

# 编译原理与技术

## 第8章：中间代码生成

王吴凡

北京邮电大学计算机学院

主页：[cswwf.github.io](https://cswwf.github.io)

邮箱：[wufanwang@bupt.edu.cn](mailto:wufanwang@bupt.edu.cn)

# 教学内容、目标与要求

## ■ 教学内容

- 中间代码形式
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译

## ■ 教学目标与要求

- 了解中间代码的形式及四元式实现；
- 理解赋值语句的翻译方案；
- 掌握回填技术；
- 理解控制语句的目标代码结构；
- 理解利用回填技术翻译布尔表达式及控制语句的翻译方案。

## ■ 教学目标与要求（续）

### □ 能够

- 分析中间代码生成的需求；
- 利用语法制导翻译技术设计中间代码生成的翻译方案；
- 利用翻译方案对输入符号串进行翻译，验证方案的有效性并得到翻译结果。

# 中间代码生成程序

## ■ 任务：

把经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。

## ■ 在编译程序中的位置：



## ■ 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

# 内容目录

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

8.5 goto语句的翻译 (\*)

8.6 CASE语句的翻译 (\*)

小 结

# 8.1 中间代码形式

## 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与语法树同样的信息。
- 在dag中，公共子表达式被标识出。

## 2. 三地址代码

### □ 三地址语句的形式

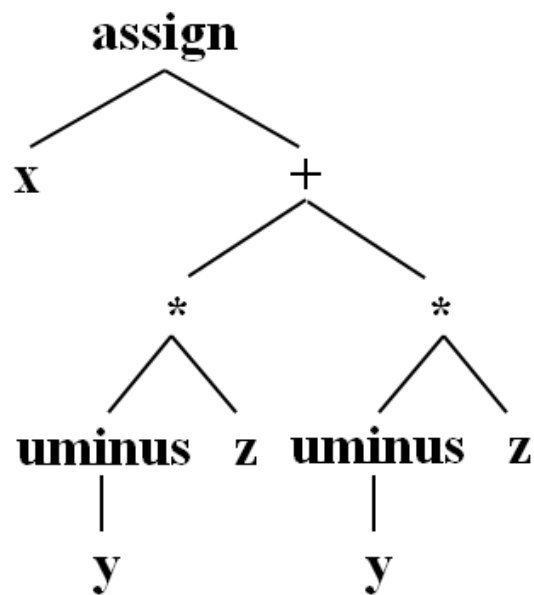
### □ 三地址语句的种类

### □ 三地址语句的实现

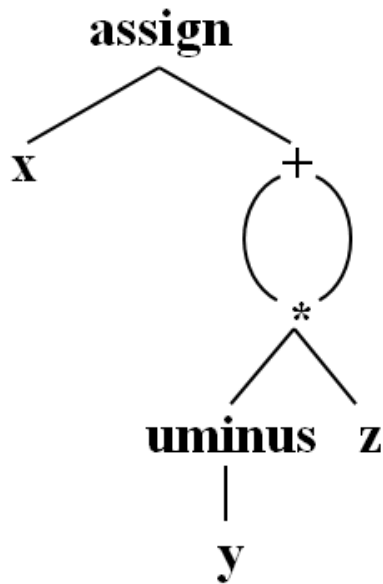
# 1. 图形表示

## ■ $x := (-y) * z + (-y) * z$ 的图形表示

### □ 语法树表示



### □ dag图形表示



## ■ 后缀式：语法树的线性表示形式

□ 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。

## ■ 对应语法树的后缀式：

$x \ y \ \text{uminus} \ z \ * \ y \ \text{uminus} \ z \ * \ + \ \text{assign}$

# 为赋值语句构造语法树的语法制导定义

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

## 2. 三地址代码

- 三地址代码：三地址语句组成的序列。
  - 类似于汇编代码
- 三地址语句的一般形式：  $x := y \text{ op } z$ 
  - $x$ ：名字、临时变量
  - $y$ 、 $z$ ：名字、常数、或临时变量
  - $\text{op}$ ：运算符号，如算术运算符、或逻辑运算符等
- 实现时，语句中的名字，指向该名字在符号表中表项的指针。



# 三地址语句的种类及形式

## ■ 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

□ goto L

□ if x relop y goto L

## ■ 过程调用语句

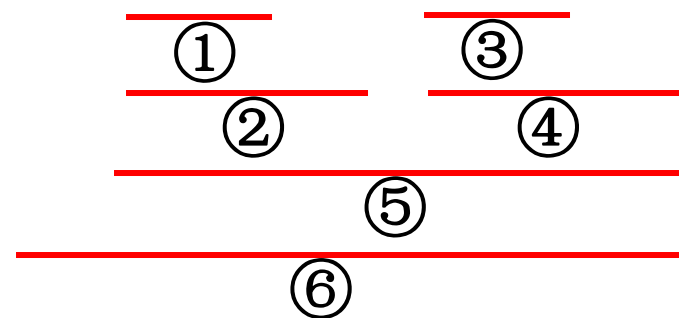
□ param x

□ call p, n

## ■ 返回语句

□ return y

■ 例如  $x := (-y) * z + (-y) * z$



## ■ 语法树的代码

(1)  $t_1 := -y$

(2)  $t_2 := t_1 * z$

(3)  $t_3 := -y$

(4)  $t_4 := t_3 * z$

(5)  $t_5 := t_2 + t_4$

(6)  $x := t_5$

## ■ dag的代码

$t_1 := -y$

$t_2 := t_1 * z$

$t_5 := t_2 + t_2$

$x := t_5$

# 三地址语句的四元式实现

## ■ 四元式

(op, arg<sub>1</sub>, arg<sub>2</sub>, result)      如:  $x := y + z$       ('+', y, z, x)  
(op, arg<sub>1</sub>, , result)      如:  $x := -y$       ('uminus', y, , x)  
(param, arg<sub>1</sub>, , )      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 <sub>1</sub>	arg <sub>2</sub>	result
(0)	uminus	y		t <sub>1</sub>
(1)	*	t <sub>1</sub>	z	t <sub>2</sub>
(2)	uminus	y		t <sub>3</sub>
(3)	*	t <sub>3</sub>	z	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	:=	t <sub>5</sub>		x

## 8.2 赋值语句的翻译

- 假定赋值语句出现的环境如下文法所描述：

$P \rightarrow \textcircled{M} D; S$

$M \rightarrow \varepsilon$

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

$N \rightarrow \varepsilon$

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

$L \rightarrow \varepsilon$

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

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

设计函数：

(1)  $p = \text{lookup}(\text{id.name})$

(2)  $\text{gettype}(p)$

(3)  $\text{newtemp}()$

(4)  $\text{outcode}(s)$

# 1. 仅涉及简单变量的赋值语句

## ■ 文法

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

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

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

## ■ 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

$p = \text{lookup}(\text{id.name})?$

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

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

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

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

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

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

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

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

## ■ 扩充符号表：

名字	类型	值存在?	值
t	real/ integer	T / F	value

## ■ $E \rightarrow \text{num}$

{  $E.\text{entry} = \text{newtemp}();$   
 $E.\text{type} = \text{integer};$   
 $\text{update}(E.\text{entry}, E.\text{type}, \text{'T'}, \text{value});$  }

## ■ $E \rightarrow \text{num.num}$

{  $E.\text{entry} = \text{newtemp}();$   
 $E.\text{type} = \text{real};$   
 $\text{update}(E.\text{entry}, E.\text{type}, \text{'T'}, \text{value});$  }

# 同时进行类型检查的翻译方案

- 假设, 仅考虑类型 **integer** 和 **real**
- $E \rightarrow id$     { **p=lookup(id.name);**  
                  **if (p!=nil) { E.entry=p;**  
                          **E.type=gettype(p); }**  
                  **else { E.type=type\_error; error(); } }**
- $E \rightarrow (E_1)$     { **E.entry=E<sub>1</sub>.entry;**  
                      **E.type=E<sub>1</sub>.type; }**
- $E \rightarrow -E_1$     { **E.entry=newtemp();**  
                  **if (E<sub>1</sub>.type==integer) || (E<sub>1</sub>.type==real) {**  
                      **outcode(E.entry ':=' 'uminus' E<sub>1</sub>.entry);**  
                      **E.type=E<sub>1</sub>.type ; }**  
                  **else { E.type=type\_error; error(); }**  
                  **}**

