数据结构课程设计项目说明文档

——表达式转换

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668154)

[1.3 项目需求分析 1](#_Toc495668155)

[2 项目设计 3](#_Toc495668156)

[2.1 数据结构设计 3](#_Toc495668157)

[2.2 类设计 3](#_Toc495668158)

2.2.1 结点类（StackNode） 3

2.2.2 链式栈类（LinkedStack） 4

2.2.3 字符串类（String） 5

2.2.4 迭代器类（iterator） 7

2.3 算法设计 7

2.3.1 算法思路 7

2.3.2 性能评估 8

2.3.3 流程图表示 8

2.3.4 代码实现 9

·对表达式进行预处理 9

·检查表达式是否合法 10

·将表达式转换为后缀表达式 12

[3 项目测试 1](#_Toc495668161)4

[3.1 正常测试6种运算符 1](#_Toc495668162)4

[3.2 嵌套括号 1](#_Toc495668166)4

[3.3 运算数超过1位整数且有非整数出现 1](#_Toc495668170)4

[3.4 运算数有正或负号 1](#_Toc495668174)5

[3.5 只有1个数字 1](#_Toc495668178)5

[3.6 数字输入错误的情况 1](#_Toc495668182)5

[3.7 括号不匹配的情况 1](#_Toc495668182)6

[3.8 操作符与操作数数量不匹配的情况 1](#_Toc495668182)6

3.9 带单目负号的情况 16

3.10 输入带空格的情况 16

**1.项目分析**

1.1 项目背景

算数表达式有前缀表示法，中缀表示法和后缀表示法等形式。日常使用的算术表达式是采用中缀表示法，即二元运算符位于两个运算数中间。请设计程序将中缀表达式转换成为后缀表达式。

1.2 项目要求

输入说明：输入在一行中给出以空格分隔不同对象的中缀表达式，可包含+, -, \*, /, -, \*, /以及左右括号，表达式不超过20个字符（不包括空格）。

输出说明：在一行中输出转换后的后缀表达式，要求不同对象（运算数，运算符号）之间以空格分隔，但是结尾不得有多余空格。

测试用例：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 输入 | 输出 | 说明 |
| 1 | 2+3\*(7–4)+8/4 | 2 3 7 4 - \* + 8 4 / + | 正常测试6种运算符 |
| 2 | ((2+3)\*4–(8+2))/5 | 2 3 + 4 \* 8 2 + - 5 / | 嵌套括号 |
| 3 | 1314+25.5\*12 | 1314 25.5 12 \* + | 运算数超过1位整数且有非整数出现 |
| 4 | -2\*(+3) | -2 3 \* | 运算数有正或负号 |
| 5 | 123 | 123 | 只有一个数字 |

1.3 项目需求分析

对于将表达式转换为后缀表达式的程序，需考虑以下需求：

**·正确性**

程序应当能够准确地将输入的表达式转换为后缀表达式，操作数或运算符之间要用空格隔开，末尾不能有多余空格。

**·健壮性**

程序应当对输入不合法表达式的行为做出判断。

**·功能提升**

程序可以对带空格的表达式进行处理

当输入表达式不合法时，程序可以输出不合法的原因。

**2.项目设计**

2.1 数据结构设计

由项目分析可以得出，该项目需要完成一个中缀表达式转换为后缀表达式的问题，设计思想如下：

（1）转换前后的表达式用字符串进行存储

（2）建立一个运算符栈，然后遍历整个表达式，若读到操作数则直接放入结果字符串中，若读到运算符则根据栈顶运算符的优先级进行对应操作。读到运算符优先级高，则压入栈中；栈顶运算符优先级高，则其弹出。进行以上循环直至遍历完成或栈空。

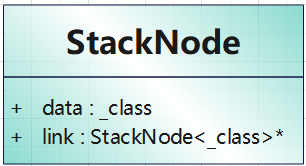
2.2 类设计

由项目分析可以得出，本项目需要用到一个栈，其本质上通过链表实现，其包括两个抽象数据类型（ADT）——链表结点类（ListNode）与链表类（List），而两个类之间的耦合关系可以采用嵌套、继承等多种关系。为了实现代码的复用性，本系统实现了一个链表为形式的栈。采用struct描述链表结点类（StackNode），这样使得链式栈类（LinkedStack）可以直接访问链表结点而不需要定义友元关系。

