



第七章 源程序的中间形式

7.1 波兰表示

7.2 N - 元表示

7.3 抽象机代码

7.4 其它形式的中间代码



7.1 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：

(逆)波兰表示、N-元表示、抽象机代码、LLVM IR

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。

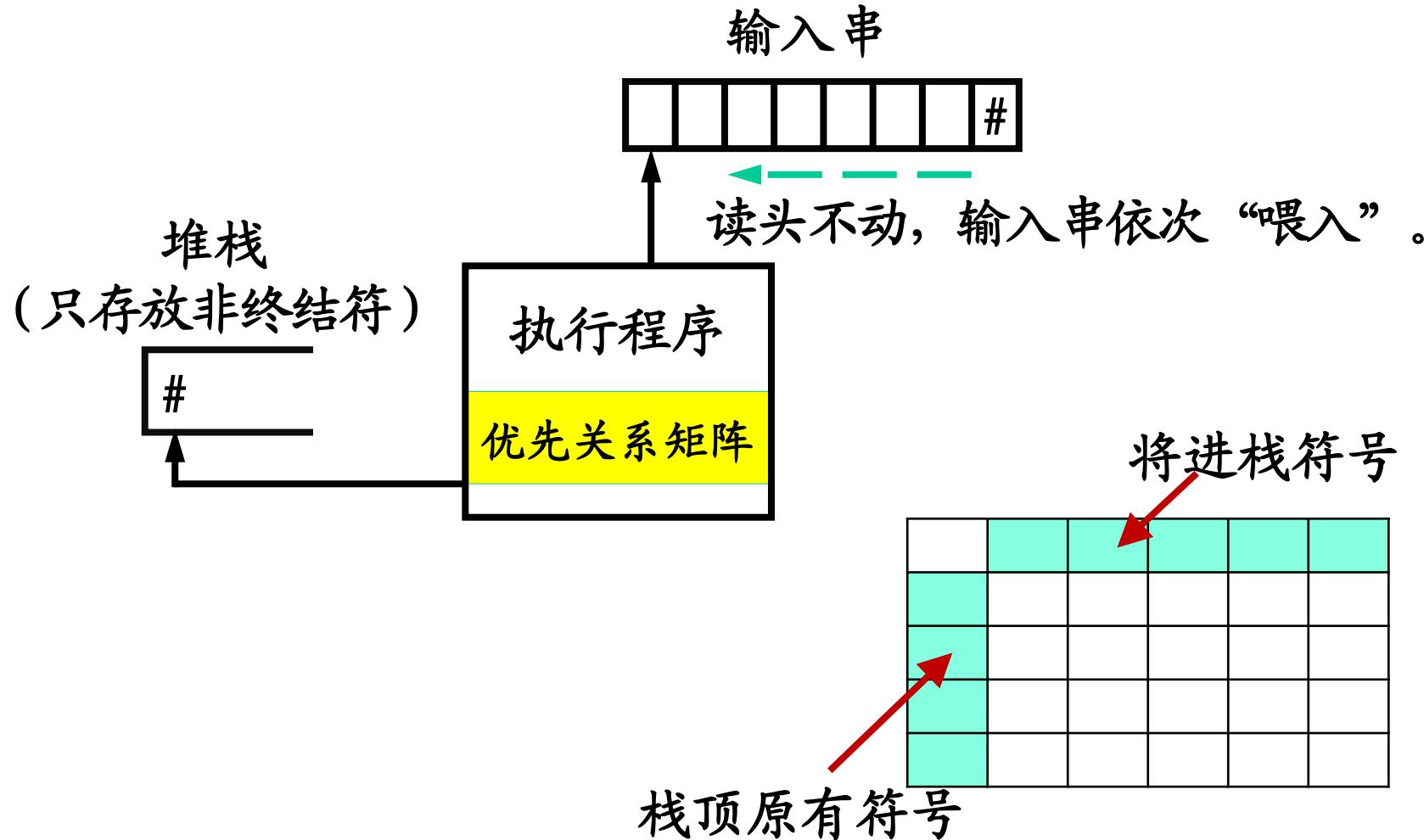
对于赋值语句，则需规定赋值符号的优先级低于其它操作符，所以：

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

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



下推自动机





优先关系矩阵

a \ b	+	*	i	()	#
+	>	<	&	&	>	>
*	>	>	&	&	>	>
i	>	>			>	>
(&	&	&	&	-	
)	>	>			>	>
#	&	&	&	&		

- 矩阵空白处表示这两个终结符不能相邻，故没有优先关系。
- #优先级最低。



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

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

F

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

F

$3.1416 * R * (H + R)$

F 3.1416

*

$* R * (H + R)$

F 3.1416 *

*

$R * (H + R)$

F 3.1416 * R

*

$* (H + R)$

F 3.1416 * R *

*

$(H + R)$

F 3.1416 * R *

*

$H + R)$

F 3.1416 * R * H

*

$(< R)$

F 3.1416 * R * H

*

$(+ R)$

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 <expr> then <stmt₁> else <stmt₂>**

