实验五 对多条语句的递归下降分析

一、 实验目的

根据给定的上下文无关文法，对高级程序设计语言中常见的几种执行语句进行语法分析

二、 实验要求

输入：一串执行语句，其中包括：赋值语句、选择语句和循环语句，变量声明等

输出：与输入对应的一颗完整的语法树或者错误

要求：

1. 基础文法以<Block>为开始符号
2. 语法分析方法采用递归子程序法
3. 赋值语句：左部为一个简单变量，右部为一个算术表达式
4. 选择语句：包含 if 单分支，if-else 双分支两种结构。只考虑分支判断条件为一个简单关系运算表达式的情况即可，暂不处理逻辑运算
5. 循环语句：包含三种循环中的一种即可

三、 实验设计

1. 首先从实验附录中可以找到参考的 little C 的文法

```
PROG→BLOCK
BLOCK→{  DECLS  STMTS  }

DECLS→DECLS  DECL | empty
DECL→TYPE  NAMES  ;
TYPE→int
NAMES→NAMES  ,  NAME | NAME
NAME→id

STMTS→STMTS  STMT | empty
STMT→id  =  EXPR ;
STMT→if  (  BOOL )  STMT
STMT→if  (  BOOL )  STMT  else  STMT
STMT→while  (  BOOL )  STMT
STMT→BLOCK

EXPR→EXPR  ADD  TERM  |  TERM
ADD→+ | -
TERM→TERM  MUL  UNARY  |  FACTOR
MUL→* | /
FACTOR→(  BOOL  )  |  id  |  number

REL→EXPR  ROP  EXPR
ROP→ > | >= | < | <= | == | !=
```

图 1 little C 的文法

可以看到该文法是含左递归的，我们需要将其改造为不含左递归的无二义性文法，改造后的文法如下：

```
改造后的消除左递归的little C文法 empty表示空
PROG → BLOCK
BLOCK → { DECLS STMTS }

DECLS → DECL DECLS | empty
DECL → INT NAMES;
NAMES → ID NAMES1
NAMES1 → , ID NAMES1 | empty

STMTS → STMT STMTS | empty
STMT → IF-STMT | WHILE-STMT | ASSIGN-STMT | BLOCK
IF-STMT → IF (BOOL) STMT ELSE-STMT
ELSE-STMT → ELSE STMT | $
WHILE-STMT → WHILE (BOOL) STMT
ASSIGN-STMT → ID = EXPR;

EXPR → TERM EXPR1
EXPR1 → ADDOP TERM EXPR1 | empty
TERM → FACTOR TERM1
TERM1 → MULOP FACTOR TERM1 | empty
FACTOR → ID | NUM | ( EXPR )
ADDOP → + | -
MULOP → * | /
BOOL→EXPR ROP EXPR
ROP→ > | >= | < | <= | == | !=
```

图 2 改造后的不含左递归的 little C 文法

2. 构造 First 集和 Follow 集

我们需要构造该文法中各个产生式的 First 集和 Follow 集，以便根据当前的输入字符进行文法分析，对于 empty，我们利用 Follow 集进行处理。

First 集：First(a)表示可以从 a 推导出的所有串首终结符构成的集合，其计算方法如下：($表示 empty)

(1)如果 X 是终结符, first(X) = {X}

(2)如果 X-> $ 是产生式, 则将{$} 加入 first(X)

(3)如果 X 是非终结符, 且 X->Y1Y2…Yk 是产生式,则:

1)若对于某个 i, 有 a 属于 first(Yi), 且$属于 first(Y1)…first(Yi-1), 则 a 属于 first(X)

2)若对于 j = 1, 2, …, k 有 $ 属于 first(Yj) 则$ 属于 first(X)

