

## 第八章 中间代码生成



# 学习内容

---

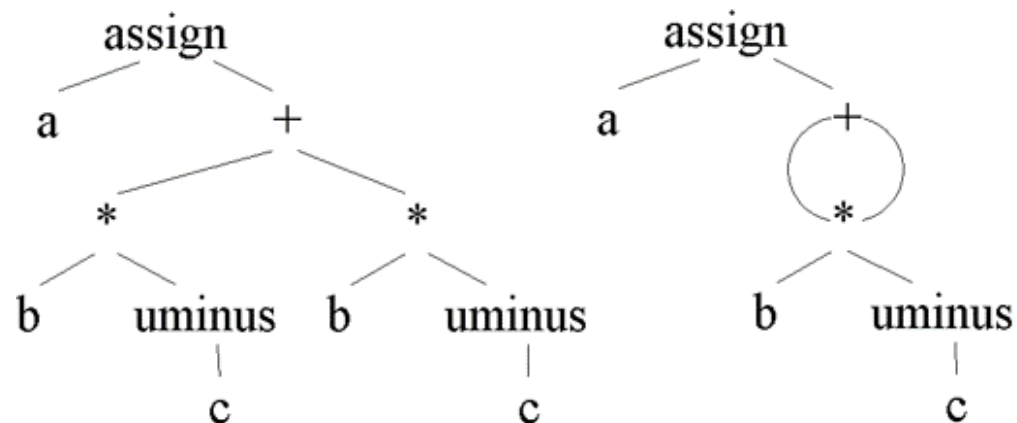
- 三地址码表示方法
- 声明语句的翻译
- 赋值语句的翻译：数组寻址
- 布尔表达式的翻译
- case语句的翻译
- backpatching技术的实现
- 过程调用的翻译

## 8.1 中间语言

### 8.1.1 图表示

$a := b * -c + b * -c$

语法树方式表示



后缀表示——语法树的线性表示方式

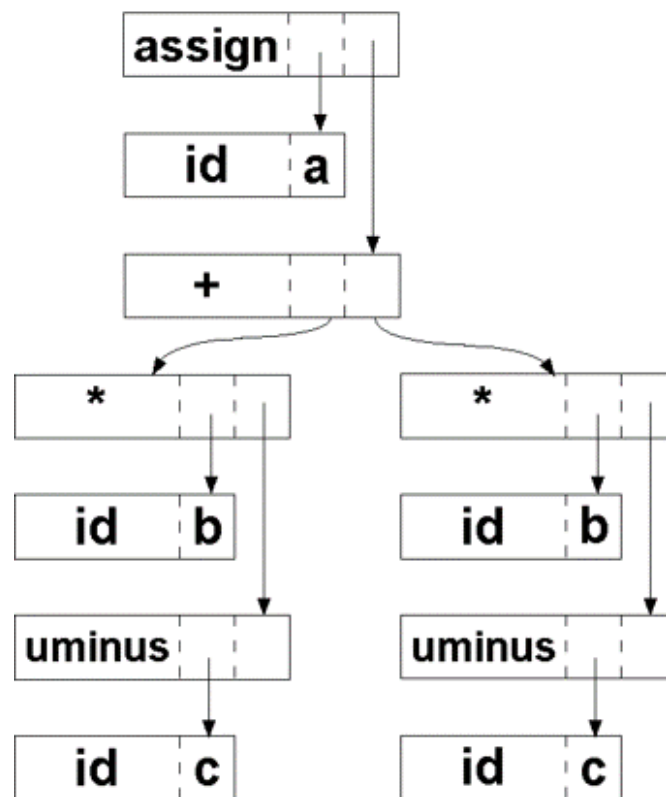
$a \ b \ c \ \text{uminus} \ * \ b \ c \ \text{uminus} \ * \ + \ \text{assign}$



## 语法制导定义构造语法树

<b>PRODUCTION</b>	<b>Semantic Rule</b>
$S \rightarrow \text{id} := E$	$\{ S.nptr = mknode('assign',$ $mkleaf(id, id.entry), E.nptr) \}$
$E \rightarrow E_1 + E_2$	$\{ E.nptr = mknode('+', E_1.nptr, E_2.nptr) \}$
$E \rightarrow E_1 * E_2$	$\{ E.nptr = mknode('*', E_1.nptr, E_2.nptr) \}$
$E \rightarrow - E_1$	$\{ E.nptr = mknode('uminus', E_1.nptr) \}$
$E \rightarrow ( E_1 )$	$\{ E.nptr = E_1.nptr \}$
$E \rightarrow \text{id}$	$\{ E.nptr = mkleaf(id, id.entry) \}$

# 语法树的计算机内部表示



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	...		



## 8.1.2 三地址码

- Three-Address Code

- 一般形式:  $x := y \text{ op } z$

- 语法树、dag的线性化表示

$t_1 := -c$

$t_1 := -c$

$t_2 := b * t_1$

$t_2 := b * t_1$

$t_3 := -c$

$t_5 := t_2 + t_2$

$t_4 := b * t_3$

$a := t_5$

$t_5 := t_2 + t_4$

$a := t_5$



## 8.1.3 三地址码语句类型

1. 二元运算:  $x := y \text{ op } z$
2. 一元运算:  $x := \text{op } y$
3. 复制语句:  $x := y$
4. 无条件转移:  $\text{goto } L$
5. 条件转移:  $\text{if } x \text{ relop } y \text{ goto } L$
6. 函数调用:  
     $\text{param } x_1$   
         $\text{param } x_2$   
         $\dots$   
         $\text{param } x_n$   
     $\text{call } p, n$
7. 数组:  $x := y[i], \quad x[i] := y$
8. 指针:  $x := \&y, \quad x := *y$



## 语法制导翻译生成三地址码

- 赋值语句:  $id := E$
- 利用属性
  - ▣  $E.place$ : 保存E的值的名字
  - ▣  $E.code$ : 计算E的三地址代码
- $newtemp$ : 生成临时变量名
- $gen$ : 输出一条三地址码指令



## 赋值语句的翻译

### PRODUCTION Semantic Rule

$S \rightarrow \text{id} := E$        $\{ S.code = E.code \parallel gen(id.place \text{ ':=' } E.place) \}$

$E \rightarrow E_1 + E_2$        $\{ E.place = newtemp ;$   
                          $E.code = E_1.code \parallel E_2.code \parallel$   
                          $\parallel gen(E.place \text{ ':=' } E_1.place \text{ '+' } E_2.place) \}$

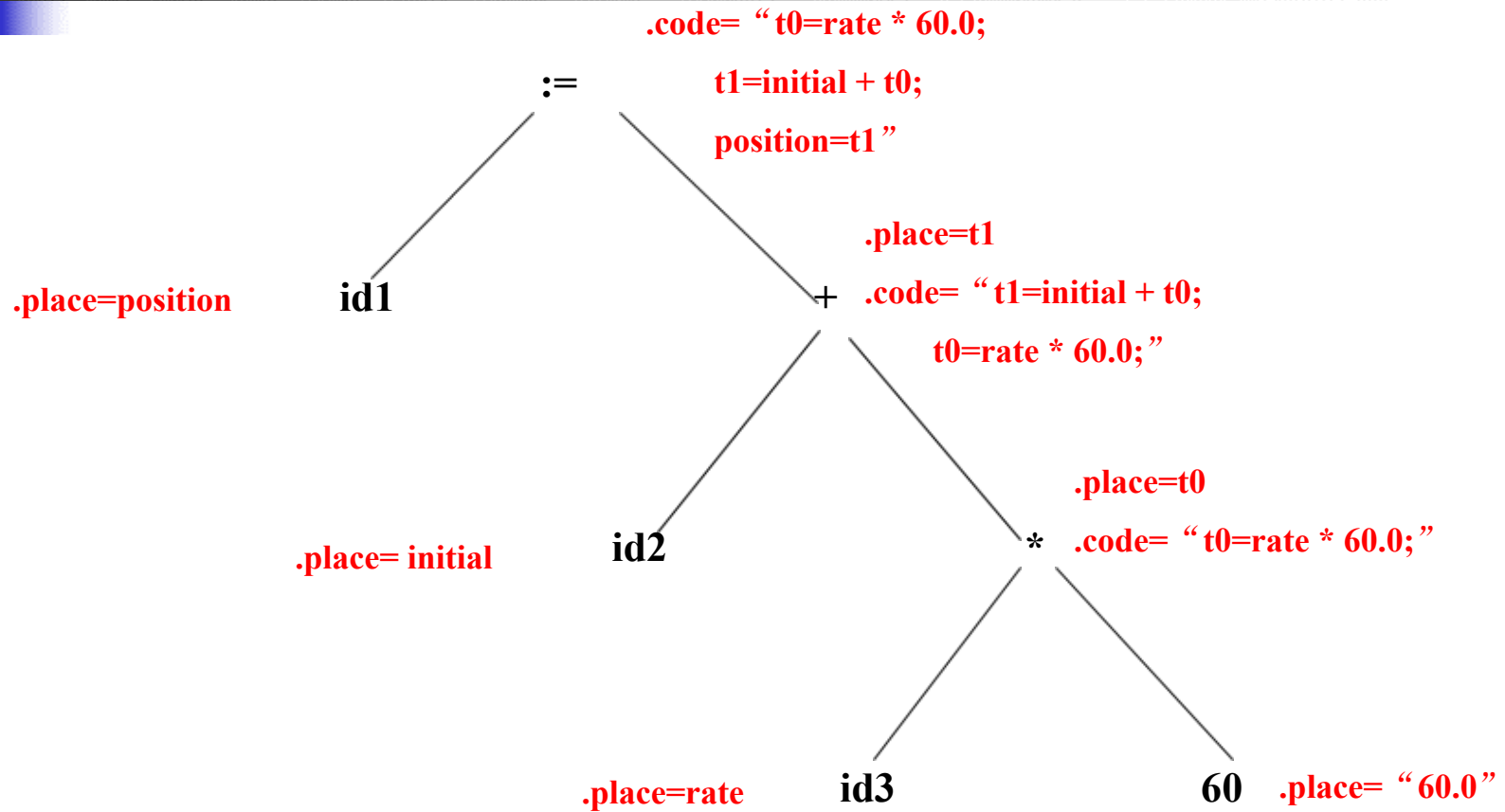
$E \rightarrow E_1 * E_2$        $\{ E.place = newtemp ;$   
                          $E.code = E_1.code \parallel E_2.code \parallel$   
                          $\parallel gen(E.place \text{ ':=' } E_1.place \text{ '*' } E_2.place) \}$

