



编译原理

第八章

代码优化

哈尔滨工业大学 陈鄞 单丽莉



代码优化

➤ 优化编译器

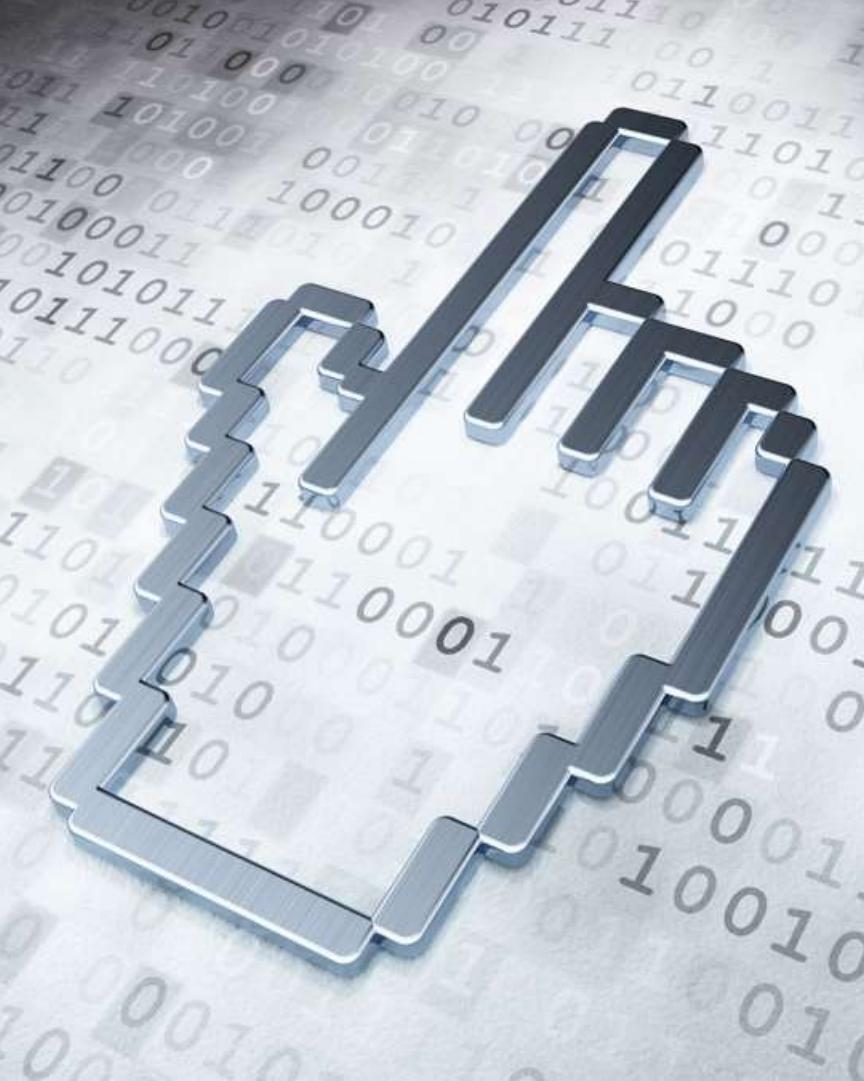
- 将程序 P 转换为与其语义等价的（具有相同输入/输出行为）程序 P' , 但 P' 可能比 P 更小或更快

➤ 完全优化编译器

- 它能将每一个程序 P 转换为与其语义等价的最小程序 P'
 - 此处最小代指最短或最快
- 可计算性理论证明了不存在完全优化编译器

➤ 充分就业定理

- 对于任何优化编译器，总是存在一个比它更好的编译器



提纲

- 8.1 流图
- 8.2 优化的分类
- 8.3 基本块的优化
- 8.4 数据流分析
- 8.5 流图中的循环
- 8.6 全局优化

8.1 流图

- 基本块(*Basic Block*)由一组总是一起执行的指令组成，即满足下列条件的**最大的连续三地址指令序列**
- 控制流只能从基本块的**第一条指令进入该块**。也就是说，没有跳转到基本块中间或末尾指令的转移指令
- 除了基本块的**最后一个指令**，控制流在离开基本块之前不会**跳转或者停机**



如何划分基本块？

基本块划分算法

- 输入：三地址指令序列
- 输出：输入序列对应的**基本块列表**，其中每个指令恰好被分配给一个基本块
- 方法：
 - 首先，确定指令序列中哪些指令是**首指令**(leaders)，即**某个基本块的第一个指令**
 1. 指令序列的第一个三地址指令是一个首指令
 2. 任意一个条件或无条件**转移指令的目标指令**是一个首指令
 3. 紧跟在一个条件或无条件**转移指令之后的指令**是一个首指令
 - 然后，每个首指令对应的基本块包括了从它自己开始，直到**下一个首指令(不含)**或者**指令序列结尾**之间的所有指令

例

```

 $i = m - 1; j = n; v = a[n];$ 
while (1) {
    do  $i = i + 1;$  while ( $a[i] < v$ );
    do  $j = j - 1;$  while ( $a[j] > v$ );
    if ( $i \geq j$ ) break;
     $x = a[i]; a[i] = a[j]; a[j] = x;$ 
}
 $x = a[i]; a[i] = a[n]; a[n] = x;$ 

```

- | | |
|--|---|
| $(1) \quad i = m - 1$
$(2) \quad j = n$
$B_1 \quad (3) \quad t_1 = 4 * n$
$\underline{(4) \quad v = a[t_1]}$
$\underline{(5) \quad i = i + 1}$
$B_2 \quad (6) \quad t_2 = 4 * i$
$\underline{(7) \quad t_3 = a[t_2]}$
$\underline{(8) \quad if \, t_3 < v \, goto(5)}$
$\underline{(9) \quad j = j - 1}$
$B_3 \quad (10) \quad t_4 = 4 * j$
$\underline{(11) \quad t_5 = a[t_4]}$
$\underline{(12) \quad if \, t_5 > v \, goto(9)}$
$B_4 \quad \underline{(13) \quad if \, i \geq j \, goto(23)}$
$\underline{(14) \quad t_6 = 4 * i}$
$\underline{(15) \quad x = a[t_6]}$ | $(16) \quad t_7 = 4 * i$
$(17) \quad t_8 = 4 * j$
$(18) \quad t_9 = a[t_8]$
$B_5 \quad (19) \quad a[t_7] = t_9$
$(20) \quad t_{10} = 4 * j$
$(21) \quad a[t_{10}] = x$
$\underline{(22) \quad goto(5)}$
$(23) \quad t_{11} = 4 * i$
$(24) \quad x = a[t_{11}]$
$(25) \quad t_{12} = 4 * i$
$B_6 \quad (26) \quad t_{13} = 4 * n$
$(27) \quad t_{14} = a[t_{13}]$
$(28) \quad a[t_{12}] = t_{14}$
$(29) \quad t_{15} = 4 * n$
$(30) \quad a[t_{15}] = x$ |
|--|---|

流图(Flow Graphs)

- 流图用来表示基本块之间的控制流，构建方法如下：
 - 流图的每个结点是一个基本块
 - 从基本块 B 到基本块 C 之间有一条边当且仅当基本块 C 的第一个指令可能紧跟在 B 的最后一条指令之后执行

此时称 B 是 C 的前驱(*predecessor*)，
 C 是 B 的后继(*successor*)

流图(Flow Graphs)

- 流图用来表示基本块之间的控制流，构建方法如下：
 - 流图的每个结点是一个基本块
 - 从基本块 B 到基本块 C 之间有一条边当且仅当基本块 C 的第一个指令可能紧跟在 B 的最后一条指令之后执行
 - 有两种方式可以确认这样的边：
 - 存在一个从 B 的结尾跳转到 C 的开头的条件或无条件跳转指令
 - 按照原来的三地址指令序列中的顺序， C 紧跟在 B 之后，且 B 的结尾不存在无条件跳转指令

例

(1) $i = m - 1$

(2) $j = n$

B₁ (3) $t_1 = 4 * n$

(4) $v = a[t_1]$

(5) $i = i + 1$

(6) $t_2 = 4 * i$

B₂ (7) $t_3 = a[t_2]$

(8) $if t_3 < v goto(5)$

(9) $j = j - 1$

B₃ (10) $t_4 = 4 * j$

(11) $t_5 = a[t_4]$

(12) $if t_5 > v goto(9)$

B₄ (13) $if i >= j goto(23)$

(14) $t_6 = 4 * i$

(15) $x = a[t_6]$

(16) $t_7 = 4 * i$

(17) $t_8 = 4 * j$

(18) $t_9 = a[t_8]$

B₅ (19) $a[t_7] = t_9$

(20) $t_{10} = 4 * j$

(21) $a[t_{10}] = x$

(22) $goto(5)$

(23) $t_{11} = 4 * i$

(24) $x = a[t_{11}]$

(25) $t_{12} = 4 * i$

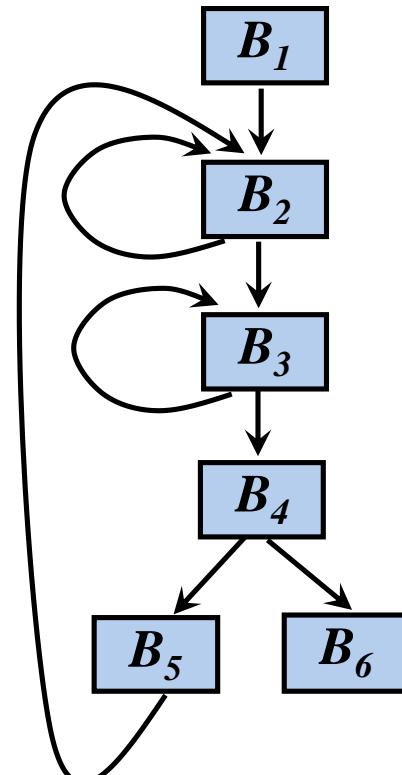
(26) $t_{13} = 4 * n$

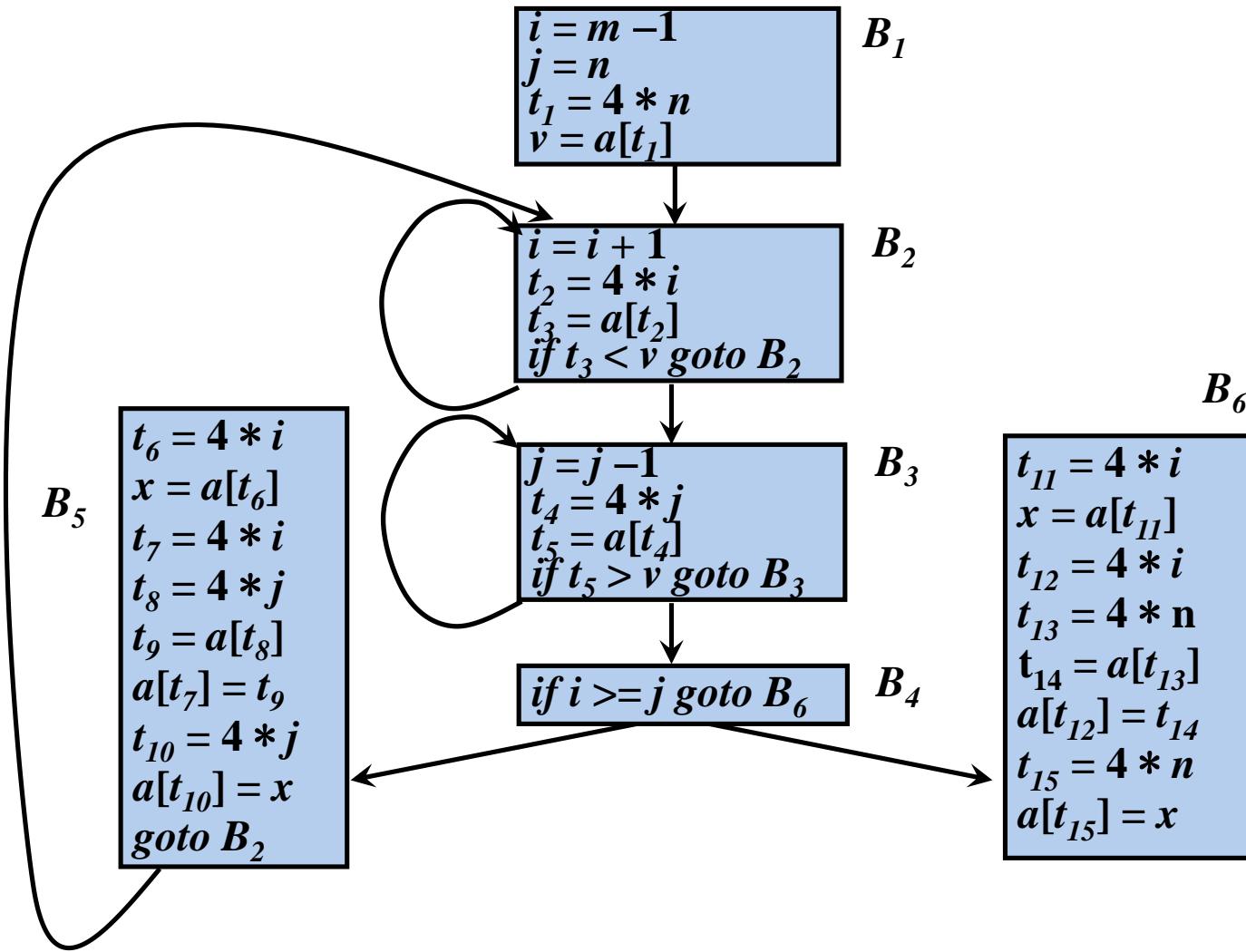
(27) $t_{14} = a[t_{13}]$

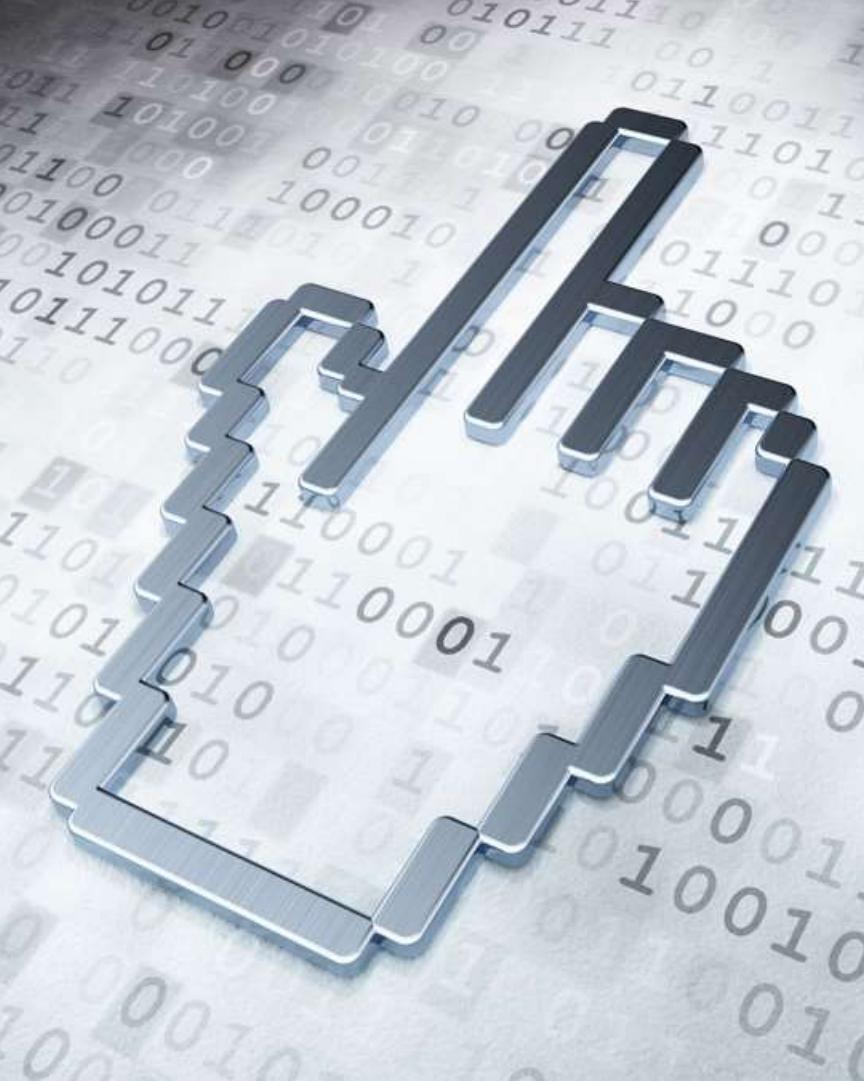
(28) $a[t_{12}] = t_{14}$

(29) $t_{15} = 4 * n$

(30) $a[t_{15}] = x$







提纲

8.1 流图

8.2 优化的分类

8.3 基本块的优化

8.4 数据流分析

8.5 流图中的循环

8.6 全局优化

8.2 优化的分类

- 机器无关优化
 - 针对中间代码
- 机器相关优化
 - 针对目标代码
- 局部代码优化
 - 单个基本块范围内的优化
- 全局代码优化
 - 面向多个基本块的优化

常用的优化方法

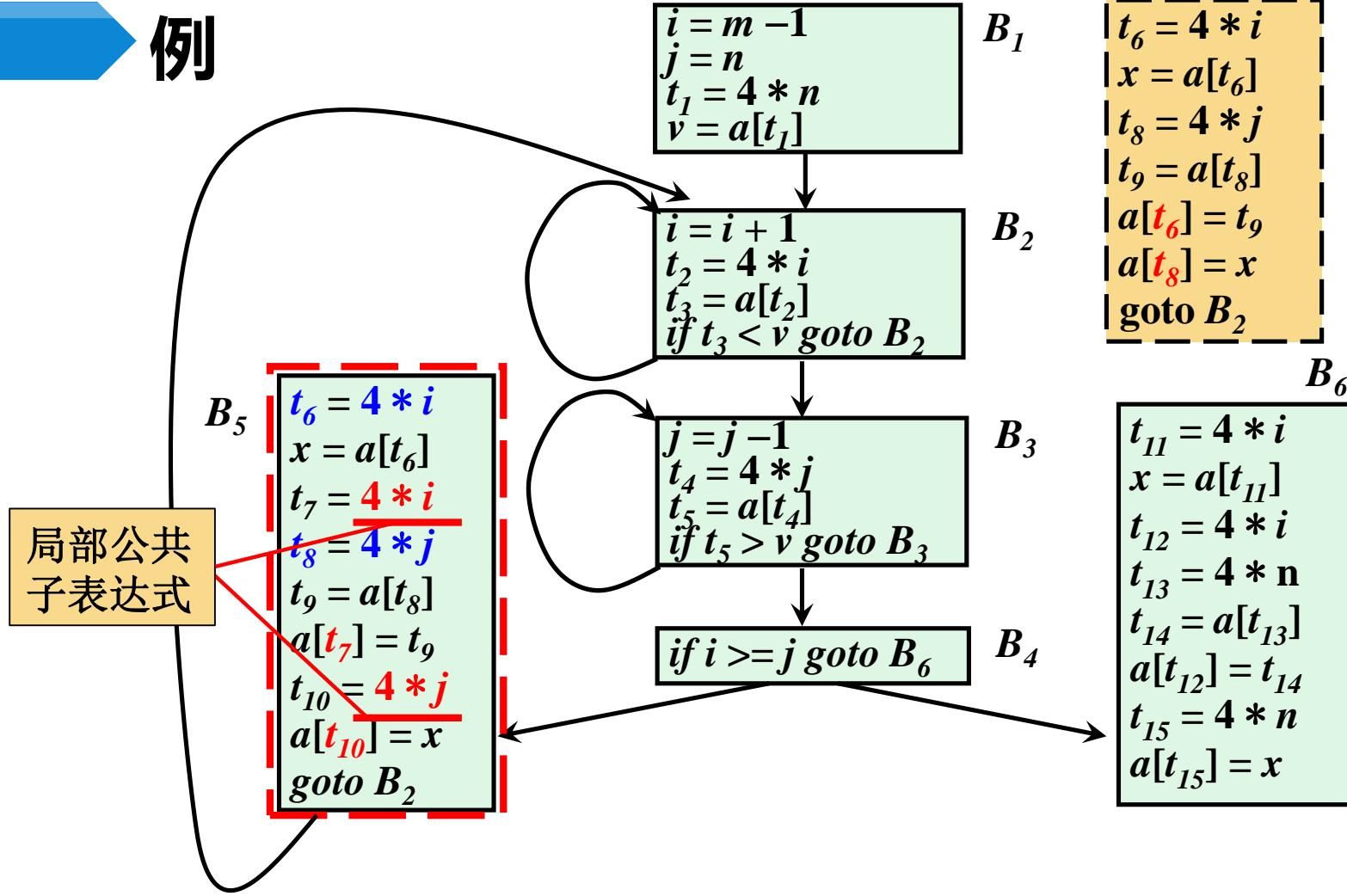
- 删除公共子表达式
 - 删除无用代码
 - 常量合并
 - 代码移动
 - 强度削弱
 - 删除归纳变量
- } 循环优化

① 删除公共子表达式

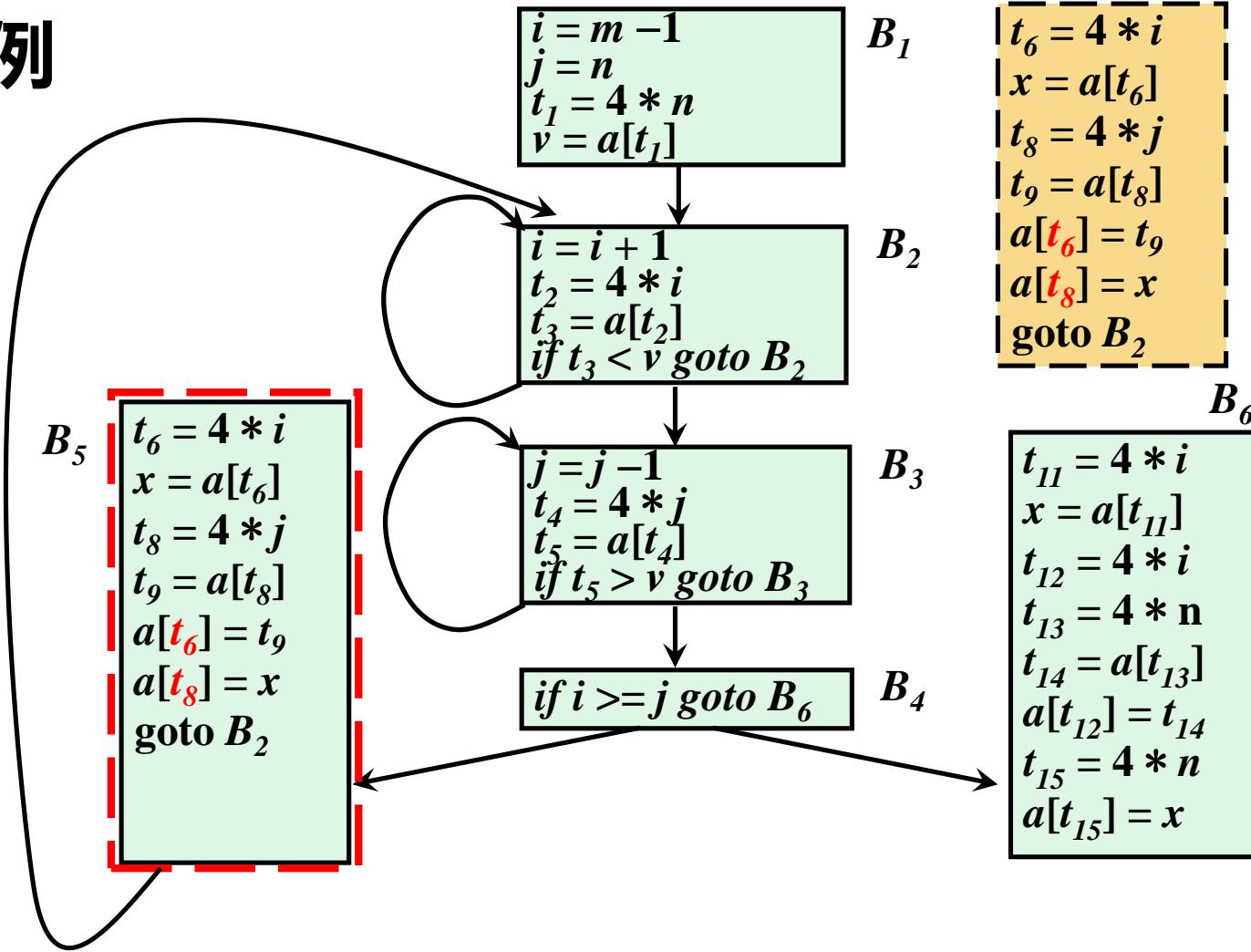
➤ 公共子表达式

- 如果表达式 $x \text{ op } y$ 先前已被计算过，并且从先前的计算到现在， $x \text{ op } y$ 中变量的值没有改变，那么 $x \text{ op } y$ 的这次出现就称为**公共子表达式**(common subexpression)
- 一个基本块范围内的公共子表达式，称为**局部公共子表达式**，跨基本块的公共子表达式，称为**全局公共子表达式**。