# $E \rightarrow E_1 + E_2$ 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 '+' 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; error(); }  
}
```

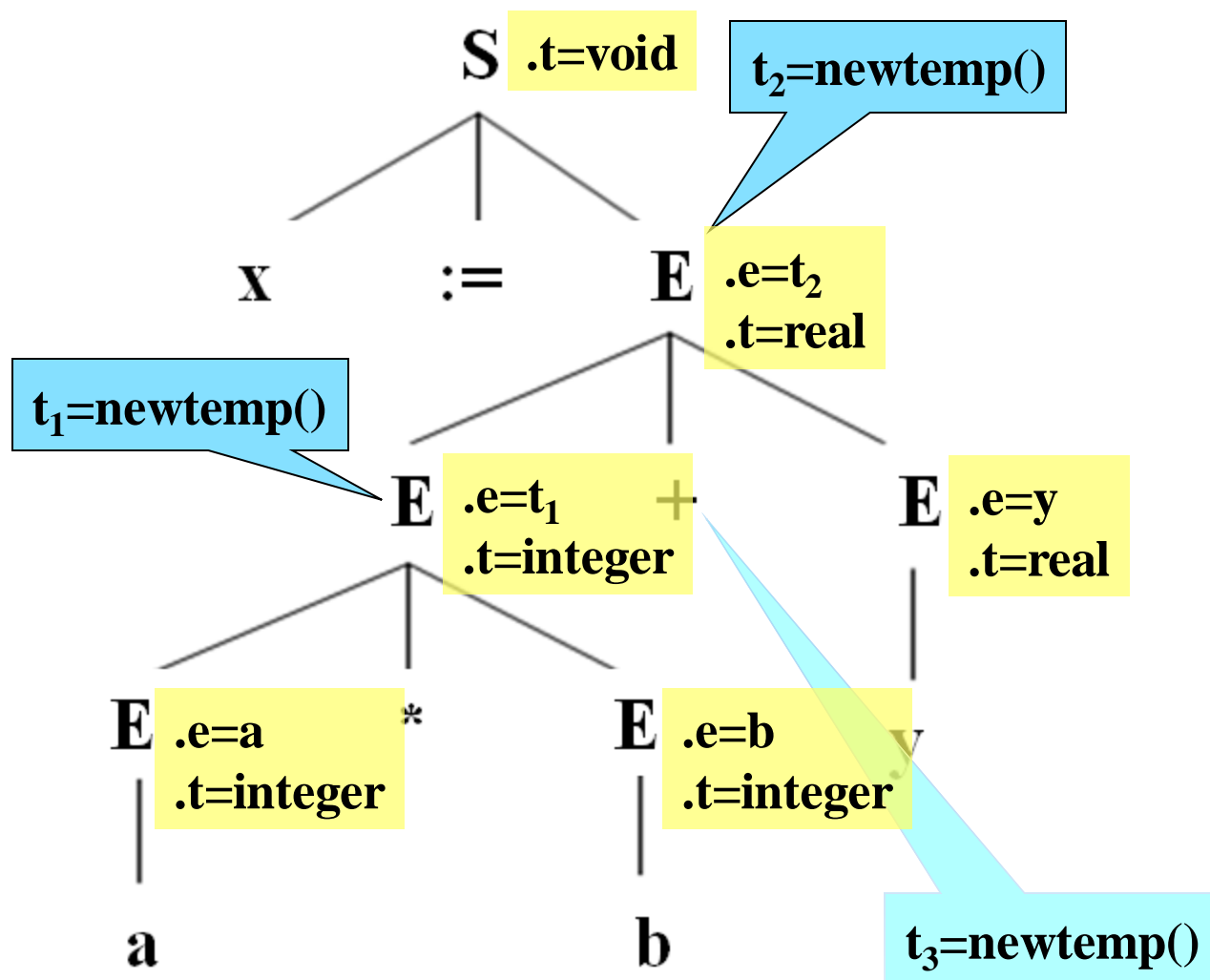
# $S \rightarrow id := E$ 带有类型检查的语义动作

```
{ 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; error(); }  
  };  
  else error();  
}
```



# 翻译赋值语句 $x := a * b + y$

- 假定x和y的类型为real, a和b的类型为integer

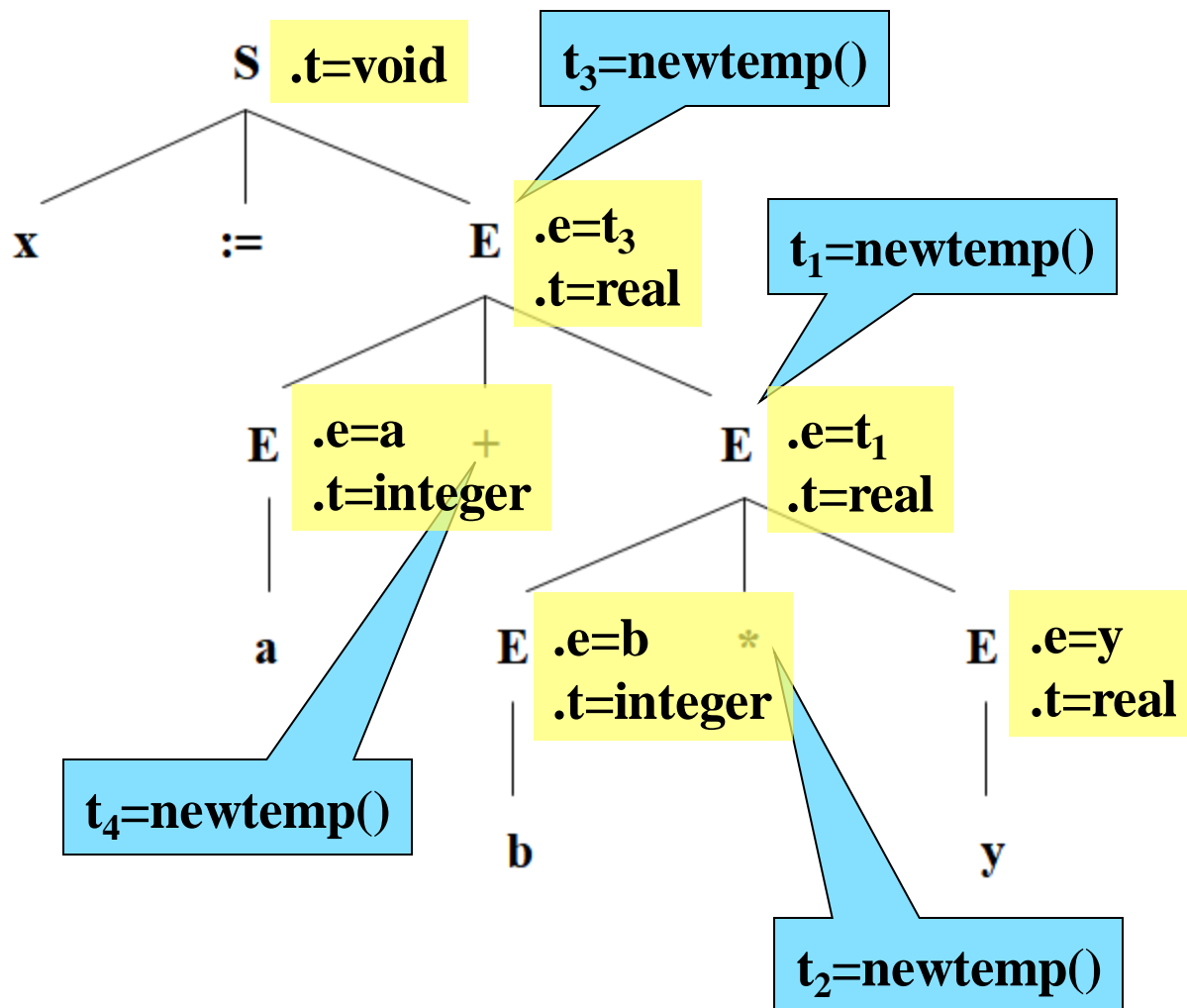


- ### ■ 三地址代码：

$$t_1 := a \text{ int}^* b$$
$$t_3 := \text{inttoreal } t_1$$
$$\mathbf{t}_2 := \mathbf{t}_3 \text{ real+ } \mathbf{y}$$
$$\mathbf{x} := \mathbf{t}_2$$

# 翻译赋值语句 $x:=a+b*y$

- 假定 $x$ 和 $y$ 的类型为 $\text{real}$ ,  $a$ 和 $b$ 的类型为 $\text{integer}$



- 三地址代码:

$t_2 := \text{intto real } b$

$t_1 := t_2 \text{ real} * y$

$t_4 := \text{intto real } a$

$t_3 := t_4 \text{ real} + t_1$

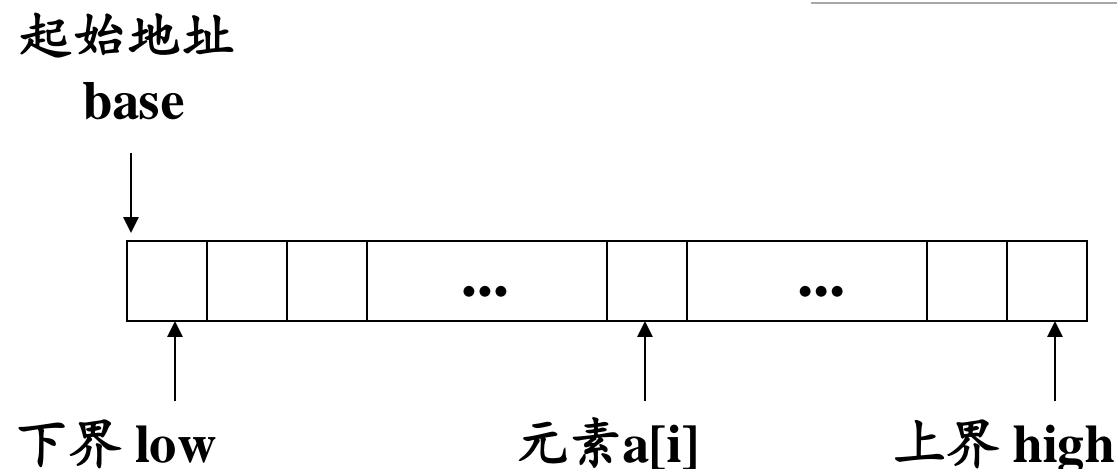
$x := t_3$

## 2. 涉及数组元素的赋值语句

## 一维数组--a[i]的地址

### ■ 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地查找每个元素。
- 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 $A[i_1, i_2, \dots, i_k]$

### ■ 数组元素个数： $high-low+1$

### ■ 数组元素 $a[i]$ 的位置：

$$\begin{aligned} & \text{base} + (i - \text{low}) \times w \\ &= i \times w + \text{base} - \text{low} \times w \quad \text{常数 } C \end{aligned}$$

# 二维数组--a[i, j]的地址

## ■ 二维数组 a[m, n]

$a_{11}$	$a_{12}$	...	$a_{1j}$	...	$a_{1n}$
$a_{21}$	$a_{22}$	...	$a_{2j}$	...	$a_{2n}$
...					
$a_{i1}$	$a_{i2}$	...	$a_{ij}$	...	$a_{in}$
...					
$a_{m1}$	$a_{m2}$	...	$a_{mj}$	...	$a_{mn}$

每维的下界:  $low_1$ 、 $low_2$

每维的上界:  $high_1$ 、 $high_2$

每维的长度:  $m=high_1-low_1+1$

$n=high_2-low_2+1$

## ■ 存储方式:

□ 按行优先存放

□ 按列优先存放

## ■ 数组元素a[i, j]的位置:

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

$$= (i \times n + j) \times w +$$

$$base - (low_1 \times n + low_2) \times w \quad \text{常数 } C$$

# 三维数组--a[i, j, k]的地址

## ■ 三维数组

- 按行优先存放

- 每维的下界:

$\text{low}_1$ 、 $\text{low}_2$ 、 $\text{low}_3$

- 每维的上界:

$\text{high}_1$ 、 $\text{high}_2$ 、 $\text{high}_3$

- 每维的长度:

$n_1 = \text{high}_1 - \text{low}_1 + 1$

$n_2 = \text{high}_2 - \text{low}_2 + 1$

