

第七章 源程序的中间形式

- 波兰表示
- N—元表示
- 中间代码的图表示
- 抽象机代码
- 静态单赋值形式 SSA

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。有多种中间代码形式：

波兰表示 N一元组表示 抽象机代码

波兰表示

由波兰逻辑学家 J.Lukasiewicz 提出

- 前缀表示：<操作符><操作数序列>
- 后缀表示：<操作数序列><操作符>

前缀表达：（波兰表达）

+ 3 5

后缀表达：（逆波兰表达）

3 5 +

Compiler Course

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。有多种中间代码形式：

波兰表示 N一元组表示 抽象机代码

波兰表示

算术表达式： $F * 3.1416 * R * (H + R)$

转换成波兰表示： $F 3.1416 * R * H R + *$

赋值语句： $A := F * 3.1416 * R * (H + R)$

波兰表示： $AF3.1416 * R * HR + * :=$

7.1 波兰表示

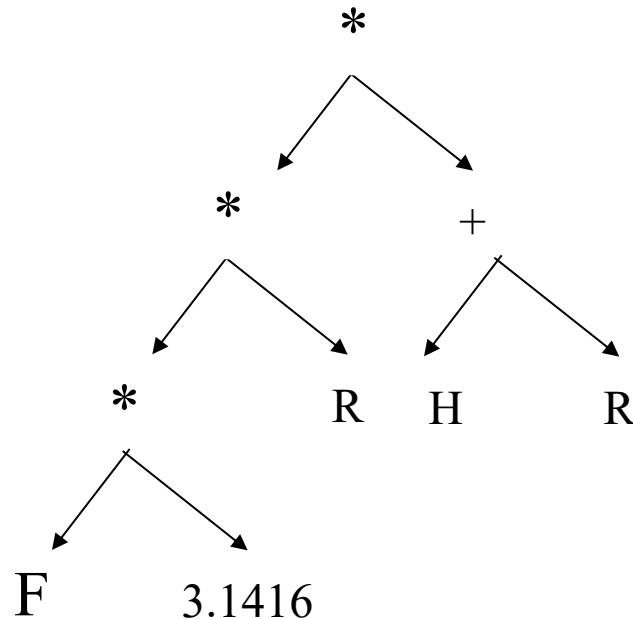
波兰表示

从语法树的角度看“波兰表示”

算术表达式：

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

$E ::= E + T \mid T$
 $T ::= T * F \mid F$
 $F ::= (E) \mid i$



前序遍历

中序遍历

后序遍历

7.1 波兰表示

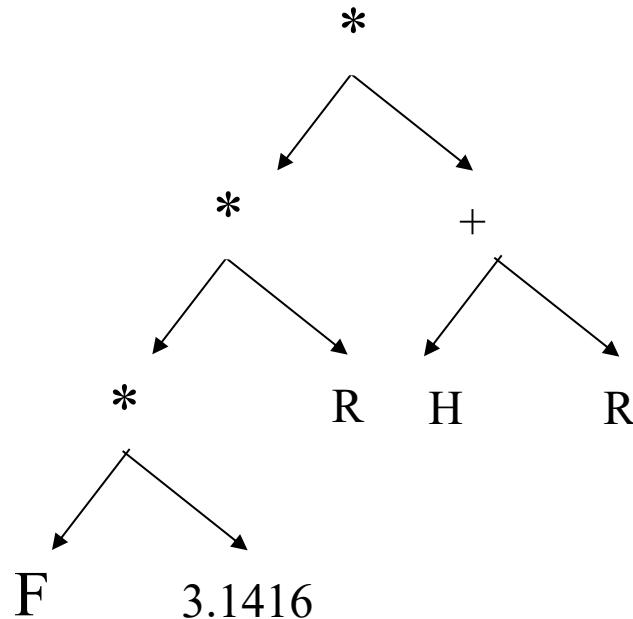
波兰表示

从语法树的角度看“波兰表示”

算术表达式:

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

$E ::= E + T \mid T$
 $T ::= T * F \mid F$
 $F ::= (E) \mid i$

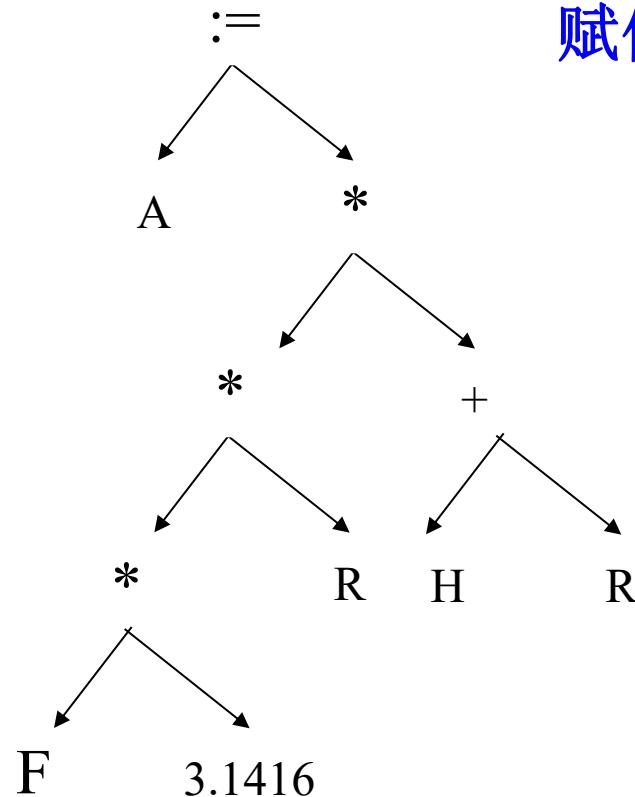


逆波兰表示: $F3.1416 * R * H R + *$

7.1 波兰表示

波兰表示

从语法树的角度看“波兰表示”



赋值语句: $A := F * 3.1416 * R * (H + R)$

前序遍历

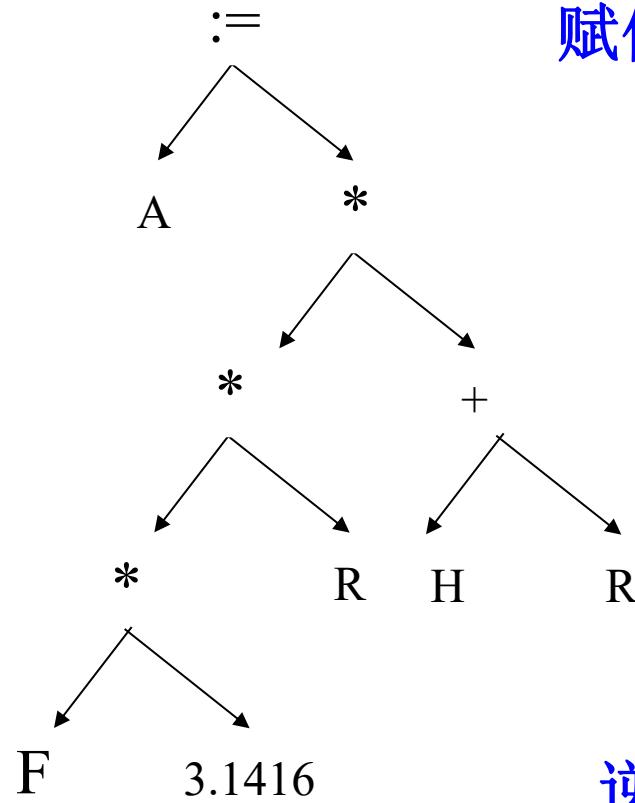
中序遍历

后序遍历

7.1 波兰表示

波兰表示

从语法树的角度看“波兰表示”



