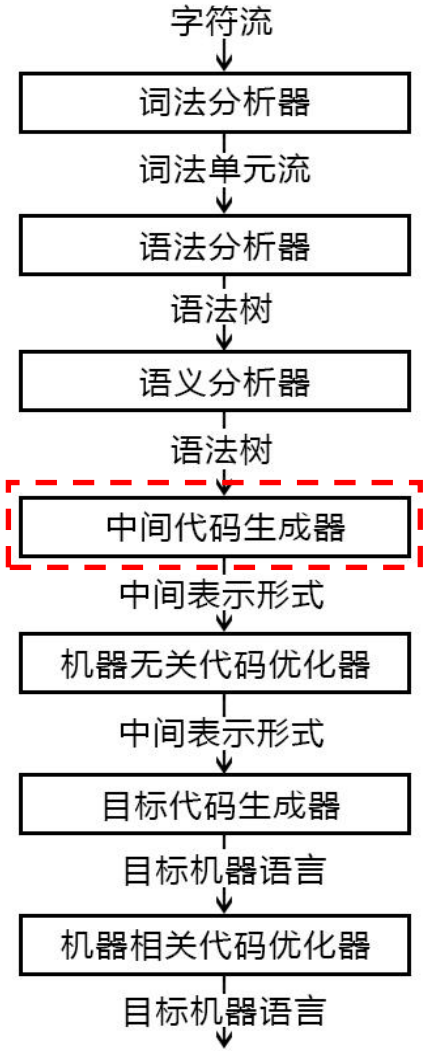
编译原理

第六章
中间代码生成

哈尔滨工业大学　陈鄞 单丽莉

编译器的结构

- 中间代码的特点
  - 简单规范
  - 易于优化与转换
  - 与机器无关
  - 易于移植



常用的三地址指令

序号	指令类型	指令形式
1	赋值指令	*x* = *y* op *z* *x* = op *y*
2	复制指令	*x* = *y*
3	条件跳转	if *x* relop *y* goto *n*
4	非条件跳转	goto *n*
5	参数传递	param *x*
6	过程调用 函数调用	call *p*, *n* *y* =call *p*, *n*
7	过程返回	return *x*
8	数组引用	*x* = *y*[*i*]
9	数组赋值	*x*[*i*] = *y*
10	地址及 指针操作	*x* =& *y* *x* =* *y*

三地址指令：一个运算符，三个地址；

三地址: (最多)两个运算分量地址，

一个结果分量地址。

地址可以具有如下形式之一

- 源程序中的名字 (*name*)
- 常量 (*constant*)
- 编译器生成的临时变量(*temporary*)

- 指令中红色的部分表示指令的操作符
- x[i]，其中的i是与首地址的偏移地址

三地址指令的四元式表示

- $x = y\ op\ z$ (op , y , z , x)
- $x = op\ y$ (op , y , _ , x)
- $x = y$ (= , y , _ , x)
- if x relop y goto n (relop , x , y , n)
- goto n (goto , _ , _ , n)
- param x (param, _ , _ , x)

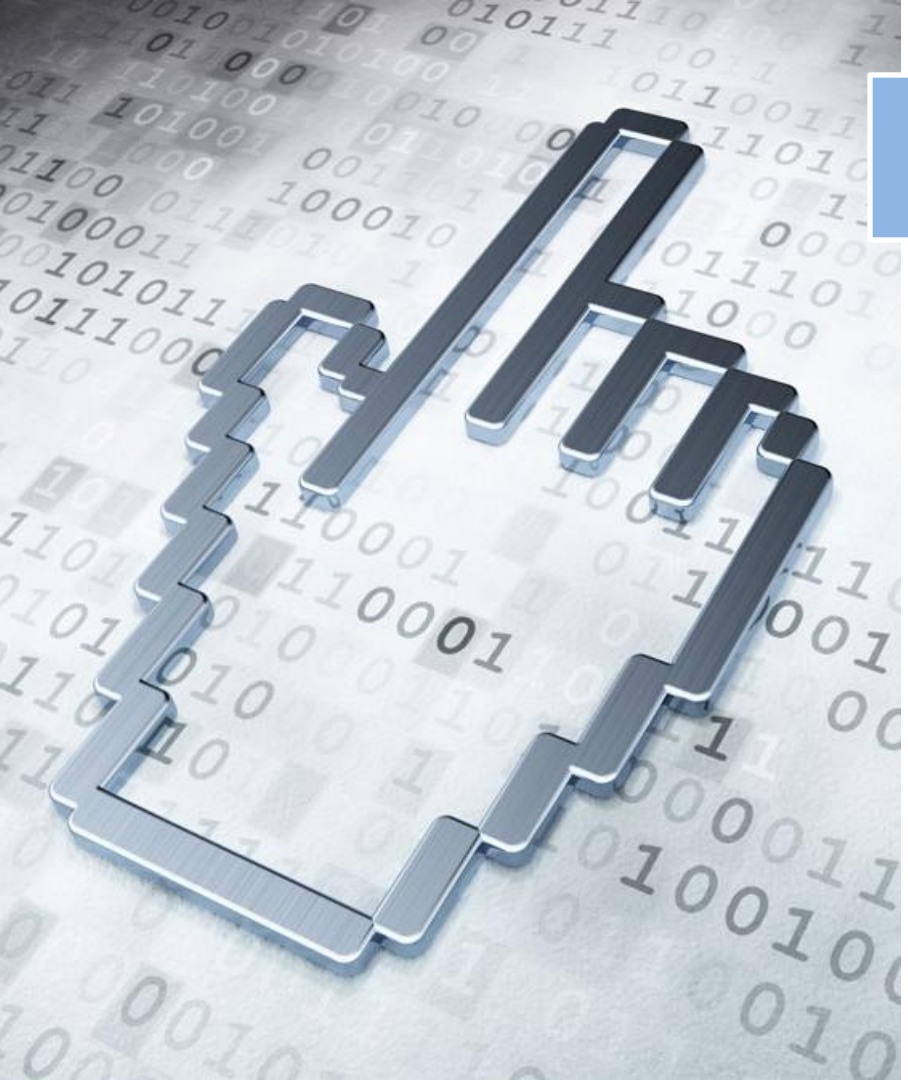
  y =call p, n (call , p , n , y)

  return x (return, _ , _ , x)
- $x = y[i]$ (=[] , y , i , x)

  $x[i] = y$ ([]= , y , x , i)
- $x = \&y$ (=& , y , _ , x)

三地址指令序列唯一确定了运算完成的顺序

- 第一个分量是操作符
- 第二三个分量是操作数
- 第四个分量是结果地址



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
- 类型的重要作用
  - 类型检查：验证程序运行时的行为是否遵守语言的类型规定。语义分析任务多数都与类型检查相关。
  - 辅助翻译：计算数组引用的地址、加入显式的类型转换、选择正确版本的算术运算符都要用到类型信息。

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 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$T_1 \quad T_2 \quad \ldots \quad T_n \rightarrow R$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 T_1) (N_2 T_2) … (N_n 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*)

类型等价

- 许多类型检查都需要判断两个类型表达式是否等价：
- 结构等价：当且仅当下列条件之一成立时，称两个类型T_1和T_2是结构等价的：
  - T_1和T_2是相同的基本类型；
  - T_1和T_2是将同一类型构造符应用于结构等价的类型上形成的；
  - T_1是表示T_2的类型名。
- 名字等价：两个类型表达式名字等价当且仅当上述前两条之一成立。 关注类型名称的一致性。

类型等价

例：有一段程序声明

```
typedef  cell*  link;
link  next ,  last;
cell * p , q;
```

其中声明变量的类型表达式

变量	类型表达式
next	link
last	link
p	pointer(cell)
q	pointer(cell)

按名字等价判断：变量next和last类型相同，
变量p，q类型相同。

按结构等价判断：所有4个变量类型都相同。

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①收集标识符的类型等属性信息，②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，可以根据类型的宽度为每一个名字分配一个相对地址（相对于数据区域开始位置的偏移量）
- 名字的类型和相对地址信息保存在符号表记录中

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

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

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

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

⑨ $C \rightarrow [\text{num}]C_1\ \{ C.type = array(num.val,$

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

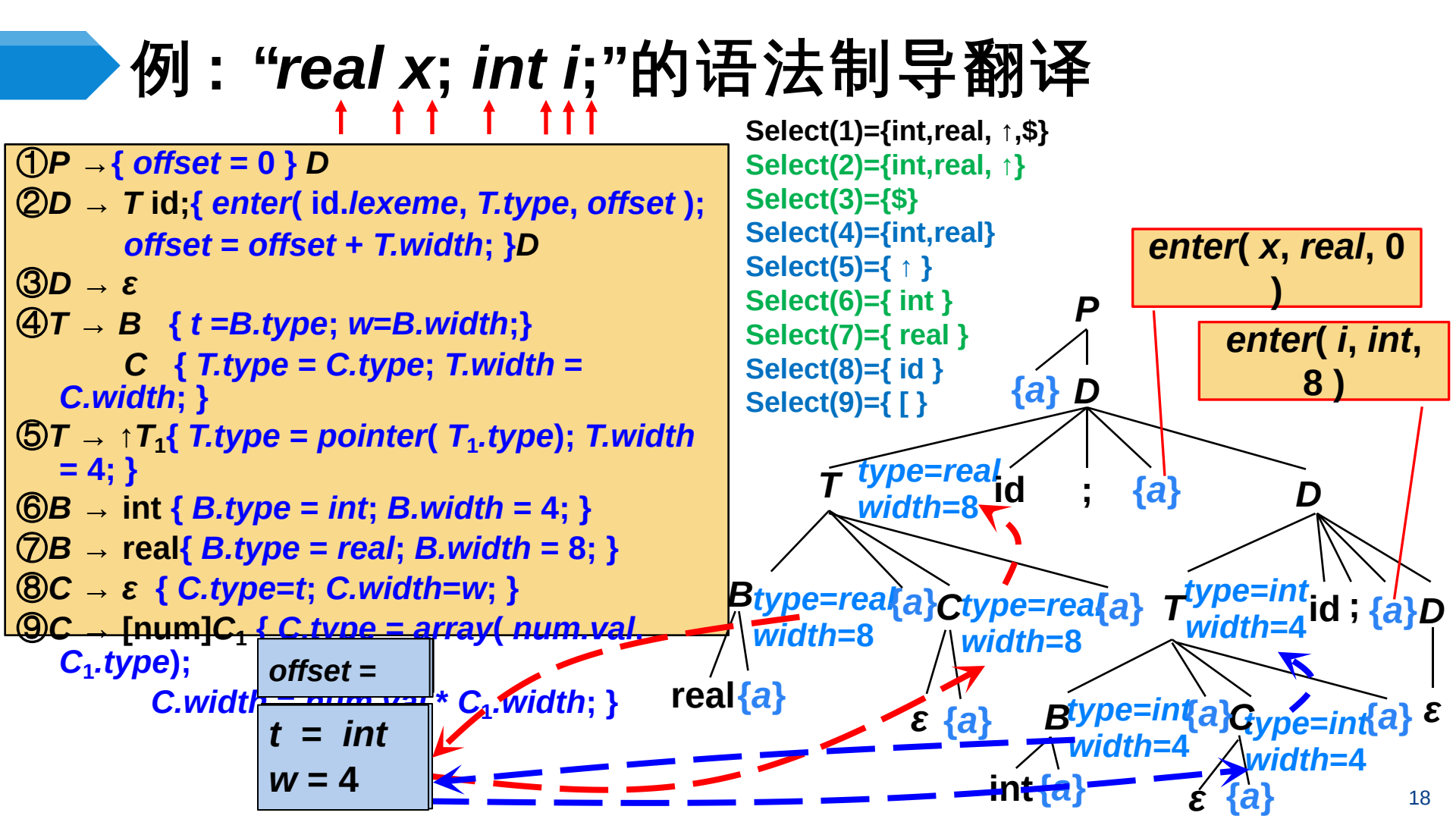
变量	作用
offset	下一个可用的相对地址
t, w	将类型和宽度信息从语法分析树中的*B*结点传递到对应于产生式$C \rightarrow \varepsilon$的结点

例：“*real x; int i;*”的语法制导翻译

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

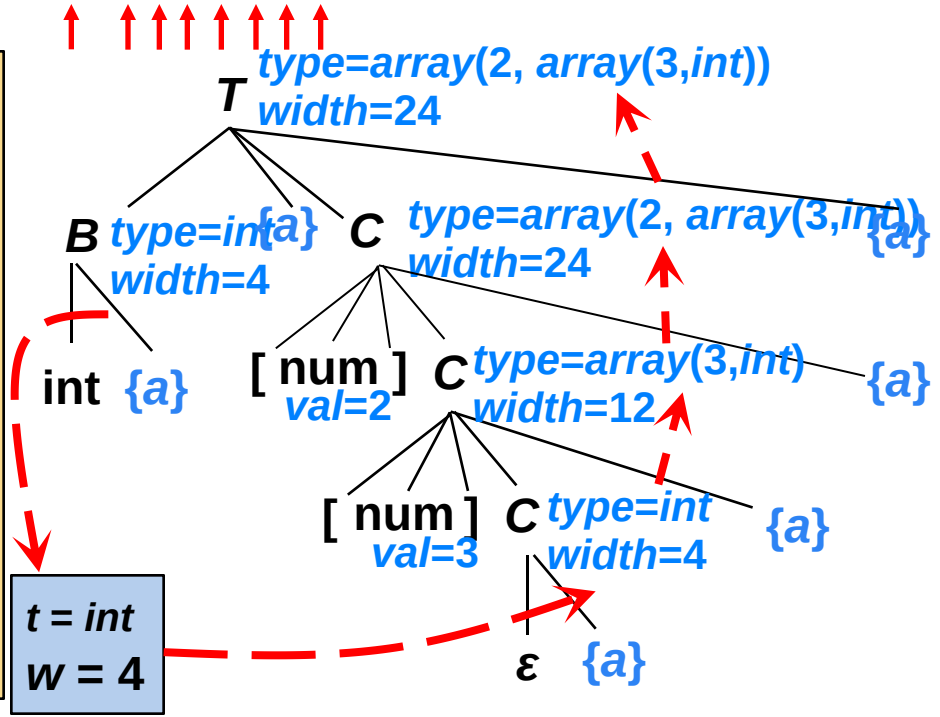
Select(1)={int,real, ↑,$}
Select(2)={int,real, ↑}
Select(3)={$}
Select(4)={int,real}
Select(5)={ ↑ }
Select(6)={ int }
Select(7)={ real }
Select(8)={ id }
Select(9)={ [}

enter(*x*, *real*, 0)
enter(*i*, *int*, 8)

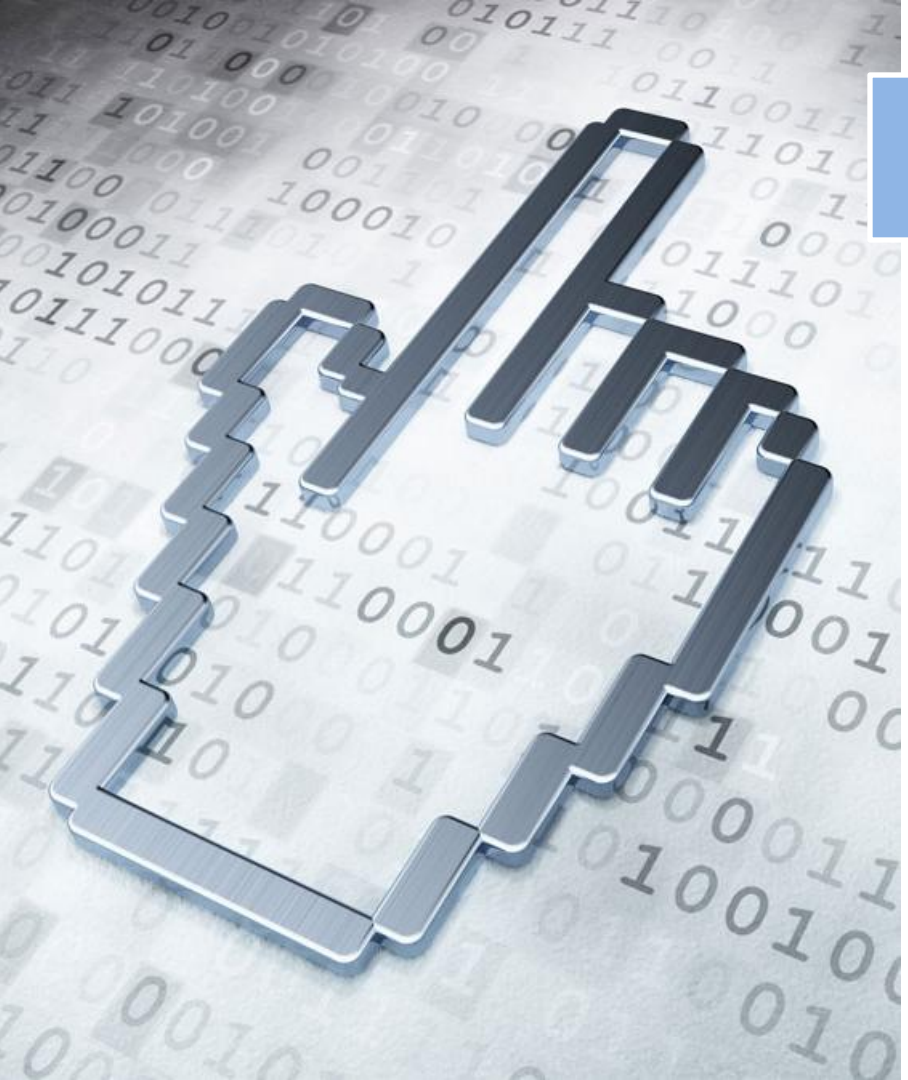


例：数组类型表达式“*int*[2][3]”的语法制导翻译

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



数组：只计算和保存整个数组的起始相对地址。



本章内容

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 id = E ;

② E $E_1 + E_2$

③ E $E_1 * E_2$

④ E E_1

⑤ E (E_1)

