PL0 编译器扩展设计实验报告

小组成员及其分工：

许同 PB15111646: 条件的短路计算；if-elif-else 语句，exit 语句；文法设计。

郭秋洋 PB15111650: 添加注释；添加各种 c 语言表达式；return 语句；goto 语句；print(),random(),callstack()内置函数。

姜也东 PB15111628: 数组实现及数组传参；函数实现及函数传参。

许道雷 PB15111674: for 语句；while 语句的修改；do while 语句；break、continue 语句。

仁青尼玛 PB15111644: 代码汇总；文法设计；语法图绘制；编写实验报告。

1）实验目的：在分析理解一个教学型编译程序（如 PL/0）的基础上，对其词法分析程序、 语法分析程序和语义处理程序进行部分修改扩充。达到进一步了解程序编译过程的基本原理和基本实现方法的目的。

2）实验要求：

- 基础要求

（1） 添加注释：包括行注释与块注释
（2） 扩展 PL0 中的“条件”：增加逻辑运算符&&、||和!；把 PL/0 语言中的“条件”概念一般化为 C 语言那样(表达式值非零即为“真”)；实现“条件”的短路计算。
（3） 添加数组：实现数组变量声明/对数组元素赋值/在表达式中引用数组元素等。可以有多维数组，数组的维度范围设为常量。
（4） 参数传递：实现传值调用，如传递常量值，或普通变量/数组元素的值。并进行简单的语义检查（如实参和形参个数/类型的对应等）
（5） 添加语句实现：else/elif 子句，exit 语句，return 语句及返回值的实现；实现 c 语言风格的 for 语句。

- 提高扩展

（6） 给 PL/0 添加内置函数 random 和 print.
（7） 给 PL/0 添加内置函数 CALLSTACK。该函数可以按照调用的先后次序，输出在运行时栈中存放的正在执行的各个过程/函数的活动记录相关信息（如程序计数器，参数值等）
（8） 更多的 C 风格的运算表达式实现。语法/语义参照 C 语言。
（9） 实现传地址调用。
（10）过程作为参数传递的实现。
（11）goto 语句/break 语句（跳出包含它的最内层循环）/continue 语句（继续执行包含它的最内层循环）的实现。语法/语义参照 C 语言。
（12）do while 语句/switch 语句的实现。语法/语义参照 C 语言。
（13）加强的 PL/0 变量的定义/初始化及其实现。

3）已实现的功能扩展：

基础修改：

（1） 添加注释：块注释由/*和*/包含，不允许嵌套；行注释由//开始直到行结束符。
（2） 添加 c 运算符：
逻辑运算：&&，||,!

位运算：&，|，^
取余运算：%
条件运算：==,!=
赋值运算：+=，-=，*=，/=，%=，&=，|=，!=,^=，<<=,>>=
自加自减：++，--
运算符优先级与 c 语言一致。

（3） 添加了 return 语句，if 语句的 then，添加了 elif 和 else 语句，取消了 while 语句的 do，添加了 c 语言的 for 语句。

（4） 添加数组。实现了多维数组的声明，赋值，运算。

（5） 参数传递：将 procedure 修改为 function，添加 function 的声明，删除 call 指令，添加 function 的调用。

（6） 条件的短路计算。

提高扩展：

（7） 添加内置函数 print()，random()，callstack()。

（8） 添加了 goto 语句，do-while 语句，break 语句，continue 语句。

（9） 添加了更多 c 语言表达式：
移位运算：<<,>>
连续赋值：i := j := k := 100
？:运算：?

(10) 实现数组传参。

4）实验环境与工具：

（1）计算机及操作系统：LENOVO G50，Windows10；
（2）实验工具:dev C++ 5.11，code :: block，TDM-GCC 4.9.2 64-bit；
（3）教学型 PL0 编译程序。

5）结构设计说明

（1）PL0 编译程序的结构图

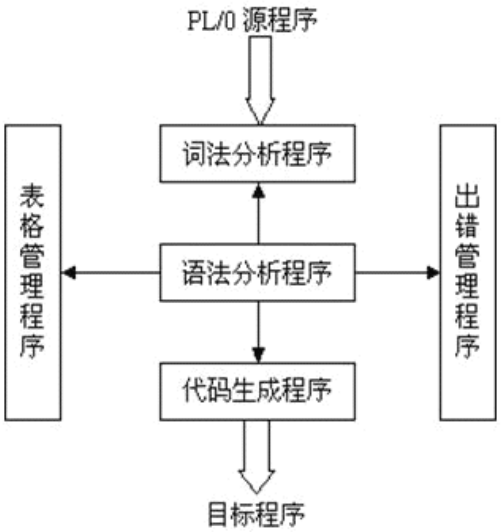


图 1　编译程序的结构图

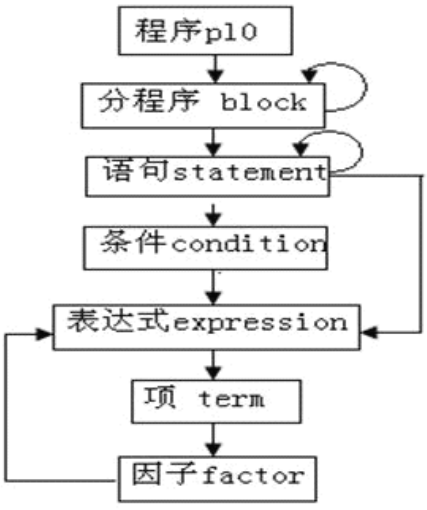


图 2 功能模块调用关系图

（2）扩展后 PL0 编译程序函数功能表

过程或函数名	简要功能说明
Main	主程序
error	出错处理，打印出错位置和错误编码
getsym	词法分析，读取一个单词
getch	漏掉空格，读取一个字符
gen	生成目标代码，并送入目标程序区
test	测试当前单词符号是否合法
block	分程序分析处理过程
enter	登录名字表
position	查找标识符在名字表中的位置
fposition	查找函数标识符在名字表中的位置
dposition	查找数组标识符在名字表中的位置
constdeclaration	常量声明处理
vardeclaration	变量声明处理
listode	列出目标代码清单
statement	语句处理
*_expr	各种表达式处理
term	项处理
factor	因子处理
interpret	对目标代码的解释执行程序
base	通过静态链求出数据区的基地址

(3)扩展后 PL0 上下文无关文法

```
body → const body1 ; body2                  //body 代表程序体，
                                            //body1 代表 ident ：= number
      | var ident  body3
      | function ident （ body7 ） body
      | stmt                                //stmt 代表语句


body1→ ident := number
body2→ body | body1;body2                   //body2 代表 const 中;后的两个分支
body3→ body6
      | body4 body6
body4 → [ number ] body5
body5 →  ε  | body4
Body6 → ,ident  body3
        | ;body


body7 → ident body8
body8 → ,body7 |
```

```
stmt→ begin  stmt1  end      // stmt 代表语句  stmt1 代表语句序列
 | { stmt1 }
 | if stmt2 stmt4
 | while （ rel_expr ） stmt
 | return  stmt5  ;  stmt
 | for ( assign_expr ;rel_expr ; assign_expr ) stmt
 | assign_expr
 | do stmt while ( rel_expr )
 | break
 | continue
 | exit （ ）
 | goto label

stmt2 →  （ assign_expr ）stmt stmt3
stmt3 →  elif stmt2 | ε
stmt4 →  else  stmt stmt4  | ε
stmt5 →  assign_expr | ε


assin_expr →question_expr := assign_expr
              | question_expr += assign_expr
              | question_expr -= assign_expr
              | question_expr *= assign_expr
              | question_expr /= assign_expr
              | question_expr %= assign_expr
              | question_expr &= assign_expr
              | question_expr |= assign_expr
              | question_expr ^= assign_expr


question_expr → or_expr ? assign_expr : assign_expr
or_expr → and_expr ||  and_expr
and_expr → or_bit_expr &&  or_bit_expr
or_bit_expr → xor_bit_expr |  xor_bit_expr
xor_bit_expr → and_bit_expr ^  and_bit_expr
and_bit_expr → rel_expr &  rel_expr
rel_expr → shift_expr  relop  shift_expr
relop  →  ==
        |  <>
        |  !=
        |  <
        |  >
```