$E \rightarrow - E_1$        $\{ E.place = newtemp ;$   
                          $E.code = E_1.code \parallel$   
                          $\parallel gen(E.place \text{ ':=' 'uminus' } E_1.place) \}$

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

$E \rightarrow \text{id}$        $\{ E.place = id.entry ; E.code = '' \}$

## 中间代码生成示例



## 控制流语句的翻译

○ while语句:  $S \rightarrow \text{while } E \text{ do } S_1$

○ 翻译为

<i>S.begin</i> :	<i>E.code</i>
	if <i>E.place</i> = 0 goto <i>S.after</i>
	<i>S<sub>1</sub>.code</i>
	goto <i>S.begin</i>

*S.after*:

○ 语法制导定义:

*S.begin* = newlabel;

*S.after* = newlabel ;

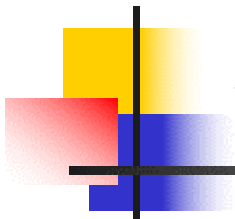
*S.code* = gen(*S.begin* ':') || *E.code* || gen('if' *E.place* '=' '0' 'goto'  
*S.after*) || *S<sub>1</sub>.code* || gen('goto' *S.begin*) || gen(*S.after* ':')

## 8.1.5 三地址码的实现

○ 四元组，三元组

	op	arg1	arg2	result
(0)	uminus	c		t <sub>1</sub>
(1)	*	b	t <sub>1</sub>	t <sub>2</sub>
(2)	uminus	c		t <sub>3</sub>
(3)	*	b	t <sub>3</sub>	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	:=	t <sub>5</sub>		a

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)



## 三元运算的实现——拆分

○  $x[i] := y$

	op	arg1	arg2
(0)	[ ]=	x	i
(1)	assign	(0)	y

○  $x := y[i]$

	op	arg1	arg2
(0)	=[ ]	y	i
(1)	assign	x	(0)



## 间接三元组实现方式

	语句		op	arg1	arg2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	:=	a	(18)

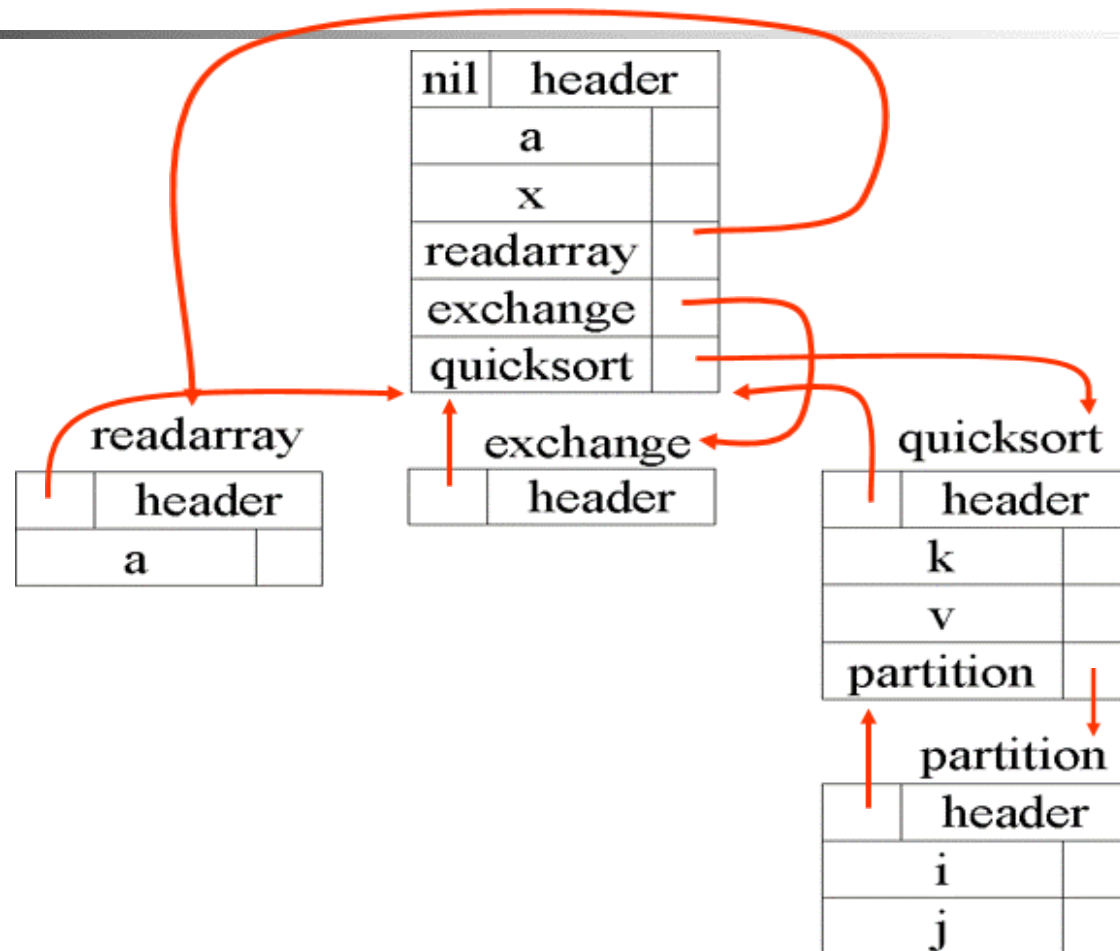
## 8.2 声明语句的翻译

### 8.2.1 过程内部的声明

#### PRODUCTION   Semantic Rule

$P \rightarrow M D$	$\{ \}$
$M \rightarrow \varepsilon$	$\{ offset := 0; \}$
$D \rightarrow id : T$	$\{ addtype(id.entry, T.type, offset);$ $offset := offset + T.width; \}$
$T \rightarrow char$	$\{ T.type = char; T.width = 1; \}$
$T \rightarrow integer$	$\{ T.type = integer; T.width = 4; \}$
$T \rightarrow array [ num ] of T_1$	$\{ T.type = array(num.val, T_1.type);$ $T.width = num.val * T_1.width; \}$
$T \rightarrow ^T_1$	$\{ T.type = pointer(T_1.type);$ $T_1.width = 4; \}$

## 8.2.2 作用域的处理





## 处理作用域的翻译模式

符号表栈

$P \rightarrow M D$       { *addwidth*(*top*(tblptr), *top*(offset));  
                          *pop*(tblptr); *pop*(offset); }

$M \rightarrow \varepsilon$       { *t*:=*mktable*(null); *push*(*t*, tblptr);  
                          *push*(0, offset); }

