# pl0 实验报告

## 一．任务分工

**基础部分**

我们小组本来一共 4 人，分别为梁聪、罗李媛、雷玓思、方林涛，罗李媛和雷玓思负责要求（1）、（4）和（5）中 return 的实现，方林涛负责要求（3），梁聪负责要求（2）和（5）中的剩余项目。

实际工作中，在我们都互相有 qq 联系的情况下，方林涛从未参与小组见面讨论，也未接受我们的 github 邀请链接，即未完成任何实际工作。最终罗李媛和雷玓思任务不变，梁聪完成（2）、（3）和（5）中剩余项目。

**提高拓展**

梁聪实现了（13），即单个变量和变量数组均可初始化，且可在语句块“{}”内定义局部常量/变量。

罗李媛实现了（6）和一部分（12），print 带间隔输出，random 函数在编译时决定数值，switch 未能在无 break 的情况下顺序执行到底。

其他人未再贡献代码。

## 二．设计框架

① 词法分析：复合符号的词法如下

id → letter(letter|digit)*

number → digit(digit)*

letter → A|B|C|...|Z|a|b|...|z

digit → 0|1|...|9

其中 id 中的 letter 区分大小写。

② 语法分析：Procedure := Block “.”

```
Block := {(ConstDeclaration “;”)
          |(VarDeclaration “;”)
          |(ProcedureDeclaration “;”)}
          Statement
ConstDeclaration := “const” (SingleInitial | ArrayInitial){“,” (SingleInitial | ArrayInitial)}
VarDeclaration := “var” (id | SingleInitial | Array | ArrayInitial)
                        {“,” (id | SingleInitial | Array | ArrayInitial)}
ProcedureDeclaration := “procedure” id “(“ Arguments “)” Block
SingleInitial := id “=” (number | const)
ArrayInitial := Array “=” Initializer
Array := id “[“ (number | const) “]” {“[“ (number | const) “]”}
Initializer := “{“ initializer {“,” initializer} “}”
             | number | const
Arguments := id {“,” id}
```

不带双引号的 const 表示已定义的常量或常量数组中的一个元素

```
Statement := Chunk | Top_expr “;” |“exit” “;” | “return” [Top_expr] “;”
             | Print_statement | If_statement | For_statement | While_statement
             | Do-While_statement | Switch_statement
Chunk := (“begin” | “{”) {ConstDeclaration “;”| VarDeclaration “;”}
```

{Statement} (“end” | “}”)
If_statement := “if” “(“ Top_expr “)” Statement [“else” Statement]
For_statement := “for” “(“ Top_expr “;” Top_expr “;” Top_expr “)” Statement
While_statement := “while” “(“ Top_expr “)” Statement
Do-While_statement := “do” “{“ Statement “}” “while” “(“ Top_expr “)” “;”
Switch_statement := “switch” “(” var “)” “{” Case_list “}”
Case_list := {“case” number “:” Statement}
Print_statement := “print” “(“ [Top_expr {“,” Top_expr}] “)” “;”
Top_expr := id “:=” Top_expr | Or_expr
Or_expr := Or_expr “||” And_expr | And_expr
And_expr := And_expr “&&” Bitor_expr | Bitor_expr
Bitor_expr := Bitor_expr “|” Bitxor_expr | Bitxor_expr
Bitxor_expr := Bitxor_expr “^” Bitand_expr | Bitand_expr
Bitand_expr := Bitand_expr “&” Rel_expr | Rel_expr
Rel_expr := Expression (“=” | “!=” | “>” | “>=” | “<” | “<=”) Expression
Expression := Term (“+” | “-”) Term
Term := Factor (“*” | “/” | “%”) Factor
Factor := id | Array | number | “(“ Top_expr “)”
| (“-” | “!” | “~”) Factor
| ProcedureCall
| Random “(” {number} “)”
ProcedureCall := id “(“ [Top_expr {“,” Top_expr}] “)”

