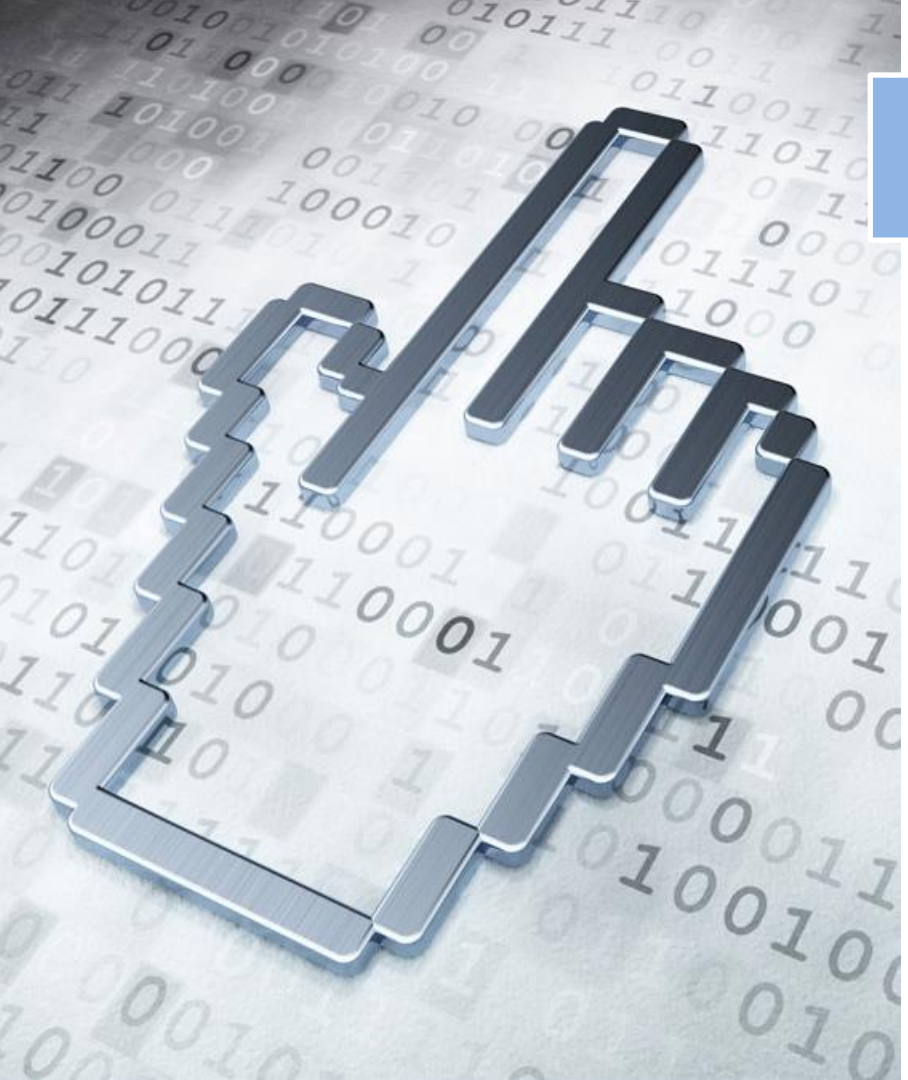
编译原理

第八章
代码优化

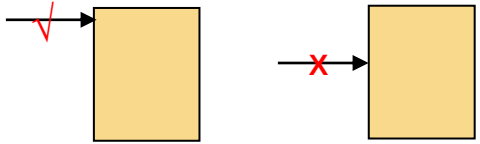
哈尔滨工业大学 陈鄞



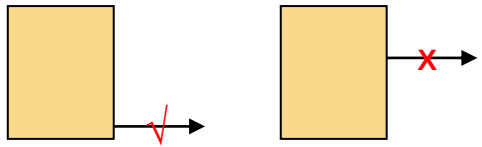
提纲

8.1 流图

- 基本块(*Basic Block*)是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条指令，控制流在离开基本块之前不会跳转或者停机

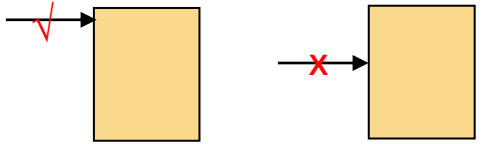


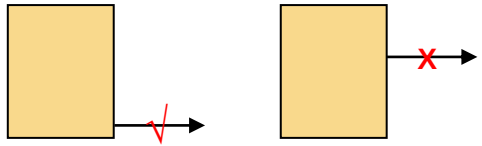
每个基本块由一组总是一起执行的指令组成

8.1 流图

- 基本块(*Basic Block*)是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条指令，控制流在离开基本块之前不会跳转或者停机

一个转移指令的目标指令





紧跟转移指令之后的指令

如何划分基本块？

是一起执行的指令组成

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 >= j) break;
    x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

B_1
(1) $i = m - 1$
(2) $j = n$
(3) $t_1 = 4 * n$
(4) $v = a[t_1]$

B_2
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) if $t_3 < v$ goto(5)

B_3
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) if $t_5 > v$ goto(9)

B_4
(13) if $i >= j$ goto(23)
(14) $t_6 = 4 * i$
(15) $x = a[t_6]$

B_5
(16) $t_7 = 4 * i$
(17) $t_8 = 4 * j$
(18) $t_9 = a[t_8]$
(19) $a[t_7] = t_9$
(20) $t_{10} = 4 * j$
(21) $a[t_{10}] = x$
(22) goto (5)

B_6
(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) $a[t_{15}] = x$

流图(*Flow Graphs*)

- 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*的第一条指令可能紧跟在*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*的第一条指令可能紧跟在*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：
    - 存在一条从*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

B_1
(1) $i = m - 1$
(2) $j = n$
(3) $t_1 = 4 * n$
(4) $v = a[t_1]$

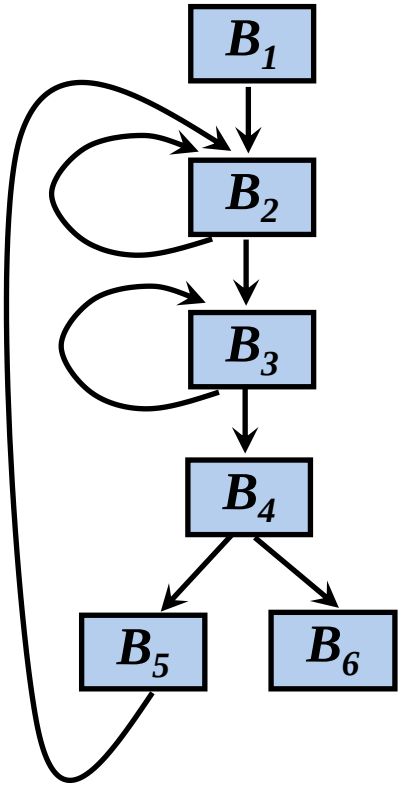
B_2
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) *if* $t_3 < v$ *goto*(5)