$D \rightarrow D_1 ; D_2$

$D \rightarrow \text{proc id} ; N D ; S$       { *t*:=*top*(tblptr);  
                          *addwidth*(*t*, *top*(offset));  
                          *pop*(tblptr); *pop*(offset);  
                          *enterproc*(*top*(tblptr), *id.name*, *t*); }

$N \rightarrow \varepsilon$       { *t*:=*mktable*(*top*(tblptr)); *push*(*t*, tblptr);  
                          *push*(0, offset); }

$D \rightarrow \text{id} : T$       { *enter*(*top*(tblptr), *id.name*, *T.type*, *top*(offset));  
                          *top*(offset):=*top*(offset) + *T.width*; }

内存占  
用量栈



## 8.2.3 记录类型的处理

### ○ 创建独立的符号表

$T \rightarrow \text{record } L \text{ D end}$       {  $T.\text{type} = \text{record}(\text{top}(\text{tblptr}))$ ;

$T.\text{width} = \text{top}(\text{offset})$ ;

$\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$

$L \rightarrow \varepsilon$

{  $t = \text{mktable}(\text{null})$ ;

$\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

## 8.3 赋值语句

在符号表中  
查找标识符

### 8.3.1 符号表中的名字

```
S → id := E    { p = lookup(id.name);  
                  if (p != null) emit(p ' := ' E.place);  
                  else error; }  
  
E → E1 + E2 { E.place = newtemp;  
                  emit(E.place ' := ' E1.place ' + ' E2.place; }  
  
E → E1 * E2 { E.place = newtemp;  
                  emit(E.place ' := ' E1.place ' * ' E2.place; }  
  
E → - E1      { E.place = newtemp;  
                  emit(E.place ' := ' 'uminus' E1.place; }  
  
E → ( E1 )    { E.place = E1.place }  
  
E → id         { p = lookup(id.name);  
                  if (p != null) E.place = p;  
                  else error; }
```



## 8.3.2 临时名字的重用

- newtemp每次产生不同名字→临时变量数=表达式中运算符数，空间占用巨大！
- 临时变量保存的中间计算结果，后续不会继续使用，浪费空间！
- 能否重用临时变量保存不同中间结果？←分析变量生存周期：语法制导翻译方法、基于中间代码进行优化
- $E \rightarrow E_1 + E_2 \rightarrow$ 
  - $t_1$ 、 $t_2$ 生存周期结束！——孩子节点、子表达式、运算对象
  - 计算 $E_1 \rightarrow t_1$
  - 计算 $E_2 \rightarrow t_2$
  - $t_3$ 生存周期开始！——父节点、运算结果
  - $t_3 = t_1 + t_2$



## 重用算法

- 观察不同临时名字生存周期期间的关系
- 嵌套、前后，但不会交叉！
- → 栈！
- 产生式翻译：
  - 孩子节点临时变量弹出栈、归还临时名字池
  - 父节点从池中分配临时名字、压栈
- 不显式使用栈——计数器c，初始为0
  - 临时名字作为运算对象使用：c--
  - 生成新的临时名字：\$c，c++



## 例8.1

○  $x = a * b + c * d - e * f$

\$0, \$1为运算对象, c减2,  
变为0→结果又保存在\$0



语句	c值
	0
<b><math>\\$0 = a * b</math></b>	1
<b><math>\\$1 = c * d</math></b>	2
<b><math>\\$0 = \\$0 + \\$1</math></b>	1
<b><math>\\$1 = e * f</math></b>	2
<b><math>\\$0 = \\$0 - \\$1</math></b>	1
<b><math>x = \\$0</math></b>	0

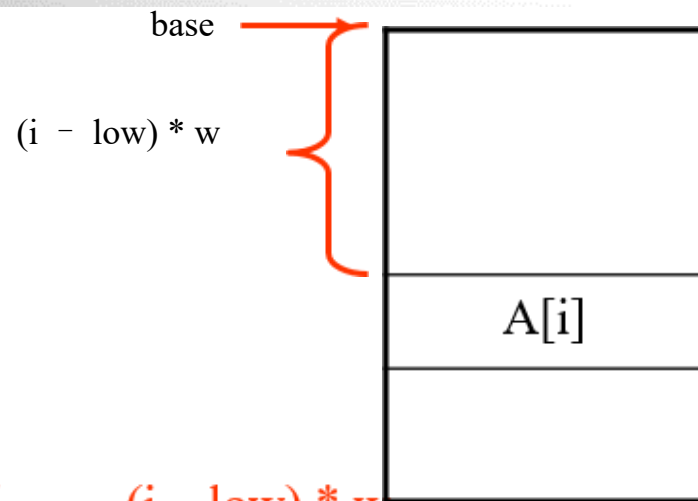
### 8.3.3 数组元素寻址

#### ○ 一维数组

□ `type A[low..high];`

□ 计算 $A[i]$ 的地址

- A的起始地址—— $\text{base}$
- 数组元素大小—— $w$
- $A[i]$ 与A起始位置的“距离”—— $(i - \text{low}) * w$
- 最终结果:  $\text{base} + (i - \text{low}) * w$   
    ➔  $i * w + (\text{base} - \text{low} * w)$
- $(\text{base} - \text{low} * w)$ 为常量, 可在编译时计算



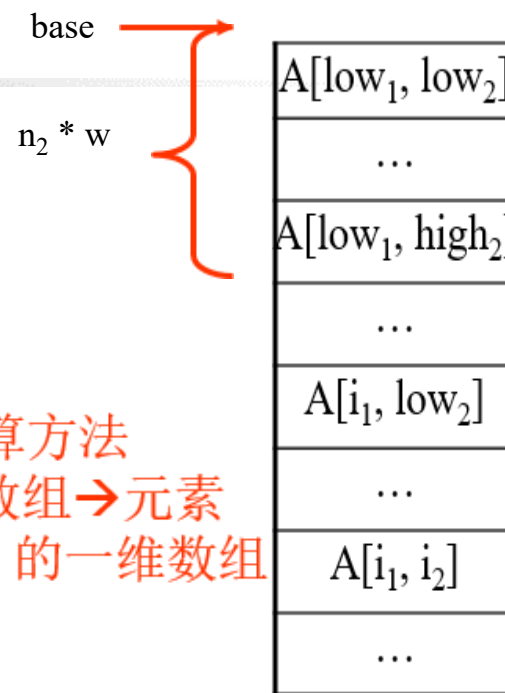
## 数组元素寻址（续）

### ○ 二维数组：

□ type  $A[\text{low}_1..\text{high}_1, \text{low}_2..\text{high}_2]$

□ 计算  $A[i_1, i_2]$  地址

- 数组的数组，两次利用一维数组计算方法
- 行：  $n_2 = \text{high}_2 - \text{low}_2 + 1$  个元素的一维数组 → 元素  
二维数组：  $n_1 = \text{high}_1 - \text{low}_1 + 1$  个“行”的一维数组
- $i_1$  行的位置  $(i_1 - \text{low}_1) * n_2$
- $A[i_1, i_2]$  距行开始位置的距离：  $i_2 - \text{low}_2$
- 最终结果  $\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$   
→  $((i_1 * n_2) + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$
- $(\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$  为常量







## 数组元素寻址（续）

- 扩展到多维情况

- type A[low<sub>1</sub>..high<sub>1</sub>, ..., low<sub>k</sub>..high<sub>k</sub>]

- 计算A[i<sub>1</sub>, i<sub>2</sub>, ..., i<sub>k</sub>]的地址

- 最终结果

- $((\cdots((i_1 n_2 + i_2) n_3 + i_3) \cdots) n_k + i_k) * w +$   
 $\text{base} - ((\cdots((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \cdots) n_k + \text{low}_k) * w$



## 实现方法

$L \rightarrow \mathbf{id} [ \text{Elist} ] \mid \mathbf{id} \rightarrow L \rightarrow \text{Elist} ] \mid \mathbf{id}$

$\text{Elist} \rightarrow \text{Elist} , E \mid E \qquad \text{Elist} \rightarrow \text{Elist} , E \mid \mathbf{id} [ E$

○  $(\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k$  的计算

$$e_1 = i_1, \quad e_m = e_{m-1} * n_m + i_m$$

○  $\text{Elist.ndim}$ : 数组维数

○  $\text{Elist.place}$ : 下标表达式计算结果,  $e_m$

○  $\text{limit}(\text{array}, m)$ : 第  $m$  维的大小,  $n_m$

○  $L.place$ : 基地址 (地址计算常量部分)

○  $L.offset$ : 偏移, 普通变量为 `null`



## 8.3.4 数组元素寻址的翻译模式

○ 文法

1.  $S \rightarrow L := E$
2.  $E \rightarrow E + E$
3.  $E \rightarrow ( E )$
4.  $E \rightarrow L$
5.  $L \rightarrow \text{Elist } ]$
6.  $L \rightarrow \mathbf{id}$
7.  $\text{Elist} \rightarrow \text{Elist } , E$
8.  $\text{Elist} \rightarrow \mathbf{id} [ E$

## 数组元素寻址的翻译模式(续)

1.  $S \rightarrow L := E$  { if (L.offset == null)

emit(L.place ‘:=’ E.place);

else

emit(L.place ‘[’ L.offset ‘]’ ‘:=’ E.place); }

2.  $E \rightarrow E_1 + E_2$  { E.place = newtemp;

emit(E.place ‘:=’ E<sub>1</sub>.place ‘+’ E<sub>2</sub>.place); }

3.  $E \rightarrow ( E_1 )$  { E.place = E<sub>1</sub>.place; }

普通变量?

