**编译原理姜庆彩组实验报告**

**已完成：基础部分全部，扩展部分（6）（10）（11）总计110分**

**1.**[**添加注释**](#添加注释)

**2.**[**条件与短路计算**](#条件)

**3.**[**添加数组**](#数组)

**4.**[**参数传递**](#传参)

**5.**[**添加语句实现**](#添加语句实现)

**6.[print/random](#print)**

**8.**[更多的C 风格的运算表达式实现。](#（）8)

**11.[goto/break/continue](#（11）)**

**14.**[**read函数**](#（14）)

**小组成员分工：**

**姜庆彩：添加注释，条件与短路计算，参数传递的一部分，print/random，read函数**

**柳学富：添加语句实现，break/continue**

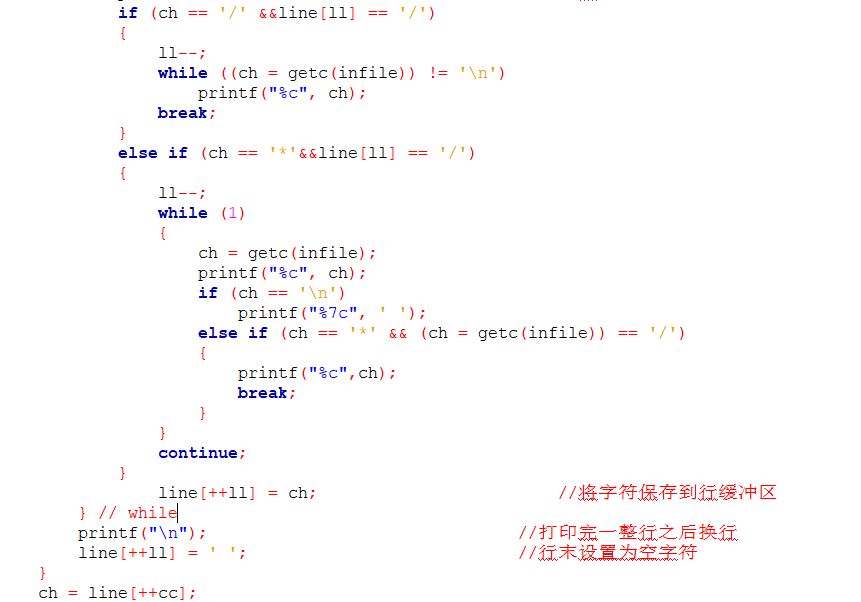
**林思捷：参数传递的一部分，更多c风格的语句**

**张瑾慧：添加数组，goto语句**

**1、添加注释（5 分） 完成人：姜庆彩**

块注释由/\*和\*/包含，不允许嵌套。

行注释由//开始直到行结束符（回车）。



这部分代码比较简单，唯一值得注意的是/\* \*/注释结束的判断标准是碰到了\*字符，而且下一个字符为/，这时表示此部分注释结束

**2、扩展PL/0 中“条件”：（15 分） 完成人：姜庆彩**

这种修改包括：

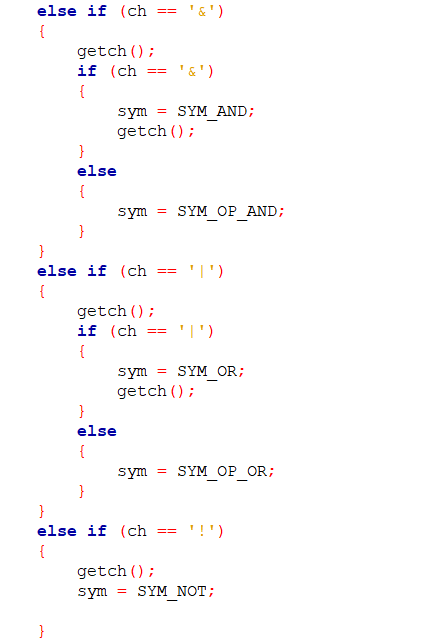
（i） 增加逻辑运算符 && 、|| 和 ！

（ii） 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”)。

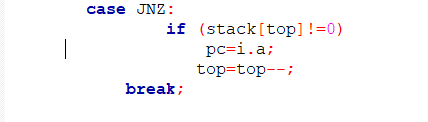
（iii） “条件”的短路计算。

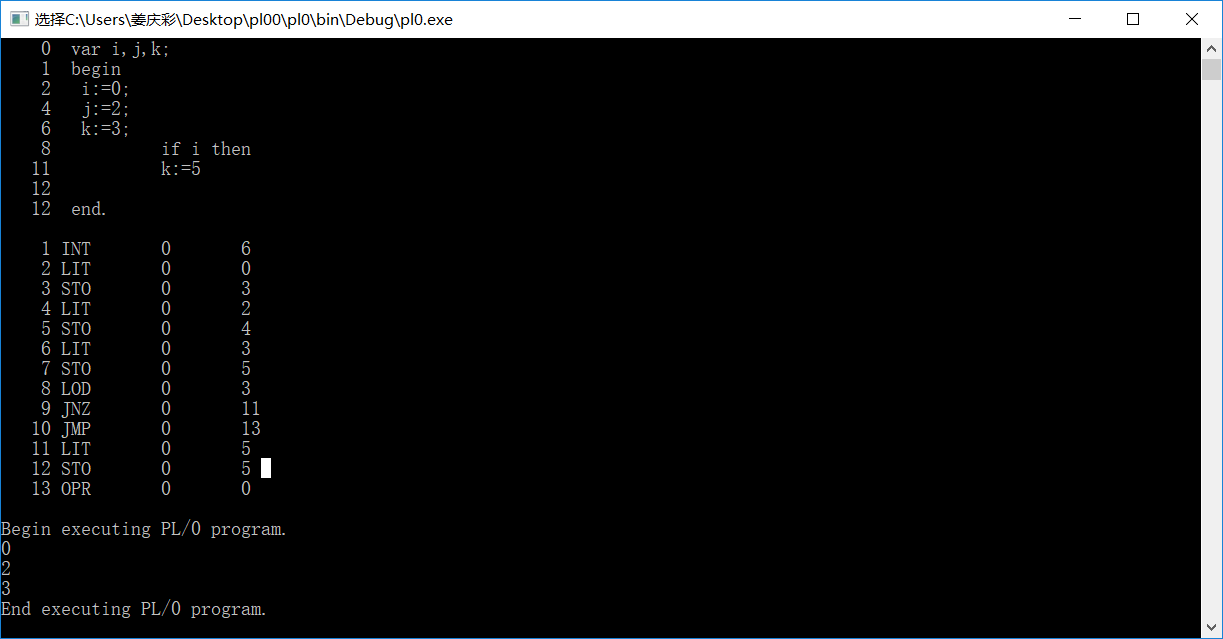
（i） 增加逻辑运算符 && 、|| 和 ！：



（ii） 把PL/0 语言中的“条件”概念一般化为C 语言那样(表达式值

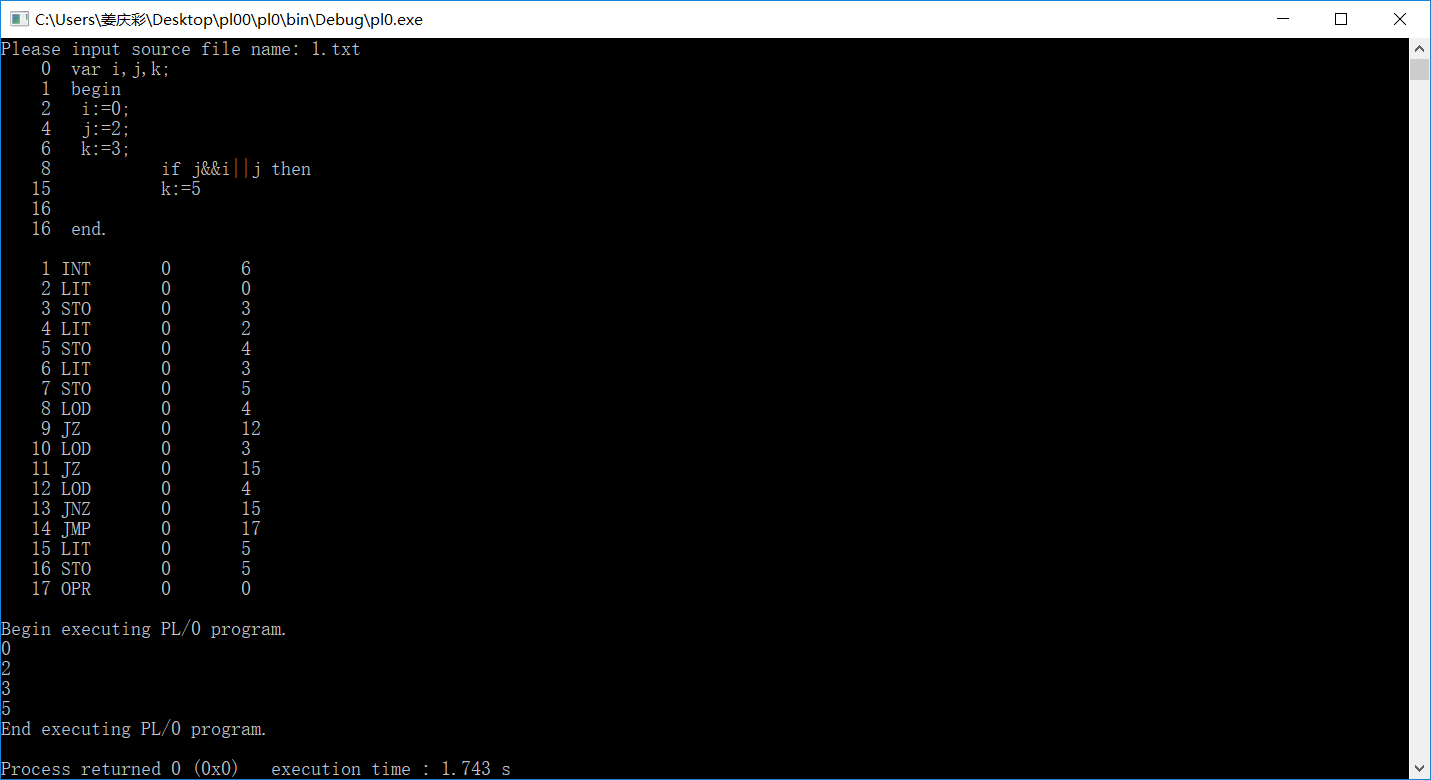
非零即为“真”)。：

这部分主要依托JNZ指令实现，JNZ指令判断栈顶值，非零时跳转

例子1：

这是一个最简单的例子，可以看到，代码第九行有JNZ 0 11指令，说明i！=0时会执行赋值，而第十行有JMP 0 13指令，说明JNZ若非真（i=0）则不赋值。运行结果也与期望一样。

例子2：



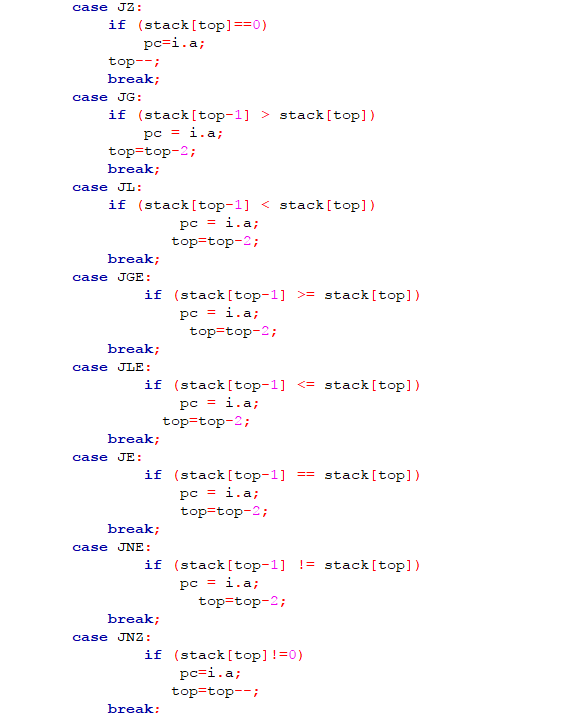
这个例子用到了后面短路计算的一些内容，首先是指令翻转，我设计时考虑了每遇到一个&&,则前后的指令都要翻转，由于我把添加的六个指令"JZ", "JG", "JL", "JE", "JNE", "JGE", "JLE", "JNZ"写成了对称的形式，因此只要code[cx - 1].f = 37 - code[cx - 1].f;（首尾enum和为37）即可。