⑥ E id

- 赋值语句翻译的主要任务
  - 生成对表达式求值的中间代码

- 例 $x = (a + b) * c$;
- 三地址码

$t_1 = a + b$

$t_2 = t_1 * c$

$x = t_2$

赋值语句的SDT

lookup(*id.lexeme*)：查询符号表返回 *id* 对应的地址

gen(*code*)：生成三地址指令 *code*

newtemp( )：生成一个新的临时变量 *t*，并返回*t*的地址

```
S   id = E ; { p = lookup(id.lexeme); if p == nil then error ;
               S.code = E.code ||
               gen( p '=' E.addr );  }
E   E1 + E2 { E.addr = newtemp( );
              E.code = E1.code || E2.code
              gen(E.addr '=' E1.addr '+' E2.addr);  }
E   E1 * E2 { E.addr = newtemp( );
              E.code = E1.code || E2.code ||
              gen(E.addr '=' E1.addr '*' E2.addr); }
E     E1          { E.addr = newtemp( );
            E.code = E1.code||
            gen(E.addr '=' 'uminus' E1.addr);  }
E   (E1) { E.addr = E1.addr;
           E.code= E1.code;   }
E   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 id = E ; { *p = lookup*(id.*lexeme*); *if p == nil then error* ;
S.code = E.code ||
gen(p '=' E.addr };

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

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

E E_1 { *E.addr = newtemp( );*
E.code = E_1.code||
gen(E.addr '=' 'uminus' E_1.addr);

E (E_1) { *E.addr = E_1.addr;*
E.code= E_1.code;

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

不再使用*code*属性，*gen( )*负责构造出一个新的三地址指令，再将它添加到至今为止已生成的指令序列之后

- 综合属性code的值总是将产生式右部符号对应的三地址码按生成顺序连接以后，在后面追加新的三地址指令

例

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

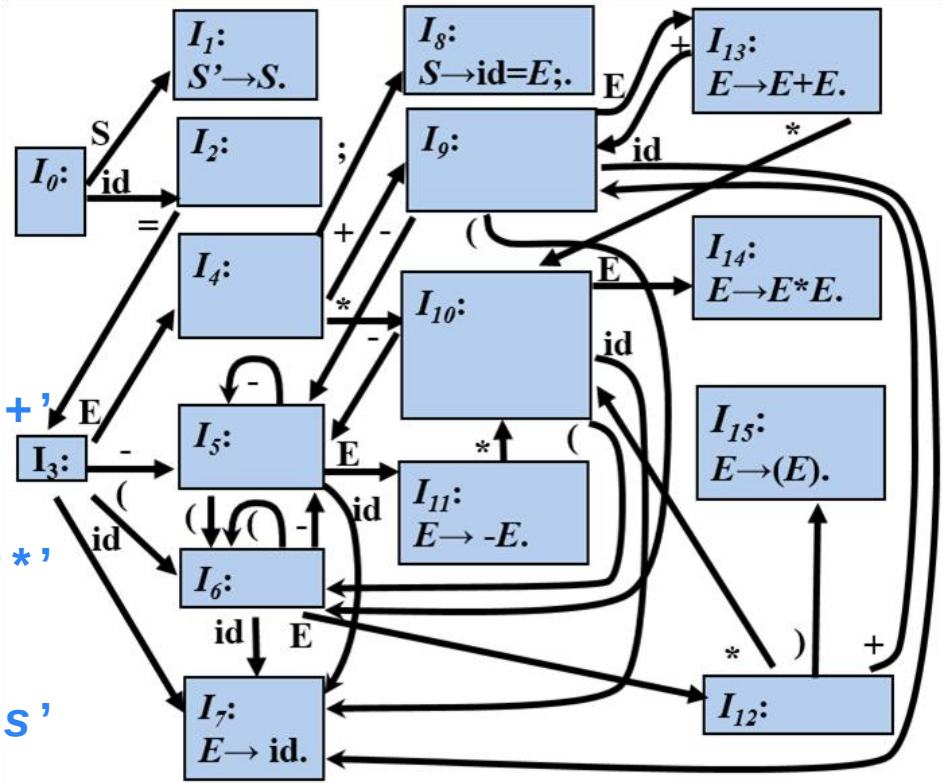
②*E*　$E_1 + E_2$ { *E.addr = newtemp( );*
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

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

⑤*E*　(*E*$_1$) { *E.addr = E_1.addr ; }*

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



例：*x* = (*a* + *b*) * *c* ;

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

例

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

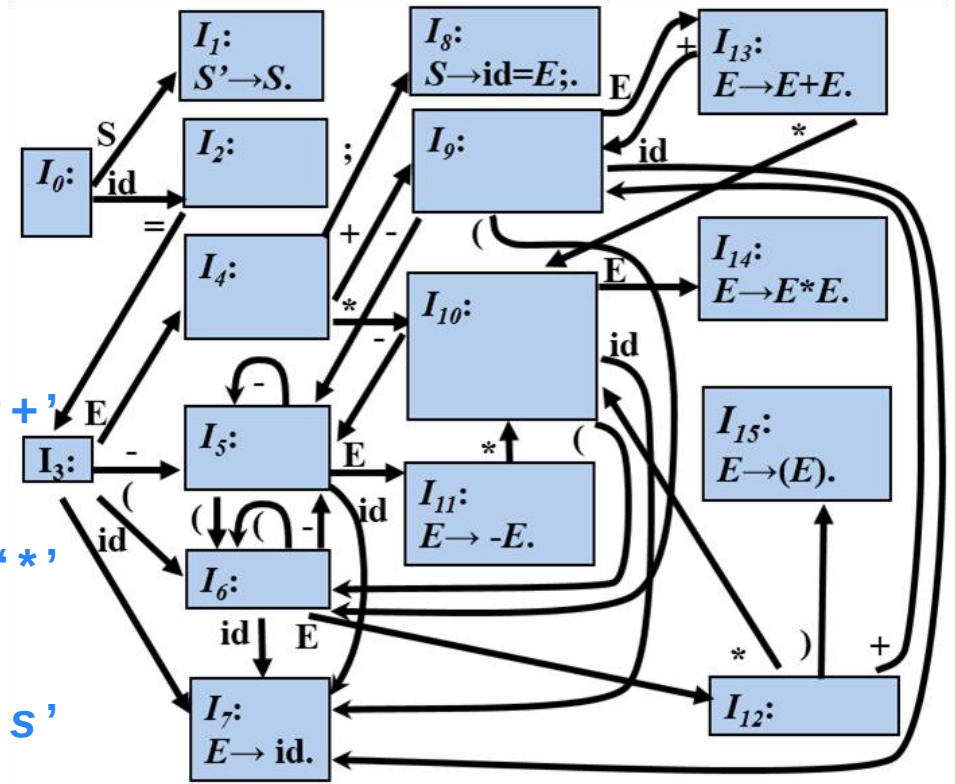
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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

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

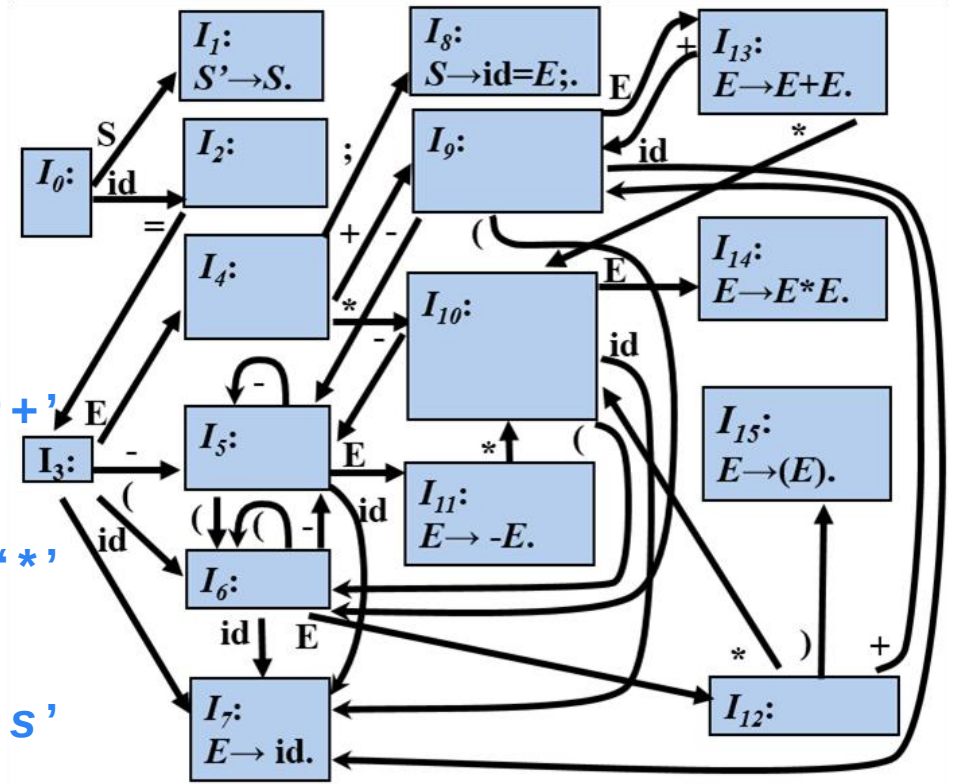
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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

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

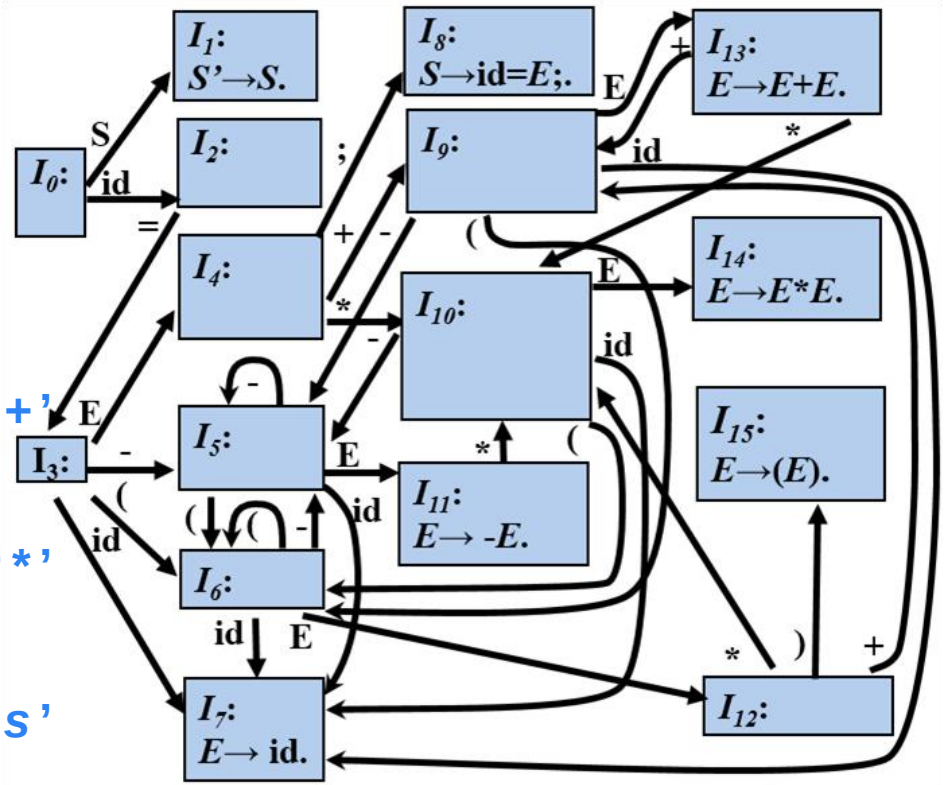
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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

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

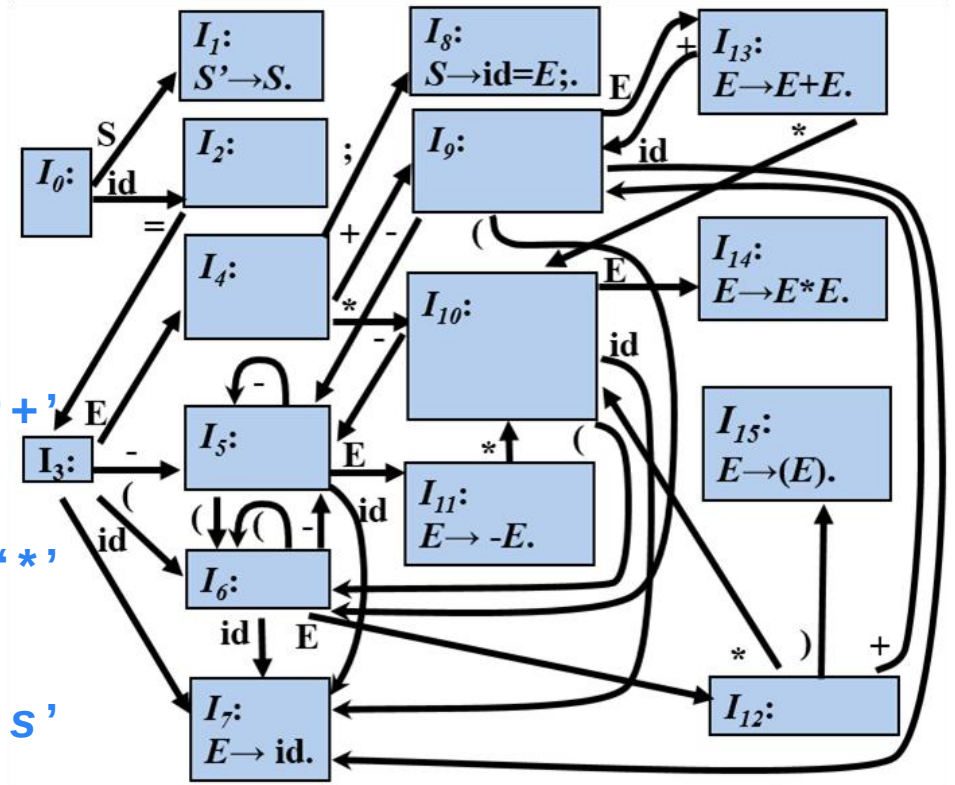
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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

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

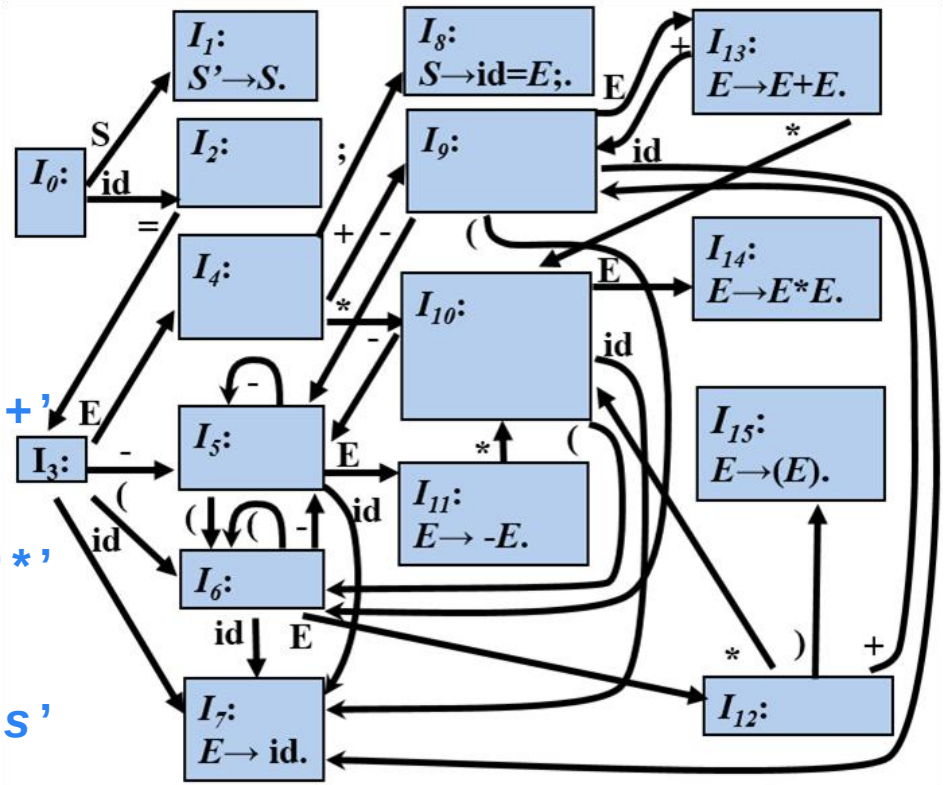
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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

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

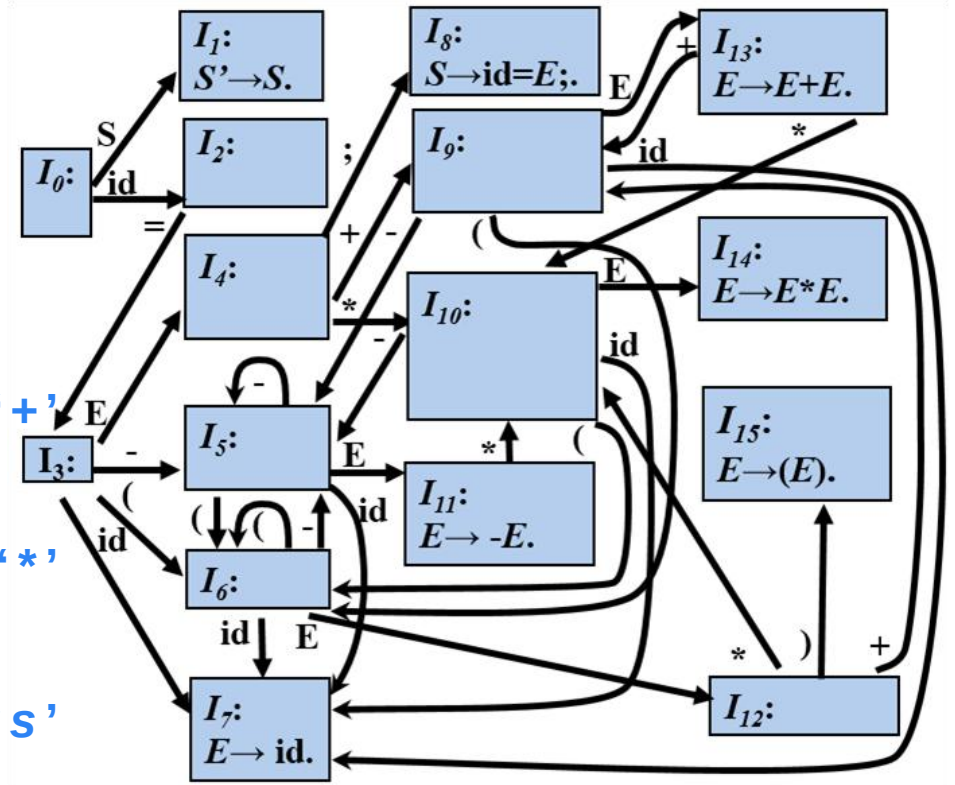
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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

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

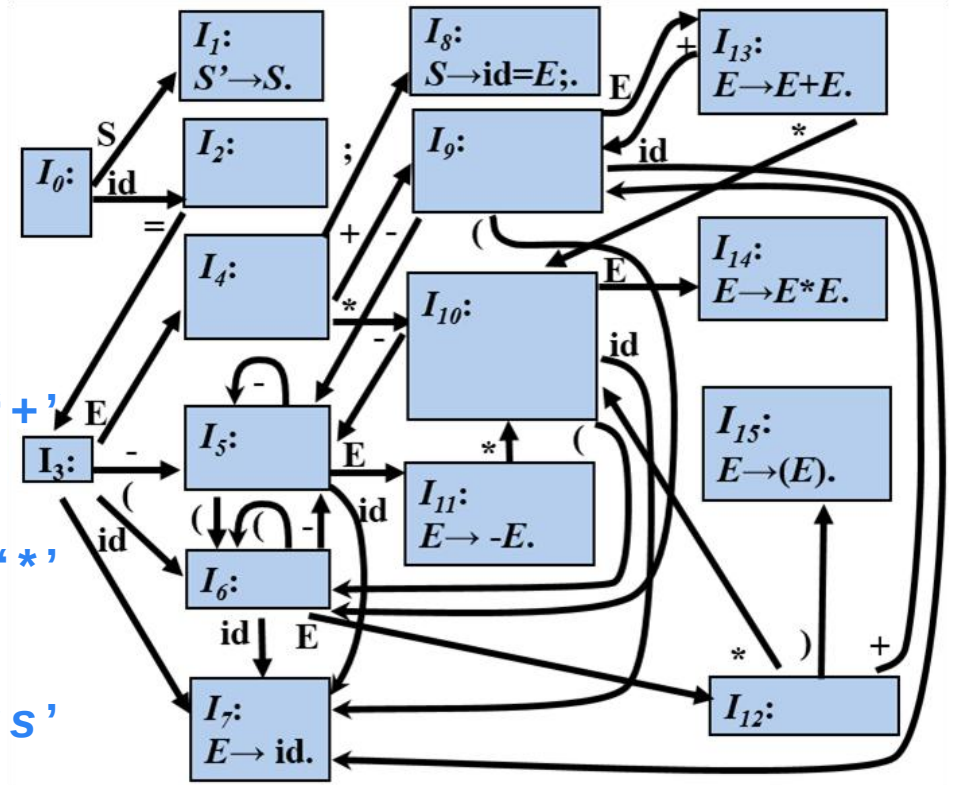
②E　$E_1 + E_2$ { E.addr = newtemp( );
　　gen(E.addr '=' E_1.addr '+' E_2.addr); }

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

④E　E_1 { E.addr = newtemp( );
　　gen(E.addr '=' 'uminus' E_1.addr); }

⑤E　(E_1) { E.addr = E_1.addr ; }

⑥E　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 → id = E; | L = E;

E → $E_1 + E_2$ | E_1 | (E_1) | id | L

L → id [E] | 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

例

- *int a*[3][5][8] ;

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

  一个整型变量占用4个字节，

  则*addr*(*a*[2][3][4]) = *base* + 2 * w_1 + 3*w_2 + 4*w_3

  = *base* + 2*160 + 3*32 + 4*4

  a[2]类型array(5, array(8, int))的宽度

  a[2][3]的类型array(8, int)宽度

  a[2][3][4]的类型 int 宽度

数组元素寻址（*Addressing Array Elements*）

- 一般情况，*k*维数组
  - 数组元素$a[i_1]\,[i_2]\,\ldots[i_k]$的相对地址是：

$$base+i_1\ w_1+i_2\ w_2+\ldots+i_k\ w_k$$

偏移地址

array(n_1, *array*(n_2, *array*(n_3, …*array*(n_k *type*) …)))

w_1 →$a[i_1]$ 的宽度
w_2 →$a[i_1]\,[i_2]$ 的宽度
…
w_k →$a[i_1]\,[i_2]\,\ldots[i_k]$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
    - $c = a[i_1][i_2]$；

    $addr(a[i_1][i_2]) = base + i_1*20 + i_2*4$

    t_1　　t_2

  - 三地址码

    $t_1 = i_1 * 20$

    $t_2 = i_2 * 4$

    $t_3 = t_1 + t_2$

    $t_4 = a\,[\,t_3\,]$

    $c = t_4$

数组元素寻址的*SDT*

- 赋值语句的基本文法

  S → id = E; | L = E;

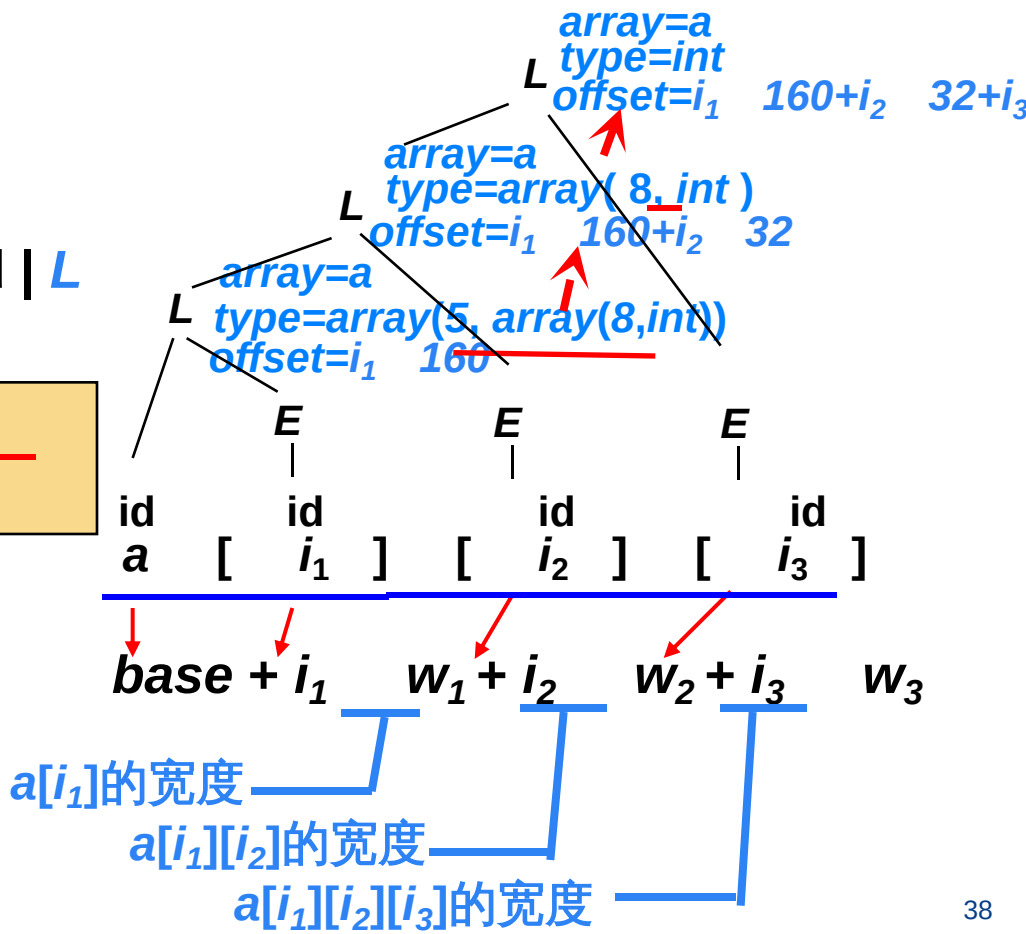
  E → $E_1 + E_2$ | E_1 | (E_1) | id | L

  L → id [E] | L [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]”

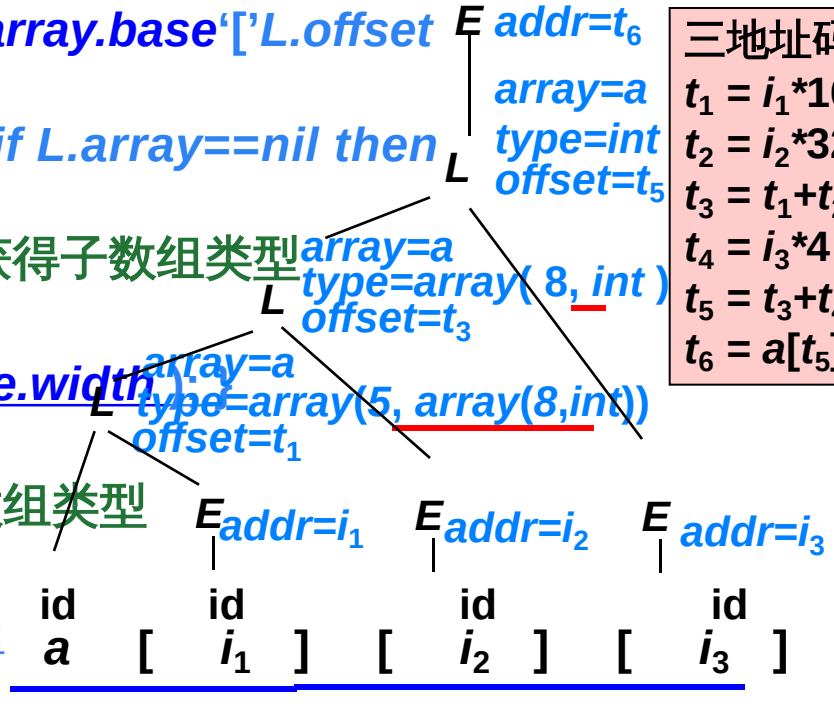
$addr(a[i_1][i_2][i_3]) = base + i_1 \times w_1 + i_2 \times w_2 + i_3 \times w_3$

t_1 t_2 t_4

S → id = E ;
| L = E; { gen(L.array.base ‘[’ L.offset ‘]’ ‘=’ E.addr); }

E → E_1 + E_2 | E_1 | (E_1) | id
| L {E.addr=newtemp(); gen(E.addr ‘=’ L.array.base‘[’L.offset ‘]’); }

L → id [E] { L.array = lookup(id.lexeme); if L.array==nil then error ;
L.type = L.array.type.elem ; //获得子数组类型
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 ; //获得子数组类型
t = newtemp();
gen(t ‘=’ E.addr ‘*’ L.type.width);
L.offset = newtemp();

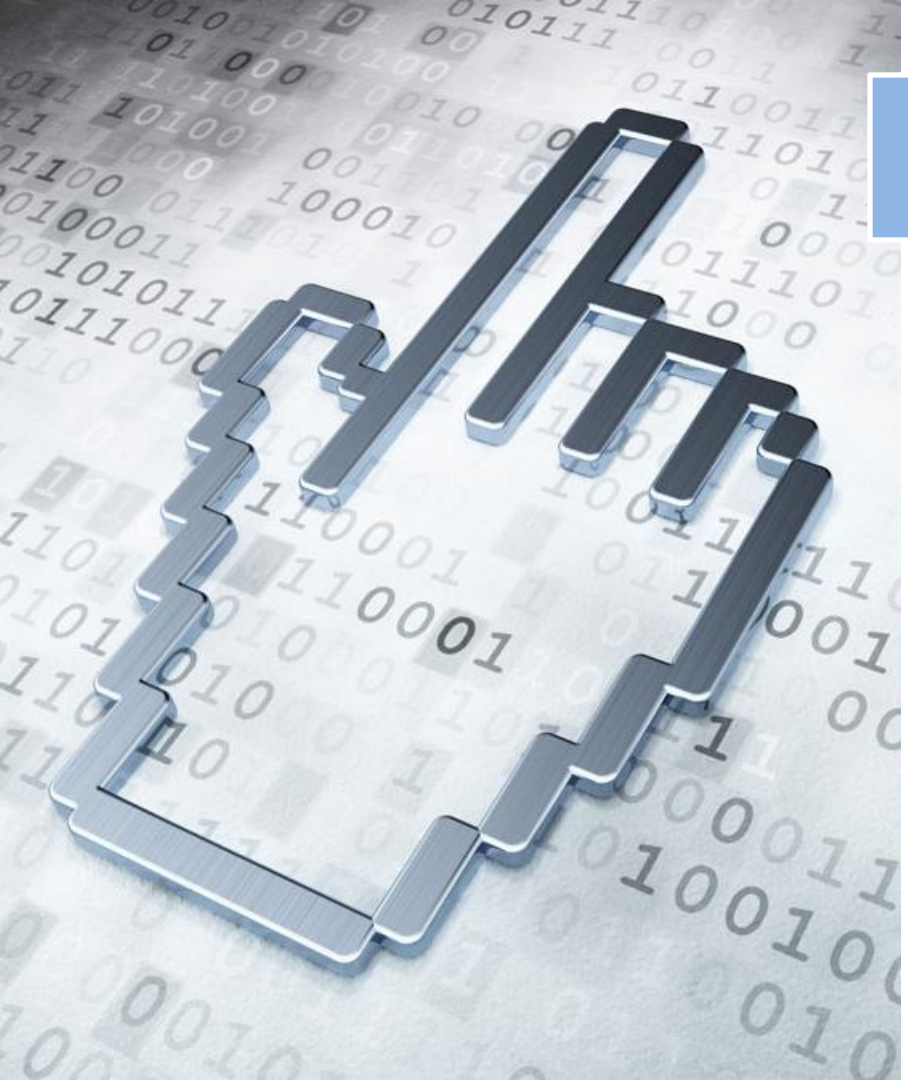


三地址码
$t_1 = i_1 * 160$
$t_2 = i_2 * 32$
$t_3 = t_1 + t_2$
$t_4 = i_3 * 4$
$t_5 = t_3 + t_4$
$t_6 = a[t_5]$

练习1

- 使用讲义中的翻译方案翻译下面赋值语句生成三地址码序列。假设*a*和*b*的类型表达式都是 array(3, array(5, real))，real类型占8个存储单元。

$$x = a[i][j] + b[i][j]$$



本章内容

6.3 类型检查*

- 为每个语法成分赋予一个类型表达式
  - 名字的类型表达式保存在符号表中
  - 其他语法成分(如表达式*E*或语句*S*)的类型表达式则作为属性保存在语义栈中。
- 类型检查的任务就是确定这些类型表达式是否符合一定的规则，这些规则的集合通常称为源程序的类型系统。
- 类型检查具有发现程序错误的能力。

6.3.1 类型检查的规则

- 类型综合（Type Synthesis）
  - 从子表达式的类型通过规则推导出整个表达式的类型（自底向上）
  - 要求名字在引用之前必须先进行声明
  - if *f*的类型为*s*→*t* and *x*的类型为*s* then 表达式*f*(*x*)的类型为*t*

