



信息与软件工程学院

编译技术

主讲教师： 陈安龙

第5章 中间代码生成

- 使用中间代码的优势
- 后缀式
- 三元式
- 四元式
- 类型检查
- 控制流和布尔表达式的翻译

程序翻译的两种方式



使用中间语言的优点

- ① 便于进行与机器无关的代码优化工作
- ② 易于移植
- ③ 使编译程序的结构在逻辑上更为简单明确



5.1 中间语言

常用的中间语言：

- 后缀式（逆波兰表示）
- 图表示： DAG、抽象语法树
- 三地址代码
 - ① 三元式
 - ② 四元式
 - ③ 间接三元式

1) 逆波兰表示法（后缀表达式）

运算量在前,运算符在后的后缀式表示法.

如: 表达式

后缀式

$a+b$

$ab+$

$(a+b)*c$

$ab+c*$

$a*(b+c)$

$abc+*$

$(a+b)*(c+d)$

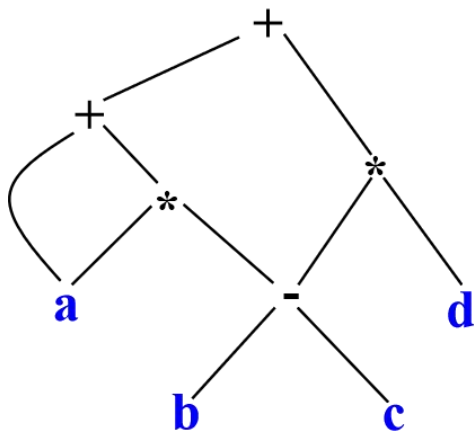
$ab+cd+*$

2) 图表示法

有向图无环(Directed Acyclic Graph, 简称DAG)

- ① 对表达式中的每个子表达式, DAG中都有一个结点
- ② 一个内部结点代表一个操作符, 它的孩子代表操作数
- ③ 在一个DAG中代表公共子表达式的结点具有多个父结点

例如: 表达式 $a+a*(b-c)+(b-c)*d$ 的DAG图表示



DAG

表达式文法语法树构造的语法制导定义

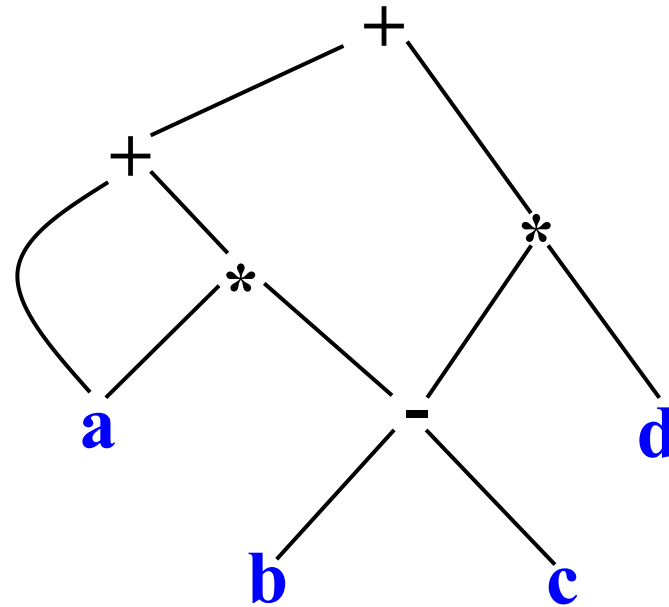
产生式

语义规则

$E \rightarrow E_1 + T$	$\{ E.node := \text{new Node}('+', E_1.node, T.node) \}$
$E \rightarrow E_1 - T$	$\{ E.node := \text{new Node}('-', E_1.node, T.node) \}$
$E \rightarrow T$	$\{ E.node := T.node \}$
$T \rightarrow T_1 * F$	$\{ T.node := \text{new Node}('*', T_1.node, F.node) \}$
$T \rightarrow (E)$	$\{ T.node := E.node \}$
$T \rightarrow \text{id}$	$\{ T.node := \text{new Leaf}(\text{id}, \text{id.entry}) \}$
$T \rightarrow \text{num}$	$\{ E.node := \text{new Leaf}(\text{num}, \text{num.val}) \}$

$a+a*(b-c)+(b-c)*d$ 的图表示法

- 1) $p_1 = \text{Leaf}(\text{id}, \text{entry}-a)$
- 2) $p_2 = \text{Leaf}(\text{id}, \text{entry}-a) = p_1$
- 3) $p_3 = \text{Leaf}(\text{id}, \text{entry}-b)$
- 4) $p_4 = \text{Leaf}(\text{id}, \text{entry}-c)$
- 5) $p_5 = \text{Node}('-', p_3, p_4)$
- 6) $p_6 = \text{Node}('*', p_1, p_5)$
- 7) $p_7 = \text{Node}('+', p_1, p_6)$
- 8) $p_8 = \text{Leaf}(\text{id}, \text{entry}-b) = p_3$
- 9) $p_9 = \text{Leaf}(\text{id}, \text{entry}-c) = p_4$
- 10) $p_{10} = \text{Node}('-', p_3, p_4) = p_5$
- 11) $p_{11} = \text{Leaf}(\text{id}, \text{entry}-d)$
- 12) $p_{12} = \text{Node}('*', p_5, p_{11})$
- 13) $p_{13} = \text{Node}('+', p_7, p_{12})$

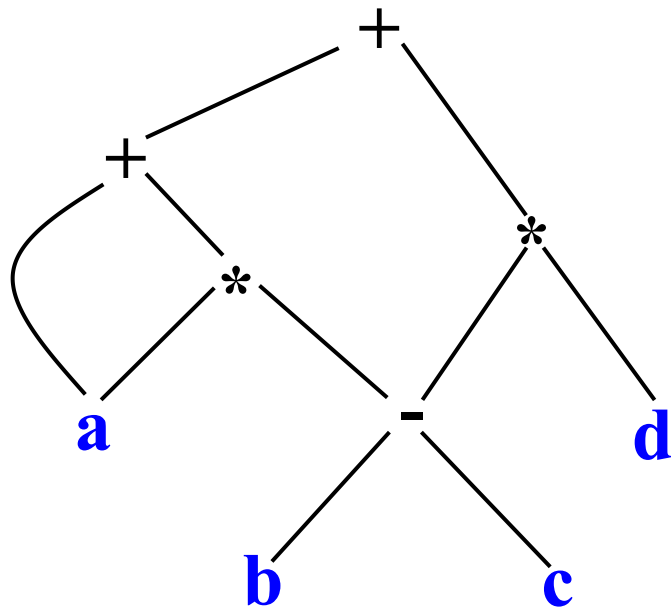


DAG

DAG图的存储结构

DAG图或语法树的结点一般存储在数组中，数组的每一行表示一个结点。

例如：表达式 $a+a*(b-c)+(b-c)*d$ 图的数组存储



DAG

0	id	a在符号表的地址	
1	id	b在符号表的地址	
2	id	c在符号表的地址	
3	-	1	2
4	*	0	3
5	+	0	4
6	id	d在符号表的地址	
7	*	3	6
8	+	5	7

值编码，类似指针或者数组下标。

三地址代码

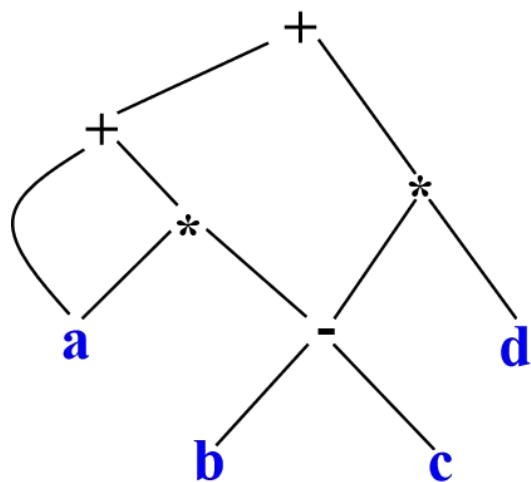
一般形式 $x := y \text{ op } z$

三地址代码可以看成是抽象语法树或DAG的一种线性表示

生成三地址代码时，临时变量的名字对应抽象语法树的内部结点

三地址代码

- 三地址代码（TAC）是大多数编译器中使用的中间代码。在三地址代码中，一条指令的右侧最多有一个运算符，不允许出现多个运算符组合的算术表达式。对于 $a + a * (b - c) + (b - c) * d$ 这样的源语言表达式可以翻译为如下三地址指令序列：



DAG

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

三地址代码

三地址语句的种类

(1) 赋值语句 $x = y \text{ op } z$, op 为二目算术算符或逻辑算符

(2) 赋值语句 $x = \text{op } y$, op 为一目算符;

如一目减 uminus 、逻辑非 not 、移位算符及转换算符

(3) 无条件转移语句 $\text{goto } L$

(4) 条件转移语句 $\text{if } x \text{ relop } y \text{ goto } L$, 关系运算符 $\text{relop}(<, =, >= \text{等等})$

(5) 复制语句 $x = y$

三地址语句的种类

(6) 过程调用语句 $\text{param } x$ 和 $\text{call } p, n$ 。过程调用语句 $p(x_1, x_2, \dots, x_n)$ 产生如下三地址代码:

$\text{param } x_1$

$\text{param } x_2$

...

$\text{param } x_n$

$\text{call } p, n$

过程返回语句 $\text{return } y$

三地址语句的种类

(7)索引赋值语句:

$$x = y[i]$$
$$x[i] = y$$

(8)地址和指针赋值语句

$$x = \&y$$
$$x = *y$$
$$*x = y$$

在设计中间代码形式时, 选择多少种算符需要平衡

语法制导翻译生成三地址代码

定义几个属性:

- (1) **E.place**表示存放E值的名字, 该名字已存放在符号表。
- (2) **E.code**表示对E求值的三地址语句序列。
- (3) **newtemp**是个函数, 对它的调用将产生一个新的临时变量。

简单赋值语句生成三地址代码 的语法制导翻译

产生式	语义规则
$S \rightarrow id = E$	$S.code = E.code gen(id.place = 'E.place)$
$E \rightarrow E_1 + E_2$	$E.place = newtemp;$ $E.code = E_1.code E_2.code gen(E.place = 'E_1.place' + 'E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place = newtemp;$ $E.code = E_1.code E_2.code gen(E.place = 'E_1.place' * 'E_2.place)$
$E \rightarrow -E_1$	$E.place = newtemp;$ $E.code = E_1.code gen(E.place = 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place = E_1.place;$ $E.code = E_1.code$
$E \rightarrow id$	$E.place = id.place;$ $E.code = ' '$

三地址代码的例子

```
do  
    i=i+1;  
while (a[i]<v);
```



三地址代码:

- (1) $t1 = i + 1$
- (2) $i = t1$
- (3) $t2 = t1 * 8$
- (4) $t3 = a[t2]$
- (5) if $t3 < v$ goto (1)

```
while (a > 10) do  
    if (b == 100)  
        while (a < 20) do  
            a = a + b - 1;
```



三地址代码

- (1) if $a \leq 10$ goto (8)
- (2) if $b \neq 100$ goto (7)
- (3) if $a \geq 20$ goto (7)
- (4) $t1 = a + b$
- (5) $a = t1 - 1$
- (6) goto (3)
- (7) goto (1)
- (8) ...

