

# 形式语言与编译大作业实验报告

第四组 王嘉禾 苏悦馨 魏欣悦 洪云 卢卓 郭涵伟

## 一、词法分析

### 1. 组合 DFA 的构造

原理：首先明确词法单位，将其区分为名字项（变量名和关键字）、数值（整型数和浮点数）、运算符（算术运算符和比较运算符）、界符（左右圆括号、方括号、分号、逗号）。

对于词法单位，在对比了题目中给出的词法单位和文法之后，我们发现两者存在一定的不统一，因此根据文法对词法单位进行了一些调整，在关键字中去掉了 `input` 和 `print`，并加上了 `void`, `int` 和 `float`。

对于名字项，考虑到关键字和变量名的本质类似，并且 `Scanner()` 通过 C++ 语言实现，故对关键字的判断采取如下方式：在识别出一个名字项后再判断这个名字是不是关键字，即 DFA 只具有读出名字项的功能，区分关键字和变量名则交给程序实现，这是容易的。

对于数值和运算符，DFA 应能实现区分数值前的正（负）号和运算符中的加（减）号。事实上如果对词法单位进行修改，将数值前面的正负号删去，这一步骤也可以交由语法分析完成。而我们选择在词法分析过程中直接进行区分：在设计 DFA 时我们对状态进行了调整，进而可以区分正负号和加減号。

最终得到的组合 DFA 如下表所示。

	B	C	D	E	F	G	H	I	J	K	L	M	N
1							12						
2	状态	数字	字母	加号	负号	乘号	等号	浮点(.)	各种括号	分界符号	<	!	换行符等
3	输出当前缓冲区回到初始状态	1	3										0
4	处理整型数	1		5	7	11	8	2	11	11	9	10	13
5	处理浮点数	2		5	7	11	8		11	11	9	10	13
6	处理变量名	3	3	5	7	11	8		11	11	9	10	12
7	正号	1											4
8	输出缓冲区并输出运算符加号	1	3	4	6		18		17				5
9	负号	1											6
10	输出缓冲区并输出运算符减号	16	3	4	6				17				7
11	输出缓冲区并存入了单个=	1	3	4	6		14		11	11			8
12	输出缓冲区并存入了单个<	1	3	4	6		15		11				9
13	输出缓冲区并存入了单个!						16		11				10
14	输出缓冲区并输出完一个其他界符/运算符	1	3	4	6	17	8	17			9	10	11
15	输出变量名并判断是不是关键字	1	3	5	7	17	8				9	10	12
16	将数字输出缓冲区	1	3	5	7	17							13
17	输出== (或读到换行符,下同)	1	3	4	6	17			17				14
18	输出<=	1	3	4	6	17			17				15
19	输出!=	1	3	4	6	17			17				16
20	输出单个界符或者运算符完成	1	3	4	6	17			17				17
21	输出+=	1	3	4	6	17			17				18

### 2. Scanner()的实现

原理：逐个字符读入输入文件中的字符串，根据 DFA 的转移表进行处理，一旦出现不能识别的字符就报错并且停止程序运行。

程序实现时的主要难点在于加号的处理。根据词法单位，加号既可以作为运算符，也可以作为数字值的前缀表示正负性。考虑到当加号作为运算符的时候，前面的单位一定是数字或者变量名，因此考虑在读入和分析词法单位的时候，对前一个单位的类型进行记录，从而在遇到加号的时候判断该加号是应该按照运算符处理还是正负号处理。

具体方法如下：

辅助标识变量：**errorflag**（判断是否出现不能识别的字符而报错），**pre\_op**（用于记录数字前面的正负号，从而确定数字的正负性），**pre\_type**（记录前一个识别单位的类型，辅助判断加号如何处理）。

对于过滤符（换行、回车、空格、制表符），不做处理，直接跳过处理下一个字符。

对于小写字母，进行连续的抓取；若在一串字母后紧跟着数字，也继续抓取。将最终得到的单词与关键词进行比较，如果不是关键词，则认为是名字项。

对于前面没有紧跟字母的数字，进行连续的抓取，若其后没有小数点字符，则识别为整型，与 **pre\_op** 配合确定该整数的值；如果其后存在小数点字符，继续连续抓取数字，识别为浮点型，与 **pre\_op** 配合确定该浮点数的值。

对于加号，需要根据 **pre\_type** 记录的类型分为两种情况讨论：第一种情况是 **pre\_type** 记录为名字项或者数字，说明加号是运算符，将其识别为求和运算符；第二种情况是 **pre\_type** 不是名字项或数字，由于词法单位中还有+=运算，所以需要查看其下一个字符进一步判断是正负号还是+=运算，查看操作在实现上，先正常读入下一个字符进行判断，再利用 **fseek** 函数将文件读入的 **FILE** 指针回退一位即可。

