第八章：语义分析和中间代码生成

语法制导翻译的延伸

- SDT
  - 实现机制（第7章）
    - 如何扩展语法分析栈
    - 如何将SDT中的抽象定义式改写成具体执行的栈操作
    - 如何扩展递归下降过程
    - 如何修改L属性定义的SDT，使之适合bottom_up翻译
  - 设计具体语法结构的SDT（第8章）
    - 声明语句的SDT
    - 赋值语句的SDT
    - 控制流语句的SDT
    - 过程调用语句的SDT

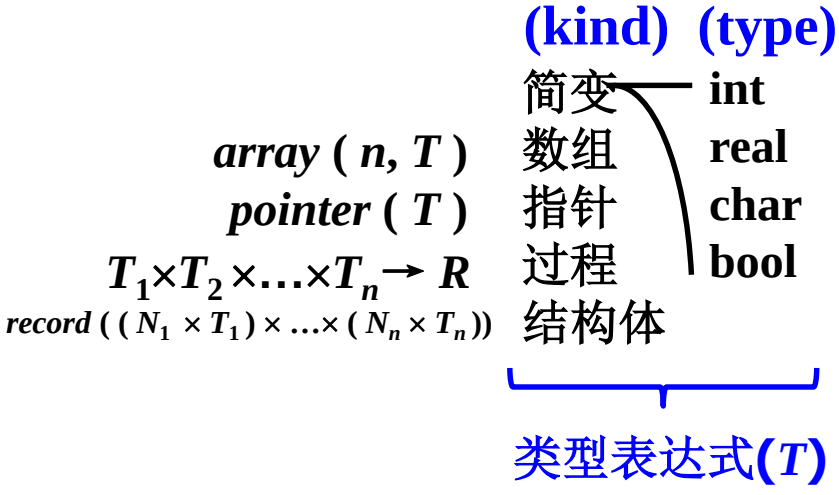
提纲

8.1 声明语句的翻译
8.2 赋值语句的翻译
8.3 控制流语句的翻译
8.4 代码回填
8.5 switch语句的翻译
8.6 过程调用语句的翻译

声明语句的翻译

- 主要任务

分析所声明id的性质、类型和地址，在符号表中为id建立一条记录



- 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）
- 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

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*

翻译的主要任务：类型表达式和地址

① $P \rightarrow D$

② $D \rightarrow T$ id; D

③ $D \rightarrow \varepsilon$

④ $T \rightarrow BC$

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

⑤ $T \rightarrow \uparrow T_1$

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

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

⑧ $C \rightarrow \varepsilon$

⑨ $C \rightarrow$ [num]C_1

变量	作用
offset	下一个可用的相对地址

$$offset_{i+1} = offset_i + T_i.width$$

变量声明语句的*SDT*

翻译的主要任务：类型表达式和地址

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 $T \rightarrow BC$

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

⑤ $T \rightarrow \uparrow T_1$

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

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

⑧ $C \rightarrow \varepsilon$

⑨ $C \rightarrow$ [num]C_1

变量	作用
offset	下一个可用的相对地址

$$offset_{i+1} = offset_i + T_i.width$$

变量声明语句的*SDT*

翻译的主要任务：类型表达式和地址

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 $T \rightarrow BC$

⑤ $T \rightarrow \uparrow T_1$

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

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

⑧ $C \rightarrow \varepsilon$

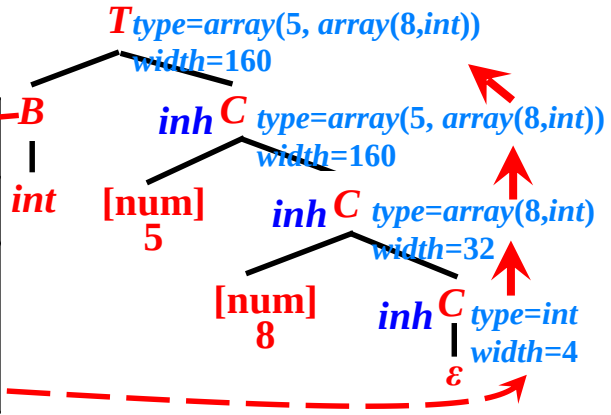
⑨ $C \rightarrow$ [num]C_1

例：int[5][8]

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

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



变量声明语句的*SDT*

翻译的主要任务：类型表达式和地址

① $P \rightarrow \{ offset = 0 \} D$

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

③ $D \rightarrow \varepsilon$

④ $T \rightarrow BC$

⑤ $T \rightarrow \uparrow T_1$

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

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

⑧ $C \rightarrow \varepsilon$

⑨ $C \rightarrow$ [num]C_1

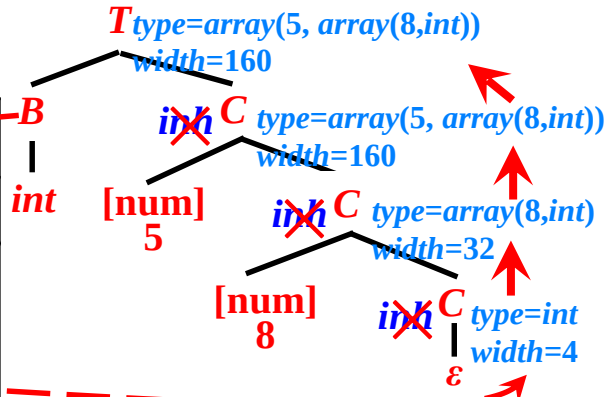
例：int[5][8]

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

右结合

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



变量声明语句的SDT

翻译的主要任务：类型表达式和地址

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;* }

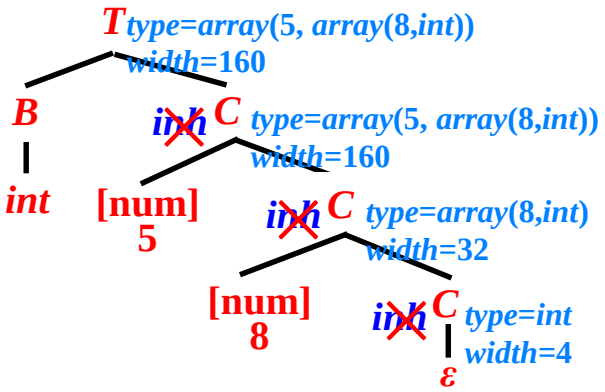
⑧ $C \rightarrow \varepsilon$ { *C.type = t ; C.width = w ;* }

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

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

例：int[5][8]

type=array(5, array(8,int))



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 $D \rightarrow T$ 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; }

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

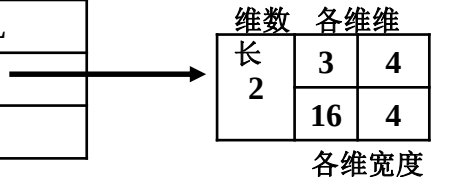
类型	类型表达式
int [3]	array (3, int)
int [2][3]	array (2, array(3,int))

符号表的组织——数组

- 例

```
int abc;
int i;
int myarray[3][4];
```

名字	基本属性				扩展属性
	种属	类型	地址	扩展属性指针	
abc	变量	int	0	NULL	
i	变量	int	4	NULL	
myarray	数组	int	8		
…	…				

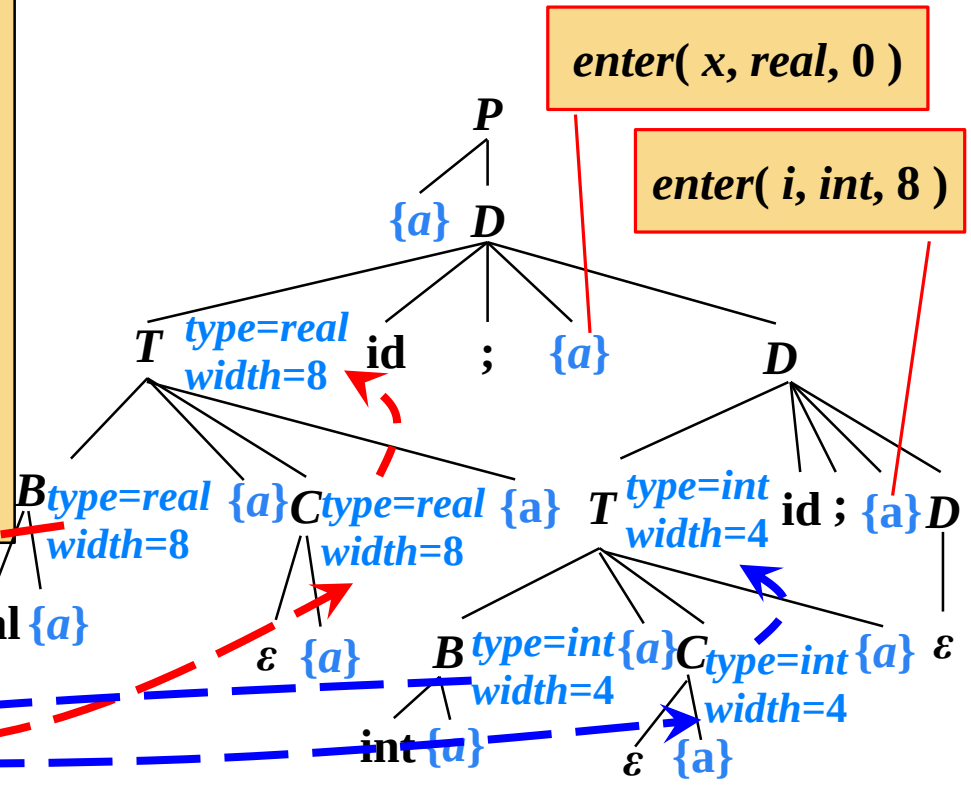


例: “real x; int i;”的语法制导翻译

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;* }

offset = 12

t = *int*
w = 4



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*; }

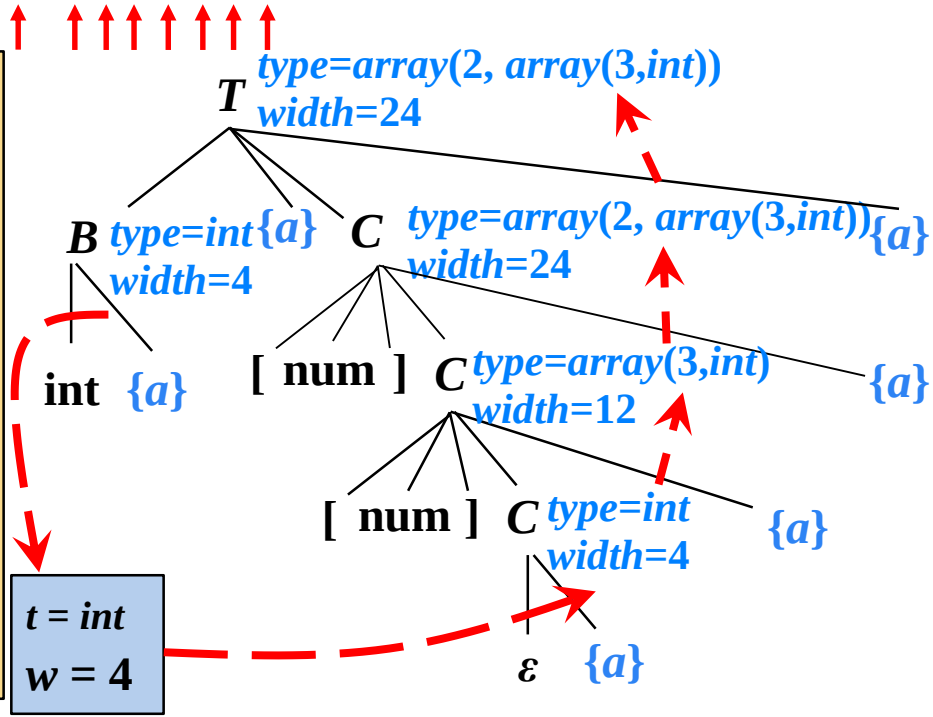
⑤$T \rightarrow \uparrow T_1$ { *T.type* = *pointer*(T_1.*type*); *T.width* = 4; }

