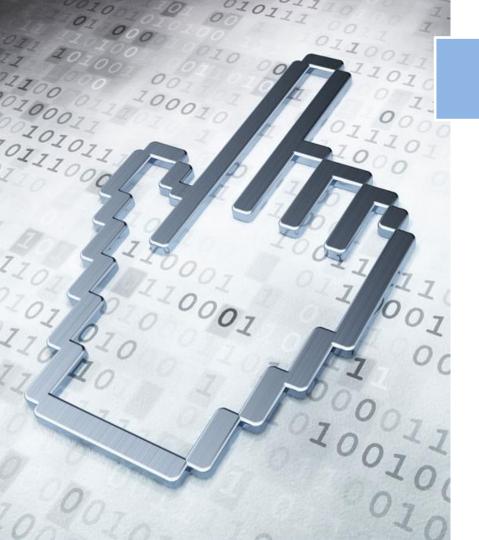


编译原理第六章中间代码生成



哈尔滨工业大学 陈鄞



本章内容

6.1 声明语句的翻译

- 6.2 赋值语句的翻译
- 6.3 控制语句的翻译
- 6.4 回填
- 6.5 switch语句的翻译
- 6.6 过程调用语句的翻译

6.1 声明语句的翻译

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

名字的类型和相对地址信息保存在相应的符号表记录中

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

- > 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
 - \triangleright 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]	<i>array</i> (2, <i>array</i> (3, <i>int</i>))

- > 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
 - \triangleright 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

- > 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×
 - \triangleright 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

- > 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→
 - \triangleright 若 T_1 、 T_2 、...、 T_n 和R是类型表达式,则 $T_1 \times T_2 \times ... \times T_n \rightarrow R$ 是类型表达式

- > 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
 - ightharpoonup 若有标识符 N_1 、 N_2 、...、 N_n 与类型表达式 T_1 、 T_2 、...、 T_n ,则 $record((N_1 \times T_1) \times (N_2 \times T_2) \times ... \times (N_n \times T_n))$ 是一个类型表达式

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

局部变量的存储分配

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

变量声明语句的SDT

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

- ① $P \rightarrow \{ offset = 0 \} D$
- ② $D \rightarrow T$ id;{ enter(id.lexeme, T.type, offset); offset = offset + T.width;}D
- $\bigcirc D \rightarrow \varepsilon$
- **4** $T \rightarrow B$ { $t = B.type; w = B.width;}$ <math>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** $B \rightarrow \text{int} \{ B.type = int; B.width = 4; \}$
- $\bigcirc B \rightarrow \text{real}\{B.type = real; B.width = 8;\}$
- (8) $C \rightarrow \varepsilon$ { C.type = t; C.width = w; }
- ⑨ $C \rightarrow [\text{num}]C_1 \{ C.type = array(num.val, C_1.type);$ $C.width = num.val * C_1.width; \}$

符号	综合属性
В	type, width
\boldsymbol{C}	type, width
T	type, width

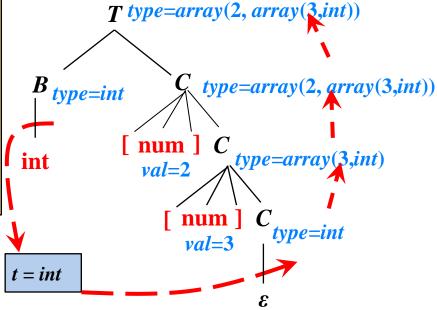
变量	作用
offset	下一个可用的偏移地址
t, w	将类型和宽度信息从语法 分析树中的 B 结点传递到 对应于产生式 $C \rightarrow \varepsilon$ 的结点

"real x; int i;"的语法制导翻译 Select(1)= $\{int,real, \uparrow,\$\}$ $(1)P \rightarrow \{ offset = 0 \} D$ Select(2)= $\{int,real, \uparrow\}$ **Select(3)={\$}** $2D \rightarrow T \text{ id}; \{ enter(id.lexeme, T.type, offset) \}$ Select(4)={int,real} offset = offset + T.width; DSelect(5)= $\{ \uparrow \}$ enter(x, real, 0) $(3)D \rightarrow \varepsilon$ **Select(6)={ int }** $\textcircled{4}T \rightarrow B \quad \{ t = B.type; w = B.width; \}$ Select(7)={ real } enter(i, int, 8) $C \{ T.type = C.type; T.width = C.width; \}$ $Select(8)=\{id\}$ $\{a\}$ D $\textcircled{5}T \rightarrow \uparrow T_1 \{ T.type = pointer(T_1.type); T.width = 4; \}$ **Select(9)={** [} $(6)B \rightarrow \text{int} \{ B.type = int; B.width = 4; \}$ type=real $7B \rightarrow \text{real}\{B.type = real; B.width = 8;\}$ width=8 $\otimes C \rightarrow \varepsilon \{ C.type=t; C.width=w; \}$ $\mathfrak{G}C \to [\text{num}]C_1 \{ C.type = array(num.val, C_1.type);$ T type=int . $B_{type=real} \{a\}_{Ctype=real} \{a\}$ $\underset{width=4}{\text{width}=4} \text{ id } ; \{a\}D$ $C.width = num.val * C_1.width;$ width=8 offset = 12real $\{a\}$ $B type=int\{a\}C_{type=int}\{a\} \varepsilon$ t = int \angle width=4

数组类型表达式的语法制导翻译

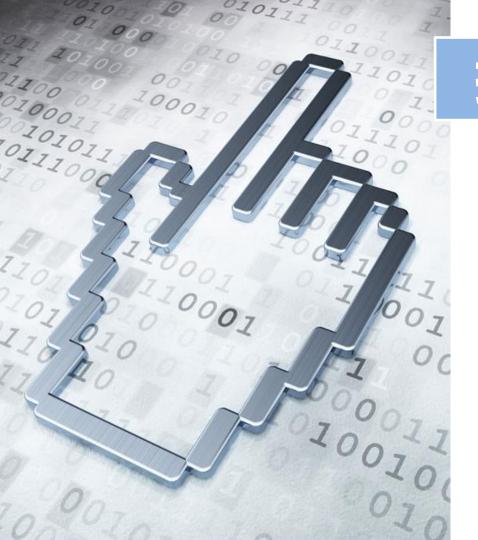
```
(1)P \rightarrow \{ offset = 0 \} D
2D \rightarrow T \text{ id}; \{ enter(id.lexeme, T.type, offset) \}
          offset = offset + T.width; D
(3)D \rightarrow \varepsilon
(4)T \rightarrow B \quad \{ t = B.type; w = B.width; \}
           C \{ T.type = C.type; T.width = C.width; \}
\textcircled{5}T \rightarrow \uparrow T_1 \{ T.type = pointer(T_1.type); T.width = 4; \}
(6)B \rightarrow \text{int} \{ B.type = int; B.width = 4; \}
\bigcirc B \rightarrow \text{real} \{ B.type = real; B.width = 8; \}
\otimes C \rightarrow \varepsilon \{ C.type=t; C.width=w; \}
\mathfrak{D}C \to [\operatorname{num}]C_1 \{ C.type = \operatorname{array}(\operatorname{num.val}, C_1.type) \}
              C.width = num.val * C_1.width;
```

例: "int[2][3]"



例: 数组类型表达式"int[2][3]"的语法制导翻译

```
_{T} type=array(2, array(3,int))
(1)P \rightarrow \{ offset = 0 \} D
                                                                                                 width=24
2D \rightarrow T \text{ id}; \{ enter(id.lexeme, T.type, offset) \}
         offset = offset + T.width; D
                                                                                 B type=int\{a\} C type=array(2, array(3, in
(3)D \rightarrow \varepsilon
                                                                                                            width=24
                                                                                   \mathbf{width} = 4
\textcircled{4}T \rightarrow B \quad \{ t = B.type; w = B.width; \}
          C \{ T.type = C.type; T.width = C.width; \}
                                                                                                              ctype=array(3,int)
                                                                               int \{a\}
\textcircled{5}T \rightarrow \uparrow T_1 \{ T.type = pointer(T_1.type); T.width = 4; \}
                                                                                                                width=12
(6)B \rightarrow \text{int} \{ B.type = int; B.width = 4; \}
7B \rightarrow \text{real} \{ B.type = real; B.width = 8; \}
                                                                                                                      c type=int
                                                                                                                                          {a}
\otimes C \rightarrow \varepsilon \{ C.type=t; C.width=w; \}
                                                                                                                       \mathbf{w}idth=4
\mathfrak{D}C \to [\operatorname{num}]C_1 \{ C.type = \operatorname{array}(\operatorname{num.val}, C_1.type) \}
                                                                             t = int
             C.width = num.val * C_1.width; 
                                                                                                                           {a}
                                                                              w = 4
```