赋值语句: $A := F * 3.1416 * R * (H + R)$

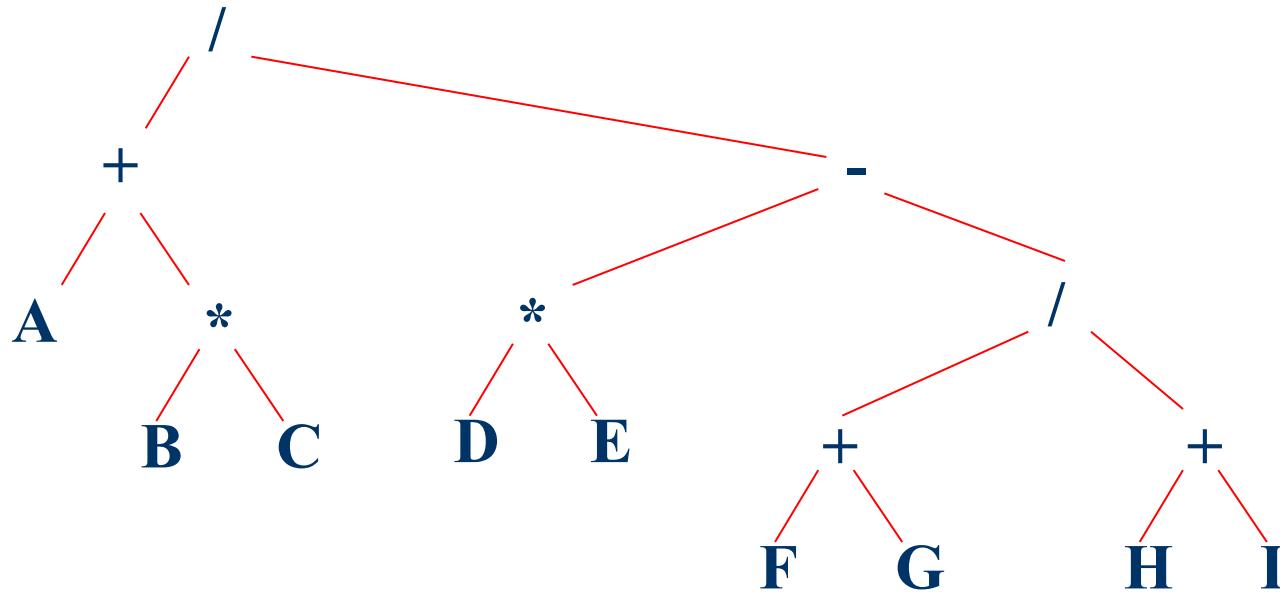
前序遍历

中序遍历

后序遍历

逆波兰表示: $AF3.1416 * R * HR + * :=$

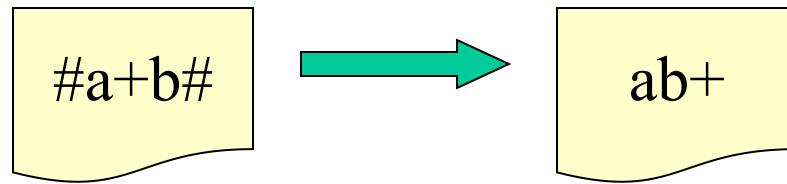
例: $(A+B * C) / (D * E - (F+G) / (H+I))$



前序遍历（根左右）: **$+A*BC-*DE/+FG+HI$** (前缀表示)

后序遍历（左右根）: **$ABC*+DE*FG+HI+/-$** (后缀表达)

中序遍历（左根右）: **$A+B*C/D*E-F+G/H+I$**



#优先级最低

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

算术表达式:

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

操作符栈

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。

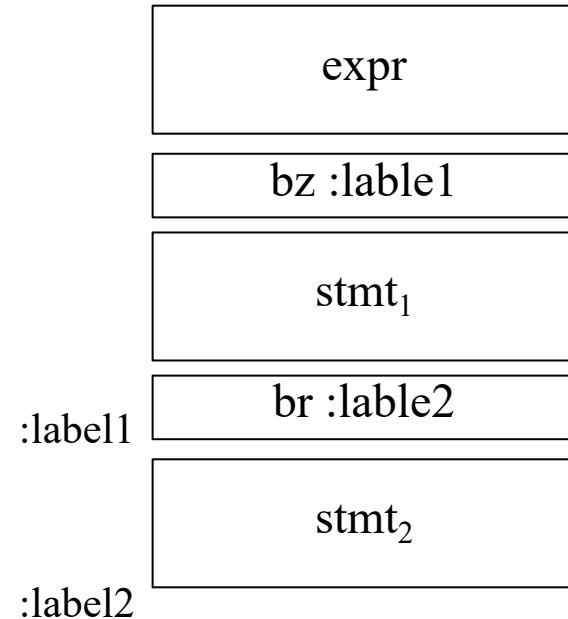
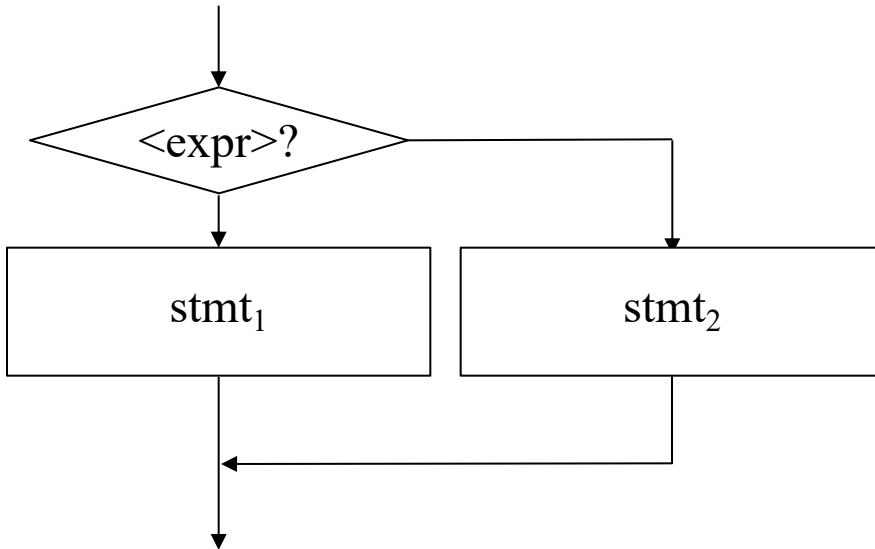
操作数出现在紧靠操作符的左边，而操作符在波兰表示中的顺序即为进行计算的顺序。

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

if 语句 : if <expr> then <stmt₁> else <stmt₂>

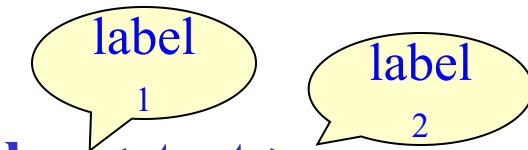
label₁ label₂



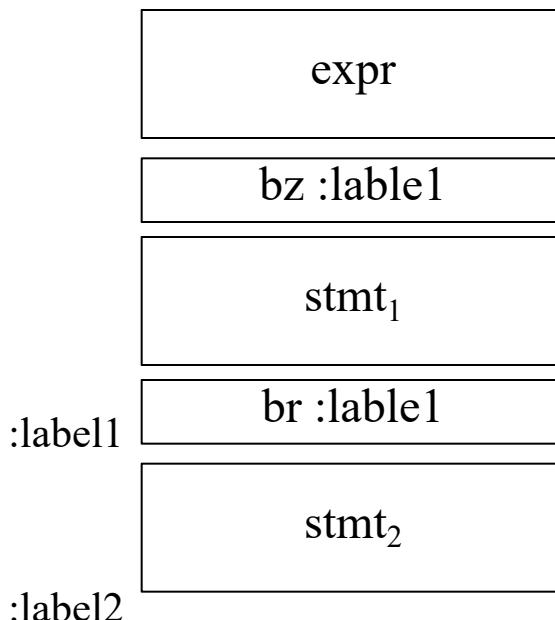
Compiler

if 语句的波兰表示

if 语句 : if <expr> then <stmt₁> else <stmt₂>



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



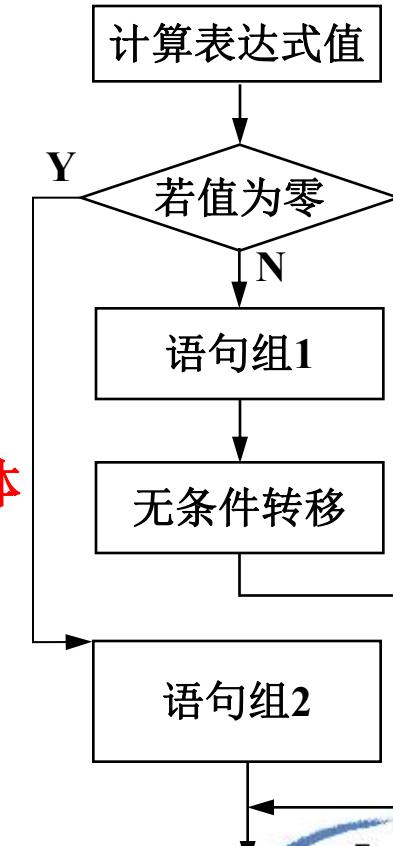
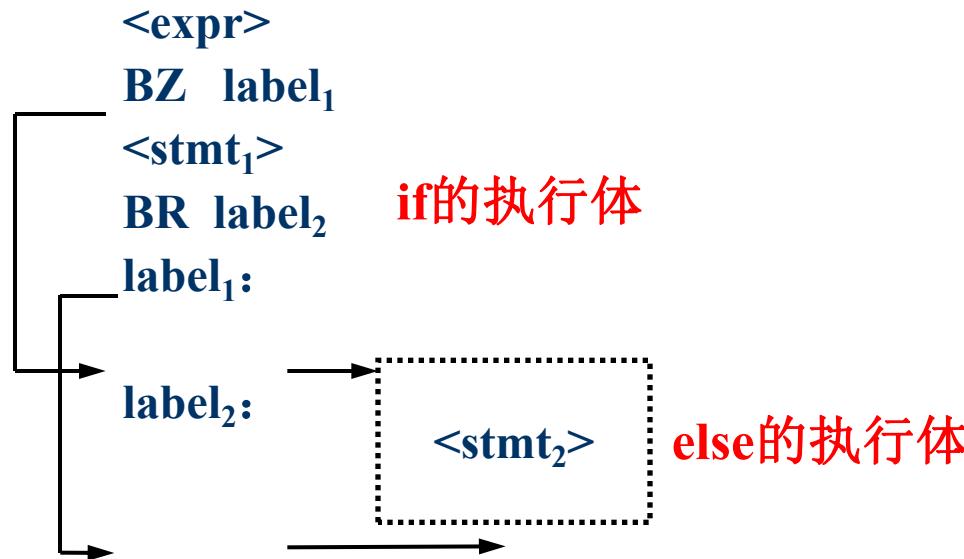
BZ: 二目操作符
若<expr>的计算结果为0 (false),
则产生一个到<label₁>的转移