数组?

## 数组元素寻址的翻译模式(续)

5.  $L \rightarrow \mathbf{id} \quad \{ L.place = \mathbf{id}.place; L.offset = \mathbf{null}; \}$

6.  $Elist \rightarrow Elist_1, E \quad \{$

$t = \text{newtemp}; m = Elist_1.ndim + 1;$

$$e_m = e_{m-1} * n_m + i_m$$

$\text{emit}(t \text{ ':=' } Elist_1.place \text{ '*' } \text{limit}(Elist_1.array, m));$

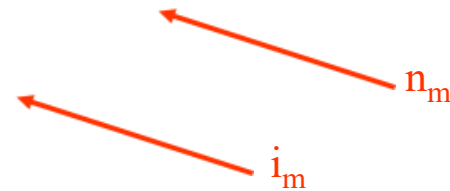
$e_{m-1} \quad \text{emit}(t \text{ ':=' } t \text{ '+' } E.place);$

$e_m \quad Elist.array = Elist_1.array;$

$Elist.place = t; Elist.ndim = m; \}$

7.  $Elist \rightarrow \mathbf{id} [ E \quad \{ Elist.array = \mathbf{id}.place;$

$Elist.place = E.place; Elist.ndim = 1; \}$



## 数组元素寻址的翻译模式(续)

```
4. E → L    { if (L.offset == null)
               E.place = L.place;
               else {
               E.place = newtemp;
               emit(E.place ‘:=’ L.place ‘[’ L.offset ‘]’ ); }}
```

```
5. L → Elist ] { L.place = newtemp; L.offset = newtemp;
                  emit(L.place ‘:=’ c(Elist.array));
                  emit(L.offset ‘:=’ Elist.place ‘*’ width(Elist.array)); }
```

地址计算常量部分

## 例8.2

10×20的数组， $low_1=low_2=1$ ,  $n_1=10$ ,  $n_2=20$ ,  $w=4$

翻译  $x := A[y, z]$

生成的三地址码：

$t_1 := y * 20$

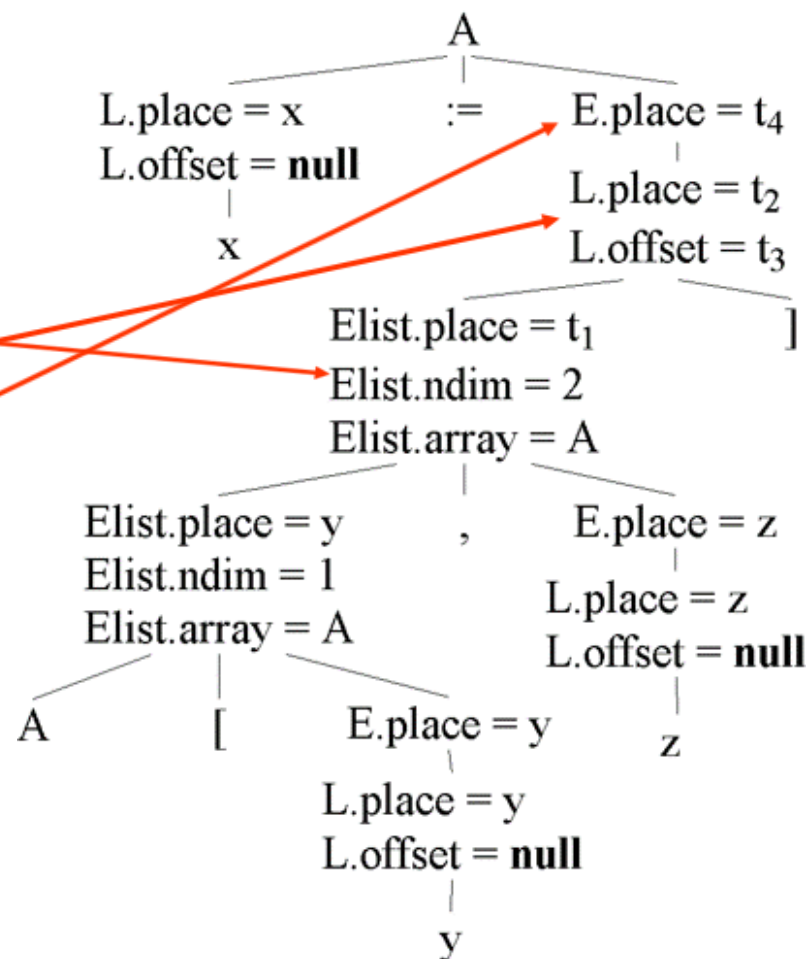
$t_1 := t_1 + z$

$t_2 := c$

$t_3 := 4 * t_1$

$t_4 := t_2[t_3]$

$x := t_4$





## 8.3.5 类型转换

○  $E \rightarrow E_1 + E_2$  的语义动作

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





## 类型转换例子

○  $x := y + i * j$

$x$ 、 $y$ ——实数

$i$ 、 $j$ ——整数

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

$t_3 := \text{intto real } t_1$

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

$x := t_2$



## 8.3.6 记录（结构）域的访问

- 保存每个域的类型和相对地址→符号表
- lookup可用于域名字
- 独立符号表
- t: 符号表指针,  $\text{record}(t) \rightarrow T.\text{type}$
- 翻译 $p^{\wedge}.\text{info} + 1$ 
  - p的类型 $\text{pointer}(\text{record}(t))$
  - $p^{\wedge}$ 的类型 $\text{record}(t)$
  - 得到t→查找info域



## 8.4 布尔表达式的翻译

- $E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id rel op id} \mid \text{true} \mid \text{false}$

- 8.4.1 两种翻译方式

- 数值编码：0—false，非0—ture

- 控制流语句

- 属性true——布尔表达式为真时跳转到的程序位置

- 属性false——布尔表达式为假时跳转到的程序位置

- 短路求值

- $E_1 \text{ or } E_2$ ,  $E_1 = \text{false}$ 才对 $E_2$ 求值

- 注意副作用问题：函数调用



## 8.4.2 用数值表示布尔值

○  $a \text{ or } b \text{ and not } c \rightarrow$

$t_1 := \text{not } c$

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

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

○  $a < b \rightarrow \text{if } a < b \text{ then } 1 \text{ else } 0$

100:       $\text{if } a < b \text{ goto } 103$

101:       $t := 0$

102:       $\text{goto } 104$

103:       $t := 1$

104:



# 翻译模式

$E \rightarrow E_1 \text{ or } E_2$  { E.place = newtemp;  
emit(E.place ':= ' E<sub>1</sub>.place 'or' E<sub>2</sub>.place); }

$E \rightarrow E_1 \text{ and } E_2$  { E.place = newtemp;  
emit(E.place ':= ' E<sub>1</sub>.place 'and' E<sub>2</sub>.place); }

$E \rightarrow \text{not } E_1$  { E.place = newtemp;  
emit(E.place ':= ' 'not' E<sub>1</sub>.place); }

$E \rightarrow ( E_1 )$  { E.place = E<sub>1</sub>.place; }

$E \rightarrow \text{id}_1 \text{ relop id}_2$  { E.place = newtemp;  
emit( 'if' id<sub>1</sub>.place relop.op id<sub>2</sub>.place 'goto' nextstat + 3);  
emit(E.place ':= ' '0' );  
emit( 'goto' nextstat + 2);  
emit(E.place ':= ' '1' ); }

