Project4 实验报告

22336044 陈圳煌

1. 程序功能简要说明

输入正确的前缀表达式，程序将前缀表达式作为先序遍历结果，构建二叉树，以运算符号为根节点，字母（a~z），数字（1~9）作为叶子节点。【每个非叶子节点必须要有子树】可以输出二叉树的中序遍历结果，即中缀表达式；可以对表达式中的字母变量赋值（相同字母只需要赋值一次），可以计算表达式结果（如果存在未赋值的结果则视为 0）；可以再添加一个表达式，并对两个表达式进行合并；可以输出二叉树的树形图。

2. 程序运行及部分代码说明

构建二叉树:

```
//利用前缀表达式（先序遍历的树）构建二叉树
tree *buildtree(string s,int &pos){
    if(pos>=s.length()){
        return NULL;
    }
    char ch=s[pos];
    pos++;
    if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^'){
        tree *p=new tree(ch);
        p->lchild=buildtree(s,pos);
        p->rchild=buildtree(s,pos);
        return p;
    }else if(ch>='a' && ch<='z'){
        tree *p=new tree(ch);
        return p;
    }else{
        tree *p=new tree(ch);
        return p;
    }
}
```

输出中缀表达式:

```
//输出构建出的树的中缀表达式
void printin(tree *root,bool isneed){
    if(root==NULL){
        return;
    }
    if(root->data=='+' ||root->data=='-' ){
        if(isneed==1){
            cout << "(" ;
        }
        printin(root->lchild,1);
        cout << root->data;
        printin(root->rchild,1);
        if(isneed==1){
            cout << ")" ;
        }
    }else if(root->data=='*' ||root->data=='/' ||root->data=='^'){
        printin(root->lchild,1);
        cout << root->data;
        printin(root->rchild,1);
    }else{
        cout << root->data ;
    }
}
```

给变量赋值:

```
//给变量赋值，如果存在多个相同变量，只需要赋值一次
void giveval(tree* root,char pa,int val){
    if(root==NULL){
        return;
    }
    if(root->data==pa){
        root->data=val+'0';
    }
    giveval(root->lchild,pa,val);
    giveval(root->rchild,pa,val);
}
```

求出表达式的结果，共分为四个模块:

（1）将二叉树以中序遍历的方式将中缀表达式存储到队列中，如果遇到未赋值的变量，则默认为 0。

```
//求出表达式的值，若存在未赋值的变量，当0处理
//1.先把中缀表达式按符号入栈
void putvector(tree *root,queue<char> &store,bool isneed){
    if(root==NULL){
        return;
    }
    if(root->data=='+' ||root->data=='-' ||root->data=='*' ||root->data=='/' ||root->data=='^'){
        store.push('(');
        putvector(root->lchild,store,1);
        store.push(root->data);
        putvector(root->rchild,store,1);
        store.push(')');
    }else{
        store.push(root->data);
    }
}
```

（2）将数字和运算符进行计算，返回 double 类型。

```
//2.进行数字间的计算
double calculate(double num1,char ope, double num2){
    switch (ope){
        case '+':
            return num1+num2;
        case '-':
            return num1-num2;
        case '*':
            return num1*num2;
        case '/':
            if(num2==0){
                return 0;
            }else{
                return num1*1.0/num2;
            }
        case '^':
            return pow(num1,num2);
        default:
            return 0;
    }
}
```

（3）判断运算符的优先级：

```
//3.判断符号的优先级
int get_pri(char op){
    switch(op){
        case '+':
            return 1;
        case '-':
            return 1;
        case '*':
            return 2;
        case '/':
            return 2;
        case '^':
            return 3;
        case '#':
            return 0;
    }
}
```