BR: 一目操作符
产生一个到<label₂>的转移

有如下 if 语句： if <expr> then <stmt₁> else <stmt₂>

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

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。

其他语言结构也很容易将其翻译成波兰表示，
使用波兰表示的问题：优化不是十分方便。

波兰表达式 隐含了“栈操作”
能否将波兰表达式拆成一组“原子操作”？

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;

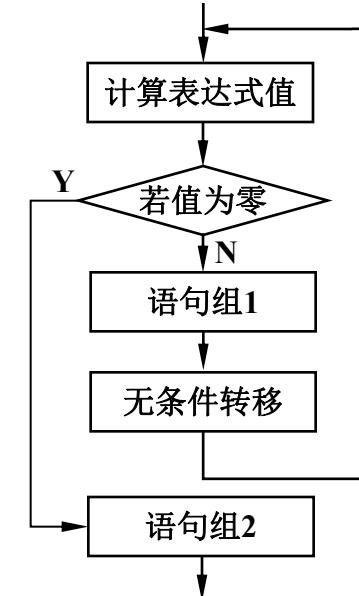
i:= 0;

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

Compiler

begin

k := 100;

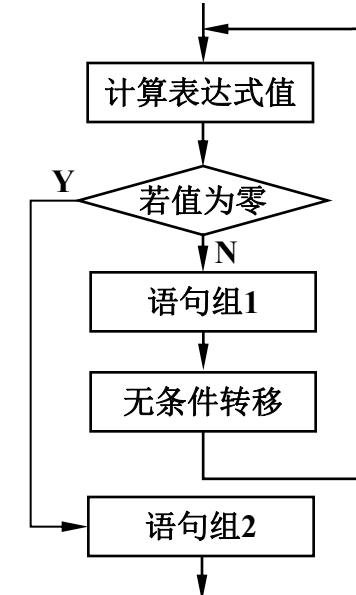
L: if k > i + j then

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

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

i:= 0;

end



1	2	3															
k	100	:=															

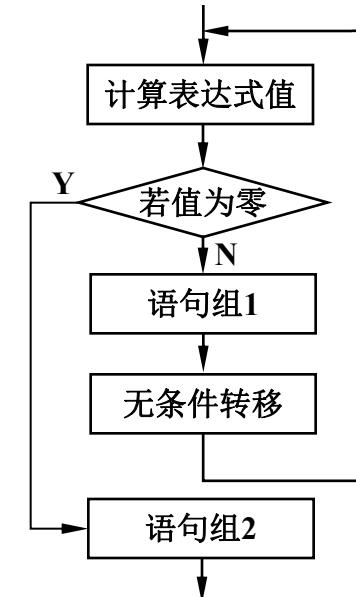
```

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;

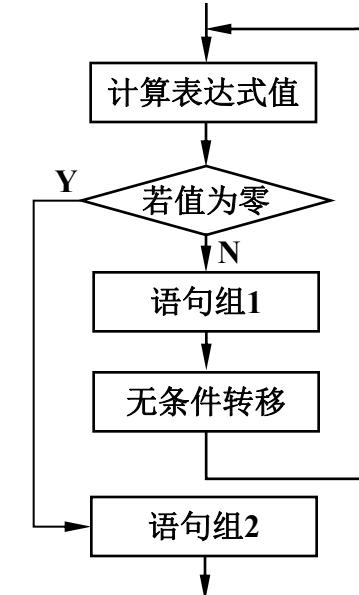
 i := 0;

end

L标号

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

↑
label1



```

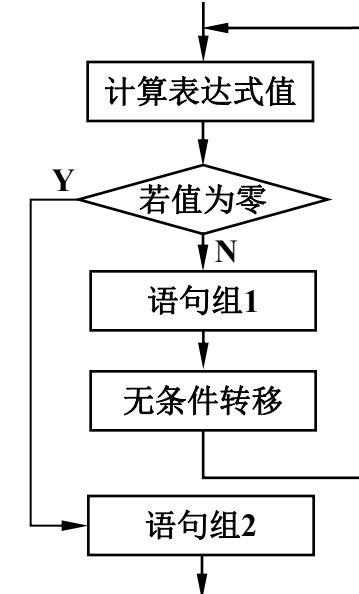
begin
    k := 100;
L: if k > i + j then
    begin k := k - 1; goto L; end
    else k := i ^ 2 - j ^ 2;
    i:= 0;
end

```

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

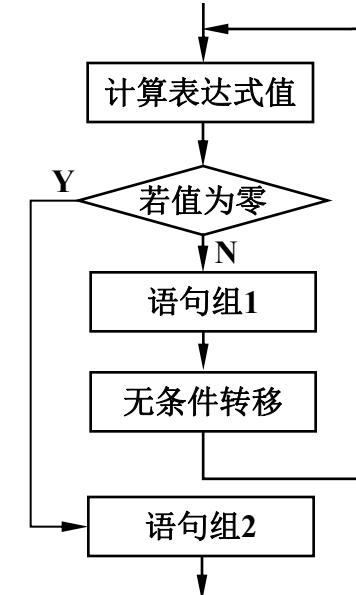
else k := i ^ 2 - j ^ 2;

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;

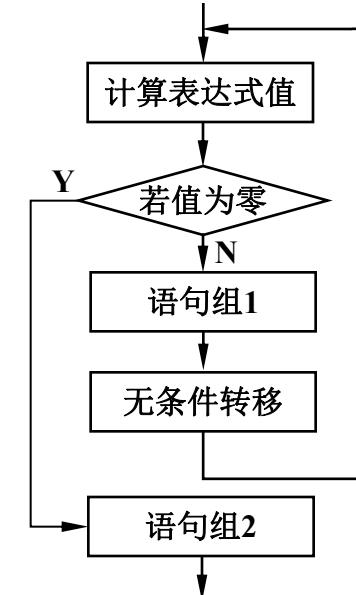
L: if k > i + j then

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	-	:=

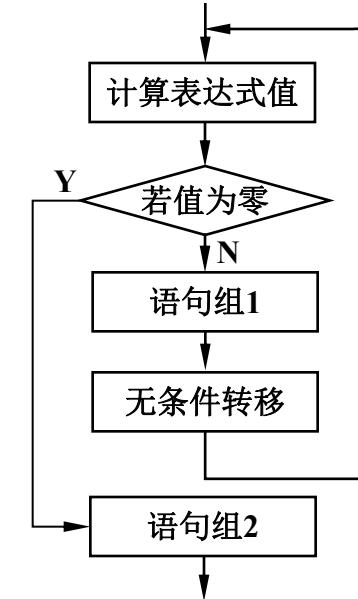
17															
jump															

Compiler

```

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	-	:=

17	18	19													
jump	?	jump													

↑
label2

Compiler

begin

k := 100;

