学　　号：	0121210680106

武漢理工大學

课 程 设 计

题　　目	算术表达式的语法分析及语义分析程序设计
学　　院	计算机科学与技术学院
专　　业	软件工程
班　　级	软件 ZY1201 班
姓　　名	王飞
指导教师	饶文碧

2015 年 1 月 15 日

课程设计任务书

学生姓名： 王飞 **专业班级：** 软件 ZY1201 班

指导教师： 饶文碧 **工作单位：** 计算机科学与技术学院

题目：算术表达式的语法分析及语义分析程序设计(递归下降法、输出逆波兰式)

1.目的

通过设计、编制、调试一个算术表达式的语法及语义分析程序，加深对语法及语义分析原理的理解，并实现词法分析程序对单词序列的词法检查和分析。

2.设计内容及要求：

算术表达式的文法：

〈无符号整数〉::= 〈数字〉{〈数字〉}

〈标志符〉::= 〈字母〉{〈字母〉 | 〈数字〉}

〈表达式〉::= [＋ | －]〈项〉{〈加法运算符〉〈项〉}

〈项〉::= 〈因子〉{〈乘法运算符〉〈因子〉}

〈因子〉::= 〈标志符〉 | 〈无符号整数〉 | ‘(’〈表达式〉‘)’

〈加法运算符〉::= ＋ | －

〈乘法运算符〉::= * | /

（1） 选择递归向下分析并输出逆波兰式

（2） 写出算术表达式的符合分析方法要求的文法，给出分析方法的思想，完成分析程序设计。

（3） 编制好分析程序后，设计若干用例，上机测试并通过所设计的分析程序。

目录

算术表达式的语法分析及语义分析程序设计

----递归下降法、输出逆波兰式

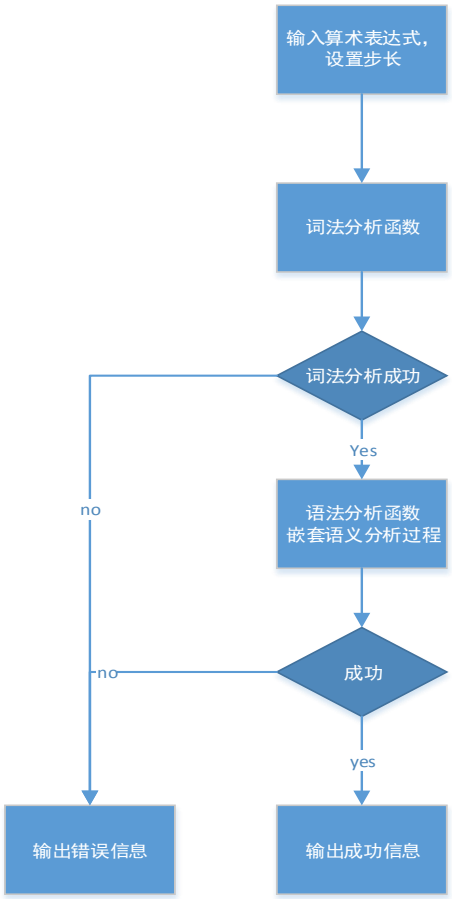
1. 系统描述

1．1 目的

通过设计、编制、调试一个算术表达式语句的语法及语义分析程序，加深对语法及语义分析原理的理解，并实现词法分析程序对单词序列的词法检查和分析。

1．2 设计内容及步骤

对算术表达式计算规则说明一下(这里只是涉及到{+，-，*，/}四个运算符)：算术表达式根据一定的优先级进行左结合运算，其中括号的优先级要大于四个运算符。这里说明下面课程设计中并不包括单目运算符，为了进一步简化程序设计的复杂性。整个过程如下：



图一：程序总体流程图

(1)规定相应的词法分析规则，规定词的种别码和值

(2)设计词法分析算法，输出词法类型，并为语法分析提供前提条件

(3)设计递归下降的子函数，进行语法分析，并提供输出结果的前提条件

(4)设计逆波兰式分析规则，嵌套到递归下降的过程中。

(5)设计输出函数，包括结果输出，错误输出

(5)测试用例和测试结果。

2. 翻译方法概述

2.1 文法设计

文法设计如下:

