



第六章 中间代码生成（2）

Intermediate Code Generation



主要内容

- 中间代码的表示
 - 抽象语法树 (AST)
 - 有向无环图 (DAG)
 - 三地址代码
- 静态类型检查
- 中间代码生成
 - 类型和声明的翻译
 - 表达式和赋值语句的翻译
 - 控制语句的翻译
 - 回填



生成表达式代码的SDD

□ 将表达式翻译成三地址指令序列

产生式	语义规则
$S \rightarrow \text{id} = E ;$	$S.\text{code} = E.\text{code} $ $\text{gen}(\text{top.get(id.lexeme)} '=' E.\text{addr})$
$E \rightarrow E_1 + E_2$	$E.\text{addr} = \text{new Temp}()$ $E.\text{code} = E_1.\text{code} E_2.\text{code} $ $\text{gen}(E.\text{addr}'=' E_1.\text{addr}'+' E_2.\text{addr})$
$ - E_1$	$E.\text{addr} = \text{new Temp}()$ $E.\text{code} = E_1.\text{code} $ $\text{gen}(E.\text{addr}'=' '\text{minus}' E_1.\text{addr})$
$ (E_1)$	$E.\text{addr} = E_1.\text{addr}$ $E.\text{code} = E_1.\text{code}$
$ \text{id}$	$E.\text{addr} = \text{top.get(id.lexeme)}$ $E.\text{code} = ''$

- 属性code表示代码
- addr表示存放表达式结果的地址（临时变量）
- top.get(...)从栈顶符号表开始，逐个向下寻找id的信息
- new Temp()可以生成一个临时变量
- gen(...)生成一个指令



增量式翻译方案

- 主属性code满足增量式翻译的条件
 - on-the-fly的代码生成

$S \rightarrow \mathbf{id} = E ; \quad \{ \text{gen}(\text{top.get(id.lexeme)} ' =' E.\text{addr}); \}$

$E \rightarrow E_1 + E_2 \quad \{ E.\text{addr} = \mathbf{new Temp}();$
 $\qquad \qquad \qquad \text{gen}(E.\text{addr} ' =' E_1.\text{addr} '+' E_2.\text{addr}); \}$

$| - E_1 \quad \{ E.\text{addr} = \mathbf{new Temp}();$
 $\qquad \qquad \qquad \text{gen}(E.\text{addr} ' =' '\mathbf{minus}' E_1.\text{addr}); \}$

$| (E_1) \quad \{ E.\text{addr} = E_1.\text{addr}; \}$

$| \mathbf{id} \quad \{ E.\text{addr} = \text{top.get(id.lexeme)}; \}$

•这里的gen()
会发出 emit
相应的代码，
而不是把代
码当做返
值



数组元素的寻址 (1)

- 假设数组元素被存放在连续的存储空间中；元素从0到n-1编号，第i个元素的地址为

$$\text{base}+i*w$$

- 其中：
 - Base为数据A的内存块的起始地址，即A[0]的相对地址
 - w为每个数组元素的宽度
 - base、w、n的值都可以从符号表中找到



数组元素的寻址 (2)

- **k 维数组的寻址**: 假设数组按行存放, 即首先存放 $A[0][i_2] \dots [i_k]$, 然后存放 $A[1][i_2] \dots [i_k]$, ...
 - 设 n_j 为第 j 维的维数
 - w_j 为第 j 维的每个子数组元素的宽度
 - w_k 为每个数组元素的宽度: $w_k = w$
 - $w_{k-1} = n_k * w_k = n_k * w$
- **$A[i_1][i_2] \dots [i_k]$ 的地址**

$$\text{base} + i_1 * w_1 + i_2 * w_2 + \dots + i_k * w_k$$

或者

$$\text{base} + (((\dots((i_1 * n_2 + i_2) * n_3 + i_3) \dots) * n_k + i_k) * w$$



数组元素的寻址 (3)

□ 多维数组的存放方法

- 行优先?
- 列优先?

□ Pascal语言的数组

- 数组A的下标为i的元素的地址

VAR a:ARRAY [low..high] OF real;

求 $a[i]$ 的地址

$$\text{base} + (i - \text{low}) * w = \text{base-low*w} + i*w$$



包含数组元素的表达式文法

□ 添加新的文法产生式

- 数组元素 **L**: $L \rightarrow L[E] \mid id[E]$
- 以数组元素为左部的赋值: $S \rightarrow L=E;$
- 数组元素作为表达式中的因子: $E \rightarrow L$



翻译方案(1)

- 对L的代码计算偏移量，将结果存放于**L.addr**所指的临时变量中
- 综合属性array记录相应数组的信息：元素类型，地址，...

```
L → id [ E ] { L.array = top.get(id.lexeme);
                  L.type = L.array.type.elem;
                  L.addr = new Temp();
                  gen(L.addr '=' E.addr '*' L.type.width); }

| L1 [ E ] { L.array = L1.array;
                  L.type = L1.type.elem;
                  t = new Temp();
                  L.addr = new Temp();
                  gen(t '=' E.addr '*' L.type.width); }
                  gen(L.addr '=' L1.addr '+' t); }
```



翻译方案(2)

□ 数组元素作为因子

- L的代码只计算了偏移量；
- 数组元素的存放地址应该根据偏移量进一步计算，即L的数组基址加上偏移量
- 使用三地址指令 $x = a[i]$

```
E → E1 + E2 { E.addr = new Temp();  
                      gen(E.addr '=' E1.addr +'+' E2.addr); }  
  
| id { E.addr = top.get(id.lexeme); }  
  
| L { E.addr = new Temp();  
          gen(E.addr '=' L.array.base '[' L.addr ']'); }
```