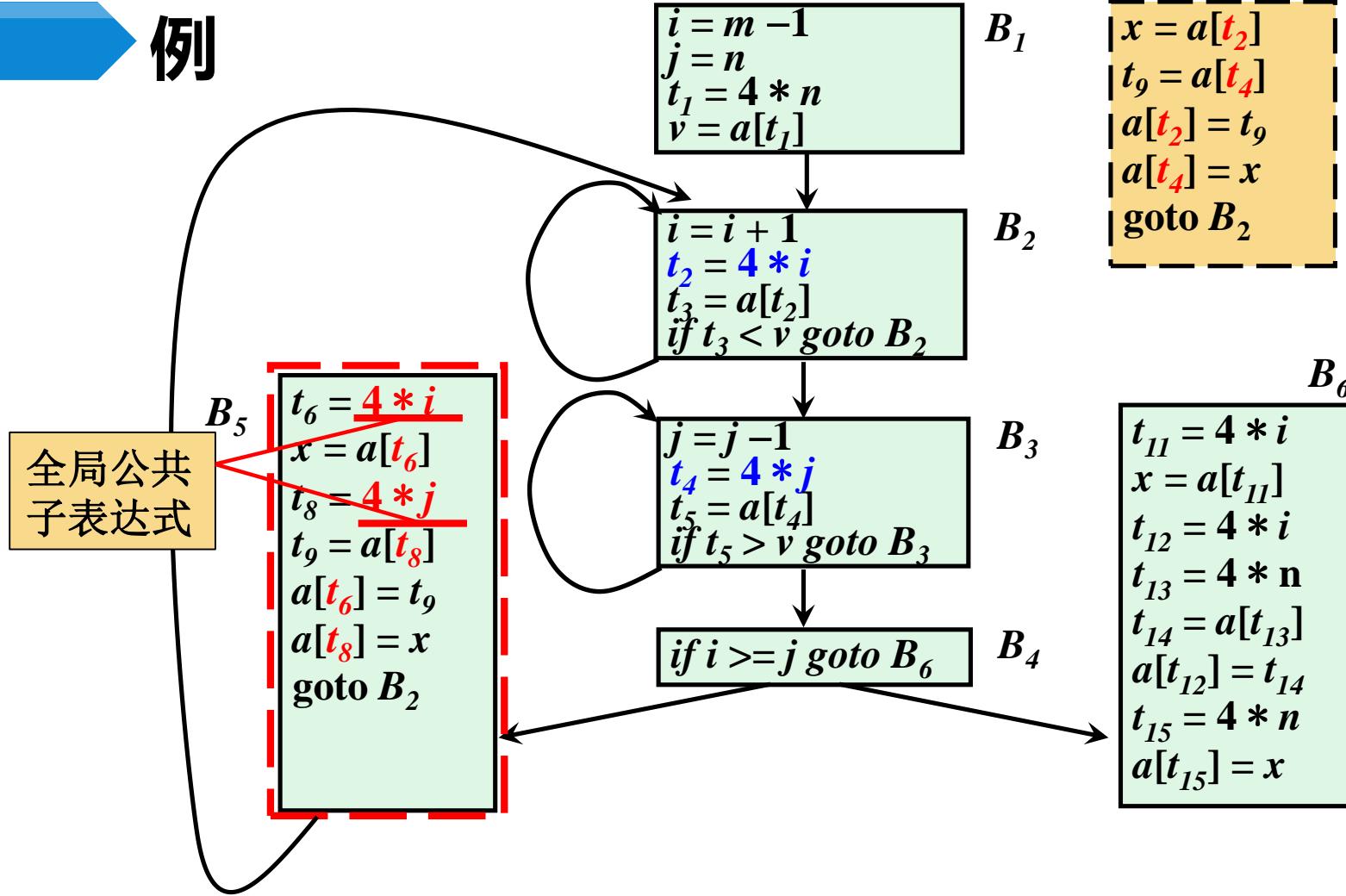
例



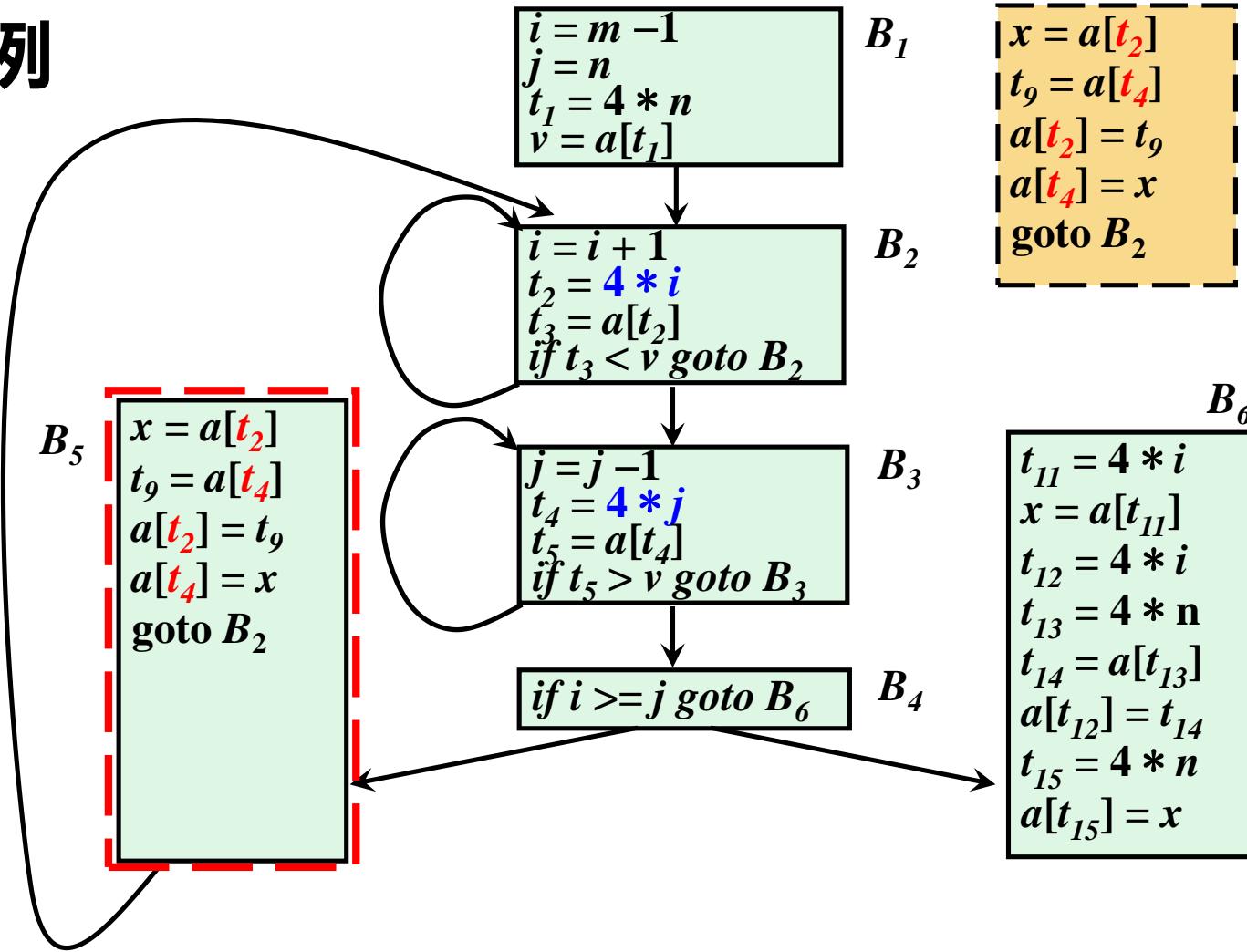
例



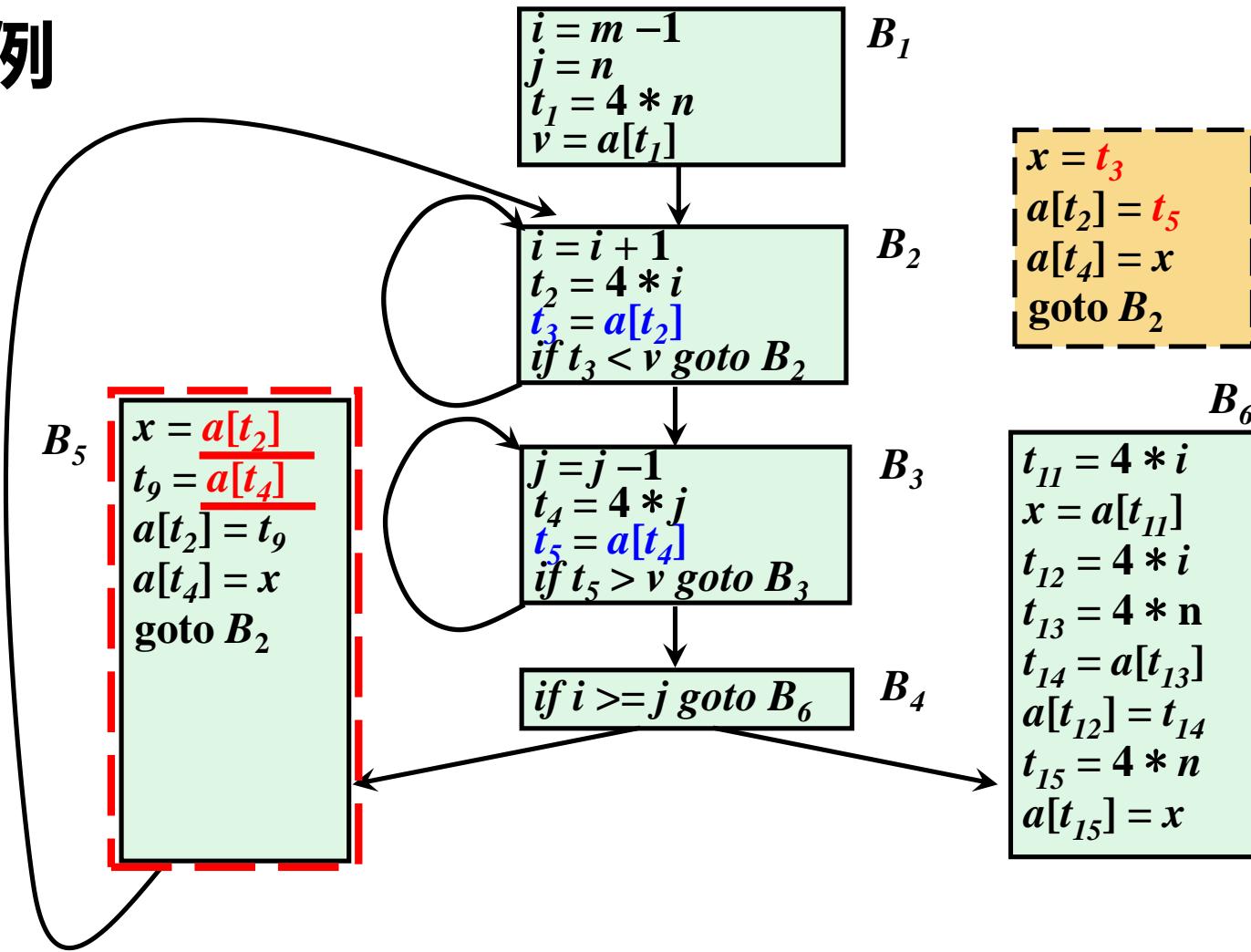
例



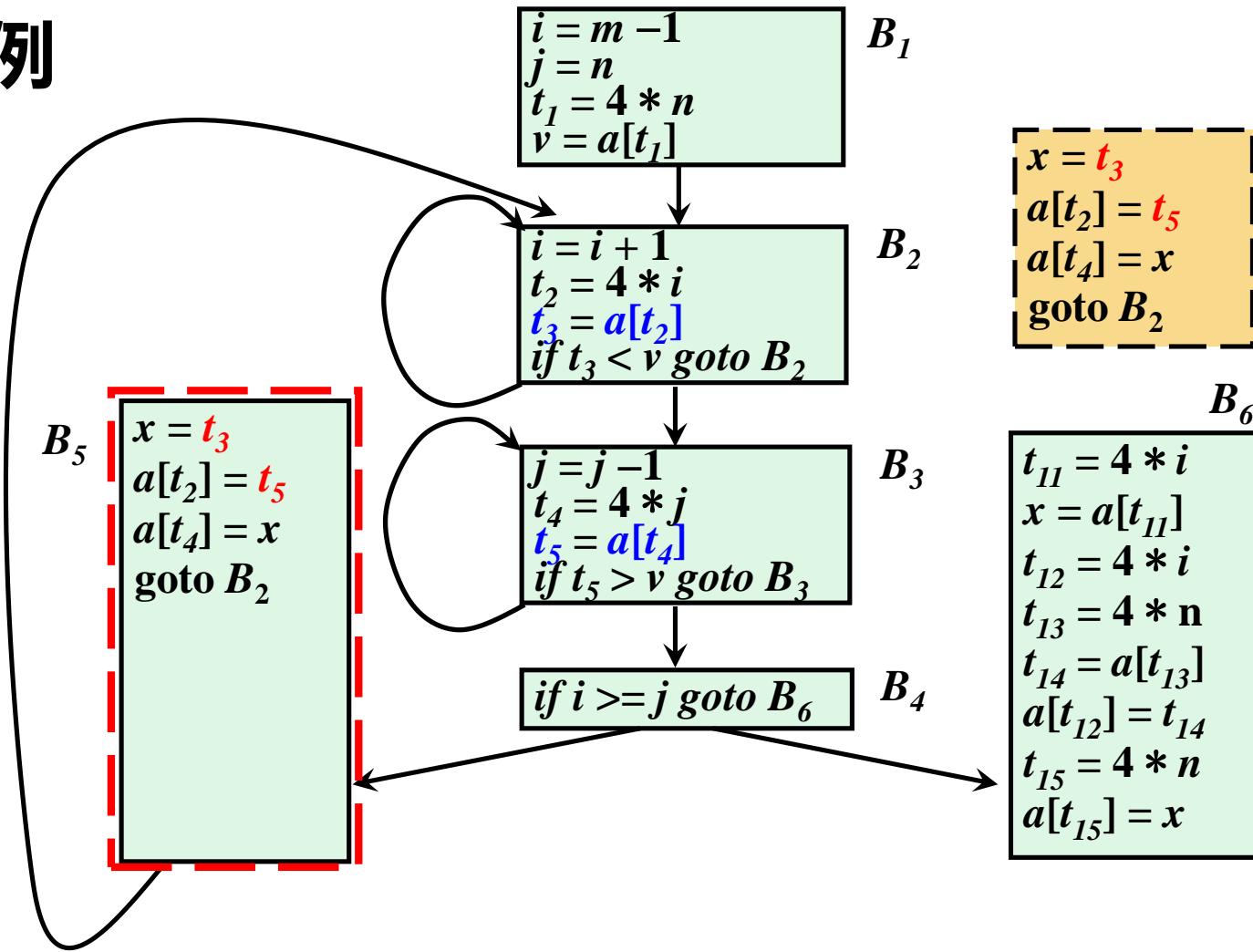
例



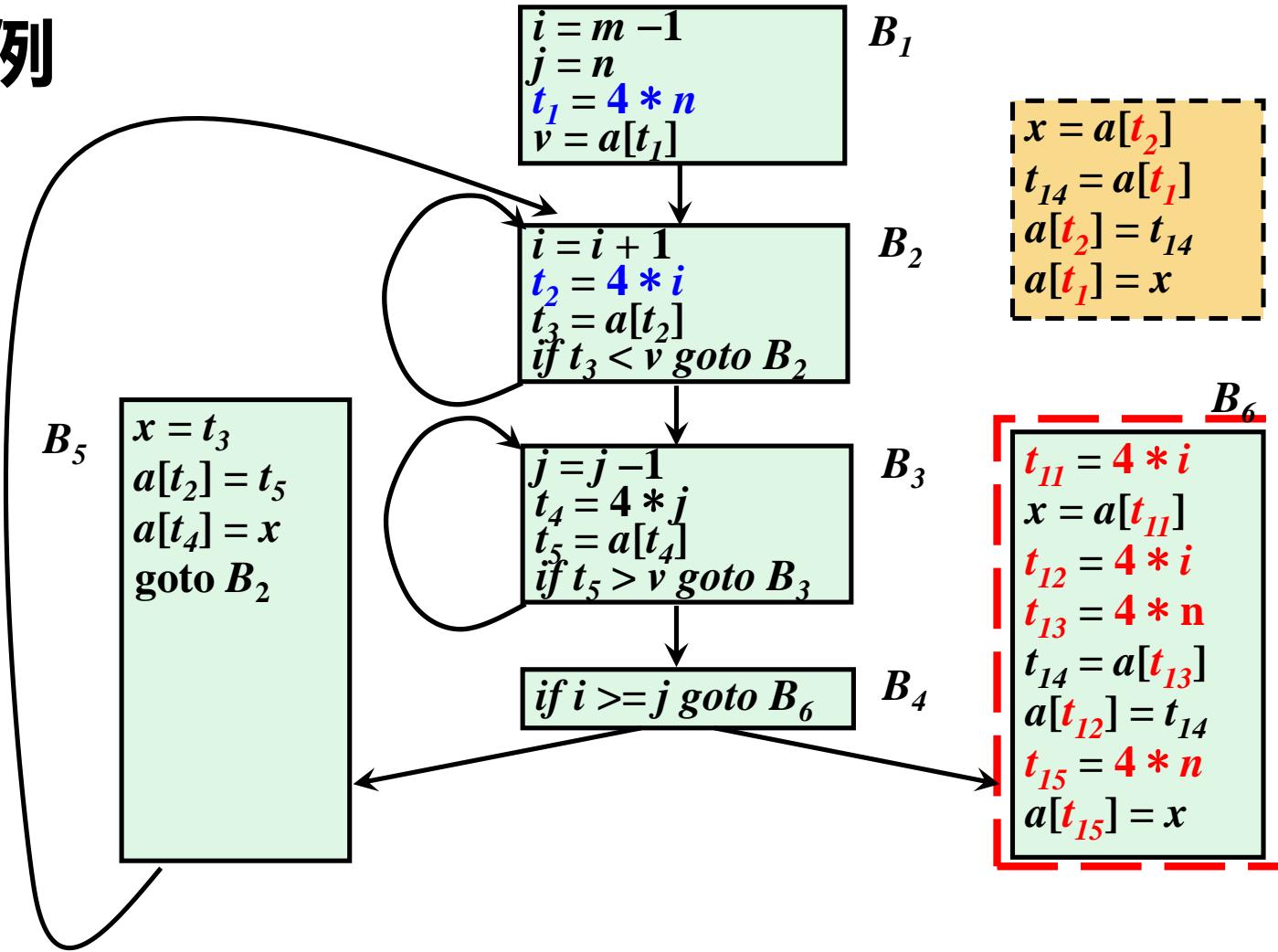
例



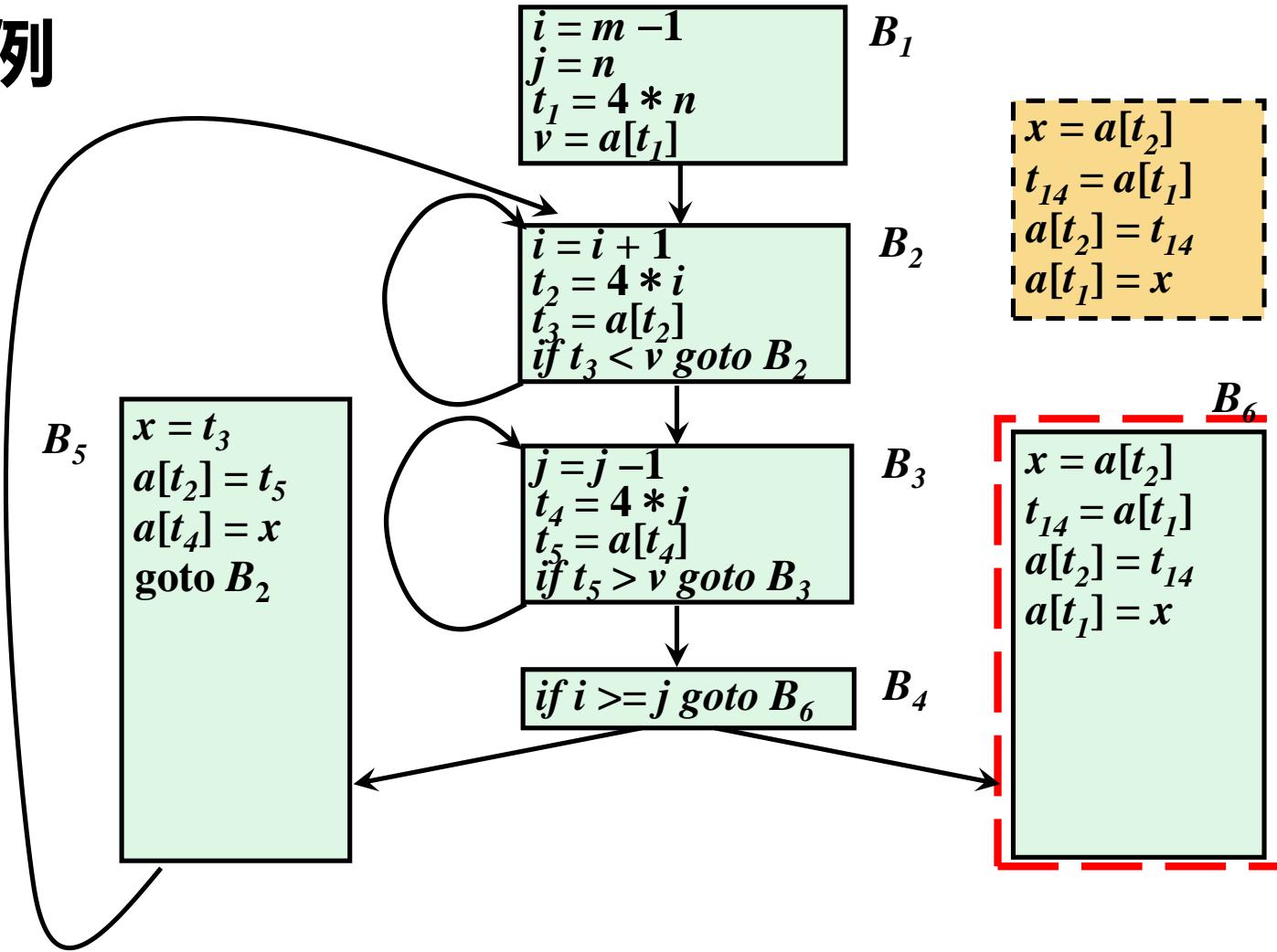
例



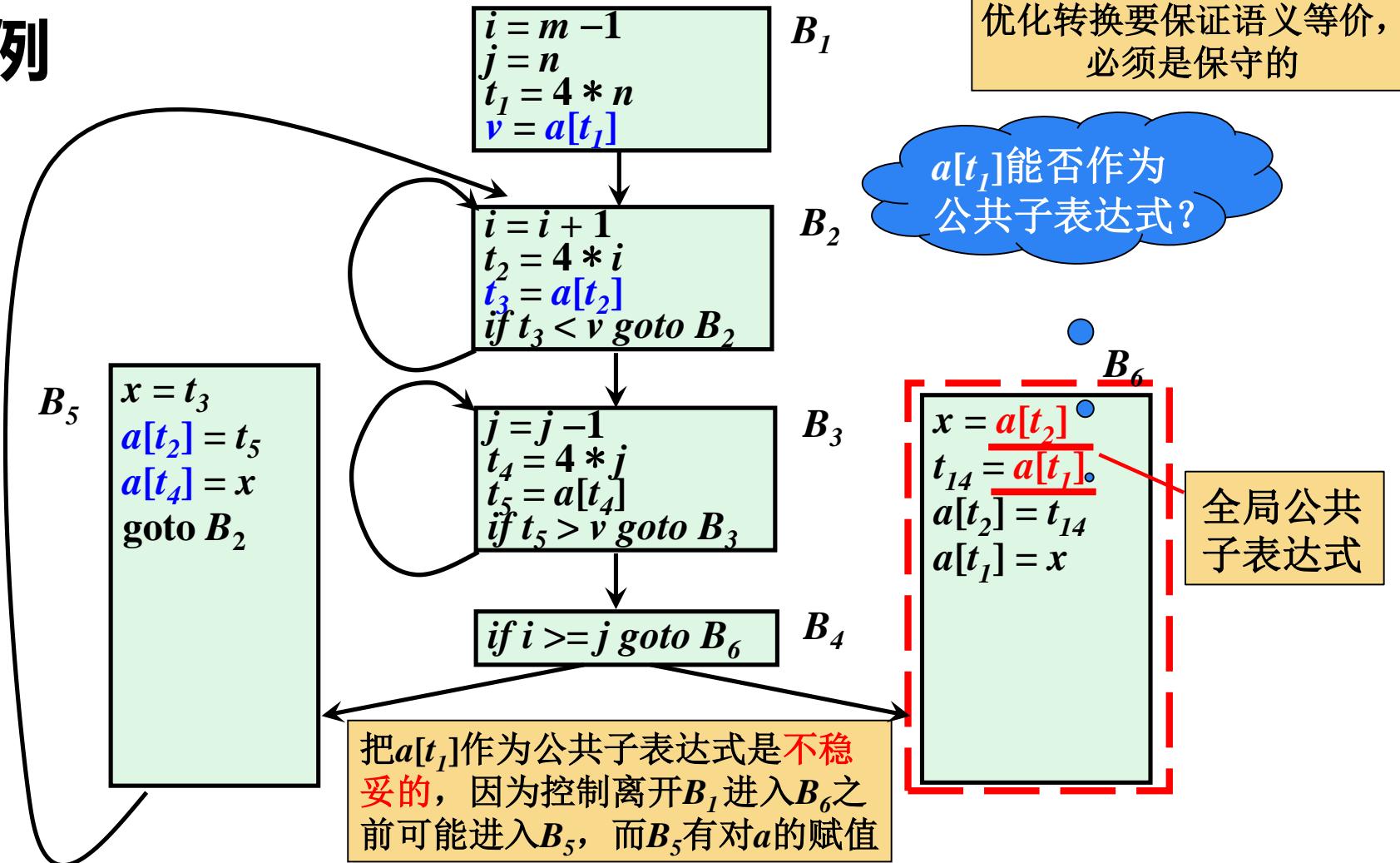
例



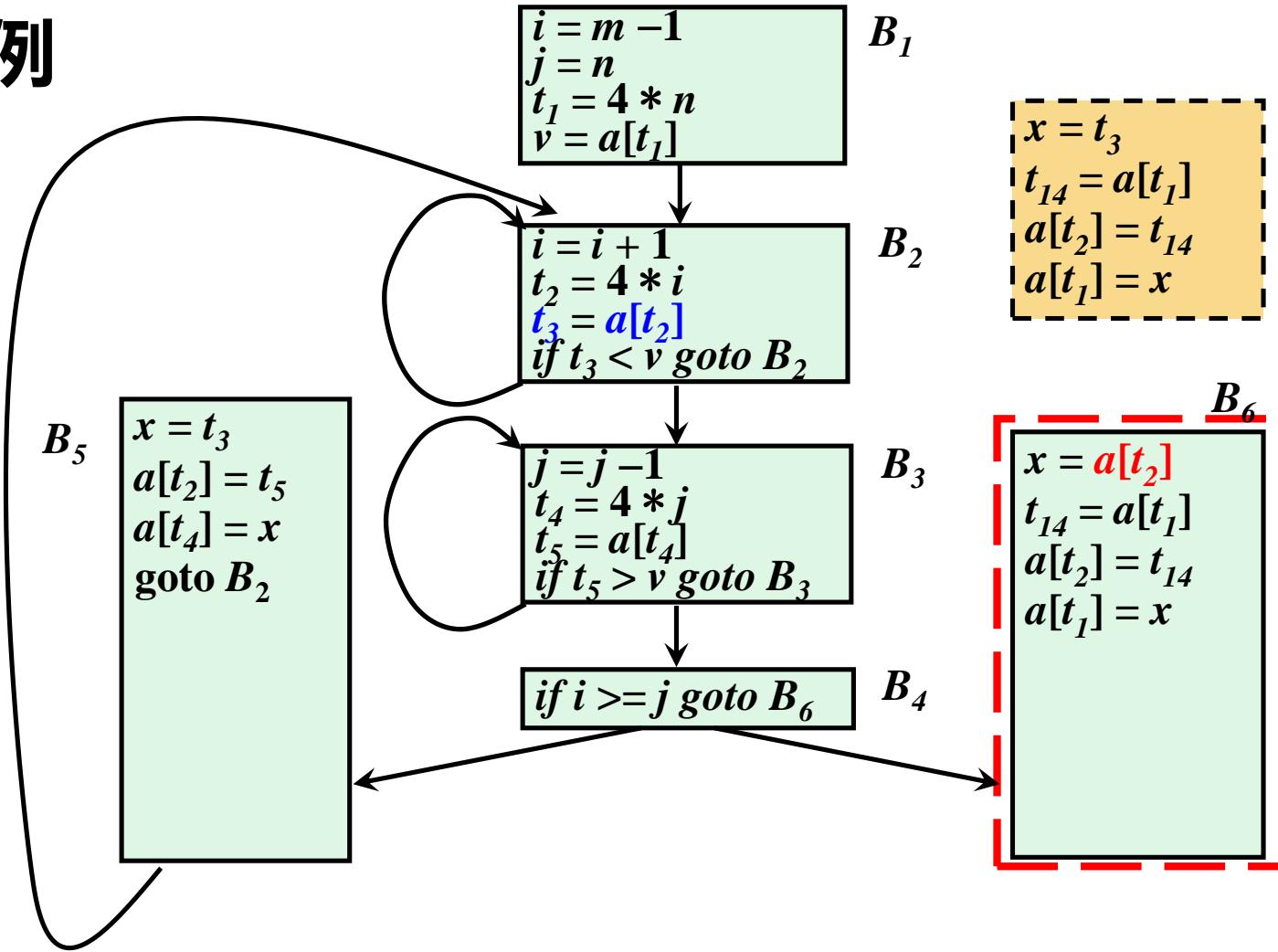
例



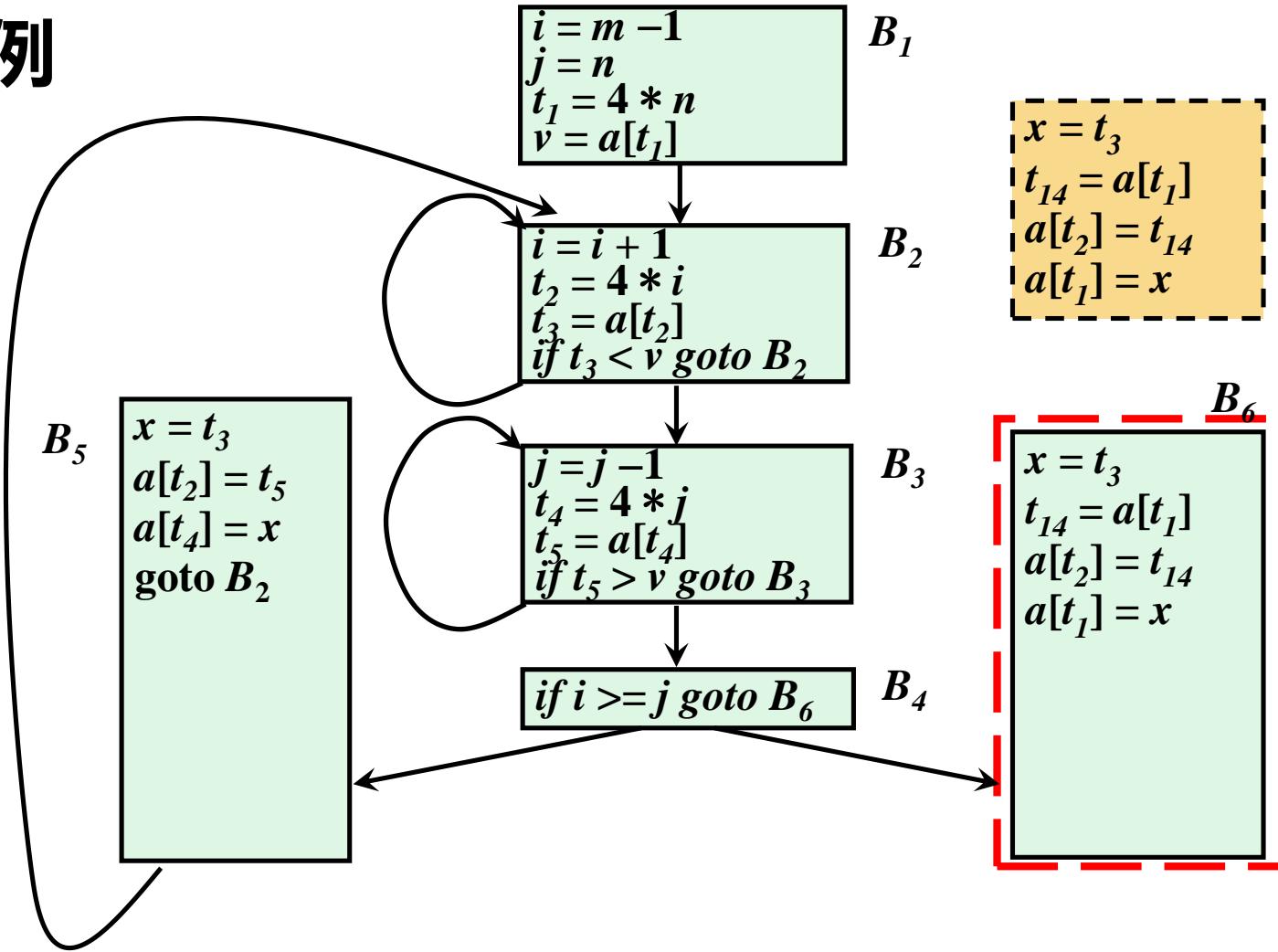
例



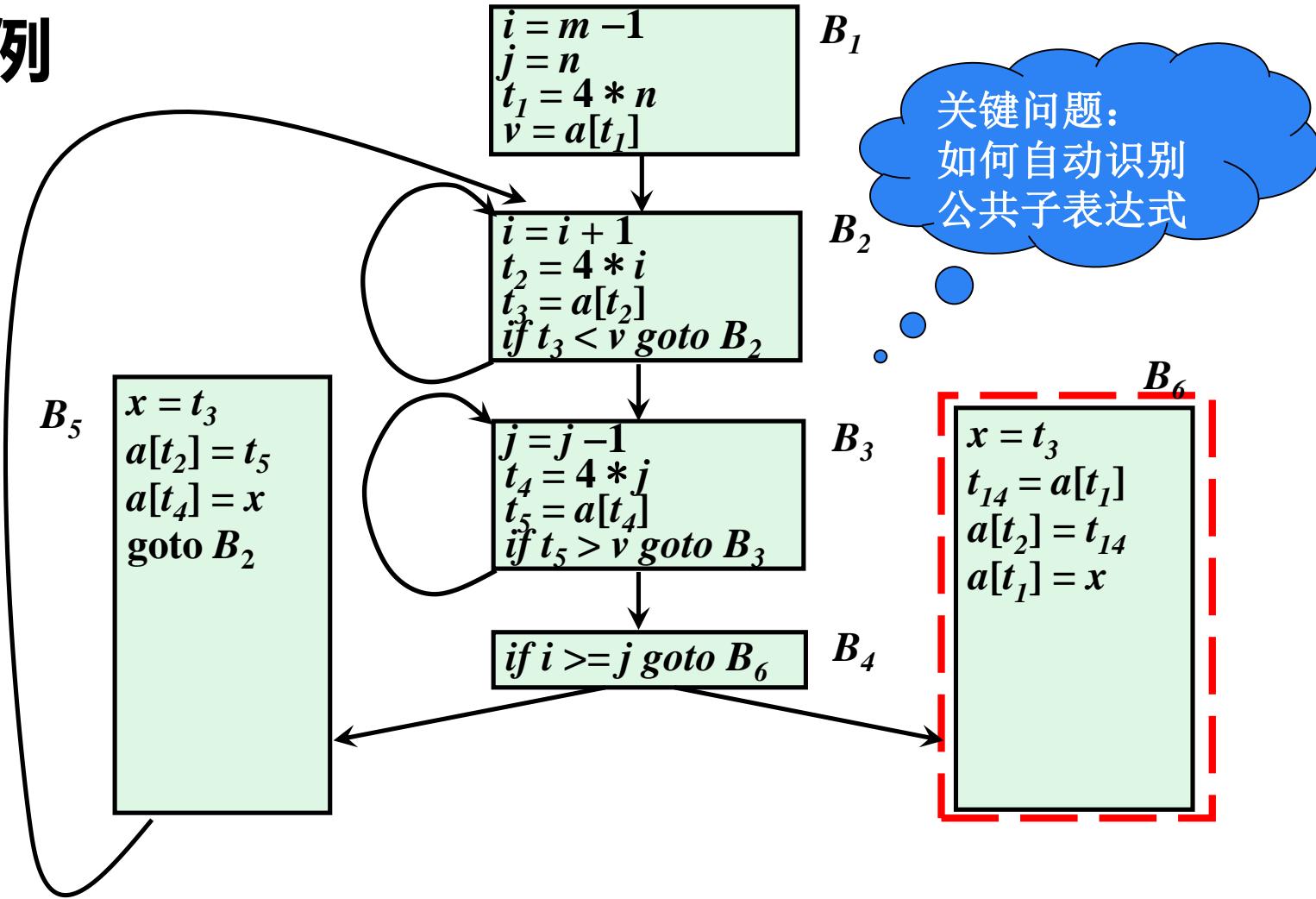
例



例

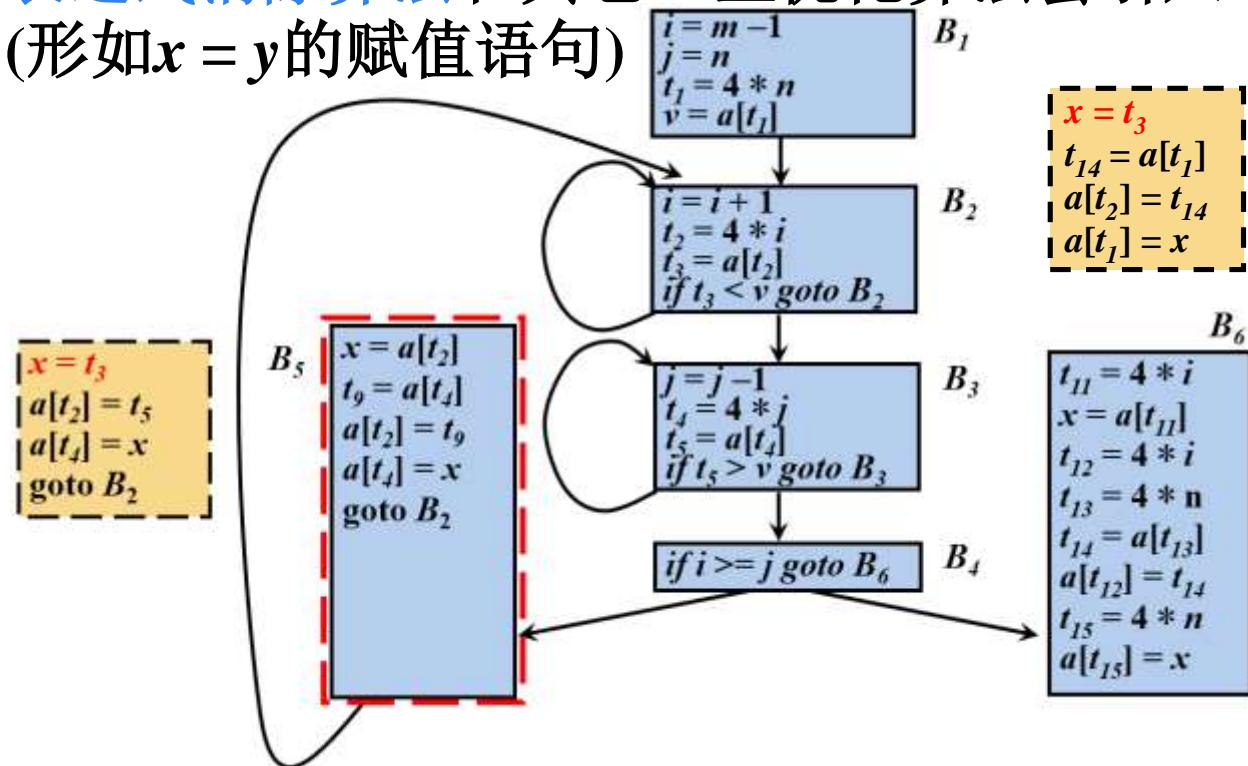


例



② 删减无用代码

- 复制传播
- 常用的公共子表达式消除算法和其它一些优化算法会引入一些复制语句(形如 $x = y$ 的赋值语句)



② 删除无用代码

- 复制传播
 - 常用的公共子表达式消除算法和其它一些优化算法会引入一些复制语句(形如 $x = y$ 的赋值语句)
 - **复制传播**: 在复制语句 $x = y$ 之后尽可能地用 y 代替 x

➤ 例 B_5

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```



```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

B_6

```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

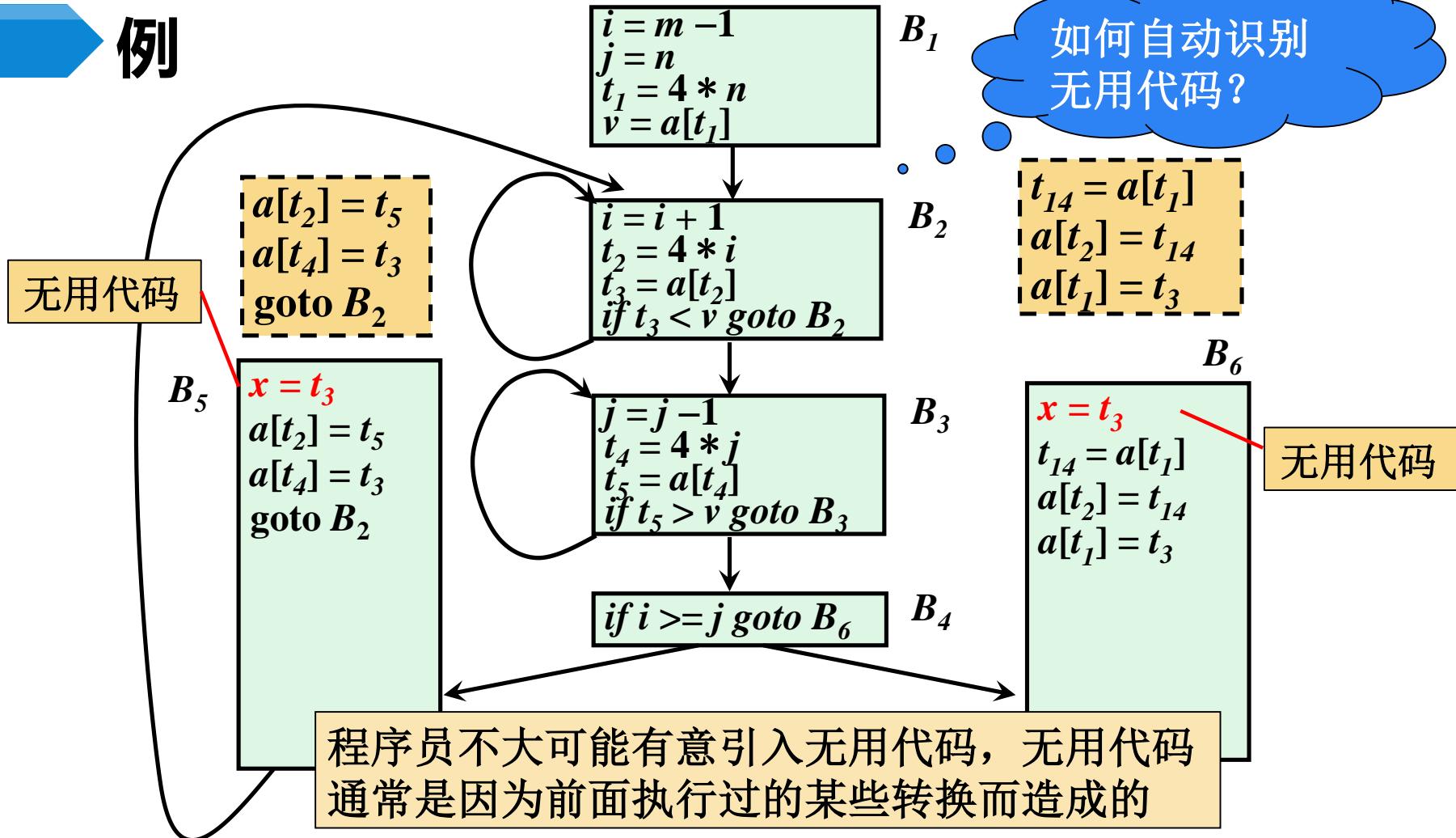


```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

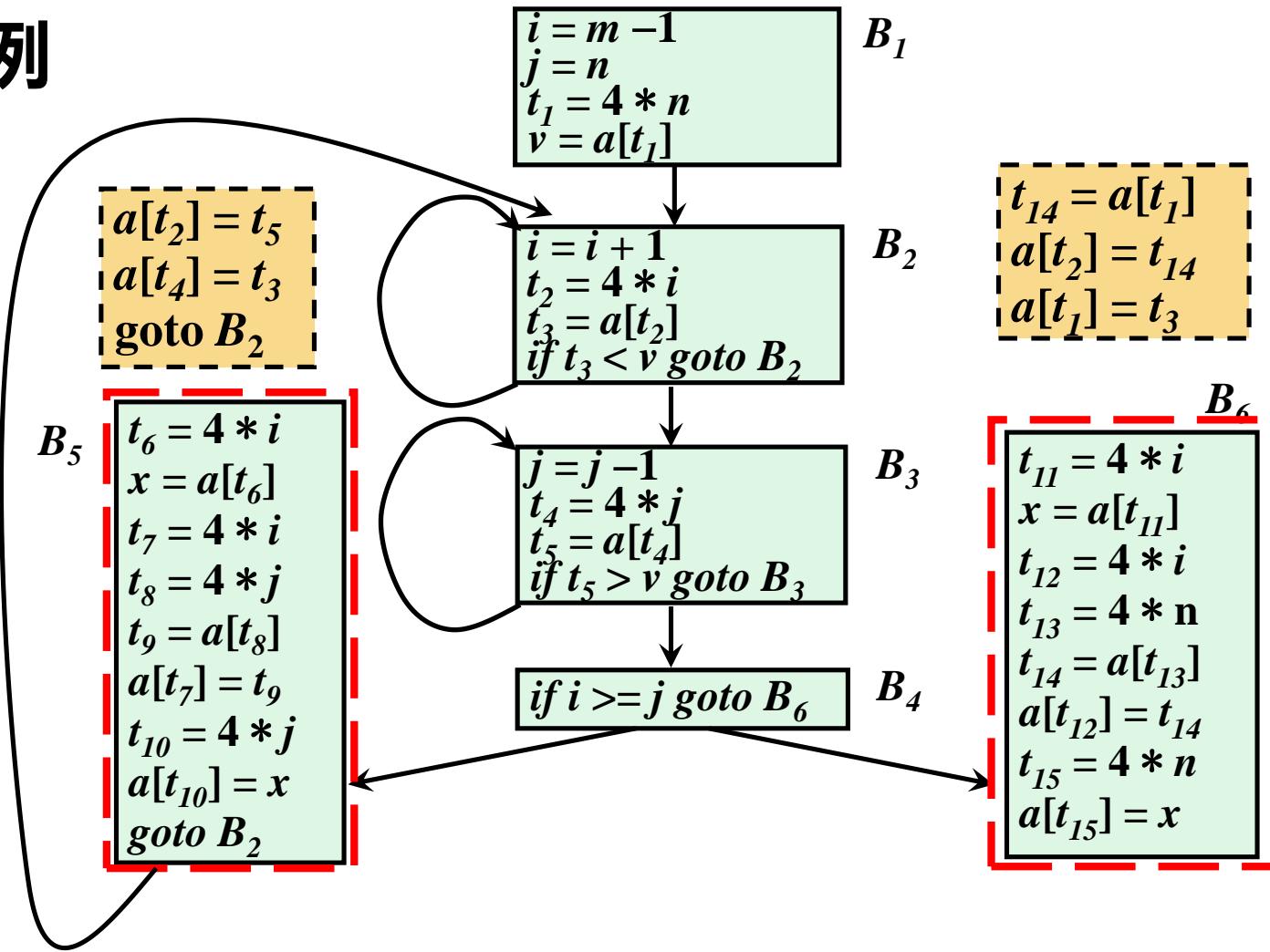
② 删除无用代码

- 复制传播
 - 常用的公共子表达式消除算法和其它一些优化算法会引入一些复制语句(形如 $x = y$ 的赋值语句)
 - **复制传播**: 在复制语句 $x = y$ 之后尽可能地用 y 代替 x
 - 复制传播给删除无用代码带来机会
- **无用代码**(死代码*Dead-Code*) : 其计算结果永远不会被使用的语句

例

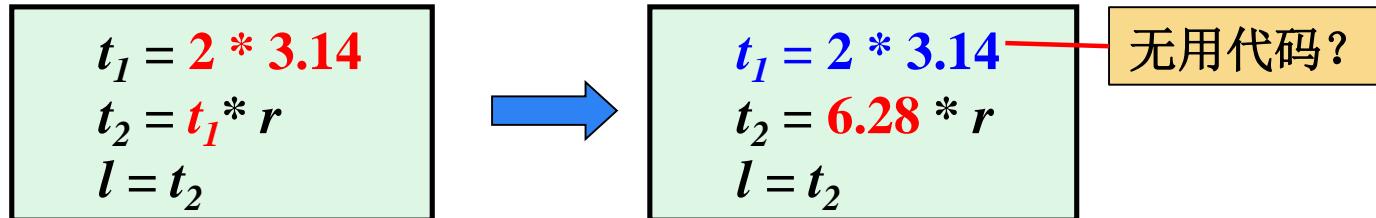


例



③ 常量合并(*Constant Folding*)

- 如果在编译时刻推导出一个表达式的值是常量，就可以使用该常量来替代这个表达式。该技术被称为**常量合并**或**常量折叠**。
- 例： $l = 2 * 3.14 * r$





④ 代码移动(*Code Motion*)

➤ 代码移动

➤ 这个转换处理的是那些**不管循环执行多少次都得到相同结果的表达式**(即**循环不变计算**, *loop-invariant computation*) , 在进入循环之前就对它们求值

例

➤ 原始程序

```
for( n = 10; n < 360; n++ ) {  
    S = 1/360*pi*r*r *n;  
    printf( “Area is %f”, S );  
}
```

循环不变计算

➤ 优化后程序

$C = 1/360*pi*r*r;$

```
for( n = 10; n < 360; n++ ) {  
    S = C * n;  
    printf( “Area is %f”, S );  
}
```

(1) $n = 1$	(8) $t_5 = t_4 * n$
(2) if $n > 360$ goto(21)	(9) $S = t_5$
(3) goto (4)	...
(4) $t_1 = 1 / 360$	(18) $t_9 = n + 1$
(5) $t_2 = t_1 * pi$	(19) $n = t_9$
(6) $t_3 = t_2 * r$	(20) goto (4)
(7) $t_4 = t_3 * r$	(21)

如何自动识别
循环不变计算?

循环不变计算的相对性

- 对于多重嵌套的循环，循环不变计算是相对于某个循环而言的。可能对于更加外层的循环，它就不是循环不变计算
- 例：