回到这个例子，第九行指令为JZ 0 12说明若j等于0，可以跳过后面i的判断直接对j的值进行判断，而十一行指令为JZ 0 15说明&&前的值已为真（因为前面的JZ没有执行），如果&&后的值为真，即可直接进行赋值，十三十四行的结构与例子1完全一样，不再赘述。

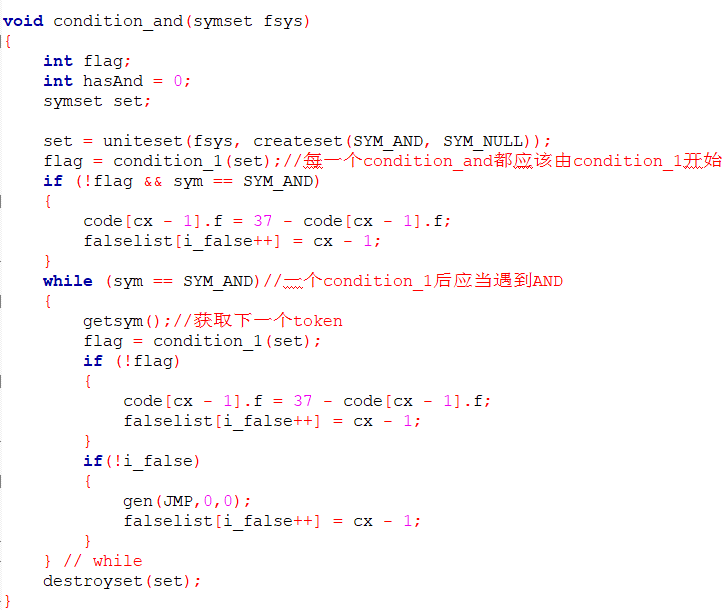
Not的实现在短路计算时一并详细叙述：

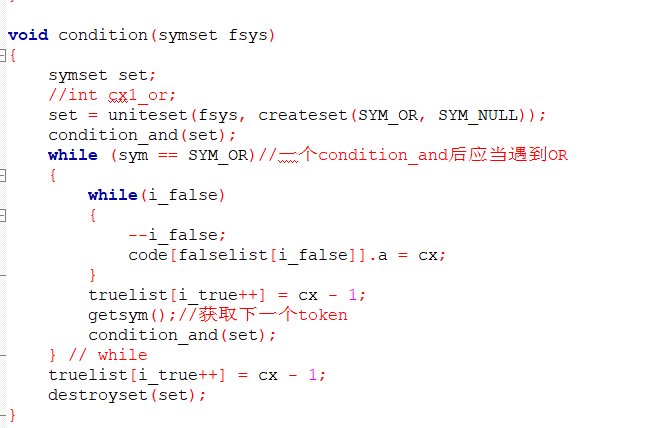
（iii） “条件”的短路计算。：

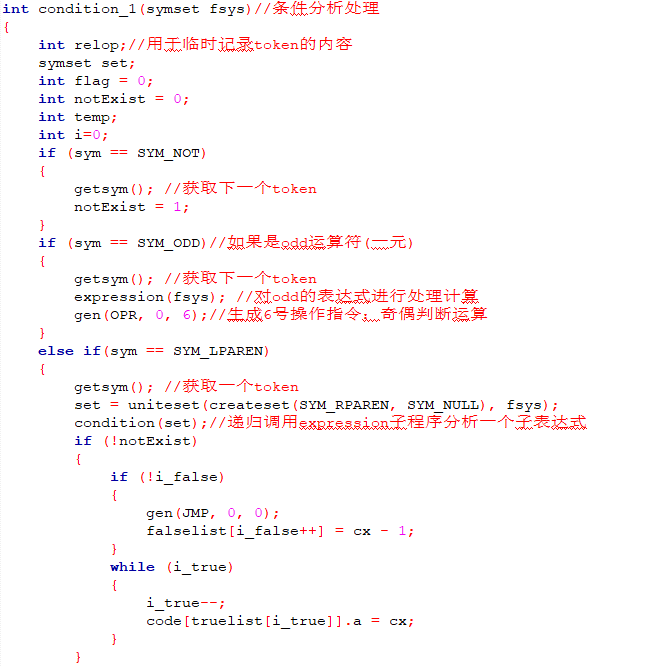
首先，当语句碰到&&时条件是需要翻转的（提供了跳出的途径），添加了8个指令"JZ", "JG", "JL", "JE", "JNE", "JGE", "JLE", "JNZ"，写成对称形式便于翻转。

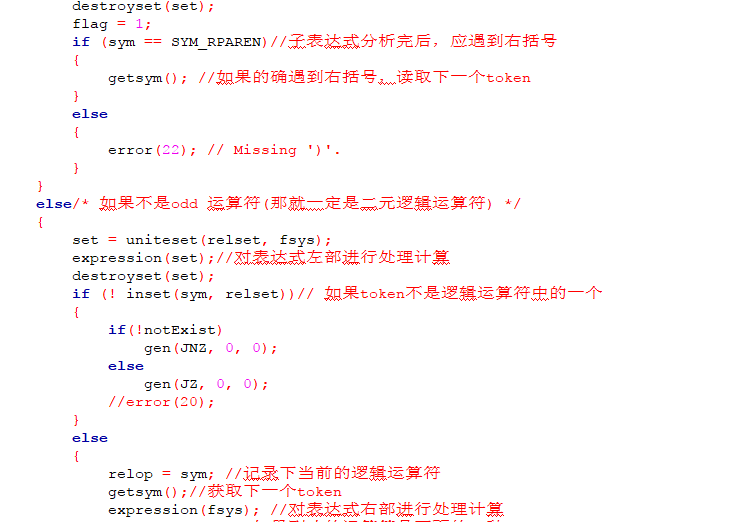


由于&&，||，!三者优先级不同，将condition改成condition，condition\_and，condotion\_1三个部分，将condition\_1的返回值改成int类型，以便判断条件表达式里面是否有括号，以便短路计算的设计。





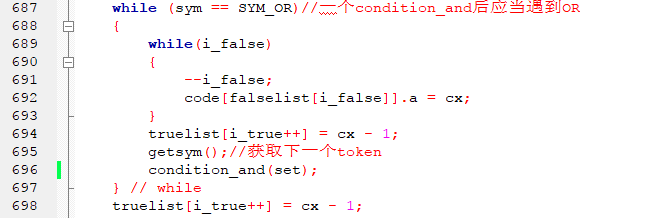




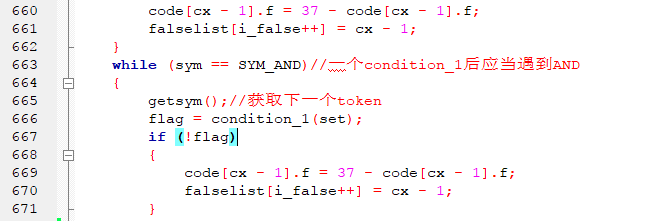


我的设计是将truelist，falselist，i\_false（falselist的个数），i\_true（truelist的个数设为全局变量）设置成全局变量

观察代码

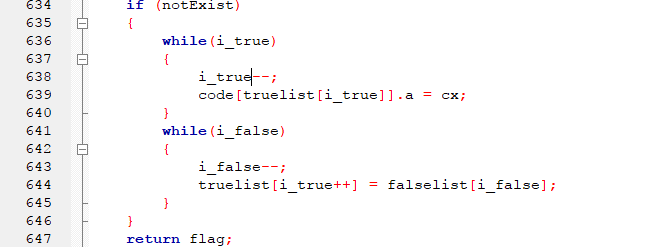


如果i\_false！=0说明or前面的语句中有falselist的值，则可以在这里进行回填，同时由于or总是伴随着truelist，则在这里要将cx-1放入truelist中。

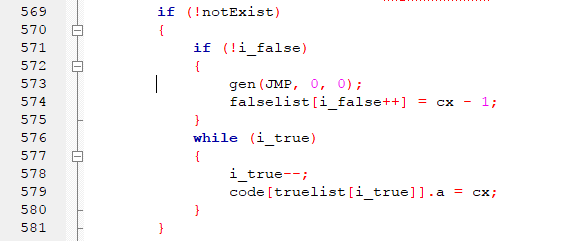


if (!flag && sym == SYM\_AND)，说明碰到&&时是没有括号表达式的，可以直接对指令翻转，并放入falselist中

condition\_1中增加了notexist变量记录前面是否有！符号

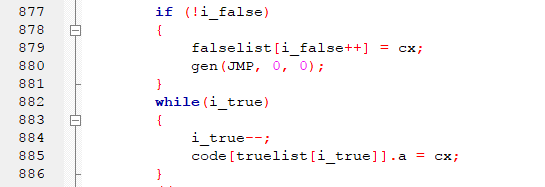


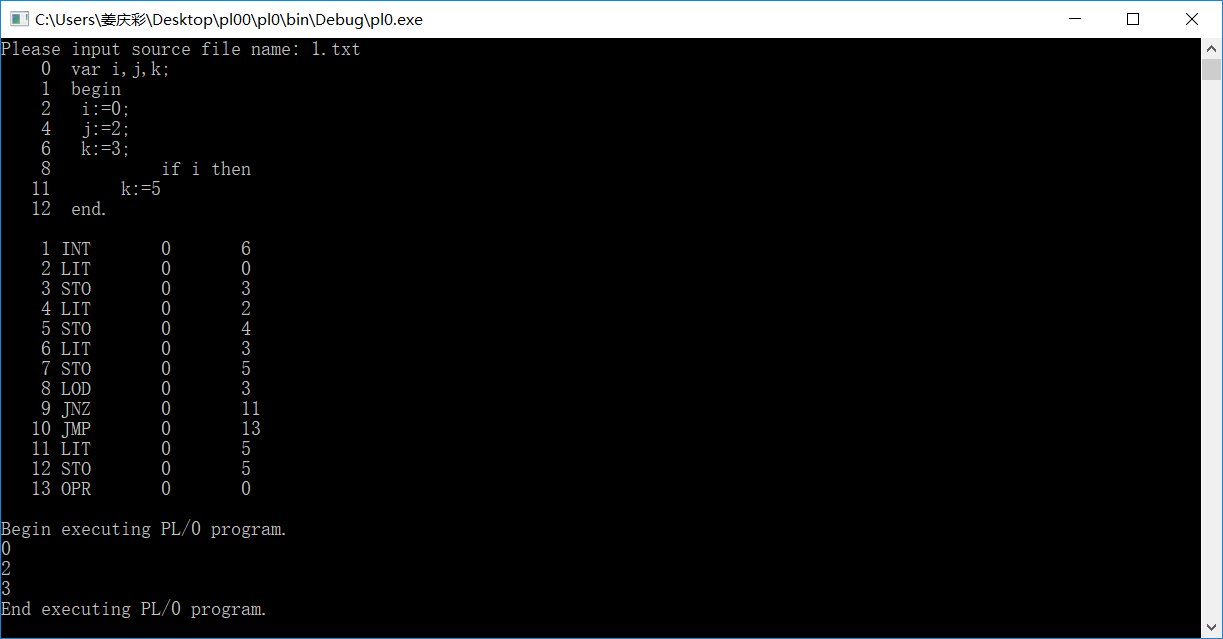
Condition\_1函数的最后，如果notexist非零，则将truelist回填，falselist转移给truelist。