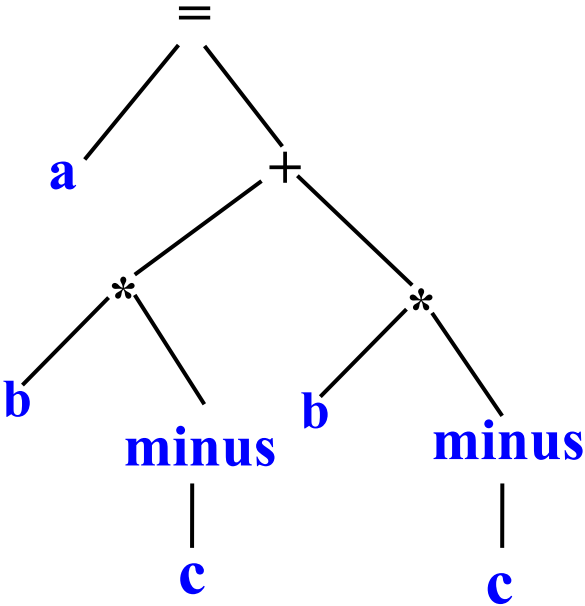
三地址代码的具体实现

- ① 四元式: $op, arg1, arg2, result$
- ② 三元式: $op, arg1, arg2$
- ③ 间接三元式: 间接码表+三元式表

四元式表示法

四元式表示通常表示为 $(op, arg1, arg2, result)$ ，其中 op 是操作符， $arg1$ 和 $arg2$ 是操作数， $result$ 是结果。

例如：给定表达式 $a = b * (-c) + b * (-c)$ ，将其转换为三地址代码和四元式表示：



语法树

三地址代码

- (0) $t1 = \text{minus } c$
- (1) $t2 = b * t1$
- (2) $t3 = \text{minus } c$
- (3) $t4 = b * t3$
- (4) $t5 = t2 + t4$
- (5) $a = t5$

四元式表示

<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
minus	c		t ₁
*	b	t ₁	t ₂
minus	c		t ₃
*	b	t ₃	t ₄
+	t ₂	t ₄	t ₅
=	t ₅		a

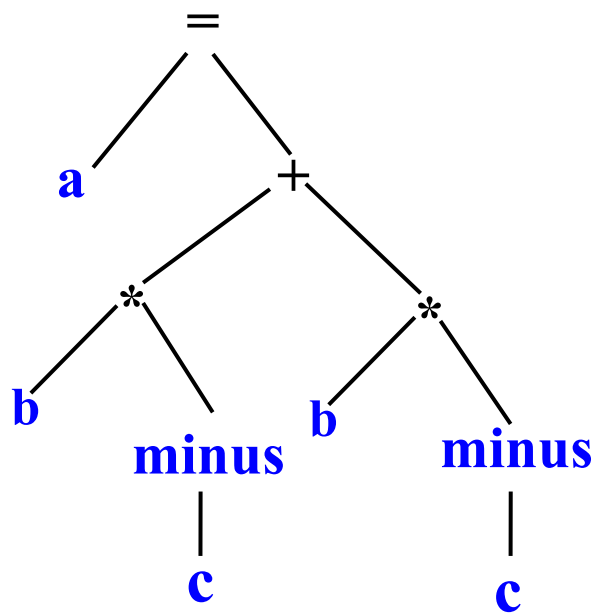
或者

- (0) $(-, c, , t1)$
- (1) $(*, b, t1, t2)$
- (2) $(-, c, , t3)$
- (3) $(*, b, t3, t4)$
- (4) $(+, t2, t4, t5)$
- (5) $(=, t5, , a)$

三元式表示法

三元式表示通常表示为 $(op, arg1, arg2)$ ，其中 op 是操作符， $arg1$ 和 $arg2$ 是操作数。结果用计算语句位置表示。

例如： 给定表达式 $a = b * (-c) + b * (-c)$ ，将其转换为三地址代码和三元式表示：



语法树

三地址代码

- (0) $t1 = \text{minus } c$
- (1) $t2 = b * t1$
- (2) $t3 = \text{minus } c$
- (3) $t4 = b * t3$
- (4) $t5 = t2 + t4$
- (5) $a = t5$

三元式

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

或者

- (0) $(-, c,)$
- (1) $(*, b, (0))$
- (2) $(-, c,)$
- (3) $(*, b, (2))$
- (4) $(+, (1), (3))$
- (5) $(=, a, (4))$

间接三元式

间接三元式是一种中间代码表示形式，它使用一个间接寻址表来存储操作数和结果。
这种表示形式可以方便地进行各种优化，如公共子表达式消除、复制传播等。

例如：给定的表达式 $a = b * (-c) + b * (-c)$ 的间接三元式表示如下：

首先，需要间接寻址表存储三元式中的指令地址：

instuction		三元式		
		op	arg ₁	arg ₂
35	(0)			
36	(1)			
37	(2)			
38	(3)			
39	(4)			
40	(5)			
	...			

0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

静态单赋值形式

- 一种便于代码优化的中间表示
- 和三地址代码的主要区别:所有赋值指令都是对不同名字的变量的赋值

三地址代码

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

静态单赋值形式

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$

$$q_2 = p_3 + q_1$$

常用语句的翻译

1、声明语句的翻译

- 作用:

声明语句用于对程序中规定范围内使用的**各类变量、常数、过程**进行说明。

- 编译要做的工作

- ① 在符号表中建立相应的表项，**填写有关的信息**。如**类型、嵌套深度、相对地址**等。

- ② **相对地址**:**相对静态数据区基址或活动记录中局部数据区基址**的一个偏移值。

声明语句的翻译

■ 变量声明语句

① 变量的类型可以确定变量需要的内存

即类型的宽度, 可变大小的数据结构只需要考虑指针

② 函数的局部变量总是分配在连续的区间;

因此给每个变量分配一个相对于这个区间开始处的相对地址

③ 变量的类型信息保存在符号表中;

C语言变量声明的文法

$P \rightarrow D$

$D \rightarrow T B ; D \mid \varepsilon$

$T \rightarrow \text{int} \mid \text{float} \mid \text{struct} \{ ' D ' \};$

$B \rightarrow \text{id } C$

$C \rightarrow \varepsilon \mid [\text{num}] C$

D 生成一系列声明;

T 生成不同的类型;

B 生成简单标识符和数组标识符;

C 表示数组分量, 生成[num]序列;

注意: struct是结构体类型声明。

声明序列的语法制导翻译

- 在处理一个过程/函数时，局部变量应该放到单独的符号表中去；
- 这些变量的内存布局独立
 - 相对地址从0开始；
 - 假设变量的放置和声明的顺序相同；
 - 变量offset记录当前可用的相对地址；
 - 每“分配”一个变量，offset的值增加相应的值
- `addtype(id.entry, C.type, offset)`;
- 在当前符号表(位于栈顶)中创建符号表条目，记录标识符的类型，偏移量

声明变量的语法制导翻译

- offset 是定义变量的偏移地址;
- T、B和C有综合属性type, width; type 为数据类型, width为变量占用内存数;
- D有综合属性width, 表示声明变量占用的总内存数;
- 全局变量t和w用于将类型和宽度信息从T传递到 $C \rightarrow \epsilon$, 相当于C的继承属性;
- struct为结构体数据类型, 该类型的变量占用的内存数为结构体元素占用的内存总数;
- addtype函数声明的变量加入符号表。

$P \rightarrow \{\text{offset} = 0;\} D$

$D \rightarrow T \{t=T.type, w=T.width;\}$

$B ;$

$D_1 \{D.width=D.width+D_1.width;\}$

$D \rightarrow \epsilon \{D.width=0;\}$

$T \rightarrow \text{int} \{T.type=integer; T.width=4;\}$

$T \rightarrow \text{float} \{T.type=float; T.width=4;\}$

$T \rightarrow \text{struct} \{'\} D '\}; \{T.type=struct; T.width=D.width\}$

$B \rightarrow \text{id } C \{\text{addtype}(\text{id.entry}, C.type, \text{offset});$

$B.type=C.type; B.width=C.width;$

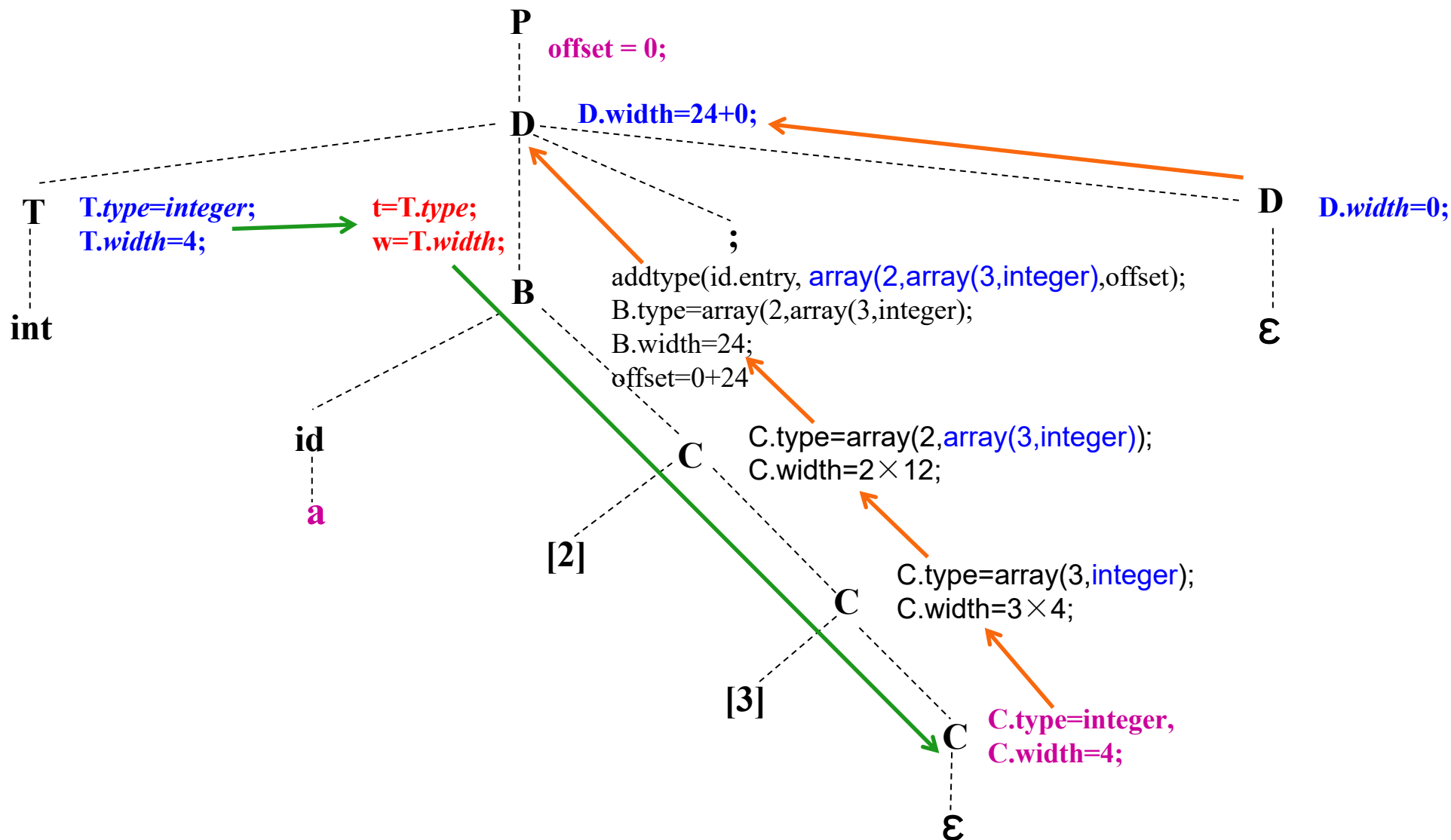
$\text{offset}=\text{offset}+C.width\}$

$C \rightarrow \epsilon \{C.type=t, C.width=w;\}$

$C \rightarrow [\text{num}] C_1 \{ C.type=\text{array}(\text{num.value}, C_1.type);$