（4）计算：（细节如注释所示）

```
//4.计算
double printval(queue<char> &store){
    if(store.empty()){//如果存储中缀表达式的队列为空则返回0
        return 0;
    }
    store.push('#');//在队列最后插入" #" 作为结束标志
    stack<double> int_sta;//数字栈
    stack<char> op_sta;//运算符栈
    op_sta.push('#');//在运算符栈栈底插入" #" 用于匹配运算
    for(1;!store.empty();){
        if(store.front()>='0' && store.front()<='9'){//如果判断队列中为数字，存入数字栈
            double num=store.front()-'0';
            store.pop();
            int_sta.push(num);
        }else if(store.front()>='a' && store.front()<='z'){//如果队列中有未赋值的字母变量，赋值为0存入数字栈
            store.pop();
            int_sta.push(0);
        }else{//运算过程伴随着符号插入进行
            if(store.front()=='('){//碰到" （" 则忽略
                op_sta.push('(');
                store.pop();
            }else if(store.front()=='+' || store.front()=='-' || store.front()=='*' || store.front()=='/' || store.front()=='^'){
                if(op_sta.top()=='(' || op_sta.top()=='#'){//碰到运算符分类讨论计算，如果栈顶为括号或#则直接存入
                    op_sta.push(store.front());
                    store.pop();
                }else{//否则判断当前运算符和栈顶运算符优先级
                    int pri1=get_pri(store.front());
                    int pri2=get_pri(op_sta.top());
                    if(pri1<=pri2){//进行计算的情况 ：取出两个数字，和运算符输入函数进行计算
                        char top_op=op_sta.top();
                        op_sta.pop();
                        double num2=int_sta.top();
                        int_sta.pop();
                        double num1=int_sta.top();
                        int_sta.pop();
                        double res1=calculate(num1,top_op,num2);
                        int_sta.push(res1);
                    }else{
                        op_sta.push(store.front());
                        store.pop();
                    }
                }
            }else if(store.front()==')'){//碰到右括号，先寻找左括号 ，若是，不计算去掉括号
                if(op_sta.top()=='('){
                    op_sta.pop();
                    store.pop();
                }else{//若不是，计算括号内内容
                    char top_op=op_sta.top();
                    op_sta.pop();
                    double num2=int_sta.top();
                    int_sta.pop();
                    double num1=int_sta.top();
                    int_sta.pop();
                    double res1=calculate(num1,top_op,num2);
                    int_sta.push(res1);
                }
            }else if(store.front()=='#'){//碰到最后一个字符，判断是否结束
                if(op_sta.top()=='#'){
                    return int_sta.top();
                }else{
                    while(op_sta.top()!='#'){
                        char top_op=op_sta.top();
                        op_sta.pop();
                        double num2=int_sta.top();
                        int_sta.pop();
                        double num1=int_sta.top();
                        int_sta.pop();
                        double res1=calculate(num1,top_op,num2);
                        int_sta.push(res1);
                    }
                    return int_sta.top();
                }
```

合并两个表达式：

首先先输入新的表达式和要进行的合并操作（+，-，*，/，^），将原有树和新的树作为左右子树，符号作为根节点合并两棵树。

```
//将两个表达式合并
tree* combination(tree* root, char opee, tree* root2){
    tree *p=new tree(opee);
    p->lchild=root;
    p->rchild=root2;
    return p;
}
```

输出二叉树的可视树形:

首先计算数的深度:

```
//计算二叉树的深度
int judgedepth(tree *root,int &depth){
    if(root==NULL){
        return 0;
    }
    depth++;
    int ldep=judgedepth(root->lchild,depth);
    int rdep=judgedepth(root->rchild,depth);
    return ((ldep>rdep)?ldep:rdep)+1;
}
```

再根据深度设置缩进空格数打印二叉树（横向）:

```
//输出二叉树的可视化树形
void printtree(tree *root,int depth){
    if(root==NULL){
        return;
    }
    printtree(root->lchild,depth-1);
    for(int i=0;i<depth;i++){
        cout << "    " ;
    }
    cout << root->data << endl;
    printtree(root->rchild,depth-1);
}
```