③ 语法特点说明：

- 整个程序以“.”结束，子过程以“;”结束，这保留了最初版本 pl0 的特点；
- 修改原有 block 使得可以读入任意顺序、数量不超过符号表长度的符号定义，所有子过程均认为有返回值，不显式使用返回语句时默认返回值为 0；
- 对数组维度的定义和常量/变量的 initializer 可以使用数字，也可以使用已定义的常量；
- Statement 中去掉了赋值语句，赋值被作为表达式放入下一层（Top_expr），修改原有的 if 语句、while 语句和 begin-end 语句，新添加 for 循环语句、程序退出（exit）语句、过程返回（return）语句、表达式（Top_expr）语句、输出（print）语句、Do-While 表达式、switch 表达式；
- 所有条件判断以非 0 为 true，0 为 false，所有表达式均有 1 个返回值（放在运行栈栈顶）；
- 赋值表达式（见 Top_expr）支持右递归；
- 由“{}”包裹起来的局部变量/常量定义以及 0 至多条语句被视为语句块，begin 等同于“{”，end 等同于“}”以向旧版本 pl0 兼容，可用内含 0 条语句的“{}”或 begin-end 代表空语句；
- 不进行数组越界检查，当下标（[...]）数量少于定义时维数时默认之后的下标为 0（等同于[0]），调用常量数组时只能以数字或常量作下标；
- 赋值时不再自动输出，需要显式使用 print 语句；

④ 亮点：

- ❖ 实现了常量数组，可用已定义常量代替数字作为 initializer；
- ❖ 空语句不是‘;’而是内部不含任何语句的“{}”或 begin-end；
- ❖ 以“return;”简化了“return 0;”，以“{}”简化 begin-end，if 语句简化掉“then”；
- ❖ 添加了位运算符‘&’，‘|’，‘~’，‘^’；

# 三．具体实现

**（1）添加注释功能**

参照老师的改法，在 getsym()中读入除法符号时对下一个字符进行判断，如果是‘/’则跳过当前行，是‘*’则跳过后续符号直到按顺序连续读入‘*’和‘/’。

```
165     else if (ch == '/')
166     {
167         getch();
168         if (ch == '/') { cc = ll; getch(); getsym(); } // line comment
169         else if (ch == '*') { // block-comment
170             getch();
171             while (1) {
172                 if (ch == '*') {
173                     getch();
174                     if (ch == '/') break;
175                 }
176                 getch();
177             }
178             getch();
179             getsym();
180         }
181         else sym = SYM_SLASH;
182     }
```

**（2）拓展“条件”**

添加新的运算符需要在 pl0.h 的`enum symtype{ }`中加入新符号，然后将单字符运算符和其枚举类型对应加入 csym 和 ssym，多字符运算符则需要在 getsym()中为其增加“else if”条目。在将不等“<>”替换为“!=”后，运算符 &&、||、！都需要在 getsym()中添加“else if”条目，而我新增了一些位运算符和括号。添加的符号有“!=”、“>”、“<”、“&&”、“||”。

```
int ssym[NSYM + 1] =
{
    SYM_NULL, SYM_PLUS, SYM_MINUS, SYM_TIMES, SYM_SLASH,
    SYM_LPAREN, SYM_RPAREN, SYM_EQU, SYM_COMMA, SYM_PERIOD, SYM_SEMICOLON,
    SYM_BITXOR, SYM_MOD, SYM_BITNOT, SYM_LBRKET, SYM_RBRKET, SYM_BEGIN, SYM_END
};

char csym[NSYM + 1] =
{
    ' ', '+', '-', '*', '/', '(', ')', '=', ',', '.', ';','^','%','~','[',']','{','}'
};
```

进行条件判断时数据位于运行栈栈顶，将 interpret 语句中对 JPC 的解释修改为

```
case JPC:
    if (stack[top] == 0)
        pc += i.a-1;
    top--;
    break;
```