E -> E+T | E-T | T

T -> T*F | T/F | F

F -> (E) | 标识符 | 无符号整数

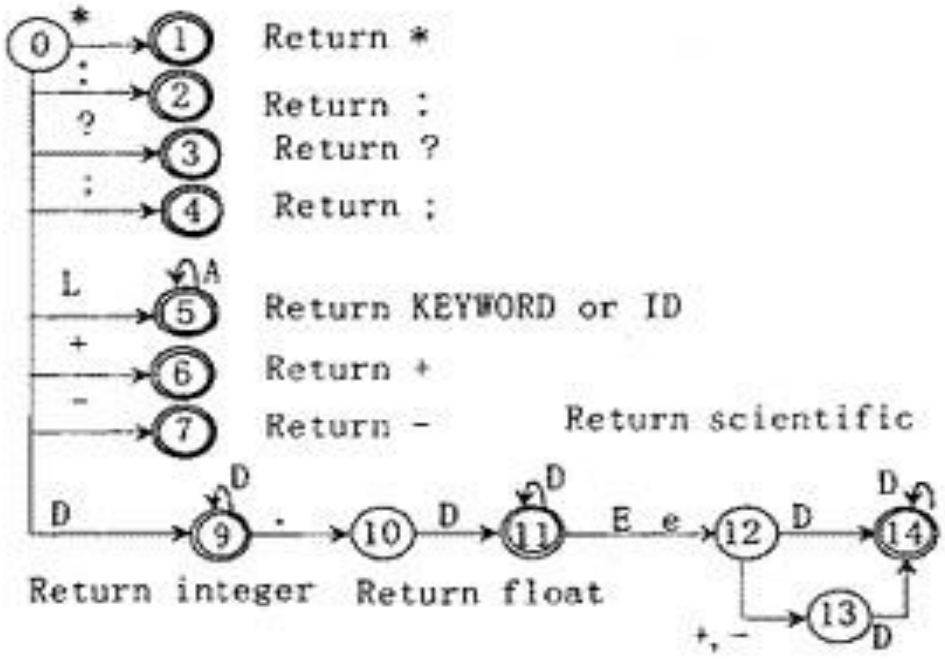
2.2 词法分析

词法分析是将源程序从左到右扫描，将单词识别出来，这就好像是人与人交谈，需要一个翻译，懂得 A 和 B 的语言，让彼此进行沟通。

词法分析单独拿出来可以简化设计，改进编译效率，词法分析是根据状态转化图来进行设计，通过标记来分出每个状态。

因此整个代码设计中词法分析的正确性直接关系到下面语法和语义分析的正确性。正确完成词法分析，就要合理设计词的属性，设计相应的 DFA，然后根据 DFA 写出相应的程序。

1. 设计的 DFA 如图二：



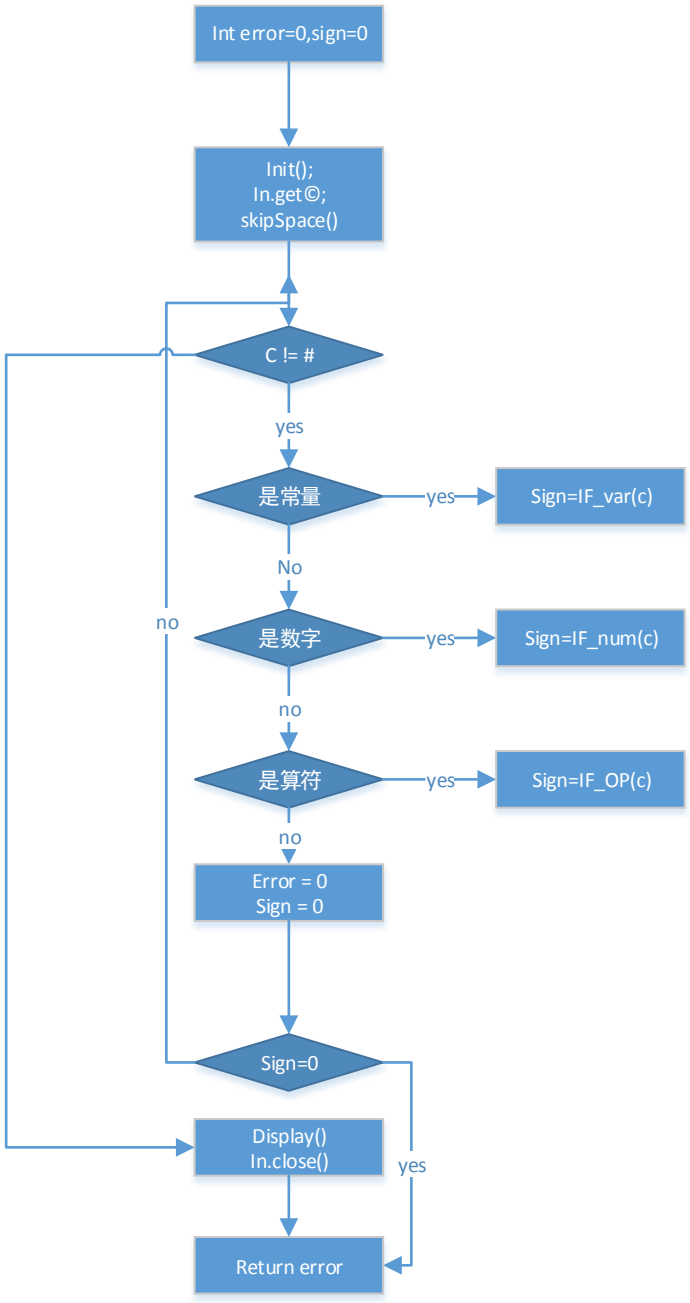
图二：DFA

2. 设计词的种别码和值如表一：

值	种别码
0	{(,),;}
1	{+, -, *, /}
2	常数
3	变量

表一：词的种别码和值

3. 词法分析子程序的流程图如图三：



图三：词法分析

说明：词法分析需要排除程序的空格，并且能够识别字符串

2.3 语法分析

1.文法设计如下:

E -> E+T | E-T | T

T -> T*F | T/F | F

F -> (E) | 标识符 | 无符号整数

根据相应的规则消除左递归设计成为LL(1)文法：

2.消除左递归如下：

E -> TG

G -〉 +TG | -TG |ε

T -> FS 子函数

S -> *FS | /FS |ε

F -> (E) | 标识符 | 无符号整数

求出相应的FIRST集合和FOLLOW集合，如表二：

	FIRST集合	FOLLOW集合
E	{(, 表示符, 整数}	
G	{+, -, ε}	{), # }
T	{(, 表示符, 整数}	
S	{*, /, ε}	{+, -,), #}
F	{(, 表示符, 整数}	

表二：FIRST集和FOLLOW集

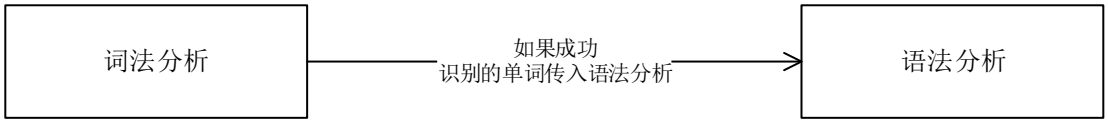
3.采用递归下降方法，根据FISRT集和FOLLOW集为每一个产生式设计一个子函数，为每一个子函数设计出相应的分析成功和失败的出口。