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

8.1 声明语句的翻译
8.2 赋值语句的翻译
8.3 控制流语句的翻译
8.4 代码回填
8.5 switch语句的翻译
8.6 过程调用语句的翻译

赋值语句翻译的任务

- 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
      - $t_1 = a + b$
      - $t_2 = t_1 * c$
      - $x = t_2$

简单赋值语句的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

$S \rightarrow$ 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 = ''; }

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); }

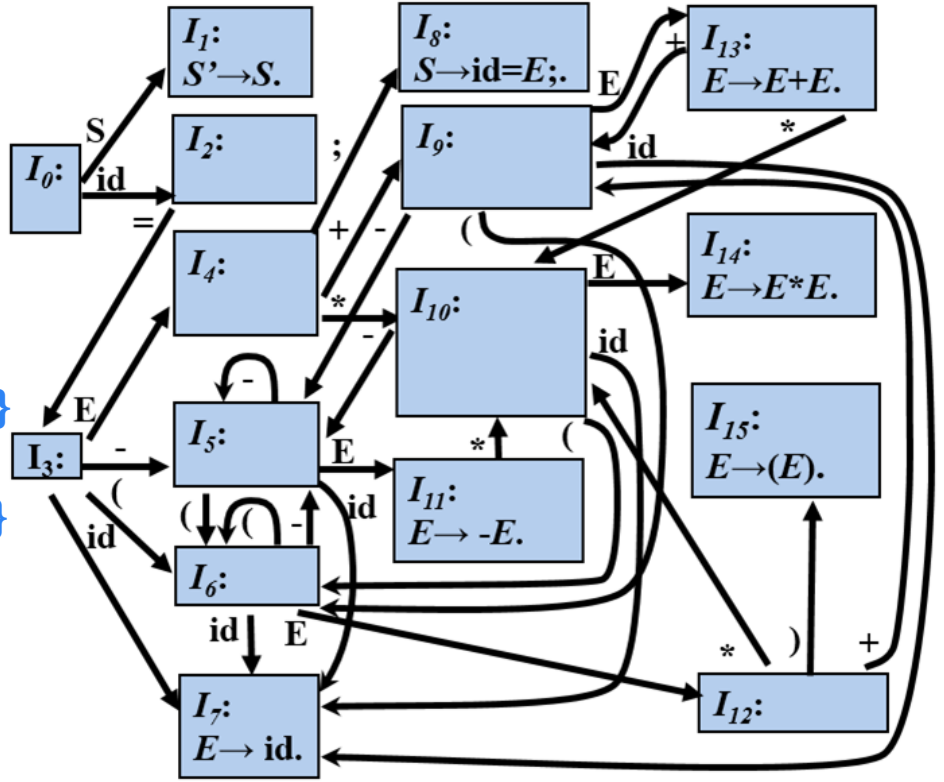
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }

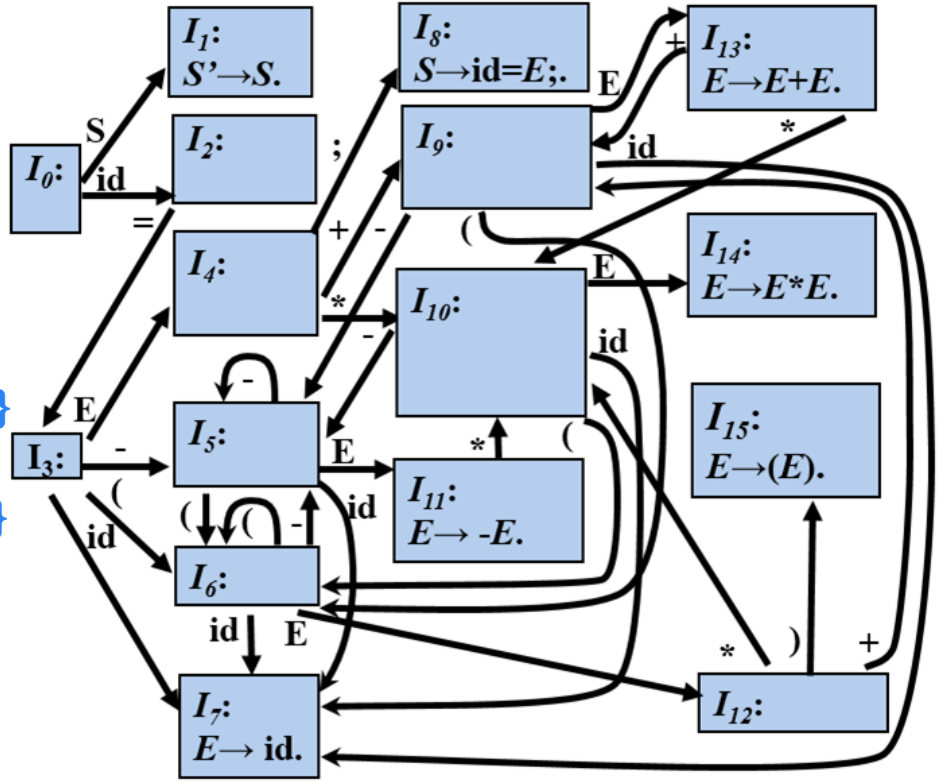
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }

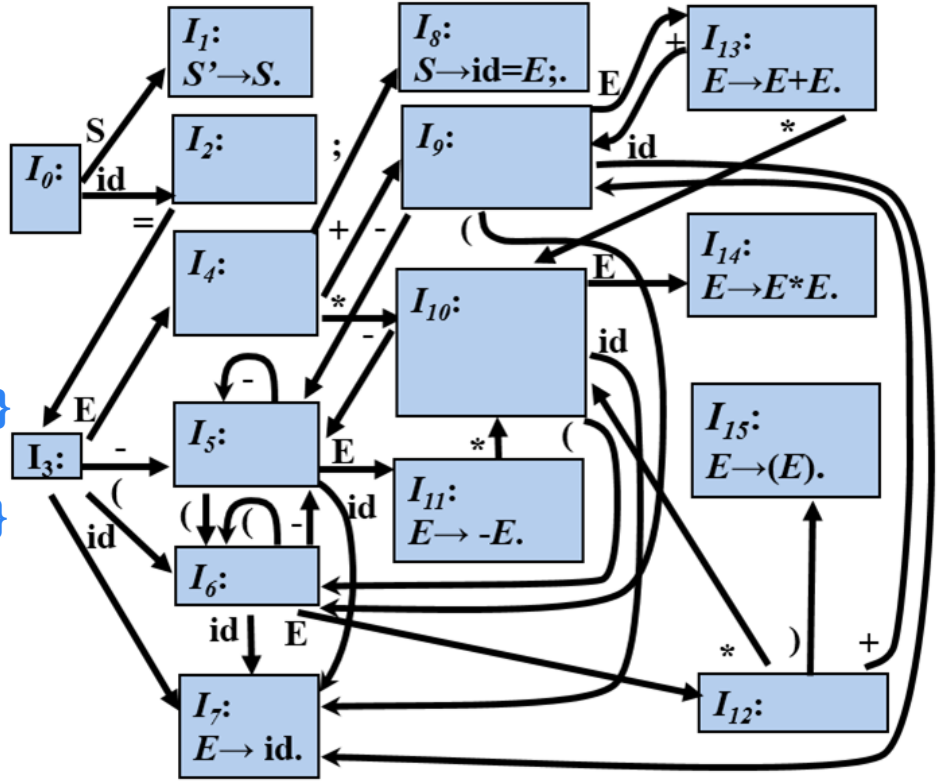
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }

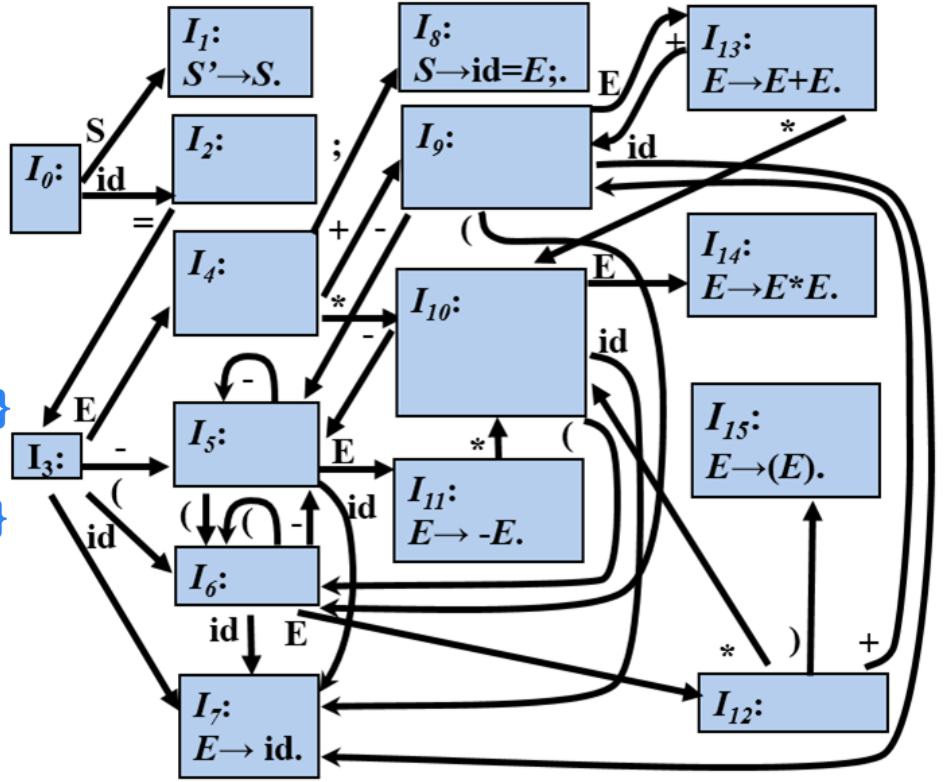
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }

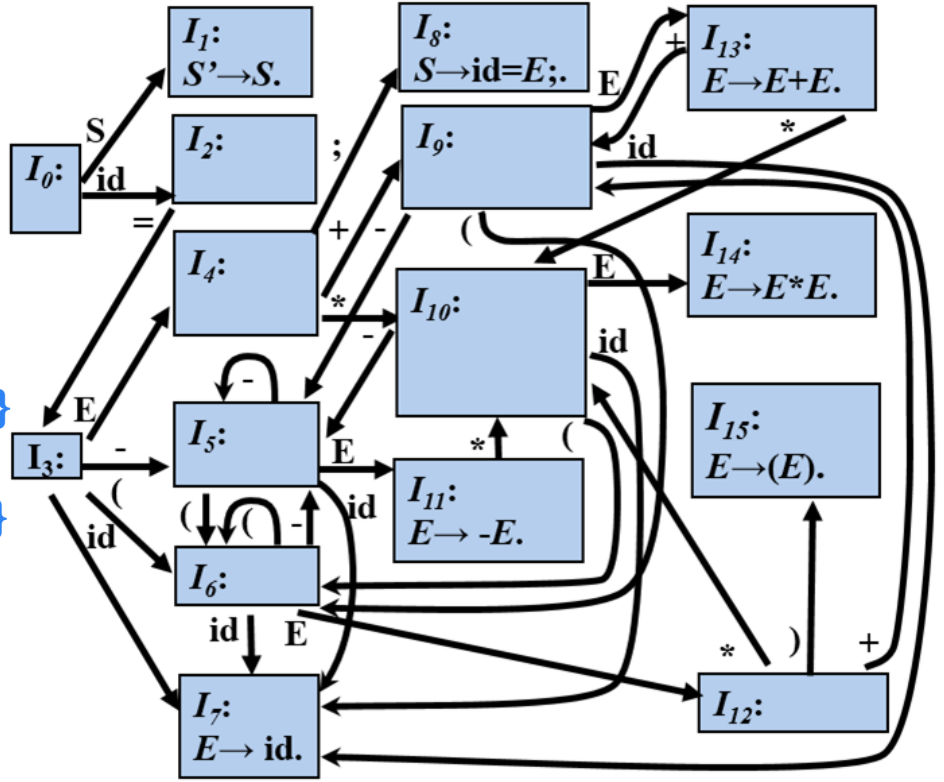
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }

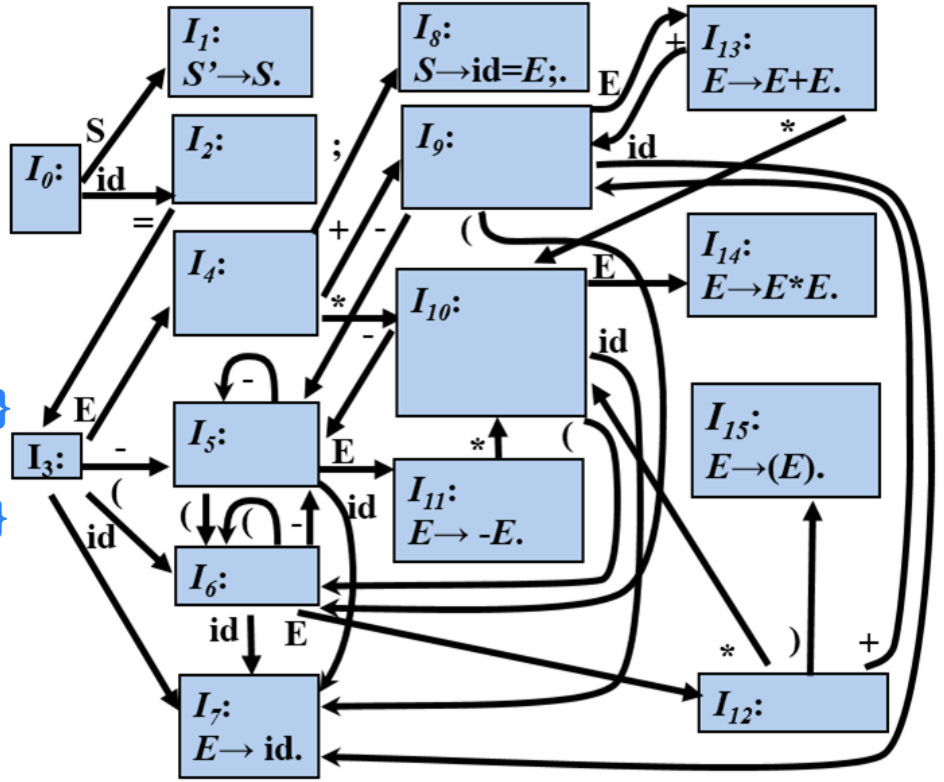
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }

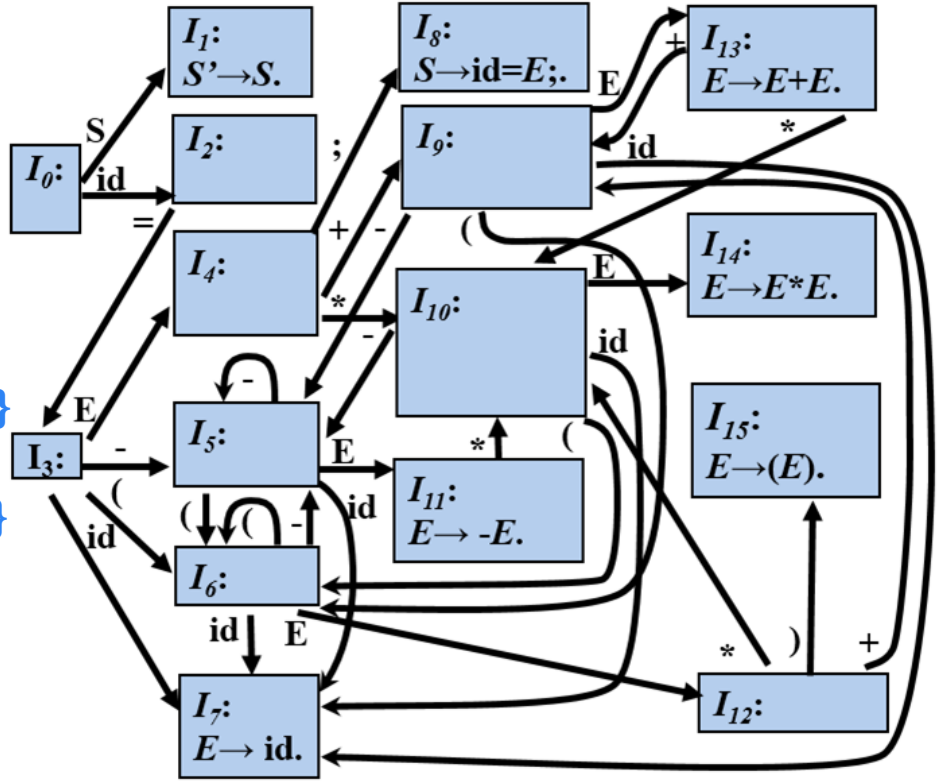
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

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); }*

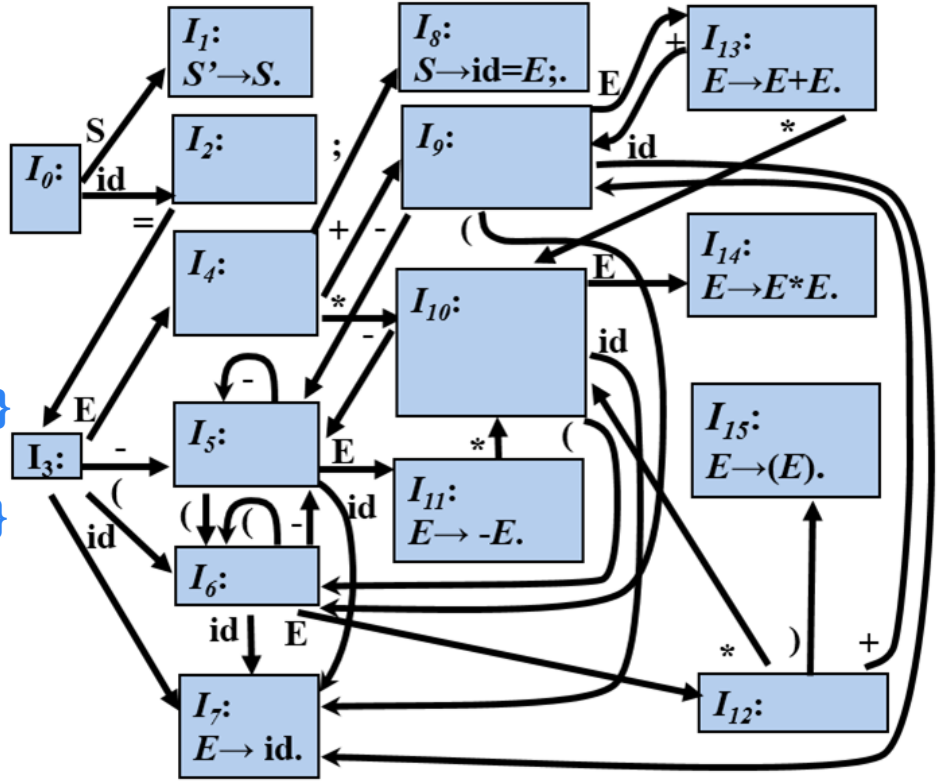
④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

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



$t_1 = a + b$

$t_2 = t_1 * c$

$x = t_2$

含有数组引用的赋值语句的翻译

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

含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
    - $t_1 = i * 4$
    - $t_2 = a[t_1]$
    - $c = t_2$

含有数组引用的赋值语句的翻译

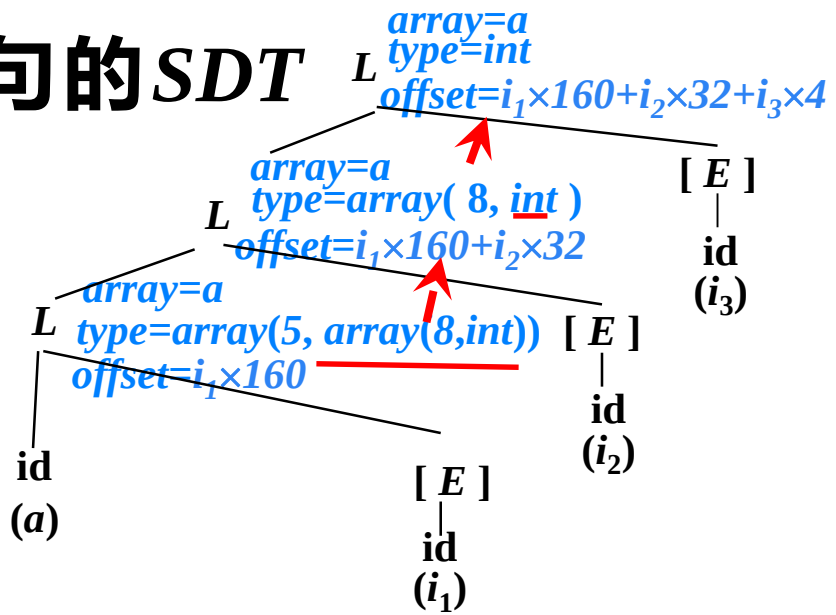
- 例2

  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

含有数组引用的赋值语句的SDT



- 赋值语句的基本文法

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

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

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

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

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，从而计算数组引用的偏移量
  - *L.array*：数组名在符号表的入口地址

含有数组引用的赋值语句的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

$S \rightarrow$ id = E;

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

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

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

$L \rightarrow$ id [E]{ $L.array$ = lookup(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

$t_1 = i_1 * 160$

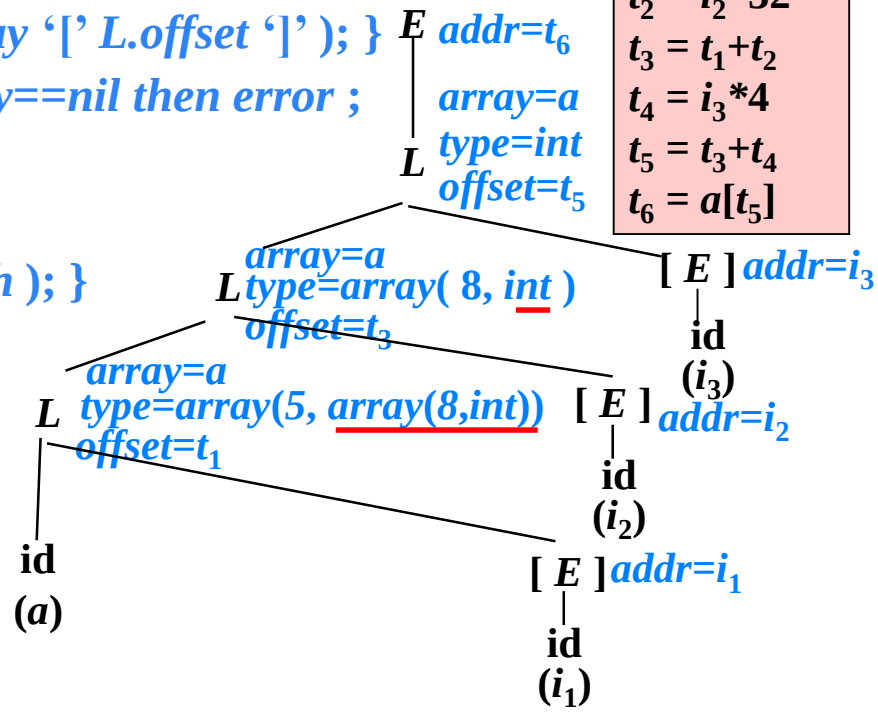
$t_2 = i_2 * 32$

$t_3 = t_1 + t_2$

$t_4 = i_3 * 4$

$t_5 = t_3 + t_4$

$t_6 = a[t_5]$



Exercise

1有文法G及其语法制导翻译如下所示(语义规则中的*和+分别是常规意义下的算术运算符):

$$E \rightarrow E^{(1)} \wedge T\{E.val = E^{(1)}.val * T.val\}$$

$$E \rightarrow T\{E.val = T.val\}$$

$$T \rightarrow T^{(1)} \# n\{T.val = T^{(1)}.val + n.val\}$$

$$T \rightarrow n\{T.val = n.val\}$$

则分析句子1 ∧ 2 ∧ 3 # 4的值为( )

