编译原理实验

实验一：将算术表达式转化为逆波兰表达式

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

1 算法原理与描述

表达式一般由操作数 (Operand)、运算符 (Operator) 组成。逆波兰表达式又称为后缀表达式，是一种特殊的、利用栈来进行运算的数学表达式。不同于把运算符放在两个操作数中间的算数表达式（又称为中缀表达式），逆波兰表达式的运算符位于操作数之后。逆波兰表达式的计算只需要压栈和出栈两种操作，在计算时其从左到右顺序读取，如果当前字符为变量或者为数字，则压栈；如果是运算符，则将栈顶的两个元素弹出，进行相应运算后将结果压栈。当表达式扫描完成后，栈里剩余的值即为结果。

将算数表达式转化为逆波兰表达式的方法有很多种，其中一种主流算法是使用栈和队列进行处理。假设运算符优先级为 $\{*\} = \{/\} > \{+\} = \{-\}$，那么算法描述如下：

1. 初始状态下栈和队列为空，从左到右扫描算数表达式：
   (a) 如果当前元素是操作数，直接输入队列；
   (b) 如果当前元素是运算符：
      i. 若当前元素为 '('，则将该元素入栈；
      ii. 若栈为空，则将该元素入栈；
      iii. 若当前元素优先级大于栈顶元素，则将该元素入栈；
      iv. 若当前元素为 ')'，则依次出栈入列，直到匹配到第一个 '(' 为止。此后当前元素直接舍弃，'(' 直接出栈舍弃；
      v. 若当前元素优先级不大于栈顶元素，则依次出栈入列，直到当前元素优先级大于栈顶元素。此后当前元素入栈；
2. 当算数表达式扫描完毕后栈中仍有运算符时，依次出栈入列，直到栈为空；
3. 此时队列输出即为逆波兰表达式。

