程序设计实训报告—
表达式求值问题

完成者：何炜

班级：计科 1501

学号：2015014278

完成日期：2016 年 7 月 14 日星期四

目录

一、 题目的内容及要求

求解形如（a+b)*((c+d)*e+f*h*g)的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。

为了实现表达式求值，可以首先读入原表达式（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（非递归），并输出逆波兰表达式，最后求解原表达式的值，同时对非法表达式格式能予以判断。

用二叉树的结构来存储表达式，后续遍历二叉树即可得到逆波兰表达式

二、 需求分析

本程序能解决形如（a+b)*((c+d)*e+f*h*g)并*以'#'作为结束标志*的简单算术表达式的求值问题。

不仅能够求解出多位浮点数，而且能够对简单的非法表达式进行判断以避免程序异常退出。

三、 概要设计

1. 用户输入中缀表达式
2. 程序将中缀表达式用二叉树的链式存储结构存储下来
3. 前序、中序遍历这颗二叉树，输出对应的前缀、中缀表达式
4. 后续遍历（非递归）这颗二叉树，并把遍历结果存储在顺序栈内，并输出后缀表达式
5. 对后缀表达式进行求值

四、 详细设计

以下对概要设计进行详细的原理分析。

1、 输入中缀表达式，并存在字符型数组里

定义字符串 char str[MAXSIZE];

通过 while(scanf("%s",str)!=EOF)来实现用户自定义结束程序的功能

并通过 strcmp("leave",str)==0 语句实现当用户输入“leave”，程序会正常退出。

2、 以中缀表达式建立一棵二叉树

定义二叉树的结构

```
typedef struct BTNode
 {
    ElemType data[length];
    struct BTNode *lchild;
    struct BTNode *rchild;
 }BTNode;
```

其中 *data[length];*存储中缀表达式中的**操作数**或**操作符**。

整体原理：

i. 已知输入的 str1=（a+b)*((c+d)*e+f*h*g)；定义 BTNode *st[MAXSIZE] **节点栈**，用来存储树的节点。

ii. 再定义一个**操作符栈** char bt[MAXSIZE]存储操作符，去扫描字符串

的每个位置，遇到数字就不做处理直接放到**节点栈**中，遇到**操作符**则将其与**操作符栈顶元素**比较优先级，决定是否送到**节点栈**中，以及决定该节点与其他节点的连接关系。

iii. 最后 BTNode* &b 记录下树的根节点，便得到一颗二叉树了。

其中第一、第二步无需过多解释，以下详细解释第二步：

① 首先通过循环 while(bt[top1]!='#'||*p!='#')来扫描字符串的每个位置的元素

② 如果字符串该位置为数字，就直接建立树的节点，并把节点放进**节点栈**中。这里会遇到两种情况：一种是**多位整数**，一种是**小数**。

- ✓ 多位整形数：遇到第一位数字时，新建树的节点，存到 data 数组里，用 while(In(*p)==0)判断，将后面的几位数字也一起存到这个节点的 data 数组里，作为一个数字串整体。
- ✓ 带小数点的数：遇到第一位数字时，新建树的节点，存到 data 数组里，如果下一位是’ . (小数点)’，则同样用 while(In(*p)==0)判断，把小数点后面的数字连同前面的一起存到 data 数组里，作为一个数字串整体。

代码如下：

```
if(In(*p)==0)//字符串该位置是操作数，则直接建立二叉树节点
{
    s=(BTNode*)malloc(sizeof(BTNode));
     s->lchild=NULL;
     s->rchild=NULL;
    s->data[i]=*p;
    p++;
    i++;
```

```
    while(In(*p)==0)
    {
        s->data[i]=*p;
        p++;
        i++;
        if(*p=='.')//如果是 a.bc 形式，则将 a.bc 一直存在节点中，直至后面不再是操作数为止
        {
            s->data[i]=*p;
            i++;
            p++;
            while(In(*p)==0)
            {
                s->data[i]=*p;
                p++;
                i++;
            }
        }
    }
    s->data[i]='\0';
     top2++;
     st[top2]=s;//把该节点放入节点栈内
}
```

