# 实验四 算数表达式的递归下降分析

## 一、 实验目的

根据给定的上下文无关文法，分析任意一个算数表达式的语法结构

输入：任意的算数表达式

输出：与输入对应的一颗语法树或错误

基础文法如下：

```
<Expr> → <Term> <Expr1>
<Expr1> → <AddOp> <Term> <Expr1> | empty
<Term> → <Factor> <Term1>
<Term1> → <MulOp> <Factor> <Term1> | empty
<Factor> → id |number | ( <Expr> )
<AddOp> → + | -
<MulOp> → * | /
```

**图 1 基础文法**

## 二、 实验要求

1. 语法分析方法采用递归子程序法
2. 输入正确时，输出其对应的语法树，树根标记为<Expr>;输入错误时，输出 error

## 三、 实验设计

### 1. 文法的分析

首先，我们将文法进行一定的等价变换，并利用《编译原理》课程中学习的相关知识，构造其 FIRST 集和 FOLLOW 集：

```
将文法改写：
E -> T E1
E1 -> + T E1 | - T E1 | C
T -> F T1
T1 -> * F T1 | / F T1 | C
F -> ID | NUMBER | (E)
```

构造FIRST集和FOLLOW集

```
FIRST(E) = {ID,NUMBER, ( }
FIRST(E1) = {+,-,C }
FIRST(T) = {ID,NUMBER, ( }
FIRST(T1) = {*,/,C}
FIRST(F) = {ID,NUMBER,( }

FOLLOW(E) = {#, )}
FOLLOW(E1) = {#, )}
FOLLOW(T) = {#, ), +, -}
FOLLOW(T1) = {#, ), +, -}
FOLLOW(F) = {#, ), *, /, +, -}
```

**图 1 文法改造**

## 2. 递归下降的分析

a. 递归下降的思想：对文法的每一个非终结符都编写一个分析过程，当根据文法和现行输入符号预测到要用某个非终结符去匹配输入串时，就调用该非终结符的分析过程，利用该方法构造的语法分析器往往含有不少递归过程，所以该方法被称为递归子程序法。

b. 按照该思路和前面的词法分析的工作，我们为改造后的五个非终结符构造分析函数，以词法分析中保存的单词表作为输入进行匹配和递归下降。

c. 对于产生式中的“空集”符号，我们使用 FOLLOW 集进行匹配，如果在 FOLLOW 集中有和当前输入字符匹配的字符，则直接返回，表示该条产生式推出“空集”

d. 对于错误的处理，这里直接报错并退出

### 3. 语法树的生成

a. 根据《编译原理》课程中的内容，我们知道在语义分析的时候。对于特定的文法，可以将语义分析生成中间代码和建树一起进行，做到一次遍历得到语法分析树和中间代码。但是对于某些文法来说，并不能做到一次遍历，所以这里选择建立一颗不含语义的语法分析树

b. 由于文法的长短不一，树的子节点的数目不固定，但是我们可以将全部的语法分析树都转变为二叉树的形式，利用孩子兄弟表示法进行表征，方便遍历和树的建立，其结构如下：

```
struct TreeNode
{
    string name;
    int type;//-1代表是非终结符
    struct TreeNode * son, *bro;//孩子兄弟表示法
    TreeNode(string n, int t):name(n),type(t), son(NULL), bro(NULL){}
};
```

**图 3 树节点的构造**

其中 string name 表示该结点的名称；type 表示其编码，两个指针分别指向其孩子和兄弟

c. 树结点的建立时机：当进入一个非终结符的分析函数时，对其存在的孩子和兄弟结点进行建立，在进行后续的递归调用。则有如下函数声明：

```
TreeNode * X(TreeNode * p, string bro);
//输入参数：p:当前结点的指针，bro:当前节点兄弟节点的名称
//输出：返回当前节点的指针
```

