



## 第八章 源程序的中间形式

8.1 波兰表示

8.2 N-元表示

8.3 抽象机代码



## 8.1 (逆)波兰表示

一般编译程序都生成中间代码，然后再生成目标代码。主要优点是可移植（与具体目标程序无关），且易于目标代码优化。

中间代码有多种形式：

波兰表示      N-元表示      抽象机代码      .....

### 波兰表示

算术表达式：       $F * 3.1416 * R * (H + R)$

转换成如波兰表示：  $F 3.1416 * R * H R + *$



由中缀表达式翻译为波兰表示算法很容易实现——  
可以构造一个类似于算符优先分析法的算法，设立一个  
操作符栈。

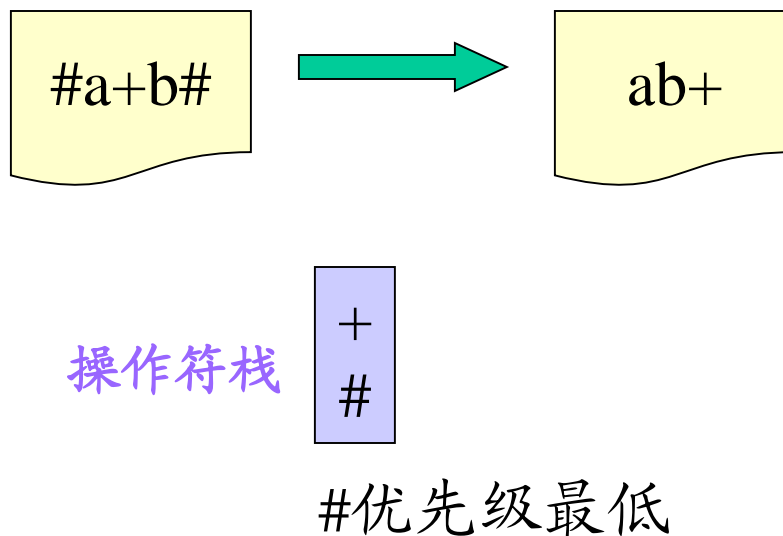
当读到操作数时，就立即输出该操作数；当遇到操  
作符时，则要与栈顶操作符比较优先级，若栈顶操作符  
优先级高于栈外，则输出该栈顶操作符；反之，则栈外  
操作符入栈。

对于赋值语句，则需规定赋值符号的优先级低于其  
它操作符，所以：

赋值语句的波兰表示  $A := F * 3.1416 * R * (H + R)$

$A F 3.1416 * R * H R + * :=$

由中缀表达式翻译  
为波兰表示算法很容易  
实现——可以构造一个  
类似于算符优先分析法的  
算法，设立一个操作  
符栈。



### 算法:

设一个操作符栈；当读到操作数时，立即输出该操作数，当扫描到操作符时，与栈顶操作符比较优先级，若栈顶操作符优先级高于栈外，则输出该栈顶操作符；反之，则栈外操作符入栈。



# 转换算法

## 波兰表示

操作符栈

算术表达式:

$F * 3.1416 * R * (H + R)$

输入

输出

		$F * 3.1416 * R * (H + R)$	
		$* 3.1416 * R * (H + R)$	$F$
		$3.1416 * R * (H + R)$	$F$
*		$* R * (H + R)$	$F 3.1416$
*	•>	$R * (H + R)$	$F 3.1416 *$
*		$* (H + R)$	$F 3.1416 * R$
*	•>	$(H + R)$	$F 3.1416 * R *$
*	<•	$H + R)$	$F 3.1416 * R *$
*		$+ R)$	$F 3.1416 * R * H$
*	<•	$R)$	$F 3.1416 * R * H$
*	+	)	$F 3.1416 * R * HR$
*	+	)	$F 3.1416 * R * HR +$
*	•>	)	$F 3.1416 * R * HR + *$

波兰表示:  $F3.1416 * R * HR + *$

## 波兰表示法的优点:

1. 在不使用括号的情况下可以无二义地说明算术表达式。
2. 波兰表示法更容易转换成机器的汇编语言或机器语言。

操作数出现在紧靠操作符的左边，而操作符在波兰表示中的顺序即为进行计算的顺序。

3. 波兰表示不仅能用来作为算术表达式的中间代码形式，而且也能作为其它语言结构的中间代码形式。

## if 语句的波兰表示

有如下 if 语句： **if**  $\langle \text{expr} \rangle$  **then**  $\langle \text{stmt}_1 \rangle$  **else**  $\langle \text{stmt}_2 \rangle$   $\text{label}_1$