即可，修改后可将所有表达式（Top_expr）作为条件，非 0 则为真。

短路运算并未加入新指令，用原有 JPC 指令实现。条件作短路判断时，产生条件的是 Bitor_expr 及更下级的表达式，进行判断动作的是 And_expr 或 Or_expr，因此在进行判断时条件已位于栈顶。只需在 And_expr 和 Or_expr 中用局部变量记录待回填的跳转指令就行，其中 And_expr 中判断栈顶为 0 则跳，Or_expr 判断栈顶非 0 则跳。

```
900     void or_expr(symset fsys)
901     {
902         symset set;
903         cxlist cx0, p;
904         int cx1, flag = 0;
905
906         cx0 = p = (cxlist)malloc(sizeof(struct cxnode));
907         p->next = NULL;
908         set = uniteset(fsys, createset(SYM_OR, SYM_NULL));
909
910         and_expr(set);
911         while (sym == SYM_OR)                //2017.10.24 , 2017.10.26
912         {                                                .
913             flag = 1;
914
915             gen(OPR, 0, 14);  // stack[top]=!stack[top];
916             p = p->next = (cxlist)malloc(sizeof(struct cxnode));
917             p->next = NULL;
918             p->cx = cx;
919             gen(JPC, 0, 0);  // short path
920
921             getsym();
922             and_expr(set);
923         } // while
924         if (flag)
925         {
926             gen(OPR, 0, OPR_NOT);
927             gen(OPR, 0, OPR_NOT);
928             cx1 = cx;
929             gen(JMP, 0, 0);    //if did not JPC, avoid restoring
930             while (cx0->next)
931             {
932                 p = cx0;
933                 cx0 = cx0->next;
934                 free(p);
935                 code[cx0->cx].a = cx-(cx0->cx);
936             }
937             free(cx0);
938             gen(LIT, 0, 1);          //if JPC, restore 1
939             code[cx1].a = cx-cx1;
940         }
941
942         destroyset(set);
943     } // or expr
```

And_expr 的短路逻辑和上图的主要区别就是没有 915 行的取反操作。其中记录短路跳转指令号的结构体如下：

```
typedef struct cxnode{
    int cx;
    struct cxnode *next;
}*cxlist;
```

**（3）添加数组**

使用宏定义`#define MAXDIM 100 // maximum array dimensions`规定最大维度，在符号表中加入 int *dim 域使用整型数组存储维界信息，将符号表中记录常量数值的 int value 换成 int *value 数组。

```
typedef struct
{
    char name[MAXIDLEN + 1];
    int  kind;
    int  *value;
    short empty;    //no use
    int  *dim;    //dim of const array
} comtab;

comtab table[TXMAX];

typedef struct
{
    char  name[MAXIDLEN + 1];
    int   kind;
    short level;
    short address;
    short prodn;
    int   *dim;     //dim of var array
} mask;
```

在定义常量和变量时判断下一个符号，如果是‘[’则调用 arraylength()读入数组维度定义存入整型数组 td，计算维界与总空间大小。td 为全局变量，后面将标识符加入符号表时 td 指向的空间交给 dim。arraylength()的代码不再贴出。

```
int *td;    //temporary dimensions
```

常量数组读完维度还要用 initialize()函数读入以“{}”包裹的初始化数据（initializers）存放于全局数组 value 中然后才加入符号表，初始化过程中用到全局变量 vx 作为 value 下标，变量数组读完维度后根据下一个符号是不是 SYM_EQU 判断有没有 initializers，有则调用 initialize()。对于多维数组采用与 c 语言相同的嵌套“{}”格式的 initializers，若 initializers 个数不足以填满数组，则以 0 填充，定义语法参考如下：

```
const K = 4;
const k[3][2][K]={2, 3, K, 5, 6, 7, 8, 9, 10};
```

