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

1. 给 PL/0 添加内置函数 random 和 print.
2. 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）
3. 更多的 C 风格的运算表达式实现。语法/语义参照 C 语言。
4. 实现传地址调用。
5. 过程作为参数传递的实现。
6. 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。
7. do while 语句/switch 语句的实现。语法/语义参照 C 语言。
8. 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。

1. 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。
2. 添加数组。实现了多维数组的声明，赋值，运算。
3. 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。
4. 条件的短路计算。

**提高扩展：**

1. 添加内置函数print()，random()，callstack()。
2. 添加了goto语句，do-while语句，break语句，continue语句。
3. 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



（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  fposition | 查找标识符在名字表中的位置  查找函数标识符在名字表中的位置 |
| dposition  constdeclaration | 查找数组标识符在名字表中的位置  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

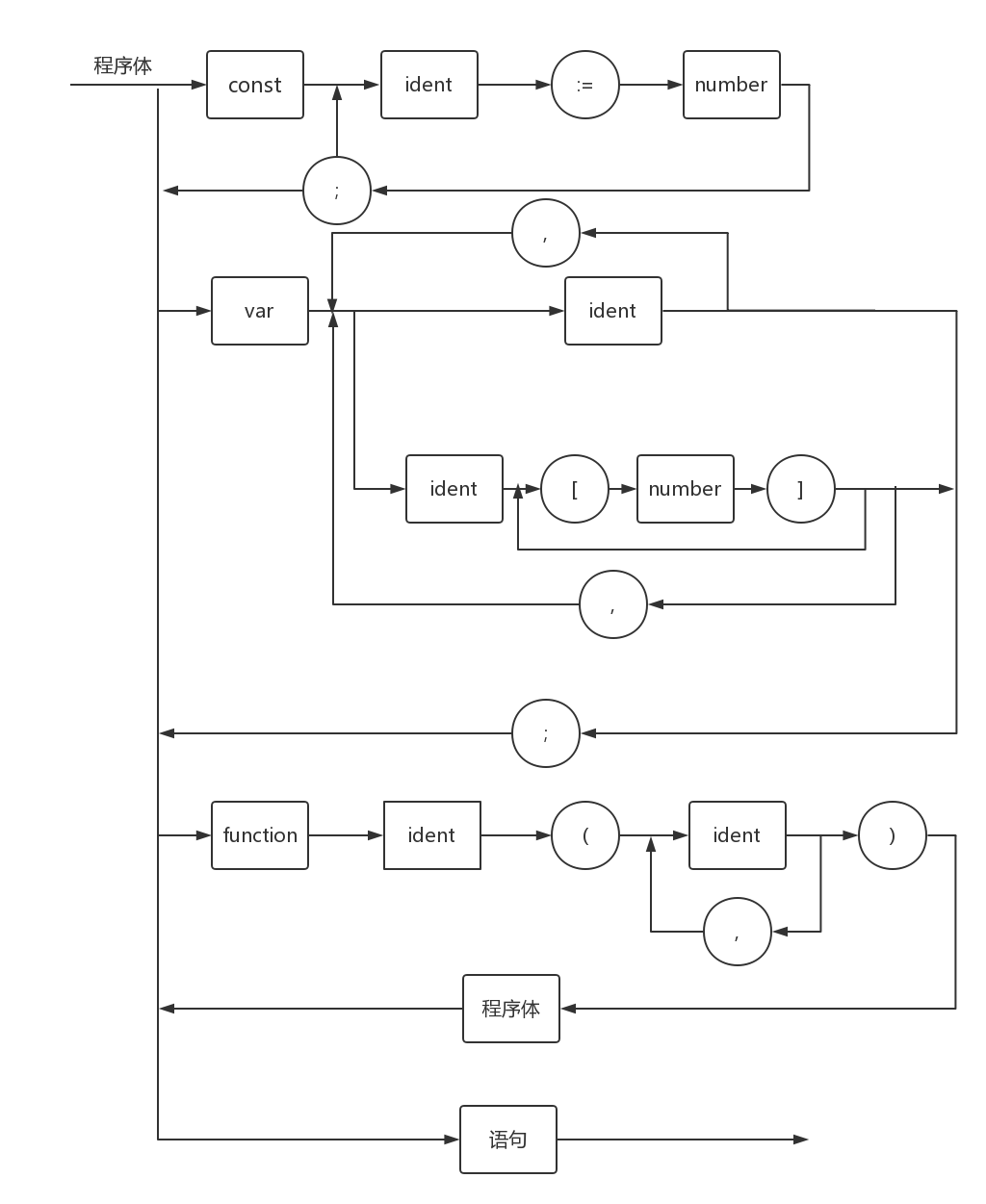
|ε

**(4)扩展后PL0语言的语法图**

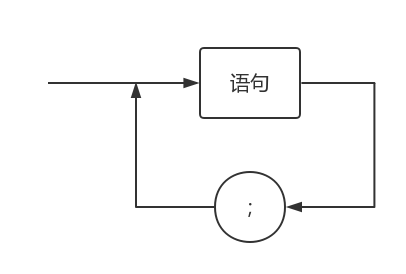
**程序：**

****

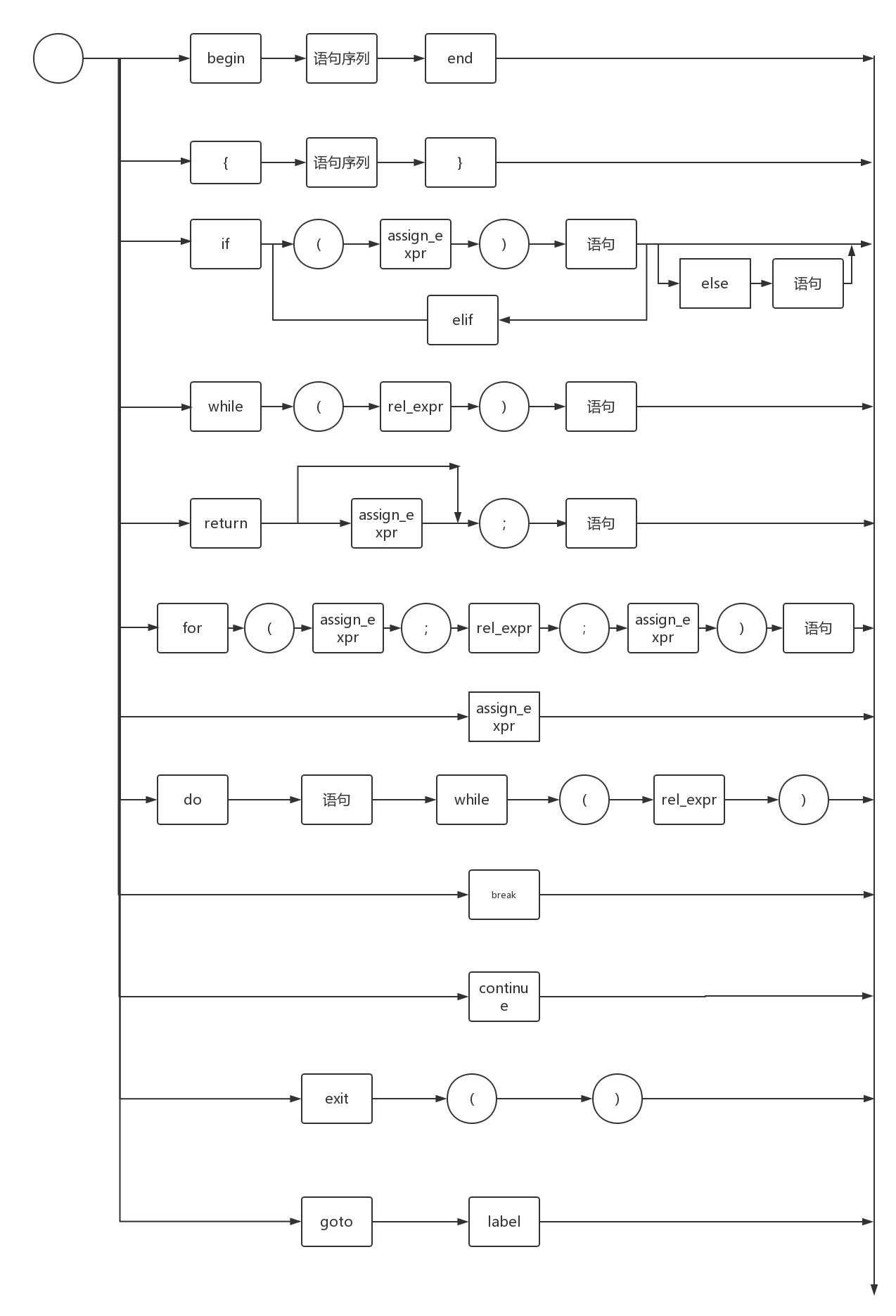
**程序体：**

****

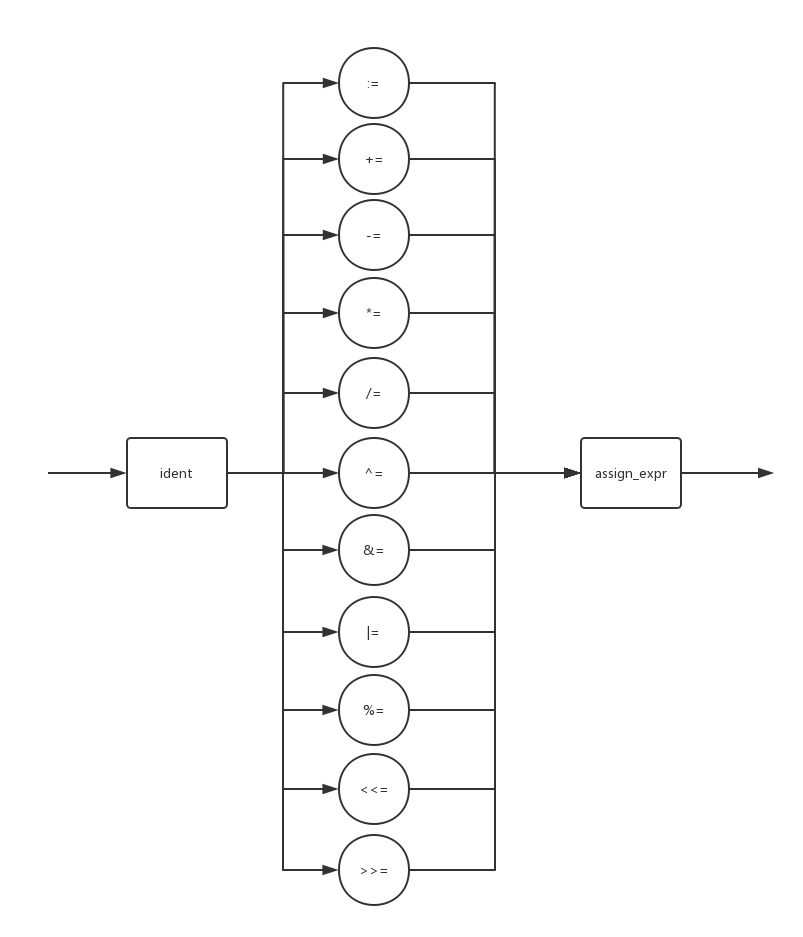
**语句序列：**



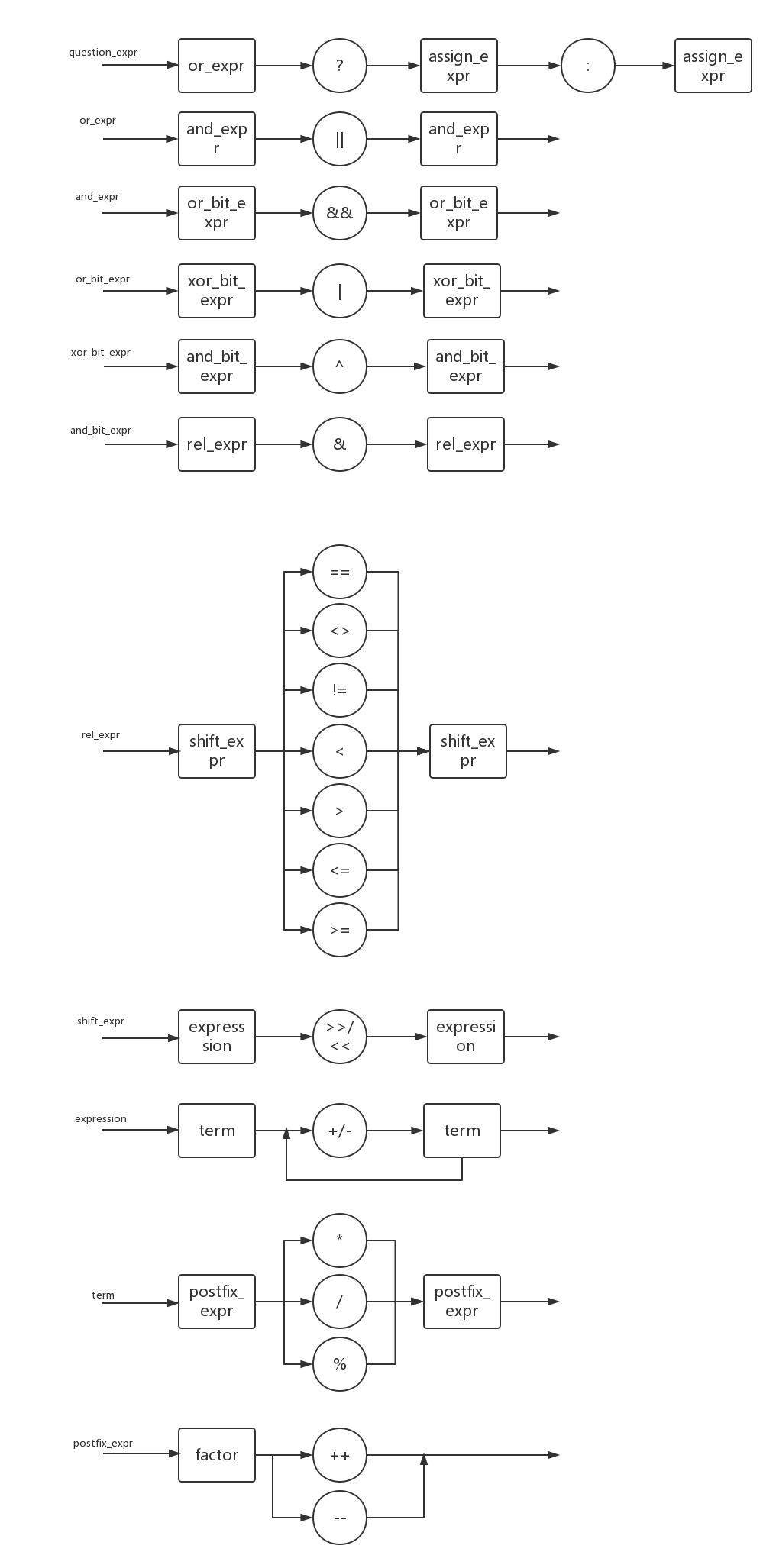
语句：



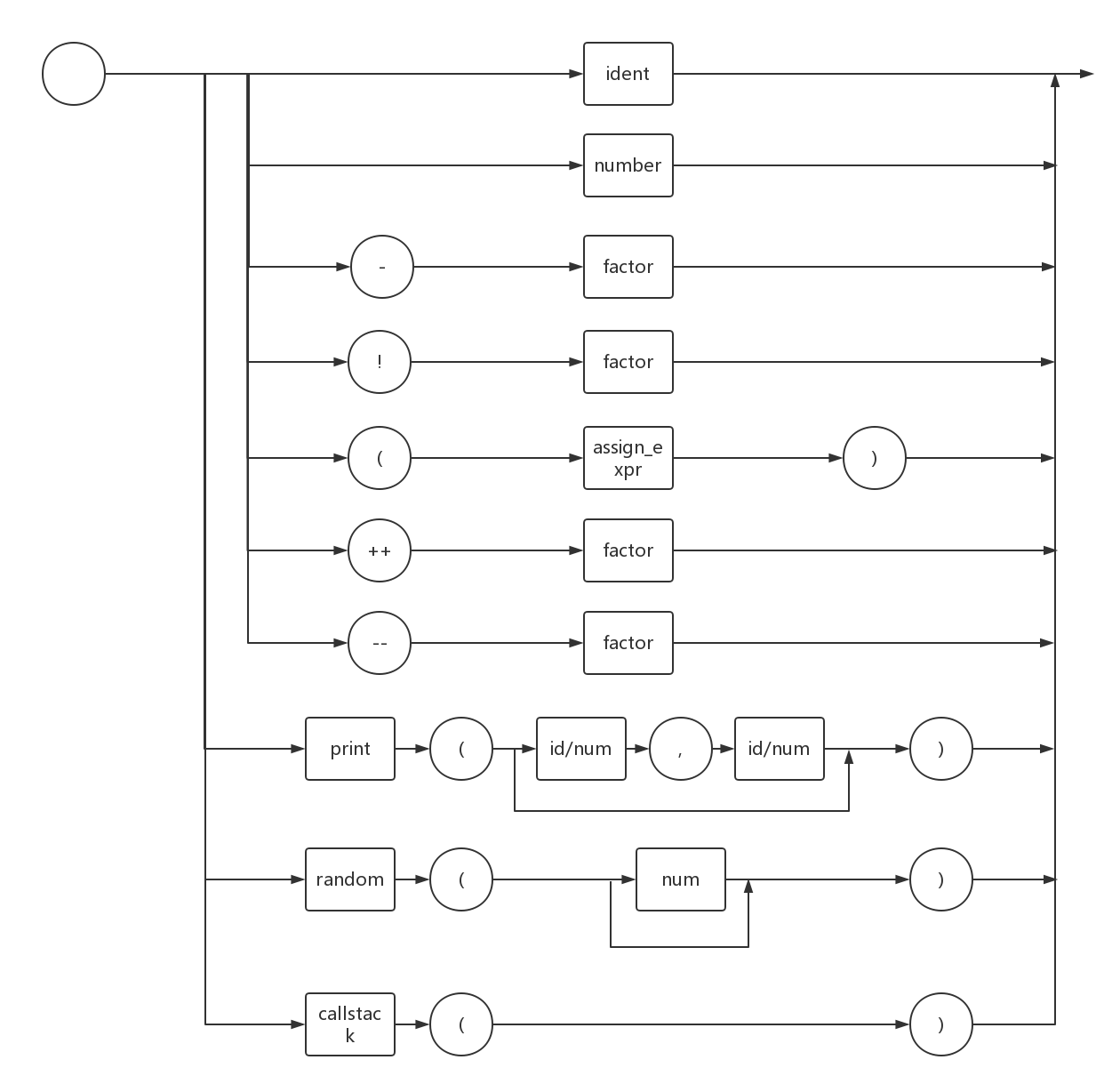
赋值表达式(assign\_expr):



其他表达式：

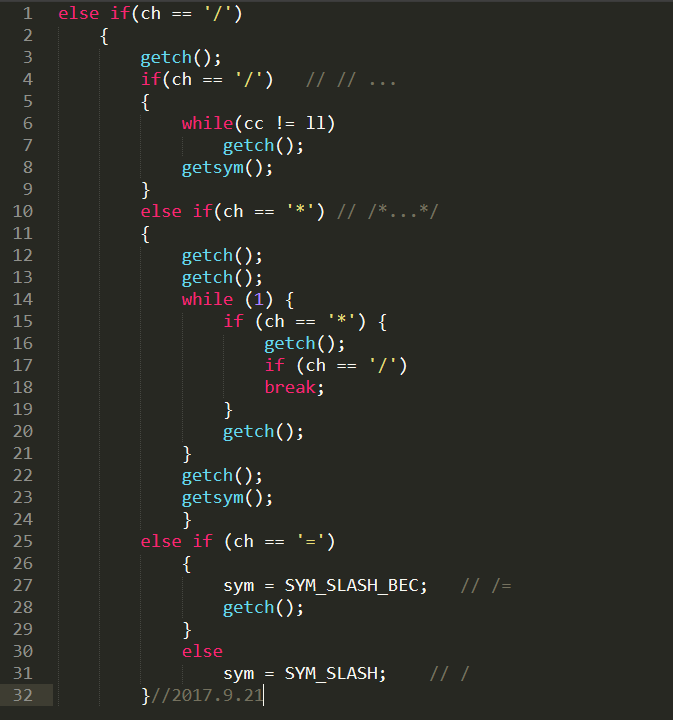


因子(factor):