波兰表示为 : <expr> <label₁>**BZ**<stmt₁> <label₂>**BR**<stmt₂>

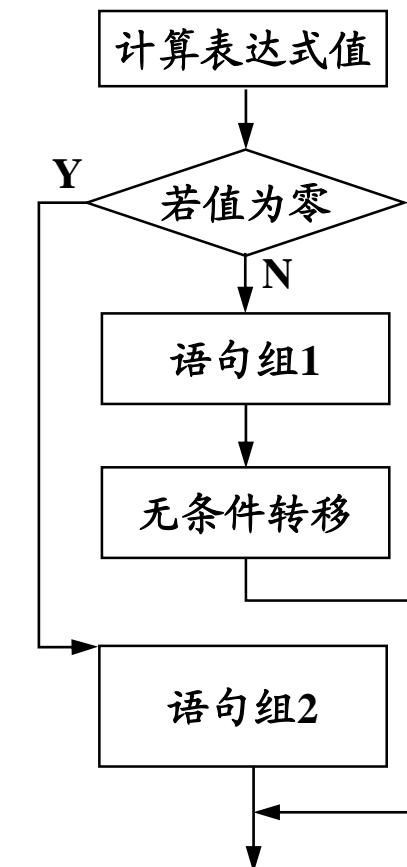
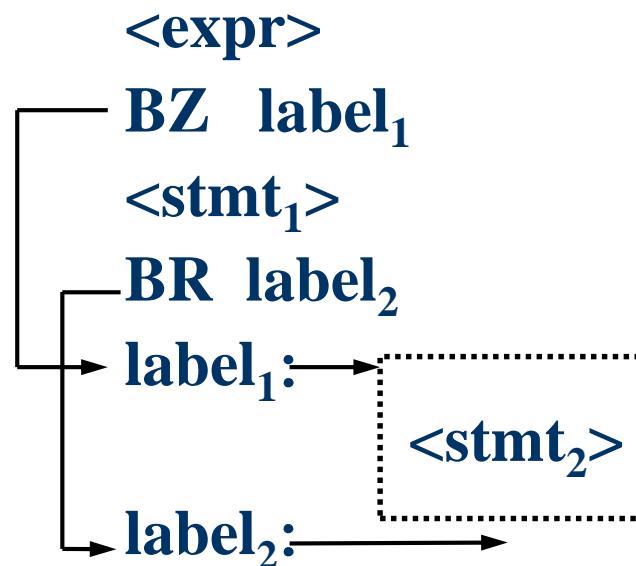
BZ: 二目操作符，如果<expr>的计算结果为0 (false)，则产生一个<label₁>的转移，而label₁是<stmt₂>的第一个符号。

BR: 一目操作符，它产生一个<label₂>的转移，而<label₂>是一个紧跟在<stmt₂>后面的符号（即 if 语句后的第一个语句的第一个符号）。



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。



中间代码生成实例——拉链与回填技术

begin

k := 100;

L: if k > i + j then

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

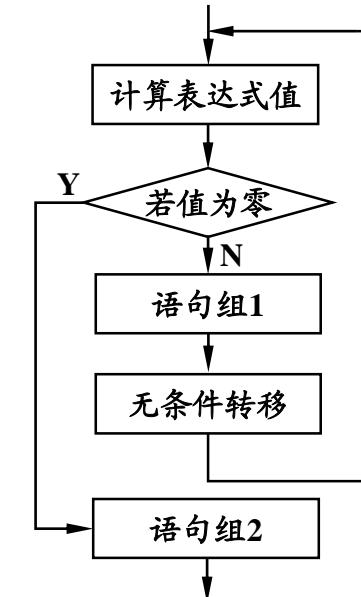
else k := i ^ 2 - j ^ 2;

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;