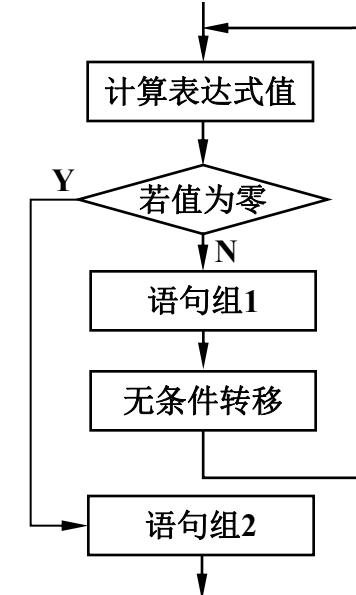
L: if k > i + j then

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	-	:=

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

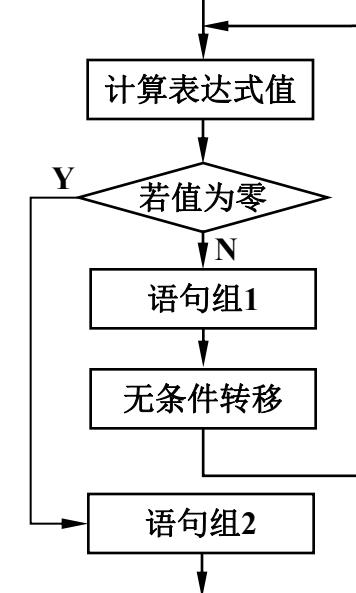
↑
label1

Compiler

```

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

```



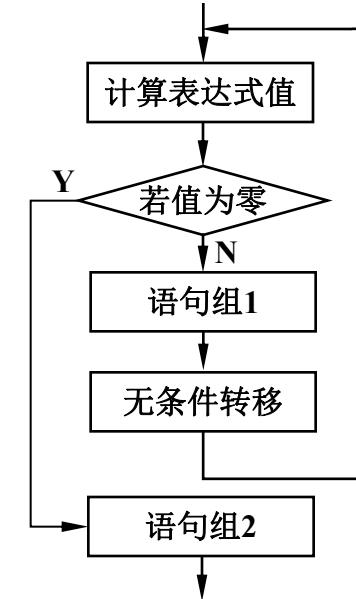
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	100	:=	k	i	j	+	>	20	jez	k	k	1	-	:=	4

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

```

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
jump	29	jump	k	i	2	^	j	2	^	-	:=	i	0	:=	...

7.2 N-元表示

在该表示中，每条指令由n个域组成，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;  
↓
```

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

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

三元式的执行次序用另一张表表示,这样在优化时，三元式可以不变，而仅仅改变其执行顺序表。

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（操作）方张表表示，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。

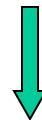
四元式表示

/三地址码(TAC)

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

结果：通常是由编译引入的临时变量，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

式中T1~T4为临时变量，
用四元式优化比较方便

四元式表示
(TAC)

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

```
int a;  
int b;  
int c;  
int d;  
  
a = b + c + d;  
b = a * a + b * b;
```

```
_t0 = b + c;  
a = _t0 + d;  
_t1 = a * a;  
_t2 = b * b;  
b = _t1 + _t2;
```

四元式表示
(TAC)

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

```
int x;  
int y;  
int z;  
  
if (x < y)  
    z = x;  
else  
    z = y;  
  
z = z * z;
```

```
_t0 = x < y;  
IfZ _t0 Goto _L0;  
z = x;  
Goto _L1;  
_L0:  
    z = y;  
_L1:  
    z = z * z;
```

四元式表示 (TAC)

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

```

int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;

```

```

_L0:
    t0 = x < y;
    IfZ t0 Goto _L1;
    x = x * 2;
    Goto _L0;
_L1:
    y = x;

```

四元式表示 (TAC)

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

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
    _L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
    _L1:
EndFunc;
```

四元式表示 (TAC)

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

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}
```

```
void main() {
    SimpleFunction(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
        _t0 = x * y;
        _t1 = _t0 * z;
        x = _t1;
    EndFunc;

main:
    BeginFunc 4;
        _t0 = 137;
        PushParam _t0;
        LCall _SimpleFn;
        PopParams 4;
    EndFunc;
```

7.3 中间代码的图结构表示

抽象语法树：

用树型图的方式表示中间代码

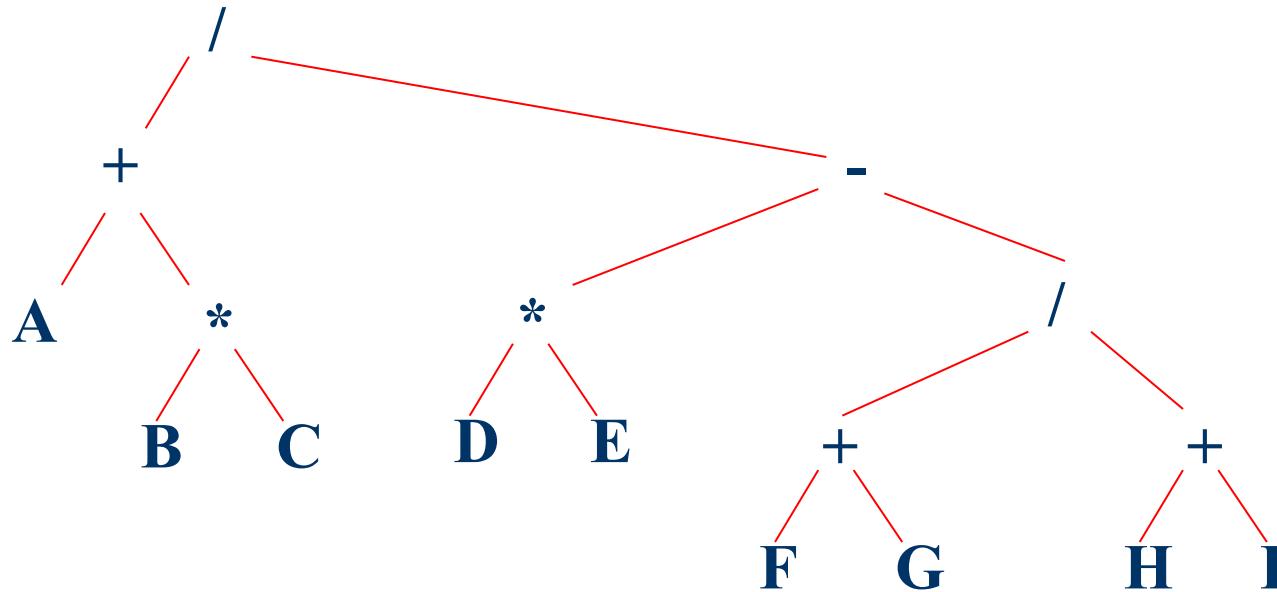
操作数出现在叶节点上，操作符出现在中间结点

DAG图：

Directed Acyclic Graphs 有向无环图

语法树的一种归约表达方式

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



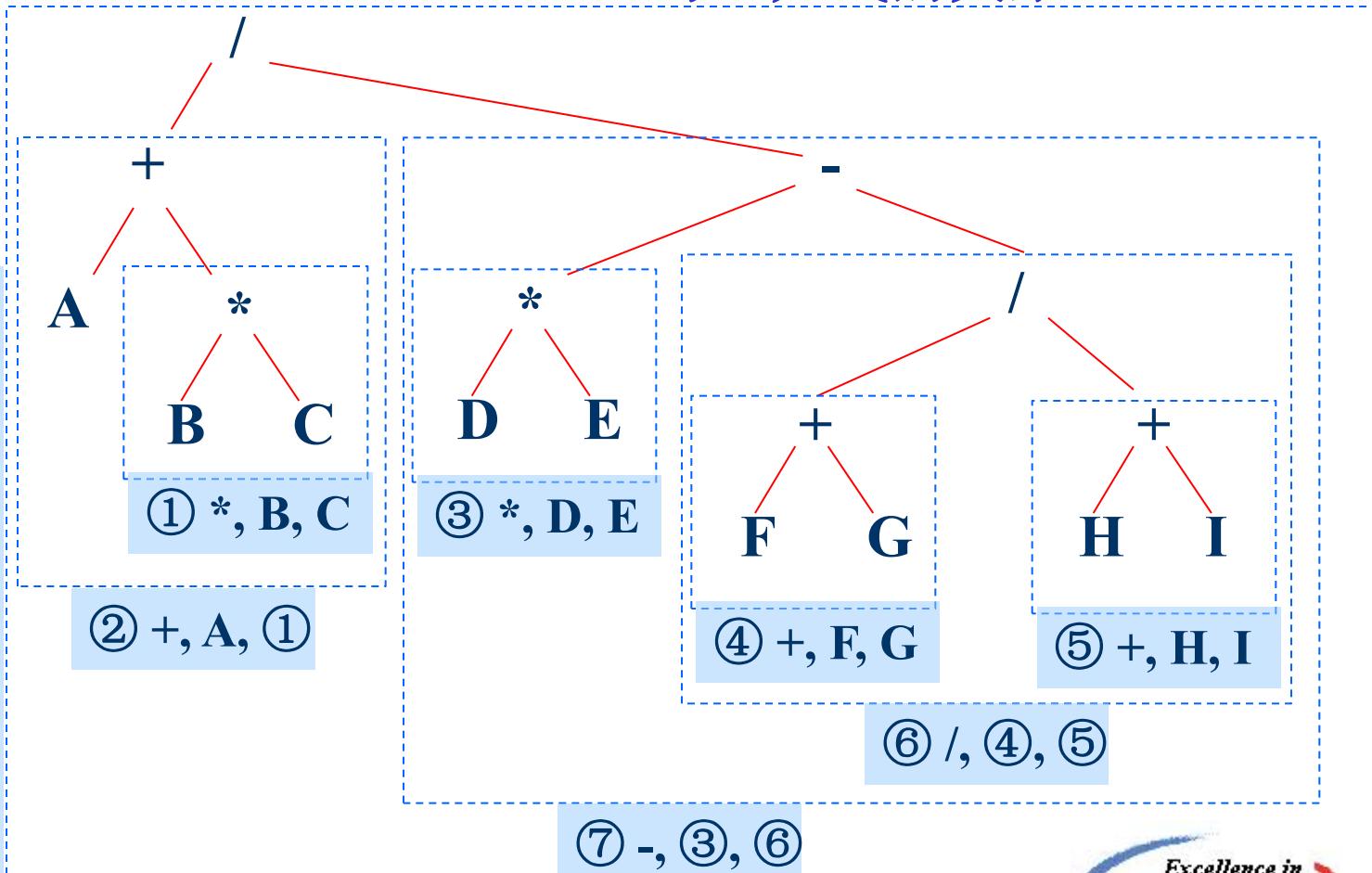
前序遍历（根左右）: **$+A*BC-*DE/+FG+HI$** (前缀表达)

