

第8章 中间代码生成

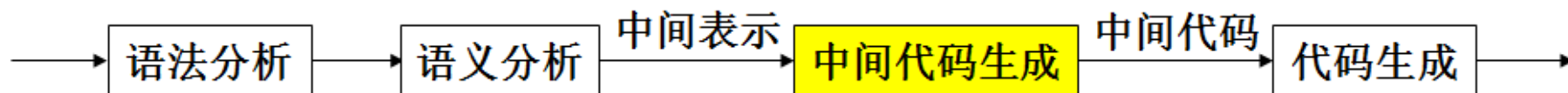


LI Wensheng, SCS, BUPT

知识点：三地址代码
语句的翻译
布尔表达式的翻译
回填技术

中间代码生成

- 中间代码生成程序的任务
 - ◆ 把经语法分析、语义分析后得到的源程序的中间表示形式翻译成中间代码表示。
- 采用中间代码作为过渡的优点
 - ◆ 便于编译程序的建立和移植
 - ◆ 便于进行与机器无关的代码优化工作
- 缺点
 - ◆ 增加了I/O操作、效率有所下降
- 中间代码生成程序的位置：



中间代码生成

- 8.1 中间代码形式
- 8.2 赋值语句的翻译
- 8.3 布尔表达式的翻译
- 8.4 控制语句的翻译
- 小 结

8.1 中间代码形式

8.1.1 图形表示

- ◆ 语法树
- ◆ dag图

8.1.2 三地址代码

三地址代码

- ◆ 三地址语句的形式
- ◆ 三地址语句的种类
- ◆ 三地址语句的实现

8.1.1 图形表示

- 语法树

- ◆ 描绘了源程序的自然层次结构。

- dag图

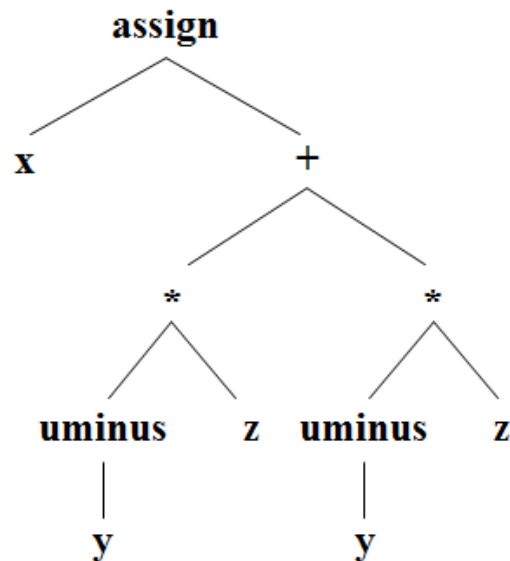
- ◆ 以更紧凑的方式给出了与语法树同样的信息。
- ◆ 在dag中，公共子表达式被标识出来了。

为赋值语句构造语法树的语法制导定义

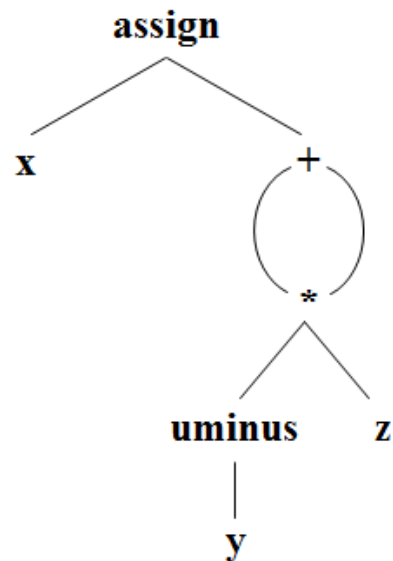
产生式	语义规则
$S \rightarrow id := E$	$S.nptr = makenode(':=', makeleaf(id, id.entry), E.nptr)$
$E \rightarrow E_1 + T$	$E.nptr = makenode('+', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = makenode('*', T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow \text{uminus } E$	$F.nptr = makeunode('uminus', E.nptr)$
$F \rightarrow id$	$F.nptr = makeleaf(id, id.entry)$
$F \rightarrow \text{num}$	$F.nptr = makeleaf(num, num.val)$

赋值语句 $x := (-y) * z + (-y) * z$ 的图表示法

■ 语法树表示:



■ dag图形表示:



- 后缀式是语法树的线性表示形式。
- 对语法树进行深度优先遍历、访问子结点先于父结点、且从左向右访问子结点，得到一个包含所有树结点的序列，即后缀式。
- 在此序列中，每个树结点出现且仅出现一次；每个结点都是在它的所有子结点出现之后立即出现。
- 与上述语法树对应的后缀式是：
 $x \ y \ \text{uminus} \ z \ * \ y \ \text{uminus} \ z \ * \ + \ \text{assign}.$

8.1.2 三地址代码

■ 三地址代码：三地址语句组成的序列。

- ◆ 类似于汇编语言的代码
- ◆ 有赋值语句、控制语句
- ◆ 语句可以有标号

■ 三地址语句的一般形式： $x := y \text{ op } z$

- ◆ x 可以是名字、临时变量
- ◆ y 、 z 可以是名字、常数、或临时变量
- ◆ op 代表运算符，如算术运算符、或逻辑运算符等
- ◆ 语句中，最多有三个地址，两个运算对象的地址和结果地址。

■ 实现时，语句中的名字，将由指向该名字在符号表中表项的指针所代替。

三地址代码.

三地址语句的种类及形式

由指令代替

■ 简单赋值语句

◆ $x := y \text{ op } z$

◆ $x := \text{op } y$

◆ $x := y$

■ 含有变址的赋值语句

◆ $x := y[i]$

◆ $x[i] := y$

■ 含有地址和指针的赋值语句

◆ $x := \&y$

◆ $x := *y$

◆ $*x := y$

■ 转移语句

◆ $\text{goto } L$

◆ $\text{if } x \text{ relop } y \text{ goto } L$

■ 过程调用语句

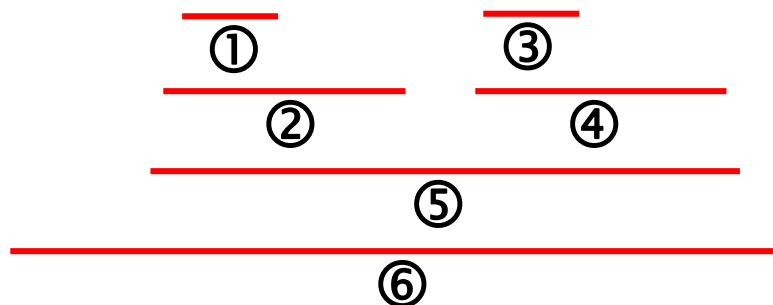
◆ $\text{param } x$

◆ $\text{call } p, n$

■ 返回语句

◆ $\text{return } y$

赋值语句 $x:=(-y)*z+(-y)*z$ 的三地址代码



■ 对应语法树的代码

$t_1 := -y$

$t_2 := t_1 * z$

$t_3 := -y$

$t_4 := t_3 * z$

$t_5 := t_2 + t_4$

$x := t_5$

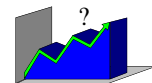
■ 对应dag的代码

$t_1 := -y$

$t_2 := t_1 * z$

$t_5 := t_2 + t_2$

$a := t_5$



- 三地址代码是语法树或dag的线性表示，其中新增加的名字对应图中的内部结点

三地址代码

三地址代码的呈现形式

三地址语句的实现——四元式

$x := -y$

$'+' , y, z, x$

■ 四元式

(op, arg₁, arg₂, result) 如: $x := y + z$ ('+', y, z, x)