A.10 B.34 C.14 D.54

则分析句子2 ∧ 3 # 4其值为( )。

A.10 B.21 C.14 D.24

Exercise

2用( )可以把a:=b+c翻译成四元式序列。
A.语法规则
B.词法规则
C.语义规则
D.等价变换规则

3中间代码生成时所依据的是( )。
A.语法规则
B.词法规则
C.语义规则
D.等价变换规则

Exercise

4以下说法不正确的是( )。

A.类型自身也有结构，用类型表达式来表示这种结构

B.基本类型不是类型表达式

C.类型名也是类型表达式

D.将类型构造符作用于类型表达式可以构成新的类型表达式

5数组元素的地址由两部分构成，一部分是基地址，另一部分是偏移量。（ ）

A.对　　B.错

6基地址通过查符号表即可获得。（ ）

A.对　　B.错

7设计数组引用的SDT的关键问题是：如何将地址计算公式和数组引用的文法关联起来。（ ）

A.对　　B.错

提纲

8.1 声明语句的翻译
8.2 赋值语句的翻译
8.3 控制流语句的翻译
8.4 代码回填
8.5 switch语句的翻译
8.6 过程调用语句的翻译

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;
```

用指令的标号标识一条三地址指令

  - 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B被翻译成由跳转指令构成的跳转代码

- *B.true*：是一个地址，该地址中存放了当*B*为真时控制流转向的指令的标号
- *B.false*：是一个地址，该地址中存放了当*B*为假时控制流转向的指令的标号

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.true*
  - *B.false*

  } 继承属性

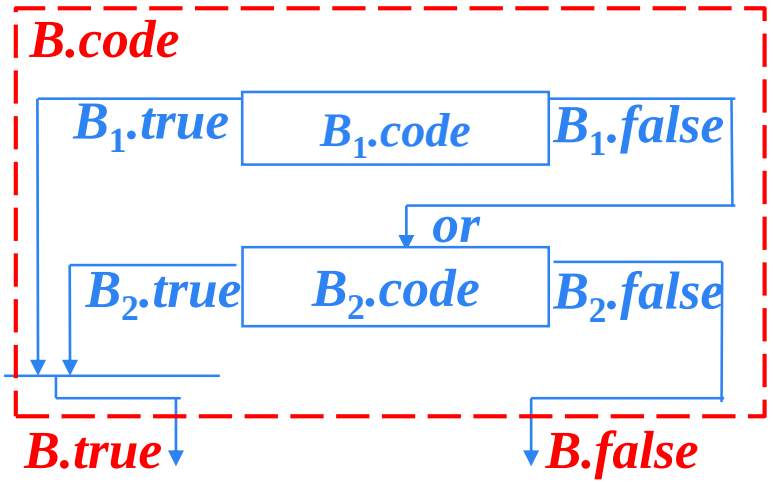
布尔表达式的SDT

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

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

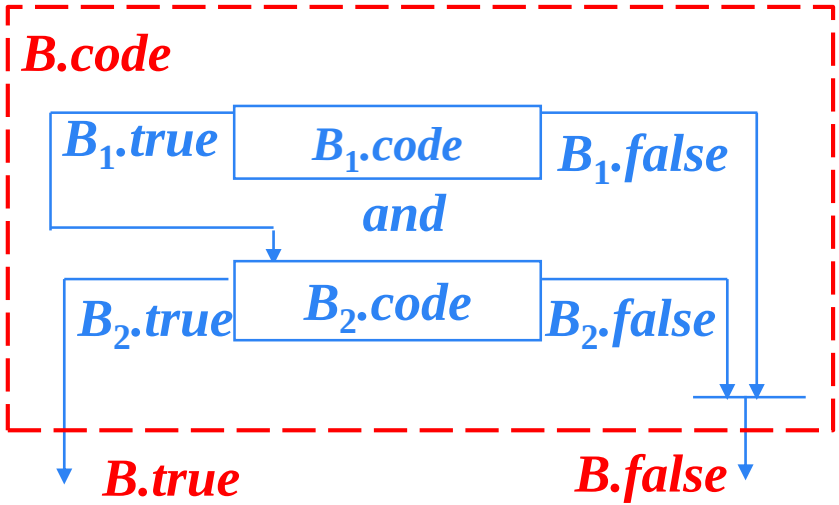
label(*L*): 将下一条三地址指令的标号赋给*L*

布尔表达式*B*被翻译成由跳转指令构成的跳转代码



布尔表达式的SDT

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

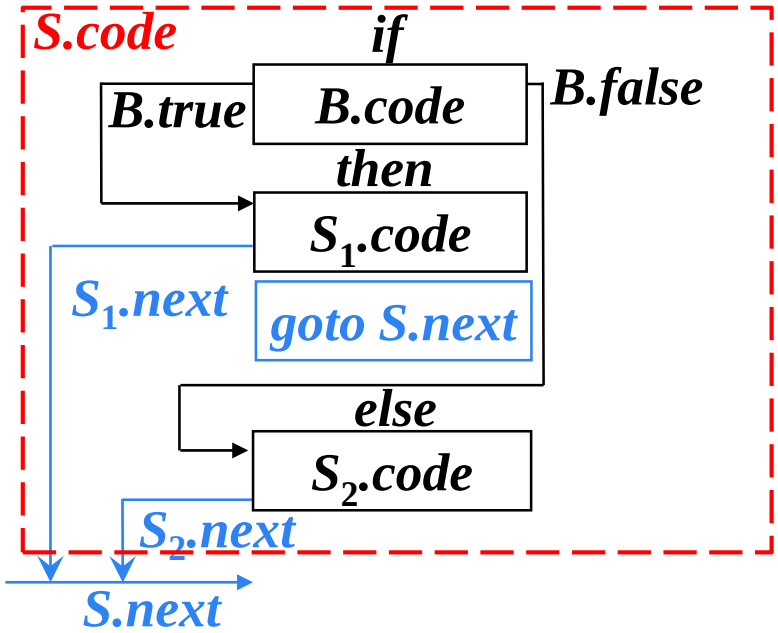


控制流语句的基本文法

- $P \rightarrow S$
- $S \rightarrow S_1 S_2$
- $S \rightarrow \text{id} = E \ ; \mid L = E \ ;$
- $S \rightarrow \text{if } B \text{ then } S_1$
  $\mid \text{if } B \text{ then } S_1 \text{ else } S_2$
  $\mid \text{while } B \text{ do } S_1$

控制流语句的代码结构

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



➢ 继承属性

- ➢ *S.next*：是一个地址，该地址中存放了紧跟在*S*代码之后的指令(*S*的后继指令)的标号
- ➢ *B.true*：是一个地址，该地址中存放了当*B*为真时控制流转向的指令的标号
- ➢ *B.false*：是一个地址，该地址中存放了当*B*为假时控制流转向的指令的标号

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

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

  $\{ label(S_1.next); S_2.next = S.next ; \} S_2$

label(*L*): 将下一条三地址指令的标号赋给*L*

- $S \rightarrow \mathbf{id} = E ; \mid L = E ;$ { *p = lookup(id.lexeme); if p == nil then error ;*
  S.code = E.code || gen(p '=' E.addr); }
  { *gen(L.array '[' L.offset ']' '=' E.addr); }*
- $S \rightarrow if\ B\ then\ S_1$

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

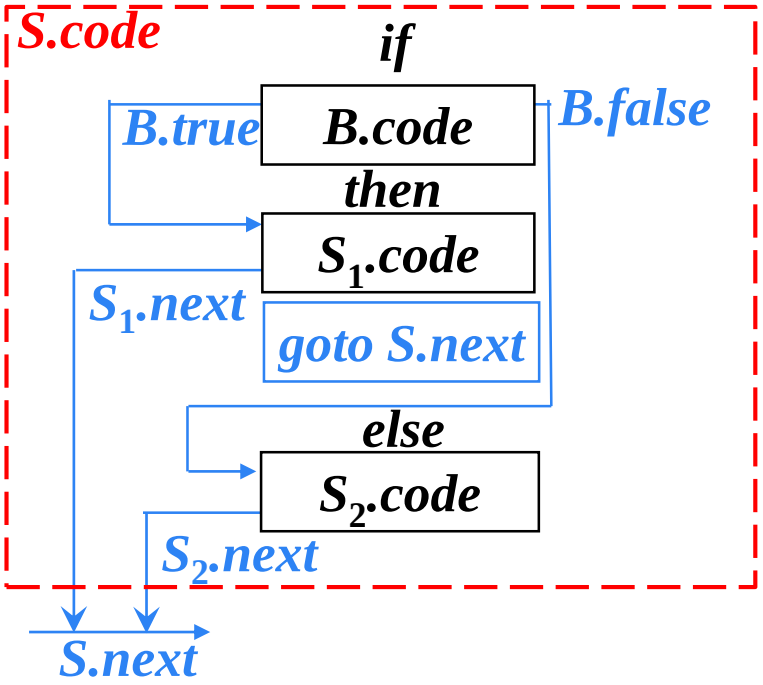
  $\mid while\ B\ do\ S_1$

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*

if-then-else语句的SDT

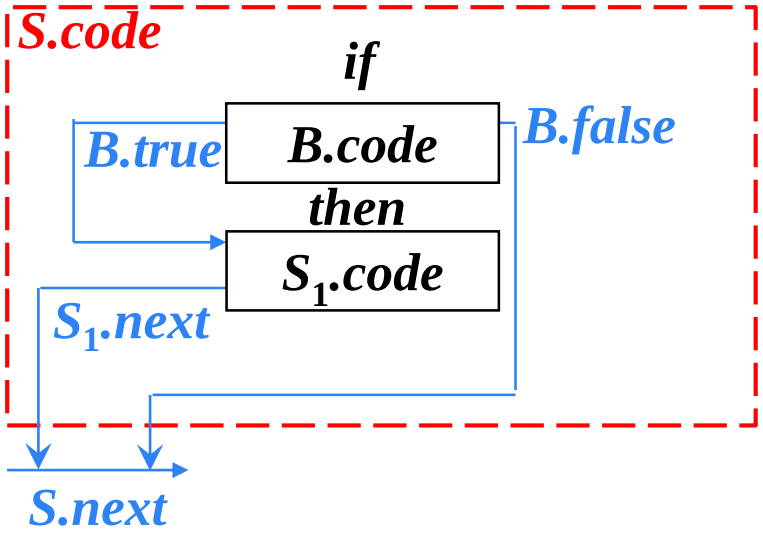
$S \rightarrow$ *if* B *then* S_1 *else* S_2