后序遍历（左右根）: **$ABC*+DE*FG+HI+/-$** (后缀表达)

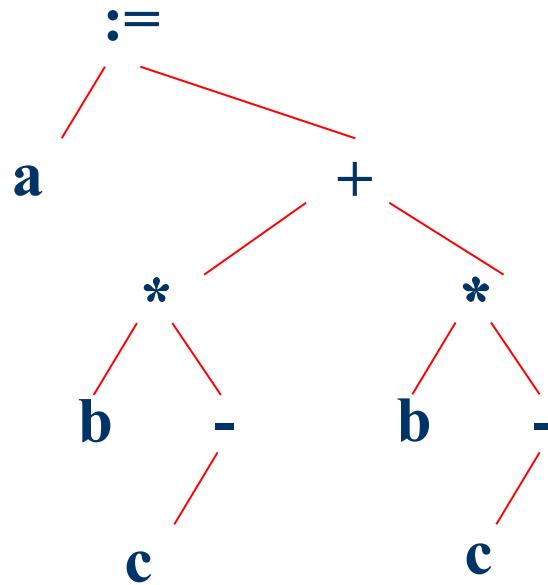
中序遍历（左根右）: **$A+B*C/D*E-F+G/H+I$**

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

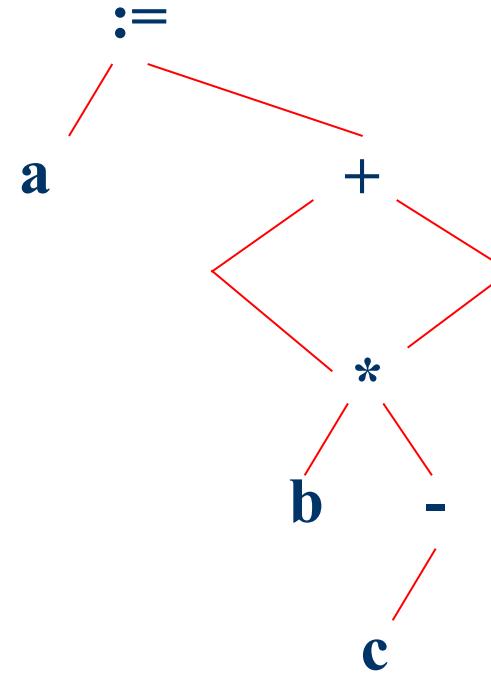
与三元式的关系



例：赋值语句： $a := b * (-c) + b * (-c)$



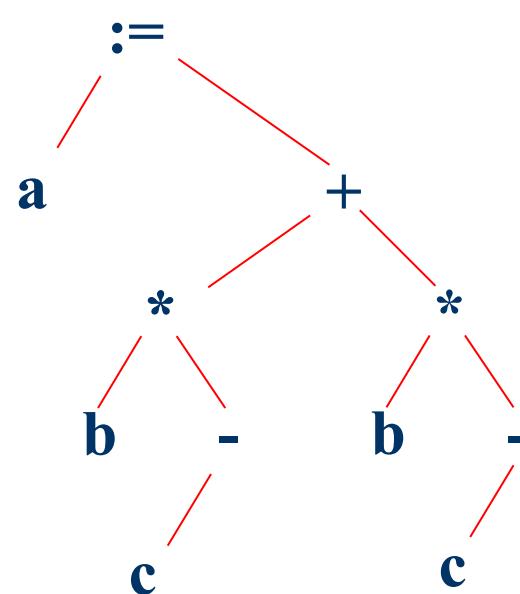
抽象语法树
(其中有重复部分)



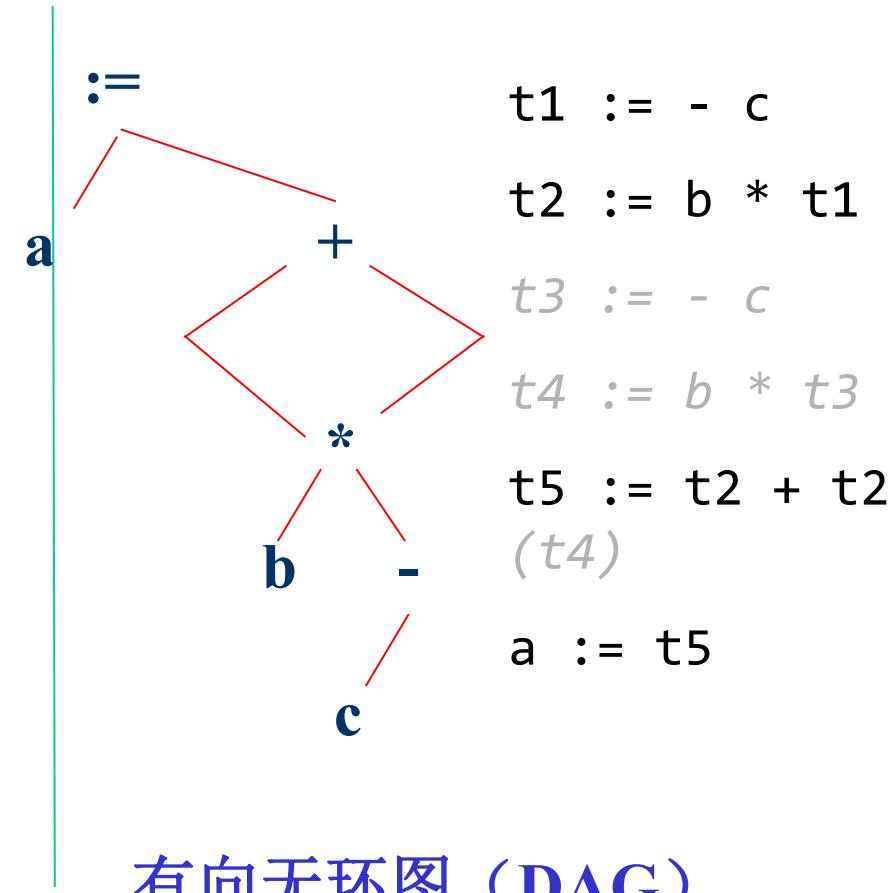
有向无环图 (DAG)

例：赋值语句： $a := b * (-c) + b * (-c)$ 对应的TAC

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



抽象语法树
(其中有重复部分)



有向无环图 (DAG)