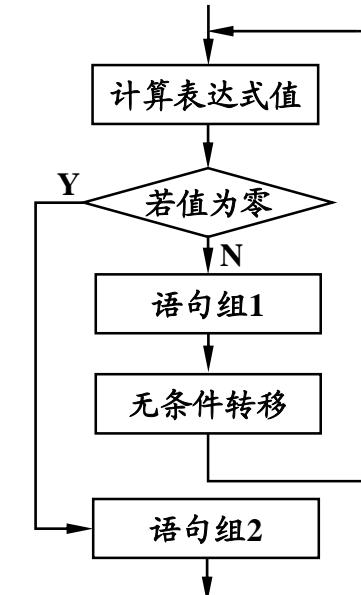
L: if k > i + j then

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

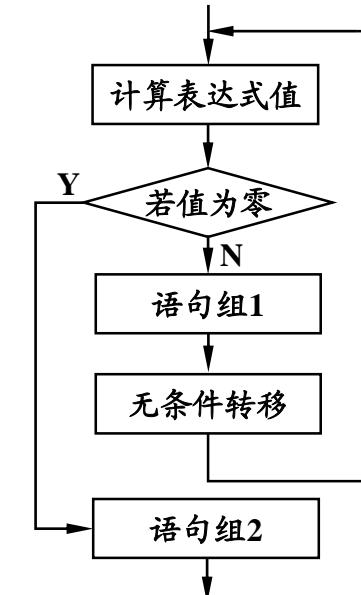
else k := i ^ 2 - j ^ 2;

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;

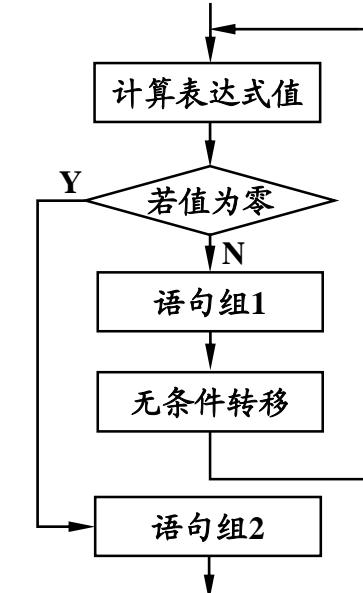
i:= 0;

end

L标号

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

↑
label1





begin

k := 100;

L: if k > i + j then

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

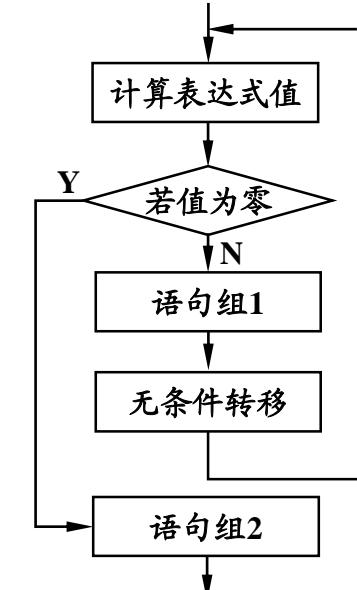
else k := i ^ 2 - j ^ 2;

i:= 0;

end

L标号

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



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

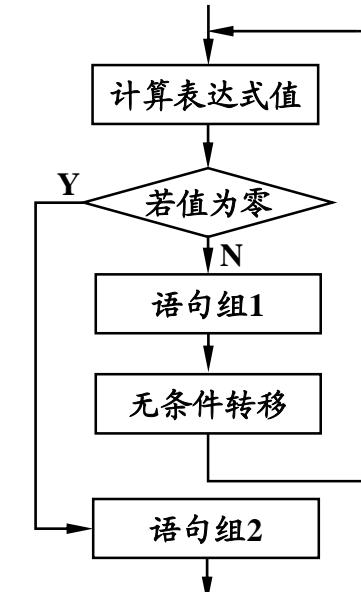
else k := i ^ 2 - j ^ 2;

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	BZ	k	k	1	-	:=	





begin

 k := 100;

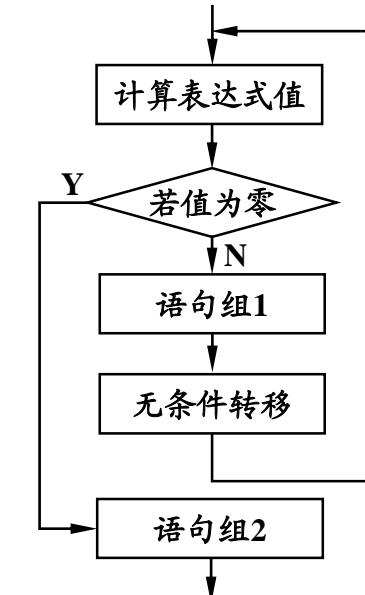
 L: if k > i + j then

 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	BZ	k	k	1	-	:=	4

goto的位置，
在这个表是4

17															
BR															



begin

 k := 100;

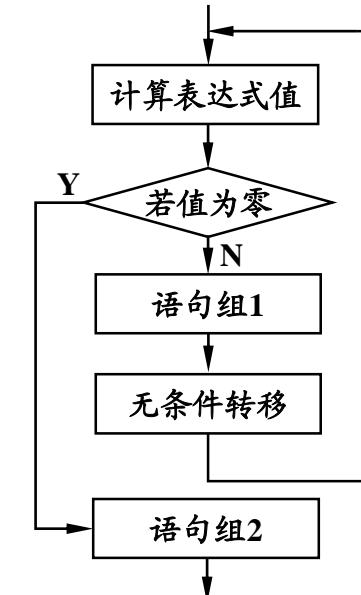
 L: if k > i + j then

 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	BZ	k	k	1	-	:=	4

17	18	19													
BR	?	BR													

↑
label2



begin

k := 100;