存储方面，本项目使用字符串类（String）对表达式进行存储。其本质上是通过字符数组实现的，同时添加了迭代器类（iterator）和一些运算符重载，以便对字符串的遍历和对字符串进行添加、赋值等操作。

**2.2.1 结点类（StackNode）**

链表结点存储了结点数据、后继节点位置，其UML图如下：

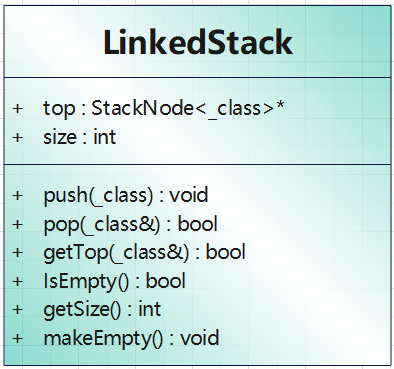


**2.2.2 链式栈类（LinkedStack）**

链式栈类本质上仍是单链表，保存了链表的基本结构。由于我们在使用栈时仅仅利用其“先进后出”的特性，所以其提供的操作并不像一般链表操作那样齐全，而是仅仅有入栈和出栈。

由于对栈的插入、删除元素等操作的位置比较固定，为了节省空间，并未采用附加头结点。

其UML图如下：



其中，主要函数如下：

//进栈

void push(const \_class x);

//出栈

bool pop(\_class& x);

//取栈顶元素

bool getTop(\_class& x)const;

//判栈空否

bool IsEmpty()const;

//返回栈元素个数

int getSize()const;

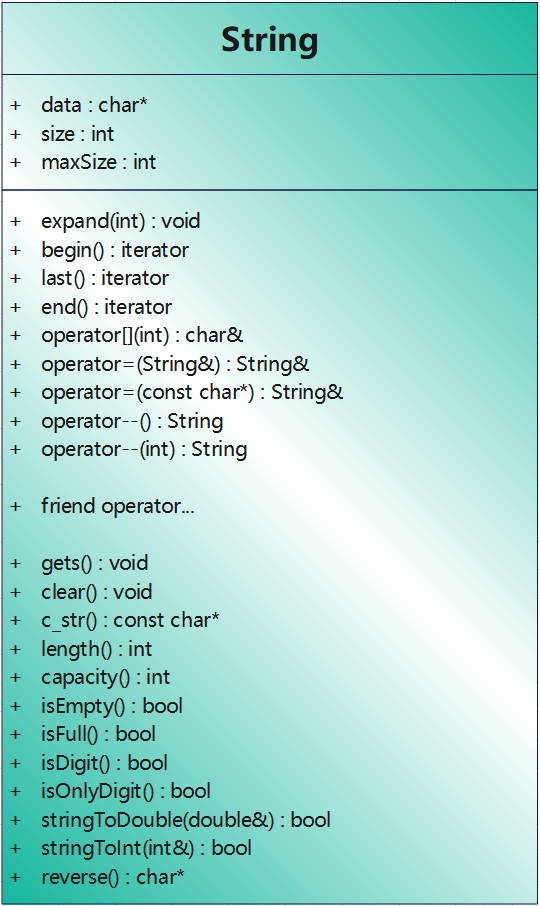
//清空栈的内容

void makeEmpty();

**2.2.3 字符串类（String）**

字符串本质上是通过字符数组实现的，通过动态开辟一个数组存放各个字符。由于字符串的添加、比较、赋值等操作使用得十分频繁，故该类重载了许多运算符的操作，使得这些操作使用起来更易上手。

其UML图如下：



其中，主要函数如下：

//增加数组空间大小

void expand(const int \_timesOfExpandingDefaultSize);

//返回字符串的起始位置

inline String::iterator begin();

inline const String::iterator begin()const;

//返回字符串末尾的后一个位置

inline String::iterator end();

inline const String::iterator end()const;

//返回字符串的末尾位置

inline String::iterator last();

inline const String::iterator last()const;