子函数名称如下：

```
int E();      // E -> TG 子函数
int G();      // G -〉 +TG | -TG |ε 子函数
int T();      // T -> FS 子函数
int S();      // S -> *FS | /FS |ε  子函数
int F();      // F -> (E) | 标识符 | 无符号整数 子函数
```

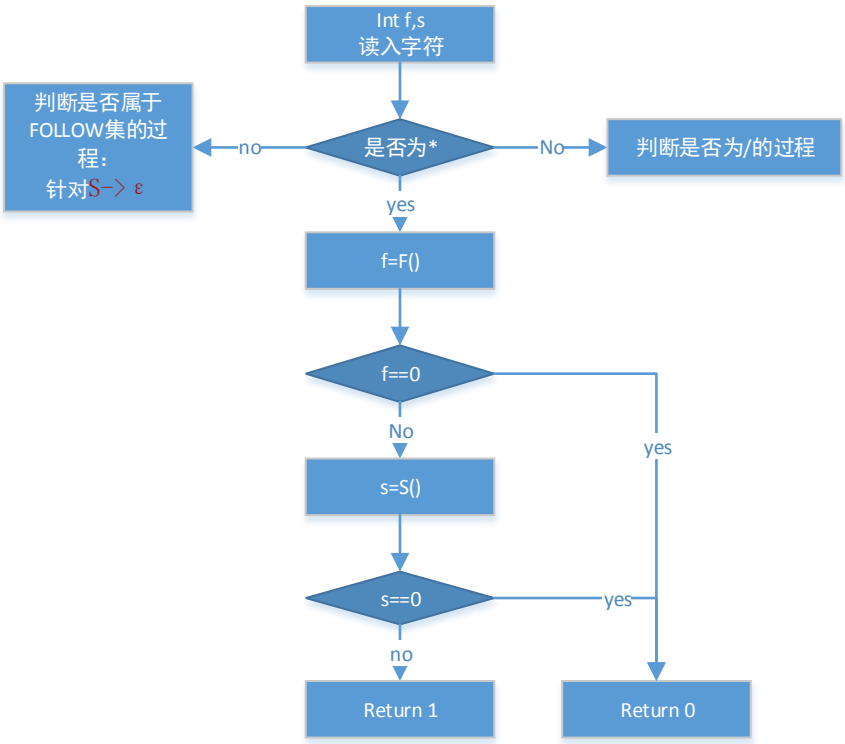
递归下降法主要采用自顶向下方法，即从文法的开始符号开始进行分析，逐渐推导的往下构造语法树，使其树叶正好构造出所给定的源程序串。因为 LL(1)文法中 SELECT 集没有重合，所以可以根据当前识别的符号与 FISRT 集和 FOLLOW 集进行匹配，当匹配成功则用当前的推导式进行推导，否则记录推导不成功。这样就解决了每一步应该选择哪一个推导式的问题。

整个过程词法分析是语法分析的前提，两者之间的关系：



图四：词法与语法分析的关系

4.语法分析过程的流程图如图五：



图五：子函数S()

说明：因为其他子函数过程与 S()函数的过程类似，所以这里只是给出 S()函数的流程图，不再重复展示。

2.3 中间代码生成

中间代码，也称中间语言，是复杂性介于源程序语言和机器语言的一种表示形式。中间代码的生成利于对后期代码的优化以及实现相应的机器无关性的转化。这里采用的是逆波兰式。

1. 逆波兰式实现的思路：

通俗的将又叫后缀式。这种形式需要进行比较算符的优先级并与栈或者队列相互结合。当当前算符与下一个字符比较，当当前算符符优先级大于下一个算符符优先级的时候，算符入栈；而遇到标示符或者整数时候一律入栈。栈里面的内容构成逆波兰式。

2. 逆波兰式的实现

因为整个过程都是自顶向下分析，并利用到递归下降的算法，如果此再重新编写算符优先算法，会显得算法结构不够紧密而且工作量重复。

- 选择将语义分析直接嵌套到语法分析(递归下降)过程中，
- 利用链表队列存储整个过程的结果；
- 借鉴算符优先的算法的思想，利用向前查看一个算符的思想，比较优先级，进行将算符入队。