$S \rightarrow$ *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*

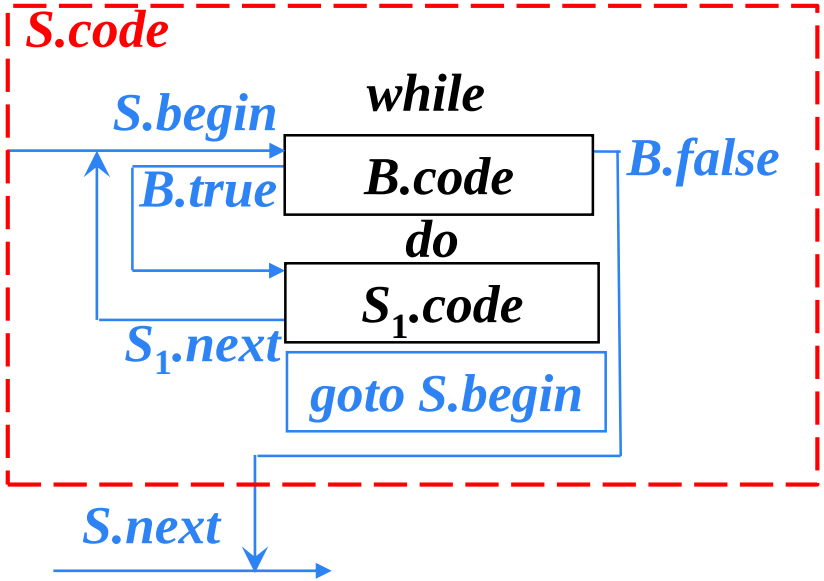
$S \rightarrow if\ B\ then\ S_1$



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

while-do语句的SDT

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



$S \rightarrow$ *while* {*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);* }

控制流语句SDT编写要点

- 分析每一个非终结符之前
  - 先计算继承属性
  - 再观察代码结构图中该非终结符对应的方框顶部是否有导入箭头。如果有，调用label( )函数
- 上一个代码框执行完不顺序执行下一个代码框时，生成一条显式跳转指令
- 有自下而上的箭头时，设置begin属性。且定义后直接调用label( )函数绑定地址

控制流语句SDT编写要点

- 分析每一个非终结符之前
  - 先计算继承属性
  - 再观察代码结构图中该非终结符对应的方框顶部是否有导入箭头。如果有，调用label( )函数
- 上一个代码框执行完不顺序执行下一个代码框时，生成一条显式跳转指令
- 有自下而上的箭头时，设置begin属性。且定义后直接调用label( )函数绑定地址

控制流语句的SDT

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

*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

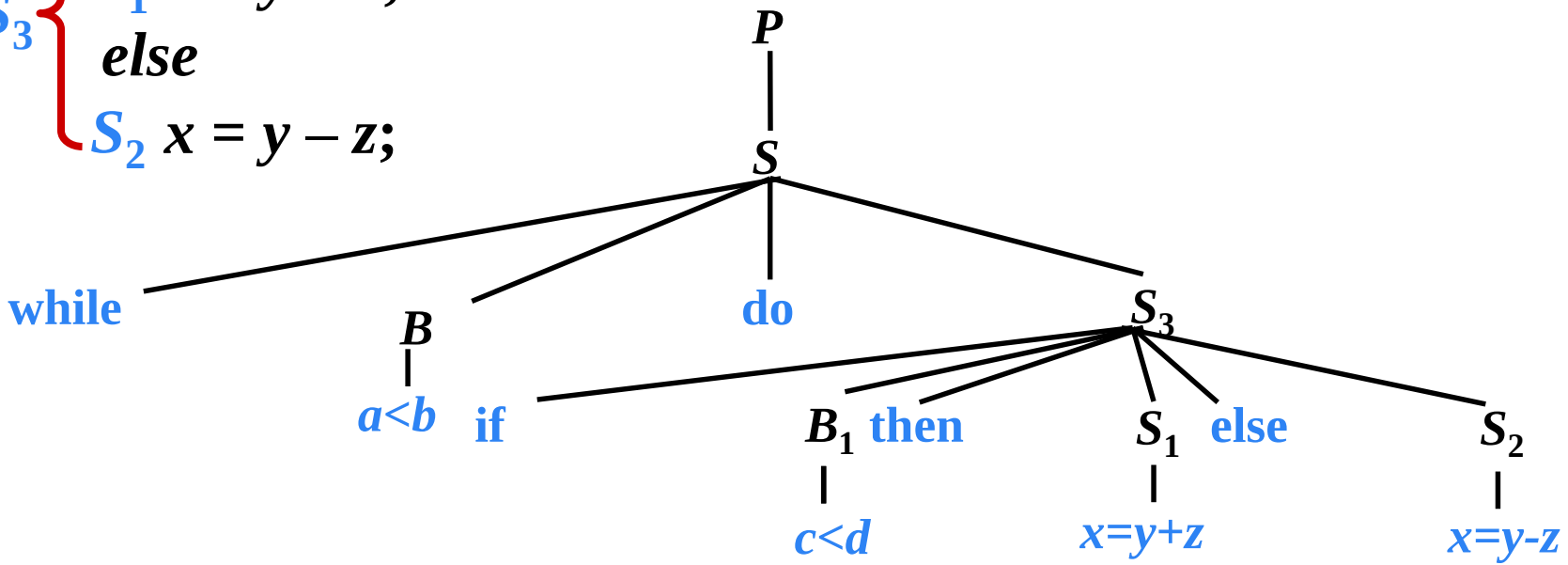
控制流语句的SDT

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

例

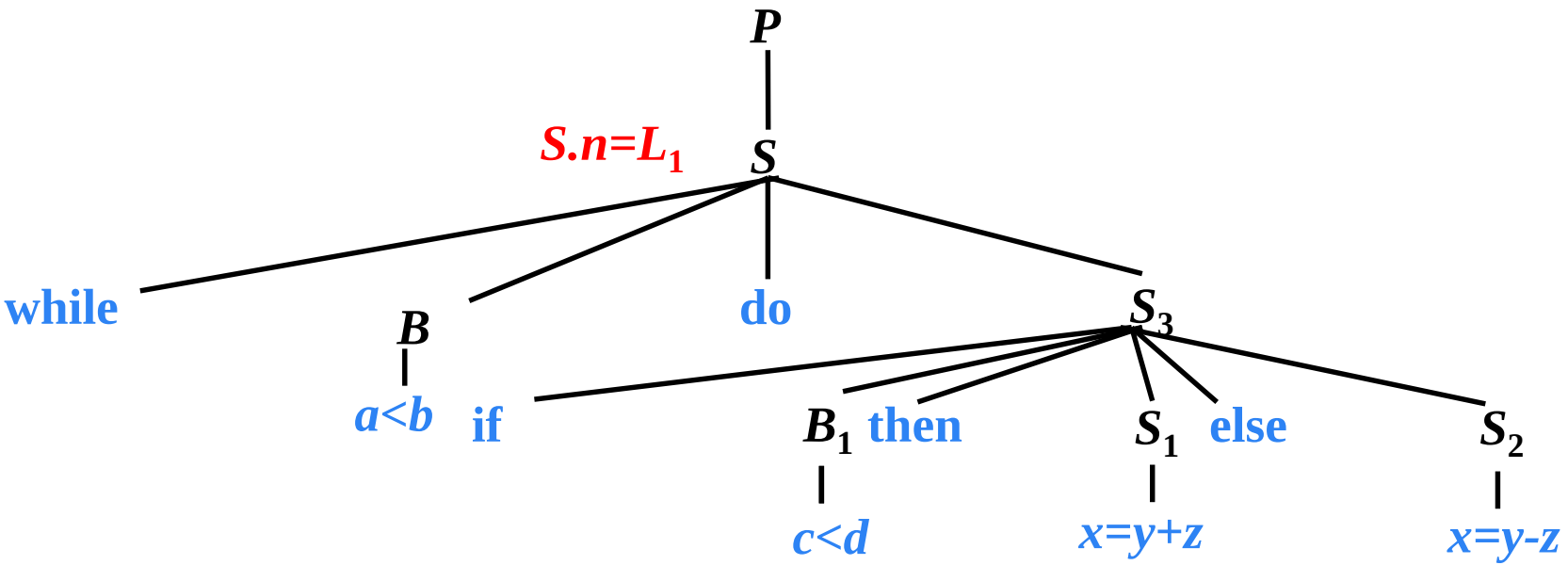
```
while a < b do
  if c ≤ d then
    S1 x = y + z;
  else
    S2 x = y – z;
```

B: `a < b`　　B_1: `c ≤ d`　　S_3: `if c ≤ d then S1 else S2`



例

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



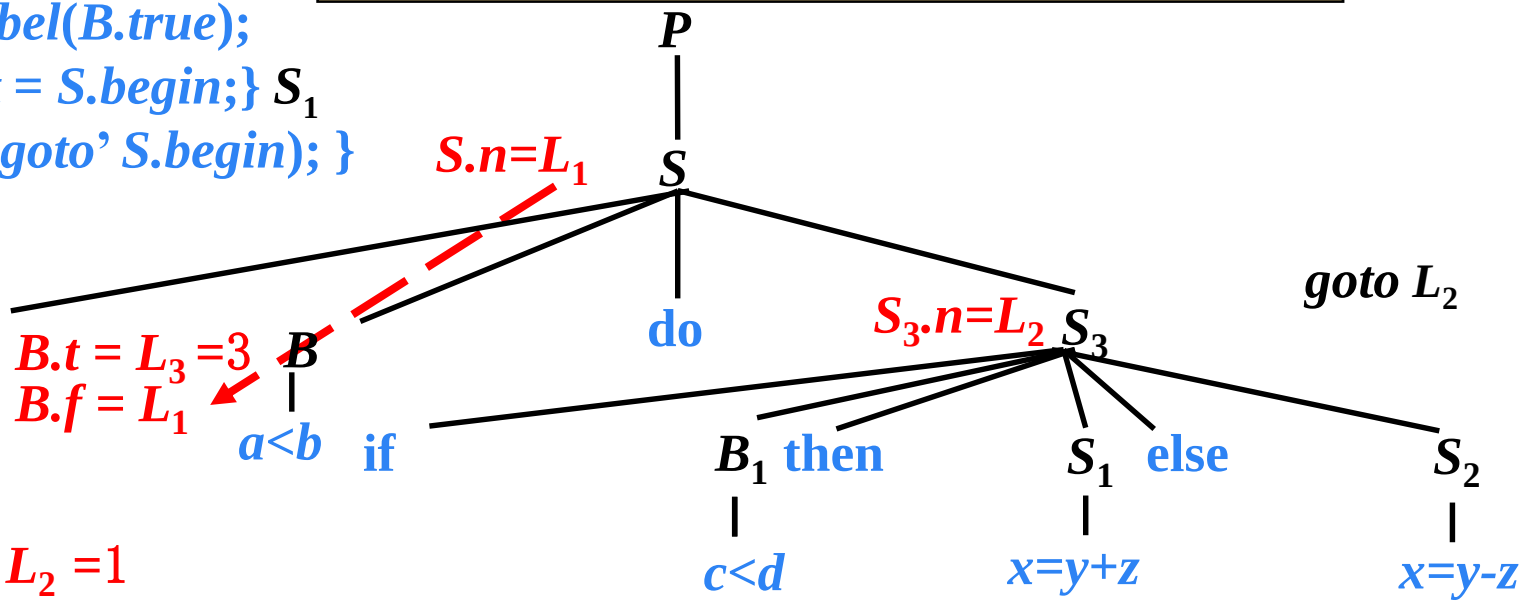
例

1: *if a < b goto* L_3

2: *goto* L_1

$S \rightarrow$ 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); }



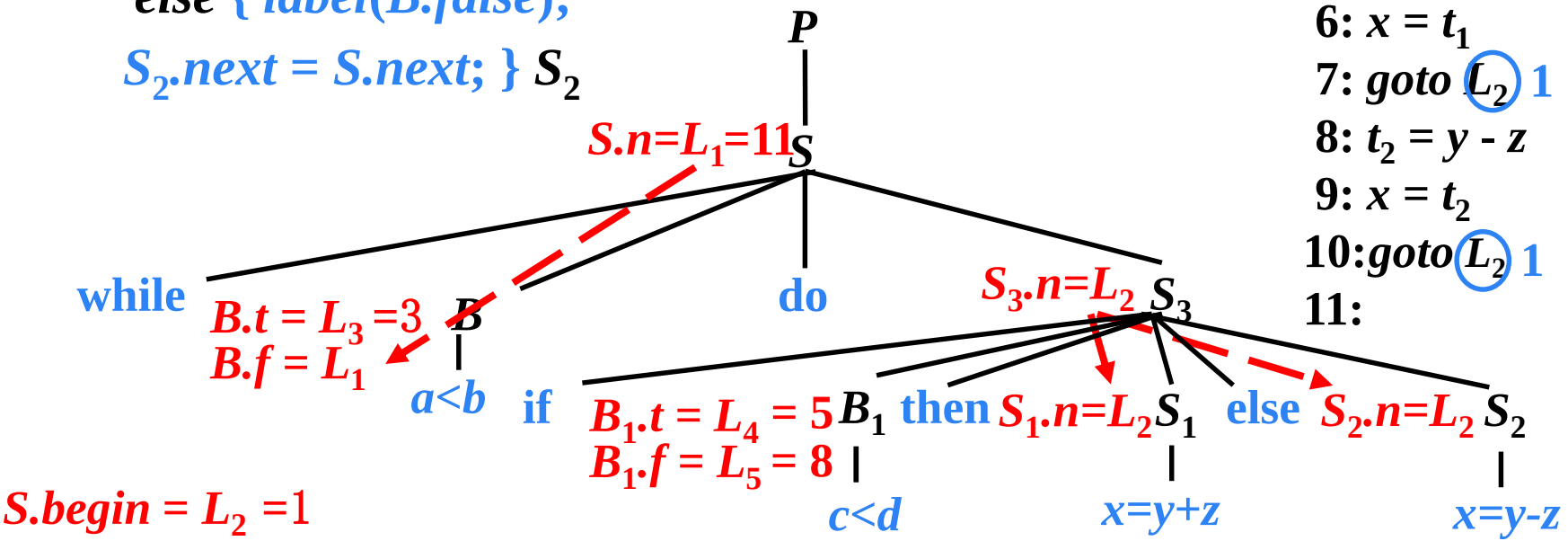
S.begin = L_2 =1

例

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

$S \rightarrow$ *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
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

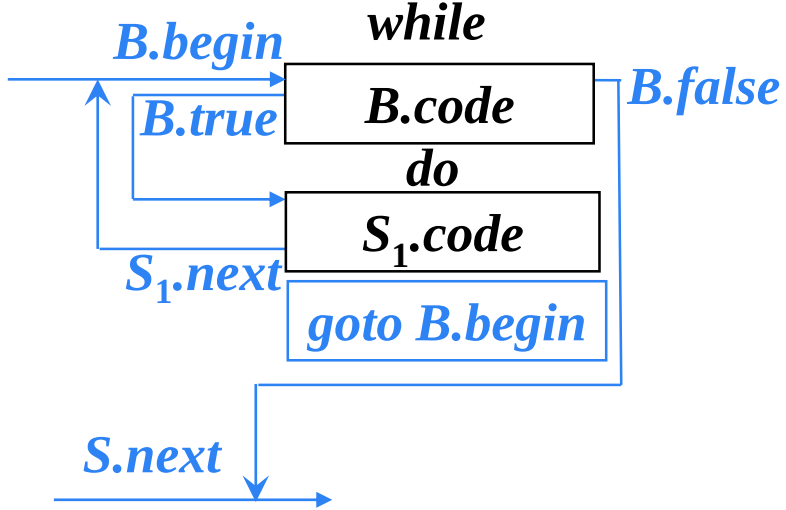
1: *if a* < *b goto* 3
2: *goto* 11
3: *if c* < *d goto* 5
4: *goto* 8
5: $t_1 = y + z$
6: $x = t_1$
7: *goto* 1
8: $t_2 = y - z$
9: $x = t_2$
10: *goto* 1
11:

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:

避免生成冗余的goto指令

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

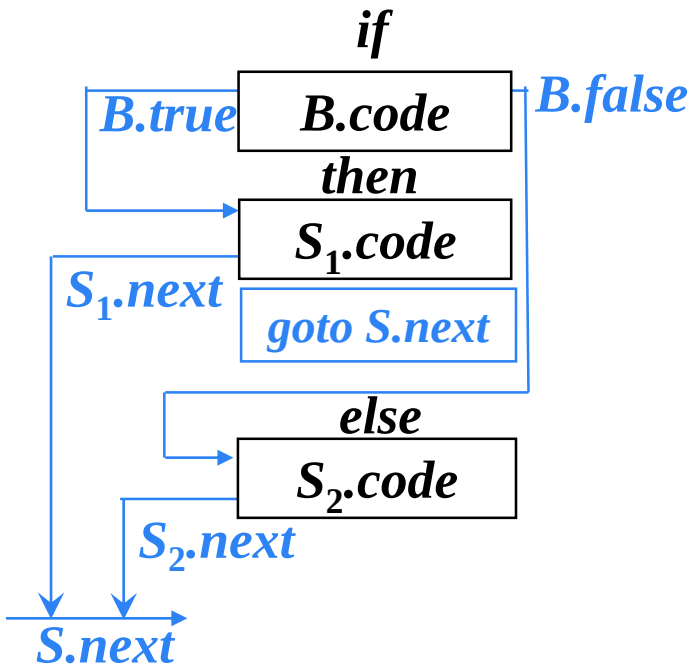
B.first → 1: *if a < b goto* 3	1: *ifFalse a < b goto* 11
2: *goto* 11	2:
S_1*.first* → 3: *if c < d goto* 5	3: *if c < d goto* 5
4: *goto* 8	4: *goto* 8
5: $t_1 = y + z$	5: $t_1 = y + z$
6: $x = t_1$	6: $x = t_1$
7: *goto* 1	7: *goto* 1
8: $t_2 = y - z$	8: $t_2 = y - z$
9: $x = t_2$	9: $x = t_2$
10: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



避免生成冗余的goto指令

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

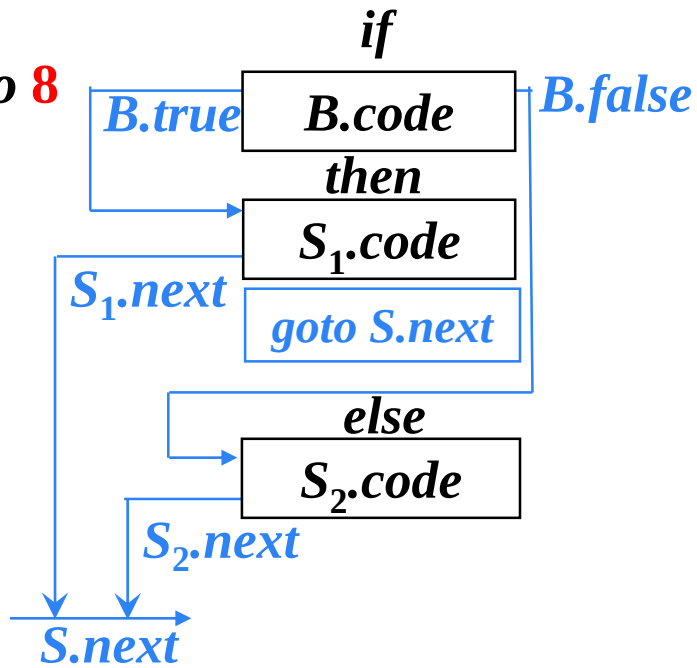