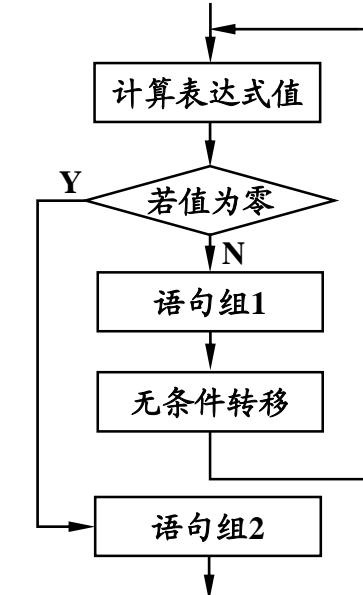
L: if k > i + j then

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	BZ	k	k	1	-	:=	4

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

↑
label1



begin

 k := 100;

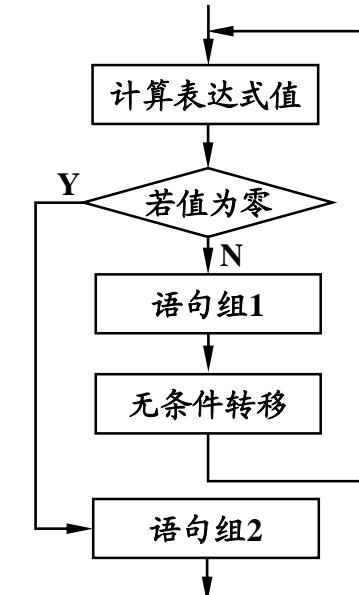
 L: if k > i + j then

 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	BZ	k	k	1	-	:=	4

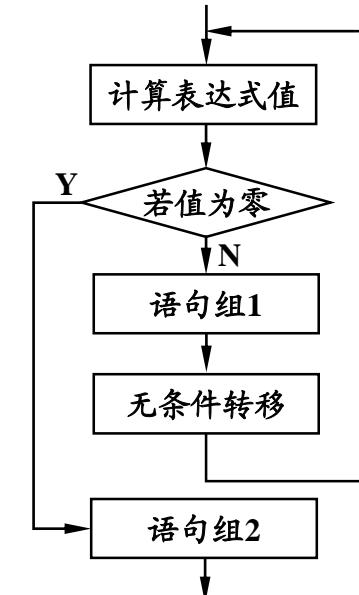
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
BR	29	BR	k	i	2	^	j	2	^	-	:=	i	0	:=	...

29 BR这个代码是死代码，因为goto的原因导致这里永远执行不到，前面已经跳转。

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	BZ	k	k	1	-	:=	4

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

该中间代码程序有死代码!



7.2 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 $z := y + 1;$
 ↓

$z := y + 1 \left\{ \begin{array}{l} (1) -, x, y \\ (2) BMZ, (1), (5) \\ (3) :=, Z, X \\ (4) BR, , (7) \\ (5) +, Y, 1 \\ (6) :=, Z, (5) \\ (7) : \\ \vdots \end{array} \right.$

其中:

BMZ: 是二元操作符, 测试第二个域的值。若 ≤ 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



7.3 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。

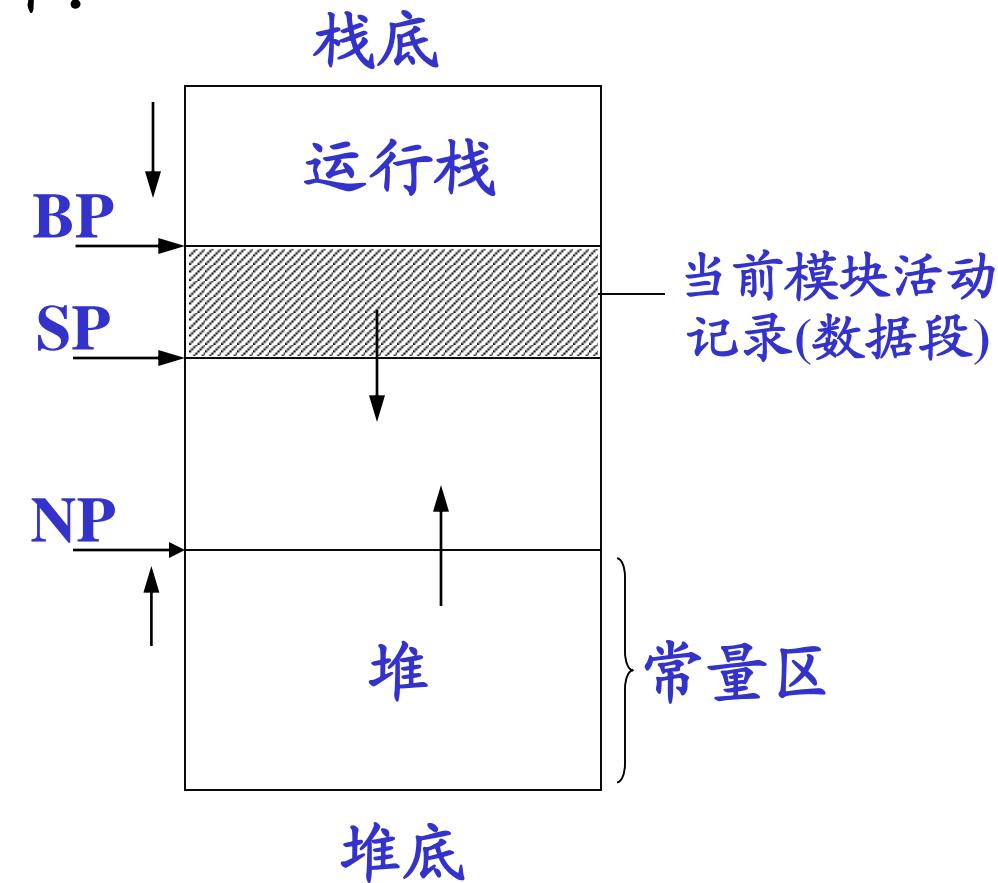
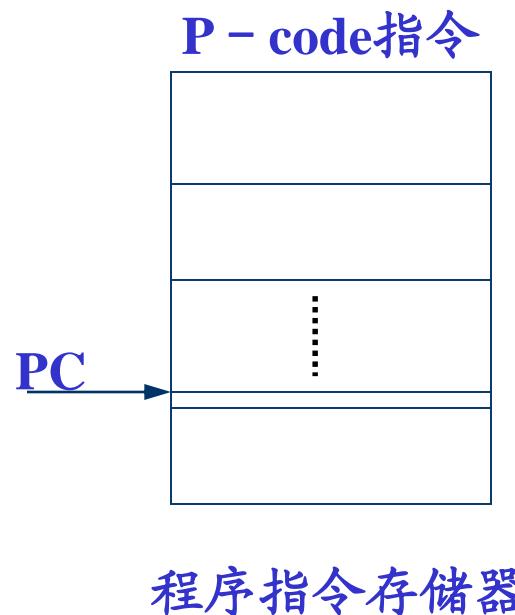