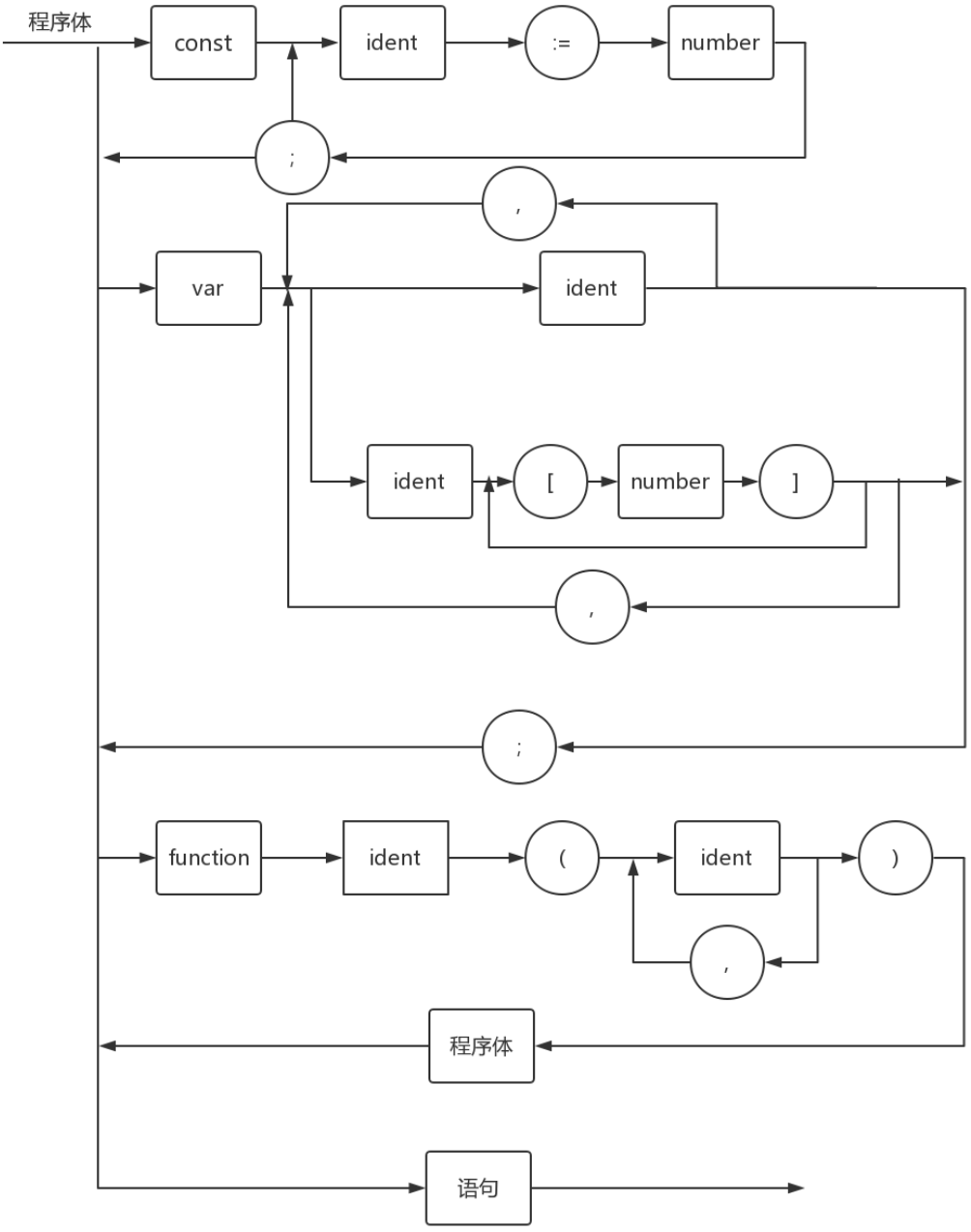
```
        |  <=
        |  >=
shift_expr → expression  relop1  expression
expression → term expre1
term   →  postfix_expr  op1   postfix_expr
postfix_expr  →  factor op2

relop1  →  >> | <<
expre1  →  + term expre2
        |   -term expre2
expre2  →  expre1 |  ε
op1  →  *
       | /
       | %

op2  →  ++
      |  --


factor → ident
         | number
         | -factor
         | !factor
         | (assign_expr)
         | ++factor
         | --factor
         | print(var1)
         | random（var2）
         | callstack（）


var1  → id
       | num
       | ε
       | var1,var1


Var2  → num
       | ε
```

(4)扩展后 PL0 语言的语法图

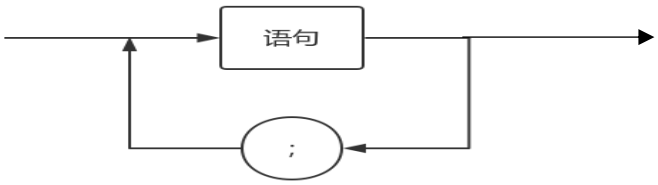
程序：


程序体
.

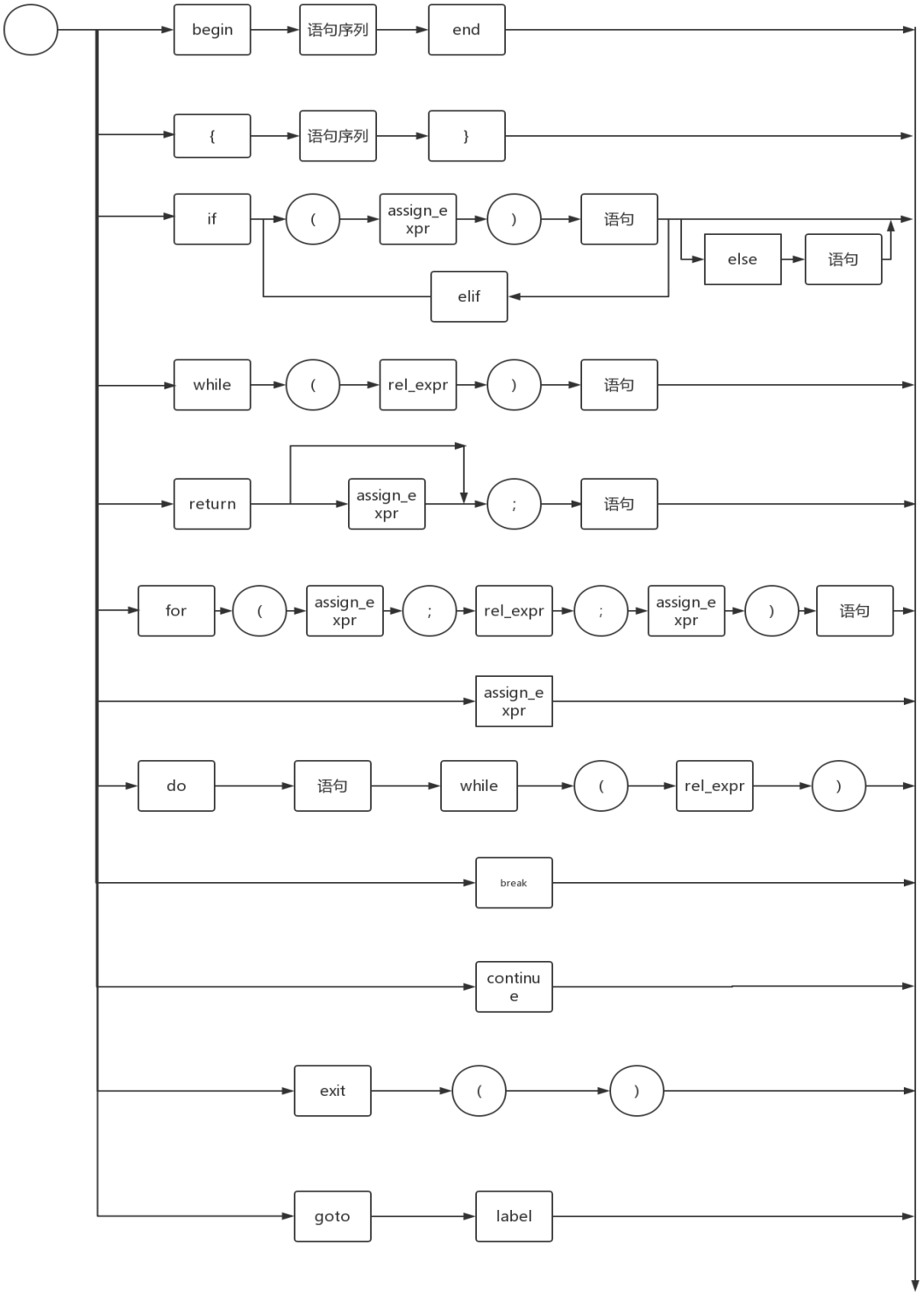
程序体：


程序体
const
ident
:=
number
;
,
var
ident
ident
[
number
]
,
;
function
ident
(
ident
)
,
程序体
语句