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。

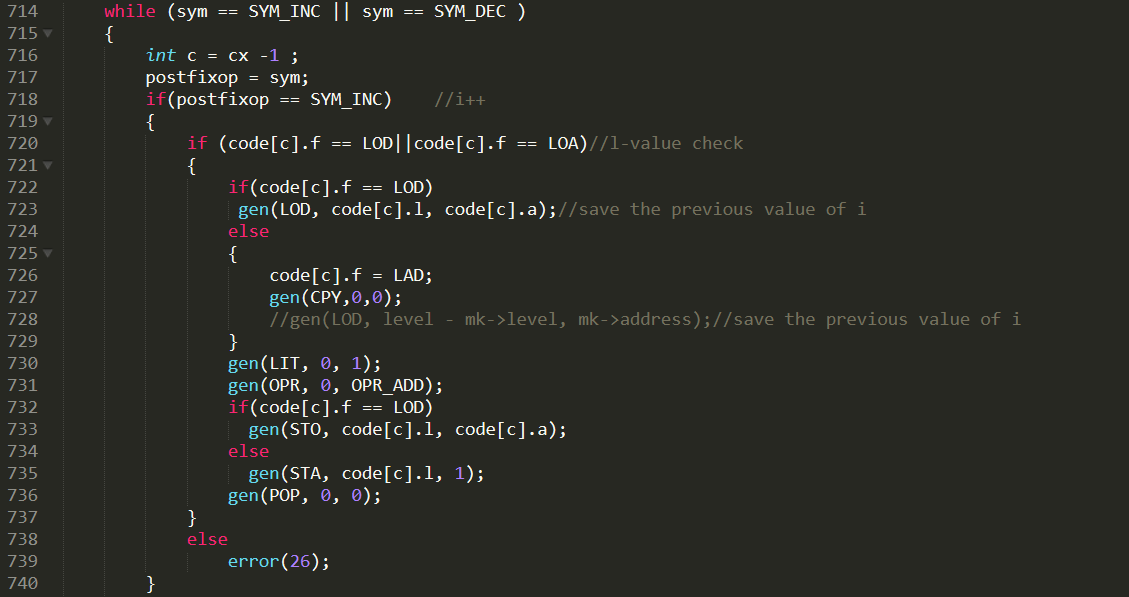


（2）添加c运算符，优先级与c语言相同。

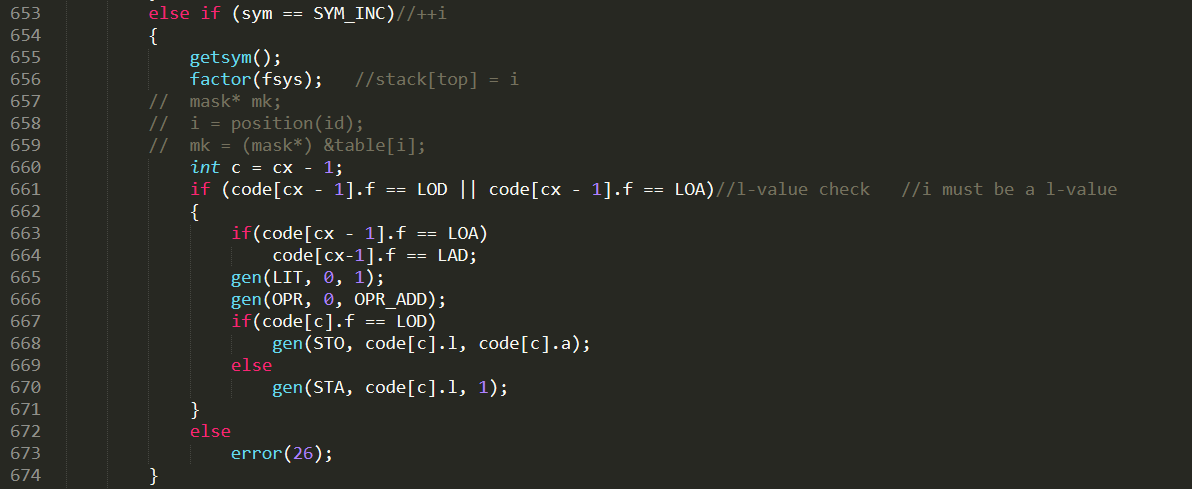
注意算符优先级和各算符对操作数的要求（例如左值检测）即可。

此处列举几个重要的实现：

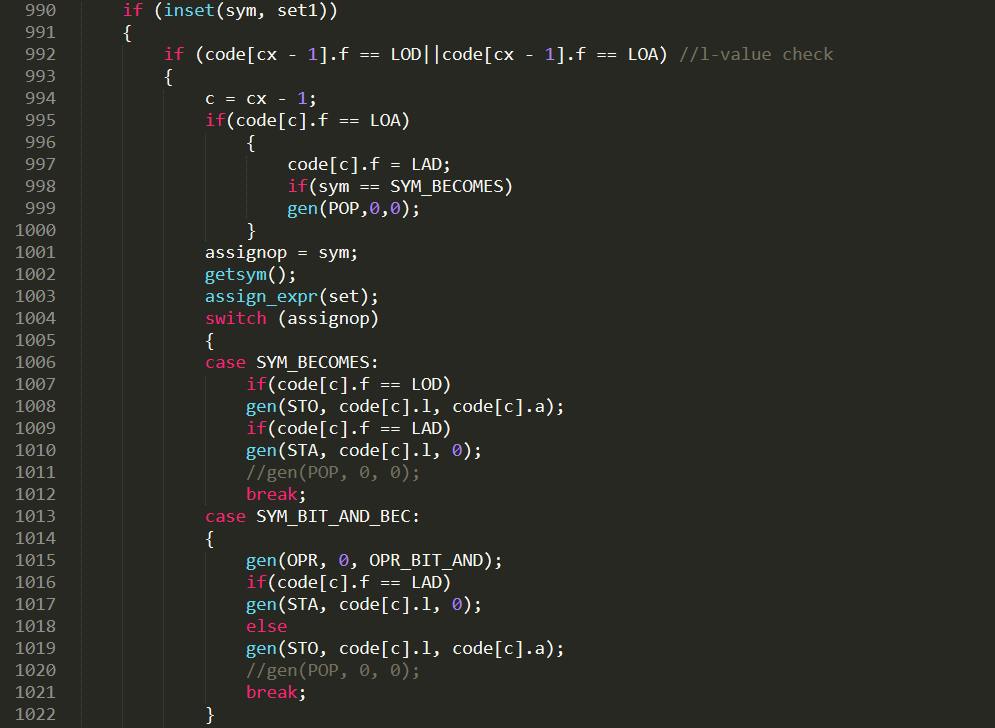
1.后++的实现：++指令要求操作数为左值，所以第720行进行左值检测。第723行保存运算前操作数的值，730行向栈顶赋1，731行执行++。CPY指令将stack[top]复制到stack[++top]，此处是为了保存数组累加操作前的值；LAD指令从栈顶得到偏移地址，并将取出的数据覆盖当前栈顶。至于STA，LOA指令，则类同于变量的STO，LOD指令，在数组实现里再作详细说明。



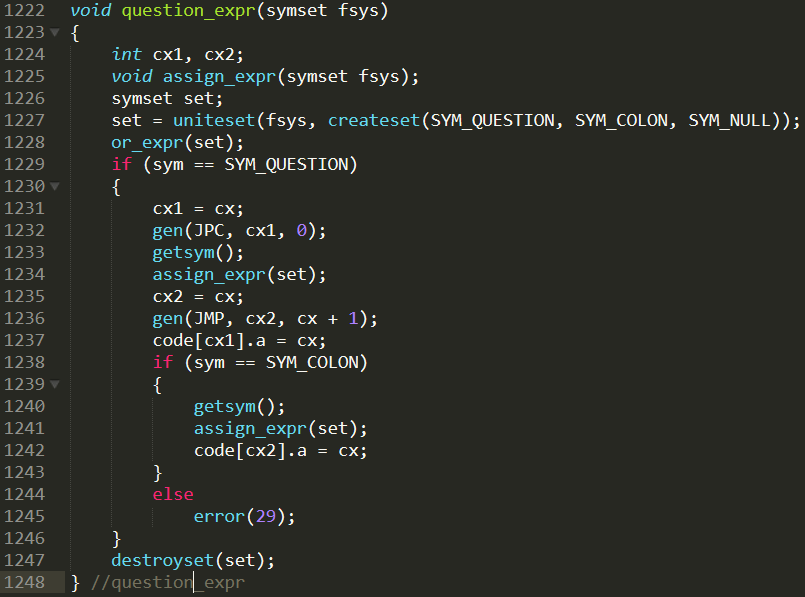
2.前++的实现：基本同后++，但不用保存运算前操作数的值。



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。



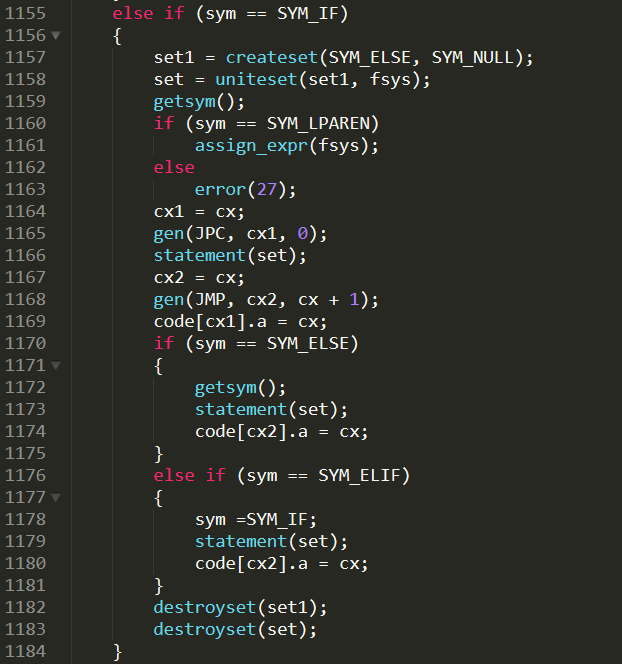