翻译方案(3)

- 数组元素作为赋值左部
 - 使用三地址指令 **a[i]=x**

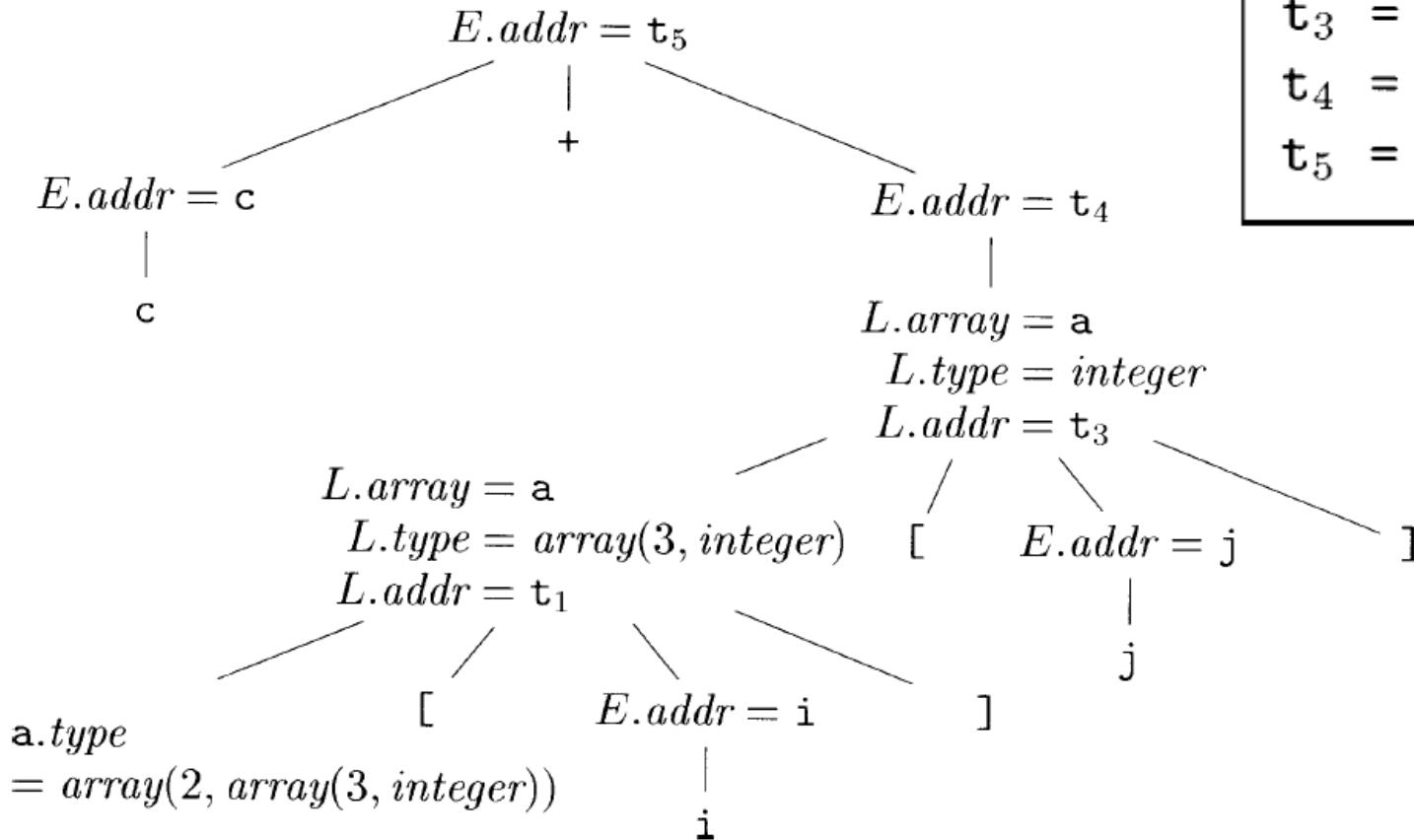
```
S → id = E ; { gen( top.get(id.lexeme) '=' E.addr); }

| L = E ; { gen(L.array.base '[' L.addr ']' '=' E.addr); }
```



数组的例子

□ 表达式: $c+a[i][j]$



```
t1 = i * 12  
t2 = j * 4  
t3 = t1 + t2  
t4 = a [ t3 ]  
t5 = c + t4
```



类型检查和转换

□ 类型系统(*type system*)：

- 给每一个组成部分赋予一个类型表达式
- 通过一组逻辑规则来表示这些类型表达式必须满足的条件

□ 设计类型系统的根本目的是用静态检查的方式来保证合法程序运行时的良行为

- 可发现错误、提高代码效率、确定临时变量的大小、.....

