实验六 对完整程序的递归下降分析

一、 实验目的

设计一个程序，输入字符串形式的源程序，输出该程序的语法分析树，有错误时报告错误。

二、 实验要求

1. 源程序及其语法规则：可以参考附录 A，也可以自定义
2. 输入为字符串形式的源程序，因此，需要调用前面实验做过的词法分析器，为语法分析器提供单词符号
3. 应该指出错误的具体位置，如：在 XX 单词之后/之前发现错误，分析终止

三、 实验设计

1. 附录文法的改造

这里选择参考附录 A 中 little C 的文法，它由花括号开始，包括以下几种类型的语句：

a. 声明语句。可以对同一个类型的变量进行并列声明，如 int a,b;

b. 赋值语句。左部为标识符，中间为=，右部为表达式

c. 选择语句。分为 if 的单分支语句，以及 if-else 的双分支语句

d. 循环语句。while-do 形式的循环语句

该文法支持花括号的嵌套，以及花括号内的内容为空

```
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

可以看到对于 DECLS、NAMES、STMTS、EXPR、TERM 的产生式都是递归定义的，直接使用将导致无限递归；而且对于选择语句来说，该文法存在二义性，也无法利用递归下降进行分析。所以我们需要将该文法进行改造，改造成能够利用递归下降分析方法进行分析的文法。

对于上述的直接左递归文法，由于我们采用自顶向下的分析方法，所以我们将其改造为右递归文法即可消除递归；对于二义性文法，我们采用改写产生式的方式将其进行改造。

另外注意到一点，对于 FACTOR 的产生式

```
FACTOR→(  BOOL  )  |  id  |  number
```

图 3 FACTOR 的产生式

括号中间的非终结符为 BOOL，而 BOOL 产生式是为了产生选

择语句或循环语句中的条件的；而 FACTOR 的产生式是用于赋值语句计算算数表达式的，BOOL 语句产生的是逻辑值；而算数表达式产生的是数值，为了两者的不混用，导致程序设计的不便，这里将 BOOL 改造为 EXPR。

另外附录中的非终结符 REL 应该改为 BOOL 才符合文法的含义

则改造后，消除左递归和二义性的 little C 文法如图所示：

```
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. 递归下降的分析方法

总体思路：对文法的每一个非终结符都编写一个分析过程，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。

设有全局变量 token，代表输入串的当前单词符号；函数 match(token)，匹配当前单词并读入下一个。从开始符号出发，对每一条产生式：

1） 设 U∈V_N，有产生式$U \to \alpha_1|\alpha_2...|\alpha_m$，定义 P(U)为如下过程：

$begin$

$if\ token \in First(\alpha_1)\ then\ P(\alpha_1);$

$else\ if\ token \in First(\alpha_2) then\ P(\alpha_2);$

$.....$

$else\ error$

end

2） 设 U∈V_N，有产生式$U \to \alpha_1|\alpha_2...|\alpha_m|\varepsilon$，定义 P(U)为：

$begin$

$if\ token \in First(\alpha_1)\ then\ P(\alpha_1);$

$else\ if\ token \in First(\alpha_2) then\ P(\alpha_2);$

$.....$

$else\ if\ token \in Follow(U)\ then\ return;$

$else\ error$

end

3） 设$\alpha = x_1x_2...x_n, x_i \in (V_N \cup V_T)$, $P(\alpha)$代表语句块：

$begin$

$P(x_1);$

$P(x_2);$

$...$

$P(x_n);$

end

其中若$x \in V_N$，则转 1)或 2）；

若$x \in V_T$，P(X)代表语句：

$if\ token = x\ then\ match(token)$;

$else\ error$;

按照上述的方法即可写出文法的递归下降分析的代码。

3. 树结点的建立方法

做完递归下降分析还不够，我们还需要建立一颗语法树将单词按语法规则保存起来，以便后续中间代码生成使用。为了方便后续的操作，这里利用孩子兄弟表示法建立一棵二叉语法树。数据项有两项：结点的名称以及编码。

结点的建立时机：在递归下降分析，当终结符匹配成功时或调用非终结符产生式的相应函数时，进行相关结点的创建，并按照孩子兄弟的方法进行树结点的排列，则递归下降分析完毕，就能够得到一个完整的语法分析树。

四、 重要代码设计

1. 不含树结点建立的非终结符函数的设计

设计时遵循从简单到复杂的设计规则，先编写不含树节点建立的非终结符函数，再将其改造为含树节点建立的函数。

我们可以参考上述递归下降的分析方法，写出相应的代码，这里以不含 empty 的产生式 BLOCK 和含 empty 的产生式 DECLS 为例进行代码的分析

```
//BLOCK→{ DECLS STMTS }
//FIRST(BLOCK) = { { }
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 5 非终结符 BLOCK 的递归下降分析函数

由于不存在 empty 产生式，所以不需要使用 Follow 集，输入匹配只需要看向 First 集即可，当前输入和 First 集中符号对应则视为匹配成功，index++；对于非终结符则直接调用其对应函数；对于终结符则继续进行匹配；匹配不成功则调用 error 报告错误，程序终止。