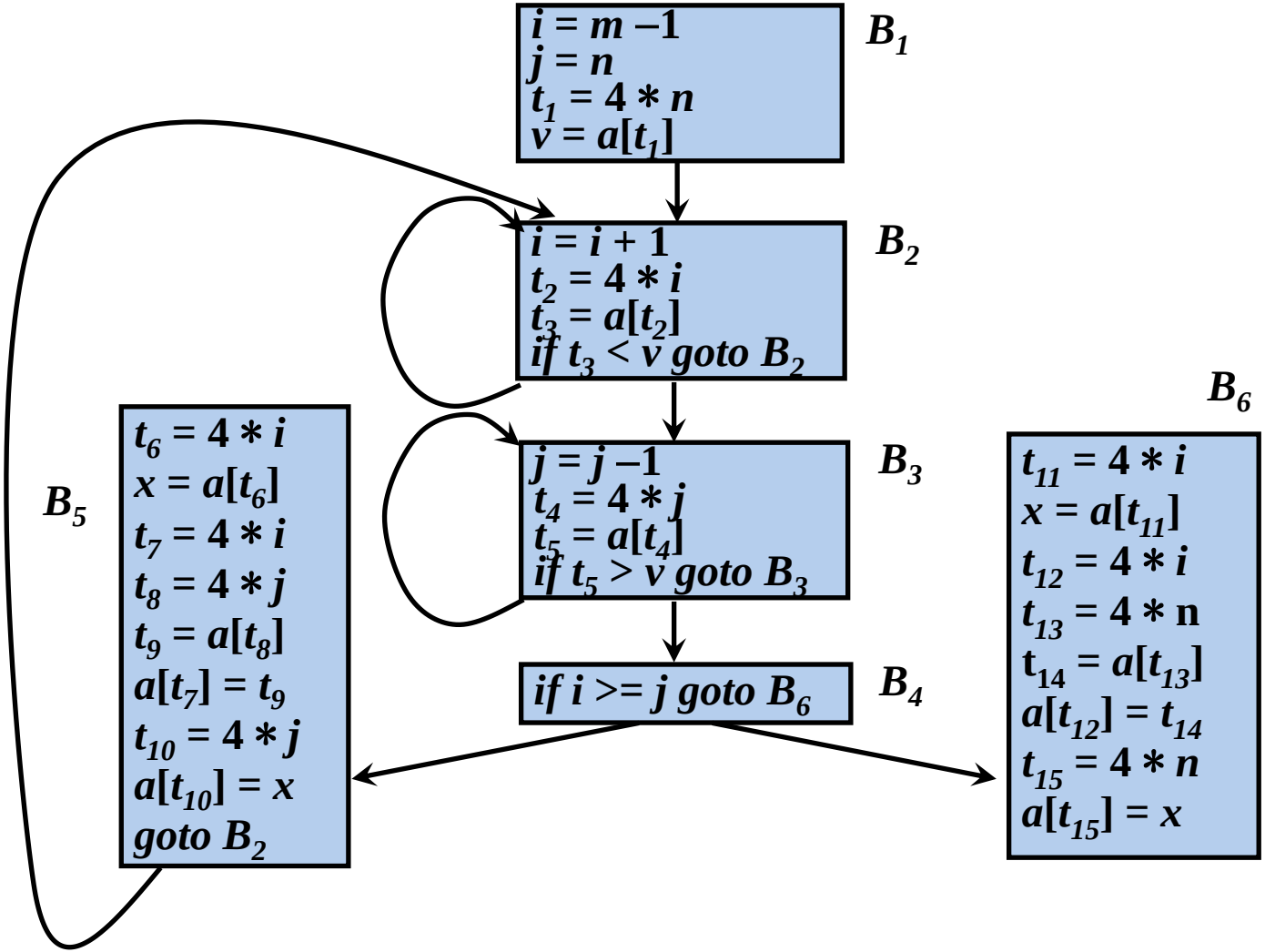
B_3
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) *if* $t_5 > v$ *goto*(9)

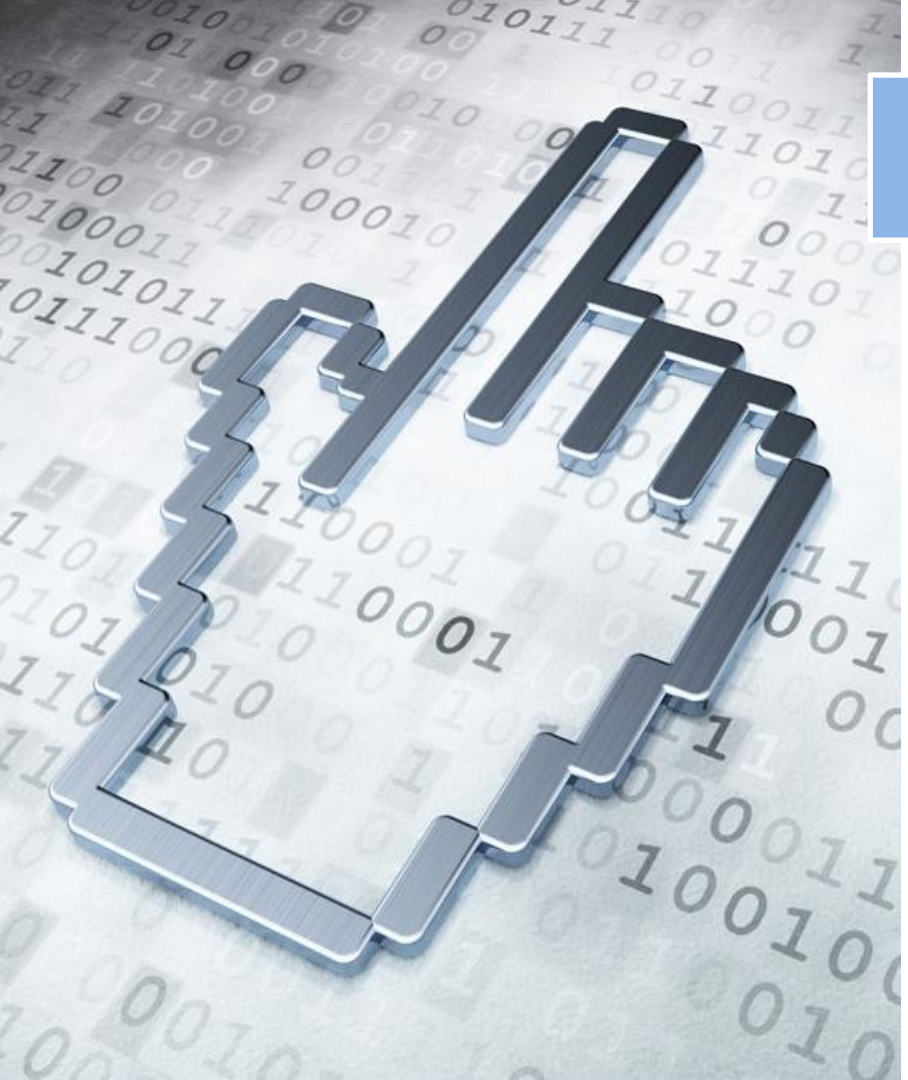
B_4
(13) *if* $i >= j$ *goto*(23)
(14) $t_6 = 4 * i$
(15) $x = a[t_6]$