$C.width=\text{num.value} \times C_1.width; \}$

示例：假如C语言声明二维数组： `int a[2][3]`



表达式代码的语法制导

- 将表达式翻译成三地址指令序列
- 表达式的语法制导
 - 属性code表示代码
 - addr表示存放表达式结果的地址
(临时变量)
 - new Temp()可以生成一个临时变量
 - gen(...)生成一个指令

产生式	语义规则
$S \rightarrow id = E ;$	$S.code = E.code \parallel$ $gen(top.get(id.lexeme) '=' E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = new Temp()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr '=' E_1.addr '+' E_2.addr)$
$E \rightarrow E_1 - E_2$	$E.addr = new Temp()$ $E.code = E_1.code \parallel$ $gen(E.addr '=' 'minus' E_1.addr)$
$E \rightarrow (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$E \rightarrow id$	$E.addr = top.get(id.lexeme)$ $E.code = ''$

增量式翻译方案

- 不再需要code属性，因为采用至下而上分析，gen连续调用已生成一系列指令，如：
 $E \rightarrow E_1 + E_2$ 生成E的指令时， E_1 和 E_2 指令已经生成；
- `top.get(...)`从栈顶符号表开始，逐个向下寻找id的信息；
- `gen`产生相应的指令代码；
- `temp()`生成临时变量。

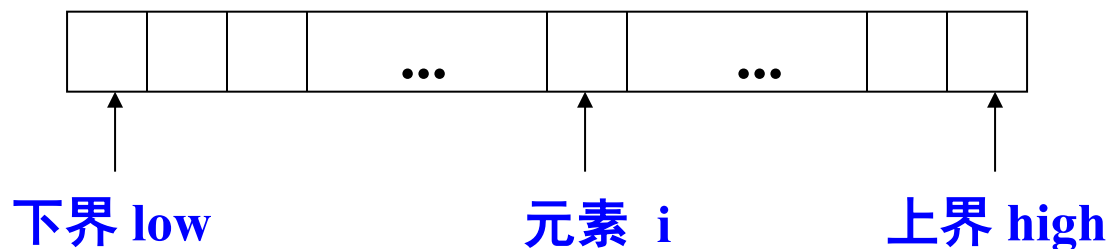
$S \rightarrow \text{id} = E ;$	$\{ \text{gen}(\text{top.get}(\text{id.lexeme}) \neq E.addr); \}$
$E \rightarrow E_1 + E_2$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr \neq E_1.addr \neq E_2.addr); \}$
$ - E_1$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr \neq \text{'minus'} E_1.addr); \}$
$ (E_1)$	$\{ E.addr = E_1.addr; \}$
$ \text{id}$	$\{ E.addr = \text{top.get}(\text{id.lexeme}); \}$

赋值语句中涉及数组元素的情况

■ 数组元素地址分配

数组元素存储在一个连续的存储块中，根据数组元素的下标查找每个元素。

■ 一维数组A



基址: base

域宽: w

元素个数: $high - low + 1$

数组元素A[i]的位置:

$$base + (i - low) \times w = i \times w + base - low \times w$$

常量，在编译
时可以确定

二维数组

$a_{1,1}$	$a_{1,2}$	$a_{1,j}$	a_{1,n_2}
$a_{2,1}$	$a_{2,2}$	$a_{2,j}$	a_{2,n_2}
...			
$a_{i,1}$	$a_{i,2}$	$a_{i,j}$	a_{i,n_2}
...			
$a_{n_1,1}$	$a_{n_1,2}$	$a_{n_1,j}$	a_{n_1,n_2}

存储方式:

按行存放
Pascal,C采用

↑	A[1,1]
第一行	A[1,2]
↓	A[1,3]
↑	A[2,1]
第二行	A[2,2]
↓	A[2,3]

按列存放
Fortran采用

A[1,1]	↑
A[2,1]	第一列
↓	
A[1,2]	↑
A[2,2]	第二列
↓	
A[1,3]	↑
A[2,3]	第三列
↓	

每维的下界: low_1 、 low_2 ,每维的上界: $high_1$ 、 $high_2$;

每维的长度: $n_1=high_1-low_1+1$, $n_2=high_2-low_2+1$; 域宽: w , 基址: $base$

数组元素A[i, j]的位置:

$$base + ((i-low_1) \times n_2 + (j-low_2)) \times w$$
$$= (i \times n_2 + j) \times w + base - (low_1 \times n_2 + low_2) \times w$$

按列存放:

$$(j \times n_1 + i) \times w + base - (low_2 \times n_1 + low_1) \times w$$

处理数组引用的翻译方案

```
S → id = E ; { gen( top.get(id.lexeme) != E.addr); }  
    | L = E ; { gen(L.addr.base '[' L.addr ']' != E.addr); }  
E → E1 + E2 { E.addr = new Temp();  
                gen(E.addr != E1.addr '+' E2.addr); }  
    | id        { E.addr = top.get(id.lexeme); }  
    | L         { E.addr = new Temp();  
                gen(E.addr != L.array.base '[' L.addr ']'); }  
L → id [ E ] { L.array = top.get(id.lexeme);  
              L.type = L.array.type.elem;  
              L.addr = new Temp();  
              gen(L.addr != E.addr '*' L.type.width); }  
    | L1 [ E ] { L.array = L1.array;  
                L.type = L1.type.elem;  
                t = new Temp();  
                L.addr = new Temp();  
                gen(t != E.addr '*' L.type.width); }  
                gen(L.addr != L1.addr '+' t); }
```

- 为有数组引用的表达式生成三地址代码；
- 将表达式赋值给数组：S→L=E；
- 表达式含有数组因子：E→E₁+E₂|L
- 数组名及下标生成：L→L[E] | id[E]

数组引用语义文法(续)

- 数组名及下标生成: $L \rightarrow L[E] \mid \text{id}[E]$
- $L \rightarrow \text{id}[E]$ 生成一维数组
- $L \rightarrow L[E]$ 生成多维数组
- $L.\text{addr}$ 临时变量, 用于在计算过程中累计数组引用偏移量;
- $L.\text{array}$ 是指向符号表的指针;
- $L.\text{type}$ 是 L 生成子数组类型;
- $L.\text{type.elem}$ 是数组元素的类型;
- $E.\text{addr}$ 存放表达式值, 这里是数组元素序号;

```

$$\begin{aligned} L \rightarrow \text{id} [ E ] & \quad \{ L.\text{array} = \text{top.get}(\text{id.lexeme}); \\ & \quad L.\text{type} = L.\text{array.type.elem}; \\ & \quad L.\text{addr} = \text{new Temp}(); \\ & \quad \text{gen}(L.\text{addr} '=' E.\text{addr} '*' L.\text{type.width}); \} \\ \\ | \quad L_1 [ E ] & \quad \{ L.\text{array} = L_1.\text{array}; \\ & \quad L.\text{type} = L_1.\text{type.elem}; \\ & \quad t = \text{new Temp}(); \\ & \quad L.\text{addr} = \text{new Temp}(); \\ & \quad \text{gen}(t '=' E.\text{addr} '*' L.\text{type.width}); \\ & \quad \text{gen}(L.\text{addr} '=' L_1.\text{addr} '+' t); \} \end{aligned}$$

```

数组元素作为因子

- 下面的语义文法片段是计算表达值，表达式中引用数组；
- **L**是代表数组的非终结符；
- **L.array.base**是数组基地址，被用于确定数组引用的左值；
- 使用三地址指令 $x=a[i]$ 。

$E \rightarrow E_1 + E_2$	$\{ E.addr = \text{new Temp}();$ $\quad gen(E.addr '=' E_1.addr '+' E_2.addr); \}$
$ \quad id$	$\{ E.addr = top.get(id.lexeme); \}$
$ \quad L$	$\{ E.addr = \text{new Temp}();$ $\quad gen(E.addr '=' L.array.base '[' L.addr ']'); \}$