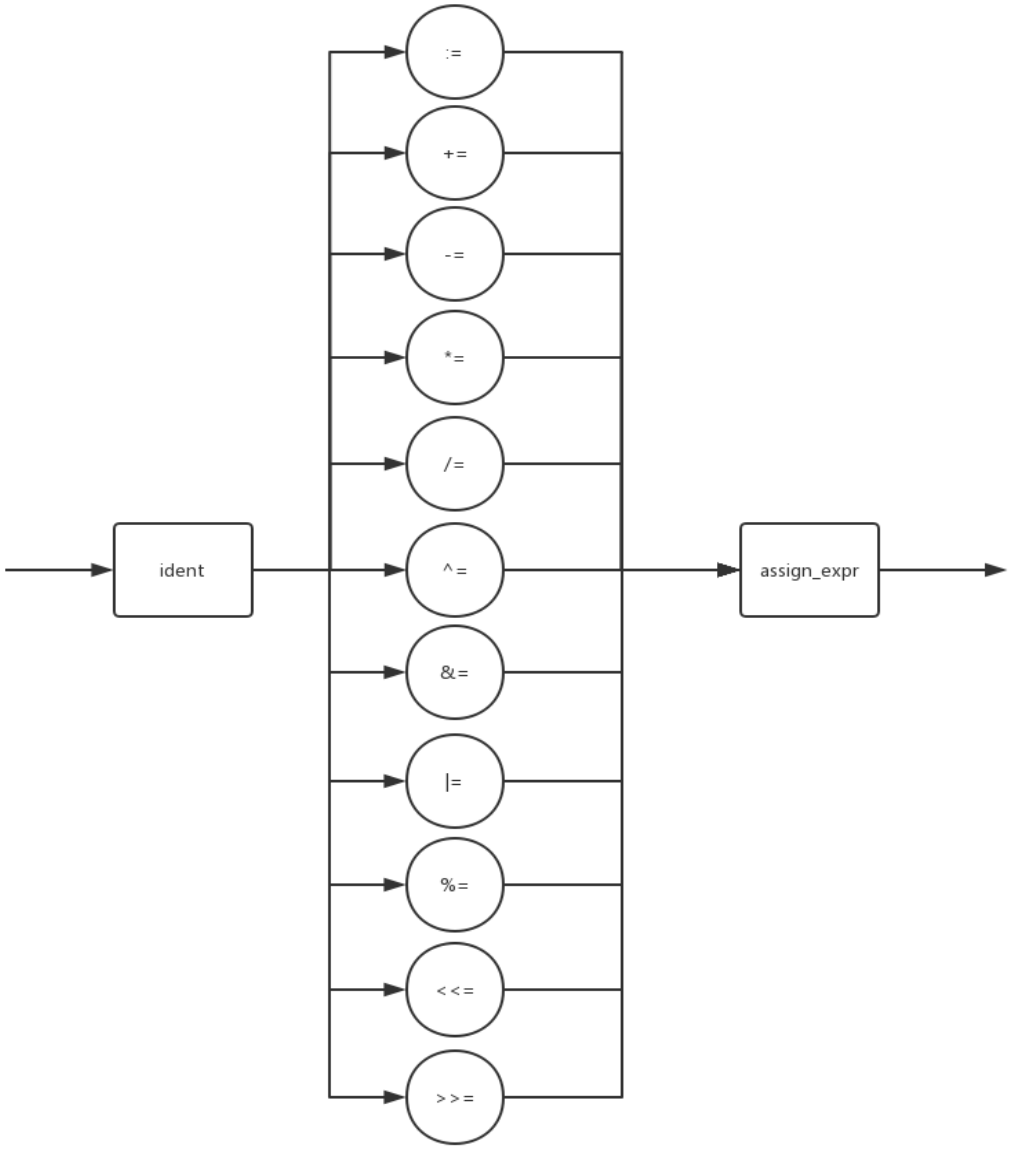
语句序列：


语句
;

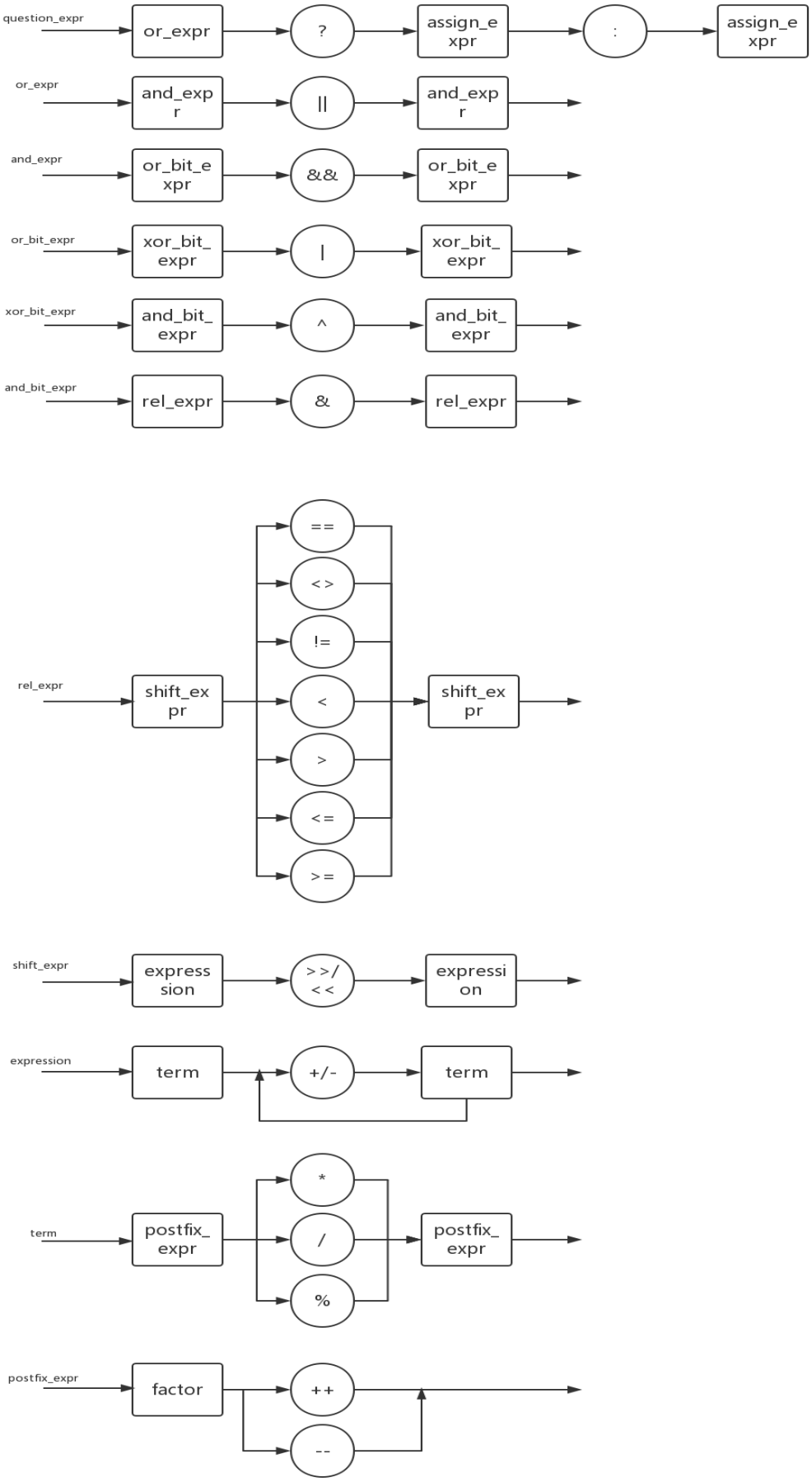
语句：


begin
语句序列
end
{
语句序列
}
if
(
assign_expr
)
语句
else
语句
elif
while
(
rel_expr
)
语句
return
assign_expr
;
语句
for
(
assign_expr
;
rel_expr
;
assign_expr
)
语句
assign_expr
do
语句
while
(
rel_expr
)
break
continue
exit
(
)
goto
label