1: *if a < b goto* 3	1: *ifFalse a < b goto* 11
2: *goto* 11	2:
B.first → 3: *if c < d goto* 5	3: *if c < d goto* 5
4: *goto* 8	4: *goto* 8
S_1*.first* → 5: $t_1 = y + z$	5: $t_1 = y + z$
6: $x = t_1$	6: $x = t_1$
7: *goto* 1	7: *goto* 1
S_2*.first* → 8: $t_2 = y - z$	8: $t_2 = y - z$
9: $x = t_2$	9: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



避免生成冗余的goto指令

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



控制流语句翻译总结

- 控制流语句翻译的一个关键是确定跳转指令的目标标号
- 存在问题：生成跳转指令时，目标标号还不能确定
- 解决办法：生成一些临时变量用来存放标号，将临时变量的地址作为继承属性传递到标号可以确定的地方。也就是说，当目标标号的值确定下来以后再赋给相应的变量
  - 缺点：需要进行两遍处理
    - 第一遍生成临时的指令
    - 第二遍将指令中的临时变量的地址改为具体的标号，从而得到最终的三地址指令

提纲

8.1 声明语句的翻译
8.2 赋值语句的翻译
8.3 控制流语句的翻译
8.4 代码回填
8.5 switch语句的翻译
8.6 过程调用语句的翻译

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

布尔表达式的回填

- 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

布尔表达式的回填

为了处理跳转指令的列表地址，定义三个函数

- *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

{

backpatch(B_1.falselist, M.quad);

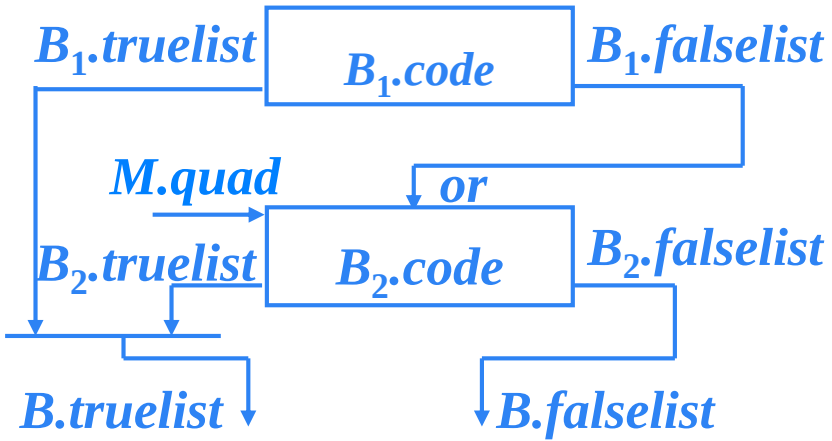
B.truelist= merge(B_1.truelist,B_2.truelist);

B.falselist = B_2.falselist ;

}

$M \rightarrow \varepsilon$

{ *M.quad = nextquad ;* }



$B \rightarrow B_1$ and B_2

$B \rightarrow B_1$ and M B_2

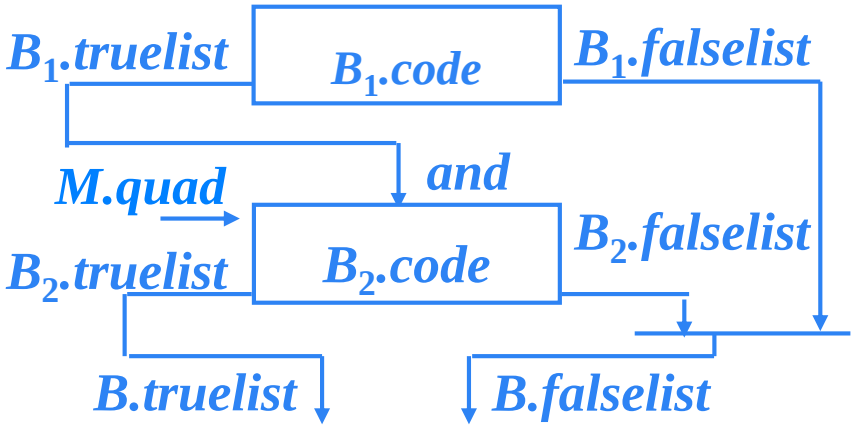
{

backpatch(B_1.truelist, M.quad);

B.truelist = B_2.truelist;

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

}



回填技术SDT编写要点

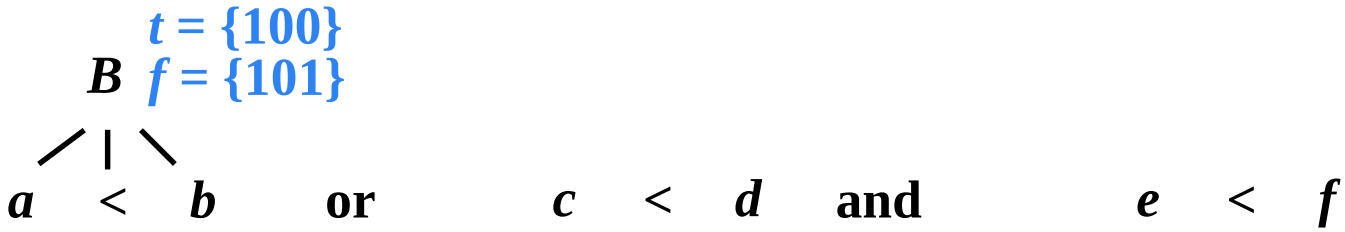
- 构造代码结构图
- 文法改造
  - 在list箭头指向的位置设置标记非终结符M
- 在产生式末尾的语义动作中
  - 计算综合属性
  - 调用backpatch ( )函数回填各个list

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 ;

}

$M \rightarrow \varepsilon$

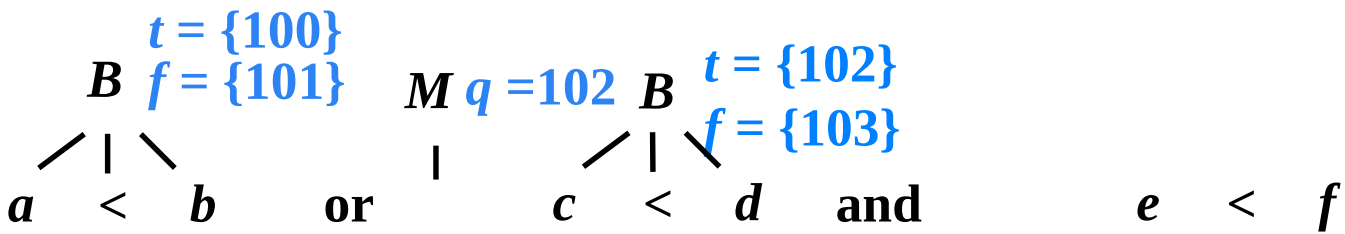
{ *M.quad = nextquad ;* }

100: *if a<b goto _*

101: *goto _*

102: *if c<d goto _*

103: *goto _*



例

$B \rightarrow B_1$ and M B_2

{

backpatch(B_1.truelist, M.quad);

B.truelist = B_2.truelist;

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

}

100: *if a<b goto _*

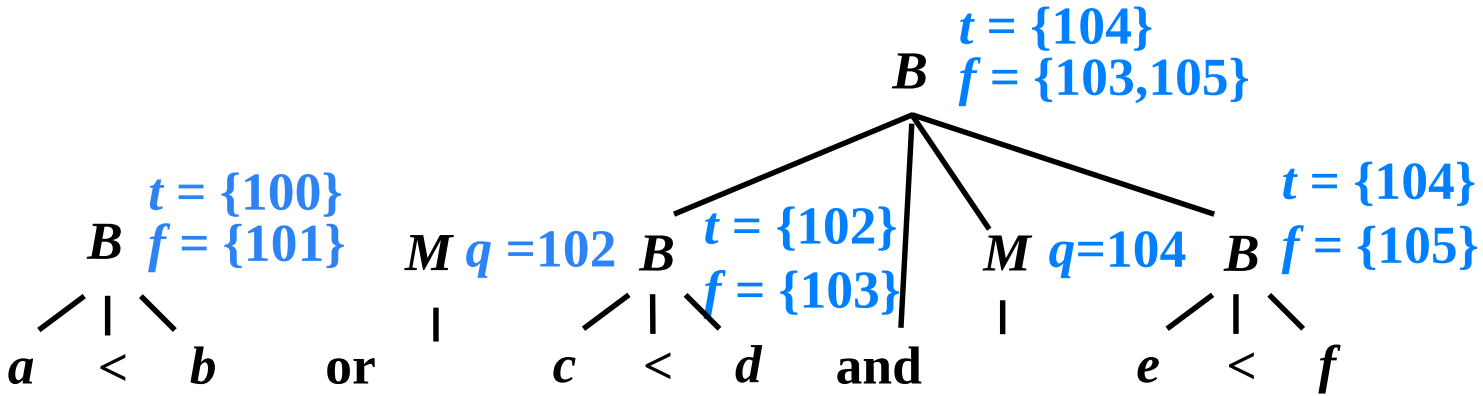
101: *goto _*

102: *if c<d goto* 104

103: *goto _*

104: *if e<f goto _*

105: *goto _*



例

$B \rightarrow B_1$ or M B_2

{

backpatch(B_1.falselist, M.quad);

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

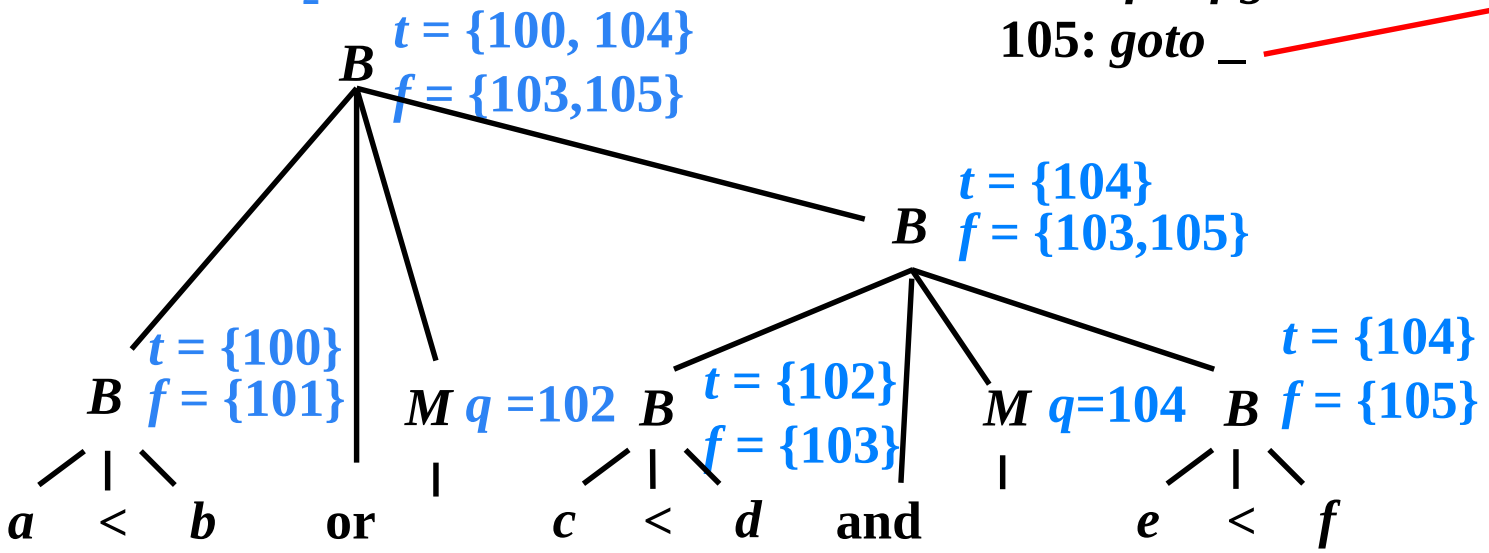
B.falselist = B_2.falselist ;

}

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

- 构造代码结构图
- 文法改造
  - 在list箭头指向的位置设置标记非终结符M
- 在产生式末尾的语义动作中
  - 计算综合属性
  - 调用backpatch ( )函数回填各个list