对 enter 也做了一定修改。

```
// enter object(constant, variable or procedre) into table.
void enter(int kind)
{
    mask* mk;

    tx++;
    strcpy(table[tx].name, id);
    *id = '\0';    //reset to empty char array
    table[tx].kind = kind;
    switch (kind)
    {
    case ID_CONSTANT:
        table[tx].value = value;
        value = NULL;
        table[tx].dim = td;
        td = NULL;
        break;
```

```
576         case ID_VARIABLE:
577             mk = (mask*)&table[tx];
578             mk->level = level;
579             mk->address = dx;
580             mk->dim = td;
581             td = NULL;
582             break;
583         case ID_PROCEDURE:
584             mk = (mask*)&table[tx];
585             mk->level = level;
586             break;
587     } // switch
588 } // enter
```

变量空间分配的 dx 在 vardeclaration()里按 arraylength()返回的数组大小修改。

数组调用时常量数组在编译时直接计算偏移量，从符号表中取出数值，和单个常量一样使用。变量数组在程序运行时计算偏移并放到栈顶，用新加入的指令 STA 和 LDA 代替 STO 和 LOD，根据栈顶偏移量和 STA/LDA 中的基址信息存取数据。

**（4）参数传递**

过程（即函数）定义时必须在 id 后面紧跟“()”，内含 0 至多个以‘,’分隔的形参。先读入形参再进入子过程的 block。

```
1380         if( sym == SYM_PROCEDURE )
1381         {
1382             int k;
1383             char argumentID[TXMAX][MAXIDLEN+1]={0},proceID[MAXIDLEN+1]={0};
1384
1385             getsym();
1386
1387             if( sym == SYM_IDENTIFIER )
1388             {
1389                 enter(ID_PROCEDURE);
1390             }
1391             else
1392             {
1393                 error(4);
1394             }
1395
1396             savedTx = tx;    //critical
1397             level++;         //critical
1398
1399             getsym();
1400             if( sym == SYM_LPAREN )
1401                 getsym();
1402
1403             else error(16);
```

```
1404             prodn = 0;
1405             if(sym == SYM_IDENTIFIER ){
1406
1407                 do{
1408                     strcpy(argumentID[prodn++], id);  //记录参数名
1409                     getsym();
1410                     if( sym == SYM_COMMA)
1411                         getsym();
1412
1413                 }while(sym == SYM_IDENTIFIER);//处理完所有参数
1414             }
```

```
1415                //printf("prodn = %d\n",prodn);
1416                k = prodn;
1417                while(k)
1418                {
1419                    dx = -k;      //b指向静态链，则实参dx为负数
1420                    strcpy(id, argumentID[prodn-(k--)]);
1421                    enter(ID_VARIABLE);
1422                }
1423
1424                if( sym == SYM_RPAREN )
1425                    getsym();
1426                else error(22);//缺少右括号
1427
1428                set1 = createset(SYM_SEMICOLON, SYM_NULL);
1429                set = uniteset(set1, fsys);
1430                block(set);
1431                destroyset(set1);
1432                destroyset(set);
1433                tx = savedTx;
1434                level--;
1436            if (sym == SYM_SEMICOLON)
1437            {
1438                getsym();
1439            }
1440            else
1441            {
1442                error(5); // Missing ',' or ';'.
1443            }
1444        } // if
1445        dx = block_dx; //restore dx after handling procedure declaration!
```

形参在符号表中与普通变量区别是形参的偏移地址依次置为负值，其他特征与未初始化的普通变量相同。最终过程的空间分配（LIT）只需考虑局部变量。

将原有“call”关键字删除，以“id(...)”进行过程调用，如果实参与形参数量不匹配则报错。Factor()读到过程 id 则调用 proceCall()，先依次计算实参放在栈顶，然后压静态链、动态链和 pc，然后进入被

```
995   int prodn;
996   ////////////////////////////////////////////////////////////////////
997   void proceCall(mask* mk,symset fsys){ // procedure call, critical
998           int j;
999           symset set,set1;
1000          //printf("proceID = %s, prodn = %d\n",mk->name,mk->prodn);
1001          set1 = createset(SYM_RPAREN,SYM_COMMA,SYM_NULL);
1002          set = uniteset(set1,fsys);
1003
1004          if( sym == SYM_LPAREN ){
1005              j = 0;
1006              do{
1007                  getsym();
1008
1009                  if(sym == SYM_RPAREN)
1010                  {
1011                      break;
1012                  }
1013                  top_expr(set);
1014                  j++;
1015
1016              }while(sym == SYM_COMMA && j <= mk->prodn);
1017              if(sym == SYM_RPAREN && j <= mk->prodn)
1018              {
1019                  getsym();
1020              }
1021              else
1022              {
1023                  error(22); //missing ')'
1024              }
1025          }
```