```
for( i = 1; i < 10; i++ )
```

```
    for( n = 1; n < 360 / ( 5 * i ); n++ )
```

```
        { S = ( 5*i ) / 360 * pi * r * r * n; ... }
```



⑤ 强度削弱(*Strength Reduction*)

➤ 强度削弱

➤ 用较快的操作代替较慢的操作，如用加代替乘

➤ 例

$$\triangleright 2*x \text{ 或 } 2.0*x \qquad \Rightarrow \qquad x+x$$

$$\triangleright x/2 \qquad \Rightarrow \qquad x*0.5$$

$$\triangleright x^2 \qquad \Rightarrow \qquad x*x$$

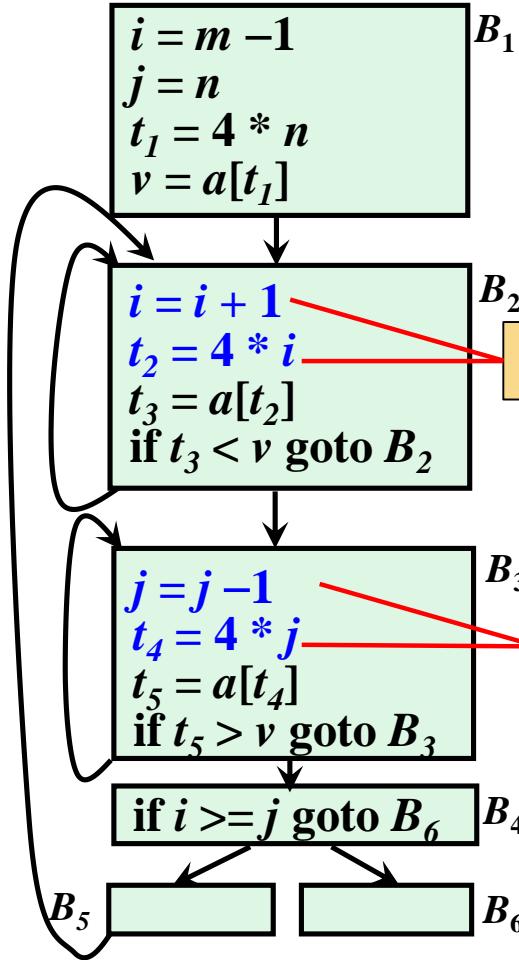
$$\triangleright a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0 \Rightarrow ((\dots((a_nx + a_{n-1})x + a_{n-2})\dots)x + a_1)x + a_0$$

循环中的强度削弱

➤ 归纳变量

➤ 对于一个变量 x ，如果存在一个正的或负的常数 c 使得每次 x 被赋值时它的值总增加 c ，那么 x 就称为归纳变量(*Induction Variable*)

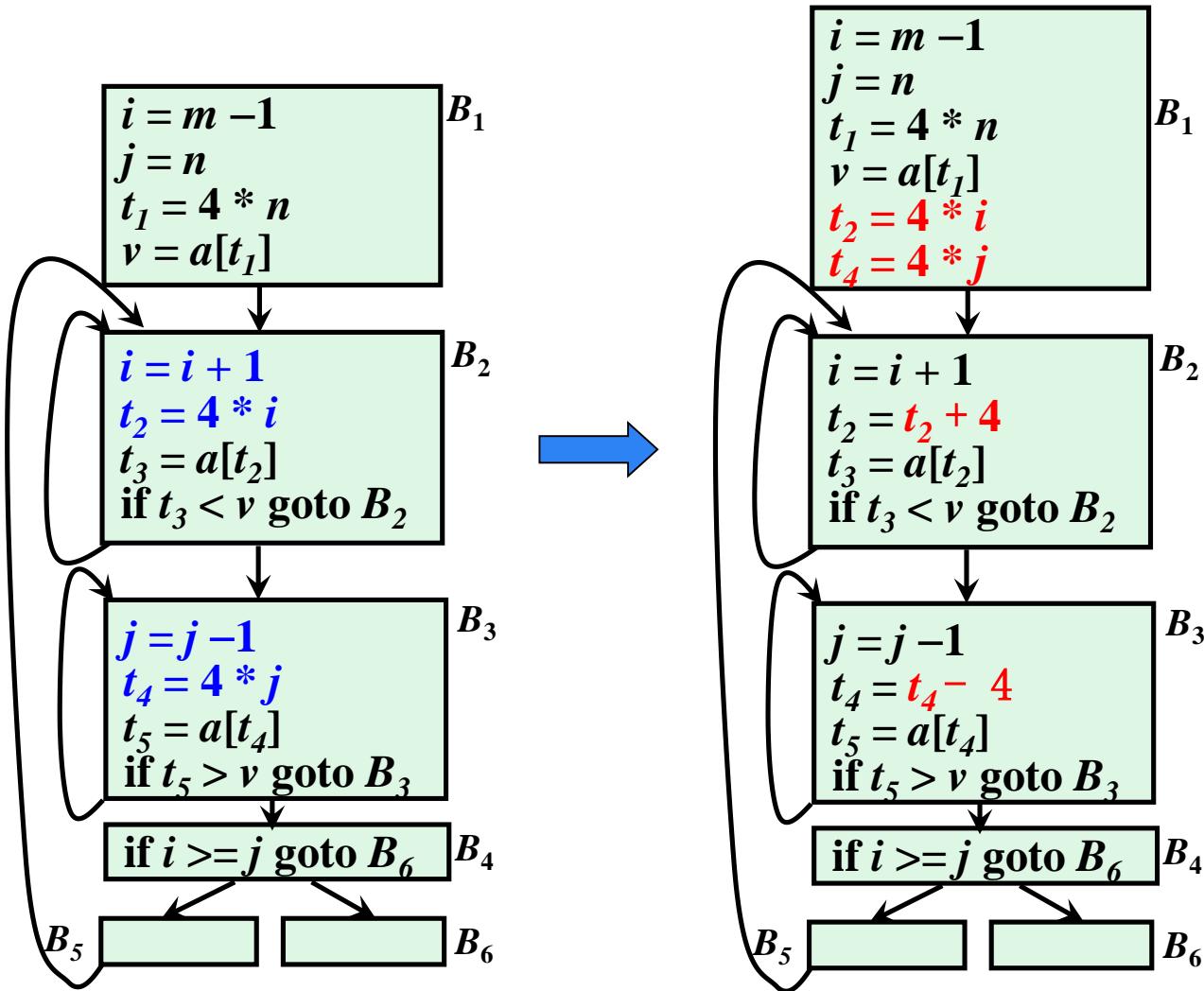
例



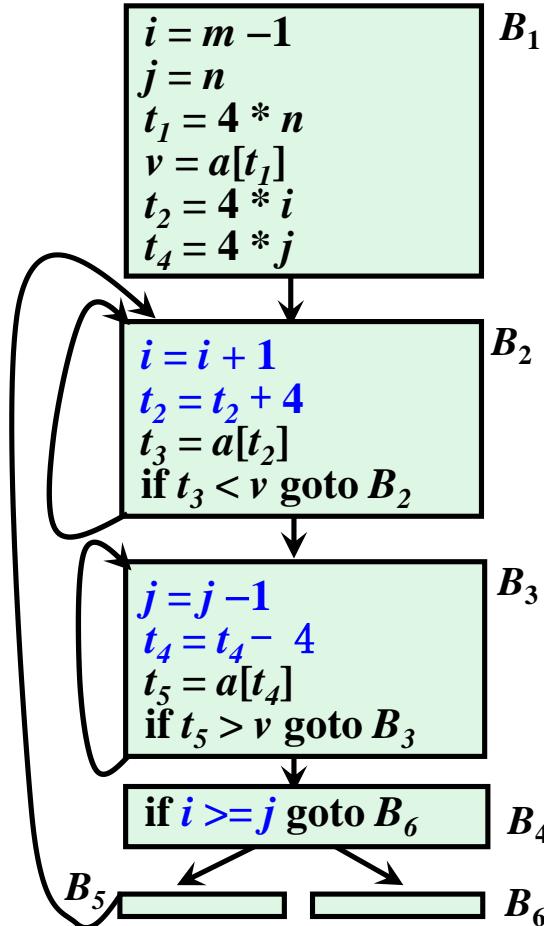
归纳变量可以通过在每次循环迭代中进行一次简单的增量运算(加法或减法)来计算,以达到计算强度削弱

问题: 如何识别
归纳变量?

例



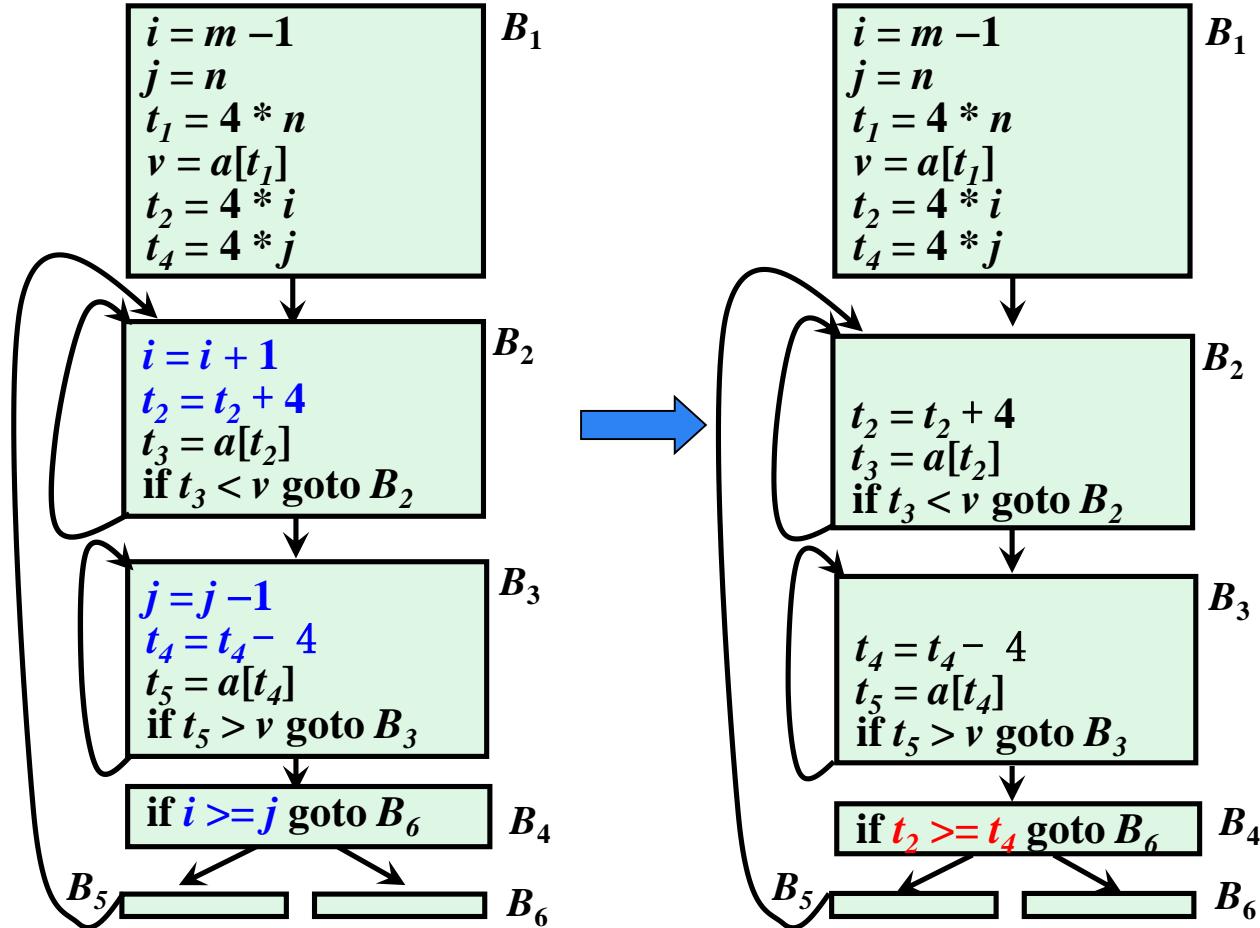
⑥ 删除归纳变量

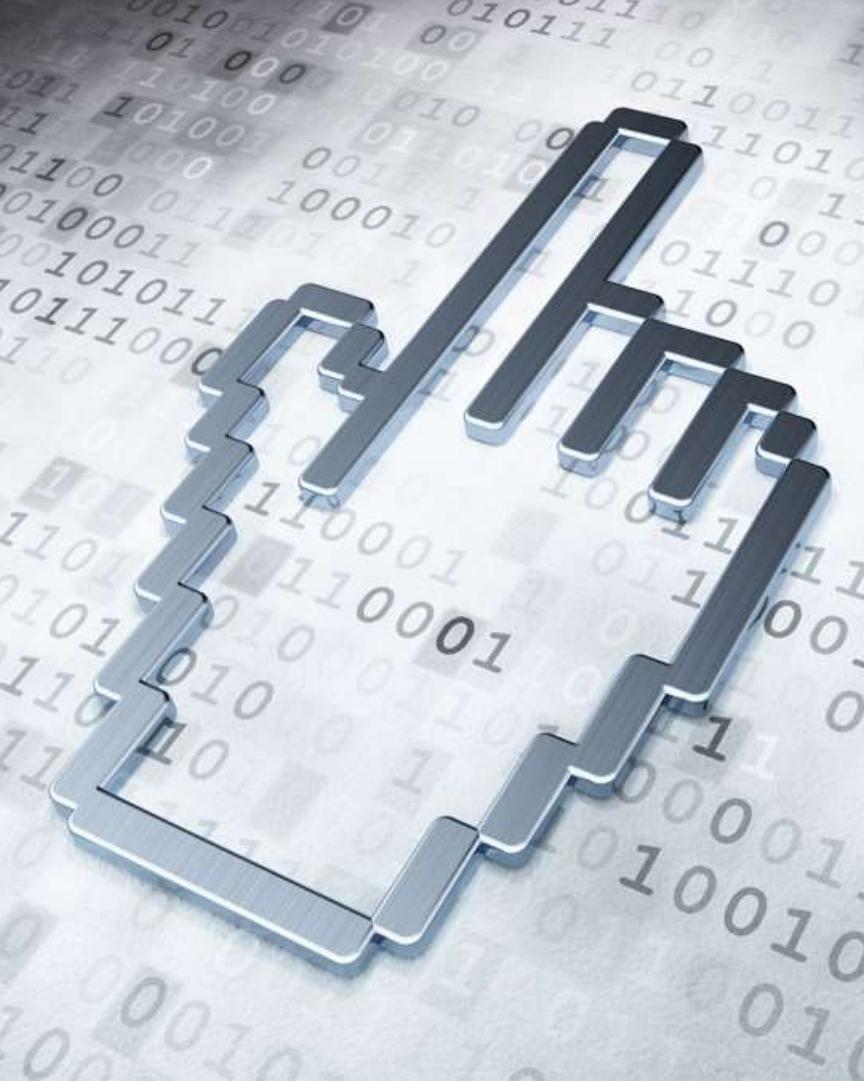


在沿着循环运行时，如果有一组归纳变量的值的变化保持步调一致，常常可以将这组变量删除为只剩一个

问题：如何删除
归纳变量？

⑥ 删除归纳变量





提纲

8.1 流图

8.2 优化的分类

8.3 基本块的优化

8.4 数据流分析

8.5 流图中的循环

8.6 全局优化

8.3 基本块的优化(局部优化)

- 很多重要的局部优化技术首先把一个基本块转换成为一个无环有向图(*directed acyclic graph, DAG*)
- 在转换成DAG过程中和转换后，都可以进行基本块的优化，最后将新DAG重组为优化后的语句序列

三地址码
语句

$$a = b$$

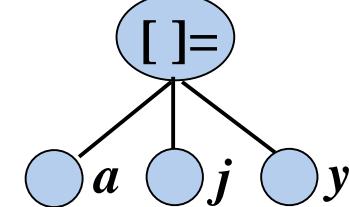
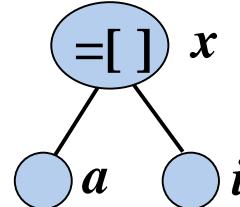
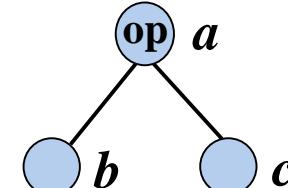
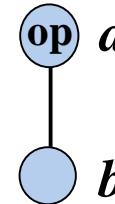
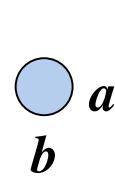
$$a = \text{op } b$$

$$a = b \text{ op } c$$

$$x = a[i]$$

$$a[j] = y$$

DAG子图



语句 $s : a = b \text{ op } c$ s 是对变量 a 的定值，对变量 b 和 c 的引用。

基本块的 DAG 表示

➤ 例

$$a = b + c$$

$$b = a - d$$

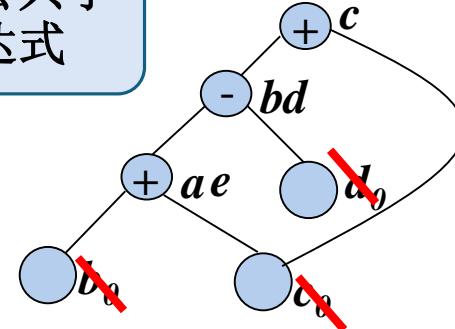
$$c = b + c$$

$$d = a - d$$

$$e = a$$

对于形如 $x=y+z$ 的三地址指令，如果已经有一个结点表示 $y+z$ ，就不向DAG中增加新的结点，而是给已经存在的结点附加定值变量 x

发现公共子表达式



➤ 基本块中的每个语句 s 都对应一个内部结点 N

➤ 结点 N 的标号是 s 中的运算符；结点 N 同时关联一个定值变量表，表示 s 是在此基本块内最晚对表中变量进行定值的语句

➤ N 的子结点是基本块中在 s 之前、最后一个对 s 所使用的运算分量进行定值的语句对应的结点。如果 s 的某个运算分量在基本块内没有在 s 之前被定值，则这个运算分量的子结点就是叶结点(其定值变量表中的变量加上下脚标0)

➤ 在为语句 $x=y+z$ 构造结点 N 的时候，如果 x 已经在某结点 M 的定值变量表中，则从 M 的定值变量表中删除变量 x

基于基本块 DAG的代数恒等式变换

► DAG构造过程可以应用以下或更多的代数变换实现优化，比如交换律和结合律。

(1) 代数恒等式的优化 —— 无需对 x 赋值

$$x+0 = 0+x = x \quad x-0 = x$$

$$x * 1 = 1 * x = x \quad x/1 = x$$

(2) 局部强度削弱，即用较快的运算符取代较慢的运算符，例如：

$$x^{**2} = x * x \quad 2.0 * x = x + x \quad x/2 = x * 0.5$$

(3) 常量合并 对常量表达式进行计算，取代常量表达式。

表达式 $2*3.14$ 可以替换为 6.28 。

(4) 交换率 $x*y = y*x$ ，寻找是否已存在运算分量是 x 和 y 的结点

基于基本块 DAG的无用代码删除

- 从一个DAG上删除所有没有附加活跃变量（活跃变量是指其值可能会在以后被使用的变量）的根结点(即没有父结点的结点)。重复应用这样的处理过程就可以从DAG中消除所有对应于无用代码的结点

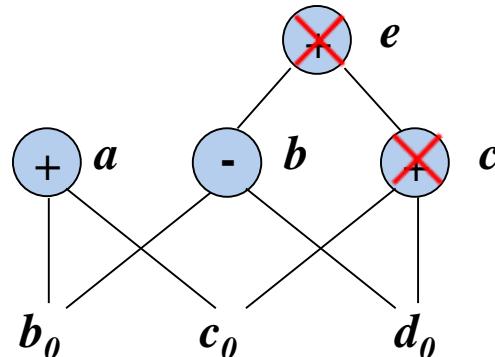
- 例

$$a = b + c$$

$$b = b - d$$

$$c = c + d$$

$$e = b + c$$



假设 a 和 b 是活跃变量，但 c 和 e 不是

数组元素赋值指令的表示

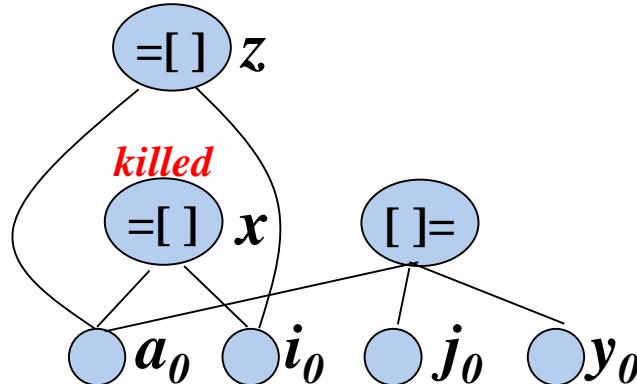
例

$x = a[i]$

$a[j] = y$

$z = a[i]$

在构造DAG时，如何防止系统将 $a[i]$ 误判为公共子表达式？



- 对于形如 $a[j] = y$ 的三地址指令，创建一个运算符为“ $[\]=$ ”的结点，这个结点有3个子结点，分别表示 a 、 j 和 y
- 该结点没有定值变量表
- 该结点的创建将注销(杀死)所有已经建立的、其值依赖于 a 的结点
- 一个被注销(杀死)的结点不能再获得任何新的定值变量，也就是说，它不可能成为一个公共子表达式

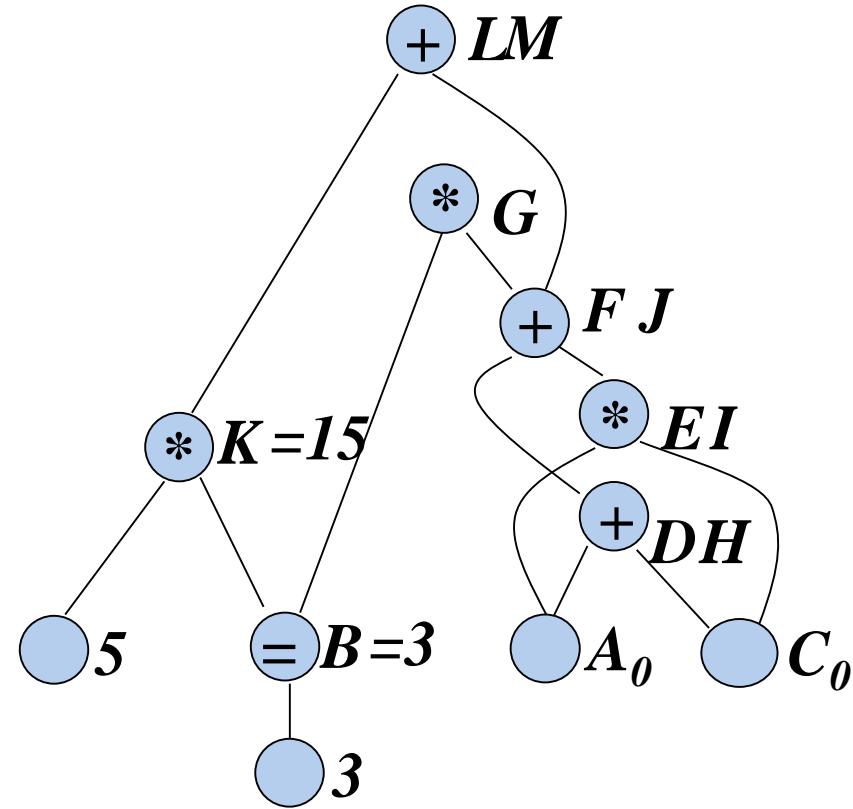
从 DAG 到基本块的重组

- 对每个具有若干定值变量的结点，构造一个三地址语句来计算其中某个定值变量的值
- 指令顺序必须遵守DAG中结点的计算顺序
- 对于每个结点，至多只生成一条实现运算的语句
- 倾向于把计算得到的结果赋给一个在基本块出口处活跃的变量(如果没有全局活跃变量的信息作为依据，就要假设所有变量都在基本块出口处活跃，除了编译器生成的临时变量)
- 如果结点有多个附加的活跃变量，就必须引入复制语句，以便给每一个变量都赋予正确的值
- 出口处不活跃且未被引用的结点，不必计算

例

➤ 给定一个基本块

- ① $B = 3$
- ② $D = A + C$
- ③ $E = A * C$
- ④ $F = E + D$
- ⑤ $G = B * F$
- ⑥ $H = A + C$
- ⑦ $I = A * C$
- ⑧ $J = H + I$
- ⑨ $K = B * 5$
- ⑩ $L = K + J$
- ⑪ $M = L$

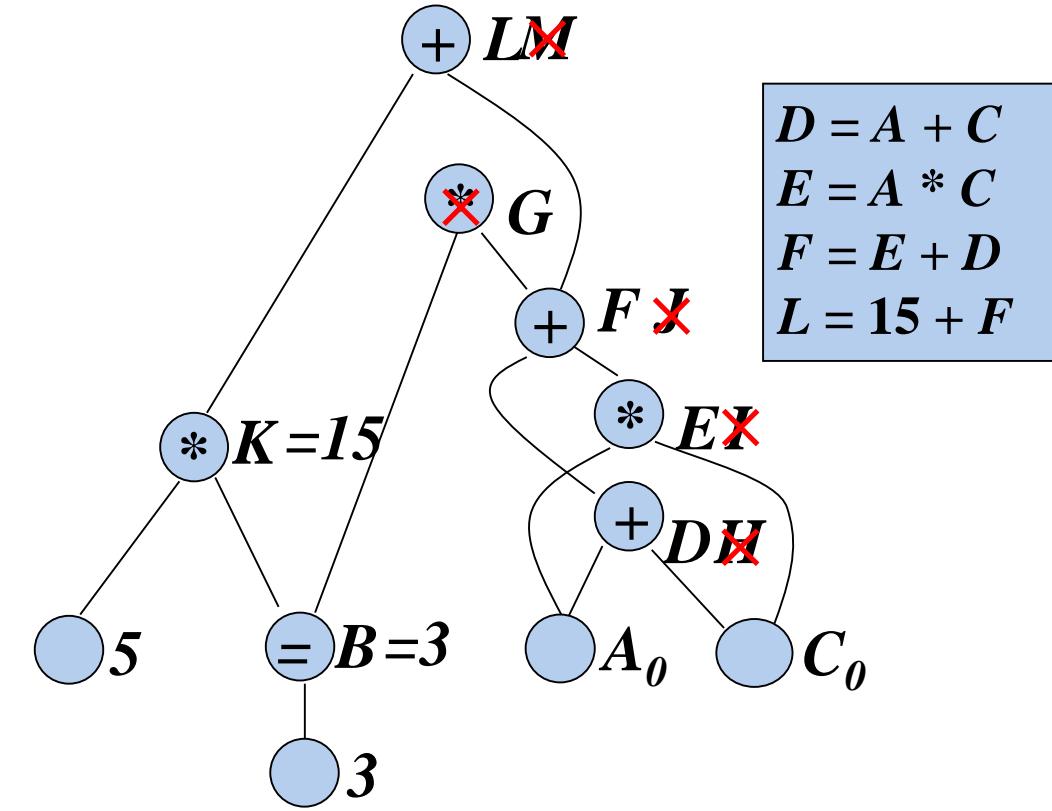


假设：仅变量 L 在基本块出口之后活跃

例

➤ 给定一个基本块

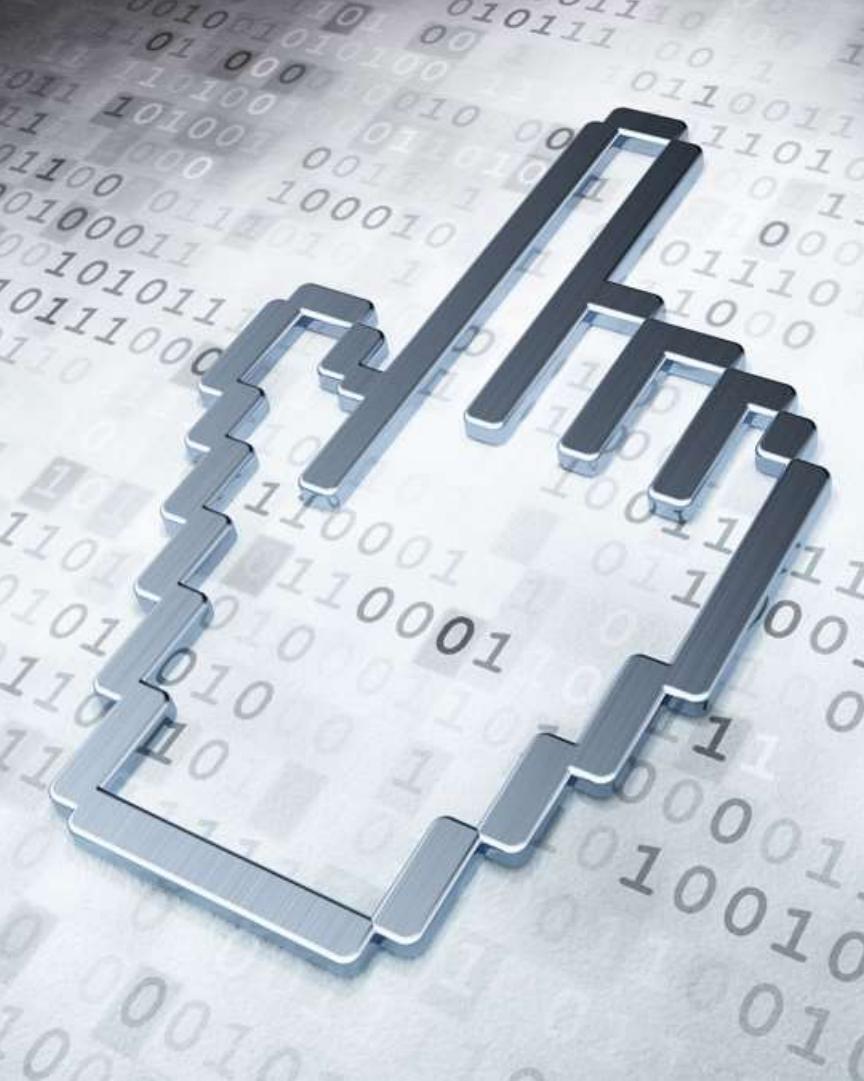
- ① $B = 3$
- ② $D = A + C$
- ③ $E = A * C$
- ④ $F = E + D$
- ⑤ $G = B * F$
- ⑥ $H = A + C$
- ⑦ $I = A * C$
- ⑧ $J = H + I$
- ⑨ $K = B * 5$
- ⑩ $L = K + J$
- ⑪ $M = L$



假设：仅变量 L 在基本块出口之后活跃

练习1

- 练习1：为下面的基本块构造DAG，假设只有b在出口处活跃，重组基本块的代码。
- $d = b * c$
- $e = a + b$
- $b = b * c$
- $a = e - d$



提纲

8.1 流图

8.2 优化的分类

8.3 基本块的优化

8.4 数据流分析

8.5 流图中的循环

8.6 全局优化

8.4 数据流分析(data-flow analysis)

➤ 程序状态

- 程序中所有的变量值以及运行栈的栈帧相关值

➤ 程序点和程序状态

- 语句之前和之后的程序点状态：某个中间代码语句的执行是对程序状态的转换，将输入状态转换为输出状态，输入状态与该语句之前的程序点相关联，而输出状态与该语句之后的程序点相关联
- 基本块之前和之后的程序点的状态也是数据流分析关注的重点

➤ 数据流抽象

- 为了简化，通常针对特定的分析目标，从各个程序点中抽象出一部分程序状态（数据流）信息，而忽略其它暂不关心的信息

8.4 数据流分析(data-flow analysis)

➤ 数据流分析

- 一组用来获取程序执行路径上的数据流信息的技术
- 在每一种数据流分析应用中，都会把每个**程序点**和一个**数据流值**关联起来

➤ 数据流分析应用

- 到达-定值分析 (*Reaching-Definition Analysis*)
- 活跃变量分析 (*Live-Variable Analysis*)
- 可用表达式分析 (*Available-Expression Analysis*)

数据流分析

- 数据流分析的途径：建立和求解数据流方程
- 数据流方程
 - 用于描述流入和流出某个程序单元的数据流信息之间的联系。程序单元可以是单条语句、基本块或循环等
- 数据流方程的构建：
 - 语义约束：一个程序单元 u 之前和之后的数据流值的关系，由语句语义推导出来，被称为传递函数
 - 控制流约束：由控制流推导出的相邻程序单元之间的数据流值的关系

数据流分析模式

➤ 语句的数据流模式

➤ $IN[s]$: 语句 s 之前的数据流值

$OUT[s]$: 语句 s 之后的数据流值

➤ 语义约束 : f_s : 语句 s 的传递函数(transfer function)

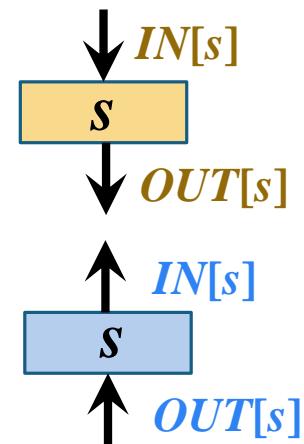
➤ 传递函数的两种风格

➤ 信息沿执行路径前向传播 (前向数据流问题)

$$OUT[s] = f_s(IN[s])$$

➤ 信息沿执行路径逆向传播 (逆向数据流问题)

$$IN[s] = f_s(OUT[s])$$



➤ 数据流分析模式

➤ 语句的数据流模式

- $IN[s]$: 语句 s 之前的数据流值
- $OUT[s]$: 语句 s 之后的数据流值
- 语义约束 : f_s : 语句 s 的传递函数(transfer function)
- 控制流约束 : 基本块内相邻两个语句之间的数据流值的关系简单, 而基本块之间相邻两个语句之间的数据流值的关系复杂, 先考虑基本块内的情况 :
 - 设基本块 B 由语句 s_1, s_2, \dots, s_n 顺序组成, 则

$$IN[s_{i+1}] = OUT[s_i] \quad i=1, 2, \dots, n-1$$

基本块上的数据流模式

- $IN[B]$: 紧靠基本块 B 之前的数据流值 $OUT[B]$
- $OUT[B]$: 紧随基本块 B 之后的数据流值 $= OUT[s_n]$
- 设基本块 B 由语句 s_1, s_2, \dots, s_n 顺序组成, 则
 - $IN[B] = IN[s_1]$ $= f_{sn}(IN[s_n])$
 - $OUT[B] = OUT[s_n]$ $= f_{sn}(OUT[s_{n-1}])$
 - 语义约束 f_B : 基本块 B 的传递函数 $= f_{sn} \cdot f_{s(n-1)}(IN[s_{n-1}])$
 - 前向数据流问题: $OUT[B] = f_B(IN[B])$ $= f_{sn} \cdot f_{s(n-1)} \cdot \dots \cdot f_{s2}(OUT[s_1])$
 - 逆向数据流问题: $IN[B] = f_B(OUT[B])$ $= f_{sn} \cdot f_{s(n-1)} \cdot \dots \cdot f_{s2} \cdot f_{s1}(IN[s_1])$
 - $f_B = f_{s1} \cdot f_{s2} \cdot \dots \cdot f_{sn}$ $= f_{sn} \cdot f_{s(n-1)} \cdot \dots \cdot f_{s2} \cdot f_{s1}(IN[B])$

基本块上的数据流模式

- $IN[B]$: 紧靠基本块 B 之前的数据流值
 $OUT[B]$: 紧随基本块 B 之后的数据流值
- 语义约束 f_B : 基本块 B 的传递函数
 - 前向数据流问题: $OUT[B] = f_B(IN[B])$
 - 逆向数据流问题: $IN[B] = f_B(OUT[B])$
- 控制流约束: 相邻基本块之间的数据流值的关系, 与具体的数据流分析目标有关。