```
double a = 5;
double b =
3.14;
int c = a + b;
```

a+b 类型是double,
可能需要类型转换

6.3.1 类型检查的规则

- 类型综合（Type Synthesis）
- 类型推断（Type Inference）
  - 类型推断是根据上下文信息，自动推导出变量或表达式的类型，无需显式声明。编译器通过分析代码的约束关系（如变量赋值、函数参数传递）来推断类型。
  - 经常被用于从函数体推断函数类型
  - if ***f*(*x*)**是一个表达式 then 对某两个类型变量***α***和***β***，***f***具有类型 **→*β* and *x*具有类型**

```
auto add( int t, double u) {
    return t + u;
}
```

add：int x double
double

6.3.2 类型转换

- 不同类型机内表示不同，且机器运算指令不同
- 编译器需要将运算分量转换为相同类型

```
int a = 5,  b = 3.14;
float c = 2;
int z = a * b + c;
```

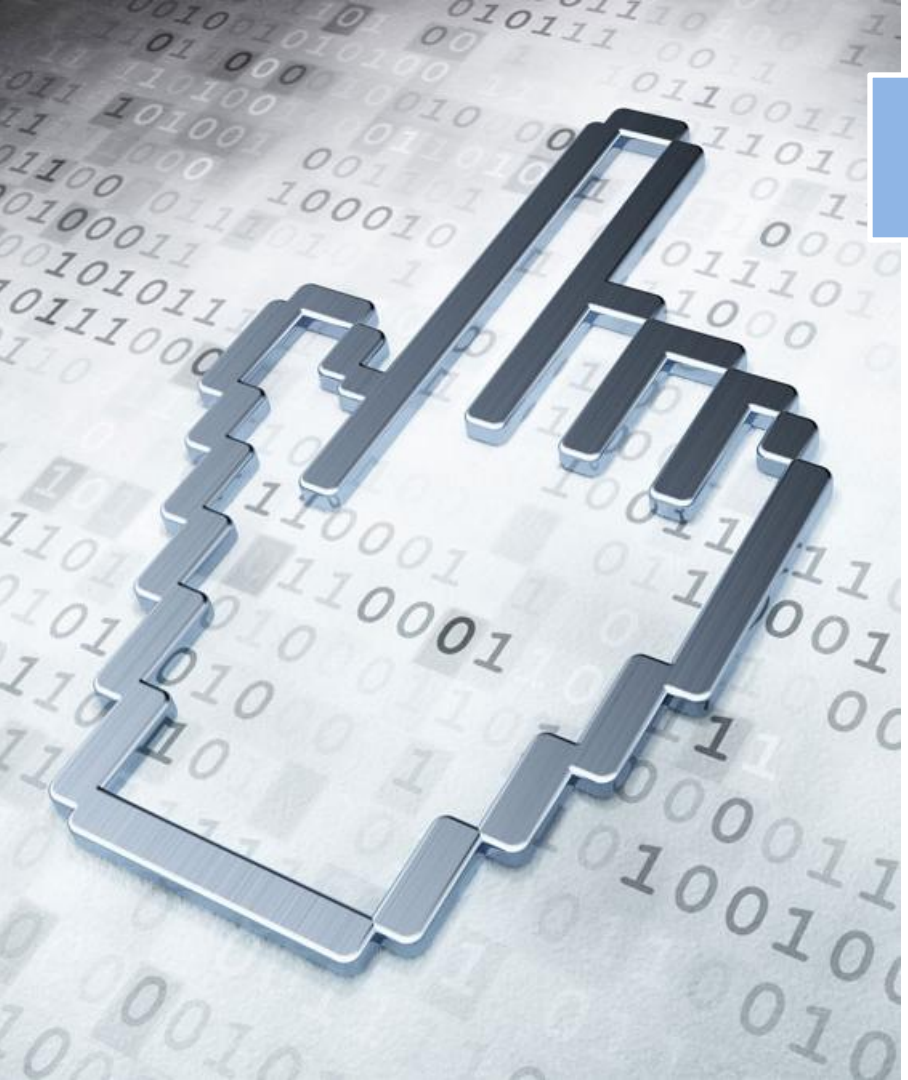
- 中间代码

t_1 = a int* b
t_2 = (float) t_1
t_3 = c float+ t_2
z = (int) t_3

6.3.2 类型转换

- 加入类型检查的E　　$E_1 + E_2$的SDT　为语法成分设置*type*属性