在此之前有一个判断是操作符还是操作数的**判断函数**，代码如下：

```
int In(char c)
{
if(c=='+'||c=='-'||c=='*'||c=='/'||c=='('||c==')'||c=='#')  return  1;//操作符
else return 0;//操作数
}
```

③ 如果是操作符，则比较该操作符与**操作符栈**顶元素的优先级

各符号的优先级表如下：

a\b	+	−	*	/	(	)	#
+	>	>	<	<	<	<	>
−	>	>	<	<	<	<	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	g
)	>	>	>	>	g	>	>
#	<	<	<	<	<	g	=

其中'g'表示 **表达式错误**。

a) 如果当前操作符优先级 > **操作符栈**顶操作符优先级，则让操作符入**操作符栈**，然后检查字符串下一个位置

b) 如果当前操作符优先级 <**操作符栈**顶操作符优先级，以**操作符栈**顶元素建立新的树节点并且取出**节点栈**顶的两个节点作为该节点的左右孩子节点，并把该节点压入**节点栈**。

c) 如果当前操作符优先级 = **操作符栈**顶操作符优先级，此时，栈顶和当前操作符为左右括号，此时，出**操作符栈**顶元素，检查字符串中的下一个元素

d) 如果返回得到 g，即反映了**表达式错误**，令全局变量 error=999，以便在主函数中控制不输出结果。

3、 前序、中序遍历这颗二叉树，并输出前缀、中缀表达式

这两种遍历采用递归方式。这种方式很简单。每次访问节点的时候，把节点存在相应表达式数组里，以便输出。

4、 后序遍历（非递归）二叉树，并把遍历结果存在字符型指针数组里，并输出后缀表达式

非递归后续遍历的思想：

采用一个顺序栈才存储需要返回的节点 BTNode *St[MAXSIZE];，先扫描根节点所有左孩子节点，并把之前经过的节点一一进栈，到了最后，出栈一个节点，即为**父亲节点**，再扫描该节点的右孩子节点的左孩子节点，并把之前经过的节点一一进栈。当一个节点的左右孩子节点均访问后再访问该父亲节点，直至栈空为止。

当访问该节点的时候，就把该节点数值存放到字符型指针数组里，并输出该值。

5、 通过后缀表达式求出表达式的值

因为本程序能解决简单的二目运算符的运算，所以

定义 double opnd1,opnd2 作为运算符的两个元素，

定义 double opnd 作为运算结果。

定义一个**数栈** double St[MAXSIZE]来存放后缀表达式结果。

依次扫描后缀表达式（即 postexp[n]数组）

i. 如果是数字，同上面建立二叉树一样，分为两种：**多位整数**和**小数**，我们通过循环 for(j=0;j<r;j++)（其中 r 为数字串的长度），只要发现有小数点，就让 flag=1；以此分辨两种情况，并定义 k，统计小数点后面的位数，以便在还原小数的值的时候乘以相应的数量级即可！还原数字串原本的数值，再讲数值压入到**数栈**内。

ii. 如果是符号，就把**数栈**内的两个数字取出来（top--）赋值给 opnd1, opnd2.计算出结果，再压入**数栈**。通过 while (i<n)控制循环条件，最后 St[0]即为**表达式的结果**。

五、 源代码

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <math.h>
/*定义区*/
#define MAXSIZE 256
#define length 200
#define ElemType char

int error=0;

 //二叉树的链式存储结构
 typedef struct BTNode
 {
    ElemType data[length];
    struct BTNode *lchild;
    struct BTNode *rchild;
 }BTNode;