```
//DECLS → DECL DECLS | empty
//FIRST(DECLS) = {INT}
//FOLLOW(DECLS) = {ID IF ELSE WHILE { } }
void DECLS()
{
    if(word_list[index].s=="int")
    {
        DECL();
        DECLS();
    }
    else if((word_list[index].type>=1&&word_list[index].type<=3)
            ||word_list[index].type==10||word_list[index].type==34)
        return;
    else
        error();
}
```

图 6 非终结符 DECLS 的递归下降分析函数

对于 DECLS 的递归下降函数来说，其存在空产生式，所以当 First 集不匹配时，还需要查看当前输入和 Follow(DECLS)中的符号是否匹

配，如果匹配则说明当前使用的是空产生式，则不需要继续往下处理，直接 return 即可。

其他非终结符递归下降函数的代码和上述函数基本一致，这里不再赘述。

2. 含树节点建立的非终结符函数的设计

写完简单的递归下降代码并调试完后，我们加入树的建立代码，在进行语法分析的同时，进行语法树的建立。首先我们建立树节点结构体：

```
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

然后定义结点的建立时机：在终结符匹配后和非终结符函数调用前，我们需要根据当前非终结符函数对应的产生式进行相关结点的创建和关联。将需要调用的非终结符函数所对应的结点作为该函数的参数传入。在实际代码实现的时候需要区分终结符匹配和仅是为了能够进行调用而进行的 First 集中字符的匹配，对于后一种情况，是不建立结点的。以 DELCS 函数为例子进行说明：

```
TreeNode *  DECLS(TreeNode * p)
{
    if(word_list[index].s=="int")//注意这里是First集的匹配，真正的match在DECL中
    {
        TreeNode * s = new TreeNode("DECL", -1);//只有第一个为孩子
        p->son = s;
        DECL(s);//将该函数对应的树结点作为参数传入
        TreeNode * cur = new TreeNode("DECLS", -1);//其余的都是孩子的兄弟
        s->bro = cur;
        DECLS(cur);
    }
    else if((word_list[index].type>=1&&word_list[index].type<=3)||
         word_list[index].type==10||word_list[index].type==34||word_list[index].type==35)
    {   //对于空产生式的匹配，考虑到后面中间代码的生成，这里建立一个表示空的结点
        TreeNode * n = new TreeNode("null", -2);//表示空节点 type=-2
        p->son = n;
    }
    else
        error(word_list[index].s);
    return p;
}
```

图 6 非终结符 DECLS 的递归下降分析函数

框架不变，在终结符匹配时和函数调用前需要进行结点的建立和连接。对于 DECLS 来说，由于存在空产生式，所以它要对当前的输入字符进行判断，所以需要和它的 First 集进行比对，以便不匹配时作出合理的判断：再运用 Follow 集进行匹配，还是不匹配则报错。

当匹配成功时，对于其需要调用的两个函数 DECL 和 DECLS 来

说，一个作为其孩子，另一个作为孩子的兄弟结点，在 DECLS 函数中建立；若 Follow 集匹配成功，则表示匹配的是空产生式，则建立一个名称为 null 的结点，表示这里进行 return——方便后续中间代码翻译的衔接；若都没有匹配成功，则将当前输入字符传给 error 进行报错，程序退出。

其余函数的编写类似，不再赘述。

3. 语法树的遍历

最后语法分析完成后，我们将得到一个完整的语法树，我们通过遍历语法树的方法，将结点输出，则利用 DFS 深度优先遍历的方法对结点进行输出。考虑到即使是少量的代码，建立的语法树结构也非常庞大，不方便查看输出，所以这里优化一下输出，仅把终结符结点输出，也方便检查语法分析的正确与否。

```
//深度优先搜索遍历二叉树
void DFS(TreeNode * root)
{
    if(root==NULL)
        return;
    if(root->type!=-1&&root->type!=-2)
        cout << root->name << "    " << root->type << endl;
    DFS(root->son);
    DFS(root->bro);
}
```

图 7 遍历二叉树的终结符结点并进行输出

4. 错误的处理

当语法分析遇到错误时，本实验的处理方法是将错误抛出，将失配的当前字符进行输出，并终止语法分析。

```
void error(string e)
{
    cout<<"语法分析错误 "<<e<<"匹配失败"<<endl;
    exit(-1);
}
```

图 8 错误处理代码

5. 主函数的设计

这里将前面实验设计的预处理模块和词法分析模块运用，从文件中读入源代码，进行注释的删除；再将删除注释后的源程序输入语法分析模块得到单词表；将单词表输入到语法分析模块得到语法树；最后遍历语法树验证结果正确与否。

图 9 语法分析器的流程图

五、 实验结果

六、 实验总结

1. 学习了利用递归下降的方法构造语法分析程序
2. 掌握了消除左递归、二义性等文法改造的知识
3. 掌握了 First 集、Follow 集的求法
4. 提高了代码的实践能力，运用编译原理、数据结构等相关知识，完成实验内容
5. 本语法分析器还有进一步改进的空间：首先，非终结符的递归下降分析函数除了 PROG 以外的函数是不需要返回值的，可以优化为 void 类型；其次，由于词法分析中将负号仅当作减号使用，所以在语法分析时，赋值语句中无法使用符号，这是需要进一步解决的问题；最后，对于错误的处理较为粗糙，可以后续考虑将代码的行号进行保存，在返回匹配失败的字符的同时，返回对应的行号，方便阅读错误。

七、 附录