2.4 属性文法

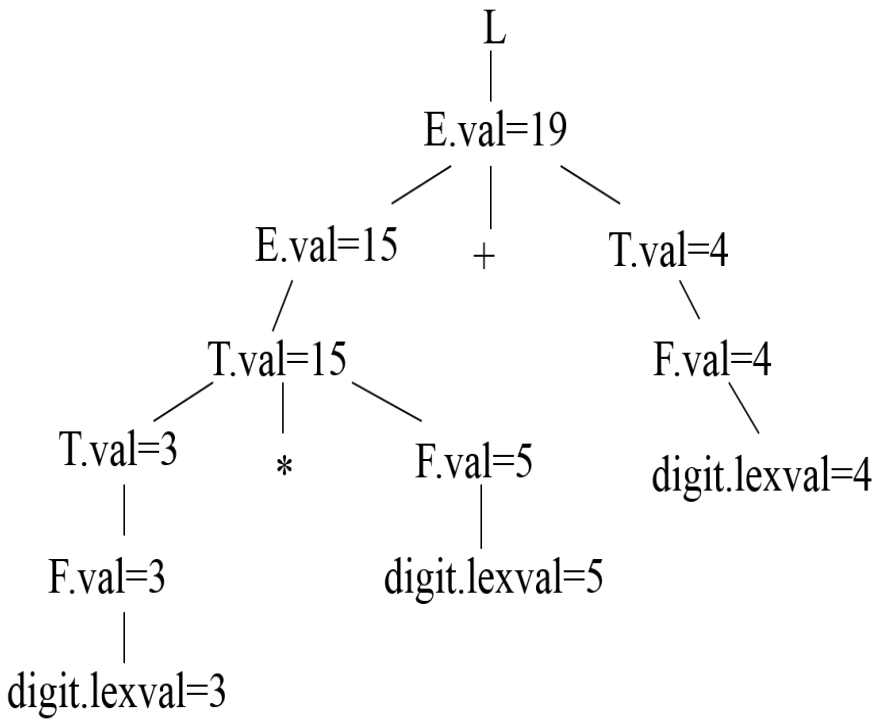
对于文法的每个产生式都配备了一组属性的计算规则，称为语义规则。

1. 思想简化说明：

- 属性文法简单说其实是对抽象的语义分析的具体化，
- 属性文法是为了实现一个句子所附加的值，不过这个值可以是具体的标示符，也可以是具体的数字，甚至可以是抽象的推导式中的某一个符号。
- 说明有属性文法的分析过程就是在语法分析过程中添加了相应的值

2. 示例语法树

对算术表达式 3*5+4 分析，语法树如图七：



图七：语法树

4. 系统的详细设计

因为前面已经涉及到文法设计、词法分析、语法分析以及词法分析的整体设计思路，所以这里不重复，只是详细说明整个程序的结构算法设计。

4.1 词法分析数据结构

1.词法分析结构体

```
struct lexical
{
   int type;
   char word[NUM];                    //这里存储字符串
   lexical *next;
};
```

2.相关常量定义

```
   #define DEL 0;    //定界符的种类编码：0-----(,),;
```

```
#define OP 1;      //算符的种别编码：1-----+,-,*,/
#define CONSTANT 2;          //常数的种别编码：3
#define    VAR 3;                    //变量的种别码

const int NUM = 10;                    //词的个数
const int OP_NUM = 8;

//算符数组
const char OP_token[OP_NUM] = {'+', '-', '*', '/',
                               '(', ')', ';', '#'};
```

3.处理空格：

```
void skipSpace(char &ch)
```

4.判断变量、标示符、数字模块

- int IF_num(char &ch, int &errorNum)----------DFA 判断是否为数字
- int IF_var(char &ch, int &errorNum)----------DFA 判断是否为变量
- int IF_OP(char &ch, int &errorNum)-----------DFA 判断是否为算符

3. 词法分析主程序

- int lexiMain(char &ch)   ----------词法分析入口程序
- void displayLexi(lexical *lhead)   ----------词法分析展示结果
- void print_ERROR_lexi(int errorNum)-------打印词法分析错误信息

4.2 语法分析数据结构

1.语法分析数据结构

```
struct semantic
{
    char word[NUM];
    semantic  *next;
};
```

2.相关变量定义

lexical *p_syntax; ----------用于语法和语义分析

3. 向前读取词法分析结果

void readPrev() ----------向前读取词法分析结果

4. 递归下降子函数

- int E(); ---------- E -> TG 子函数
- int G(); ----------G -〉 +TG | -TG | ε 子函数
- int T(); ----------T -> FS 子函数
- int S(); ----------S -> *FS | /FS | ε 子函数
- int F(); -----F -> (E) | 标识符 | 无符号整数 子函数

5. 语法分析主函数

int syntaxMain() ----------语法分析入口

4.3 逆波兰式生成

1.相关变量定义

```
char OP_sem[2] = {' ', '\0'};     ---用于比较前后出现的四则运算符号
int OP_order;                     ----------记录算法的优先级
int memory_sem = 0, num_sem = 0;-----记录识别的四则运算符号的个数
semantic *p_semantic, *shead, *send;----用于语义分析，存取逆波兰式
```

2.嵌套在递归下降中的语义分析

- 嵌套在 int S()中的语义分析设置

```
OP_sem[0] = '*';
OP_order = 2;
memory_sem++;
num_sem ++;
-------------------------------------------------当读取到*时候存取
OP_sem[0] = '/';
OP_order = 2;
memory_sem++;
```

num_sem ++;

----------当读取到/时候存取

- 嵌套在 int G()中的语义分析设置

当读取到+或者-的时候设置与上面相似，只是将 OP_sem[0] = '*';

OP_order = 2;处分别设置为+,1;-,1 即可

- 嵌套在 int E()中的语义分析设置

**-2:括号前没算符

**0:括号前构造后缀式，一个算符

**-1：两个算符,op < OP_order

**1:两个算符：op>=OP_order

**出括号时：分别操作：对于-1，OP_order 进入队列

(1)未出现括号

基本设计思路如下：当读到两个算符的时候，比较，当以前算符大于当前算符，之前算符入队，否则，先入队一个标示符，再将当前的算符入队。

```
if(memory_sem == 2)
    {
        if(order >= OP_order)
        {
            addElement_s(op);
            strcpy(op, OP_sem);
            order = OP_order;

            strcpy(semTemp,p_syntax->word);
            addElement_s(semTemp);
        }
        else
        {
            strcpy(semTemp,p_syntax->word);
            addElement_s(semTemp);
```

```
            addElement_s(OP_sem);
        }
        memory_sem--;
    }
```

(2)出现括号的设计思路

出现括号的时候，用临时变量存储当前剩下的算符，括号里面的内容按照(1)规则进行分析；括号分析成功则将临时变量重新设置为当前和以前算符进行分析。

```
if(sign == -1)
{
      addElement_s(temp_2);
      strcpy(op, temp);
      to = order;
      //memory_sem = 1;
      num_sem = 2;
  }
      else if(sign == -2)
  {
      num_sem = 0;
  }
  else
  {
      strcpy(op, temp);
      to = order;
      //memory_sem = 1;
      num_sem = 2;
  }
  //向前读取存取字
  readPrev();
```

```
if( strcmp(p_syntax->word, ";") == 0)
{
    addElement_s(op);
    memory_sem--;
}
return 1;
```