提纲

- 6.1 声明语句的翻译
- 6.2 赋值语句的翻译
- 6.3 控制语句的翻译
- 6.4 回填
- 6.5 switch语句的翻译
- 6.6 过程调用语句的翻译

6.2 赋值语句的翻译

- > 6.2.1简单赋值语句的翻译
- > 6.2.2数组引用的翻译

6.2.1 简单赋值语句的翻译

- > 赋值语句的基本文法
 - (1) $S \rightarrow id = E$;
 - $② E \rightarrow E_1 + E_2$

 - 6 $E \rightarrow \text{id}$

- > 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

$$\succ t_1 = a + b$$

$$> t_2 = t_1 * c$$

$$> x = t_2$$

赋值语句的SDT

```
符号 综合属性
S code
E code addr
```

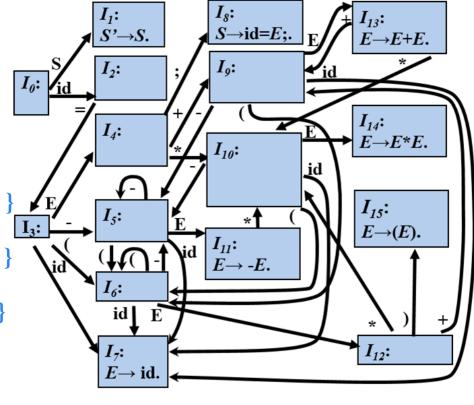
```
S \rightarrow id = E; { p = lookup(id.lexeme); if p == nil then error;
                S.code = E.code \parallel
               gen(p'='E.addr); }
                                               newtemp(): 生成一个新的临时变量t,
E \rightarrow E_1 + E_2 \{ E.addr = newtemp() \}
                                               返回t的地址
              E.code = E_1.code \parallel E_2.code \parallel
              gen(E.addr '=' E_1.addr '+' E_2.addr);
E \rightarrow E_1 * E_2 \{ E.addr = newtemp() \}
                                                       gen(code): 生成三地址指令code
             E.code = E_1.code \parallel E_2.code \parallel
             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);
                                                  |lookup(name): 查询符号表
E \rightarrow (E_1) \{ E.addr = E_1.addr;
                                                  返回name 对应的记录
           E.code = E_1.code;
E \rightarrow id { E.addr = lookup(id.lexeme); if E.addr == nil then error;
           E.code = ``; }
```

增量翻译 (Incremental Translation)

```
S \rightarrow id = E; { p = lookup(id.lexeme); if p == nil then error;
               S.code = E.code \parallel
                                                 在增量方法中, gen()不仅要构造出
               gen(p'='E.addr);
                                                 一个新的三地址指令, 还要将它添加
E \rightarrow E_1 + E_2 \{ E.addr = newtemp() \}
                                                 到至今为止已生成的指令序列之后
             E.code = E_1.code \parallel E_2.code \parallel
              gen(E.addr'='E_1.addr'+'E_2.addr);
E \rightarrow E_1 * E_2 \{ E.addr = newtemp() \}
             E.code = E_1.code \parallel E_2.code \parallel
             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 = ";
```

```
(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); 
\Im 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); 
6E \rightarrow id { E.addr = lookup(id.lexeme);
             if E.addr==nil then error; }
    例: x = (a + b) * c;
                                  $ | id | = |
```

 \boldsymbol{a}



 $S \rightarrow id = E;$

 $E \rightarrow -E$.

 $E \rightarrow E + E$.

 $E \rightarrow E * E$.

 $E \rightarrow (E)$.

```
S \rightarrow id = E;
                                                                      S'→S.
                                                           I_{\theta}: I_{\mathrm{id}}
(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); \} 
\Im E \rightarrow E_1 * E_2 \{ E.addr = newtemp() \}
           gen(E.addr '=' E_1.addr '*' E_2.addr); 
                                                                                    E \rightarrow -E.
(4)E \rightarrow -E_1 \{ E.addr = newtemp() \}
             gen(E.addr'=''uminus'E_1.addr); \}
                                                                       id
\textcircled{6}E \rightarrow \textbf{id} \quad \{ E.addr = lookup(\textbf{id}.lexeme); \}
                                                                      E \rightarrow id.
                if E.addr==nil then error; }
                                                                                         t_1 = a + b
    例: x = (a + b) * c;
                                         $ | id | = |
```

 $E \rightarrow E + E$.

 $E \rightarrow E * E$.

 $E \rightarrow (E)$.

$$\begin{array}{c} \text{(1)} S \rightarrow \text{id} = E; \{ p = lookup(\text{id}.lexeme); \\ if p == nil \ then \ error; \\ gen(p '=' E.addr); \} \\ \text{(2)} E \rightarrow E_1 + E_2 \{ E.addr = newtemp(); \\ gen(E.addr '=' E_1.addr '+' E_2.addr); \} \\ \text{(3)} E \rightarrow E_1 * E_2 \{ E.addr = newtemp(); \\ gen(E.addr '=' E_1.addr '*' E_2.addr); \} \\ \text{(4)} E \rightarrow -E_1 \{ E.addr = newtemp(); \\ gen(E.addr '=' 'uminus' E_1.addr); \} \\ \text{(5)} E \rightarrow (E_1) \{ E.addr = E_1.addr ; \} \\ \text{(6)} E \rightarrow \text{id} \{ E.addr = lookup(\text{id}.lexeme); \\ if E.addr == nil \ then \ error; \} \\ \text{(5)} E \rightarrow (E_1) \{ E.addr = lookup(\text{id}.lexeme); \\ if E.addr == nil \ then \ error; \} \\ \text{(6)} E \rightarrow \text{id} \{ E.addr = (B + b) * c; \} \\ \text{(6)} E \rightarrow \text{(6$$

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

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

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

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

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

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

例:
$$x = (a + b) * c$$
;

$$t_1 = a + b$$

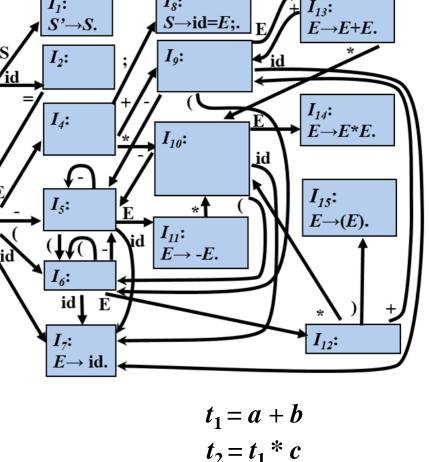
 $E \rightarrow E + E$.

 $E \rightarrow E * E$.