如果此时i\_false的值为零，则需要产生一条jmp指令强制跳转，，否则处理完括号表达式程序将不知道自己将跳往何处。此时可以将truelist同时回填。

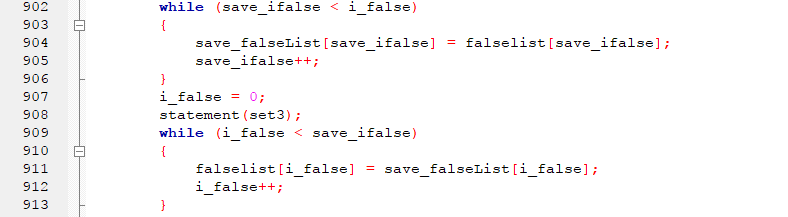
处理if表达式时类似：





第十行产生了JMP指令，强制跳转。

处理嵌套的if时，需要设置一个变量save\_ifalse保存i\_false的值，处理完之后再放回去。



Elif中处理情况类似，不再赘述

While指令唯一的区别在于设置了cx1保存当前代码序号，处理完while内容之后会跳回cx1

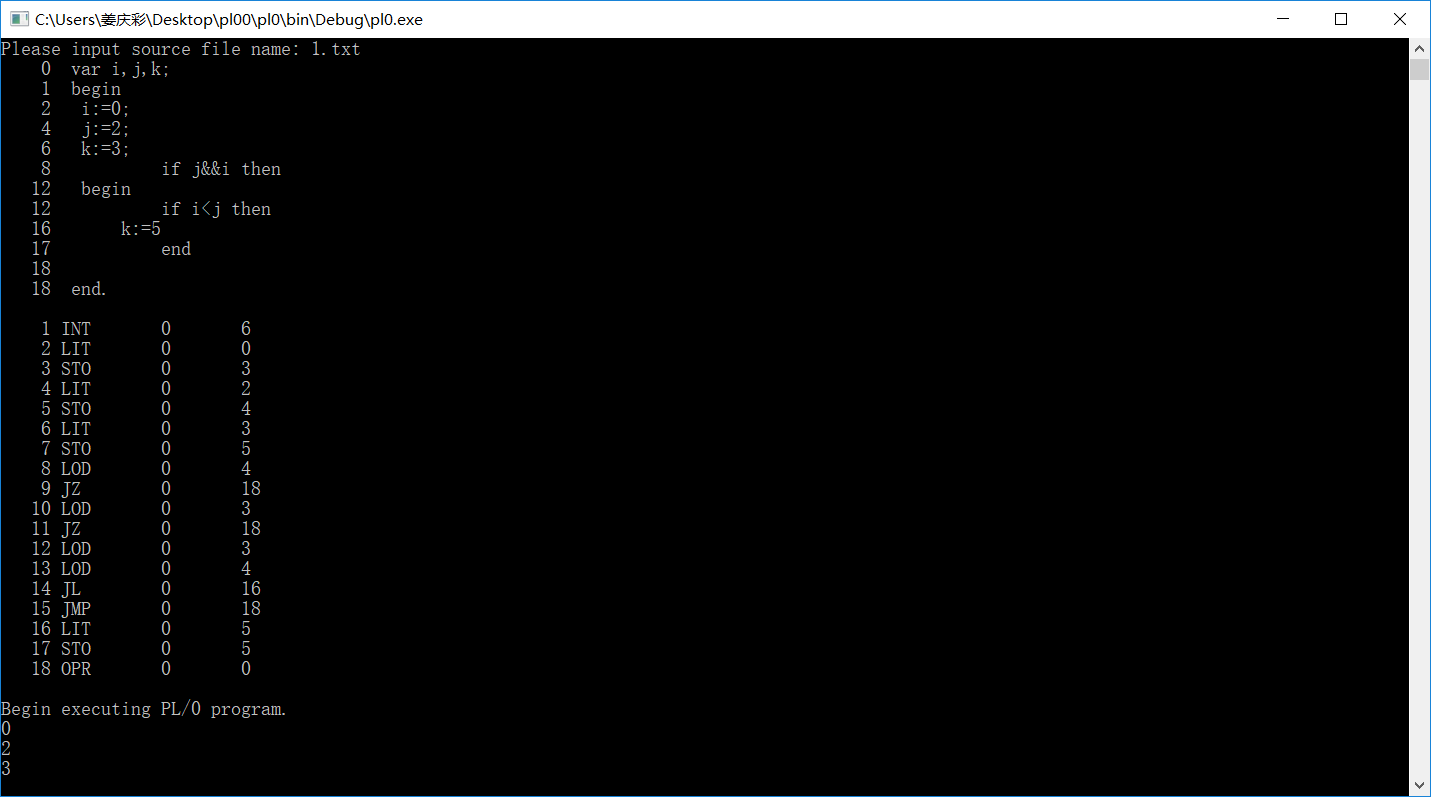
继续执行。

For语句类似，不再赘述。



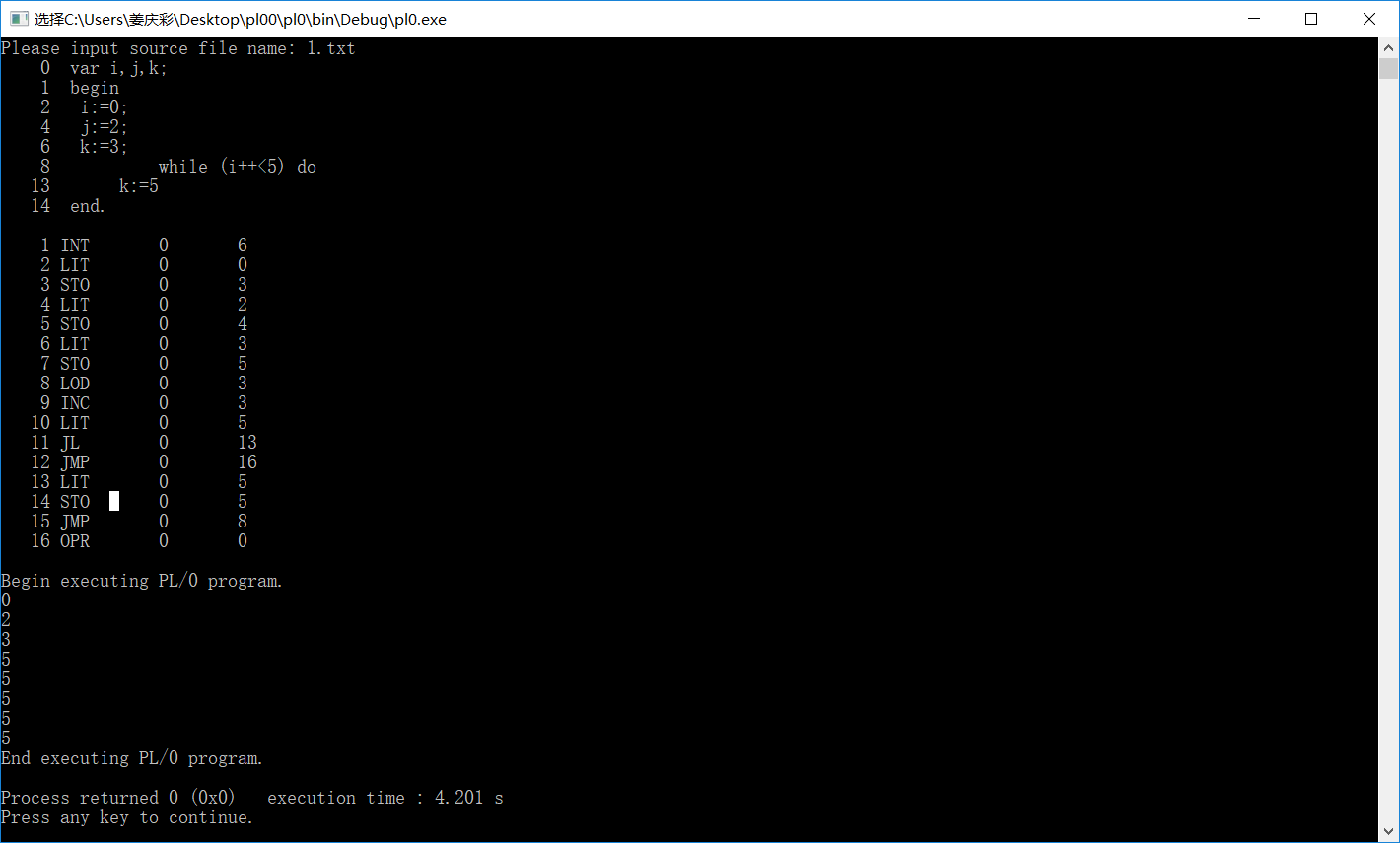


例子1.



第一层if中产生了两条jz 0 18 说明i或j等于0都会导致程序停止执行，第二层if中产生了jl 0 16指令，说明这层条件正确时会进行赋值，不正确时会产生JMP指令强制跳转。

例子2.



十五行产生了JMP 0 8表示跳回第八行进行判断

**（3） 添加数组（10 分）完成人：张瑾慧**

主要把数组作为一组特殊变量和若干常量（维度范围）来处理。

添加了函数arraydeclaration()以及函数addr(),修改了getsym（）使之能读取’[’和’]’，修改了enter（）函数使之能把同一名字下的数据作为数组读入，修改了vardeclaration()函数使之能判断并调用arraydeclaration（）定义数组，同时修改了所有使用变量的地方，若读到’[’则调用addr定位数组元素位置

以下为部分代码：

else if(ch == '[')

{

getch();

sym = SYM\_DIMSTART;

}

else if(ch == ']')

{

getch();

sym = SYM\_DIMEND;

}//dim tokens，属于getsym（）

if(position(id)>0)

{

mk = (mask\*)&table[tx];

mk->level = level;

mk->address = dx++;

}//属于enter（）

if(position(id)>0)

{

mk = (mask\*)&table[tx];

mk->level = level;

mk->address = dx++;

}//same name ,array,nothing changed but the address add 1

if (sym == SYM\_IDENTIFIER)

{

getsym();

if(sym == SYM\_DIMSTART)

arraydeclaration();//if '[' ,deal with array

else

enter(ID\_VARIABLE);//属于vardeclaration

void addr(mask\*mk)

{

static int space = 0;

int i,j;

j=position(id);//名字表中位置

getsym();

if(sym == SYM\_NUMBER)

{

space = space\*table[j].value+num;

j++;

getsym();

if(sym == SYM\_DIMEND)

{

getsym();

if(sym == SYM\_DIMSTART)

addr(mk);

while(space>0)

{

space--;

mk->address++;

}

getsym();

}

else error(28);

}

else if(sym == SYM\_IDENTIFIER)

{

space = space\*table[j].value+num;

j++;

expression(sym);

gen(STO,0,&i);

space = space\*table[j].value+i;

getsym();

if(sym == SYM\_DIMEND)

{

getsym();

if(sym == SYM\_DIMSTART)

addr(mk);

while(space>0)

{

space--;

mk->address++;

}

getsym();

}

else error(28);

}

}

//address the element of array

case ID\_VARIABLE:

mk = (mask\*) &table[i];

getsym();

if(sym == SYM\_DIMSTART)//array element , deal with address

{

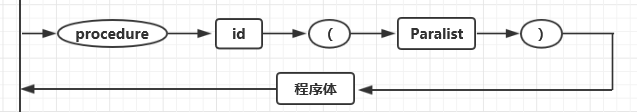
addr(mk);

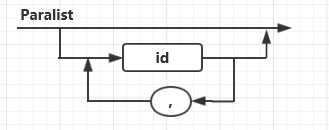
}//定位

**（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）。（15 分）完成人：林思捷，姜庆彩**

语法分析：





实现方式：

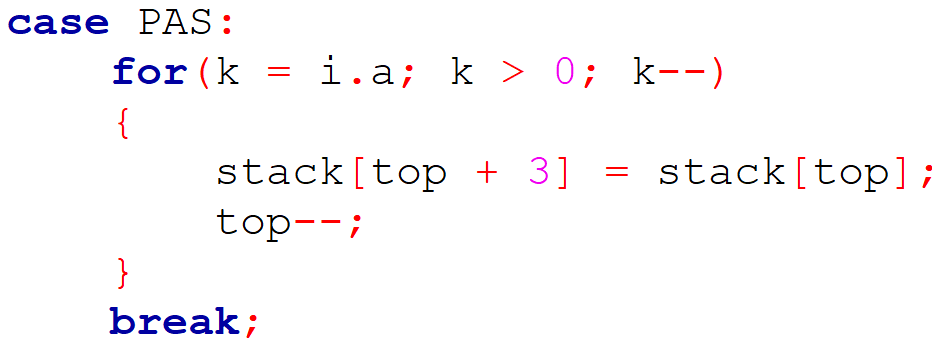
这里实现参数传递的方法是将需要传递的参数当作是过程的变量，如p(a, b)，就将a, b声明为过程p的两个局部变量。在调用过程的时候，把各个参数的值先依次放在栈顶，然后通过指令PAS把栈顶的参数的值都向上移动三位，然后再进行CAL，这就达到了把参数传递进去的目的。