调过程。原 CAL 指令被拆成 DIP 和 CAL 两条指令，为后续更复杂功能作准备。

```
1026            else
1027                error(16);  //'(' expected
1028
1029
1030            destroyset(set);
1031            destroyset(set1);
1032            if(j != mk->prodn)
1033            {
1034                error(34); //"The number of actual parameters and virtual paramet
1035                if(j > mk->prodn){
1036                    set = createset(SYM_RPAREN,SYM_NULL);
1037                    set1 = createset(SYM_RPAREN,SYM_SEMICOLON,SYM_END,SYM_NULL);
1038                    test(set,set1,0);
1039                    destroyset(set);
1040                    destroyset(set1);
1041                }
1042            }
1043
1044            gen(DIP,level-mk->level,mk->prodn);
1045            gen(CAL, 0, mk->address);
1046    }//proceCall
```

**（5）exit、return、for 语句和 else 子句**

“exit;”产生新指令“EXT 0 0”将 pc 置零并输出`printf("Exit Program");`。

“return”先把返回值放栈顶，然后利用记录当前所在过程参数数量的全局变量 prodn，将栈顶元素放在第一个实参的位置然后调整 top、b 和 pc。

Statement 中：

```
1262    else if (sym == SYM_RET) {
1263        getsym();
1264        if (sym == SYM_SEMICOLON) {//return; 则把0放在被调用的栈顶，然后再放到原栈栈顶
1265            gen(LIT, 0, 0);
1266            gen(OPR, prodn, OPR_RET);
1267            getsym();
1268        }
1269        else {//return 1;return 1+x;return fact(n-1);
1270            top_expr(fsys);//调用后的结果存在栈顶
1271            gen(OPR, prodn, OPR_RET);
1272            if (sym != SYM_SEMICOLON)
1273                error(10);// missing ';'.
1274            else
1275                getsym();
1276        }
1277    }
```

Interpret 中：

```
case OPR_RET: //put the return value in stack[b-i.l],since the argument isn't useful anymore,
    pc = stack[b+2];
    stack[b-i.l] = stack[top];
    //printf("ret %d from [%d] to [%d]\n",stack[top],top,b-i.l);
    top = b-i.l;
    b = stack[top + i.l + 1];
    break;
```

else 子句很简单，直接在处理 if 时最后回填条件转移指令（JPC）时判断下一个符号就行。参考 c 语言的语法，我没有单独做 elif，使用 else if 可以达到相同效果。

```
if (sym == SYM_ELSE)                          //2017.10.22
{
    gen(JMP, 0, 0);
    code[cx1].a = cx-cx1;  //code[cx1] = JPC, 0, cx-cx1
    cx1 = cx - 1;          //cx1 = JMP,0,0
    getsym();
    statement(fsys);
    code[cx1].a = cx-cx1;
}
else
{
    code[cx1].a = cx-cx1;      //cx1 = JPC,0,0
}
```