```
{ E.addr := newtemp()
if E1.type = integer and E2.type = integer then begin
    gencode( E.addr, ':=', E1.addr, 'int+', E2.addr ) ;
    E.type = integer
end
else if E1.type = integer and E2.type = float then begin
    u := newtemp();
    gencode( u, ':=', '(float)', E1.addr ) ;
    gencode( E.addr, ':=', u, 'float+', E2.addr ) ;
    E.type := float
end        . . . }
```



本章内容

6.4 控制语句的翻译

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

控制流语句的代码结构

- 例 S *if B then* S_1 *else* S_2

例：if a<b then y=0 else y=1

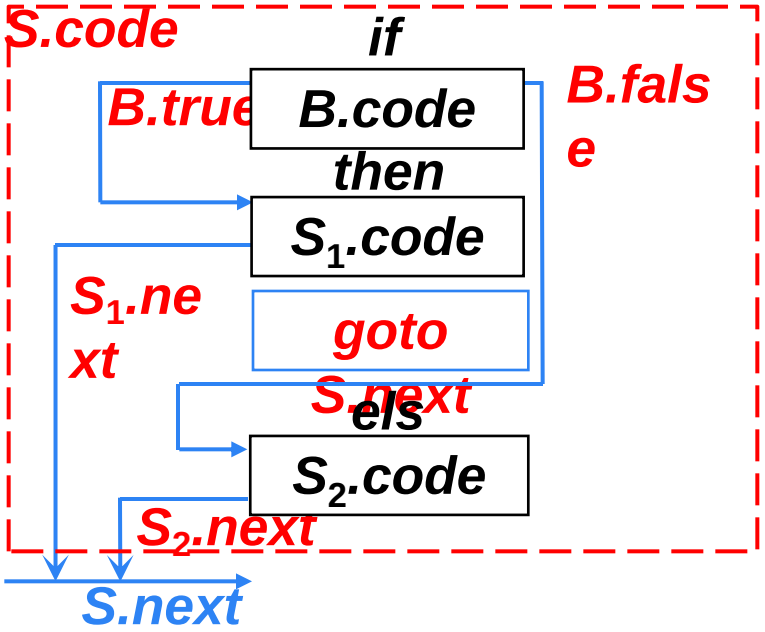
```
1       if  a < b  goto L1      } B
:       goto L2
2  L1:  y=0          ——S1
:       goto L3
3  L2:  y=1          —— S2
:  L3:
4
5
```

- 难点：确定跳转指令中目标地址
  - 第一种方法：为跳转目标预分配标号并通过继承属性传递到标注位置
    - 需第二遍扫描绑定标号与地址
  - 第二种方法：回填技术
    - 一遍扫描完成

布尔表达式*B*被翻译成由跳转指令构成的跳转代码

控制流语句的代码结构

- 例 S *if B then* S_1 *else* S_2



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

布尔表达式*B*被翻译成由跳转指令构成的跳转代码

用指令的标号标识一条三地址指令

控制流语句的SDT

newlabel( ): 生成一个新指令标号并返回

- *P* { *S.next = newlabel*(); }*S*{ *label*(*S.next*);}
- *S* { S_1*.next = newlabel*(); } S_1

  { *label*(S_1*.next*); S_2*.next*= *S.next* ;
- *S* id = *E* ; | *L* = *E* ;
- *S* *if B then* S_1

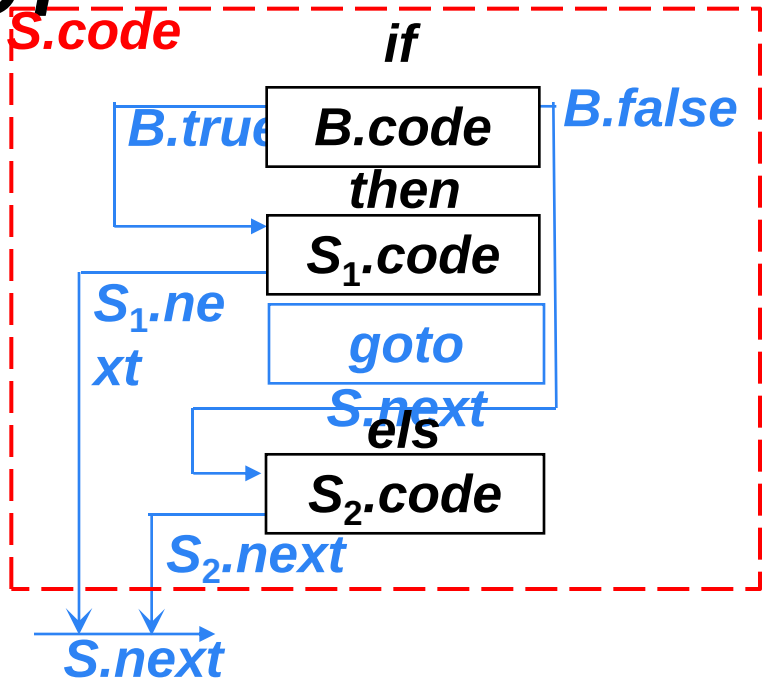
  | *if B then* S_1 *else* S_2

  | *while B do* S_1

label(*L*): 将标号*L*附加到下一条三地址指令前

if-then-else语句的SDT

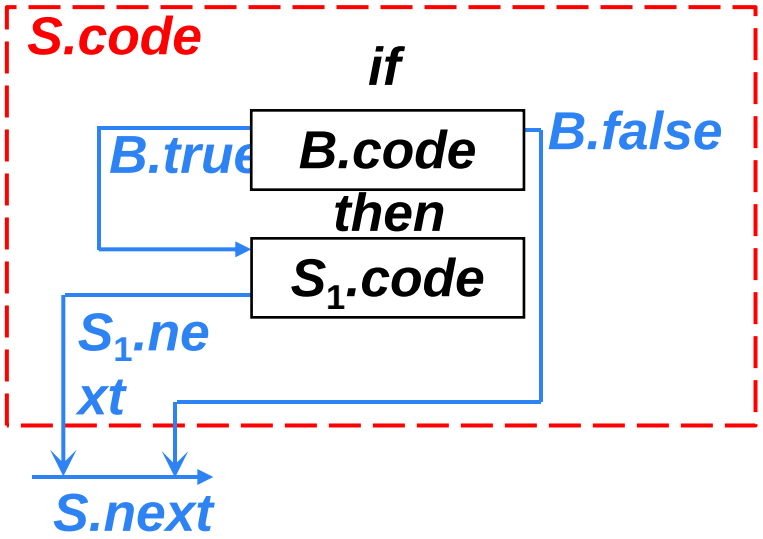
S $\quad$ *if* B *then* S_1 *else* S_2



S $\quad$ *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*

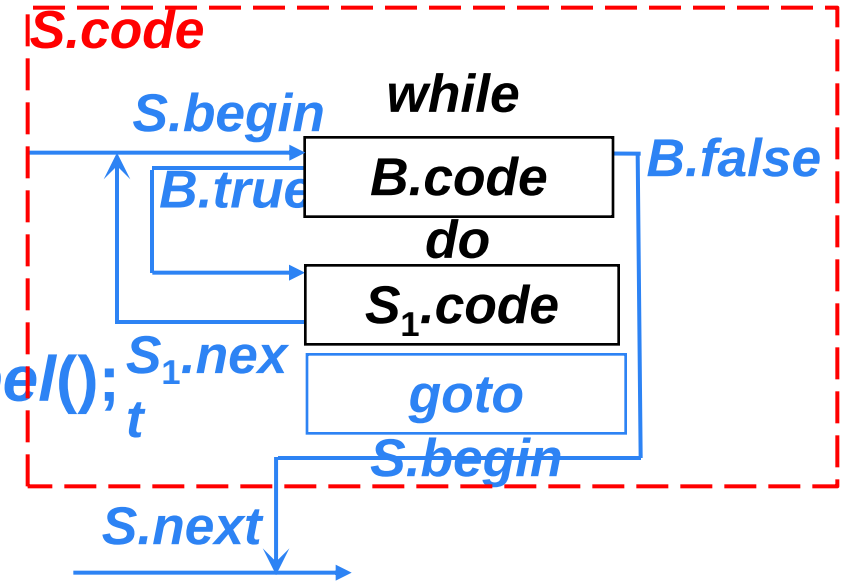
$S \rightarrow$ *if B then* S_1



$S \rightarrow$ *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

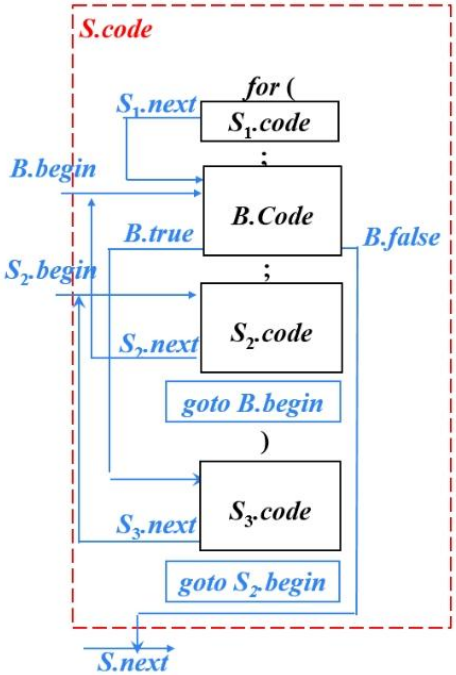
S → while B do S_1



S → 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); }

练习2

- 根据给定的代码结构图，参考6.4控制流语法的翻译方案，构造预标号的SDT，实现for控制流的翻译。



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）：
<, <=, >, >= , ==, ! =

布尔表达式的翻译

- 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跳转位置来表示的
- 例
  - 语句
  - 三地址代码

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

```
        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*

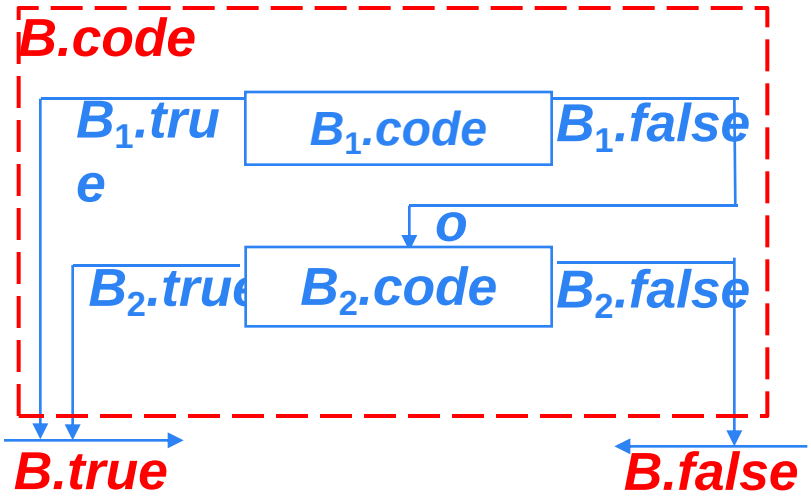
- $B \rightarrow E_1$ relop E_2 { *gen*(‘*if* ’ E_1.*addr relop* E_2.*addr* ‘*goto*’ *B.true*) ;

  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

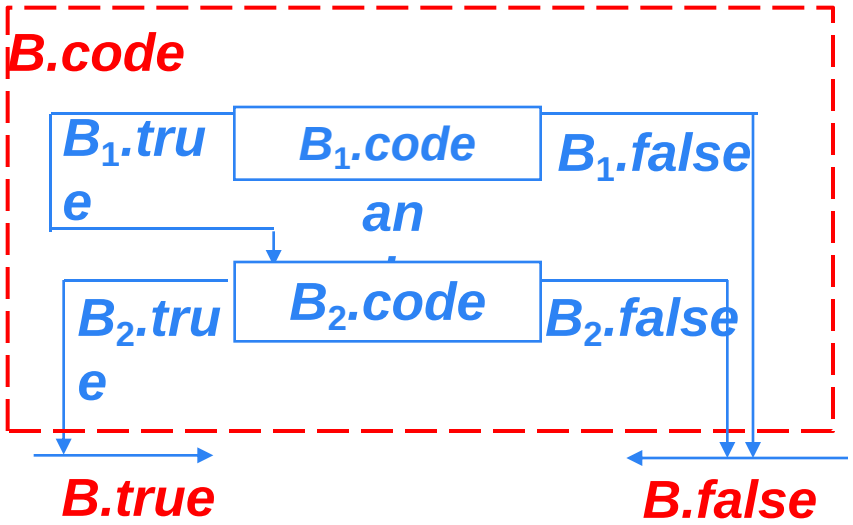
生成短路代码



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

生成短路代码



控制流语句的SDT

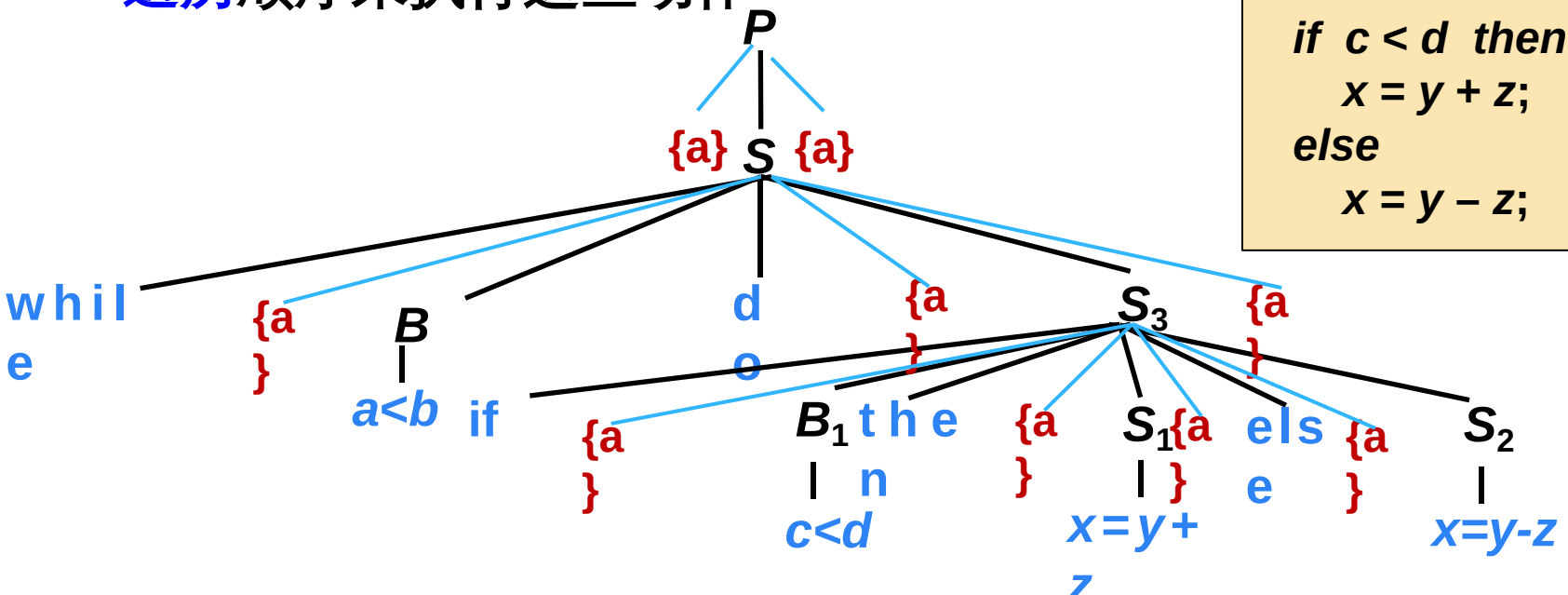
- P {a}S{a}
- S {a}S_1{a}S_2
- S id=E;{a} | L=E;{a}
- E E_1+E_2{a} | E_1{a} | (E_1){a}| id{a}| L{a}
- L id[E]{a} | L_1[E]{a}
- S if {a}B then {a}S_1
  | if {a}B then {a}S_1 {a} else {a}S_2
  | while {a}B do {a}S_1{a}
- B → {a}B or {a}B | {a}B and {a}B | not {a}B | ({a}B)
  | E relop E{a} | true{a} | 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遍历顺序来执行这些动作