//判断是操作符还是操作数
```

```
int In(char c)
{
if(c=='+'||c=='-'||c=='*'||c=='/'||c=='('||c==')'||c=='#') return 1;//操作符

else return 0;//操作数
}

//判断操作符的优先级
char Precede(char a,char b)
{
	char r;
	switch(a)
	{
		case '+':
			case '-':if(b=='*'||b=='/'||b=='(') r='<';else r='>';break;    //加减比乘除和左括号的优先级低
			case '*':
				case'/':if(b=='(')r='<';else r='>';break;//乘除比左括号的优先级低
				case'(':if(b==')')//左右括号的优先级相同
						r='=';
				  else if(b=='#')
				  {
					printf("表达式错误\n");
					r='g';
				  }
				  else r='<';//除此之外，左括号比其他符号优先级低
				  break;

				case')':if(b=='(')
					{
						printf("表达式错误\n");
						r='g';
					}
					else r='>';//右括号比其他符号优先级高
					break;
				case'#':if(b==')')
					{
						printf("表达式错误\n");
						r='g';
					}
					else if(b=='#')r='=';
					else r='<'; //#比其他的符号优先级低
					break;
	}
```

```
    return r;
}

//将中缀表达式建立一棵二叉树
int ExpBTree(BTNode* &b,char *str)
{

   int i;
    BTNode *st[MAXSIZE];//节点栈
    BTNode *s;//新建节点指针
    char bt[MAXSIZE];//操作符栈
    char *p;//字符串
    int top1= -1;//操作符栈指针
    int top2= -1;//节点栈指针
    p=str;
    top1++;
    bt[top1]='#';
    while(bt[top1]!='#'||*p!='#')//当且仅当操作符栈顶元素为'#'且字符串该位置为'#'时退出循环
    {
        i=0;
         if(In(*p)==0)//字符串该位置是操作数，则直接建立二叉树节点
         {
              s=(BTNode*)malloc(sizeof(BTNode));
              s->lchild=NULL;
              s->rchild=NULL;
            s->data[i]=*p;
            p++;
            i++;
            while(In(*p)==0)
            {
                 s->data[i]=*p;
                 p++;
                 i++;
                 if(*p=='.')//如果是 a.bc 形式，则将 a.bc 一直存在节点中，直至后面不再是操作数为止
                 {
                      s->data[i]=*p;
                      i++;
                      p++;
                      while(In(*p)==0)//当下一个位置是操作符，就存储到节点数组中
                      {
                           s->data[i]=*p;
```

```
                    p++;
                    i++;
                }
            }
        }
        s->data[i]='\0';
          top2++;
          st[top2]=s;//把该节点放入节点栈内
    }
    else //字符串该位置是操作符，比较该操作符和操作符栈顶元素的优先级
    {

        switch(Precede(bt[top1],*p))
        {
            case '<':
              top1++;
              bt[top1]=*p;
              p++;
              break;
              //如果当前操作符优先级 > 栈顶操作符优先级，则让操作符入操作符栈
            case '>':
              s=(BTNode*)malloc(sizeof(BTNode));
              s->data[i]=bt[top1];
              i++;
              s->data[i]='\0';
              top1--;
              s->rchild=st[top2];
              top2--;
              s->lchild=st[top2];
              st[top2]=s;
              break;
              //如果当前操作符优先级 < 栈顶操作符优先级，以操作符栈顶元素建立新的树节点并且取出节点栈顶的两个节点作为该节点的左右孩子节点，并把该节点压入节点栈。
          case '=':
              top1--;
              p++;
              break;
              //如果当前操作符优先级 = 栈顶操作符优先级，此时，栈顶和当前操作符为左右括号，此时，出栈顶元素，检查字符串中的下一个元素
          case 'g':
              error=999;
```

```
                        goto loop1;
                }
            }

        }
        loop1:b=st[top2];//树的根节点
        return 1;

}

//前序遍历得到前缀表达式

char *preexp[MAXSIZE];
int n1=0;
void PreOrder(BTNode *b)
{
    if(b!=NULL)
    {
        preexp[n1]=b->data;
        printf("%s ",b->data);
        n1++;
        PreOrder(b->lchild);
        PreOrder(b->rchild);
    }
}


//中序遍历得到前缀表达式

char *inexp[MAXSIZE];
int n2=0;
void InOrder(BTNode *b)
{
    if(b!=NULL)
    {
        InOrder(b->lchild);
        inexp[n2]=b->data;
        printf("%s ",b->data);
        n1++;
        InOrder(b->rchild);
    }
}
```

