括判断表达式的合法性，不合法的情况包括：1.预处理不合法（括号不匹配，非法符号）；2.中缀表达式符号、运算数入栈时出现栈空时仍需要进行pop()操作的情况（运算不合法）3.根据栈计算最终结果时出现栈空时仍需要进行pop()操作的情况（运算不合法）

```
class Calculator {
public:
    //运行程序
    void run();
private:
    LinkStack<double> operandStack;    //操作数栈，存放表达式中的式子
    LinkStack<char> operatorStack;    //操作符栈，存放操作符并按优先级出栈入栈

    int prior(const char ch)const;    //获取各个操作符的优先级

    bool inputInfixExpression(string& infix);    //输入中缀表达式并初步检查合法性

    bool processInfixExpression(const string& infixExpression); //将每个数字分解入栈

    bool processOperator(const string& infixExpression, int index); //处理双目运算符以及括号等

    bool process2Operands();    //从数字栈顶端弹出两个数字，符号栈顶端弹出一个符号，计算，放回数字栈

    double getAnswer(double a, char b, double c)const;

    bool calculateResult();    //对分解后的两个栈进行计算得到最终结果
};
```

2.3 符号优先级设定

按照题目给定要求进行设定

优先级	0	1	2	3
符号	()	+ -	* / %	^

3.核心功能实现

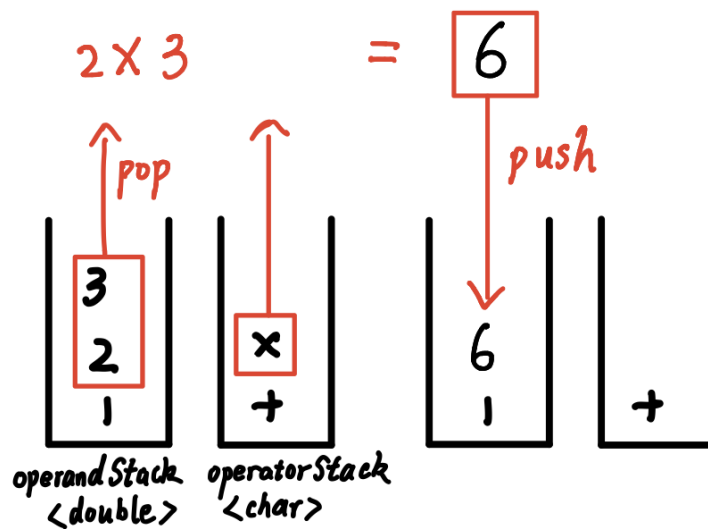
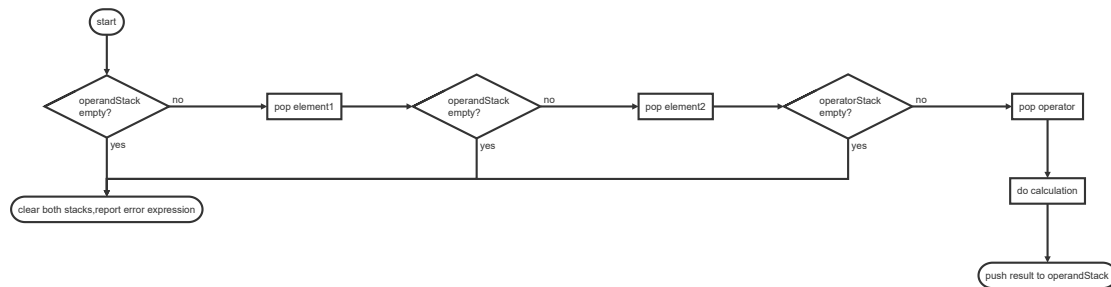
3.1 核心运算操作

相关函数：**bool Calculator::process2Operands()**

在栈顶非空的前提下，从数字栈顶端弹出两个数字，从操作符栈顶端弹出一个操作符，进行相应的运算后放入数字栈顶端。如果栈顶为空却出现了出栈的操作时，说明此时表达式非法，清空栈内元素并返回**false**指示程序提示表达式非法。

(注：由于直接使用%计算浮点数的取余是非法的，故使用了中的**fmod**函数处理两个浮点数的取余)