对于负号，改变 **pre\_op** 的值即可。

对于等号，同样需要查看下一个字符来确定其类型：如果下一个字符也是等号，则将其识别为关系运算符==，否则将其识别为赋值符号。

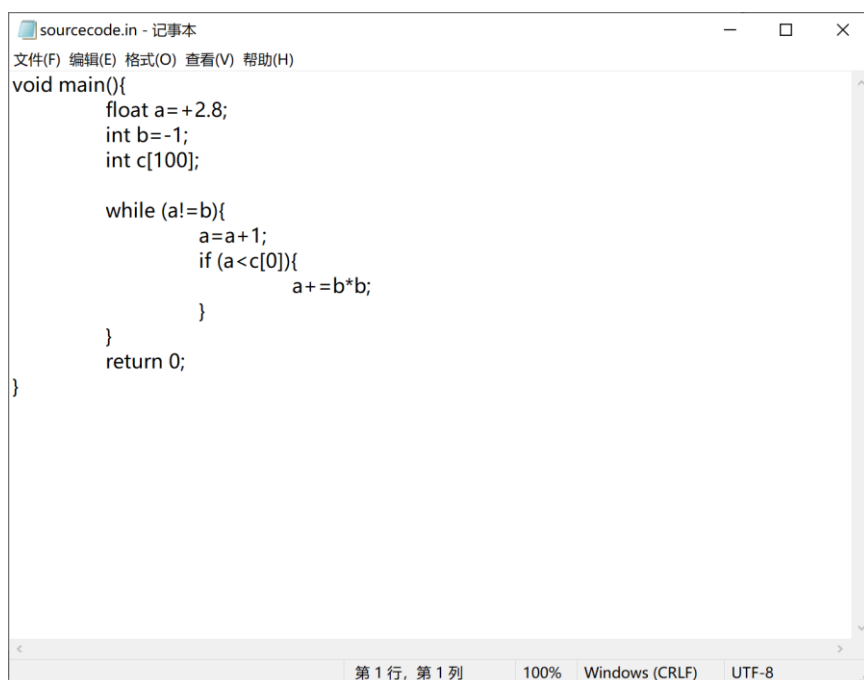
对于感叹号，由于合法的词法单位中其只能和一个等号配合成为关系运算符中的不等号，因此查看一下下一个字符进行识别：如果是等号，将其读入，两个字符一起识别为不等号；否则直接报错。

其他的符号没有特别的操作，只要根据词法单位进行单符号的识别即可。

代码实现：见 `alexcd.cpp`

运行结果演示：

输入文件：`sourcecode.in`



The screenshot shows a Notepad window titled "sourcecode.in - 记事本". The menu bar includes "文件(F)", "编辑(E)", "格式(O)", "查看(V)", and "帮助(H)". The code content is as follows:

```
void main(){
    float a=+2.8;
    int b=-1;
    int c[100];

    while (a!=b){
        a=a+1;
        if (a<c[0]){
            a+=b*b;
        }
    }
    return 0;
}
```

The status bar at the bottom indicates "第 1 行, 第 1 列", "100%", "Windows (CRLF)", and "UTF-8".

输出文件：mstr.in（将词法分析的结果作为后续分析的输入文件）

```
mstr.in - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
void      #
d         # main
(         #
)         #
{         #
float     #
d         # a
=         #
f         # 2.8
;         #
int       #
d         # b
=         #
i         # -1
;         #
int       #
d         # c
[         #
i         # 100
]         #
;         #
while     #
(         #
d         # a
r         # !=
d         # b
)         #
{         #
d         # a
=         #
d         # a
+         #
i         # 1
;         #
if        #

(         #
d         # a
r         # <
d         # c
[         #
i         # 0
]         #
)         #
{         #
d         # a
+         #
=         #
d         # b
*         #
d         # b
;         #
}         #
}         #
return    #
i         # 0
;         #
}         #
#         #
```

## 二、 语法分析

### 1. FIRST 和 FOLLOW 集合的生成

在对文法进行正式的语法分析之前,我们首先生成了文法的非终结符以及每条产生式的 FIRST 集合再通过文法的 FIRST 集合进而求出所有非终结符元素的 FOLLOW 集合,用于后续对 SLR(1) 分析表的消除冲突分析。

原理:

(1) FIRST 集合:

对文法  $G$  的任何符号串  $\alpha = X_1X_2\cdots X_n$  构造集合  $FIRST(\alpha)$

- 置  $FIRST(\alpha) = FIRST(X_1) \setminus \{\epsilon\}$ ;
- 若对任何  $1 \leq j \leq i-1$ ,  $\epsilon \in FIRST(X_j)$ , 则把  $FIRST(X_i) \setminus \{\epsilon\}$  加至  $FIRST(\alpha)$  中; 特别是, 若所有的  $FIRST(X_j)$  均含有  $\epsilon$ ,  $1 \leq j \leq n$ , 则把  $\epsilon$  也加至  $FIRST(\alpha)$  中。显然, 若  $\alpha = \epsilon$  则  $FIRST(\alpha) = \{\epsilon\}$ 。

(2) FOLLOW 集合:

原理: 对于文法  $G$  的每个非终结符  $A$  构造  $FOLLOW(A)$  的办法是, 连续使用下面的规则, 直至每个 FOLLOW 不再增大为止:

- 对于文法的开始符号  $S$ ，置  $\#$  于  $FOLLOW(S)$  中；
- 若  $A \rightarrow \alpha B \beta$  是一个产生式，则把  $FIRST(\beta) \setminus \{\epsilon\}$  加至  $FOLLOW(B)$  中；
- 若  $A \rightarrow \alpha B$  是一个产生式，或  $A \rightarrow \alpha B \beta$  是一个产生式而  $\beta \Rightarrow * \epsilon$  (即  $\epsilon \in FIRST(\beta)$ )，则把  $FOLLOW(A)$  加至  $FOLLOW(B)$  中

结果展示：FIRST 集

```
T: float 'int' 'void'
D: 'void' 'float' 'int'
C: 'void' 'float' 'int' 'e'
S: 'if' '{' 'return' 'while' 'd'
V: 'if' '{' 'return' 'while' 'd'
P: 'if' '{' 'return' 'void' 'e' 'float' 'int' 'while' 'd'
A: 'if' '{' 'return' 'void' 'e' 'float' 'int' 'while' 'd'
I: '}'
G: '}' 'd'
E: '}' 'd'
F: '}' 'd'
B: '}' 'd'

A->P('if' '{' 'return' 'void' 'e' 'float' 'int' 'while' 'd')
P->CV('if' '{' 'return' 'void' 'float' 'int' 'while' 'd')
C->e('e')
C->CD('void' 'float' 'int')
V->CV('if' '{' 'return' 'while' 'd')
V->VS('if' '{' 'return' 'while' 'd')
D->Td('float' 'int' 'void')
D->Td(B: 'float' 'int' 'void')
D->Td(CV: 'float' 'int' 'void')
T->int('int')
T->float('float')
T->void('void')
I->i('}')
I->i('}')
S->d=E: 'd')
S->if(BS: 'if')
S->if(BSelse: 'if')
S->while(BS: 'while')
S->returnE: 'return')
S->V: '}' 'd')
S->dF: 'd')
F->E: '}' 'd')
F->F.E: '}' 'd')
E->E.G: '}' 'd')
E->G: '}' 'd')
G->i: '}'
G->d: 'd')
G->dF: 'd')
G->dF: 'd')
G->G.G: '}' 'd')
B->E: '}' 'd')
B->E: '}' 'd')
```

FOLLOW 集

```
A: {'#'}
P: {'#'}
C: {'if', '{', 'return', 'void', 'float', 'int', 'while', 'd', '}' }
V: {'#', 'if', '{', 'return', '}', 'while', 'd'}
D: {';'}
T: {'d'}
I: {'.', '}' }
S: {'#', 'if', '{', 'return', '}', 'while', 'd', 'else'}
F: {'}', '}', '}' }
E: {'r', '+', '}', '}', '}', '}' }
G: {'r', '+', '}', '}', '}', '}', '*' }
B: {'}' }
```

## 2. LR (0) 规范集的实现

原理：给定 CFG  $G=(V,T,P,S)$ ，构造识别活前缀的 DFA( $Q, \Sigma, u, q_0, F$ )。DFA 的状态是项目集合，初始状态  $q_0=\omega([S' \rightarrow \bullet S])$ 。

项目集  $q$  的  $\epsilon$ -闭包  $\omega(q)$  定义为：如果  $[A \rightarrow \alpha \bullet N \beta] \in \omega(q)$ ，那么  $[N \rightarrow \bullet \gamma] \in \omega(q)$ （其中  $A \rightarrow \alpha N \beta$  和  $N \rightarrow \gamma$  都在文法的产生式  $A \rightarrow \alpha \bullet N \beta$  到  $[N \rightarrow \bullet \gamma]$  有  $\epsilon$ -转移。此外没有别的  $\epsilon$ -转移。

对于  $q \in Q$ ，令  $p=u(q,X)$ ， $X \in V \cup T$ ，那么， $p=\omega(\{[A \rightarrow \alpha X \bullet \eta] \mid [A \rightarrow \alpha \bullet X \eta] \in q\})$ ， $p \in Q$ ，若