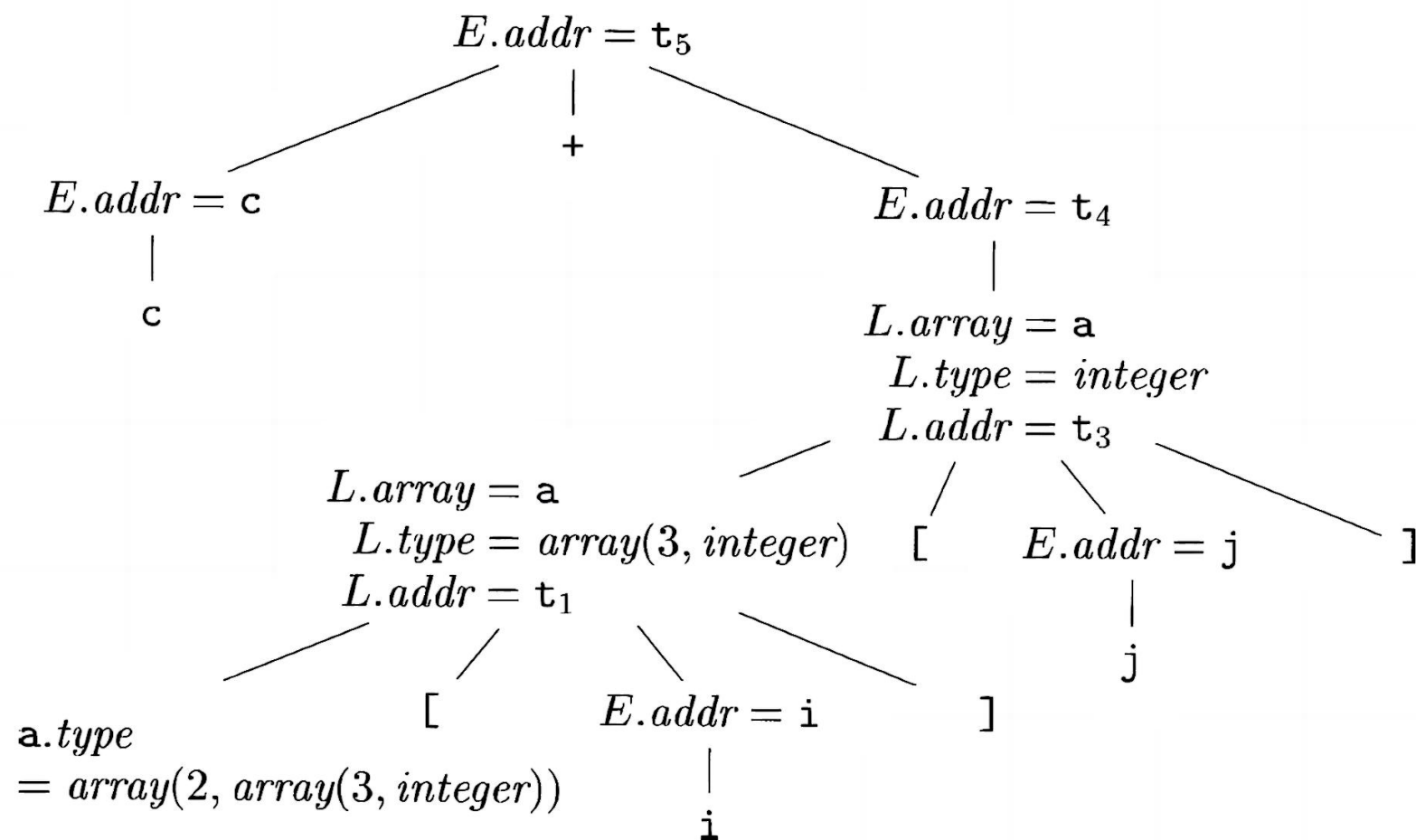
数组元素作为赋值左部

- 下面是表达式赋值的语义文法;
- 将表达式值赋给标识符: $S \rightarrow \text{id} = E;$
- 将表达式值赋给数组: $S \rightarrow L = E;$
- 使用三地址指令 $a[i] = x;$

$\begin{array}{l} S \rightarrow \text{id} = E ; \quad \{ \text{gen}(\text{top.get}(\text{id.lexeme}) \neq E.addr); \} \\ \quad \quad L = E ; \quad \{ \text{gen}(L.array.base '[' L.addr ']' \neq E.addr); \} \end{array}$

带数组元素的表达式例子

- 表达式: $c + a[i][j]$



t_1	$=$	i	$*$	12	
t_2	$=$	j	$*$	4	
t_3	$=$	t_1	$+$	t_2	
t_4	$=$	a	$[$	t_3	$]$
t_5	$=$	c	$+$	t_4	

布尔表达式和控制流语句

■ 布尔表达式

- ① 与一般的算术表达式类似
- ② 运算符是 or、and、not等
- ③ 运算量是布尔常量(true、false)或关系表达式(其值也是布尔量)

■ 布尔表达式的作用

- ① 计算逻辑值
- ② 用作控制流语句中的条件表达式

■ 产生布尔表达式的文法

$E \rightarrow E \text{ or } E$
 $E \rightarrow E \text{ and } E$
 $E \rightarrow \text{not } E$
 $E \rightarrow (E)$
 $E \rightarrow \text{id relop id}$
 $E \rightarrow \text{true}$
 $E \rightarrow \text{false}$

翻译布尔表达式的方法

- 翻译布尔表达式是为了得到表达式的运算结果
- 布尔表达式的运算结果是 true 或 false
- 如何体现运算结果？

翻译布尔表达式的方法

■ 布尔表达式的值的表示方法

① 数值表示法：

- 1 — true 0 — false
- 非0 — true 0 — false

② 控制流表示法：

利用控制流到达程序中的位置来表示true或false

■ 布尔表达式的翻译方法

① 数值方法

② 控制流方法

布尔表达式的数值翻译方法

- 把true、false数值化(true 为1, false 为0);
- 布尔表达式的求值类似于算术表达式的求值

例如:

a or b and not c

①

②

③

- 关系表达式 $a < b$ 等价于:
if $a < b$
then 1
else 0

- 三地址代码

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

- $a < b$ 的三地址代码:
100: if $a < b$ goto 103
101: $t := 0$
102: goto 104
103: $t := 1$
104:

布尔表达式的数值翻译方法

■ 属性、函数及变量说明

- ① **属性E.place**: 存放布尔表达式E的值的临时变量(临时变量在符号表中的入口位置)
- ② **函数gen**: 产生并输出一条三地址语句
- ③ **变量nextstat**: 输出序列中下一条三地址语句的位置, 函数gen执行之后, **nextstat自动加1**。

布尔表达式的数值翻译方案

$E \rightarrow E_1 \text{ or } E_2$ { E.place:=newtemp; gen(E.place '=' E₁.place '**or**' E₂.place) }

$E \rightarrow E_1 \text{ and } E_2$ { E.place:=newtemp; gen(E.place '=' E₁.place '**and**' E₂.place) }

$E \rightarrow \text{not } E_1$ { E.place:=newtemp; gen(E.place '=' '**not**' E₁.place) }

$E \rightarrow (E_1)$ { E.place:= E₁.place) }

$E \rightarrow \text{id}_1 \text{ relop id}_2$ { E.place:=newtemp;
gen('**if**' id₁.place relop.op id₂.place '**goto**' nextstat+3);
gen(E.place '=' '0');
gen('**goto**' nextstat+2);
gen(E.place '=' '1') }

$E \rightarrow \text{true}$ { E.place:=newtemp; gen(E.place '=' '1') }

$E \rightarrow \text{false}$ { E.place:=newtemp; gen(E.place '=' '0') }

示例: $a < b$ or $c < d$ and $e < f$

$E \rightarrow E \text{ or } E$

$E \rightarrow E \text{ and } E$

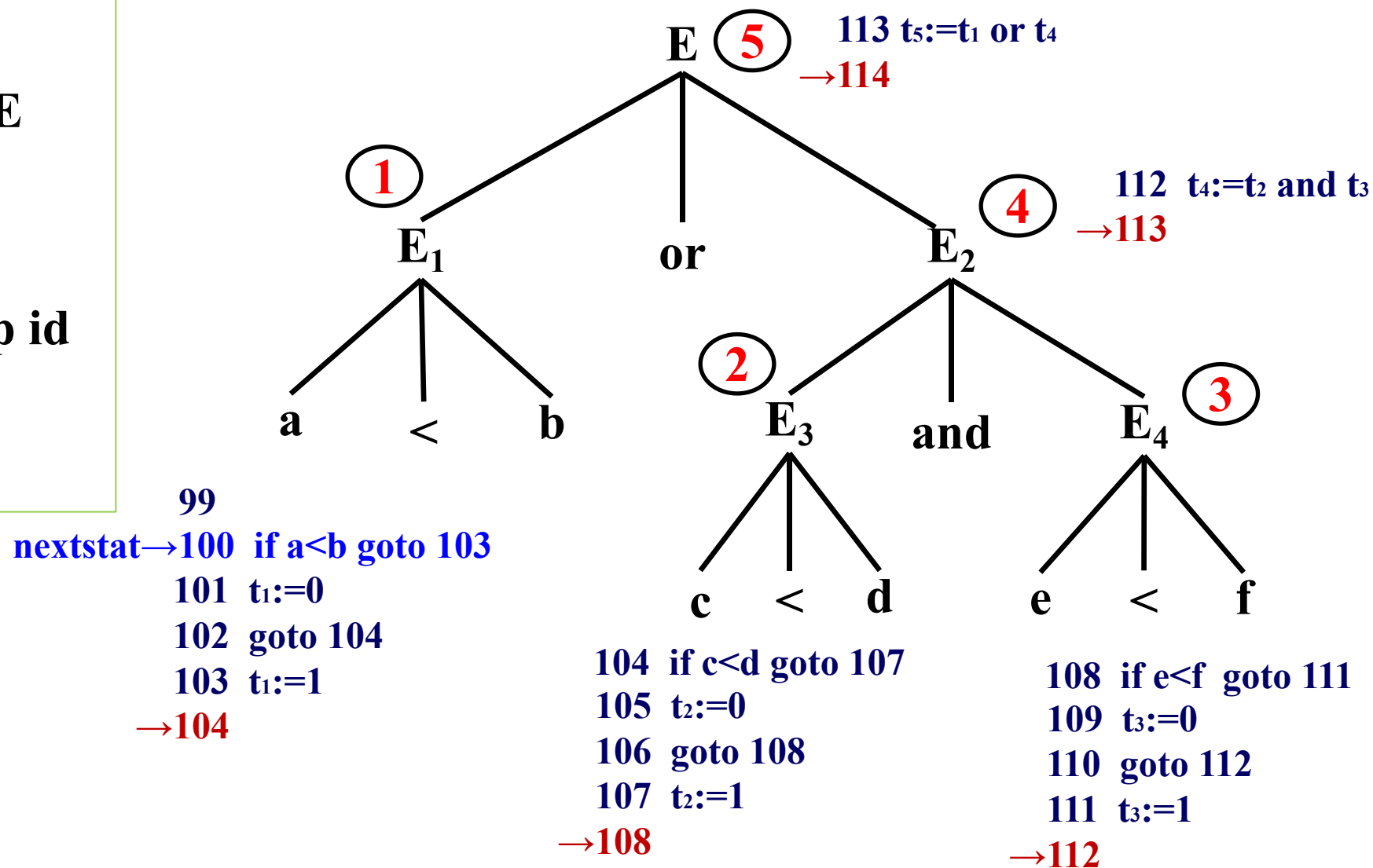
$E \rightarrow \text{not } E$

$E \rightarrow (E)$

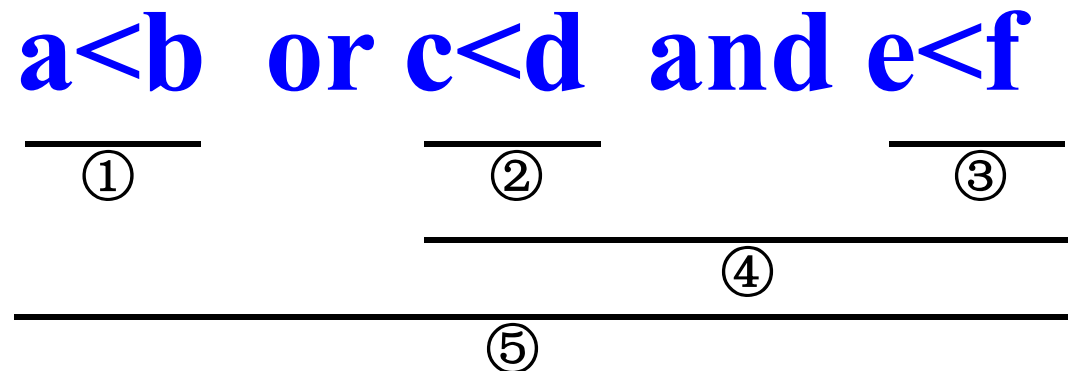
$E \rightarrow \text{id relop id}$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$