B_5
(16) $t_7 = 4 * i$
(17) $t_8 = 4 * j$
(18) $t_9 = a[t_8]$
(19) $a[t_7] = t_9$
(20) $t_{10} = 4 * j$
(21) $a[t_{10}] = x$
(22) *goto* (5)

B_6
(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) $a[t_{15}] = x$




i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
B5
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x



提纲

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

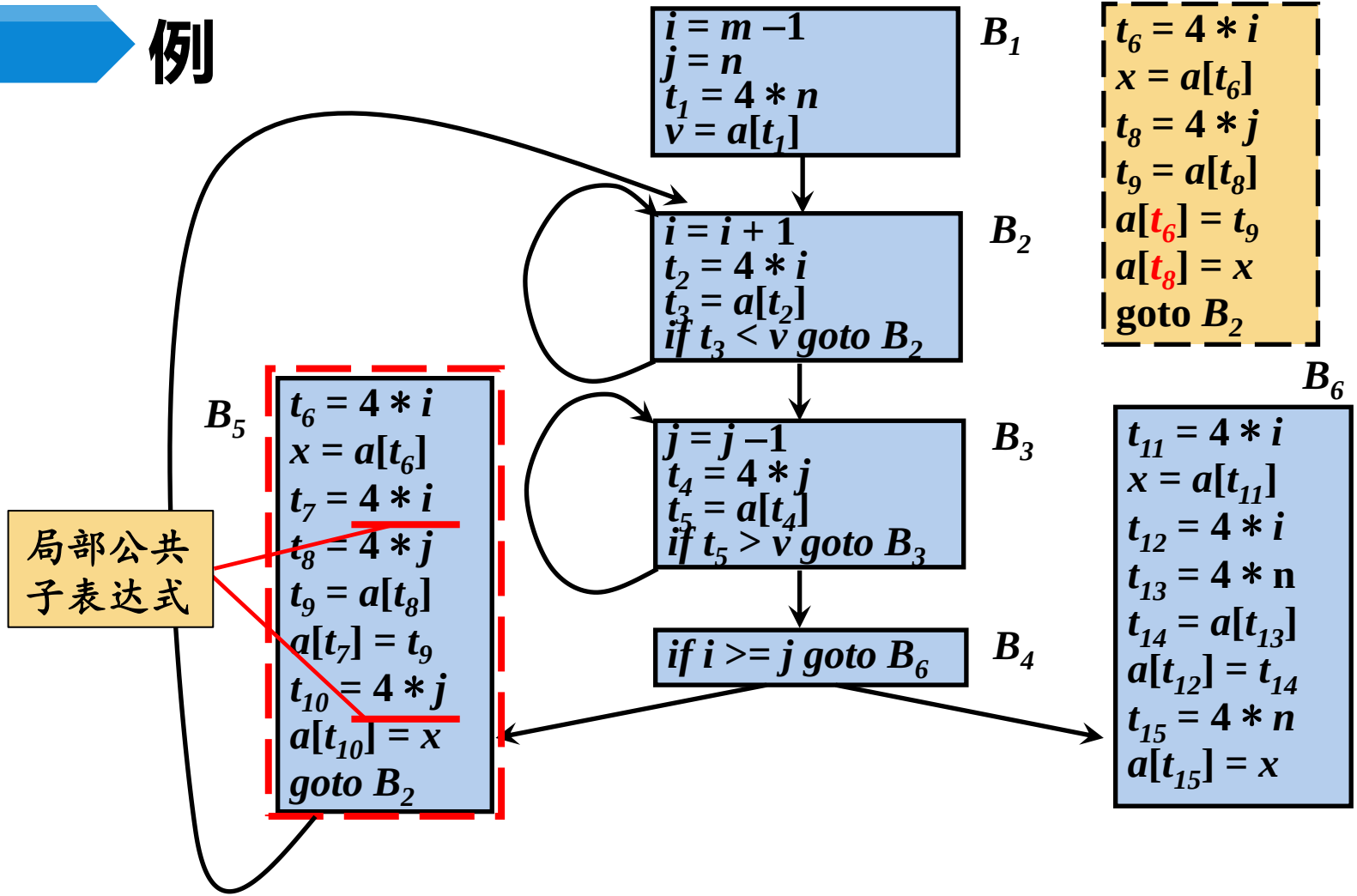
常用的优化方法

- 删除公共子表达式
- 删除无用代码
- 代码移动
- 强度削弱
- 删除归纳变量

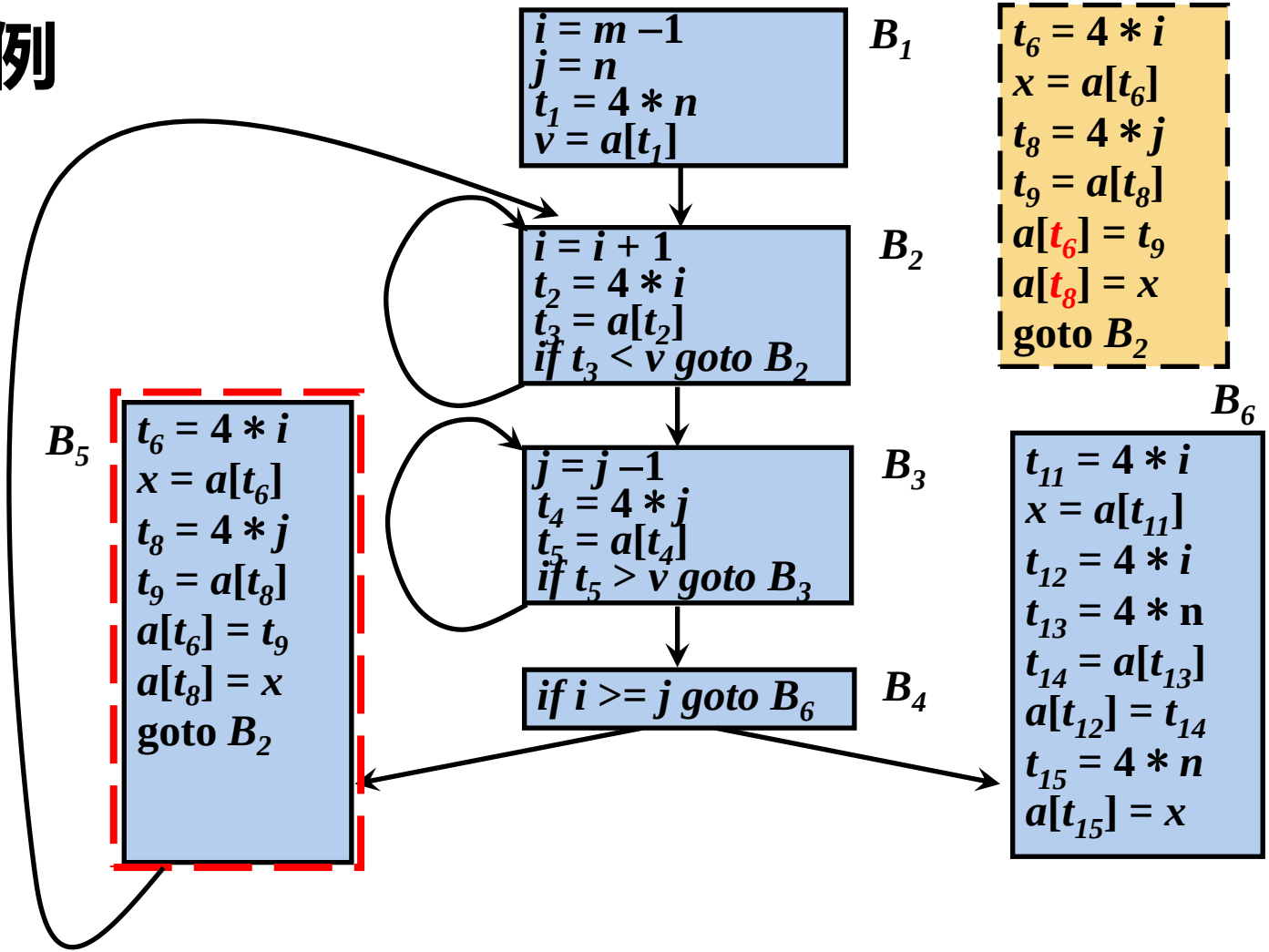
① 删除公共子表达式

- 公共子表达式
  - 如果表达式x op y先前已被计算过，并且从先前的计算到现在，x op y中变量的值没有改变，那么x op y的这次出现就称为公共子表达式(*common subexpression*)

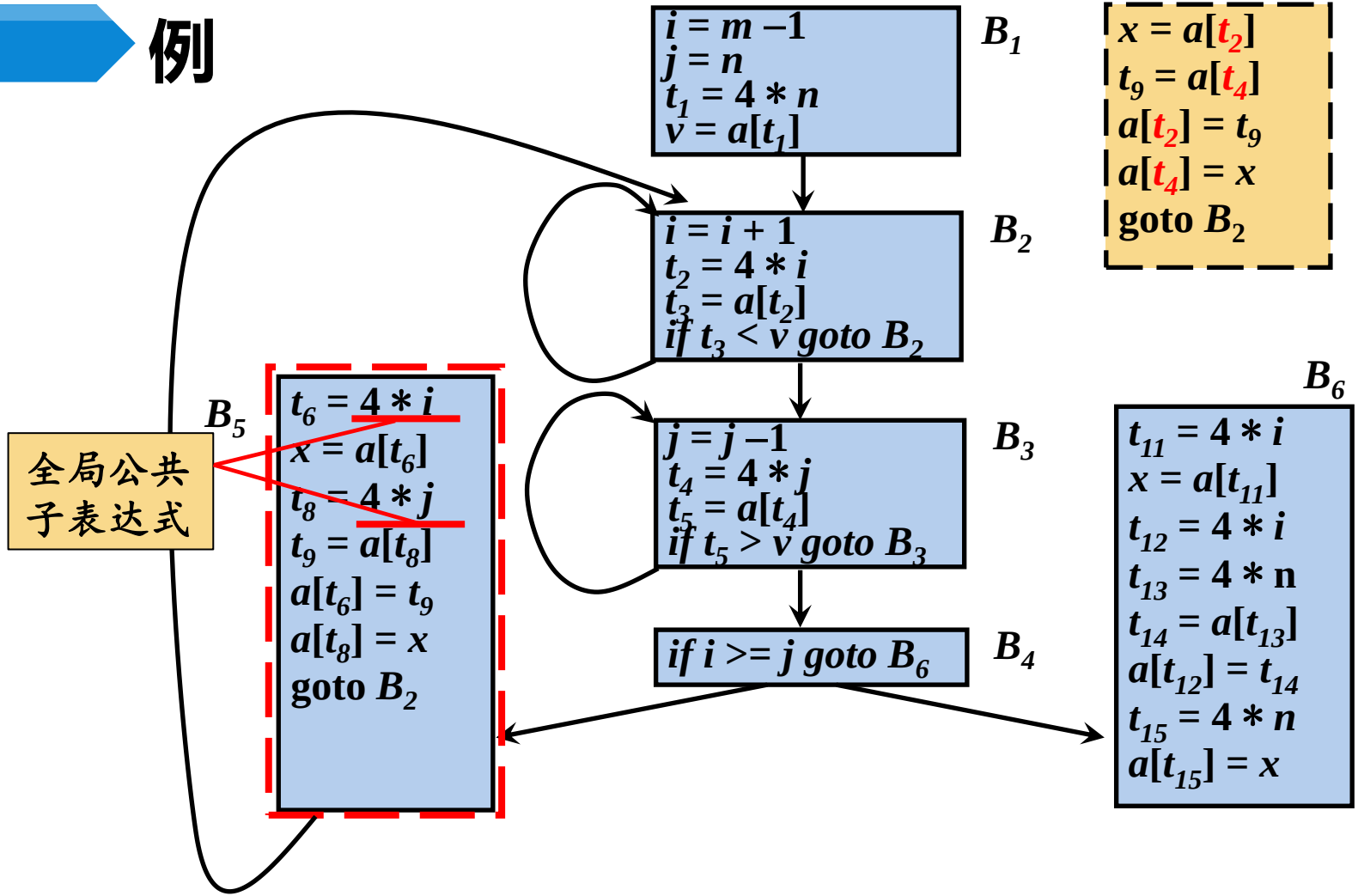
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
局部公共
子表达式
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

