编译原理

第六章

中间代码生成

哈尔滨工业大学 陈鄞

本章内容

6.1 声明语句的翻译

- 声明语句翻译的主要任务：收集标识符的类型等属性信息，并为每一个名字分配一个相对地址

名字的类型和相对地址信息保存在相应的符号表记录中

类型表达式 (*Type Expressions*)

- 基本类型是类型表达式
  - *integer*
  - *real*
  - *char*
  - *boolean*
  - *type_error* (出错类型)
  - *void* (无类型)

类型表达式 (*Type Expressions*)

- 基本类型是类型表达式
- 可以为类型表达式命名，类型名也是类型表达式
- 将类型构造符(*type constructor*)作用于类型表达式可以构成新的类型表达式
  - 数组构造符*array*
    - 若*T*是类型表达式，则*array* (*I*, *T*)是类型表达式(*I*是一个整数)

类型	类型表达式
int [3]	*array* (3, *int*)
int [2][3]	*array* (2, *array*(3,*int*))

类型表达式 (*Type Expressions*)

- 基本类型是类型表达式
- 可以为类型表达式命名，类型名也是类型表达式
- 将类型构造符(*type constructor*)作用于类型表达式可以构成新的类型表达式
  - 数组构造符*array*
  - 指针构造符*pointer*
    - 若*T*是类型表达式，则*pointer* (*T*) 是类型表达式，它表示一个指针类型

类型表达式 (*Type Expressions*)

- 基本类型是类型表达式
- 可以为类型表达式命名，类型名也是类型表达式
- 将类型构造符(*type constructor*)作用于类型表达式可以构成新的类型表达式
  - 数组构造符*array*
  - 指针构造符*pointer*
  - 笛卡尔乘积构造符×
    - 若T_1和T_2是类型表达式，则笛卡尔乘积$T_1 \times T_2$是类型表达式

类型表达式 (*Type Expressions*)

- 基本类型是类型表达式
- 可以为类型表达式命名，类型名也是类型表达式
- 将类型构造符(*type constructor*)作用于类型表达式可以构成新的类型表达式
  - 数组构造符*array*
  - 指针构造符*pointer*
  - 笛卡尔乘积构造符×
  - 函数构造符→
    - 若T_1、T_2、…、T_n和R是类型表达式，则$T_1 \times T_2 \times \ldots \times T_n \rightarrow R$是类型表达式

类型表达式 (*Type Expressions*)

- 基本类型是类型表达式
- 可以为类型表达式命名，类型名也是类型表达式
- 将类型构造符(*type constructor*)作用于类型表达式可以构成新的类型表达式
  - 数组构造符*array*
  - 指针构造符*pointer*
  - 笛卡尔乘积构造符×
  - 函数构造符→
  - 记录构造符*record*
    - 若有标识符N_1、N_2、…、N_n与类型表达式T_1、T_2、…、T_n，则 *record* $((N_1 \times T_1) \times (N_2 \times T_2) \times \ldots \times (N_n \times T_n))$ 是一个类型表达式

例

- 设有C程序片段：

```
struct   stype
{   char[8] name;
    int  score;
};
stype[50] table;
stype* p;
```

- 和*stype*绑定的类型表达式
  - *record* ((*name*×*array*(8, *char*)) × (*score* × *integer*))
- 和*table*绑定的类型表达式
  - *array* (50, *stype*)
- 和*p*绑定的类型表达式
  - *pointer* (*stype*)

局部变量的存储分配

- 对于声明语句，语义分析的主要任务就是收集标识符的类型等属性信息，并为每一个名字分配一个相对地址
  - 从类型表达式可以知道该类型在运行时刻所需的存储单元数量称为类型的宽度(*width*)
  - 在编译时刻，可以使用类型的宽度为每一个名字分配一个相对地址
- 名字的类型和相对地址信息保存在相应的符号表记录中

变量声明语句的*SDT*

enter(name, type, offset):在符号表中为名字*name*创建记录,将*name*的类型设置为*type*,相对地址设置为*offset*

① $P \rightarrow \{\ offset = 0\ \}\ D$

② $D \rightarrow T\ \mathrm{id};\{\ enter(\ \mathrm{id}.lexeme,\ T.type,\ offset\ );\ offset = offset + T.width;\ \}D$

③ $D \rightarrow \varepsilon$

④ $T \rightarrow B\ \ \{\ t = B.type;\ w = B.width;\}\ C\ \ \{\ T.type = C.type;\ T.width = C.width;\ \}$

⑤ $T \rightarrow \uparrow T_1\{\ T.type = pointer(\ T_1.type);\ T.width = 4;\ \}$

⑥ $B \rightarrow \mathrm{int}\ \{\ B.type = int;\ B.width = 4;\ \}$

⑦ $B \rightarrow \mathrm{real}\{\ B.type = real;\ B.width = 8;\ \}$

⑧ $C \rightarrow \varepsilon\ \ \{\ C.type = t;\ C.width = w;\ \}$

⑨ $C \rightarrow [\mathrm{num}]C_1\ \{\ C.type = array(\ num.val,\ C_1.type);\ C.width = num.val * C_1.width;\ \}$

符号	综合属性
B	*type, width*
C	*type, width*
T	*type, width*

变量	作用
offset	下一个可用的相对地址
t, w	将类型和宽度信息从语法分析树中的*B*结点传递到对应于产生式$C \rightarrow \varepsilon$的结点

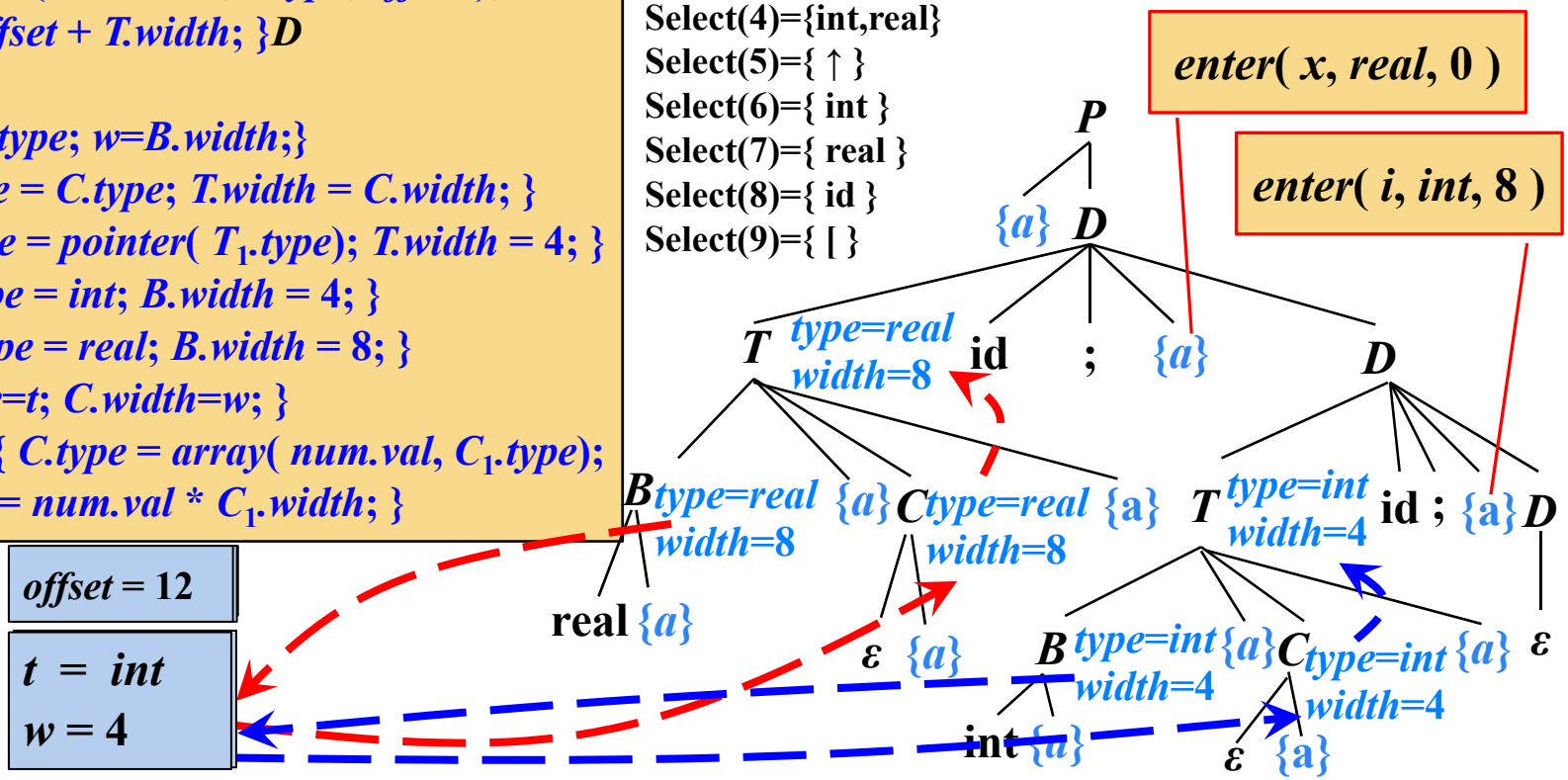
例：“*real x; int i;*”的语法制导翻译

①$P \rightarrow$ { *offset = 0* } D
②$D \rightarrow T$ id;{ *enter(id.lexeme, T.type, offset);*
offset = offset + T.width; }D
③$D \rightarrow \varepsilon$
④$T \rightarrow B$ { *t =B.type; w=B.width;*}
C { *T.type = C.type; T.width = C.width;* }
⑤$T \rightarrow \uparrow T_1$ { *T.type = pointer(T_1.type); T.width = 4;* }
⑥$B \rightarrow$ int { *B.type = int; B.width = 4;* }
⑦$B \rightarrow$ real{ *B.type = real; B.width = 8;* }
⑧$C \rightarrow \varepsilon$ { *C.type=t; C.width=w;* }
⑨$C \rightarrow$ [num]C_1 { *C.type = array(num.val, C_1.type);*
*C.width = num.val * C_1.width;* }

Select(1)={int,real, ↑,$}
Select(2)={int,real, ↑}
Select(3)={$}
Select(4)={int,real}
Select(5)={ ↑ }
Select(6)={ int }
Select(7)={ real }
Select(8)={ id }
Select(9)={ [}

enter(x, real, 0)

enter(i, int, 8)



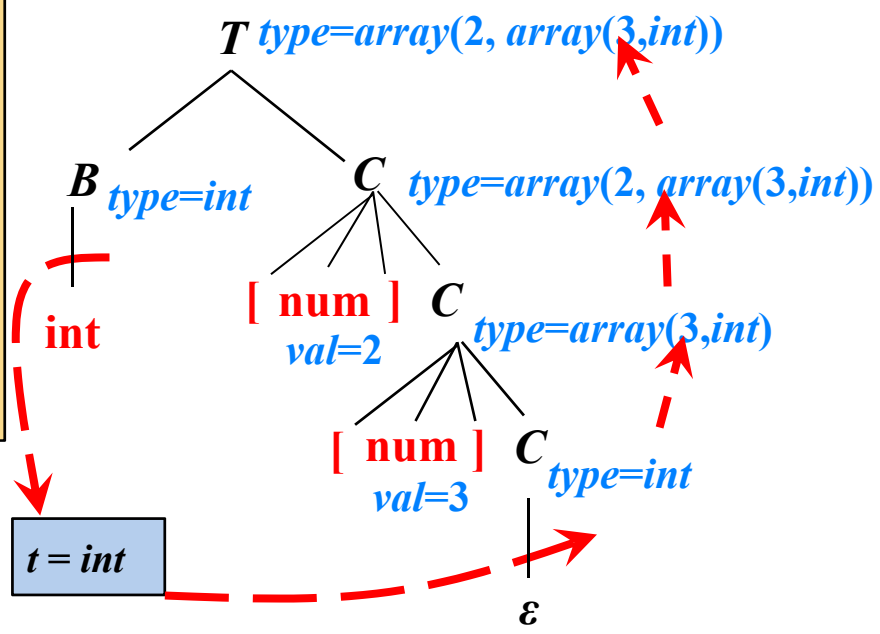
offset = 12

t = *int*
w = 4

数组类型表达式的语法制导翻译

①$P \rightarrow$ { $offset = 0$ } D
②$D \rightarrow T$ id; { $enter($ id.$lexeme, T.type, offset$); $offset = offset + T.width$; } D
③$D \rightarrow \varepsilon$
④$T \rightarrow B$ { $t = B.type$; $w = B.width$; } C { $T.type = C.type$; $T.width = C.width$; }
⑤$T \rightarrow \uparrow T_1$ { $T.type = pointer(T_1.type)$; $T.width = 4$; }
⑥$B \rightarrow$ int { $B.type = int$; $B.width = 4$; }
⑦$B \rightarrow$ real { $B.type = real$; $B.width = 8$; }
⑧$C \rightarrow \varepsilon$ { $C.type = t$; $C.width = w$; }
⑨$C \rightarrow$ [num]C_1 { $C.type = array(num.val, C_1.type)$; $C.width = num.val * C_1.width$; }

例： “*int*[2][3]”



例：数组类型表达式“*int*[2][3]”的语法制导翻译

①$P \rightarrow$ { *offset* = 0 } *D*

②$D \rightarrow T$ id;{ *enter*(id.*lexeme*, *T.type*, *offset*);
offset = *offset* + *T.width*; }*D*

③$D \rightarrow \varepsilon$

④$T \rightarrow B$ { *t* =*B.type*; *w*=*B.width*;}
C { *T.type* = *C.type*; *T.width* = *C.width*; }

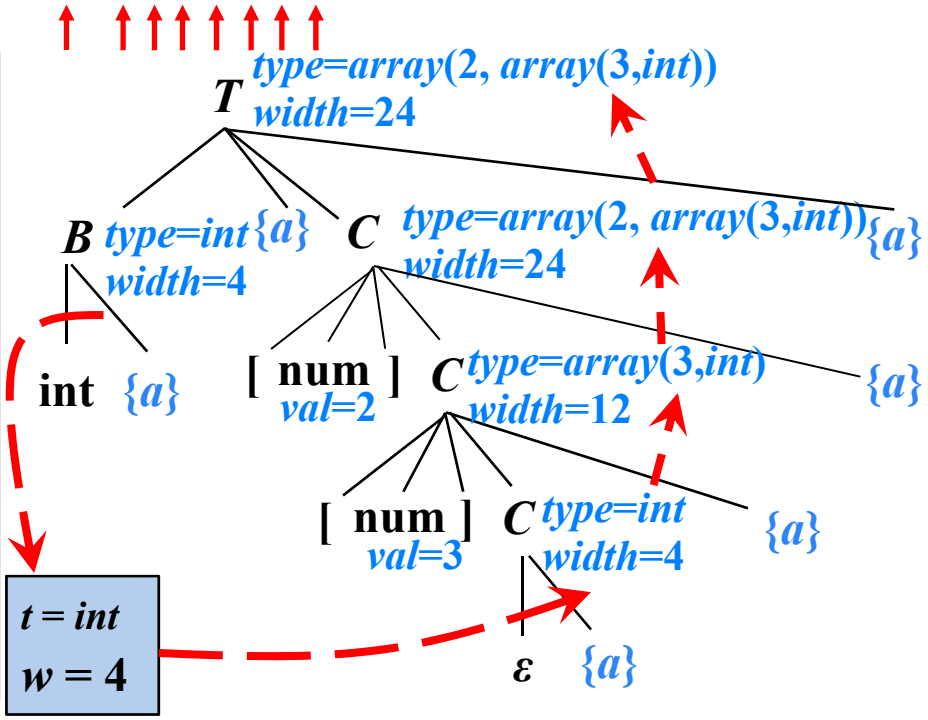
⑤$T \rightarrow \uparrow T_1$ { *T.type* = *pointer*(T_1.*type*); *T.width* = 4; }