4.？表达式的实现：类似if-else语句采用回填技术，cx1保存？后第一个指令的地址，cx2保存：后第一指令的地址。然后分别回填到gen(JPC, cx1, 0)和gen(JMP, cx2, cx + 1)。



（3）添加/修改语句：

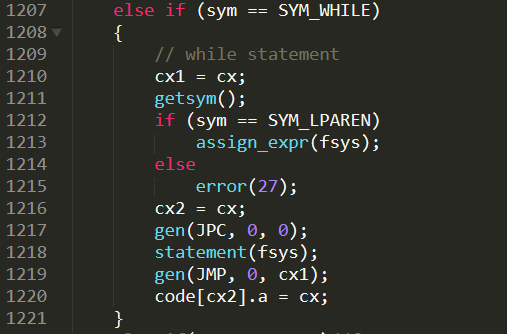
1.修改if语句：

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。



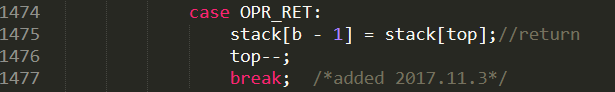
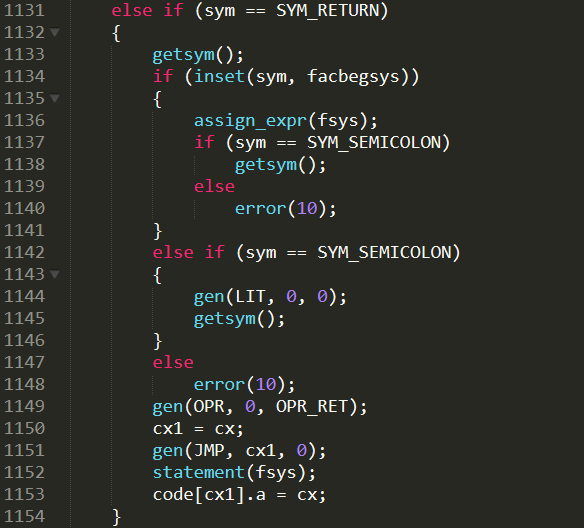
2.修改while语句：

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。



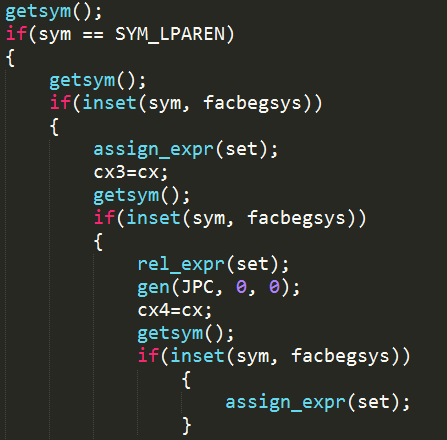
3.添加return语句;

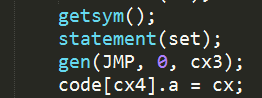
return后无表达式时，1144行返回0；否则返回stack[b-1]=stack[top]。b为被调用函数活动记录栈的基指针，则b-1则为设定的返回值所在位置。函数活动记录栈的详细情况在函数部分再作详细说明。