For 语句处理时用临时空间将( Top_expr ; Top_expr ; Top_expr )中第三个 Top_expr 产生的语句保存，最后处理完循环体内的 statement 后再将保存的语句接在其后就行。实现 For 主要的工作量是修改程序整体的语法结构，将赋值语句改成赋值表达式，以及将 JMP 和 JPC 都改成相对地址跳转。

```
1143        else if (sym == SYM_FOR)  //2017.10.25
1144        { // for statement
1145            int i;
1146
1147            getsym();
1148            if( sym != SYM_LPAREN)
1149            {
1150                error(16);   // '(' expected
1151            }
1152            else
1153            {
1154                getsym();
1155                set1 = createset(SYM_SEMICOLON, SYM_RPAREN, SYM_NULL);
1156                set = uniteset(set1, fsys);
1157                top_expr(set);
1158                if(sym != SYM_SEMICOLON)
1159                {
1160                    error(10);   //';' expected
1161                    test(fsys,set1,28);   //Incomplete 'for' statement.
1162                }
1163                else
1164                {
1165                    getsym();
1166                    gen(POP, 0, 0);
1167                    cx1 = cx;  //come back here after loop
1168                    top_expr(set);
1169                    if(sym != SYM_SEMICOLON)
1170                    {
1171                        error(10);   //';' expected
1172                        test(fsys,set1,28);    //Incomplete 'for' statement.
1173                    }
```

```
1174                    else
1175                    {
1176                        getsym();
1177
1178                        cx2 = cx;  //if false, skip loop
1179                        gen(JPC, 0, 0);
1180
1181                        cx3 = cx;  //beginning of codes that need move
1182                        top_expr(set);
```

```
1183                        if(sym == SYM_RPAREN)
1184                        {
1185                            getsym();
1186                        }
1187                        else
1188                        {
1189                            error(22);
1190                        }
1191                        destroyset(set);
1192                        destroyset(set1);
1193                        gen(POP, 0, 0);
1194
1195                        instruction temp[CXMAX];
1196                        for(i = cx3; i<cx; i++)
1197                        {
1198                            temp[i-cx3]=code[i];
1199                        }
1200                        cx = cx3;
1201                        cx3 = i-cx3; //the length of temp
1202                        statement(fsys);
1203                        for(i = 0; i<cx3; i++)
1204                        {
1205                            code[cx++]=temp[i];
1206                        }
1207                        cx1 = cx1-cx;      //offset
1208                        gen(JMP, 0, cx1); //come back to condition
1209                        code[cx2].a = cx-cx2; //destination of false condition
1210                    }
1211                }
1212            }
1213        } // sym == SYM_FOR
```

**（6）random 和 print**

Random 的实现：

```
else if (sym == SYM_RDM) {

    getsym();
    if (sym == SYM_LPAREN)
        getsym();
    else
        error(16);   //'(' expected
    if (sym == SYM_RPAREN) {
        srand((unsigned)time(NULL));
        tmp_num = rand();
        gen(LIT, 0, tmp_num);
        getsym();
        if (sym == SYM_SEMICOLON)
            getsym();
        else
            error(10);//';'expected
    }
    else if (sym == SYM_NUMBER){ ... }
```

把 random 作为一个保留字，当 getsym（）识别出当前读入的符号是 random，则在 statement（）中进入 random 的部分。

再次 getsym（），若 sym 不为左括号则语法错误；在 random 部分以当前时间为随机种子，调用 c 语言内置的 random 函数，存在临时变量中。

再次 getsym（），若为 SYM_NUMBER 则将临时变量除以 num 并放在栈顶；若为右括号则将临时变量直接放在栈顶，并对分号进行错误检查。

print 则是将 print 作为保留字，当 getsym（）识别出当前读入的符号是 random，则在 statement（）中进入 print 的部分。加入新指令 PRT，输出栈顶的值并在其后加上 2 个空格。

对于( Top_expr ,Top_expr , … , Top_expr)会将其中表达式的值依次带间隔输出，若为空括号则换行。

**（12）do-while 和 switch**

Do-while:

```
getsym();
if (sym != SYM_BEGIN) error(39);//'{'expected.
getsym();
set1 = createset(SYM_END, SYM_SEMICOLON, SYM_NULL);
set = uniteset(set1, fsys);
statement(set);
while (inset(sym, statbegsys) || sym == SYM_SEMICOLON){ ... }
destroyset(set1);
destroyset(set);
if (sym == SYM_END)
{
    getsym();
    if (sym != SYM_WHILE)   error(43);// condition expected
    else getsym();
    if (sym != SYM_LPAREN)      error(16);//'(' expected
    getsym();
    set1 = creatset(SYM_RPAREN, SYM_NULL);
    set = uniteset(set1, fsys);
    else top_expr(set);
    gen(JPC, 0, 0);
```