⑥$B \rightarrow$ int { *B.type* = *int*; *B.width* = 4; }

⑦$B \rightarrow$ real{ *B.type* = *real*; *B.width* = 8; }

⑧$C \rightarrow \varepsilon$ { *C.type*=*t*; *C.width*=*w*; }

⑨$C \rightarrow$ [num]C_1 { *C.type* = *array*(*num.val*, C_1.*type*);
C.width = *num.val* * C_1.*width*; }



提纲

6.2 赋值语句的翻译

- 6.2.1简单赋值语句的翻译
- 6.2.2数组引用的翻译

6.2.1 简单赋值语句的翻译

- 赋值语句的基本文法
  - ① $S \rightarrow \mathrm{id} = E;$
  - ② $E \rightarrow E_1 + E_2$
  - ③ $E \rightarrow E_1 * E_2$
  - ④ $E \rightarrow -E_1$
  - ⑤ $E \rightarrow (E_1)$
  - ⑥ $E \rightarrow \mathrm{id}$
- 赋值语句翻译的主要任务
  - 生成对表达式求值的三地址码
- 例
  - 源程序片段
    - $x = (a + b) * c;$
  - 三地址码
    - $t_1 = a + b$
    - $t_2 = t_1 * c$
    - $x = t_2$

赋值语句的*SDT*

lookup(name)：查询符号表，返回*name* 对应的记录

gen(code)：生成三地址指令*code*

newtemp( )：生成一个新的临时变量*t*，返回*t*的地址

$S \rightarrow \text{id} = E;$ { *p = lookup*(id.*lexeme*); *if p == nil then error* ;
S.code = E.code ||
gen(p '=' *E.addr*); }

$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( )*;
E.code = E_1*.code* || E_2*.code*||
gen(E.addr '=' E_1*.addr* '+' E_2*.addr*); }

$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( )*;
E.code = E_1*.code* || E_2*.code* ||
gen(E.addr '=' E_1*.addr* '*' E_2*.addr*); }

$E \rightarrow -E_1$ { *E.addr = newtemp( )*;
E.code = E_1*.code*||
gen(E.addr '=' '*uminus*' E_1*.addr*); }

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

$E \rightarrow \text{id}$ { *E.addr = lookup*(id.*lexeme*); *if E.addr == nil then error* ;
E.code = ''; }

符号	综合属性
S	*code*
E	*code* *addr*

增量翻译 (*Incremental Translation*)

$S \rightarrow$ id $= E$; { *p = lookup*(id.*lexeme*); *if p == nil then error* ;
S.code = E.code ||
gen(*p* '=' *E.addr*); }

$E \rightarrow E_1 + E_2$ { *E.addr = newtemp*( );
E.code = E_1*.code* || E_2*.code*||
gen(*E.addr* '=' E_1*.addr* '+' E_2*.addr*); }

$E \rightarrow E_1 * E_2$ { *E.addr = newtemp*( );
E.code = E_1*.code* || E_2*.code* ||
gen(*E.addr* '=' E_1*.addr* '*' E_2*.addr*); }

$E \rightarrow -E_1$ { *E.addr = newtemp*( );
E.code = E_1*.code*||
gen(*E.addr* '=' '*uminus*' E_1*.addr*); }

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

$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*); *if E.addr == nil then error* ;
E.code = ''; }

在增量方法中，*gen*( )不仅要构造出一个新的三地址指令，还要将它添加到至今为止已生成的指令序列之后

例

①$S \rightarrow$ id = E; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

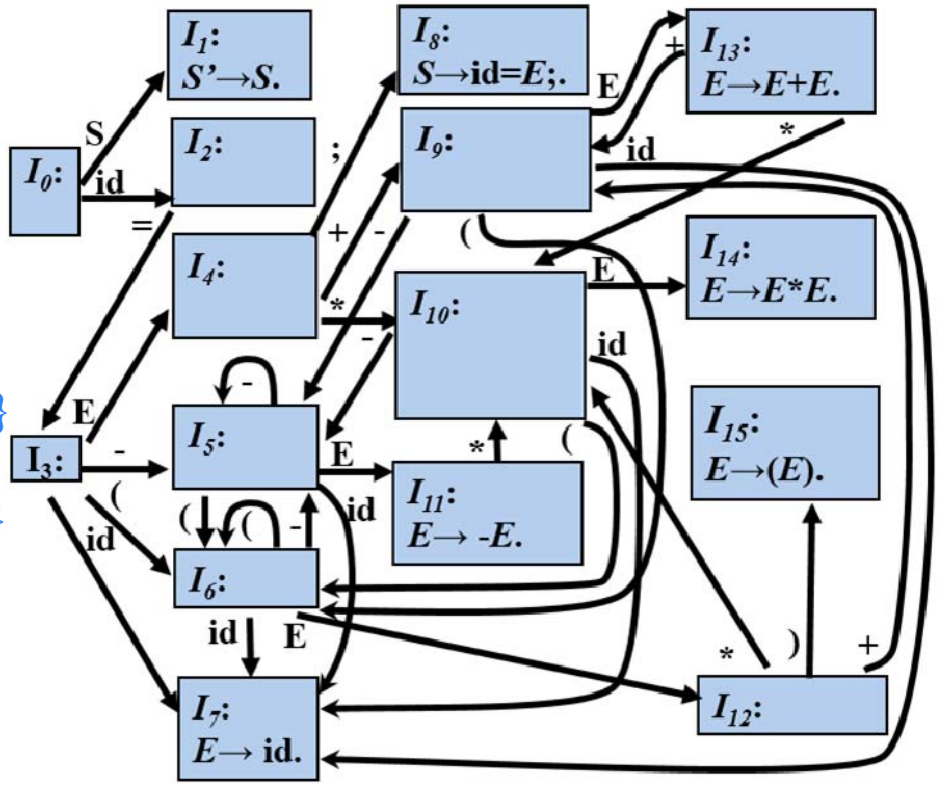
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	6	7
$	id	=	(	id
	x			*a*

例

①$S \rightarrow$ id = E; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

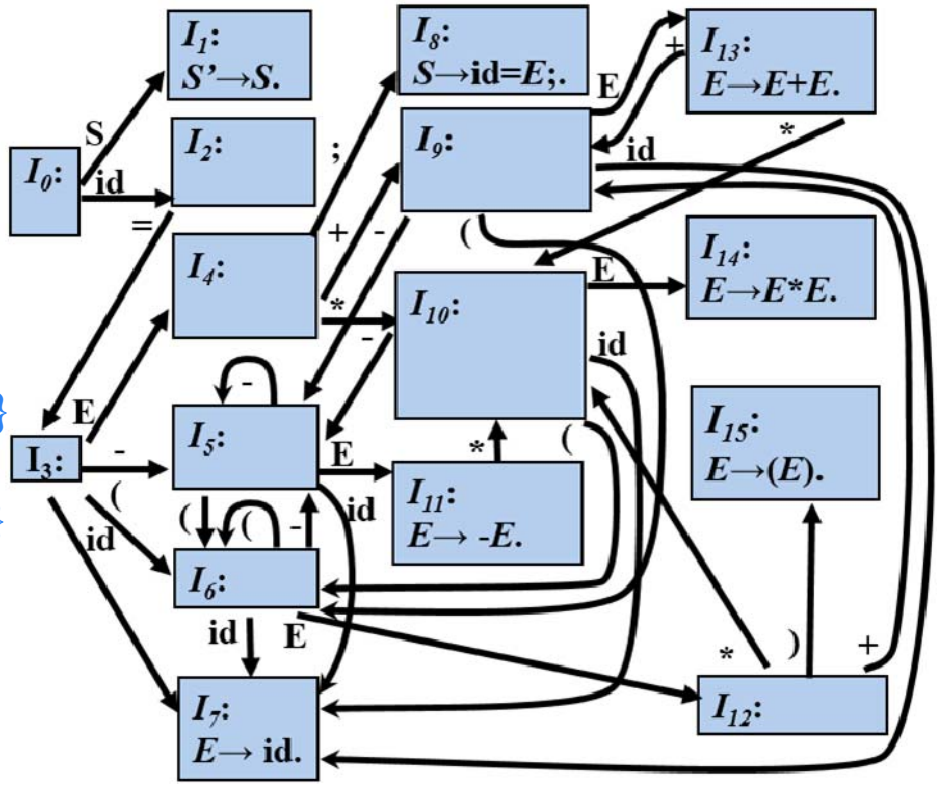
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	6	12	9	7
$	id	=	(	*E*	+	id
	x			*a*		*b*

例

①$S \rightarrow$ id = E; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

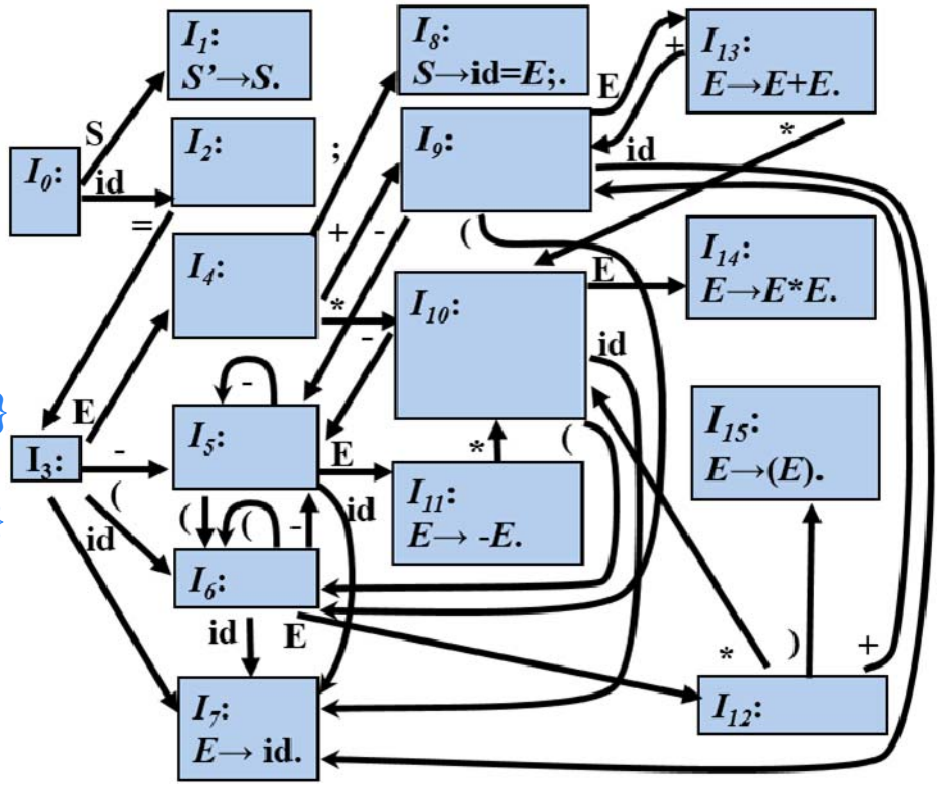
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	6	12	9	13
$	id	=	(	E	+	E
	x			a		b

$t_1 = a + b$

例

①$S \rightarrow$ id $= E$; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

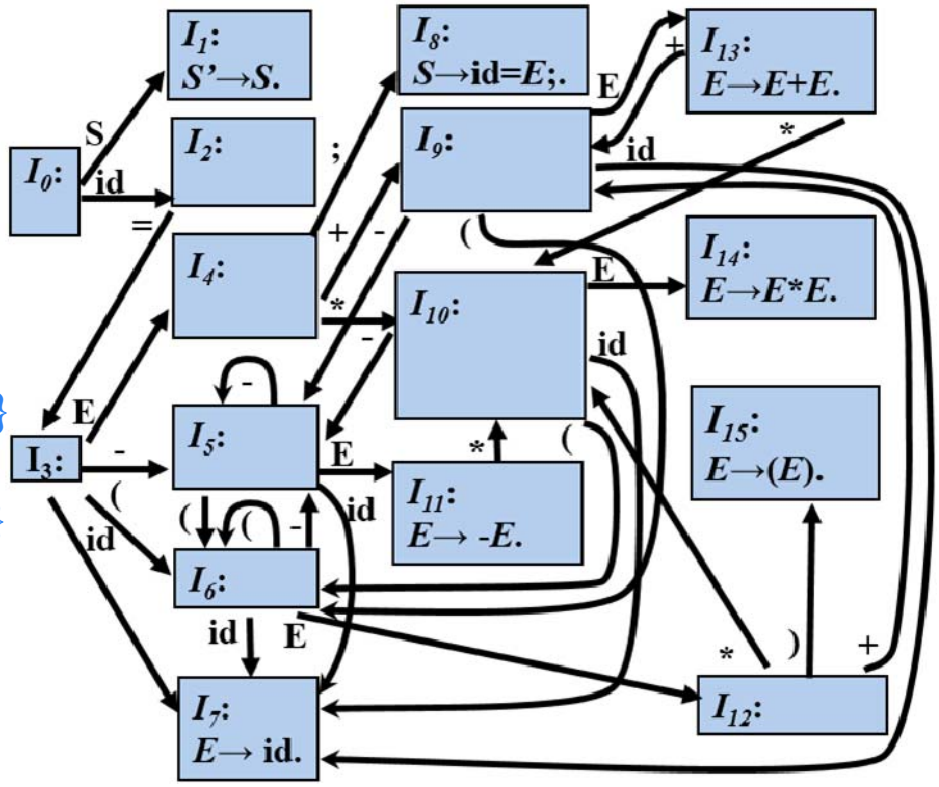
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	6	12	15
$	id	=	(	*E*	)
	x			t_1	

$t_1 = a + b$

例

①$S \rightarrow$ id = E; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

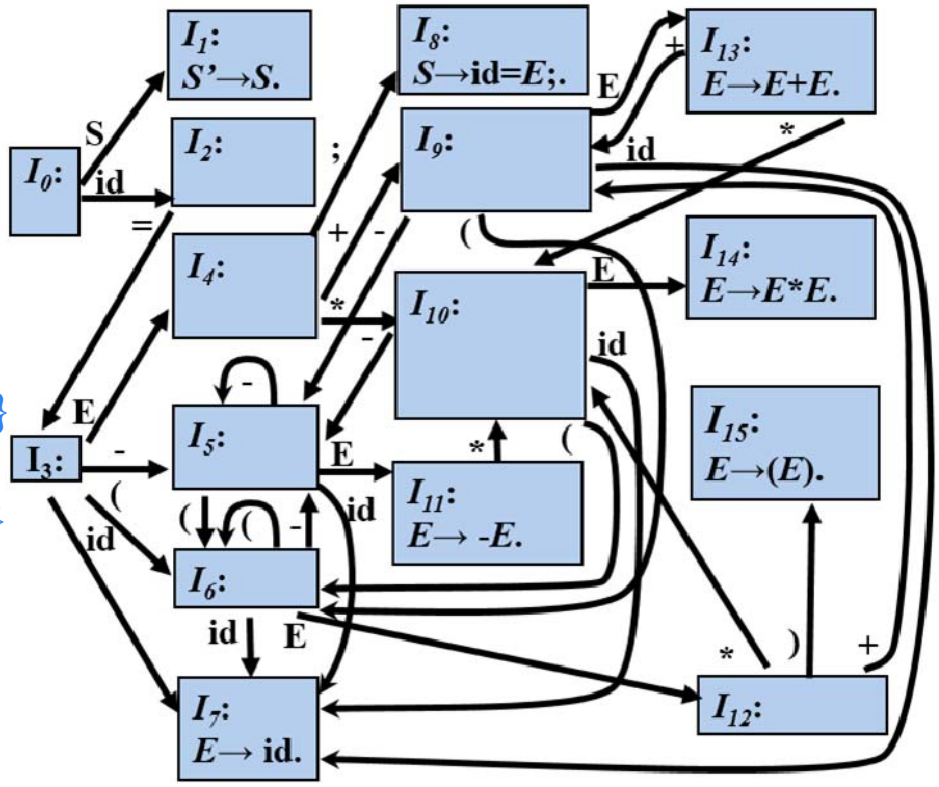
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	4	10	7
$	id	=	E	*	id
	x		t_1		c