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。





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。



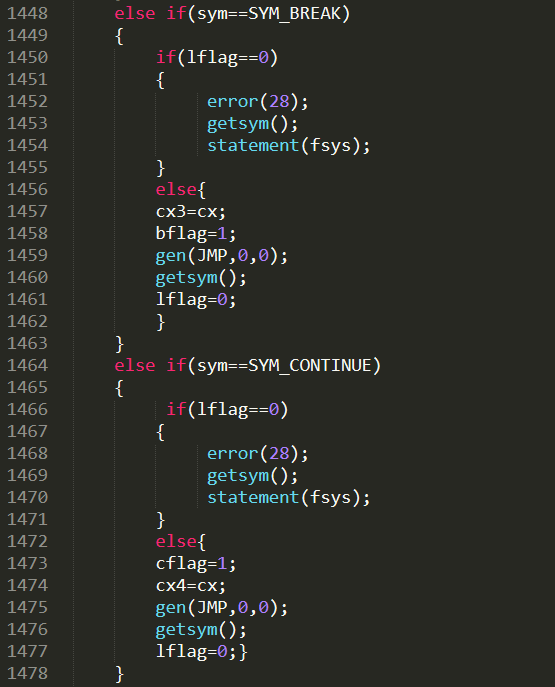
5．添加do-while语句：

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)。

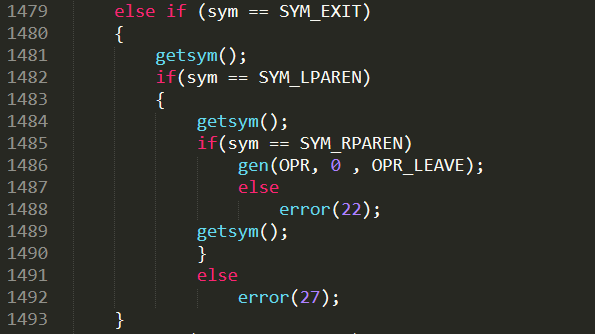


6.添加break/continue语句;

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。

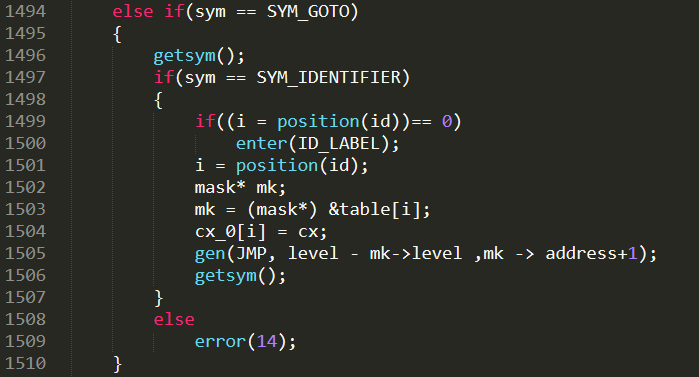


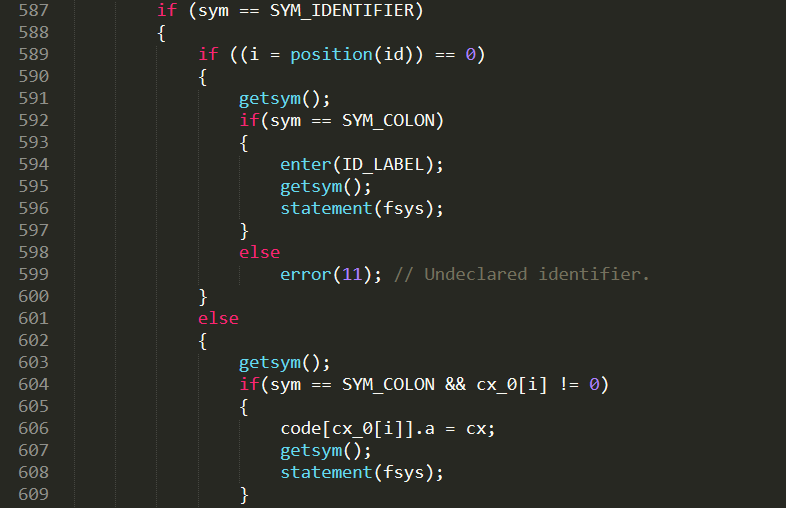
7.exit语句实现：直接生成leave指令即可。



8.goto语句的实现：

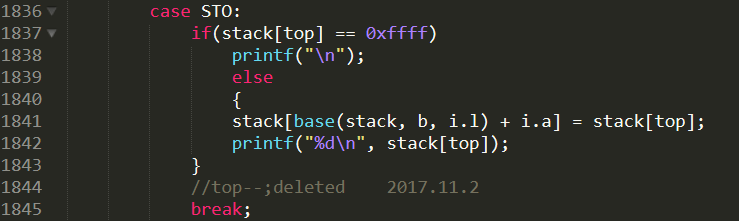
为了与ID\_VARIABLE区分，添加新的类型ID\_LABEL，该类型在名字表中具有name,kind和address属性。如果读到goto语句，查名字表，没有则登录名字表，用全局数组cx\_0[i]保存当前指令地址，然后生成jmp指令，其跳转地址未定，否则直接跳转到名字表中LABEL的地址之后一个地址；如果读到ID\_LABEL，查名字表，没有则登录名字表，否则将当前cx赋给code[cx\_0[i]]（回填）。

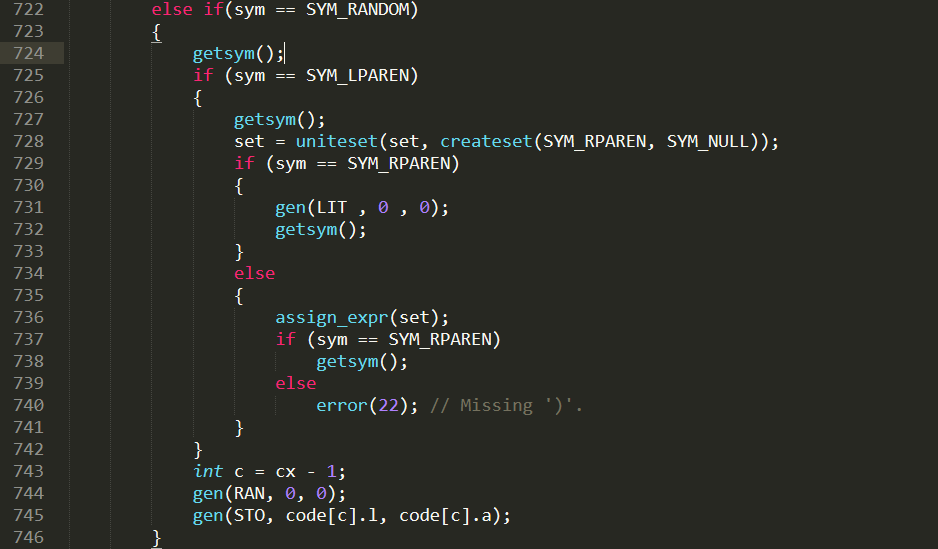




（4）添加print(),random(),callstack()内置函数

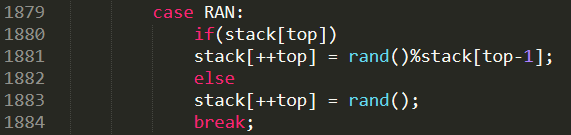
1.print()函数：为了尽量减少新指令的数量，考虑借用原有STO指令里的printf实现print()的功能。在表达式后直接在调用STO，这样不影响栈顶和表达式的值，并且输出了表达式的值。为了实现print()输出换行，观察到stack为int型，所以通过向栈顶存入0xffff来标志换行。

