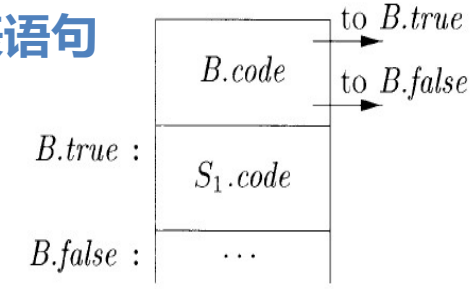
Lecture 12: 中间代码生成-II

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

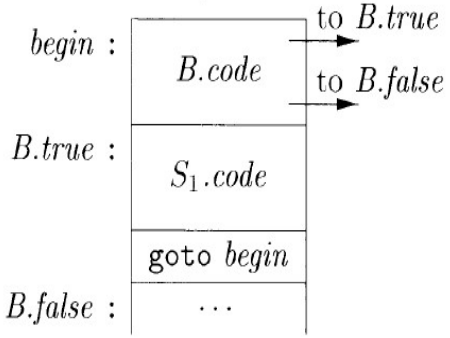
控制流翻译

控制流语句的翻译

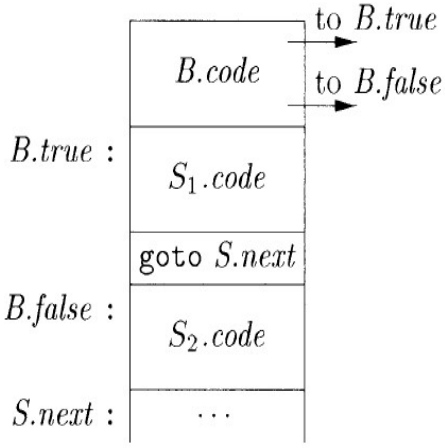
- **文法：B表示布尔表达式，S代表语句**
  - S → **if** (B) S_1
  - S → **if** (B) S_1 **else** S_2
  - S → **while** (B) S_1
- **代码的布局见右图**
- **继承属性**
  - B.true：B为真的跳转目标
  - B.false：B为假的跳转目标
  - S.next：S执行完毕时的跳转目标



a) if



c) while



b) if-else

控制流语句的翻译

控制流语句的翻译

$S \rightarrow$ if B then S_1

| if B then S_1 else S_2

| while B do S_1

| S_1; S_2

产生式	语义规则
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

图 6-36　控制流语句的语法制导定义

控制流语句的翻译

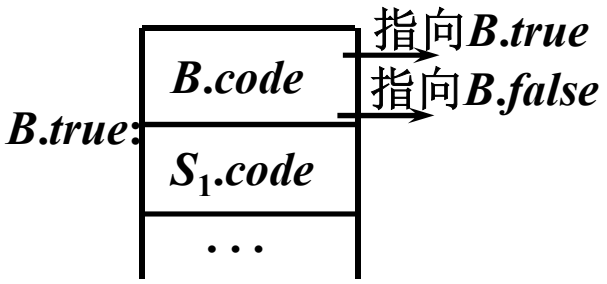
$S \rightarrow$ if B then S_1

{*B.true = newLabel*();

B.false = S.next;

S_1*.next = S.next*;

S.code = B.code || *gen*(*B.true*, ':') || S_1*.code* }



(a) if-then

label(B.true)

控制流语句的翻译

$S \rightarrow$ if B then S_1 else S_2

{*B.true* = *newLabel*();

B.false = *newLabel*();

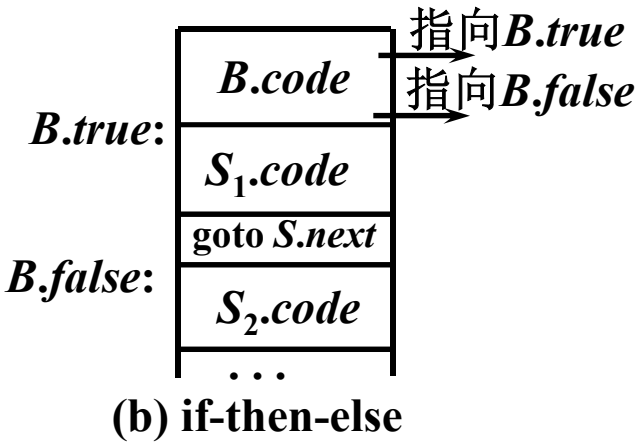
S_1.*next* = *S.next*;