过程示意：  
 $p$  仅含完全项目，那么  $p \in F$ 。

PROCEDURE ITEMSETS( $G$ ):

BEGIN

$C:=\{CLOSURE(\{S \rightarrow \bullet S\})\};$

```

REPEAT
    FOR C 中每个项目集 I 和 G 的每个符号 X DO
        IF GO(I, X)非空且不属于 C THEN
            把 GO(I, X)放入 C 族中;
    UNTIL C 不再增大
END

```

代码实现：

通过栈进行实现，从初始状态开始，没进行一步对点后面的项目进行检索，如果是非终结符则进行状态集的扩充循环到不能扩充为止。一个状态结束，创建新状态继续原来的操作直到栈里没有元素。

```

State CLOSURE(State target){
    State newState;
    newState.items = copyItems(target);
    newState.top = target.top;

    Item *itemStack = copyItems(target);
    int stackTop = target.top;

    while(stackTop != 0){
        // 出栈
        stackTop--;
        int idx = itemStack[stackTop].idx;
        Production tempPro = productions[itemStack[stackTop].pld];
        // 判断该 item 是否还能扩展
        if(idx < tempPro.rightPartLength){
            string tempVT = tempPro.rightPart[idx + 1]; // 点右边的第一个字符
            if(isNonTerminal(tempVT)){ // 如果是非终结符
                int* pld = existLeftPro(tempVT);
                for(int i = 0; i < productionTop; i++){
                    int id = *(pld + i);
                    if(id != -1 && !itemRepeat(newState, id, -1)){
                        // 添加新的 item，点标在最左边
                        newState.items[newState.top].pld = id;
                        newState.items[newState.top].idx = -1;
                        // 入栈
                        itemStack[stackTop++] = newState.items[newState.top];
                        newState.top++;
                    }
                }
            }
        }
    }
}

```

}

[illegible]

原理:

1. 若  $a$  是某个  $a_i$ ,  $i=1,2,\dots,m$ , 则移进;

- 方法描述: 通过 STL 库的 set 数据结构, 首先为每个活前缀构造各自的规范文集, 之后一个文集开始, 搜寻每一个转移状态, 如果之前状态没有则插入新状态, 若有则不需要。转移之后在 STL(1)表中填入相应的移进项。所有状态构造结束之后, 再从头开始根据 w 表填入规约项, 有移进规约冲突则优先规约。程序首先输出的是每个活前缀的规范文合并之前), 然后输出 STR (1) 分析表, 最后输出每个状态形成规范集簇。



### 3. 属性文法的构建（按归约式给出）

- (1) **A->P** 无
- (2) **P->CV** 合并两部分代码即可 `code[sp-1]=code[sp-1]+"|"+code[sp]`
- (3) **C->e** 无
- (4) **C->CD;** 合并两部分代码即可 `code[sp-2]=code[sp-2]+"|"+code[sp-1]`
- (5) **V->S** 无
- (6) **V->VS** 合并两部分代码即可 `code[sp-1]=code[sp-1]+"|"+code[sp]`
- (7) **D->Td** 调用符号表操作, **d** 的名字登记为 **T** 的类型
- (8) **D->Td[I]** 调用符号表操作, **d** 的名字登记为 **array**, 元素类型登记为 **T** 的类型, 并存储维数维长等信息 (存储在 **I** 的属性中)
- (9) **D->Td(C){CV}** 由于形参和实参已在符号表中登记 (具体方法在第 4 部分中说明), 并且代码部分 **V** 也已经生成完毕, 因此将 **d** 登记为 **function**, 返回类型为 **T** 的类型, 添加上调取形参的代码即可  
`code[sp-8]=prop[sp-7]+"proc"+"mcode"+"|"+code[sp-1]+"|"+prop[sp-7]+"endp"` (其中 **mcode** 部分为 **pop** 形参名字的代码)
- (10) **T->int** 将 **T** 的类型置为 **int**
- (11) **T->float** 将 **T** 的类型置为 **float**
- (12) **T->void** 将 **T** 的类型置为 **void**
- (13) **I->i** 将 **I** 的维数置为 1, 第 1 维的维长置为 **i** 的数值 (从实际意义来说, **I** 是数值序列, 为数组定义服务)
- (14) **I->I,i** 将 **I[0]** 的维数加一, 且新的维长为 **i** 的数值
- (15) **S->d=E** 赋值语句, 在 **E** 的代码上添加 `d.name=E.place` 即可  
`code[sp-3]=code[sp-1]+"|"+prop[sp-3]+"="+place[sp-1]`
- (16) **S->if(B)S** **B** 中以自带 **tc** 和 **fc**, 分别为布尔表达式成立时跳转下标和不成立时跳转下标, 分别置 **S** 的代码之前和之后即可  
`code[sp-4]=code[sp-2]+"|"+tc[sp-2]+":|"+code[sp]+"|"+fc[sp-2]+":|"+code[sp-4]`
- (17) **S->if(B)SelseS** 和 (16) 的情况基本相似, 不同的是在 **S[1]** 的代码执行完毕后需要跳转到 **S[2]** 之后, 这是一个新的地址标号  
`l=getnewlabel()`  
`code[sp-6]=code[sp-4]+"|"+tc[sp-4]+":|"+code[sp-2]+"|"+l+"|"+fc[sp-4]+":|"+code[sp-6]`
- (18) **S->while(B)S**  
`l=getnewlabel()`  
`code[sp-4]=l+":|"+code[sp-2]+"|"+tc[sp-2]+":|"+code[sp]+"|"+l+"|"+fc[sp-2]+":|"+code[sp-4]`
- (19) **S->returnE** 在 **E** 的代码基础上将 **E.place** 压进栈即可  
`code[sp-2]=code[sp-2]+"|push "+place[sp-1]`
- (20) **S->d(F);** 这里由于 **S** 是一个语句, 不具有 **place** 属性, 因此只调用函数的过程而不保存返回值, 因此若函数的类型不为 **int**, 需要将栈中保存的返回值结果弹出  
`code[sp-4]=code[sp-1]+"|"+mcode+"|call "+prop[sp-4]` (其中 **mcode** 为 **push** 形参名字代码)
- (21) **F->E** 无
- (22) **F->F,E** 将二者代码合并即可  
`code[sp-2]=code[sp-2]+"|"+code[sp];`  
`place[sp-2]=place[sp-2]+"|"+place[sp];`