$n_3 = \text{high}_3 - \text{low}_3 + 1$

## ■ 数组元素a[i, j, k]的位置:

$$\begin{aligned} & \text{base} + (((i - \text{low}_1) \times n_2 + (j - \text{low}_2)) \times n_3 + (k - \text{low}_3)) \times w \\ &= ((i \times n_2 + j) \times n_3 + k) \times w \\ &+ \text{base} - ((\text{low}_1 \times n_2 + \text{low}_2) \times n_3 + \text{low}_3) \times w \quad \text{常数C} \end{aligned}$$

# 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:

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

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

常数

递归计算:

$$e_1 = i_1$$

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

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

...

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

动态数组?  
常数 C?

# 涉及数组元素的赋值语句的翻译

## ■ 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)  $\text{Elist} \rightarrow E$

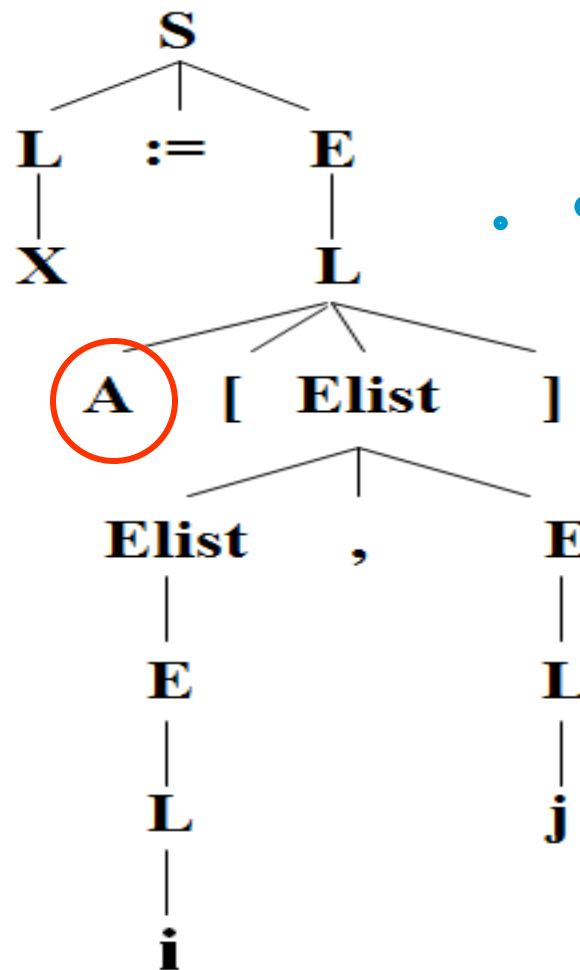
(5)  $\text{Elist} \rightarrow \text{Elist}_1, E$

(6)  $E \rightarrow E_1 + E_2$

(7)  $E \rightarrow (E_1)$

(8)  $E \rightarrow L$

语句  $X := A[i, j]$  的分析树



继承属性?

$n_2$  ?

$$\begin{aligned} e_1 &= i \\ e_2 &= e_1 \times n_2 + j \end{aligned}$$

# 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)  $\text{Elist} \rightarrow \text{Elist}_1, E$

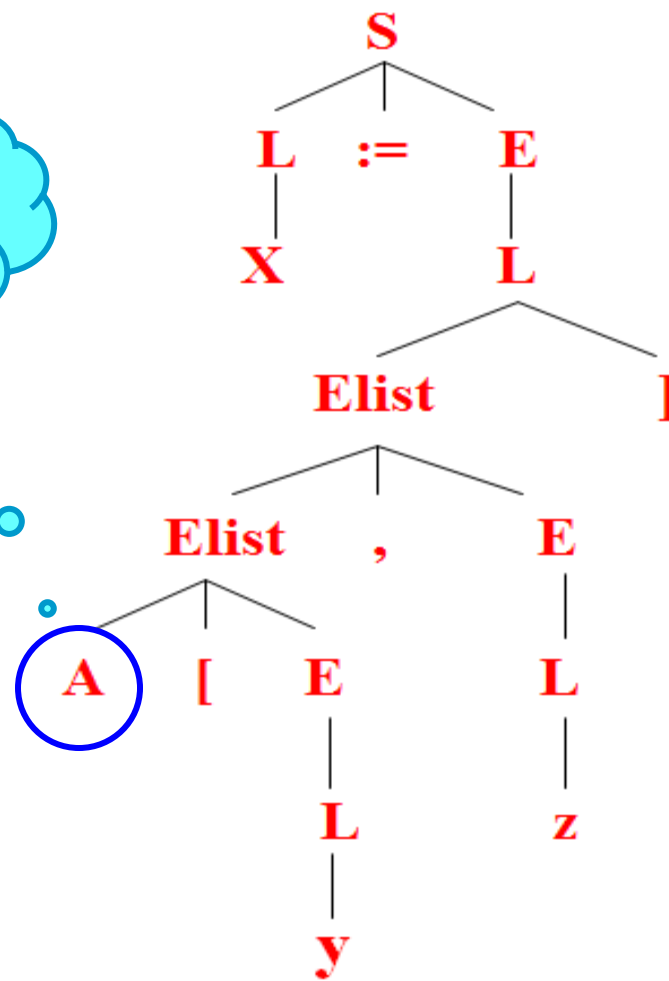
(6)  $E \rightarrow E_1 + E_2$

(7)  $E \rightarrow (E_1)$

(8)  $E \rightarrow L$

语句  $X := A[y, z]$  的分析树

综合  
属性



$n_2$  ?

改写文法:

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

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

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

$$\begin{aligned} e_1 &= y \\ e_2 &= e_1 \times n_2 + z \end{aligned}$$



# S属性定义翻译方案：属性及函数设计

## ■ L 综合属性 L.entry 和 L.offset

- 简单变量:  $L.offset = \text{null}$        $L.entry = \text{符号表入口指针}$
- 数组元素:  $L.offset = \text{公式第一项}$        $L.entry = \text{公式第二项}$

## ■ 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（访问符号表）

- getaddr(array): 返回 array 指向的数组的空间起始位置 base
- limit(array, j): 返回 array 指向的数组第 j 维的长度
- invariant(array): 返回 array 指向的数组的地址计算公式中的常数 C


# S属性定义翻译方案