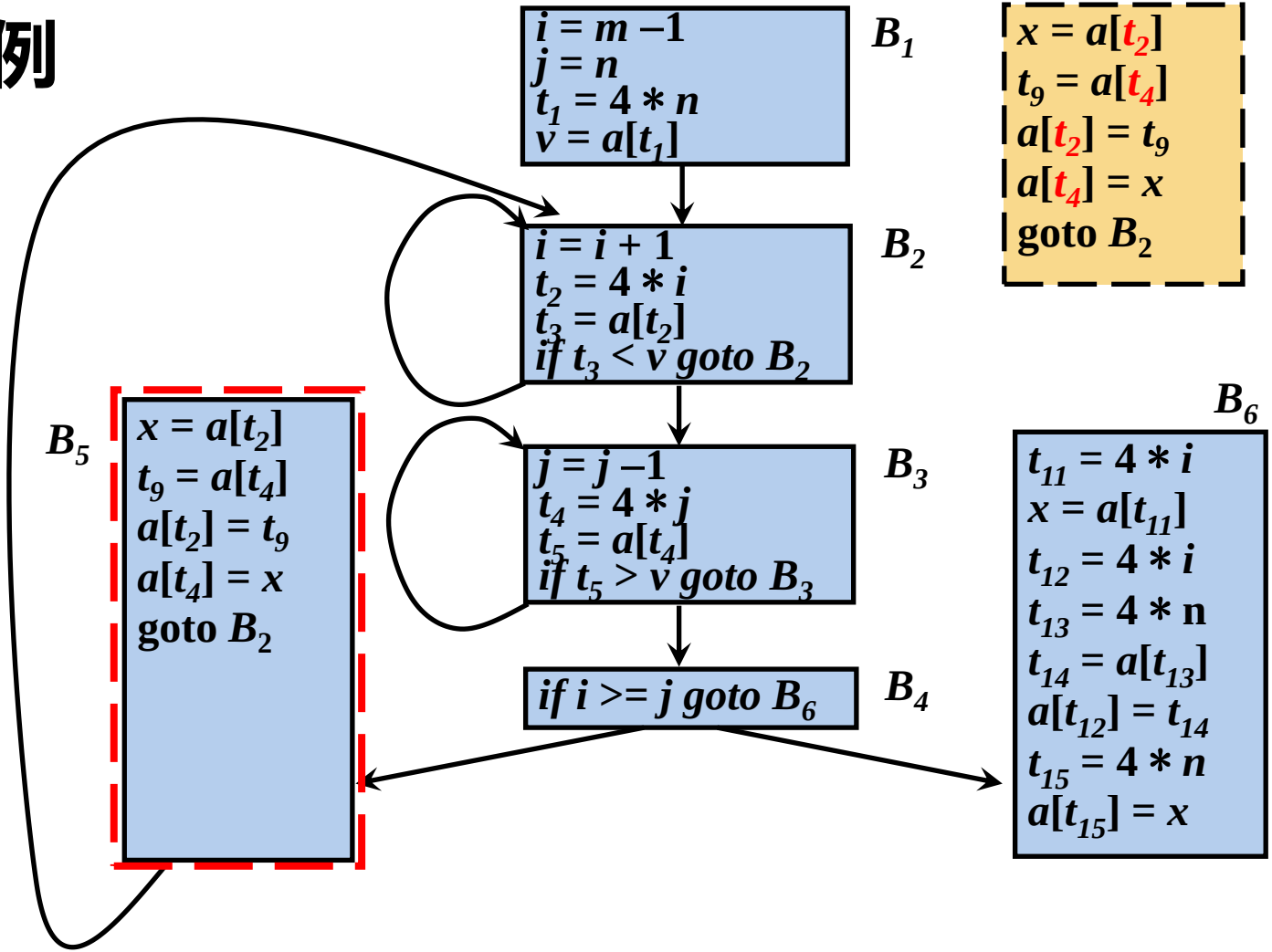
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
B5
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

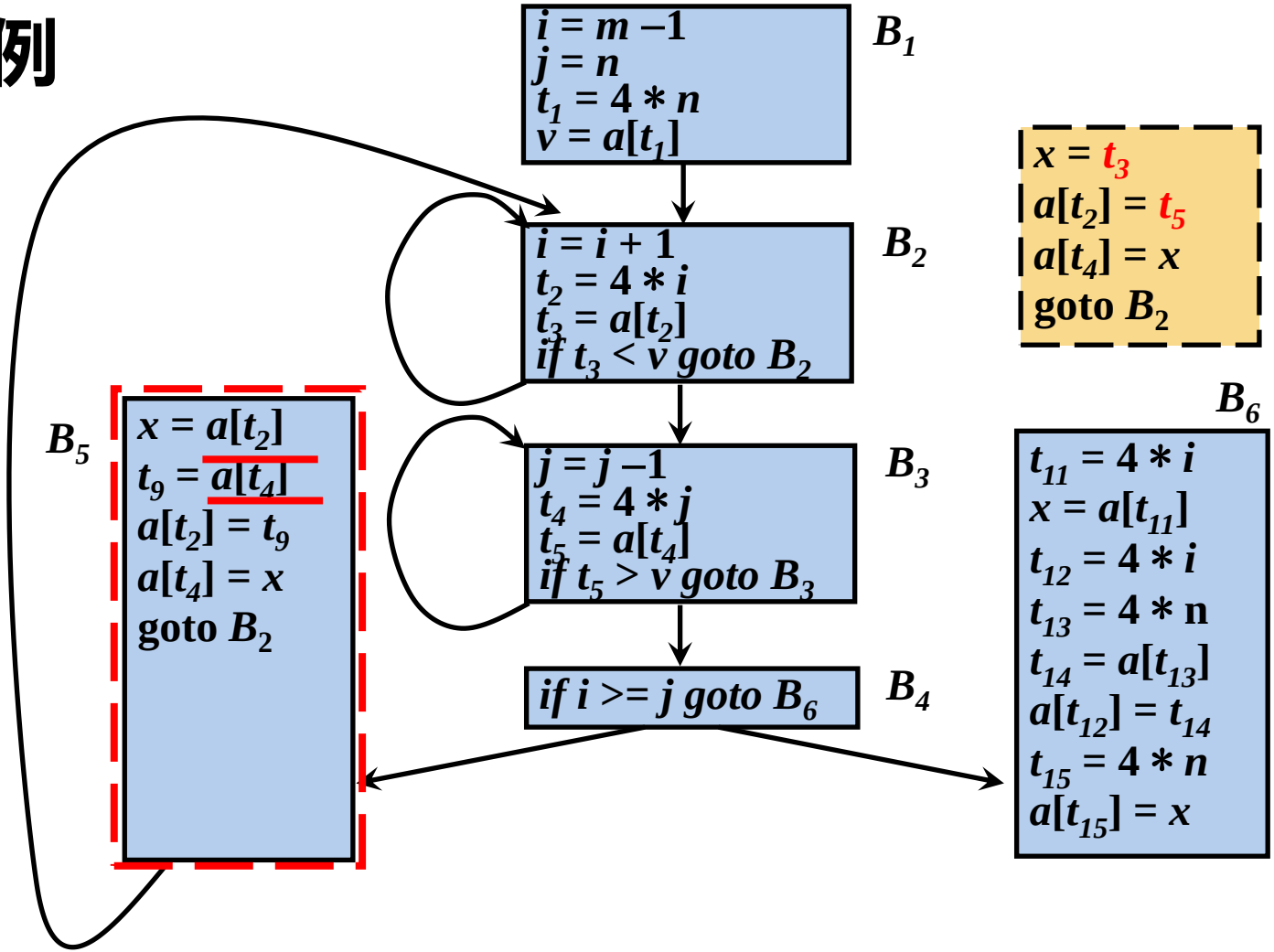
例


$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
B_1
$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2
B_2
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3
B_3
if $i >= j$ goto B_6
B_4
$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2
B_5
全局公共子表达式
$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2
B_6
$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