$t_1 = a + b$

例

①$S \rightarrow$ id = E; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

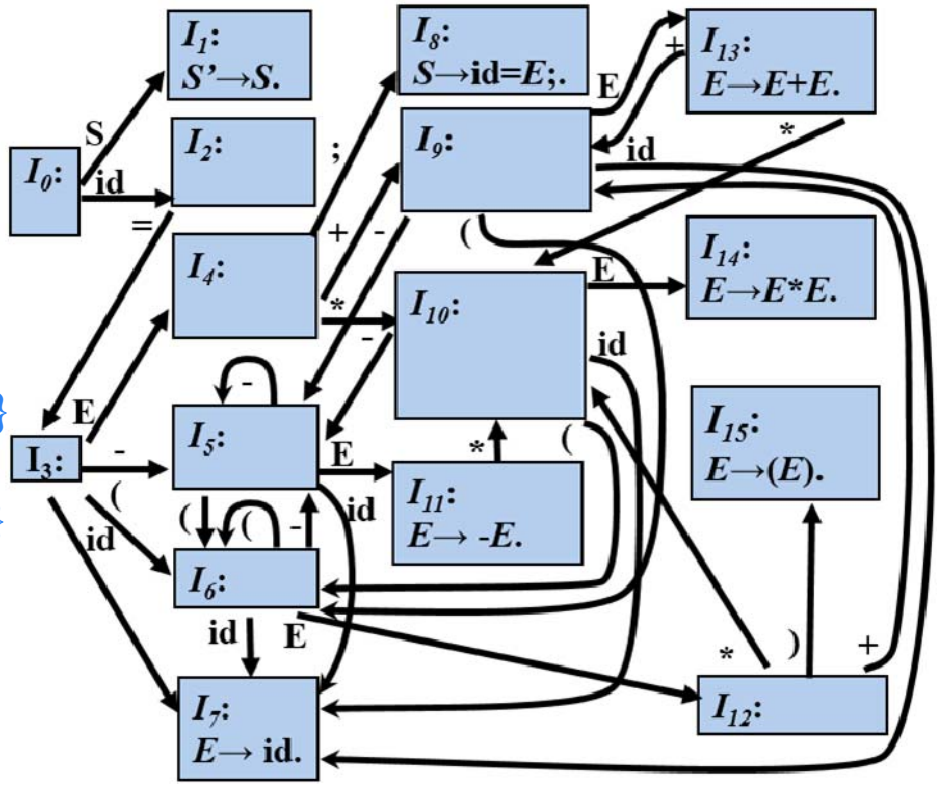
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	4	10	14
$	id	=	E	*	E
	x		t_1		c

$t_1 = a + b$

$t_2 = t_1 * c$

例

①$S \rightarrow$ id $= E$; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

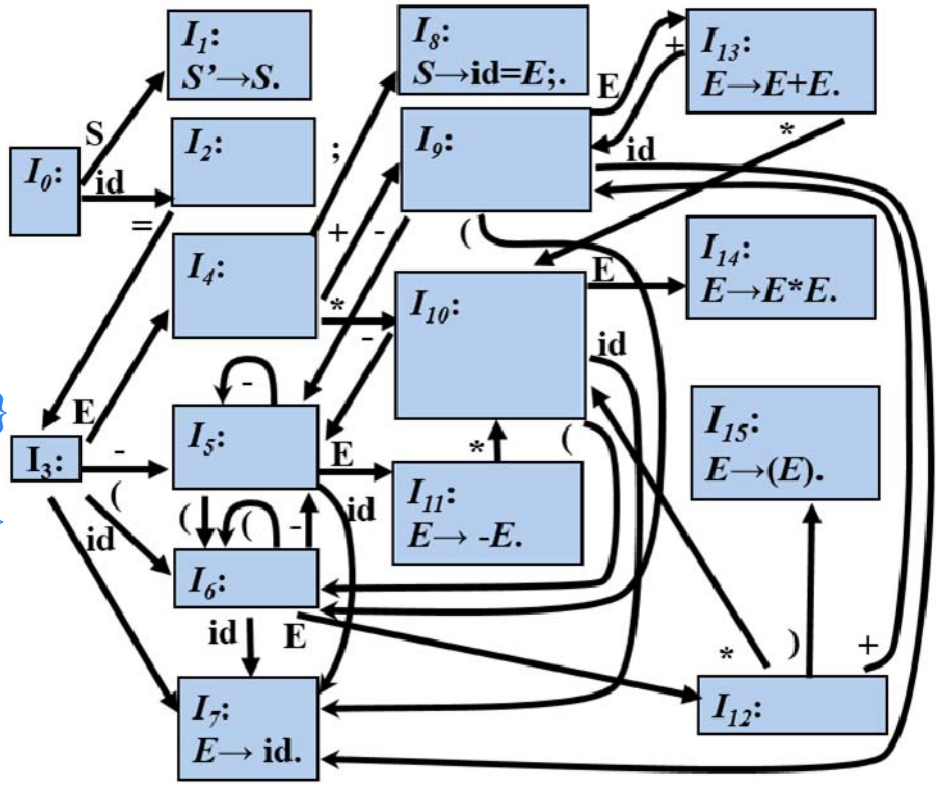
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr);* }

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ;* }

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	4	8
$	id	=	E	;
	x		t_2	

$t_1 = a + b$

$t_2 = t_1 * c$

$x = t_2$

例

①$S \rightarrow$ id $= E$; { *p = lookup*(id.*lexeme*);
if p==nil then error ;
gen(p '=' E.addr); }

②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr);* }

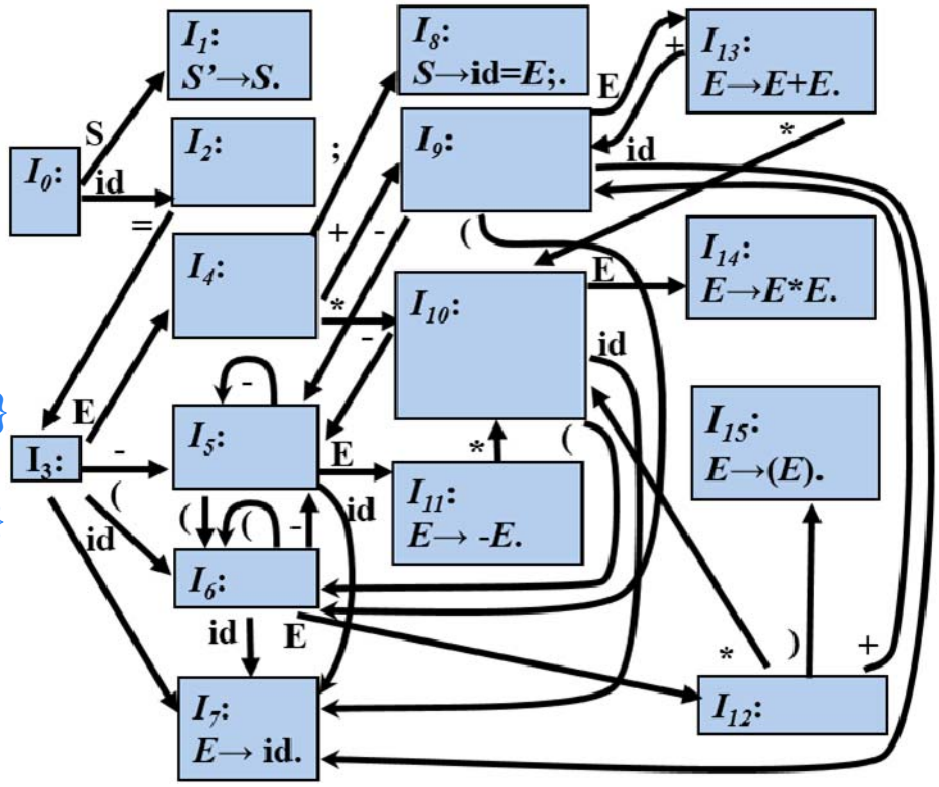
④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ;* }

⑥$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*);
if E.addr==nil then error ; }

例：$x = (a + b) * c$;

0	1
$	S



$t_1 = a + b$

$t_2 = t_1 * c$

$x = t_2$

6.2.2 数组引用的翻译

- 赋值语句的基本文法

$S \rightarrow \mathrm{id} = E; \mid L = E;$

$E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \mathrm{id} \mid L$

$L \rightarrow \mathrm{id}\ [E] \mid L_1\ [E]$

将数组引用翻译成三地址码时要解决的主要问题是确定数组元素的存放地址，也就是数组元素的寻址

数组元素寻址 (*Addressing Array Elements*)

- 一维数组
  - 假设每个数组元素的宽度是w，则数组元素$a[i]$的相对地址是：

$$base+i\times w$$

其中，$base$是数组的基地址，$i\times w$是偏移地址

- 二维数组
  - 假设一行的宽度是w_1，同一行中每个数组元素的宽度是w_2，则数组元素$a[i_1]\ [i_2]$的相对地址是：

$$base+\underbrace{i_1\times w_1+i_2\times w_2}_{\text{偏移地址}}$$

- k维数组
  - 数组元素$a[i_1]\ [i_2]\ \ldots[i_k]$的相对地址是：

$$base+\underbrace{i_1\times w_1+i_2\times w_2+\ldots+i_k\times w_k}_{\text{偏移地址}}$$

$w_1 \rightarrow a[i_1]$ 的宽度
$w_2 \rightarrow a[i_1]\ [i_2]$ 的宽度
…
$w_k \rightarrow a[i_1]\ [i_2]\ \ldots[i_k]$的宽度

例

- 假设*type*(*a*)= *array*(3, *array*(5, *array*(8, *int*))),

  一个整型变量占用4个字节，

  则*addr*($a[i_1][i_2][i_3]$) = *base* + i_1 * w_1 + i_2*w_2 + i_3*w_3

  = *base* + i_1*160 + i_2*32 + i_3*4

$a[i_1]$的宽度

$a[i_1][i_2]$的宽度

$a[i_1][i_2][i_3]$的宽度

带有数组引用的赋值语句的翻译

- 例1

  假设 *type*(*a*)=*array*(*n*, *int*),

  - 源程序片段
    - $c = a[i];$

    $addr(a[i]) = base + i*4$

  - 三地址码
    - $t_1 = i * 4$
    - $t_2 = a[t_1]$
    - $c = t_2$

带有数组引用的赋值语句的翻译

- 例2

  假设 $type(a) = array(3, array(5, int))$，

  - 源程序片段
    - $c = a[i_1][i_2];$ $addr(a[i_1][i_2]) = base + i_1*20 + i_2*4$
  - 三地址码
    - $t_1 = i_1 * 20$
    - $t_2 = i_2 * 4$
    - $t_3 = t_1 + t_2$
    - $t_4 = a[t_3]$
    - $c = t_4$

t_1 t_2

数组元素寻址的*SDT*

- 赋值语句的基本文法

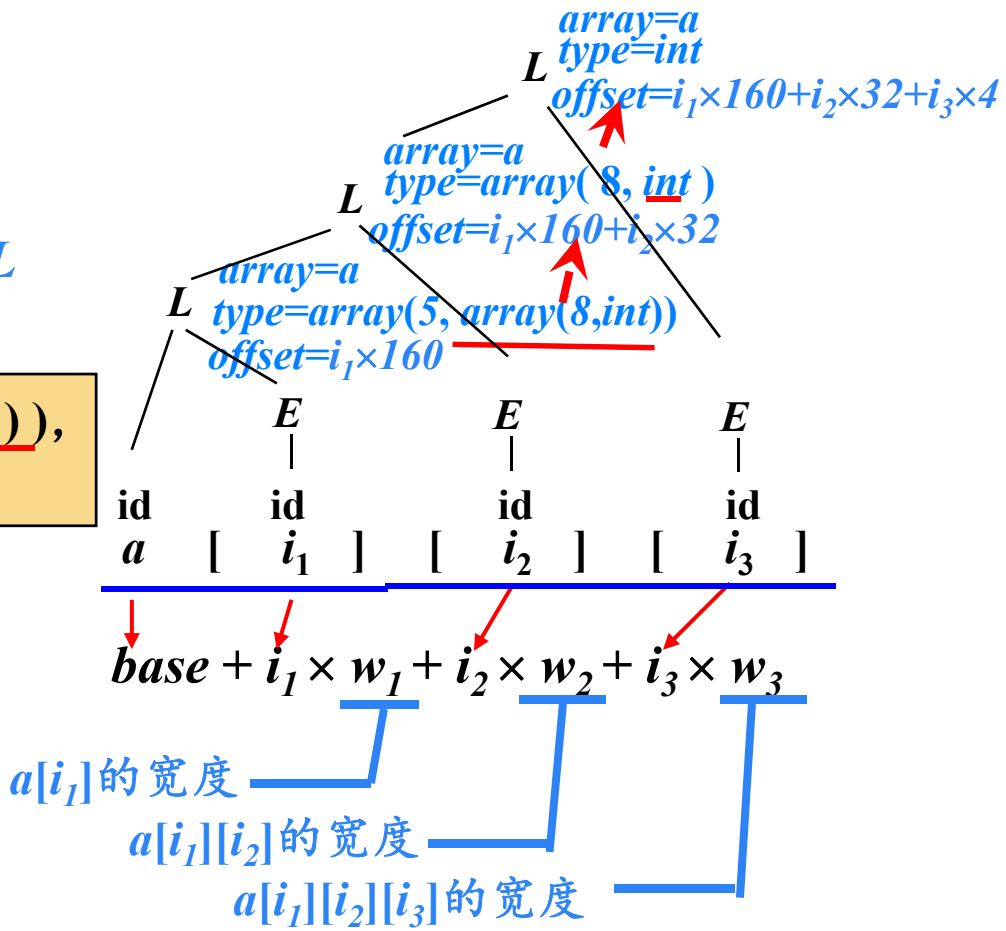
$S \rightarrow \mathrm{id} = E; \mid L = E;$

$E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \mathrm{id} \mid L$

$L \rightarrow \mathrm{id}\ [E] \mid L_1\ [E]$

假设 *type*(*a*)= *array*(3, *array*(5, *array*(8, *int*))),
翻译语句片段“$a[i_1][i_2][i_3]$”

- *L*的综合属性
  - *L.type*：*L*生成的数组元素的类型
  - *L.offset*：指示一个临时变量，该临时变量用于累加公式中的$i_j \times w_j$项，从而计算数组元素的偏移量
  - *L.array*：数组名在符号表的入口地址



数组元素寻址的SDT

假设 $type(a)= array(3, array(5, array(8, int))$，翻译语句片段“$a[i_1][i_2][i_3]$”

$addr(a[i_1][i_2][i_3]) = base + i_1 \times w_1 + i_2 \times w_2 + i_3 \times w_3$

t_1 t_2 t_4

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

| $L = E;$ { $gen(L.array$ ‘[’ $L.offset$ ‘]’ ‘=’ $E.addr$); }

$E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \text{id}$

| L { $E.addr = newtemp()$; $gen(E.addr$ ‘=’ $L.array$ ‘[’ $L.offset$ ‘]’); }

$L \rightarrow \text{id}\ [E]$ { $L.array = lookup(\text{id}.lexeme)$; if $L.array == nil$ then error ;

$L.type = L.array.type.elem$;