```
S → 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); }  
;
```

```
L → id { L.entry = id.entry; L.offset = null; }  
;
```

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

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



$e_1 = i_1$

# S属性定义翻译方案

```
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.ndim=m;  
    Elist.entry=t; }
```

$e_2 = e_1 \times n_2 + i_2$   
 $e_3 = e_2 \times n_3 + i_3$   
...  
 $e_k = e_{k-1} \times n_k + i_k$

```
E→E1+E2 { E.entry=newtemp();  
    outcode(E.entry ':=' E1.entry '+' E2.entry); }
```

```
E→(E1) { E.entry=E1.entry; }
```

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

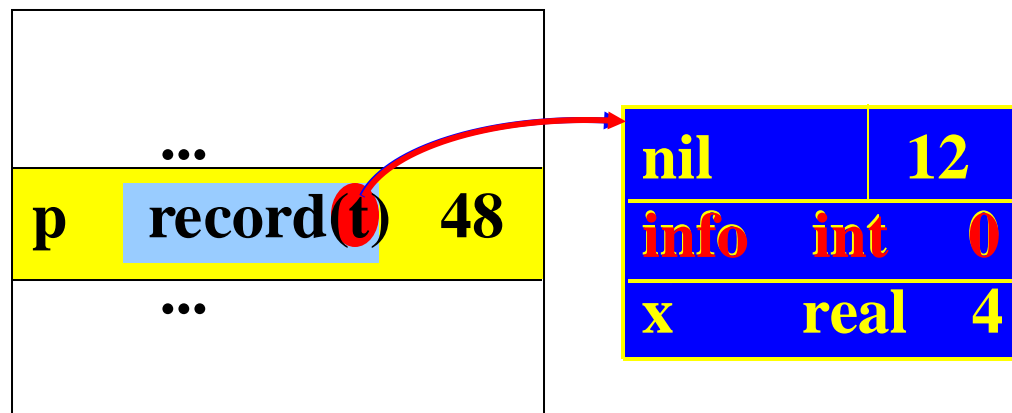


### 3. 记录结构中域的访问

#### ■ 声明:

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

name    type    address



#### ■ 编译器的动作