把 do 作为一个保留字，当 getsym（）识别出当前符号是 do，在 statement（）中进入 do（）的部分，调用 statement（）直到遇到 SYM_END，再从 while 中调用 top_expr（）判断条件是否成立，不成立则从第一行语句重新执行。

Switch 未完成：

```
set1 = createset(SYM_RPAREN, SYM_NULL);
set = uniteset(set1, fsys);
if (sym != SYM_LPAREN) error(16);       //'（' expected.
else {
    getsym();
    if (sym != SYM_IDENTIFIER)
        error(38);  // "There must be an identifier to follow the 'switch'."
    else {
        mask* mk;
        if (!(i = position(id))){ ... }
        else if (table[i].kind == ID_VARIABLE){ ... }
        getsym();
        if (sym != SYM_RPAREN) error(22); //MISSING ')'.
        else getsym();
        if (sym != SYM_BEGIN) error(39); //'{' expected
        else getsym();
        while (sym != SYM_END){ ... }
    }
    for (int i = 0; i < 50 && cxend[i] != -1; i++){ ... }
    getsym();
```

本次设计的 case 只能与数字相比，且只能执行单条语句。先判断 switch（）后的标识符是否存在，然后将 case 后的数字与 a 在 table[]中的位置取出来依次放在栈顶，调用 OPR 指令比较两者是否相等，相等则调用 statement（），否则与下一个 case 相比较。

**（13）数组初始化**

使用自递归的函数 initialize()读取由嵌套“{}”包裹的初始化列表，返回值为当前层次实际初始化了的元素（元素指下一级数组，或者最低一级指单个变量/常量）个数。如果维界已定义而初始化列表元素个数少于维界，则以 0 填充。常量数组定义后必须有初始化列表，第一维维界未定义（a[][n]…[m]，n 和 m 为数字或已定义常量）的变量数组也必须有初始化列表，单个变量和维界完全定义的变量数组可初始化也可不初始化。单个变量和单个常量分别在 vardeclaration()和 constdeclaration()中初始化。