流程图



3.2 输入中缀表达式并进行合法性预先检查

输入一个中缀表达式，先遍历表达式查看是否出现非法字符（不是数字，小数点，包括左右括号在内的运算符；或者出现合法运算符如 '=' 出现在非法位置的情况），并且以左括号为+1，右括号为-1的计算法则考虑括号统计的权值。如果检查非法（出现非法字符，或在遍历过程中或遍历完成后括号统计权值小于0），输出相应的错误提示并要求重新输入，返回**false**。若在这些方面没有问题（但不排除计算非法的情况），则返回**true**继续后续计算

```
for (int i = 0; i < infix.size(); i++)
{
    char ch = infix.at(i);
    if (!(isExpressionOperator(ch) || (ch >= '0' && ch <= '9') || ch == '.'))
```

```

{
    if (ch != '=' || (ch == '=' && i < infix.size() - 1))
    {
        std::cerr << "错误：出现非法字符,请重新";
        return false;
    }
}

//考虑括号匹配情况
if (ch == '(') bracketCounter++;
if (ch == ')')bracketCounter--;
if (bracketCounter < 0) break;
}

if (bracketCounter != 0)
{
    std::cerr << "错误：括号不匹配,请重新";
    return false;
}

```

3.3 中缀表达式符号，运算数入栈

遍历中缀表达式，处理每一个字符串，并根据字符串的性质进行相应的处理

处理数字（浮点数）

处理string中一段含有数字与小数点的序列，该序列即为一个操作数.将这段序列用一个**string**复制下来并使用**std::stringstream**的方法将其从代表数字的字符串转换为浮点数，同时根据前面是否有单目符号标记，决定是将这个操作数本身还是相反数形式放入运算数栈**operandStack**中

```

//处理浮点数，用stringstream方法将浮点数字字符串转化为浮点数double
while (!isExpressionOperator(currentChar) && i < infixExpression.size() - 1)
{
    number.push_back(currentChar);
    currentChar = infixExpression.at(++i);
}
--i;
std::stringstream stringToDouble;
stringToDouble << number;
stringToDouble >> data;
if (isNegative)
{
    //对负数的处理
    operandStack.push(-data);
    isNegative = false;
}
else
{
    operandStack.push(data);
}
number.clear();

```

处理运算符

处理string中一个表示运算符的字符（包括+ - * / % ^ ()）先判断是否为单目运算符。若为单目负号，设置标记，在后续处理运算数的操作中，根据此标记将操作数取反。若为双目运算符以及括号，用**processOperator(infixExpression, i)**进行处理

```

//单目运算符+ -的处理方式
if (currentChar == '-' && isUnary(infixExpression, i))
{
    isNegative = true;
}

```

```
else if (currentChar == '+' && isUnary(infixExpression, i))
{
    continue;
}
else
{
    //处理双目运算符以及括号
    if (!processOperator(infixExpression, i))
    {
        return false;
    }
}
```

3.4 处理双目运算符与括号情况

根据运算符的性质进行相应处理

直接入栈情况

当以下情况发生时，直接将操作符推入操作符栈

	直接入栈情况	对应判断条件
1	运算符栈空	operatorStack.empty()
2	当前运算符为左括号	curChar == '('
3	当前栈顶为左括号但当前不为右括号	operatorStack.top()->data == '(' && curChar!=')
4	栈顶元素优先级小于当前运算符优先级	prior(curChar) > prior(operatorStack.top()->data)