(op, arg₁, ~~无参数~~, result) 如: $x := -y$ ('uminus', y, , x)

(param, arg₁, ,) 如: param x (param, x)

(goto, , , 语句标号) 如: goto L (goto, , , L)

■ 赋值语句 $x := (-y) * z + (-y) * z$ 的四元式表示

	op	arg ₁	arg ₂	result
(0)	uminus	y		t ₁
(1)	*	t ₁	z	t ₂
(2)	uminus	y		t ₃
(3)	*	t ₃	z	t ₄
(4)	+	t ₂	t ₄	t ₅
(5)	:=	t ₅		x

三地址语句的实现——三元式

- 三元式：(op, arg₁, arg₂)
 - ◆ 为避免把临时变量名也存入符号表，不引入临时变量
 - ◆ 一个语句计算出来的中间结果直接提供给引用它的语句
 - ◆ 用计算中间结果的语句的指针代替存放中间结果的临时变量
- 赋值语句 $x := (-y) * z + (-y) * z$ 的三元式表示

语句序号	op	arg1	arg2
(0)	uminus	y	
(1)	*	(0) 直接用 (0)、(1)、(2)	z
(2)	uminus	y	
(3)	*	(2)	z
(4)	+	(1)	(3)
(5)	assign	x	(4)

语句 $x[i] := y$ 和 $x := y[i]$ 的三元式序列

■ 语句 $x[i] := y$ *arg1 op arg2.*

语句序号	op	arg1	arg2
(0)	$[] =$	<u>x</u>	<u>i</u>
(1)	assign	(0)	y

■ 语句 $x := y[i]$ *(0)*

语句序号	op	arg1	arg2
(0)	$= []$	y	i
(1)	assign	<u>x</u>	<u>(0)</u>

$[] =$ y i

assign 相当于拼接.

三地址语句的实现——间接三元式

■ 间接三元式

- ◆ 间接码表：为三元式序列建立的一个指针数组，其每个元素依次指向三元式序列中的一项

■ 赋值语句 $x := (-y) * z + (-y) * z$ 的间接三元式表示

间接码表

序号	三地址语句序号
(14)	(0)
(15)	(1)
(16)	(2)
(17)	(3)
(18)	(4)
(19)	(5)

三元式

语句序号	op	arg1	arg2
(0)	uminus	y	
(1)	*	(0)	z
(2)	uminus	y	
(3)	*	(2)	z
(4)	+	(1)	(3)
(5)	assign	x	(4)

间接码表

8.2 赋值语句的翻译

- 假定赋值语句出现的环境可用下面的文法描述：

$P \rightarrow M D; S$

$M \rightarrow \varepsilon$

$D \rightarrow D; D \mid D \rightarrow \text{id}: T \mid D \rightarrow \text{proc id}; N D; S$

$N \rightarrow \varepsilon$

$T \rightarrow \text{integer} \mid \text{real}$
 $\mid \text{array} [\text{num}] \text{ of } T_1$
 $\mid \uparrow T_1$
 $\mid \text{record } L D \text{ end}$

$L \rightarrow \varepsilon$

$S \rightarrow \text{id} := E$

$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \text{id} \mid \text{num} \mid \text{num.num}$

设计函数：

- (1) $p = \text{lookup}(\text{id.name})$
- (2) $\text{gettype}(p)$
- (3) $\text{newtemp}()$
- (4) $\text{outcode}(s)$

8.2.1 仅涉及简单变量的赋值语句

■ 文法 $S \rightarrow \text{id} := E$

$S \rightarrow \text{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

$E \rightarrow \text{num.num}$

- 属性 $E.\text{entry}$:
记录与E相应的临时变量
在符号表中的表项位置

翻译方案8.1

$S \rightarrow id := E$ { $p = \text{lookup}(id.name);$
if ($p \neq \text{nil}$) $\text{outcode}(p := E.entry);$
else $\text{error}();$ }

$E \rightarrow E_1 + E_2$ { $E.entry = \text{newtemp}();$
 $\text{outcode}(E.entry := E_1.entry + E_2.entry)$ }

$E \rightarrow E_1 * E_2$ { $E.entry = \text{newtemp}();$
 $\text{outcode}(E.entry := E_1.entry * E_2.entry)$ }

$E \rightarrow -E_1$ { $E.entry = \text{newtemp}();$
 $\text{outcode}(E.entry := \text{'uminus'} E_1.entry)$ }

$E \rightarrow (E_1)$ { $E.entry = E_1.entry$ }

$E \rightarrow id$ { $p = \text{lookup}(id.name);$
if ($p \neq \text{nil}$) $E.entry = p;$
else $\text{error}();$ }

思考?

$E \rightarrow \text{num}$??

$E \rightarrow \text{num.num}$??

同时进行类型检查的翻译方案

- 假设，仅考虑类型 `integer` 和 `real`

- $E \rightarrow E_1 + E_2$ 的类型检查动作：

```
{ if (E1.type==integer) && (E2.type==integer)
    E.type=integer;
  else E.type =real; }
```

integer 和 real.

$E \rightarrow E_1 + E_2$ 带有类型检查的语义动作

```
{ E.entry=newtemp();  
  if (E1.type==integer) && (E2.type==integer) {  
    outcode(E.entry ':=' E1.entry 'int+' E2.entry);  
    E.type=integer; }  
  else if (E1.type==real) && (E2.type==real) {  
    outcode(E.entry ':=' E1.entry 'real+' E2.entry);  
    E.type=real; }  
  else if (E1.type==integer) && (E2.type==real) {  
    u=newtemp();  
    outcode(u ':=' 'inttoreal' E1.entry);  
    outcode(E.entry ':=' u 'real+' E2.entry);  
    E.type=real; }  
  else if (E1.type==real) && (E2.type==integer) {  
    u=newtemp();  
    outcode(u ':=' 'inttoreal' E2.pace);  
    outcode(E.entry ':=' E1.entry 'real+' u);  
    E.type=real; }  
  else E.type=type_error; }
```