□ 类型系统的形式化

- 类型表达式、定型断言、定型规则



类型检查规则分类

□ 类型综合

- 根据子表达式的类型构造出表达式的类型

if f 的类型为 $s \rightarrow t$ 且 x 的类型为 s
then $f(x)$ 的类型为 t

□ 类型推导

- 根据语言结构的使用方式来确定该结构的类型

if $f(x)$ 是一个表达式

then 对于某些类型 α, β ; f 的类型为 $\alpha \rightarrow \beta$ 且 x 的类型为 α

□ α, β 可以是未知类型



类型转换

□ 假设在表达式 $x*i$ 中， x 为浮点数、 i 为整数，则结果应该是浮点数

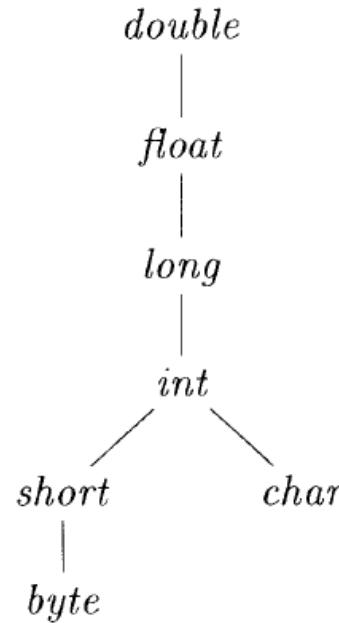
- x 和*i*使用不同的二进制表示方式
- 浮点*和整数*使用不同的指令
- $t1 = (\text{float}) i$
- $t2 = x \text{ fmul } t1$

□ 处理简单的类型转换的SDD

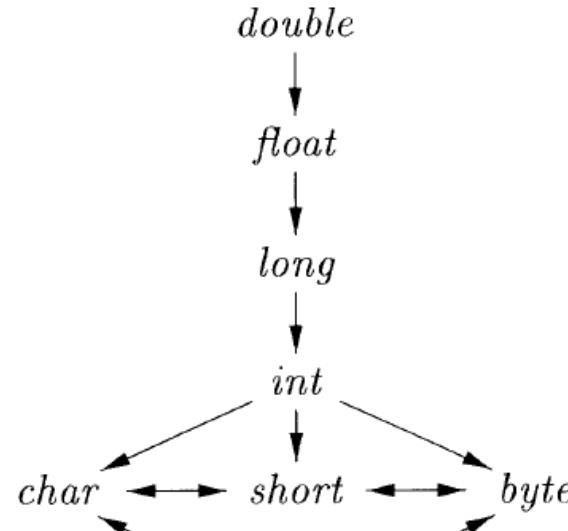
```
E → E1 + E2 {  
    if(E1.type = integer and E2.type = integer) E.type = integer;  
    else if (E1.type = float and E2.type= integer) E.type = float;}  
}
```

类型的widening和narrowing

□ Java的类型转换规则



a) 拓宽类型转换



b) 窄化类型转换

- 编译器自动完成的转换为隐式转换(coercion)
- 程序员用代码指定的强制转换为显式转换(cast)



处理类型转换的SDT

```
 $E \rightarrow E_1 + E_2 \quad \{ \quad E.type = max(E_1.type, E_2.type);$ 
 $a_1 = widen(E_1.addr, E_1.type, E.type);$ 
 $a_2 = widen(E_2.addr, E_2.type, E.type);$ 
 $E.addr = \text{new Temp}();$ 
 $gen(E.addr '=' a_1 '+' a_2); \}$ 
```

```
Addr widen(Addr a, Type t, Type w)
  if ( t = w ) return a;
  else if ( t = integer and w = float ) {
    temp = new Temp();
    gen(temp '=' '(float)' a);
    return temp;
  }
  else error;
}
```

- 函数**max**求的是两个参数在拓宽层次结构中的最小公共祖先
- **widen**函数已经生成了必要的类型转换代码



函数/运算符的重载

- 通过查看参数来解决函数重载问题
- $E \rightarrow f(E_1)$
{ if $f.typeSet = \{s_i \rightarrow t_i \mid 1 \leq i \leq k\}$ and $E_1.type = s_k$
then $E.type = t_k$ }



控制流的翻译

- 布尔表达式可以用于改变控制流/计算逻辑值
- 文法
 - $B \rightarrow B \parallel B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$
- 语义
 - $B_1 \parallel B_2$ 中 B_1 为真时，不用计算 B_2 ，整个表达式为真 → 当 B_1 为真时应该跳过 B_2 的代码
 - $B_1 \&\& B_2$ 中 B_1 为假时，不用计算 B_2 ，整个表达式为假
- 短路代码
 - 通过跳转指令实现控制流的处理
 - 逻辑运算符本身不在代码中出现



短路代码的例子

□ 语句

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

□ 代码

<code>if</code>	<code>x < 100</code>	<code>goto L2</code>
<code>ifFalse</code>	<code>x > 200</code>	<code>goto L1</code>
<code>ifFalse</code>	<code>x != y</code>	<code>goto L1</code>