7.3 抽象机代码

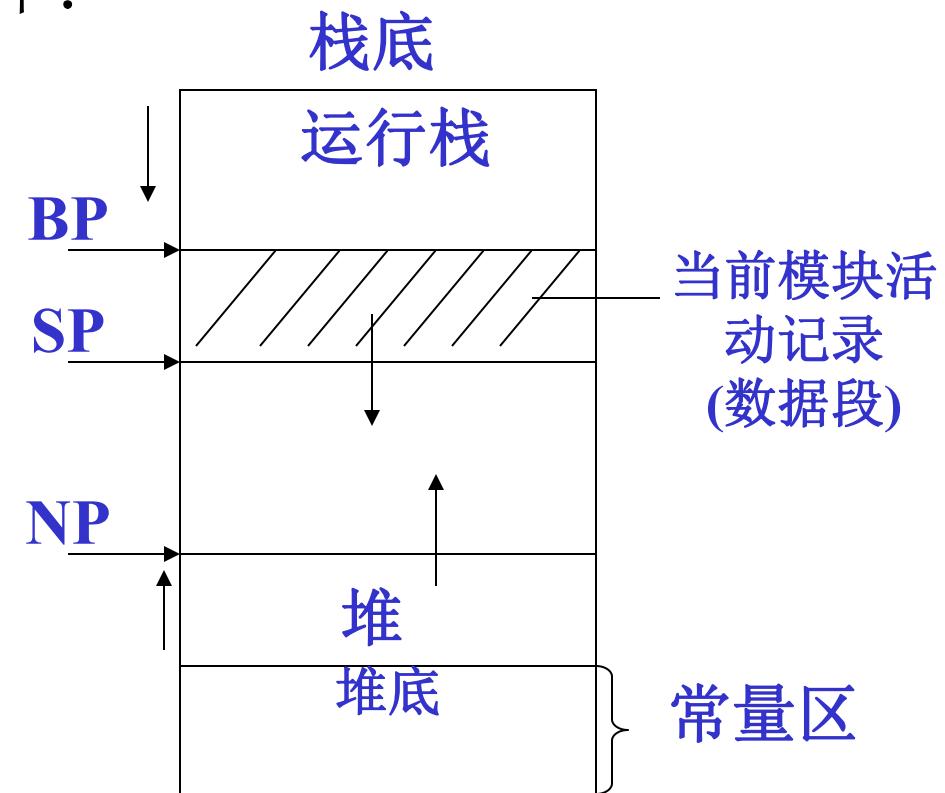
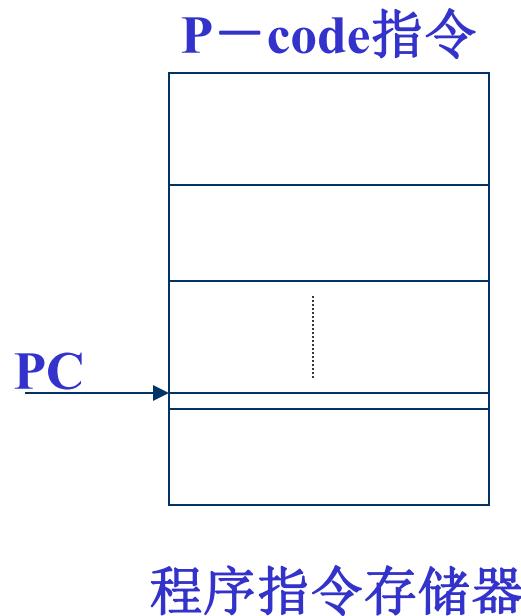
许多Pascal编译系统生成的中间代码是一种称为P—code的抽象代码。P—code的“P”即“Pseudo”。

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：

栈底：



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

P-Code指令集 (p184, 10.2)

40 ABI 取整数绝对值
41 ABR 取实数绝对值
28 ADI 整数加
29 ADR 实数加
53 DVI 整数除
54 DVR 实数除
42 NOT 布尔“非”
43 AND 布尔“与”
26 CHK Q 检查越界

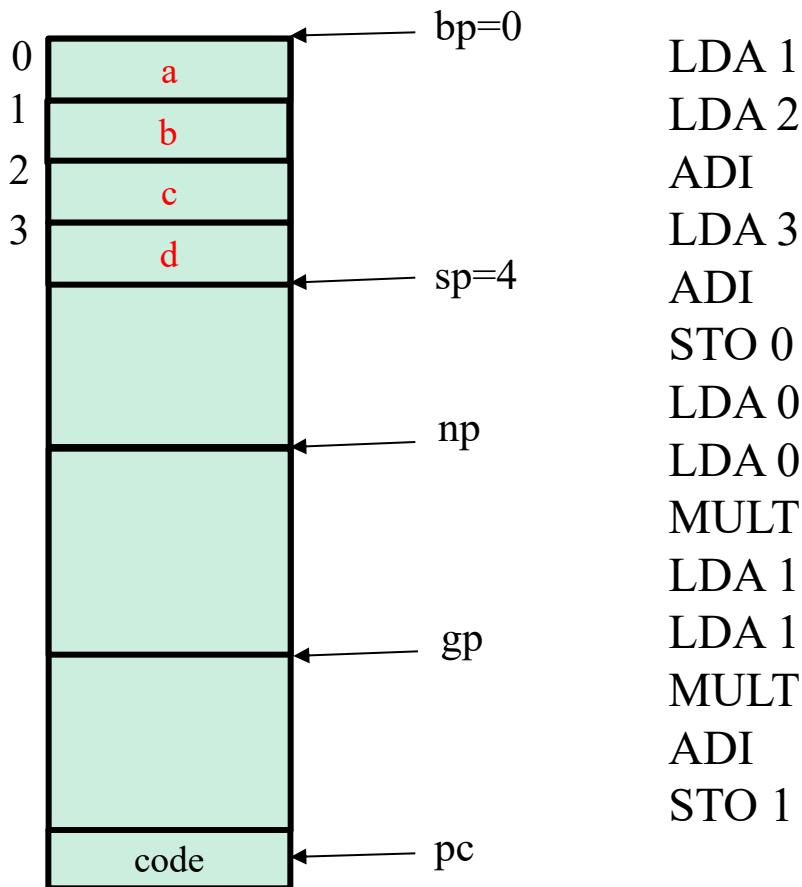
15 CSP Q 调用标准过程
12 CUP P, Q 调用用户过程
19 GEQ P, Q 大于等于
20 GRT P, Q 大于
48 INN 判定集合成员
48 INT 取交集
4 LDA P, Q 加载地址
7 LDC P, Q 加载常量
23 UJP Q 无条件跳转
24 FJP Q 为假时条件跳转

P-Code指令集

40 ABI 取整数绝对值
41 ABR 取实数绝对值
28 ADI 整数加
29 ADR 实数加
53 DVI 整数除
54 DVR 实数除
42 NOT 布尔“非”
43 AND 布尔“与”
26 CHK Q 检查越界

```
int a;  
int b;  
int c;  
int d;  
  
a = b + c + d;  
b = a * a + b * b;
```

15 CSP Q 调用标准过程
12 CUP P, Q 调用用户过程
19 GEQ P, Q 大于等于
20 GRT P, Q 大于
48 INN 判定集合成员
48 INT 取交集
4 LDA P, Q 加载地址
7 LDC P, Q 加载常量
23 UJP Q 无条件跳转
24 FJP O 为假时条件跳转

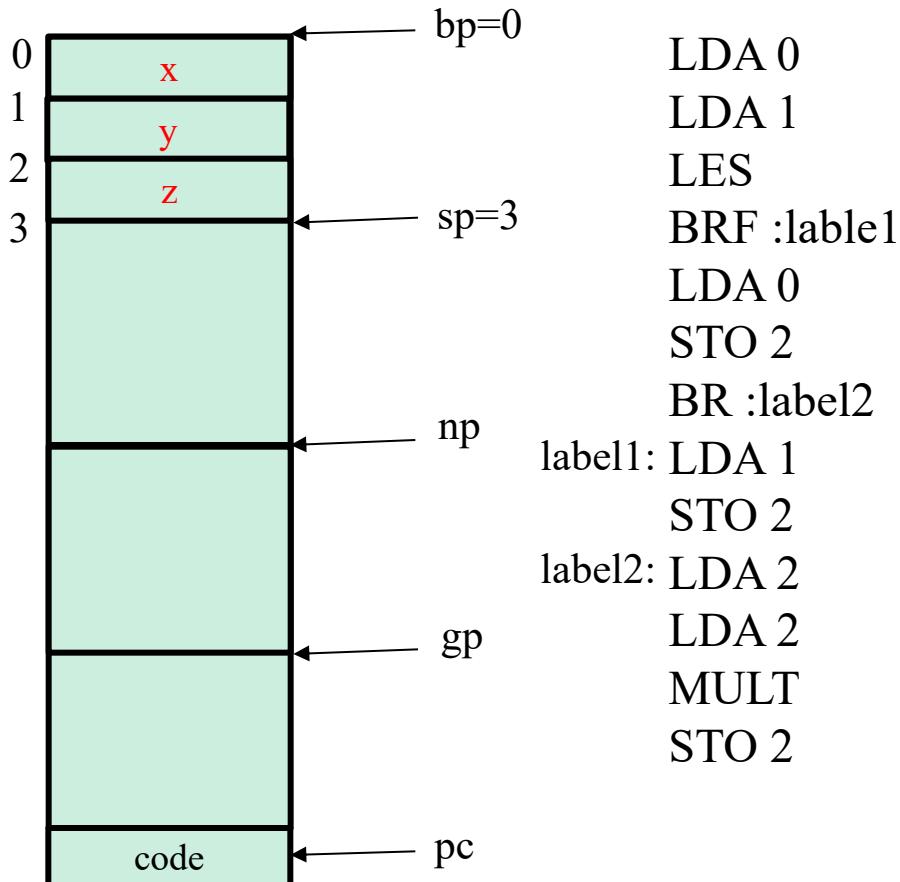


P-Code指令集

40 ABI 取整数绝对值
41 ABR 取实数绝对值
28 ADI 整数加
29 ADR 实数加
53 DVI 整数除
54 DVR 实数除
42 NOT 布尔“非”
43 AND 布尔“与”
26 CHK Q 检查越界

```
int x;  
int y;  
int z;  
  
if (x < y)  
    z = x;  
else  
    z = y;  
  
z = z * z;
```

15 CSP Q 调用标准过程
12 CUP P, Q 调用用户过程
19 GEQ P, Q 大于等于
20 GRT P, Q 大于
48 INN 判定集合成员
48 INT 取交集
4 LDA P, Q 加载地址
7 LDC P, Q 加载常量
23 UJP Q 无条件跳转
24 FJP Q 为假时条件跳转