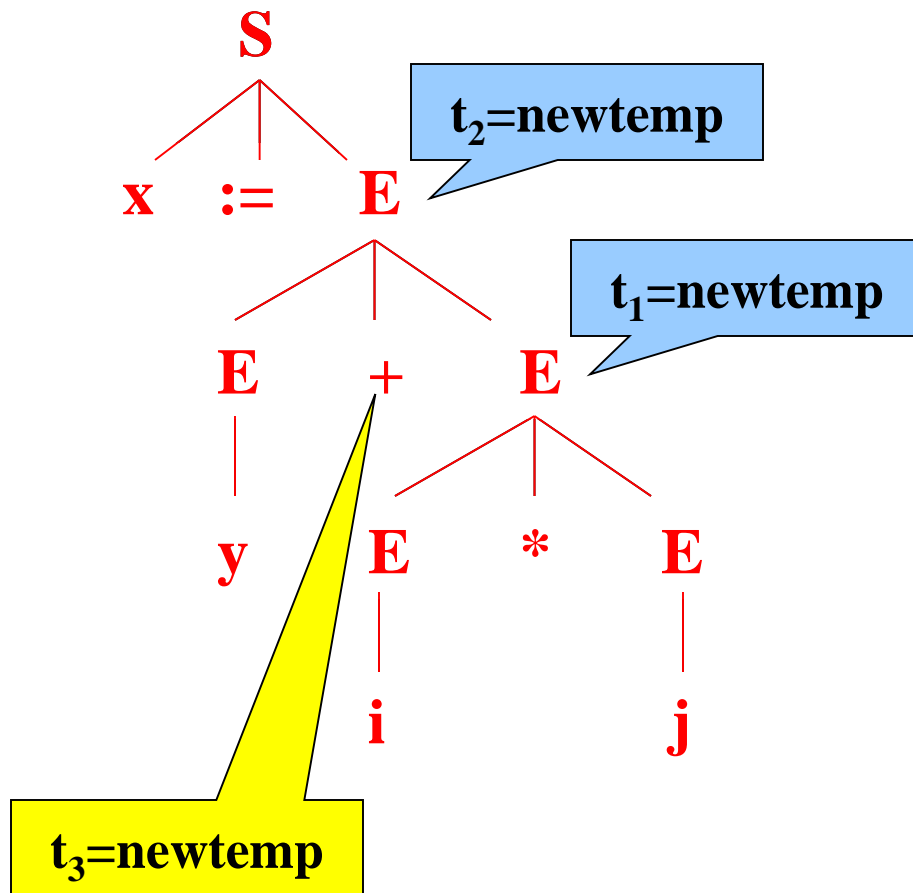


$S \rightarrow id := E$ 带有类型检查的语义动作

```
{ p=lookup(id.name);  
  if (p!=nil) {  
    t=gettype(p);  
    if (t==E.type) {  
      outcode(p ':=' E.entry);  
      S.type=void;  };  
    else if (t==real) && (E.type==integer) {  
      u=newtemp();  
      outcode(u ':=' 'inttoreal' E.entry);  
      outcode(p ':=' u);  
      S.type=void;  }  
    else S.type=type_error;  
  };  
  else error(); }
```

翻译赋值语句 $x := y + i * j$

- 假定 x 和 y 的类型为 real ， i 和 j 的类型为 integer



- 三地址代码:

$t_1 := i \text{ int} * j$

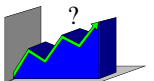
$t_3 := \text{inttoreal } t_1$

$t_2 := y \text{ real} + t_3$

$x := t_2$

$t_1 := i \text{ int} * j$

$t_3 :=$



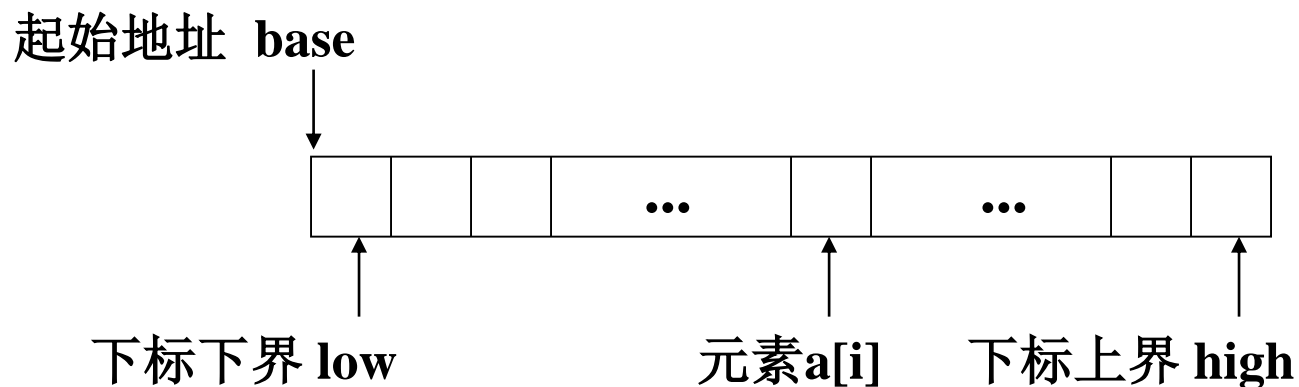
8.2.2 涉及数组元素的赋值语句

1. 计算数组元素的地址

- ◆ 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可以快速查找每个元素。
- ◆ 数组空间起始地址：base
- ◆ 每个元素的域宽：w

- 一维数组 $A[i]$
- 二维数组 $A[i, j]$
- k 维数组 $A[i_1, i_2, \dots, i_k]$

一维数组--A[i]的地址



数组元素个数: $\text{high} - \text{low} + 1$

数组元素A[i]的位置:

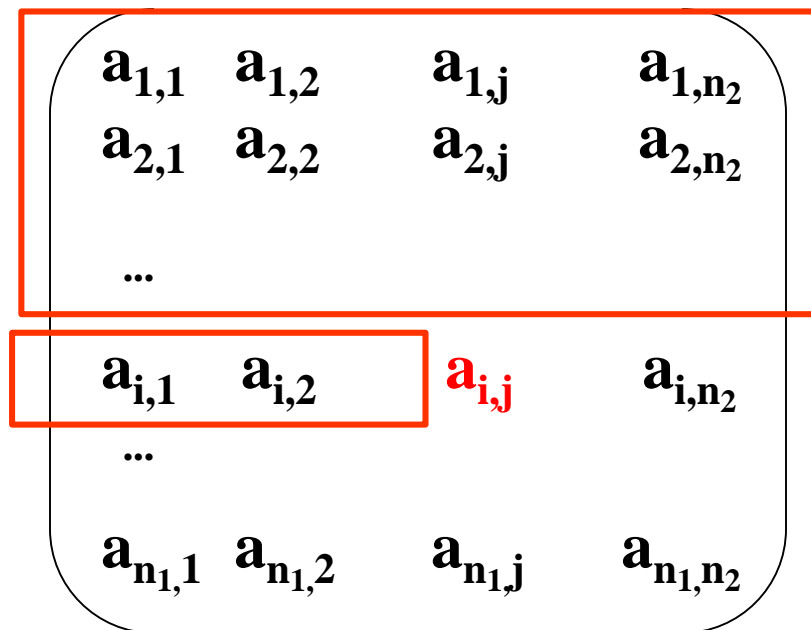
$$\begin{aligned} & \text{base} + (\textcolor{red}{i} - \text{low}) \times w \\ &= \textcolor{red}{i} \times w + \text{base} - \text{low} \times w \end{aligned}$$

常数C

编译时刻确定的常数，
保存在符号表中

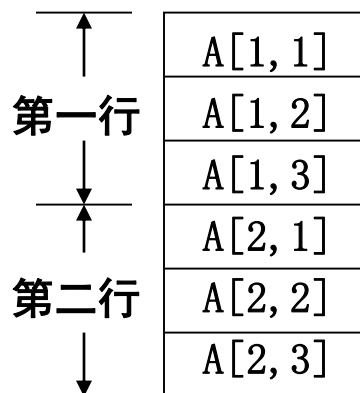
二维数组--A[i, j]的地址

存储方式:



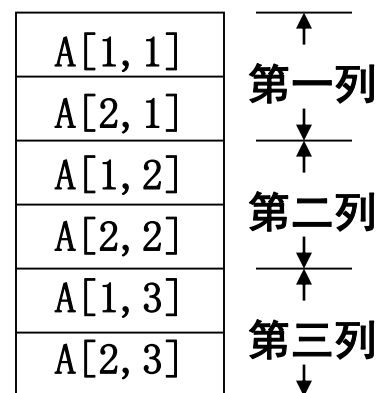
按行优先存放

Pascal,C采用



按列优先存放

Fortran采用



每维的下界: low_1 、 low_2

每维的长度: $n_1=high_1-low_1$

域宽: w

数组元素A[i,j]的位置:

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

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

每维的上界: $high_1$ 、 $high_2$

$n_2=high_2-low_2$

基址: $base$

编译时刻确定的常数,
保存在符号表中

常数C

k维数组-- $A[i_1, i_2, \dots, i_k]$ 的地址

每维的下界: $\text{low}_1, \text{low}_2, \dots, \text{low}_k$

每维的长度: n_1, n_2, \dots, n_k

存储方式: 按行存放

数组元素 $A[i_1, i_2, \dots, i_k]$ 的位置:

$$((\dots((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w$$

$$+ \text{base} - ((\dots((\text{low}_1 \times n_2 + \text{low}_2) \times n_3 + \text{low}_3) \dots) \times n_k + \text{low}_k) \times w$$

递归计算:

$$e_1 = i_1$$

$$e_2 = e_1 \times n_2 + i_2$$

$$e_3 = e_2 \times n_3 + i_3$$

...

$$e_k = e_{k-1} \times n_k + i_k$$

编译时刻确定的常数C,
保存在符号表中



涉及数组元素的赋值语句的翻译

——S属性定义

■ 赋值语句的文法：

(1) $S \rightarrow L := E$

(2) $L \rightarrow \text{id}$

(3) $L \rightarrow \text{id} [\text{Elist}]$

(4) $\text{Elist} \rightarrow E$

(5) $\text{Elist} \rightarrow \text{Elist}_1, E$

(6) $E \rightarrow E_1 + E_2$

(7) $E \rightarrow (E_1)$

(8) $E \rightarrow L$

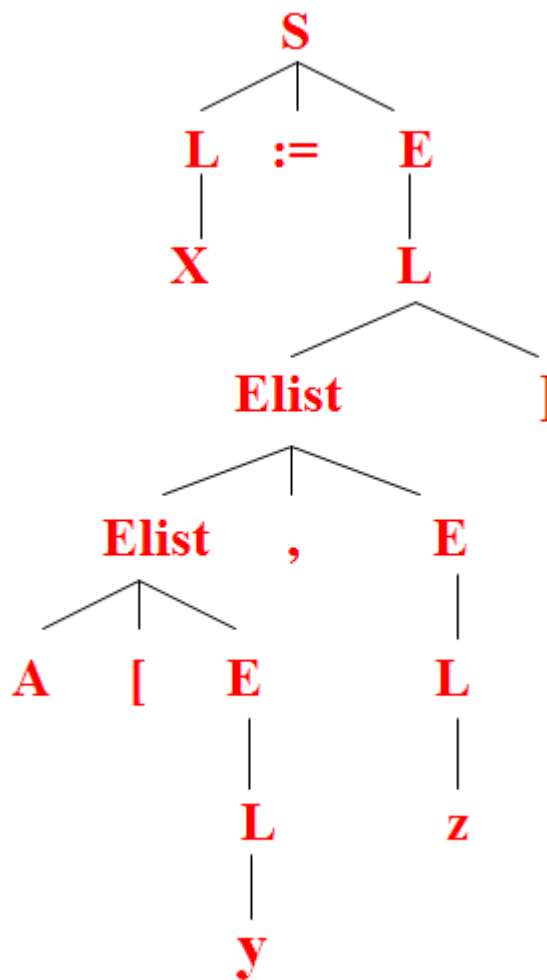
改写文法：

(3) $L \rightarrow \text{Elist}]$

(4) $\text{Elist} \rightarrow \text{id}[E$

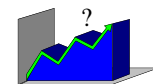
(5) $\text{Elist} \rightarrow \text{Elist}_1, E$

语句 $X := A [y, z]$ 的分析树



属性及函数设计

- **L 综合属性** L.entry 和 L.offset（符号表入口指针）
 - 简单变量：
L.offset=null L.entry=变量在符号表中的入口指针
 - 数组元素：（临时变量在符号表中的入口指针）
L.offset=计算公式第一项 L.entry=计算公式第二项 (base-C)



- **E 综合属性** E.entry，保存E值的变量在符号表中的位置
- **Elist 综合属性** Elist.array, ndim, entry
 - Elist.array：数组名在符号表中的位置
 - Elist.ndim：目前已经识别出的下标表达式的个数
 - Elist.entry：保存递推公式中 e_m 值的临时变量在符号表中的位置

■ 函数

- getaddr(array)：根据 指针array访问符号表，返回该表项中存放的数组空间的起始位置 base。
- limit(array, j)：返回array指向的数组的第 j 维的长度： n_j 。
- invariant(array)：返回array指向的数组的地址计算公式中的常量C。

S属性定义翻译方案

$S \rightarrow L := E$ { if (L.offset==null) /* L是简单变量 */
 outcode(L.entry ':=' E.entry);
 else outcode (L.entry '[' L.offset ']' ':=' E.entry); }

$E \rightarrow E_1 + E_2$ { E.entry=newtemp();
 outcode (E.entry ':=' E₁.entry '+' E₂.entry) }

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

$E \rightarrow L$ { if (L.offset == null) /* L是简单变量 */
 E.entry=L.entry;
 else { E.entry=newtemp();
 outcode (E.entry ':=' L.entry '[' L.offset ']'); }
 }

$L \rightarrow id$ { L.entry=id.entry; L.offset=null }

S属性定义翻译方案（续）

Elist → **id[E** { **Elist.entry=E.entry;**
Elist.ndim=1;
Elist.array=id.entry }

$$e_1 = i_1$$

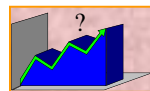
Elist → **Elist₁,E** { **t=newtemp();**
m=Elist₁.ndim+1;
outcode(t ':=' Elist₁.entry '×' limit(Elist₁.array,m));
outcode(t ':=' t '+' E.entry);
Elist.array=Elist₁.array;
Elist.entry=t;
Elist.ndim=m }

$$e_k = e_{k-1} \times n_k + i_k$$

base - C

L → **Elist]** { **L.entry=newtemp();**
outcode(L.entry ':=' getaddr(Elist.array) '-'
invariant(Elist.array));
L.offset=newtemp();
outcode(L.offset ':=' w '×' Elist.entry) }

$$e_m \times w$$



示例：翻译语句 $x := A[i, j]$

已知：

A是一个 10×20 的数组，即 $n_1=10$, $n_2=20$;

设数组元素的域宽 $w=4$;