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

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



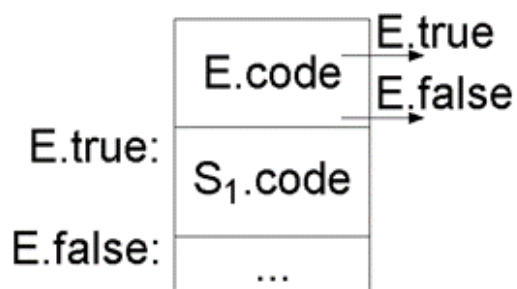
## 例8.3

○ 翻译  $a < b$  or  $c < d$  and  $e < f$

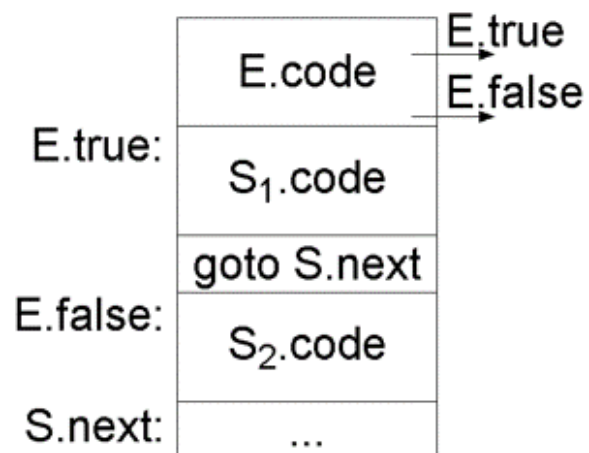
100: if $a < b$ goto 103	107: $t_2 := 1$
101: $t_1 := 0$	108: if $e < f$ goto 111
102: goto 104	109: $t_3 := 0$
103: $t_1 := 1$	110: goto 112
104: if $c < d$ goto 107	111: $t_3 := 1$
105: $t_2 := 0$	112: $t_4 := t_2$ and $t_3$
106: goto 108	113: $t_5 := t_1$ or $t_4$

## 8.4.3 控制流语句的翻译

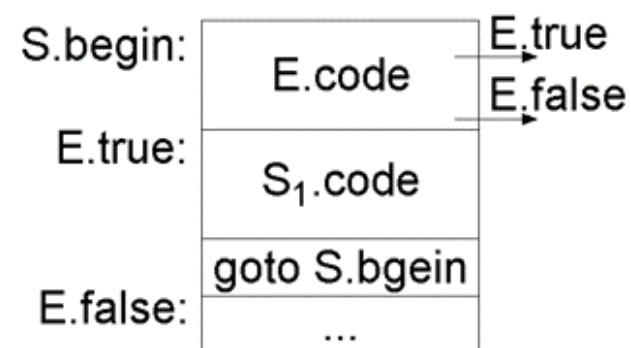
$S \rightarrow \text{if } E \text{ then } S_1 \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S_1$



(a) if-then

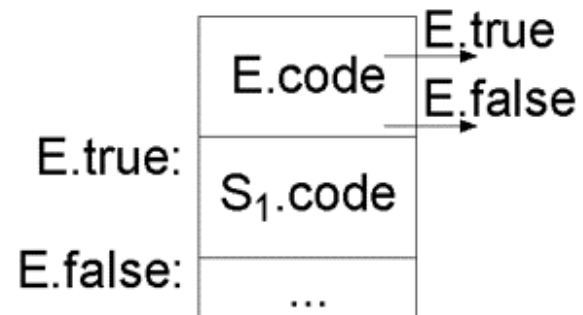


(b) if-then-else



(a) while-do

## 语法制导定义—if



(a) if-then

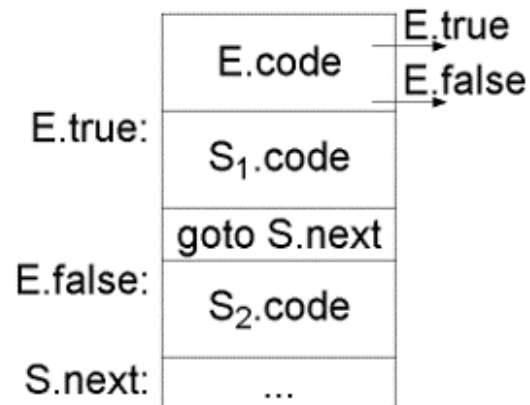
$S \rightarrow \text{if } E \text{ then } S_1$      $E.\text{true} = \text{newlabel}; E.\text{false} = S.\text{next};$

$S_1.\text{next} = S.\text{next};$

$S.\text{code} = E.\text{code} \parallel \text{gen}(E.\text{true} \text{ ':' } ) \parallel S_1.\text{code}$



## 语法制导定义—if-else



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

(b) if-then-else

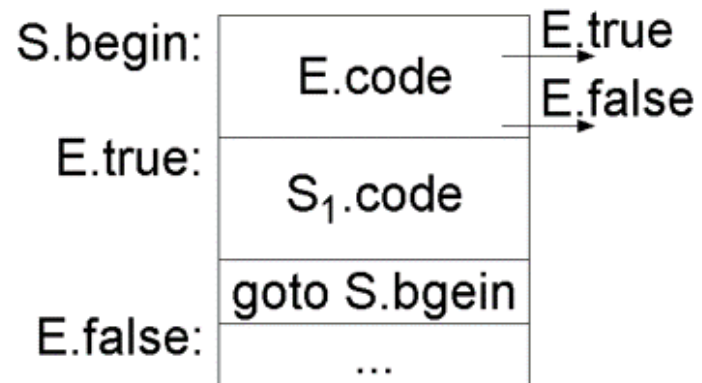
$E.\text{true} = \text{newlabel}; E.\text{false} = \text{newlabel};$

$S_1.\text{next} = S.\text{next}; S_2.\text{next} = S.\text{next};$

$S.\text{code} = E.\text{code} \parallel \text{gen}(E.\text{true} \text{ ':' }) \parallel S_1.\text{code}$

$\parallel \text{gen}(\text{'goto' } S.\text{next}) \parallel \text{gen}(E.\text{false} \text{ ':' }) \parallel S_2.\text{code}$

# 语法制导定义



(a) while-do

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

S.begin = newlabel; E.true = newlabel;

E.false = S.next; S<sub>1</sub>.next = S.begin;

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

S<sub>1</sub>.code || gen( 'goto' S.begin)



## 8.4.5 布尔表达式翻译为控制流语句

$E \rightarrow E_1 \text{ or } E_2$      $E_1.\text{true} = E.\text{true}; E_1.\text{false} = \text{newlabel};$   
                               $E_2.\text{true} = E.\text{true}; E_2.\text{false} = E.\text{false};$   
                               $E.\text{code} = E_1.\text{code} \parallel \text{gen}(E_1.\text{false} \text{ ‘:’ }) \parallel E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$      $E_1.\text{true} = \text{newlabel}; E_1.\text{false} = E.\text{false};$   
                               $E_2.\text{true} = E.\text{true}; E_2.\text{false} = E.\text{false};$   
                               $E.\text{code} = E_1.\text{code} \parallel \text{gen}(E_1.\text{true} \text{ ‘:’ }) \parallel E_2.\text{code}$

$E \rightarrow \text{not } E_1$      $E_1.\text{true} = E.\text{false}; E_1.\text{false} = E.\text{true}; E.\text{code} = E_1.\text{code};$

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

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

$E \rightarrow \text{true}$      $E.\text{code} = \text{gen}( \text{‘goto’ } E.\text{true});$

$E \rightarrow \text{false}$      $E.\text{code} = \text{gen}( \text{‘goto’ } E.\text{false});$

## 例8.4

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

生成代码:

if  $a < b$  goto Ltrue

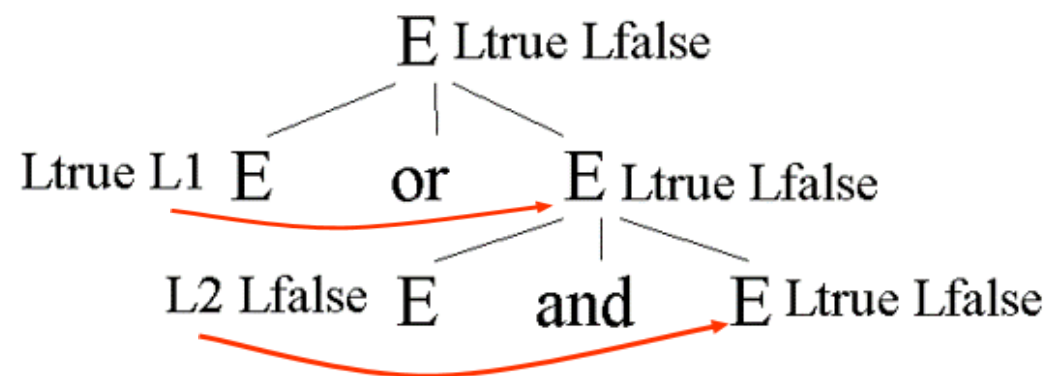
goto L1

L1: if  $c < d$  goto L2

goto Lfalse

L2: if  $e < f$  goto Ltrue

goto Lfalse



## 例8.5

```

while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
  
```

生成代码为:

L1: if a < b goto L2

goto Lnext

L2: if c < d goto L3

goto L4

L3:  $t_1 := y + z$

$x := t_1$

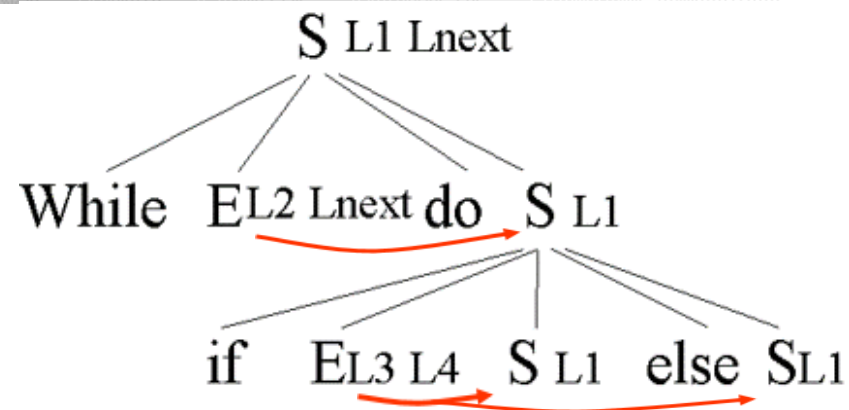
goto L1

L4:  $t_2 := y - z$

$x := t_2$

goto L1

Lnext:





## 8.4.6 混合翻译方式

$E \rightarrow E + E \mid E \text{ and } E \mid E \text{ relop } E \mid \text{id}$

$E \rightarrow E_1 + E_2$  对应的语义规则为:

$E.\text{type} = \text{arith};$

**if** ( $E_1.\text{type} = \text{arith}$  **and**  $E_2.\text{type} = \text{arith}$ ) {

$E.\text{place} = \text{newtemp};$

$E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel \text{gen}(E.\text{place} \text{ ‘:=’ } E_1.\text{place} \text{ ‘+’ } E_2.\text{place});$

**} else if** ( $E_1.\text{type} = \text{arith}$  **and**  $E_2.\text{type} = \text{bool}$ ) {

$E.\text{place} = \text{newtemp}; E_2.\text{true} = \text{newlabel}; E_2.\text{false} = \text{newlabel};$

$E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel$

$\text{gen}(E_2.\text{true} \text{ ‘:’ } E.\text{place} \text{ ‘:=’ } E_1.\text{place} + 1) \parallel$

$\text{gen}(\text{ ‘goto’ } \text{nextstat} + 1) \parallel$

$\text{gen}(E_2.\text{false} \text{ ‘:’ } E.\text{place} \text{ ‘:=’ } E_1.\text{place})$

**} else if ...**



## 8.5 case语句的翻译

switch E

begin

case  $V_1 : S_1$

case  $V_2 : S_2$

...

case  $V_{n-1} : S_{n-1}$

default :  $S_n$

end

1. 计算表达式E
2. 寻找哪个V与E的值相等，都不等，则选择default
3. 运行对应的语句



# 翻译方法

- 第(2)步是一个n-路分支
  - 翻译为简单的条件分支语句
  - 表驱动
    - 表项——<值, 标号>
    - 利用一循环进行检测
  - hash表
  - 直接定位
    - 取值范围小,  $i_{\min} \sim i_{\max}$
    - 标号数组  $\text{label}[i_{\max} - i_{\min} + 1]$
    - 值  $j \rightarrow$  对应语句标号  $\text{label}[j - i_{\min}]$
    - 表达式求值  $\rightarrow$  直接定位标号





# 翻译方法1

```
      计算 $E \rightarrow t$   
      goto test  
L1: S1代码  
      goto next  
L2: S2代码  
      goto next  
      ...  
Ln-1: Sn-1代码  
      goto next  
Ln: Sn代码  
      goto next  
test:  if ( $t == V_1$ ) goto L1  
      if ( $t == V_2$ ) goto L2  
      ...  
      if ( $t == V_{n-1}$ ) goto Ln-1  
      goto Ln  
next:
```



## 翻译方法2

```
      计算  $E \rightarrow t$   
      if  $t \diamond V_1$  goto  $L_1$   
       $S_1$  代码  
      goto test  
 $L_1$ :  if  $t \diamond V_2$  goto  $L_2$   
       $S_2$  代码  
      goto next  
 $L_2$ :  
      ...  
 $L_{n-2}$ : if  $t \diamond V_{n-1}$  goto  $L_{n-1}$   
       $S_{n-1}$  代码  
      goto next  
 $L_n$ :   $S_n$  代码  
next:
```



## 8.6 BackPatching技术

- 第4节语法制导定义实现方法

- 两遍扫描：创建语法树，然后计算属性值
- 若单遍扫描：生成转移语句时目的标号未知

- backpatching

- 语句→四元式数组，标号——数组索引
- makelist(i): 创建列表，仅包含i——指向四元式数组的索引
- merge( $p_1, p_2$ ): 合并列表 $p_1$ 、 $p_2$
- backpatch( $p, i$ ): 将列表 $p$ 指向的所有语句中的空白地址用 $i$ 填入

## 8.6.1 语法制导定义

下一语句编号

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

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

## 语法制导定义（续）

下一语句编号

```
E → id1 relop id2 { E.truelist = makelist(nextquad);  
                        E.falselist = makelist(nextquad + 1);  
                        emit( 'if' id1.place relop.op id2.place 'goto _' );  
                        emit( 'goto _' ); }
```

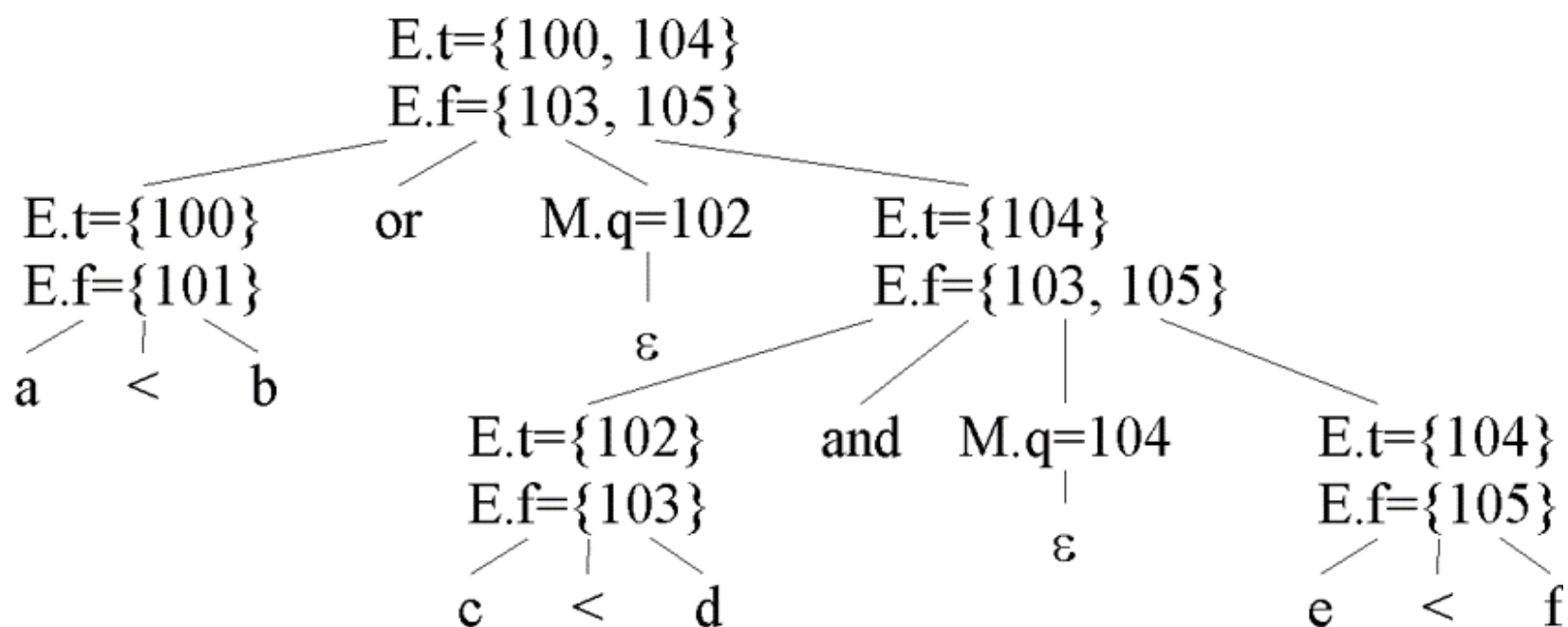
```
E → true { E.truelist = makelist(nextquad);  
          emit( 'goto _' ); }
```

```
E → false { E.falselist = makelist(nextquad);  
           emit( 'goto _' ); }
```

```
M → ε { M.quad = nextquad; }
```