$L.offset = newtemp()$;

$gen(L.offset$ ‘=’ $E.addr$ ‘*’ $L.type.width$); }

| $L_1[E]$ { $L.array = L_1.array$;

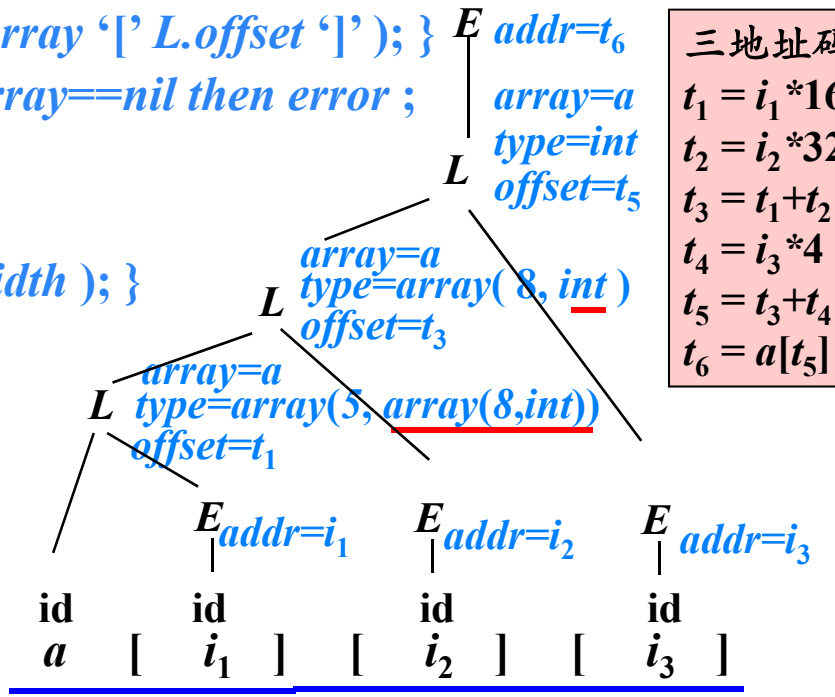
$L.type = L_1.type.elem$;

$t = newtemp()$;

$gen(t$ ‘=’ $E.addr$ ‘*’ $L.type.width$);

$L.offset = newtemp()$;

$gen(L.offset$ ‘=’ $L_1.offset$ ‘+’ t); }



三地址码

$t_1 = i_1 * 160$

$t_2 = i_2 * 32$

$t_3 = t_1 + t_2$

$t_4 = i_3 * 4$

$t_5 = t_3 + t_4$

$t_6 = a[t_5]$

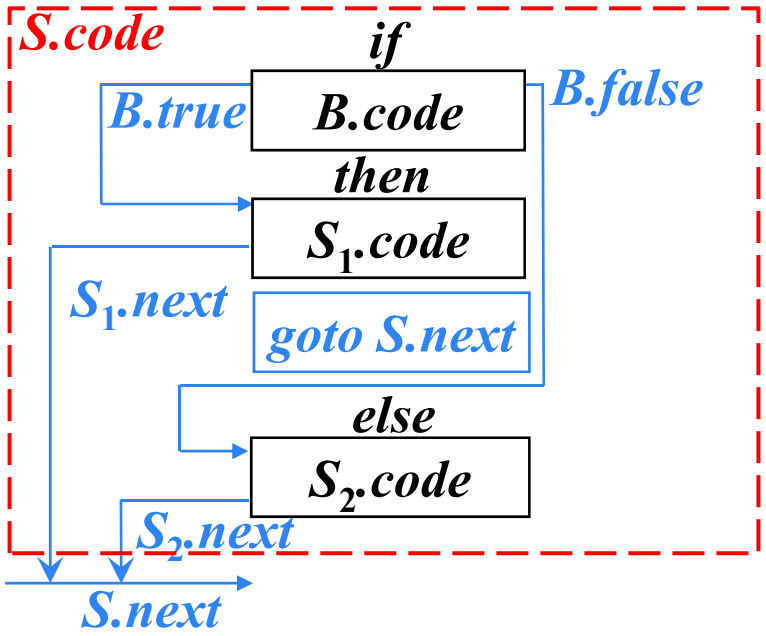
提纲

6.3 控制语句的翻译

- 控制流语句的基本文法
  - $P \rightarrow S$
  - $S \rightarrow S_1 S_2$
  - $S \rightarrow \text{id} = E\ ; \mid L = E\ ;$
  - $S \rightarrow \text{if } B \text{ then } S_1$
    $\mid \text{if } B \text{ then } S_1 \text{ else } S_2$
    $\mid \text{while } B \text{ do } S_1$

控制流语句的代码结构

➢ 例 $S \rightarrow if\ B\ then\ S_1\ else\ S_2$



布尔表达式*B*被翻译成由跳转指令构成的跳转代码

➢ 继承属性

- ***B.true***：是一个地址，该地址用来存放当*B*为真时控制流转向的指令的标号
- ***B.false***：是一个地址，该地址用来存放当*B*为假时控制流转向的指令的标号
- ***S.next***：是一个地址，该地址用来存放紧跟在*S*代码之后的指令(*S*的后继指令)的标号

用指令的标号标识一条三地址指令

控制流语句的*SDT*

newlabel( ): 生成一个用于存放标号的新的临时变量*L*，返回变量地址

- $P \rightarrow \{\ S.next = newlabel();\ \} S \{\ label(S.next);\}$
- $S \rightarrow \{\ S_1.next = newlabel();\ \}\ S_1$

  $\{\ label(S_1.next);\ S_2.next = S.next\ ;\ \}\ S_2$
- $S \rightarrow \mathbf{id} = E\ ;\ |\ L = E\ ;$
- $S \rightarrow if\ B\ then\ S_1$

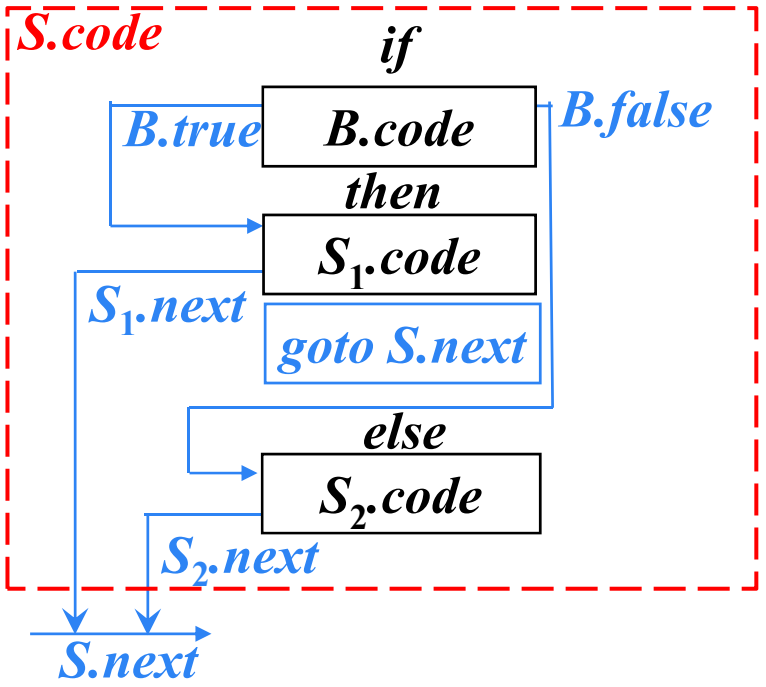
  $|\ if\ B\ then\ S_1\ else\ S_2$

  $|\ while\ B\ do\ S_1$

label(*L*): 将下一条三地址指令的标号存放到地址*L*中

*if-then-else*语句的*SDT*

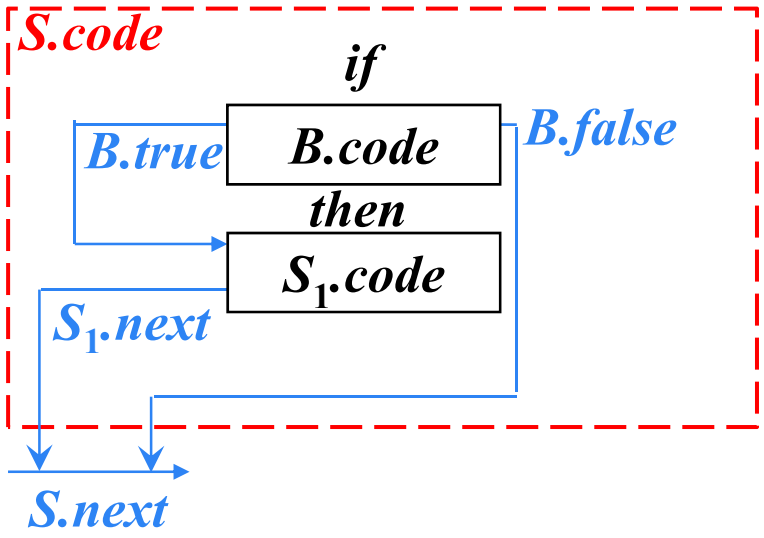
$S \rightarrow$ *if B then* S_1 *else* S_2



$S \rightarrow$ *if* { *B.true = newlabel(); B.false = newlabel();* } *B*
then { *label(B.true);* S_1*.next = S.next;* } S_1 { *gen('goto' S.next)* }
else { *label(B.false);* S_2*.next = S.next;* }S_2

*if-then*语句的*SDT*

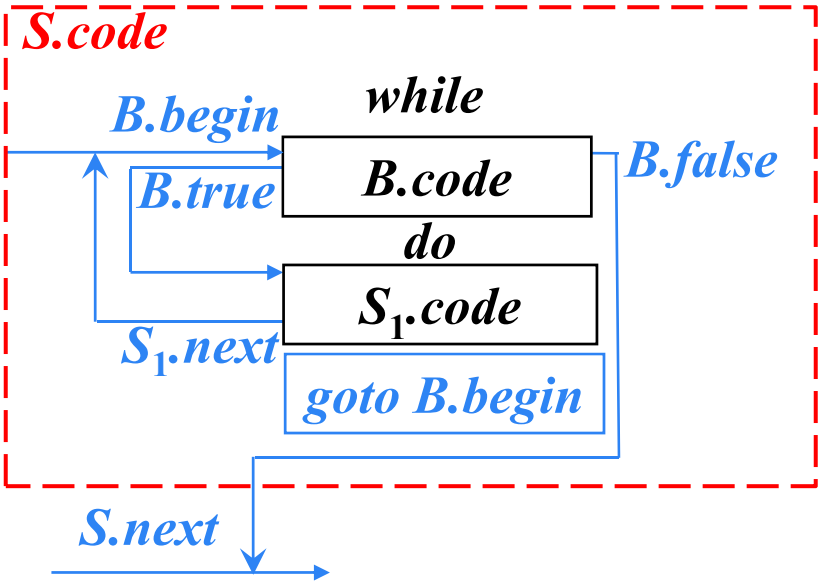
$S \rightarrow if\ B\ then\ S_1$



$S \rightarrow if\ \{\ B.true = newlabel();\ B.false = S.next;\ \}\ B$
$then\ \{\ label(B.true);\ S_1.next = S.next;\ \}\ S_1$

*while-do*语句的*SDT*

$S \rightarrow$ *while* B *do* S_1



$S \rightarrow$ *while* {*B.begin* = *newlabel*();
label(*B.begin*);
B.true = *newlabel*();
B.false = *S.next*;} B
do { *label*(*B.true*); S_1*.next* = *B.begin*;} S_1
{ *gen*('*goto*' *B.begin*); }

布尔表达式的翻译

➢布尔表达式的基本文法

$B \rightarrow B$ or B
| B and B
| not B
| (B)
| E relop E
| true
| false

优先级：not > and > or

关系表达式

relop（关系运算符）：
<, <=, >, >=, ==, !=

- 在跳转代码中，逻辑运算符&&、||和！被翻译成跳转指令。运算符本身不出现在代码中，布尔表达式的值是通过代码序列中的位置来表示的
- 例
  - 语句

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

  - 三地址代码

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

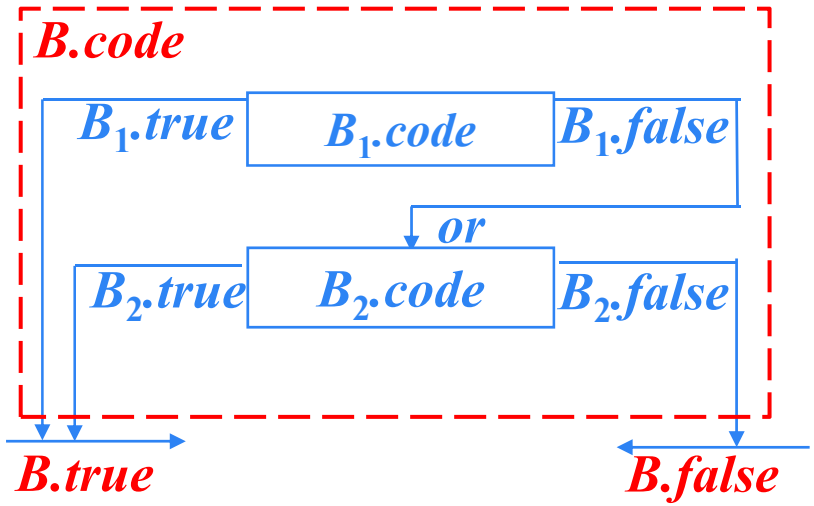
布尔表达式的*SDT*

- $B \rightarrow E_1$ relop E_2 { *gen*(‘*if* ’ E_1.*addr relop* E_2.*addr* ‘*goto*’ *B.true*);
  gen(‘*goto*’ *B.false*); }
- $B \rightarrow$ true { *gen*(‘*goto*’ *B.true*); }
- $B \rightarrow$ false { *gen*(‘*goto*’ *B.false*); }
- $B \rightarrow$ ({B_1.*true* =*B. true*; B_1.*false* =*B.false*; } B_1)
- $B \rightarrow$ not { B_1.*true* = *B.false*; B_1.*false* = *B.true*; } B_1

$B \rightarrow B_1$ or B_2 的SDT

- $B \rightarrow B_1$ *or* B_2
  - $B \rightarrow \{ B_1.true = B.true;\ B_1.false = newlabel(); \} B_1$

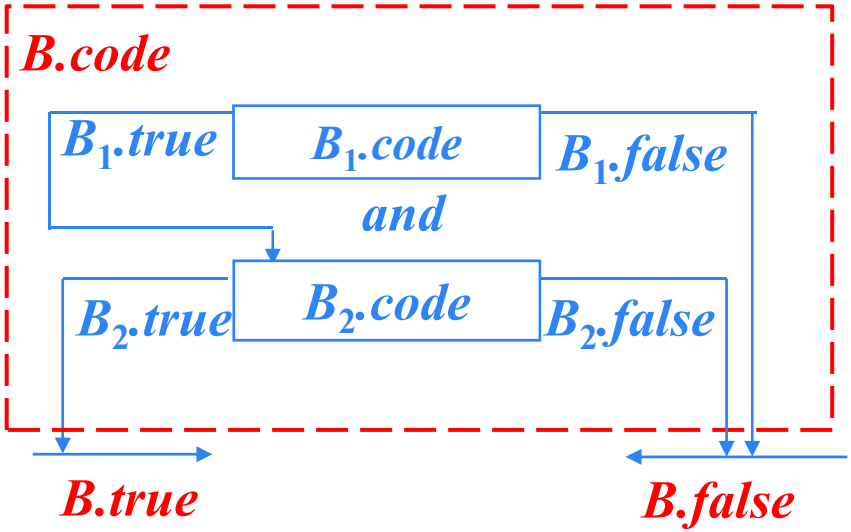
    or $\{ label(B_1.false);\ B_2.true = B.true;\ B_2.false = B.false; \} B_2$