设数组的第一个元素为：A[1,1],

则有 $low_1=1$, $low_2=1$

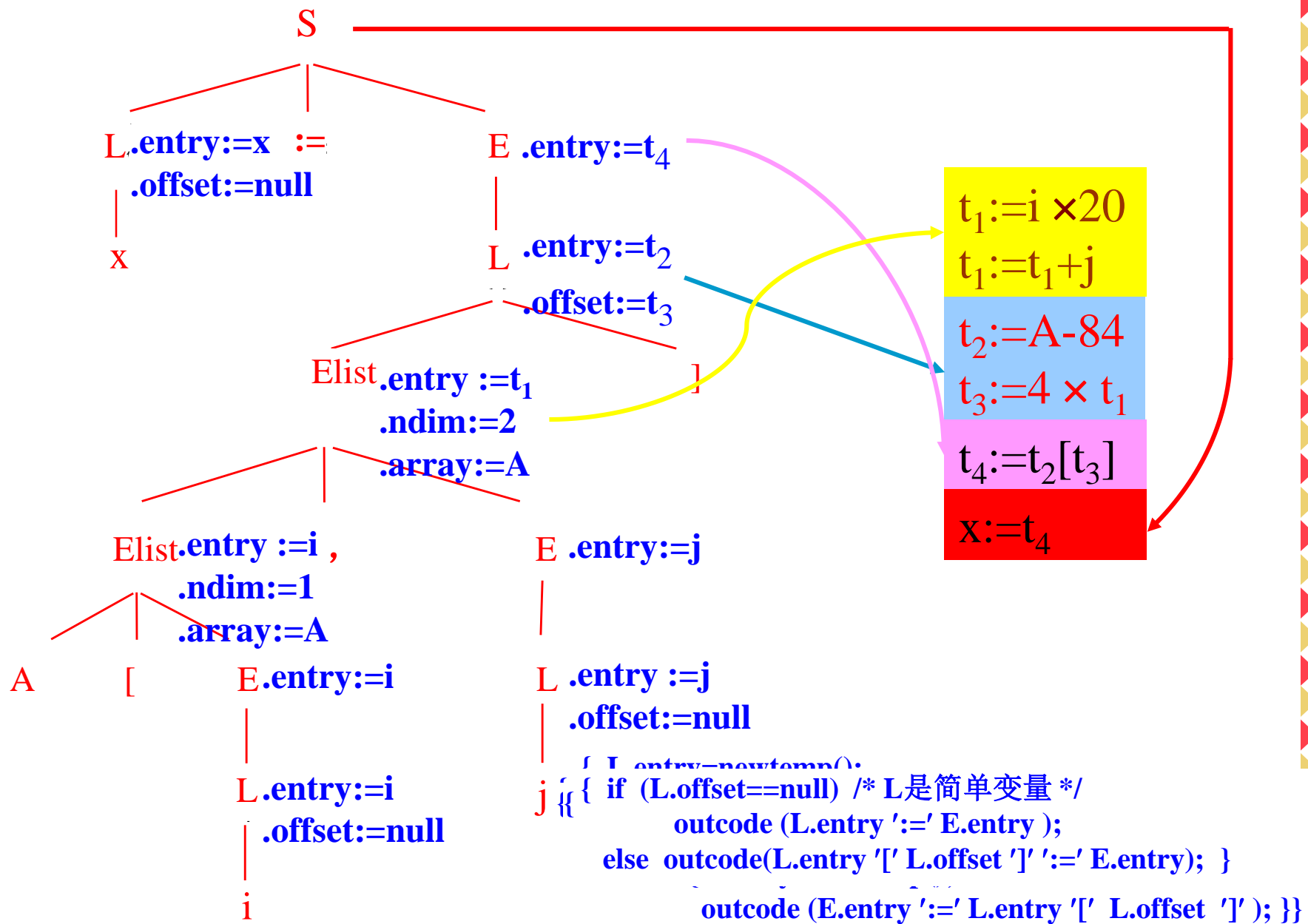
所以：

$$(low_1 \times n_2 + low_2) \times w = (1 \times 20 + 1) \times 4 = 84$$

要求：

将赋值语句 $x := A[i, j]$ 翻译为三地址代码。

赋值语句 $x:=A[i,j]$ 的翻译



3. 记录结构中域的访问

■ 声明:

```
p: record  
    info: integer;  
    x: real  
end;
```

name type address

...		
p	record(t)	48
...		

nil		12
info	int	0
x	real	4

■ 引用

```
p.info = p.info + 1;
```

■ 编译器的动作

lookup(p)

Gettype

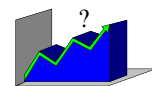
根据t, 找到记录的符号表

根据info在表中找



访问记录域的翻译动作

```
L→L1.L2 { L.entry=newtemp( );  
                if (L1.offset==null) L.entry= L1.entry;  
                else outcode( L.entry ':=' L1.entry '[' L1.offset ']' );  
                L.offset=newtemp( );  
                if (L2.offset==null) L.offset= L2.entry;  
                else outcode( L.offset ':=' L2.entry '[' L2.offset ']' );  
                }
```



8.3 布尔表达式的翻译

- 布尔表达式的作用
 - ◆ 计算逻辑值
 - ◆ 用作控制语句中的条件表达式
- 产生布尔表达式的文法

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

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

$E \rightarrow \text{not } E$

$E \rightarrow (E)$

$E \rightarrow \text{id rel op id}$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

8.3.1 翻译布尔表达式的方法

- 布尔表达式的真值的表示方法

- ◆ 数值表示法:

- 1 — true 0 — false

- 非0 — true 0 — false

- ◆ 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

- 布尔运算符的短路运算

- ◆ 短路运算, 如C、C++、java支持, Pascal不支持

- ◆ Ada语言, 非短路运算符: and, or

- 短路运算符: and then, or else

- 布尔表达式的翻译方法

- ◆ 数值表示法

- ◆ 控制流表示法

8.3.2 数值表示法

- 布尔表达式的求值类似于算术表达式的求值

■ 例如: $a \text{ or } \underbrace{\text{not } b}_{\textcircled{1}} \text{ and } c$

$\underbrace{\hspace{10em}}_{\textcircled{2}}$

$\underbrace{\hspace{15em}}_{\textcircled{3}}$

- 三地址代码

$t_1 := \text{not } b$

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

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

- 关系表达式 $x > y$

等价于:

if $x > y$

then 1

else 0

- $x > y$ 的三地址代码:

100: if $x > y$ goto 103

101: $t := 0$

102: goto 104

103: $t := 1$

104:



语义动作中变量、属性及函数说明

- 变量nextstat: 写指针, 指示输出序列中下一条三地址语句的位置。
- 属性E.entry: 存放布尔表达式E的真值的临时变量在符号表中的入口位置。
- 函数outcode(s): 根据nextstat的指示将三地址语句写到输出序列中。
 - ◆ outcode(s)输出一条三地址语句之后, nextstat自动加1。

数值表示法翻译方案

$E \rightarrow E_1 \text{ or } E_2$ { $E.\text{entry} = \text{newtemp}();$
 $\text{outcode}(E.\text{entry} ' := ' E_1.\text{entry} \text{ 'or' } E_2.\text{entry});$ }

$E \rightarrow E_1 \text{ and } E_2$ { $E.\text{entry} = \text{newtemp}();$
 $\text{outcode}(E.\text{entry} ' := ' E_1.\text{entry} \text{ 'and' } E_2.\text{entry});$ }

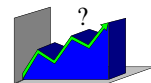
$E \rightarrow \text{not } E_1$ { $E.\text{entry} = \text{newtemp}();$
 $\text{outcode}(E.\text{entry} ' := ' \text{ 'not' } E_1.\text{entry});$ }

$E \rightarrow (E_1)$ { $E.\text{entry} = E_1.\text{entry};$ }

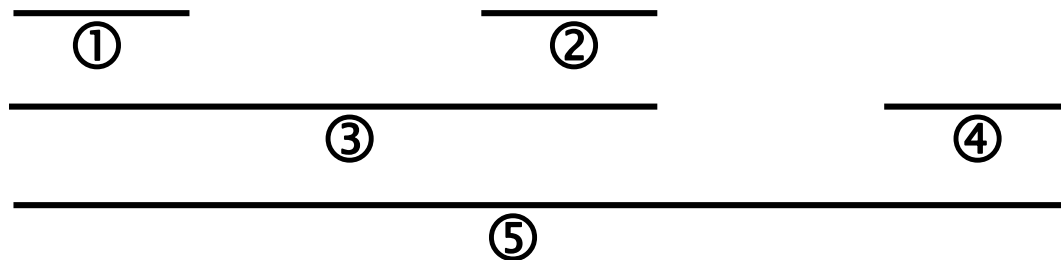
$E \rightarrow \text{id}_1 \text{ relop id}_2$ { $E.\text{entry} = \text{newtemp}();$
 $\text{outcode}(\text{ 'if' id}_1.\text{entry relop.op id}_2.\text{entry} \text{ 'goto' nextstat}+3);$
 $\text{outcode}(E.\text{entry} ' := ' \text{ '0' });$
 $\text{outcode}(\text{ 'goto' nextstat}+2);$
 $\text{outcode}(E.\text{entry}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