S_2.*next* = *S.next*;

S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code* ||

gen('goto', *S.next*) || *gen*(*B.false*, ':') ||

S_2.*code*}



(b) if-then-else

控制流语句的翻译

$S \rightarrow$ while B do S_1

{*S.begin = newLabel*();

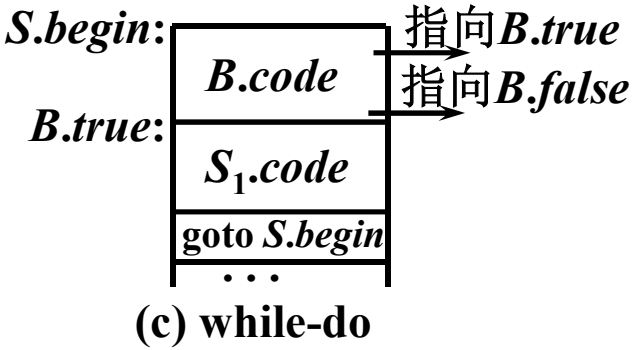
B.true = newLabel();

B.false = S.next;

S_1*.next = S.begin*;

S.code = gen(*S.begin*, ':') || *B.code* ||

gen(*B.true*, ':') || S_1*.code* || *gen*('goto', *S.begin*) }



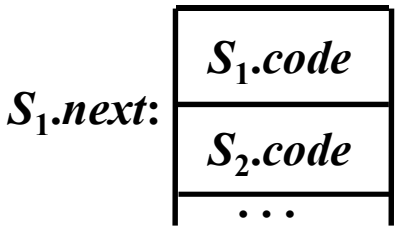
(c) while-do

控制流语句的翻译

$S \rightarrow S_1; S_2$

{$S_1.next = newLabel()$; $S_2.next = S.next$;

$S.code = S_1.code \,||\, gen(S_1.next, ':') \,||\, S_2.code$ }



(d) $S_1; S_2$

布尔表达式翻译

布尔表达式

- **布尔表达式有两个基本目的**
  - 计算逻辑值
  - 在控制流语句中用作条件表达式
- **本节所用的布尔表达式文法**

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

布尔表达式

布尔表达式的完全计算

- 布尔值计算和算数表达式计算非常类似，可仿照算数表达式翻译方法，为每个产生式写出语义子程序：值的表示数值化

产生式	语义子程序
(1)$E \rightarrow E_a^{(1)}$ rop $E_a^{(2)}$	{T=NEWTEMP; GEN(rop, $E_a^{(1)}$•PLACE , $E_a^{(2)}$•PLACE ,T); E•PLACE =T }
(2)$E \rightarrow E^{(1)}$ bop $E^{(2)}$	{T=NEWTEMP; GEN(bop, $E_a^{(1)}$•PLACE , $E_a^{(2)}$•PLACE ,T); E•PLACE =T }

效率低下！！

(3)$E \rightarrow \neg E^{(1)}$	{T=NEWTEMP; GEN(¬ , $E^{(1)}$•PLACE , _ ,T); E•PLACE =T }
(4)$E \rightarrow (E^{(1)})$	{E•PLACE =$E^{(1)}$•PLACE }
(5)$E \rightarrow i$	{E•PLACE = ENTRY(i)}

布尔表达式

- **如何优化？布尔表达式的“短路”计算**
  - 用控制流来实现计算，即用程序中的位置来表示值，因为布尔表达式通常用来决定控制流走向
  - B_1 or B_2　定义成　if B_1 then true else B_2
  - B_1 and B_2 定义成　if B_1 then B_2 else false

两种不同计算方式会导致程序的结果不一样
有时B中有副作用

布尔表达式的控制流语句的翻译

- **布尔表达式可以用于改变控制流/计算逻辑值**
  - B → B||B | B && B | !B | (B) | E **rel** E | **true** | false
- **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
- **语义**
  - $B_1||B_2$中B_1为真时，不计算B_2，整个表达式为真，因此，当B_1为真时可以跳过B_2的代码：定义成 if B_1 then true else B_2
  - $B_1\&\&B_2$中B_1为假时，可以不计算B_2，整个表达式为假：定义成 if B_1 then B_2 else false