 $E \rightarrow (E)$.

$$\begin{array}{c} \text{(1)} S \rightarrow \text{id} = E; \{ p = lookup(\text{id.}lexeme); \\ if p == nil \ then \ error; \\ gen(p'='E.addr); \} \\ \text{(2)} E \rightarrow E_1 + E_2 \{ E.addr = newtemp(); \\ gen(E.addr'='E_1.addr'+'E_2.addr); \} \\ \text{(3)} E \rightarrow E_1 * E_2 \{ E.addr = newtemp(); \\ gen(E.addr'='E_1.addr'**E_2.addr); \} \\ \text{(4)} E \rightarrow -E_1 \{ E.addr = newtemp(); \\ gen(E.addr'='uminus'E_1.addr); \} \\ \text{(5)} E \rightarrow (E_1) \{ E.addr = E_1.addr; \} \\ \text{(6)} E \rightarrow \text{id} \quad \{ E.addr = lookup(\text{id.}lexeme); \\ if E.addr== nil \ then \ error; \} \\ \end{array}$$

例:
$$x = (a + b) * c$$
;



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

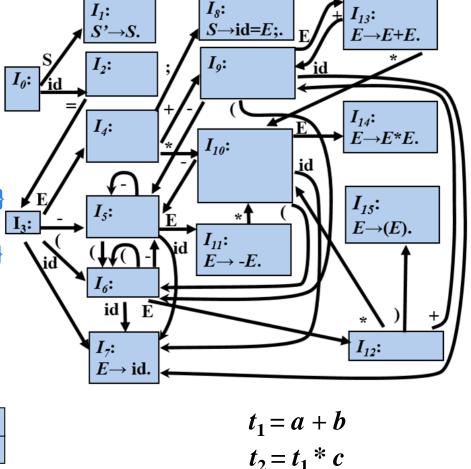
$$(2)E \rightarrow E_1 + E_2 \{ E.addr = newtemp(); gen(E.addr '=' E_1.addr '+' E_2.addr); \}$$

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

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

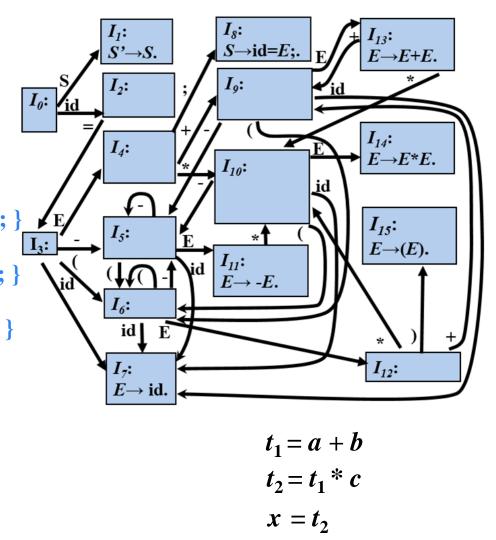
$$(5)E \rightarrow (E_1) \{ E.addr = E_1.addr ; \}$$

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



 $x=t_2$

①
$$S o id = E$$
; { $p = lookup(id.lexeme)$; if $p = nil$ then error; $gen(p `=` E.addr)$; }
② $E o E_1 + E_2$ { $E.addr = newtemp()$; $gen(E.addr `=` E_1.addr `+` E_2.addr)$; }
③ $E o E_1 * E_2$ { $E.addr = newtemp()$; $gen(E.addr `=` E_1.addr `*` E_2.addr)$; }
④ $E o -E_1$ { $E.addr = newtemp()$; $gen(E.addr `=` `uminus` E_1.addr)$; }
⑤ $E o (E_1)$ { $E.addr = E_1.addr$; }
⑥ $E o id$ { $E.addr = lookup(id.lexeme)$; if $E.addr = nil$ then error; }



上一讲内容回顾

- > 6.1 声明语句的翻译
- > 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

6.2.2 数组引用的翻译

ightarrow 赋值语句的文法 $S \to id = E; | L = E;$ $E \to E_1 + E_2 | -E_1 | (E_1) | id | L$ $L \to id [E] | L_1 [E]$

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

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, ixw是偏移地址

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

 $base+i_1\times w_1+i_2\times w_2$

>k维数组

偏移地址

 \rightarrow 数组元素 $a[i_1][i_2]...[i_k]$ 的相对地址是:

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

偏移地址

 $w_1 \rightarrow a[i_I]$ 的宽度 $w_2 \rightarrow a[i_I] [i_2]$ 的宽度 ... $w_k \rightarrow a[i_I] [i_2] ...[i_k]$ 的宽度

》一假设type(a) = array(3, array(5, array(8, int))), 一个整型变量占用4个字节, \mathbb{N} $addr(a[i_1][i_2][i_3]) = base + i_1 * w_1 + i_2 * w_2 + i_3 * w_3$ $= base + i_1*160 + i_2*32 + i_3*4$ $a[i_i]$ 的宽度 $a[i_1][i_2]$ 的宽度 $a[i_1][i_2][i_3]$ 的宽度'

带有数组引用的赋值语句的翻译

>例1

假设 type(a)=array(n, int),

>源程序片段

$$\succ c = a[i];$$

>三地址码

$$> t_1 = i * 4$$

$$> t_2 = a [t_1]$$

$$\geq c = t_2$$

$$\frac{addr(a[i]) = base + i*4}{t_1}$$

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

带有数组引用的赋值语句的翻译

>例2 ▶源程序片段 >三地址码 $> t_1 = i_1 * 20$ $>t_2=i_2*4$ $>t_3=t_1+t_2$ $\triangleright t_{A} = a [t_{3}]$

 $>c = t_{\perp}$

假设 type(a) = array(3, array(5, int)), $> c = a[i_1][i_2]; |addr(a[i_1][i_2]) = base + i_1*20 + i_2*4$

数组元素寻址的SDT

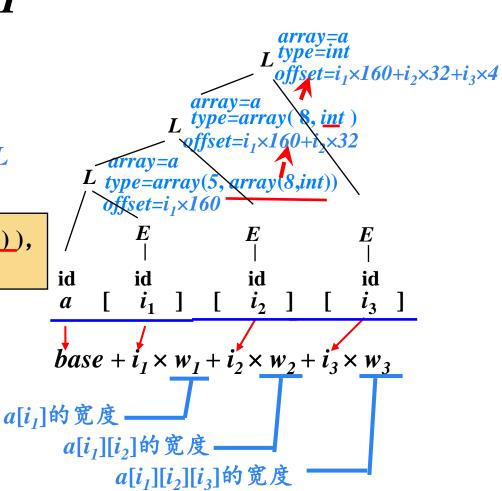
▶ 赋值语句的基本文法 $S \rightarrow id = E; | L = E;$ $E \rightarrow E_1 + E_2 | -E_1 | (E_1) | id | L$ $L \rightarrow id [E] | L_1[E]$

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

翻译语句片段 " $a[i_1][i_2][i_3]$ "

> L的综合属性

- > L.type: L生成的数组元素的类型
- ho L.offset: 指示一个临时变量,该临时变量用于ho 变量用于ho 之式中的 $i_j imes w_j$ 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
S \rightarrow id = E;
   L = E; { gen(L.array '['L.offset ']' '=' E.addr); }
E \to E_1 + E_2 | -E_1 | (E_1) | id
  L \{ E.addr = newtemp(); gen(E.addr'='L.array'['L.offset']'); \} E addr=t_6
                                                                                                  三地址码
L \rightarrow id [E] \{ L.array = lookup(id.lexeme); if L.array == nil then error; \}
                                                                                                  t_1 = i_1 * 160
                                                                                      array=a
                                                                                      type=int
                                                                                                  t_2 = i_2 * 32
              L.type = L.array.type.elem;
                                                                                       offset=t_
                                                                                                  t_3 = t_1 + t_2
              L.offset = newtemp();
                                                                                                  t_4 = i_3*4
              gen(L.offset '=' E.addr '*' L.type.width ); }
                                                                                                  t_5 = t_3 + t_4
                                                                           ottset=t2
   L_1[E]{ L.array = L_1. array;
                                                                                                  t_6 = a[t_5]
            L.type = L_1.type.elem;
                                                              L type=array(5, array(8,int)
                                                                 offset=t_1
            t = newtemp();
                                                                    E_{addr=i_1}
            gen(t'='E.addr''*'L.type.width);
                                                                                 E_{1}addr=i_{2}
            L.offset = newtemp();
                                                           id
            gen(L.offset '=' L_1.offset '+' t); \}
```

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

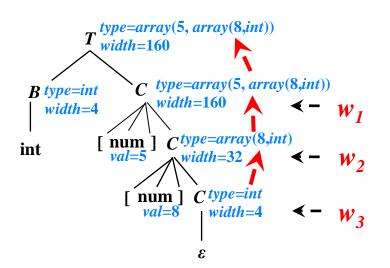
翻译声明语句时已经计算过 $addr(a[i_1][i_2][i_3])=base+i_1\times w_1+i_2\times w_2+i_3\times w_3$

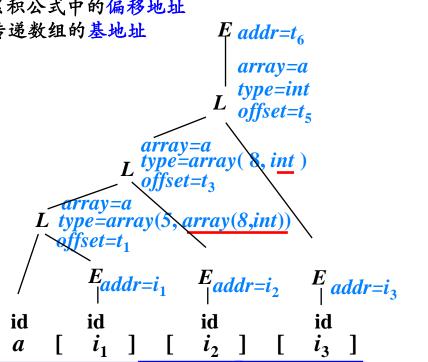
设置type属性: 计算宽度w

设置offset属性:累积公式中的偏移地址

设置array属性:传递数组的基地址

数组声明语句的翻译





符号表的组织

〉例

