**数据结构课程设计《表达式计算》项目说明文档**

2253893 苗君文 软件工程

**1. 项目简介**

**1.1. 项目背景**

表达式求值是程序设计语言编译中的一个最基本问题，就是将一个表达式转化为逆波兰表达式并求值。具体要求是以字符序列的形式从终端输入语法正确的，不含变量的整数表达式，并利用给定的优先关系实现对算术四则混合表达式的求值，并延时在求值过程中运算符栈，操作数栈，输入字符和主要操作变化过程。

要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求值，首先要能正确解释表达式。任何一个表达式都是由操作符，运算符和界限符组成，我们称它们为单词。一般来说，操作数既可以是常数，又可以是被说明为变量或常量的标识符；运算符可以分成算术运算符，关系运算符和逻辑运算符3类；基本界限符有左右括号和表达式结束符等。为了叙述的简洁，我们仅仅讨论简单算术表达式的求值问题。这种表达式只包括加，减，乘，除4种运算符。

人们在书写表达式时通常采用的是“中缀”表达形式，也就是将运算符放在两个操作数中间，用这种“中缀”形式表示的表达式称为中缀表达式。但是，这种表达式表示形式对计算机处理来说是不大合适的。对于表达式的表示还有另一种形式，称之为“后缀表达式“，也就是将运算符紧跟在两个操作书的后面。这种表达式比较合适计算机的处理方式，因此要用计算机来处理，计算表达式的问题，首先要将中缀表达式转化成后缀表达式，又称为逆波兰表达式。

**1.2. 项目要求**

为了实现表达式求值，本项目要求首先读入表达式（包括括号）并创建对应二叉树，其次对二叉树进行前序遍历，中序遍历，后续遍历，输出对应的逆波兰式，中序表达式和波兰表达式。

**1.3. 输入格式**

本项目要求用户首先输入表达式，要求表达式中的数值只能是一位数，且表达式逻辑正确，否则需要重新输入，直到输入合法的表达式，完成后，用户需要输入以确定是否继续运行程序。

**1.4. 输出格式**

本项目针对输入有详细的输入错误处理，若用户输入错误，则程序会输出相关提示信息，并要求重新输入。获得正确的表达式以后，程序会依次输出中缀表达式、波兰表达式和逆波兰表达式，并询问用户是否需要继续运行程序。

**2. 设计思路**

**2.1. 总体设计思路**

中缀表达式就是用户一般看得懂且经常使用的表达式。其特征是运算符在两个操作数之间，方便理解与输出。本项目的难点在于如何用中缀表达式转换为波兰表达式和逆波兰表达式，首先需要明晰如何构造这两种表达式。

这三种表达式的主要区别在于运算符的位置和如何解析表达式。中缀表达式通常需要使用括号来明确运算顺序，而波兰表达式和逆波兰表达式则通过运算符的位置来明确运算顺序，不需要括号。

转换中缀表达式为前缀（波兰表达式）和后缀表达式（逆波兰表达式）通常使用栈的数据结构，通过栈来保存运算符并确保正确的运算顺序。

**中缀表达式转前缀表达式（波兰表达式）：**

1. 从右到左遍历中缀表达式。

2. 遇到操作数，直接输出到结果。

3. 遇到运算符，将其压入栈中。如果遇到优先级更低或相等的运算符，就将栈中的运算符弹出，输出到结果，直到遇到优先级更低的运算符或栈为空，然后将当前运算符压入栈。

4. 遇到左括号时，将右括号之前的运算符全部弹出栈。

最终，输出的结果即为前缀表达式。

**中缀表达式转后缀表达式（逆波兰表达式）：**

1. 从左到右遍历中缀表达式。

2. 遇到操作数，直接输出到结果。

3. 遇到运算符，将其压入栈中。如果遇到优先级更低或相等的运算符，就将栈中的运算符弹出，输出到结果，直到遇到优先级更低的运算符或栈为空，然后将当前运算符压入栈。

4. 遇到右括号时，将左括号之前的运算符全部弹出栈。

最终，输出的结果即为后缀表达式。

通过这种方式，就可以将中缀表达式转换为不同形式的表达式，使得计算机更容易处理和计算。

**2.2. 数据结构设计**

为实现本项目的功能，选择使用栈和树的数据结构。

使用栈的原因如下：在算术表达式中，括号的匹配是其中重要的步骤。栈的先进后出（FILO）特性使其方便处理括号的匹配。通过在栈中将左括号入栈，遇到右括号则出栈，从而可以确保表达式的合法性。