- (23) **E→E+G** 创建一个新变量，将两个 place 相加即可
- ```
t=getnewvar()
code[sp-2]=code[sp-2]+"|"+code[sp]+"|"+t+"="+place[sp-2]+"+"+place[sp]
place[sp-2]=t;
```
- (24) **E→G** 无
- (25) **G→I** 新建一个变量，赋值为 i 的数值即可
- ```
t=getnewvar()
code[sp]=t+"="+prop[sp]
```
- (26) **G→d** G.place=d.name
- (27) **G→d[F]** 调用数组，首先通过查找符号表得到维数和维长信息，再通过 F 中的所有变量集合可结合维长算出偏移地址，具体代码见源程序 case (27)
- (28) **G→d(F)** 与第 (20) 中情况基本相同，区别在于 G 是为复制服务的，因此如果 d 的类型是 void 不具有返回值，应当报错，并且调用 d 完毕后，应当新建一个变量名保存返回值
- (29) **G→G\*G** 与第 (23) 中情况基本相同，将+改为\*即可
- ```
t=getnewvar()
code[sp-2]+="|"+code[sp]+"|"+t+"="+place[sp-2]+"*"+place[sp]
place[sp-2]=t
```
- (30) **B→ErE** 对于布尔表达式需要创建 tc 和 fc 标号并生成 if 语句
- ```
tc[sp-2]=getnewlabel()
fc[sp-2]=getnewlabel()
code[sp-2]=code[sp-2]+"|"+code[sp]+"|"+t+"if"+place[sp-2]+prop[sp-1]+place[sp]+"th
en jmp "+tc[sp-2]+" else jmp "+fc[sp-2]
```
- (31) **B→E** 和第 (30) 中情况基本相同，区别在于将 ErE 的表达式替换为 E!=0
- ```
tc[sp]=getnewlabel()
fc[sp]=getnewlabel()
code[sp]=code[sp]+"|"+t+"if "+place[sp]+"!=0 then jmp "+tc[sp]+" else jmp "+fc[sp]
```

#### 4. 登记符号表操作的具体实现

登记符号表采用提前登记函数名称的方式实现分层，即开始定义某一函数 d 时，先登记 d 为 function 类型，此时所有的登记都会被视为在 d 之内，当函数声明结束，即出现归约式 **D→Td(C){CV}** 时，调用符号表的 end ( ) 功能，恢复到直接外侧。因此我们采取提前声明函数的方式，在每次移进动作之后，当到达项目 **D→Td(·C){CV}** 所在的项目集状态 (41 号状态) 时，视为开始定义函数，从这时起所有定义变量登记符号表时均置为“形参”。当到达项目 **D→Td(C){·CV}**，视为开始定义函数中的实参，从这时开始所有变量登记符号表时均置为“实参”。通过这种方式实现了归约出定义式时立刻登记，可节省存储空间，简化代码，并且，这两种状态也只可能通过函数声明到达。

```
if (state[sp]==41)
{
    if (prop[sp-2]=="void") globaltab.regis(prop[sp-1],0); else globaltab.regis(prop[sp-1],1);
    //cout << "function " << prop[sp-1] << " registered\n";
    vartype=1;
}
else if (state[sp]==67 && inputsym[ip]=="") vartype=0;
```