- 语义分析出口

当读到)或者;的时候为结束

```
if( strcmp(p_syntax->word, ";") == 0||  strcmp(p_syntax->word, ")")
== 0)
{
    addElement_s(op);
    memory_sem--;
}
```

3.输出逆波兰式

void displaySem(semantic *shead)----------------------输出逆波兰式

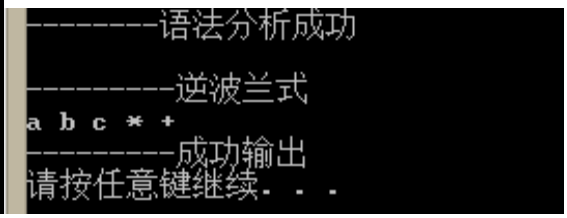
4.4 课设结果分析

说明：测试内容存储在 txt 文本中，程序读取文本进行分析

1. 课程结果

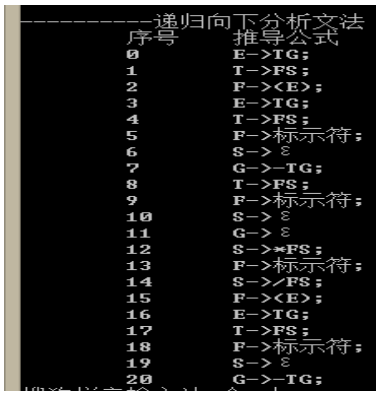
(1) 测试结果一：

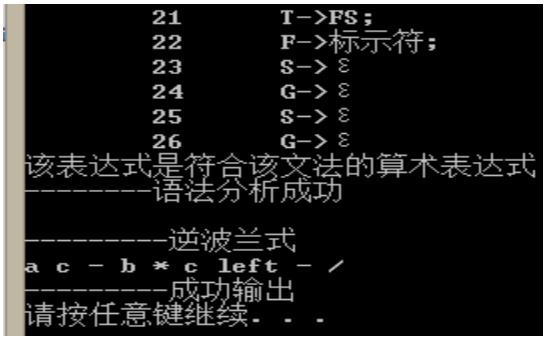
测试数据：a+b*c;#



(2) 测试结果 2：

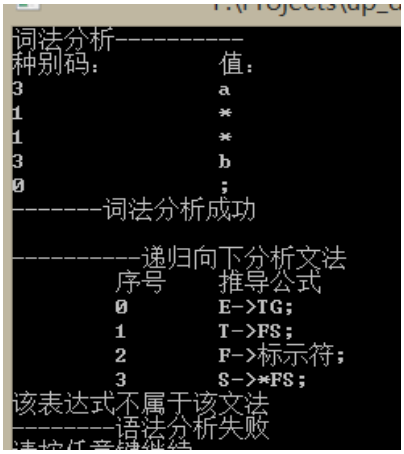
测试数据：(a-c)*b/(c-left);#





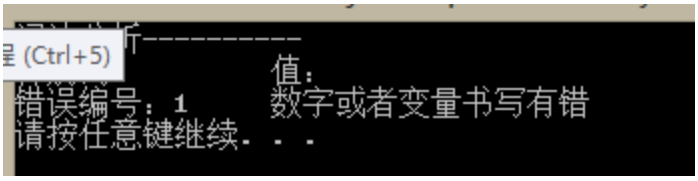
(3) 测试结果 3：

测试数据 3：a**b;#



(4) 测试结果 4：

测试数据：a + b@$c -c;#



2.课设分析

课设程序整体而言实现基本的功能，但存在一定的缺陷。首先对于单目运算符并没有实现分析；第二，并不能实现对嵌套括号的逆波兰式的正确输出

5. 程序的评价及心得体会

此次课设让自己更加深刻理解了词法分析、语法分析、语义分析之间的关系，并让自己加强了学科之间的联系。在编写整个程序过程中，遇到了不少困难，首先是对递归下降算法的理解。不过最大的困难可以说是如何在语法分析过程中直接分析出逆波兰式，因为自己并不想直接利用算符优先算法来实现，这样的话使得工作量重复。所以自己就选择借鉴其思想，并向前读取算符，进行比较实现。

程序的整体上也存在着一些缺陷，首先是不能对单目运算符进行分析，这是考虑到了生成逆波兰式的难易问题。接着便是因为逆波兰式生成还不是太完善，不能正确的将嵌套括号的算数表达式正确的分析出逆波兰式。