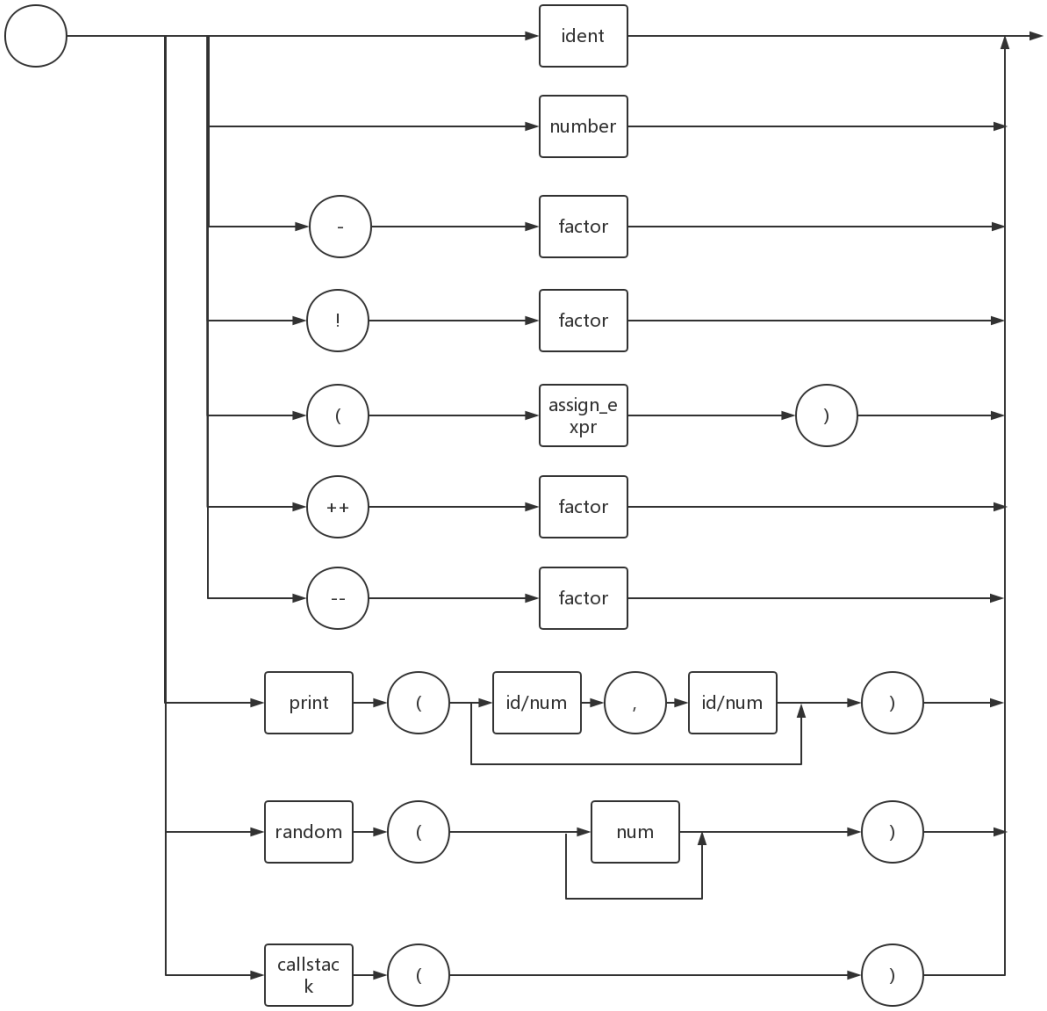
赋值表达式(assign_expr):



其他表达式:


question_expr
or_expr
?
assign_expr
:
assign_expr
or_expr
and_expr
||
and_expr
and_expr
or_bit_expr
&&
or_bit_expr
or_bit_expr
xor_bit_expr
|
xor_bit_expr
xor_bit_expr
and_bit_expr
^
and_bit_expr
and_bit_expr
rel_expr
&
rel_expr
rel_expr
shift_expr
==
<>
!=
<
>
<=
>=
shift_expr
shift_expr
expression
>>/<<
expression
expression
term
+/-
term
term
postfix_expr
*
/
%
postfix_expr
postfix_expr
factor
++
--

因子(factor):



6）设计实现

（1）添加注释：包括行注释//和块注释/*…*/，块注释不允许嵌套。

```
else if(ch == '/')
    {
        getch();
        if(ch == '/')   // // ...
        {
            while(cc != ll)
                getch();
            getsym();
        }
        else if(ch == '*') // /*...*/
        {
            getch();
            getch();
            while (1) {
                if (ch == '*') {
                    getch();
                    if (ch == '/')
                    break;
                }
                getch();
            }
            getch();
            getsym();
            }
        else if (ch == '=')
            {
                sym = SYM_SLASH_BEC;    // /=
                getch();
            }
            else
                sym = SYM_SLASH;     // /
        }//2017.9.21
```

（2）添加 c 运算符，优先级与 c 语言相同。

注意算符优先级和各算符对操作数的要求（例如左值检测）即可。

此处列举几个重要的实现：

1. 后++的实现：++指令要求操作数为左值，所以第 720 行进行左值检测。第 723 行保存运算前操作数的值，730 行向栈顶赋 1，731 行执行++。CPY 指令将 stack[top]复制到 stack[++top]，此处是为了保存数组累加操作前的值；LAD 指令从栈顶得到偏移地址，并将取出的数据覆盖当前栈顶。至于 STA，LOA 指令，则类同于变量的 STO，LOD 指令，在数组实现里再作详细说明。

```
714     while (sym == SYM_INC || sym == SYM_DEC )
715     {
716         int c = cx -1 ;
717         postfixop = sym;
718         if(postfixop == SYM_INC)    //i++
719         {
720             if (code[c].f == LOD||code[c].f == LOA)//l-value check
721             {
722                 if(code[c].f == LOD)
723                  gen(LOD, code[c].l, code[c].a);//save the previous value of i
724                 else
725                 {
726                     code[c].f = LAD;
727                     gen(CPY,0,0);
728                     //gen(LOD, level - mk->level, mk->address);//save the previous value of i
729                 }
730                 gen(LIT, 0, 1);
731                 gen(OPR, 0, OPR_ADD);
732                 if(code[c].f == LOD)
733                   gen(STO, code[c].l, code[c].a);
734                 else
735                   gen(STA, code[c].l, 1);
736                 gen(POP, 0, 0);
737             }
738             else
739                 error(26);
740         }
```

2. 前++的实现：基本同后++，但不用保存运算前操作数的值。

```
653         else if (sym == SYM_INC)//++i
654         {
655             getsym();
656             factor(fsys);   //stack[top] = i
657         //  mask* mk;
658         //  i = position(id);
659         //  mk = (mask*) &table[i];
660             int c = cx - 1;
661             if (code[cx - 1].f == LOD || code[cx - 1].f == LOA)//l-value check   //i must be a l-value
662             {
663                 if(code[cx - 1].f == LOA)
664                     code[cx-1].f == LAD;
665                 gen(LIT, 0, 1);
666                 gen(OPR, 0, OPR_ADD);
667                 if(code[c].f == LOD)
668                     gen(STO, code[c].l, code[c].a);
669                 else
670                     gen(STA, code[c].l, 1);
671             }
672             else
673                 error(26);
674         }
```