#### 5. 活动记录操作代码的生成

程序运行结束后给出对活动记录操作的规则，当出现调用函数的归约式 (20 号和 28 号归约串)，则生成该函数的活动记录，这里调用了符号表的查找特定函数的形参个数，

实参个数，类型，和嵌套层数等信息，可以很方便地实现生成操作。

(图片代码仅为构建参数区和控制链部分,完整代码见 slr1.cpp 中 genactionrecord 函数)

```
arip++;
arcode[arip]="\n//"+funcname+"(start)";
pos=globaltab.look(funcname);
pt=globaltab.nametab[pos].ref;
kt=globaltab.btab[pt].lastpar;
while(kt!=-1)
{
    arip++;
    arcode[arip]="sp++;";
    arip++;
    arcode[arip]="M[sp]="+globaltab.nametab[kt].name+";";
    kt=globaltab.nametab[kt].link;
}
if(globaltab.nametab[pos].type!=0)
{
    arip++;
    arcode[arip]="sp++;";
    arip++;
    arcode[arip]="M[sp]="+funcname+"@value;";
}
```

## 6. SLR1 分析过程的实现

Sp 指针指向状态栈顶，ip 指向输入串当前位置，每一次操作，用最顶状态和当前输入符号查询 action 表，获得 act (pair 类型)，第一项为操作类型，第二项为对应的附属信息 (对于移进操作，为新的状态编号，对于归约操作，为所用归约式编号)

```
ch=inputSYM[ip];
tpro=inputpro[ip];
act=action[state[sp]][ch];
```

(其中 inputSYM 为输入串，inputpro 为变量名，数据或类型等基础属性，act 为操作)

- (1) 若为接受项目遇到“#”，则输出“done”，跳转到中间代码输出 printcode () 函数，程序结束
- (2) 若没有操作，则输入串的出现是出错，输出 error，结束程序
- (3) 若为移进操作，则 ip+1, sp+1, state[sp]=act.second(刚查找的新的状态) sym[sp]=ch (此次输入符号)
- (4) 若为归约操作，则进行两个步骤。先根据所用的归约式编号进行归约，主要任务在属性的计算 (详见代码 case (0) 至 case (31) 的部分，分别对应 32 条归约式) 归约之后，根据弹栈之后栈顶的状态和归约所得的非终结符查找 goto 表，得到新的栈顶状态

```

state[sp]=gt[state[sp-1]][sym[sp]];
if (state[sp]==-1)
{
    cout << "state error\n";
    break;
}

```

## 7. 实验结果（示例）

| sourcecode.in - 记事本                                                                                                                                                 | code.out - 记事本                                                                                                                                                                        | actionrecordcode.out - 记事本                                                                                                                                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 文件 编辑 查看                                                                                                                                                            | 文件 编辑 查看                                                                                                                                                                              | 文件 编辑 查看                                                                                                                                                                                                                            |
| <pre> int q(int m;) {     m=m+1;     return m; }; int p(int m;int n;) {     int t;     t=m+q(n);     return t; }; int a; int b; int c; a=a+1; b=b+1; c=p(a); </pre> | <pre> q proc pop m T0=1 T1=m+T0 m=T1 push m q endp p proc pop n pop m push n call q pop T2 T3=m+T2 t=T3 push t p endp T4=1 T5=a+T4 a=T5 T6=1 T7=b+T6 b=T7 push a call p pop T8 </pre> | <pre> //q(start) sp++; M[sp]=m; sp++; M[sp]=q@value; sp++; M[sp]=M[fp-1]; sp++; M[sp]=fp; fp=sp; sp++; M[sp]=code@address; sp=sp+2; sp--; M[sp]=m;  //q(finish) fp=M[fp]; sp=sp-6;  //p(start) sp++; M[sp]=n; sp++; M[sp]=m; </pre> |
| 行 1, 列 1                                                                                                                                                            | 行 1, 列 1                                                                                                                                                                              | 行 1, 列 1                                                                                                                                                                                                                            |