在构建表达式树之前，通常需要将中缀表达式转换为后缀表达式，使用栈来管理运算符，根据运算符的优先级进行推送和弹出，可以有效地实现这一转换。在构建表达式树时，使用两个栈，一个用于值栈，另一个用于符号栈。这种栈的结构提供了一种递归结构，与树的构建相符合。

由于需要生成中缀表达式，后缀表达式，前缀表达式，最方便使用的结构就是树，因为树可以方便地实现前序、中序、后序的遍历从而形成三种表达式。树的层次结构可以清晰地反映表达式中的操作和运算符的关系。通过表达式树，可以更容易地进行 各种表达式的操作，如求值、输出不同形式的表达式等。

通过将树的操作封装到BinaryTree类中，可以使代码更加模块化，易于理解和维护。在表达式中，操作树和运算符的组合可以通过树的分支和叶子结点来表示，有助于实现表达式的构建和处理。

**2.3. 类设计**

**2.3.1. StackNode 类（栈结点类）**

成员变量：

* data: 存储节点数据。
* link: 指向下一个节点的指针。

成员函数：

* 构造函数:StackNode(Type d = 0, StackNode<Type>\* l = NULL): 初始化节点，可以指定数据和下一个节点。

**2.3.2. Stack类（栈类）**

成员变量：

* top：指向栈顶的指针。

成员函数：

* Push(const Type& item): 将元素压入栈顶。
* Pop(): 弹出栈顶元素并返回其值。
* GetTop() const: 返回栈顶元素的值。
* MakeEmpty(): 清空栈。
* IsEmpty() const: 检查栈是否为空。

2.3.3. BinTreeNode类（树的结点类）

成员变量：

* leftChild: 左子节点指针。
* rightChild: 右子节点指针。
* data: 节点数据。

成员函数：

* 构造函数BinTreeNode: 初始化节点，可以指定数据、左子节点和右子节点。
* GetData() const: 返回节点数据。
* GetLeft() const: 返回左子节点指针。
* GetRight() const: 返回右子节点指针。
* SetData(const Type& item): 设置节点数据。
* SetLeft(BinTreeNode<Type>\* L): 设置左子节点指针。
* SetRight(BinTreeNode<Type>\* R): 设置右子节点指针。

**2.3.4. BinaryTree 类（二叉树类）**

成员变量:

* root: 指向树根的指针。
* RefValue: 用于比较的参考值。

成员函数：

* Insert(const Type& item): 插入元素到二叉树。
* Find(const Type& item) const: 在二叉树中查找元素。
* Destroy(BinTreeNode<Type>\* current): 递归销毁二叉树。
* GetPrecedence(char op) const: 获取运算符的优先级。
* IsValidExpression(const char\* expression) const: 验证算术表达式的格式。
* BuildExpressionTree(const char\* expression): 从中缀表达式构建表达式树。
* PrintExpressions() const: 打印表达式树的不同形式。
* PreOrderTraversal(BinTreeNode<Type>\* current, ostream& out) const: 先序遍历。
* InOrderTraversal(BinTreeNode<Type>\* current, ostream& out) const: 中序遍历。
* PostOrderTraversal(BinTreeNode<Type>\* current, ostream& out) const: 后序遍历。

Stack和StackNode构成了一个基本的栈数据结构，用于处理括号匹配和中缀表达式转后缀表达式的操作。BinTreeNode作为二叉树节点类，包含数据和指向左右子节点的指针。BinaryTree则包含根节点指针，以及与二叉树相关的各种操作，如插入、查找、删除、构建表达式树等。通过这些类的设计，程序实现了一个基于栈和二叉树的算术表达式处理系统，提高了代码的可读性和可维护性。

**2.4. 主程序设计**

使用while循环让用户可以多次输入和处理表达式，在循环中，创建一个BinaryTree对象用于处理算术表达式。让用户输入表达式，并用IsValidExpression函数判断表达式是否合法，不合法则需要重新输入。合法则调用BuildExpressionTree函数构建表达式，而后调用PrintExpressions函数输出中缀、后缀和前缀形式的表达式，最后让用户选择是否继续运行程序。

