PL/0 实验报告

课程名称：编译原理

题目名称：PL/0 编译程序

学生学院：计算机科学与技术学院

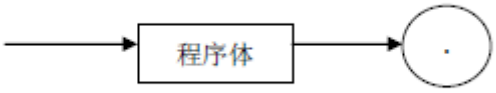
小组成员：朱河勤，张世聪，徐瑞，詹慧悠

实验项目	PL/0 编译程序的分析	指导教师	郑启龙
实验目的	1、熟悉 pl/0 语言并能编写小程序 2、掌握 pl/0 编译程序的编译过程(词法分析、语法分析、语义分析等)		
小组分工	组长：朱河勤（整体框架设计与实现包括词法分析,语法分析代码生成等） 组员：张世聪（do-while、switch 语句的实现） 詹慧悠（do-while、switch 语句的实现及实验报告撰写） 徐瑞（实现传值调用）		
实验仪器（编号） 材料、工具	Liinux, python		

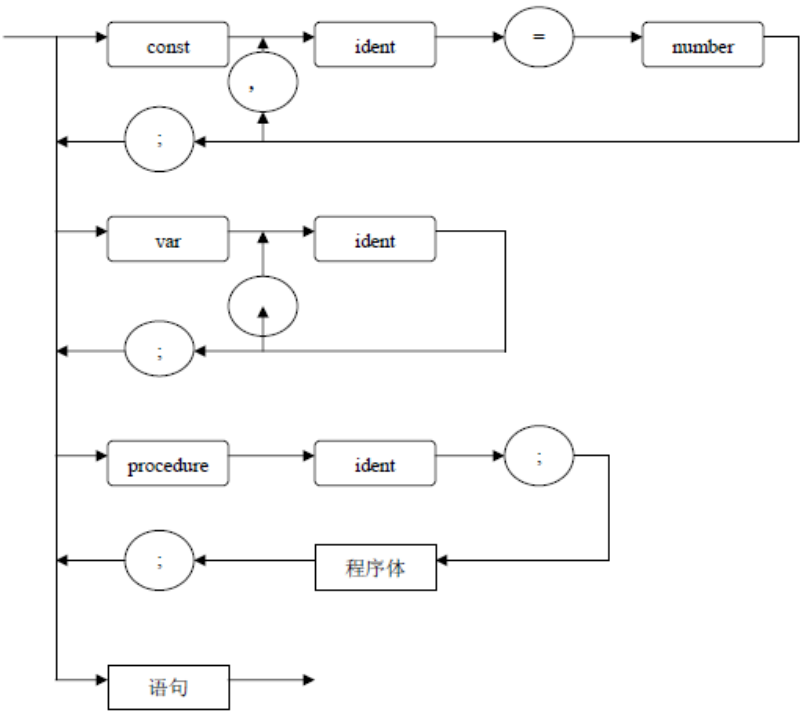
（原理概述）

pl/0 语言编译程序采用以语法分析为核心、一遍扫描的编译方法。词法分析和代码生成作为独立的子程序供语法分析程序调用。语法分析的同时，提供了出错报告和出错恢复的功能。在源程序没有错误编译通过的情况下，调用类 pcode 解释程序解释执行生成的类 pcode 代码。