 - 前向流问题: 如可用表达式分析: $IN[B] = \cap_{P \text{ 是 } B \text{ 的前驱}} OUT[P]$
 - 逆向流问题: 如活跃变量分析: $OUT[B] = \cup_{S \text{ 是 } B \text{ 的后继}} IN[S]$

8.4.1 到达定值分析(*Reaching definitions*)

➤ 定值 (*Definition*)

➤ 变量 x 的定值是(可能)将一个值赋给 x 的语句

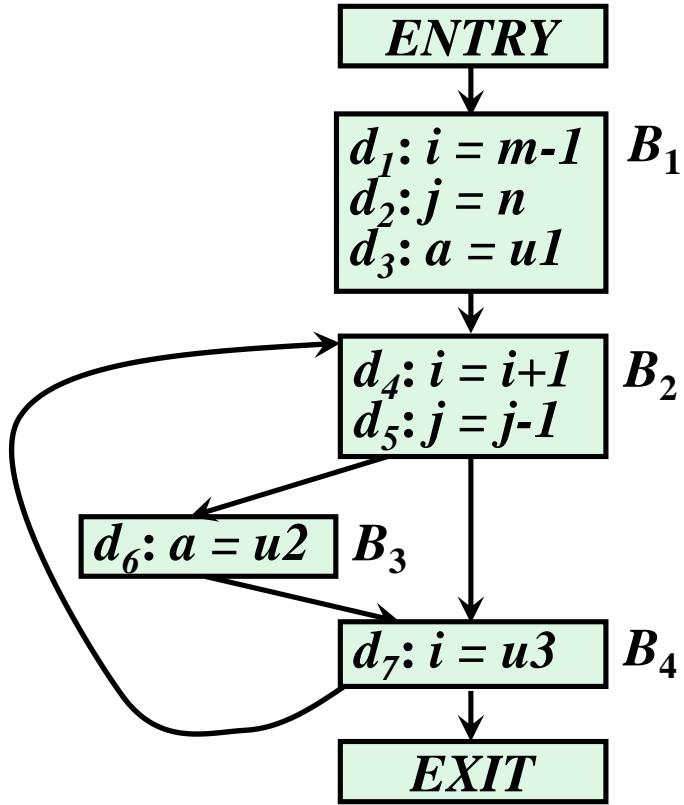
➤ 到达定值 (*Reaching Definition*)

➤ 如果存在一条从紧跟在 x 的定值 d 后面的点到达某一程序点 p 的路径，而且在此路径上没有对 x 的其它定值，则称定值 d 到达程序点 p

➤ 如果在此路径上有对变量 x 的其它定值 d' ，则称定值 d 被定值 d' “注销(杀死)”了，即定值 d 不能到达程序点 p

➤ 直观地讲，如果某个变量 x 的一个定值 d 到达点 p ，在点 p 处使用的 x 的值可能就是由 d 最后赋予的

例：可以到达各基本块的入口处的定值



假设每个控制流图都有两个**空基本块**，分别是表示流图的开始点的**ENTRY**结点和结束点的**EXIT**结点（所有离开该图的控制流都流向它）

$IN[B]$	B_2	B_3	B_4
d_1	✓	✗	✗
d_2	✓	✗	✗
d_3	✓	✓	✓
d_4	✗	✓	✓
d_5	✓	✓	✓
d_6	✓	✓	✓
d_7	✓	✗	✗

到达定值分析的主要用途

- 循环不变计算的检测
- 如果循环中含有赋值 $x=y+z$ ，而 y 和 z 所有可能的定值都在循环外面(包括 y 或 z 是常数的特殊情况)，那么 $y+z$ 就是循环不变计算

到达定值分析的主要用途

- 循环不变计算的检测
- 常量合并
 - 如果对变量 x 的某次使用只有一个定值可以到达，并且该定值把一个常量赋给 x ，那么可以简单地把 x 替换为该常量

到达定值分析的主要用途

- 循环不变计算的检测
- 常量合并
- 判定变量 x 在 p 点上是否有可能未经定值就被引用
 - 在流图的入口点对每个变量 x 引入一个哑定值。如果 x 的哑定值到达了一个可能使用 x 的程序点 p ,那么 x , 就可能在定值之前被引用

“生成”与“注销”定值

这里，“+”代表一个
一般性的二元运算符

- 定值 $d: u = v + w$
- 该语句“生成”了一个对变量 u 的定值 d ，并“注销（杀死）”了程序中所有其它对 u 的定值
- 维持其它变量的定值没有变化

到达定值的传递函数

➤ f_d : 定值 $d: u = v + w$ 的传递函数

生成-注销(杀死)形式

➤ $OUT[S] = f_d(IN[S]) = gen_d \cup (IN[S] - kill_d)$

➤ gen_d : 由语句 d 生成的定值的集合 $gen_d = \{ d \}$

➤ $kill_d$: 由语句 d 注销(杀死)的定值的集合 (即程序中所有其它对 u 的定值)

到达定值的传递函数

➤ f_d : 定值 $d: u = v + w$ 的传递函数

➤ $OUT[S] = f_d(IN[S]) = gen_d \cup (IN[S] - kill_d)$

➤ f_B : 基本块 B 的传递函数, 假设基本块 B 包含 n 条语句

➤ $OUT[B] = f_B(IN[B]) = gen_B \cup (IN[B] - kill_B)$

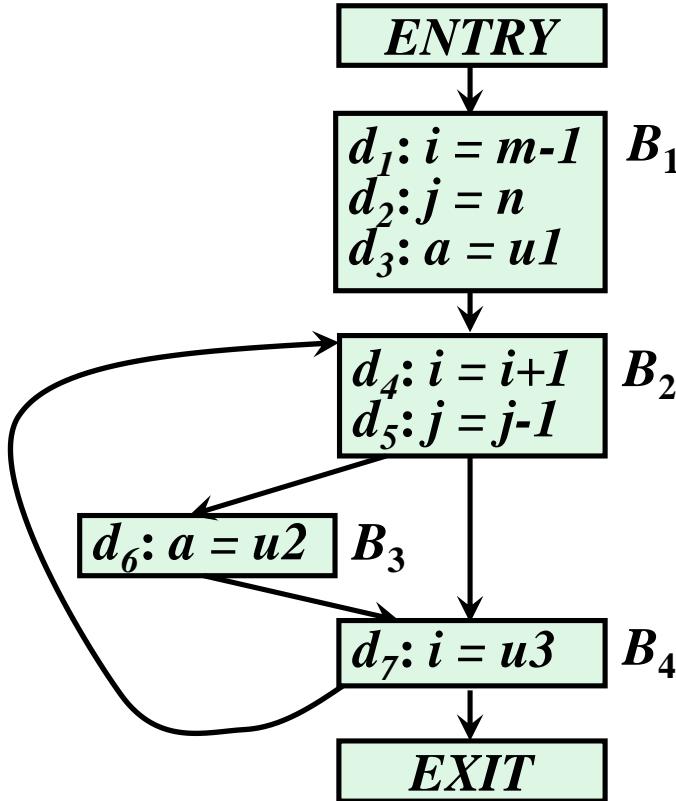
➤ $kill_B = kill_1 \cup kill_2 \cup \dots \cup kill_n$

➤ 被基本块中各个语句 “注销(杀死)” 的所有定值的集合

➤ $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \dots \cup (gen_1 - kill_2 - kill_3 - \dots - kill_n)$

➤ 基本块生成且未被其后的语句 “注销(杀死)” 的定值的集合

例：各基本块 B 的 gen_B 和 $kill_B$

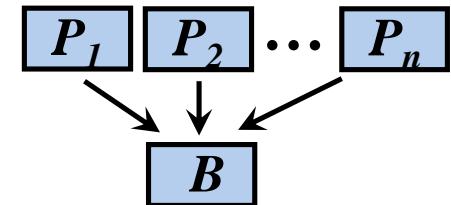


- $gen_{B1} = \{ d_1, d_2, d_3 \}$
- $kill_{B1} = \{ d_4, d_5, d_6, d_7 \}$
- $gen_{B2} = \{ d_4, d_5 \}$
- $kill_{B2} = \{ d_1, d_2, d_7 \}$
- $gen_{B3} = \{ d_6 \}$
- $kill_{B3} = \{ d_3 \}$
- $gen_{B4} = \{ d_7 \}$
- $kill_{B4} = \{ d_1, d_4 \}$

到达定值的数据流方程

- $IN[B]$: 到达流图中基本块 B 的入口处的定值的集合
- $OUT[B]$: 到达流图中基本块 B 的出口处的定值的集合
- 建立数据流方程
 - $OUT[ENTRY] = \Phi$
 - $OUT[B] = f_B(IN[B]) = gen_B \cup (IN[B] - kill_B)$ ($B \neq ENTRY$)
 - $IN[B] = \bigcup_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P]$ ($B \neq ENTRY$)

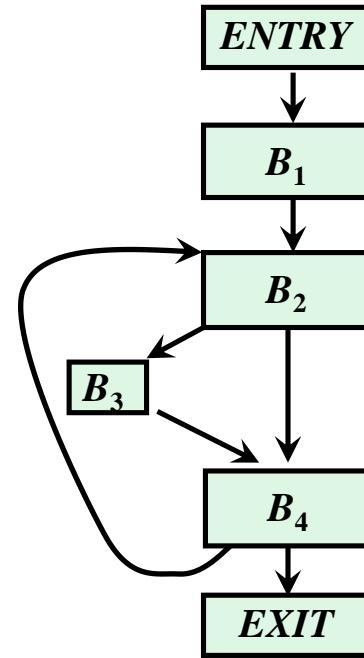
gen_B 和 $kill_B$ 的值可以根据流图预先计算出来，因此在方程中作为已知量



计算到达定值的迭代算法

- 输入：
 - 流图 G , 其中每个基本块 B 的 gen_B 和 $kill_B$ 都已计算出来
- 输出：
 - $IN[B]$ 和 $OUT[B]$
- 方法：
 - $OUT[ENTRY] = \Phi;$
 - for** (除 $ENTRY$ 之外的每个基本块 B) $OUT[B] = \Phi;$
 - while** (某个 OUT 值发生了改变)
 - for** (除 $ENTRY$ 之外的每个基本块 B) {
 - $IN[B] = \cup_{P \text{是}B\text{的一个前驱}} OUT[P];$
 - $OUT[B] = gen_B \cup (IN[B] - kill_B)$
 - }

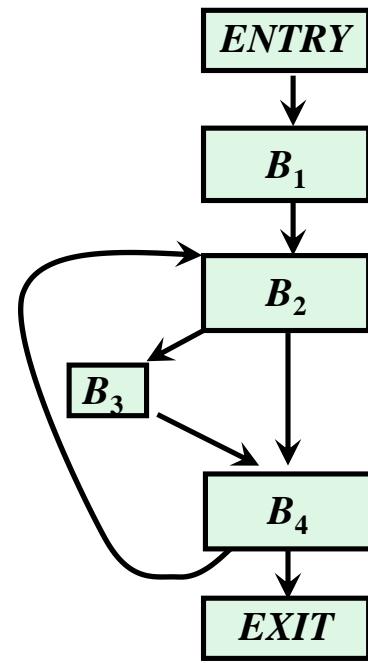
例



$gen_{BI} = \{ d_1, d_2, d_3 \}$
 $kill_{BI} = \{ d_4, d_5, d_6, d_7 \}$
 $gen_{B2} = \{ d_4, d_5 \}$
 $kill_{B2} = \{ d_1, d_2, d_7 \}$
 $gen_{B3} = \{ d_6 \}$
 $kill_{B3} = \{ d_3 \}$
 $gen_{B4} = \{ d_7 \}$
 $kill_{B4} = \{ d_1, d_4 \}$

$OUT[ENTRY] = \Phi;$
for (除 $ENTRY$ 之外的每个基本块 B) $OUT[B] = \Phi$;
while (某个 OUT 值发生了改变)
for (除 $ENTRY$ 之外的每个基本块 B) {
 $IN[B] = \cup_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P];$
 $OUT[B] = gen_B \cup (IN[B]-kill_B)$
}

例

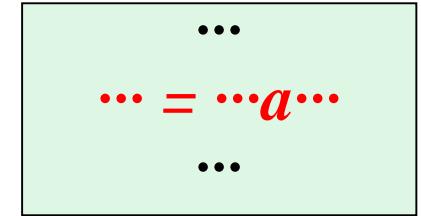
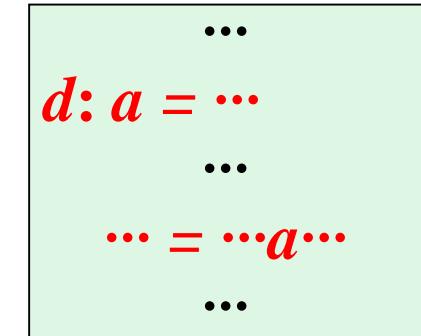


$gen_{BI} = \{ d_1, d_2, d_3 \}$
 $kill_{BI} = \{ d_4, d_5, d_6, d_7 \}$
 $gen_{B2} = \{ d_4, d_5 \}$
 $kill_{B2} = \{ d_1, d_2, d_7 \}$
 $gen_{B3} = \{ d_6 \}$
 $kill_{B3} = \{ d_3 \}$
 $gen_{B4} = \{ d_7 \}$
 $kill_{B4} = \{ d_1, d_4 \}$

<i>IN[B]</i>	<i>B</i>₂	<i>B</i>₃	<i>B</i>₄
<i>d</i> ₁	√	✗	✗
<i>d</i> ₂	√	✗	✗
<i>d</i> ₃	√	√	√
<i>d</i> ₄	✗	√	√
<i>d</i> ₅	√	√	√
<i>d</i> ₆	√	√	√
<i>d</i> ₇	√	✗	✗

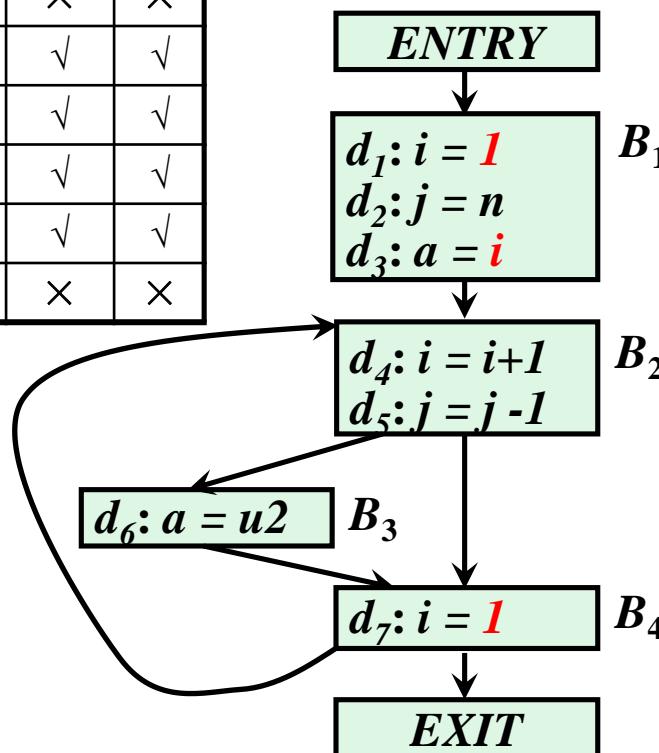
引用-定值链 (Use-Definition Chains)

- 引用-定值链(简称ud链)是一个列表，对于变量的每一次引用，到达该引用的所有定值都在该列表中
- 如果块B中变量 a 的引用之前有 a 的定值，那么只有 a 的最后一次定值会在该引用的ud链中
- 如果块B中变量 a 的引用之前没有 a 的定值，那么 a 的这次引用的ud链就是IN[B]中 a 的定值的集合



引用-定值链 (Use-Definition Chains)

$IN[B]$	B_2	B_3	B_4
d_1	✓	✗	✗
d_2	✓	✗	✗
d_3	✓	✓	✓
d_4	✗	✓	✓
d_5	✓	✓	✓
d_6	✓	✓	✓
d_7	✓	✗	✗



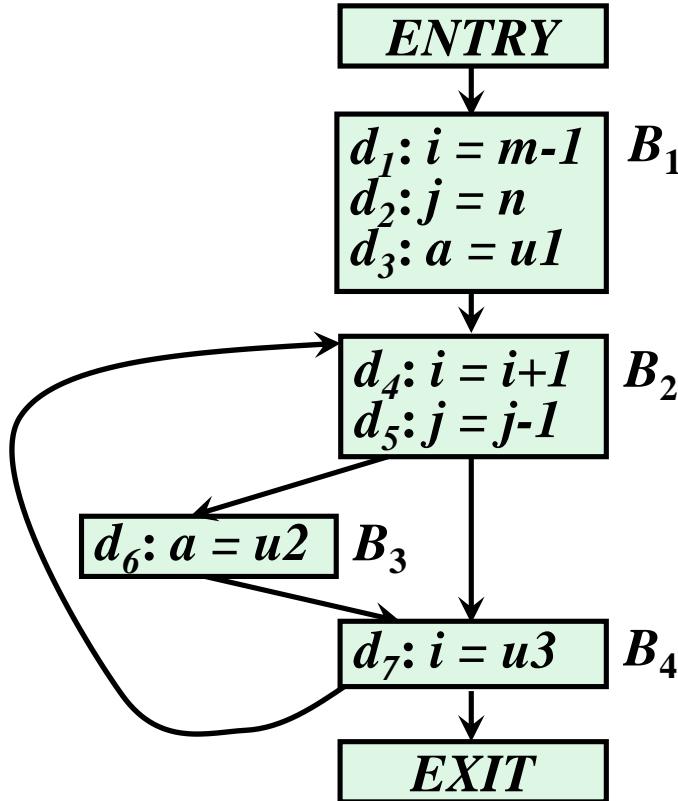
- i 在引用点 d_3 的 UD 链:
 - 块内最近一次定值 { d_1 }
 - 定值为1, 常量合并
- i 在引用点 d_4 的 UD 链:
 - $IN[B_2]$ 中 i 的定值的集合即 { d_1, d_7 }
 - 都是定值为1, 常量合并
- j 在引用点 d_5 的 UD 链:
 - $IN[B_2]$ 中 j 的定值的集合即 { d_2, d_5 }
- u_2 在引用点 d_6 的 UD 链:
 - $IN[B_3]$ 中 u_2 的定值的集合
 - 一定在循环之外, 循环不变计算。

8.4.2 活跃变量分析(*Live-variable Analysis*)

➤ 活跃变量

➤ 对于变量 x 和程序点 p , 如果在流图中沿着从 p 开始的
某条路径会引用变量 x 在 p 点的值, 则称变量 x 在点 p
是**活跃(live)**的, 否则称变量 x 在点 p **不活跃(dead)**

例：各基本块的出口处的活跃变量



$OUT[B]$	B_1	B_2	B_3	B_4
a	✗	✗	✗	✗
i	✓	✗	✗	✓
j	✓	✓	✓	✓
m	✗	✗	✗	✗
n	✗	✗	✗	✗
$u1$	✗	✗	✗	✗
$u2$	✓	✓	✓	✓
$u3$	✓	✓	✓	✓

▶ 活跃变量信息的主要用途

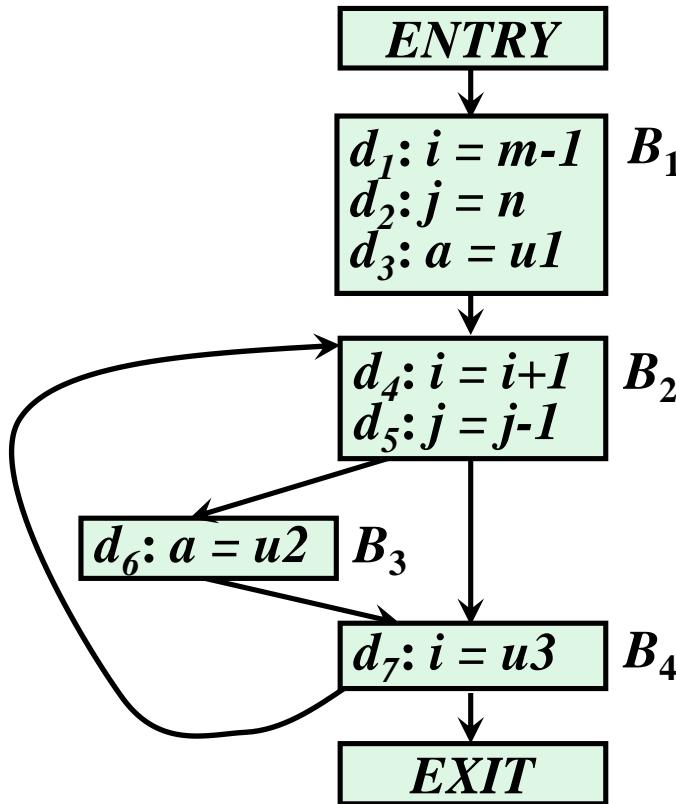
- 删除无用赋值
 - 无用赋值：如果 x 在点 p 的定值在基本块内所有后继点都不被引用，且 x 在基本块出口之后又是不活跃的，那么 x 在点 p 的定值就是无用的
- 为基本块分配寄存器
 - 如果所有寄存器都被占用，并且还需要申请一个寄存器，则应该考虑使用已经存放了不活跃变量(死亡)值的寄存器，因为这个值不需要保存到内存
 - 如果一个值在基本块结尾处是不活跃(死)的就不必在结尾处保存这个值

▶ 活跃变量的传递函数

➤ 逆向数据流问题

- $IN[B] = f_B(OUT[B])$
- $IN[B] = f_B(OUT[B]) = use_B \cup (OUT[B] - def_B)$
- use_B : 在基本块 B 中引用，但是引用前在 B 中没有被定值的变量集合 —— 引用的是 B 的入口处的变量值
- def_B : 在基本块 B 中定值，但是定值前在 B 中没有被引用的变量的集合 —— 引用的不是 B 的入口处的变量值

例：各基本块 B 的 use_B 和 def_B



➤ $use_{B_1} = \{ m, n, u1 \}$

➤ $def_{B_1} = \{ i, j, a \}$

➤ $use_{B_2} = \{ i, j \}$

➤ $def_{B_2} = \Phi$

➤ $use_{B_3} = \{ u2 \}$

➤ $def_{B_3} = \{ a \}$

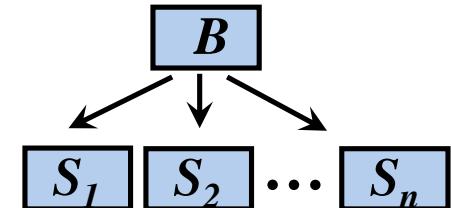
➤ $use_{B_4} = \{ u3 \}$

➤ $def_{B_4} = \{ i \}$

活跃变量数据流方程

- $IN[B]$: 在基本块 B 的入口处的活跃变量集合
- $OUT[B]$: 在基本块 B 的出口处的活跃变量集合
- 建立逆向流方程
 - $IN[EXIT] = \Phi$
 - $IN[B] = f_B(OUT[B]) = use_B \cup (OUT[B] - def_B) \quad (B \neq EXIT)$
 - $OUT[B] = \bigcup_{S \text{ 是 } B \text{ 的一个后继}} IN[S] \quad (B \neq EXIT)$

use_B 和 def_B 的值可以直接从流图计算出来，因此在方程中作为已知量



计算活跃变量的迭代算法

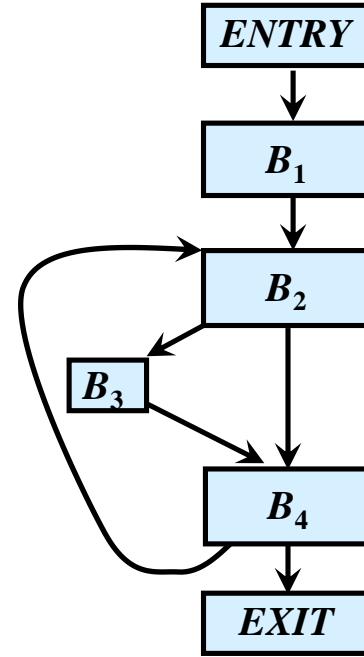
➤ 输入：流图 G ，其中每个基本块 B 的 use_B 和 def_B 都已计算出来

➤ 输出： $IN[B]$ 和 $OUT[B]$

➤ 方法：