```
ptr=lookup(p)
```

```
gettype(ptr)
```

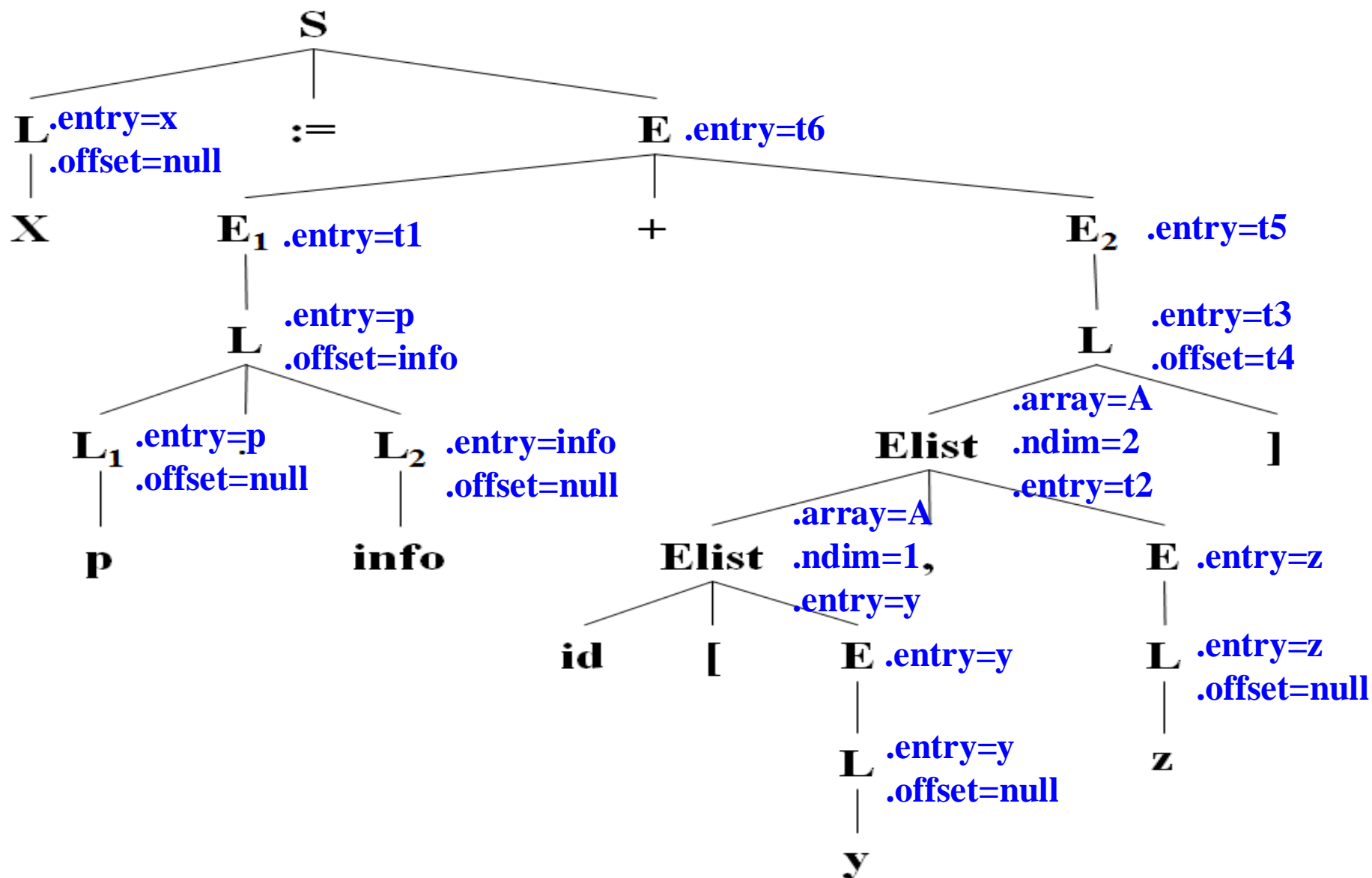
由t找到记录的符号表  
根据info在表中找

#### 语句:

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

```
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 ']' );
               }
```

# 翻译语句 $X := p.info + A[y, z]$



**t1:=p[info]**

**t2:=y\*20**

**t2:=t2+z**

**t3:=A-84**

**t4:=4\*t2**

**t5:=t3[t4]**

**t6:=t1+t5**

**X:=t6**



## 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

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

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

$E \rightarrow \text{not } E$

$E \rightarrow (E)$

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

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

## 1. 翻译布尔表达式的方法

### ■ 真值的表示方法

- 数值表示法:

➤ 1/非0 — true    0 — false

- 控制流表示法:

用控制流到达的位置表示 true 或 false

### ■ 短路运算

- 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

### ■ 翻译方法

- 数值表示法
- 控制流表示法

## 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s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

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

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

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

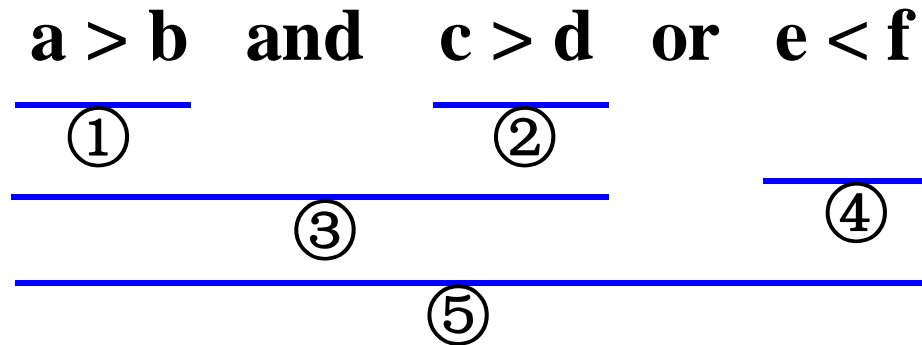
$E \rightarrow (E_1)$             {  $E.entry = E_1.entry;$  }

$E \rightarrow id_1 \text{ relop } id_2$  {  $E.entry = \text{newtemp}();$   
                               $\text{outcode}(\text{'if' } id_1.entry \text{ relop.op } id_2.entry \text{ 'goto' nextstat+3});$   
                               $\text{outcode}(E.entry \text{ ':=' '0'});$   
                               $\text{outcode}(\text{'goto' nextstat+2});$   
                               $\text{outcode}(E.entry \text{ ':=' '1'});$     }

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

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

# 示例



100: if a>b goto 103

101: t<sub>1</sub>:=0

102: goto 104

103: t<sub>1</sub>:=1

104: if c>d goto 107

105: t<sub>2</sub>:=0

106: goto 108

107: t<sub>2</sub>:=1

108: t<sub>3</sub>:=t<sub>1</sub> and t<sub>2</sub>

109: if e<f goto 112

110: t<sub>4</sub>:=0

111: goto 113

112: t<sub>4</sub>:=1

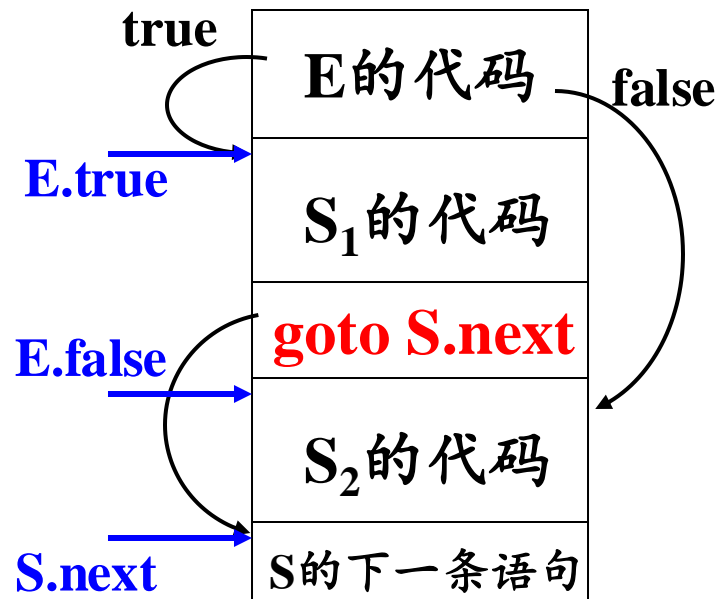
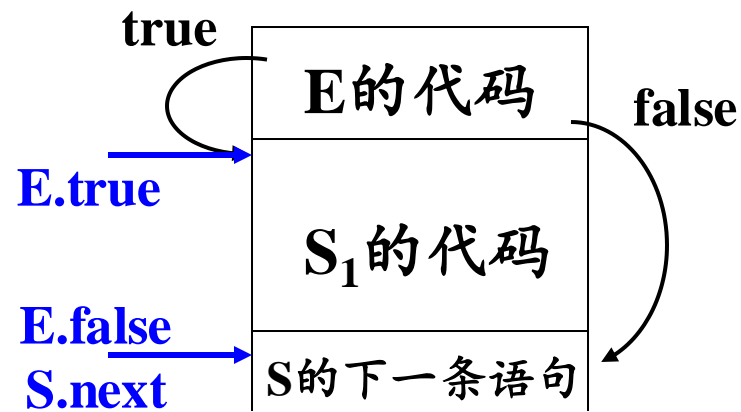
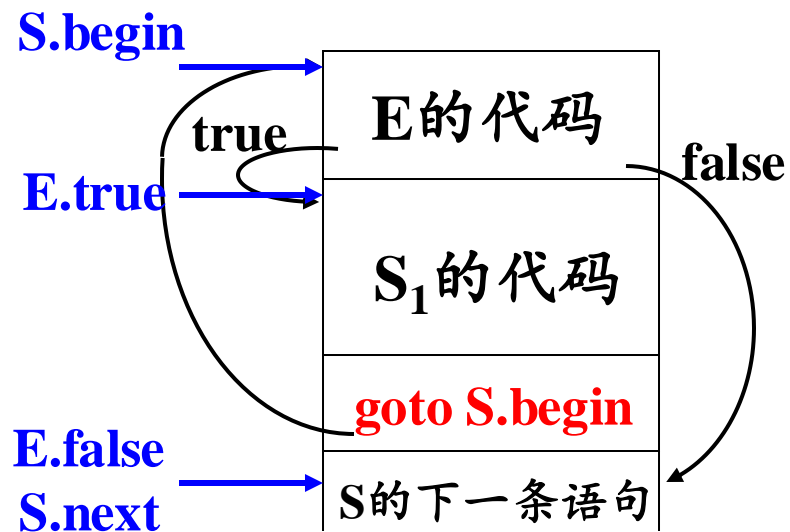
113: t<sub>5</sub>:=t<sub>3</sub> or t<sub>4</sub>

### 3. 控制流表示法及回填技术

#### ■ 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



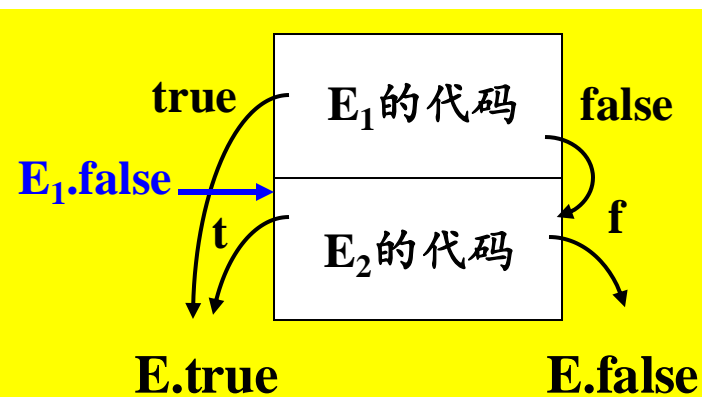
# 控制流表示法翻译布尔表达式

- 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
  - 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
- 如  $a < b$  翻译为:  
if  $a < b$  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

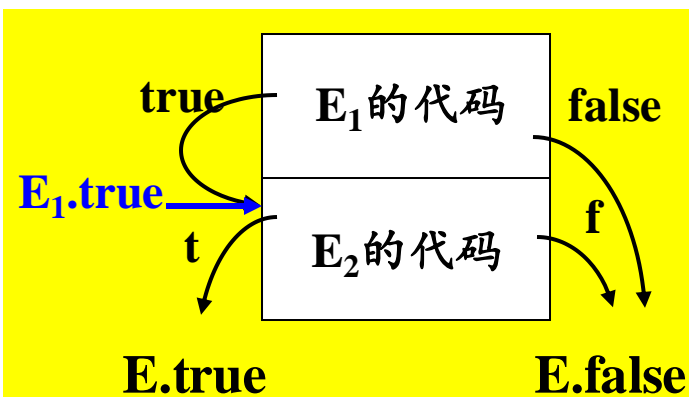
# 布尔表达式的代码结构

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

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



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

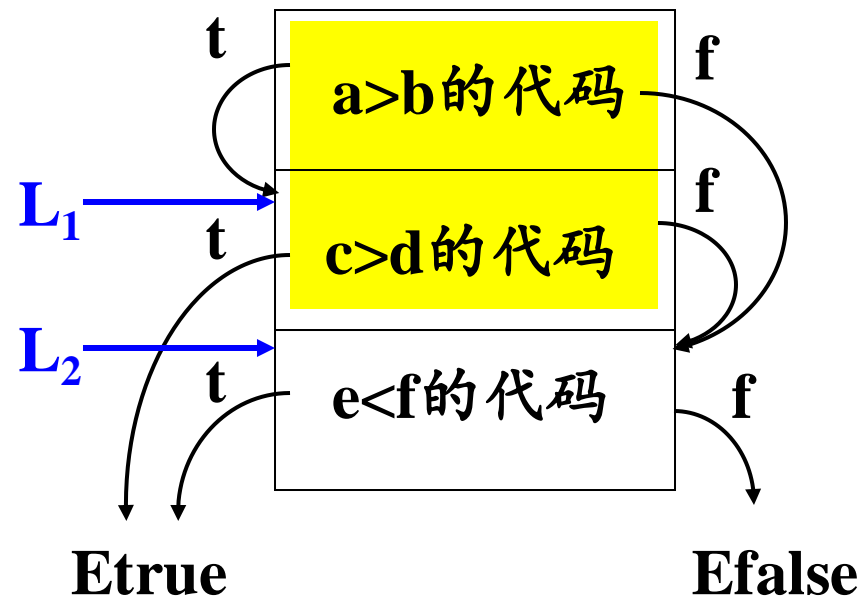


$E \rightarrow \text{not } E_1$



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

'if'  $\text{id}_1.\text{entry relop.op id}_2.\text{entry}$  'goto'  $E.\text{true}$   
'goto'  $E.\text{false}$