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

布尔表达式的控制流翻译

产生式	语义规则
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ // 短路 $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ // 短路 $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$

布尔表达式的控制流翻译

$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\|\| \ gen('\texttt{goto}' \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\texttt{goto}' \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\texttt{goto}' \ B.false)$

如果*B*是*a* < *b*的形式，
那么代码是：

if *a* < *b* goto *B.true*
goto *B.false*

布尔表达式的控制流语句的翻译

- **例 表达式*a* < *b* or *c* < *d* and *e* < *f* 的代码是：**

if $a < b$ goto L_{true}

goto L_1

L_1: if $c < d$ goto L_2

goto L_{false}

L_2: if $e < f$ goto L_{true}

goto L_{false}

场景一：布尔表达式用于决定控制流走向

- **以 if (B) then S1 else S2 为例进行说明**
  - if (T) then S1 else S2
  - if (E1 rop E2) then S1 else S2
  - if (B1 || B2) then S1 else S2
  - if (B1 && B2) then S1 else S2
  - if (!B) then S1 else S2

场景一：布尔表达式用于决定控制流走向

- **短路代码例子：if (x<100 || x>200 && x!= y) x = 0;**

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

生成的中间代码

```
        if x < 100 goto L2
        if False x > 200 goto L1
        if False x != y goto L1
L2:     x = 0
L1:     接下来的代码
```

优化过的中间代码

场景二：布尔表达式用于赋值

- **程序中出现布尔表达式的目的可能就是求出它的值**

x=a<b && c<d

```
        ifFalse a < b goto L1
        ifFalse c < d goto L1
        t = true
        goto L2
L1:     t = false
L2:     x = t
```

图 6-42　通过计算一个临时变量的值来翻译一个布尔类型的赋值语句

场景三：General cases

- **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 → id = E; | if (E) S | while (E) S | S S
  - E → E||E | E && E | E rel E | …
- **根据E的语法树结点所在的位置**
  - S → while (E) S_1中的E，生成跳转代码
  - 对于S → id = E，生成计算右值的代码

Switch语句的翻译

Switch语句翻译

- **分支数较少时**

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

t := E的代码
if t≠ V_1 goto L_1
S_1 的代码
goto next
L_1: if t≠ V_2 goto L_2
S_2 的代码
goto next
L_2: ...
...
L_{n-2}: if t≠ V_{n-1} goto L_{n-1}
S_{n-1}的代码
goto next
L_{n-1}: S_n 的代码
next:

Switch语句翻译

- **分支较多时，将分支测试代码集中在一起，便于生成较好的分支测试代码**

```
        t := E的代码
        goto test
L1:     S1 的代码
        goto next
L2:     S2 的代码
        goto next
        ...
Ln-1:   Sn-1的代码
        goto next
Ln:     Sn 的代码
        goto next
test:   if t= V1 goto L1
        if t= V2 goto L2
        ...
        if t=Vn-1 goto Ln-1
        goto Ln
next:
```

Switch语句翻译

- **中间代码增加一种case语句，便于代码生成器对它进行特别处理**

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

简洁的办法，构造一个关系对照表，表中每个关系包含一个敞亮和对应的**label**，运行时进行查找。

N较小的时候，效率可接受
N较大的时候：二分查找，**hash table**，为每个可能值建立一个单元空间

过程调用的翻译

过程调用的翻译

例 6.25 假定 a 是一个整数数组，并且 f 是一个从整数到整数的函数。那么赋值语句

```
n = f(a[i]);
```

可以被翻译成如下的三地址代码。

```
1)  t1 = i * 4
2)  t2 = a [ t1 ]
3)  param t2
4)  t3 = call f, 1
5)  n = t3
```

语言中函数定义

D	$\rightarrow$	**define** T **id** (F) { S }
F	$\rightarrow$	ϵ | T **id** , F
S	$\rightarrow$	**return** E ;
E	$\rightarrow$	**id** (A)
A	$\rightarrow$	ϵ | E , A