```
int abc;
int i;
```

int myarray[3][4];

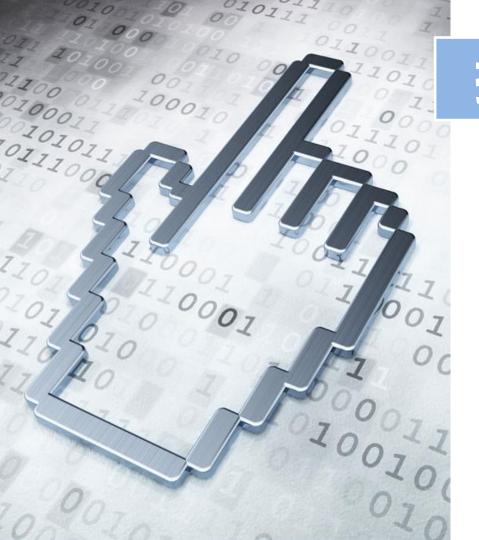
名字	基本属性					扩展属性		
	种属	类型	地址	扩展属性指针	,			
abc	变量	int	0	NULL				
i	变量	int	4	NULL		维数	各维	
myarray	数组	int	8			2	3	4
•••		<u> </u>	•••	I			16	4
•••							各组	主宽.

各维宽度

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 " $a[i_1][i_2][i_3]$ "

 $addr(a[i_1][i_2][i_3]) = base + i_1 \times w_1 + i_2 \times w_2 + i_3 \times w_3$ $S \rightarrow id = E$; L = E; { gen(L.array '['L.offset ']' '=' E.addr); } $E \to E_1 + E_2 | -E_1 | (E_1) | id$ $L \{ E.addr = newtemp(); gen(E.addr'='L.array'['L.offset']'); \} E addr=t_6$ 三地址码 $L \rightarrow id [E] \{ L.array = lookup(id.lexeme); if L.array == nil then error; \}$ $t_1 = i_1 * 160$ array=a type=int $t_2 = i_2 * 32$ L.type = L.array.type.elem; offset=t_ $t_3 = t_1 + t_2$ L.offset = newtemp(); $t_4 = i_3*4$ gen(L.offset '=' E.addr '*' L.type.width); } $t_5 = t_3 + t_4$ ottset=t2 $L_1[E]$ { $L.array = L_1$. array; $t_6 = a[t_5]$ $L.type = L_1.type.elem$; L type=array(5, array(8,int) $offset=t_1$ t = newtemp(); $E_{addr=i_1}$ gen(t'='E.addr''*'L.type.width); $E_{1}addr=i_{2}$ L.offset = newtemp();id $gen(L.offset '=' L_1.offset '+' t); \}$



提纲

- 6.1 声明语句的翻译
- 6.2 赋值语句的翻译
- 6.3 控制语句的翻译
- 6.4 回填
- 6.5 switch语句的翻译
- 6.6 过程调用语句的翻译

6.3 控制语句的翻译

户控制流语句的基本文法

$$\triangleright P \rightarrow S$$

$$> S \rightarrow S_1 S_2$$

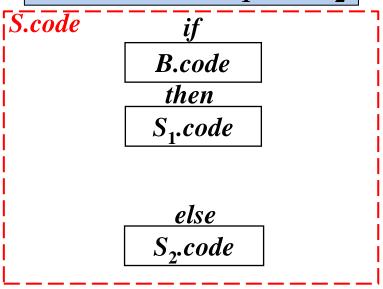
$$> S \rightarrow id = E ; | L = E ;$$

$$\gt S \rightarrow \text{if } B \text{ then } S_1$$

 \mid if B then S_1 else S_2

| while B do S_1

 \triangleright 例 $S \rightarrow if B then <math>S_1$ else S_2



例1: B="a<b"
三地址指令 标号(常量)
if a<b goto S₁.first
goto S₂.first

问题: 生成跳转指令的时候, S₁.first和
S₂.first的值还不知道

例2: B="a<b || a>200 && b<100 "

布尔表达式B被翻译成由 跳转指令构成的跳转代码

〉例 $S \rightarrow if B then S_1 else S_2$ 例1: B="a<b" S.code 三地址指令 标号(常量) if a < b goto S₁.first **B.**code then goto S₂.first S_1 .code 临时指令 变量 if a < b goto **B.true** else goto B.false S₂.code

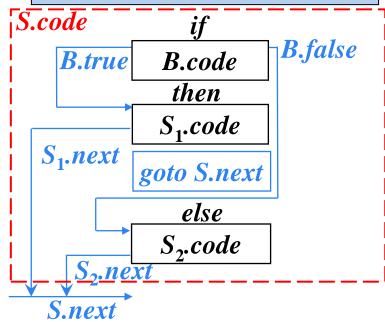
> 布尔表达式B被翻译成由 跳转指令构成的跳转代码

〉例 $S \rightarrow if B then S_1 else S_2$ S.code **B.**code then S_1 .code else S₂.code

例1: B="a<b" 三地址指令 if a < b goto S₁.first goto S₂.first 临时指令 if a<b goto B.true ___if a<b goto (B.true) goto **B.false** goto (B.false)

布尔表达式B被翻译成由 跳转指令构成的跳转代码

 \triangleright 例 $S \rightarrow if B then <math>S_1$ else S_2



布尔表达式B被翻译成由 跳转指令构成的跳转代码

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

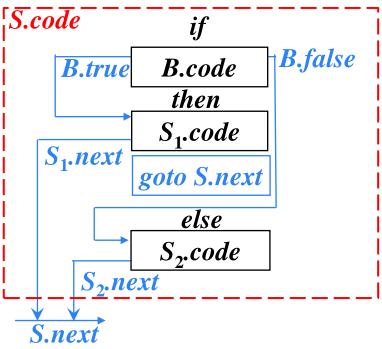
控制流语句的SDT

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

```
\triangleright P \rightarrow \{ S.next = newlabel(); \} S \{ label(S.next); \}
\gt S \rightarrow \{S_1.next = newlabel();\} S_1
                                                            label(L): 将下一条
                                                            三地址指令的标号
         { label(S_1.next); S_2.next = S.next; } S_2
                                                             存放到地址L中
> S \rightarrow id = E ; | L = E ;
\triangleright S \rightarrow if B then S_1
          | if B then S_1 else S_2
           | while B do S_1
```

if-then-else语句的SDT_{[S.code}]

 $S \rightarrow if B then S_1 else S_2$



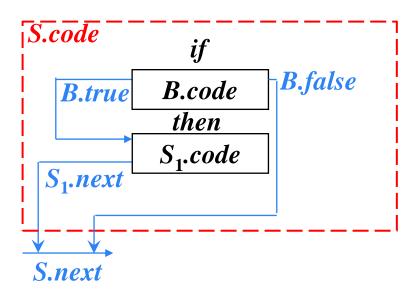
```
S \rightarrow if \{ B.true = newlabel(); B.false = newlabel(); \} B

then \{ S_1.next = S.next; label(B.true); \} S_1 \{ gen(`goto` S.next) \}

else \{ S_2.next = S.next; label(B.false); \} 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 \{S_1.next = S.next; label(B.true); \} S_1
```

while-do语句的SDT

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

布尔表达式的翻译

> 布尔表达式的基本文法