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



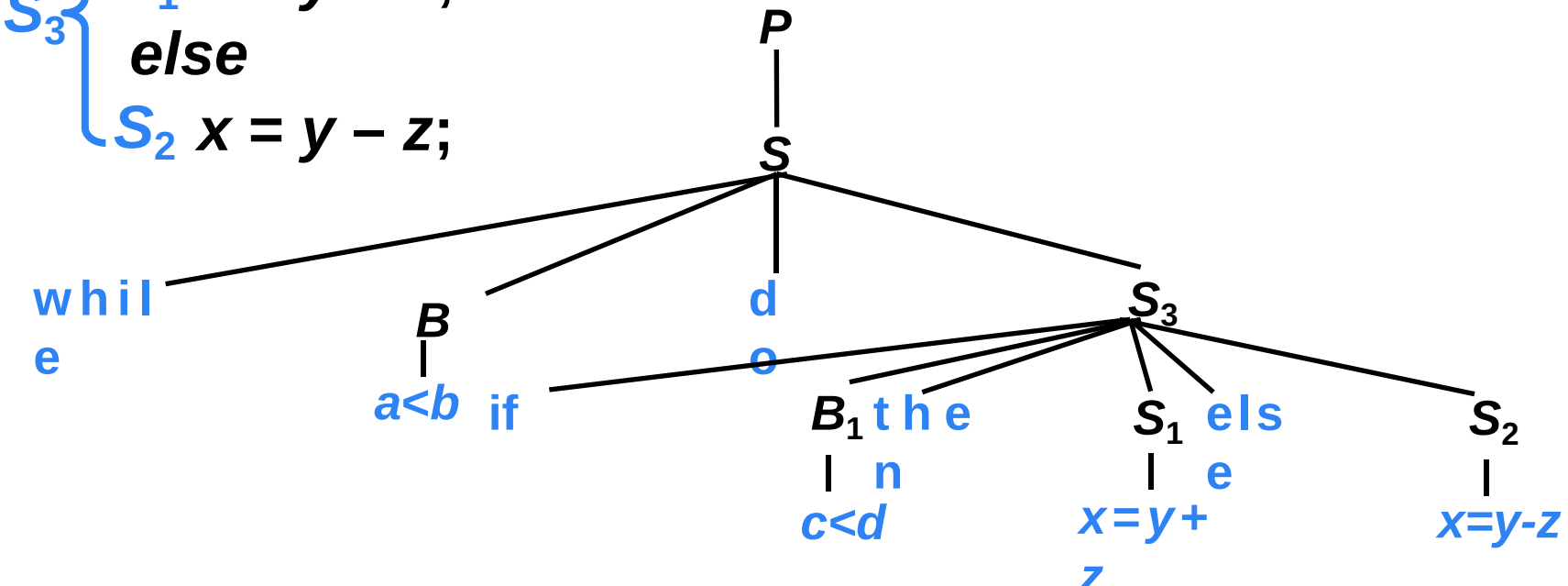
例

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

B

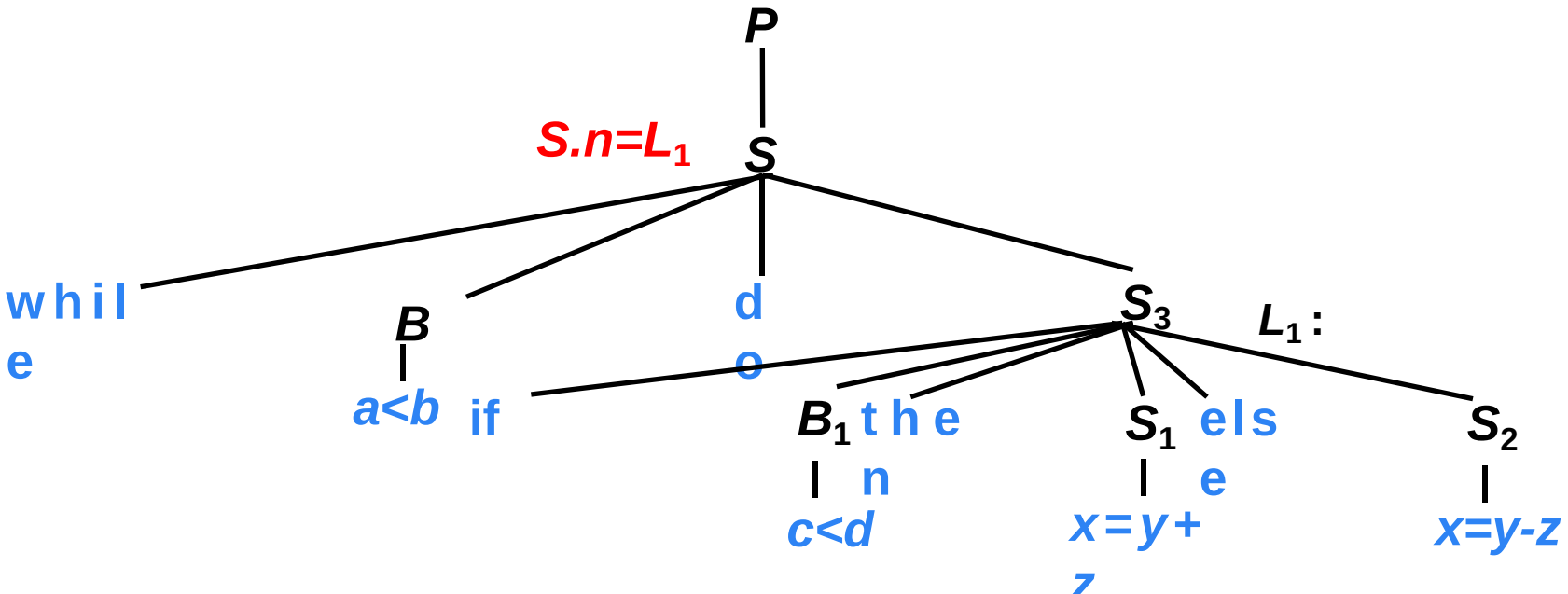
B1

S3



例

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

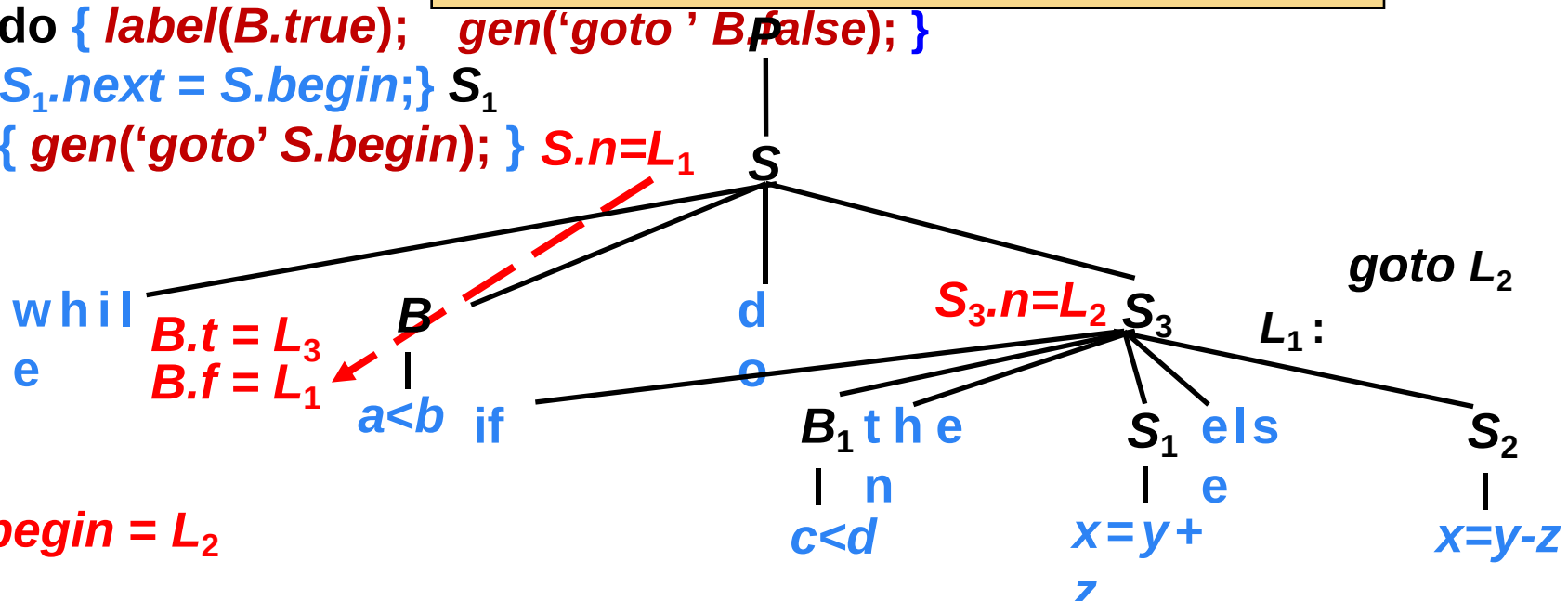


例

L_2 : *if a < b goto L*
goto L_1

S while {*S.begin = newlabel( );*
label(S.begin);
B.true = newlabel(
B.false = S.next;} **B**
do { *label(B.true);* *gen('goto ' B.false);* **}**
S_1*.next = S.begin;}* S_1
{ *gen('goto' S.begin);* } *S.n=*L_1

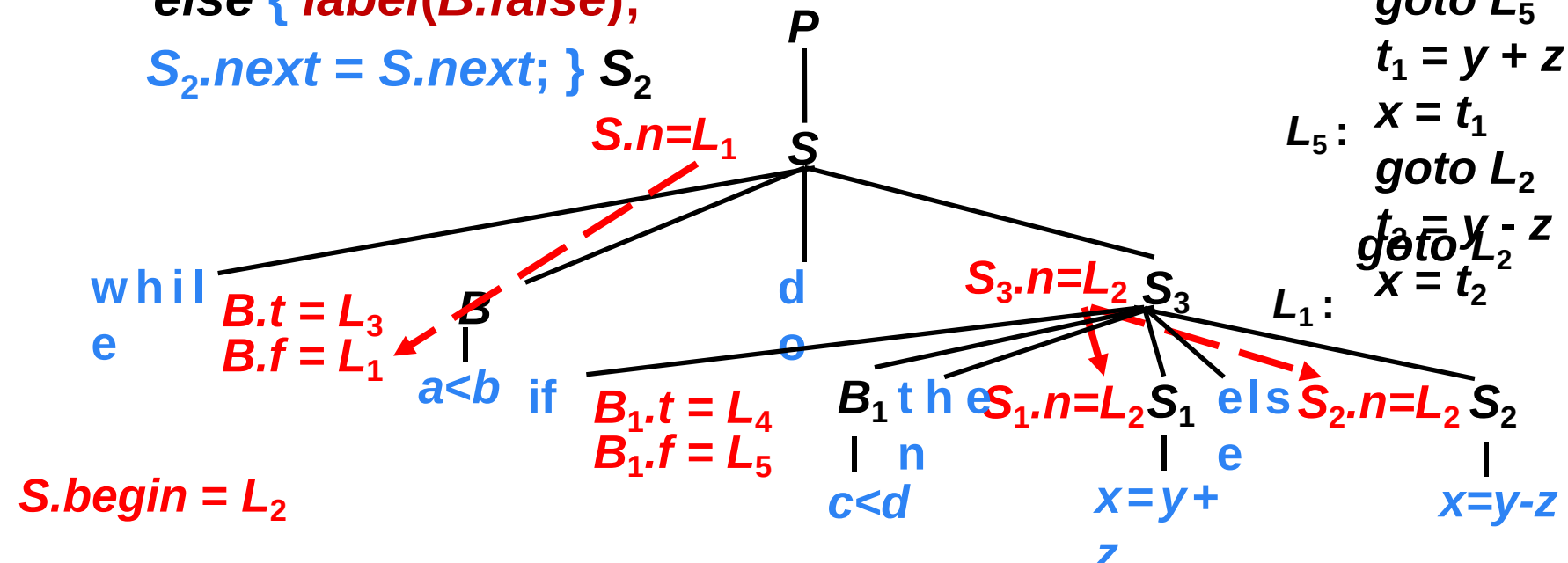
B → E_1 relop E_2 L_3 :
{ *gen('if '* E_1*.addr relop* E_2*.addr 'goto' B.true);*



S.begin = L_2

例

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



L_2 : *if* $a < b$ *goto* $B.t$
L_3 : *goto* L_1
if $c < d$ *goto* L_4
L_4 : *goto* L_5
$t_1 = y + z$
L_5 : $x = t_1$
goto L_2
$t_2 = y - z$
L_1 : $x = t_2$

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

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

需要第二趟扫描
绑定标号与
指令序号

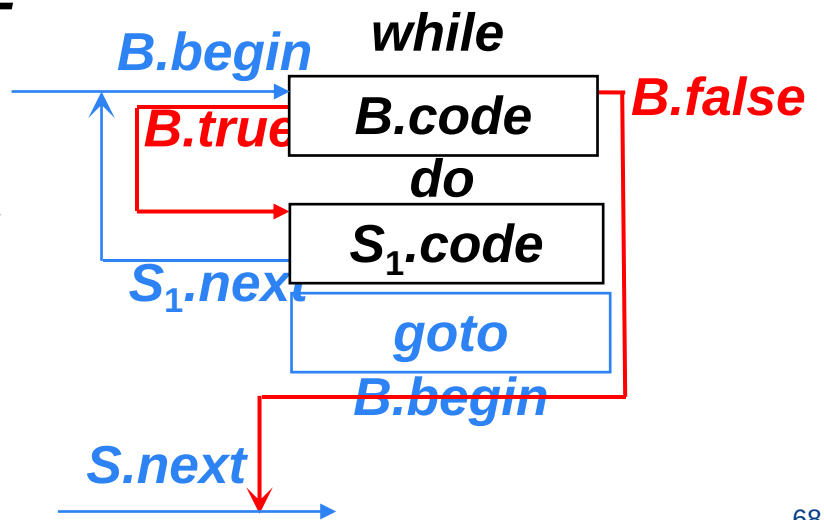
1: (*j*<, *a*, *b* , 3)
2: (*j* , - , - , 11)
3: (*j*<, *c*, *d* , 5)
4: (*j* , - , - , 8)
5: (+ , *y* , *z* , t_1)
6: (= , t_1, - , *x*)
7: (*j* , - , - , 1)
8: (- , *y*, *z* , t_2)
9: (= , t_2, - , *x*)
10: (*j* , - , - , 1)
11:

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

B.first →
1: *if a < b goto* **3**
2: *goto* 11
S_1.first →
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

1: *if***False** *a < b goto* **11**
2:
3: *if***False** *c < d goto* **8**
4:
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:

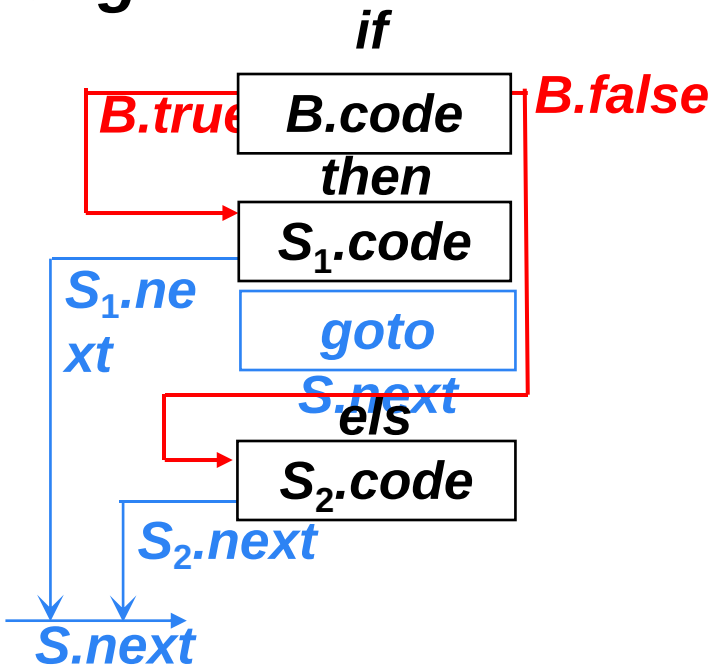
如何避免生成冗余的goto指令？



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

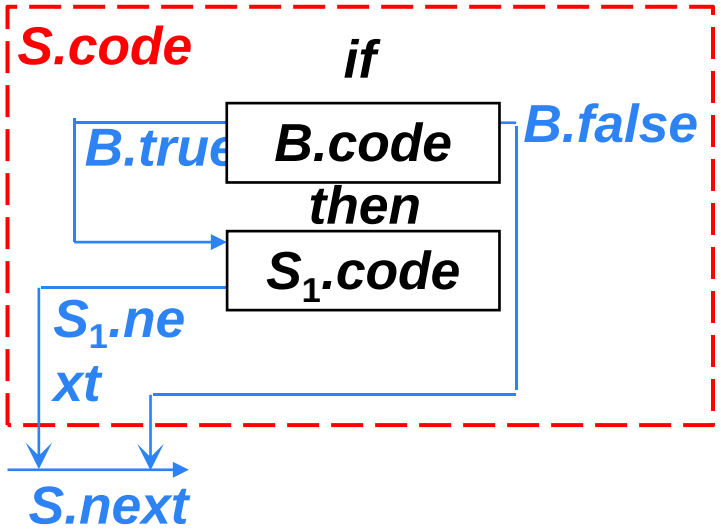
1: *if a < b goto 3*
B.first → 2: *goto 11*
3: *if c < d goto 5*
S_1.first → 4: *goto 8*
5: $t_1 = y + z$
6: $x = t_1$
S_2.first → 7: *goto 1*
8: $t_2 = y - z$
9: $x = t_2$
10: *goto 1*

1: *ifFalse a < b goto 11*
2:
3: *ifFalse c < d goto 8*
4:
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:



避免生成冗余的*goto*指令

修改*if-then*语句的*SDT*



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

不要生成任何跳转指令

S → *if* { $B.true =$ ***fall***; $B.false = S.next$; } B
then { $S_1.next = S.next$; } S_1

修改$B \rightarrow E_1$ relop E_2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); }

- $B \rightarrow E_1$ relop E_2

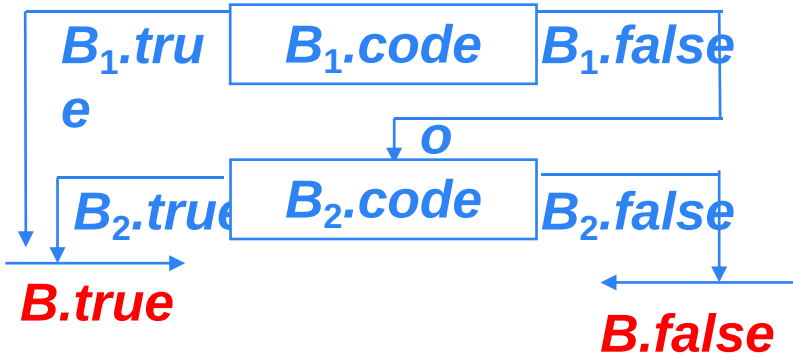
{ if B.true ≠fall and B.false ≠fall then

gen(‘if ’ E_1.addr relop E_2.addr ‘goto’ B.true) ; gen(‘goto’ B.false); }

else if B.true ≠ fall then gen(‘if ’ E_1.addr relop E_2.addr ‘goto’ B.true);

else if B.false ≠ fall then gen(‘ifFalse ’ E_1.addr relop E_2.addr‘goto’ B.false);

修改$B \rightarrow B_1$ or B_2 的SDT



$B \rightarrow$ { $B_1.true = B.true$; $B_1.false = newlabel()$; }B_1

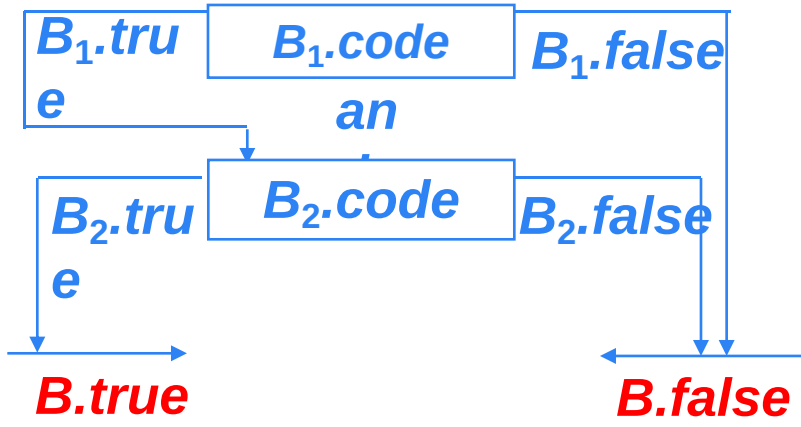
or { $label(B_1.false)$; $B_2.true = B.true$; $B_2.false = B.false$;

}B_2

$B \rightarrow$ { $B_1.true$ = if $B.true \neq fall$ then $B.true$ else $newlabel()$; $B_1.false = fall$; }B_1

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

修改$B \rightarrow B_1$ and B_2 的SDT



$B \rightarrow \{ B_1.true = newlabel(); B_1.false = B.false; \} B_1$

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

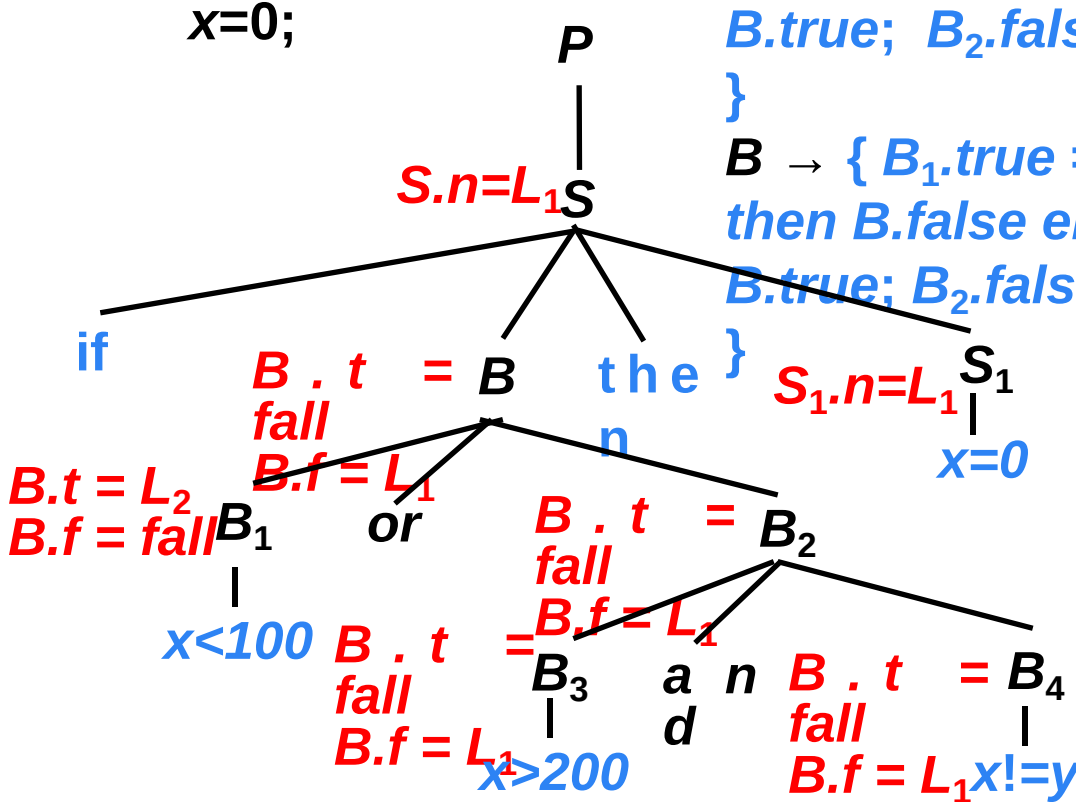
$B \rightarrow \{ B_1.true = fall; B_1.false =$ if $B.false \neq fall$ then $B.false$ else $newlabel(); \} B_1$

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

例

if (*x*<100 || *x*>200 && *x*!=*y*)

x=0;



S *if* { *B.true* = *fall*; *B.false* = *S.next*; } *B* *then* { S_1*.next* = *S.next*; } S_1

B → { B_1*.true* = if *B.true* ≠*fall* then *B.true* else *newlabel*(); B_1*.false* = *fall*; }B_1 or {B_2*.true* = *B.true*; B_2*.false* = *B.false*; }B_2 { *label*(B_1*.true*) }