3. 测试案例

均以表达式-+a*b-cd/ef 为例，

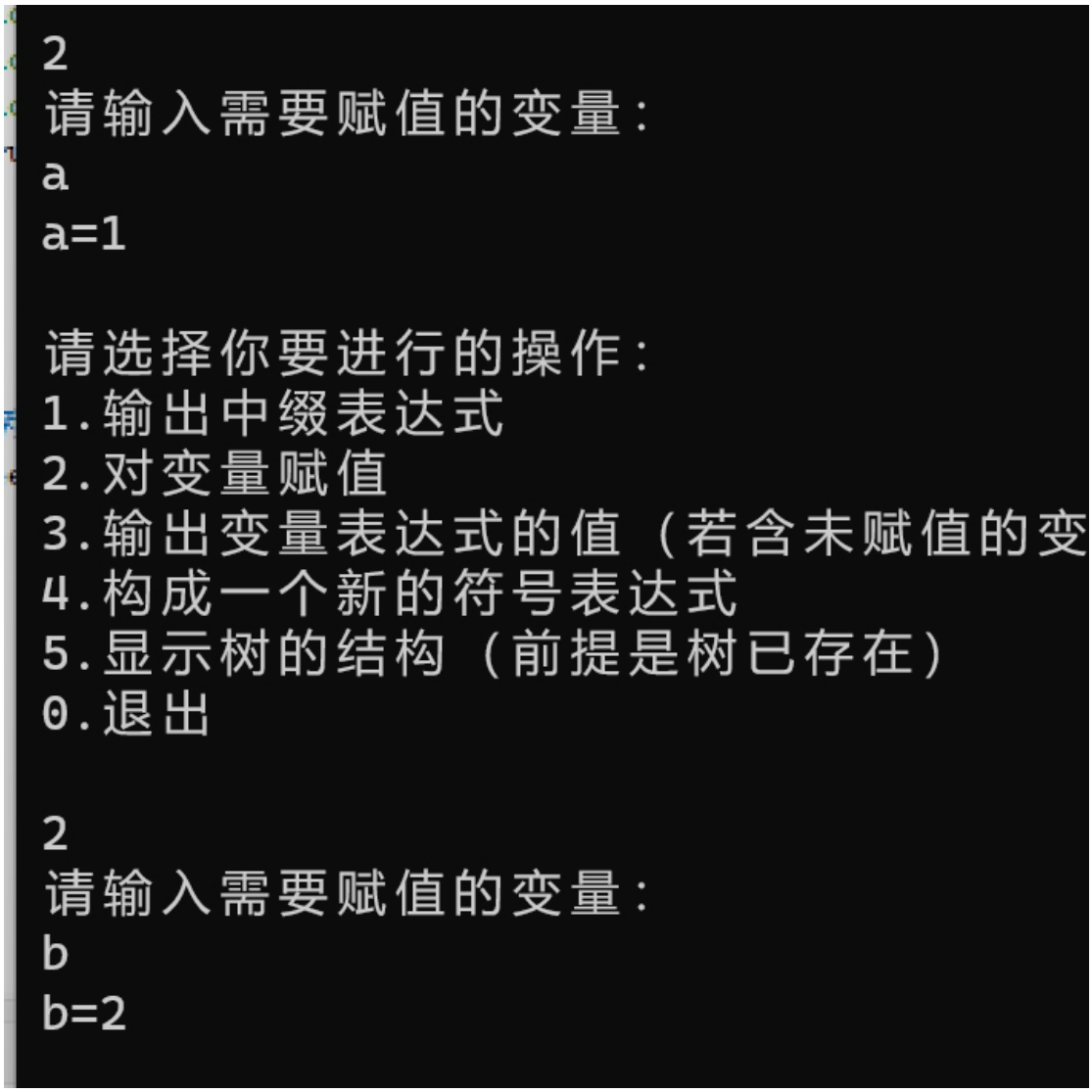
输出中缀表达式:

```
请输入前缀表达式：（请确保表达式合理有效,不含空格）
-+a*b-cd/ef

请选择你要进行的操作：
1.输出中缀表达式
2.对变量赋值
3.输出变量表达式的值（若含未赋值的变量，一律当作0）
4.构成一个新的符号表达式
5.显示树的结构（前提是树已存在）
0.退出

1
(a+b*(c-d))-e/f
```

对变量赋值：a=1,b=2c=3,d=3,e=5,f=2



赋值后的表达式为：

```
1
(1+2*(3-3))-5/2
```

计算表达式的值:

```
1
(1+2*(3-3))-5/2
请选择你要进行的操作:
1.输出中缀表达式
2.对变量赋值
3.输出变量表达式的值（若含未赋值的变量，一律
4.构成一个新的符号表达式
5.显示树的结构（前提是树已存在）
0.退出

3
表达式的值为: -1.5
```