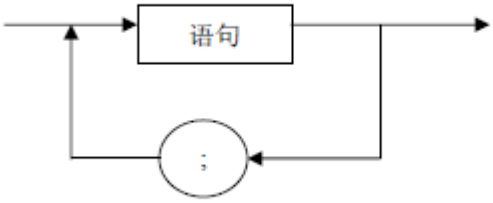
程序



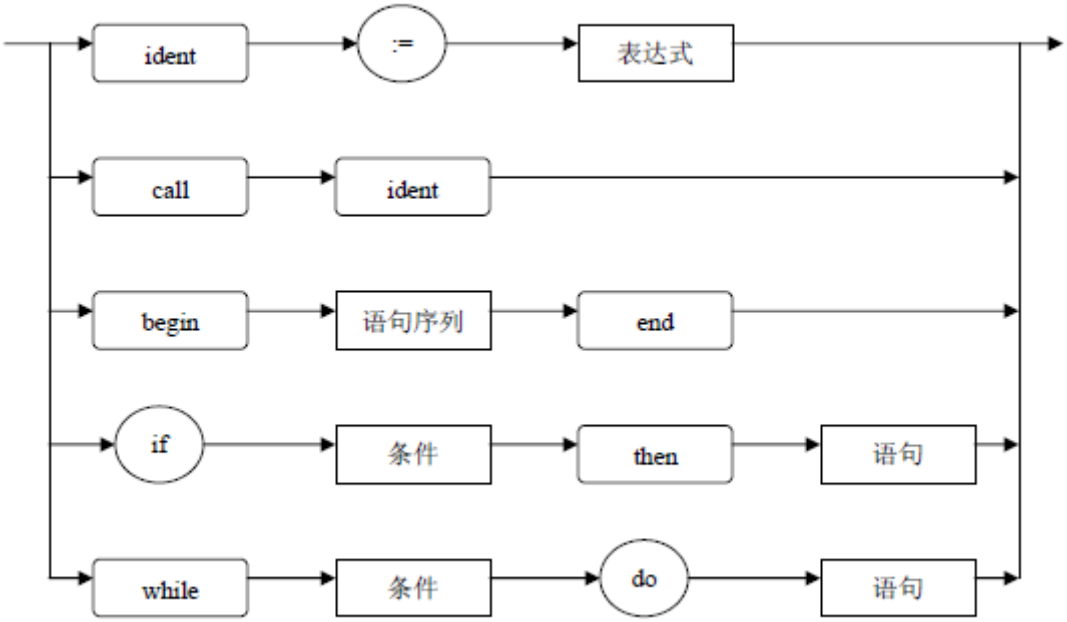
程序体



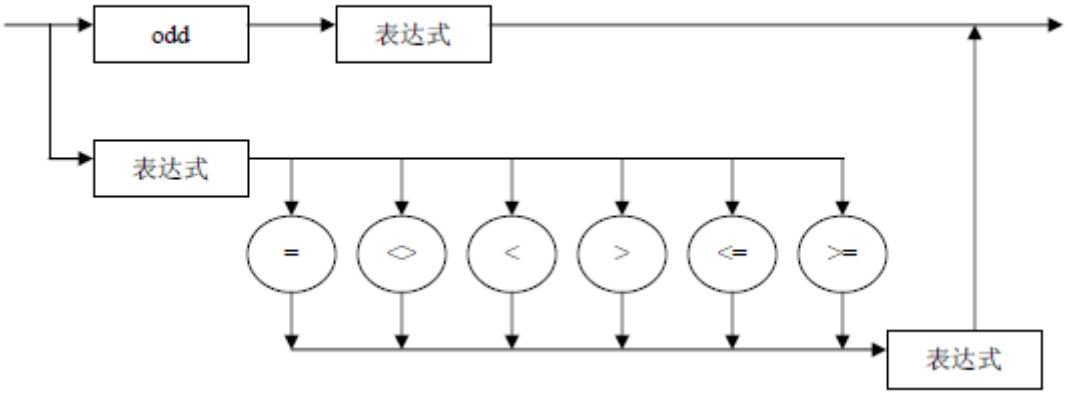
语句序列


语句
;