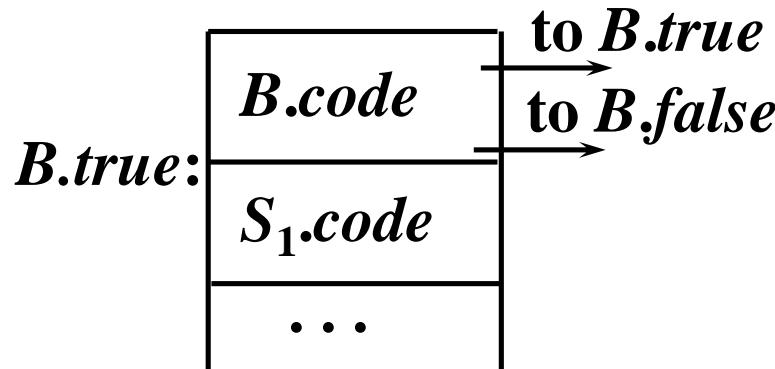
`L2: x=0`

`L1: 接下来的代码`

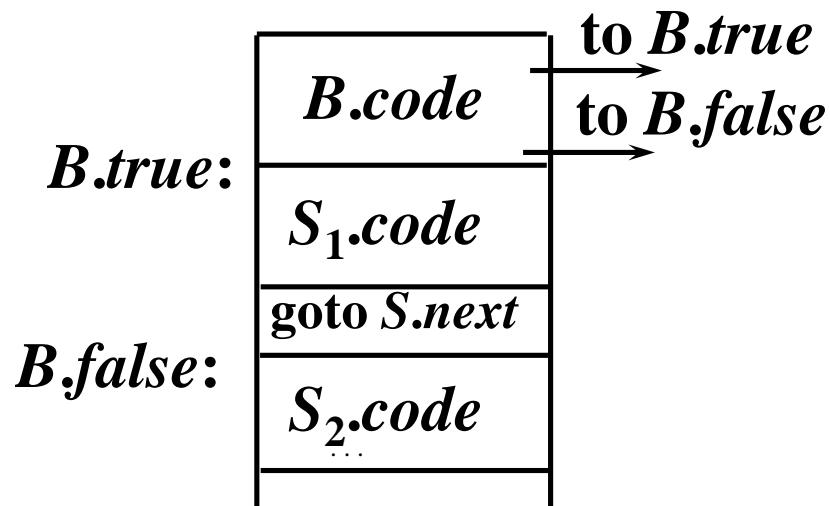
- 当 $x < 100$ 为真时，直接执行 $x = 0$ ；
- 所以执行 $x > 200$ 时， $x < 100$ 必然为假
- 同理：计算 $x \neq y$ 时， $x < 100$ 为假，而 $x > 200$ 为真



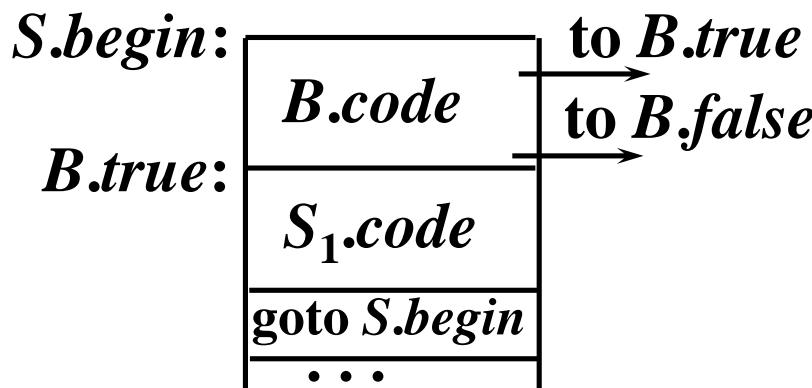
控制流语句的中间代码结构



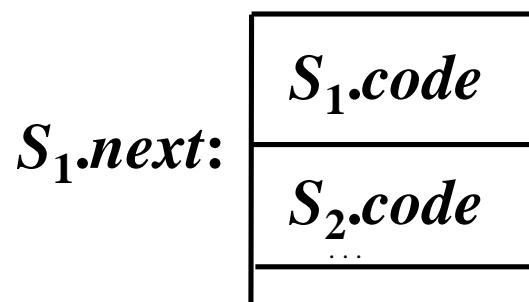
(a) **if (B) S_1**



(b) **if (B) S_1 else S_2**



(c) **while (B) S_1**



(d) **$S_1 S_2$**

- **$B.true$:** B 为真的跳转目标
- **$B.false$:** B 为假的跳转目标
- **$S.next$:** S 执行完毕时的跳转目标



控制流语句的SDD (1)

产生式	语义规则
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{if} (B) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \parallel label(B.true) \parallel S_1.code$ $\quad \parallel gen('goto' S.next)$ $\quad \parallel label(B.false) \parallel S_2.code$



控制流语句的SDD (2)

$S \rightarrow \text{while} (B) S_1$

```
begin = newlabel()
B.true = newlabel()
B.false = S.next
S1.next = begin
S.code = label(begin) || B.code
|| label(B.true) || S1.code
|| gen('goto' begin)
```

$S \rightarrow S_1 S_2$

```
S1.next = newlabel()
S2.next = S.next
S.code = S1.code || label(S1.next) || S2.code
```

改为用增量式生成代码的方式：

$S \rightarrow \text{while} (\{ \text{begin} = \text{newlabel}(); B.\text{true} = \text{newlabel};$
 $\quad \quad \quad B.\text{false} = S.\text{next}; \text{emit}(\text{begin} ':'); \}$
 $\quad \quad \quad B \} \{ \text{emit}(B.\text{true} ':'); S1.\text{next} = \text{begin}; \} \quad S1 \{ \text{emit}('goto' begin); \}$



布尔表达式的控制流翻译

- 生成的代码执行时跳转到两个标号之一。
 - 表达式的值为真时，跳转到**B.true**
 - 表达式的值为假时，跳转到**B.false**
- **B.true**和**B.false**是两个继承属性，根据**B**所在的上下文指向不同的位置
 - 如果**B**是if语句的条件表达式，分别指向then分支和else分支；如果没有else分支，则指向if语句的下一条指令
 - 如果**B**是while语句的条件表达式，分别指向循环体的开头和循环出口处