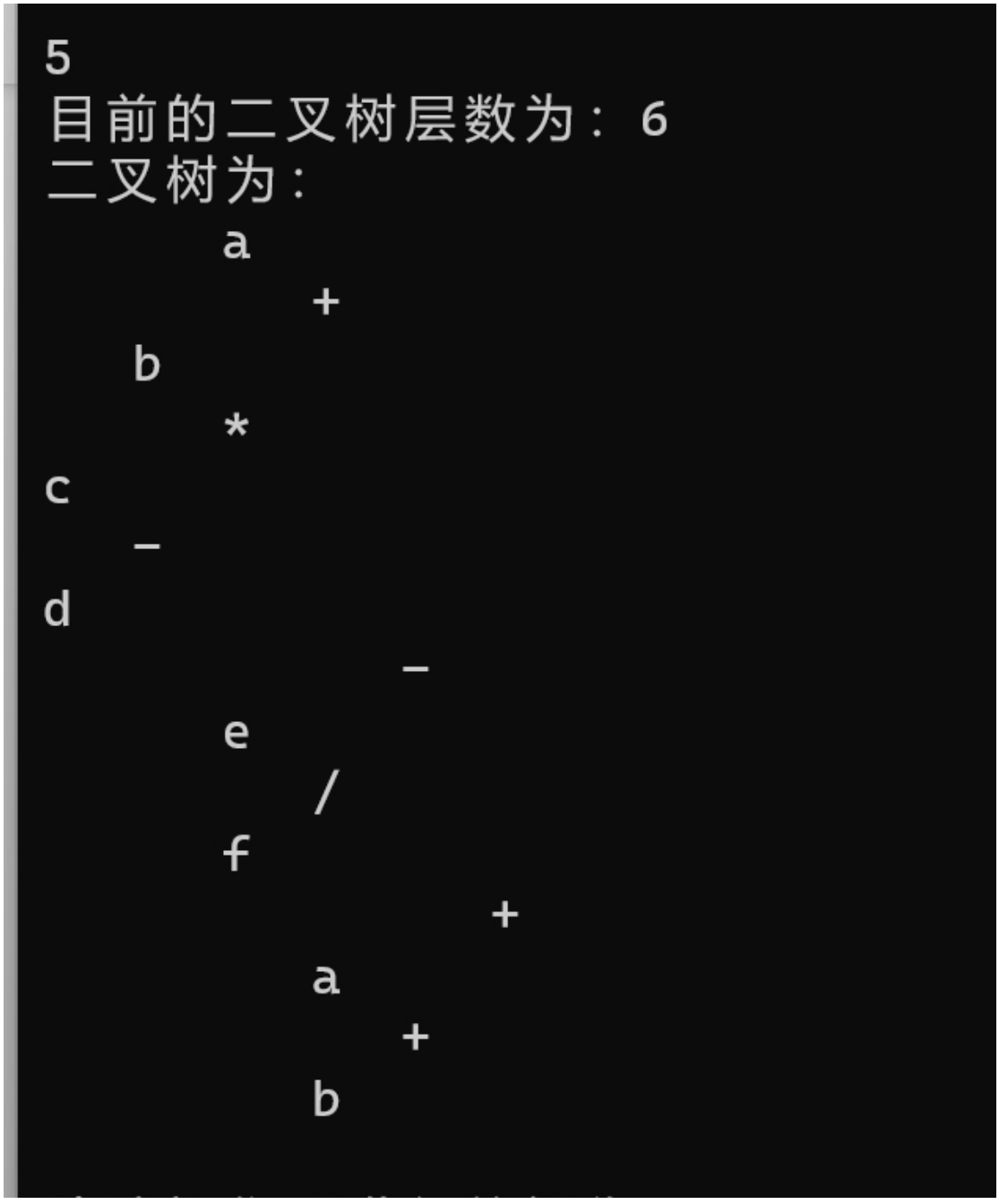
合并表达式:【加上（a+b)】

```
4
请输入要添加的表达式:
+ab
要进行的操作:
+

请选择你要进行的操作:
1.输出中缀表达式
2.对变量赋值
3.输出变量表达式的值（若含未赋值的变量，一律当
4.构成一个新的符号表达式
5.显示树的结构（前提是树已存在）
0.退出

1
((a+b*(c-d))-e/f)+(a+b)
```

输出二叉树的可视化树形图（横向）：



经检验，以上结果均与理论相符合。

4. 程序运行方式

首先输入一个正确的前缀表达式，例如-+a*b-cd/ef。

之后可以进行操作选择：

```
cout << "请选择你要进行的操作: " << endl;
cout << "1.输出中缀表达式" << endl;
cout << "2.对变量赋值" << endl;
cout << "3.输出变量表达式的值（若含未赋值的变量，一律当作0）" << endl;
cout << "4.构成一个新的符号表达式" << endl;
cout << "5.显示树的结构（前提是树已存在）" << endl;
cout << "0.退出" << endl << endl;
```

当对变量赋值后表达式的字母会替换成相应数字，输入 0 结束程序。