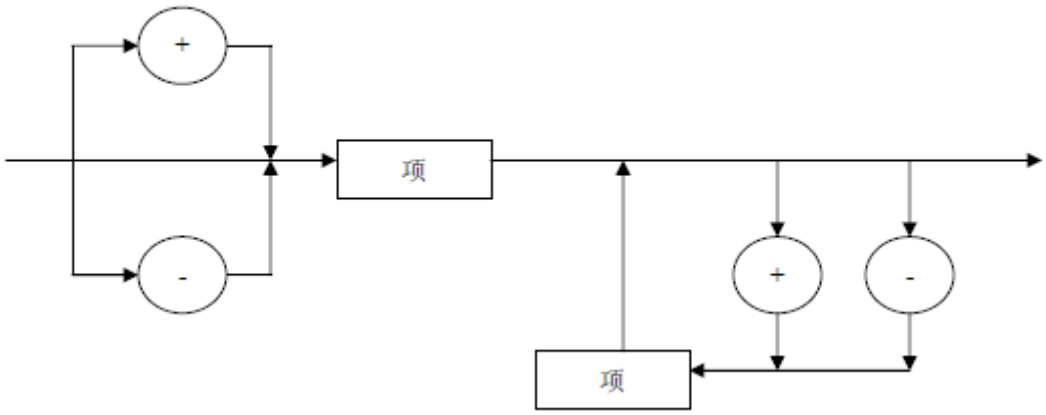
语句


ident
:=
表达式
call
ident
begin
语句序列
end
if
条件
then
语句
while
条件
do
语句

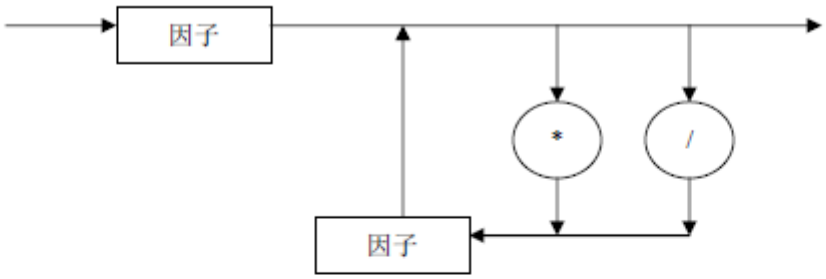
条件


odd
表达式
表达式
=
<>
<
>
<=
>=
表达式

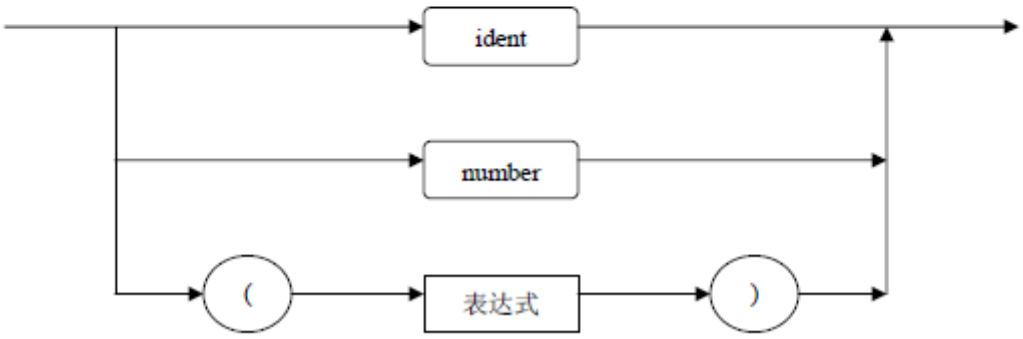
表达式



项



因子



PL/0 语言文法的 EBNF 表示：

EBNF 表示的符号说明。

〈 〉：用左右尖括号括起来的中文字表示语法构造成分，或称语法单位，为非终结符。

∷= ：该符号的左部由右部定义，可读作'定义为'。

| ：表示'或'，为左部可由多个右部定义。

{ } ：花括号表示其内的语法成分可以重复。在不加上下界时可重复 0 到任意次数，

有上下界时为可重复次数的限制。
如：{*}表示*重复任意次，$\{*\}^3_8$表示*重复 3-8 次。
[ ] ：方括号表示其内的成分为任选项。
( ) ：表示圆括号内的成分优先。

PL/0 语言文法的 EBNF 表示为：

〈程序〉::=〈分程序〉.
〈分程序〉::=[〈常量说明部分〉][〈变量说明部分〉][〈过程说明部分〉]〈语句〉
〈常量说明部分〉::=CONST〈常量定义〉 {，〈常量定义〉};
〈常量定义〉::=〈标识符〉=〈无符号整数〉
〈无符号整数〉::=〈数字〉{〈数字〉}
〈变量说明部分〉::=VAR〈标识符〉{，〈标识符〉};
〈标识符〉::=〈字母〉{〈字母〉|〈数字〉}
〈过程说明部分〉::=〈过程首部〉〈分程序〉{;〈过程说明部分〉};
〈过程首部〉::=PROCEDURE〈标识符〉;
〈语句〉::=〈赋值语句〉|〈条件语句〉|〈当型循环语句〉|
〈过程调用语句〉|〈读语句〉|〈写语句〉|〈复合语句〉|〈空〉
〈赋值语句〉::=〈标识符〉:=〈表达式〉
〈复合语句〉::=BEGIN〈语句〉{;〈语句〉}END
〈条件〉::=〈表达式〉〈关系运算符〉〈表达式〉|ODD〈表达式〉
〈表达式〉::=[+|-]〈项〉{〈加法运算符〉〈项〉}
〈项〉::=〈因子〉{〈乘法运算符〉〈因子〉}
〈因子〉::=〈标识符〉|〈无符号整数〉|'('〈表达式〉')'
〈加法运算符〉::=+|-
〈乘法运算符〉::=*|/
〈关系运算符〉::=#|=|<|<=|>|>=
〈条件语句〉::=IF〈条件〉THEN〈语句〉
〈过程调用语句〉::=CALL〈标识符〉
〈当型循环语句〉::=WHILE〈条件〉DO〈语句〉
〈读语句〉::=READ'('〈标识符〉{，〈标识符〉}')'
〈写语句〉::=WRITE'('〈表达式〉{，〈表达式〉}')'
〈字母〉::=a|b|…|X|Y|Z
〈数字〉::=0|1|2|…|8|9