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

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



举例: $a > b$ and $c > d$ or $e < f$



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: $t_3 := t_1$ and t_2

109: if $e < f$ goto 112

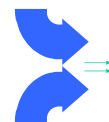
110: $t_4 := 0$

111: goto 113

112: $t_4 := 1$

113: $t_5 := t_3$ or t_4

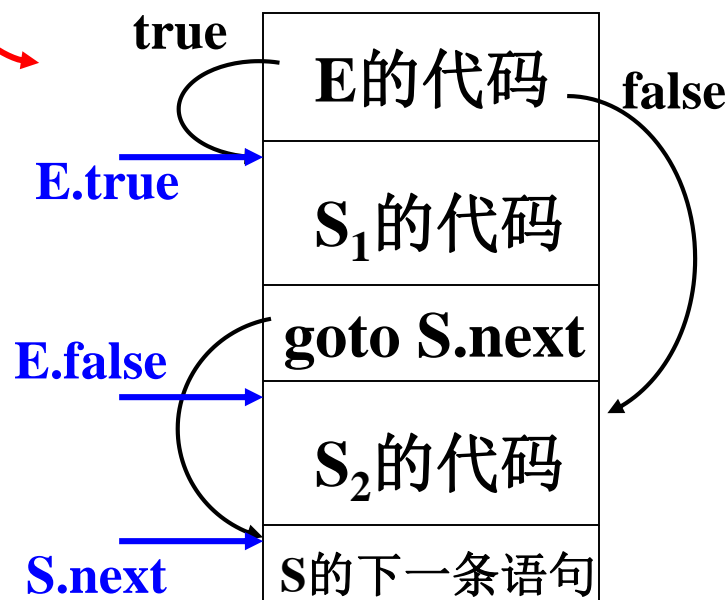
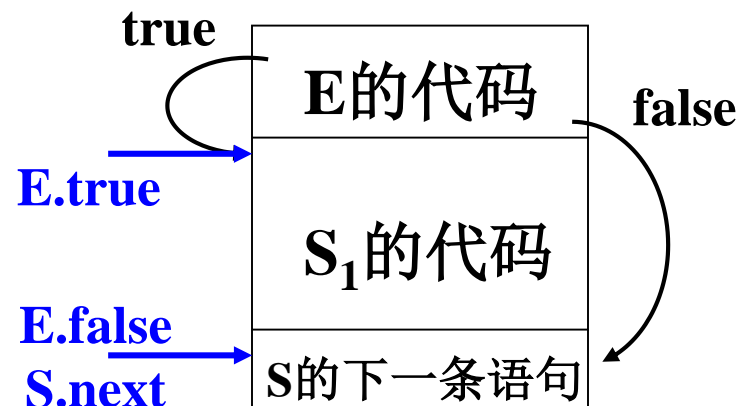
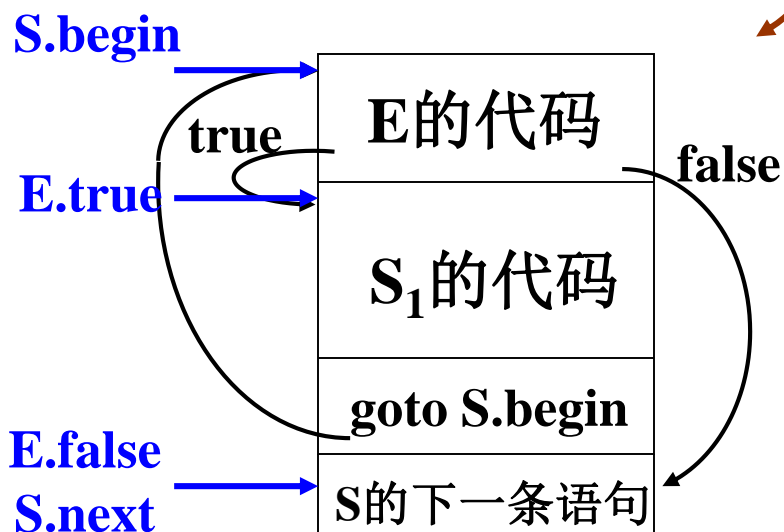
8.3.3 控制流表示法



■ 控制语句

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

■ 控制语句的代码结构



变量、属性及函数说明

- 函数newlabel(): 产生并返回一个新的语句标号。
- 继承属性：三地址语句标号
 - ◆ E.true: E的值为真时应执行的第一条语句的标号
 - ◆ E.false: E的值为假时应执行的第一条语句的标号
 - ◆ S.next: 紧跟在语句S之后的下一条三地址语句的标号
 - ◆ S.begin: 语句S的第一条三地址语句的标号

控制流表示法翻译布尔表达式

- 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
- 这些语句转移到的位置是E.true、E.false之一
- 例如 $a < b$ 翻译为：
 if $a < b$ goto E.true
 goto E.false
- 属性说明
 - ◆ 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

控制流翻译方法的基本思想

- 条件表达式 $x > y$ 翻译为：

if $x > y$ goto E.true
goto E.false

- $E \rightarrow id_1 \text{ relop } id_2$

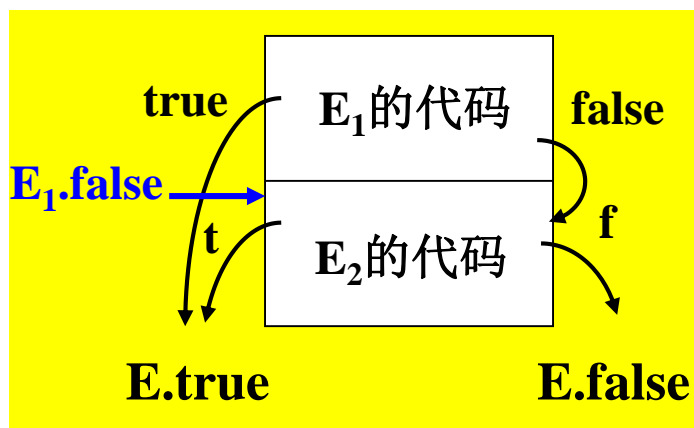
'if' $id_1.entry \text{ relop.op } id_2.entry$ 'goto' E.true
'goto' E.false