```
//对二叉树进行后序遍历得到后缀表达书
//后序遍历

char*  postexp[MAXSIZE];//存放后缀表达式,每个节点都是一个字符型数组，故这里使用字符型指针数组
int n3=0;

void PostOrder(BTNode *b)
{
    BTNode *St[MAXSIZE];//保存需要返回节点指针
    BTNode *p;
    int flag,top=-1;
    if(b!=NULL)
    {
        do
        {
            while(b!=NULL)
            {

                top++;
                St[top]=b;
                b=b->lchild;
            }
            p=NULL;
            flag=1;
            while(top!=-1 && flag)
            {
                b=St[top];
                if(b->rchild==p)
                {
                    postexp[n3]=b->data;
                    printf("%s ",b->data);
                    n3++;
                    top--;
                    p=b;
                }
                else
                {
                    b=b->rchild;
                    flag=0;
                }

            }
        }while(top!=-1);
```

```
            printf("\n");
    }
}



  //后缀表达式求值
double CompValue()
{
    int r;//字符串的长度
     double St[MAXSIZE];//数栈
    double  opnd,opnd1,opnd2;//opnd1,opnd2 分别为数栈栈顶的前两个元素，
opnd 为计算结果
     char ch[length];
     int top=-1;
     int i=0;
    int j;
    double sum;//不能写成 int sum，否则当小数点位数过多，会出现错误！
     while (i<n3)
     {
          strcpy(ch,postexp[i]);
        i++;
        if(strcmp(ch,"+")==0)
        {
            opnd1=St[top];top--;
              opnd2=St[top];top--;
              opnd=opnd1+opnd2;
              top++;
              St[top]=opnd;

        }

        else if(strcmp(ch,"-")==0)
        {
            opnd1=St[top];top--;
              opnd2=St[top];top--;
              opnd=opnd2-opnd1;
              top++;
              St[top]=opnd;
        }
        else if(strcmp(ch,"*")==0)
        {
            opnd1=St[top];top--;
              opnd2=St[top];top--;
```

```
        opnd=opnd2*opnd1;
        top++;
        St[top]=opnd;
}
else if(strcmp(ch,"/")==0)
{
    opnd1=St[top];top--;
        opnd2=St[top];top--;
        if(opnd1==0)
        {
            printf("不能除以 0\n");
            error=999;//error 为错误变量，以便最后不输出任何结果
            return 0;
        }
        opnd=opnd2/opnd1;
        top++;
        St[top]=opnd;
}

else
{
    int k;
    int flag=0;//判断是小数还是多位整数、单位整数三种情况
    sum =0;
    r=strlen(ch);
    for(j=0;j<r;j++)
    {
        if(ch[j]=='.')
        {
            k=-1;
            flag=1;
        }
        else
            sum=sum*10+(ch[j]-'0');
        k++;
    }
    top++;
    if(flag==0)
        St[top]=sum;

    else
        St[top]=sum*pow(10,-k);
//  printf("sum=%");
//  printf("还原=%f\n",St[top]);
```

```
            //  printf("%f\n",St[top]);//这个地方如果用%d，就没办法输出浮点数!!!!

            }

        }
        return St[0];
}

double ExpValuel(BTNode *b)
{
      printf("后缀表达式： ");
      PostOrder(b);
       return (CompValue());
}


int main ()
{
      BTNode *B;//根节点的指针
      double re;//结果变量
      char str[MAXSIZE];//中缀表达式字符串
      printf("*************************************************\n");
      printf("              这里是计算器程序，包含以下功能\n");
      printf(" 1.支持二目运算符的多位整数、单个整数、小数混合的加减乘除\n");
      printf(" 2.支持对一些简单的非法表达式的判断\n");
      printf(" 3.你可以在任何时候输入 leave  退出程序\n");
      printf(" 4.所输入必须是英文符号，以#号键结束!! \n");
      printf("*************************************************\n");
      printf("                          现在开始！\n");
      while(scanf("%s",str)!=EOF)
      {
            if(strcmp("leave",str)==0)
            {
                  printf("程序正确退出！\n");
                  break;
            }
            ExpBTree(B,str);//建立二叉树
            if(error!=999)
            {
                  printf("前缀表达式： ");
                  PreOrder(B);
                  printf("\n");
                  printf("中缀表达式： ");
```