遇到右括号)的情况

由于已经在预处理阶段进行了括号匹配的检查，故可以确保在栈中必然有与之匹配的左括号，此时分以下几种情况：

	栈顶情况	对应的数据存储情况	执行的操作
1	栈顶为左括号	左右括号之间只存在操作数	直接将栈顶左括号取出
2	栈顶为运算符	左右括号之间存在计算式	反复执行process2Operands()直到运算符栈栈顶为左括号，再将左括号取出

```
char ctop = operatorStack.top()->data; //获取栈顶元素

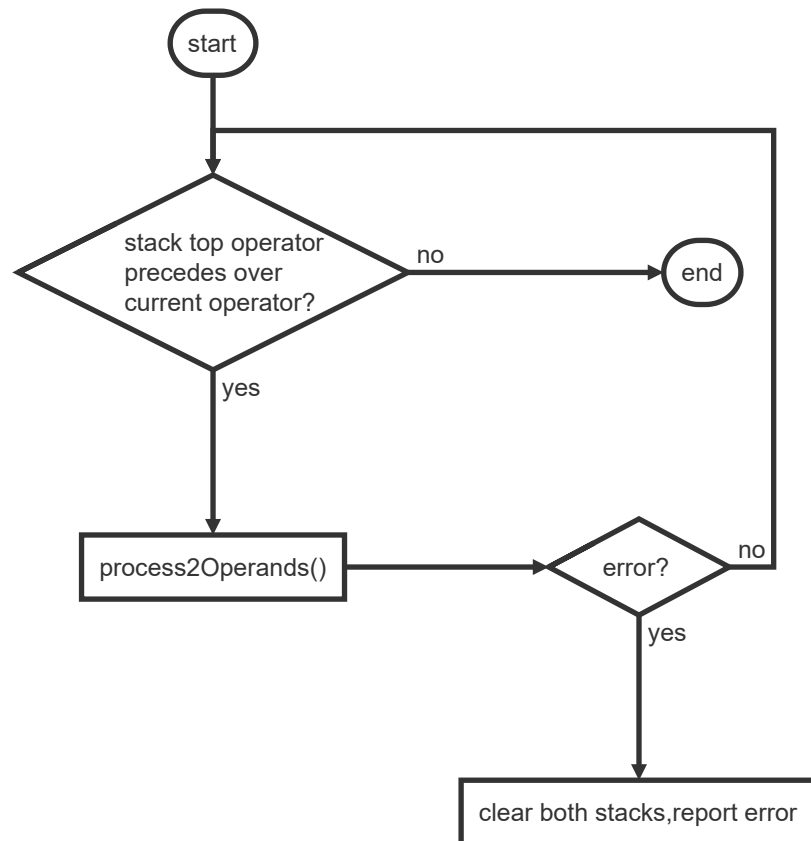
if (ctop == '(')
{
    operatorStack.pop();
}
else
{
    while (ctop != '(')
    {
        //反复执行计算操作，直到operatorStack栈顶为'('，再将'('取出
        if (!process2Operands())
        {
            return false;
        }
        ctop = operatorStack.top()->data;
    }
}
```

```
operatorStack.pop();  
}
```

栈顶元素优先级大于当前运算符优先级

当栈顶元素优先级大于当前运算符优先级时，譬如当前栈顶为 '*' 但当前运算符为 '+', 此时应优先运算乘号再运算加号。这种情况发生时，反复对栈顶进行运算操作，直至栈顶的运算符优先级比当前的优先级小时，才停止计算并让当前运算符入运算符栈

流程图



```
while (true)
{
    if (operatorStack.empty()) break;
    else if ((prior(operatorStack.top()->data) >= prior(curChar)))
    {
        char ctop = operatorStack.top()->data;
        if (!process2Operands())
        {
            return false;
        }
    }
    else break;
}
operatorStack.push(curChar);
```