（实验内容步骤）

(1) 根据 PL/0 语言的语法图，理解 PL/0 语言各级语法单位的结构，掌握 PL/0 语言合法程序的结构；
(2)从总体上分析整个系统的体系结构、各功能模块的功能、各模块之间的调用关系、各模块之间的接口；
(3)详细分析各子程序和函数的代码结构、程序流程、采用的主要算法及实现的功能；
(4)撰写分析报告，主要内容包括系统结构框图、模块接口、主要算法、各模块程序流

程图等

1. 分析PL/0源程序

词法分析

PL/0 的语言的词法分析器将要完成以下工作：

（1） 跳过分隔符（如空格，回车，制表符）；

（2） 识别诸如 begin，end，if，while 等保留字；

（3） 识别非保留字的一般标识符，此标识符值（字符序列）赋给全局量 id，而全局量 sym 赋值为 SYM_IDENTIFIER。

（4） 识别数字序列，当前值赋给全局量 NUM，sym 则置为 SYM_NUMBER；

（5） 识别:=，<=，>=之类的特殊符号，全局量 sym 则分别被赋值为 SYM_BECOMES，SYM_LEQ，SYM_GEQ 等。

相关过程（函数）有 getsym()，getch()，其中 getch()为获取单个字符的过程，除此之外，它还完成：

（1） 识别且跳过行结束符；

（2） 将输入源文件复写到输出文件；

（3） 产生一份程序列表，输出相应行号或指令计数器的值。

语法分析

我们采用递归下降的方法来设计 PL/0 编译器。以下我们给出该语言的 FIRST 和 FOLLOW 集合。

非终结符（S）	FIRST(S)	FOLLOW(S)
程序体	const var procedure ident call if begin while	. ;
语句	ident call begin if while	. ; end
条件	odd + - (ident number	then do
表达式	+ - (ident number	. ;) R end then do
项	ident number (	. ;) R + - end then do
因子	ident number (	. ;) R + - * / end then do

注：表中 R 代表六个关系运算符。

不难证明，PL/0 语言属于 LL（1）文法。（证明从略。）

以下是我们给出如何结合语法图编写（递归下降）语法分析程序的一般方法。假定图 S 所对应的程序段为 T（S），则：

（1） 用合适的替换将语法约化成尽可能少的单个图；

（2） 将每一个图按下面的规则（3）-（7）翻译成一个过程说明；

（3） 顺序图对应复合语句：

对应：begin T(S1); T(S2); ...; T(Sn) end

（4） 选择：

对应：case 语句或者条件语句：

```
case ch of          if ch in L1 then T(S1) else
 L1: T(S1);         if ch in L2 then T(S2) else
 L2: T(S2);  或      ...
...                 if ch in Ln then T(Sn) else
Ln: T(Sn);          error
```

其中 Li∈FIRST（Si），ch 为当前输入符号。（下同）

（5） 循环

对应：while ch in L do T(S)

（6） 表示另一个图 A 的图：

对应：过程调用 A。

（7） 表示终结符的单元图：

对应：if ch == x then read(ch) else error

相关过程有：
block(), constdeclaration(), vardeclaration(), statement(), condition(), expression_r(), term(), factor()等。

语义分析

PL/0 的语义分析主要进行以下检查：

（1） 是否存在标识符先引用未声明的情况；
（2） 是否存在已声明的标识符的错误引用；
（3） 是否存在一般标识符的多重声明

代码生成

PL/0 编译程序不仅完成通常的词法分析、语法分析，而且还产生中间代码和“目标”代码。最终我们要“运行”该目标码。为了使我们的编译程序保持适当简单的水平，不致陷入与本课程无关的实际机器的特有性质的考虑中去，我们假想有台适合 PL/0 程序运行的计算机，我们称之为 PL/0 处理机。PL/0 处理机顺序解释生成的目标代码，我们称之为解释程序。注意：这里的假设与我们的编译概念并不矛盾，在本课程中我们写的只是一个示范性的编译程序，它的后端无法完整地实现，因而只能在一个解释性的环境下予以模拟。从另一个角度上讲，把解释程序就看成是 PL/0 机硬件，把解释执行看成是 PL/0 的硬件执行，那么我们所做的工作：由 PL/0 源语言程序到 PL/0 机器指令的变换，就是一个完整的编译程序。