$B \rightarrow B_1$ and B_2 的*SDT*

- $B \rightarrow B_1$ *and* B_2
  - $B \rightarrow \{\ B_1.true = newlabel();\ B_1.false = B.false;\ \}\ B_1$

    and $\{\ label(B_1.true);\ B_2.true = B.true;\ B_2.false = B.false;\ \} B_2$



控制流语句的SDT

- $P \rightarrow \{a\}S\{a\}$
- $S \rightarrow \{a\}S_1\{a\}S_2$
- $S \rightarrow \text{id}=E;\{a\} \mid L=E;\{a\}$
- $E \rightarrow E_1+E_2\{a\} \mid -E_1\{a\} \mid (E_1)\{a\} \mid \text{id}\{a\} \mid L\{a\}$
- $L \rightarrow \text{id}[E]\{a\} \mid L_1[E]\{a\}$
- $S \rightarrow \text{if } \{a\}B \text{ then } \{a\}S_1$
  $\mid \text{if } \{a\}B \text{ then } \{a\}S_1 \text{ else } \{a\}S_2$
  $\mid \text{while } \{a\}B \text{ do } \{a\}S_1\{a\}$
- $B \rightarrow \{a\}B \text{ or } \{a\}B \mid \{a\}B \text{ and } \{a\}B \mid \text{not } \{a\}B \mid (\{a\}B)$
  $\mid E \text{ relop } E\{a\} \mid \text{true}\{a\} \mid \text{false}\{a\}$

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

*SDT*的通用实现方法

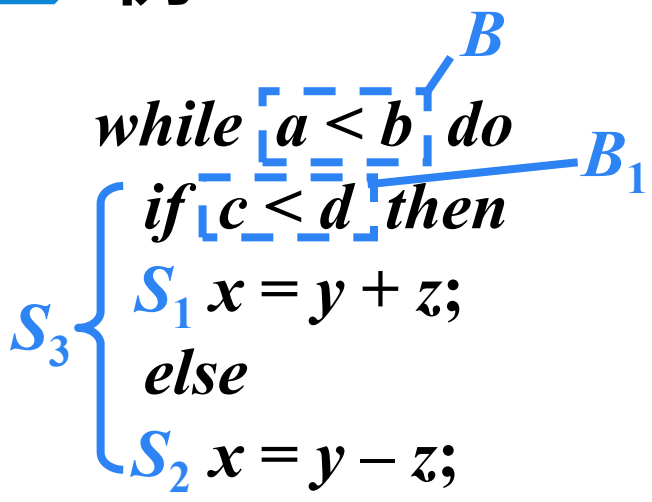
- 任何*SDT*都可以通过下面的方法实现
  - 首先建立一棵语法分析树，然后按照从左到右的深度优先顺序来执行这些动作

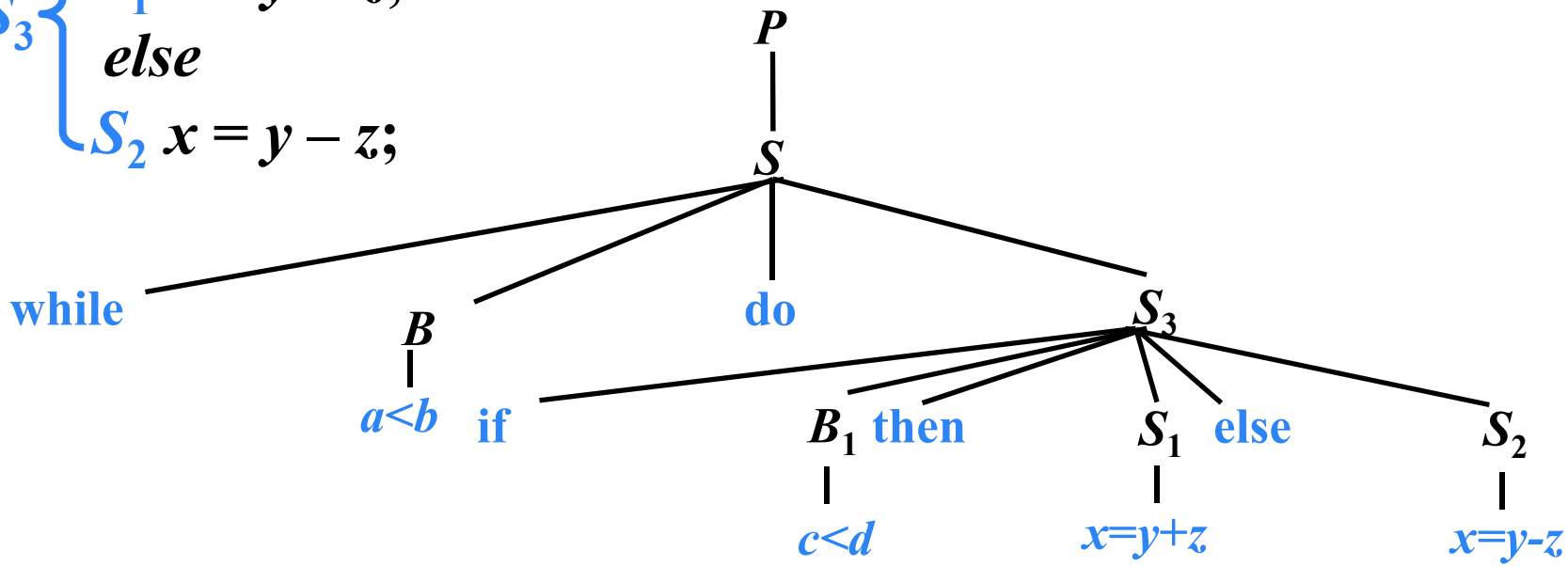
控制流语句的*SDT*

- $P \rightarrow \{a\}S\{a\}$
- $S \rightarrow \{a\}S_1\{a\}S_2$
- $S \rightarrow \text{id}=E;\{a\} \mid L=E;\{a\}$
- $E \rightarrow E_1+E_2\{a\} \mid -E_1\{a\} \mid (E_1)\{a\} \mid \text{id}\{a\} \mid L\{a\}$
- $L \rightarrow \text{id}[E]\{a\} \mid L_1[E]\{a\}$
- $S \rightarrow \text{if } \{a\}B \text{ then } \{a\}S_1$
  $\mid \text{if } \{a\}B \text{ then } \{a\}S_1 \text{ else } \{a\}S_2$
  $\mid \text{while } \{a\}B \text{ do } \{a\}S_1\{a\}$
- $B \rightarrow \{a\}B \text{ or } \{a\}B \mid \{a\}B \text{ and } \{a\}B \mid \text{not } \{a\}B \mid (\{a\}B)$
  $\mid E \text{ relop } E\{a\} \mid \text{true}\{a\} \mid \text{false}\{a\}$

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

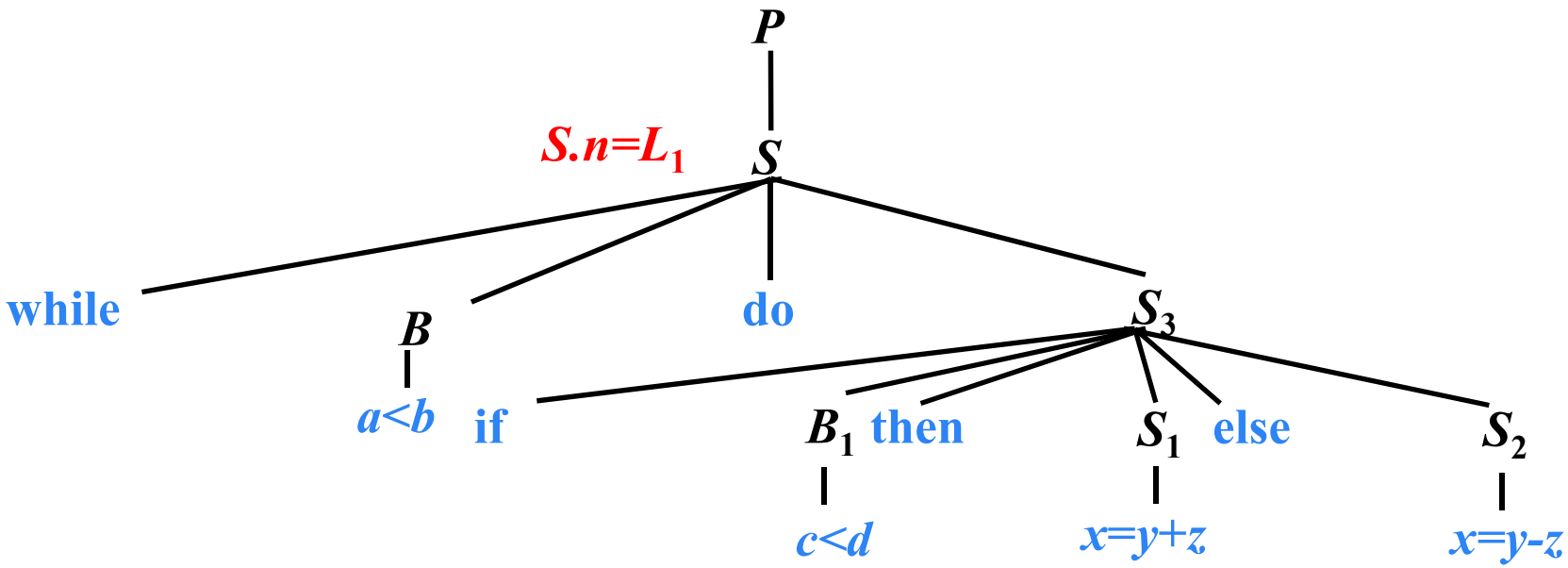
例


B
while a < b do
B1
if c < d then
S3
S1 x = y + z;
else
S2 x = y – z;


P
S
while
B
a<b
do
S3
if
B1
c<d
then
S1
x=y+z
else
S2
x=y-z

例

$P \rightarrow \{\ S.next = newlabel(\ );\ \}S\{\ label(S.next);\}$

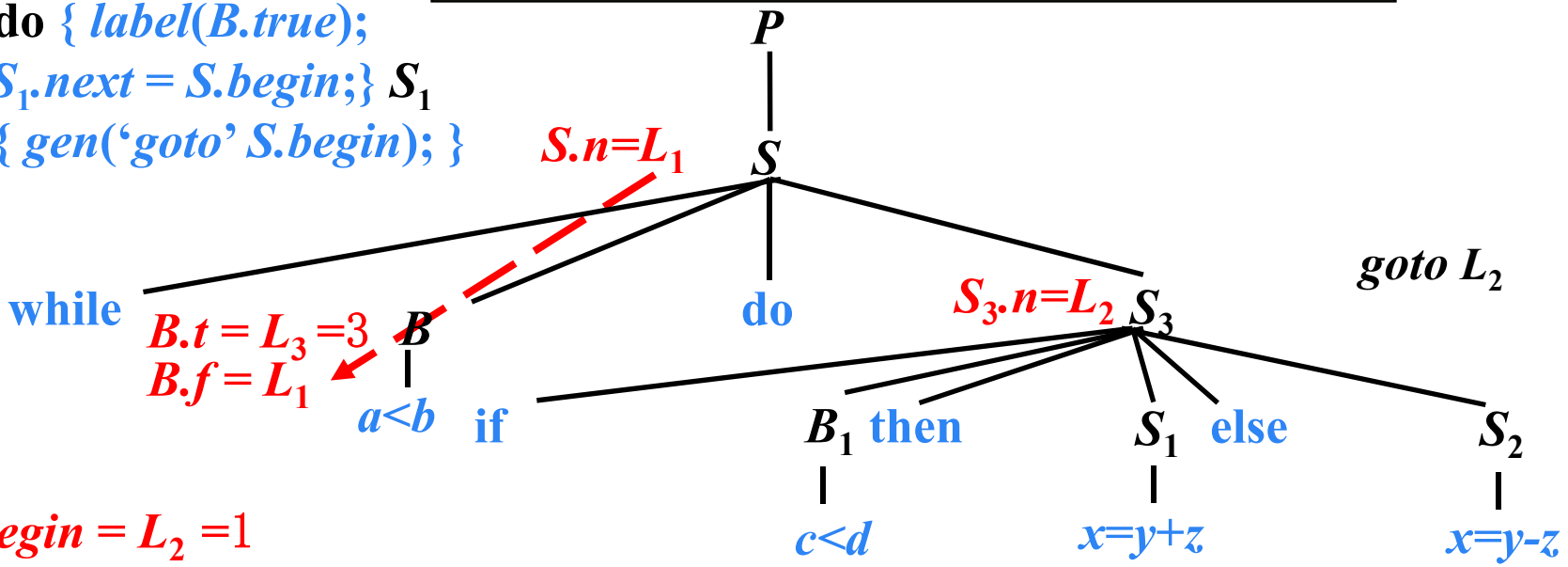


例

$S \rightarrow$ while {*S.begin = newlabel( );*
label(S.begin);
B.true = newlabel( );
B.false = S.next;} B
do { *label(B.true);*
S_1*.next = S.begin;*} S_1
{ *gen('goto' S.begin);* }

$B \rightarrow E_1$ relop E_2
{ *gen('if' E_1.addr relop E_2.addr 'goto' B.true);*
gen('goto ' B.false); }

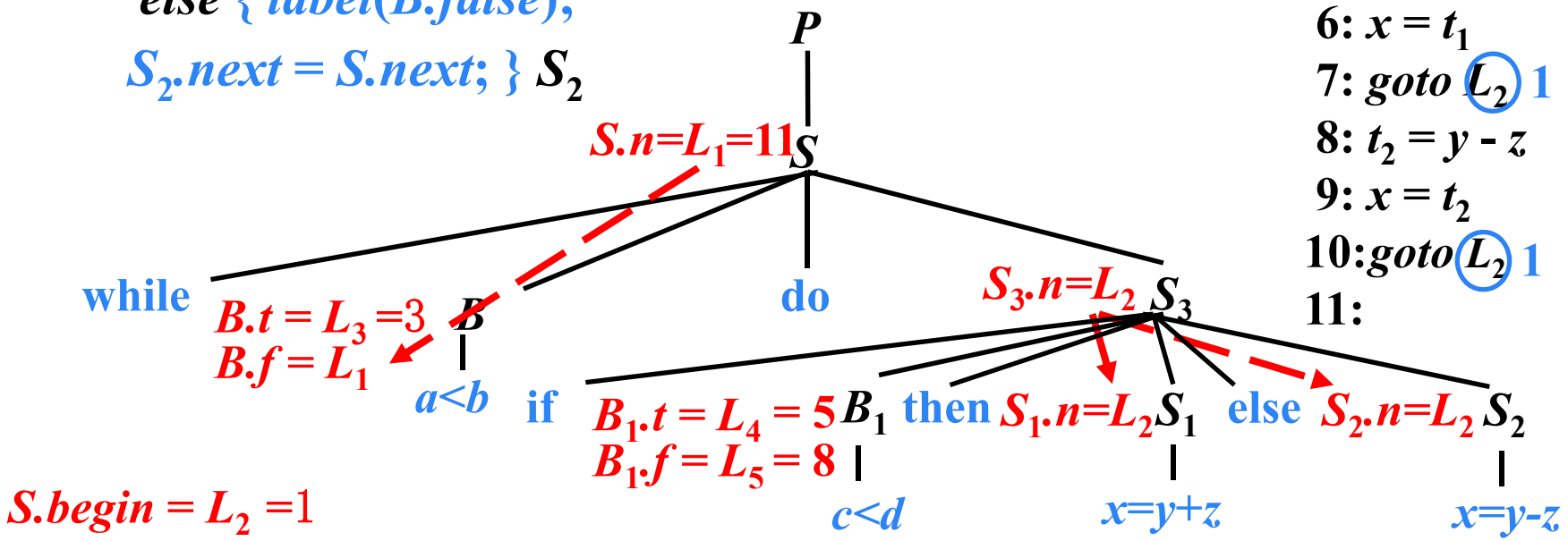
1: *if a < b goto* L_3
2: *goto* L_1



S.begin = L_2 =1

例

$S \rightarrow$ *if* { *B.true* = *newlabel*( ); *B.false* = *newlabel*( ); } *B*
then { *label*(*B.true*); S_1.*next* = *S.next*; } S_1
{ *gen*('goto' *S.next*); }
else { *label*(*B.false*);
S_2.*next* = *S.next*; } S_2