3. 赋值表达式的实现：同样先进行左值检测，assignop 保存 sym 然后递归调用 assign_expr,这样就实现了连续赋值的操作，然后计算，结果放在栈顶，然后赋给左操作数。注意，根据老师的建议，我们将 STO 指令的弹栈操作取消，改为在 assign_expr 返回到 statement 后再统一弹栈。

```
990     if (inset(sym, set1))
991     {
992         if (code[cx - 1].f == LOD||code[cx - 1].f == LOA) //l-value check
993         {
994             c = cx - 1;
995             if(code[c].f == LOA)
996                 {
997                     code[c].f = LAD;
998                     if(sym == SYM_BECOMES)
999                     gen(POP,0,0);
1000                }
1001            assignop = sym;
1002            getsym();
1003            assign_expr(set);
1004            switch (assignop)
1005            {
1006            case SYM_BECOMES:
1007                if(code[c].f == LOD)
1008                gen(STO, code[c].l, code[c].a);
1009                if(code[c].f == LAD)
1010                gen(STA, code[c].l, 0);
1011                //gen(POP, 0, 0);
1012                break;
1013            case SYM_BIT_AND_BEC:
1014            {
1015                gen(OPR, 0, OPR_BIT_AND);
1016                if(code[c].f == LAD)
1017                gen(STA, code[c].l, 0);
1018                else
1019                gen(STO, code[c].l, code[c].a);
1020                //gen(POP, 0, 0);
1021                break;
1022            }
```

4. ? 表达式的实现：类似 if-else 语句采用回填技术，cx1 保存？后第一个指令的地址，cx2 保存：后第一指令的地址。然后分别回填到 gen(JPC, cx1, 0)和 gen(JMP, cx2, cx + 1)。

```
1222  void question_expr(symset fsys)
1223  {
1224      int cx1, cx2;
1225      void assign_expr(symset fsys);
1226      symset set;
1227      set = uniteset(fsys, createset(SYM_QUESTION, SYM_COLON, SYM_NULL));
1228      or_expr(set);
1229      if (sym == SYM_QUESTION)
1230      {
1231          cx1 = cx;
1232          gen(JPC, cx1, 0);
1233          getsym();
1234          assign_expr(set);
1235          cx2 = cx;
1236          gen(JMP, cx2, cx + 1);
1237          code[cx1].a = cx;
1238          if (sym == SYM_COLON)
1239          {
1240              getsym();
1241              assign_expr(set);
1242              code[cx2].a = cx;
1243          }
1244          else
1245              error(29);
1246      }
1247      destroyset(set);
1248  } //question_expr
```

（3）添加/修改语句：

1. 修改 if 语句：

删除了 then，添加了 else，elif。运用回填技术，用 cx1 保存条件为假时需跳转的地址，cx2 保存条件为真时需跳转的地址。1165 行即为条件为假的跳转，地址为 else 或 elif 中的语句，1168 行是条件为真且执行了 else 前语句后的跳转，地址为 else 后的下一地址。由于 elif 与 if 执行操作相同，所以 1178 行直接执行 if 的操作。

```
1155        else if (sym == SYM_IF)
1156        {
1157            set1 = createset(SYM_ELSE, SYM_NULL);
1158            set = uniteset(set1, fsys);
1159            getsym();
1160            if (sym == SYM_LPAREN)
1161                assign_expr(fsys);
1162            else
1163                error(27);
1164            cx1 = cx;
1165            gen(JPC, cx1, 0);
1166            statement(set);
1167            cx2 = cx;
1168            gen(JMP, cx2, cx + 1);
1169            code[cx1].a = cx;
1170            if (sym == SYM_ELSE)
1171            {
1172                getsym();
1173                statement(set);
1174                code[cx2].a = cx;
1175            }
1176            else if (sym == SYM_ELIF)
1177            {
1178                sym =SYM_IF;
1179                statement(set);
1180                code[cx2].a = cx;
1181            }
1182            destroyset(set1);
1183            destroyset(set);
1184        }
```

2. 修改 while 语句：

删除 do。仅仅是删除了 do，改用左右括号区分循环条件和语句，此处不做赘述。

```
1207        else if (sym == SYM_WHILE)
1208        {
1209            // while statement
1210            cx1 = cx;
1211            getsym();
1212            if (sym == SYM_LPAREN)
1213                assign_expr(fsys);
1214            else
1215                error(27);
1216            cx2 = cx;
1217            gen(JPC, 0, 0);
1218            statement(fsys);
1219            gen(JMP, 0, cx1);
1220            code[cx2].a = cx;
1221        }
```

3. 添加 return 语句;

return 后无表达式时，1144 行返回 0；否则返回 stack[b-1]=stack[top]。

b 为被调用函数活动记录栈的基指针，则 b-1 则为设定的返回值所在位置。函数活动记录栈的详细情况在函数部分再作详细说明。

```
1131    else if (sym == SYM_RETURN)
1132    {
1133        getsym();
1134        if (inset(sym, facbegsys))
1135        {
1136            assign_expr(fsys);
1137            if (sym == SYM_SEMICOLON)
1138                getsym();
1139            else
1140                error(10);
1141        }
1142        else if (sym == SYM_SEMICOLON)
1143        {
1144            gen(LIT, 0, 0);
1145            getsym();
1146        }
1147        else
1148            error(10);
1149        gen(OPR, 0, OPR_RET);
1150        cx1 = cx;
1151        gen(JMP, cx1, 0);
1152        statement(fsys);
1153        code[cx1].a = cx;
1154    }
```

```
1474            case OPR_RET:
1475                stack[b - 1] = stack[top];//return
1476                top--;
1477                break;  /*added 2017.11.3*/
```

4. 添加 c 风格的 for 语句：

只需在 statement 函数中加入 sym = SYM_FOR 的情况即可。首先考虑 C 语言风格 for 语句结构 for(赋值;条件;赋值)，读取 for 以后读到左括号(后调用 assign_expr 函数读取赋值表达式，记录此时代码位置 cx3，然后调用 rel_expr 读取条件表达式，并生成跳转代码，记录代码位置 cx4，最后再调用 assign_expr 读取赋值表达式，然后读取循环内容，调用 statement 函数。然后返回 ex3 继续判断条件，直到条件不成立，将此处 cx 值填入 JPC 中 0 的位置，循环结束。

```
getsym();
if(sym == SYM_LPAREN)
{
    getsym();
    if(inset(sym, facbegsys))
    {
        assign_expr(set);
        cx3=cx;
        getsym();
        if(inset(sym, facbegsys))
        {
            rel_expr(set);
            gen(JPC, 0, 0);
            cx4=cx;
            getsym();
            if(inset(sym, facbegsys))
                {
                    assign_expr(set);
                }
```

```
    getsym();
    statement(set);
    gen(JMP, 0, cx3);
    code[cx4].a = cx;
```

由于 for 语句第三个赋值表达式不是以分号;结尾，将右括号)加入结束符集合 fsys。

```
set1=createset(SYM_RPAREN);
set=uniteset(set1,fsys);
```

5. 添加 do-while 语句：

statement 函数读到保留字 do 以后，记录此时的代码位置 cx1，调用 statement 读取执行内容，结束后取下一个因子 while 后调用 rel_expr 读取条件语句，然后记录此时代码位置 cx2，产生 JPC 0，然后产生 JMP cx1,，读取完后将此时代码位置 cx 回填给 cx2 位置的 JPC 0，使其修改为 JPC cx(回填时的代码位置)。

```
1411        else if(sym == SYM_DO)
1412        {
1413            lflag=1;
1414            cx1=cx;
1415            getsym();
1416            statement(fsys);
1417            if(sym==SYM_WHILE)
1418            {
1419                getsym();
1420                if(sym==SYM_LPAREN)
1421                {
1422                    getsym();
1423                    if(inset(sym, facbegsys))
1424                    {
1425                        if(cflag==1)code[cx4].a = cx;
1426                        rel_expr(fsys);
1427                        cx2=cx;
1428                        gen(JPC, 0, 0);
1429                        gen(JMP, 0, cx1);
1430                        if(bflag==1)code[cx3].a = cx;
1431                        bflag=0;
1432                        cflag=0;
1433                        code[cx2].a = cx;
1434                        getsym();
1435                    }
1436                }
1437            }
1438        }
```