寄存器有：

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）

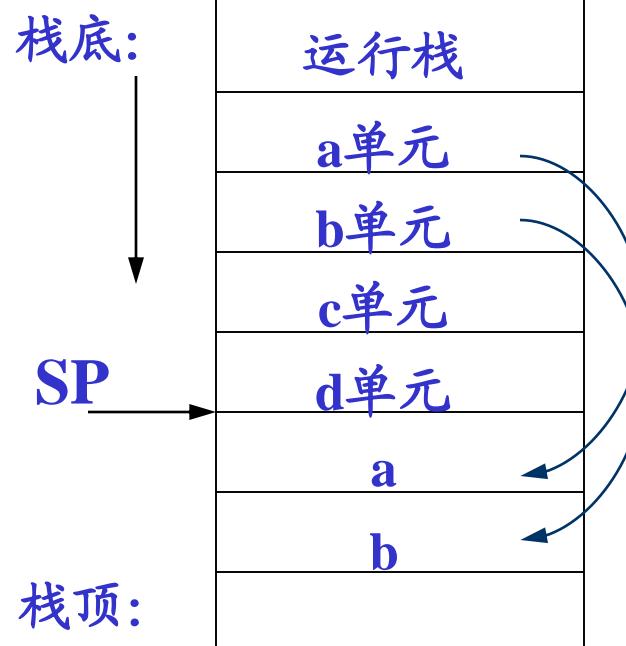


计算机的存储大致情况如下：





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 ($d\ a\ b\ +\ c\ *\ :=$)。



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。



7.4 其它形式的中间代码

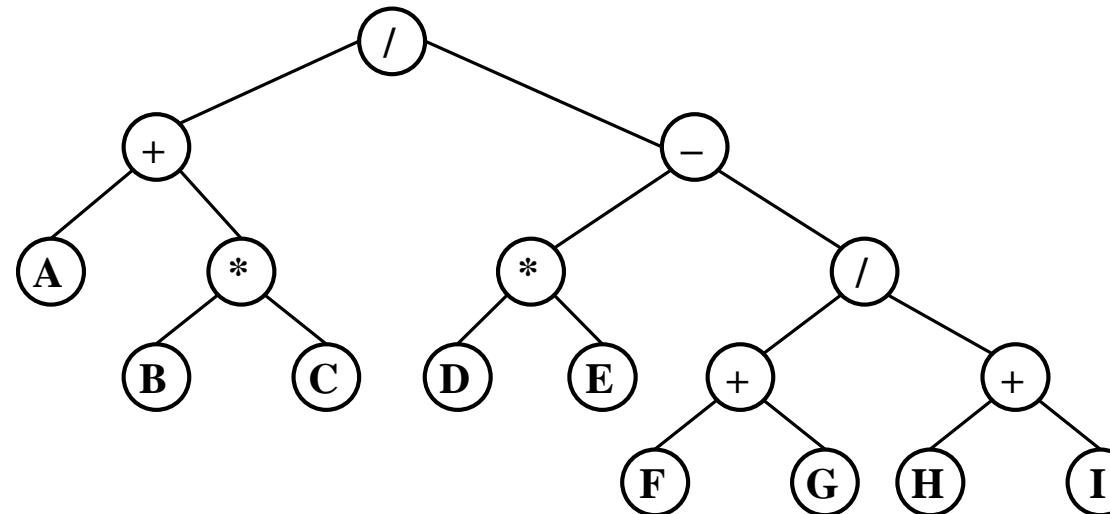
中间代码的二叉树表示

- 抽象语法树 (AST: Abstract Syntax Tree)

用树型图的方式表示中间代码；

操作数出现在叶节点上，操作符出现在中间结点。

例如： $(A + B * C) / (D * E - (F + G) / (H + I))$





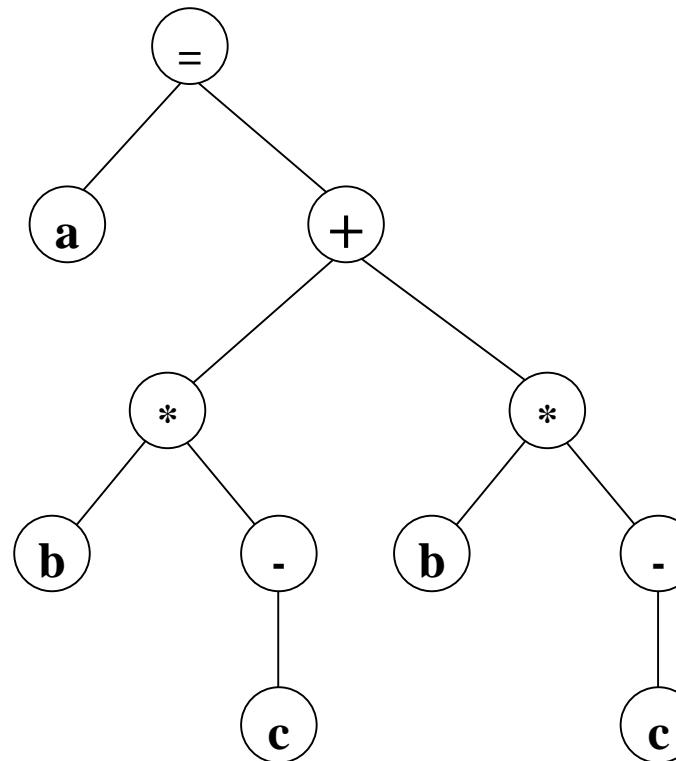
中间代码的图表示

- DAG图（Directed Acyclic Graphs 有向无环图）
—语法树的一种归约表达方式

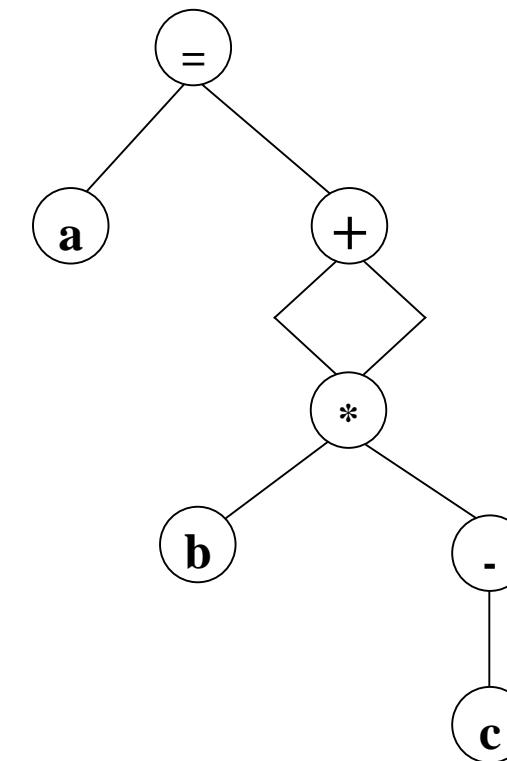
1. 图的叶节点由变量名或常量所标记。对于那些在基本块内先引用再赋值的变量，可以采用变量名加下标0的方式命名其初值。
2. 图的中间节点由中间代码的操作符所标记，代表着基本块中一条或多条中间代码。
3. 基本块中变量的最终计算结果，都对应着图中的一个节点；具有初值的变量，其初值和最终值可以分别对应不同的节点。