//重载函数：下标访问

char& operator[](const int pos)const;

//重载函数：复制（从String对象复制）

String& operator=(const String& str);

//重载函数：复制（从const char\*变量复制）

String& operator=(const char\* str);

//输入字符串（支持空格输入）

void gets();

//将字符串置为空

void clear();

//返回字符串首地址（以const char\*形式返回）

const char\* c\_str()const;

//返回字符串长度

int length()const;

//返回字符串最大容量

int capacity()const;

//判断字符串是否为空

bool isEmpty()const;

//判断字符串是否已满

bool isFull()const;

//判断字符串是否表示一个数值

bool isDigit()const;

//判断字符串是否仅有数字

bool isOnlyDigit()const;

//将字符串转成双精度数

bool stringToDouble(double& d)const;

//将字符串转成整型数

bool stringToInt(int& i)const;

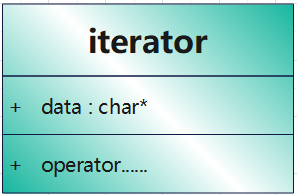
//将字符串反转

char\* reverse();

**2.2.4 迭代器类（iterator）**

另一种常用的字符串操作是对其的遍历，在此添加了迭代器类（iterator）和一些运算符重载，以便对字符串的遍历等操作。由于这个类名在标准库中已有实现，且此处的迭代器类为String类特有，故使用了嵌套类的形式，使得iterator只能在String类内访问。

其UML图如下：



2.3 算法设计

**2.3.1 算法思路**

将一个表达式转换为后缀表达式可以分为四步：

（1）对表达式进行预处理：去除所有空格，并将单目正号、单目负号转换为别的字符进行标记。

（2）判断表达式是否合法，若不合法，输出不合法原因并结束。

（3）遍历整个表达式，若读到操作数则直接放入结果字符串中，若读到运算符则根据栈顶运算符的优先级进行对应操作。读到运算符优先级高，则压入栈中；栈顶运算符优先级高，则其弹出（优先级参考下表，#是表达式结束符，@为单目负号）。进行以上循环直至遍历完成或栈空。