```
 $IN[EXIT] = \Phi;$ 
for (除 $EXIT$ 之外的每个基本块 $B$ )  $IN[B] = \Phi$ ;
while (某个 $IN$ 值发生了改变)
    for (除 $EXIT$ 之外的每个基本块 $B$ ) {
         $OUT[B] = \bigcup_{S \text{ 是 } B \text{ 的一个后继}} IN[S];$ 
         $IN[B] = use_B \cup (OUT[B] - def_B);$ 
    }
```

例

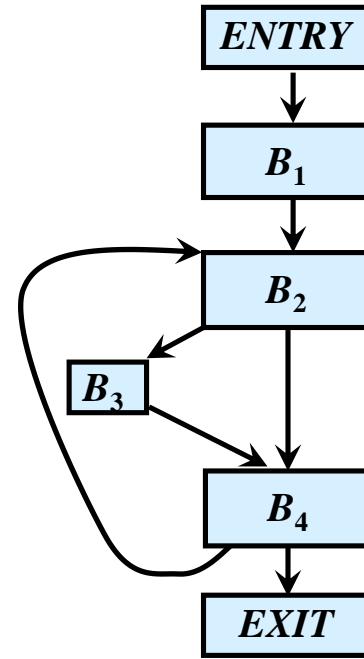


$use_{B1} = \{ m, n, u1 \}$
$def_{B1} = \{ i, j, a \}$
$use_{B2} = \{ i, j \}$
$def_{B2} = \Phi$
$use_{B3} = \{ u2 \}$
$def_{B3} = \{ a \}$
$use_{B4} = \{ u3 \}$
$def_{B4} = \{ i \}$

$IN[EXIT] = \Phi;$
for (除EXIT之外的每个基本块 B) $IN[B] = \Phi$;
while (某个 IN 值发生了改变)
for (除EXIT之外的每个基本块 B) {
 $OUT[B] = \cup_{S \text{是 } B \text{ 的一个后继}} IN[S];$
 $IN[B] = use_B \cup (OUT[B] - def_B);$
}

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4	Φ	$u3$	$i, j, u2, u3$	$j, u2, u3$	$i, j, u2, u3$	$j, u2, u3$
B_3	$u3$	$u2, u3$	$j, u2, u3$	$j, u2, u3$	$j, u2, u3$	$j, u2, u3$
B_2	$u2, u3$	$i, j, u2, u3$	$j, u2, u3$	$i, j, u2, u3$	$j, u2, u3$	$i, j, u2, u3$
B_1	$i, j, u2, u3$	$m, n, u1, u2, u3$	$i, j, u2, u3$	$m, n, u1, u2, u3$	$i, j, u2, u3$	$m, n, u1, u2, u3$

例



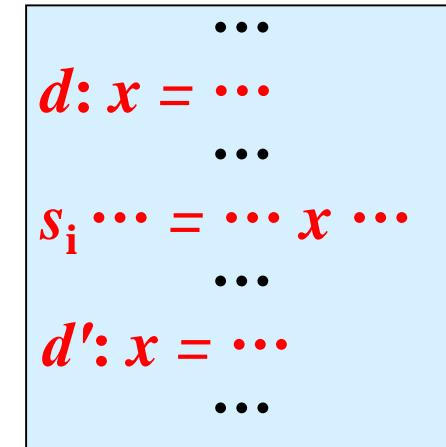
$use_{B1} = \{ m, n, u1 \}$
 $def_{B1} = \{ i, j, a \}$
 $use_{B2} = \{ i, j \}$
 $def_{B2} = \Phi$
 $use_{B3} = \{ u2 \}$
 $def_{B3} = \{ a \}$
 $use_{B4} = \{ u3 \}$
 $def_{B4} = \{ i \}$

$OUT[B]$	B_1	B_2	B_3	B_4
a	✗	✗	✗	✗
i	✓	✗	✗	✓
j	✓	✓	✓	✓
m	✗	✗	✗	✗
n	✗	✗	✗	✗
u_1	✗	✗	✗	✗
u_2	✓	✓	✓	✓
u_3	✓	✓	✓	✓

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4	Φ	$u3$	$i,j,u2,u3$	$j,u2,u3$	$i,j,u2,u3$	$j,u2,u3$
B_3	$u3$	$u2,u3$	$j,u2,u3$	$j,u2,u3$	$j,u2,u3$	$j,u2,u3$
B_2	$u2,u3$	$i,j,u2,u3$	$j,u2,u3$	$i,j,u2,u3$	$j,u2,u3$	$i,j,u2,u3$
B_1	$i,j,u2,u3$	$m,n,u1,u2,u3$	$i,j,u2,u3$	$m,n,u1,u2,u3$	$i,j,u2,u3$	$m,n,u1,u2,u3$

定值-引用链 (*Definition-Use Chains*)

- 定值-引用链：设变量 x 有一个定值 d ，该定值能够到达的所有引用 u 的集合称为 x 在 d 处的定值-引用链，简称 du 链
- 如果在求解活跃变量数据流方程中的 $OUT[B]$ 时，将 $OUT[B]$ 表示成从 B 的末尾处能够到达的引用的集合，那么，可以直接利用这些信息计算基本块 B 中每个变量 x 在其定值处的 du 链
- 如果 B 中 x 的定值 d 之后有 x 的第一个定值 d' ，则 d 和 d' 之间 x 的所有引用构成 d 的 du 链

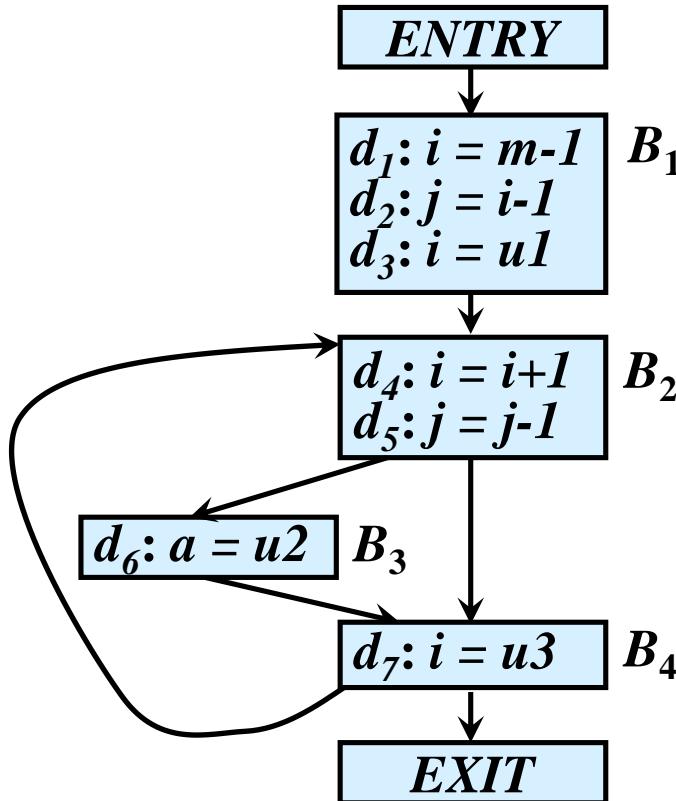


定值-引用链 (*Definition-Use Chains*)

- 定值-引用链：设变量 x 有一个定值 d ，该定值能够到达的所有引用 u 的集合称为 x 在 d 处的定值-引用链，简称 du 链
- 如果在求解活跃变量数据流方程中的 $OUT[B]$ 时，将 $OUT[B]$ 表示成从 B 的末尾处能够到达的引用的集合，那么，可以直接利用这些信息计算基本块 B 中每个变量 x 在其定值处的 du 链
- 如果 B 中 x 的定值 d 之后有 x 的第一个定值 d' ，
则 d 和 d' 之间 x 的所有引用构成 d 的 du 链
- 如果 B 中 x 的定值 d 之后没有 x 的新的定值，
则 B 中 d 之后 x 的所有引用以及 $OUT[B]$ 中 x 的所有引用构成 d 的 du 链

\cdots
 $d: x = \cdots$
 \cdots
 $s_i \cdots = \cdots x \cdots$
 \cdots

定值-引用链 (Definition-Use Chains)



	到达-定值 $OUT[B]$	D_u 链的 $OUT[B]$
B_4	$i, j, u2, u3$	$i \{d_4\}, j \{d_5\}, u2 \{d_6\}, u3 \{d_7\}$
B_3	$j, u2, u3$	$j \{d_5\}, u2 \{d_6\}, u3 \{d_7\}$
B_2	$j, u2, u3$	$j \{d_5\}, u2 \{d_6\}, u3 \{d_7\}$
B_1	$i, j, u2, u3$	$i \{d_4\}, j \{d_5\}, u2 \{d_6\}, u3 \{d_7\}$

- i 在定值点 d_1 的 DU 链为: $\{ d_2 \}$
- j 在定值点 d_2 的 DU 链为: $\{ d_5 \}$
- i 在定值点 d_3 的 DU 链为: $\{ d_4 \}$
- i 在定值点 d_4 的 DU 链为: \emptyset
- j 在定值点 d_5 的 DU 链为: $\{ d_5 \}$
- a 在定值点 d_6 的 DU 链为: \emptyset
- i 在定值点 d_7 的 DU 链为: $\{ d_4 \}$

8.4.3 可用表达式分析(Available Expressions)

➤ 可用表达式

➤ 如果从流图的首结点到达程序点 p 的每条路径都对表达式 $x \text{ op } y$ 进行计算，并且从最后一个这样的计算到点 p 之间没有再次对 x 或 y 定值，那么表达式 $x \text{ op } y$ 在点 p 是可用的(*available*)

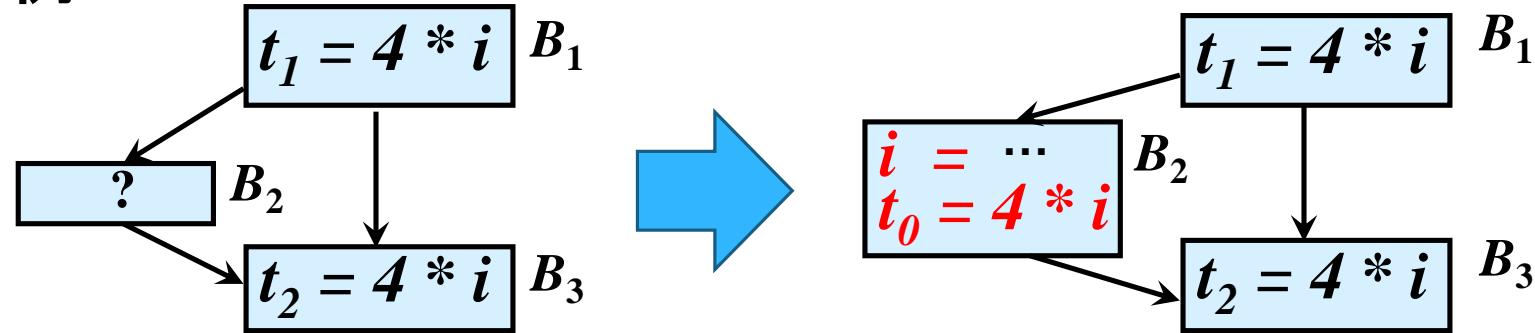
➤ 表达式可用的直观意义

➤ 在点 p 上， $x \text{ op } y$ 已经在之前被计算过，不需要重新计算

可用表达式信息的主要用途

➤ 消除全局公共子表达式

➤ 例



B_2 中的代码满足什么条件， $4 * i$ 在基本块 B_3 的入口点才可用？

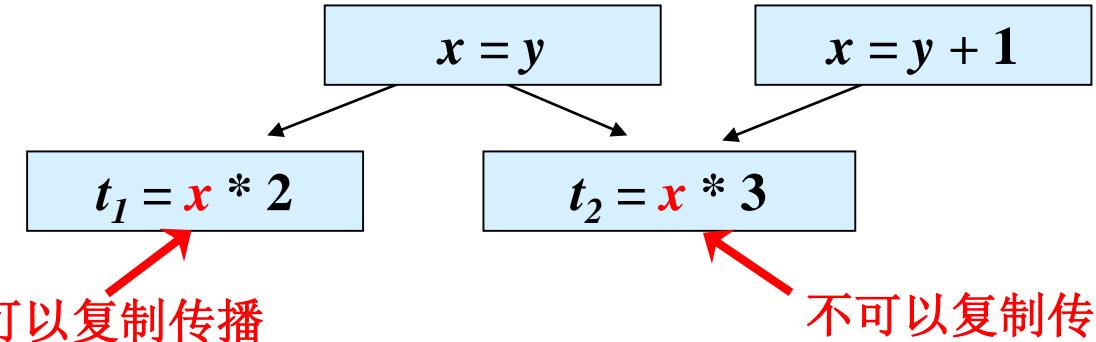
- 未对 i 重新定值
- 或 对 i 重新定值了，又重新计算了 $4 * i$

可用表达式信息的主要用途

➤ 消除全局公共子表达式

➤ 进行复制传播

➤ 例



在 x 的引用点 u 可以用 y 代替 x 的条件: 复制语句 $x = y$ 在引用点 u 处可用

从流图的首结点到达 u 的每条路径都存在复制语句 $x = y$, 并且从最后一条复制语句 $x = y$ 到点 u 之间没有再次对 x 或 y 定值

可用表达式的传递函数

- 如果基本块 B 对 $x \text{ op } y$ 进行计算，并且之后没有重新定值 x 或 y ，则称 B 生成表达式 $x \text{ op } y$ ；如果基本块 B 对 x 或者 y 进行了(或可能进行)定值，且以后没有重新计算 $x \text{ op } y$ ，则称 B 注销(杀死)表达式 $x \text{ op } y$
- $OUT[B] = f_B(IN[B]) = e_gen_B \cup (IN[B] - e_kill_B)$
 - e_gen_B ：基本块 B 所生成的可用表达式的集合
 - e_kill_B ：基本块 B 所注销(杀死)的 U 中的可用表达式的集合
 - U ：所有出现在程序语句右部的表达式的全集

e_gen_B 的计算

- 初始化: $e_gen_B = \Phi$
 - 顺序扫描基本块的每个语句: $z = x \text{ op } y$
 - 把 $x \text{ op } y$ 加入 e_gen_B
 - 从 e_gen_B 中删除和 z 相关的表达式
- } 顺序不能颠倒

块B语句	e_gen_B	$U = \{ b+c, a-d \}$
$a := b+c$	\emptyset	
$b := a-d$	$\{ b+c \}$	—— 加入 $b+c$, 删除 $a-b$
$c := b+c$	$\{ a-d \}$	—— 加入 $a-d$, 删除 $b+c$
$d := a-d$	$\{ a-d \}$	—— 加入 $b+c$, 删除 $b+c$
.....	\emptyset	—— 加入 $a-d$, 删除 $a-d$

e_kill_B 的计算

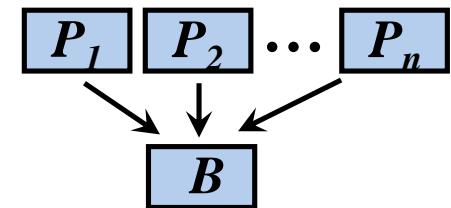
- 初始化: $e_kill_B = \emptyset$
- 顺序扫描基本块的每个语句: $z = x \text{ op } y$
 - 从 e_kill_B 中删除表达式 $x \text{ op } y$
 - 把所有和 z 相关的表达式加入到 e_kill_B 中

块B语句	e_kill_B	$\mathbf{U} = \{ b+c, a-d \}$
$a := b+c$	\emptyset	
$b := a-d$	$\{ a-d \}$	—— 删除 $b+c$, 加入 $a-d$
$c := b+c$	$\{ b+c \}$	—— 删除 $a - d$, 加入 $b+c$
$d := a-d$	$\{ b+c \}$	—— 删除 $b+c$, 加入 $b+c$
.....	$\{ b+c, a-d \}$	—— 删除 $a-d$, 加入 $a-d$

可用表达式的数据流方程

- $IN[B]$: 在 B 的入口处可用的 U 中的表达式集合
- $OUT[B]$: 在 B 的出口处可用的 U 中的表达式集合
- 方程
 - $OUT[ENTRY] = \Phi$
 - $OUT[B] = f_B(IN[B]) = e_gen_B \cup (IN[B] - e_kill_B) \quad (B \neq ENTRY)$
 - $IN[B] = \bigcap_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P] \quad (B \neq ENTRY)$

e_gen_B 和 e_kill_B 的值可以直接从流图计算出来，
因此在方程中作为已知量



计算可用表达式的迭代算法

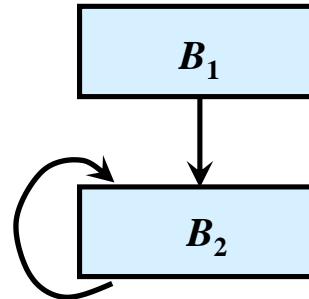
- 输入：流图G，其中每个基本块B的 e_gen_B 和 e_kill_B 都已计算出来
- 输出： $IN[B]$ 和 $OUT[B]$
- 方法：