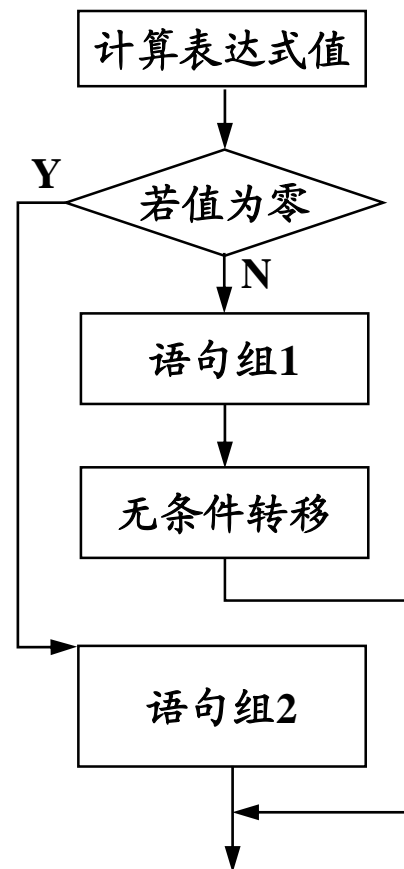
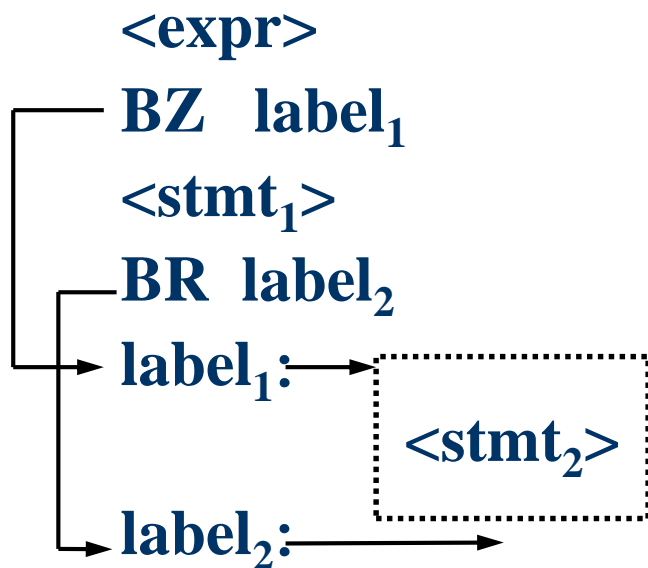
波兰表示为：  $\langle \text{expr} \rangle$   $\langle \text{label}_1 \rangle$  **BZ**  $\langle \text{stmt}_1 \rangle$   $\langle \text{label}_2 \rangle$  **BR**  $\langle \text{stmt}_2 \rangle$   $\text{label}_2$

**BZ:** 二目操作符，如果 $\langle \text{expr} \rangle$ 的计算结果为0 (false)，则产生一个 $\langle \text{label}_1 \rangle$ 的转移，而 $\text{label}_1$ 是 $\langle \text{stmt}_2 \rangle$ 的头一个符号。

**BR:** 一目操作符，它产生一个 $\langle \text{label}_2 \rangle$ 的转移，而 $\langle \text{label}_2 \rangle$ 是一个紧跟在 $\langle \text{stmt}_2 \rangle$ 后面的符号（即 if 语句后的第一个语句的头一个符号）。

波兰表示为： $\langle \text{expr} \rangle \langle \text{label}_1 \rangle \text{BZ} \langle \text{stmt}_1 \rangle \langle \text{label}_2 \rangle \text{BR} \langle \text{stmt}_2 \rangle$