PL/0 处理机有两类存贮，目标代码放在一个固定的存贮数组 code 中，而所需数据组织成一个栈形式存放。

PL/0 处理机的指令集根据 PL/0 语言的要求而设计，它包括以下的指令：LIT 、LOD 、STO CAL、INT 、JMP、 JPC 、OPR

上述指令的格式由三部分组成：

F	L	A

其中，f, l, a 的含义见下表：

F	L	a
INT	————	常 量
LIT	————	常 量
LOD	层次差	数据地址
STO	层次差	数据地址
CAL	层次差	程序地址
JMP	————	程序地址

JPC	————	程序地址
OPR	————	运算类别

表 2-1 PL/0 处理机指令

上表中，层次差为变量名或过程名引用和声明之间的静态层次差别，程序地址为目标数组 code 的下标，数据地址为变量在局部存贮中的相对地址。

PL/0 的编译程序为每一条 PL/0 源程序的可执行语句生成后缀式目标代码。这种代码生成方式对于表达式、赋值语句、过程调用等的翻译较简单。

如赋值语句 X := Y op Z（op 为某个运算符），将被翻译成下面的目标代码序列：（设指令计数从第 100 号开始）

No.	f	L	a
100	LOD	Level_diff_Y	Addr_Y
101	LOD	Level_diff_Z	Addr_Z
102	OPR	————————	op
103	STO	Level_diff_X	Addr_X

而对 if 和 while 语句稍繁琐一点，因为此时要生成一些跳转指令，而跳转的目标地址大都是未知的。为解决这一问题，我们在 PL/0 编译程序中采用了回填技术，即产生跳转目标地址不明确的指令时，先保留这些指令的地址（code 数组的下标），等到目标地址明确后再回过来将该跳转指令的目标地址补上，使其成为完整的指令。下表是 if、while 语句目标代码生成的模式。（L1，L2 是代码地址）

if C then S	While C do S
条件 C 的目标代码 JPC -- L1 语句 S 的目标代码 L1: ...	L1: 条件 C 的目标代码 JPC – L2 语句 S 的目标代码 JMP L1 L2: ...

表 2-2 if-while 语句目标代码生成模式

相关过程（函数）有：gen()，其任务是把三个参数 f、l、a 组装成一条目标指令并存放于 code 数组中，增加 CX 的值，CX 表示下一条即将生成的目标指令的地址。

代码执行

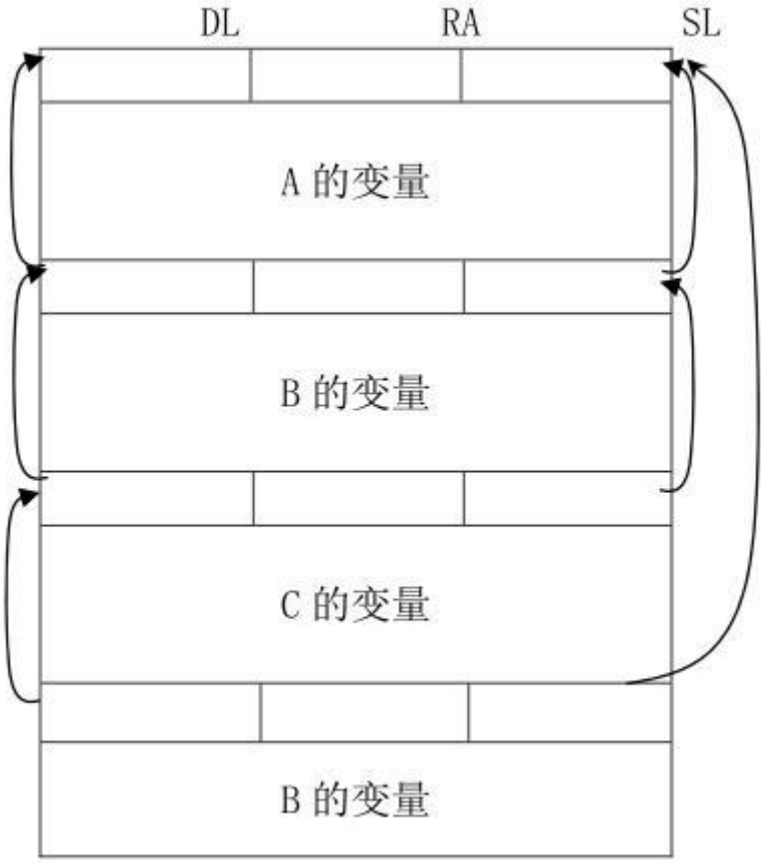
为了简单起见，我们假设有一个 PL/0 处理机，它能够解释执行 PL/0 编译程序所生成的目标代码。这个 PL/0 处理机有两类存贮、一个指令寄存器和三个地址寄存器组成。程序（目标代码）存贮称为 code，由编译程序装入，在目标代码执行过程中保持不变，因此它可被看成是“只读”存贮器。数据存贮 S 组织成为一个栈，所有的算术运算均对栈顶元和次栈顶元进行（一元运算仅作用于栈顶元），并用结果值代替原来的运算对象。栈顶元的地址（下标）记在栈顶寄存器 T 中，指令寄存器 I 包含着当前正在解释执行的指令，程序地址寄存器 P 指向下一条将取出的指令。