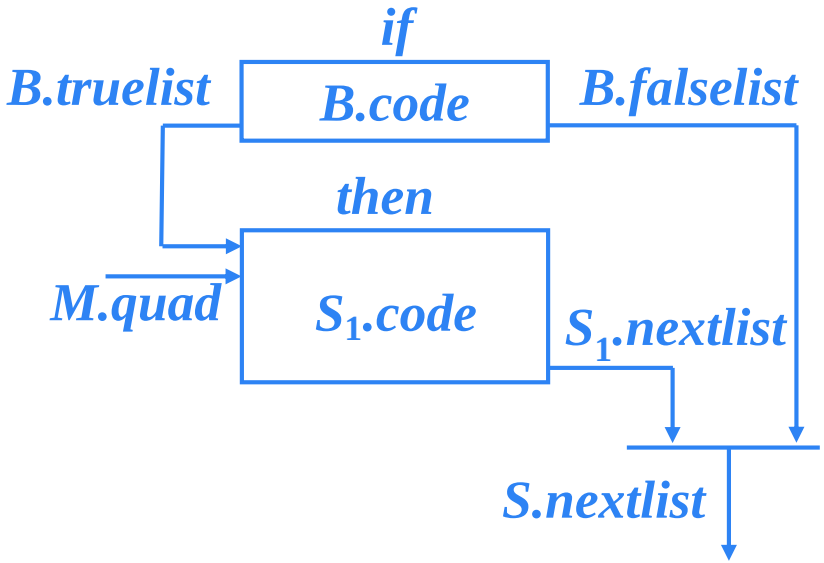
$S \rightarrow$ if B then M S_1

{

backpatch(B.truelist, M.quad);

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

}



$S \rightarrow$ if B then S_1 else S_2

- 构造代码结构图
- 文法改造
  - 在list箭头指向的位置设置标记非终结符M
- 在产生式末尾的语义动作中
  - 计算综合属性
  - 调用backpatch ( )函数回填各个list

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

{

backpatch(B.truelist, M_1.quad);

backpatch(B.falselist, M_2.quad);

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

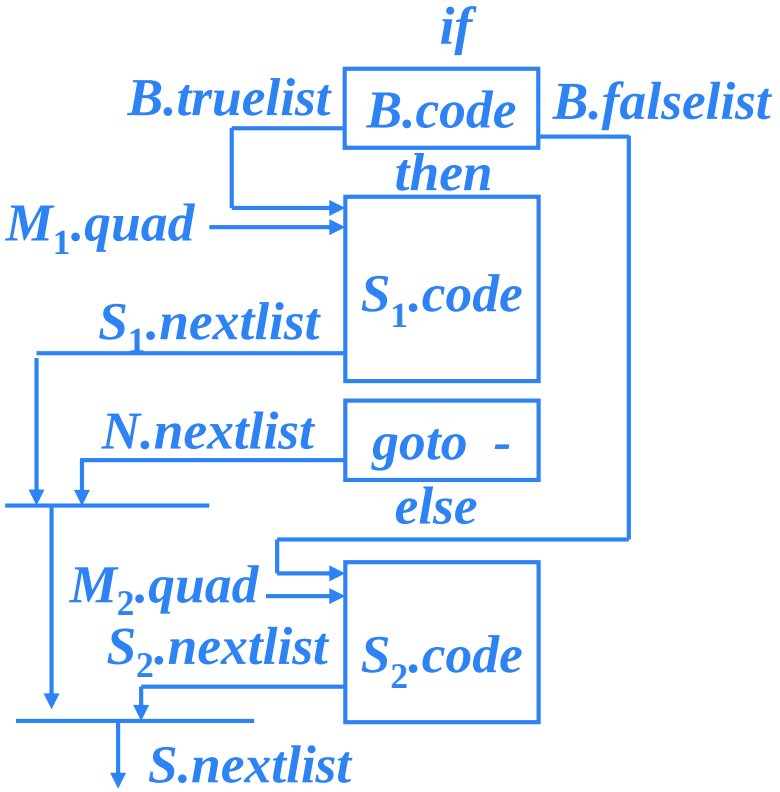
}

$N \rightarrow \varepsilon$

{ N.nextlist = makelist(nextquad);

gen('goto _');

}



$S \rightarrow$ while B do S_1

- 构造代码结构图
- 文法改造
  - 在list箭头指向的位置设置标记非终结符M
- 在产生式末尾的语义动作中
  - 计算综合属性
  - 调用backpatch ( )函数回填各个list

$S \rightarrow$ while M_1 B do M_2 S_1

{

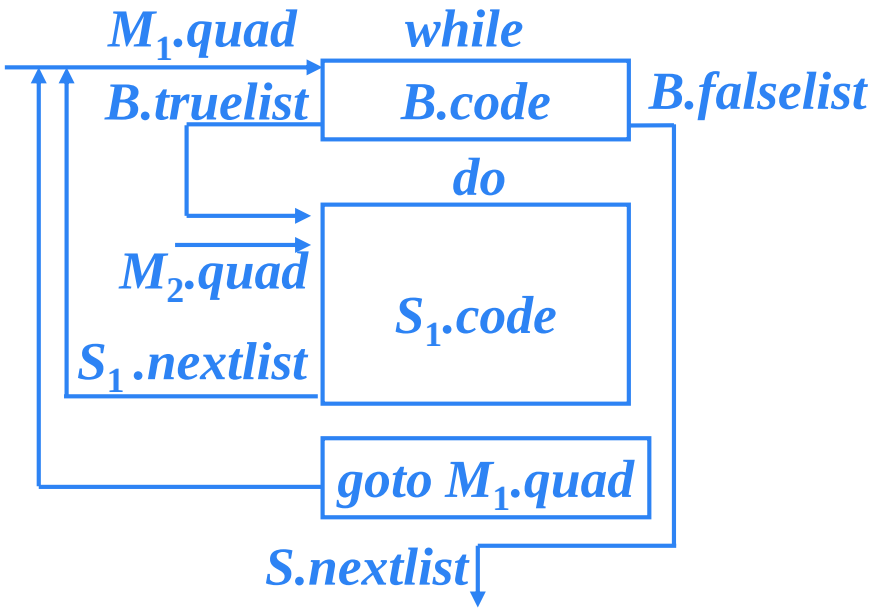
backpatch(S_1.nextlist, M_1.quad);

backpatch(B.truelist, M_2.quad);

S.nextlist = B.falselist;

gen('goto' M_1.quad);

}



$S \rightarrow S_1\ S_2$

- 构造代码结构图
- 文法改造
  - 在list箭头指向的位置设置标记非终结符M
- 在产生式末尾的语义动作中
  - 计算综合属性
  - 调用backpatch ( )函数回填各个list

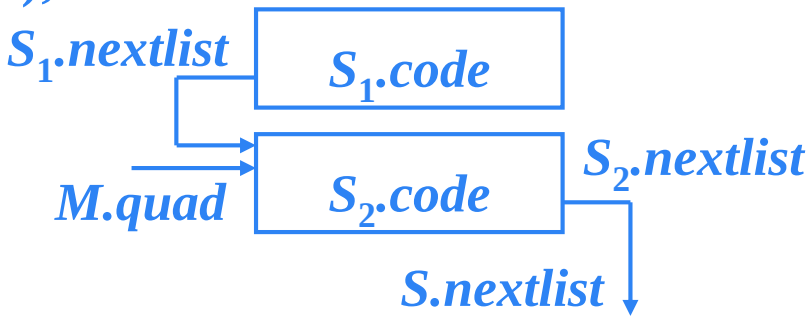
$S \rightarrow S_1\ M\ S_2$

{

backpatch(S_1.nextlist, M.quad);

S.nextlist = S_2.nextlist ;

}



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

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

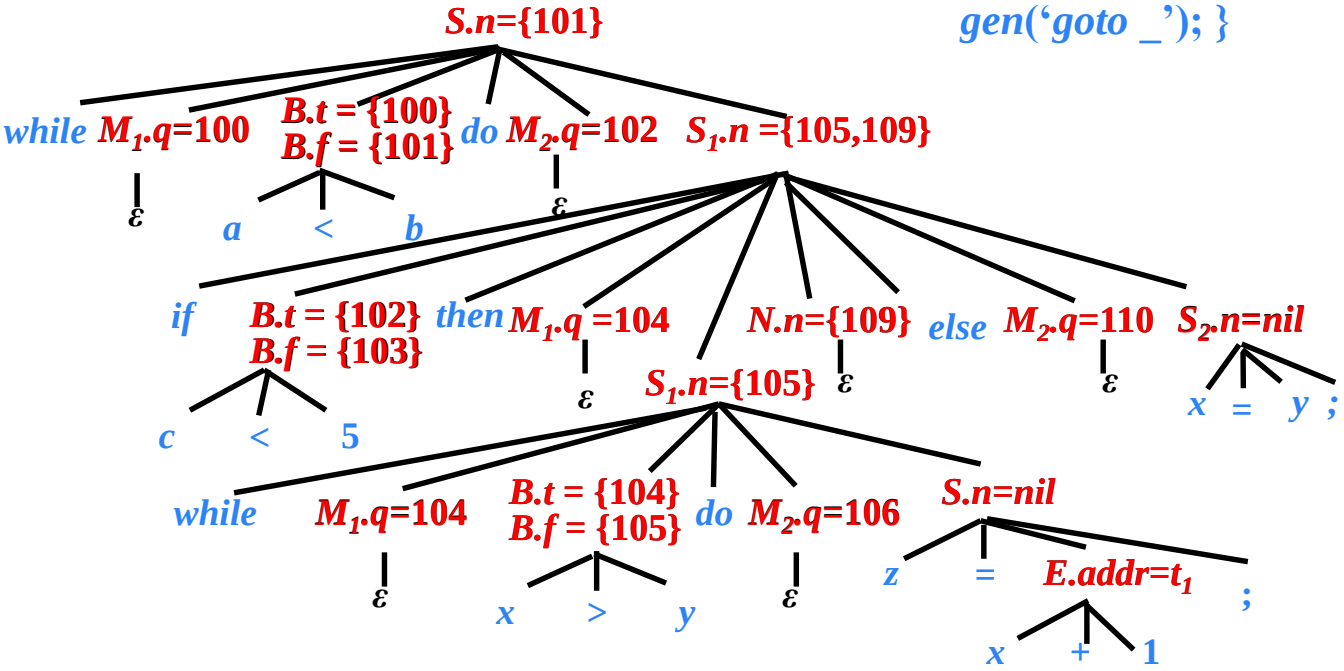
例

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

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



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:

提纲

8.1 声明语句的翻译
8.2 赋值语句的翻译
8.3 控制流语句的翻译
8.4 代码回填
8.5 switch语句的翻译
8.6 过程调用语句的翻译

提纲

8.1 声明语句的翻译
8.2 赋值语句的翻译
8.3 控制流语句的翻译
8.4 代码回填
8.5 switch语句的翻译
8.6 过程调用语句的翻译

过程调用的翻译

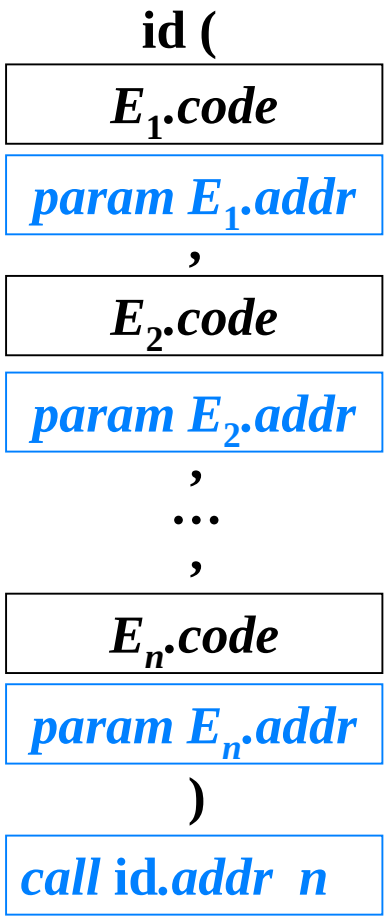
- 文法
  - $S \rightarrow$ call id ($Elist$)

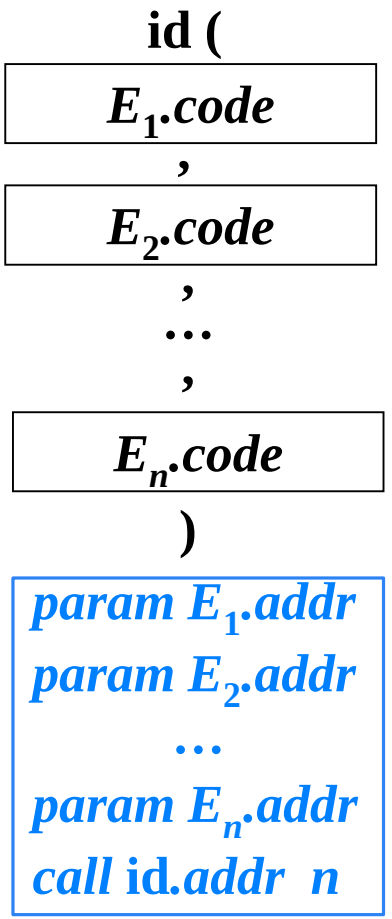
    $Elist \rightarrow Elist, E$

    $Elist \rightarrow E$

过程调用语句的代码结构

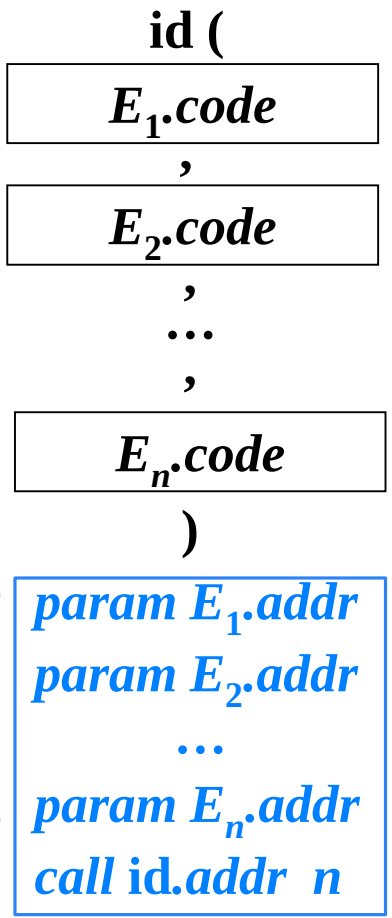
id(E_1, E_2, … , E_n)





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*)

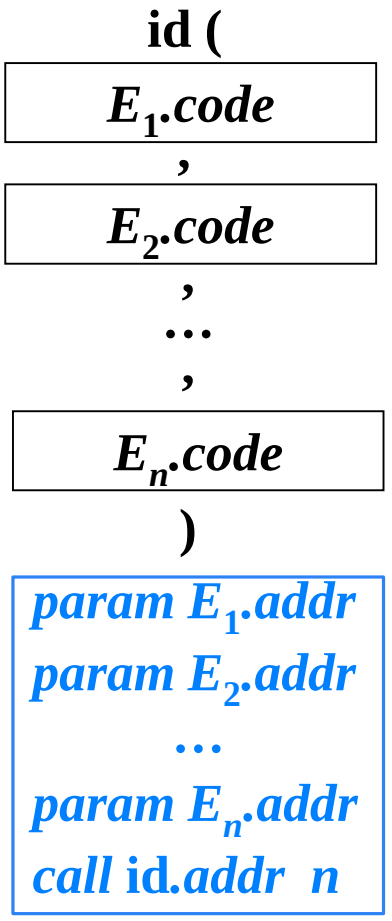
```
{       n=0;
        for queue中的每个t  do
        {       gen('param' t );
                n = n+1;
        }
        gen('call' id.addr ',' n);
}
```

- *Elist* → *E*

{ 将*queue*初始化为只包含*E.addr*; }

- $Elist \rightarrow Elist_1, E$

{ 将*E.addr*添加到*queue*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

中间代码生成要点总结