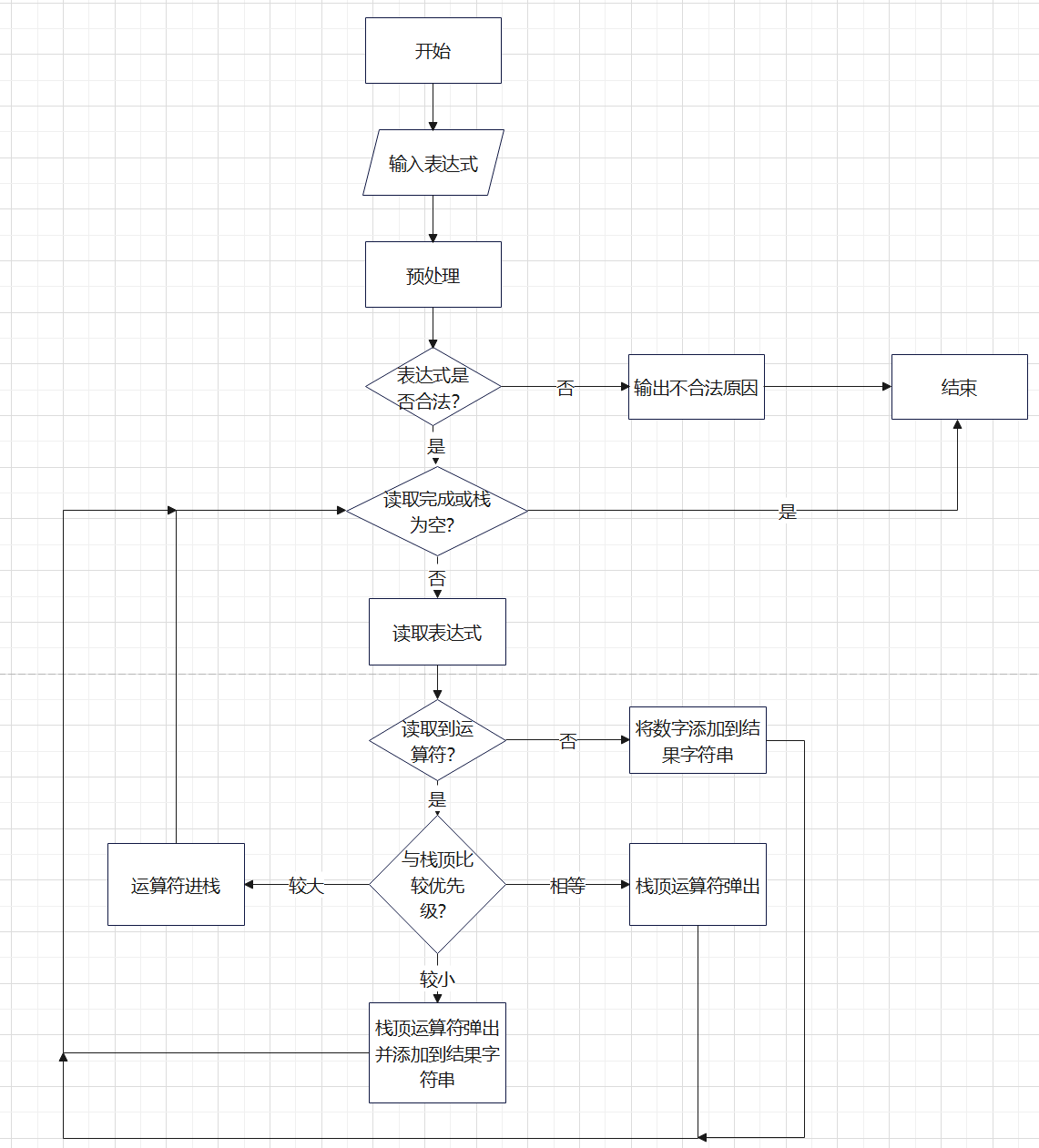
（4）打印后缀表达式，不带多余空格

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | # | ( | @ | \*、/ | +、- | ) |
| isp | 0 | 1 | 7 | 5 | 3 | 8 |
| icp | 0 | 8 | 6 | 4 | 2 | 1 |

**2.3.2 性能评估**

设输入的原始表达式有n个字符，对表达式的预处理、判是否合法、改变至后缀形式都最多遍历一次字符串，且过程中字符串长度不会增长，所需时间都是O(n)。综上，总的时间复杂度仍是O(n)。

**2.3.3 流程图表示**

****

**2.3.4 代码实现**

·对表达式进行预处理

void postfix\_prep(String& str)

{

    if (str.isEmpty())

        return;

    String ostr;  /\*存放处理结果的辅助字符串\*/

    String::iterator iter = str.begin();  /\*游标对象\*/

    while (iter != str.end()) {

        if (\*iter == '+') {  /\*去除单目正号\*/

            if (iter != str.begin() && \*(iter - 1) != '(')  /\*若该正号不在表达式开头且前面不是左括号，则是双目正号，需放进结果字符串；否则忽略\*/

                ostr += \*iter;

        }

        else if (\*iter != ' ') {  /\*去除空格\*/

            ostr += \*iter;

        }

        iter++;

    }

    iter = ostr.begin();/\*准备进行第二次扫描\*/

    while (iter != ostr.end()) {

        if (\*iter == '-') {  /\*标记单目负号\*/

            /\*若该负号在表达式开头，根据后面的字符做对应更改\*/

            if (iter == ostr.begin()) {

                if (isDigit(\*(iter + 1)))

                    \*iter = '$';

                else

                    \*iter = '@';

            }

            /\*若该负号不在表达式开头，则当它前面不是数字、不是运算符、不是右括号时，它就是单目负号。根据后面的字符做对应更改\*/

            else if (!isDigit(\*(iter - 1)) && !isOp(\*(iter - 1)) && \*(iter - 1) != ')') {

                if (isDigit(\*(iter + 1)))

                    \*iter = '$';

                else

                    \*iter = '@';

            }

        }

        iter++;