6. 参考文献

[1]《编译原理》.吕映芝、张素琴、蒋维杜.清华大学出版.2004 年 11 月

7.源代码附件

```
/*
**课设：算术表达式的语法分析及语义分析程序设计----逆波兰式输出+递归向下
-----------------------------------------------------------------
**单位：武汉理工大学
**作者：油纸伞
**时间：2015/1/9-1/11
*/

/*
**文法描述：
**E -> E+T | E-T | T
**T -> T*F | T/F | F
**F -> (E) | 标识符 | 无符号整数
-----------------------------------------------------------------
**消除左递归如下：
**int E();     // E -> TG 子函数
**int G();     // G -> +TG | -TG | ε 子函数
**int T();     // T -> FS 子函数
**int S();     // S -> *FS | /FS | ε  子函数
**int F();     // F -> (E) | 标识符 | 无符号整数 子函数
*/
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

#define DEL 0;    //定界符的种类编码：0-----(),;
#define OP    1;  //算符的种别编码：1-----+,-,*,/
#define CONSTANT 2;   //常数的种别编码：3
#define  VAR 3;   //变量的种别码

const int NUM = 10;//词的个数????????
const int OP_NUM = 8;

//算符数组
```

```
const char OP_token[OP_NUM] = {'+', '-', '*', '/',
                               '(', ')', ';', '#'};

int n = 0;   //用于标记递归次序、分析层次
char OP_sem[2] = {' ', '\0'}; //用于比较前后出现的四则运算符号
int OP_order;
int memory_sem = 0, num_sem = 0;   //记录识别的四则运算符号的个数

char semTemp[NUM] = "";
ifstream in("1.txt");

//词法结构体
struct lexical
{
    //enum type t1;
    int type;
    char word[NUM];
    lexical *next;
};

lexical *lhead, *lend;
lexical *p_syntax;   //用于语法和语义分析

void initLexical()
{
    lhead = NULL;
    lend = lhead;
}

//尾插入元素
lexical *addElement_l(char *str, int type)
{
    lexical *p;
    p = new lexical();
    strcpy(p->word, str);
    p->type = type;
    if(lhead == NULL)
    {
        lhead = p;
    }
    else
    {
        lend->next = p;
    }
```

```
    lend = p;
    lend->next = NULL;
    return lhead;
}

struct semantic
{
    char word[NUM];
    semantic *next;
};

semantic *p_semantic, *shead, *send;    //用于语义分析，存取逆波兰式

void initSemantic()
{
    shead = NULL;
    send = shead;
}

//尾插入元素
semantic *addElement_s(char *str)
{
    semantic *p;
    p = new semantic();
    strcpy(p->word, str);
    if(shead == NULL)
    {
        shead = p;
    }
    else
    {
        send->next = p;
    }
    send = p;
    send->next = NULL;
    return shead;
}

//处理空格
void skipSpace(char &ch)
{
    if(ch == ' ')
    {
        while(ch == ' ')
```

```
        {
            in.get(ch);
    }
}
        }

//判断算符
int OP_fair(char ch)
{
    for(int i = 0; i < OP_NUM; ++i)
    {
        if(ch == OP_token[i])
            return 1;
    }
        return 0;
}

//判断字符
int CONST_fair(char ch)
{
    if(ch >='a' && ch <='z' || ch >='A' && ch <='Z')
        return 1;
    else
        return 0;

}

//判断数字
int NUM_fair(char ch)
{
    if( ch >='0' && ch <='9')
        return 1;
    else
        return 0;
}

//DFA判断是否为数字
int IF_num(char &ch, int &errorNum)
{

    int type = 2;
    int i = 0;
    char temp[NUM] = "";
    while(NUM_fair(ch))
```

```
    {
        temp[i] = ch;
        i++;
        in.get(ch);
    }
    temp[i] = '\0'
;   if(ch == ' ')
        skipSpace(ch);
    else if(!OP_fair(ch))
    {
        errorNum = 1;//错误编号1：后缀连接错误
        return 0;
    }

    //此处插入节点
    addElement_1(temp, type);
    return 1;
}

//DFA判断是否为变量
int IF_var(char &ch, int &errorNum)
{

    int type = 3;
    int i = 0;
    char temp[NUM] = "";
    while(CONST_fair(ch) || NUM_fair(ch))
    {
        temp[i] = ch;
        i++;
        in.get(ch);
    }
    temp[i] = '\0';
    if(ch == ' ')
        skipSpace(ch);//此处处理空格
    else if(!OP_fair(ch))
    {
        errorNum = 1;//错误编号：后缀连接错误
        return 0;
    }

    //此处插入节点
    addElement_1(temp, type);
    return 1;
```

```
}

//DFA判断是否为算符
int IF_OP(char &ch, int &errorNum)
{

    int type;
    char temp[2] = {' ', '\0'};
    int i = 0;
    temp[i] = ch;
    switch(ch)
    {
    case ';':
        type = 0;
        in.get(ch);
        if(ch == ' ')
            skipSpace(ch);
        break;
    case '(':
        type = 0;
        in.get(ch);
        if(ch == ' ')
            skipSpace(ch);//这里处理空格
        else if(!(CONST_fair(ch) || NUM_fair(ch)) && ch != '(')
        {
            if(ch == ')')
                errorNum = 3;//算符括号里面无变量运算
            else errorNum = 2;    //算符重叠
            return 0;
        }

        break;
    case ')':
        type = 0;
        in.get(ch);
        if(ch == ' ')
            skipSpace(ch);//这里处理空格

        break;
    case '+':
        type = 1;
        in.get(ch);
        if(ch == ' ')
```

```
            skipSpace(ch);//这里处理空格
        else if(!(CONST_fair(ch) || NUM_fair(ch)) && ch != '(')
        {
            errorNum = 2;//算符重叠
            return 0;
        }
        break;
    case '-':
        type = 1;
        in.get(ch);
        if(ch == ' ')
            skipSpace(ch);//这里处理空格
        else if(!(CONST_fair(ch) || NUM_fair(ch)) && ch != '(')
        {
            errorNum = 2;//算符重叠
            return 0;
        }
        break;
    case '*':
        type = 1;
        in.get(ch);
        if(ch == ' ')
            skipSpace(ch);//这里处理空格
        /*else if(!(CONST_fair(ch) || NUM_fair(ch)) && ch != '(')
        {