```
B \rightarrow B \text{ or } B
 |B| and |B| 逻辑运算符优先级: not > and > or
 |  not B
 | (B) \rangle
                      关系表达式
 ||E|| relop E
  true
               relop (关系运算符):
  false
               <, <=, >, >=, !=
```

- ▶ 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

> 三地址代码

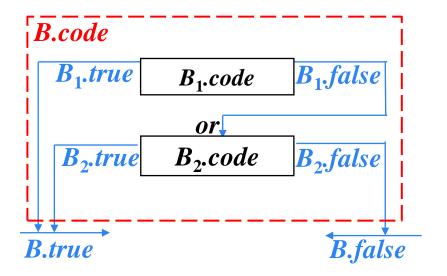
```
if(x<100 \mid x>200 \&\& x!=y)
        x=0:
        if x < 100 goto L_2
        goto L<sub>3</sub>
 L_3: if x>200 goto L_4
       goto L<sub>1</sub>
 L_4: if x!=y goto L_2
        goto L_1
 L_2: x=0
 L_1:
```

布尔表达式的SDT

```
holdsymbol{>} B 
ightharpoonup E_1 	ext{relop } E_2 	ext{gen('if' } E_1.addr relop } E_2.addr 'goto' B.true); 
holdsymbol{>} B 
ightharpoonup true } 	ext{gen('goto' B.true); } 	ext{``a<b`'} 	ext{``a<b''} 	ext{``a<b''} 	ext{``f a<b goto B.true} 	ext{``b''} 	ext{``if a<b goto B.true} 	ext{``b''} 	ext{``b''} 	ext{``b''} 	ext{``a<b''} 	ext{``b''} 	ext{``a<b''} 	ext{``b''} 	ext{``a<b''} 	ext{``b''} 	ext{``a<b''} 	ext{``a
```

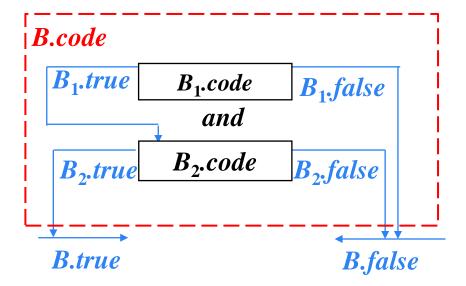
$B \rightarrow B_1 \text{ or } B_2 \text{ 的}SDT$

```
 PB \rightarrow B_1 \text{ or } B_2 
 PB \rightarrow \{B_1.true = B.true; B_1.false = newlabel(); \}B_1 
 PB \rightarrow \{B_2.true = B.true; B_2.false = B.false; label(B_1.false); \}B_2
```



$B \rightarrow B_1$ and B_2 的SDT

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



控制流语句的SDT

- $> P \rightarrow \{a\}S\{a\}$
- $\gt S \rightarrow \{a\}S_1\{a\}S_2$
- $\gt S \rightarrow id=E;\{a\} \mid L=E;\{a\}$

SDD: L-SDD

- 赋值语句: 只定义了综合属性
- 分支、循环语句: 只定义了继承属性, 且不依赖右兄弟节点属性值

基础文法:可以使用LR分析技术

- $F \to E_1 + E_2\{a\} \mid -E_1\{a\} \mid (E_1)\{a\} \mid id\{a\} \mid L\{a\}\}$
- $ightharpoonup L
 ightharpoonup \operatorname{id}[E]\{\mathbf{a}\} \mid L_1[E]\{\mathbf{a}\}$
- > $S \rightarrow \text{if } \{a\}B \text{ then } \{a\}S_1$ | if $\{a\}B \text{ then } \{a\}S_1 \text{ else } \{a\}S_2$ | while $\{a\}B \text{ do } \{a\}S_1\{a\}$
- $\triangleright B \rightarrow \{a\}B \text{ or } \{a\}B \mid \{a\}B \text{ and } \{a\}B \mid \text{not } \{a\}B \mid (\{a\}B) \mid E \text{ relop } E\{a\} \mid \text{true}\{a\} \mid \text{false}\{a\}$

例:

else

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

$$x = y - z$$
;

SDT的通用实现方法

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

控制流语句的SDT

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

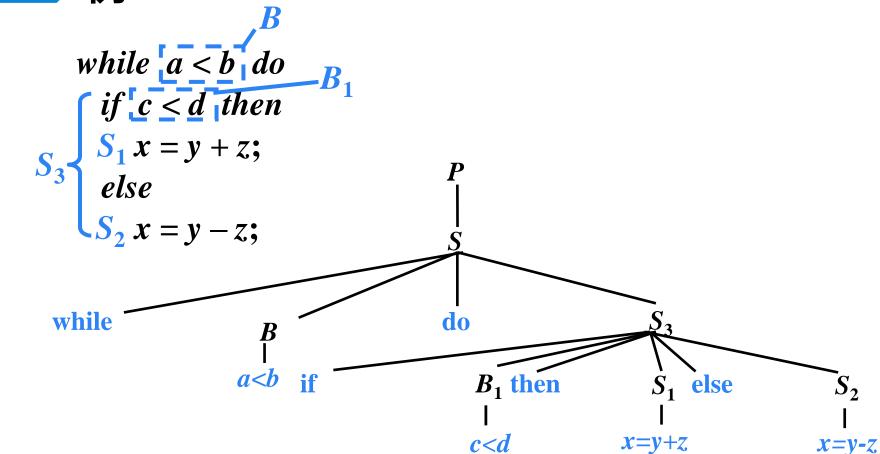
```
while a < b do

if c < d then

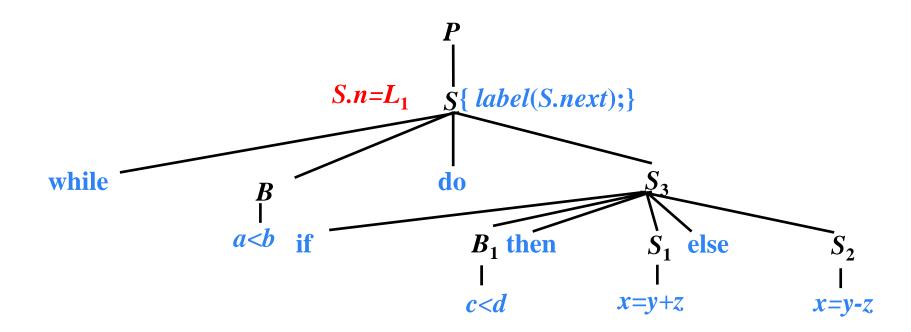
x = y + z;