B → { B_1*.true* = *fall*; B_1*.false* = if *B.false* ≠ *fall* then *B.false* else *newlabel*();} B_1 and { B_2*.true* *B.true*; B_2*.false* = *B.false*; }B_2 { *label*(B_1*.false*) }

if x<100 *goto* L_2

ifFalse x>200 *goto* L_1

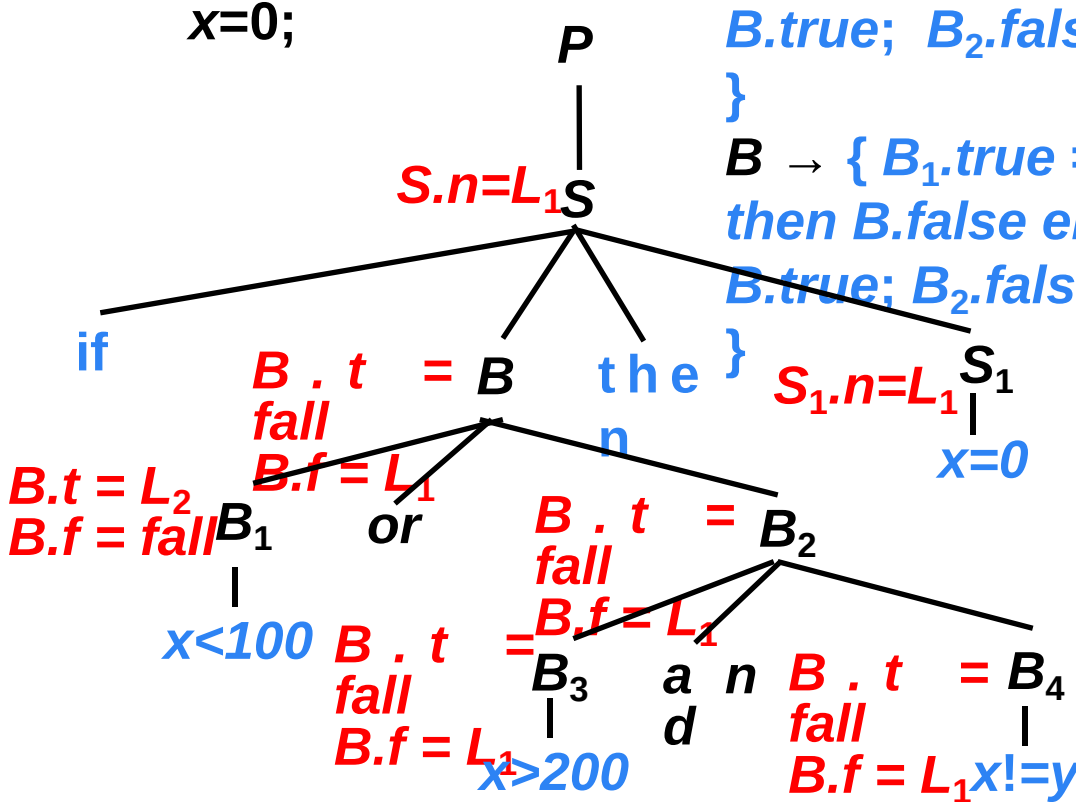
ifFalse x!=*y goto* L_1^1 L_2 :

x=0

例

if (*x*<100 || *x*>200 && *x*!=*y*)

x=0;



S *if* { *B.true* = *fall*; *B.false* = *S.next*; } *B*
then { S_1*.next* = *S.next*; } S_1

B → { B_1*.true* = if *B.true* ≠*fall then B.true else newlabel*(); B_1*.false* = *fall*; }B_1 or {B_2*.true* = *B.true*; B_2*.false* = *B.false*; }B_2 { *label*(B_1*.true*) }

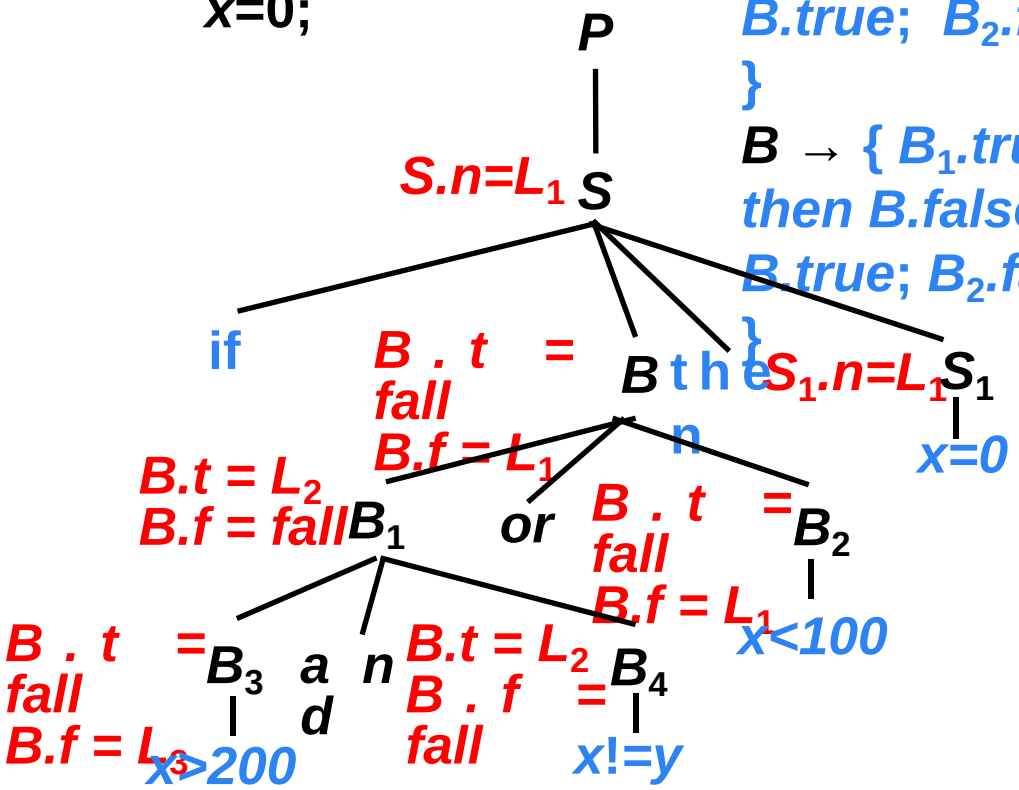
B → { B_1*.true* = *fall*; B_1*.false* = if *B.false* ≠ *fall then B.false else newlabel*();} B_1 and { B_2*.true* *B.true*; B_2*.false* = *B.false*; }B_2 { *label*(B_1*.false*) }

if x<100 *goto* L_2
ifFalse x>200 *goto* L_1
ifFalse x!=*y goto* L_1
L_2 :
L_1 : *x*=0

例

if (*x*>200 && *x*!=*y* || *x*<100)

x=0;



S → *if* { *B.true* = *fall*; *B.false* = *S.next*; } *B* *then* { S_1*.next* = *S.next*; } S_1

B → { B_1*.true* = if *B.true* ≠*fall then B.true else newlabel*(); B_1*.false* = *fall*; }B_1 or {B_2*.true* = *B.true*; B_2*.false* = *B.false*; }B_2 { *label*(B_1*.true*) }

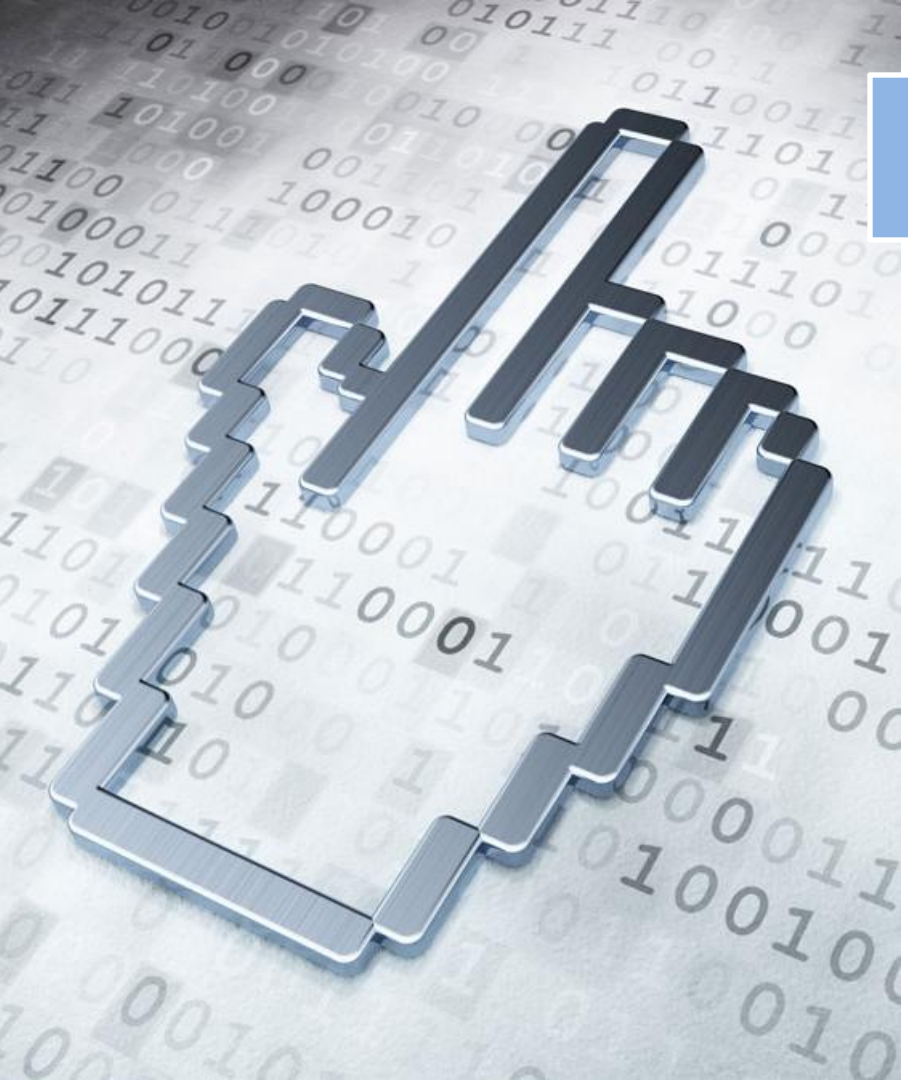
B → { B_1*.true* = *fall*; B_1*.false* = if *B.false* ≠ *fall then B.false else newlabel*();} B_1 and { B_2*.true* *B.true*; B_2*.false* = *B.false*; }B_2 { *label*(B_1*.false*) }

ifFalse x>200 goto L_3

L_3 : *if x!=y goto* L_2

L_2 : *ifFalse x<100 goto* L_1

L_1 : *x*=0



本章内容

6.5 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。具有相同的目标标号的跳转指令组成一个列表并以综合属性的形式进行传递。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。
  - 优点：在一趟扫描中即可完成跳转语句生成与目标标号的填充。

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

- 将生成的指令按顺序编号，指令的序号作为标号

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, j)***
  - 将 *j*作为目标标号插入到 *p*所指列表中的各跳转指令中

布尔表达式的回填

- $B \quad E_1$ relop E_2

{

$B.truelist = makelist(nextquad);$

$B.falselist = makelist(nextquad+1);$

$gen(\text{‘if ’ } E_1.addr \text{ relop } E_2.addr \text{ ‘goto _’});$

$gen(\text{‘goto _’});$

}

nextquad：即将生成的下一条指令的标号，也就是下一条跳转指令的标号

- 语义动作都在产生式末尾，可以在自底向上语法分析中实现的SDT

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

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

{

B.falselist = makelist(nextquad);

gen('goto _');

}

布尔表达式的回填

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

{

$B.truelist = B_1.truelist$;

$B.falselist = B_1.falselist$;

}

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

{

$B.truelist = B_1.falselist$;

$B.falselist = B_1.truelist$;

}

$B \quad B_1$ or B_2

$B \quad B_1$ or M B_2

{

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

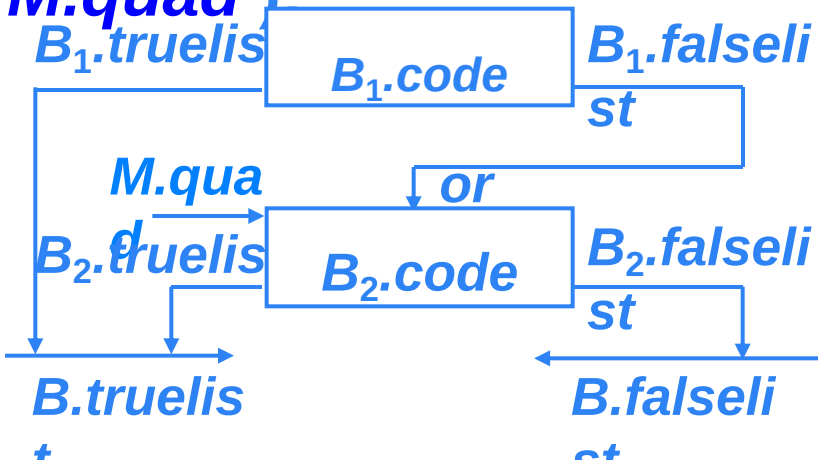
$B.falselist = B_2.falselist$;

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

}

$M \quad \varepsilon$

{ $M.quad = nextquad$; }



$B \quad B_1$ and B_2

$B \quad B_1$ and M B_2

{

$B.truelist = B_2.truelist$;

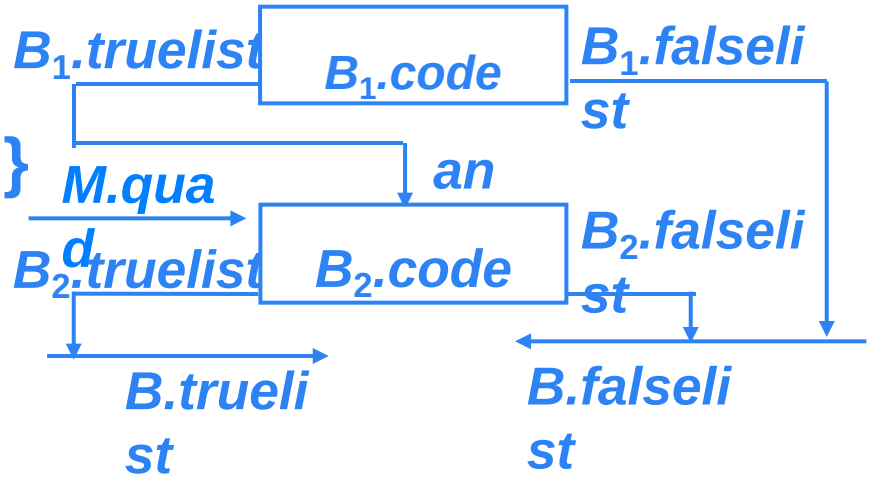
$B.falselist = merge(B_1.falselist, B_2.falselist)$;

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

}

$M \quad \varepsilon$

{ $M.quad = nextquad$; }

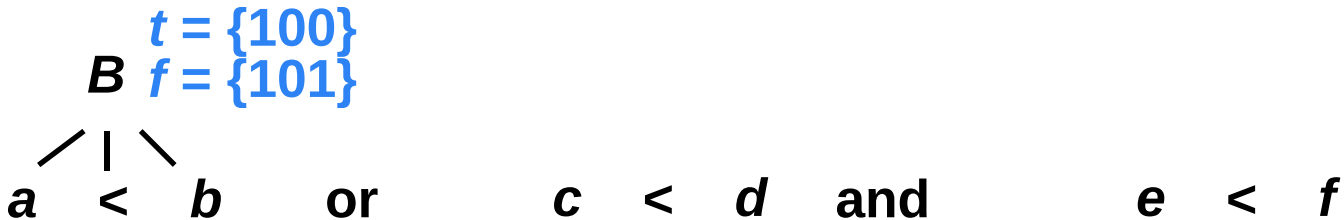


例

$B \quad E_1$ 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* ;

}

$M \rightarrow \varepsilon$

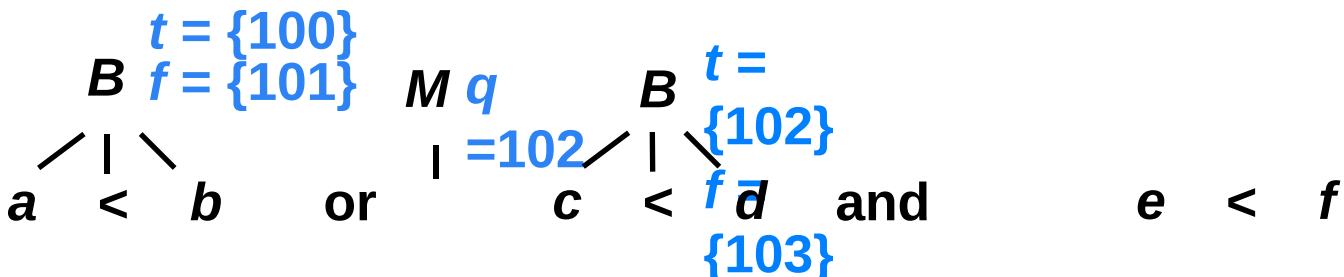
{ *M.quad* = *nextquad* ; }

100: *if a<b goto* _

101: *goto* _

102: *if c<d goto* _

103: *goto* _



例

$B \rightarrow B_1$ and M B_2

{

$B.truelist = B_2.truelist$;