生成作为控制条件的布尔表达式中间代码的SDD



产生式	语义规则
$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}; B_1.\text{false} = \text{newlabel}();$ $B_2.\text{true} = B.\text{true}; B_2.\text{false} = B.\text{false};$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
$B \rightarrow B_1 \&& B_2$	$B_1.\text{true} = \text{newlabel}(); B_1.\text{false} = B.\text{false};$ $B_2.\text{true} = B.\text{true}; B_2.\text{false} = B.\text{false};$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$
$B \rightarrow ! B_1$	$B_1.\text{true} = B.\text{false}; B_1.\text{false} = B.\text{true};$ $B.\text{code} = B_1.\text{code}$
$B \rightarrow (B_1)$	$B_1.\text{true} = B.\text{true}; B_1.\text{false} = B.\text{false};$ $B.\text{code} = B_1.\text{code}$
$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} =$ $\quad \text{gen(' if ' } E_1.\text{addr } \text{rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\quad \parallel \text{gen('goto' } B.\text{false})$
$B \rightarrow \text{true}$	$B.\text{code} = \text{gen('goto ' } B.\text{true})$
$B \rightarrow \text{false}$	$B.\text{code} = \text{gen('goto ' } B.\text{false})$



布尔表达式代码的例子

- **if (x<100 || x > 200 && x!= y) x = 0;** 的代码

```
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:
```



布尔值和跳转代码

- 程序中出现布尔表达式的目的也有可能就是求出它的值。比如 $x=a < b$
- 处理方法：
 - 首先建立表达式的语法树，然后根据表达式的不同角色来处理。
- 文法：
 - $S \rightarrow id = E ; \mid if (E) S \mid while (E) S \mid S\ S$
 - $E \rightarrow E \# E \mid E \&& E \mid E \text{ rel } E \mid \dots$
- 根据E的语法树结点所在的位置：
 - $S \rightarrow while (E) S_1$ 中的E，生成跳转代码
 - 对于 $S \rightarrow id = E$ ，生成计算右值的代码



回填 (1)

- 为布尔表达式和控制流语句生成目标代码的关键问题：**某些跳转指令应该跳转到哪里？**
- 例如： **if (B) S**
 - 按照短路代码的翻译方法， B的代码中有一些跳转指令在B为假时执行，
 - 这些跳转指令的目标应该跳过S对应的代码。生成这些指令时， S的代码尚未生成，因此目标不确定
 - 如果通过语句的继承属性**next**来传递，当中间代码不允许符号标号时，则需要第二趟处理。
- 如何一趟处理完毕呢？



回填 (2)

□ 基本思想：

- 记录B的代码中跳转指令 `goto S.next, if ... goto S.next` 的位置，但是不生成跳转目标
- 这些位置被记录到B的综合属性 `B.falseList` 中
- 当 `S.next` 的值已知时（即S的代码生成完毕时），把 `B.falseList` 中的所有指令的目标都填上这个值

□ 回填技术：

- 生成跳转指令时暂时不指定跳转目标标号，而是使用列表记录这些不完整的指令
- 等知道正确的目标时再填写目标标号
- 每个列表中的指令都指向同一个目标
 - `truelist, falselist, nextlist`



布尔表达式的回填翻译 (1)

改写布尔表达式文法：

- (1) $B \rightarrow B1 \parallel M B2$
- (2) | $B1 \&& M B2$
- (3) | $! B1$
- (4) | $(B1)$
- (5) | $E1 \text{ rel } E2$
- (6) | true
- (7) | false
- (8) $M \rightarrow \epsilon$

- 插入非终结符号M是为了引入一个语义动作，以便获得即将产生的下一个三地址语句的索引（编号）。



布尔表达式的回填翻译（2）

- 布尔表达式用于语句的控制流时，它总是在取值true时和取值false时分别跳转到某个位置
- 引入两个综合属性
 - **truelist**: 包含跳转指令（位置）的列表，这些指令在取值true时执行
 - **falselist**: 包含跳转指令（位置）的列表，这些指令在取值false时执行
- 辅助函数
 - **makelist(i)**: 构造一个列表
 - **merge(p1,p2)**: 合并两个列表
 - **backpatch(p,i)**: 用i回填p指向的语句列表中的跳转语句的跳转地址



布尔表达式的回填翻译 (3)

$B \rightarrow B1 \parallel M B2$

```
{ backpatch(B1.falselist, M.instr);
  B.truelist=merge(B1.truelist, B2.truelist);
  B.falselist=B2.falselist;
}
```

$B \rightarrow B1 \&& M B2$

```
{ backpatch(B1.truelist, M.instr);
  B.truelist=B2.truelist;
  B.falselist=merge(B1.falselist, B2.falselist);
}
```

$B \rightarrow ! B1 \{ B.truelist=B1.falselist;
 B.falselist=B1.truelist \}$



布尔表达式的回填翻译 (4)

B → (B1) { B.truelist= B1.truelist;
B.falselist=B1.falselist; }

B → E1 rel E2

{ B.truelist= makelist(nextinstr);
B.falselist= makelist(nextinstr+1);
emit('if' E1.addr rel.op E2.addr 'goto ____');
emit('goto ____'); }

M→ε { M.instr=nextinstr;}

B→true { B.truelist=makelist(nextinstr);
 emit('goto ____'); B.falselist=null;}

B→false { B.falselist=makelist(nextinstr);
 emit('goto ____'); B.truelist=null;}



控制流语句的翻译方案

文法：

- (1) $S \rightarrow \text{if } (B) \ S$
- (2) | $\text{if } (B) \ S \ \text{else } S$
- (3) | $\text{while } (B) \ S$
- (4) | $\{ L \}$
- (5) | A
- (6) $L \rightarrow L \ S$
- (7) | S