在block中读到ID\_PROCEDUCE时，先将过程的名字记下来为tem，变量个数n初始化为0，并且把此时的dx存到save\_dx上，把dx赋值为3，level ++。这是为了在后面声明变量的时候把变量声明到下一个level中，并且有着正确的偏移地址，另外声明变量时需要用n记录变量的个数。声明完毕变量之后，将之前存的dx恢复，然后将之前的过程名tem放入id中，并且将过程名进表。之后调用新的block的时候一开始就将dx = 3 + n，这是因为变量在上一个level中声明，在这里把dx补回。

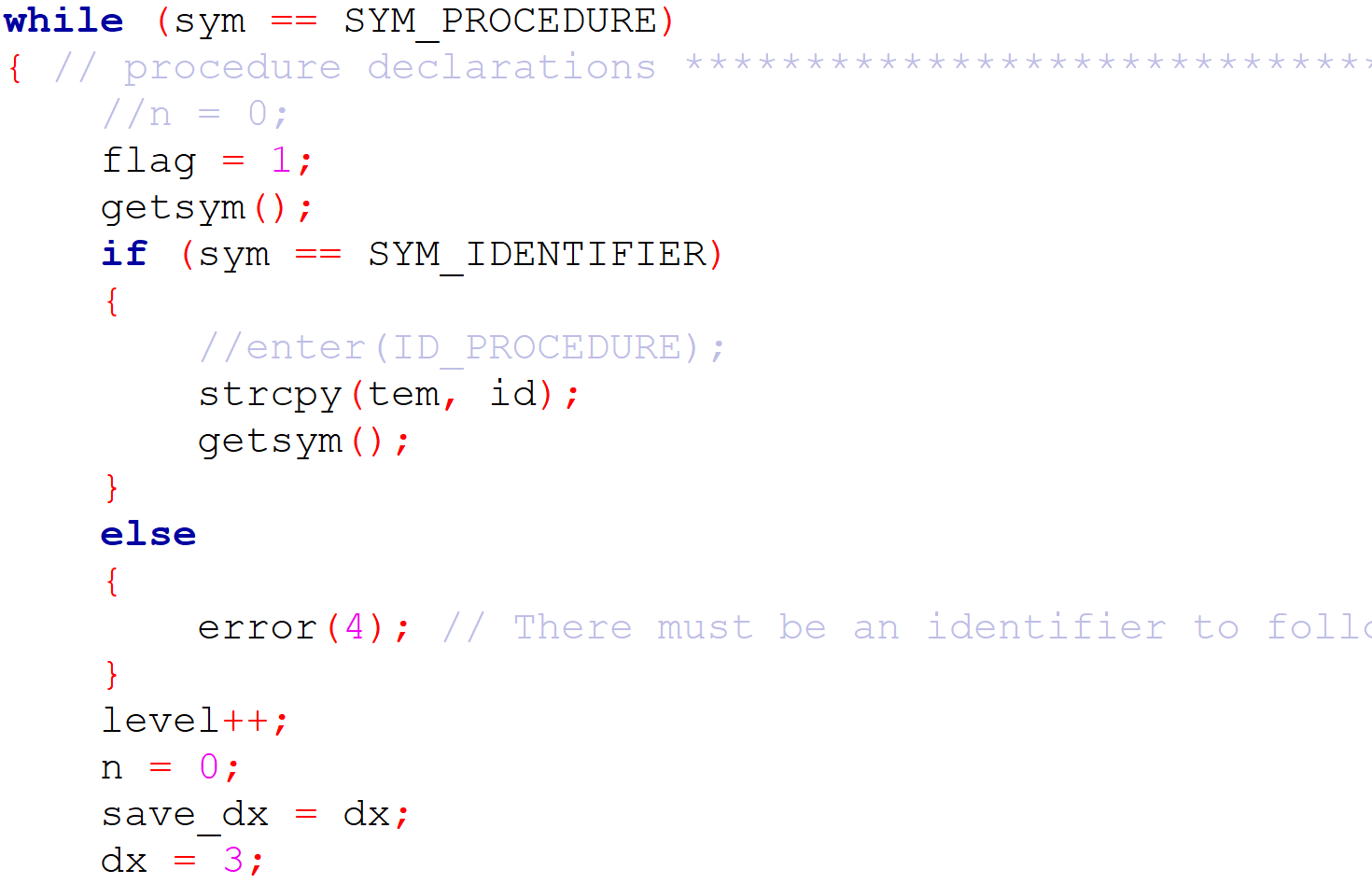
需要调用时，读到过程名就直接调用函数call。这个函数的作用就是将各个参数的值放在栈顶，生成PAS指令和CAL指令，完成调用的需求。其中PAS中会指定参数的个数，如

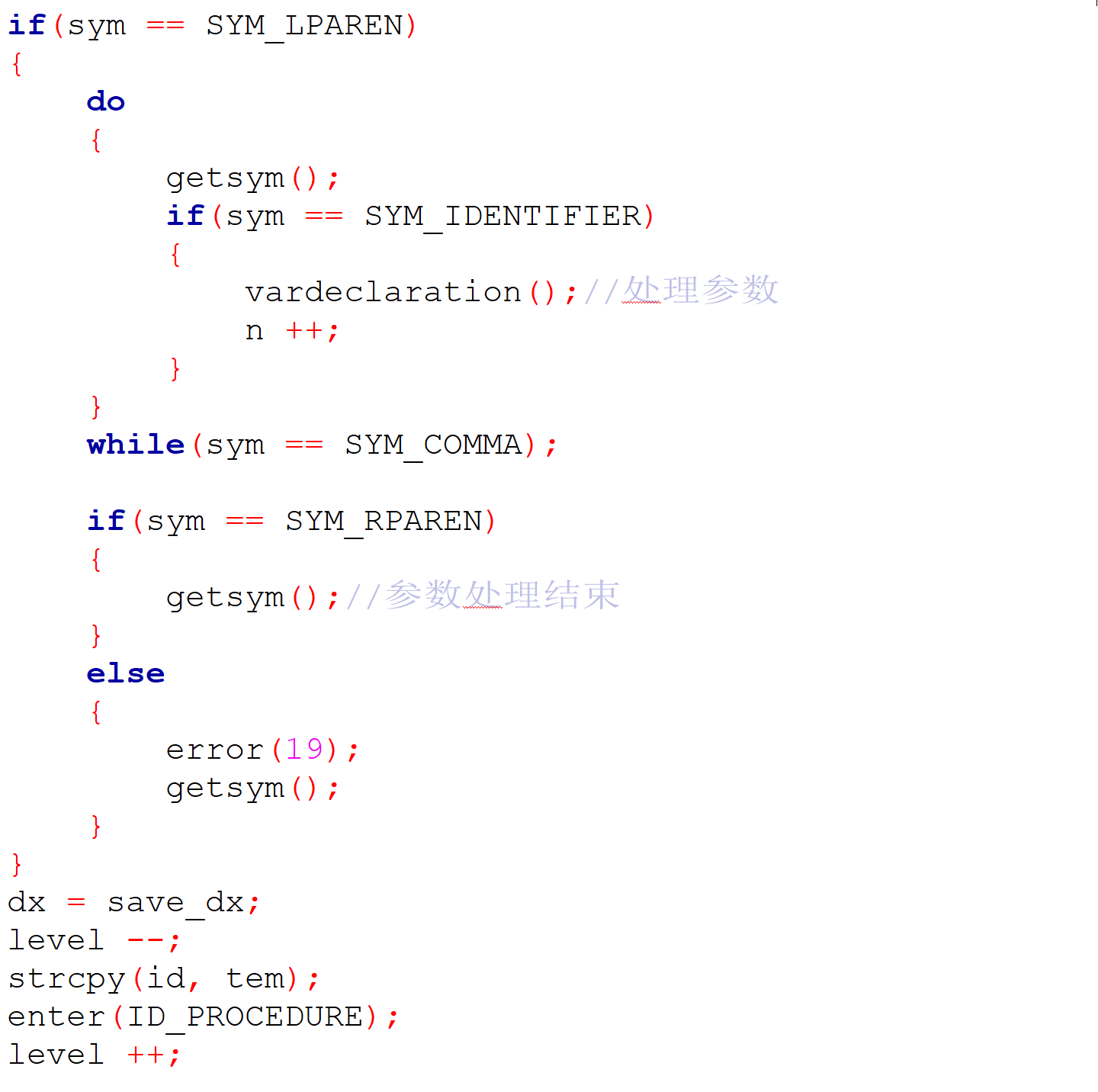
在p(a, b, c)时，GEN(PAS, 0, 3)，PAS指令如下，就可以留出三个位置留给等会CAL指令放置信息。