else

x = y - z;
```



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





 $S \rightarrow \text{while } \{B.begin = newlabel(); \ 2: goto \ L_1 \}$

label(B.begin); $B \rightarrow E_1 \text{ relop } E_2$ $B.true = newlabel(); \{ gen('if' E_1.addr relop E_2.addr' goto' B.true); \}$ B.false = S.next; B | gen('goto' B.false); do { label(B.true); $S_1.next = B.begin; S_1$ { gen('goto' B.begin); } $S.n=L_1$ S{ label(S.next);} goto L_2 $S_3.n=L_2S_1$ while do B_1 then else B.begin = c < dx=y-z

1: if a < b goto (L_2) $S \rightarrow if \{ B.true = newlabel(); B.false = newlabel(); \} B$ 2: $goto(L_1)$ 11 3: if $c < \overline{d}$ goto $(L_A)^3$ then { label(B.true); $S_1.next = S.next$; } S_1 4: goto (L₂) 8 { gen('goto' S.next); } 5: $t_1 = y + z$ else { label(B,false); 6: $x = t_1$ $S_2.next = S.next; \} S_2$ 7: goto (L_2) 1 8: $t_2 = y - z$ S{ label(S.next);} 9: $x = t_2$ $10:goto(L_2)$ S_3 , $n=L_2$ S_1 while do 11: B_1 then S_1 . $n=L_2S_1$ else S_2 . $n=L_2S_2$ $B_1.t =$ B.begin =

· 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: $x = t_2$

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

1: ifFalse a < b goto 11 避免生成冗余的 B.first $\rightarrow 1$: if a < b goto 3

2: goto 11 goto指令 S₁. first \rightarrow 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:** $x = t_1$

11:

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:

do S_1 .code S_1 .next goto B.begin S.next

while

B.code

B.false

B.begin

B.true

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: if False a < b goto 11

2: goto 11 2:

B.first $\rightarrow 3$: if c < d goto 5 3: if c < d goto 5

4: goto 8 4: goto 8

7. gold 0

S₁. first $\to 5$: $t_1 = y + z$ 5: $t_1 = y + z$

6: $x = t_1$ 6: $x = t_1$

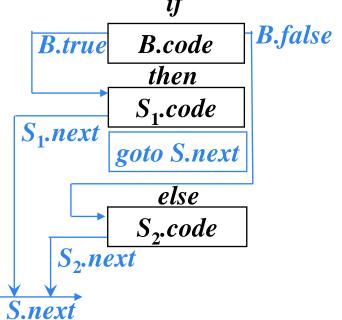
7: goto 1 7: goto 1

S₂. first $\rightarrow 8$: $t_2 = y - z$ 8: $t_2 = y - z$

9: $x = t_2$ 9: $x = t_2$

10: goto 1 10: goto 1

11:



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: if False a < b goto 11

2: goto 11 2:

B.first $\rightarrow 3$: if c < d goto 5 3: if False c < d goto 8

4: goto 8 4:

7 6 4:

S₁. first $\to 5$: $t_1 = y + z$ 5: $t_1 = y + z$

6: $x = t_1$ 6: $x = t_1$

7: goto 1 7: goto 1

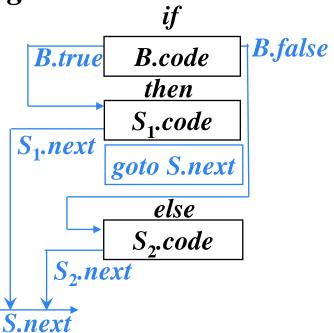
 S_2 . first $\rightarrow 8$: $t_2 = y - z$ 8: $t_2 = y - z$

9: $x = t_2$ 9: $x = t_2$

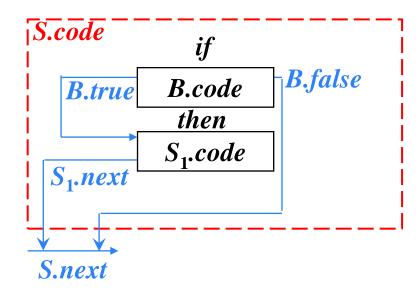
11:

10: goto 1 10: goto 1

11:



修改if-then语句的SDT



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

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

省略。不生成任何跳转指令

$$S \rightarrow if \{ B.true = fall; B.false = S.next; \} B$$

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

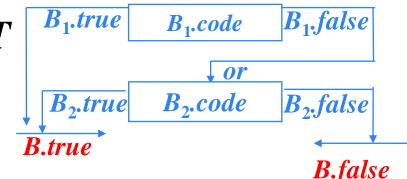
修改 $B \rightarrow E_1$ relop E_2 的SDT

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



```
 \begin{array}{l} \nearrow B \rightarrow E_1 \ \text{relop} \ E_2 \\ \text{ if } \textit{B.true} \neq \textit{fall and B.false} \neq \textit{fall then} \\ \textit{gen('if'} \ E_1.\textit{addr relop} \ E_2.\textit{addr 'goto'} \ \textit{B.true}); \ \textit{gen('goto'} \ \textit{B.false}); \ \text{else if } \textit{B.true} \neq \textit{fall then gen('if'} \ E_1.\textit{addr relop} \ E_2.\textit{addr 'goto'} \ \textit{B.true}); \\ \text{else if } \textit{B.false} \neq \textit{fall then gen('ifFalse'} \ E_1.\textit{addr relop} \ E_2.\textit{addr'goto'} \ \textit{B.false}); \\ \text{else''} \ \} \end{array}
```

修改 $B \rightarrow B_1$ or B_2 的SDT

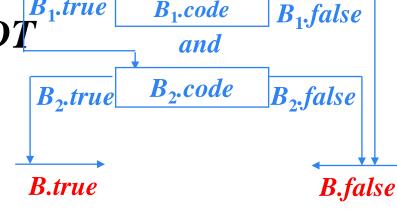


$$B \rightarrow \{ B_1.true = B.true; B_1.false = newlabel(); \} B_1$$
or $\{ B_2.true = B.true; B_2.false = B.false; label(B_1.false); \} B_2$

$$B \rightarrow \{B_1.true = B.true = = fall ? newlabel() : B.true; B_1.false = fall; \}B_1 \text{ or }$$

 $\{\textit{B}_{2}.true = \textit{B}.true; \textit{B}_{2}.false = \textit{B}.false; \}\textit{B}_{2} \{\text{if } \textit{B}.true = = fall \text{ then } label(\textit{B}_{1}.true); \}$

修改 $B \rightarrow B_1$ and B_2 的SDT



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

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



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

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

 $S \rightarrow if \{ B.true = fall; B.false = S.next; \} B$ then $\{S_1.next = S.next;\} S_1$ $B \rightarrow \{ B_1.true = B.true = = fall? newlabel():B.true; \}$ if(x<100 || x>200 && x!=y) B_1 -false = fall; B_1 or B_2 -true = B.true; B_2 -false =

B.false; B_2 {if B.true == fall then label(B_1 .true);}

x=0;

 $B \rightarrow \{B_1.true = fall; B_1.false = B.false = = fall ?$ newlabel(): B.false; B_1 and $\{B_2.true = B.true; B_2.false\}$ $S.n=L_1$ then

if B.t = fall B $S_1.n = L_1.1$ x=0 $B.t = L_2$ $B.f = f\tilde{a}ll B_1$ or x<100 B.t = fall and $B.t = fall B_4$

if x<100 goto L_{2} ifFalse x>200 goto L_1 if False x!=y goto L_1 L_2 : x=0

if

 $B.t = L_2$

 $B.f = f\tilde{a}ll B_1$

 $S \rightarrow if \{ B.true = fall; B.false = S.next; \} B$ then $\{S_1.next = S.next;\} S_1$ $B \rightarrow \{ B_1.true = B.true = = fall? newlabel():B.true; \}$ if(x<100 || x>200 && x!=y) B_1 -false = fall; B_1 or B_2 -true = B.true; B_2 -false =

x=0; $S.n=L_1$

B.t = fall B

or

x<100 B.t = fall

 $S_1.n = L_1.1$ then x=0 $B.t = fall B_2$

and $B.t = fall B_4$

B.false; B_2 {if B.true == fall then label(B_1 .true);} $B \rightarrow \{B_1.true = fall; B_1.false = B.false = = fall ?$ newlabel(): B.false; B_1 and $\{B_2.true = B.true; B_2.false\}$ if x<100 goto L_{2}

ifFalse x>200 goto L_1 if False x!=y goto L_1 x=0

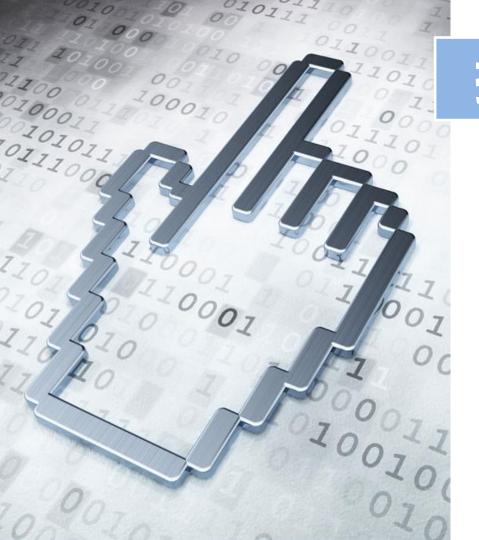
 L_2 :

 L_1 :

 $S \rightarrow if \{ B.true = fall; B.false = S.next; \} B$ then $\{S_1.next = S.next;\} S_1$

 $B \rightarrow \{ B_1.true = B.true = = fall? newlabel():B.true; \}$ if(x>200 && x!=y || x<100) B_1 .false = fall; B_1 or B_2 .true = B.true; B_2 .false = x=0; B.false; B_2 {if B.true == fall then label(B_1 .true);} $B \rightarrow \{B_1.true = fall; B_1.false = B.false = = fall ?$ newlabel(): B.false; B_1 and $\{B_2.true = B.true; B_2.false\}$ $S.n=L_1$ \dot{B} thèn S_1 . $n=L_1S_1$ if False x>200 goto L_3

 $B.t = L_2$ if x!=y goto L, $B.f = fall B_1$ L_3 : ifFalse x<100 goto L_1 x=0x < 100



提纲

- 6.1 声明语句的翻译
- 6.2 赋值语句的翻译
- 6.3 控制语句的翻译

6.4 回填

- 6.5 switch语句的翻译
- 6.6 过程调用语句的翻译

6.4 回填 (Backpatching)

>基本思想

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

非终结符B的综合属性

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

函数

- >makelist(i)
 - ▶创建一个只包含i的列表,i是跳转指令的标号,函数 返回指向新创建的列表的指针
- $\succ merge(p_1, p_2)$
 - \triangleright 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
- \succ 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