示例: $a < b$ or $c < d$ and $e < f$



100: if $a < b$ goto 103

101: $t_1 := 0$

102: goto 104

103: $t_1 := 1$

104: if $c < d$ goto 107

105: $t_2 := 0$

106: goto 108

107: $t_2 := 1$

108: if $e < f$ goto 111

109: $t_3 := 0$

110: goto 112

111: $t_3 := 1$

112: $t_4 := t_2$ and t_3

113: $t_5 := t_1$ or t_4

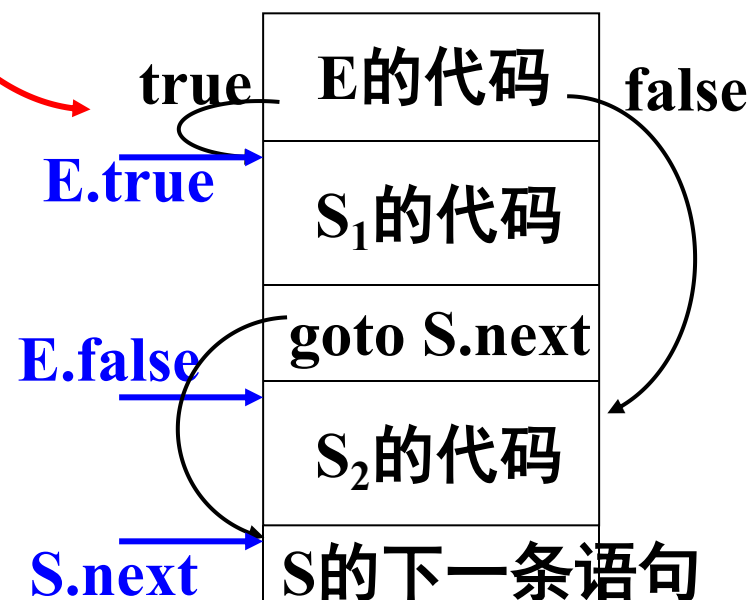
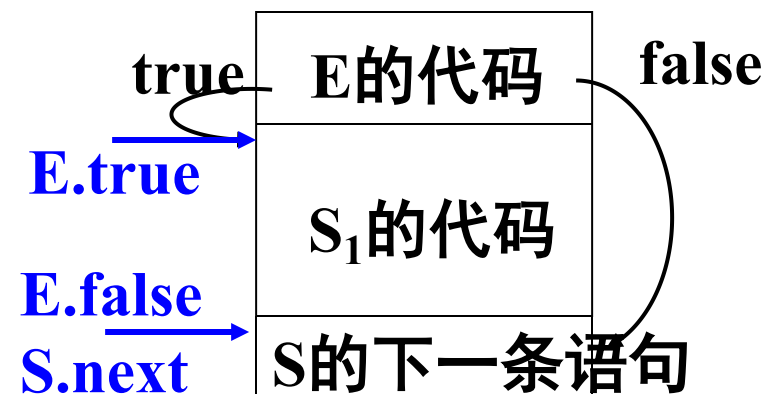
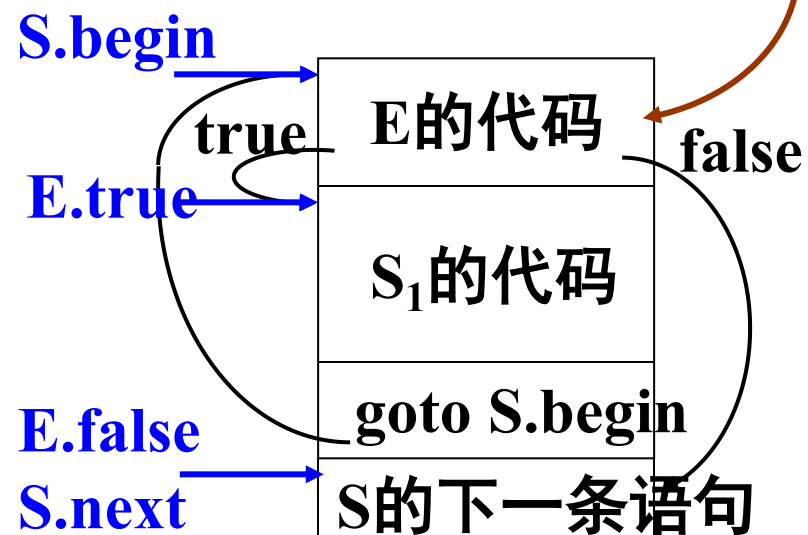
布尔表达式的控制流翻译方法

- 把 true、false 体现在程序(三地址代码形式)的位置上
 - ① 一般用此方法来实现控制流语句中的布尔表达式
 - ② 如true 跳转到标号为 109 的语句，false 跳转到标号为 103 的语句
 - ③ 布尔表达式的值体现在控制转移到的位置上

布尔表达式的控制流翻译方法

- 控制流语句

$S \rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1



布尔表达式的控制流翻译方法

■ 属性说明

① 继承属性：三地址语句标号

E.true: E值为真时应执行的第一条语句的标号

E.false: E值为假时应执行的第一条语句的标号

S.next: 紧跟在语句S之后的三地址语句的标号

S.begin: 语句S的第一条三地址语句的标号

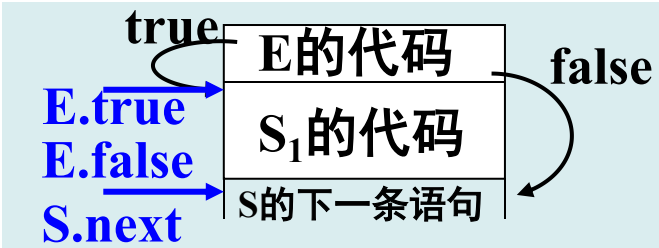
② 综合属性：三地址代码

E.code: 布尔表达式E的三地址代码

S.code: 语句S的三地址代码

翻译控制流语句的语法制导定义

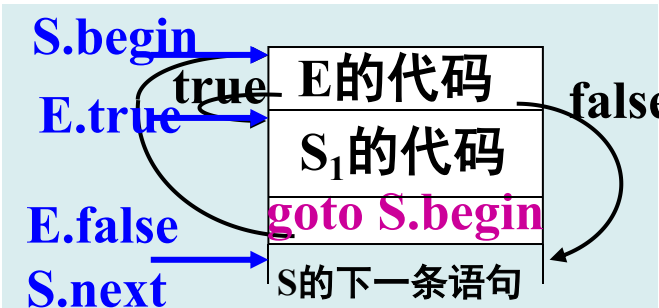
$S \rightarrow \text{if } E \text{ then } S_1$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$



$S \rightarrow \text{while } E \text{ do } S_1$



E.true = newlabel; E.false = S.next;

S₁.next = S.next;

S.code = E.code || gen(E.true':') || S₁.code

E.true = newlabel; E.flase = newlabel;

S₁.next = S.next; S₂.next = S.next;

S.code = E.code || gen(E.true':') || S₁.code

|| gen('goto' S.next)

|| gen(E.false':') || S₂.code

S.begin = newlabel; S₁.next = S.begin;

E.true = newlabel; E.false = S.next;

S.code = gen(S.begin':') || E.code

|| gen(E.true':') || S₁.code

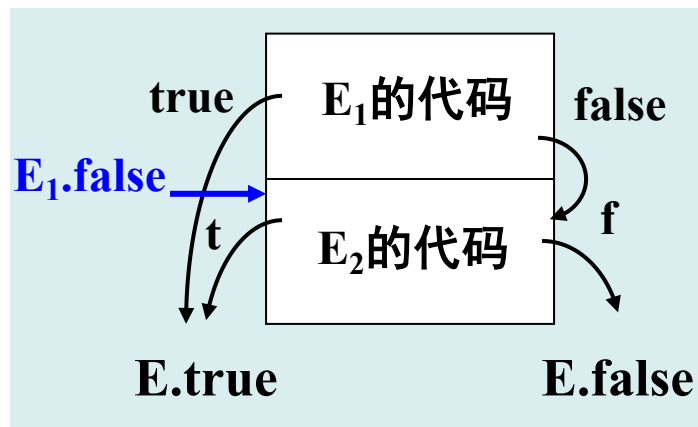
|| gen('goto' S.begin)

控制流语句中布尔表达式的翻译

- 产生布尔表达式三地址代码的语法制导定义
 - ① 布尔表达式被翻译为一系列条件转移和无条件转移三地址语句
 - ② 这些语句转移到的位置是E.true、E.false之一
 - ③ 例如 $a < b$ 翻译为：
if $a < b$ goto E.true
goto E.false
- 属性说明
 - ① 继承属性
 - E.true: E为真时转移到的三地址语句的标号
 - E.false: E为假时转移到的三地址语句的标号
 - ② 综合属性
 - E.code: 为E生成的三地址代码

翻译布尔表达式的语法制导定义

$E \rightarrow E_1 \text{ or } E_2$

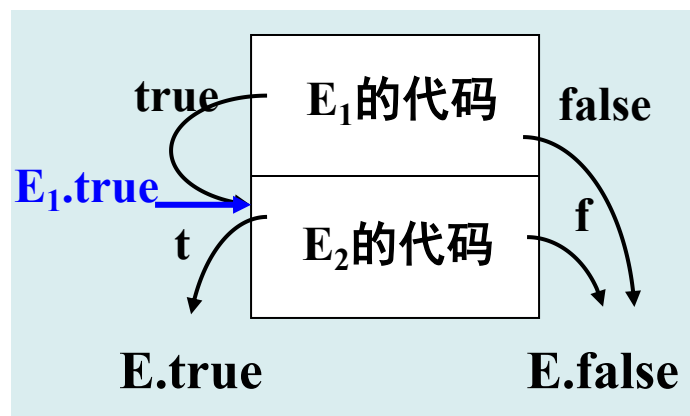


$E_1.\text{true} := E.\text{true}; \quad E_1.\text{false} := \text{newlable};$

$E_2.\text{true} := E.\text{true}; \quad E_2.\text{false} := E.\text{false};$

$E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false}':') \parallel E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$



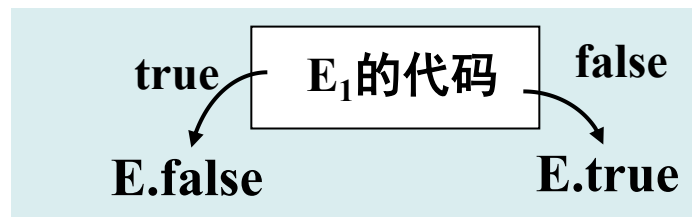
$E_1.\text{true} := \text{newlable}; \quad E_1.\text{false} := E.\text{false};$

$E_2.\text{true} := E.\text{true}; \quad E_2.\text{false} := E.\text{false};$

$E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true}':') \parallel E_2.\text{code}$

翻译布尔表达式的语法制导定义

$E \rightarrow \text{not } E_1$



$E_1.\text{true} := E.\text{false}; \quad E_1.\text{false} := E.\text{true}$

$E.\text{code} := E_1.\text{code}$

$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$

$E.\text{code} := \text{gen}(\text{'if' id}_1.\text{place relop.op id}_2.\text{place 'goto' E.true}) \parallel \text{gen}(\text{'goto' E.false})$

$E \rightarrow (E_1) \quad E_1.\text{true} := E.\text{true}; \quad E_1.\text{false} := E.\text{false};$