D定义了函数的signature;
F定义了形参；
S定义了(返回)语句；
E定义了表达式，包含函数调用；
A定义了实参

过程调用翻译需要考虑的点

- **函数类型：包含返回类型和形参类型 --- 在其列表上应用fun构造算子构造；**
- **符号表：用栈来管理**
- **类型检查：注意类型转换的规则**
- **函数调用：为id(E, E, …, E)生成三地址代码时，只需对各个参数E生成三地址执行，将所有E.addr放到一个队列中，然后为每个参数生成一条param指令，清除队列**

回填

回填（1）

很重要

- **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
  - 通过语句的继承属性next来传递。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

布尔表达式的回填翻译（1）

- **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)：创建一个只包含i的列表
  - Merge(p1,p2)：将p1和p2指向的列表合并
  - 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

布尔表达式的回填翻译（2）

1)	$B \rightarrow B_1 \ \| \| \ M \ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \rightarrow B_1 \ \&\& \ M \ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \rightarrow \ ! \ B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \rightarrow (\ B_1 \ )$	{ $B.truelist = B_1.truelist$; $B.falselist \ = \ B_1.falselist$; }
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}' \ E_1.addr$ **rel**$.op \ E_2.addr \ '\texttt{goto} \ \_')$; $gen('\texttt{goto} \ \_')$; }
6)	$B \rightarrow$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto} \ \_')$; }
7)	$B \rightarrow$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto} \ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

回填和非回填方法的比较

- 回填时生成指令坯，然后加入相应的list
- 原来跳转到B.true的指令，现在被加入到B.truelist中

$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$
$B \rightarrow$ **true**	$B.code = gen(\text{'goto'}\ B.true)$
$B \rightarrow$ **false**	$B.code = gen(\text{'goto'}\ B.false)$

$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ \_')$; $gen(\text{'goto'}\ \_')$; }
$B \rightarrow$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen(\text{'goto'}\ \_')$; }
$B \rightarrow$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen(\text{'goto'}\ \_')$; }

回填和非回填方法的比较

$$
\begin{array}{l|l}
B \rightarrow B_1 \;||\; B_2 & B_1.true = B.true \\
 & B_1.false = newlabel() \\
 & B_2.true = B.true \\
 & B_2.false = B.false \\
 & B.code = B_1.code \;||\; label(B_1.false) \;||\; B_2.code
\end{array}
$$

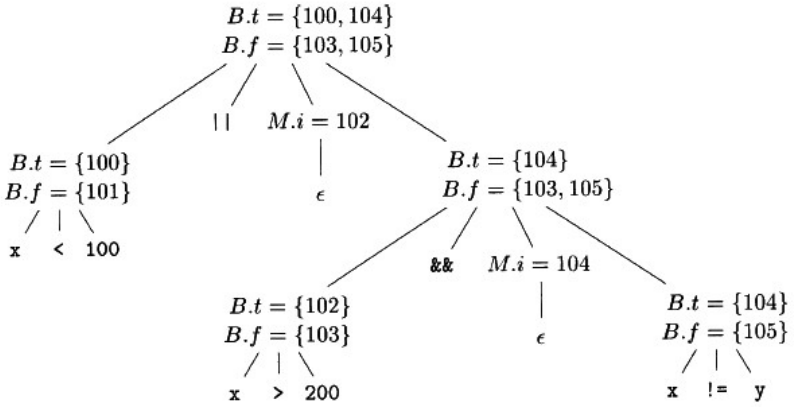
回填；并将真假出口相对应

$$
\begin{array}{ll}
B \rightarrow B_1 \;||\; M \; B_2 & \{\; backpatch(B_1.falselist, M.instr); \\
 & \quad B.truelist = merge(B_1.truelist, B_2.truelist); \\
 & \quad B.falselist = B_2.falselist; \;\} \\
M \rightarrow \epsilon & \{\; M.instr = nextinstr; \;\}
\end{array}
$$

- true/false属性的赋值，在回填方案中对应为相应的list的赋值或者merge
- 原来生成label的地方，在回填方案中使用M来记录相应的代码位置，M.inst需要对应label的标号
- 原方案生成的指令goto B_1.false，现在生成了goto M.inst

布尔表达式的回填例子

- **x<100 || x>200 && x!=y**




```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto _
103:  goto _
104:  if x != y goto _
105:  goto _
```