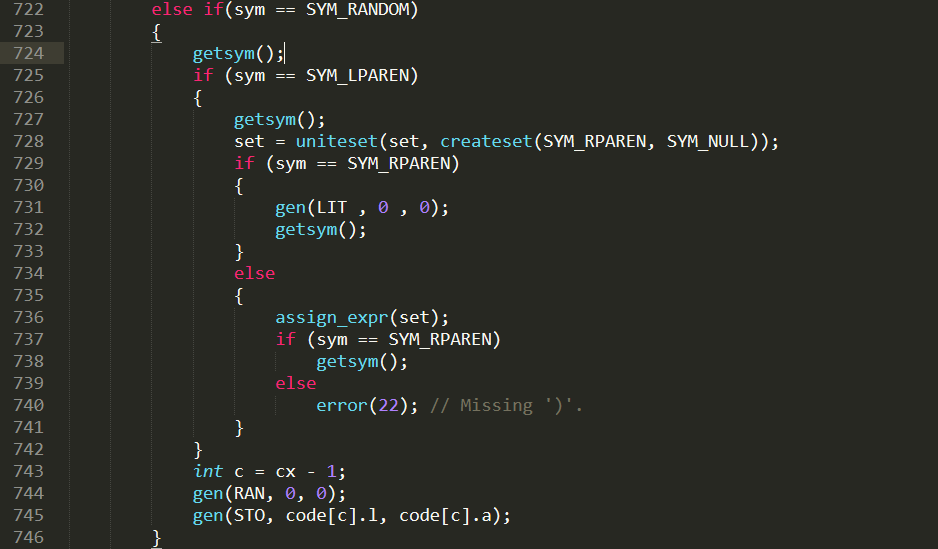




2.random()函数：

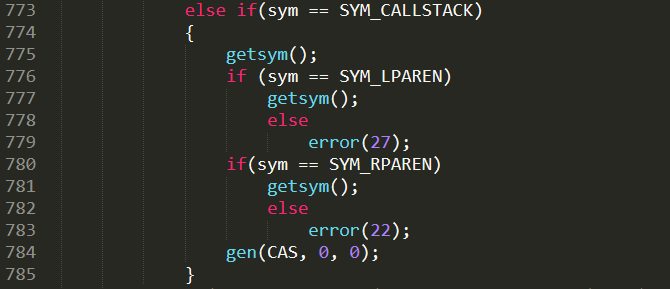
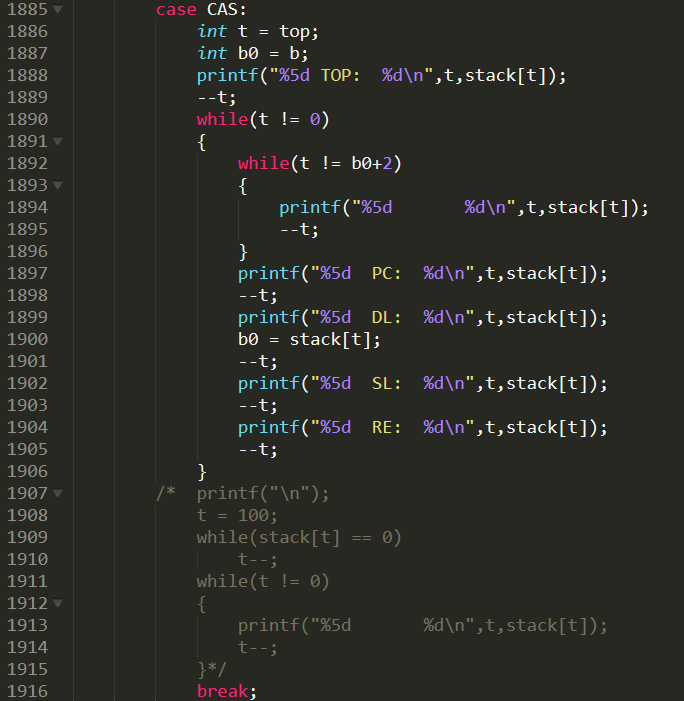
添加RAN指令，通过栈顶值调用rand()函数实现生成随机数。





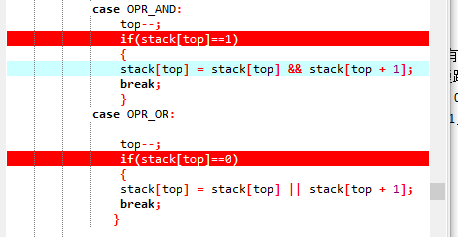
3.callstack()函数

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。



（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；



（6）实现数组：

1．数组声明的语法结构：

**ident ->id | dim**

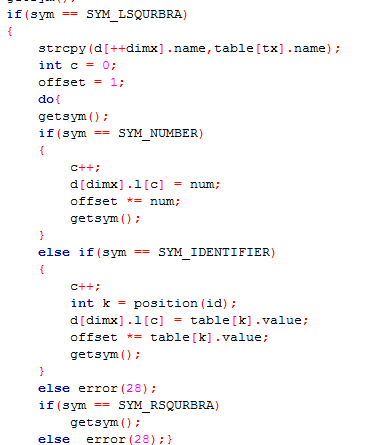
**dim -> idB[N]**

**B -> [N]B |ε**

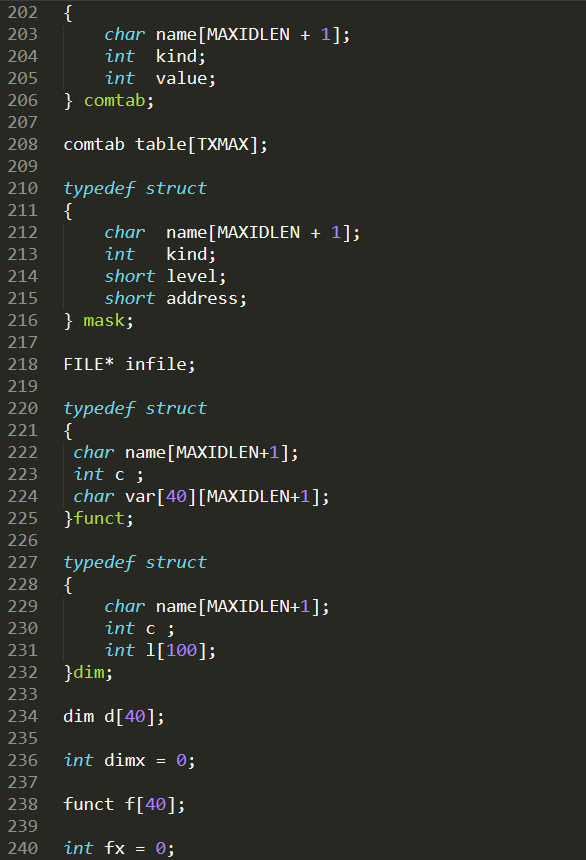
**N –> num**

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：

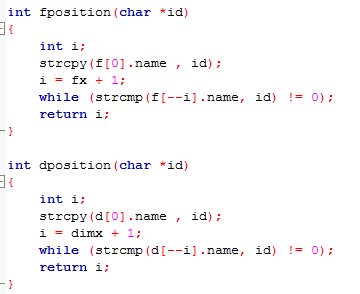


新建立的名字表：

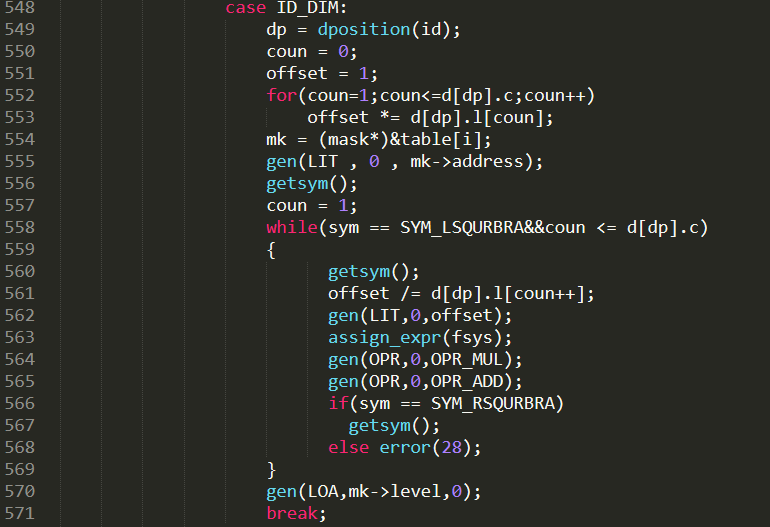


3．数组的寻址：在factor函数中，如果遇到数组变量，先将0放在栈顶，使全局变量offset等于所有维度长度之积，从“[assign\_expr]”中的表达式得到最高维的位置，放在栈顶，offset除以这个维度定义的长度，放在栈顶加入乘法指令，再加入加法指令。 对下一个维度执行同样操作，直到最后一维。这样数组元素的地址就会在操作结束后被放在栈顶。