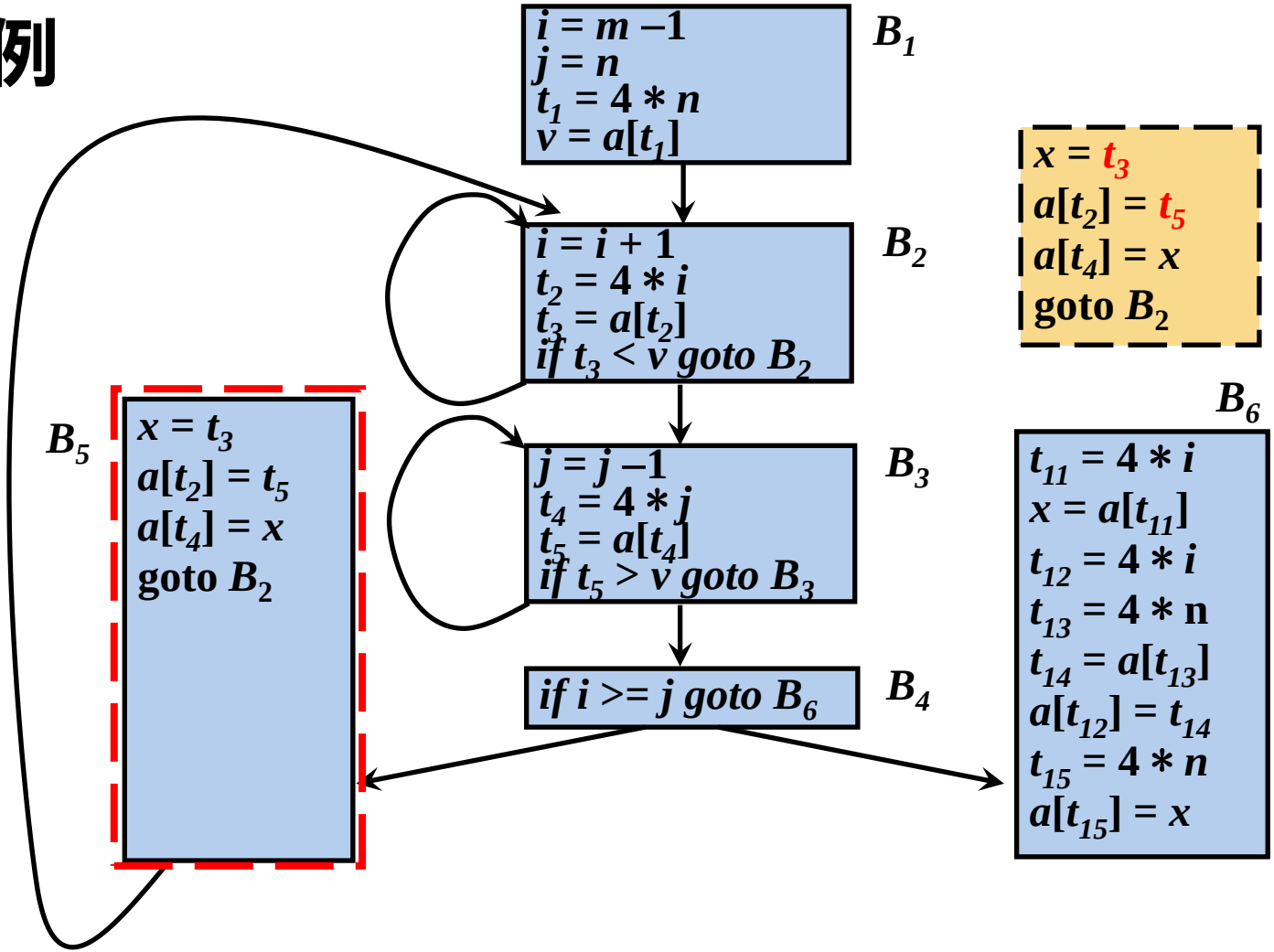
例


$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
B_1
$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2
B_2
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3
B_3
if $i >= j$ goto B_6
B_4
$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2
B_5
$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$
B_6
$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

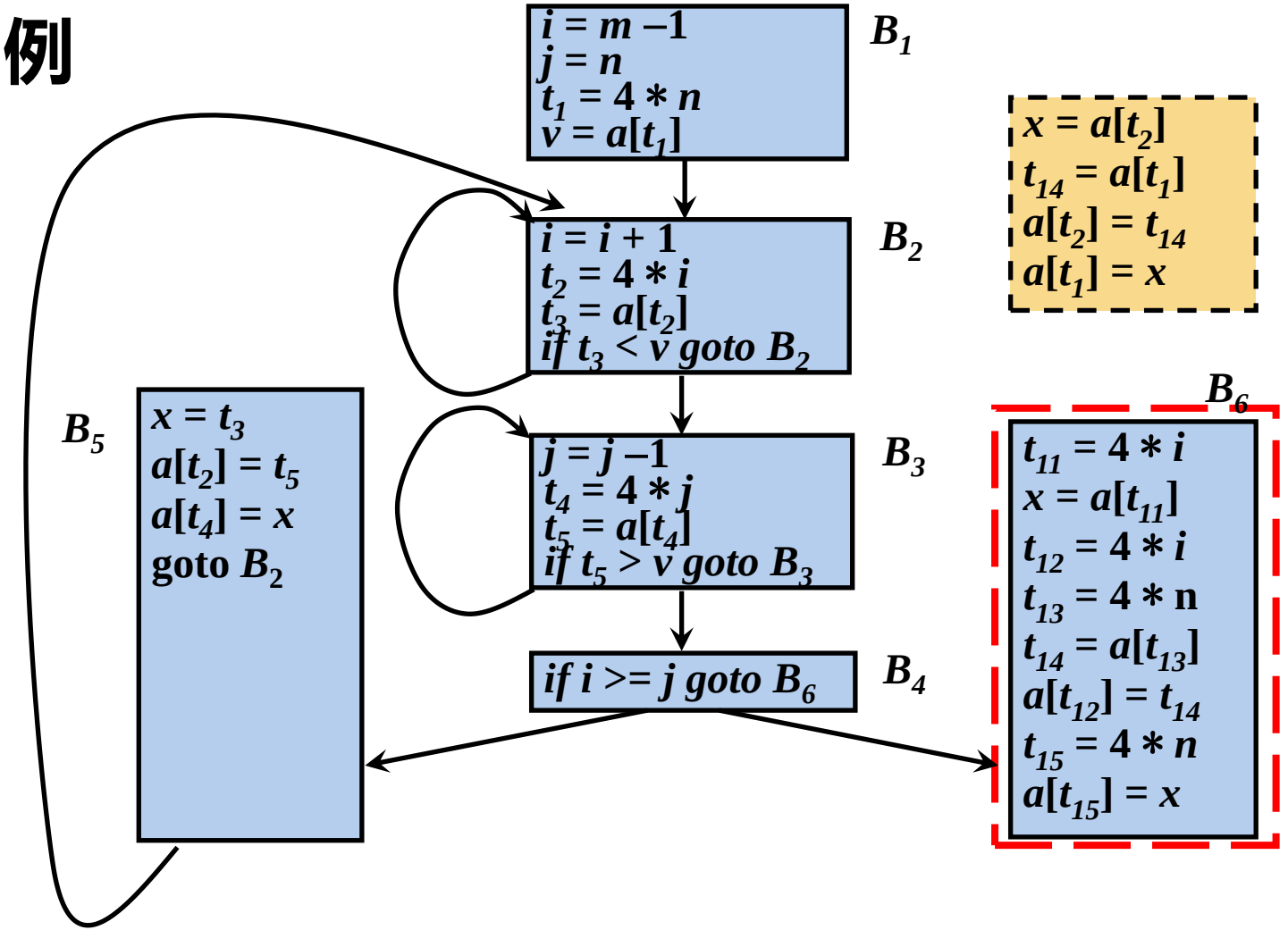
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2
B5
x = t3
a[t2] = t5
a[t4] = x
goto B2
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