6. 添加 break/continue 语句;

break 和 continue 相似，首先定义全局变量 bflag 和 cflag 来标记代码中出现过 break 或者 continue，同时定义全局变量 cx3 和 cx4 来记录 break 和 continue 生成跳转指令的位置。在读取到 break 时，记录代码位置 cx3，生成 JMP 0；在循环语句编译的末尾将此时的代码编号 cx 回填给 cx3 位置的 JMP 0，变成 JMP cx(循环终止位置)，这样就完成了跳出循环的功能。Continue 类似，读取到 continue 时，记录代码位置 cx4，生成 JMP0，将循环开始的条件判断之前的代码位置 cx2 回填到 cx4 对应的指令 JMP 0，产生 JMP cx2（循环条件判断位置)，Continue 功能完成。另外定义全局变量 lflag 用于判断 break/continue 是否在循环中。

```
1448        else if(sym==SYM_BREAK)
1449        {
1450            if(lflag==0)
1451            {
1452                error(28);
1453                getsym();
1454                statement(fsys);
1455            }
1456            else{
1457            cx3=cx;
1458            bflag=1;
1459            gen(JMP,0,0);
1460            getsym();
1461            lflag=0;
1462            }
1463        }
1464        else if(sym==SYM_CONTINUE)
1465        {
1466             if(lflag==0)
1467            {
1468                error(28);
1469                getsym();
1470                statement(fsys);
1471            }
1472            else{
1473            cflag=1;
1474            cx4=cx;
1475            gen(JMP,0,0);
1476            getsym();
1477            lflag=0;}
1478        }
```

7. exit 语句实现：直接生成 leave 指令即可。

```
1479        else if (sym == SYM_EXIT)
1480        {
1481            getsym();
1482            if(sym == SYM_LPAREN)
1483            {
1484                getsym();
1485                if(sym == SYM_RPAREN)
1486                    gen(OPR, 0 , OPR_LEAVE);
1487                    else
1488                        error(22);
1489                getsym();
1490                }
1491                else
1492                    error(27);
1493        }
```

8. goto 语句的实现：

为了与 ID_VARIABLE 区分，添加新的类型 ID_LABEL，该类型在名字表中具有 name,kind 和 address 属性。如果读到 goto 语句，查名字表，没有则登录名字

表，用全局数组 cx_0[i]保存当前指令地址，然后生成 jmp 指令，其跳转地址未定，否则直接跳转到名字表中 LABEL 的地址之后一个地址；如果读到 ID_LABEL，查名字表，没有则登录名字表，否则将当前 cx 赋给 code[cx_0[i]]（回填）。

```
1494        else if(sym == SYM_GOTO)
1495        {
1496            getsym();
1497            if(sym == SYM_IDENTIFIER)
1498            {
1499                if((i = position(id))== 0)
1500                    enter(ID_LABEL);
1501                i = position(id);
1502                mask* mk;
1503                mk = (mask*) &table[i];
1504                cx_0[i] = cx;
1505                gen(JMP, level - mk->level ,mk -> address+1);
1506                getsym();
1507            }
1508            else
1509                error(14);
1510        }
```

```
587         if (sym == SYM_IDENTIFIER)
588         {
589             if ((i = position(id)) == 0)
590             {
591                 getsym();
592                 if(sym == SYM_COLON)
593                 {
594                     enter(ID_LABEL);
595                     getsym();
596                     statement(fsys);
597                 }
598                 else
599                     error(11); // Undeclared identifier.
600             }
601             else
602             {
603                 getsym();
604                 if(sym == SYM_COLON && cx_0[i] != 0)
605                 {
606                     code[cx_0[i]].a = cx;
607                     getsym();
608                     statement(fsys);
609                 }
```

（4）添加 print(),random(),callstack()内置函数

1.print()函数：为了尽量减少新指令的数量，考虑借用原有 STO 指令里的 printf 实现 print()的功能。在表达式后直接在调用 STO，这样不影响栈顶和表达式的值，并且输出了表达式的值。为了实现 print()输出换行，观察到 stack 为 int 型，所以通过向栈顶存入 0xffff 来标志换行。

```
1836            case STO:
1837                if(stack[top] == 0xffff)
1838                    printf("\n");
1839                    else
1840                    {
1841                    stack[base(stack, b, i.l) + i.a] = stack[top];
1842                    printf("%d\n", stack[top]);
1843                }
1844                //top--;deleted    2017.11.2
1845                break;
```

```
722         else if(sym == SYM_RANDOM)
723         {
724             getsym();
725             if (sym == SYM_LPAREN)
726             {
727                 getsym();
728                 set = uniteset(set, createset(SYM_RPAREN, SYM_NULL));
729                 if (sym == SYM_RPAREN)
730                 {
731                     gen(LIT , 0 , 0);
732                     getsym();
733                 }
734                 else
735                 {
736                     assign_expr(set);
737                     if (sym == SYM_RPAREN)
738                         getsym();
739                     else
740                         error(22); // Missing ')'.
741                 }
742             }
743             int c = cx - 1;
744             gen(RAN, 0, 0);
745             gen(STO, code[c].l, code[c].a);
746         }
```

2. random()函数：

添加 RAN 指令，通过栈顶值调用 rand()函数实现生成随机数。

```
1879            case RAN:
1880                if(stack[top])
1881                stack[++top] = rand()%stack[top-1];
1882                else
1883                stack[++top] = rand();
1884                break;
```

```
722         else if(sym == SYM_RANDOM)
723         {
724             getsym();
725             if (sym == SYM_LPAREN)
726             {
727                 getsym();
728                 set = uniteset(set, createset(SYM_RPAREN, SYM_NULL));
729                 if (sym == SYM_RPAREN)
730                 {
731                     gen(LIT , 0 , 0);
732                     getsym();
733                 }
734                 else
735                 {
736                     assign_expr(set);
737                     if (sym == SYM_RPAREN)
738                         getsym();
739                     else
740                         error(22); // Missing ')'.
741                 }
742             }
743             int c = cx - 1;
744             gen(RAN, 0, 0);
745             gen(STO, code[c].l, code[c].a);
746         }
```

3.callstack()函数

添加 CAS 指令，用 t 指针定位来打印当前 stack，并通过 b0 指针对返回值，静态链，动态链，返回地址做出标记。

```
1885        case CAS:
1886            int t = top;
1887            int b0 = b;
1888            printf("%5d TOP:  %d\n",t,stack[t]);
1889            --t;
1890            while(t != 0)
1891            {
1892                while(t != b0+2)
1893                {
1894                    printf("%5d       %d\n",t,stack[t]);
1895                    --t;
1896                }
1897                printf("%5d  PC:  %d\n",t,stack[t]);
1898                --t;
1899                printf("%5d  DL:  %d\n",t,stack[t]);
1900                b0 = stack[t];
1901                --t;
1902                printf("%5d  SL:  %d\n",t,stack[t]);
1903                --t;
1904                printf("%5d  RE:  %d\n",t,stack[t]);
1905                --t;
1906            }
```

```
773             else if(sym == SYM_CALLSTACK)
774             {
775                 getsym();
776                 if (sym == SYM_LPAREN)
777                     getsym();
778                     else
779                         error(27);
780                 if(sym == SYM_RPAREN)
781                     getsym();
782                     else
783                         error(22);
784                 gen(CAS, 0, 0);
785             }
```