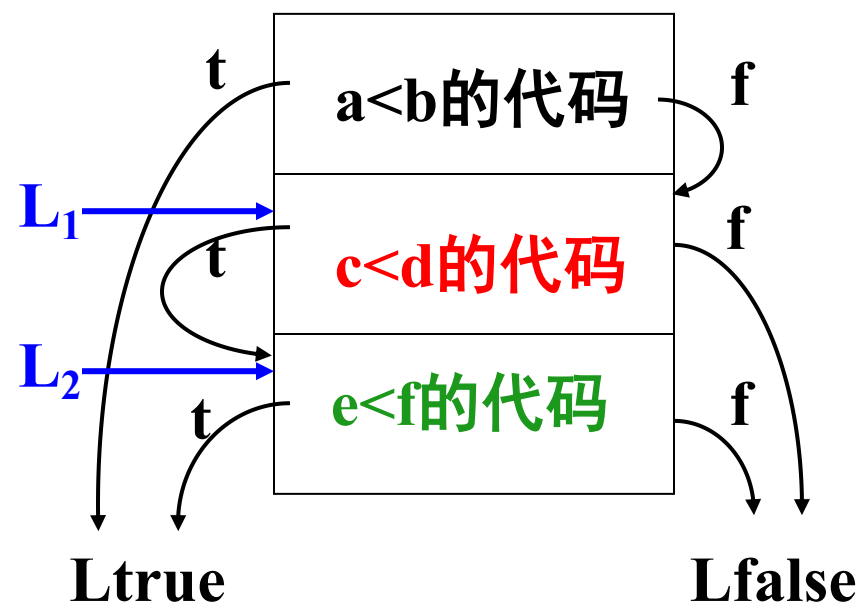
$E.\text{code} := E_1.\text{code}$

$E \rightarrow \text{true} \quad E.\text{code} := \text{gen}(\text{'goto' E.true})$

$E \rightarrow \text{false} \quad E.\text{code} := \text{gen}(\text{'goto' E.false})$

用控制流方法翻译布尔表达式

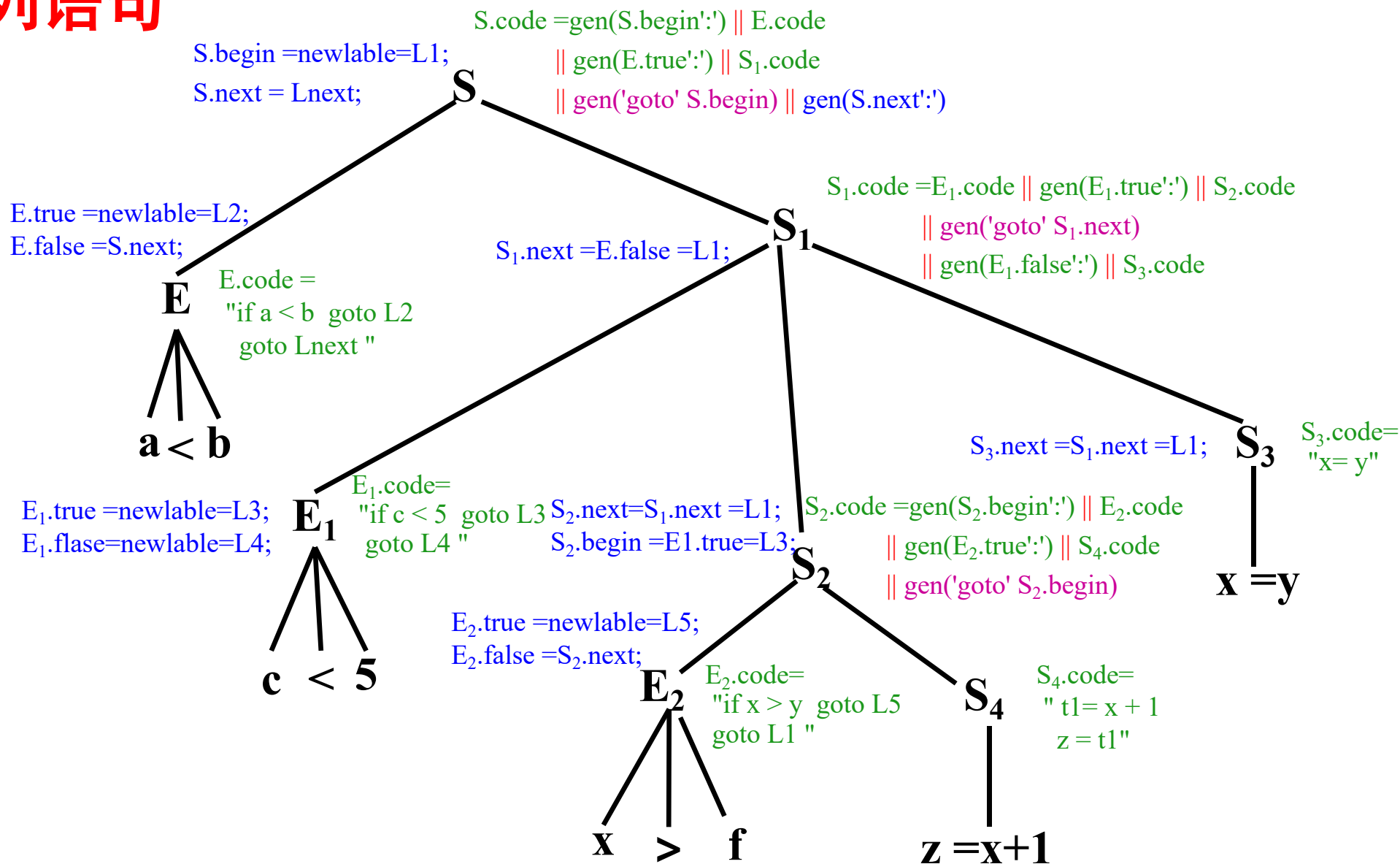
例: $a < b$ or $c < d$ and $e < f$



```
if a < b goto Ltrue
goto L1
L1: if c < d goto L2
      goto Lfalse
L2: if e < f goto Ltrue
      goto Lfalse
```

例：翻译下列语句

while a<b do
if c<5 then
while x>y do
z=x+1;
else x=y;



例：翻译下列语句

while a<b do

if c<5 then

while x>y do

z=x+1;

else x=y;

生成的三地址代码序列

L1: if a < b goto L2
goto Lnext

L2: if c < 5 goto L3
goto L4

L3: if x > y goto L5
goto L1

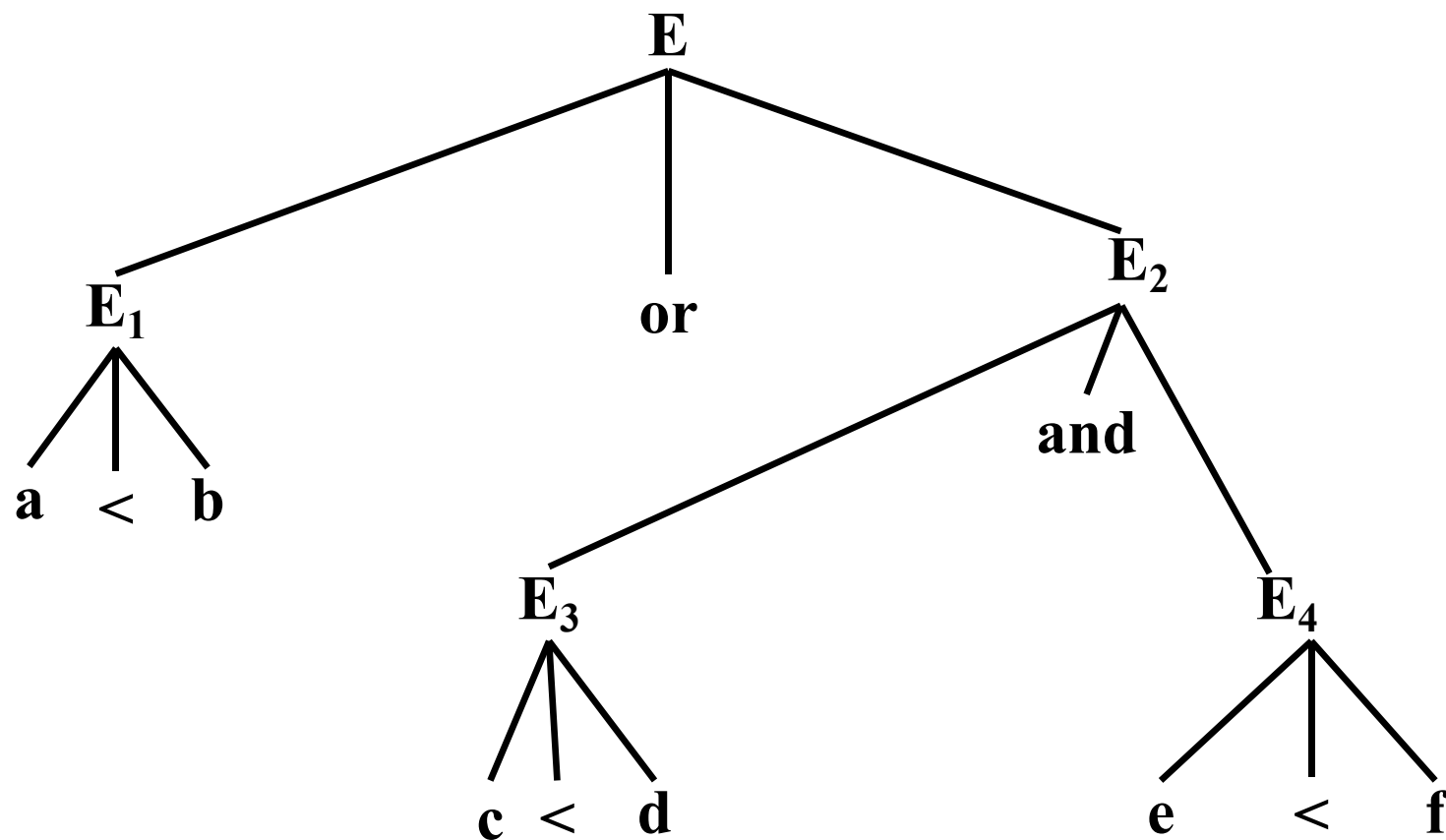
L5: $t_1 = x + 1$
z = t_1
goto L3

L4: x = y
goto L1

Lnext:

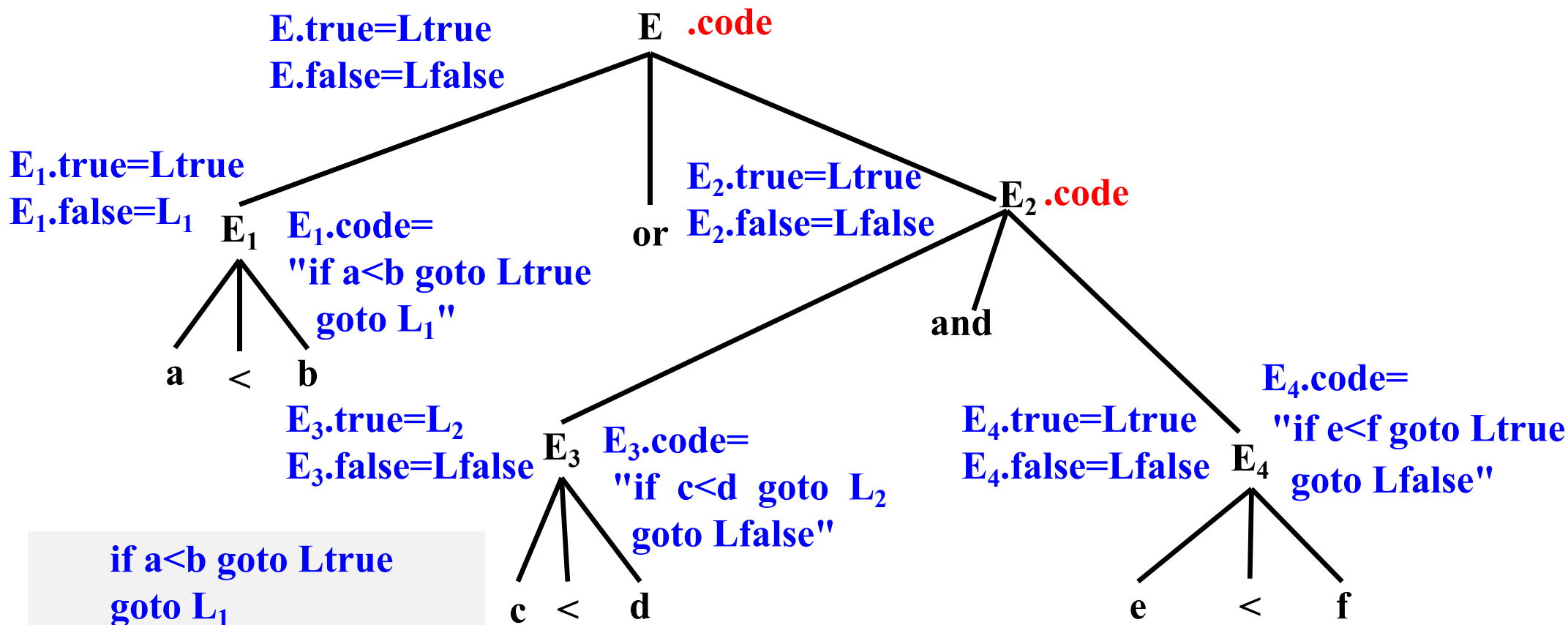
用控制流方法翻译布尔表达式

例: $a < b$ or $c < d$ and $e < f$



用控制流方法翻译布尔表达式

例: $a < b$ or $c < d$ and $e < f$



if $a < b$ goto Ltrue
goto L₁

L1: if $c < d$ goto L₂
goto Lfalse

L2: if $e < f$ goto Ltrue
goto Lfalse

两遍扫描:

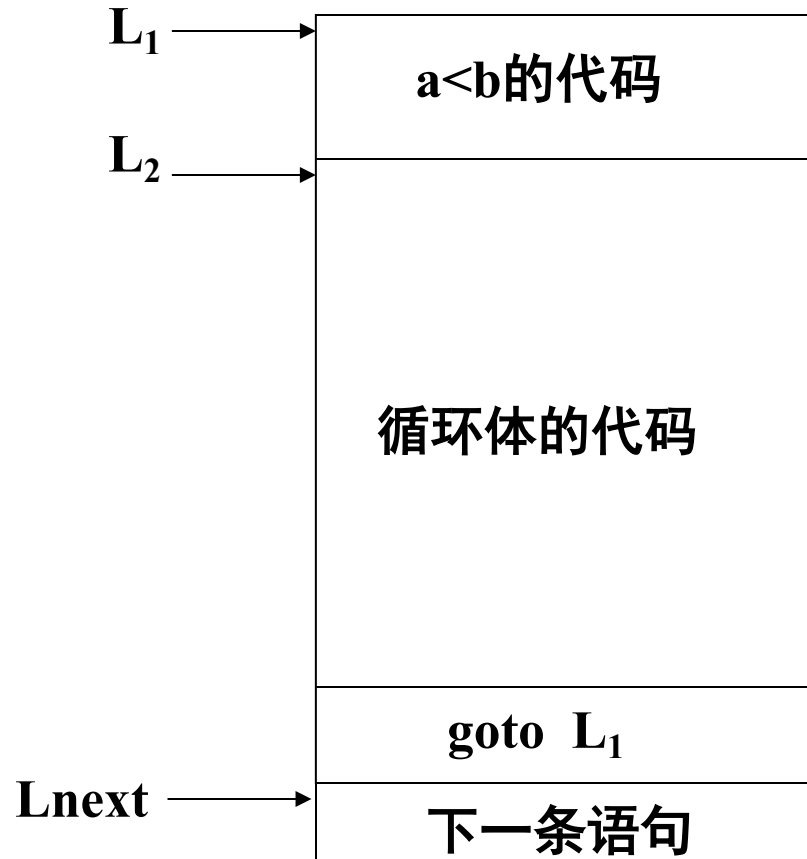
1. 生成分析树
2. 翻译, 在深度优先遍历分析树的过程中, 计算属性值

例:将下面的语句翻译为三地址代码

while a<b do

if c<d then x:=y+z

else x:=y-z;

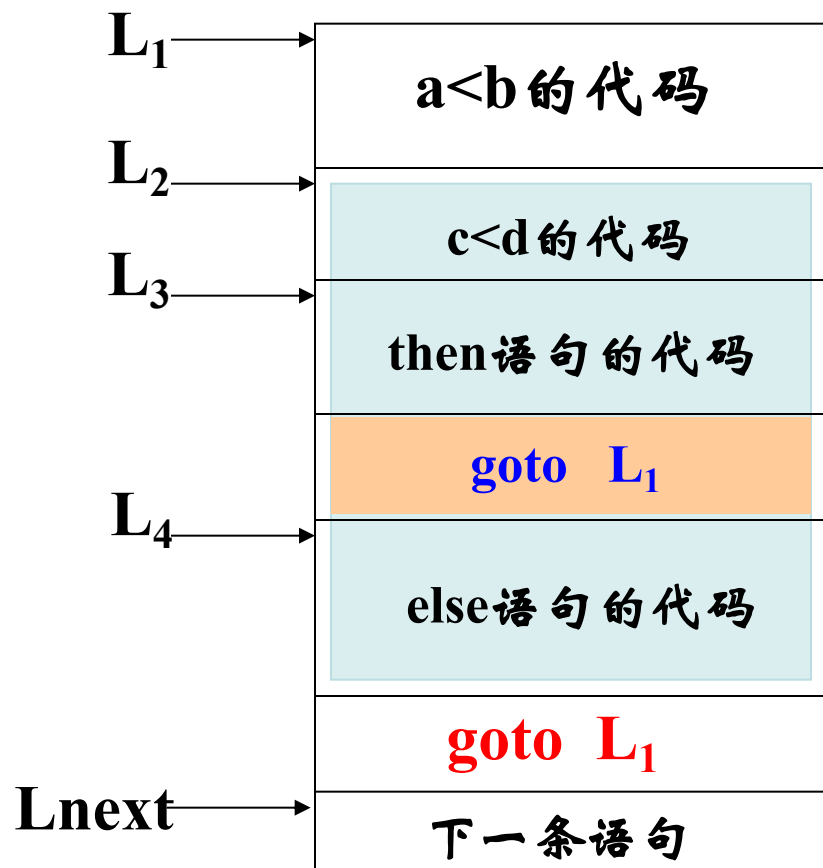


例:将下面的语句翻译为三地址代码

while a<b do

if c<d then x:=y+z

else x:=y-z;



• 三地址代码为:

L₁: if a<b goto L₂

goto L_{next}

L₂: if c<d goto L₃

goto L₄

L₃: t₁:=y+z

x:=t₁

goto L₁

L₄: t₂:=y-z

x:=t₂

goto L₁

L_{next}:

布尔表达式代码的例子

- `if (x < 100 || x > 200 && x != y) x = 0;` 的代码

```
        if x < 100 goto L2
        goto L3
L3:    if x > 200 goto L4
        goto L1
L4:    if x != y goto L2
        goto L1
L2:    x = 0
L1:
```

回填技术

- 为布尔表达式和控制流语句生成目标代码的**关键问题**：**某些跳转指令应该跳转到哪里**
- 例如：**if (E) S**
 - 按照前面的翻译方法，E的代码中有一些跳转指令在E为假时执行，
 - 这些跳转指令的目标应该跳过S对应的代码。**生成这些指令时，S的代码尚未生成，因此目标不确定**
 - **通过语句的继承属性next来传递**；需要第二趟处理。
- 如何一趟处理完毕呢？

回填技术

- 基本思想：
 - 在B的代码中，记录跳转指令位置，如：goto S.next, if ... goto S.next, 但是不生成跳转目标。
 - 这些位置被记录到B的综合属性B.falseList中；
 - 当S.next的值已知时（即S的代码生成完毕时），把B.nextList中的所有指令的目标都填上这个值。
- 回填技术：
 - 生成跳转指令时暂时不指定跳转目标标号，而是使用列表记录这些不完整的指令；
 - 等知道正确的目标时再填写目标标号；
 - 每个列表中的指令都指向同一个目标

利用回填技术翻译布尔表达式

- 为明确起见，生成的中间代码用四元式表示
- 四元式存放在数组中
- 用数组下标表示三地址语句的标号
- 综合属性
 - ① E.truelist: 记录转移到E的真出口的指令的链表指针
 - ② E.falselist: 记录转移到E的假出口的指令的链表指针
 - ③ M.instr: M所标识的三地址语句的地址
- 变量nextinstr: 下一个可用的四元式地址,(产生的下一条三地址语句在数组中的位置)

利用回填技术翻译布尔表达式(辅助函数说明)

① makelist(i):

- a) 建立新链表，其中只包括四元式指令在数组中的下标i;
- b) 返回所建链表的指针。

② merge(p_1 , p_2):

- a) 合并由指针 p_1 和 p_2 所指向的两个链表
- b) 返回结果链表的指针。

③ backpatch(p, i):

用目标地址i回填 p所指链表中记录的每一条转移指令。

④ gen(S)

- a) 产生一条三地址语句S，并写入输出数组中
- b) 该函数执行完后，变量 nextinstr 加 1

利用回填技术翻译布尔表达式

- 布尔表达式的文法

$E \rightarrow E_1 \text{ or } M E_2$

$E \rightarrow E_1 \text{ and } M E_2$

$E \rightarrow \text{not } E_1$

$E \rightarrow (E_1)$

$E \rightarrow \text{id}_1 \text{ relop id}_2$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