```
if a>b goto L1
goto L2
L1: if c>d goto Etrue
    goto L2
L2: if e<f goto Etrue
    goto Efalse
```

# 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“ $a > b \text{ or } c > d$ ”和“ $a > b \text{ and } c > d$ ”，“ $a > b$ ”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

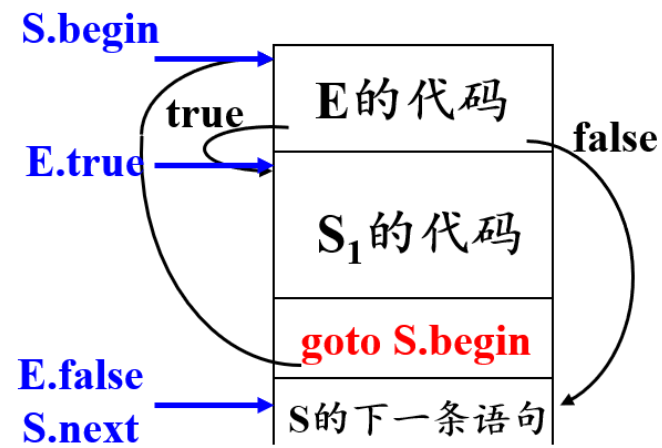
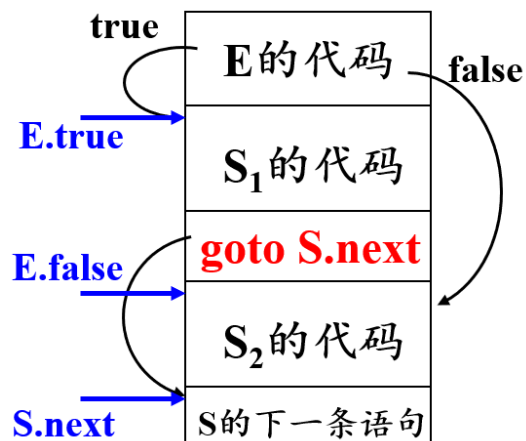
## ■ 两遍扫描的翻译技术

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。

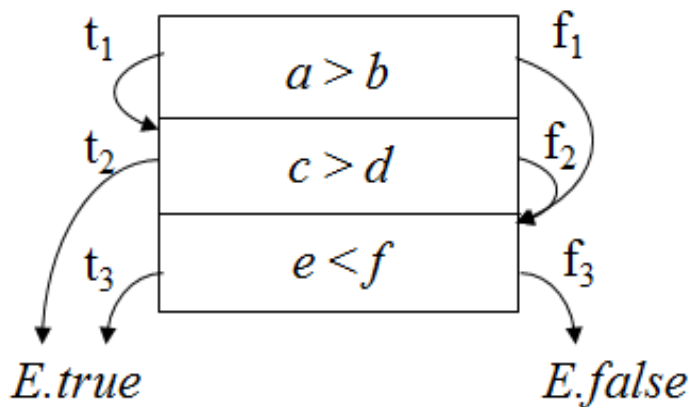


# 翻译布尔表达式——回填技术

- 先产生没有填写目标标号的转移指令；
- 建立一个链表，把转向此位置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指令中。

- 用回填技术翻译  
 $a > b \text{ and } c > d \text{ or } e < f$

$$\frac{\frac{a > b}{(1)} \text{ and } \frac{c > d}{(2)} \text{ or } \frac{e < f}{(4)}}{(3)} \quad (5)$$



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

$a > b \text{ and } c > d$

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

100: if  $a > b$  goto 102  
101: goto 104

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

102: if  $c > d$  goto 103  
103: goto 104

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

104: if  $e < f$  goto 105  
105: goto

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

.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

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

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

$E \rightarrow \text{not } E_1$

$E \rightarrow (E_1)$

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

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

$M \rightarrow \epsilon$

## ■ 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.quad, 记录 $E_2$ 的第一条三地址语句的地址

□  $M \rightarrow \epsilon$  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的真出口位置的指令链表的**指针**
- **E.falselist:**  
记录转移到E的假出口位置的指令链表的**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  $p_1$  和  $p_2$  所指的两个链表, 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 -'); }

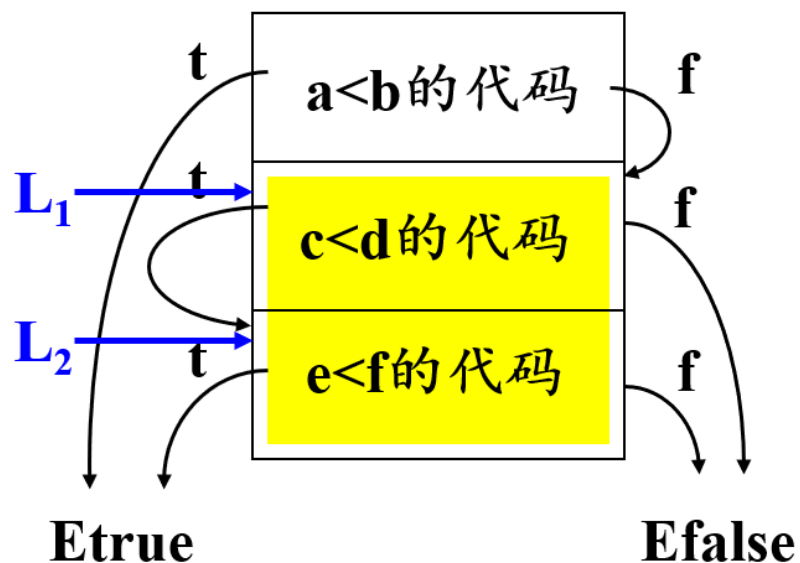
$E \rightarrow \text{true}$  {  $E$ .truelist=makelist(nextquad);  $E$ .falselist=NULL; outcode('goto -'); }

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

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

# 示例

翻译:  $a < b$  or  $c < d$  and  $e < f$   
假定nextquad的初值为100



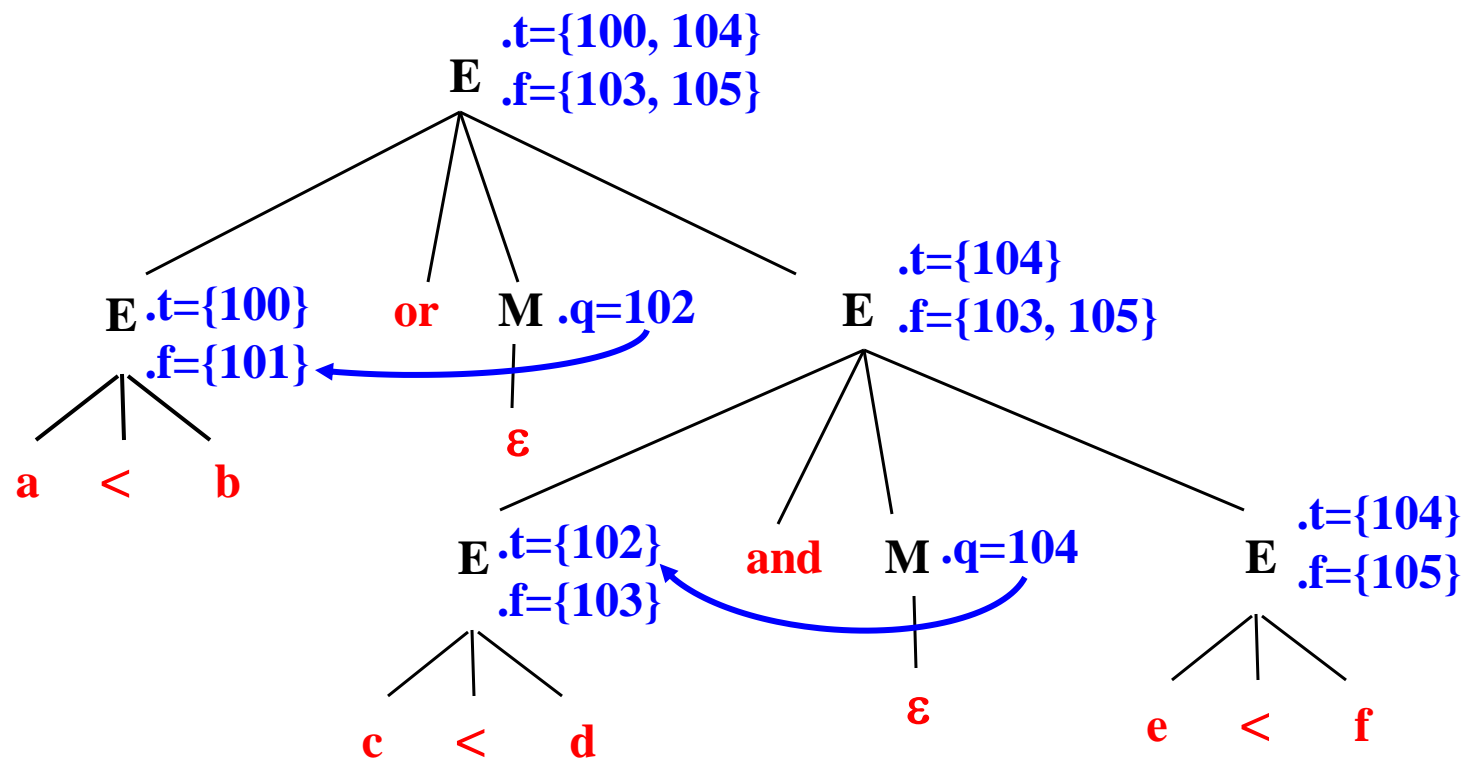
E.truelist={100, 104}  
E.falselist={103, 105}

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