- 将布尔表达式E翻译为一系列条件转移和无条件转移三地址语句。

- ◆ 转移语句转移到的位置是 E.true 或者 E.false
- ◆ E的值为真或为假时，控制转移到的位置

布尔表达式的代码结构（短路运算）

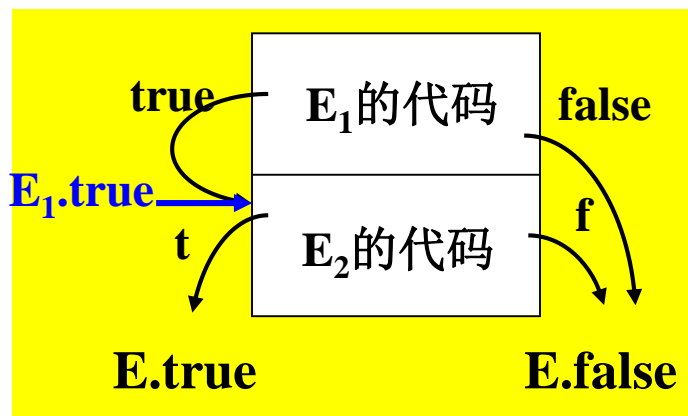
$E \rightarrow E_1 \text{ or } E_2$



$E \rightarrow \text{not } E_1$



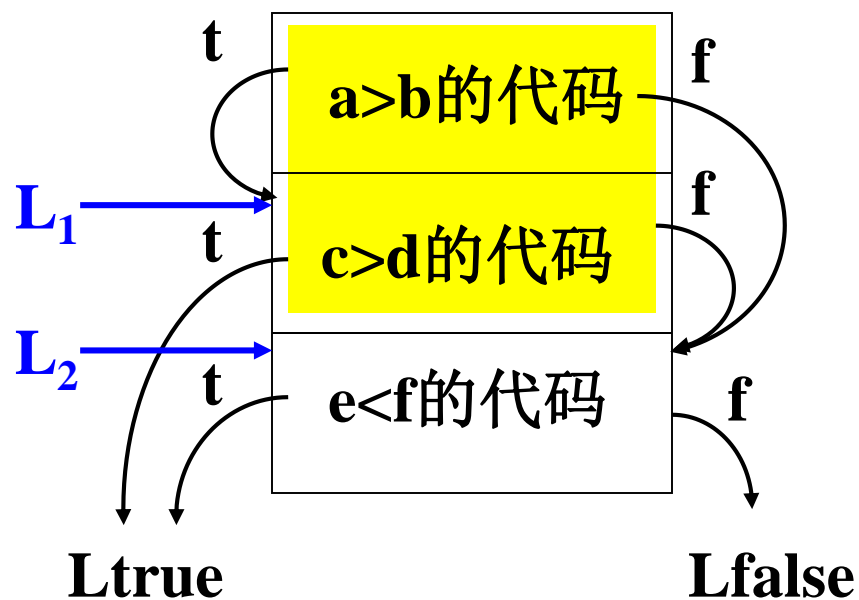
$E \rightarrow E_1 \text{ and } E_2$



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

**'if' id₁.entry relop.op id₂.entry 'goto' E.true
'goto' E.false**

例: $a > b$ and $c > d$ or $e < f$ 的代码结构及三地址语句



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

只有一个是真的

控制流表示法翻译布尔表达式

- 布尔表达式的真假出口位置不但与表达式本身的结构有关，还与表达式出现的上下文有关。
- 考虑表达式 “ $a > b$ or $c > d$ ” 和 “ $a > b$ and $c > d$ ”，“ $a > b$ ” 的真假出口依赖于：
 - ◆ 布尔表达式的结构
 - ◆ 布尔表达式所在控制语句的结构
- 两遍扫描的翻译技术
 - Pass 1. 生成分析树
 - Pass 2. 为分析树加注释——翻译
- 可否在一遍扫描过程中，同时完成分析和翻译？

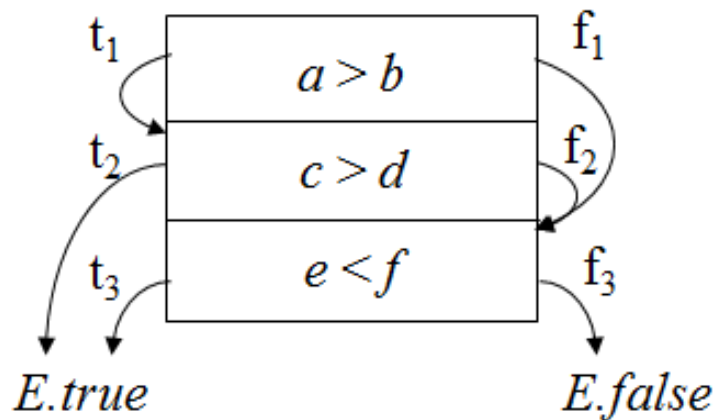
问题：当生成某些转移指令时，目标地址可能还不知道。

8.3.4 控制流表示法翻译布尔表达式 ——回填技术

- 先产生没有填写目标标号的转移指令；
- 建立一个链表，把转向这个目标的所有转移指令的标号填入该链表；
- 目标地址确定后，再把目标地址填入该链表中记录的所有转移指令中。

例：用回填技术翻译 $a > b$ and $c > d$ or $e < f$

$a > b$ and $c > d$ or $e < f$
① ② ④
③
⑤



~~.t={102}~~
~~.f={101, 103}~~

$a > b$ and $c > d$

~~.t={100}~~
~~.f={101}~~

100: if $a > b$ goto 102

101: goto 104

~~.t={102}~~
~~.f={103}~~

102: if $c > d$ goto t_2

103: goto 104

~~.t={104}~~
~~.f={105}~~

104: if $e < f$ goto t_3

105: goto f_3

$a > b$ and $c > d$ or $e < f$

~~.t={102, 104}~~
~~.f={105}~~

利用回填技术翻译布尔表达式

■ 布尔表达式文法

$E \rightarrow E_1 \text{ or } ME_2$

$E \rightarrow E_1 \text{ and } ME_2$

$E \rightarrow \text{not } E_1$

$E \rightarrow (E_1)$

$E \rightarrow id_1 \text{ relop } id_2$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

$M \rightarrow \varepsilon$

■ 说明

– 三地址语句用**四元式**表示

– 四元式存放在数组中

– 数组下标：三地址语句的标号

■ 变量nextquad：记录将要产生的下一条三地址语句在四元式数组中的位置

■ 标记非终结符号**M**

– 标识 E_2 的开始位置

– 属性 $M.\text{quad}$ ，记录 E_2 的第一条三地址语句的地址

– **$M \rightarrow \varepsilon$ 的动作：**

$M.\text{quad} = \text{nextquad}$

属性定义及函数说明

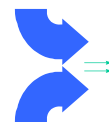
■ 综合属性

- ◆ E.truelist: 记录转移到E的真出口的指令链表的指针
- ◆ E.falselist: 记录转移到E的假出口的指令链表的指针
- ◆ M.quad: M所标识的三地址语句的地址

■ 函数

- ◆ makelist(i): 建立新链表, 其中只包括待回填的转移指令在数组中的位置 i, 返回所建链表的指针。
- ◆ merge(p_1, p_2): 合并由指针 p_1 和 p_2 所指向的两个链表, 返回结果链表的指针。
- ◆ 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。
- ◆ outcode(S): 产生一条三地址语句S, 并写入输出数组中, 该函数执行完后, 变量 nextquad 加 1。

布尔表达式的翻译方案



$E \rightarrow E_1 \text{ or } ME_2$ { backpatch(E_1 .falselist, M.quad);
E.truelist= merge(E_1 .truelist, E_2 .truelist);
E.falselist= E_2 .falselist; }

$E \rightarrow E_1 \text{ and } ME_2$ { backpatch(E_1 .truelist, M.quad);
E.truelist= E_2 .truelist;
E.falselist= merge(E_1 .falselist, E_2 .falselist); }

$E \rightarrow \text{not } E_1$ { E.truelist= E_1 .falselist; E.falselist= E_1 .truelist; }

$E \rightarrow (E_1)$ { E.truelist= E_1 .truelist; E.falselist= E_1 .falselist; }

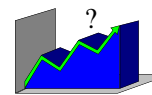
$E \rightarrow id_1 \text{ relop } id_2$ { E.truelist=makelist(nextquad);
E.falselist=makelist(nextquad+1);
outcode('if' id_1 .entry relop.op id_2 .entry 'goto —');
outcode('goto —'); }

$E \rightarrow \text{true}$ { E.truelist=makelist(nextquad); outcode('goto —'); }

$E \rightarrow \text{false}$ { E.falselist=makelist(nextquad); outcode('goto —'); }