**2.5. 输入错误处理设计**

**2.5.1. 单个参数的输入错误处理**

在让用户输入以选择是否继续运行程序时，使用 while 循环不断尝试获取用户输入，直到输入满足要求。使用 cin.fail() 及y/Y/n/N来判断输入是否出错，检查输入是否在有效范围内。如果输入无效，则输出错误信息，清除输入缓冲区并忽略之后的字符，并重新输出输入的提示信息。通过一个字符来获取输入一个数之后的字符以检查输入字符的个数，若不正确，也输出错误信息，并重新输出输入的提示信息，执行相应清除操作。如果输入有效，则跳出循环。

**2.5.2. 判断表达式是否合法的输入错误处理**

为判断用户输入的表达式是否需要重新输入，使用了IsValidExpression来判断。此函数确保了左右括号的使用是匹配的，即每个左括号都有相应的右括号，嵌套关系是正确的；表达式中所有的字符都只能是数字字符，括号或加减乘除；首字符不可以是乘除符号；尾字符只能是右括号或数字字符；右括号后只能是运算符，除非作为最后一个字符；左括号前只能是运算符，除非作为第一个字符；不可以有多个数字字符连续，不满足以上规则都会返回false，要求用户重新输入。

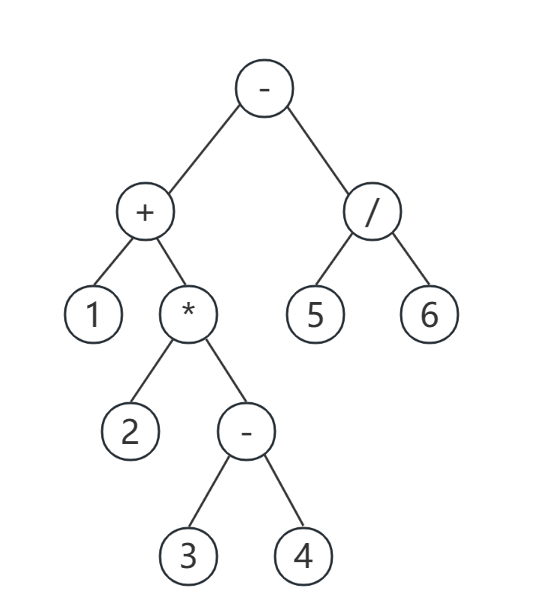
**3. 功能实现**

**3.1. BuildExpressionTree函数**

此函数根据给定的中缀表达式构建一个二叉表达式树。这个树的节点包含操作数和运算符，通过树的结构反映了表达式中的运算次序。

首先，创建两个栈，一个用于存储操作数和表达式树节点，另一个用于存储运算符。同时，定义运算符的优先级。然后，对中缀表达式进行遍历，处理每个字符。如果遇到数字字符，创建一个包含该数字的表达式树节点，并将节点推入树节点栈。如果遇到运算符，进行以下操作：如果运算符栈为空，或者当前运算符的优先级大于栈顶运算符的优先级，将当前运算符推入运算符栈。否则，从运算符栈弹出运算符，同时从树节点栈弹出两个节点，构建一个新的表达式树节点，将新节点推入树节点栈。重复这一过程，直到满足上述条件。遍历结束后，将运算符栈中剩余的运算符依次弹出，同时从树节点栈中弹出两个节点，构建新的表达式树节点，将新节点推入树节点栈。树节点栈中最终的根节点即为构建好的表达式树。

例如，对于中缀表达式“1+2\*(3-4)-5/6”用BuildExpressionTree函数构建出的表达式树如下：



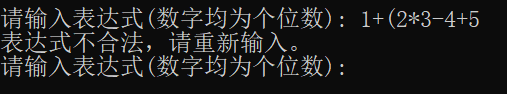
**3.2. PrintExpressions函数**

该函数可以打印中缀表达式，波兰表达式，逆波兰表达式。调用InOrderTraversal函数，使用中序遍历方式访问表达式树节点，同时在访问每个节点时打印节点的值,打印中缀表达式。类似地，用PostOrderTraversal，PreOrderTraversal函数分别进行后序、前序遍历从而打印逆波兰表达式与波兰表达式。