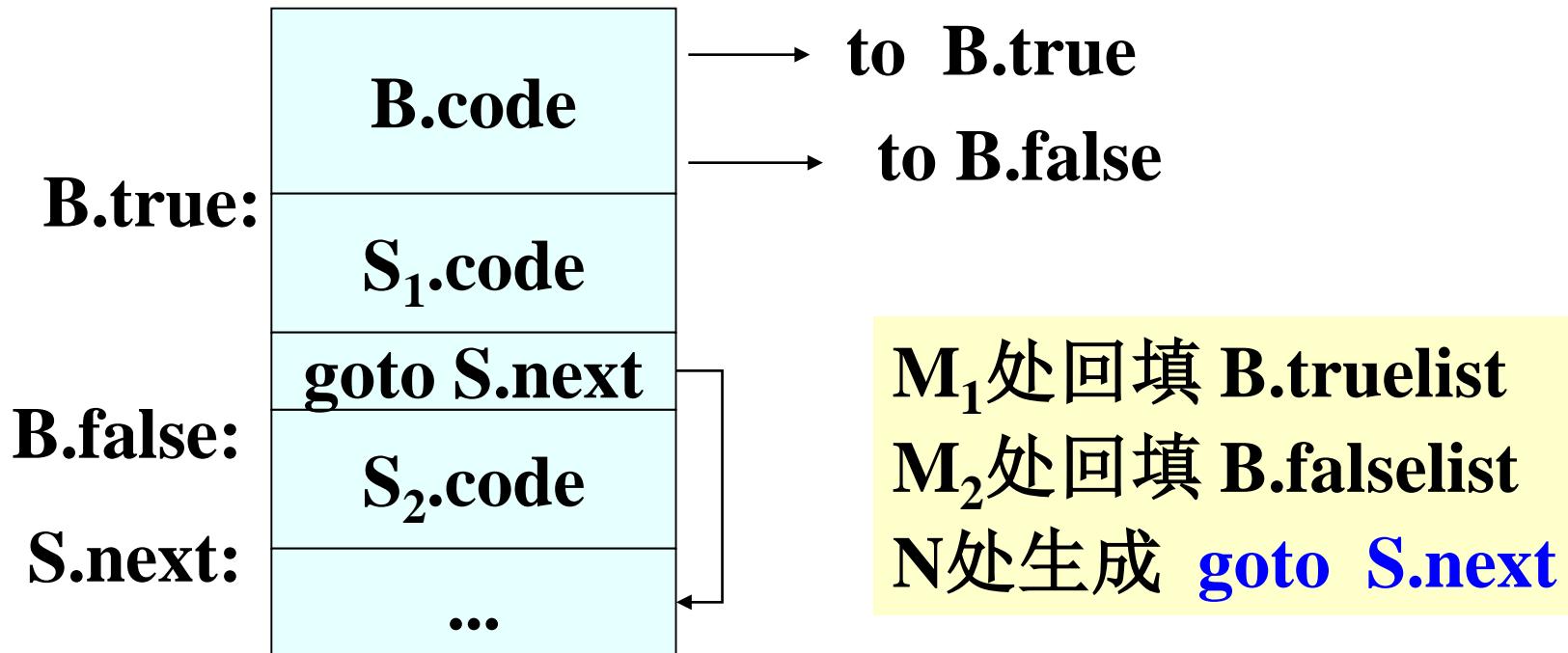
S 表示语句
A 为赋值语句

L 表示语句串
B 为布尔表达式



续：控制流语句的翻译方案：

1. $S \rightarrow \text{if } (B) M_1 S_1 N \text{ else } M_2 S_2$



- 为语句添加综合属性: **nextlist**
- **nextlist**中的跳转指令的目标是S执行完之后紧接着执行的下一条指令的位置。



续：控制流语句的翻译方案：

1. $S \rightarrow if (B) M_1 S_1 N else M_2 S_2$

```
{ backpatch( B.truelist, M1.instr);  
  backpatch( B.falselist, M2.instr);  
  temp = merge(S1.nextlist, N.nextlist);  
  S.nextlist= merge(temp, S2.nextlist); }
```

2. $S \rightarrow if (B) then M S_1$

```
{ backpatch (B.truelist, M.instr);  
  S.nextlist= merge(B.falselist, S1.nextlist); }
```

3. $N \rightarrow \epsilon \{ N.nextlist= makelist(nextinstr);$

```
  emit( 'goto—'); }
```

4. $M \rightarrow \epsilon \{ M.instr =nextinstr; \}$



续：控制流语句的翻译方案：

5. $S \rightarrow \text{while } M_1 \text{ (B) do } M_2 \text{ } S_1$

/* M_1 处生成标号 $S.\text{begin}$ ，回填 $S_1.\text{nextlist}$ 。

M_2 处回填 $B.\text{truelist}$ 。 */

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

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

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

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

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



例1：

```
if ((x+y>z) && (a==b))  
    while m<n { m=n+10; }  
else { a=b-m; }
```

相应的中间代码如下：

(1) t1=x+y
(2) if t1>z goto (4)
(3) goto (12)
(4) if a==b goto (6)
(5) goto (12)

(6) if m<n goto (8)
(7) goto ()
(8) t2=n+10
(9) m=t2
(10) goto (6)
(11) goto ()