```
 $OUT[ENTRY] = \Phi;$ 
for (除ENTRY之外的每个基本块B)  $OUT[B] = U;$ 
while (某个 $OUT$ 值发生了改变)
    for (除ENTRY之外的每个基本块B) {
         $IN[B] = \bigcap_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P]$ 
         $OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B);$ 
    }
```

为什么将 $OUT[B]$ 集合初始化为 U ？

➤ 将 OUT 集合初始化为 Φ 局限性太大

➤ 例



➤ 如果 $OUT[B_2]^0 = \Phi$

那么 $IN[B_2]^I = OUT[B_1]^I \cap OUT[B_2]^0 = \Phi$

➤ 如果 $OUT[B_2]^0 = U$

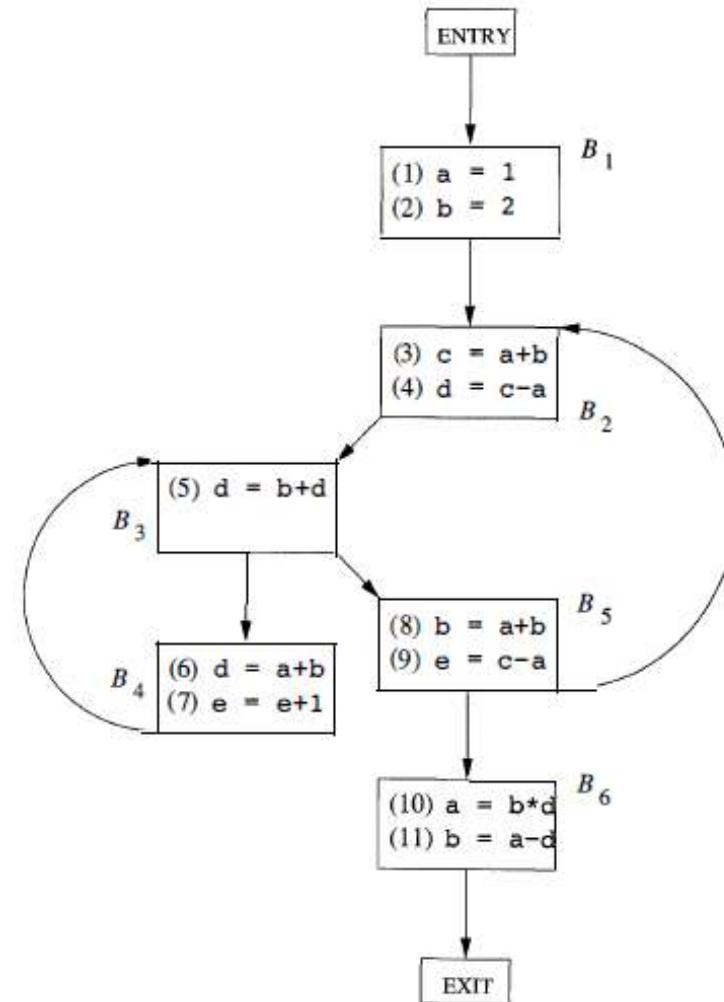
那么 $IN[B_2]^I = OUT[B_1]^I \cap OUT[B_2]^0 = OUT[B_1]$

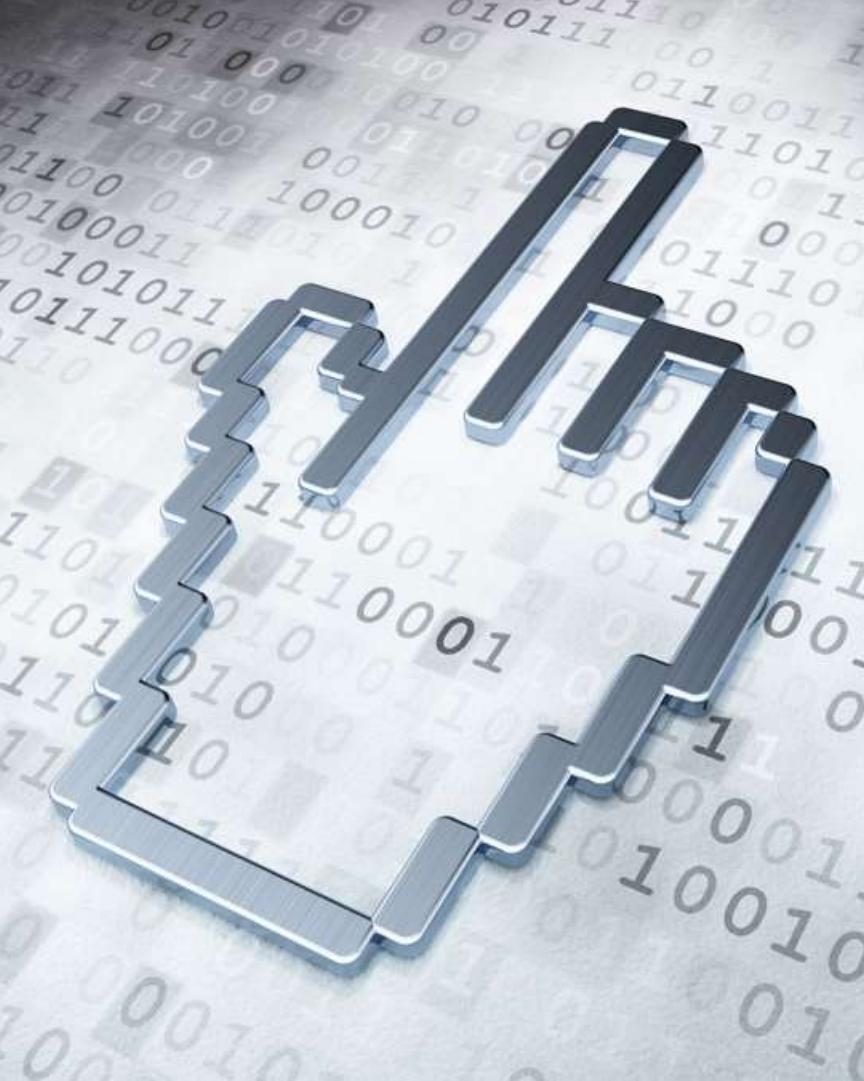
数据流分析小结

	到达定值	活跃变量	可用表达式
域	定值的集合	变量的集合	表达式的集合
方向	前向	逆向	前向
传递函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e_gen_B \cup (x - e_kill_B)$
边界条件	$OUT[ENTRY] = \Phi$	$OUT[EXIT] = \Phi$	$OUT[ENTRY] = \Phi$
交汇运算	交运算	交运算	并运算
方程组	$IN[B] = \bigcup_{P \text{是 } B \text{ 的一个前驱}} OUT[P]$ $OUT[B] = gen_B \cup (IN[B] - kill_B)$	$OUT[B] = \bigcup_{S \text{是 } B \text{ 的一个后继}} IN[S]$ $IN[B] = use_B \cup (OUT[B] - def_B)$	$IN[B] = \bigcap_{P \text{是 } B \text{ 的一个前驱}} OUT[P]$ $OUT[B] = e_gen_B \cup (IN[B] - e_kill_B)$
初始值	$OUT[B] = \Phi$	$IN[B] = \Phi$	$OUT[B] = U$

练习2

➤ 练习2：给出如图所示的流图，计算活跃变量分析中每个基本块的def、use、IN和OUT集合。





提纲

8.1 流图

8.2 优化的分类

8.3 基本块的优化

8.4 数据流分析

8.5 流图中的循环

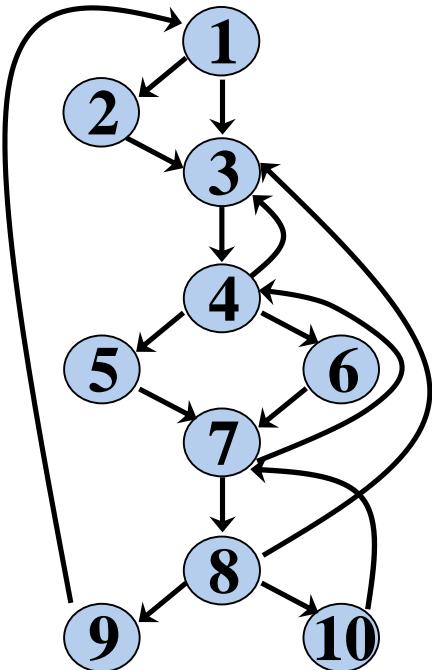
8.6 全局优化

8.5 流图中的循环

- 支配结点 (*Dominators*)
- 如果从流图的入口结点到结点 n 的每条路径都经过结点 d , 则称结点 d 支配(*dominate*)结点 n , 记为 $d \text{ dom } n$

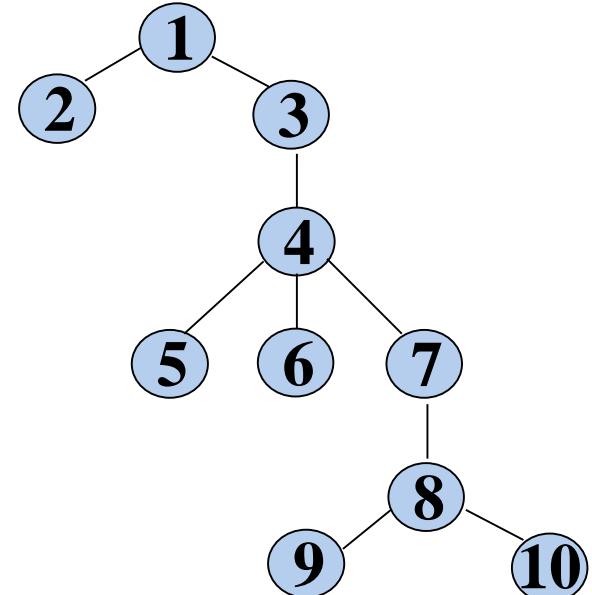
每个结点都支配它自己

例



支配结点	支配对象
1	1~10
2	2
3	3~10
4	4~10
5	5
6	6
7	7~10
8	8~10
9	9
10	10

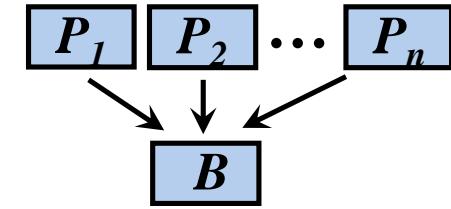
➤ 支配结点树 (*Dominator Tree*)



每个结点只支配它和它的后代结点

寻找支配结点

- 支配结点的数据流方程
 - $IN[B]$: 在基本块 B 入口处的支配结点集合
 - $OUT[B]$: 在基本块 B 出口处的支配结点集合
- 方程
 - $OUT[ENTRY] = \{ ENTRY \}$
 - $OUT[B] = f(IN[B]) = IN[B] \cup \{ B \} \quad (B \neq ENTRY)$
 - $IN[B] = \bigcap_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P] \quad (B \neq ENTRY)$



根据支配结点的定义， $OUT[B]$ 是结点 B 的所有支配结点的集合

计算支配结点的迭代算法

- 输入：流图 G , G 的结点集是 N , 边集是 E , 入口结点是 $ENTRY$
- 输出：对于 N 中的各个结点 n , 给出 $D(n)$, 即支配 n 的所有结点的集合, $D(n) = OUT[n]$
- 方法:

```
     $OUT[ENTRY] = \{ ENTRY \}$ 
    for(除 $ENTRY$ 之外的每个基本块 $B$ )
         $OUT[B] = N$ 
    while(某个 $OUT$ 值发生了改变)
        for(除 $ENTRY$ 之外的每个基本块 $B$ )
            {  $IN[B] = \bigcap_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P]$ 
               $OUT[B] = IN[B] \cup \{ B \}$ 
            }
    }
```

例

$OUT[ENTRY]=\{ENTRY\}$

for(除ENTRY之外的每个基本块B) $OUT[B]=N$

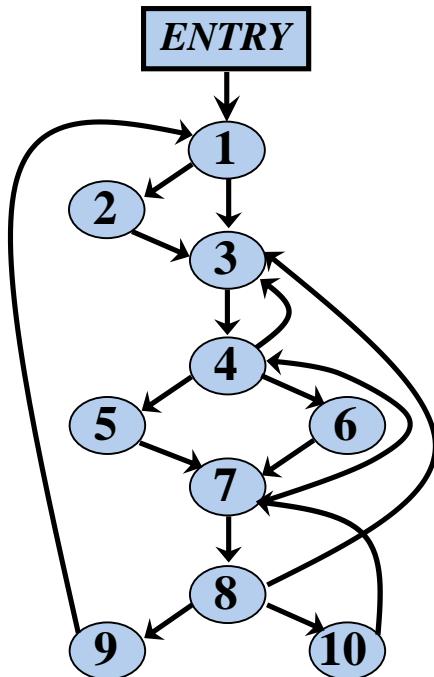
while(某个 OUT 值发生了改变)

for(除ENTRY之外的每个基本块B)

{ $IN[B]=\cap_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P]$

$OUT[B]=IN[B] \cup \{B\}$

}

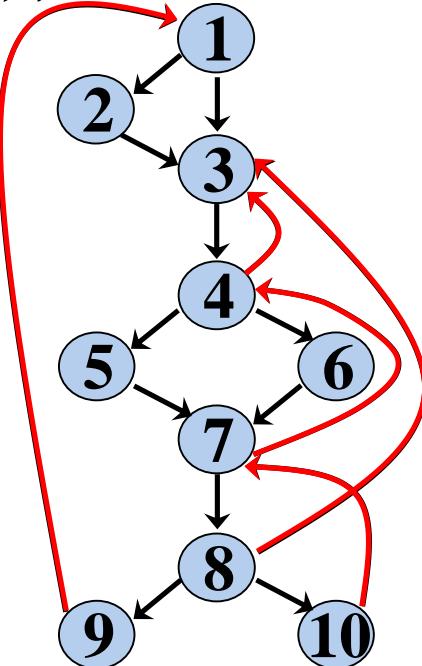


	$OUT^0[B]$	$IN^1[B]$	$OUT^1[B]$
E	{ E }		
①	N	{ E }	{ E ① }
②	N	{ E ① }	{ E ① ② }
③	N	{ E ① }	{ E ① ③ }
④	N	{ E ① ③ }	{ E ① ③ ④ }
⑤	N	{ E ① ③ ④ }	{ E ① ③ ④ ⑤ }
⑥	N	{ E ① ③ ④ }	{ E ① ③ ④ ⑥ }
⑦	N	{ E ① ③ ④ }	{ E ① ③ ④ ⑦ }
⑧	N	{ E ① ③ ④ ⑦ }	{ E ① ③ ④ ⑦ ⑧ }
⑨	N	{ E ① ③ ④ ⑦ ⑧ }	{ E ① ③ ④ ⑦ ⑧ ⑨ }
⑩	N	{ E ① ③ ④ ⑦ ⑧ }	{ E ① ③ ④ ⑦ ⑧ ⑩ }

回边 (Back Edges)

- 如果存在从结点 n 到 d 的有向边 $n \rightarrow d$, 且 $d \text{ dom } n$, 那么这条边称为回边

例



B	$OUT^0[B]$	$IN^i[B]$	$OUT^i[B]$
E	E		
①	N	E	$E \ ①$
②	N	$E \ ①$	$E \ ① \ ②$
③	N	$E \ ①$	$E \ ① \ ③$
④	N	$E \ ① \ ③$	$E \ ① \ ③ \ ④$
⑤	N	$E \ ① \ ③ \ ④$	$E \ ① \ ③ \ ④ \ ⑤$
⑥	N	$E \ ① \ ③ \ ④$	$E \ ① \ ③ \ ④ \ ⑥$
⑦	N	$E \ ① \ ③ \ ④$	$E \ ① \ ③ \ ④ \ ⑦$
⑧	N	$E \ ① \ ③ \ ④ \ ⑦$	$E \ ① \ ③ \ ④ \ ⑦ \ ⑧$
⑨	N	$E \ ① \ ③ \ ④ \ ⑦ \ ⑧$	$E \ ① \ ③ \ ④ \ ⑦ \ ⑧ \ ⑨$
⑩	N	$E \ ① \ ③ \ ④ \ ⑦ \ ⑧$	$E \ ① \ ③ \ ④ \ ⑦ \ ⑧ \ ⑩$

$\downarrow d$

回边:

$4 \rightarrow 3$

$7 \rightarrow 4$

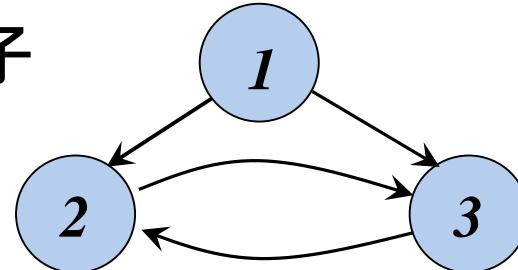
$8 \rightarrow 3$

$9 \rightarrow 1$

$10 \rightarrow 7$

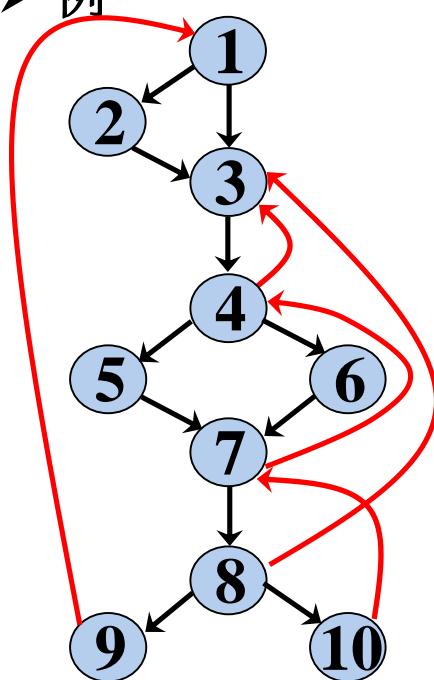
自然循环 (Natural Loops)

- 从程序分析的角度来看，**循环**在代码中以什么形式出现并不重要，重要的是它是否具有易于优化的性质
- **自然循环**是一种**适合于优化**的循环，它满足以下性质
 - 具有唯一的入口结点，称为**首结点**(循环头结点，*header*)
 - 首结点支配循环中的所有结点
 - 循环中至少有一条返回首结点的**回边**，否则，控制就不可能从“循环”中直接回到循环头，也就无法构成循环
- 非自然循环的例子

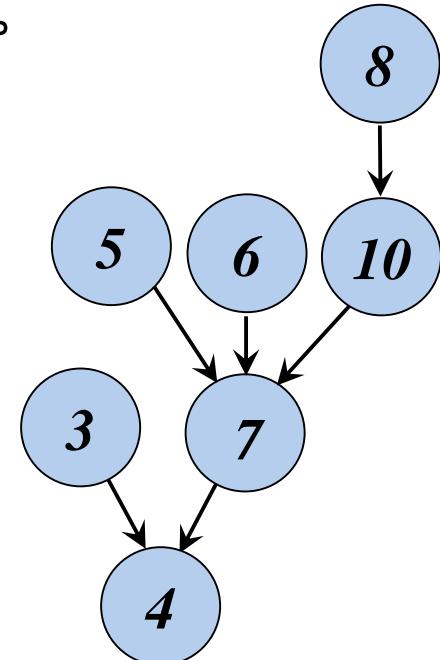


自然循环的识别

- 给定一个回边 $n \rightarrow d$, 该回边的自然循环为: d , 以及所有可以不经过 d 而到达 n 的结点。 d 为该循环的首结点。
 - 例



回边	自然循环
$4 \rightarrow 3$	(3) (4) (5) (6) (7) (8) (10)
$7 \rightarrow 4$	(4) (5) (6) (7) (8) (10)
$8 \rightarrow 3$	(3) (4) (5) (6) (7) (8) (10)
$9 \rightarrow 1$	(1) ~ (10)
$10 \rightarrow 7$	(7) (8) (10)



算法：构造一条回边的自然循环

- 输入：流图 G 和回边 $n \rightarrow d$
- 输出：由回边 $n \rightarrow d$ 的自然循环中的所有结点组成的集合

➤ 方法：
 $stack = \Phi; loop = \{ n, \textcolor{red}{d} \}; push(stack, \textcolor{red}{n});$

while $stack$ 不空 **do**

{ $m = top(stack); pop(stack);$

for m 的每个前驱 p

if p 不在 $loop$ 中 **then**

$loop = loop \cup \{ p \};$

$push(stack, p);$ }

 }

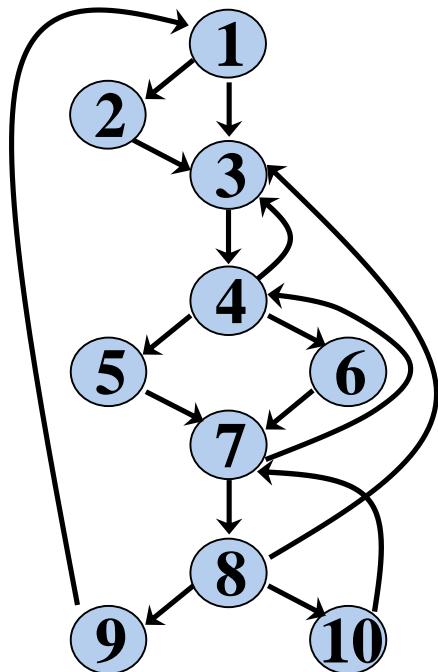
}

结点 d 在初始时刻已经在 $loop$ 中，不会去考虑它的前驱。因此，找出的都是不经过 d 而能到达 n 的结点。

➤ 自然循环的一个重要性质

- 如果两个自然循环的首结点不相同，则这两个循环要么互不相交，要么一个完全包含(嵌入)在另外一个里面

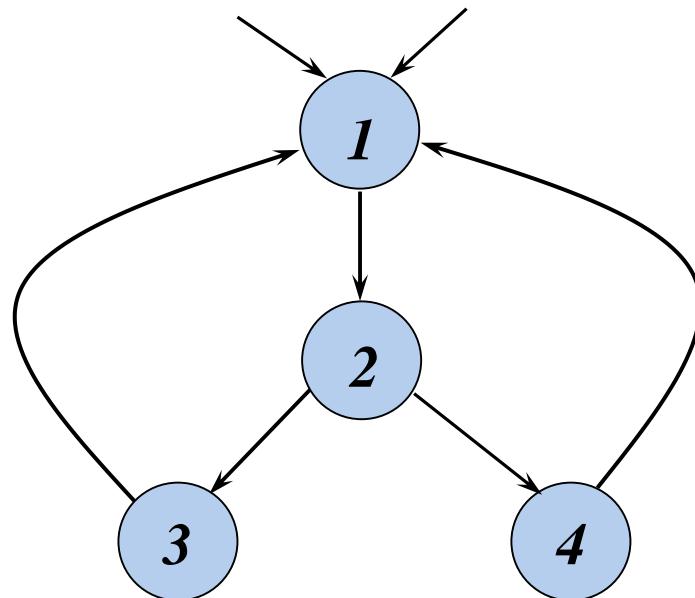
➤ 例

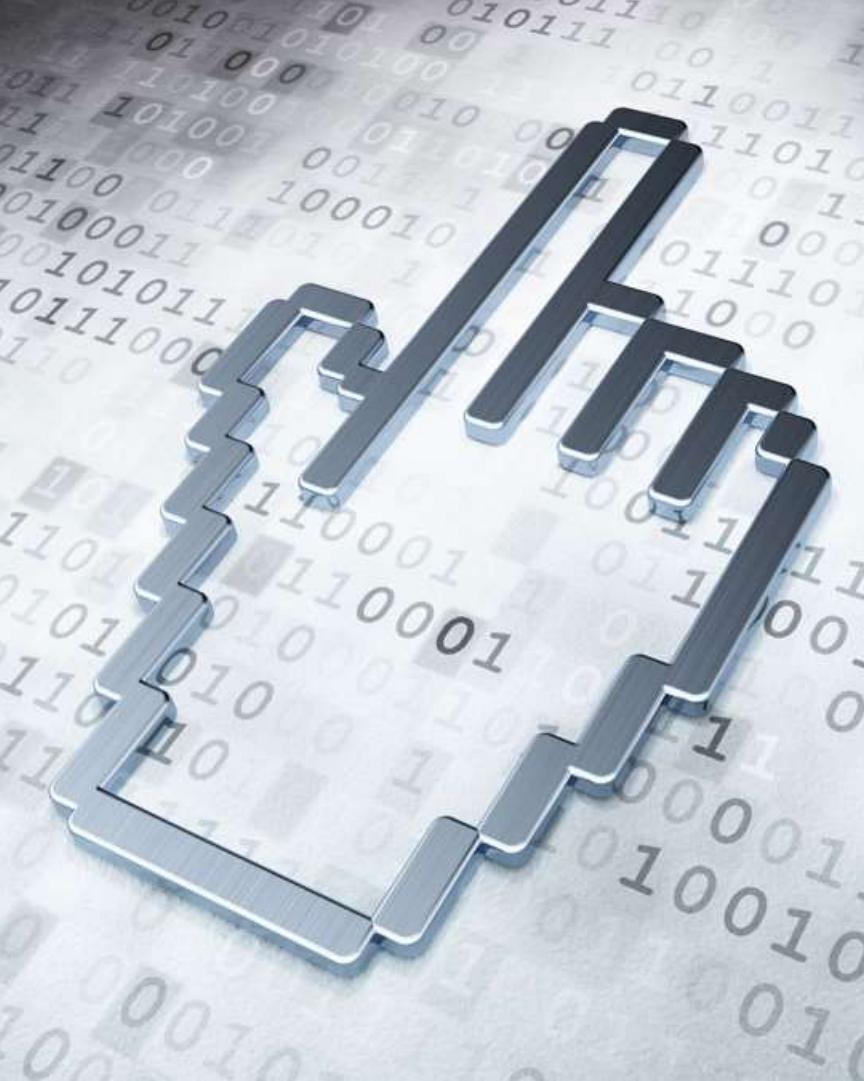


回边	自然循环
4->3	(3)④⑤⑥⑦⑧⑩
7->4	④⑤⑥⑦⑧⑩
8->3	(3)④⑤⑥⑦⑧⑩
9->1	① ~ ⑩
10->7	⑦⑧⑩

最内循环 (Innermost Loops):
不包含其它循环的循环

- 如果两个循环具有相同的首结点，那么很难说哪个是最内循环。此时把两个循环合并





提纲

8.1 流图

8.2 优化的分类

8.3 基本块的优化

8.4 数据流分析

8.5 流图中的循环

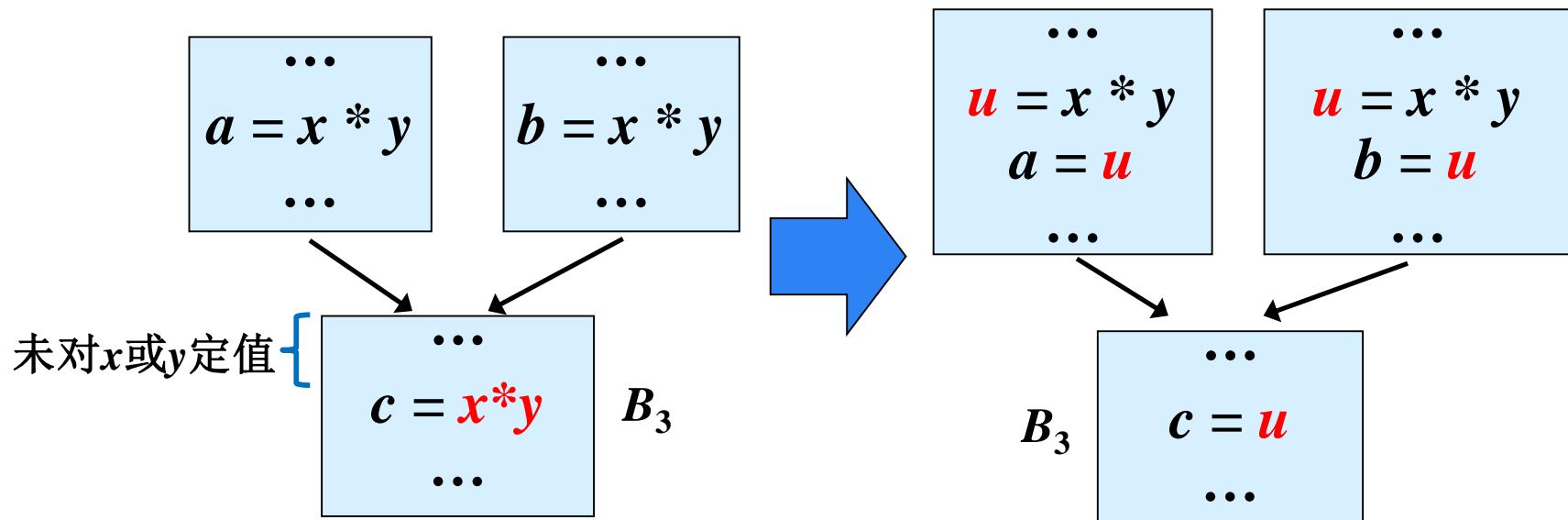
8.6 全局优化

8.6 全局优化

- 删除全局公共子表达式
 - 删除复制语句
 - 代码移动
 - 作用于递归变量的强度削弱
 - 删除递归变量
- } 循环优化

① 删除全局公共子表达式

- 可用表达式的数据流问题可以帮助确定位于流图中 p 点的表达式是否为全局公共子表达式
- 例



全局公共子表达式删除算法

- 输入：带有可用表达式信息的流图
- 输出：修正后的流图
- 方法：
 - 对于语句 $s : z = x \text{ op } y$, 如果 $x \text{ op } y$ 在 s 之前可用, 那么执行如下步骤：
 - ① 从 s 开始逆向搜索, 但不穿过任何计算了 $x \text{ op } y$ 的块·找到所有离 s 最近的计算了 $x \text{ op } y$ 的语句
 - ② 建立新的临时变量 u
 - ③ 把步骤①中找到的语句 $w = x \text{ op } y$ 用下列语句代替：

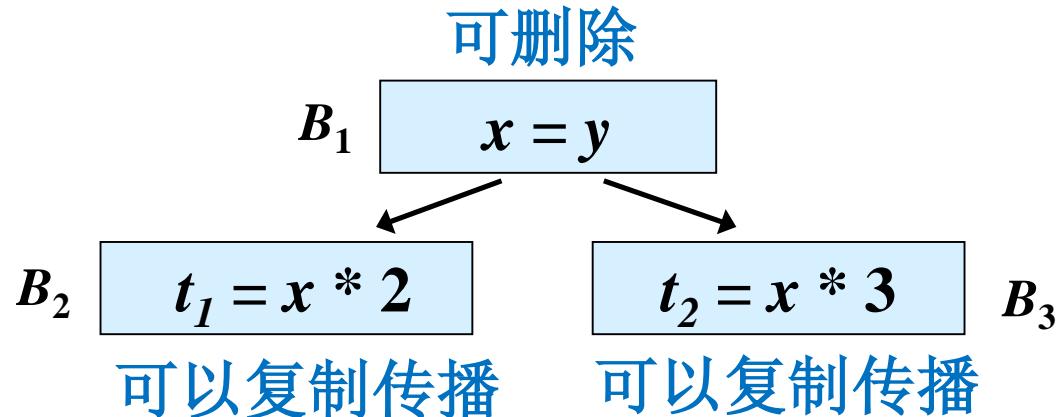
$u = x \text{ op } y$

$w = u$

- ④ 用 $z = u$ 替代 s

② 删除复制语句

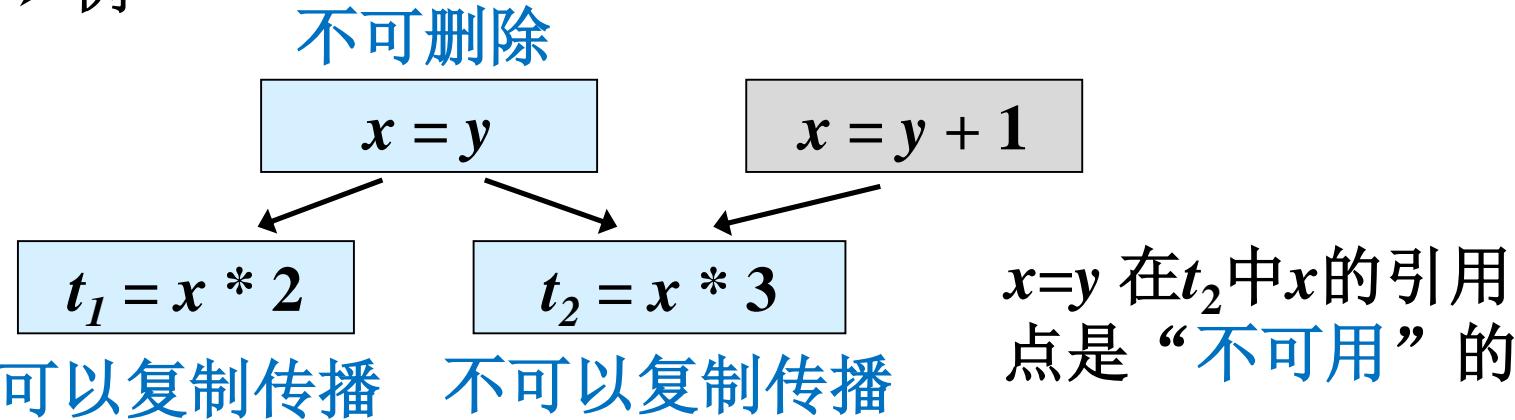
- 对于复制语句 $x=y$, 如果在 x 的所有引用点都可以用对 y 的引用代替对 x 的引用(复制传播), 那么可以删除复制语句 $x=y$
- 例



$x=y$ 在 t_1 和 t_2 中 x 的引用点都是“可用”的

② 删除复制语句

- 对于复制语句 $s: x=y$, 如果在 x 的所有引用点都可以用对 y 的引用代替对 x 的引用(复制传播), 那么可以删除复制语句 $x=y$
- 例

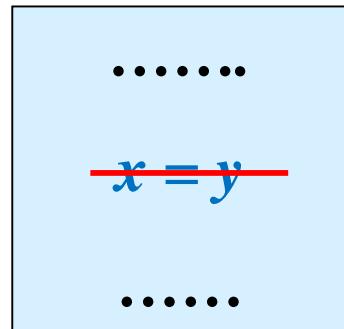


- $s: x=y$ 在 x 的引用点 u 可用 y 代替 x 的条件

➤ $s: x=y$ 在 u 点 “可用”

② 删除复制语句

$IN(X)$: 可用表达式



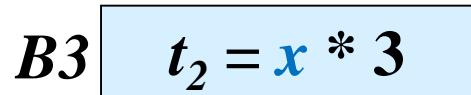
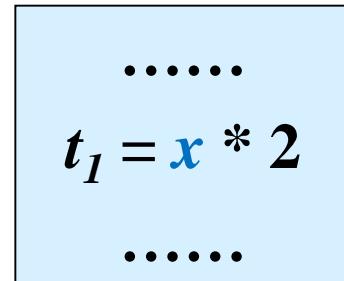
$B1$

$$du(x=y) = \{ t_1 = x * 2, t_2 = x * 3 \}$$

$$IN(B2) = \{ \dots, x=y, \dots \}$$

未对x或y定值

$B2$

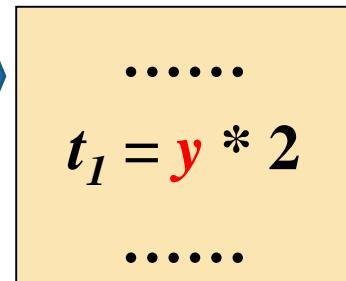


$$IN(B3) = \{ \dots, x=y, \dots \}$$

y 替换 x

$t_2 = \underline{y} * 3$

y 替换 x



➤ 删除复制语句的算法

- 输入：流图 G 、 du 链、各基本块 B 入口处的可用复制语句集合
- 输出：修改后的流图
- 方法：
 - 对于每个复制语句 $x=y$ ，执行下列步骤
 - ① 根据 du 链找出该定值所能够到达的那些对 x 的引用
 - ② 确定是否对于每个这样的引用， $x=y$ 都在 $IN[B]$ 中 (B 是包含这个引用的基本块)，并且 B 中该引用的前面没有 x 或者 y 的定值(即对于每个引用点， $x=y$ 都是可用的)
 - ③ 如果 $x=y$ 满足第②步的条件，删除 $x=y$ ，且把步骤①中找到的对 x 的引用用 y 代替



③ 代码移动

- 循环不变计算的检测
- 代码外提

循环不变计算检测算法

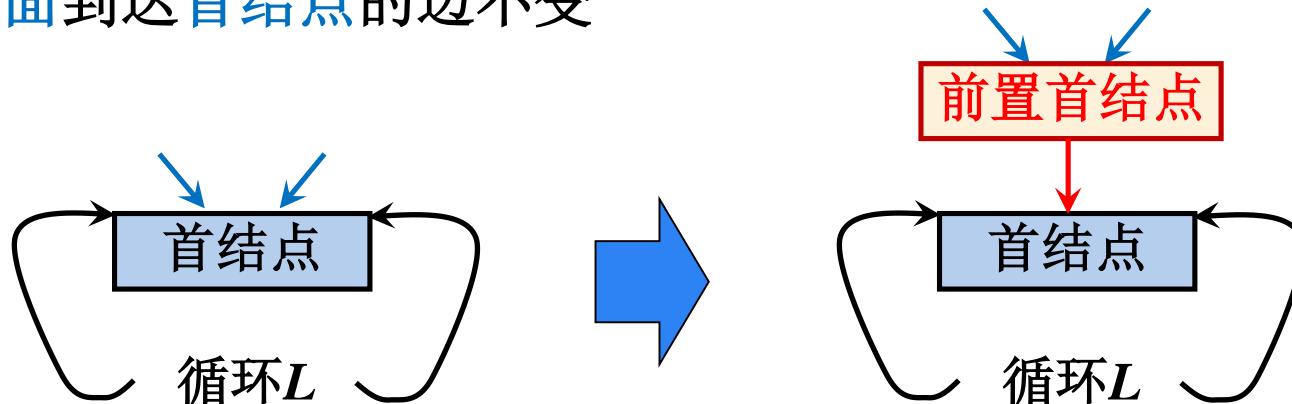
- 输入：循环 L ，每个三地址指令的ud链
- 输出： L 的循环不变计算语句
- 方法