中间代码的图表示

- 例如赋值语句： $a = b * (-c) + b * (-c)$ 的DAG图



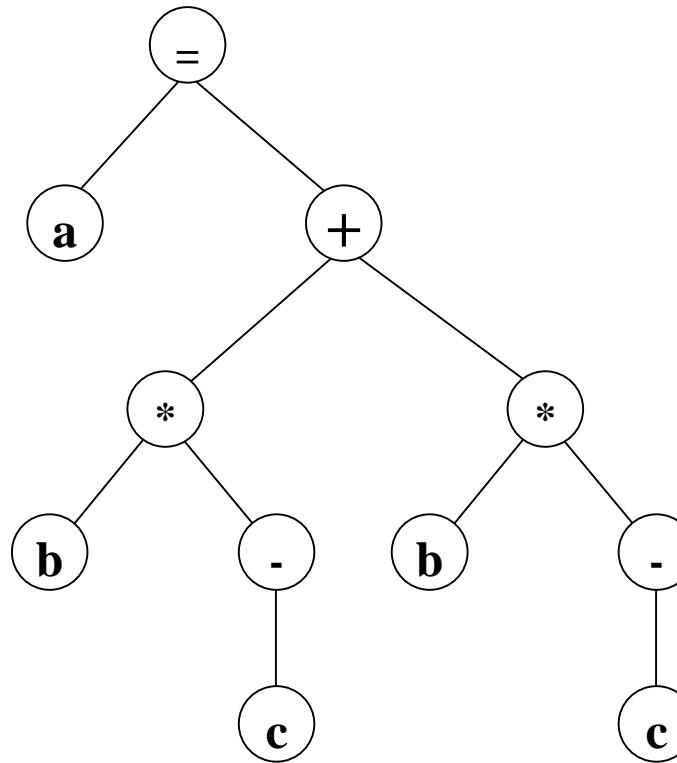
语法树



DAG图

三地址码与语法树的对应关系

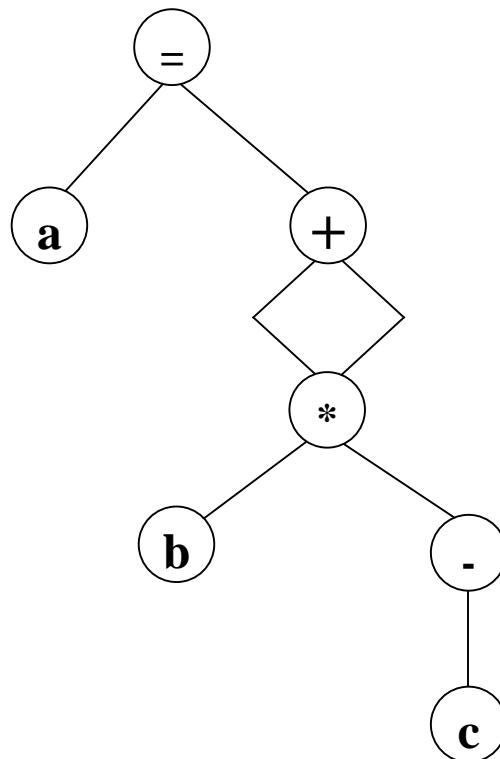
- 赋值语句: $a = b * (-c) + b * (-c)$



语法树

t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5

三地址码与DAG图的对应关系



DAG图

$t1 := - c$

$t2 := b * t1$

$t3 := - c$

$t4 := b * t3$

$t5 := t2 + t4$

$a := t5$



一种特殊的四元式表达方式：SSA

Single Static Assignment form(SSA form)静态单一赋值形式的 IR 主要特征是每个变量只赋值一次。

SSA的优点：
1) 可以简化很多优化的过程；
2) 可以获得更好的优化结果。

$y := 1$

...

$y := 2$

$x := y + z$

$y1 := 1$

...