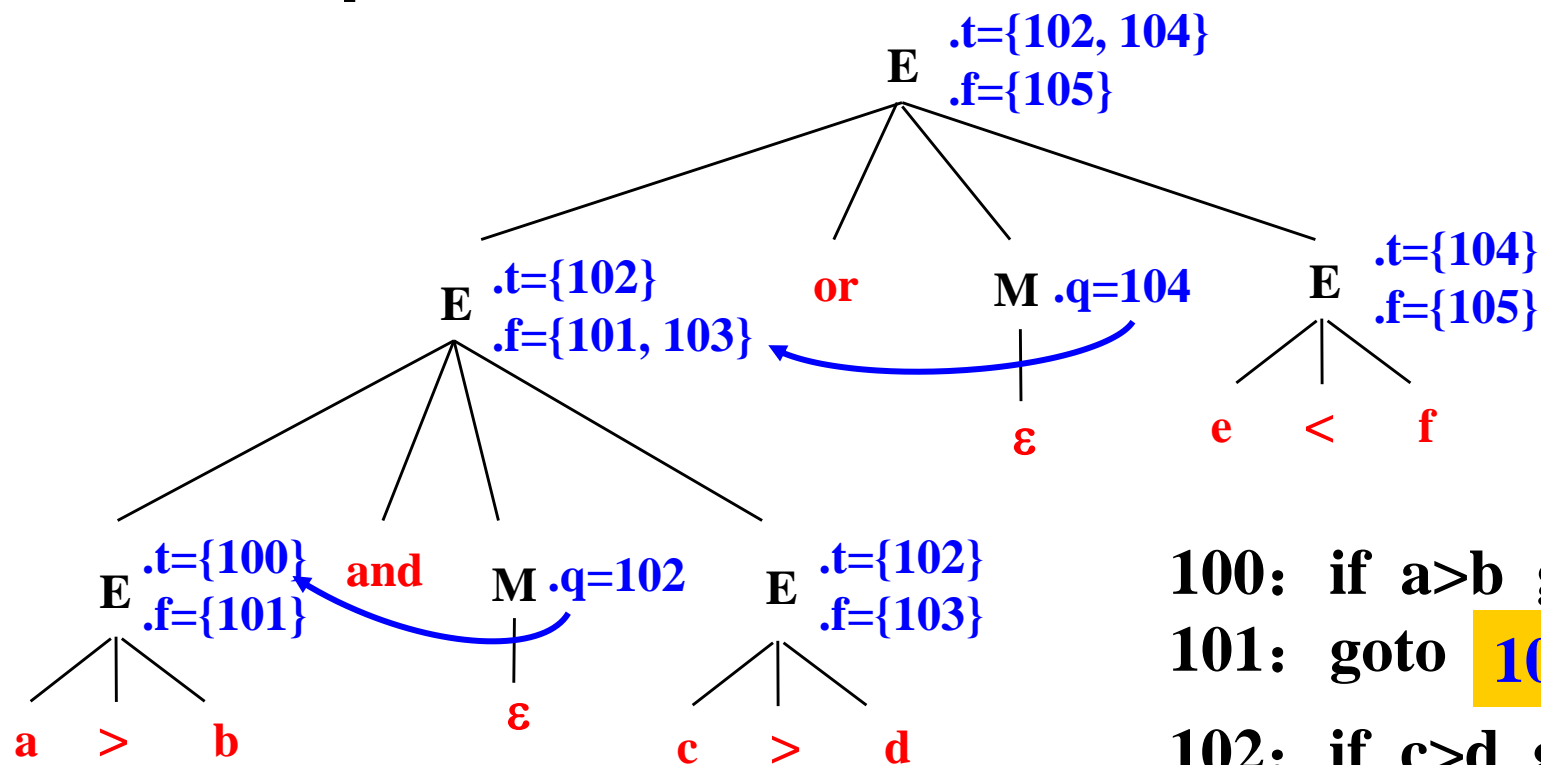
$M \rightarrow \epsilon$ { M.quad=nextquad; }

利用翻译方案翻译布尔表达式



$a > b$ and $c > d$ or $e < f$

假定nextquad的当前值为100



100: if $a > b$ goto 102

101: goto 104

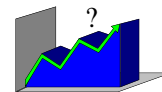
102: if $c > d$ goto —

103: goto 104

104: if $e < f$ goto —

105: goto —

8.4 控制语句的翻译



■ 文法

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

$S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$

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

$S \rightarrow A$

$\text{Slist} \rightarrow \text{Slist}_1; \ M \ S$

$\text{Slist} \rightarrow S$

$M \rightarrow \epsilon$

$N \rightarrow \epsilon$

转移到下一条语句
的指令链表的指针

属性:

$E.\text{truelist}$

$E.\text{falselist}$

$M.\text{quad}$

$S.\text{nextlist}$

$\text{Slist}.\text{nextlist}$

$N.\text{nextlist}$

变量: nextquad

函数:

$\text{makelist}(i)$

$\text{backpatch}(p, i)$

$\text{merge}(p_1, p_2)$

$\text{outcode}(s)$

▲ 记录变量 nextquad 的当前, 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



控制语句的翻译方案

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

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

$M \rightarrow \epsilon \ \{ M.\text{quad} = \text{nextquad} \}$

$N \rightarrow \epsilon \ \{ N.\text{nextlist} = \text{makelist}(\text{nextquad}); \ \text{outcode}('goto \text{---}'); \}$

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1 \ \{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{quad});$
 $\quad \text{backpatch}(E.\text{truelist}, M_2.\text{quad});$
 $\quad S.\text{nextlist} = E.\text{falselist};$
 $\quad \text{outcode}('goto' \ M_1.\text{quad}); \}$

$S \rightarrow \text{begin } Slist \ \text{end} \ \{ S.\text{nextlist} = Slist.\text{nextlist}; \}$

$S \rightarrow A \ \{ S.\text{nextlist} = \text{makelist}(); \}$

$Slist \rightarrow Slist_1; \ M \ S \ \{ \text{backpatch}(Slist_1.\text{nextlist}, M.\text{quad});$
 $\quad Slist.\text{nextlist} = S.\text{nextlist} \}$

$Slist \rightarrow S \ \{ Slist.\text{nextlist} = S.\text{nextlist} \}$

例:

if $a > b$ and $c > d$ or $e < f$ then $\overset{106}{\triangle} A_1$ $\overset{116}{\triangle}$ else $\overset{117}{\triangle} A_2$;

① ② ③

⑧

$\overset{127}{\triangle}$ while $\overset{127}{\triangle} a < b$ do $\overset{129}{\triangle} A_3$

⑤ ⑥

⑦



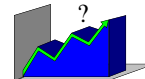
~~.t={102,104}~~
~~.f={105}~~

~~.n={116}~~

~~.t={127}~~
~~.f={128}~~

```

100: if a>b goto 102
101: goto 104
102: if c>d goto 106
103: goto 104
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
    
```



小结

■ 中间语言

◆ 图形表示

- 树、dag

◆ 三地址代码

- 三地址语句的形式: $x := y \text{ op } z$
- 三地址语句的种类
 - 简单赋值语句
 - 涉及数组元素的赋值语句
 - 涉及指针的赋值语句
 - 转移语句
 - 过程调用语句
- 三地址语句的具体实现
 - 三元式、四元式、间接三元式

小结（续1）

■ 赋值语句的翻译

- ◆ 文法（赋值语句出现的环境）
- ◆ 仅涉及简单变量的赋值语句的翻译
- ◆ 涉及数组元素的赋值语句的翻译
 - 计算数组元素的地址
- ◆ 访问记录中的域

■ 布尔表达式的翻译

- ◆ 数值方法
- ◆ 控制流方法：代码结构
- ◆ 回填技术
 - 思想、问题、方法
 - 与链表操作有关的函数
 - makelist
 - merge
 - backpatch
 - 属性设计
 - 布尔表达式的翻译

■ 控制语句的翻译