（5）条件的短路计算：

在解释执行程序中，以语句 X:=Y op Z 为例，其中 Y，Z 为布尔表达式，其值只有两个，TRUE 和 FALSE，其中真为 1，假为 0. op 为逻辑与&&和逻辑非||布尔运算符。短路计算是指，当 op 为&&时，只有 Y 为 1 时，才会执行此运算符，否则直接另表达式结果为 0，即 Y 本身的值就代表了表达式的值；当 op 为||时，只有 Y 为 0 时，才会执行此运算符，否则直接另表达式结果为 1，即此时 Y 本身就代表了表达式的值；

```
case OPR_AND:
    top--;
    if(stack[top]==1)
    {
    stack[top] = stack[top] && stack[top + 1];
    break;
    }
case OPR_OR:

    top--;
    if(stack[top]==0)
    {
    stack[top] = stack[top] || stack[top + 1];
    break;
   }
```

（6）实现数组：

1．数组声明的语法结构：

ident ->id | dim

dim -> idB[N]

B -> [N]B | ε

N -> num

2．数组的声明：

用一个新的名词表 d 存放数组的名称，维度与每维的长度。在声明变量时向前一个符号，如果是‘[’，则作为数组存进名词表 table 与 d，并继续向前一个符号，如果是数字或常数，则维度加一，并记录长度，继续向前一个，期望匹配到‘]’，匹配后继续向前。若符号为‘[’，重复上述操作。为数组分配对应大小（所有维度长度之积）的空间。以下为数组的声明：

```
if(sym == SYM_LSQURBRA)
{
    strcpy(d[++dimx].name,table[tx].name);
    int c = 0;
    offset = 1;
    do{
    getsym();
    if(sym == SYM_NUMBER)
    {
        c++;
        d[dimx].l[c] = num;
        offset *= num;
        getsym();
    }
    else if(sym == SYM_IDENTIFIER)
    {
        c++;
        int k = position(id);
        d[dimx].l[c] = table[k].value;
        offset *= table[k].value;
        getsym();
    }
    else error(28);
    if(sym == SYM_RSQURBRA)
        getsym();
    else  error(28);}
```

新建立的名字表：

```
{
    char name[MAXIDLEN + 1];
    int  kind;
    int  value;
} comtab;

comtab table[TXMAX];

typedef struct
{
    char  name[MAXIDLEN + 1];
    int   kind;
    short level;
    short address;
} mask;

FILE* infile;

typedef struct
{
 char name[MAXIDLEN+1];
 int c ;
 char var[40][MAXIDLEN+1];
}funct;

typedef struct
{
    char name[MAXIDLEN+1];
    int c ;
    int l[100];
}dim;

dim d[40];

int dimx = 0;

funct f[40];

int fx = 0;
```

3. 数组的寻址：在 factor 函数中，如果遇到数组变量，先将 0 放在栈顶，使全局变量 offset 等于所有维度长度之积，从“[assign_expr]”中的表达式得到最高维的位置，放在栈顶，offset 除以这个维度定义的长度，放在栈顶加入乘法指令，再加入加法指令。 对下一个维度执行同样操作，直到最后一维。这样数组元素的地址就会在操作结束后被放在栈顶。

查找数组名字表的函数：

```
int dposition(char *id)
{
    int i;
    strcpy(d[0].name , id);
    i = dimx + 1;
    while (strcmp(d[--i].name, id) != 0);
    return i;
}
```

factor 中数组寻址的具体操作：

```
548                 case ID_DIM:
549                     dp = dposition(id);
550                     coun = 0;
551                     offset = 1;
552                     for(coun=1;coun<=d[dp].c;coun++)
553                         offset *= d[dp].l[coun];
554                     mk = (mask*)&table[i];
555                     gen(LIT , 0 , mk->address);
556                     getsym();
557                     coun = 1;
558                     while(sym == SYM_LSQURBRA&&coun <= d[dp].c)
559                     {
560                           getsym();
561                           offset /= d[dp].l[coun++];
562                           gen(LIT,0,offset);
563                           assign_expr(fsys);
564                           gen(OPR,0,OPR_MUL);
565                           gen(OPR,0,OPR_ADD);
566                           if(sym == SYM_RSQURBRA)
567                             getsym();
568                           else error(28);
569                     }
570                     gen(LOA,mk->level,0);
571                     break;
```

4. 数组的存取：添加了三个指令用于数组的存取

STA：从栈顶减 1 减 i.a 处得到偏移地址，其余同 STO

LOA：从栈顶得到偏移地址，并将取出的数据覆盖当前栈顶，其余同 LOD

LAD：从栈顶得到偏移地址，其余同 LOD。

```
1575        case LOD:
1576            stack[++top] = stack[base(stack, b, i.l) + i.a];
1577            break;
1578        case LOA:
1579            stack[top] = stack[base(stack, b, i.l) + stack[top]];
1580           // top++;
1581            break;
1582        case LAD:
1583            stack[top+1] = stack[base(stack, b, i.l) + stack[top]];
1584            top++;
1585            break;
1586        case STO:
1587            stack[base(stack, b, i.l) + i.a] = stack[top];
1588            printf("%d\n", stack[top]);
1589            //top--;deleted    2017.11.2
1590            break;
1591        case STA:
1592            stack[base(stack, b, i.l) + stack[top-i.a-1]] = stack[top];
1593            printf("%d\n", stack[top]);
1594            //top--;deleted    2017.11.2
1595            break;
1596        case CPY:
1597            stack[top+1] = stack[top];
1598            top++;
1599            break;
```

5. 数组传参：

在函数声明局部变量时考虑数组，实现与上面相似。在 factor 函数中处理时，使用 LOD 指令将数组的首地址入栈，其他操作与普通数组相同。处理函数时，如果需要传递的参数是数组，则将传递进去的数组地址装入栈中。

```
case ID_PDIM:
    dp = dposition(id);
    coun = 0;
    offset = 1;
    for(coun=1;coun<=d[dp].c;coun++)
        offset *= d[dp].l[coun];
    mk = (mask*)&table[i];
    //gen(LOD , 0 , mk->address+1);
    gen(LOD , 0 , mk->address);
//  getsym();
    coun = 1;
    while(sym == SYM_LSQURBRA&&coun <= d[dp].c)
    {
          getsym();
          offset /= d[dp].l[coun++];
          gen(LIT,0,offset);
          assign_expr(fsys);
          gen(OPR,0,OPR_MUL);
          gen(OPR,0,OPR_ADD);
          if(sym == SYM_RSQURBRA)
            getsym();
          else error(32);
    }
    gen(LOA,level - mk->level,0);
    break;
```

```
else if(t[k].kind[flag] == ID_PDIM)
{
   int x = position(id);
   mk = (mask *) &table[x];
   gen(LIT , 0 , mk->address) ;
   //gen(LIT , 0 , mk->level) ;
   //v++;
   getsym();
}
```

（7）实现函数参数传递：

1. 函数声明：

修改了 enter 函数，可以处理 ID_PVARIBLE（局部变量），level 填 0，addres 填偏移值 offset，偏移值加一。在处理了函数名称后，向前一个期望左括号，再向前一个，如果是变量名，offset 置为 3，enter(ID_PVARIBLE)，再依次对余下的局部变量 enter。函数体开始时，站上预分配 4 加局部变量的个数。

Block 中函数声明部分：

```
offset = 3 ;
getsym();
if (sym == SYM_RPAREN)
    getsym();
else if(sym == SYM_IDENTIFIER){
        int v = 1;
        strcpy(f[fx].var[v++] , id);
        enter(ID_PVARIABLE);
        getsym();
        while(sym != SYM_RPAREN){
        if(sym == SYM_COMMA)
        {
            getsym();
            if(sym == SYM_IDENTIFIER)
               {
                   strcpy(f[fx].var[v++] , id);
                   enter(ID_PVARIABLE);
                   getsym();
               }
            else error(28);
        }
        else  error(5);
        }
```

修改后的 enter 函数：其中的 ID_PVARIABLE 用于维护局部变量，作用域中将会谈到。

```
375       case ID_PVARIABLE:
376           mk = (mask*)&table[tx];
377           mk->level = 0;
378           mk->address = offset++;
379           break;
380       case ID_FUNCTION:
381           mk = (mask*)&table[tx];
382           mk->address = cx;
383           mk->level = level;
384           fx ++ ;
385           strcpy(f[fx].name , id);
386           f[fx].c = 0;
387           break;
388       } // switch
389   } // enter
```