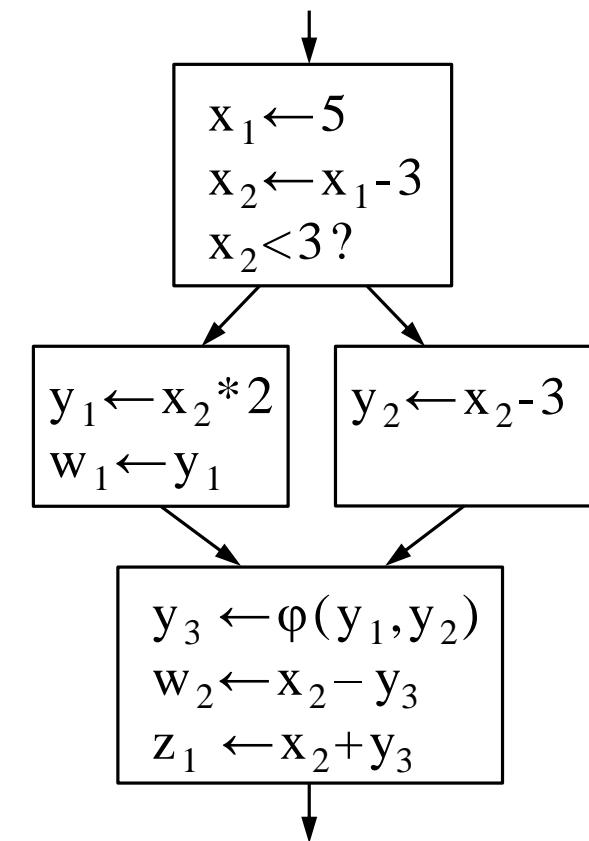
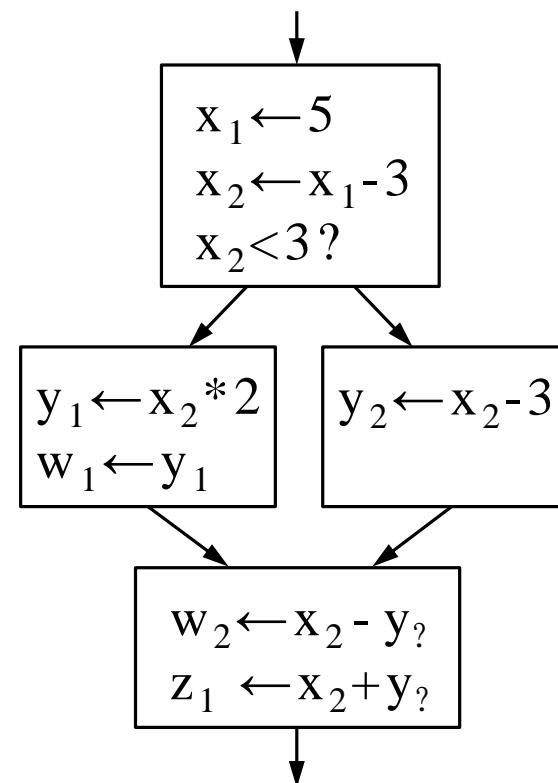
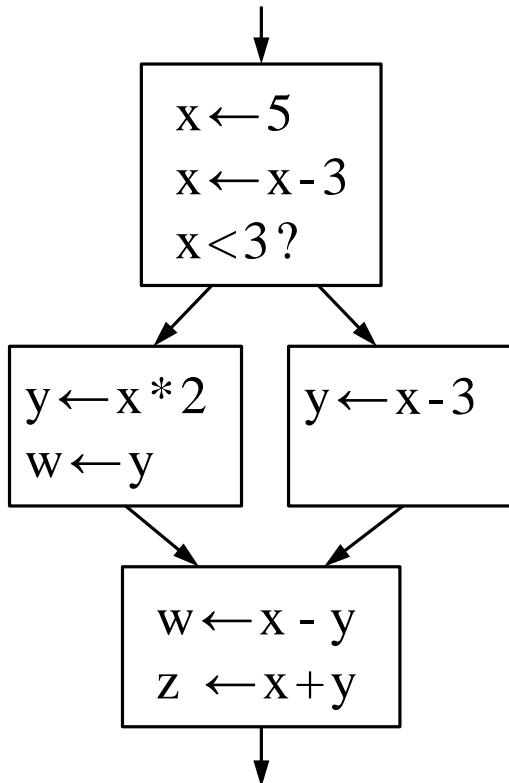
$y2 := 2$

$x := y2 + z$

例，很容易分析出y1
是可以优化掉的变量



SSA可以从普通的四元式转化而来。如何转化？





SSA的关键问题——如何加入 Φ 节点？

Φ 节点的参数应包括所有可能到达其位置的同一个变量的所有定义： $x_{n+1} = \Phi(x_1, x_2, x_3, \dots, x_n)$ 。

在某个分支汇聚点，如果有同一个变量的多个定义点可能到达，就需要在此处增加相应的 Φ 节点，汇聚所有可能到达的定义，将其转化为新的定义。

作业： P144 1, 2, 3, 4