| sourcecode.in - 记事本                                                            | code.out - 记事本                                                                                                                                                  | actionrecordcode.out - 记事本                                                                                                  |
|--------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| 文件 编辑 查看                                                                       | 文件 编辑 查看                                                                                                                                                        | 文件 编辑 查看                                                                                                                    |
| <pre>int a; int b; int c; int t[10,10,10]; a=1; b=1+2*3; c=t[1,2,3]+a+b;</pre> | <pre>T0=1 a=T0 T1=1 T2=2 T3=3 T4=T2*T3 T5=T1+T4 b=T5 T6=1 T7=2 T8=3 T9=T6 T10=T9*10 T11=T10+T7 T12=T11*10 T13=T12+T8 T14=t[T13] T15=T14+a T16=T15+b c=T16</pre> | <pre>  //global M[0]=NULL; M[1]=NULL; M[2]=0; fp=1; sp=2; sp++; M[sp]=t; sp++; M[sp]=c; sp++; M[sp]=b; sp++; M[sp]=a;</pre> |
| 行 1, 列 1                                                                       | 行 1, 列 1                                                                                                                                                        | 行 1, 列 1                                                                                                                    |

| sourcecode.in - 记事本                                                                                                     | code.out - 记事本                                                                                                                                                | actionrecordcode.out - 记事本                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int r(int a;int b;) {     if (a!=b) return 1; else return 0; }; int p; int q; int ans; p=1; q=2; ans=r(p,q);</pre> | <pre>r proc pop b pop a if a!=b then jmp L0 else jmp L1 L0: push T0 jmp L2 L1: push T1 L2 r endp T2=1 p=T2 T3=2 q=T3 push p push q call r pop T4 ans=T4</pre> | <pre>//r(start) sp++; M[sp]=b; sp++; M[sp]=a; sp++; M[sp]=r@value; sp++; M[sp]=M[fp-1]; sp++; M[sp]=fp; fp=sp; sp++; M[sp]=code@address; sp=sp+3; sp--; M[sp]=b; sp--; M[sp]=a; sp=sp+1;  //r(finish) fp=M[fp]; sp=sp-8;</pre> |
| 行 1, 列 1   100%                                                                                                         | 行 1, 列 1   100%                                                                                                                                               | 行 1, 列 1   100%                                                                                                                                                                                                                |

(附件中给出了若干样例，置于输入文件 **sourcecode.in** 执行 **slr1.cpp**，中间代码输出在 **code.out** 中，活动记录操作代码输出在 **actionrecordcode.out** 中)

## 四、 符号表实现

原理描述

### 1) 符号表总述:

为每个函数创建一个符号表，符号表内登记该函数的形式参数和局部变量(包括所有的名字)。主程序看做一个大的函数。

### 2) 符号表框架:

- 程序中所有的名字登记在 **nametab** 表中。登记在 **nametab** 表中的数组名有 **ref** 指针指向数组内情表 **atab**；登记在 **nametab** 表中的函数名有 **ref** 指针指向函数内情表 **btab**；
- 登记在 **nametab** 表中的所有名字都有 **lev** 记录其当前嵌套层级，**outer** 记录其上一级嵌套层级，并有 **link** 指针指向同一层上一个定义的名字。循着 **outer** 指针可以找到该定义的所有嵌套外层；循着 **link** 指针可以找到同一层所有的名字。
- 函数内情表 **btab** 管理全体名字登记表 **nametab**；函数嵌套表 **display** 管理函数内情表 **btab**。

### 3) 符号表各登记项详述:

- 普通变量、形参：登记变量的名字 **name**，类型（**int** 型和 **float** 型）**type**，当前嵌套层次 **lev**，定义的直接外层的嵌套层次 **outer**，是否为形参（**para**），相对于该函数的栈帧的偏移地址，指向同一层级上一个名字的指针 **link**

- ii. 数组：登记数组的名字 **name**，类型（**int** 型和 **float** 型）**type**，当前嵌套层次 **lev**，定义的直接外层的嵌套层次 **outer**，是否为形参（**para**），相对于该函数的栈帧的偏移地址，指向同一层级上一个名字的指针 **link**，指向数组内情表的指针 **ref**
- iii. 函数：登记函数的名字 **name**，函数返回类型（**int** 型和 **float** 型）**type**，当前嵌套层次 **lev**，定义的直接外层的嵌套层次 **outer**，函数内定义的所有静态变量所需要开辟栈帧的大小，指向同一层级上一个名字的指针 **link**，指向函数内情表的指针 **ref**

## 五、方法描述

### 1) 一遍扫描创建符号表方法总述：

- i. 由于该文法允许定义嵌套，故在自下而上一遍扫描翻译程序时，如果要一遍扫描的同时创建符号表，则需要准确记录各名字之间的嵌套层级关系。
- ii. **display** 表和 **btabs** 表同时也是管理表。**display** 表中以函数的定义出现为标志，不断更新记录当前的嵌套层级。**display** 表中的每一个嵌套层级，对应一个函数定义，也即对应 **btabs** 表中的一行。
- iii. **btabs** 表中的 **lastpar** 指针指向该嵌套层级在 **nametab** 表中定义的最后一个形式参数，**last** 指针指向该嵌套层级在 **nametab** 表中定义的最后一个名字。