源码分析：



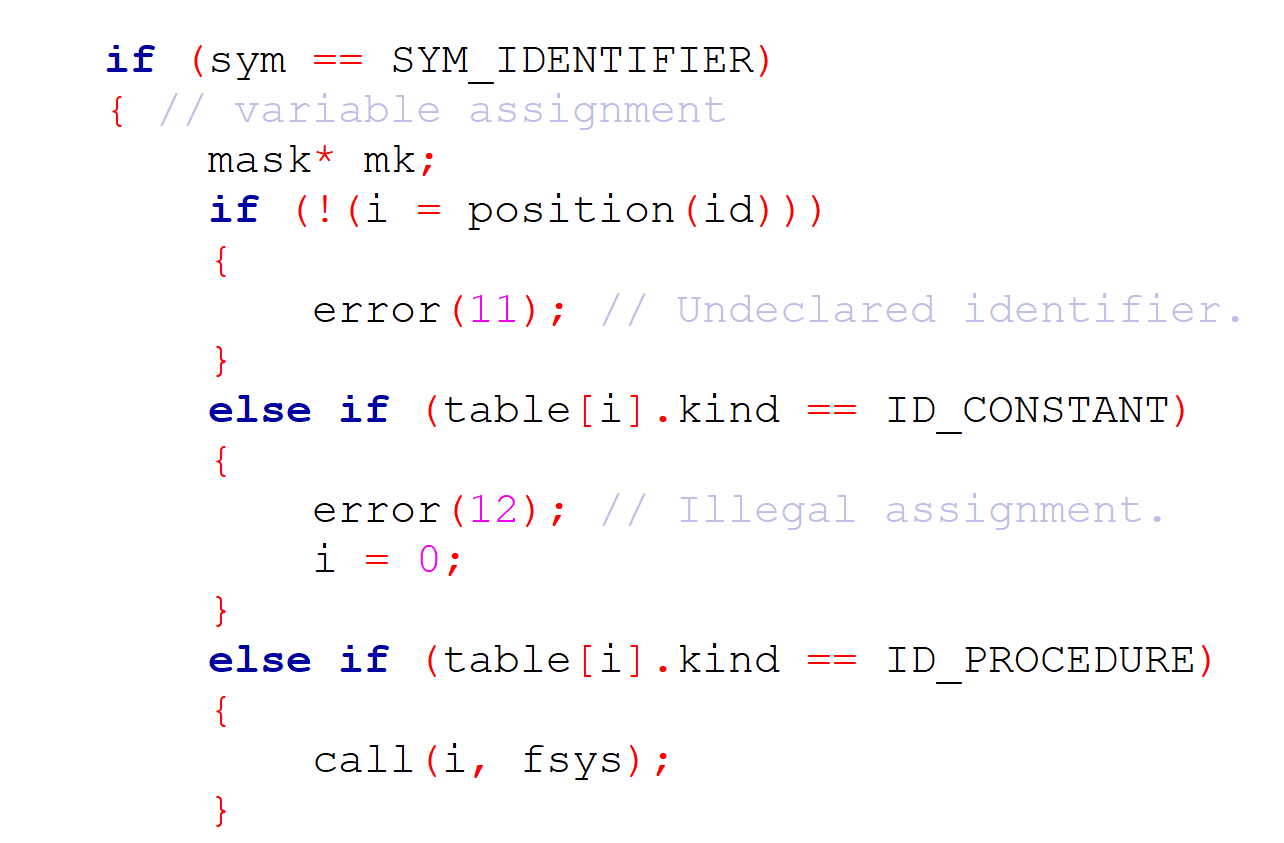
添加的PAS指令如上



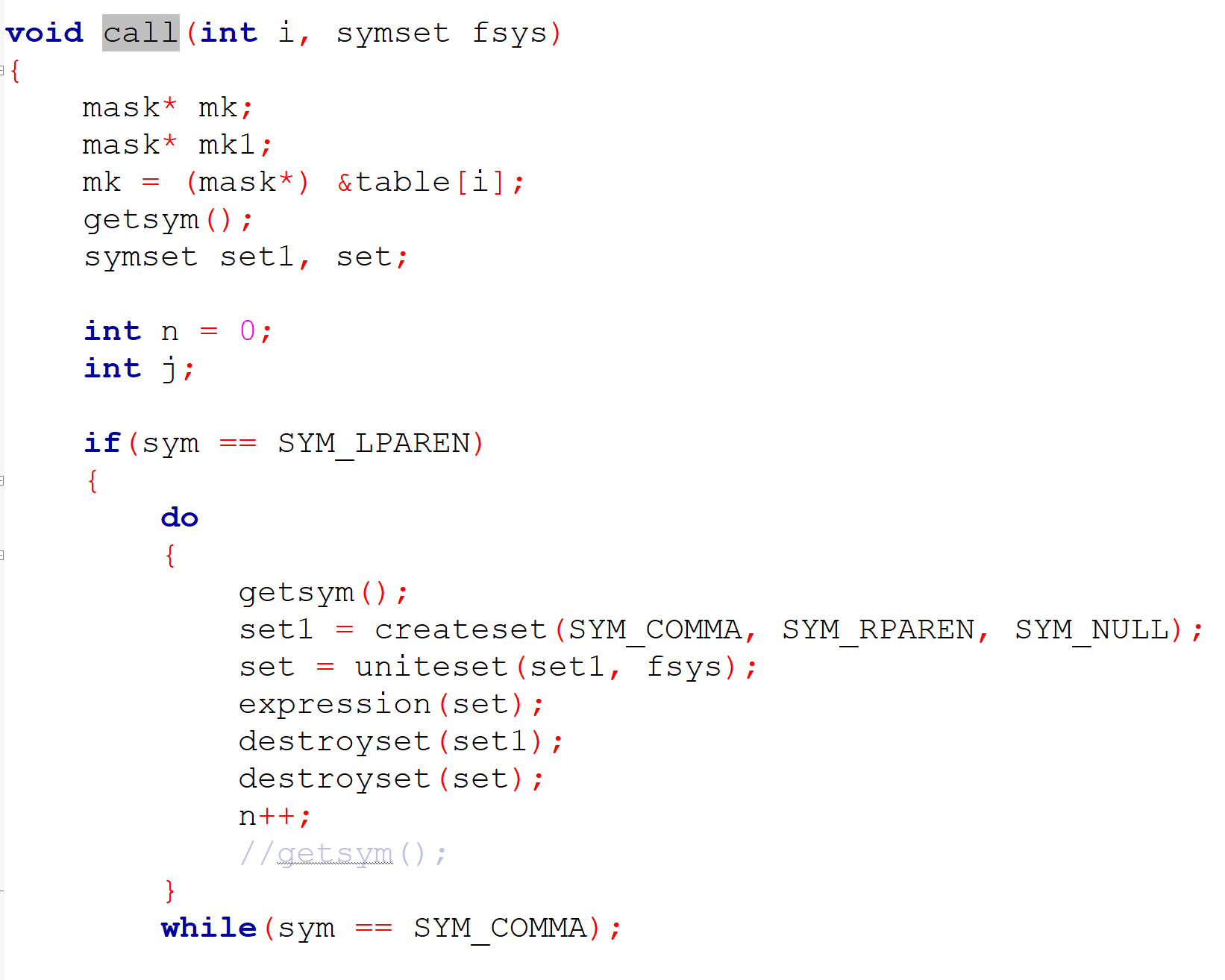


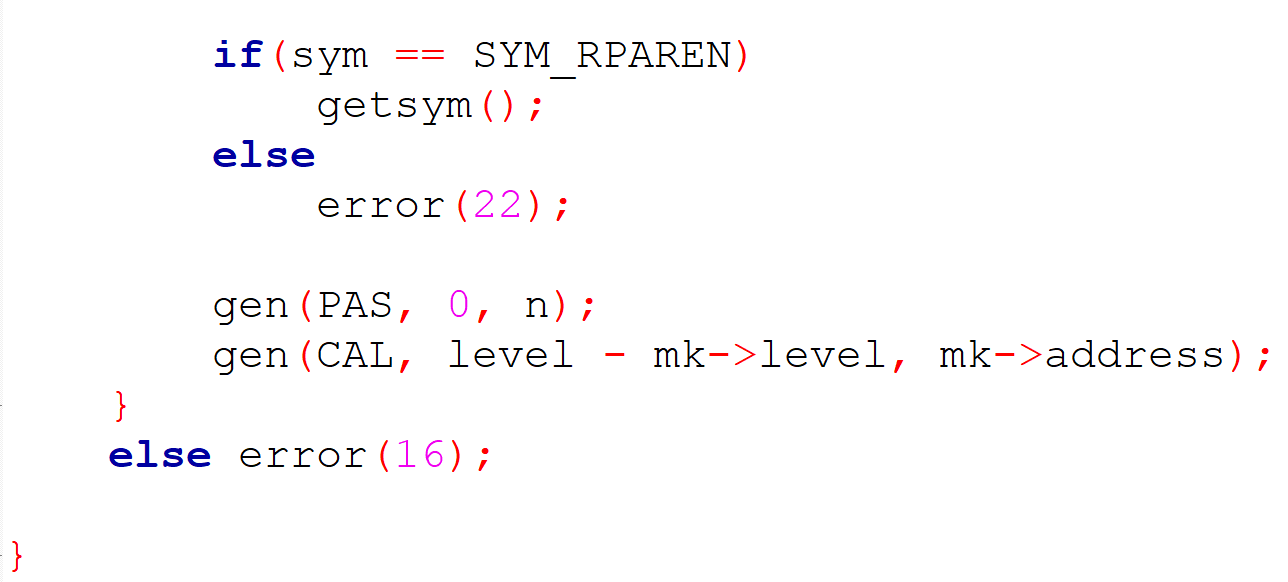
上图是在block中声明过程做的改动

可见在读到参数名的时候先将其存起来然后进行语法分析并且声明参数，最后再将过程名放入表中。



上图是statement里面，当读到ID\_PROCEDURE时，直接调用函数call

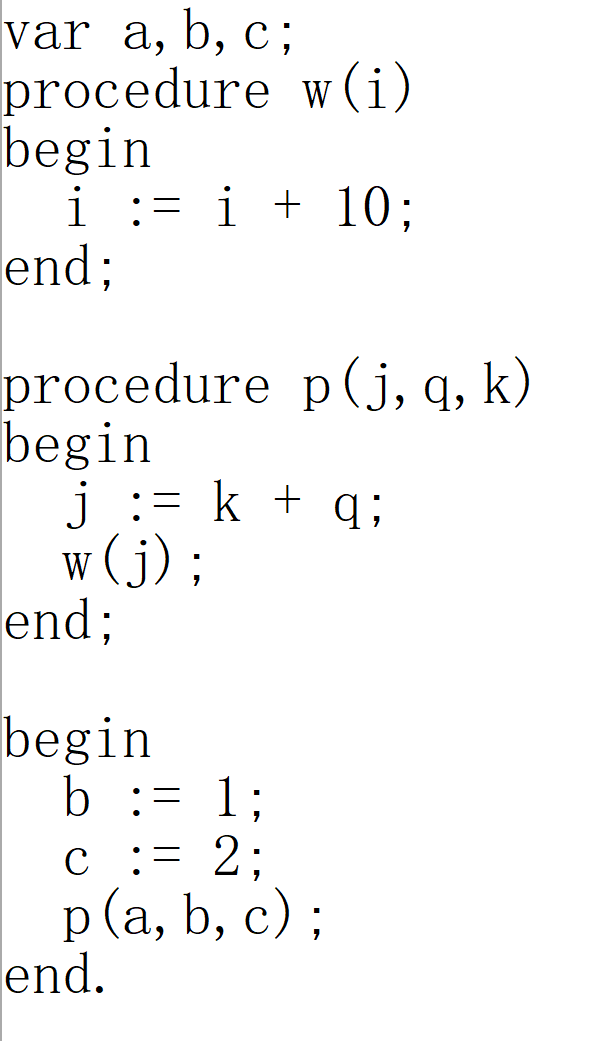
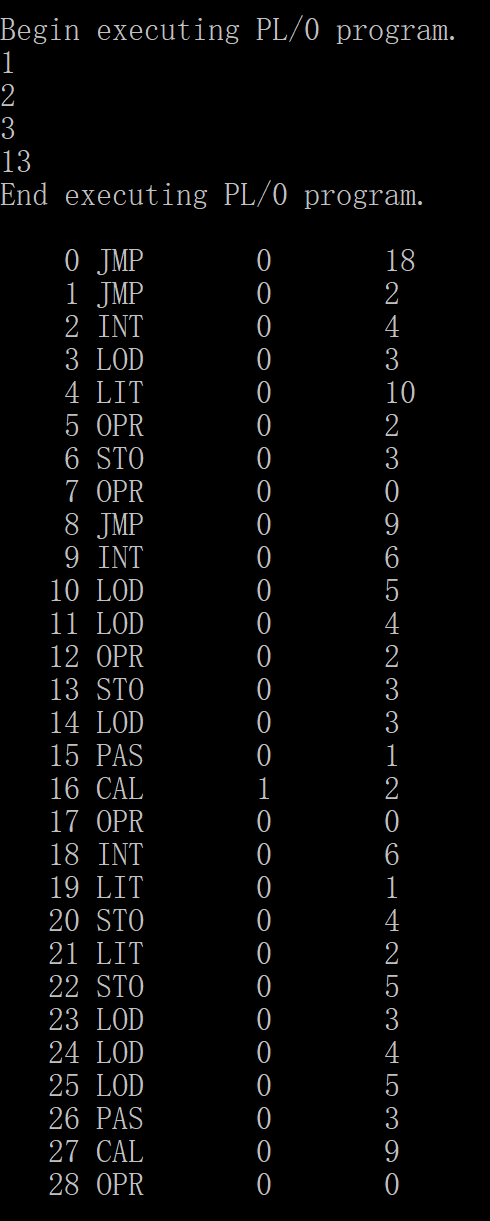




上图是call函数的内容，读到左括号之后，将每一个参数当成一个expression，使用while循环把每个参数的值放在栈顶，在其间用n来记录参数的个数。读完SYM\_RPAREN之后，就先生成PAS最后生成CAL。

测试例子：

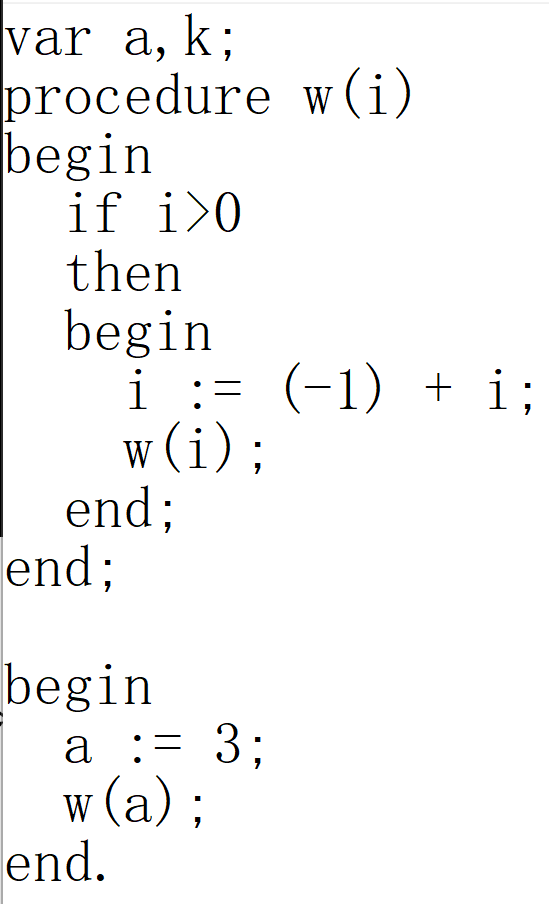
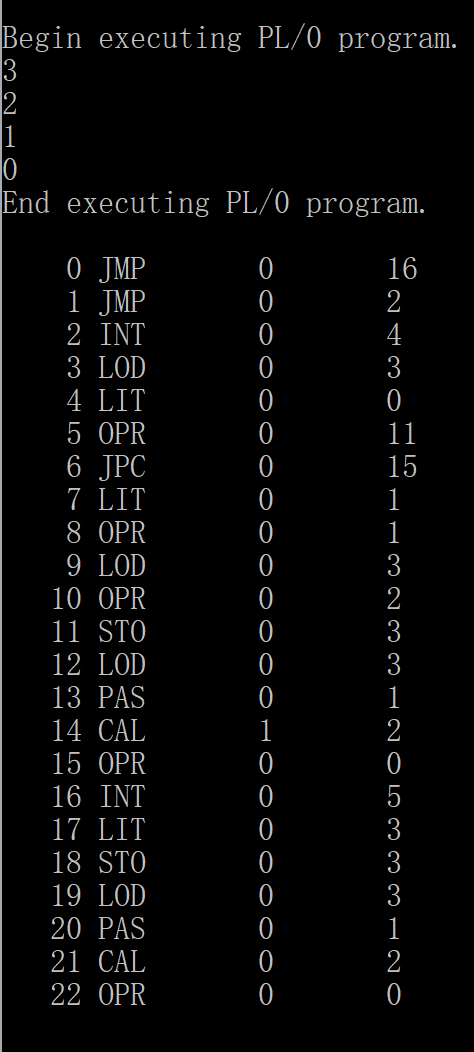
例1.