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

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

}

$M \rightarrow \varepsilon$

{ $M.quad = nextquad$; }

100: *if a<b goto* _

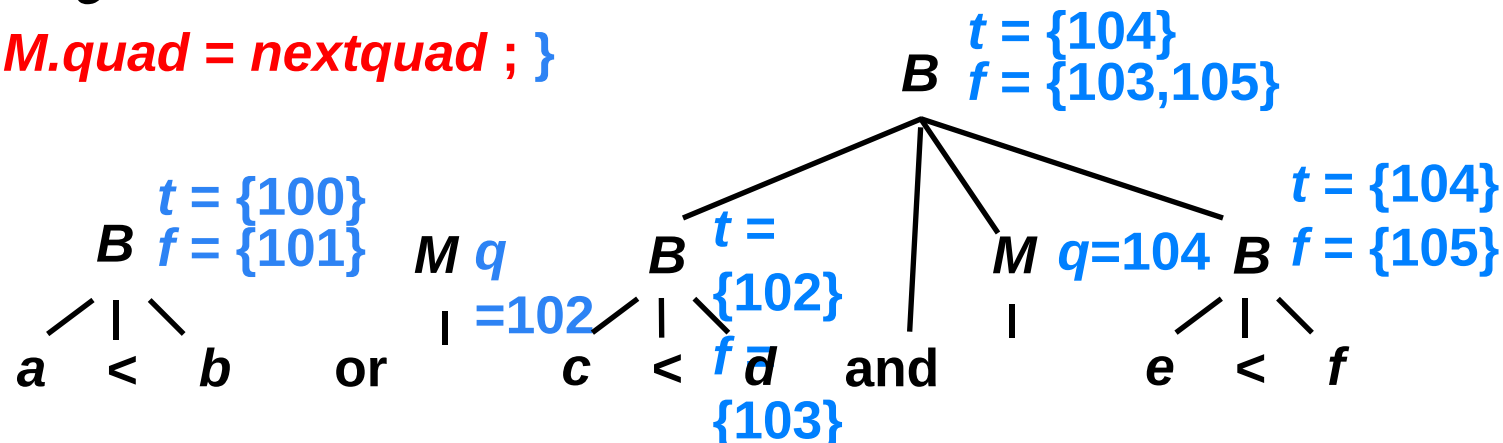
101: *goto* _

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: *if c<d goto* _

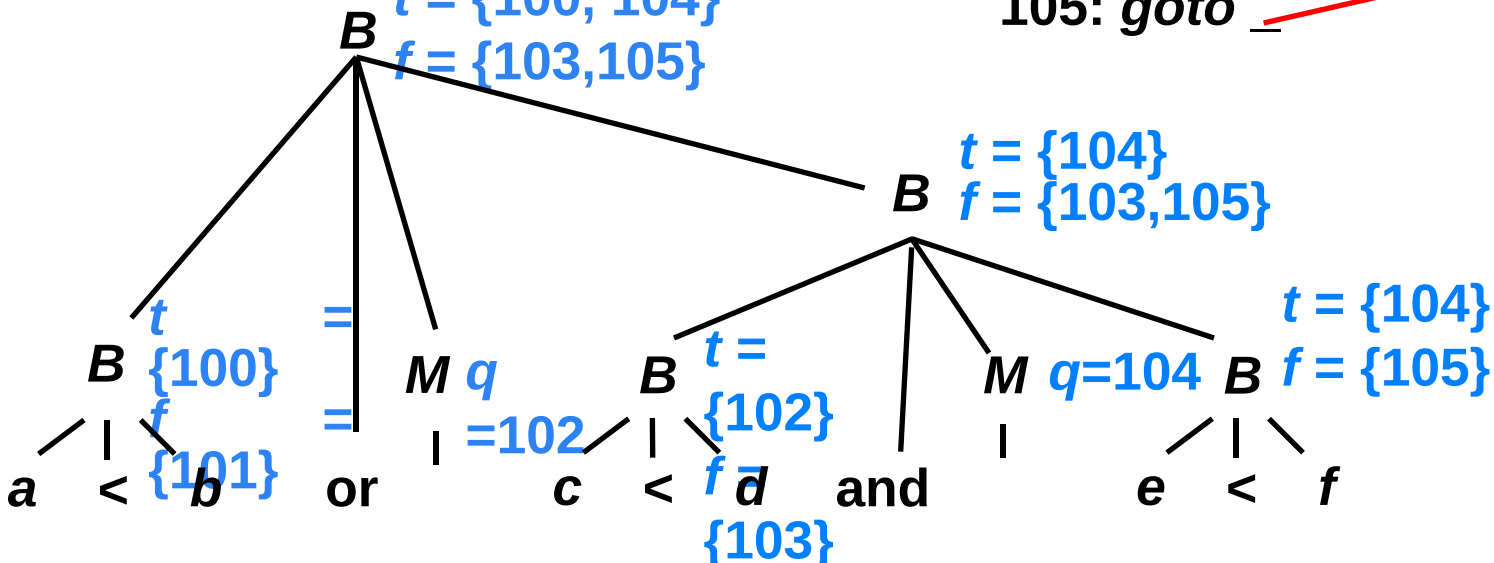
103: *goto* _

104: *if e<f goto* _

105: *goto* _

true

false



控制流语句的回填

- 文法
  - S S_1 S_2
  - S id = E ; | L = E ;
  - S if B then S_1
    | if B then S_1 else S_2
    | while B 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 if B then S_1

S if B then M S_1

{

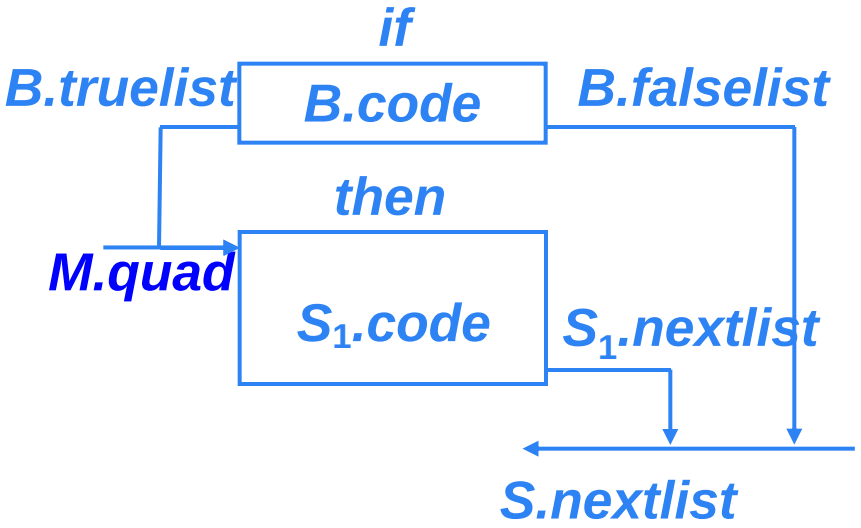
$S.nextlist = merge(B.falselist, S_1.nextlist)$;

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

}

M ε

{ $M.quad = nextquad$; }



S if B then S_1 else S_2

S if B then M_1 S_1 N else M_2 S_2

{

$S.nextlist =$

$merge(\ merge(S_1.nextlist,$

$N.nextlist),$

$S_2.nextlist\ );$

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

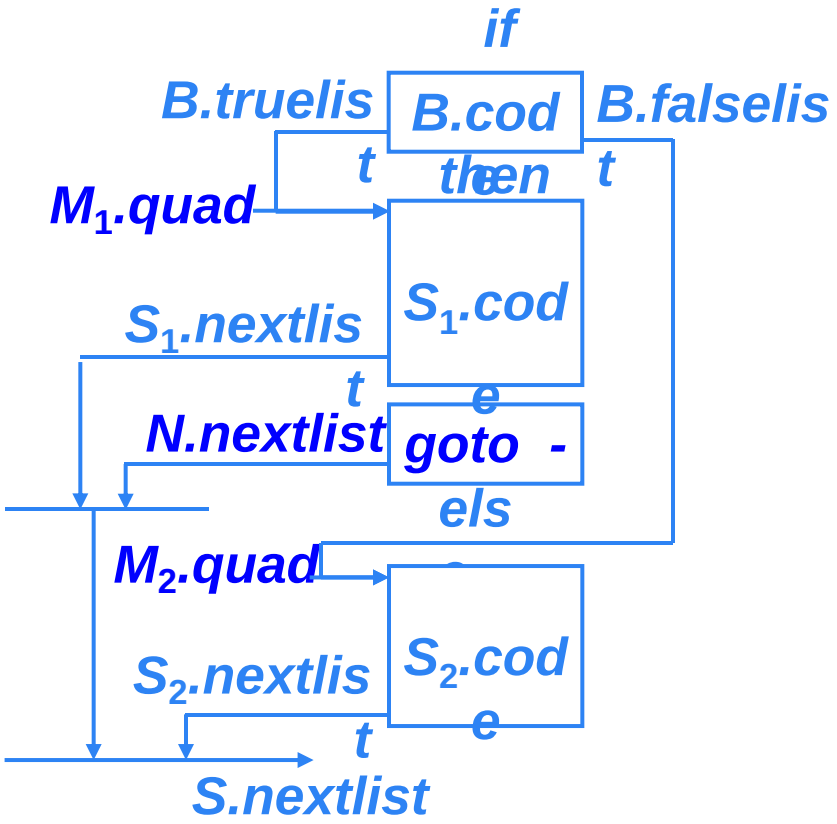
$backpatch(B.falselist, M_2.quad);$

}

N ε

{

$N.nextlist = makelist(nextquad);$



S while B do S_1

S while M_1 B do M_2 S_1

{

$S.nextlist = B.falselist$;

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

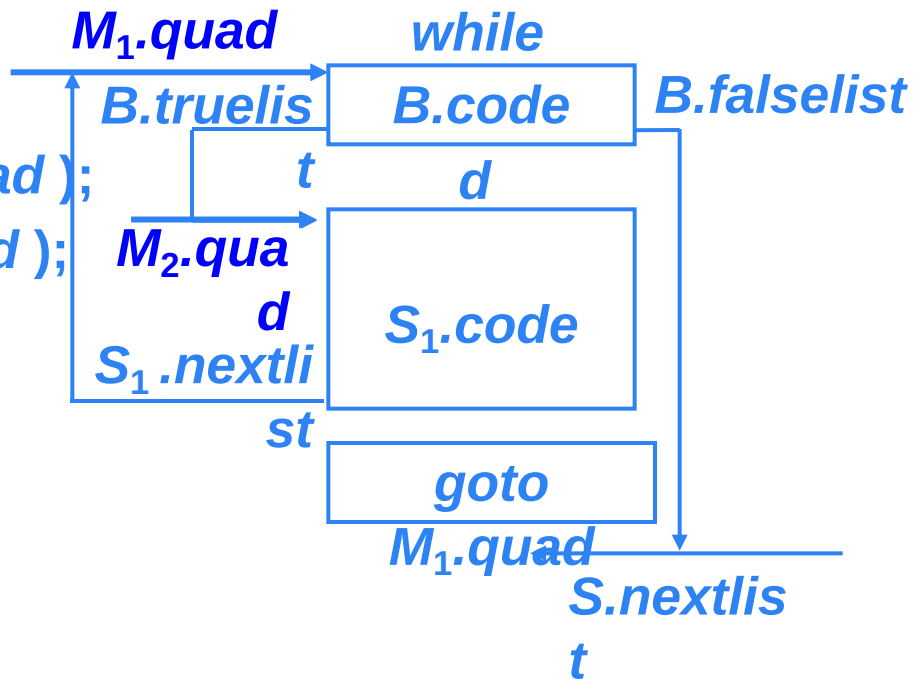
$backpatch(B.truelist, M_2.quad)$;

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

}

M ε

{ $M.quad = nextquad$; }



$S \quad S_1 \; S_2$

$S \quad S_1 \; M \; S_2$

{

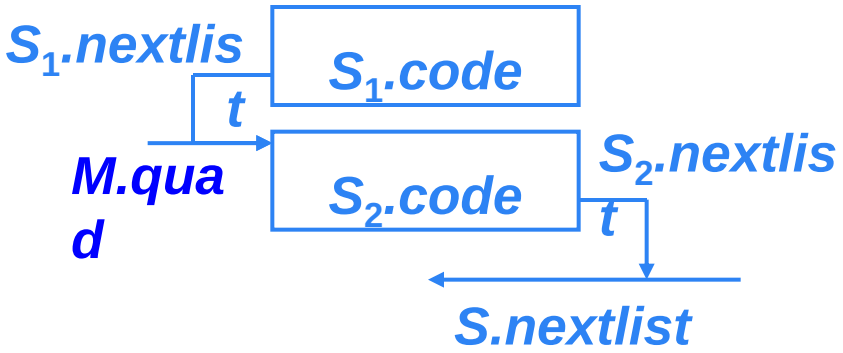
$S.nextlist = S_2.nextlist$;

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

}

$M \quad \varepsilon$

{ $M.quad = nextquad$; }



S id = E ; | L = E ;

S id = E ; | L = E ;{ $S.nextlist = null$;
}

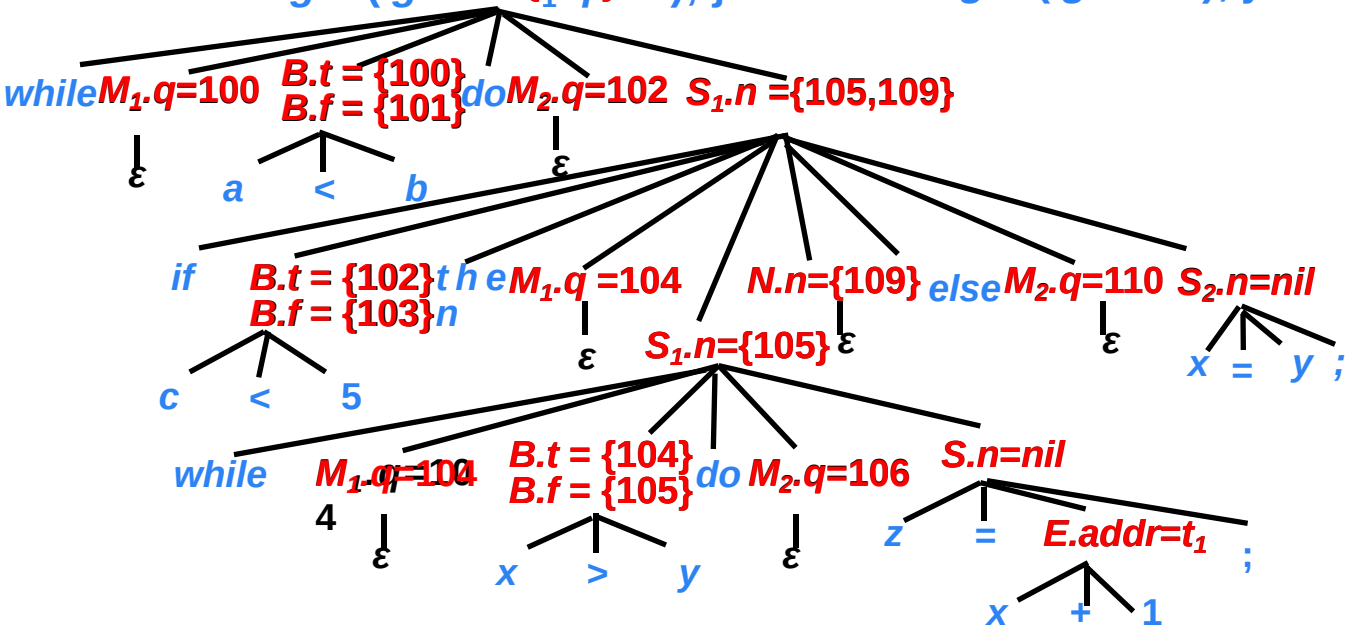
例

```
while  a < b  do

    if  c < 5  then

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

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

S 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 ε { $N.nextlist = makelist(nextquad)$;
$gen('goto\ \_')$; }



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
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:

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

练习3

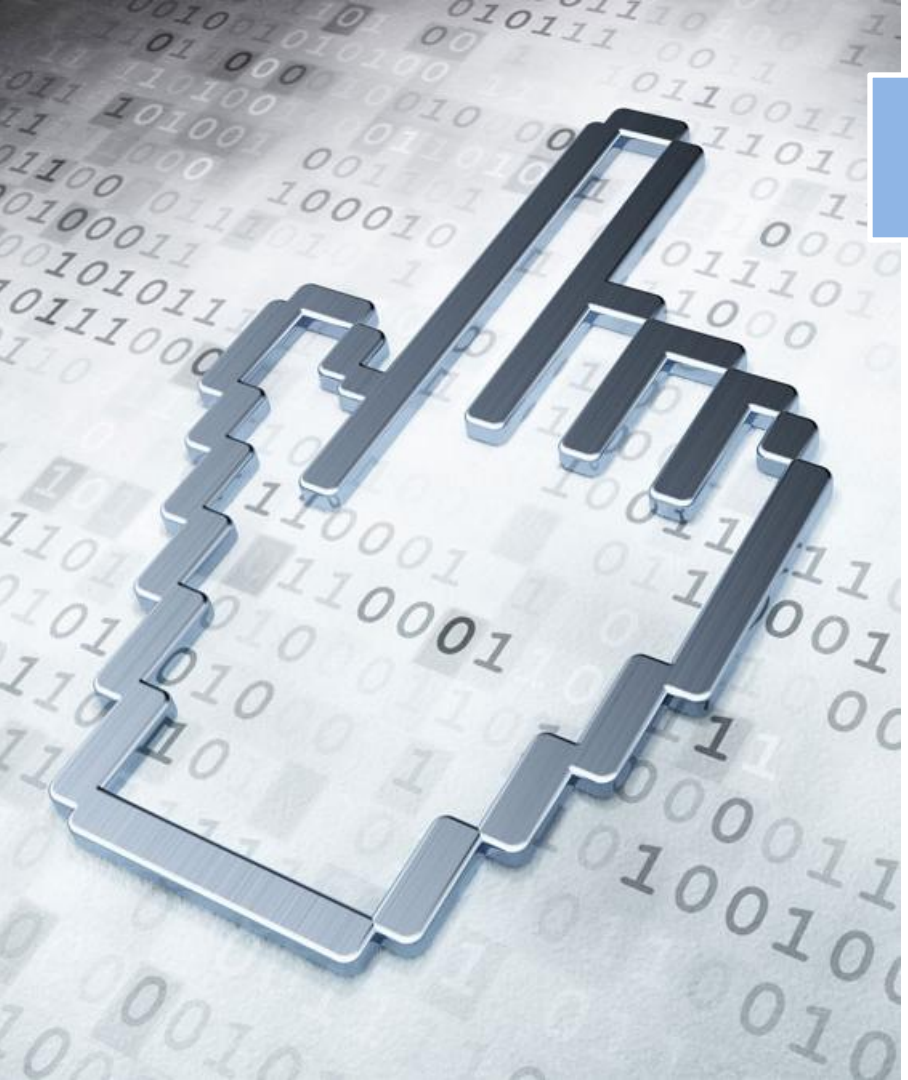
- 试将下面的语句翻译成三地址码和对应的四元式序列（指令标号从100开始，跳转指令目标标号是数字代表的序号）。不必考虑避免生成冗余的goto语句。

（1）while a<c and b<d do

if a=1 then c=c+1

else while a<=d do

a=a+2



本章内容

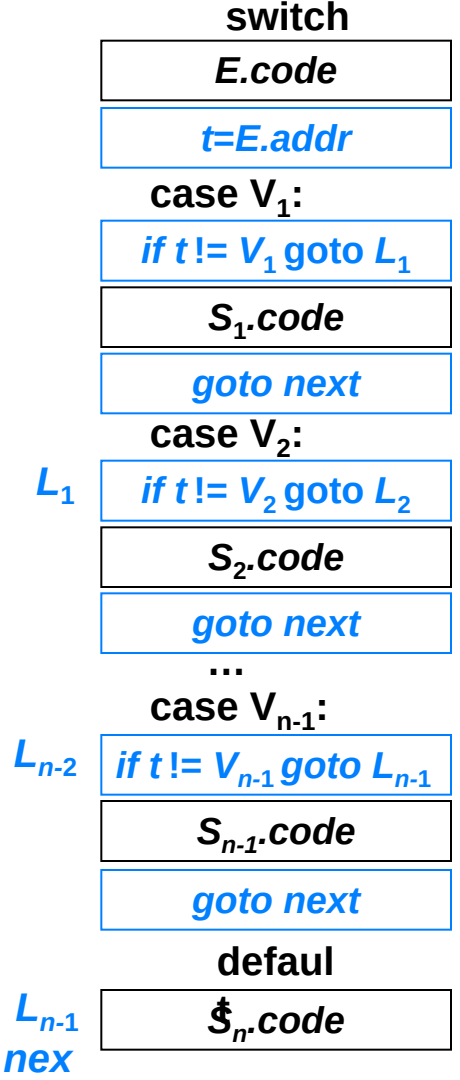
6.6 *switch*语句的翻译

- switch语句语法

```
switch E
   begin
          case V1: S1
          case V2: S2
          . . .
          case Vn – 1: Sn – 1
          default: Sn
   end
```

- switch语句文法