PL/0 的每一个过程可能包含着局部变量，因为这些过程可以被递归地调用，故在实际调用前，无法为这些局部变量分配存贮地址。各个过程的数据区在存贮栈 S 内顺序叠起来，每个过程，除用户定义的变量外，还应当有它自己的内部信息，即调用它的程序段地址（返回地址）和它的调用者的数据区地址。在过程终止后，为了恢复原来程序的执行，这两个地址都是必须的。我们可将这两个内部值作为位于该过程数据区的内部式隐式局部变量。我们把它们分别称为返回地址（return address）RA 和动态链（dynamic link）DL。动态链的头，即最新分配的数据区的地址，保存在某地址寄存器 B 内。

因为实际的存贮分配是运行（解释）时进行的，编译程序不能为其生成的代码提供绝对地址，它只能确定变量在数据区内的位置，因此它只能提供相对地址。为了正确地存取数据，解释程序需将某个修正量加到相应的数据区的基地址上去。若变量是局部于当前正在解释的过程，则此基

地址由寄存器 B 给出，否则，就需要顺着数据区的链逐层上去找。然而遗憾的是，编译程序只能知道存取路线的表的长度，同时动态链保存的则是过程活动的动态历史，而这两条存取路线并不总是一样。

例如，假定有过程 A，B，C，其中过程 C 的说明局部于过程 B，而过程 B 的说明局部于过程 A，程序运行时，过程 A 调用过程 B，过程 B 则调用过程 C，过程 C 又调用过程 B。 从静态的角度我们可以说 A 是在第一层说明的，B 是在第二层说明的，C 则是在第三层说明的。若在 B 中存取 A 中说明的变量 a，由于编译程序只知道 A，B 间的静态层差为 1，如果这时沿着动态链下降一步，将导致对 C 的局部变量的操作。为防止这种情况发生，有必要设置第二条链，它以编译程序能明了的方式将各个数据区连接起来。我们称之为静态链（static link）SL。这样，编译程序所生成的代码地址是一对数，指示着静态层差和数据区的相对修正量。下面我们给出的是过程 A、B 和 C 运行时刻的数据区图示：



有了以上认识，我们就不难明白 PL/0 源程序的目标代码是如何被解释执行的。以语句 X：= Y op Z 为例，（该语句的目标代码序列我们已在 2.4 节给出），PL/0 处理机解释该指令的“步骤”如下：

```
step 1,
  S[++T] ← S[base(level_diff_Y) + addr_Y];
   // 将变量 Y 的值放在栈顶
step 2,
  S[++T] ← S[base(level_diff_Z) + addr_Z];
   // 将变量 Z 的值放在栈顶，此栈顶元为变量 Y 的值
step 3,
  T--;
   // 栈顶指针指向次栈顶元，即存放结果的单元
step 4,
```

S[T] ← S[T] op S[T + 1];
// 变量 Y 和变量 Z 之间进行“op”操作
step 5,
S[base(level_diff_X) + addr_X] ← S[T];
// 将栈顶的值存放到变量 X 所在的单元
step 6,
T--;
// 栈顶指针减一