    }

    ostr += '#';  /\*添加结束符号\*/

    \*(ostr.end()) = '\0';  /\*手动添加尾零\*/

    str = ostr;  /\*将结果赋值给原来的字符串\*/

}

·检查表达式是否合法

bool expCheck(const String& str)

{

    if (\*str.last() != '#') {

        cout << "表达式结尾缺少符号#" << endl;

        return false;

    }

    if (\*str.begin() == '#') {

        cout << "表达式为空" << endl;

        return false;

    }

    String::iterator iter = str.begin();  /\*游标对象\*/

    LinkedStack<char> brackets;  /\*记录括号情况的栈\*/

    int opNum = 0, numNum = 0;  /\*操作符（双目）个数，数字个数\*/

    while (iter != str.end()) {

        /\*1、若扫描到字符用来表示数字，则不停扫描，直到扫描到的字符有其他用途为止\*/

        if (isDigit(\*iter)) {

            while (isDigit(\*iter) && iter != str.end()) {

                if (\*iter == '.' && (!isDigit(\*(iter - 1)) || !isDigit(\*(iter + 1)))) {  /\*若小数点前后都不是数字，则非法\*/

                    cout << "数字输入错误" << endl;

                    return false;

                }

                else

                    iter++;

            }

            numNum++;  /\*一个数字扫描完成，进行计数\*/

        }

        if (\*iter == '(')  /\*若遇到左括号，压入栈中\*/

            brackets.push(\*iter);

        else if (\*iter == ')') {  /\*若遇到右括号，栈不为空则弹出一个元素；栈为空则说明右括号多余，非法\*/

            if (!brackets.IsEmpty()) {

                char c;

                brackets.pop(c);

            }

            else {

                cout << "括号不匹配" << endl;

                return false;

            }

        }

        else if (isOp(\*iter))  /\*若遇到双目操作符，进行计数\*/

            opNum++;

        if (iter != str.end())

            iter++;

        else

            break;

    }

    if (!brackets.IsEmpty()) {  /\*若扫描完成栈仍不为空，则左括号多余，非法\*/

        cout << "括号不匹配" << endl;

        return false;

    }

    if (numNum != opNum + 1) {  /\*若双目运算符个数不等于数字个数+1，则不匹配，非法\*/

        cout << "操作符与操作数数量不匹配" << endl;

        return false;

    }

    return true;  /\*所有标准均符合，则表达式合法\*/

}

·将表达式转换为后缀表达式

void postfix(String& src)

{

    LinkedStack<char> stack;  /\*存放运算符的栈\*/

    String::iterator iter = src.begin();  /\*游标对象\*/

    String result;  /\*存放转换结果的字符串\*/

    char ch = '#', ch1, op;  /\*当前字符、栈顶字符、弹出字符\*/

    bool cut = false;  /\*是否添加空格的标志\*/

    stack.push(ch);  /\*将#压入栈中，使栈不为空\*/

    ch = \*iter;  /\*读入第一个字符\*/

    iter++;

    while (!stack.IsEmpty()) {

        /\*1、当前字符用来表示数字\*/

        if (isDigit(ch)) {

            if (ch == '$') {  /\*若为单目负号，则不需要分隔\*/

                cut = false;

                if (result.length())  /\*若当前字符串为空，则仅添加一个$；若不为空，则添加一个空格和$\*/

                    result += " $";

                else

                    result += '$';

                ch = \*iter;  /\*读入下一个字符\*/

                iter++;

            }

            else if (cut) {  /\*若需要分隔，则添加一个空格，标志置为假\*/

                cut = false;

                result += ' ';

            }

            else if (result.last() != result.begin() && !isDigit(\*(result.last())))  /\*若字符串不止一个字符且最后一个字符不用来表示数字，则添加一个空格\*/

                result += ' ';

            result += ch;  /\*添加当前字符\*/

            ch = \*iter;    /\*读入下一个字符\*/

            iter++;

        }

        /\*2、当前字符是括号或运算符\*/

        else {

            if (result.length())  /\*若当前字符串不为空，则分隔标志设为真\*/

                cut = true;

            stack.getTop(ch1);  /\*取栈顶运算符\*/

            if (isp(ch1) < icp(ch)) {  /\*若当前字符（栈外）优先级大于栈顶运算符（栈内）的优先级，则将当前字符压入栈中\*/

                stack.push(ch);

                ch = \*iter;  /\*读入下一个字符\*/

                iter++;