1: *if a < b goto* L_3 3
2: *goto* L_1 11
3: *if c < d goto* L_4 5
4: *goto* L_5 8
5: $t_1 = y + z$
6: $x = t_1$
7: *goto* L_2 1
8: $t_2 = y - z$
9: $x = t_2$
10: *goto* L_2 1
11:

语句“*while* a<b *do if* c<d *then* x=y+z *else* x=y-z”的三地址代码

1: *if* $a < b$ *goto* 3
2: *goto* 11
3: *if* $c < d$ *goto* 5
4: *goto* 8
5: $t_1 = y + z$
6: $x = t_1$
7: *goto* 1
8: $t_2 = y - z$
9: $x = t_2$
10: *goto* 1
11:

1: (*j*<, *a*, *b* , 3)
2: (*j* , - , - , 11)
3: (*j*<, *c*, *d* , 5)
4: (*j* , - , - , 8)
5: (+ , *y* , *z* , t_1)
6: (= , t_1, - , *x*)
7: (*j* , - , - , 1)
8: (- , *y*, *z* , t_2)
9: (= , t_2, - , *x*)
10: (*j* , - , - , 1)
11:

提纲

6.1 声明语句的翻译

6.2 赋值语句的翻译

6.3 控制语句的翻译

6.4 回填

6.5 switch语句的翻译

6.6 过程调用语句的翻译

6.4 回填 (*Backpatching*)

- 基本思想
  - 生成一个跳转指令时，暂时不指定该跳转指令的目标标号。这样的指令都被放入由跳转指令组成的列表中。同一个列表中的所有跳转指令具有相同的目标标号。等到能够确定正确的目标标号时，才去填充这些指令的目标标号

非终结符*B*的综合属性

- ***B.truelist***：指向一个包含跳转指令的列表，这些指令最终获得的目标标号就是当*B*为真时控制流应该转向的指令的标号
- ***B.falselist***：指向一个包含跳转指令的列表，这些指令最终获得的目标标号就是当*B*为假时控制流应该转向的指令的标号

函数

- ***makelist*(*i*)**
  - 创建一个只包含*i*的列表，*i*是跳转指令的标号，函数返回指向新创建的列表的指针
- ***merge*(p_1, p_2)**
  - 将 p_1 和 p_2 指向的列表进行合并，返回指向合并后的列表的指针
- ***backpatch*(*p*, *i*)**
  - 将 *i* 作为目标标号插入到 *p* 所指列表中的各指令中

布尔表达式的回填

- $B \rightarrow E_1$ relop E_2

{

B.truelist = makelist(nextquad);

B.falselist = makelist(nextquad+1);

gen('if' E_1.addr relop E_2.addr 'goto _');

gen('goto _');

}

布尔表达式的回填

- $B \rightarrow E_1$ relop E_2
- $B \rightarrow$ true

{

B.truelist = makelist(nextquad);

gen('goto _');

}

布尔表达式的回填

- $B \rightarrow E_1$ relop E_2
- $B \rightarrow$ true
- $B \rightarrow$ false

{

B.falselist = makelist(nextquad);

gen('goto _');

}

布尔表达式的回填

- $B \rightarrow E_1$ relop E_2
- $B \rightarrow$ true
- $B \rightarrow$ false
- $B \rightarrow (B_1)$

{

$B.truelist = B_1.truelist$;

$B.falselist = B_1.falselist$;

}

布尔表达式的回填

- $B \rightarrow E_1$ relop E_2
- $B \rightarrow$ true
- $B \rightarrow$ false
- $B \rightarrow (B_1)$
- $B \rightarrow$ not B_1

{

$B.truelist = B_1.falselist$;

$B.falselist = B_1.truelist$;

}

$B \rightarrow B_1$ or B_2

$B \rightarrow B_1$ or M B_2

{

$B.truelist = merge(B_1.truelist, B_2.truelist)$;

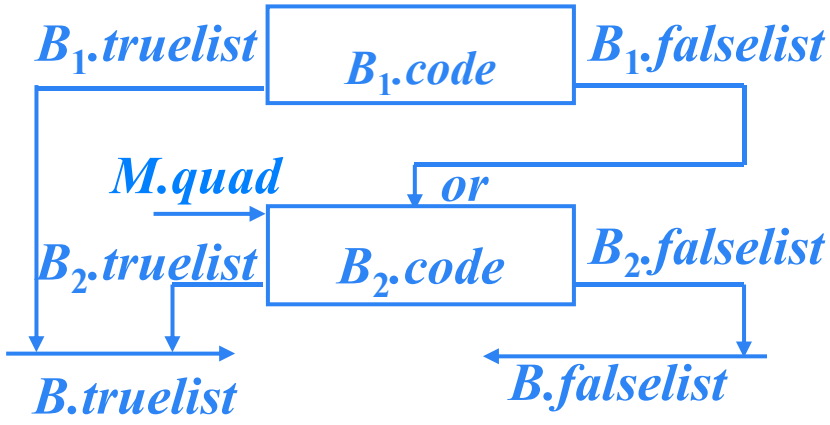
$B.falselist = B_2.falselist$;

$backpatch(B_1.falselist, M.quad)$;

}

$M \rightarrow \varepsilon$

{ $M.quad = nextquad$; }



$B \rightarrow B_1$ and B_2

$B \rightarrow B_1$ and M B_2

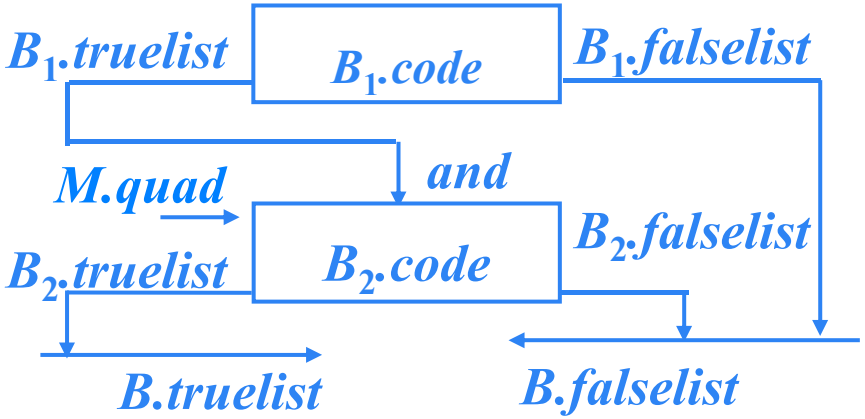
{

B.truelist = B_2.truelist;

B.falselist = merge(B_1.falselist, B_2.falselist);

backpatch(B_1.truelist, M.quad);

}

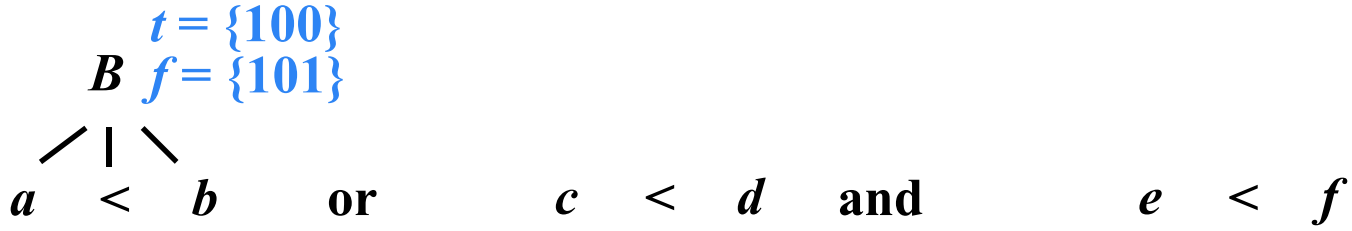


例

$B \rightarrow E_1$ relop E_2

{ *B.truelist = makelist(nextquad);*
B.falselist = makelist(nextquad+1);
gen('if' E_1.addr relop E_2.addr 'goto _');
gen('goto _');
}

100: *if a<b goto _*
101: *goto _*



例

$B \rightarrow B_1$ or M B_2

{

backpatch(B_1.*falselist*, *M.quad*);

B.*truelist* = *merge*(B_1.*truelist*, B_2.*truelist*);

B.*falselist* = B_2.*falselist* ;

}

$M \rightarrow \varepsilon$

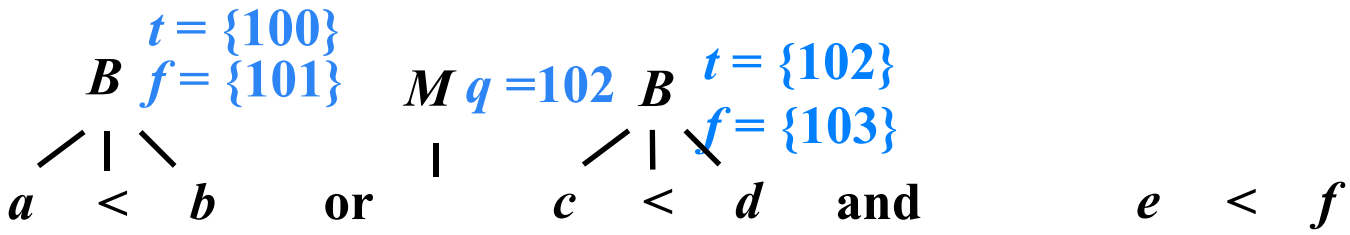
{ *M.quad* = *nextquad* ; }

100: *if a<b goto* _

101: *goto* _

102: *if c<d goto* _

103: *goto* _



例

$B \rightarrow B_1$ and M B_2

{

$B.truelist = B_2.truelist$;

$B.falselist = merge(B_1.falselist, B_2.falselist)$;

$backpatch(B_1.truelist, M.quad)$;

}

100: *if a<b goto* _

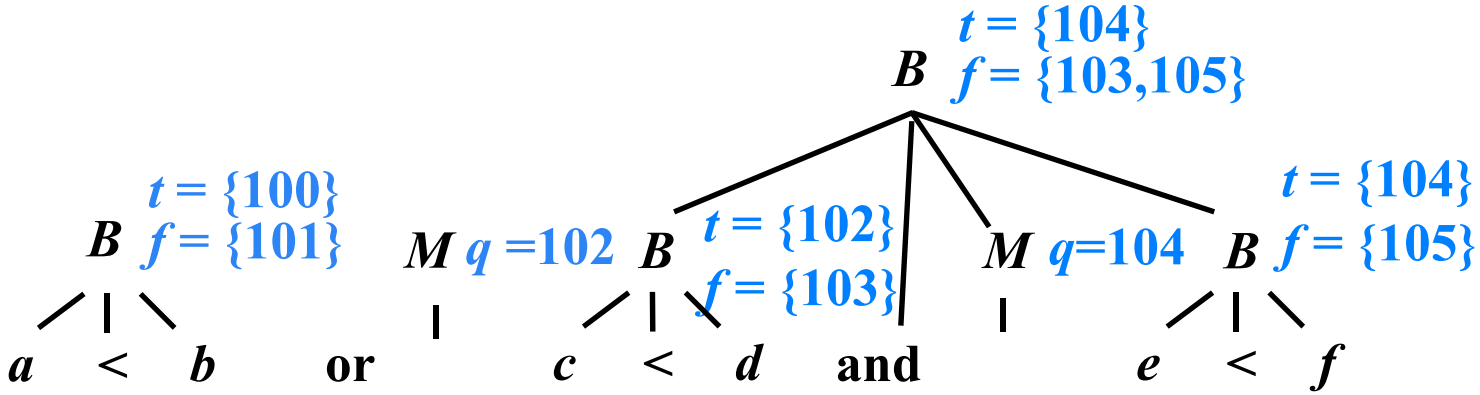
101: *goto* _

102: *if c<d goto* 104

103: *goto* _

104: *if e<f goto* _

105: *goto* _



例

$B \rightarrow B_1$ or M B_2

{

B.truelist = *merge*(B_1.*truelist*, B_2.*truelist*);

B.falselist = B_2.*falselist* ;

backpatch(B_1.*falselist*, *M.quad*);

}

100: *if a<b goto* _

101: *goto* 102

102: *if c<d goto* 104

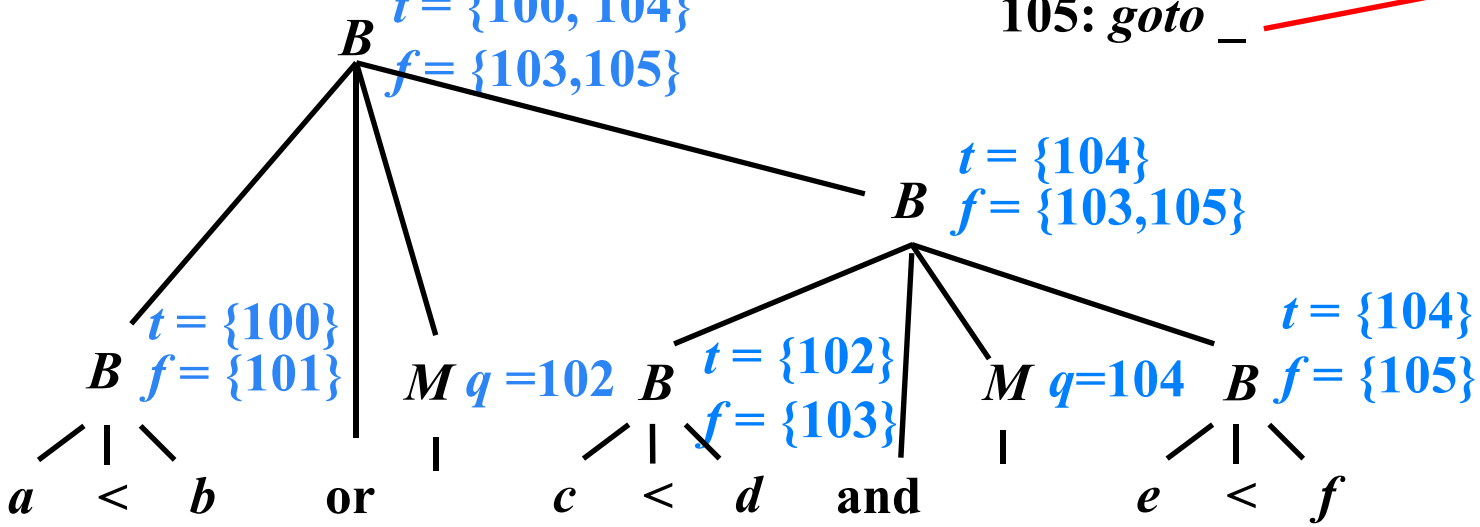
103: *goto* _

104: *if e<f goto* _

105: *goto* _

t

f



控制流语句的回填

- 文法
  - $S \rightarrow S_1 S_2$
  - $S \rightarrow \text{id} = E\ ; \mid L = E\ ;$
  - $S \rightarrow \text{if } B \text{ then } S_1$
    $\mid \text{if } B \text{ then } S_1 \text{ else } S_2$
    $\mid \text{while } B \text{ do } S_1$
- 综合属性
  - *S.next1ist*：指向一个包含跳转指令的列表，这些指令最终获得的目标标号就是按照运行顺序紧跟在S代码之后的指令的标号