由 if 语句的波兰表示可生成如下的目标程序框架：



其它语言结构也很容易将其翻译成波兰表示，但使用波兰表示优化不是十分方便。





## 8.2 N-元表示

在该表示中，每条指令由  $n$  个域所组成，通常第一个域表示操作符，其余为操作数。

常用的  $n$  元表示是：三元式 四元式

三元式

操作符	左操作数	右操作数
-----	------	------

表达式的三元式：

$w * x + (y + z)$



(1)  $*$ ,  $w$ ,  $x$

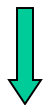
(2)  $+$ ,  $y$ ,  $z$

(3)  $+$ , (1), (2)

第三个三元式  
中的操作数(1) (2)  
表示第(1)和第(2)  
条三元式的计算结果。

## 条件语句的三元式:

If  $x > y$  then  
     $z := x$ ;  
else  $z := y + 1$ ;



$z := y + 1$  {  
    (1)  $- , x , y$   
    (2) **BMZ** , (1) , (5)  
    (3)  $:= , Z , X$   
    (4) **BR** , , (7)  
    (5)  $+, Y , 1$   
    (6)  $:= , Z , (5)$   
    (7)  $:$   
     $:$

其中:

**BMZ**: 是二元操作符, 测试第二个域的值。若 $\leq 0$ , 则按第3个域的地址转移, 若为正值则该指令作废。

**BR**: 一元操作符, 按第3个域作无条件转移。



使用三元式也不便于代码优化，因为优化要删除一些三元式，或对某些三元式的位置要进行变更，由于三元式的结果（表示为编号），可以是某个三元式的操作数，随着三元式位置的变更也将作相应的修改，很费事！

间接三元式：

为了便于在三元式上作优化处理，可使用间接三元式。

例:  $A := B + C * D / E$   
 $F := C * D$

用直接三元式表示为:

(1)	*	C	D
(2)	/	(1)	E
(3)	+	B	(2)
(4)	:=	A	(3)
(5)	*	C	D
(6)	:=	F	(5)



(1)	*	C	D
(2)	/	(1)	E
(3)	+	B	(2)
(4)	:=	A	(3)
(5)	:=	F	(1)

## 用间接三元式表示为:

操作	三元式
1. (1)	(1) *, C, D
2. (2)	(2) /, (1), E
3. (3)	(3) +, B, (2)
4. (4)	(4) :=, A, (3)
5. (1)	(5) :=, F, (1)
6. (5)	