            }

            else if (isp(ch1) > icp(ch)) {  /\*若当前字符（栈外）优先级小于栈顶运算符（栈内）的优先级，则弹出栈顶元素，并添加空格和栈顶元素\*/

                stack.pop(op);

                result += ' ';

                result += op;

            }

            else {  /\*若两者优先级相等，则弹出栈顶元素\*/

                stack.pop(op);

                if (op == '(') {  /\*若栈顶元素是左括号，直接读入下一个字符\*/

                    ch = \*iter;

                    iter++;

                }

            }

        }

    }

    src = result;  /\*将结果赋值给原来的字符串\*/

}

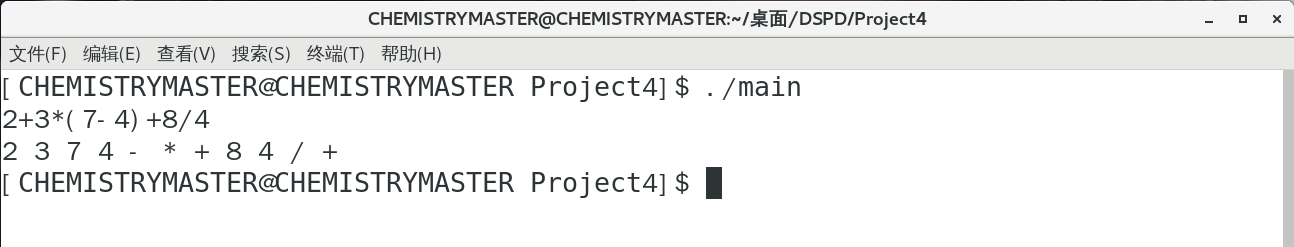
**3.项目测试**

3.1 正常测试6种运算符

输入内容：2+3\*(7-4)+8/4

预期结果：2 3 7 4 - \* + 8 4 / +

测试结果：

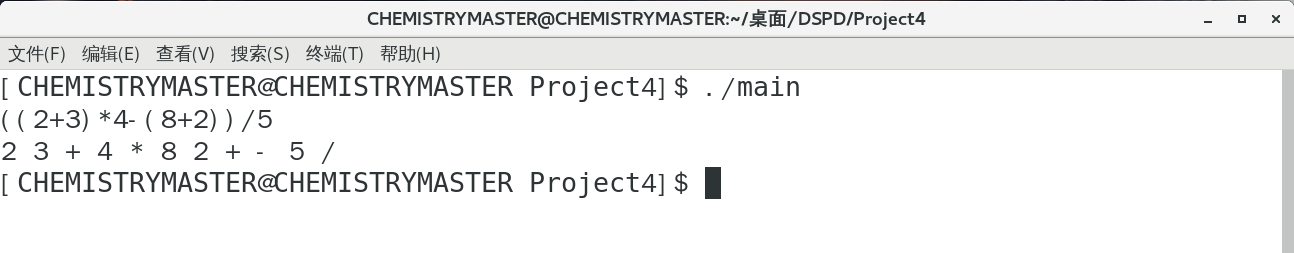


3.2 嵌套括号

输入内容：((2+3)\*4-(8+2))/5

预期结果：2 3 + 4 \* 8 2 + - 5 /

测试结果：

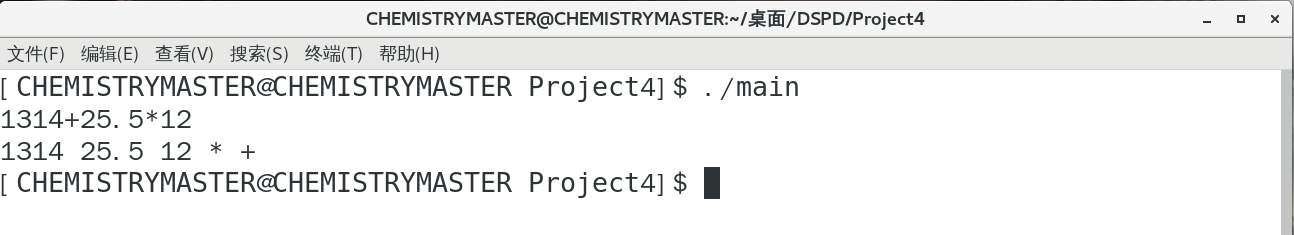


3.3 运算数超过1位整数且有非整数出现

输入内容：1314+25.5\*12

预期结果：1314 25.5 12 \* +

测试结果：

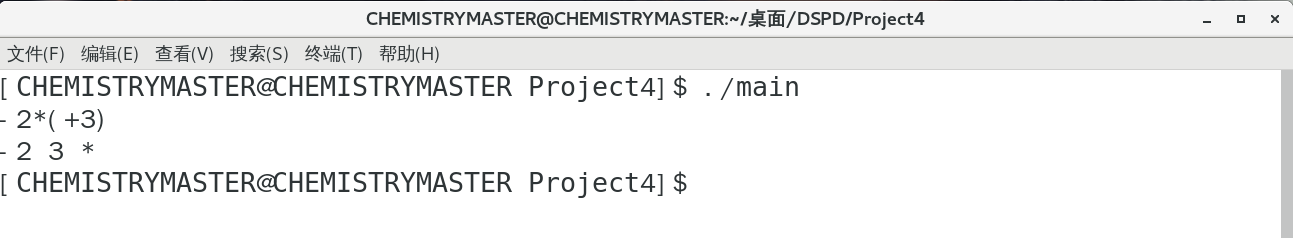


3.4 运算数有正或负号

输入内容：-2\*(+3)