            errorNum = 2;//算符重叠
            return 0;
        }*/
        break;
    case '/':
        type = 1;
        in.get(ch);
        if(ch == ' ')
            skipSpace(ch);//这里处理空格
        else if(!(CONST_fair(ch) || NUM_fair(ch)) && ch != '(')
        {
            errorNum = 2;//算符重叠
            return 0;
        }
        break;
    default:
        errorNum = 4;//未识别算符
        cout << ch;
        in.get(ch);
```

```
        if(ch == ' ')
            skipSpace(ch);//这里处理空格
        return 0;
    }
    //添加节点
    addElement_l(temp, type);
    return 1;
}

//展示词法分析结果
void displayLexi(lexical *lhead)
{
    lexical *ph;
    ph = lhead;
    while(ph != NULL)
    {
        cout << ph->type <<"\t\t" << ph->word<<endl;
        ph = ph->next;
    }
}

//词法分析入口程序
int lexiMain(char &ch)
{
    int errorNum = 0;
    int sign_lexi = 0;

    initLexical();
    in.get(ch);
    skipSpace(ch);

    cout << "词法分析---------\n";
    cout << "种别码：\t" << "值：\n";
    while(ch != '#')
    {
        if(CONST_fair(ch))
            sign_lexi = IF_var(ch, errorNum);
        else if(NUM_fair(ch))
            sign_lexi = IF_num(ch, errorNum);
        else if(OP_fair(ch))
            sign_lexi = IF_OP(ch, errorNum);
        else
        {
            cout << ch;
```

```
                errorNum = 4;//未识别的字符
                sign_lexi = 0;
            }
            if(sign_lexi == 0) return errorNum;
        }
        displayLexi(lhead);
        in.close();
        return errorNum;
}

//打印词法分析错误信息
void print_ERROR_lexi(int errorNum)
{
        switch(errorNum)
        {
        case 0:
            cout << "-------词法分析成功\n";break;
        case 1:
            cout << "错误编号：" << errorNum << "\t数字或者变量书写有错\n";break;
        case 2:
            cout << "错误编号：" << errorNum <<"\t出现算符重叠\n";break;
        case 3:
            cout << "错误编号：" << errorNum << "\t出现空括号\n";break;
        case 4:
            cout << "错误编号：" << errorNum  << "\t出现未识别字符\n";break;
        default:
            cout << "错误编号：" << errorNum << "\t未知错误\n";break;
        }
}

void initSyntax()
{
        p_syntax = lhead;
}

//当与SELECT集合中符号相互匹配时候，指针右移，向前读入一个符号
void readPrev()
{
        p_syntax = p_syntax->next;
}
int E();
/*void print_program()
{
        while(p_syntax!=NULL)
```

```
    {
        cout << p_syntax->word;
        p_syntax = p_syntax->next;
    }
    cout << endl;
}*/

/*
**FOLLOW集
**S->ε :FOLLOW(S)={+,-,#,)}
**G->ε :FOLLOW(G)={#,)}
*/

int F()
{
    static char op[2];
    static int order;

    //当遇见括号时候存取括号前面的算符
    char temp[2], temp_2[2];
    int to;

    /*
    **-2:括号前没算符
    **0:括号前构造后缀式，一个算符
    **-1：两个算符,op < OP_order
    **1:两个算符：op>=OP_order
    **出括号时：分别操作：对于-1，OP_order进入队列
    */
    int sign;

    int e;

    if((memory_sem == 1) && (num_sem == 1))
    {
        strcpy(op, OP_sem);
        order = OP_order;
    }
    if(strcmp(p_syntax->word, "(") == 0)
    {
        if(memory_sem == 0)
        {
            sign = -2;
        }
```

```
//括号前只有一个算符
else if(memory_sem == 1)
{
    sign = 0;
    strcpy(temp, op);
    to = order;
}

//括号前只有一个算符
else if(memory_sem == 2)
{
    if(order >= OP_order)
    {
        sign = 1;
        addElement_s(op);
        strcpy(temp, OP_sem);
        to = OP_order;
    }
    else
    {
        sign = -1;
        strcpy(temp, op);
        to = order;

        strcpy(temp_2, OP_sem);
    }
}

num_sem = 0;
memory_sem = 0;

cout << '\t' << n++ << '\t' << "F->(E);" << endl;

//向前读取存取字
readPrev();
e = E();
if(!e)
    return 0;
if(strcmp(p_syntax->word, ")") == 0)
{
    if(sign == -1)
    {
        addElement_s(temp_2);
        strcpy(op, temp);
```

```
                to = order;
                //memory_sem = 1;
                num_sem = 2;
            }
            else if(sign == -2)
            {
                num_sem = 0;
            }
            else
            {
                strcpy(op, temp);
                to = order;
                //memory_sem = 1;
                num_sem = 2;
            }


            //向前读取存取字
            readPrev();

            if( strcmp(p_syntax->word, ";") == 0)
            {
                addElement_s(op);
                memory_sem--;
            }
            return 1;
        }
        else
        {
            cout << "括号匹配不成功" << endl;
            return 0;
        }
    }
    else if(p_syntax->type == 2)
    {
        cout << '\t' << n++ << '\t' << "F->整数;" << endl;
        if(memory_sem == 2)
        {
            if(order >= OP_order)
            {
                addElement_s(op);
                strcpy(op, OP_sem);
                order = OP_order;
```

```
                strcpy(semTemp,p_syntax->word);
                addElement_s(semTemp);
            }
            else
            {
                strcpy(semTemp,p_syntax->word);
                addElement_s(semTemp);
                addElement_s(OP_sem);
            }
            memory_sem--;
        }
        else
        {
            strcpy(semTemp,p_syntax->word);