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

a) 将 104 回填到指令 102 中之后

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

b) 将 102 回填到指令 101 中之后

1)	$B \rightarrow B_1$ **||** $M\ B_2$	{ *backpatch*(B_1.*falselist*, M.*instr*); B.*truelist* = *merge*(B_1.*truelist*, B_2.*truelist*); B.*falselist* = B_2.*falselist*; }
2)	$B \rightarrow B_1$ **&&** $M\ B_2$	{ *backpatch*(B_1.*truelist*, M.*instr*); B.*truelist* = B_2.*truelist*; B.*falselist* = *merge*(B_1.*falselist*, B_2.*falselist*); }
3)	$B \rightarrow$ **!** B_1	{ B.*truelist* = B_1.*falselist*; B.*falselist* = B_1.*truelist*; }
4)	$B \rightarrow$ **(** B_1 **)**	{ B.*truelist* = B_1.*truelist*; B.*falselist* = B_1.*falselist*; }
5)	$B \rightarrow E_1$ **rel** E_2	{ B.*truelist* = *makelist*(*nextinstr*); B.*falselist* = *makelist*(*nextinstr* + 1); *gen*('**if**' E_1.*addr* **rel**.*op* E_2.*addr* '**goto** _'); *gen*('**goto** _'); }
6)	$B \rightarrow$ **true**	{ B.*truelist* = *makelist*(*nextinstr*); *gen*('**goto** _'); }
7)	$B \rightarrow$ **false**	{ B.*falselist* = *makelist*(*nextinstr*); *gen*('**goto** _'); }
8)	$M \rightarrow \epsilon$	{ M.*instr* = *nextinstr*; }

```
      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:
```

控制转移语句的回填

$$S \rightarrow \textbf{if}\,(B)\,S \mid \textbf{if}\,(B)\,S\,\textbf{else}\,S \mid \textbf{while}\,(B)\,S$$
$$\mid \{L\} \mid A$$
$$L \rightarrow L\,S \mid S$$

- **语句的综合属性：nextlist**
  - nextlist中的跳转指令的目标应该是S执行完毕之后紧接着执行的下一条指令的位置
  - 考虑S是while语句、if语句的子语句时，分别应该跳转到哪里

控制转移语句的回填

- **M的作用就是用M.instr记录下一个指令的位置**
- **N的作用是生成goto指令坯，N.nextlist只包含这个指令的位置**

1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

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

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

控制转移语句的回填

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

4) $S \rightarrow$ { L } { S.*nextlist* = L.*nextlist*; }

5) $S \rightarrow A$; { S.*nextlist* = **null**; }

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

9) $L \rightarrow S$ { L.*nextlist* = S.*nextlist*; }

Break、Continue的处理

- **虽然break、continue在语法上是一个独立的句子，但是它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

作业

- **教材P263: 6.6.1, 6.6.3**
- **教材P268: 6.7.1**

类型检查

什么是类型

- **一个程序变量在程序执行期间的值可以设想为有一个范围，这个范围的一个界叫做该变量的类型。**
  - 变量都被给定类型的语言叫做类型化语言（typed language）。
  - 语言若不限制变量值的范围，则被称作未类型化的语言（untyped language ）

类型系统

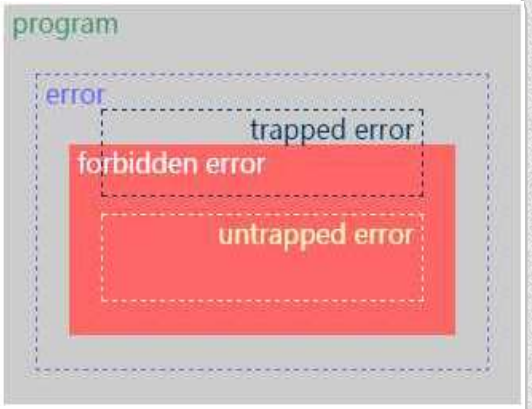
- 类型化语言的类型系统（type system）是该语言的一个组成部分，它始终监视着程序中变量的类型，通常还包括所有表达式的类型。
- 一个类型系统主要由一组定型规则（typing rules）构成，这组规则用来给各种语言构造（程序、语句、表达式等）指派类型。