预期结果：-2 3 \*

测试结果：

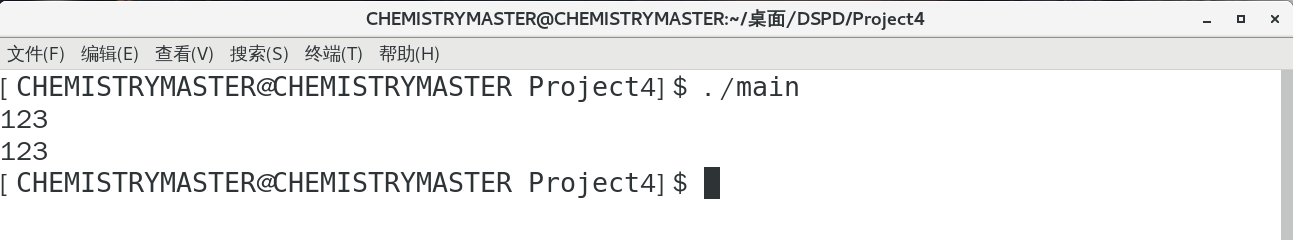


3.5 只有1个数字

输入内容：123

预期结果：123

测试结果：

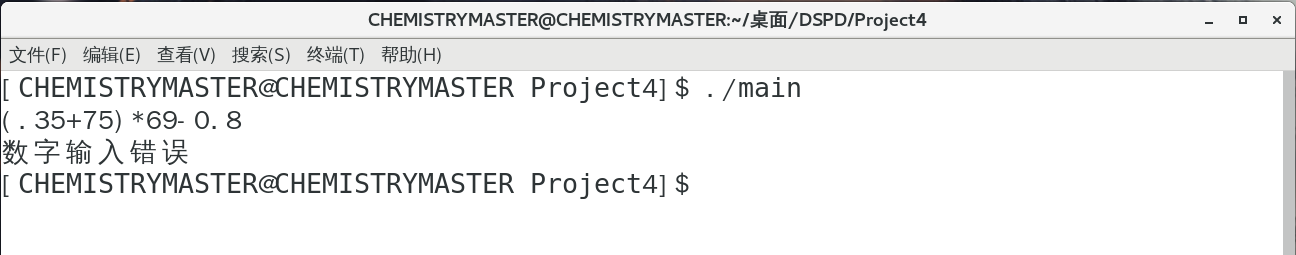


3.6 数字输入错误的情况

输入内容：(.35+75)\*69-0.8

预期结果：数字输入错误

测试结果：

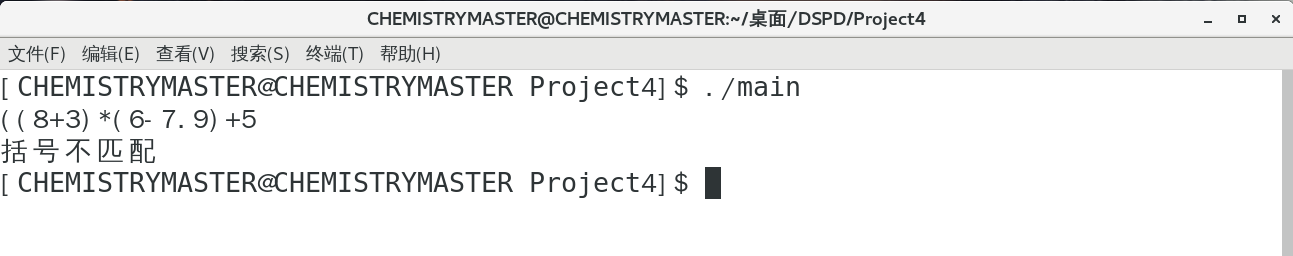


3.7 括号不匹配的情况

输入内容：((8+3)\*(6-7.9)+5

预期结果：括号不匹配

测试结果：

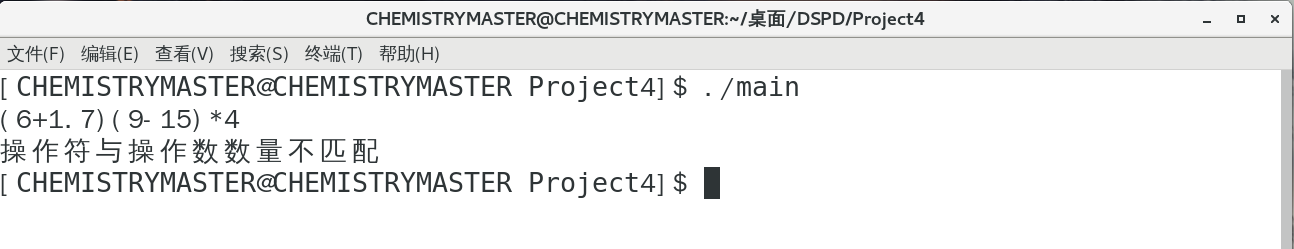