```
            InOrder(B);
            printf("\n");
            re=ExpValuel(B);//后序遍历并且计算结果
            printf("运算结果：%.2f\n",re);//输出结果保留两位小数
        }
        n1=0;
        n2=0;
        n3=0;
        error=0;
    }
        return 1;
}
```

六、 运行结果及分析

```
**********************************************************
        这里是计算器程序，包含以下功能
 1.支持二目运算符的多位整数、单个整数、小数混合的加减乘除
 2.支持对一些简单的非法表达式的判断
 3.你可以在任何时候输入leave 退出程序
 4.所输入必须是英文符号，以#号键结束！！
**********************************************************
                现在开始!
3+(6*7)+34-10/2#
前缀表达式：- + + 3 * 6 7 34 / 10 2
中缀表达式：3 + 6 * 7 + 34 - 10 / 2
后缀表达式：3 6 7 * + 34 + 10 2 / -
运算结果：74.00
3+9-8*(4/2)+2#
前缀表达式：+ - + 3 9 * 8 / 4 2 2
中缀表达式：3 + 9 - 8 * 4 / 2 + 2
后缀表达式：3 9 + 8 4 2 / * - 2 +
运算结果：-2.00
9-99999999999#
前缀表达式：- 9 99999999999
中缀表达式：9 - 99999999999
后缀表达式：9 99999999999 -
运算结果：-99999999990.00
```

```
3.1415*2.13+21-90#
前缀表达式：- + * 3.1415 2.13 21 90
中缀表达式：3.1415 * 2.13 + 21 - 90
后缀表达式：3.1415 2.13 * 21 + 90 -
运算结果：-62.31
(43*2#
表达式错误
)43+2#
表达式错误
leave
程序正确退出!
Press any key to continue
```

结果分析：

- ✓ 输入的括号必须为英文括号，而且表达式需要以#结束，否则无法输出正确结果。
- ✓ 该表达式求值程序能够求多位整数、小数单独或者混合加减运算，不仅能够得到正确结果，而且能够输出前缀、中缀、后缀表达式。
- ✓ 该程序能够对简单的**非法表达式**进行进行判断，并且输出**表达式错误**信息而且不输出任何数值。但是对复杂的**非法表达式**不能进行主动判断，只能异常退出。
- ✓ 该程序能够实现主动退出，输入“leave”就会正确退出。

七、 收获及体会

1、 没有完美的程序，一个程序是需要不断修改，不断找出问题并更改的。

这个程序一开始只能算个位数的加减乘除，因为一开始，中缀表达式是以字符串的形式存到内存里的，我们需要对字符串的每个字符进行扫描，如何判断一串连续数字是一个整体，这是一个问题，如何判断这一连串数字时小数，也是一个问题。但是经过思考，其实发现，这个也并不会很难。只需要对下一个字符进行判断即可！后面修订的的程序对非法表达式也不能判断，一旦输入出错，会直接异常退出，而不会给用户提示信息。所以，在后面，我又加了几项，以便能够对简单的非法表达式进行判断。但，事实上，限于能力，还有一部分非法

表达式，这个程序是无法判断，只能异常结束。一个小小程序完成，我发现尽管可能基本满足需要，但是需要改进的地方总是无穷无尽的，要想一个程序达到尽善尽美，是需要很大耐心、毅力和能力的。

2、 关注细节错误

写程序最难的是找错误！我的程序往往会有一些小低级错误使我花费很多精力，这个是需要我们在编写程序时候专心、细心才能解决的。比如，我定义了 double sum，结果再输出的时候却以%d 的格式输出，这样结果就会出现问题！避免这样错误，是需要靠平时的经验积累和细心的。