```
S     switch E  begin  Clist end

Clist     case V: S  Clist
```

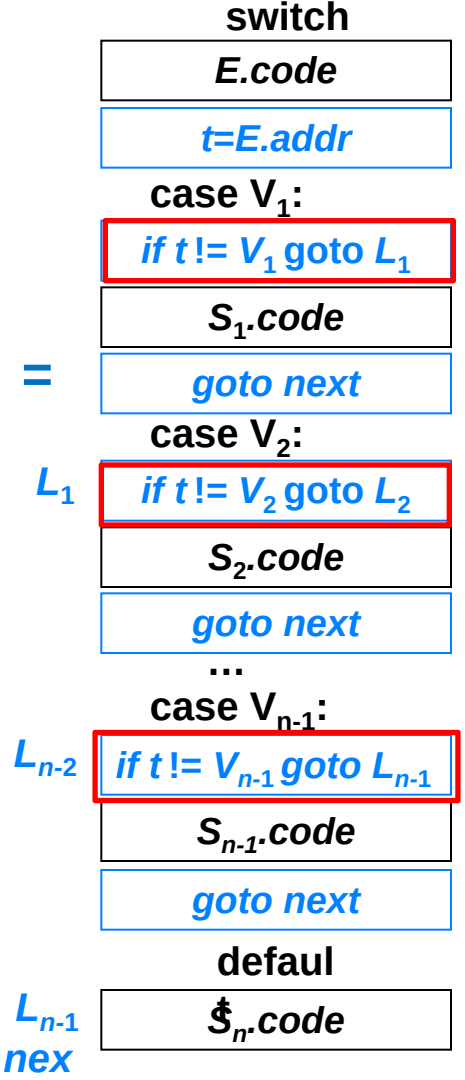


6.6 *switch*语句的翻译

S switch *E* { *t = newtemp*();
gen(t '=' E.addr);
next = newlabel(); l = newlabel(); }
begin *Clist* end

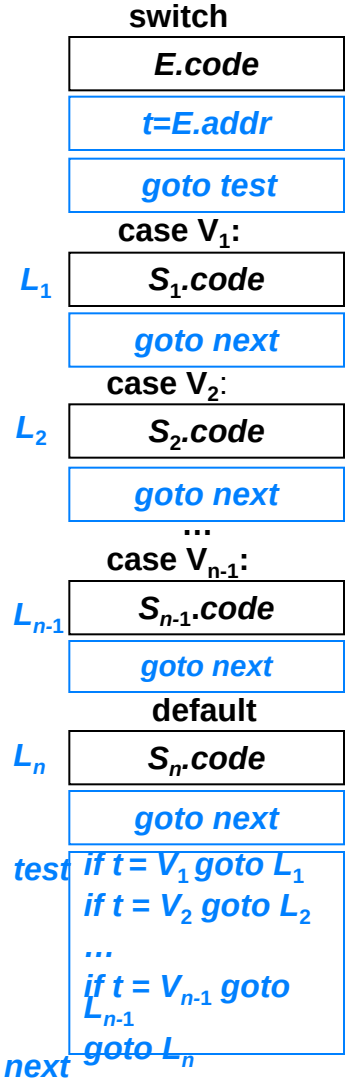
Clist case V: { *label(l); l = newlabel*();
gen('if' t '!=' V 'goto' l); }
S { *gen('goto' next*); }
Clist

Clist default: { *label(l); }*
S { *label (next*); }



*switch*语句的另一种翻译

```
switch E
   begin
         case V1: S1
         case V2: S2
         . . .
         case Vn – 1: Sn – 1
         default: Sn
   end
```

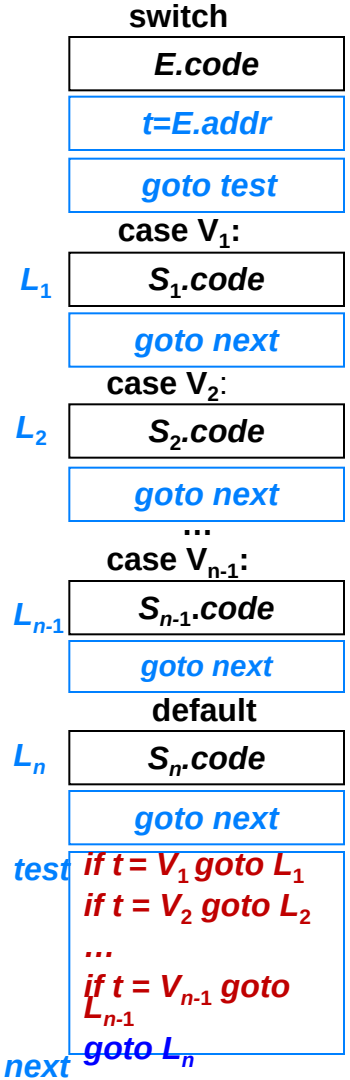


优势：根据分支的个数以及这些值是否在一个较小的范围内，这种条件跳转指令序列可以被翻译成最高效的*n*路分支(散列表)

switch语句的另一种翻译

```
S     switch E { t = newtemp();  gen( t '=' E.addr );
                 test = newlabel();  next = newlabel();
                 gen('goto' test); }
      begin Clist end
Clist     case V: { I = newlabel(); label( I ); q(V, I ); }
                  S { gen('goto' next); } Clist
Clist     default: { I = newlabel(); label( I ); }
                   S { gen('goto' next) ; label(test);
                       for 队列 q 中的每对 ( Vi , Ii ) do
                        {
                            gen('if ' t '=' Vi 'goto' Ii ) ;
                        }
                       gen( 'goto' I ) ;  // 跳转到 default S
                         label (next); }
```

值-标号对加入队列q

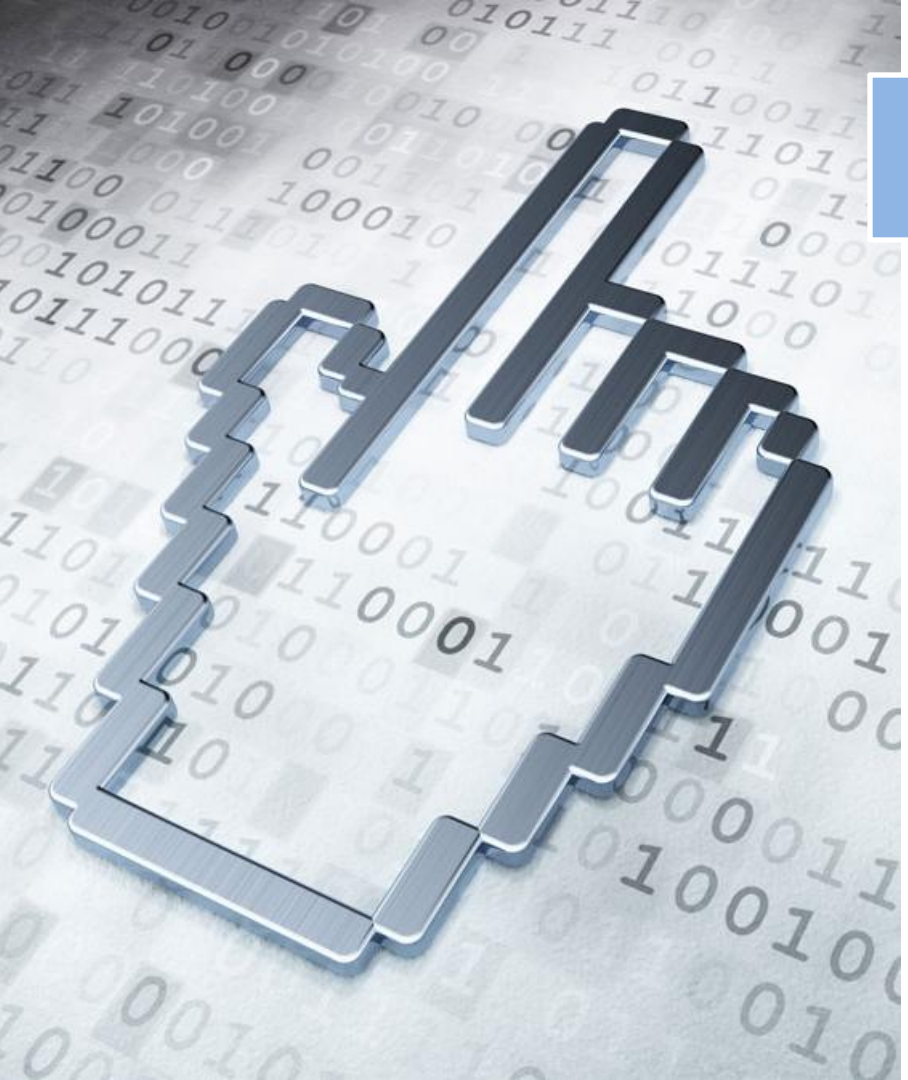


增加一种case指令

```
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



本章内容

6.7 过程调用的翻译

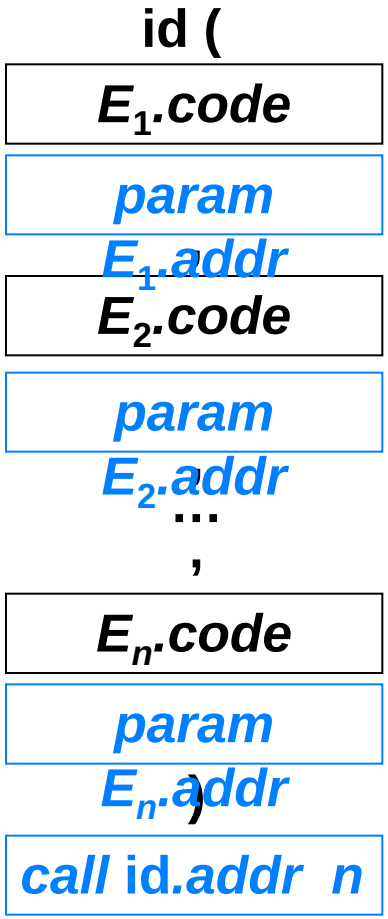
- 文法
  - ***S*** → **call** id (***Elist***)

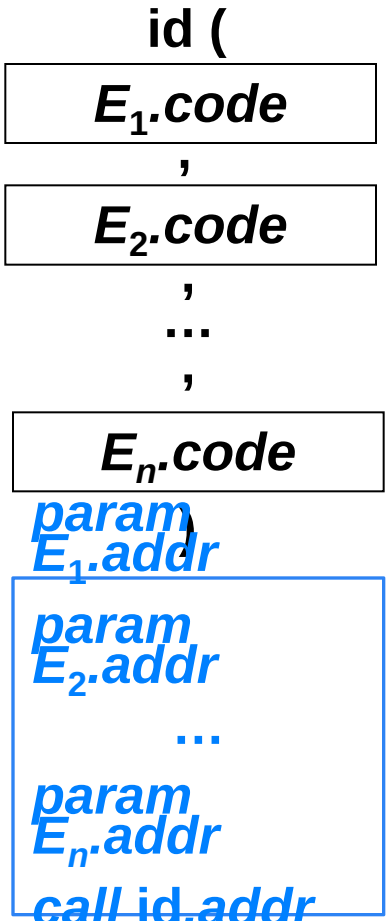
    Elist → ***Elist***, ***E***

    Elist → ***E***

过程调用语句的代码结构

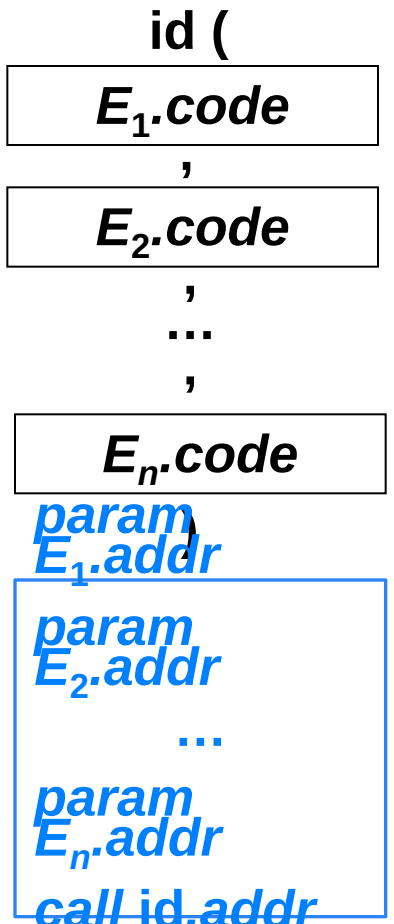
id(E_1, E_2, … , E_n)





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* → call id (*Elist*)

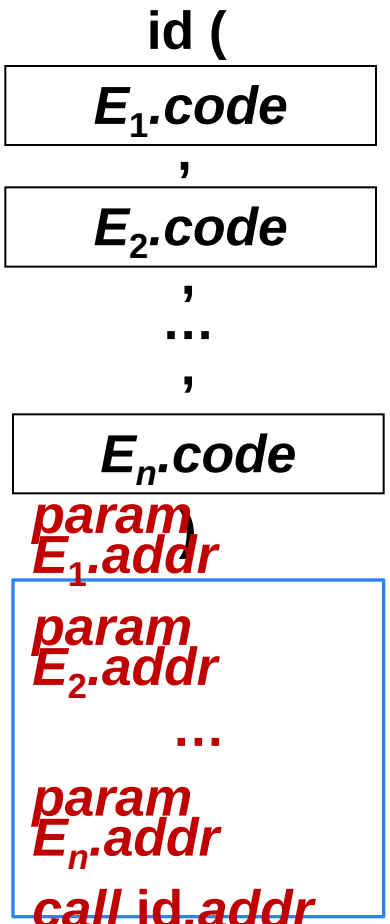
  { *n*=0;
  *for q*中的每个*t do*
  { *gen*(‘*param*’ *t*);
  n = *n*+1 ;
  }
  gen(‘*call*’ id.*addr* ‘,’ *n*);
  }

- *Elist* → *E*

  { 将*q*初始化为只包含*E.addr*; }

- *Elist* → *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

实验二指导

- 语义检查的核心任务是收集各种语法成分的类型信息，并进行类型匹配检查。
  - 类型的内部表示 ：P64 3.2.4类型表示
    - 基本数据类型、数组和结构体
  - 类型的计算：综合属性或继承属性
  - 类型等价性：名字等价或结构等价
- 声明语句翻译的核心任务是收集标识符的类型信息并存入符号表
  - 不同种别的标识符可以建立不同的符号表
  - 符号表的数据结构 P60 3.2.2 符号表

实验二指导

- 语义检查的总体思想：类似递归的预测分析
  - 先构建语法分析树
  - 自顶向下深度优先遍历语法分析树，根据内部节点，分别调用对应的非终结符处理函数，按照对应的产生式进行解析，同时计算需要的属性值，或者进行类型检查。
  - 为非终结符编写解析函数
    - 函数计算并返回综合属性：如EXP的数据类型
    - 将继承属性作为函数参数：如声明语句中，ID的类型由父辈节点的左部符号specifier确定。

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）

```
EXP     ID
EXP     ID LP  Args RP
EXP     ID LP  RP
{
  //在符号表中查找ID，若未找到，则未声明
}
```

语义分析中的错误检测

- 变量或函数未经声明就使用 （表达式/函数调用语句翻译）
- 变量或函数名重复声明 （变量/函数声明语句翻译）

```
VarDec      ID
FunDec      ID LR VarList RP
                | ID  LP RP
{
      //在符号表中查找ID，若找到了，则为重复声明
}
```

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配

```
Dec → VarDec ASSIGNOP Exp
Exp → Exp ASSIGNOP Exp
{
    // 收集 ASSIGNOP 两侧的非终结符的 数据类型，并进行类型比较
}
```

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配
- 赋值号左边出现一个只有右值的表达式

```
Exp → Exp ASSIGNOP Exp
{
    //根据左值的可能形式，使用启发性规则对EXP的子
   结点进行判断
}
```

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配
- 赋值号左边出现一个只有右值的表达式
- 操作数类型不匹配或操作数类型与操作符不匹配

Stmt →IF LP Exp RP Stmt|IF LP Exp RP Stmt ELSE Stmt

Stmt →WHILE LP Exp RP Stmt

Exp →NOT Exp | Exp AND Exp | Exp OR Exp | MINUS Exp | Exp RELOP Exp

Exp → Exp PLUS Exp| → Exp MINUS Exp | Exp STAR Exp | Exp DIV Exp

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配
- 赋值号左边出现一个只有右值的表达式
- 操作数类型不匹配或操作数类型与操作符不匹配
- return语句的返回类型与函数定义的返回类型不匹配

ExtDef → Specifier FunDec { CompSt.type = Specifier.type } CompSt

CompSt → LC DefList { StmtList.type = CompSt.type } StmtList RC

StmtList → {Stmt.type = StmtList.type }Stmt

Stmt →RETURN Exp SEMI { if Exp.type = Stmt.type ? }

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配
- 赋值号左边出现一个只有右值的表达式
- 操作数类型不匹配或操作数类型与操作符不匹配
- return语句的返回类型与函数定义的返回类型不匹配
- 函数调用时形参的数目或类型不匹配

```
Exp → ID LP Args RP
Exp → ID LP RP
{ //在函数表中查找ID可知函数的形参数目和类型，解析Args可知实参数目和类型，然后进行比较。}
```

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配
- 赋值号左边出现一个只有右值的表达式
- 操作数类型不匹配或操作数类型与操作符不匹配
- return语句的返回类型与函数定义的返回类型不匹配
- 函数调用时形参的数目或类型不匹配
- 对非数组型变量使用数组访问操作符，数组访问[]中出现非整数

Exp → Exp LB Exp RB { if Exp.type = 整型 ? }

语义分析中的错误检测

- 变量或函数未经声明就使用（表达式/函数调用语句翻译）
- 变量或函数名重复声明（变量/函数声明语句翻译）
- 赋值号两边的表达式类型不匹配
- 赋值号左边出现一个只有右值的表达式
- 操作数类型不匹配或操作数类型与操作符不匹配
- return语句的返回类型与函数定义的返回类型不匹配
- 函数调用时形参的数目或类型不匹配
- 对非数组型变量使用数组访问操作符，数组访问[]中出现非整数
- 对普通变量使用函数调用操作符

Exp →ID LP RP | ID LP Args RP
{ //在符号表中查找ID，是否函数？ }

语义分析中的错误检测

- 结构体类错误
  - 对非结构体型变量使用”.”操作符
  - 结构体域名重复定义，或在定义时对域进行了初始化
  - 结构体的名字与前面定义过的结构体或变量的名字重复
  - 使用未定义过的结构体来定义变量

```
Exp → Exp DOT ID
{ //解析Exp的子结点，收集Exp.type，是否结构体？}
```

```
VarDec → ID { //若是声明结构体中的域，要检查是否重复定义，需要记录是哪种声明使用了这个产生式，函数声明/定义？结构体声明？变量声明？}
Dec → VarDec ASSIGNOP Exp{ //若是声明结构体的域，则不合法 }
```

```
OptTag → ID { //在符号表中查找ID，若找到则重名 }
```

```
Tag → ID { //在符号表中查找ID，若未找到或种别不是结构体，则未定义 }
```

本章小结

- 声明语句的翻译
  - 变量声明的翻译
  - 数组声明的翻译
- 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

课程主要内容

1．绪论	（2学时）
2. 语言及其文法	（2学时）
3．词法分析	（3学时）
4．语法分析	（9学时）
5．语法制导翻译	（6学时）
6．中间代码生成	（7学时）
7．运行时的存贮组织	（3学时）
8．代码优化	（6学时）
9．代码生成	（2学时）

结束