$M \rightarrow \epsilon$

- 标记非终结符号 M

① 标识布尔表达式 E_2 的开始位置

② 用属性 $M.\text{instr}$ 记录其所标识的布尔表达式的第一条三地址语句的地址

③ 相应于产生式 $M \rightarrow \epsilon$ 的动作: $M.\text{instr} := \text{nextinstr}$

利用回填技术翻译布尔表达式

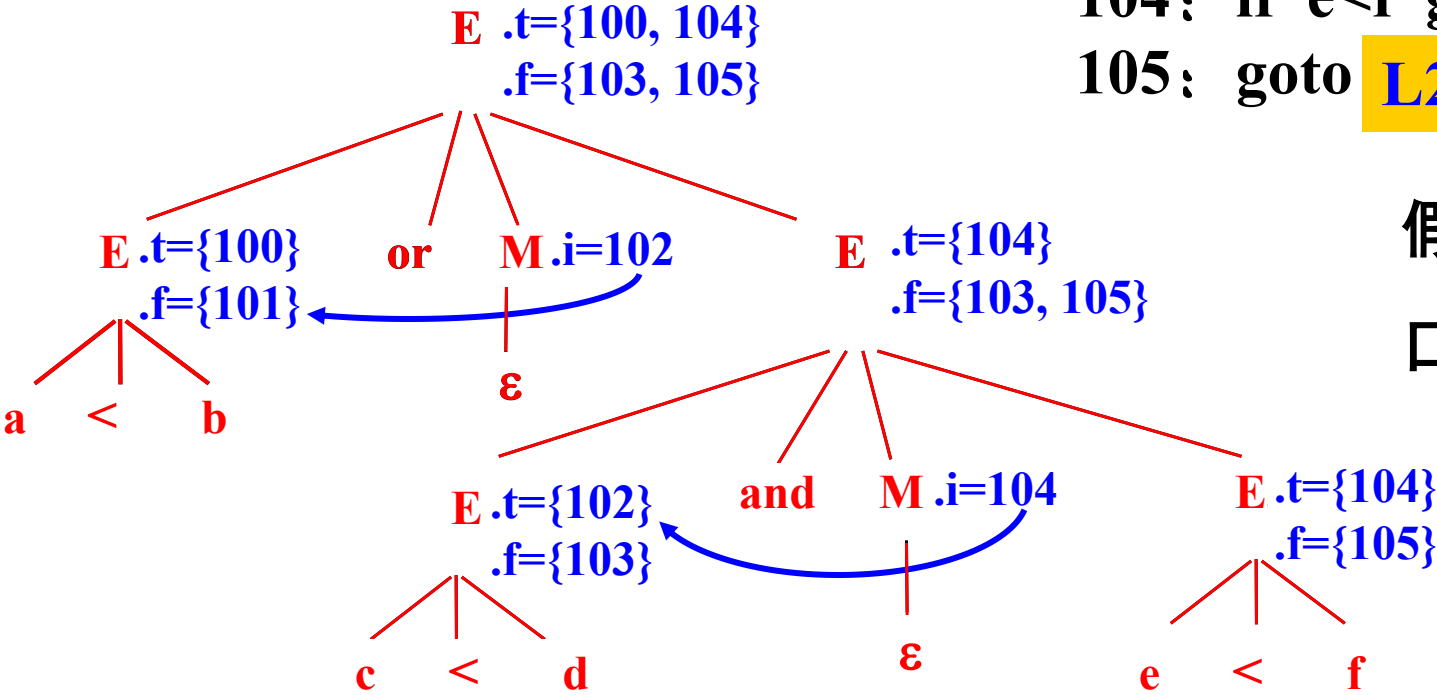
$E \rightarrow E_1 \text{ or } M E_2$ { <i>backpatch</i> ($E_1.\text{falselist}$, $M.\text{instr}$) ; $E.\text{truelist} := \text{merge} (E_1.\text{truelist}, E_2.\text{truelist})$; $E.\text{falselist} := E_2.\text{falselist}$ }	$E \rightarrow \text{id}_1 \text{ relop id}_2$ { $E.\text{truelist} := \text{makelist} (\text{nextinstr})$; $E.\text{falselist} := \text{makelist} (\text{nextinstr} + 1)$; <i>gen</i> ('if' $\text{id}_1.\text{place}$ <i>relop.op</i> $\text{id}_2.\text{place}$ 'goto_') ; <i>gen</i> ('goto_') }
$E \rightarrow E_1 \text{ and } M E_2$ { <i>backpatch</i> ($E_1.\text{truelist}$, $M.\text{instr}$) ; $E.\text{truelist} := E_2.\text{truelist}$; $E.\text{falselist} := \text{merge} (E_1.\text{falselist}, E_2.\text{falselist})$ }	$E \rightarrow \text{true}$ { $E.\text{truelist} = \text{makelist} (\text{nextinstr})$; <i>gen</i> ('goto_') }
$E \rightarrow \text{not } E_1$ { $E.\text{truelist} := E_1.\text{falselist}$; $E.\text{falselist} := E_1.\text{truelist}$ }	$E \rightarrow \text{false}$ { $E.\text{falselist} = \text{makelist} (\text{nextinstr})$; <i>gen</i> ('goto_') }
$E \rightarrow (E_1)$ { $E.\text{truelist} := E_1.\text{truelist}$; $E.\text{falselist} := E_1.\text{falselist}$ }	$M \rightarrow \varepsilon$ { $M.\text{instr} = \text{nextinstr}$ }

利用回填技术翻译布尔表达式

■ $a < b \text{ or } c < d \text{ and } e < f$

假定变量nextinstr的初值为100

```
100: if a<b goto L1
101: goto 102
102: if c<d goto 104
103: goto L2
104: if e<f goto L1
105: goto L2
```



假设已知E的真假出口分别是L1和L2

利用回填技术翻译控制流语句

■ 文法

$S \rightarrow \text{if } E \text{ then } \# S_1$

$S \rightarrow \text{if } E \text{ then } \# S_1 \ \& \ \text{else } \# S_2$

$S \rightarrow \text{while } \# E \text{ do } \# S_1$

$S \rightarrow \text{begin } L \text{ end}$

$S \rightarrow A$

$L \rightarrow L_1; \ \# S$

$L \rightarrow S$

记录变量 `nextinstr` 的当前值，以便回填转移到此的指令

& 产生一条不完整的goto指令，并记录下它的位置

利用回填技术翻译控制流语句

■ 改写后的文法

$S \rightarrow \text{if } E \text{ then } \mathbf{M} S_1$

$S \rightarrow \text{if } E \text{ then } \mathbf{M}_1 S_1 \mathbf{N} \text{ else } \mathbf{M}_2 S_2$

$S \rightarrow \text{while } \mathbf{M}_1 E \text{ do } \mathbf{M}_2 S_1$

$S \rightarrow \text{begin } L \text{ end}$

$S \rightarrow A$

$L \rightarrow L_1; \mathbf{M} S$

$L \rightarrow S$

$\mathbf{M} \rightarrow \epsilon$

$\mathbf{N} \rightarrow \epsilon$

属性:

$E.\text{truelist}$

$E.\text{falselist}$

$M.\text{instr}$

$\mathbf{N}.\text{nextlist}$

$\mathbf{S}.\text{nextlist}$

变量: nextinstr

函数:

makelist

backpatch

merge

gen

转移到下一条语句
的指令链表的指针

\mathbf{M} (标记非终结符)— 为了引进语义动作, 产生一个标号, 标识 E 或 S 的第一条四元式的位置

\mathbf{N} (标记非终结符)— 为了引进语义动作, 在 S_1 的最后产生一条转移语句跳过 S_2 的代码

利用回填技术翻译控制流语句（翻译方案）

$S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$
{ *backpatch* (*E.truelist* , $M_1.instr$) ;
 backpatch (*E.falselist* , $M_2.instr$) ;
 $S.nextlist = \text{merge} (S_1.nextlist , \text{merge}(N.nextlist , S_2.nextlist))$ }

$N \rightarrow \epsilon$
{ $N.nextlist = \text{makelist} (nextinstr)$;
 gen ('*goto_*') }




$M \rightarrow \epsilon$
{ $M.instr = nextinstr$ }




$S \rightarrow \text{if } E \text{ then } M S_1$
{ *backpatch* (*E.truelist* , $M.instr$) ;
 $S.nextlist = \text{merge} (E.falselist , S_1.nextlist)$ }

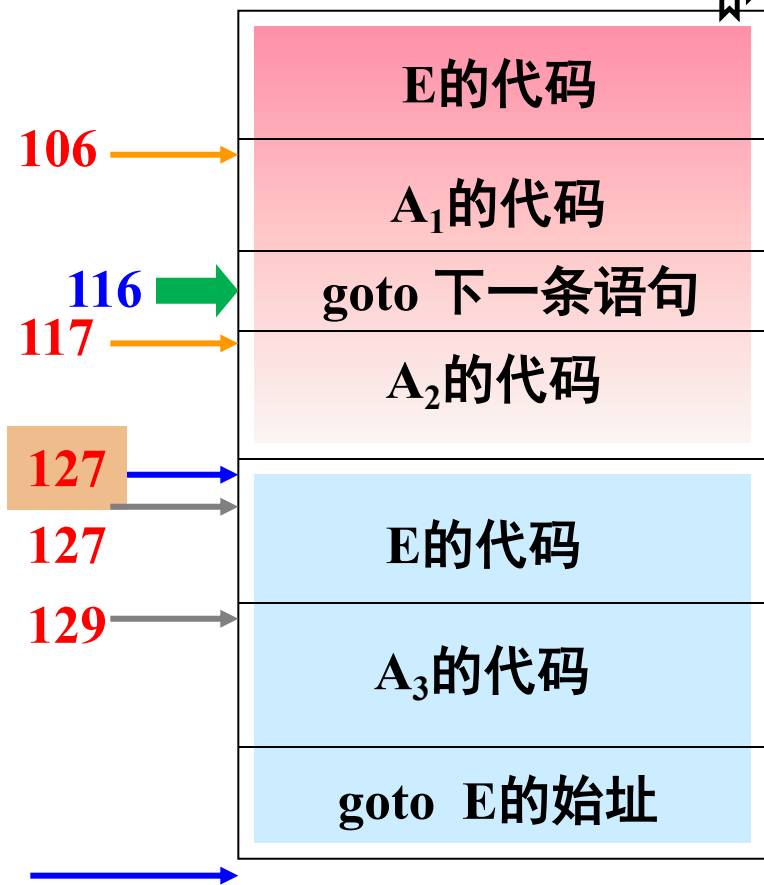
利用回填技术翻译控制流语句（翻译方案）

$\begin{aligned} S \rightarrow & \text{while } M_1 E \text{ do } M_2 S_1 \\ & \{ \text{backpatch} (S_1.\text{nextlist}, M_1.\text{instr}); \\ & \quad \text{backpatch} (E.\text{truelist}, M_2.\text{instr}); \\ & \quad S.\text{nextlist} := E.\text{falselist}; \\ & \quad \text{gen} (\text{'goto'} M_1.\text{instr}) \} \end{aligned}$
$\begin{aligned} S \rightarrow & \text{begin } L \text{ end} \\ & \{ S.\text{nextlist} := L.\text{nextlist} \} \end{aligned}$
$\begin{aligned} S \rightarrow & A \\ & \{ S.\text{nextlist} := \text{nil} \} \end{aligned}$
$\begin{aligned} L \rightarrow & L_1 ; M S \\ & \{ \text{backpatch} (L_1.\text{nextlist}, M.\text{instr}); \\ & \quad L.\text{nextlist} := S.\text{nextlist} \} \end{aligned}$

例:

if a<b or c<d and e<f then ¹⁰⁶ A₁ ¹¹⁶ else ¹¹⁷ A₂;

¹²⁷ while ¹²⁷ a<b do ¹²⁹ A₃



100: if a<b goto 106
101: goto 102
102: if c<d goto 104
103: goto 117
104: if e<f goto 106
105: goto 117
106: A₁的代码
115:
116: goto 127
117: A₂的代码
126:
127: if a<b goto 129
128: goto —
129: A₃的代码
138:
139: goto 127