查找数组名字表的函数：



factor中数组寻址的具体操作：

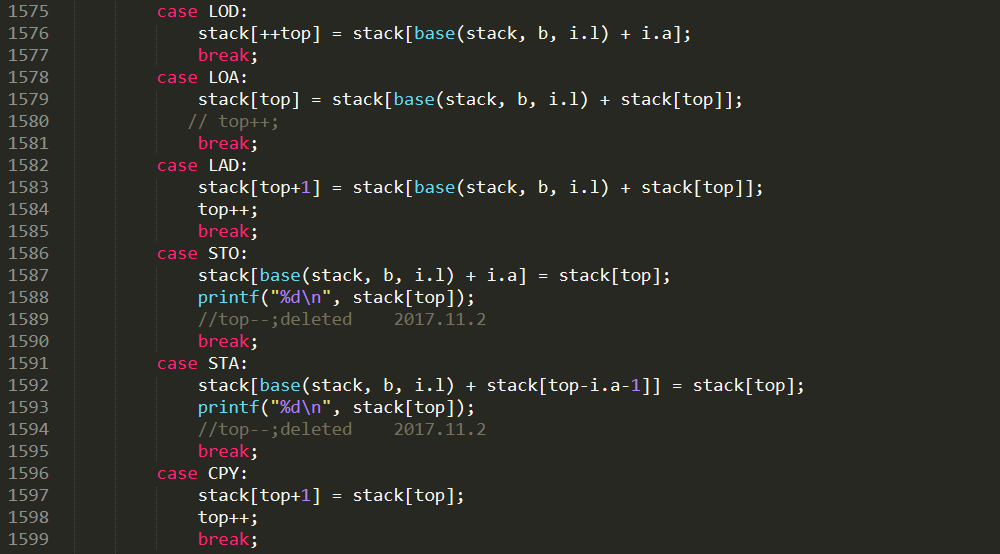


4．数组的存取：添加了三个指令用于数组的存取

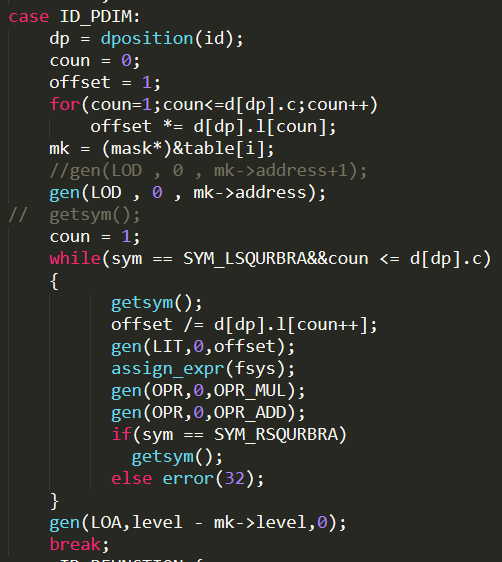
STA：从栈顶减1减i.a处得到偏移地址，其余同STO

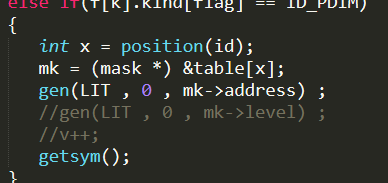
LOA: 从栈顶得到偏移地址，并将取出的数据覆盖当前栈顶，其余同LOD

LAD: 从栈顶得到偏移地址，其余同LOD。



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。 

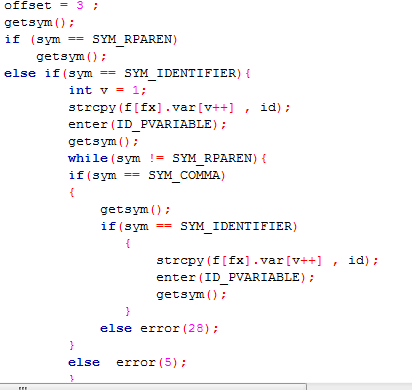


（7）实现函数参数传递：

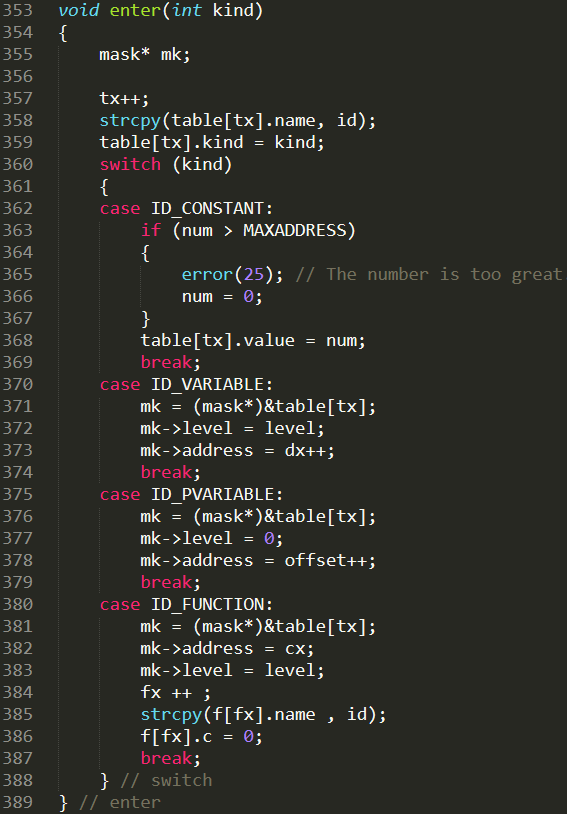
1．函数声明：

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。

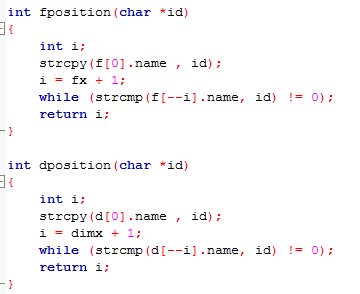
Block中函数声明部分：



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。



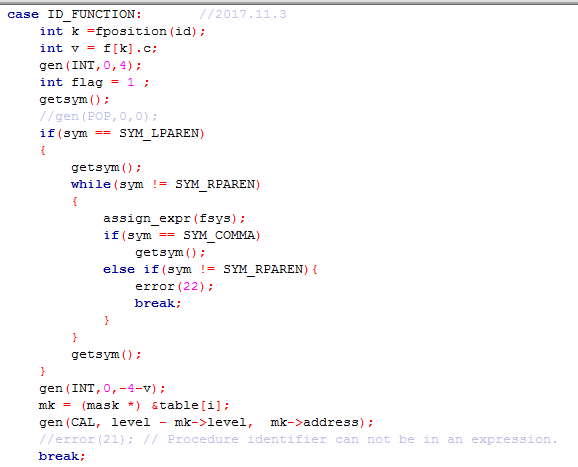
查找函数名字表的函数：



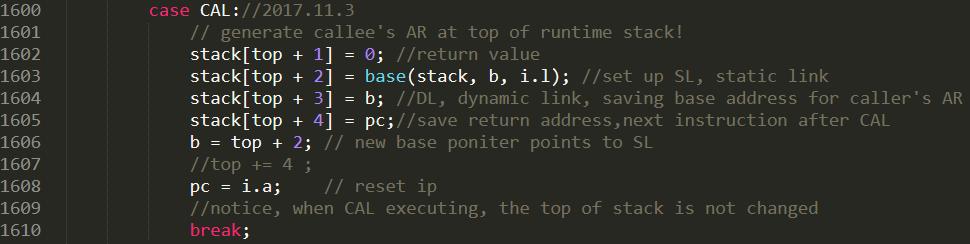
2．函数调用：

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：



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。

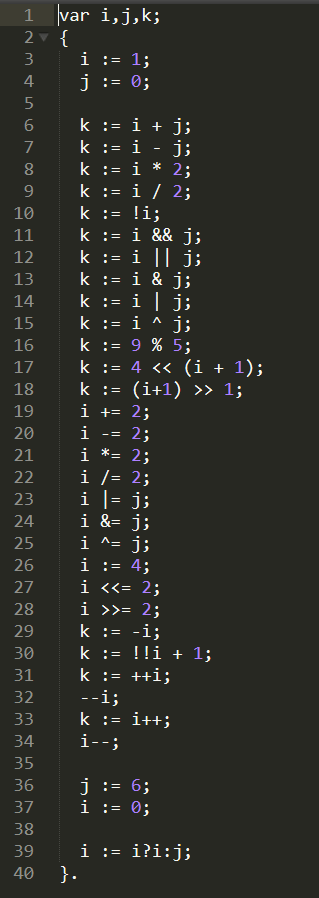


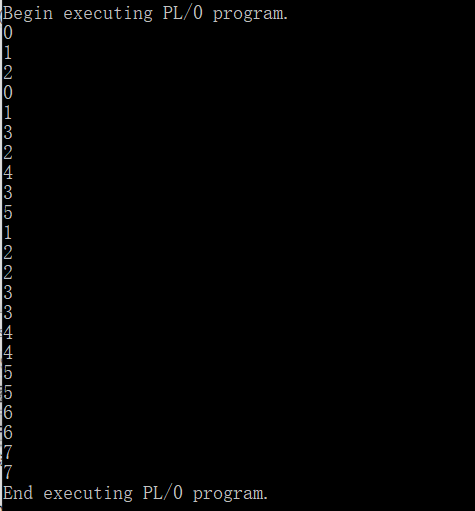