查找函数名字表的函数：

```
int fposition(char *id)
{
    int i;
    strcpy(f[0].name , id);
    i = fx + 1;
    while (strcmp(f[--i].name, id) != 0);
    return i;
}
```

2. 函数调用：

调用函数，先在站上分配 CALL 所需的四个位置，再依次加载个传入的实参，再使 top 回到调用前的位置，再加入 CAL 命令。
Factor 中函数调用部分：

```
case ID_FUNCTION:        //2017.11.3
    int k =fposition(id);
    int v = f[k].c;
    gen(INT,0,4);
    int flag = 1 ;
    getsym();
    //gen(POP,0,0);
    if(sym == SYM_LPAREN)
    {
        getsym();
        while(sym != SYM_RPAREN)
        {
            assign_expr(fsys);
            if(sym == SYM_COMMA)
                getsym();
            else if(sym != SYM_RPAREN){
                error(22);
                break;
            }
        }
        getsym();
    }
    gen(INT,0,-4-v);
    mk = (mask *) &table[i];
    gen(CAL, level - mk->level,  mk->address);
    //error(21); // Procedure identifier can not be in an expression.
    break;
```

Cal 指令的实现：活动记录栈从栈底方向到栈顶依次为：返回值，静态链（被调用函数新 bp 指针），动态链（老 bp），pc，局部变量。

```
1600        case CAL://2017.11.3
1601            // generate callee's AR at top of runtime stack!
1602            stack[top + 1] = 0; //return value
1603            stack[top + 2] = base(stack, b, i.l); //set up SL, static link
1604            stack[top + 3] = b; //DL, dynamic link, saving base address for caller's AR
1605            stack[top + 4] = pc;//save return address,next instruction after CAL
1606            b = top + 2; // new base poniter points to SL
1607            //top += 4 ;
1608            pc = i.a;    // reset ip
1609            //notice, when CAL executing, the top of stack is not changed
1610            break;
```

3. 函数体结束后，从名次表中删除 ID_PVARIBLE，局部变量不再有效。

7）测试实现

1.测试各运算符：
编译运行 ex1 文件。

```
var i,j,k;
{
  i := 1;
  j := 0;

  k := i + j;
  k := i - j;
  k := i * 2;
  k := i / 2;
  k := !i;
  k := i && j;
  k := i || j;
  k := i & j;
  k := i | j;
  k := i ^ j;
  k := 9 % 5;
  k := 4 << (i + 1);
  k := (i+1) >> 1;
  i += 2;
  i -= 2;
  i *= 2;
  i /= 2;
  i |= j;
  i &= j;
  i ^= j;
  i := 4;
  i <<= 2;
  i >>= 2;
  k := -i;
  k := !!i + 1;
  k := ++i;
  --i;
  k := i++;
  i--;

  j := 6;
  i := 0;

  i := i?i:j;
}.
```

```
Begin executing PL/0 program.
1
0
1
1
2
0
0
0
1
0
1
1
4
16
1
3
1
2
1
1
0
0
4
16
4
-4
2
5
5
4
5
4
4
6
0
6
End executing PL/0 program.
```

2. 测试 if-elif-else 语句，while 语句，for 语句：

```
var i,j,k;
{
        i := 0;
        j := 1;
if(i == j)
{
     k := i++;
     j := ++i;

    if((i < 0||j >= 3)&&k > 0)
{
     k := 10;
}
    else
{
    k := 20;
}

}
    elif(i < j)
{
    k := !i + j;
}
else
{
    k:= 1;
};
    i:= 0;
while(i <=2 )
{
    i ++;
    k ++;

};
for(i :=1;i<= 6; i++)
{
    j++;
}

}.
```

```
Begin executing PL/0 program.
0
1
2
0
1
3
2
4
3
5
1
2
2
3
3
4
4
5
5
6
6
7
7
End executing PL/0 program.
```

编译运行 ex2 文件。

3. 测试短路计算：

编译运行 ex3 文件。

```
var a,b,c,d,f,g;
{
    a:=1;
    b:=2;
    c:=3;
    d:=4;
    f:=0;
    g:=6;
  if((a<b ||c<d) && (f && g))
    {a := 20; }
  else {b := 30;}

}.
```

```
Begin executing PL/0 program.
1
2
3
4
0
6
30
End executing PL/0 program.
```

4.测试函数传参，数组传参，print()函数：
编译运行 ex4 文件。

```

var i , x[1][2][3] , y[9];


function q(var m,var n)
{
  if(m>1){
  return m * q(m-1,n-1) ;}
  else {return n;};
};
function p(var a[1][2][3])
{
  a[0][0][0] := 2;
  i := a[0][0][0] + a[0][1][1];
  return i;
};


{
  x[0][1][1] := 1 ;
  i := q(6,4);
  i := p(x);
  print(x[0][0][0]);
if(i>1)
  i := i+1;
}.
```

```
Begin executing PL/0 program.
1
-720
2
3
3
2
4
End executing PL/0 program.
```

5.测试数组的声明，存取：
编译运行 ex5 文件。

```
var i,j,a[3][11][4];
{

    i:= 10;
    j:= 2;
    a[2][i][3] := 12;
    j := a[2][10][3]++;
    j := --a[2][10][3];
    j += a[2][10][3];
}.

```

```
Begin executing PL/0 program.
10
2
12
13
12
12
12
24
End executing PL/0 program.
```

6.测试函数传参，callstack()函数，print()函数：
编译运行 ex6 文件。

```
var i,j,k , x[1][2][3] , y[9];
function q(var m,var n)
{
  if(m>1){
  return m * q(m-1,n-1);}
  else {
callstack();
return n;
}
};

function p()
{

  i := 4;
  return 3;
};

{
  i := q(6,4);
if(i>1)
  i := i+1;

  i := 5;
  k := 6;
  print();
  print(i,k);
  print();
  j := 1;
  print();
  print(i,j,k);
  print(100);
}.
```

运行结果：

```
Begin executing PL/0 program.
   68 TOP:   -1
   67        1
   66  PC:   17
   65  DL:   57
   64  SL:   2
   63  RE:   0
   62        2
   61        0
   60        2
   59  PC:   17
   58  DL:   50
   57  SL:   2
   56  RE:   0
   55        3
   54        1
   53        3
   52  PC:   17
   51  DL:   43
   50  SL:   2
   49  RE:   0
   48        4
   47        2
   46        4
   45  PC:   17
   44  DL:   36
   43  SL:   2
   42  RE:   0
   41        5
   40        3
   39        5
   38  PC:   17
   37  DL:   29
   36  SL:   2
   35  RE:   0
   34        6
   33        4
   32        6
   31  PC:   44
   30  DL:   2
   29  SL:   2
   28  RE:   0
   27        0
   26        0
   25        0
   24        0
   23        0
   22        0
```

```
   22        0
   21        0
   20        0
   19        0
   18        0
   17        0
   16        0
   15        0
   14        0
   13        0
   12        0
   11        0
   10        0
    9        0
    8        0
    7        0
    6        0
    5        0
    4  PC:   0
    3  DL:   0
    2  SL:   0
    1  RE:   0
-720
5
6

5
6

1

5
1
6
100
```

7. 测试 goto 语句，random()函数：
编译运行 ex7 文件。

```
var i;
{
    i := 1;
first:  i := random();
goto second;
    i := 2;
third:  i := 3;
second: i := random(23);
goto forth;
goto first;
forth:  i := 100;
}.
```

```
Begin executing PL/0 program.
1
8379
8379
10
10
100
End executing PL/0 program.
```

8. 测试 do-while 语句，for 语句，break/continue 语句：
编译运行 ex8 文件。

```
var a,b;
{
a:=0;
do
{
    a++;
    if(a==3)continue;
    a++;
}while(a<6);
    a++;

    for(a:=0;a<6;a++)
    {
        a++;
        if(a<3)
{
        a:= 4;
        continue;
};
        if(a==2)
        a++;
    };
    a++;
}.
```

```
Begin executing PL/0 program.
0
1
2
3
4
5
6
7
8
0
1
2
4
5
6
7
End executing PL/0 program.
```