Follow 集：Follow(A)表示可能在某个句型中紧跟 A 后边的终结符的集合，其具体计算方式如下：

(1)如果 S 是开始符号, 则$属于 follow(S), $是记号串的结束符号。

(2)如果存在产生式 A-> aBb 则将 first(b)中除了$ 以外的符号加入到 follow(B)中。

(3)如果存在产生式 A-> aB, 或 A-> aBb 且$ 属于 first(b),则 将 follow(A)加 入到 follow(B)中。

对于我们分析的 little C 的文法运用上述方法，求得其 First 集和 Follow 集如下图所示：

图 3 little C 文法所对应的 FIRST 集

图 4 little C 文法所对应的 FOLLOW 集

3. 语法分析的思路：

a. 以前面词法分析得到的单词作为输入，设置全局变量 index 进行单词的遍历，同时利用递归下降分析的方法对其进行语法分析

b. 在输入匹配时，则自动生成相应的树结点，index+1，指向下一个输入单词；当输入不匹配时，执行 error 函数，进行报错并退出

c. 沿用对算数表达式进行递归下降的语法分析的思路，构造以下树节点：

```
struct TreeNode//用孩子兄弟表示法将树转换为二叉树
{
    string name;//结点名称
    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 5 树结点结构体展示

树节点主要存储了结点对应的名称和编码，对于终结符的编码，沿用在词法分析中的编码，对于非终结符结点，type=-1；对于 empty 结点，type=-2

表 1 字符编码表

单词符号	编码	单词符号	编码	单词符号	编码
main	0	-	22	<	30
if	1	*	23	<=	31

else	2	/	24	(	32
while	3	=	25	)	33
int	4	==	26	{	34
id	10	!=	27	}	35
num	20	>	28	;	36
+	21	>=	29	,	37

d. 当语法分析结束并没有报错时，则执行 DFS 进行构建的语法树的深度优先搜索，将对应的树节点的名称和编码进行输出，以验证二叉树的构建是否正确。

```
void DFS(TreeNode * root)
{
    if(root==NULL)
        return;
    cout << root->name << " " << root->type << endl;
    DFS(root->son);
    DFS(root->bro);
}
```

图 6 DFS 遍历二叉树代码

四、 重要代码设计

1. 对实验四对算术表达式的递归下降分析的改进

在实验四中，为了更好的表示父子、兄弟的关系，在参数设置时，设置了 string bro 参数进行该结点的兄弟结点的名称传递。但是经过分析后，发现该参数是不需要传递的，可以在父亲结点创建后，将其对应产生式中的子结点、子结点的兄弟结点一并创建，再分别传入即可。改进之后可以使得递归下降分析的函数形式一致，代码思路更加清晰，方便后续步骤的进行。

2. 对于函数的设计和实现，同样是先写出不建立树结点的递归下降分析的代码后，再将其改造为构建树节点的递归下降分析代码。

这里以 BLOCK 为例子进行分析

a. 构建不含树结点建立的递归下降代码

首先将 BLOCK 对应的产生式、First 集在这里给出：

```
BLOCK→{ DECLS STMTS }
FIRST(BLOCK) = { { }
```

图 7 BLOCK 对应的产生式和 Follow 集

根据《编译原理》课程中提到的递归下降分析的方法，写出如下递归下降的代码：

```
void BLOCK()
{
    if(word_list[index].s=="{")
    {
        index++;//当前输入匹配后index指向下一个token
        DECLS();
        STMTS();
        if(word_list[index].s=="}")
        {
            index++;
        }
        else
        {
            error();//不匹配的情况则调用error函数进行报错
        }
    }
    else
    {
        error();
    }
}
```

图 8 递归下降的代码

b. 构建含树节点建立的递归下降代码

根据已有的代码，分两步建立含树节点建立的递归下降代码：1）在

每次匹配成功后，为匹配的字符建立结点，并链上二叉树中；2）为将要递归调用的函数(非终结符)也建立结点，并将该结点作为参数传入该函数中。

以 BLOCK 函数为例进行说明

```
TreeNode *  BLOCK(TreeNode * p)
{
    if(word_list[index].s=="{")//当前匹配成功
    {
        TreeNode * cur1 = new TreeNode("{", 34);//建立该终结符的结点
        p->son = cur1;//只有第一个是孩子，其余的都是孩子的兄弟
        index++;
        TreeNode * cur2 = new TreeNode("DECLS", -1);//建立将要调用的函数对应的结点
        cur1->bro = cur2;
        DECLS(cur2);
        TreeNode * cur3 = new TreeNode("STMTS", -1);//非终结符结点的创建
        cur2->bro = cur3;
        STMTS(cur3);
        if(word_list[index].s=="}")
        {
            TreeNode * cur4 = new TreeNode("}", 35);//非终结符匹配后创建结点
            cur3->bro = cur4;
            index++;
        }
        else
        {
            error();
        }
    }
    else
    {
        error();
    }
    return p;
}
```

图 9 包含树节点建立的递归下降分析的代码

从代码中可以看到，对于非终结符和终结符都分别建立的结点，并进行对应关系的连接，所有的函数都按照这个思路来写，则很容易将 little C 的全部文法进行实现，并得到一颗语法树。

3. 主程序依然从文件中读取代码，进行注释的删除，并将无注释的代码传入词法分析器得到单词表，传入语法分析中，无错误

则得到一颗语法树，返回语法树的根节点；含错误则提示错误，直接返回。

五、 实验结果

六、 实验总结

1. 学习了利用递归下降的思想构造语法分析程序
2. 学习了如何在递归下降的同时生成语法树
3. 掌握了 First 集和 Follow 集的求法
4. 掌握了消除左递归以及二义性的相关知识

七、 附录