(12) t3=b-m
(13) a=t3

S.nextlist={7,11}

显然，(11)句是多余的，在后面优化中会被删除



例2：语句 {

```
while (!((x<=y) || (z>=x))) do
    if (a !=b) {x=a+b;} else
        {while (y>100) { y=x-1;}}
    a=x+y;
```

}

相应的中间代码如下：

(1) if x<=y goto(16)
(2) goto (3)
(3) if z>=x goto (16)
(4) goto (5)

(5) if a!=b goto(7)
(6) goto (10)
(7) t1=a+b
(8) x=t1
(9) goto (1)

S.nextlist={ }

(10) if y>100 goto(12)
(11) goto (1)
(12) t2=x-1
(13) y=t2
(14) goto (10)
(15) goto (1)
(16) t3=x+y
(17) a=t3

其中多余的语句，如2, 4, 15等在后面的优化中会被删除。



Break、Continue的处理

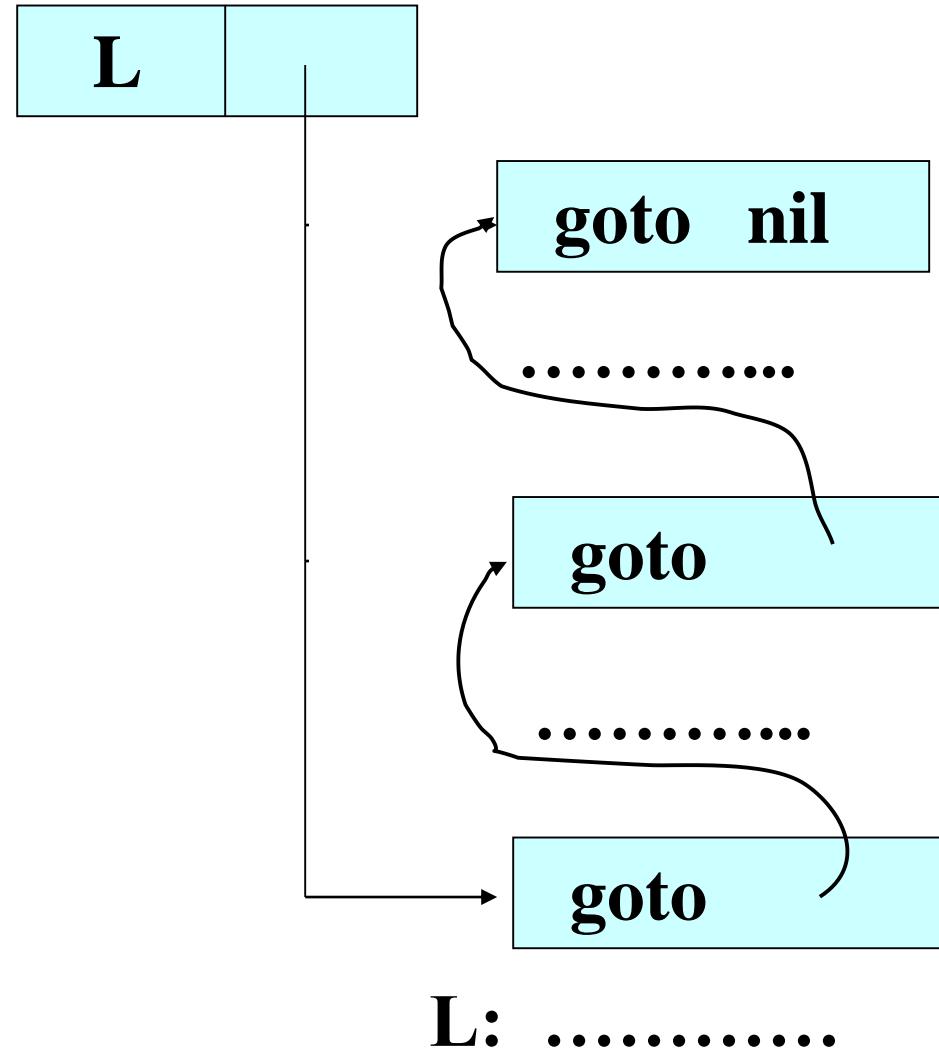
- 虽然break、continue在语法上是一个独立的句子，但是它们的代码和外围语句相关
- 方法(break语句)
 - 跟踪外围语句S
 - 生成一个跳转指令坯
 - 将这个指令坯的位置加入到S的nextlist中
- 跟踪的方法
 - 在符号表中设置break条目，令其指向外围语句
 - 在符号表中设置指向S的nextlist的指针，然后把这个指令坯的位置直接加入到nextlist中

标号和转移语句



```
.....  
goto L;  
.....  
goto L;  
.....  
goto L;  
.....  
L: .....
```

拉链—回填





switch语句的翻译

假定switch语句的形式如下：

```
switch (E) {  
    case v1: LS1  
    case v2: LS2  
    .....  
    case vn: LSn  
    default: LS  
}
```



Switch语句的中间代码形式

计算E的代码;

goto test;

L₁: 语句串LS₁的中间代码LS₁.code;

 goto next;

L₂: 语句串LS₂的中间代码LS₂.code;

 goto next;

.....