```
100: if a<b goto —
101: goto 102
102: if c<d goto 104
103: goto —
104: if c<d goto —
105: goto —
```

# 示例

利用LR技术分析并翻译： $a < b$  or  $c < d$  and  $e < f$   
假定nextquad的初值为100



100: if  $a < b$  goto —

101: goto 102

102: if  $c < d$  goto 104

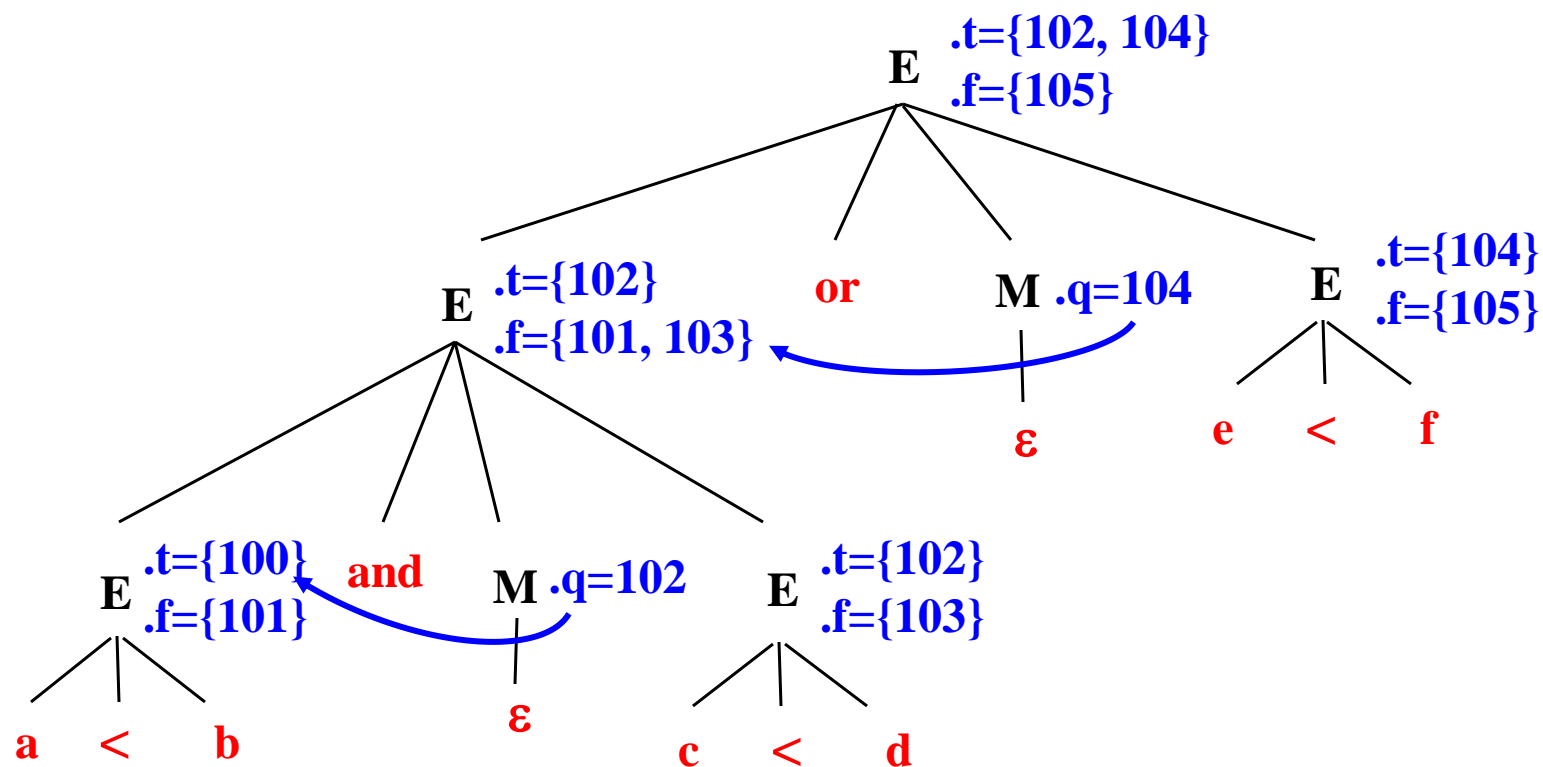
103: goto —

104: if  $e < f$  goto —

105: goto —

# 示例

利用LR技术分析并翻译： $a < b$  and  $c < d$  or  $e < f$   
假定nextquad的初值为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

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

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

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

$S \rightarrow \text{begin } S_{\text{list}} \ \text{end}$

$S \rightarrow A$

$S_{\text{list}} \rightarrow S_{\text{list}}_1; \ \mathbf{M} \ S$

$S_{\text{list}} \rightarrow S$

$\mathbf{M} \rightarrow \epsilon$

$\mathbf{N} \rightarrow \epsilon$

属性:

$E.\text{truelist}$

$E.\text{falselist}$

$M.\text{quad}$

$S.\text{nextlist}$

$S_{\text{list}}.\text{nextlist}$

$N.\text{nextlist}$

变量:  $\text{nextquad}$

函数:

$\text{makelist}(i)$

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

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

$\text{outcode}(s)$

转移到下一条  
语句的语句链  
表的指针

▲ 记录变量  $\text{nextquad}$  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

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

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

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

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

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

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

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

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

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



## 例：

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

$\overset{127}{\blacktriangle}$  while  $\overset{127}{\blacktriangle} \underline{a < b}$  do  $\overset{129}{\blacktriangle} \underline{A_3}$   
 $\textcircled{5}$   $\textcircled{6}$   
 $\textcircled{7}$

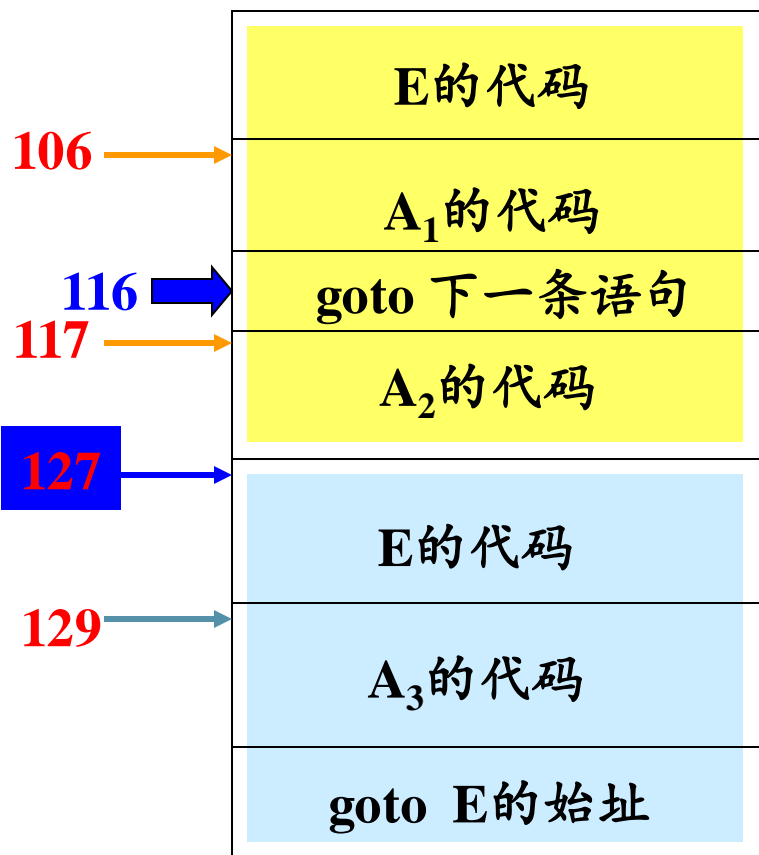


Diagram illustrating a control flow graph with three basic blocks (A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>) and their associated instructions. The instructions are numbered 100 through 139.

**Block A<sub>1</sub> (Instructions 100-105):**

- 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

**Block A<sub>2</sub> (Instructions 106-117):**

- 106: (Block Header)
- 115: (Block Header)
- 116: goto 127
- 117: (Block Header)
- 126: (Block Header)

**Block A<sub>3</sub> (Instructions 126-139):**

- 126: (Block Header)
- 127: if a<b goto 129
- 128: goto —
- 129: (Block Header)
- 138: (Block Header)
- 139: goto 127

**Callout Boxes (Instruction Sets):**

- Box 1 (Left of A<sub>1</sub>):  $\{102, 104\}$  and  $\{105\}$
- Box 2 (Left of A<sub>2</sub>):  $\{116\}$
- Box 3 (Left of A<sub>3</sub>):  $\{127\}$  and  $\{128\}$



## 8.5 goto语句的翻译

### ■ goto 语句的一般形式

- goto lable
- if expr goto lable

### ■ 语句标号的出现形式

- 定义性出现，形式为 lable: stmt
- 引用性出现，作为转移目标出现在 goto语句中

### ■ 程序中应用形式:

先定义后引用:  
lable: stme;  
...  
goto lable

先引用后定义:  
goto lable;  
...  
lable: stme;

### ■ 标号的声明

- Pascal: 使用前先声明
- C语言, 不要求

### ■ 符号表中的语句标号

名字	类型	定义标志	地址
<i>L</i>	Label	F	-1

名字	类型	定义标志	地址
<i>L</i>	Label	T	<i>V</i>

# 用户定义的循环控制机制

## ■ continue