将执行顺序和三元式编号分离

三元式的执行次序用另一张表表示, 这样在优化时 (三元式位置的变更实际是执行顺序的变化), 三元式可以不变, 而仅仅改变其执行顺序表。

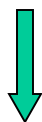


## 四元式表示

操作符	操作数1	操作数2	结果
-----	------	------	----

结果：通常是编译时分配的临时变量，可由编译程序分配一个寄存器或主存单元。

例：  $(A + B) * (C + D) - E$



$+$ , A, B, T1  
 $+$ , C, D, T2  
 $*$ , T1, T2, T3  
 $-$ , T3, E, T4

其中T1 ~ T4为临时变量。  
用四元式优化比较方便

## 8.3 抽象机代码

许多Pascal编译系统生成的中间代码是一种称为P-code的抽象代码。P-code的“P”即“Pseudo”。

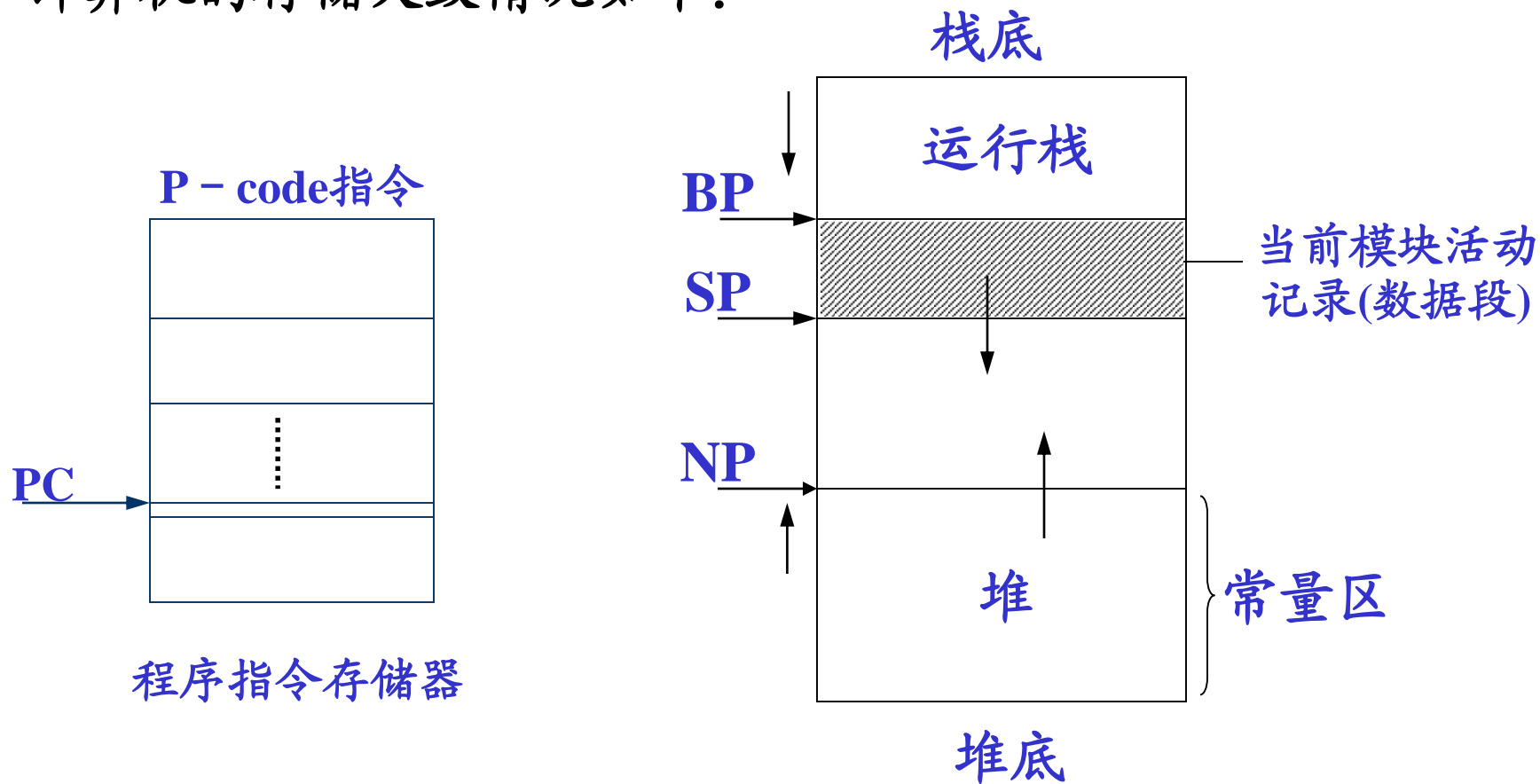
既然是“抽象机”，就是表示它并不是实际的物理目标机器而通常是虚拟的一台“堆栈计算机”。该堆栈式计算机主要由若干寄存器、一个保存程序指令的储存器和一个堆栈式数据及操作存储组成。

## 寄存器有：

1. PC —— 程序计数器。
2. NP —— New指针，指向“堆”的顶部。“堆”用来存放由NEW生成的动态数据。
3. SP —— 运行栈指针，存放所有可按源程序的数据声明直接寻址的数据。
4. BP —— 基地址指针，即指向当前活动记录的起始位置指针。
5. 其他（如MP—栈标志指针，EP—极限栈指针等）

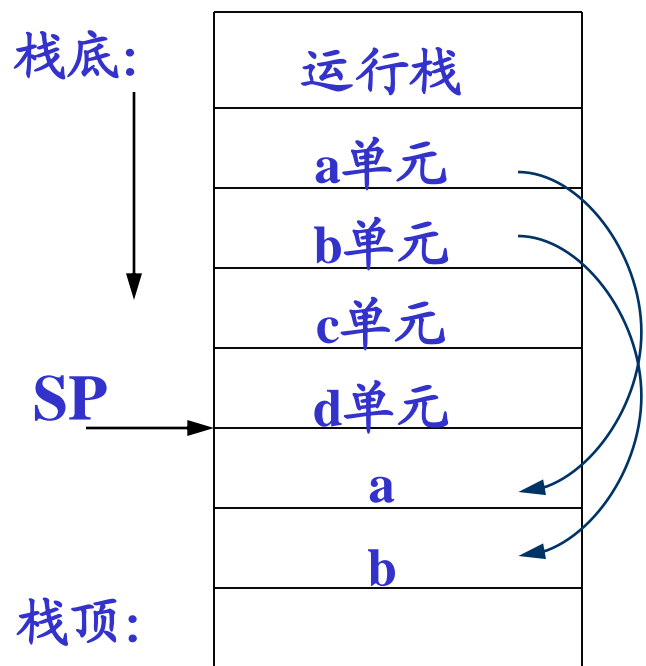


计算机的存储大致情况如下:





运行P-code的抽象机没有专门的运算器或累加器，所有的运算（操作）都在运行栈的栈顶进行，如要进行 $d := (a + b) * c$ 的运算，生成P-code序列为：



取a	LOD a
取b	LOD b
+	ADD
取c	LOD c
*	MUL
送d	STO d

P-code实际上是波兰表示形式的中间代码。



编译程序生成P-code指令程序后，我们可以用一个解释执行程序（interpreter）来解释执行P-code，当然也可以把P-code再变成某一机器的目标代码。

显然，生成抽象机P-code的编译程序是很容易移植的。

**作业： P175    1, 2, 4**

## 补充：中间代码生成实例——翻译成后缀式

**begin**

**k := 100;**

**L: if k > i + j then**

**begin k := k - 1; goto L; end**

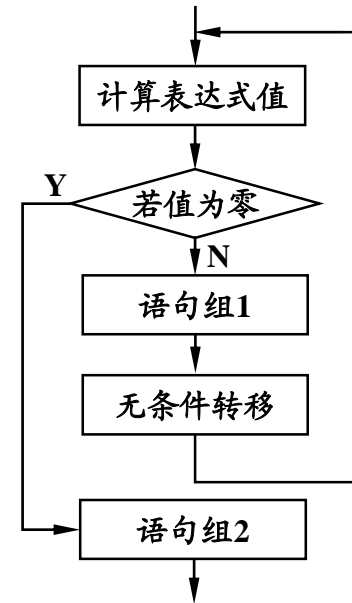
**else k := i ^ 2 - j ^ 2;**

**i := 0;**

**end**



```
begin
    k := 100;
L: if k > i + j then
    begin k := k - 1; goto L; end
    else k := i ^ 2 - j ^ 2;
    i := 0;
end
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



**begin**

**k := 100;**

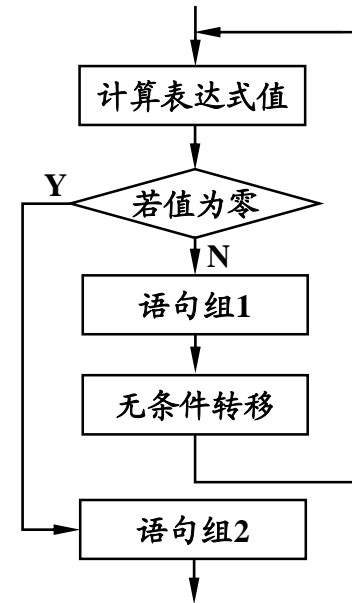
**L: if k > i + j then**

**begin k := k - 1; goto L; end**

**else k := i ^ 2 - j ^ 2;**

**i := 0;**

**end**



1	2	3													
k	100	:=													

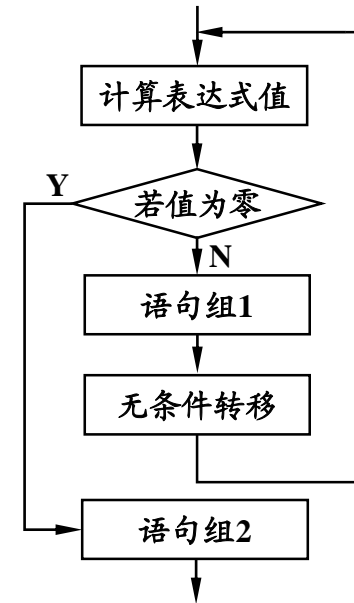