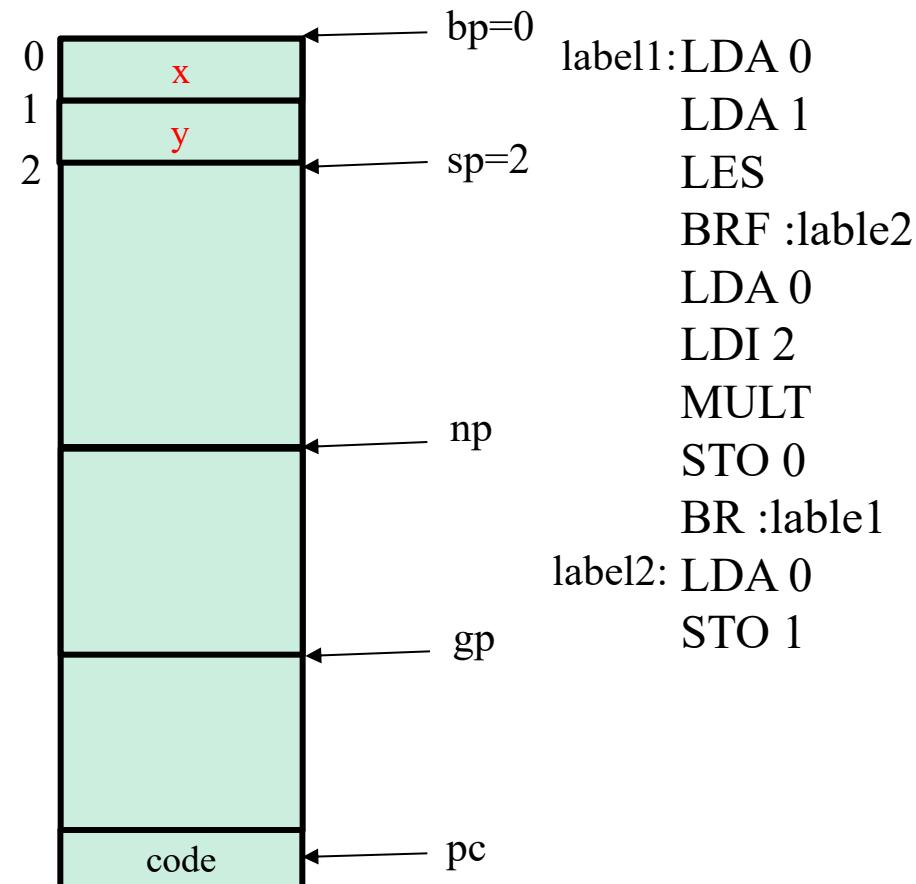


P-Code指令集

40 ABI 取整数绝对值
41 ABR 取实数绝对值
28 ADI 整数加
29 ADR 实数加
53 DVI 整数除
54 DVR 实数除
42 NOT 布尔“非”
43 AND 布尔“与”
26 CHK Q 检查越界

15 CSP Q 调用标准过程
12 CUP P, Q 调用用户过程
19 GEQ P, Q 大于等于
20 GRT P, Q 大于
48 INN 判定集合成员
48 INT 取交集
4 LDA P, Q 加载地址
7 LDC P, Q 加载常量
23 UJP Q 无条件跳转
24 FJP Q 为假时条件跳转

```
int x;  
int y;  
  
while (x < y) {  
    x = x * 2;  
}  
  
y = x;
```



P-Code指令集

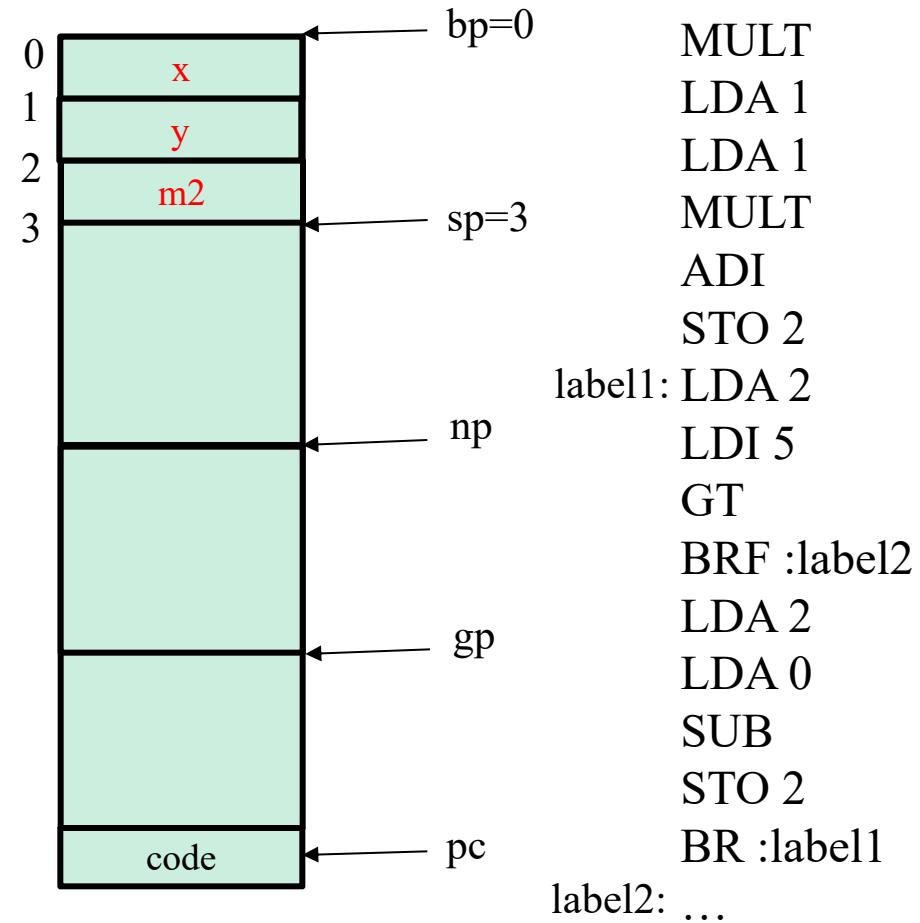
40 ABI 取整数绝对值
 41 ABR 取实数绝对值
 28 ADI 整数加
 29 ADR 实数加
 53 DVI 整数除
 54 DVR 实数除
 42 NOT 布尔“非”
 43 AND 布尔“与”
 26 CHK Q 检查越界

15 CSP Q 调用标准过程
 12 CUP P, Q 调用用户过程
 19 GEQ P, Q 大于等于
 20 GRT P, Q 大于
 48 INN 判定集合成员
 48 INT 取交集
 4 LDA P, Q 加载地址
 7 LDC P, Q 加载常量
 23 UJP Q 无条件跳转
 24 FJP Q 为假时条件跳转

```

void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
  
```



P-Code指令集

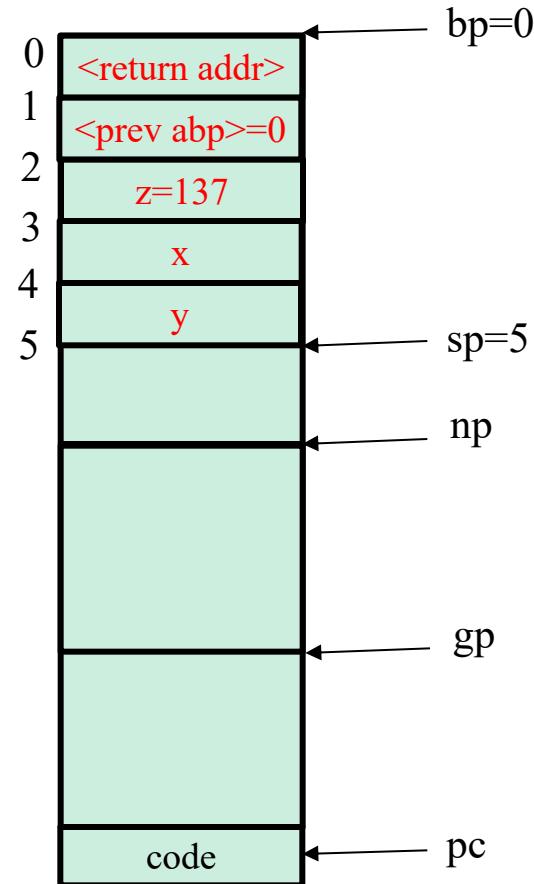
40 ABI 取整数绝对值
 41 ABR 取实数绝对值
 28 ADI 整数加
 29 ADR 实数加
 53 DVI 整数除
 54 DVR 实数除
 42 NOT 布尔“非”
 43 AND 布尔“与”
 26 CHK Q 检查越界

15 CSP Q 调用标准过程
 12 CUP P, Q 调用用户过程
 19 GEQ P, Q 大于等于
 20 GRT P, Q 大于
 48 INN 判定集合成员
 48 INT 取交集
 4 LDA P, Q 加载地址
 7 LDC P, Q 加载常量
 23 UJP Q 无条件跳转
 24 FJP Q 为假时条件跳转

```

void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
  
```



label1: ALLOCATE 3
 LDA 3
 LDA 4
 MULT
 LDA 2
 MULT
 STO 3
 LDA 1
 STO bp
 BR :label2
 ...
 LDI :label2
 LDI 0
 LDI 137
 BR :label1
 label2: ...