$S \rightarrow$ if B then S_1

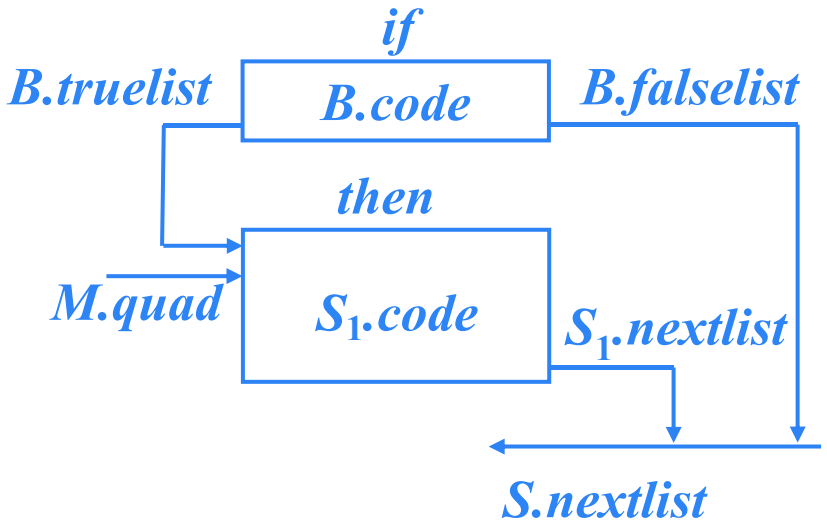
$S \rightarrow$ if B then M S_1

{

S.nextlist=merge(B.falselist, S_1.nextlist);

backpatch(B.truelist, M.quad);

}



$S \rightarrow$ if B then S_1 else S_2

$S \rightarrow$ if B then M_1 S_1 N else M_2 S_2

{

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

$backpatch(B.truelist, M_1.quad);$

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

}

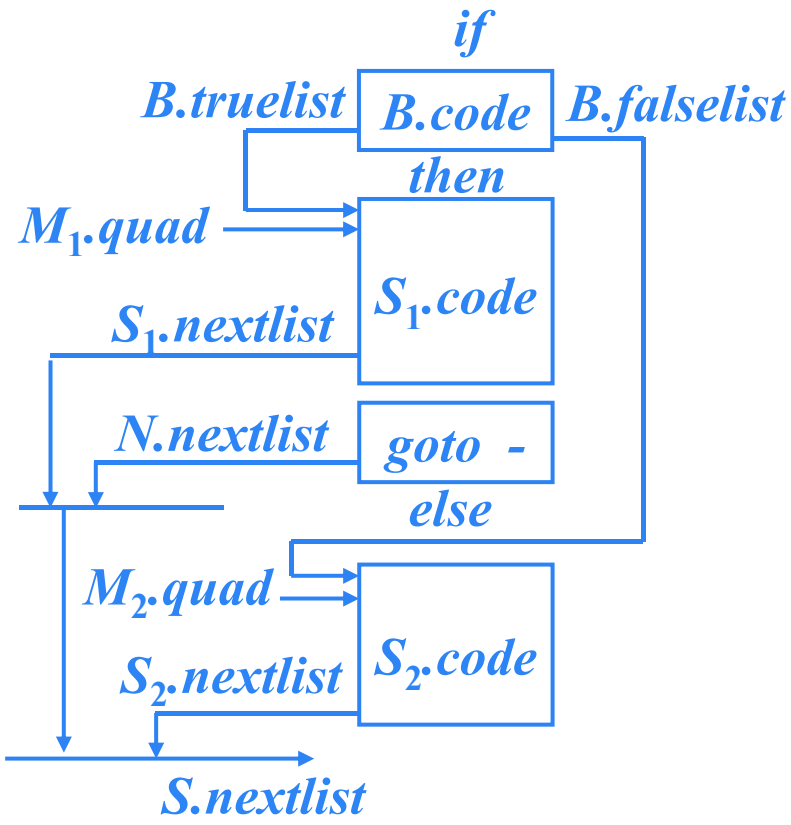
$N \rightarrow \varepsilon$

{

$N.nextlist = makelist(nextquad);$

$gen(\text{`}goto\ \_\text{'});$

}



S → while *B* do S_1

$S \rightarrow$ while M_1 B do M_2 S_1

{

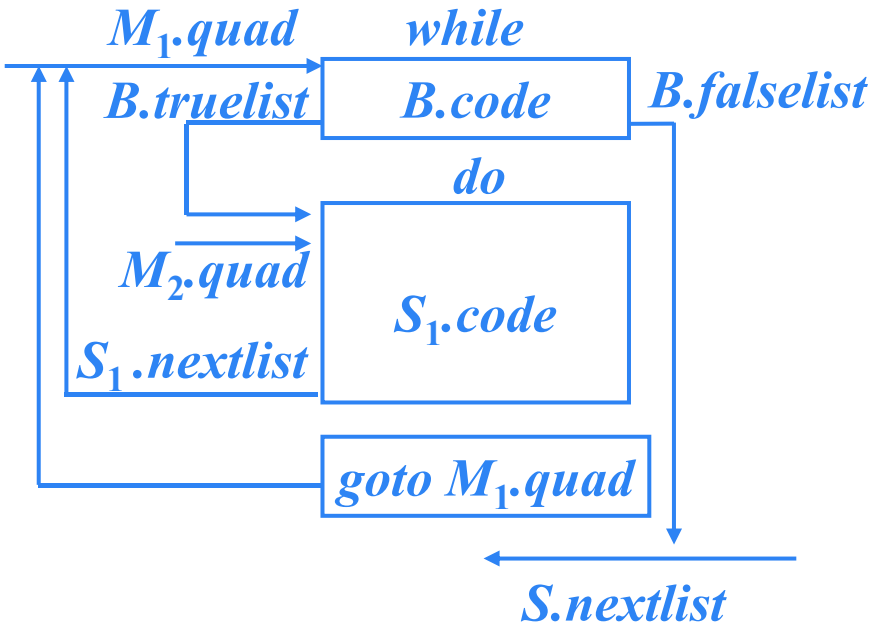
$S.nextlist = B.falselist$;

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

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

$gen(\text{'goto'}\ M_1.quad)$;

}



$S \rightarrow S_1\ S_2$

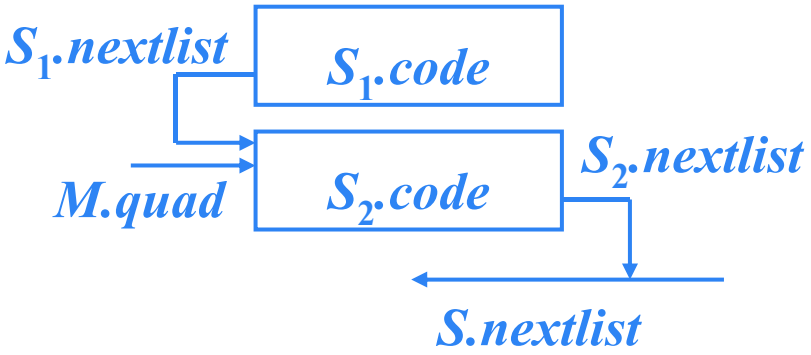
$S \rightarrow S_1\ M\ S_2$

{

$S.nextlist = S_2.nextlist$;

$backpatch(\ S_1.nextlist,\ M.quad\ )$;

}



$S \rightarrow \mathbf{id} = E\ ; \mid L = E\ ;$

$S \rightarrow \mathbf{id} = E\ ; \mid L = E\ ;$ { *S.nextlist = null*; }

例

```
while  a < b  do

    if  c < 5  then

            while x > y do z = x + 1;
    else
            x = y;
```

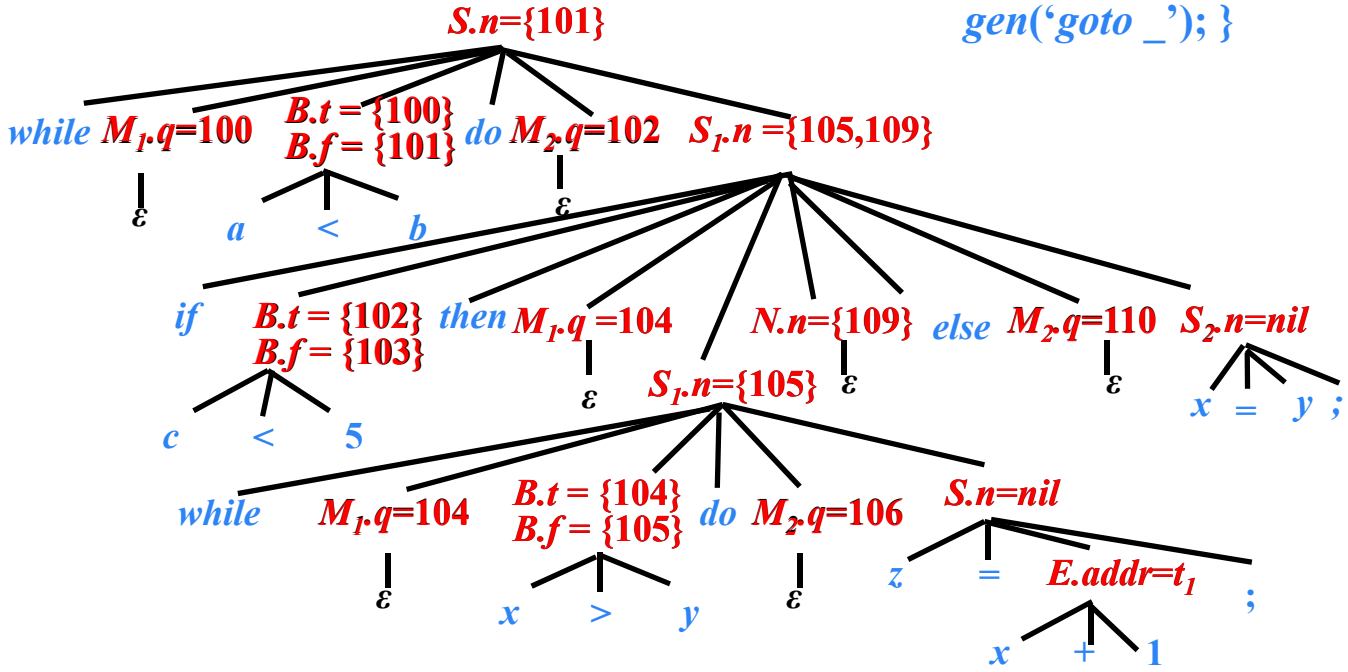
$S \rightarrow$ while M_1 B do M_2 S_1

{ $S.nextlist = B.falselist$;
$backpatch(S_1.nextlist, M_1.quad)$;
$backpatch(B.truelist, M_2.quad)$;
$gen(\text{'goto'}\ M_1.quad)$; }

$S \rightarrow$ if B then M_1 S_1 N else M_2 S_2

{ $S.nextlist = merge(merge(S_1.nextlist, N.nextlist), S_2.nextlist)$;
$backpatch(B.truelist, M_1.quad)$;
$backpatch(B.falselist, M_2.quad)$; }

$N \rightarrow \varepsilon$ { $N.nextlist = makelist(nextquad)$;
$gen(\text{'goto}\ \_\text{'})$; }



100: *if* $a < b$ *goto* 102
101: *goto* _
102: *if* $c < 5$ *goto* 104
103: *goto* 110
104: *if* $x > y$ *goto* 106
105: *goto* 100
106: $t_1 = x + 1$
107: $z = t_1$
108: *goto* 104
109: *goto* 100
110: $x = y$
111: *goto* 100
112:

语句“*while a<b do if c<5 then while x>y do z=x+1; else x=y;*”的注释分析树

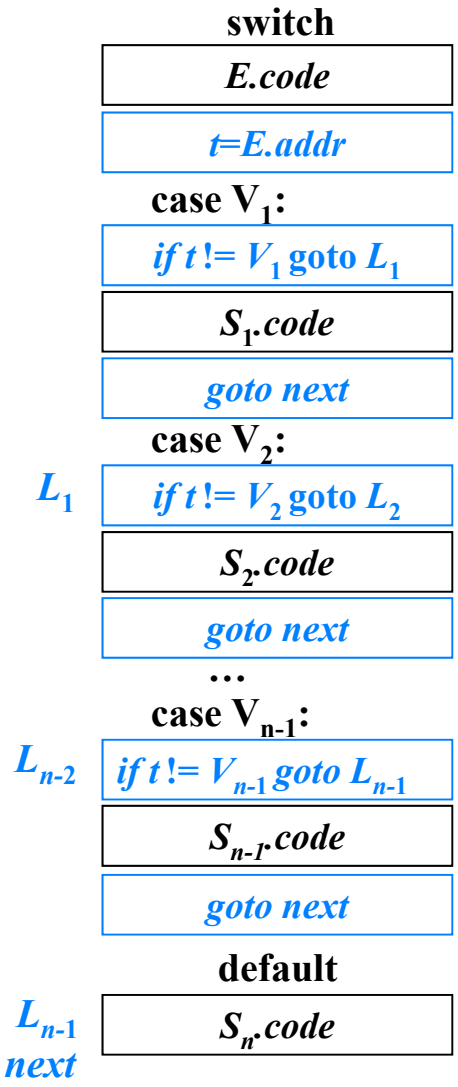
while a<b do if c<5 then while x>y do *z=x+1*; *else x=y*;

100: *if a < b goto* 102	100: (*j<*, *a* ,*b* , 102)
101: *goto* _	101: (*j* , - , - , -)
102: *if c < 5 goto* 104	102: (*j<*, *c* , 5 , 104)
103: *goto* 110	103: (*j* , - , - , 110)
104: *if x > y goto* 106	104: (*j>*, *x* , *y* , 106)
105: *goto* 100	105: (*j* , - , - , 100)
106: $t_1 = x + 1$	106: (+ , *x* , 1 , t_1)
107: $z = t_1$	107: (= , t_1 , - , *z*)
108: *goto* 104	108: (*j* , - , - , 104)
109: *goto* 100	109: (*j* , - , - , 100)
110: *x = y*	110: (= , *y* , - , *x*)
111: *goto* 100	111: (*j* , - , - , 100)
112:	112:

提纲

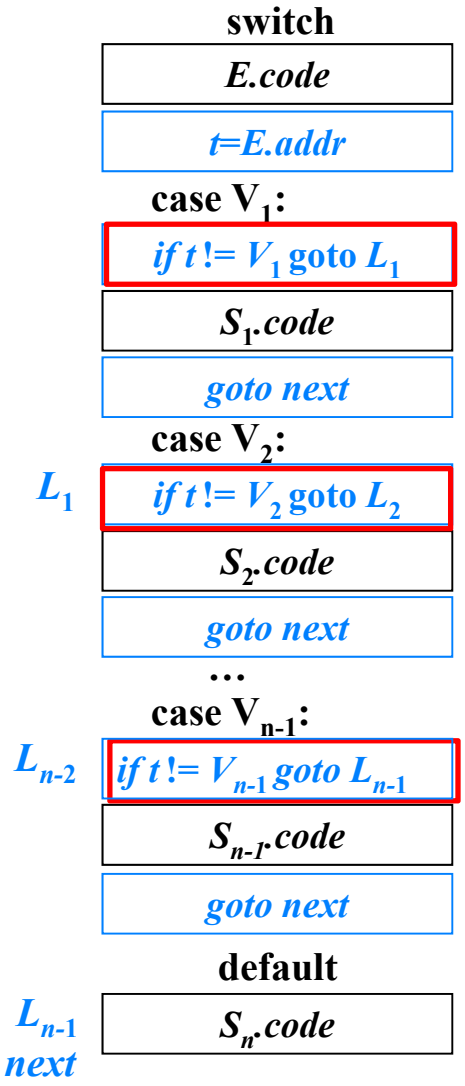
6.5 *switch*语句的翻译

switch *E*
  begin
    case V_1: S_1
    case V_2: S_2
    . . .
    case V_{n-1}: S_{n-1}
    default: S_n
  end