 1. 将下面这样的语句标记为“不变”：语句的各运算分量或者是常数，或者其所有定值点都在循环 L 外部
 2. 重复执行步骤(3)，直到某次没有新的语句可标记为“不变”为止
 3. 将下面这样的语句标记为“不变”：先前没有被标记过，且各运算分量或者是常数，或者其所有定值点都在循环 L 外部，或者只有一个到达定值，该定值是循环中已经被标记为“不变”的语句

代码外提



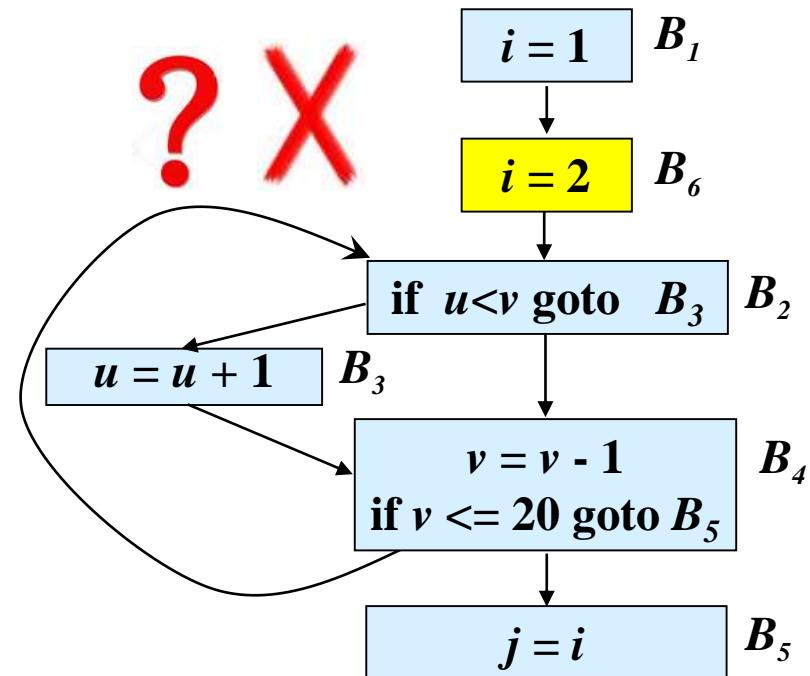
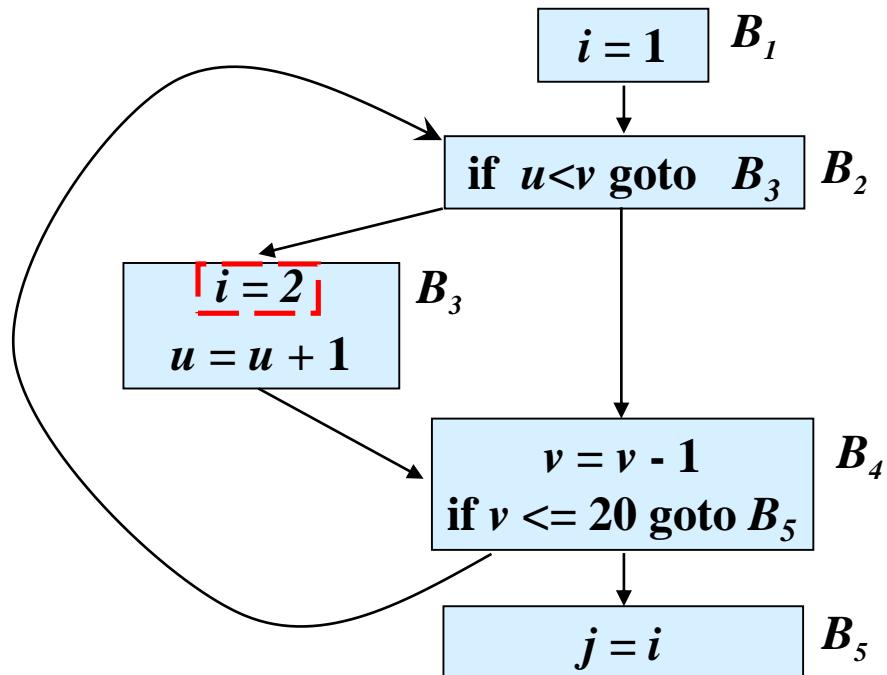
➤ 前置首结点 (*preheader*)

➤ 循环不变计算将被移至首结点之前，为此创建一个称为前置首结点的新块。前置首结点的唯一后继是首结点，并且原来从循环L外到达L首结点的边都改成进入前置首结点。从循环L里面到达首结点的边不变



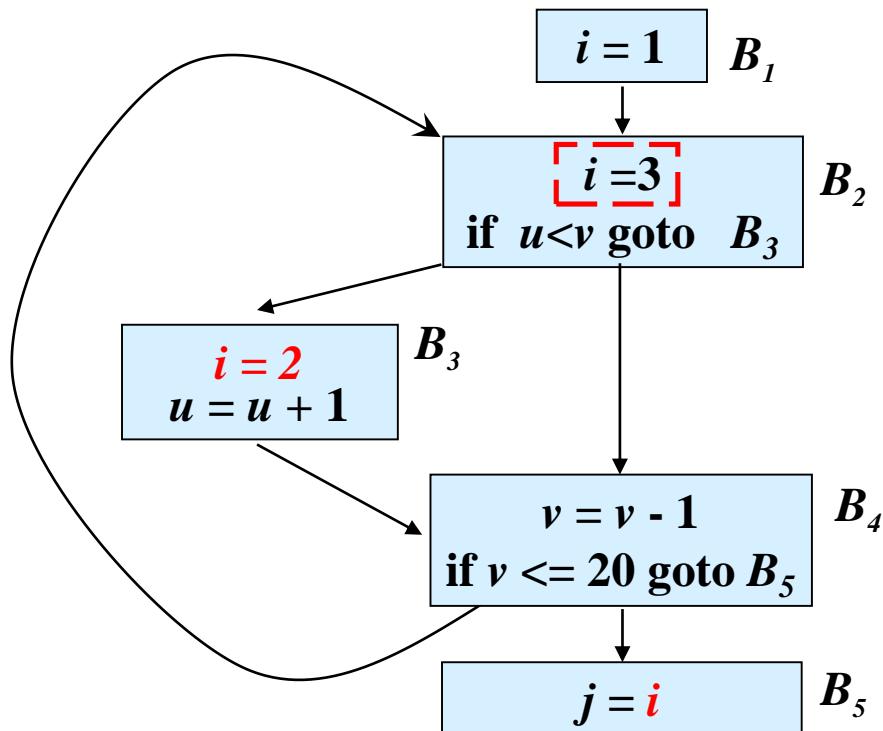
循环不变计算语句 $s : x = y + z$ 移动的条件

(1) s 所在的基本块是循环所有出口结点(有后继结点在循环外的结点)的支配结点



循环不变计算语句 $s : x = y + z$ 移动的条件

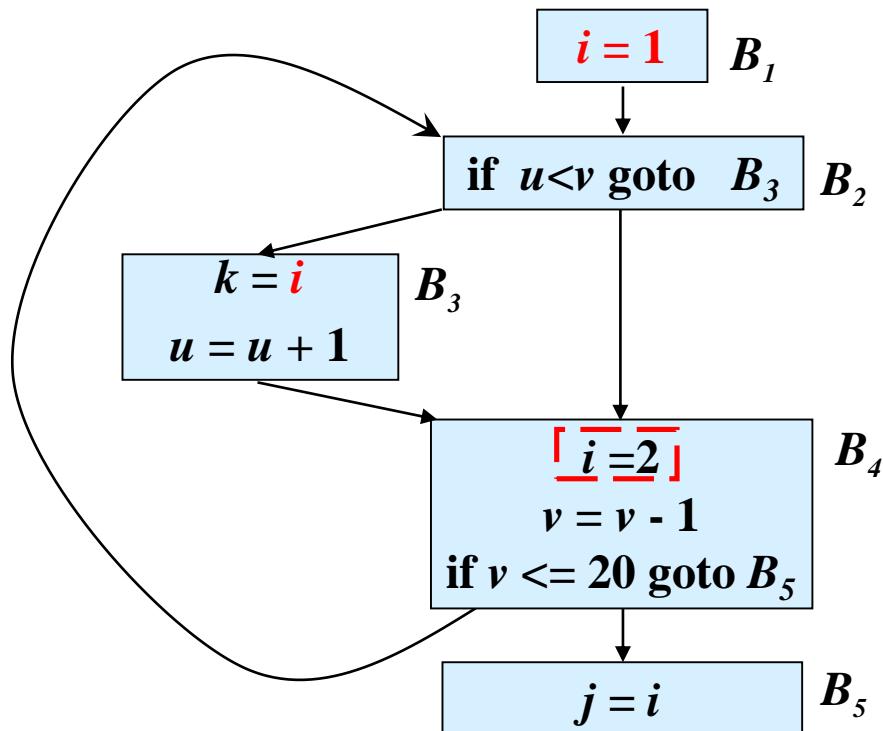
(2) 循环中没有其它语句对 x 赋值



- 如果循环中还存在 $i=2$
- 外提前
 - j 的值是否等于2取决于循环最后一次迭代时，是否执行了 B_3
- 外提后
 - 只要 B_3 执行过一次， j 的值就等于2

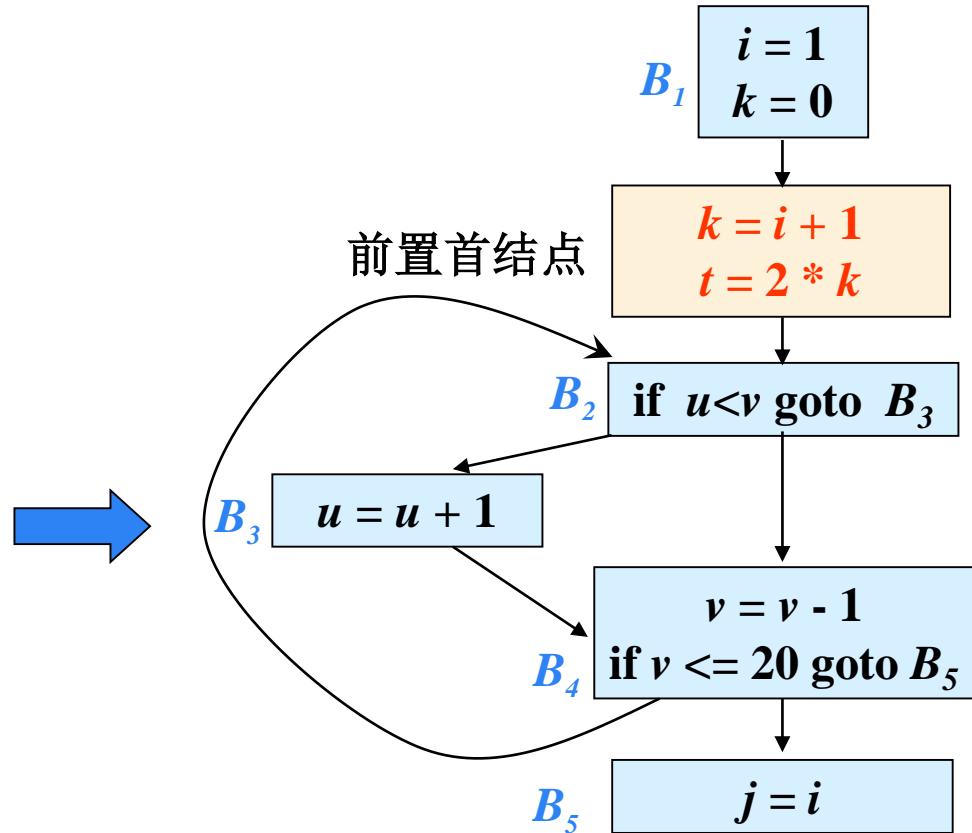
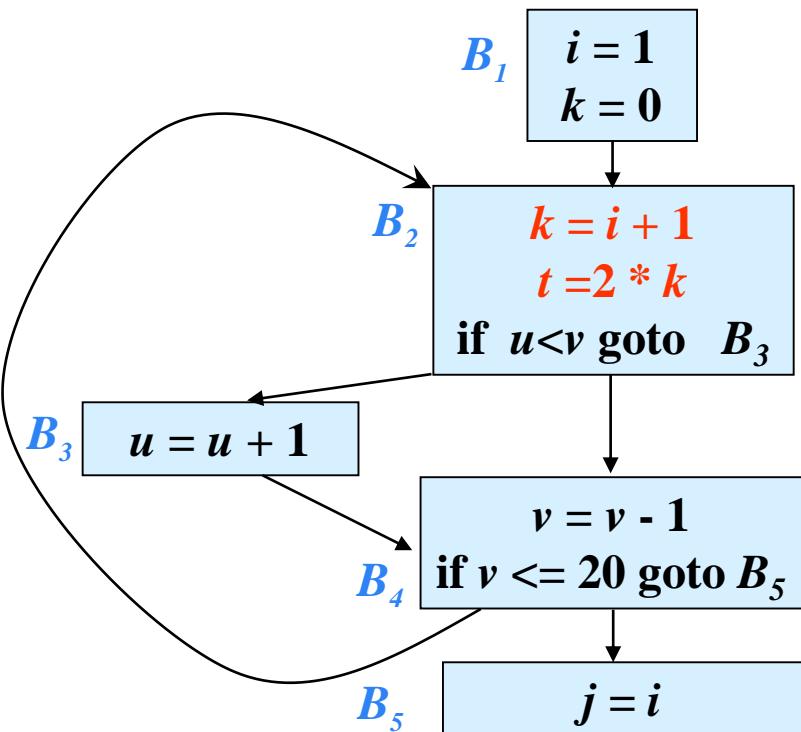
循环不变计算语句 $s : x = y + z$ 移动的条件

(3) 循环中对 x 的引用仅由 s 到达



- 如果还有其它定值能到达 $k = i$
- 外提前
 - k 的值有可能等于 1，也可能等于 2
- 外提后
 - k 的值只能等于 2

例



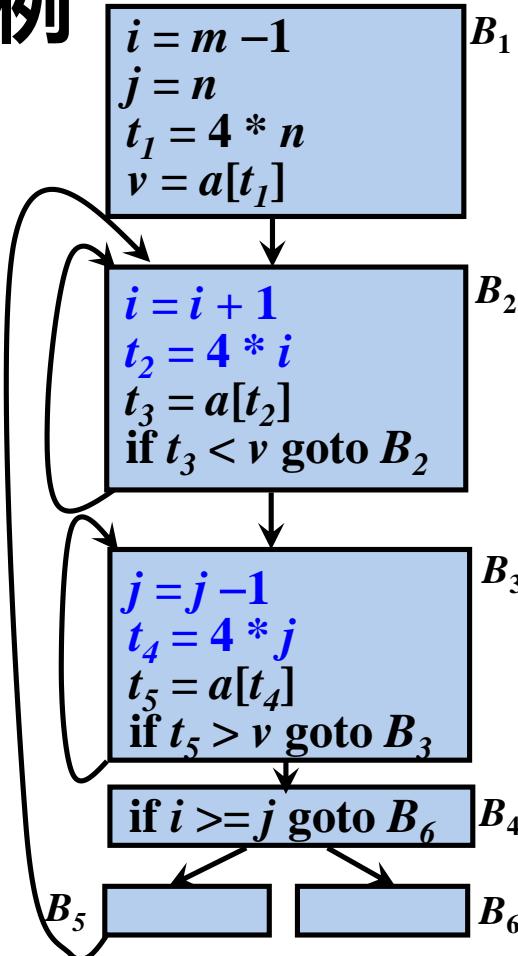
代码移动算法

- 输入：循环 L 、*ud*链、支配结点信息
- 输出：修改后的循环
- 方法：
 1. 寻找循环不变计算
 2. 对于步骤(1)中找到的每个循环不变计算，检查是否满足上面的三个条件
 3. 按照循环不变计算找出的次序，把所找到的满足上述条件的循环不变计算外提到前置首结点中。如果循环不变计算有分量在循环中定值，只有将定值点外提后，该循环不变计算才可以外提

④ 作用于归纳变量的强度削弱

- 如果循环 L 中的变量 i 只有形如 $i = i + n$ 的定值， n 是常量(或循环不变量)，则称 i 为循环 L 的基本归纳变量
- 如果循环 L 中的变量 j 只有形如 $j = c \times i + d$ 的定值，其中 i 是基本归纳变量， c 和 d 是常量(或循环不变量)，则 j 也是一个归纳变量，称 j 属于 i 族
 - 基本归纳变量 i 属于它自己的族
 - 每个归纳变量都关联一个三元组。如果 $j = c \times i + d$ ，其中 i 是基本归纳变量， c 和 d 是常量，则与 j 相关联的三元组是 (i, c, d)

例



- 本节主要研究的目的 是找到它与基本归纳变量之间的函数关系，以期使用代价低的运算代替代价高的运算
- 若 i 是基本归纳变量: $i = i + n$
 j 是 i 族 归纳变量, 表示为 (i, c, d)
那么有, i 增加 n , j 增加 $c * n$

i 是 基本归纳变量, $i = i + 1$
 t_2 是 i 族 归纳变量, 可以表示为 $(i, 4, 0)$
 i 增加 1, t_2 增加 4

j 是 基本归纳变量, $j = j - 1$
 t_4 是 j 族 归纳变量, 可以表示为 $(j, 4, 0)$
 j 增加 -1, t_4 增加 -4

归纳变量检测算法

- 输入： 带有循环不变计算信息和到达定值信息的循环 L
- 输出： 一组归纳变量
- 方法：
 1. 扫描 L 的语句，找出所有基本归纳变量。在此要用到循环不变计算信息。与每个基本归纳变量 i 相关联的三元组是($i, 1, 0$)

归纳变量检测算法 (续)

2: 寻找 L 中只有一次定值的变量 k , 它具有下面的形式: $k=c' \times j + d'$ 。其中 c' 和 d' 是常量(或循环不变计算), j 是基本的或非基本的归纳变量

- 如果 j 是基本归纳变量, 那么 k 属于 j 族。 k 对应的三元组可以通过其定值语句确定, 即 (j, c', d')
- 如果 j 不是基本归纳变量, 假设其属于 i 族, 那么 k 也属于 i 族, 且 k 的三元组可以通过 j 的三元组 (i, c, d) 和 k 的定值语句 $(k=c' \times j + d')$ 来计算, 即 k 的三元组为 $(i, c' \times c, c' \times d + d')$, 此时我们还要求:
 - 循环 L 中对 j 的唯一定值和对 k 的定值之间没有对 i 的定值
 - 循环 L 外没有 j 的定值可以到达 k

这两个条件是为了保证对 k 与 i 的函数关系是符合同步调变化(线性)的。

$$j=c \times i + d$$



$$k=c' \times j + d'$$

例

强度削弱

$i = m - 1$
 $j = n$
 $t_1 = 4 * n$
 $v = a[t_1]$

B_1

新归纳变量初始化
(根据三元组)
 $s=c*i$
 $s=s+d$

$i = i + 1$
 $t_2 = 4 * i$
 $t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_2

i 增加 $n(1)$,
 t_2 增加 $c(4)*n(1)$

$j = j - 1$
 $t_4 = 4 * j$
 $t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_3

$t_2: (i, 4, 0)$

$t_4: (j, 4, 0)$
 j 增加 $n(-1)$,
 t_4 增加 $c(4)*n(-1)$

if $i \geq j$ goto B_6

B_4

B_5

B_6

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

$s_2 = 4 * i$

$s_4 = 4 * j$

B_1

$i = i + 1$

$s_2 = s_2 + 4$

$t_2 = s_2$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_2

$j = j - 1$

$s_4 = s_4 - 4$

$t_4 = s_4$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_3

if $i \geq j$ goto B_6

B_4

B_5

B_6

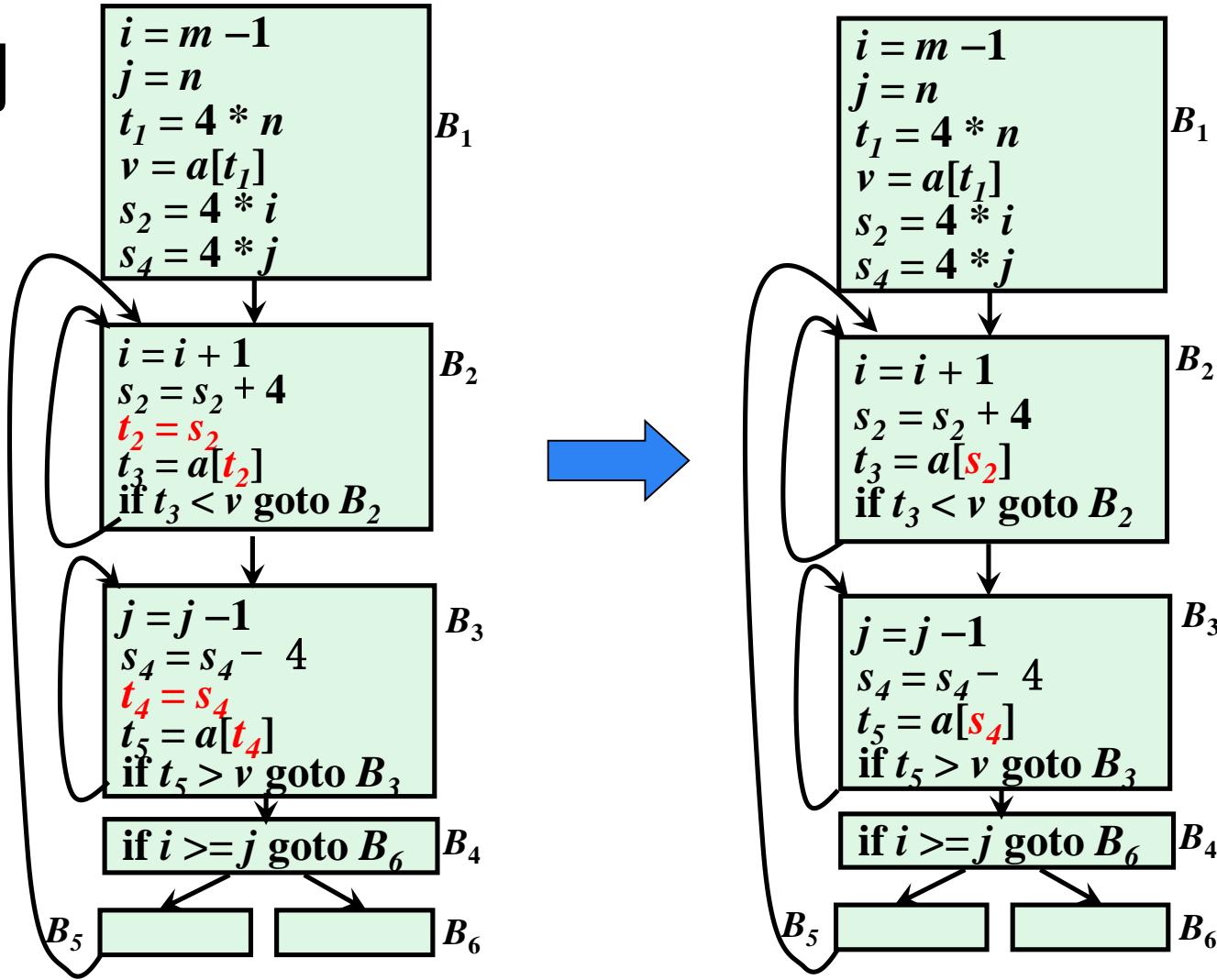
作用于归纳变量的强度削弱算法

- 输入：带有到达定值信息和已计算出的归纳变量族的循环 L
- 输出：修改后的循环
- 方法：对于每个基本归纳变量 i ，对其族中的每个归纳变量 j : (i, c, d) 执行下列步骤
 1. 建立新的临时变量 t 。如果变量 j_1 和 j_2 具有相同的三元组，则只为它们建立一个新变量
 2. 在前置结点的末尾，添加语句 $t = c*i$ 和 $t = t+d$ ，使得在循环开始的时候
$$t = c*i + d$$
 3. 在 L 中紧跟定值 $i = i+n$ 之后，添加 $t = t+c*n$ 。将 t 放入 i 族，其三元组为 (i, c, d)
 4. 用 $j=t$ 代替对 j 的赋值

⑤ 归纳变量的删除

- 对于在强度削弱算法中引入的复制语句 $j=t$ ，如果在归纳变量 j 的所有引用点都可以用对 t 的引用代替对 j 的引用，并且 j 在循环的出口处不活跃，则可以删除复制语句 $j=t$

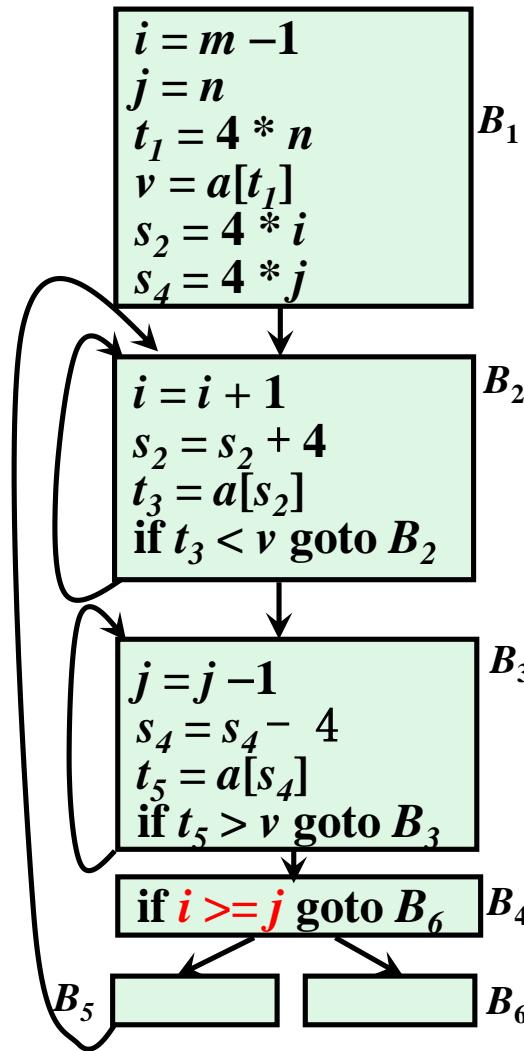
例



归纳变量的删除

- 对于在强度削弱算法中引入的复制语句 $j=t$ ，如果在归纳变量 j 的所有引用点都可以用对 t 的引用代替对 j 的引用，并且 j 在循环的出口处不活跃，则可以删除复制语句 $j=t$
- 强度削弱后，有些归纳变量的作用只是用于测试。如果可以用对其它归纳变量的测试代替对这种归纳变量的测试，那么可以删除这种归纳变量

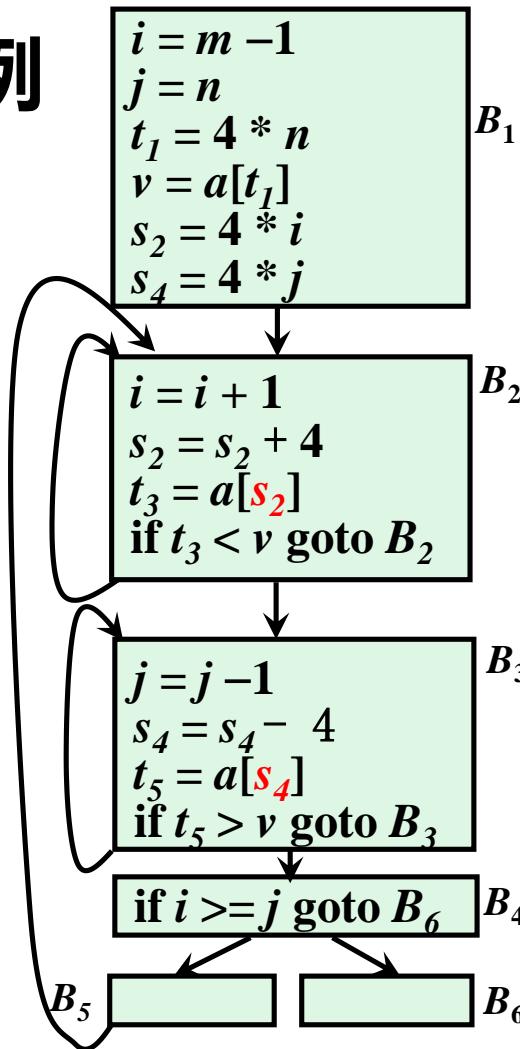
例



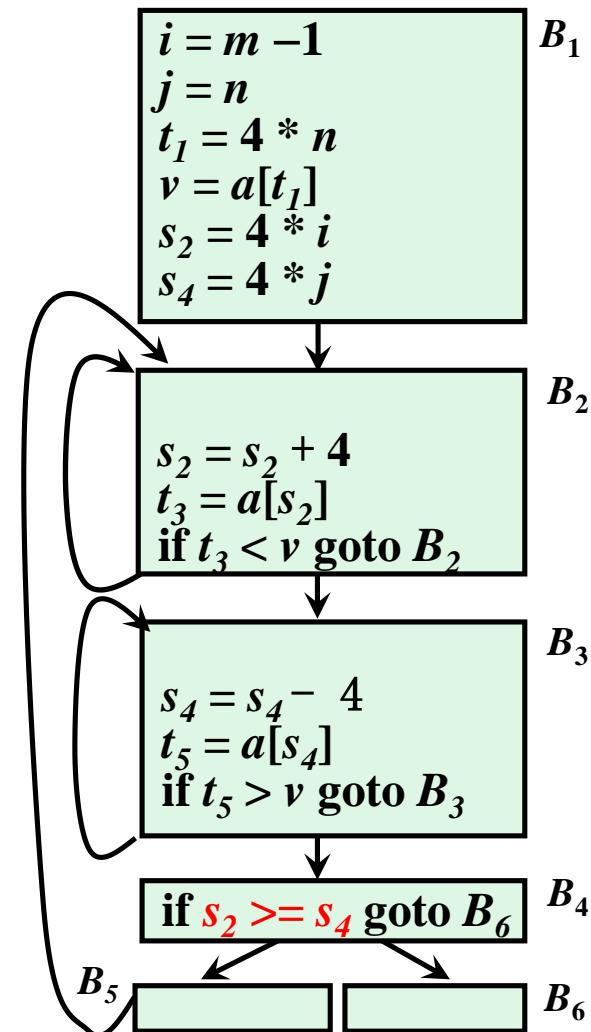
➤ 删除仅用于测试的归纳变量

- 对于仅用于测试的基本归纳变量 i , 取 i 族的某个归纳变量 j , 设其三元组为 (i, c, d) (尽量选择使得 c 、 d 简单, 即 $c=1$ 或 $d=0$ 的情况)。把每个对 i 的测试替换成对 j 的测试
- $(\text{relop } i \ x \ B)$ 替换为 $(\text{relop } j \ c*x+d \ B)$, 其中 x 不是归纳变量, 并假设 $c>0$
- $(\text{relop } i_1 \ i_2 \ B)$, 如果能够找到三元组 $j_1 (i_1, c, d)$ 和 $j_2 (i_2, c, d)$, 那么可以将其替换为 $(\text{relop } j_1 \ j_2 \ B)$ (假设 $c>0$)。否则, 测试的替换可能是没有价值的
- 如果归纳变量 i 不再被引用, 那么可以删除和它相关的指令

例



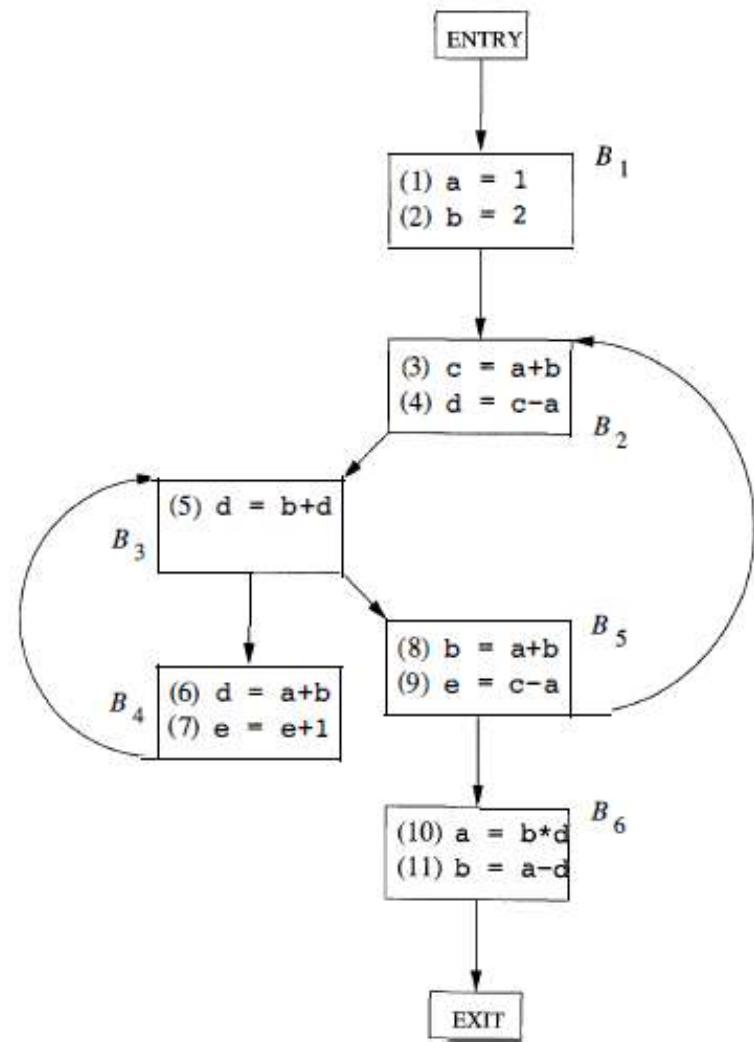
$$\begin{aligned} s_2: & (i, 4, 0) \\ s_4: & (j, 4, 0) \end{aligned}$$



练习3

➤ 练习3. 对于下图中的流图：

- (1) 找出流图中的自然循环。
- (2) B_1 中的语句 (1) 和 (2) 都是复制语句且 a 和 b 都被赋予了常量值。我们可以对 a 和 b 的哪些引用进行复制传播，并把对它们的引用替换为对一个常量的引用？在所有可能的地方进行这种替换。
- (3) 找出所有的全局公共子表达式。





本章小结

- 流图
- 优化的分类
- 基本块的优化
- 数据流分析
 - 到达-定值分析
 - 活跃变量分析
 - 可用表达式分析
- 流图中的循环
- 全局优化



课程主要内容

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



结束