类型系统

- **程序运行时的执行错误分成两类**
  - 会被捕获的错误（*trapped error*）
    - 例：非法指令错误、非法内存访问、除数为零
    - 引起计算立即停止
  - 不会被捕获的错误（*untrapped error*）
    - 例：下标变量的访问越过了数组的末端
    - 例：跳到一个错误的地址，该地址开始的内存正好代表一个指令序列
    - 错误可能会有一段时间未引起注意

类型系统

- **禁止错误（*forbidden error*）**
  - *untrapped error*集合 + *trapped error*的一个子集
  - 为语言设计类型系统的目标是在排除禁止错误



- **良行为的程序(well-behaved)**
  - A program fragment that will not produce forbidden errors at run time (不同场合对良行为的定义略有区别)
- **安全语言**
  - 任何合法程序都没有`forbidden error`

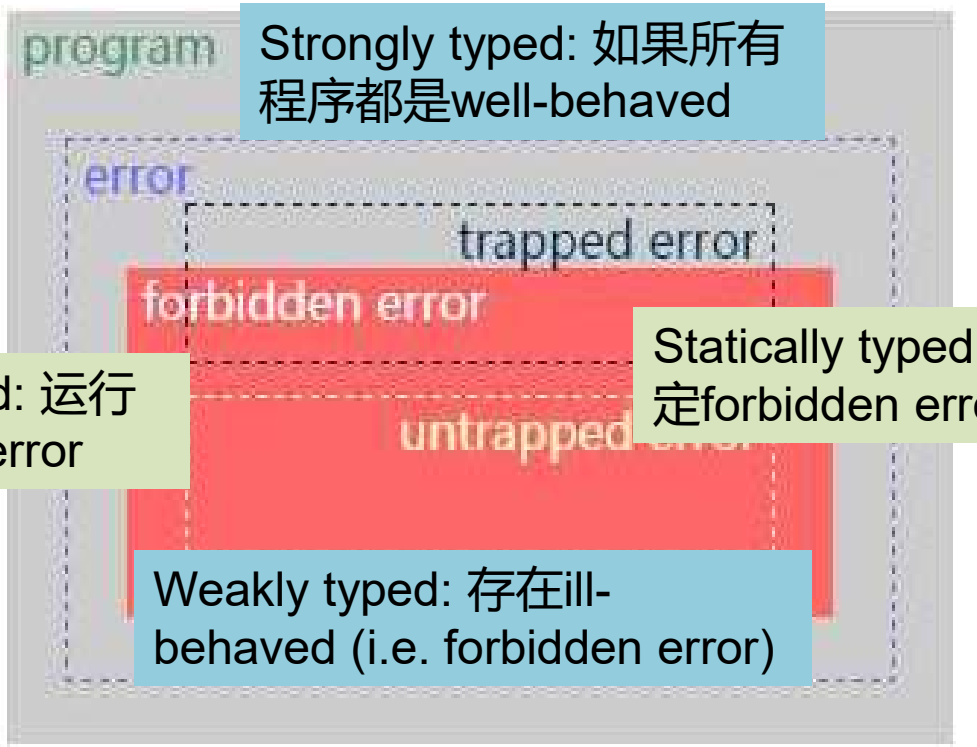
类型系统

- **类型化的语言**
  - 变量都被给定类型的语言：表达式、语句等程序构造的类型都可以静态确定，例如，类型*boolean*的变量*x*在程序每次运行时的值只能是布尔值，*not* (*x*)总有意义
- **未类型化的语言**
  - 不限制变量值范围的语言：一个运算可以作用到任意的运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个语言未加定义的结果, 例如：LISP语言

类型系统

- **显式类型化语言**
  - 类型是语法的一部分
- **隐式类型化的语言**
  - 不存在隐式类型化的主流语言，但可能存在忽略类型信息的程序片段，例如不需要程序员声明函数的参数类型

类型系统


program
error
trapped error
forbidden error
untrapped
Strongly typed: 如果所有程序都是well-behaved
Dynamically typed: 运行时确定forbidden error
Statically typed: 编译时确定forbidden error
Weakly typed: 存在ill-behaved (i.e. forbidden error)

类型表达式

- **类型本身也有结构，我们使用类型表达式 (type expression)来表示这种结构**
  - 基本类型：Boolean, integer, float, char, void；或
  - 类名；或