程序和运行结果如上所示

可见在每一次CAL之前都会进行一次PAS指令，并且PAS中含有参数的个数，为了可以确认移动参数的个数。而在PAS之前的指令，是将所要传递的值放在栈的顶部，传递顺序是过程括号内靠左的参数先放在栈顶。

例二

程序和运行结果如上所示

可见源程序是一个简单的递归程序，本例子测试了过程的递归调用是否可行，在这里看到输出结果是3 2 1 0，则显然得到了正确的结果。

**（5）添加语句实现:（25 分）完成人：柳学富**

1. if-elif-else结构语句

else if (sym == SYM\_IF)

{ // if statement

ifAppear = 1;

getsym();

set1 = createset(SYM\_THEN, SYM\_DO, SYM\_NULL);

set = uniteset(set1, fsys);

condition(set);

if (!i\_false)

{

falselist[i\_false++] = cx;

gen(JMP, 0, 0);

}

while(i\_true)

{

i\_true--;

code[truelist[i\_true]].a = cx;

}

//destroyset(set);

//destroyset(set1);

if (sym == SYM\_THEN)

{

getsym();

}

else

{

error(16); // 'then' expected.

}

//cx1 = cx;

//gen(JPC, 0, 0);

set2 = createset(SYM\_ELSE, SYM\_ELIF, SYM\_NULL);

set3 = uniteset(set2, fsys);

save\_ifalse = 0;

while (save\_ifalse < i\_false)

{

save\_falseList[save\_ifalse] = falselist[save\_ifalse];

save\_ifalse++;

}

i\_false = 0;

statement(set3);

while (i\_false < save\_ifalse)

{

falselist[i\_false] = save\_falseList[i\_false];

i\_false++;

}

while (sym == SYM\_ELIF)

{

jumpList[list++] = cx;

gen(JMP, 0, 0);

while(i\_false)

{

i\_false--;

code[falselist[i\_false]].a = cx;

}

getsym();

condition(set);

if (!i\_false)

{

falselist[i\_false++] = cx;

gen(JMP, 0, 0);

}

while (i\_true)

{

i\_true--;

code[truelist[i\_true]].a = cx;

}

if (sym == SYM\_THEN)

{

getsym();

}

else

{

error(16); // 'then' expected.

}

//cx1 = cx;

//gen(JPC, 0, 0);

save\_ifalse = 0;

while (save\_ifalse < i\_false)

{

save\_falseList[save\_ifalse] = falselist[save\_ifalse];

save\_ifalse++;

}

i\_false = 0;

statement(set3);

while (i\_false < save\_ifalse)

{

falselist[i\_false] = save\_falseList[i\_false];

i\_false++;

}

}

destroyset(set);

destroyset(set1);

if (sym == SYM\_ELSE)

{

jumpList[list++] = cx;

gen(JMP, 0, 0);

while(i\_false)

{

i\_false--;

code[falselist[i\_false]].a = cx;

}

getsym();

statement(fsys);

}

while(i\_false)

{

i\_false--;

code[falselist[i\_false]].a = cx;

}

while (list--)

code[jumpList[list]].a = cx;

destroyset(set3);

destroyset(set2);

}

其中本人设计if-elif-else的语法规则是整个if-elif-else语句以分号作为结尾，比如

if i>0 then

j:=1;

在这种情况下if语句当做终结，不能再跟elif和else，如需elif和else，则需要加begin-end，如下所示：

if i=0 then

begin

j:=0;

end

elif i>0 then

begin

j:=1;

end

else

j:=-1;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

程序示例和运行结果如下：

var i,j,k,l;

begin

i:= -2;

if i=0 then

begin

j:=0;

end

elif i>0 then

begin

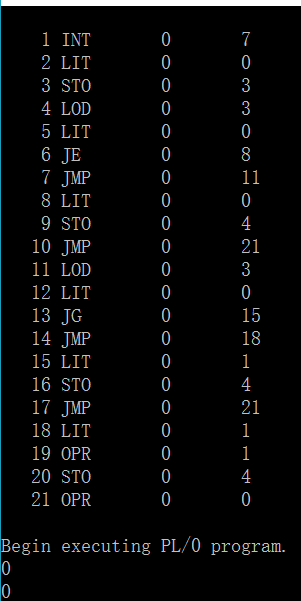
j:=1;

end

else

j:=-1;

end.

生成代码介绍：第六行为if条件判断开始，条件为真转入第8行赋值，赋值完成后跳到第21行结束，条件为假，进入第10行强制跳转到elif条件判断开始处，即第11行，条件为真则跳到第15行赋值，赋值完成后跳到第21行结束，条件为假，进入第10行强制跳转到else后的语句进行赋值，然后顺序退出。

C程序实现：在原来if的基础上，读入if符号后，完成条件判断和语句执行，则用一个while循环判断其后是否有elif，elif的条件判断和语句执行与if的写法类似，最后判断是否有else，在完成对语句的翻译。其中最大的改进是利用了短路运算的结果，再结合回填实现多处跳转，摒弃了JPC指令。同时由于每个if或者elif后的语句执行完成后都要强制跳转到整个if-elif-else语句后面，所以用一个jumplist来保存要回填的强制跳转地址，在这之前还要强制生成一条JMP指令，正如上述运行结果所述。

1. for结构语句以及自增自减

else if (sym == SYM\_FOR)

{

getsym();

if (sym == SYM\_LPAREN)

{

getsym();

if (sym != SYM\_SEMICOLON)

statement(fsys);

if (sym != SYM\_SEMICOLON)

error(10);

getsym();

if (sym != SYM\_SEMICOLON)

{

cx1 = cx;

condition(fsys);

if (!i\_false)

{

falselist[i\_false++] = cx;

gen(JMP, 0, 0);

}

}

if (sym != SYM\_SEMICOLON)

error(10);

getsym();

cx4 = cx;

if (sym != SYM\_RPAREN)

{

set1 = createset(SYM\_RPAREN, SYM\_NULL);

set = uniteset(fsys, set1);

statement(set);

destroyset(set);

destroyset(set1);

gen(JMP, 0, cx1);

}

else

{

cx--;

cx4 = cx1;

}

if (sym != SYM\_RPAREN)

error(22);

getsym();

while (i\_true)

{

i\_true--;

code[truelist[i\_true]].a = cx;

}

save\_ifalse = 0;

while (save\_ifalse < i\_false)

{

save\_falseList[save\_ifalse] = falselist[save\_ifalse];

save\_ifalse++;

}

i\_false = 0;

statement(fsys);

while (i\_false < save\_ifalse)

{

falselist[i\_false] = save\_falseList[i\_false] ;

i\_false++;

}

gen(JMP, 0, cx4);

while (i\_false)

{

i\_false--;

code[falselist[i\_false]].a = cx;

}

//code[cx2].a = cx;

}

else

{

error(26); //'(' expected;

}

}

for语句设计完全参照C语言语法规则，读入for关键字之后，对for后面括号内的语句进行解析，三个语句均可缺省，先执行第一个分号前的语句（如果为空不用执行），然后执行第二个语句的条件判断（为空则不用执行），然后读入第三个语句，对其进行解析，之后执行整个循环体。处理for的关键是各个基本块的跳转。首先执行第一个初始化语句，然后执行条件判断，为真转入循环体，循环体结束后强制跳转到for后面括号中的第三个语句，执行完强制跳转到条件判断，如此不断循环，知道条件判断为假之后跳出循环体，整个for结构语句执行结束。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

程序示例和运行结果如下：

var i,j,k;

begin

k:=0;

j:=0;

for(i:=0;i<3;++i)

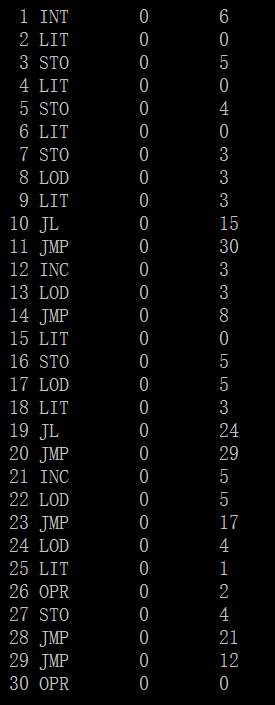
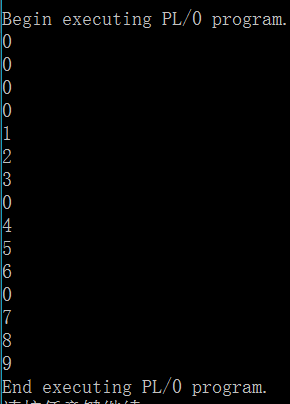
begin

for(k:=0;k<3;++k)

j:=j+1;

end;

end.

以外层循环为例，第8行开始是条件判断，正确则进入循环体，第15行开始则是整个循环体，循环体语句执行结束后跳转到第12行执行增值语句，如第29行所示，在第12行完成增值之后跳转到第8行进行条件判断，知道条件判断为假，跳转到第30行，整个循环结束。

1. 自增自减的实现

首先在符号表中添加了++、--，在设计上是每读入一个’+’（或者’-’）就会判断下一个输入符号是不是’+’（或者’-’），如果是，则将sym赋值为SYM\_INC（或者SYM\_DEC）。由于类似i++、++i都可以做为语句和因子，所以在statement和factor函数中都添加了关于++、--的解析。在statement还有所区别，就是如果是在statement完成自增自减，由于其值会保存在栈顶，所以需要弹出，在factor部分则不需要。

程序实现如下（factor和statement处理方式类似，区别在于是否生成弹栈指令）：

if (sym == SYM\_IDENTIFIER)

{ // variable assignment

mask\* mk;

if (!(i = position(id)))

{

error(11); // Undeclared identifier.

}

else if (table[i].kind == ID\_CONSTANT)

{

error(12); // Illegal assignment.

i = 0;

}

else if (table[i].kind == ID\_PROCEDURE)

{

call(i, fsys);

gen(POP, 0, 0);

}

else

{

getsym();

mk = (mask\*)&table[i];

if (sym == SYM\_BECOMES)

{

getsym();

expression(fsys);

gen(STO, level - mk->level, mk->address);

}

else if (sym == SYM\_INC)

{

gen(LOD, level - mk->level, mk->address);

gen(INC, level - mk->level, mk->address);

gen(POP, 0, 1);

getsym();

}

else if (sym == SYM\_DEC)

{

gen(LOD, level - mk->level, mk->address);

gen(DEC, level - mk->level, mk->address);

gen(POP, 0, 1);

getsym();

}

}

}

else if (sym == SYM\_INC)

{

getsym();

if (sym == SYM\_IDENTIFIER)

{

if ((i = position(id)) == 0)

{

error(11); // Undeclared identifier.