6.5 *switch*语句的翻译

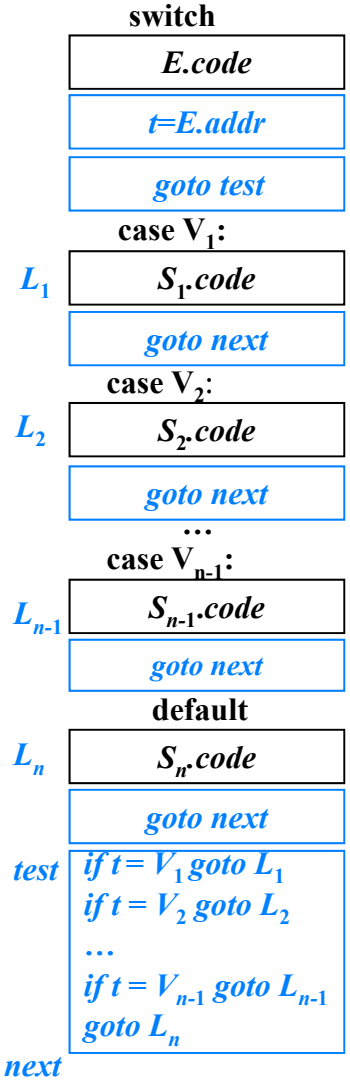
switch *E* { *t* = *newtemp*(); *gen*(*t* '=' *E.addr*); }
case V_1:{ L_1= *newlabel*();
gen('*if*' *t* '!=' V_1 '*goto*' L_1); }
S_1 { *next* = *newlabel*(); *gen*('*goto*' *next*); }
case V_2:{ *label*(L_1); L_2=*newlabel*();
gen('*if*' *t* '!= ' V_2 '*goto*' L_2); }
S_2 { *gen*('*goto*' *next*); }
. . .
case V_{n-1}:{ *label*(L_{n-2}); L_{n-1}=*newlabel*();
gen('*if*' *t* '!=' V_{n-1} '*goto*' L_{n-1});}
S_{n-1} { *gen*('*goto*' *next*);}
default:{ *label*(L_{n-1});}
S_n {*label*(*next*);}



*switch*语句的另一种翻译

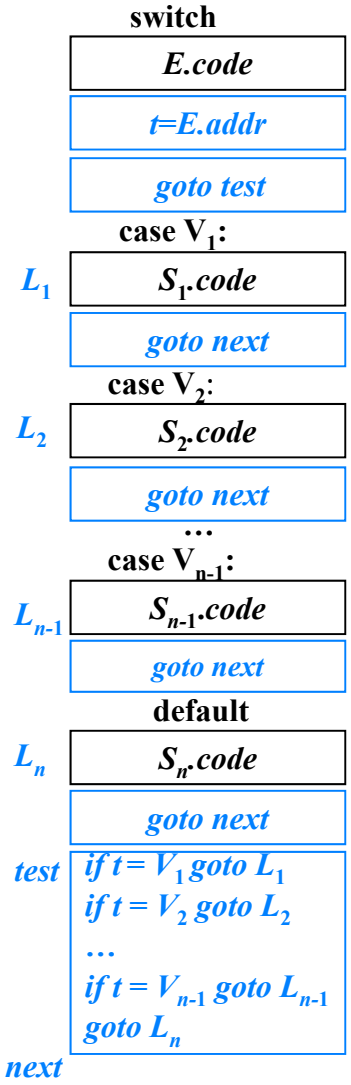
switch *E*
  begin
    case V_1: S_1
    case V_2: S_2
    . . .
    case V_{n-1}: S_{n-1}
    default: S_n
  end

在代码生成阶段，根据分支的个数以及这些值是否在一个较小的范围内，这种条件跳转指令序列可以被翻译成最高效的*n*路分支



*switch*语句的另一种翻译

switch *E* { *t* = *newtemp*(); *gen*(*t* '=' *E.addr*);
test = *newlabel*(); *gen*('goto' *test*); }
case V_1 : { L_1 = *newlabel*(); *label*(L_1); *map*(V_1, L_1); }
S_1 { *next* = *newlabel*(); *gen*('goto' *next*); }
case V_2 : { L_2 = *newlabel*(); *label*(L_2); *map*(V_2, L_2); }
S_2 { *gen*('goto' *next*); }
. . .
case V_{n-1}:{ L_{n-1} = *newlabel*(); *label*(L_{n-1}); *map*(V_{n-1}, L_{n-1}); }
S_{n-1} { *gen*('goto' *next*); }
default : { L_n = *newlabel*(); *label*(L_n); }
S_n { *gen*('goto' *next*);
label(*test*);
gen('if' *t* '=' V_1 'goto' L_1);
gen('if' *t* '=' V_2 'goto' L_2);
. . .
gen('if' *t* '=' V_{n-1} 'goto' L_{n-1});
gen('goto' L_n);
label(*next*);
}



增加一种*case*指令

```
test :  if t = V1 goto L1
        if t = V2 goto L2
        . . .
        if t = Vn-1 goto Ln-1
        goto Ln
next :
```

```
test :  case t  V1    L1
        case t  V2    L2
           . . .
        case t  Vn-1  Ln-1
        case t  t     Ln
next :
```

指令 *case t* V_i L_i 和 *if* $t = V_i$ *goto* L_i 的含义相同，
但是*case*指令更加容易被最终的代码生成器探测到，
从而对这些指令进行特殊处理

提纲

6.6 过程调用的翻译

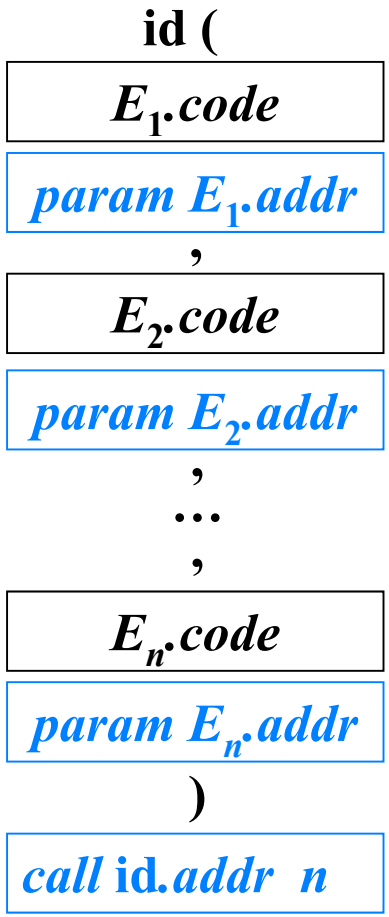
- 文法
  - $S \rightarrow$ call id ($Elist$)

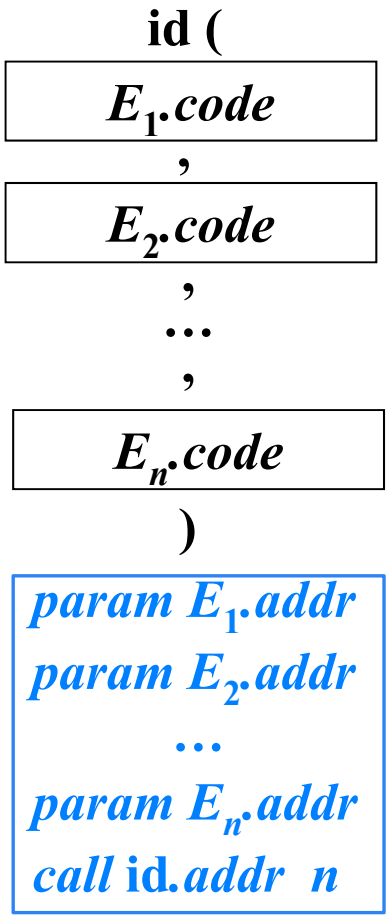
    $Elist \rightarrow Elist, E$

    $Elist \rightarrow E$

过程调用语句的代码结构

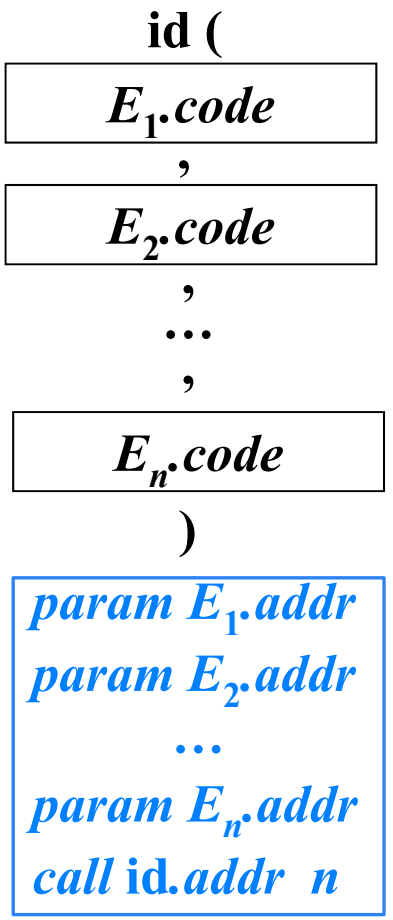
id(E_1, E_2, … , E_n)





过程调用语句的代码结构

id(E_1, E_2, … , E_n)



需要一个队列q存放E_1.addr 、E_2.addr、…、E_n.addr，以生成

过程调用语句的*SDD*

- $S \rightarrow$ call id (*Elist*)

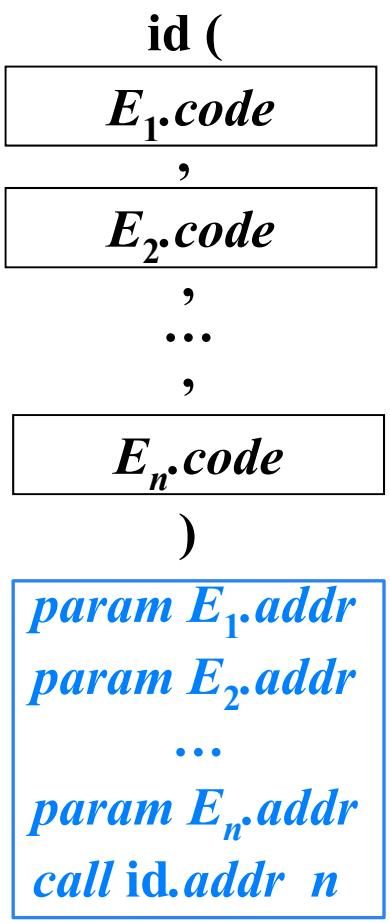
```
{   n=0;
    for q中的每个t do
    {   gen('param' t );
        n = n+1;
    }
    gen('call' id.addr ',' n);
}
```

- *Elist* → *E*

```
{   将q初始化为只包含E.addr; }
```

- $Elist \rightarrow Elist_1, E$

```
{   将E.addr添加到q的队尾;   }
```



例：翻译以下语句 $f(b*c-1, x+y, x, y)$

$t_1 = b*c$

$t_2 = t_1 - 1$

$t_3 = x+y$

param t_2

param t_3

param x

param y

call f, 4

语义分析中的错误检测

- 变量或过程未经声明就使用 （赋值/过程调用语句翻译）

```
S → id = E;
{   p = lookup(id.lexeme); if p == nil then error ;
    gen( p '=' E.addr ); }
E → id
{   E.addr = lookup(id.lexeme); if E.addr == nil then error ; }
```

```
S → call id ( Elist )
{    n=0;
     for q中的每个t  do
     {  gen('param' t );    n = n+1;   }
     gen('call' id.addr ',' n);
}
```

语义分析中的错误检测

- 变量或过程未经声明就使用 （赋值/过程调用语句翻译）
- 变量或过程名重复声明 （声明语句翻译）

$D \rightarrow T\ \mathrm{id};\{\ enter(\ \mathrm{id}.lexeme,\ T.type,\ offset\ );\ \ offset = offset + T.width;\ \}D$

语义分析中的错误检测

- 变量或过程未经声明就使用 （赋值/过程调用语句翻译）
- 变量或过程名重复声明 （声明语句翻译）
- 运算分量类型不匹配 （赋值语句翻译）

$E \rightarrow E_1 + E_2$

```
{   E.addr = newtemp()
    if E1.type == integer and E2.type == integer then
            { gen( E.addr ‘=’ E1.addr ‘int+’ E2.addr ); E.type = integer；  }
    else if E1.type == integer and E2.type == real then
            {u = newtemp();
            gen( u ‘=’ ‘inttoreal’ E1.addr );
            gen( E.addr ‘=’ u ‘real+’ E2.addr );
            E.type = real；  }
    . . .
}
```

语义分析中的错误检测

- 变量或过程未经声明就使用（赋值/过程调用语句翻译）
- 变量或过程名重复声明（声明语句翻译）
- 运算分量类型不匹配（赋值语句翻译）
- 操作符与操作数之间的类型不匹配
  - 数组下标不是整数（赋值语句翻译）

$$L \rightarrow \mathrm{id}\,[E] \mid L_1\,[E]$$

语义分析中的错误检测

- 变量或过程未经声明就使用（赋值/过程调用语句翻译）
- 变量或过程名重复声明（声明语句翻译）
- 运算分量类型不匹配（赋值语句翻译）
- 操作符与操作数之间的类型不匹配
  - 数组下标不是整数（赋值语句翻译）
  - 对非数组变量使用数组访问操作符（赋值语句翻译）

$$L \rightarrow \mathrm{id}\,[E] \mid L_1\,[E]$$

语义分析中的错误检测

- 变量或过程未经声明就使用（赋值/过程调用语句翻译）
- 变量或过程名重复声明（声明语句翻译）
- 运算分量类型不匹配（赋值语句翻译）
- 操作符与操作数之间的类型不匹配
  - 数组下标不是整数（赋值语句翻译）
  - 对非数组变量使用数组访问操作符（赋值语句翻译）
  - 对非过程名使用过程调用操作符（过程调用翻译）

$S \rightarrow$ **call** id (*Elist*)

语义分析中的错误检测

- 变量或过程未经声明就使用（赋值/过程调用语句翻译）
- 变量或过程名重复声明（声明语句翻译）
- 运算分量类型不匹配（赋值语句翻译）
- 操作符与操作数之间的类型不匹配
  - 数组下标不是整数（赋值语句翻译）
  - 对非数组变量使用数组访问操作符（赋值语句翻译）
  - 对非过程名使用过程调用操作符（过程调用翻译）
  - 过程调用的参数类型或数目不匹配

$S \rightarrow$ call id (*Elist*)
{ *n*=0;
*for q*中的每个*t do*
{ *gen*(‘*param*’ *t*); *n* = *n*+1; }
gen(‘*call*’ id.*addr* ‘,’ *n*); }

语义分析中的错误检测

- 变量或过程未经声明就使用（赋值/过程调用语句翻译）
- 变量或过程名重复声明（声明语句翻译）
- 运算分量类型不匹配（赋值语句翻译）
- 操作符与操作数之间的类型不匹配
  - 数组下标不是整数（赋值语句翻译）
  - 对非数组变量使用数组访问操作符（赋值语句翻译）
  - 对非过程名使用过程调用操作符（过程调用翻译）
  - 过程调用的参数类型或数目不匹配
  - 函数返回类型有误

本章小结

- 声明语句的翻译
- 赋值语句的翻译
  - 简单赋值语句的翻译
  - 数组引用的翻译
- 控制语句的翻译
- 回填
- switch语句的翻译
- 过程调用语句的翻译

结束