            addElement_s(semTemp);
        }

        readPrev();

        //遇到结界当前最后一算符入队
        if( strcmp(p_syntax->word, ";") == 0||  strcmp(p_syntax->word, ")") == 0)
        {
            addElement_s(op);
            memory_sem--;
        }
        return 1;
    }
    else if(p_syntax->type == 3)
    {
        cout << '\t' << n++ << '\t' << "F->标示符;" << endl;

        if(memory_sem == 2)
        {
            if(order >= OP_order)
            {
                addElement_s(op);
                strcpy(op, OP_sem);
                order = OP_order;

                strcpy(semTemp,p_syntax->word);
                addElement_s(semTemp);
            }
            else
            {
```

```
                strcpy(semTemp,p_syntax->word);
                addElement_s(semTemp);
                addElement_s(OP_sem);
            }
            memory_sem--;
        }
        else
        {
            strcpy(semTemp,p_syntax->word);
            addElement_s(semTemp);
        }
        readPrev();

        //此处应该用字符比较函数，用==直接比较不出结果
        if( strcmp(p_syntax->word, ";") == 0||  strcmp(p_syntax->word, ")") == 0)
        {
            addElement_s(op);
            memory_sem--;
        }
        return 1;
    }
    else return 0;

}

int S()
{
    int f, s;
    if(strcmp(p_syntax->word, "*") == 0)
    {
        cout << '\t' << n++ << '\t' << "S->*FS;" << endl;

        //用于语义分析
        OP_sem[0] = '*';
        OP_order = 2;
        memory_sem++;
        num_sem ++;

        //向前读取存取字
        readPrev();
        f = F();
        if(!f) return 0;
        s = S();
        if(!s) return 0;
```

```
        return 1;
    }
    else if(strcmp(p_syntax->word, "/") == 0)
    {
        cout << '\t' << n++ << '\t' << "S->/FS;" << endl;

        //用于语义分析
        OP_sem[0] = '/';
        OP_order = 2;
        memory_sem++;
        num_sem ++;

        //向前读取存取字
        readPrev();
        f = F();
        if(!f) return 0;
        s = S();
        if(!s) return 0;

        return 1;
    }
    //此处是关于S->空的情况
    else if(strcmp(p_syntax->word, ";") == 0 || strcmp(p_syntax->word, ")") == 0 ||
            strcmp(p_syntax->word, "+") == 0 || strcmp(p_syntax->word, "-") == 0)
    {
        cout << '\t' << n++ << '\t' << "S-> ε " << endl;
        return 1;
    }
    else return 0;

}

int T()
{
    int f, s;
    cout << '\t' << n++ << '\t' << "T->FS;" << endl;
    f = F();
    if(!f)
        return 0;
    s = S();
    if(!s)
        return 0;
```

```
    return 1;
}

int G()
{
    int t, g;
    if(strcmp(p_syntax->word, "+") == 0)//这里记着在看一看，单引号，双引号
    {
        cout << '\t' << n++ << '\t' << "G->+TG;" << endl;

        //用于语义分析
        OP_sem[0] = '+';
        OP_order = 1;
        memory_sem++;
        num_sem ++;

        //向前读取存取字
        readPrev();
        t = T();
        if(!t) return 0;
        g = G();
        if(!g) return 0;

        return 1;
    }
    else if(strcmp(p_syntax->word, "-") == 0)
    {
        cout << '\t' << n++ << '\t' << "G->-TG;" << endl;

        //用于语义分析
        OP_sem[0] = '-';
        OP_order = 1;
        memory_sem++;
        num_sem ++;

        //向前读取存取字
        readPrev();
        t = T();
        if(!t) return 0;
        g = G();
        if(!g) return 0;

        return 1;
    }
```

```
    //此处写G->空的情况
    else if(strcmp(p_syntax->word, ";") == 0 || strcmp(p_syntax->word, ")") == 0)
    {
        cout << '\t' << n++ << '\t' << "G-> ε " << endl;
        return 1;
    }
    else return 0;
}

int E()
{
    int t, g;
    if ((strcmp(p_syntax->word,"+") == 0)|| (strcmp(p_syntax->word,"-") == 0))
        readPrev();
    cout << '\t' << n++ << '\t' << "E->TG;" << endl;
    t = T();
    if(t == 0)
        return 0;
    g = G();
    if(g == 0)
        return 0;

    return 1;
}

//语法分析入口
int syntaxMain()
{
    int n = 0;

    initSyntax();
    initSemantic();
    //print_program();

    cout << endl;

    cout << "---------递归向下分析文法\n";
    cout << "\t序号" << "\t推导公式\n";
    n = E();
    if(!n)
    {
        cout << "该表达式不属于该文法\n";
        cout << "--------语法分析失败\n";
        return 0;
```

```
    }
    else
    {
        cout << "该表达式是符合该文法的算术表达式\n";
        cout << "--------语法分析成功\n";
        return 1;
    }
}

//存在缺陷：不能转换嵌套括号的逆波兰式
void displaySem(semantic *shead)
{
    semantic *ph;
    ph = shead;

    cout << "\n--------逆波兰式\n";
    while(ph != NULL)
    {
        cout << ph->word<<" ";
        ph = ph->next;
    }
    cout << "\n--------成功输出\n";
}

int main()
{
    char ch;
    int errorNum;
    int t;

    errorNum = lexiMain(ch);
    print_ERROR_lexi(errorNum);
    if(errorNum == 0)
    {
        if(syntaxMain())
            displaySem(shead);
    }
    system("pause");
    return 0;
}
```

本科生课程设计成绩评定表

班级：软件 ZY1201 班　　姓名：王飞　　学号：0121210680106

序号	评分项目	满分	实得分
1	学习态度认真、遵守纪律	10	
2	设计分析合理性	10	
3	设计方案正确性、可行性、创造性	20	
4	设计结果正确性	40	
5	设计报告的规范性	10	
6	设计验收	10	
		总得分/等级	
评语：			

注：最终成绩以五级分制记。优（90-100 分）、良（80-89 分）、中（70-79 分）、及格（60-69 分）、60 分以下为不及格

指导教师签名：

2015 年　月　日