2 算法流程图

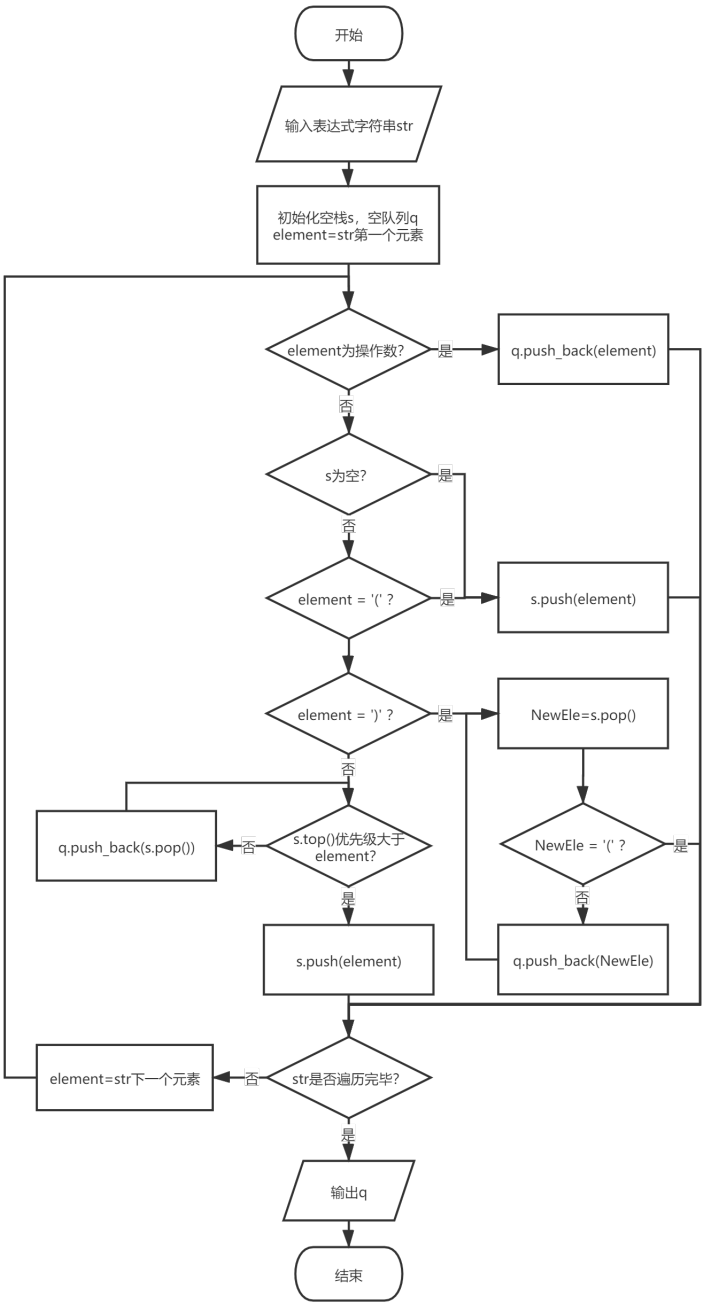

图 1: 算法流程图

3 关键实现代码

```
// 判断字符c是否是操作符并返回优先级
int GetPriority(char c) {
    switch (c) {
        case '(':
            return 1;
            break;
        case '+':
```

```
        case '-':
            return 2;
            break;
        case '*':
        case '/':
            return 3;
            break;
        case ')':
            return 4;
            break;
        default:// 字符c为非操作符时返回-1
            return -1;
            break;
    }
}
// 将表达式字符串exp转换为后缀表达式vector
vector<string> GetPostfixExpression(string exp) {
    vector<string> output;
    string temp;
    stack<char> s;
    for (int i=0; i<exp.length(); i++) {
        // 当前字符为非操作符时
        if (GetPriority(exp[i]) == -1) {
            // 获取操作数
            string temp;
            while(i<exp.length() && GetPriority(exp[i])==-1) {
                temp.push_back(exp[i]);
                i++;
            }
            // 操作数压栈
            output.push_back(temp);
            i-=1;
        }
        // 当前字符为操作符时
        else {
            // 栈空，压栈
            if (s.empty())
                s.push(exp[i]);
            //操作符为'('，压栈
            else if (exp[i]=='(')
                s.push(exp[i]);
            // 操作符为')'，弹栈
            else if (exp[i]==')') {
                while (!s.empty()) {
                    if (s.top()=='(') {
                        // 栈顶为'('时结束
                        s.pop();
                        break;
```

```
                    }
                    string temp(1, s.top());
                    output.push_back(temp);
                    s.pop();
                }
            }
            else {
                // 当前操作符优先级大于栈顶元素，压栈
                if (GetPriority(s.top()) < GetPriority(exp[i]))
                    s.push(exp[i]);
                // 否则弹栈
                else {
                    while (!s.empty() && GetPriority(s.top())>=GetPriority(exp[i])) {
                        string temp(1, s.top());
                        output.push_back(temp);
                        s.pop();
                    }
                    s.push(exp[i]);
                }
            }
        }
    }
    // 扫描字符串结束，清空栈
    while (!s.empty()) {
        string temp(1, s.top());
        output.push_back(temp);
        s.pop();
    }
    return output;
}
```

4 实验结果

运行结果如图2、图3和图4所示，可见程序运行正确。

图 2: 实验结果 1

图 3: 实验结果 2

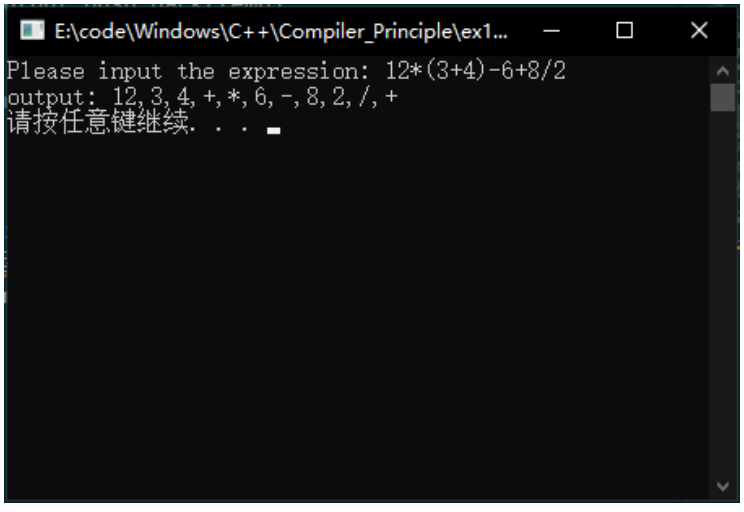

图 4: 实验结果 3