相关过程：base()，interpret()。其中 base()的功能是根据层次差并从当前数据区沿着静态链查找，以便获取变量实际所在的数据区其地址；interpret()则完成各种指令的执行工作。

错误诊断处理

一个编译程序，在多数情况下，所接受的源程序正文都是有错误的。发现错误，并给出合适的诊断信息且继续编译下去从而发现更多的错误，对于编译程序而言是完全必要的。一个好的编译器，其特征在于：

任何输入序列都不会引起编译程序的崩溃。

一切按语言定义为非法的结构，都能被发现和标志出来。

经常出现的错误，程序员的粗心或误解造成的错误能被正确地诊断出来，而不致引起进一步的株连错误。

符号表管理

为了组成一条指令，编译程序必须知道其操作码及其参数（数或地址）。这些值是由编译程序本身联系到相应标识符上去的。这种联系是在处理常数、变量和过程说明完成的。为此，标识符表应包含每一标识符所联系的属性；如果标识符被说明为常数，其属性值为常数值；如果标识符被说明成变量，其属性就是由层次和修正量（偏移量）组成的地址；如果标识符被说明为过程，其属性就是过程的入口地址及层次。

常数的值由程序正文提供，编译的任务就是确定存放该值的地址。我们选择顺序分配变量和代码的方法；每遇到一个变量说明，就将数据单元的下标加一（PL/0 机中，每个变量占一个存贮单元）。开始编译一个过程时，要对数据单元的下标 dx 赋初值，表示新开辟一个数据区。dx 的初值为 3，因为每个数据区包含三个内部变量 RA，DL 和 SL。

相关过程：enter()，该函数用于向符号表添加新的符号，并确定标识符的有关属性。

说明

（1） 每一个分程序(过程)被编译结束后,将列出该部分 PL/0 程序代码。这个工作由过程 listcode()完成。注意，每个分程序（过程）的第一条指令未被列出。该指令是跳转指令。其作用是绕过该分程序的说明部分所产生的代码（含过程说明所产生的代码）。

（2） 解释程序作为 PL/0 编译程序的一个过程，若被编译的源代码没有错误，则编译结束时调用这个过程。

PL/0 编译程序实现的功能：

```
# Description
## ident type
* constant
* varible
* function
## operator
### relation opr
* \<
* \>
* \<=
* \>=
* = equal
* !=
* odd
### bit opr
* \& bitand
* \| bitor
* \~ bitnot
* \<\< left shift
* \>\> right shift
### arithmetic opr
* \+  add/plus
* \-  sub/minus
* \*  multiply
* \/  divide
* \/\% integer div
* %  mod
* \^  power
* \!  factorial
### conditon opr
*  ?:  eg   a\>b ? c:d
## control structure
* if elif else
* for
* while-do
* do-while
* swtich
* break
* continue
* return

## builtin function
* print(a,b,c...)
* random(), random(n)
```

PL/0 编译程序语法的具体实现为：

```
# Grammar
program =  body "."
body = {varDeclaration ";" |  constDeclaration ";" |  "func" ident "(" arg_list
")" body ";"}  sentence

varDeclaration = "var"  varIdent { "," varIdent}
varIdent  = ident ["=" number] | ident  { "[" number "]" }
constDeclaration = "const" ident "=" number {"," ident "=" number}

sentence = [ ident ":=" { ident ":=" } sentenceValue
               |  "begin" sentence { ";" sentence}  "end"
               |  "if" sentenceValue "then" sentence {"elif" sentence} ["else"
sentence]
               |  "while" sentenceValue "do" sentence
               |  "do" sentence "while" sentenceValue
               |  "switch" sentenceValue {"case" sentenceValue {","
sentenceValue} ":" [setenceValue]}  (* ["default" ":" sentenceValue]   to do *)
               |  "break"
               |  "continue"
               |  ["return"] sentenceValue
               |  "print" "(" str,real_arg_list ")" ]

sentenceValue =   condition

arg_list =  ident { "," ident}

real_arg_list = sentenceValue {"," sentenceValue }


condition = condition_or [ "?" sentenceValue ":" sentenceValue ]
condition_or  = condition_and { "||" condition_or }
condition_and = condition_not { condition_not "&&" condition_and}
condition_not = {"!"} condition_unit
condiiton_unit = ["odd"] expression
                     | expression ("<" | ">" | "<=" | ">=" | "=" | "!=") expression

expression =  level1 { ("<<"| ">>" | "&" | "|") level1 }
level1  = level2 { ( "+" | "-" ) level2 }
level2 = level3 { "*" | "/" | "/%" | "%" ) level3 }
level3 = level4 {"^" level4}
level4 = item {"!"}         (*  factorial *)
item =  number|"true"|"false" | ident { "(" real_arg_list ")" }| "("
sentenceValue" )" | ("+" | "-" | "~" ) item
```