➢ 本章是上一章（语法制导翻译）的延伸

- SDT
  - 实现机制（第7章）
    - 如何扩展语法分析栈
    - 如何将SDT中的抽象定义式改写成具体执行的栈操作
    - 如何扩展递归下降过程
    - 如何修改L属性定义的SDT，使之适合bottom_up翻译
  - 设计具体语法结构的SDT（第8章）
    - 声明语句的SDT
    - 赋值语句的SDT
    - 控制流语句的SDT
    - 过程调用语句的SDT

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）

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

$S \rightarrow$ 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）

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

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 $E_1.type == integer$ and $E_2.type == integer$ then

{ $gen(E.addr$ '=' $E_1.addr$ 'int+' $E_2.addr)$; $E.type = integer$; }

else if $E_1.type == integer$ and $E_2.type == real$ then

{$u = newtemp()$;

$gen(u$ '=' '$inttoreal$' $E_1.addr)$;

$gen(E.addr$ '=' u '$real$+' $E_2.addr)$;

$E.type = real$; }

. . .

}

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）
  - 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）
  - 过程调用的参数类型或数目不匹配
  - 函数返回类型有误

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

基本语句的语法制导翻译过程

基本控制流语句的翻译方案如下：

请给出把语句while(a>b) do if (c<d) then x=x*a翻译成中间代码的过程

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

$M \rightarrow \varepsilon$ {*M.quad* = *nextquad* ; }

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

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

$L \rightarrow S$
{ *L.nextlist* = *S.nextlist* }

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

$B \rightarrow (B_1)$
{ *B.truelist* = B_1.*truelist* ;
B.falselist = B_1.*falselist* ; }

$B \rightarrow$ false
{ *B.falselist* = *makelist*(*nextquad*);
gen('goto _'); }

$B \rightarrow$ true
{ *B.truelist* = *makelist*(*nextquad*);
gen('goto _'); }

$A \rightarrow$ id=E
{ p= *lookup*(id.name);
if p=null *then error else gen*('goto _'); }

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

把下面的语句翻译成三地址码或四元式序列

if A *and* B *and* (C>D)

then if A<B *then* F=1

else F=0

else G=G+1

1:if A goto 3
2:goto 13
3: if B goto 5
4:Goto 13
5:if C>D goto 7
6:Goto 13
7:If A<B goto 9
8:Goto 11
9:F=1
10:Goto 15
11:F=0
12:Goto 15
13:t1=G+1
14:G=t1
15:

if A *and* B *and* (C>D) *then if* A<B *then* F=1 *else* F=0 *else* G=G+1

把下面的语句翻译成三地址码或四元式序列

1、*if* x>0 *and* y>0 *then* z=x+y *else begin* x=x+2; y=y+3 *end*

2、*if* (A<C) *and* (B<D) *then if* A=1 *then* c=c+1 *else if* A≤D *then* A=A+*2*

3、*while* a<c *and* b<d *do if* a=1 *then* c=c+1 *else if a<d then* a=a+2