```
\triangleright B \rightarrow E_1 \text{ relop } E_2
                         B.truelist = makelist(nextquad); nextquad: physical ph
                          B.falselist = makelist(nextquad+1);
                        gen(if' E_1.addr\ relop\ E_2.addr\ goto\ ');
                       gen('goto');
```

```
 PB \rightarrow E_1 \text{ relop } E_2 
 PB \rightarrow \text{true} 
 \{ B.truelist = makelist(nextquad); \\ gen(`goto \_`); 
 \}
```

```
\triangleright B \rightarrow E_1 \text{ relop } E_2
\triangleright B \rightarrow \text{true}
\triangleright B \rightarrow \text{false}
   B.falselist = makelist(nextquad);
   gen('goto ');
```

```
\triangleright B \rightarrow E_1 \text{ relop } E_2
\triangleright B \rightarrow \text{true}
\triangleright B \rightarrow \text{false}
\triangleright B \rightarrow (B_1)
    B.truelist = B_1.truelist;
    B.falselist = B_1.falselist;
```

```
\triangleright B \rightarrow E_1 \text{ relop } E_2
\triangleright B \rightarrow \text{true}
\triangleright B \rightarrow \text{false}
\triangleright B \rightarrow (B_1)
\triangleright B \rightarrow \text{not } B_1
    B.truelist = B_1.falselist;
    B.falselist = B_1.truelist;
```

$B \rightarrow B_1 \text{ or } B_2$

```
B \rightarrow B_1 or MB_2
   B.truelist = merge(B_1.truelist, B_2.truelist);
   B.falselist = B_{2}.falselist;
   backpatch(B_1, falselist, M.quad);
                                                                       B_1 false list
                                          B<sub>1</sub>.truelist
                                                           B_1.code
M \to \varepsilon
\{ M.quad = nextquad; \}
                                              M.quad
                                                              1 or
                                                                       B<sub>2</sub>.falselist
                                          B<sub>2</sub>.truelist
                                                          B_{2}.code
                                                                   B.falselist
                                          B.truelist
```

$^{\circ}B \rightarrow B_{1}$ and B_{2} $B \rightarrow B_1$ and MB_2 $B.truelist = B_{\gamma}.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);$ $backpatch(B_1.truelist, M.quad);$ B_1 .falselist B_1 .truelist B_1 .code and M.quad B₂:falselist B_2 .code B₂.truelist

B.truelist

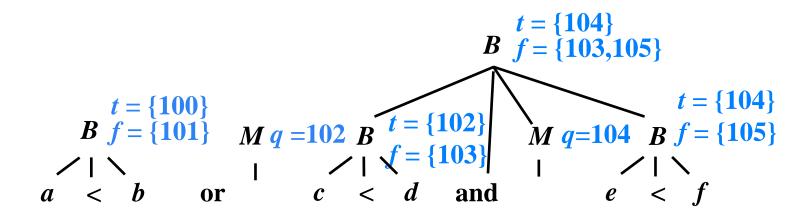
B. falselist

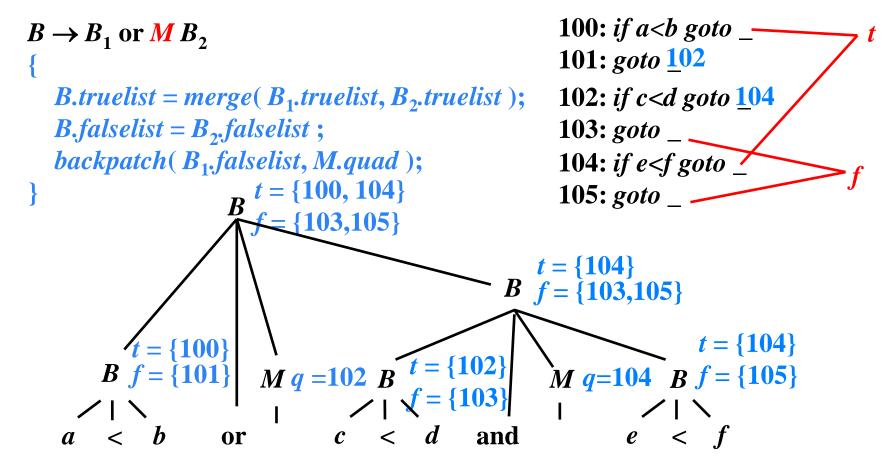
```
B \rightarrow E_1 \text{ relop } E_2 100: if a < b goto _ 101: go
```

$$\begin{array}{c}
t = \{100\} \\
B \ f = \{101\}
\end{array}$$
\(\sim 1 \)
$$a < b \ or \ c < d \ and \ e < f$$

```
100: if a<b goto _
B \rightarrow B_1 or MB_2
                                              101: goto _
                                              102: if c<d goto _
  backpatch(B_1.falselist, M.quad);
                                              103: goto _
  B.truelist = merge(B_1.truelist, B_2.truelist);
  B.falselist = B_{2}.falselist;
M \to \varepsilon
\{ M.quad = nextquad; \}
```

```
B \rightarrow B_1 \text{ and } M B_2 \\ \{B.truelist = B_2.truelist; \\ B.falselist = merge(B_1.falselist, B_2.falselist); \\ backpatch(B_1.truelist, M.quad); \\ \}
100: if a < b \ goto \\ 101: goto \\ 102: if c < d \ goto \\ 103: goto \\ 104: if e < f \ goto \\ 105: goto \\ 105:
```





控制流语句的回填

- 〉文法
 - $\triangleright S \rightarrow S_1 S_2$
 - $>S \rightarrow id = E ; | L = E ;$
 - $> S \rightarrow \text{if } B \text{ then } S_1$
 - \mid if B then S_1 else S_2
 - | while B do S_1
- 户综合属性
 - ▶S.nextlist: 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

$S \rightarrow \text{if } B \text{ then } S_1$

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

B.falselist

 S_1 .nextlist

S.nextlist

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

```
S \rightarrow \text{if } B \text{ then } M_1 S_1 N \text{ else } M_2 S_2
                                                                           if
                                                         B.truelist
                                                                                 B.falselist
  S.nextlist = merge(merge(S_1.nextlist,
                                                                       B.code
                                                                        then
   N.nextlist), S_2.nextlist);
                                                   M_1.quad
   backpatch(B.truelist, M_1.quad);
                                                                      S_1.code
                                                        S_1.nextlist
   backpatch(B.falselist, M<sub>2</sub>.quad);
                                                        N.nextlist
                                                                       goto -
N \to \varepsilon
                                                                         else
                                                      M_2, quad
   N.nextlist = makelist(nextquad);
                                                        S_2.nextlist | S_2.code
   gen('goto');
                                                           S.nextlist
```

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

```
S \rightarrow \text{while } M_1 B \text{ do } M_2 S_1
                                                 M_1.quad
                                                                  while
                                                                               B.falselist
   S.nextlist = B.falselist;
                                                 B.truelist
                                                                 B.code
   backpatch(S_1.nextlist, M_1.quad);
                                                                    do
   backpatch(B.truelist, M<sub>2</sub>.quad);
                                                  M_2.quad
                                                                 S_1.code
   gen(`goto` M_1.quad);
                                                S_1.nextlist
                                                              goto M_1.quad
                                                                        S.nextlist
```

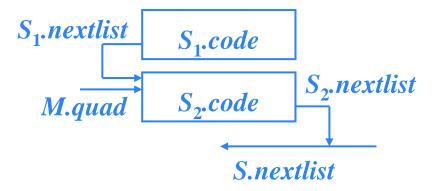
```
S \rightarrow S_1 S_2