}

else

{

if (table[i].kind == ID\_VARIABLE)

{

mask \*mk;

mk = (mask \*)&table[i];

gen(INC, level - mk->level, mk->address);

gen(LOD, level - mk->level, mk->address);

gen(POP, 0, 1);

}

else

error(12);

}

}

else

error(14);

getsym();

}

else if (sym == SYM\_DEC)

{

getsym();

if (sym == SYM\_IDENTIFIER)

{

if ((i = position(id)) == 0)

{

error(11); // Undeclared identifier.

}

else

{

if (table[i].kind == ID\_VARIABLE)

{

mask \*mk;

mk = (mask \*)&table[i];

gen(DEC, level - mk->level, mk->address);

gen(LOD, level - mk->level, mk->address);

gen(POP, 0, 1);

}

else

error(12);

}

}

else

error(14);

getsym();

}

其中利用了新添加的INC和DEC指令，其作用是利用变量的层差和偏移修改其在栈中的值。程序的实现还区别了自增自减运算符是位于标识符前还是表示符后，如果是在标识符前，则先INC（或者DEC）后LOD，如果是在标识符后，则先INC（或者DEC）后LOD。还需要说明一点的是添加了POP指令，POP指令适用于两种情况的弹栈，一种是函数返回值的弹栈，一种是用于自增自减的弹栈，这两种方式的a域不同，前者的a域为0，后者的a域为1

POP指令实现如下：

case POP:

if (i.a)

top--;

else if (use)

{

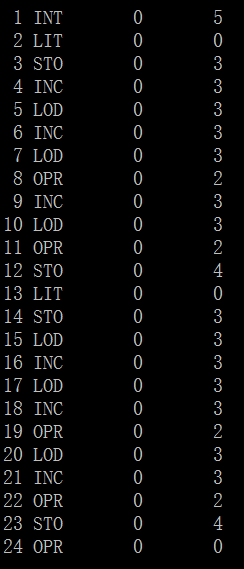
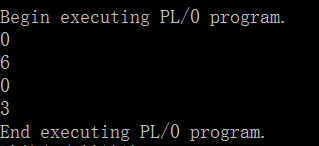
top--;

use = 0;

}

break;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*程序示例和运行结果如下：

1. exit语句的实现

程序实现如下：

else if (sym == SYM\_EXIT)

{

gen(STP, 0, 0);

getsym();

}

当读入exit语句时会产生停机指令，即STP（stop）指令，STP指令翻译如下：

if (i.f == STP)

{

printf("The program exited\n");

break;

}

首先判断指令的指令域是否为STP，如果是就跳出interpret中的循环，整个pl0处理机结束运行。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*程序示例和运行结果如下所示：

var i,j;

begin

i:=0;

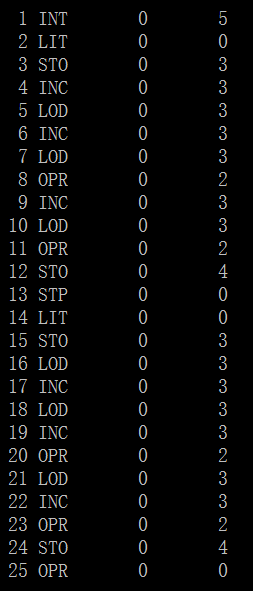
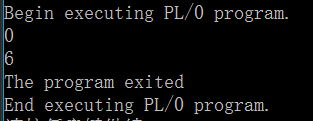
j:=(++i)+(++i)+(++i);

exit;

i:=0;

j:=(i++)+(i++)+(i++);

end.

可以看出一共有四条赋值语句只执行两条，因为在执行第二条赋值语句之后就产生了停机指令，后面的赋值语句停止执行。

1. return语句和返回值

程序实现如下：

else if (sym == SYM\_RETURN)

{

getsym();

expression(fsys);

gen(REV, 0, 0);

}

REV指令翻译：

case REV:

temp = stack[top];

top = b;

pc = stack[top + 2];

b = stack[top + 1];

stack[top] = temp;

use = 1;

printf("%d\n", stack[top]);

break;

当读入return关键字之后会处理后面的表达式，然后产生一条REV指令，REV指令的作用就是退出当前函数栈，控制权回到调用者，同时将表达式的值做为临时变量放在栈顶，同时输出该值，将use值设为1，供弹栈的时候判断。如果函数返回值作为因子的话，那么在使用时会自动退栈，如果作为语句的会产生一条弹栈指令弹出该值，从而不占用栈空间。设计弹栈指令时会利用产生返回值时设置的use值来决定是否要弹栈，弹栈完成后还会将该值复位。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

程序示例和运行结果如下：

var i,j;

procedure p(i)

begin

return i+1;

end;

procedure q(i)

begin

j:=i;

end;

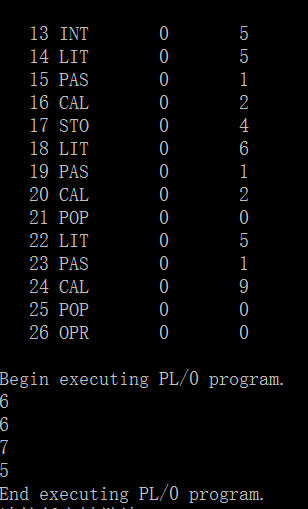
begin

j:=p(5);

p(6);

q(5);

end.

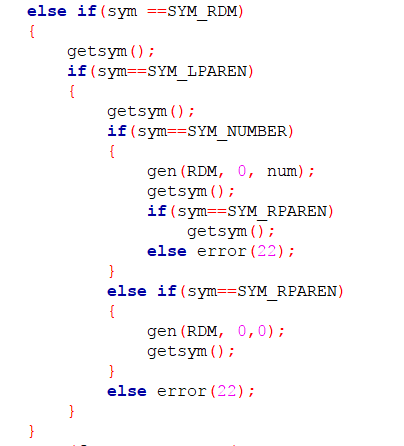


当返回值做为因子时没有产生POP指令，而单独做为语句时产生了POP指令弹栈。输出中第一个6出现是因为遇到REV指令，输出栈顶的值，第二个6是返回值赋值给j，7是第二次调用p时产生的返回值，5是调用q时内部的赋值。

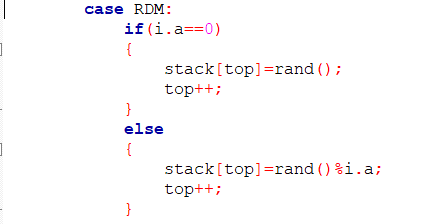
**扩展部分：**

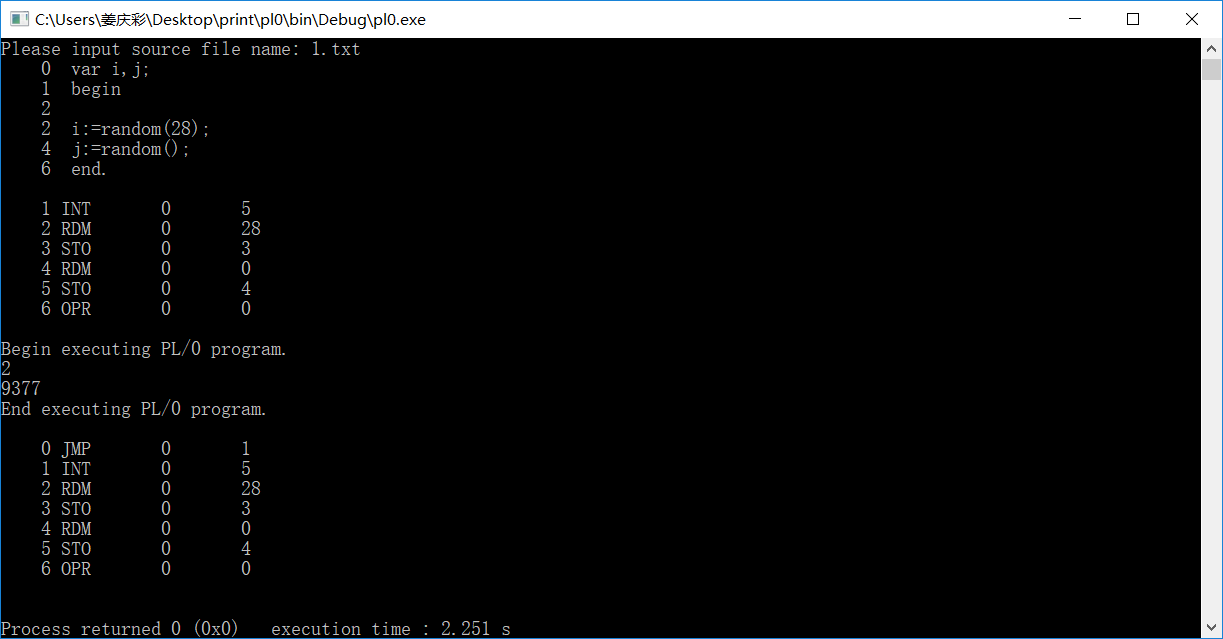
**（6）给PL/0 添加内置函数random 和print（10分） 完成人：姜庆彩**