```
begin
  k := 100;
L: if k > i + j then
  begin k := k - 1; goto L; end
  else k := i ^ 2 - j ^ 2;
  i := 0;
end
```

L标号

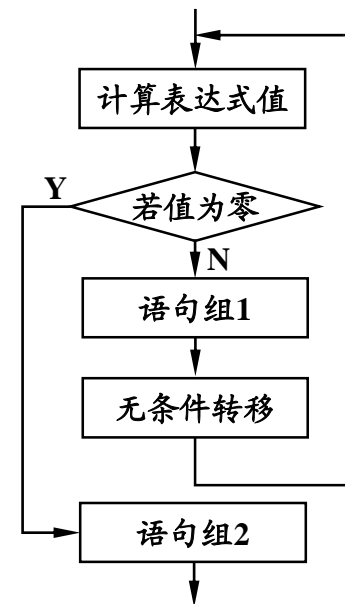


1	2	3	4	5	6	7	8								
k	100	:=	k	i	j	+	>								



```

begin
  k := 100;
  L: if k > i + j then
    begin k := k - 1; goto L; end
  else k := i ^ 2 - j ^ 2;
  i := 0;
end
  
```



L标号

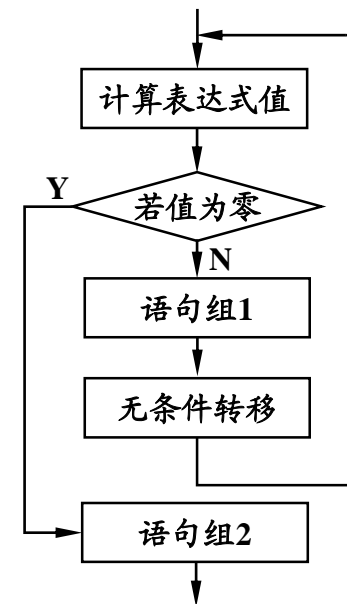
1	2	3	4	5	6	7	8	9	10						
k	100	:=	k	i	j	+	>	?	jez						

label1



```

begin
  k := 100;
L: if k > i + j then
  begin k := k - 1; goto L; end
  else k := i ^ 2 - j ^ 2;
  i := 0;
end
  
```



L标号

1	2	3	4	5	6	7	8	9	10						
k	100	:=	k	i	j	+	>	0	jez						

无法确定的地址先填入0。  
一旦地址确定“回填”之！

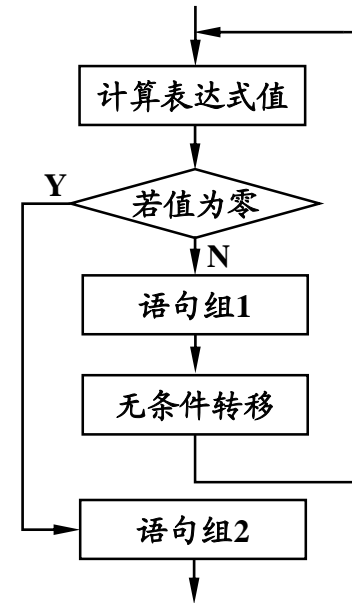


```
begin
  k := 100;
  L: if k > i + j then
    begin k := k - 1; goto L; end
  else k := i ^ 2 - j ^ 2;
  i := 0;
end
```

L标号

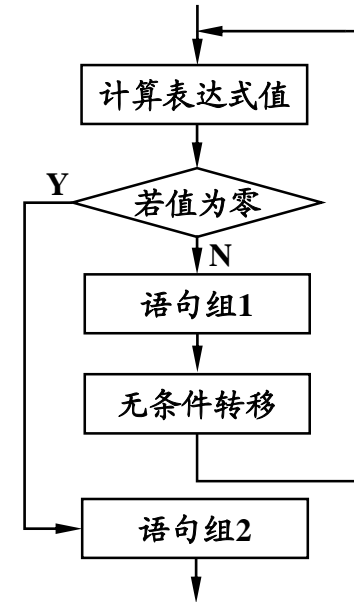


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
k	100	:=	k	i	j	+	>	0	jez	k	k	1	-	:=	