## 例8.6

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





## 例8.6（续）

- $a < b$  归约为E，产生两个四元式

100: if  $a < b$  goto \_

101: goto \_

- $E \rightarrow E_1 \text{ or } M E_2$  中的M，属性值为102

- $c < d$  归约为E，产生两个四元式

102: if  $c < d$  goto \_

103: goto \_

- $E \rightarrow E_1 \text{ and } M E_2$  中的M，属性值为104

- $e < f$  归约为E，产生两个四元式

104: if  $e < f$  goto \_

105: goto \_

## 例8.6（续）

- 利用 $E \rightarrow E_1 \text{ and } M E_2$  归约,  $\text{backpatch}(\{102\}, 104)$

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

- 规约 $E \rightarrow E_1 \text{ or } M E_2$ ,  $\text{backpatch}(\{101\}, 102)$

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

**E.truelist**

**E.falselist**





## 8.6.2 控制流语句的翻译

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

{ backpatch(E.truelist,  $M_1$ .quad);

backpatch(E.falselist,  $M_2$ .quad);

$S$ .nextlist = merge( $S_1$ .nextlist, merge( $N$ .nextlist,  $S_2$ .nextlist)); }

$N \rightarrow \epsilon$  {  $N$ .nextlist = makelist(nextquad); emit( 'goto \_' ); }

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

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

{ backpatch( $S_1$ .nextlist,  $M_1$ .quad);

backpatch(E.truelist,  $M_2$ .quad);

$S$ .nextlist = E.falselist;

emit( 'goto'  $M_1$ .quad); }

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



## 控制流语句的翻译（续）

---

$S \rightarrow A \quad \{ S.nextlist = \text{null}; \}$

$L \rightarrow L_1 ; M S \quad \{ \text{backpatch}(L_1.nextlist, M.quad);$

$L.nextlist = S.nextlist; \}$

$L \rightarrow S \quad \{ L.nextlist = S.nextlist; \}$



## 8.7 过程调用的翻译

$S \rightarrow \text{call id} ( \text{Elist} )$

$\text{Elist} \rightarrow \text{Elist}, \text{E}$

$\text{Elist} \rightarrow \text{E}$

### ○ 调用序列

- 为被调用过程活动记录分配内存
- 计算参数值，传递给被调用过程
- 保存调用过程状态，保存返回地址
- 生成转移语句  $\rightarrow$  被调用过程

### ○ 返回序列

- 返回值传递给调用过程
- 恢复调用过程的活动记录
- 跳转回返回地址



## 简单例子

- 参数传地址方式，内存静态分配
  - param语句——参数的占位符
  - 第一个param语句指针传递给被调用过程
  - 调用序列
    - 计算参数
    - param语句列表——罗列参数
    - 所有参数表达式place属性保存在队列queue中



## 简单例子（续）

### ○ 翻译模式

$S \rightarrow \text{call id ( Elist )}$

{ for queue中每个项p do

emit( 'param' p);

emit( 'call' id.place); }

$\text{Elist} \rightarrow \text{Elist , E}$  { 将E.place添加到queue尾部; }

$\text{Elist} \rightarrow \text{E}$  { 初始化queue, 仅包含E.place; }



# 递归遍历语法树生成代码

```
void tree::gen_code(ostream &out)
{
    gen_header(out);           // 生成汇编程序头部
    Node *p = root->children[0];
    if (p->kind == DECL_NODE)
        gen_decl(out, p);      // 生成变量声明
    out << endl << endl << "\t.code" << endl;
    recursive_gen_code(out, root); // 汇编程序生成递归函数
    if (root->label.next_label != "")
        out << root->label.next_label << ":" << endl;
    out << "\tinvoke ExitProcess, 0" << endl;
    out << "end " << root->label.begin_label << endl;
}
```



# 打印程序头部

```
void tree::gen_header(ostream &out)
{
    out << "\t.586" << endl;
    out << "\t.model flat, stdcall" << endl;
    out << "\toption casemap :none" << endl;
    out << endl;
    out << "\tinclude \\masm32\\include\\windows.inc" << endl;
    out << "\tinclude \\masm32\\include\\user32.inc" << endl;
    out << "\tinclude \\masm32\\include\\kernel32.inc" << endl;
    out << "\tinclude \\masm32\\include\\masm32.inc" << endl;
    out << endl;
    out << "\tincludelib \\masm32\\lib\\user32.lib" << endl;
    out << "\tincludelib \\masm32\\lib\\kernel32.lib" << endl;
    out << "\tincludelib \\masm32\\lib\\masm32.lib" << endl;
}
```



## 变量声明

```
void tree::gen_decl(ostream &out, Node *t)
{
    out << endl << endl << "\t.data" << endl;

    for (; t->kind == DECL_NODE; t = t->sibling)
    {
        for (Node *p = t->children[1]; p; p = p->sibling)
            if (p->type == Integer)
                out << "\t\t_" << symtbl.getname(p->attr.symtbl_seq) << " DWORD 0"
<< endl;                                // “_”+C源程序变量名→汇编程序变量名
            else if (p->type == Char)
                out << "\t\t_" << symtbl.getname(p->attr.symtbl_seq) << " BYTE 0" <<
endl;
    }
}
```





## 变量声明（续）

```
for (int i = 0; i < temp_var_seq; i++) // 声明临时变量 ( "t" + 编号)
{
    out << "\t\t" << i << " DWORD 0" << endl;
}

out << "\t\tbuffer BYTE 128 dup(0)" << endl;
out << "\t\tLF BYTE 13, 10, 0" << endl;
}
```



## 递归代码生成函数入口

```
void tree::recursive_gen_code(ostream &out, Node *t)
{
    if (t->kind == STMT_NODE)
    {
        stmt_gen_code(out, t);          // 生成语句的汇编码
    }
    else if (t->kind == EXPR_NODE && (t->kind_kind == OP_EXPR || t-
>kind_kind == NOT_EXPR))
    {
        expr_gen_code(out, t);          // 生成表达式的汇编码
    }
}
```



# 表达式代码生成

```
void tree::expr_gen_code(ostream &out, Node *t)
{
    Node *e1 = t->children[0];    // 第一个运算对象
    Node *e2 = t->children[1];    // 第二个运算对象
    switch (t->attr.op)
    {
    case PLUS:                    // 加法表达式
        out << "\tMOV eax, " ;    // 取出第一个运算对象到eax
        if (e1->kind_kind == ID_EXPR) // 变量打印名字
            out << "_" << symtbl.getname(e1->attr.symtbl_seq);
        else if (e1->kind_kind == CONST_EXPR) // 常量打印数值
            out << e1->attr.vali;
        else out << "t" << e1->temp_var; // 运算表达式打印中间结果临时变量
        out << endl;
        out << "\tADD eax, " ;    // eax与第二个运算对象进行加法
```



## 表达式代码生成（续）

```
if (e2->kind_kind == ID_EXPR)
    out << "_" << symtbl.getname(e2->attr.symtbl_seq);
else if (e2->kind_kind == CONST_EXPR)
    out << e2->attr.vali;
else out << "t" << e2->temp_var;
out << endl;
out << "\tMOV t" << t->temp_var << ", eax" << endl;    // 保存结果
break;
```



## 表达式代码生成（续）

```
case LT:
    out << "\tMOV eax, ";
    if (e1->kind_kind == ID_EXPR)
        out << "_" << symtbl.getname(e1->attr.symtbl_seq);
    else if (e1->kind_kind == CONST_EXPR)
        out << e1->attr.vali;
    else out << "t" << e1->temp_var;
    out << endl;
    out << "\tCMP eax, ";
    if (e2->kind_kind == ID_EXPR)
        out << "_" << symtbl.getname(e2->attr.symtbl_seq);
    else if (e2->kind_kind == CONST_EXPR)
        out << e2->attr.vali;
    else out << "t" << e2->temp_var;
    out << endl;
    out << "\tjl " << t->label.true_label << endl;    // 跳转到真值出口
    out << "\tjmp " << t->label.false_label << endl; // 跳转到假值出口
    break;
```