对常量数组的初始化为将数字放入 value 数组中响应位置，然后最终随标识符加入符号表。对变量数组初始化为依次生成 LIT，STO，POP 指令，这些初始化指令会被转移到每个过程 INT 指令之后其他指令之前，在程序运行时完成对数组元素的赋值。

```
318     ////////////////////////////////////////////////////////////////////
319 int initialize(int ID_type, int arrayLevel, int n) {  // n = boundary of current dimension
320     int initer = 0, i;
321     printf("arrayLevel = %d\n", arrayLevel);
322     if (td[arrayLevel]) {
323         if (sym == SYM_BEGIN) { //'{'
324             getsym();
325             while (sym != SYM_END) {
326                 if (td[arrayLevel + 1])
327                     i = td[arrayLevel] / td[arrayLevel + 1];
328                 else
329                     i = 1;
330                 initialize(ID_type, arrayLevel + 1, i);
331                 initer++;
332                 if (sym == SYM_COMMA) {
333                     getsym();
334                 }
335             }
336             if (n == -1)
337                 n = initer;
338             if (initer > n) {
339                 error(33);   // too many initializers
340             }
341             else if (initer < n) {
342                 // fill by 0
343                 if (ID_type == ID_CONSTANT) {
344                     vx += (n - initer)*td[arrayLevel];
345                 }

346                 else { // ID_VARIABLE
347                     initer = (n - initer)*td[arrayLevel];
348                     gen(LIT, 0, 0);
349                     while (initer--) {
350                         gen(STO, 0, dx++);
351                     }
352                     gen(POP, 0, 0);
353                 }
354             }
355             if (sym == SYM_END) { //'}'
356                 getsym();
357             }
358             else {
359                 error(30);  //missing '}'
360             }
361         }
362         else {
363             error(38); //'{' expected
364         }
365     }

366     else {
367         if (sym == SYM_NUMBER || sym == SYM_IDENTIFIER) {
368             if (sym == SYM_IDENTIFIER)
369             {
370                 if ((i = position(id)) == 0)
371                 {
372                     error(11); // Undeclared identifier.
373                 }
374                 else {
375                     getsym();
376                     if (table[i].kind == ID_CONSTANT)
377                     {
378                         num = constvalue(i);
379                     }
380                     else
381                     {
382                         error(2); // There must be a number or const to follow '='
383                     }
384                 }
385             }
386             else { //number
387                 getsym();;
388             }
```

```
386             else { //number
387                 getsym();;
388             }
389             if (ID_type == ID_CONSTANT) {
390                 value[vx++] = num;
391             }
392             else { //ID_type == ID_VARIABLE
393                 gen(LIT, 0, num);
394                 gen(STO, 0, dx++);
395                 gen(POP, 0, 0);
396             }
397         }
398         else if (sym == SYM_END) {
399             // {} means initial by 0
400         }
401         else {
402             symset set, set1;
403             set = createset(SYM_COMMA, SYM_NULL);
404             set1 = createset(SYM_NULL);
405             test(set, set1, 29);    //There must be an number or const in declaration
406             destroyset(set);
407             destroyset(set1);
408         }
409     }
410     return initer;
411 }
```

对于在复合语句（begin-end 或“{}”之内的语句）中定义局部变量的情况，只需在 statement 中 begin 条目内前段加上读取变量/常量定义的部分，并且在退出复合语句时将符号表指针 tx 还原（即将方才定义的变量、常量删除）即可。

**（7）其他**

加入的位运算符优先级与 c 语言相同，暂未设计自增（++）自减（--）运算符。

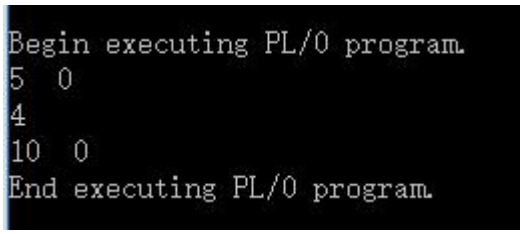
# 四．测试样例与运行结果

**样例 1：**

```
const K = 2,A = 5,k[3][4][K]={{{2,3},{4,K},{10}}};
procedure p()
var a[A][2] = {{2, 5}};
{
   print(a[0][1]， a[4][1]);
};

{
   p();
   print();
   {
   var i = 4;
   const B = k[0][2];
      print(i);
      print();
      print(B,k[1][1]);
   }
}.
```

运行结果截图：




**样例 2：**

```
var i;
const A = 10;

procedure add(a,b)
{
   return a+b;
};

procedure fibo()
var a[A];
{
   a[0]:=a[1]:=1;
   print(a[0],a[1]);
   for(i:=2;i<10;i:=i+1)
      print(a[i]:=add(a[i-1],a[i-2]));
};

procedure factorial(n)
{
   if(n=1 || n=0)
      return 1;
   else if(n<0)
      print(-1);
   else
      return n*factorial(n-1);
   return n;
};

var j;
{
   fibo();
   print();

   print(factorial(6),factorial(-5));
   print();

   if((j:=0) && (j:=4) && (j:=6) || (j:=j+1) || (j:=2))
   {
      print(j);
      print();
```

```
        if((j:=8) && (j:=4) && (j:=6) || (j:=j+1) || (j:=2))
            print(j);
        print();

        exit;
    }
    print(9);
}.
```

运行结果：

```
Begin executing PL/0 program.
1  1  2  3  5  8  13  21  34  55
720  -1  -5
1
6
Exit Program
End executing PL/0 program.
```