3.8 操作符与操作数数量不匹配的情况

输入内容：(6+1.7)(9-15)\*4

预期结果：操作符与操作数数量不匹配

测试结果：

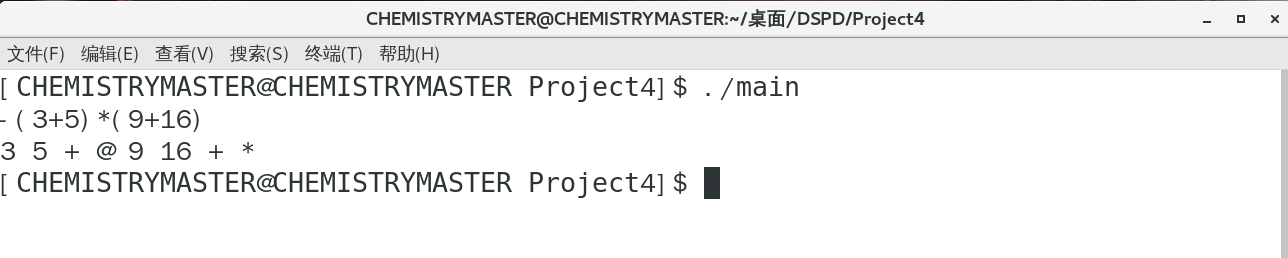


3.9 带单目负号的情况

输入内容：-(3+5)\*(9+16)

预期结果：3 5 + @ 9 16 + \*（单目取负用@来代替）

测试结果：



3.10 输入带空格的情况

输入内容：+(42.195-38.95) + (-6)\*(-88 +99)-40\*(+8)

预期结果：42.195 38.95 - -6 -88 99 + \* + 40 8 \* -

测试结果：