PL/0 编译程序测试结果：

（指令说明）

```
# QuickStart
```shell
usage: parser.py [-h] [-i] [-s] [-t] [-v] [-f FILE]

optional arguments:
  -h, --help           show this help message and exit
  -i, --instruction     output instructions
  -s, --stack          output data stack when executing each instruction
  -t, --token          output tokens when parsing
  -v, --varible        output varibles for every static environment
  -f FILE, --file FILE  compile and run codes. Without this arg, enter
                      interactive REPL
```

Run `python parse.py` and enter a REPL state, you can type and run sentences and expressions interactively
```

（测试样例）

```
Therer are some expressions and sentence in file expr.txt, now test it.
`python parser.py -f test/expr.txt`

```c
>> codes:
1     // expression
2     var a=3,b=2,c;.

>>  c:=a+1.
>> begin c; c+1!=1 ; c+1=5 end.
result: 4.0; True; True;
>> for(;b>=0;b:=b-1) print('random(100): %d',random(100)) .
random(100): 14
random(100): 60
random(100): 58
>> begin ++1--1; 1<<2+3%2; 2&1 end.
result: 2.0; 8; 0;
>>   -1+2*3/%2.
result: 2.0;
>>    (1+2.
line 1: ( 1 + 2 .
               ^
[Error]: Expected ")", got "."
>> 4!!.
```
```

```
result: 620448401733239439360000;
>> codes:
1     if   0 then 1
2     elif 1>2 then 2
3     elif false then 3
4     else 4.

result: 4.0;
```
fibonacci
Run `python parser.py  -f test/fibonacci.txt`

```c
>> codes:
1     func fib(n)
2     begin
3         if n=1 || n=2 then return 1;
4         return fib(n-1)+fib(n-2);
5     end ;
6     var n=1;
7     begin
8         while n<15 do
9         begin
10            print('fib[%d]=%d',n,fib(n));
11            n :=n+1;
12        end;
13    end
14    .

fib[1]=1
fib[2]=1
fib[3]=2
fib[4]=3
fib[5]=5
fib[6]=8
fib[7]=13
fib[8]=21
fib[9]=34
fib[10]=55
fib[11]=89
fib[12]=144
fib[13]=233
fib[14]=377


## switch
```

run
`python parser.py  -f test/switch.txt`

```c
>> var x=3;.
>> File: "test/switch.txt"
1    begin
2    switch (x)
3    case 2:print('222')
4    case 3:print(x)
5    case 3:
6    case 3:begin print('x');  end
7    case 3:print(x)
8    case 4:print('a')
9    case 5:print('a')
10   default:print('def');
11   end.
3.0
x
3.0
def
>> File: "test/switch.txt"
1
2    var n=2;.
>> File: "test/switch.txt"
1    begin
2    switch (n)
3    case 1:print('222')
4    case 2:print('n')
5    case 2:begin print('bread'); break;end
6    case 2:print('else')
7    case 4:print('a')
8    case 5:print('a')
9    default:print('def');
10   end.
n
bread
```

do-while

run
`python parser.py  -f test/dowhile.txt`

```c
>> File: "test/dowhile.txt"
1    var n=1;
2    begin
```

```
3         do
4         begin
5             print(n);
6             n:=n+1;
7         end
8         while n<10;
9         print('ZHQnb')
10    end
11    .1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
ZHQnb
```

（实验结论及问题讨论）

这次实验给我们带来的收获都是受益匪浅的，它让我们对编译原理的词法分析、语法语义分析等都有了更进一步的了解。一个算法的好坏决定了一段程序的好坏，它会影响它的运行时间及效率，而一种思想对一个程序来讲也起着至关重要的作用，那一行行代码不仅仅只是一些简单的字符，它还包含了设计者的理念。我们觉得我们学习就是要把原理搞清楚，这样才能把思想与理念运用到更大的层面上去，通过实验我们也深深体会到了这门课的重要性。过程是艰辛的，但结果又是喜人的，相信这门课会对我们的将来起到至关重要的作用。