# 语句代码生成

```
void tree::stmt_gen_code(ostream &out, Node *t)
{
    if (t->kind_kind == COMP_STMT)
    {
        for (int i = 0; t->children[i]; i++)
        {
            recursive_gen_code(out, t->children[i]);
            for (Node *p = t->children[i]->sibling; p; p = p->sibling)
                recursive_gen_code(out, p);
        }
    }
}
```



## 语句代码生成（续）

```
else if (t->kind_kind == WHILE_STMT)
{
    if (t->label.begin_label != "" ) // while循环开始标号
        out << t->label.begin_label << ":" << endl;
    recursive_gen_code(out, t->children[0]); // 生成布尔表达式代码
    recursive_gen_code(out, t->children[1]); // 生成循环体代码
    out << "\tjmp " << t->label.begin_label << endl; // 跳转到循环开始
}
```



## 语句代码生成（续）

```
void tree::get_temp_var(Node *t)
{
    if (t->kind != EXPR_NODE)
        return;
    if (t->attr.op < PLUS || t->attr.op > OVER)
        return;

    Node *arg1 = t->children[0];
    Node *arg2 = t->children[1];
    // 临时变量重用（收回不用的临时变量）
    if (arg1->kind_kind == OP_EXPR)
        temp_var_seq--;
    if (arg2 && arg2->kind_kind == OP_EXPR)
        tree::temp_var_seq--;
    t->temp_var = tree::temp_var_seq; // 分配临时变量
    tree::temp_var_seq++;
}
```



## 语句代码生成（续）

```
void tree::stmt_get_label(Node *t) // 为语句生成标号
{
    switch (t->kind_kind)
    { ...
    case WHILE_STMT:
    {
        Node *e = t->children[0]; // 循环判定条件——布尔表达式
        Node *s = t->children[1]; // 循环体
        if (t->label.begin_label == "")
            t->label.begin_label = new_label(); // 生成循环开始标号
        s->label.next_label = t->label.begin_label; // 循环体的“下一条语句”是循环开始标号——继续循环
        s->label.begin_label = e->label.true_label = new_label(); // 生成循环体开始标号——也是循环条件真值出口

        if (t->label.next_label == "")
            t->label.next_label = new_label(); // 生成整个循环的下一条语句标号（循环结束）
        e->label.false_label = t->label.next_label; // 循环条件假值出口即循环结束
        if (t->sibling)
            t->sibling->label.begin_label = t->label.next_label; // 兄弟结点的开始标号即这条语句的下一个标号
        recursive_get_label(e); // 递归地生成布尔表达式内标号
        recursive_get_label(s); // 递归地生成循环体内标号
    }
    }
}
```

显然，标号是继承属性



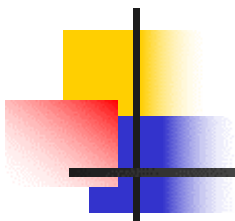
## 语句代码生成（续）

```
void tree::expr_get_label(Node *t) // 布尔表达式生成标号
{
    if (t->type != Boolean) return;
    Node *e1 = t->children[0];      // 第一子表达式
    Node *e2 = t->children[1];      // 第二子表达式
    switch (t->attr.op)
    {
        case AND:
            e1->label.true_label = new_label(); // 子表达式一为真, 跳转到子表达式二代码
            e2->label.true_label = t->label.true_label; // 子表达式二也为真, 与操作结果为真——真值出口是相同标号
            e1->label.false_label = e2->label.false_label = t->label.false_label; // 三者假值出口是相同标号
            break;
        case OR:
            e1->label.false_label = new_label();
            e2->label.false_label = t->label.false_label;
            e1->label.true_label = e2->label.true_label = t->label.true_label;
            break;
        case NOT:
            e1->label.true_label = t->label.false_label;
            e1->label.false_label = t->label.true_label;
            break;
    }
    if (e1) recursive_get_label(e1);
    if (e2) recursive_get_label(e2);
}
```



## 语句代码生成（续）

```
int main(int argc, char *argv[])
{
    int n = 1;
    lexer lexer;
    parser parser;
    if (parser.yycreate(&lexer)) {
        if (lexer.yycreate(&parser)) {
            lexer.yyin = new ifstream(argv[1]);
            lexer.yyout = new ofstream(argv[2]);
            n = parser.yyparse();
            parse_tree.get_label();
            parse_tree.gen_code(*lexer.yyout);
        }
    }
    getchar();
    return n;
}
```



$\varepsilon$ 是\_\_\_\_\_。

- A 终结符
- B 非终结符
- C 空符号
- D 空字符串

提交

对下面CFG构造预测分析表，不正确的说法是

$$\begin{aligned} S &\rightarrow i E t S S' \mid a \\ E &\rightarrow b \\ S' &\rightarrow e S \mid \varepsilon \end{aligned}$$

- A  $S \rightarrow a$ 应填入列a
- B  $S' \rightarrow eS$ 应填入列e
- C  $S' \rightarrow \varepsilon$ 应填入列 $\varepsilon$
- D  $S' \rightarrow \varepsilon$ 应填入列e

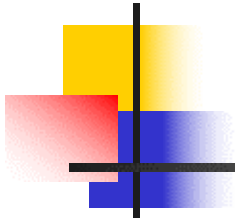
提交

对下面CFG, 正确的说法是\_\_\_\_\_。

$$S \rightarrow TU \quad T \rightarrow 0T1 \mid \varepsilon \quad U \rightarrow 1U0 \mid \varepsilon$$

- ☐ A 接受所有0、1个数相等的0、1串
- ☐ B 存在等价的正则表达式
- ☐ C 是LL(1)文法
- ☐ D 是算符文法

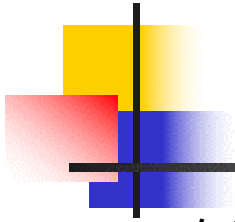
提交



文法G:  $S \rightarrow xSx \mid y$  能够识别的语言是\_\_\_\_\_。

- A  $xyx$
- B  $(xyx)^*$
- C  $x^n y x^n (n \geq 0)$
- D  $x^* y x^*$

提交



在进行同心集合并时可能会产生新的\_\_\_\_冲突。

- A 移进
- B 归约/归约
- C 归约
- D 移进/归约

提交





在LR语法分析栈中存放的状态是识别\_\_\_\_ \_\_\_\_的DFA  
状态。

- ☐ A 最右句型
- ☐ B 最右句柄
- ☐ C 活前缀
- ☐ D 项目

提交



上下文无关文法\_\_\_\_\_产生语言

$$L = \{a^n b^n c^i \mid i \geq 1, n \geq 1\}。$$

- ☐ A 可以
- ☐ B 不可以

提交

对下面CFG，说法错误的是\_\_\_\_\_。

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

- A 消除左递归后即可用预测分析法进行分析
- B 可用算符优先分析算法进行分析
- C 可消除LR分析表中冲突，从而用SLR分析
- D 是二义性文法


提交

对下面CFG, 说法错误的是\_\_\_\_\_。

$S \rightarrow aA$      $A \rightarrow Bb$      $B \rightarrow Ba \mid a$   
 $C \rightarrow Ab$

- A C是无用的
- B 与 $aa^+b$ 对应相同的语言
- C 是算符文法
- D  $aaab$ 是其活前缀

提交

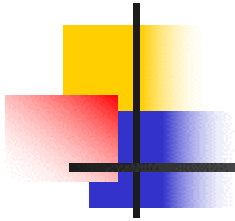


用DAG表示下面表达式，需\_\_\_\_\_个结点。

$$a + (a + a + a * (a + a)) * a$$

- A 4
- B 5
- C 6
- D 7

提交



S属性定义更不容易和\_\_\_\_\_相结合。

- A 预测分析
- B 算符优先分析
- C SLR分析
- D 规范LR分析

提交



如果对类型采用名字等价判定，则披着羊皮的灰太狼会被认为是\_\_\_\_。

- ☐ A 羊
- ☐ B 狼
- ☐ C 灰太狼
- ☐ D 以上皆错

提交



关于下面类型表达式, \_\_\_\_ 是正确的。  
 **$(\text{pointer}(\text{char}) \rightarrow \text{int}) \rightarrow \text{pointer}(\text{char})$**

- A** C语言对这种类型的等价判定采用名字等价方式
- B** Pascal语言采用结构等价方式
- C** 在C语言中, 这种类型会引发类型错误
- D** 以上皆错

提交