**图 4 树节点定义**

考虑到当前节点的兄弟节点由其父节点所确定，所以这里需要从父节点额外传入兄弟节点的名称，进行该节点的建立

## 四、 重要代码分析

这里只对重点的函数进行讲解，完整的代码见附录

1. 不含树节点建立的递归下降分析代码

首先我们编写不含树节点建立的递归下降分析的代码，比较简单只需要根据产生式进行编写即可，注意 error 的判断和 FOLLOW 集的运用。这里以 E1 为例子进行说明

```
void E1()//E1 -> +TE1 | -TE1 | C
{
    if(word_list[index].s=="+")
    {
        index++;
        T();
        E1();
    }
    else if(word_list[index].s=="-")
    {
        index++;
        T();
        E1();
    }
    else if(word_list[index].s==")"||word_list[index].s=="#")
        return;
    else
    {
        error();
    }
}
```

当输入的终结符 word_list[index].s 与产生式中终结符匹配成功后，index 后移一位，调用后续非终结符对应的函数；不匹配时，与 E1 的 FOLLOW 集进行匹配，成功则直接返回，否则报错，error 函数的代码如下：

```
void error()
{
    cout << "error!" << endl;
    exit(-1);
}
```

其余非终结符的分析函数的编写思路一致，不再赘述

2. 含树节点建立的递归下降分析代码

按照上述的分析，需要给函数加上返回类型和输入参数。这里依然以 E1 的代码为例：

```
TreeNode* E1(TreeNode * p)
{
    if(word_list[index].s=="+")
    {
        TreeNode * cur = new TreeNode("+", 21);
        p->son = cur;//创建其儿子结点
        index++;//进行匹配
        TreeNode * right = new TreeNode("Term", -1);
        cur->bro = right;//创建儿子结点的兄弟结点
        TreeNode * b = T(right, "Expr1");
        E1(b->bro);
        return p;
    }
    else if(word_list[index].s=="-")
    {
        TreeNode * cur = new TreeNode("-", 22);
        p->son = cur;
        index++;
        TreeNode * right = new TreeNode("Term", -1);
        cur->bro = right;
        TreeNode * b = T(right, "Expr1");
        E1(b->bro);
        return p;
    }
    else if(word_list[index].s==")"||word_list[index].s=="#")
    {
        TreeNode * cur = new TreeNode("null", -2);
        p->son = cur;
        return p;
    }
    else
        error();
    return p;
}
```

需要注意几点：

a. 首先 E1 不存在兄弟节点，所以输入中的 string bro 可以省略

b. 对于输入的 p 指针不可以将其所指地址进行改动，否则将无法正确返回该节点，导致树建立的失败

**c. 对于 E1 或 T1 需要完成的是：输入节点的孩子节点的建立，以**

**及输入节点的孩子节点的兄弟节点的建立；（这个非常重要）而对于 E 或 T 或 F 函数来说，需要完成的是输入节点的兄弟节点的建立和孩子节点的建立。**

这里给出 E 的递归下降的代码进行对比

```
TreeNode* E(TreeNode * p, string bro)
{
    TreeNode * son = new TreeNode("Term", -1);//E->TE1
    if(bro!="")//存在 )
    {
        TreeNode * b = new TreeNode(bro, 33);
        p->bro = b;
    }
    p->son = son;
    TreeNode * b = T(p->son, "Expr1");
    E1(b->bro);
    return p;
}
```

## 五、 实验结果

这里依然利用前面的词法分析器进行词法分析得到单词表，将单词表用于此处的语法分析和建立语法树，最后通过简单的 DFS 进行语法分析树的遍历和输出。

这里测试三组数据：

不带括号的数据

带括号的数据

存在错误的数据

## 六、 实验总结

1. 学习了如何利用递归下降子程序法对算术表达式进行语法分析

2. 学习了语法分析树的基本建立方法，为后续的其他结构的语法分析奠定了基础
3. 正确的应用《编译原理》课程中的理论知识并进行实践，能够大大降低代码实现的难度
4. 语法分析与词法分析和语义分析紧密联系，在实现时，需要同时考虑与词法分析和语义分析的衔接

## 七、 附录