### 2) 符号表登记方法 **regis()**

**regis()**函数有三个重载，分别为在符号表中登记变量，数组，函数  
符号表登记时，如需创建内情表，同时更改内情表指针。

### 3) 符号表查找方法 **look()**

```
int look(string n); // 查找有没有定义，没有定义则返回-1，有定义则返回其在符号表nametab中的位置
```

符号表查找时，先在本嵌套层次沿 **link** 查找。若本嵌套层次未查找到，沿着 **display** 表的当前嵌套层次记录循外层嵌套层级查找。

### 4) 符号表输出方法 **out()**

```
void out(string n); // 传入函数的名字，打印该函数对应的符号表的所有信息
```

结果示例

如下代码示例，生成的符号表为：

```
int P()
    int a,b;
    int P1 (int i1,int j1);
    int c,d
    ...
end;
int P2 (int i2,int j2)
    int a[4,5,6,7];
    int P21();
    float b1,b2
```

```
void regis(string n, int t, int nor); // 登记变量，传入参数依次为名称，类型（0为int,1为float），是否为形参（为形参则为1，其他为0）
void regis(string n, int t, int nor, vector<int> par); // 登记数组，调用时机为整个数组都翻译完时
// 传入参数依次是数组名，数组元素类型（规定同上），是否为形参（规定同上），以及数组维数列表
void regis(string n, int t); // 登记函数，调用时机应该是开始规约函数内的形式参数时。即T d（出现时调用

return b1
return 0
return 0
```

```

1 func_name      func_return_type  func_level      func_outer      size_of_frame
2 p              int                0               -1              4
3
4 variables and parameters:
5 label  name    kind  level  outer  type  para?  ref  seg  link
6 8      p2      func  1      0      int   3      3    1684 3
7 3      p1      func  1      0      int   2      8      2
8 2      b       var  1      0      int   -1     2      1
9 1      a       var  1      0      int   -1     0     -1
10

```

```

1 func_name      func_return_type  func_level      func_outer      size_of_frame
2 p1             int                1               0              8
3
4 variables and parameters:
5 label  name    kind  level  outer  type  para?  ref  seg  link
6 7      d       var  2      1      int   -1     6      6
7 6      c       var  2      1      int   -1     4      5
8 5      j1      var  2      1      int   par    -1     2      4
9 4      i1      var  2      1      int   par    -1     0     -1
10

```

```

1 func_name      func_return_type  func_level      func_outer      size_of_frame
2 p2             int                1               0             1684
3
4 variables and parameters:
5 label  name    kind  level  outer  type  para?  ref  seg  link
6 12     p21     func  2      1      int   4      8     11
7 11     a       array 2      1      int   0      4     10
8 10     j2      var  2      1      int   par    -1     2      9
9 9      i2      var  2      1      int   par    -1     0     -1
10

```

```

1 func_name      func_return_type  func_level      func_outer      size_of_frame
2 p21            int                2               1              8
3
4 variables and parameters:
5 label  name    kind  level  outer  type  para?  ref  seg  link
6 14     b2      var  3      3      float -1     4     13
7 13     b1      var  3      3      float -1     0     -1
8

```

符号表全体

```

2 variables and parameters:
3 label  name    kind  level  outer  type  para?  ref  seg  link
4 0      p       func  0      -1     int   1      4     -1
5 1      a       var  1      0      int   -1     0     -1
6 2      b       var  1      0      int   -1     2      1
7 3      p1      func  1      0      int   2      8      2
8 4      i1      var  2      1      int   par    -1     0     -1
9 5      j1      var  2      1      int   par    -1     2      4
10 6      c       var  2      1      int   -1     4      5
11 7      d       var  2      1      int   -1     6      6
12 8      p2      func  1      0      int   3      1684 3
13 9      i2      var  2      1      int   par    -1     0     -1
14 10     j2      var  2      1      int   par    -1     2      9
15 11     a       array 2      1      int   0      4     10
16 12     p21     func  2      1      int   4      8     11
17 13     b1      var  3      3      float -1     0     -1
18 14     b2      var  3      3      float -1     4     13

```

# 五、目标程序的内存映像格式

目标程序的内存映像格式如右图所示  
分为代码段，数据段，未初始化数据段和堆

代码段存放程序翻译后的目标代码

数据段和未初始化数据段存放定义的变量

栈运行时使用

堆在运行时动态分配内存使用