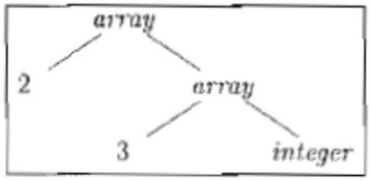


图 6-14　int[2][3]的类型表达式

类型表达式

- **类型本身也有结构，我们使用类型表达式 (type expression)来表示这种结构**
  - 通过将“类型构造算子”作用于类型表达式而得，例如：
    - array[数字，类型表达式]
    - record[字段/类型对的列表]（可以用符号表表示）
    - 函数类型构造算子→ ：参数类型→结果类型
    - 笛卡尔积：s X t: struct { int a[10]; float f;} st 对应于：record((a array(0..9, int)) (f real))
    - 可以包含取值为类型表达式的变量

类型表达式的例子

- **类型例子**
  - 元素个数为3X4的二维数组
  - 数组的元素的记录类型
  - 该记录类型中包含两个字段: x和y,其类型分别是float和integer
- **类型表达式**
  - array[3,
    array[4, record[(x,float),(y,integer)]
    ]
    ]

类型等价

- **不同的语言有不同的类型等价的定义**
- **结构等价**
  - 或者它们是相同的基本类型
  - 或者是相同的构造算子作用于结构等价的类型而得到的。
  - 或者一个类型是另一个类型表达式的名字
- **名等价**
  - 类型名仅仅代表其自身

静态类型信息在编译中的作用

- **应用一：静态类型检查**
  - 编译时确定forbidden error

类型检查和转换

- **类型系统**
  - 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
- **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 → 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

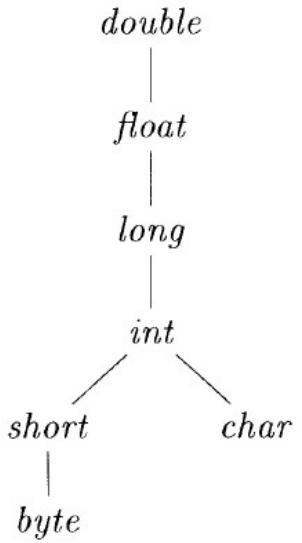
    then 对于某些类型α,β；f的类型为α → β且x的类型为α

类型转换

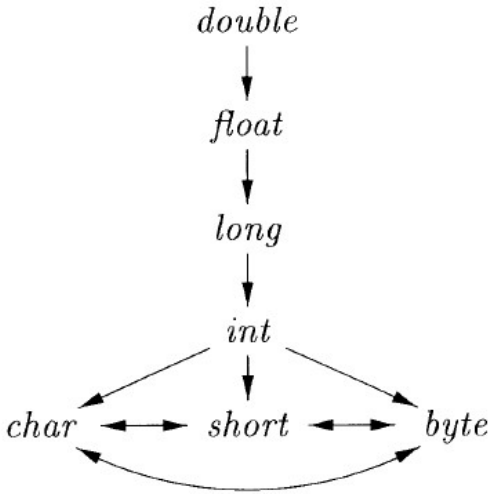
- **假设在表达式x*i中，x为浮点数、i为整数，则结果应该是浮点数**
  - x和i使用不同的二进制表示方式
  - 浮点*和整数*使用不同的指令
  - t1 = (float) i
  - t2 = x fmul t1
- **类型转换比较简单时的SDD**
  - E → E1 + E2 {
    if(E1.type = integer and E2.type = integer) E.type = integer;
    else if (E1.type = float and E2.type= integer) E.type = float;
    }
  - 这个规则没有考虑生成类型转换代码

类型的widening和narrowing

- **编译器自动完成的转换为隐式转换，程序员用代码指定的转换为显式转换**



a) 拓宽类型转换



b) 窄化类型转换

处理类型转换的SDT

- **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**

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{\ E.type = max(E_1.type, E_2.type); \\
& \quad a_1 = widen(E_1.addr, E_1.type, E.type); \\
& \quad a_2 = widen(E_2.addr, E_2.type, E.type); \\
& \quad E.addr = \textbf{new}\ Temp\,(); \\
& \quad gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{array}
$$

```
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/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 = $\{s_i \rightarrow t_i | 1 <= i <= k\}$ and E_1.type=s_k

  then E.type = t_k

  }

Thank you!