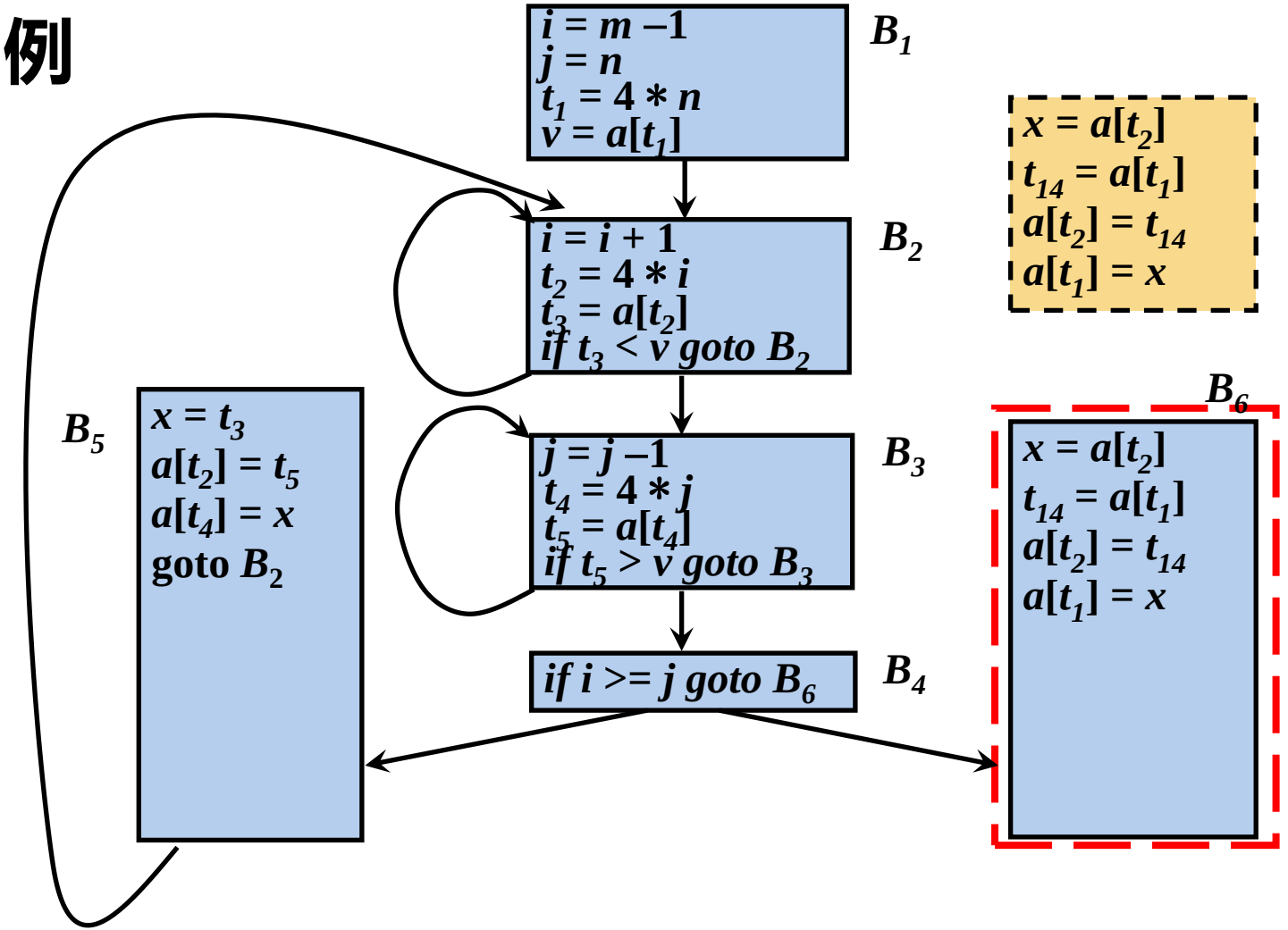
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
a[t2] = t5
a[t4] = x
goto B2
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