Random函数的实现依赖于，每当发现random指令时，创建一条RDM命令，i.a的值就是括号内的值（括号内没有数时i.a缺省）

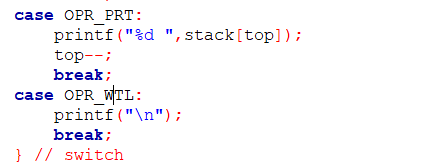


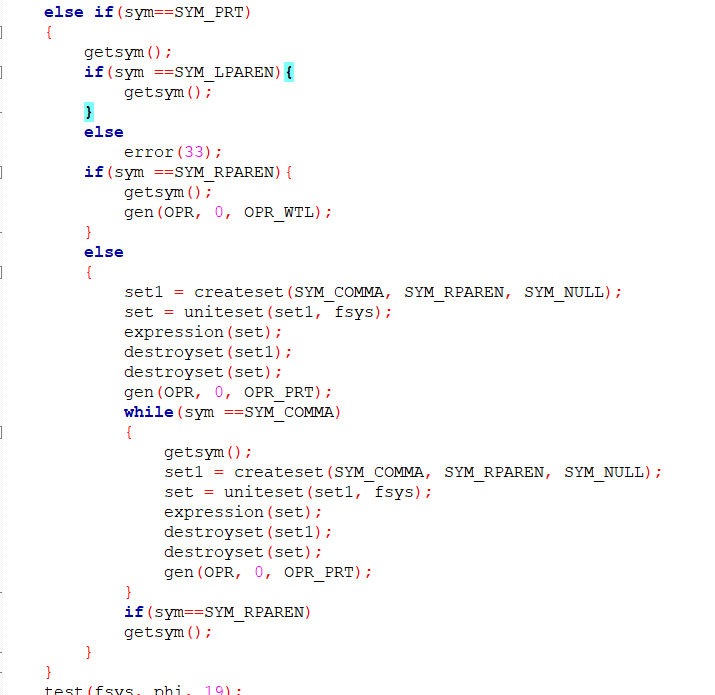
同时，在处理RDM指令时调用c库的rand（函数）

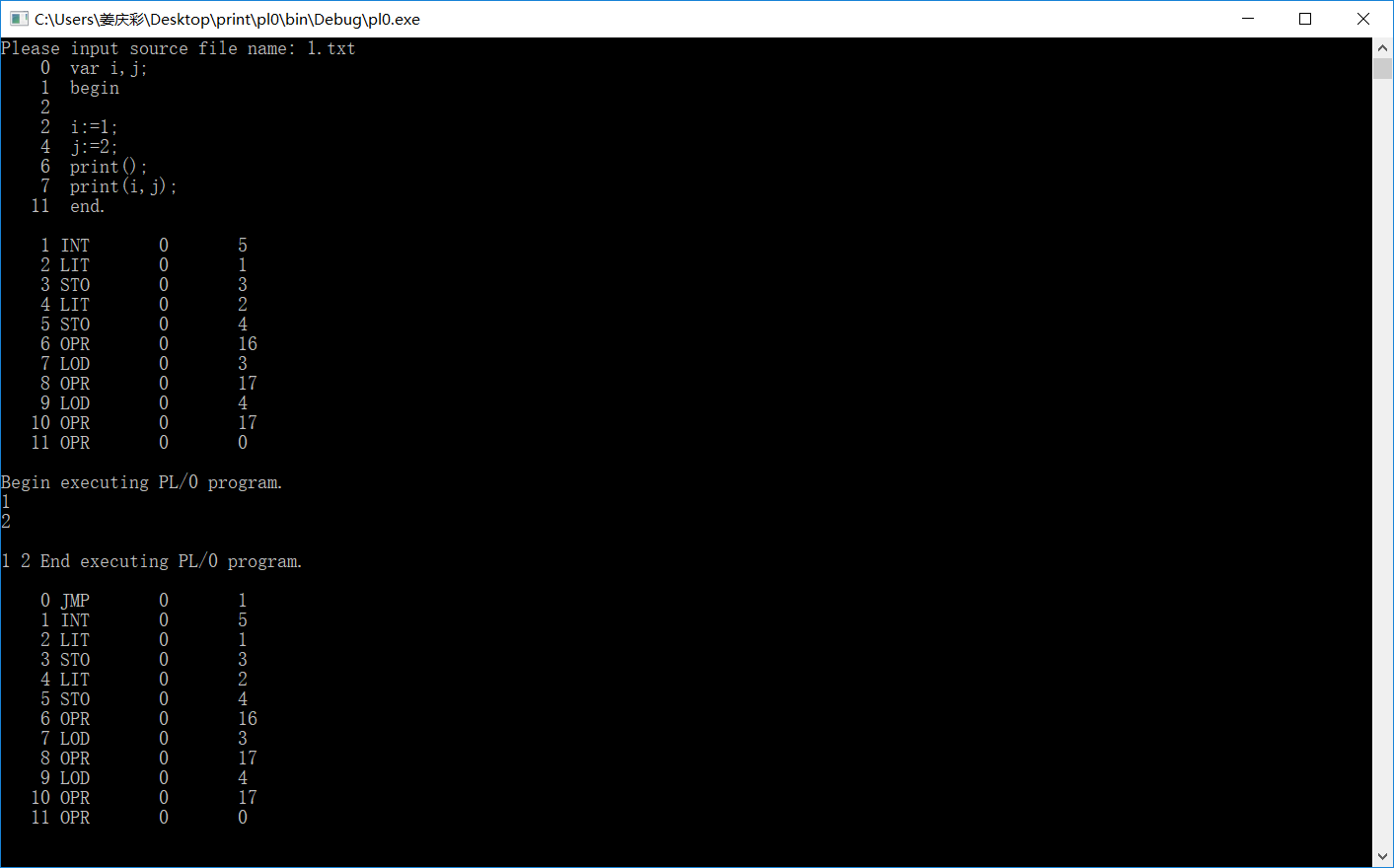




Print函数设计成可以打印空行（OPR\_WLT）与可以打印任意参数个数字（OPR\_PRT）





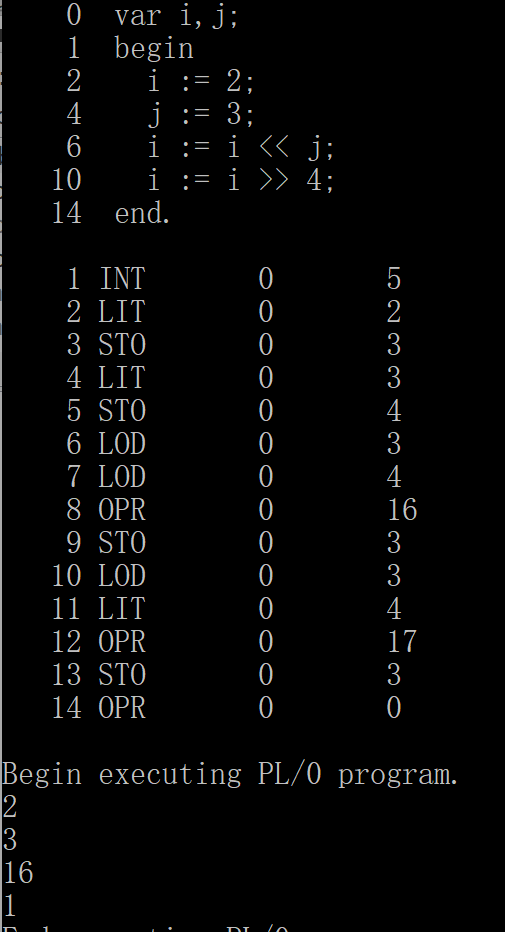


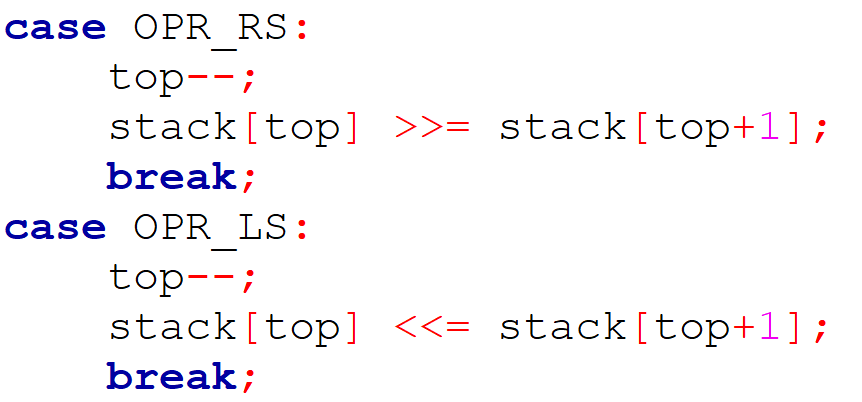
可以看到，在打印一个空行之后，i与j的值在同一行中打印了

**（8）更多的C 风格的运算表达式实现。**

在本拓展中实现了c语言风格表达式中的 >>, <<, +=, -=, /=, \*=运算符和赋值表达式：=的多重赋值。下面分成三个部分进行介绍。

**一、左移和右移（<<, >>）**



首先看上图中的一个例子，由输出结果可知程序运行正确

首先定义了OPR\_LS和OPR\_RS，分别是将次栈顶的值左/右移栈顶的值的次数。



然后新加入函数expression，将原来的expression改成expression2。在这个函数中生成了左移和右移的操作。

（11）**（15分）此部分分为（goto）与（break、continue）两部分，goto由张瑾慧完成，break/continue由柳学富完成**

int labeltotal = 0;

int gotototal = 0;

char labelchar[10][11];

int labelcx[10];

char gotochar[10][11];

int gotocx[10];

记录goto的和goto点的地址和名称

把SYM\_GOTO添到statbegsys和对应位置

if (sym == SYM\_IDENTIFIER)

{ // variable assignment

mask\* mk;

if (!(i = position(id)))

{

getsym();

if(sym == SYM\_COLON)

{

set1 = createset(SYM\_COLON,SYM\_NULL);

fsys = uniteset(set1, fsys);

destroyset(set1);

flag=0;

for(j=0;j<labeltotal;j++)

{

if(!strcmp(labelchar[j],id))

flag=1;

}

if(flag)

error(31);

else{

strcpy(labelchar[labeltotal],id);

if(labeltotal >= 10)

error(33);

else

{

labelcx[labeltotal] = cx;

labeltotal++;

}

}

getsym();

}

else

error(11);

test(fsys, phi, 19);

statement(fsys);

statement中若是不明标示符开头并接：为goto的可能点，记录位置和名称

else if ( sym == SYM\_GOTO)

{

flag = 0;

getsym();

if(sym != SYM\_IDENTIFIER)

error(8);

else

{

getsym();

if(sym != SYM\_SEMICOLON)

error(5);

else

{

strcpy(gotochar[gotototal],id);

gotocx[gotototal]=cx;

gotototal++;

gen(JMP,0,0);

}

}

statement中出现goto，记录位置和名称

block(set);

for(i=0;i<gotototal;i++)

{

for(j=0;j<labeltotal;j++)

if(!strcmp(gotochar[i],labelchar[j]))

code[gotocx[i]].a = labelcx[j];

}

主函数block后面回填jmp地址Break、continue：

1. break语句

代码实现如下所示：

PART1：

else if (sym == SYM\_BREAK)

{

if (!loopLevel)

error(28);

else

{

getsym();

breakList[breakCount].cx = cx;

breakList[breakCount].level = loopLevel;

breakCount++;

gen(JMP, 0, 0);

}

}

PART2：

while (breakCount)

{

if (breakList[breakCount - 1].level == loopLevel)

{

breakCount--;

code[breakList[breakCount].cx].a = cx;

}

else

break;

}

其中PART1是对break语句的解析，每次读到break之后，先判断其是否在循环中（即第一个if判断），如果不在循环中则报错，如果在循环中则进行后续操作，首先有一个breakList保存待回填的break语句对应的代码行数，同时还保存其所在的循环层级，PART2出现在处理while语句与for语句中的循环语句后，处理完循环体之后开始回填，只回填当层循环的break语句，其中looplevel就保存了当前的循环层级。

程序示例如下：

var i,j;

begin

for(i:=1;i<5;i++)

begin

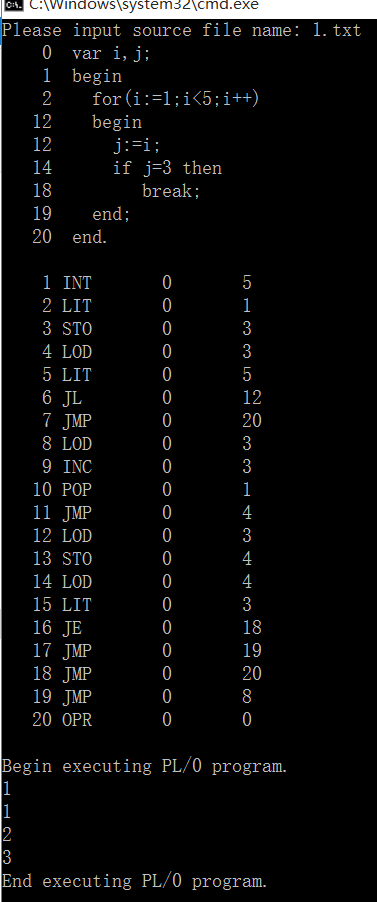
j:=i;

if j=3 then

break;

end;

end.



从结果上看，当j为3的时候跳出循环，运行结果正确。

1. continue语句

代码实现如下所示：

else if (sym == SYM\_CONTINUE)

{

if (!loopLevel)

error(28);

else

{

getsym();

gen(JMP, 0, loopStart);

}

}

其中loopStart保存了当前循环开始所在的代码行数，同时考虑循环嵌套，会用到下面的代码，

saveLoopStart = loopStart;

statement(fsys);

loopStart = saveLoopStart;

其做法就是保存当前的循环开始的行数，在处理statement语句之后再恢复该数。

程序示例如下：

var i,j, k,l;

begin

i:=0;

k:=0;

for( ;i<5; )

begin

i++;

if i=2 then

continue;

for ( ;k<2;k++)

begin

if k=1 then

continue;

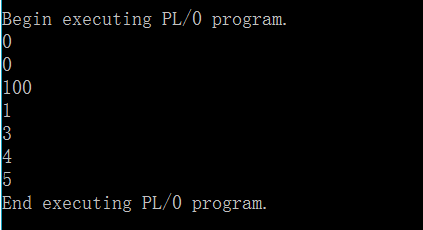
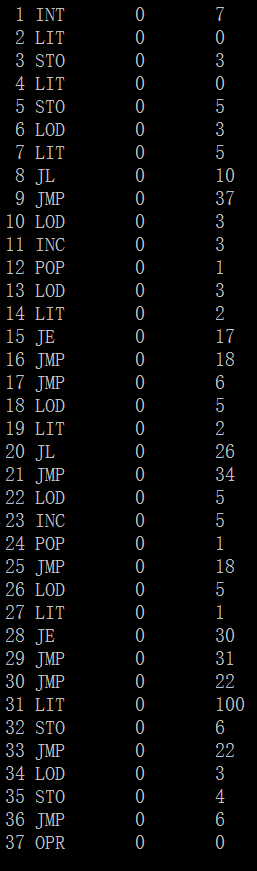
l:=100;

end;

j:=i;

end;

end.



内层continue会导致只输出一次100（在没有continue的情况下会输出两次），外层的continue导致i为2时没有对j执行赋值，所以没有输出2，所以运行结果正确。

（14）read函数

此read函数支持读入20个以内的变量，可以存入数组中,statement里加入如下语句