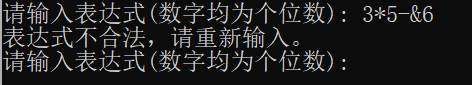
**4. 测试结果（包含边界测试）**

**4.1. 输入表达式非法情况**

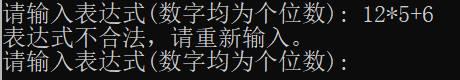
（1）输入的表达式括号不匹配



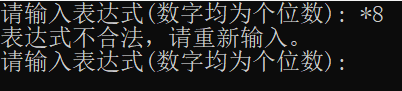
（2）输入的表达式有非法字符



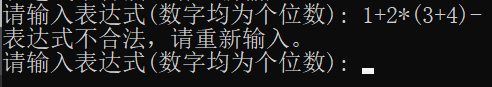
（3）输入的表达式中数字存在非个位数数字



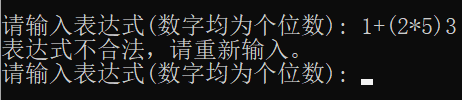
（4）输入的表达式中第一个字符为\*或/



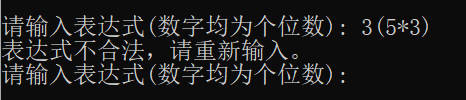
（5）输入的表达式中最后一个字符不为合法字符（即数字、右括号）



（6）输入的表达式中右括号后不为运算符

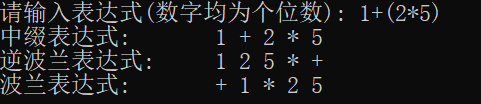


（7）输入的表达式中左括号前不为运算符

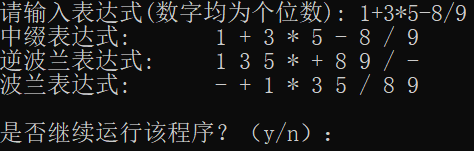


**4.2. 输入的表达式正确的情况**

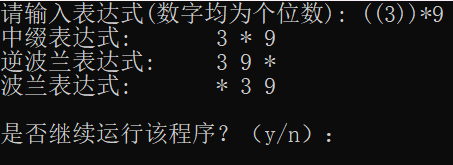
（1）有一组括号



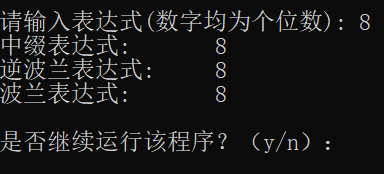
（2）无括号



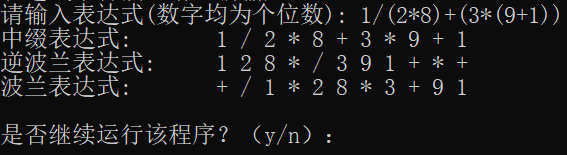
（3）括号嵌套



（4）单个数字

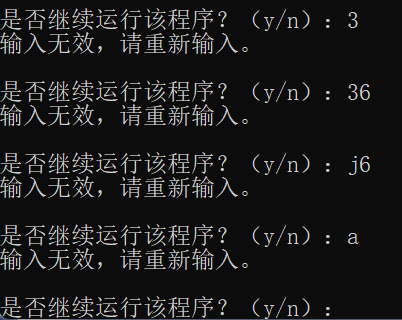


（5）多组括号同时存在

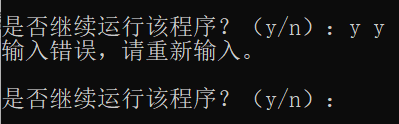


**4.3. 选择是否继续运行程序**

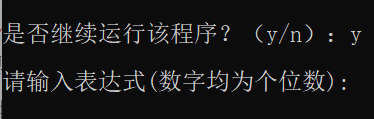
（1）输入的内容错误（多种情况，具体如下）



（2）输入的个数错误



（2）输入y/Y/n/N均输入正确（下图以y为例）



**4.4. 程序测试结果总览（此处仅展示输入正确有效的情况）**