```
begin
  k := 100;
  L: if k > i + j then
    begin k := k - 1; goto L; end
  else k := i ^ 2 - j ^ 2;
  i := 0;
end
```



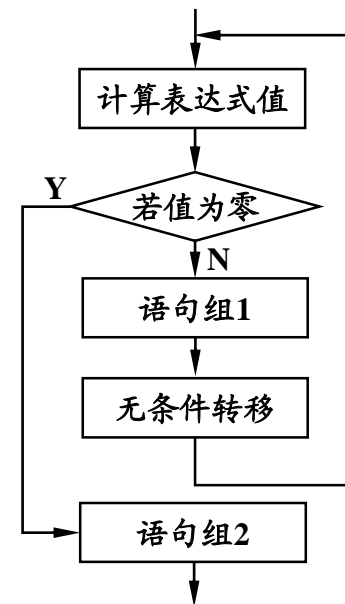
L标号

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	100	:=	k	i	j	+	>	0	jez	k	k	1	-	:=	4

17															
j															



```
begin
    k := 100;
L: if k > i + j then
    begin k := k - 1; goto L; end
    else k := i ^ 2 - j ^ 2;
    i := 0;
end
```



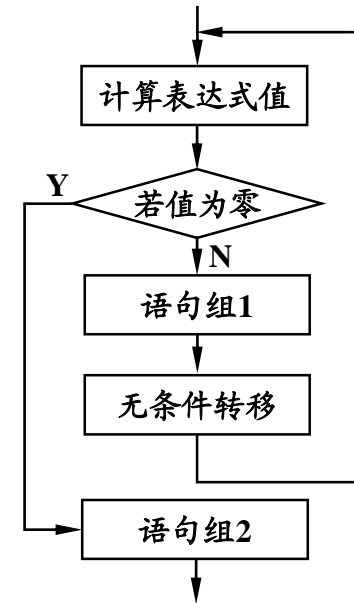
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	100	:=	k	i	j	+	>	0	jez	k	k	1	-	:=	4

17	18	19													
j	?	j													

↑  
label2

```

begin
    k := 100;
L: if k > i + j then
    begin k := k - 1; goto L; end
    else k := i ^ 2 - j ^ 2;
    i := 0;
end
    
```



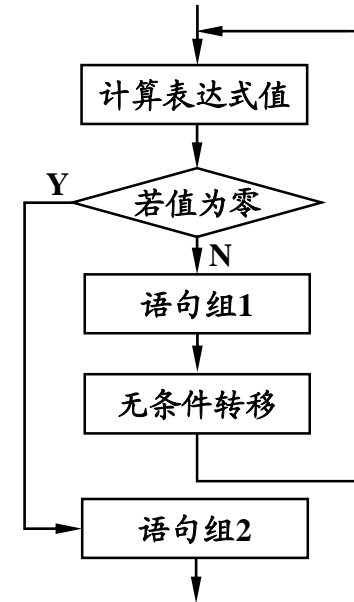
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	100	:=	k	i	j	+	>	20	jez	k	k	1	-	:=	4

17	18	19	20	21	22	23	24	25	26	27	28				
j	0	j	k	i	2	^	j	2	^	-	:=				

↑  
label1

```

begin
    k := 100;
L: if k > i + j then
    begin k := k - 1; goto L; end
    else k := i ^ 2 - j ^ 2;
    i := 0;
end
    
```



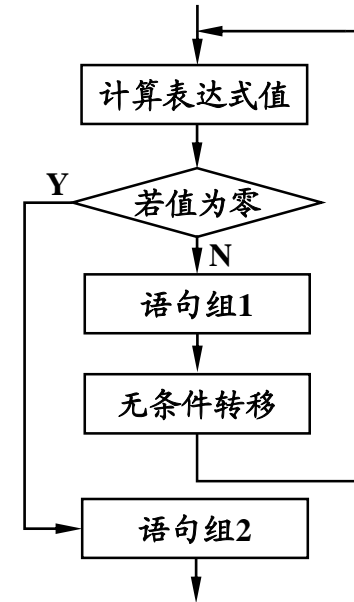
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	100	:=	k	i	j	+	>	20	jez	k	k	1	-	:=	4

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
j	29	j	k	i	2	^	j	2	^	-	:=	i	0	:=	...

label2

```

begin
    k := 100;
L: if k > i + j then
    begin k := k - 1; goto L; end
    else k := i ^ 2 - j ^ 2;
    i := 0;
end
    
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	100	:=	k	i	j	+	>	20	jez	k	k	1	-	:=	4

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
j	29	j	k	i	2	^	j	2	^	-	:=	i	0	:=	...

该中间代码程序有死代码!