L_n: 语句串LS_n的中间代码LS_n.code;

 goto next;

L: 语句串LS的中间代码LS.code;

 goto next;

test: if E.place==v1 goto L1;

 if E.place==v2 goto L2;

.....

 if E.place==vn goto Ln;

 goto L;

next:



描述switch语句的生成式

$S \rightarrow \text{switch } (E) \{ M \text{ default:}L \}$

$M \rightarrow M \text{ case } C:L \mid \text{case } C:L$

$L \rightarrow L1 S \mid S$

- 为了构造switch语句的翻译方案，设置一个队列变量q
- q的元素是记录，包含c和d两个成员，分别用于存储case后面的常量值v和各语句串中间代码第一个三地址语句地址，以便生成test后面的条件转移语句时使用
- S为语句



switch语句的翻译方案

$S \rightarrow \text{switch } (E) H \{ M \text{ default:} F L \}$

{ S.nextlist =

merge(M.nextlist, L.nextlist, makelist(nextinstr));

emit('goto-'); backpatch(H.list, nextinstr);

对于队列q中的每个元素t,

执行gen('if' E.addr'=='t.c' goto' t.d);

emit('goto' F.instr);

}

$H \rightarrow \epsilon$ { 把队列q初始化为空队列;

H.list=makelist(nextinstr); emit('goto-');

}

$F \rightarrow \epsilon$ {F.instr=nextinstr;}



M→case C:F L

```
{ t.c=C.val; t.d=F.instr; 把t插入队列q;  
M.nextlist=merge(L.nextlist,makelist(nextinstr));  
emit('goto-');  
}
```

M→M1 case C: F L

```
{ t.c=C.val;t.d=F.instr; 把t插入队列q;  
M.nextlist=  
    merge(M1.nextlist,L.nextlist,makelist(nextinstr));  
emit('goto-');  
}
```

L→S

{L.nextlist=S.nextlist;}

L→L1 F S

{backpatch(L1.nextlist,F.instr);
L.nextlist=S.nextlist; }



switch语句的例子

```
switch (a*b+c) {  
    case 20: if ((x>y) && (x>z)) {x=y-12;} else {z=x+y;}  
        while (x>y) x=x-1;  
    case 10: a=b+c; if (m==n) m=a*n;  
    case 30: x=m+n; switch x{  
        case 100: if (a>b) a=a-1;  
        case 200: if (b !=c) {b=m+2;} c=n*10;  
        default: x=a+b;  
    }  
    default: while (x<z) x=x*2;  
}
```

(1) $t1=a*b$
(2) $t2=t1+c$
(3) goto (54)
(4) if $x>y$ goto (6)
(5) goto (11)
(6) if $x>z$ goto (8)
(7) goto (11)
(8) $t3=y-12$
(9) $x=t3$
(10) goto (13)
(11) $t4=x+y$
(12) $z=t4$
(13) if $x>y$ goto (15)
(14) goto ()
(15) $t5=x-1$
(16) $x=t5$
(17) goto (13)
(18) goto ()

(19) $t6=b+c$
(20) $a=t6$
(21) if $m==n$ goto (23)
(22) goto ()
(23) $t7=a*n$
(24) $m=t7$
(25) goto ()

(48) if $x<z$ goto (50)
(49) goto ()
(50) $t13=x^2$
(51) $x=t13$
(52) goto (48)
(53) goto ()
(54) if $t2==20$ goto(4)
(55) if $t2==10$ goto(19)
(56) if $t2==30$ goto(26)
(57) goto (48)

(26) $t8=m+n$
(27) $x=t8$
(28) goto (44)
(29) if $a>b$ goto (31)
(30) goto ()
(31) $t9=a-1$
(32) $a=t9$
(33) goto ()
(34) if $b!=c$ goto (36)
(35) goto (38)
(36) $t10=m+2$
(37) $b=t10$
(38) $t11=n^{*}10$
(39) $c=t11$
(40) goto ()
(41) $t12=a+b$
(42) $x=t12$
(43) goto ()
(44) if $x==100$ goto(29)
(45) if $x==200$ goto (34)
(46) goto (41)
(47) goto()

S.nextlist={14, 18, 22, 25, 30, 33, 40, 43, 47, 49, 53}



本章小结

- 中间代码通常使用三地址代码来表示
 - 三地址代码的实现形式
- 掌握产生如下语句中间代码的翻译方案
 - 赋值语句
 - 含有数组与记录变量的赋值语句
 - 布尔表达式
 - 作为控制条件的布尔表达式
 - if, while 等控制语句的翻译
 - switch, do-while, for语句？



作业

- 11月20日交
- 表达式和类型
 - 6.4.3(2), 6.4.6(2), 6.4.8(2)
 - 6.5.2
 - 本题文法: $S \rightarrow id = E \quad E \rightarrow E1 + E2 \mid -E1 \mid (E1) \mid id \mid E1(E2)$
- 控制流语句
 - 6.6.1
 - 6.7.3 (题目有误, 这里应该使用nextlist, truelist 和 falselist)



补充作业（选做）

□ Ex. 6-1

设有说明

```
int a[5][6][9];
```

```
int i,j,k;
```

给出下面语句的三地址中间代码：

```
a[i*2+j, a[2,i+j,k]+i, i*4+k*2] = i*j + k*a[0,4,2*k]
```

□ Ex. 6-2

给出语句 S: `if ((a+b*c > m+m) && (x==y)) {while (m>n)
m=m-2;} else {a=b+c;}` 的三地址中间代码和 S.nextlist。

□ Ex. 6-3

给出语句 S: `while ((a+b*c > x+y) && (m==n)) { if
(x<=y) { while (a<b) { a=a+10; b=c*m; } } else a=b+c; }`
的三地址中间代码和 S.nextlist。