3.5 根据栈的结果进行最终计算

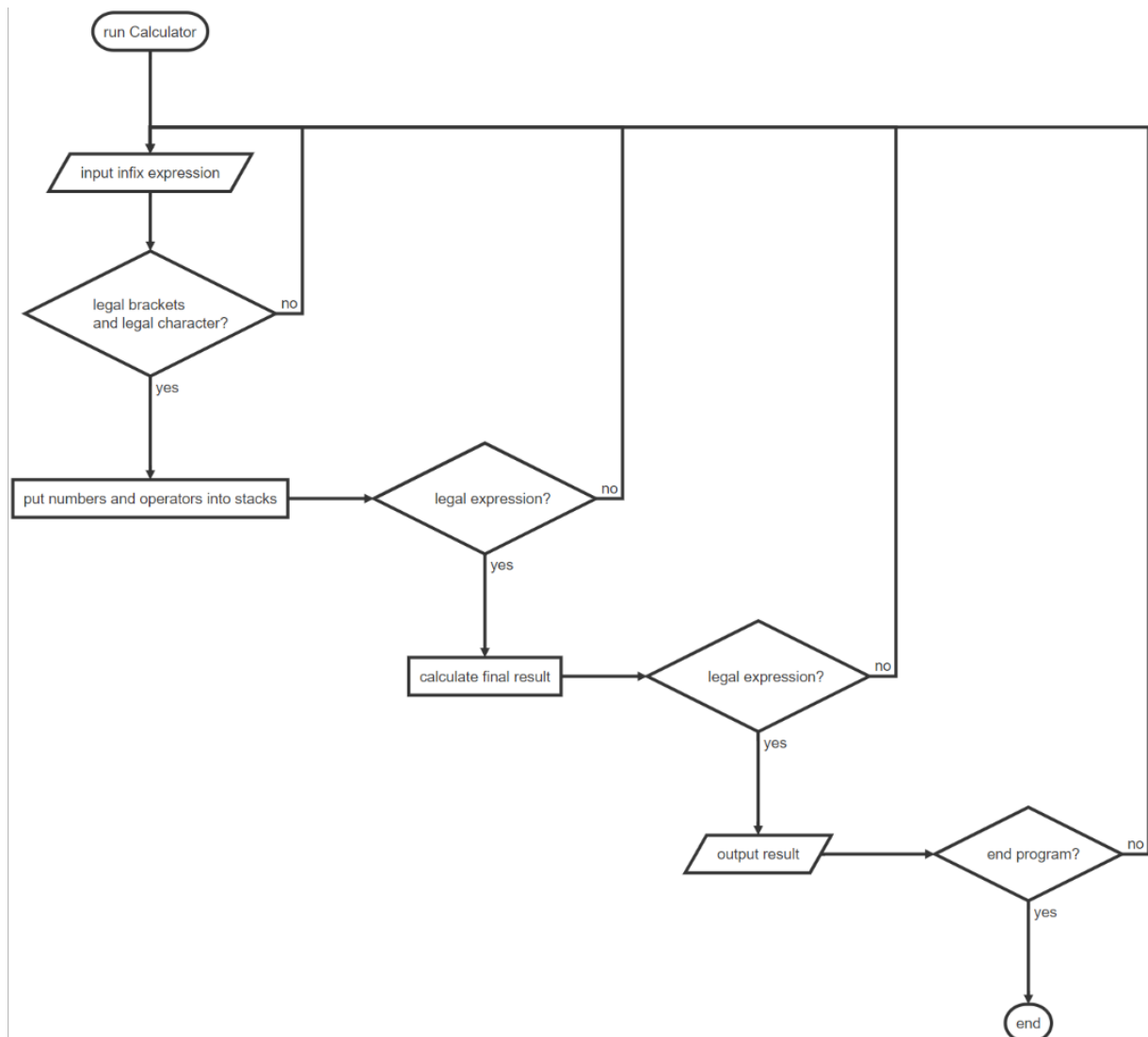
通过**bool Calculator::calculateResult()**函数实现，不断进行**process2Operands()**计算直至运算符栈为空，计算数栈只剩一个结果，此结果便为表达式的求值

```
bool Calculator::calculateResult()
{
    while (!operatorStack.empty())
    {
        if (!process2Operands())
        {
            return false;
        }
    }
    return true;
}
```

3.6 系统类运行

系统类通过**void Calculator::run()**函数运行计算器程序。运行时，按照**输入中缀表达式--将中缀表达式转化为后缀表达式的存储栈形式--计算求出最终结果**进行操作。过程中对每一步进行合法性检查。不合法的情况包括：1.预处理不合法（括号不匹配，非法符号）；2.中缀表达式符号、运算数入栈时出现栈空时仍需要进行pop()操作的情况（运算不合法）3.根据栈计算最终结果时出现栈空时仍需要进行pop()操作的情况（运算不合法）。当不合法的情况出现时，重新进行输入中缀表达式操作。若计算无误且选择退出程序时，程序退出。

流程图



```

std::cout << "输入表达式:" << std::endl;
if (!inputInfixExpression(infixExpression)) //输入表达式
{
    continue;
}

if (!processInfixExpression(infixExpression)) //将表达式数字与符号放入栈中
{
    std::cerr << "错误: 表达式非法,请重新";
    continue;
}
if (!calculateResult()) //根据栈中的情况计算结果
{
    std::cerr << "错误: 表达式非法,请重新";
    continue;
}
if (!_finite(operandStack.top()->data)) //若除数为0时用_finite函数判断是否有理数范围内
{
    std::cout << operandStack.top()->data << std::endl; //合法, 正常输出
}
else
{
    std::cerr << "错误: 表达式非法,出现除数为0的非法运算,请重新";
    continue;
}

```

4.测试

4.1 常规结果测试

Task 1 范例

测试样例1：

$-2*(3+5)+2^3/4=$

预期结果1：

-14

实际结果1：

```
ckx@VM-0-10-ubuntu:~$  
输入表达式：  
-2*(3+5)+2^3/4=  
-14  
是否继续(y,n)?
```