S \rightarrow S_1 M S_2

{

S.nextlist = S_2.nextlist;

backpatch(S_1.nextlist, M.quad);
```



$$S \rightarrow id = E ; | L = E ;$$

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

回填技术SDT编写要点

- 〉文法改造
 - ▶在list箭头指向的位置设置标记非终结符M
- 产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

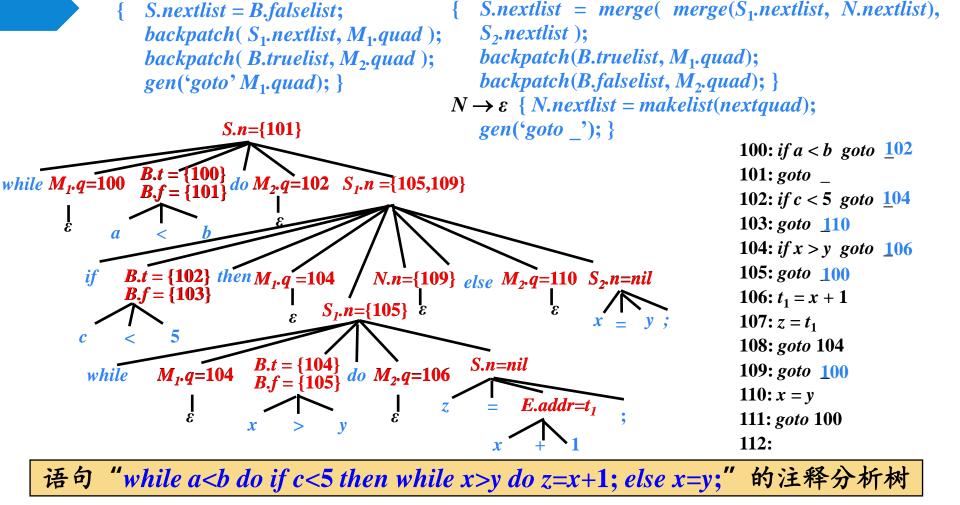
while a < b do

if c < 5 then

while $x > y \ do \ z = x + 1;$

else

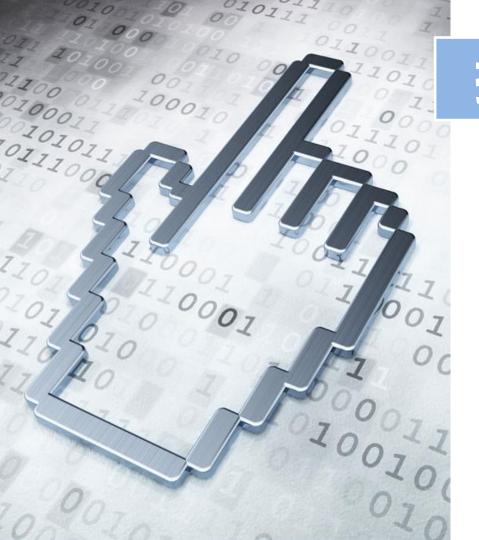
x = y;



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

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

while a < b do if c < 5 then while x > y do z = x+1; else x = y;100: if a < b goto 102 100: (j<,a,b,102)**101:** *goto* _ 101: (i, -, -, -)102: (j <, c, 5, 104)102: if c < 5 goto 104 103: goto 110 103: (j, -, -, 110)104: *if* x > y *goto* 106 104: (j>, x, y, 106)(j, -, -, 100)105: goto 100 105: 106: $(+,x,1,t_1)$ 106: $t_1 = x + 1$ 107: $(=,t_1,-,z)$ 107: $z = t_1$ 108: goto 104 108: (i, -, -, 104)109: goto 100 109: (j, -, -, 100)110: x = y110: (=,y,-,x)111: goto 100 111: (j, -, -, 100)112: 112:



提纲

- 6.1 声明语句的翻译
- 6.2 赋值语句的翻译
- 6.3 控制语句的翻译
- 6.4 回填
- 6.5 switch语句的翻译
- 6.6 过程调用语句的翻译

6.5 switch语句的翻译

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

```
E.code
           case V_1:
          if E.addr != V_1 goto L_1
                  S_1.code
                  goto next
           case V_2:
          if E.addr != V_2 goto L_2
                  S<sub>2</sub>.code
                  goto next
           case V_{n-1}:
        if E.addr !=V_{n-1} goto L_{n-1}
                  S_{n-1}.code
                  goto next
                default
L_{n-1}
                  S_n.code
next
```

switch

switch语句的另一种翻译

```
switch E
begin
case V_1: S_1
case V_2: S_2
```

• • •

case V_{n-1} : S_{n-1} default: S_n

end

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

goto test case V_1 : S_1 .code goto next case V₂: L_2 S₂.code goto next case $\underline{V_{n-1}}$: S_{n-1} .code L_{n-1} goto next default L_n S_n .code goto next test | $if E.addr = V_1 goto L_1$ if E.addr = V, goto L, if $E.addr = V_{n-1}$ goto L_{n-1} goto L_n next

switch

E.code

增加一种case指令

```
test: if t = V_1 goto L_1

if t = V_2 goto L_2

...

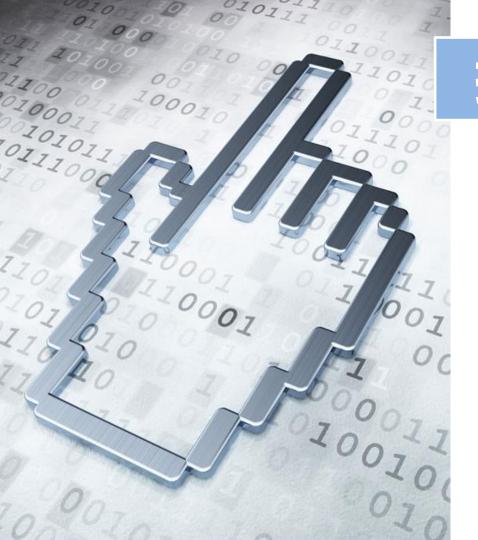
if t = V_{n-1} goto L_{n-1}

goto L_n

next:
```

```
test: case t V_1 L_1 case t V_2 L_2 ... case t V_{n-1} L_{n-1} case t t L_n
```

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



提纲

- 6.1 声明语句的翻译
- 6.2 赋值语句的翻译
- 6.3 控制语句的翻译
- 6.4 回填
- 6.5 switch语句的翻译
- 6.6 过程调用语句的翻译

6.6 过程调用的翻译

ightarrow文法 ightarrow S
ightarrow call id (Elist) Elist
ightarrow Elist, E Elist
ightarrow E

过程调用语句的代码结构

call id(E_1, E_2, \ldots, E_n)

call id (call id (E_1 .code E_1 .code $param E_1.addr$ $E_{\gamma}.code$ $E_{\gamma}.code$ param E_{2} .addr E_n .code $param E_1.addr$ E_n .code param E2.addr $param E_n.addr$ $param E_n.addr$ call id.addr n call id.addr n

过程调用语句的代码结构

call id(E_1, E_2, \ldots, E_n)

call id (E_1 .code $E_{\gamma}.code$ E_n .code $\overline{param E_1.a}ddr$ $param E_{2}$. addr $param E_n.addr$ call id.addr 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

```
\gt S \rightarrow \text{call id } (Elist)
                                                            call id (
        n=0;
                                                            E_1.code
        for q中的每个t do
                                                            E_{\gamma}.code
                 gen('param' t );
                 n = n+1:
                                                            E_n.code
        gen('call' id.addr','n);
                                                        param E_1.addr
\triangleright Elist \rightarrow E
                                                        param E_{2}.addr
        将q初始化为只包含E.addr; }
\gt{Elist} \rightarrow Elist_1, E
                                                        param E_n.addr
        将E.addr添加到q的队尾;
                                                        call id.addr n
```

例:翻译以下语句f(b*c-1,x+y,x,y)

$$t_1 = b*c$$
 $t_2 = t_1 - 1$
 $t_3 = x + y$
 $param t_2$
 $param t_3$
 $param x$
 $param y$
 $call f, 4$

本章小结

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