3．函数体结束后，从名次表中删除ID\_PVARIBLE，局部变量不再有效。

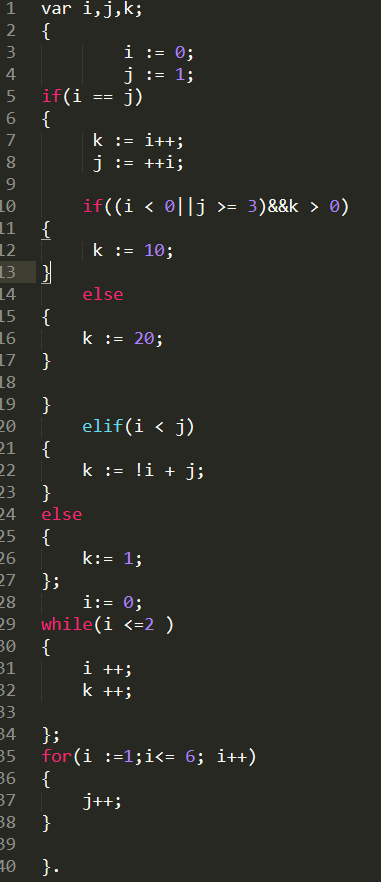
7）测试实现

1.测试各运算符：

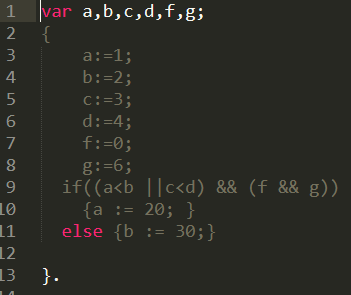
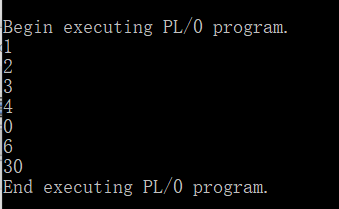
编译运行ex1文件。



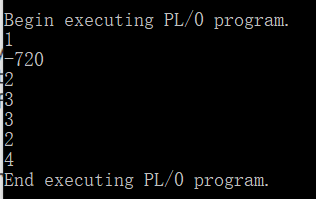
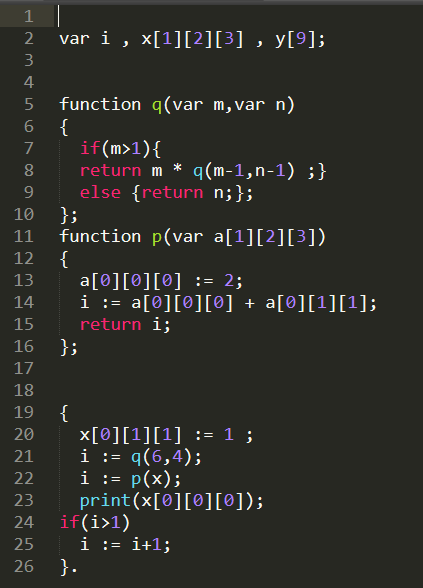
2.测试if-elif-else语句，while语句，for语句：

编译运行ex2文件。

3.测试短路计算：

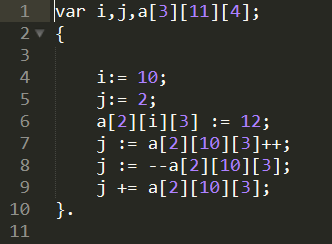
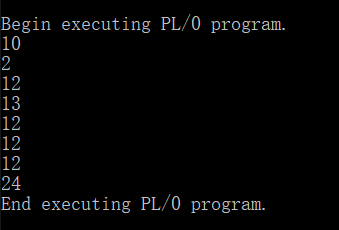
编译运行ex3文件。

4.测试函数传参，数组传参，print()函数：

编译运行ex4文件。

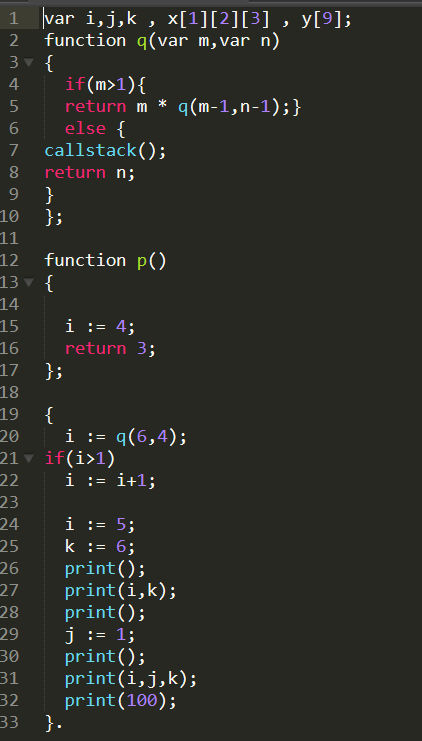
5.测试数组的声明，存取：

编译运行ex5文件。

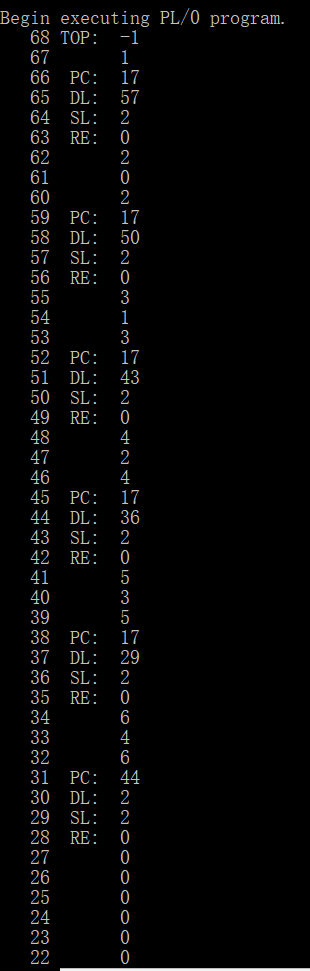
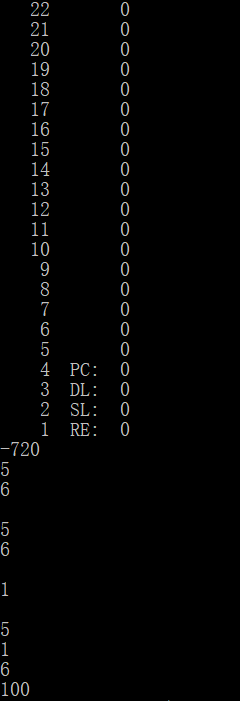


6.测试函数传参，callstack()函数，print()函数：

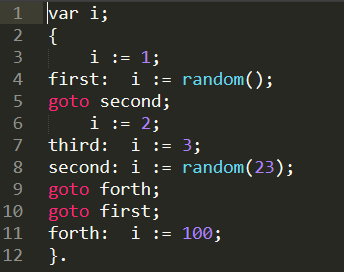
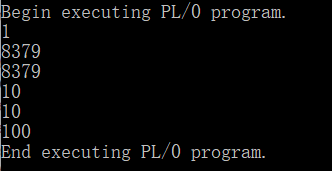
编译运行ex6文件。



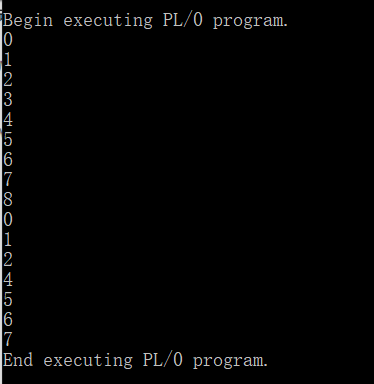
运行结果：



7.测试goto语句，random()函数：

编译运行ex7文件。

8.测试do-while语句，for语句，break/continue语句：

编译运行ex8文件。