测试样例2：

$2^4/8-(+2+8)\%3=$

预期结果1：

1

实际结果1：

```
ckx@VM-0-10-ubuntu:~$  
输入表达式：  
2^4/8-(+2+8)%3=  
1  
是否继续(y,n)?
```

Task 2 一般四则运算

测试样例：

$12+3678*256-2670/2=$

预期结果：

940245

实际结果：

```
是否继续(y,n)?  
y  
输入表达式：  
12+3678*256-2670/2=  
940245  
是否继续(y,n)?
```

Task 3 含单对括号的运算

测试样例：

```
265*(1+273%3^4+2^4)=
```

预期结果：

```
12455
```

实际结果：

```
输入表达式：
265*(1+273%3^4+2^4)=
12455
是否继续(y,n)?
```

Task 4 含多重括号的运算

测试样例：

```
278*((2567-2673)+111)=
```

预期结果：

```
1390
```

实际结果：

```
是否继续(y,n)?
y
输入表达式：
278*((2567-2673)+111)=
1390
```

Task 5 含单目运算符的运算

测试样例：

```
-1+(-1*2)=
```

预期结果：

```
-3
```

实际结果：

```
输入表达式：
-1+(-1*2)=
-3
是否继续(y,n)?
```

Task 6 浮点数运算功能

测试样例：

```
1.1919*114.514/7777=
```

预期结果：

```
0.0175504
```

实际结果：

```
ckx@VM-0-10-ubuntu:
输入表达式:
1.1919*114.514/7777=
0.0175504
是否继续(y,n)?
```

Task 7 浮点数的取余运算功能

测试样例：

1.21%1.1

预期结果：

0.11

实际结果：

```
输入表达式:
1.21%1.1=
0.11
是否继续(y,n)?
```

4.2 边界测试

Task 1 单个数的输出

测试样例：

12=
-12=

预期结果：

分别为12 -12

实际结果：

```
输入表达式:
12=
12
是否继续(y,n)?
y
输入表达式:
-12=
-12
```

Task 2 括号在输入表达式的首位

测试样例：

(3+7)%2+1=

预期结果：

1

实际结果：

```
是否继续(y,n)?
```

```
y
```

```
输入表达式:
```

```
(3+7)%2+1=
```

```
1
```

Task 3 括号中只有一个数字而非运算式

测试样例1:

```
1-(-2)=
```

预期结果1:

```
3
```

实际结果1:

```
输入表达式:
```

```
1-(-2)=
```

```
3
```

测试样例2:

```
-(-3)^2-1=
```

预期结果2:

```
-10
```

实际结果2:

```
输入表达式:
```

```
-(-3)^2-1=
```

```
-10
```

```
是否继续(y,n)?
```

4.3 错误结果测试

Task 1 括号数目不匹配

测试样例:

```
278*(((2567-2673)+111)=
```

预期结果:

```
错误: 括号不匹配,请重新输入表达式:
```

实际结果:

```
输入表达式:
```

```
278*(((2567-2673)+111)=
```

```
错误: 括号不匹配,请重新输入表达式:
```

Task 2 出现除以0的操作

测试样例:

```
1+2/0=
```

预期结果：

错误：表达式非法,出现除数为0的非法运算,请重新输入表达式：

实际结果：

```
错误：括号不匹配,请重新输入表达式：
1+2/0=
错误：表达式非法,出现除数为0的非法运算,请重新输入表达式：
```

Task 3 输入表达式中含有非法符号

测试样例：

3&9!0=

预期结果：

错误：出现非法字符,请重新输入表达式：

实际结果：

```
错误：表达式非法,出现除数为0的非法运算,请重新输入表达式：
3&9!0=
错误：出现非法字符,请重新输入表达式：
```

Task 4 输入表达式中不含有=

测试样例：

1+2+3+4+5

预期结果：

为用户添加=号，结果仍为正确结果 15

实际结果：

```
错误：出现非法字符,请重新输入表达式：
1+2+3+4+5
15
```

Task 5 输入表达式中=未在末尾出现

测试样例：

1+2=13+35=

预期结果：

错误：出现非法字符,请重新输入表达式：

实际结果：

```
输入表达式：
1+2=13+35=
错误：出现非法字符,请重新输入表达式：
```

Task 6 输入表达式中未按照合法的形式安排符号与计算式

测试样例：

1+-39+27895-123-53(124)=

预期结果：

错误：表达式非法,请重新输入表达式：

实际结果：

1+-39+27895-123-53(124)=
错误：表达式非法,请重新输入表达式：