```
while (sum<1000) {  
    getNext(value);  
    if (value<0) continue;  
    sum+=value;  
}
```

## ■ break

```
while (sum<1000) {  
    getNext(value);  
    if (value<0) break;  
    sum+=value;  
}
```

## 8.6 CASE语句的翻译

### ■ Pascal语言的CASE语句

```
case E of  
    V1: S1 ;  
    V2: S2 ;  
    ...  
    Vn-1: Sn-1 ;  
    [else Sn ;]  
end
```

### ■ C语言的CASE语句

```
switch (E) {  
    case V1: S1 ; break;  
    case V2: S2 ; break;  
    ...  
    case Vn-1: Sn-1 ; break;  
    [default: Sn ; ]  
}
```

- 对情况表达式E求值。
- 在列出的常量V<sub>1</sub>、V<sub>2</sub>、...、V<sub>n-1</sub>中寻找与表达式E的值相等的值V<sub>i</sub>。
  - 如果不存在这样的值，则让“默认值”与之匹配（如果有缺省分支的话）。
- 执行与找到的值V<sub>i</sub>相联系的语句S<sub>i</sub>。

# CASE语句的代码结构

对E求值的代码

把求值结果置于临时变量 t 中

if  $t \neq V_1$  goto  $L_2$

$S_1$  的代码

goto next

$L_2$ : if  $t \neq V_2$  goto  $L_3$

$S_2$  的代码

goto next

$L_3$ : ...

$L_{n-1}$ : if  $t \neq V_{n-1}$  goto  $L_n$

$S_{n-1}$  的代码

goto next

$L_n$ :  $S_n$  的代码

next:

控制结构复杂  
goto语句生成  
时不完整

对E求值的代码

把求值结果置于临时变量 t 中

goto test

$L_1$ :  $S_1$  的代码

goto next

$L_2$ :  $S_2$  的代码

goto next

$L_3$ : ...

$L_{n-1}$ :  $S_{n-1}$  的代码

goto next

$L_n$ :  $S_n$  的代码

goto next

test: if  $t = V_1$  goto  $L_1$

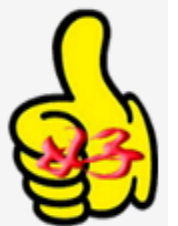
if  $t = V_2$  goto  $L_2$

...

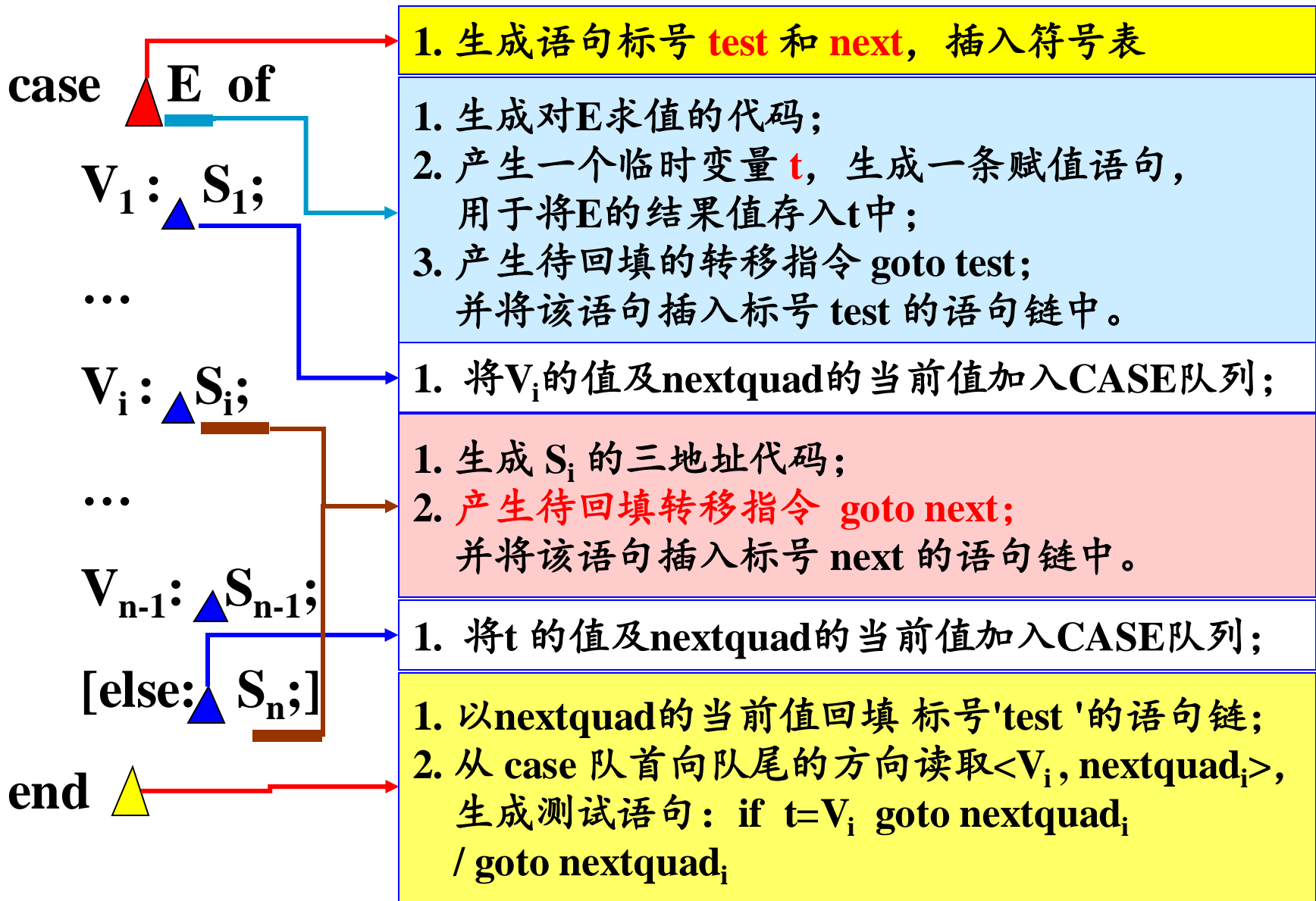
if  $t = V_{n-1}$  goto  $L_{n-1}$

goto  $L_n$

next:

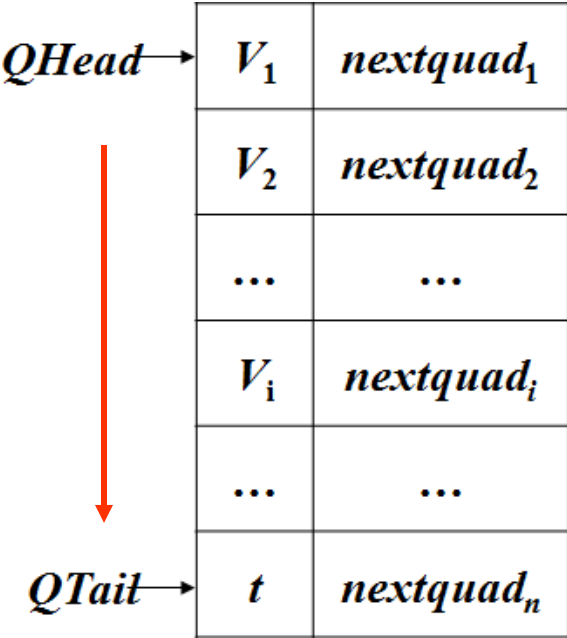


# CASE语句的翻译

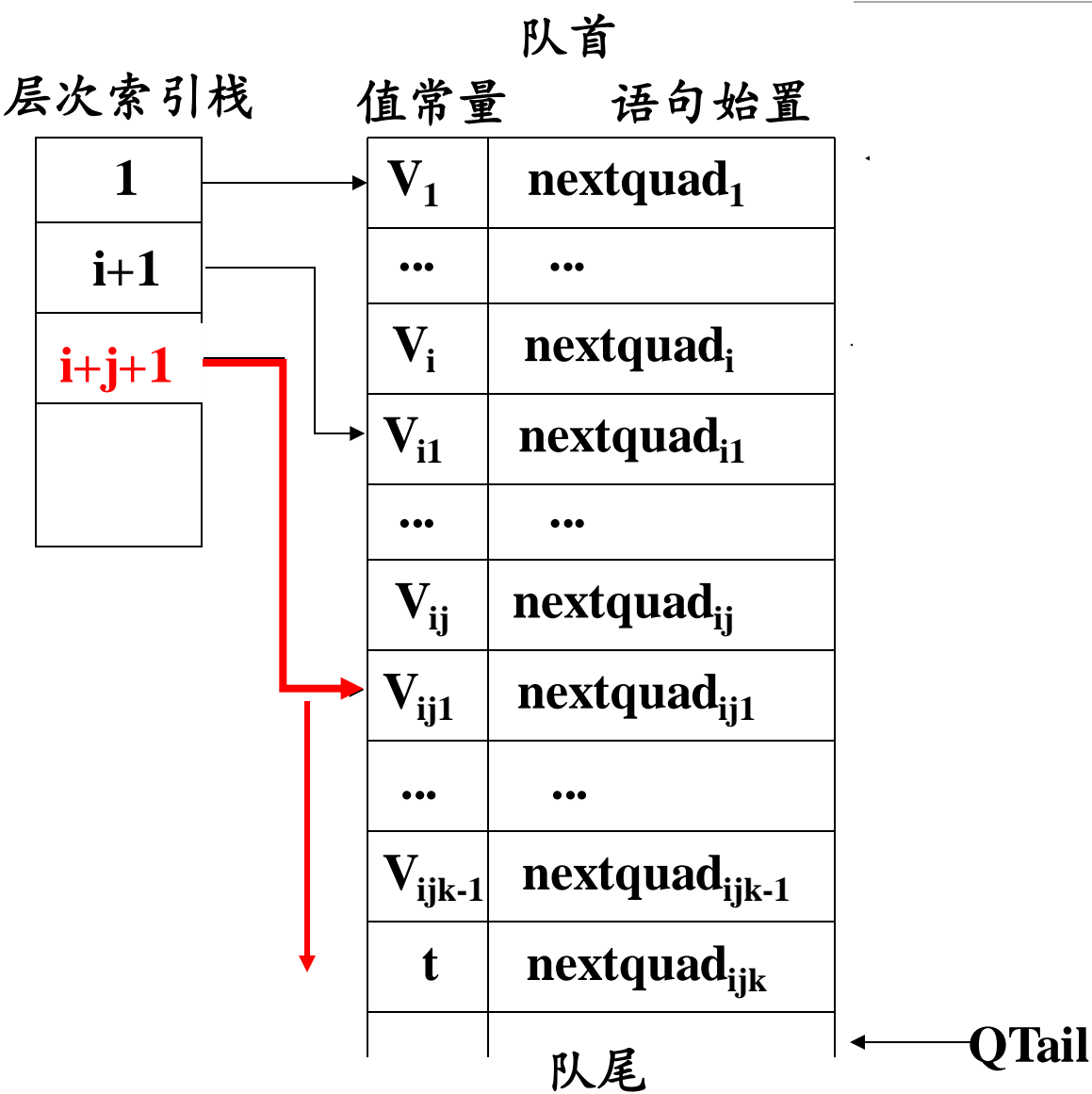


名字	类型	定义标志	地址
test	label	F	-1
next	label	F	-1

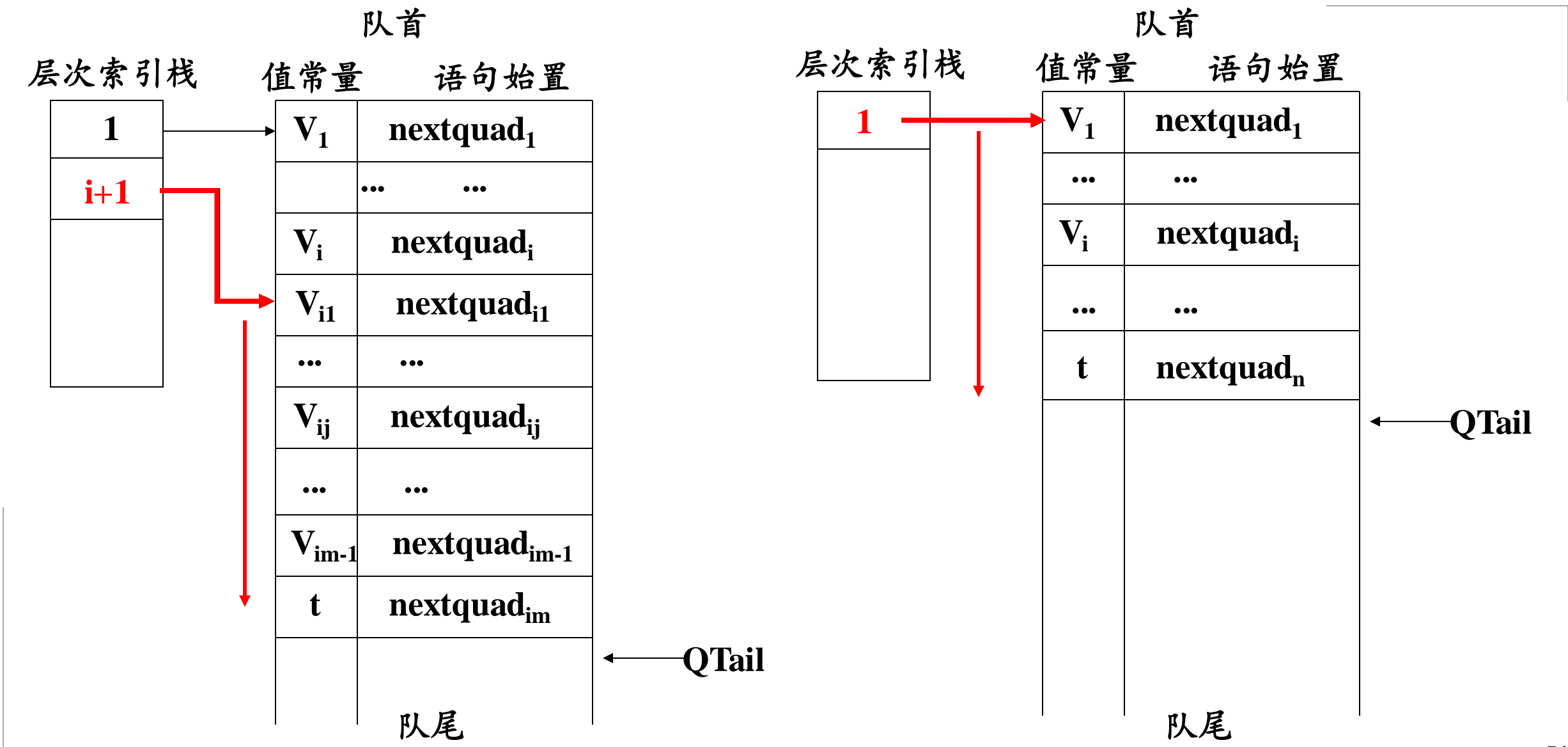
$QHead \rightarrow$	$V_1$	$\text{nextquad}_1$
	$V_2$	$\text{nextquad}_2$
	...	...
	$V_i$	$\text{nextquad}_i$
	...	...
$QTail \rightarrow$	$t$	$\text{nextquad}_n$



```
if t= $V_1$  goto nextquad $_1$ 
if t= $V_2$  goto nextquad $_2$ 
...
if t= $V_{n-1}$  goto nextquad $_{n-1}$ 
goto nextquad $_n$ 
```



# 多层case语句嵌套的情况





# 本章小结

## ■ 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
  - 三地址语句的四元式实现

## ■ 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
- 涉及数组元素的赋值语句
  - 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

# 学习任务

## ■ 作业

- 利用所给翻译方案，将输入的表达式、赋值语句、控制语句、语句序列等翻译为中间代码表示。

## ■ 研究性学习

- 其他控制语句的翻译（如for语句）
- 函数/过程调用语句的翻译。