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。

7.5 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
```

很容易分析出y1是
可以优化掉的变量

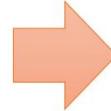
7.5 一种特殊的四元式表达方式：SSA

静态单赋值形式

Single Static Assignment

- 每个变量都只被赋值一次

x=10;
y=y+1;
x=y+x;
y=y+1;
z=y;



x0=10;
y0=y0+1;
x1=y0+x0;
y1=y0+1;
z0=y1;

练习：把以下程序转成静态 单赋值形式

x=10;

x0=10;

x+=y;

x1=x0+y;

if (x>10)

if (x1>10)

z=10;

z0=10;

else

else

z=20;

z1=20;

x+=z;

x2=x1+z?;

作业： P144 1, 2, 4

1. 将条件语句 if X = Y+2 then Z:=X else Z:=Y+1 转换成波兰后缀表示。
2. 将下面的语句：

$A := (B+C) \uparrow E + (B+C) * F$

转换成三元式、间接三元式和四元式序列。

3. 将下列语句转换成四元式序列：

① $A[1]:=B$

② $B:=A[1]$

4. 为第 2 题的语句构造抽象语法树。

小结：本节+上一节：为代码生成做“需求分析”

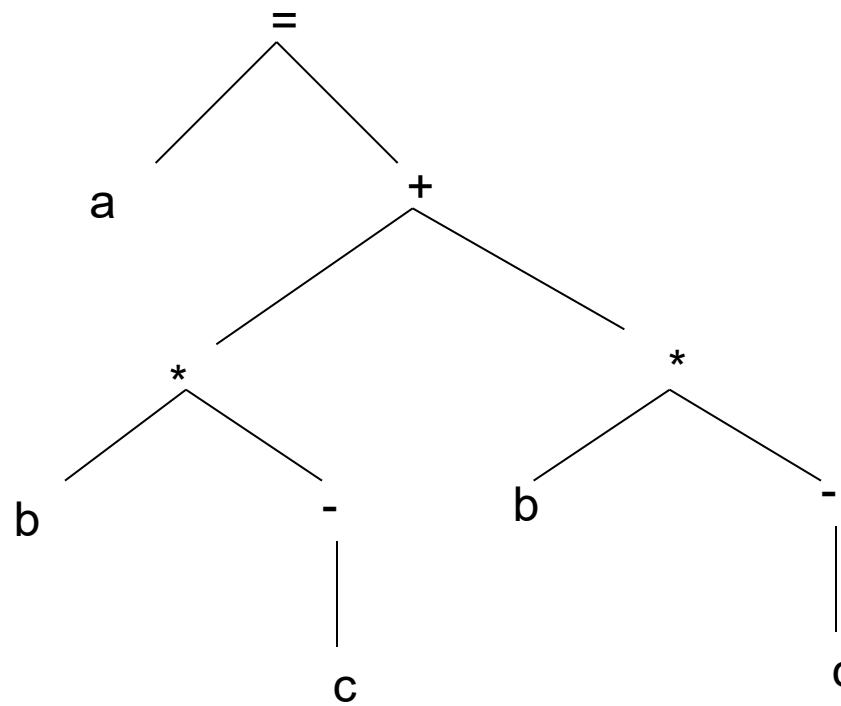
- 数据结构在内存里是如何表示的
 - 函数在内存中是如何表示的
 - 他们在内存中如何放置，如何管理
 - 中间代码如何表示
- 让编译器聪明一些：错误处理！

作业： p144 1,2,3,4

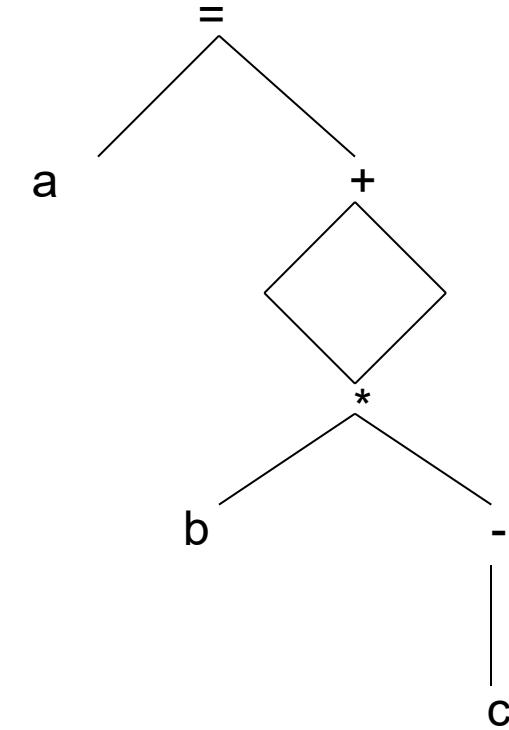
7.3 中间代码的图表示

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

- 赋值语句: $a := b * (-c) + b * (-c)$



语法树

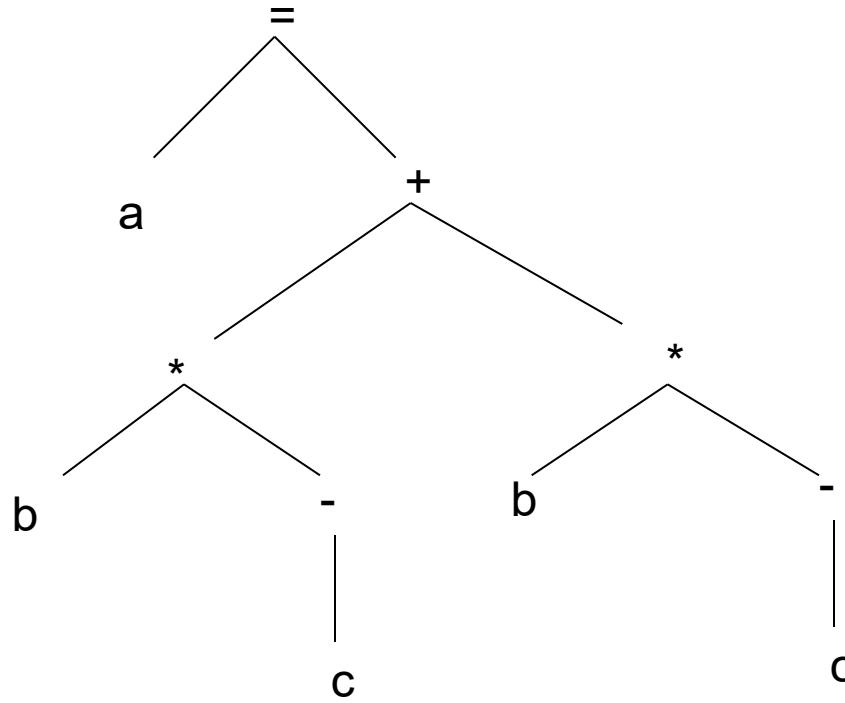


DAG图

中间代码：三地址码

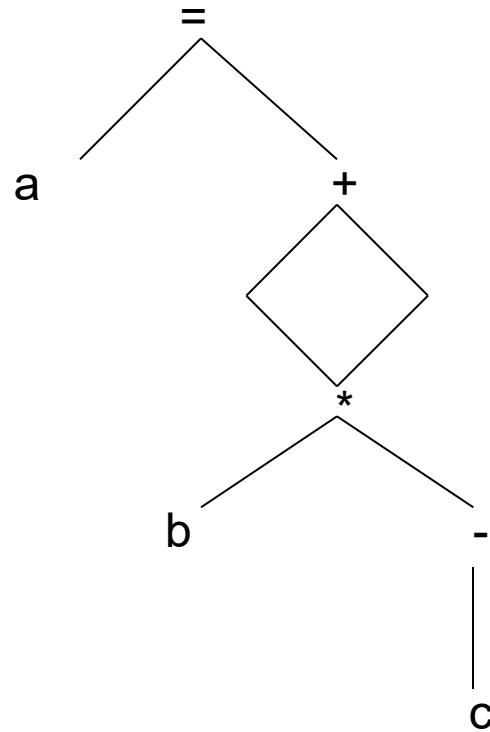
- 适合目标代码生成和优化的一种表达形式
- 三地址码是语法树或者DAG图的线性表示
- 树的中间结点由临时变量表示

三地址码与语法树的对应关系



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

三地址码与DAG图的对应关系



DAG图

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