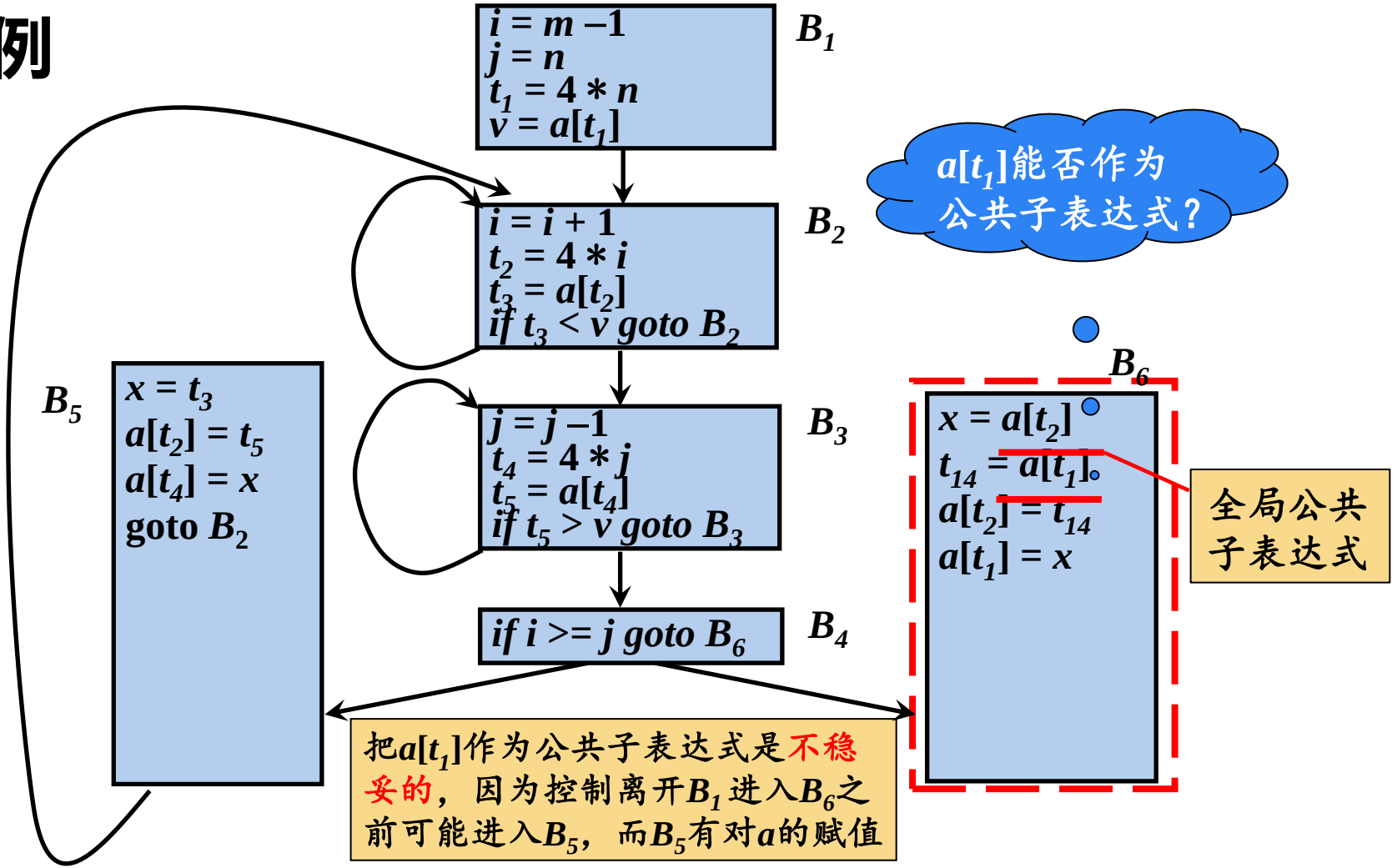
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

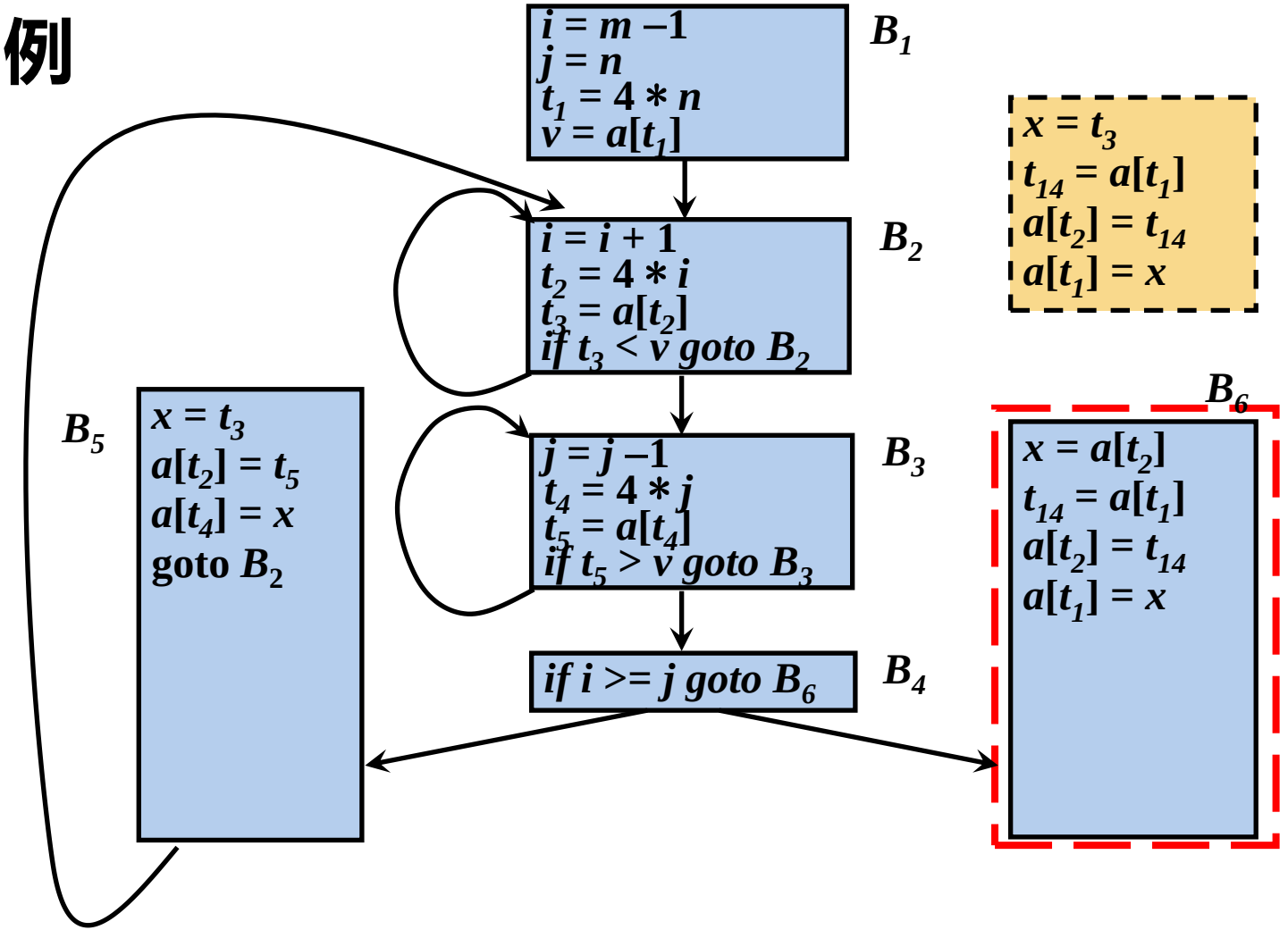
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x

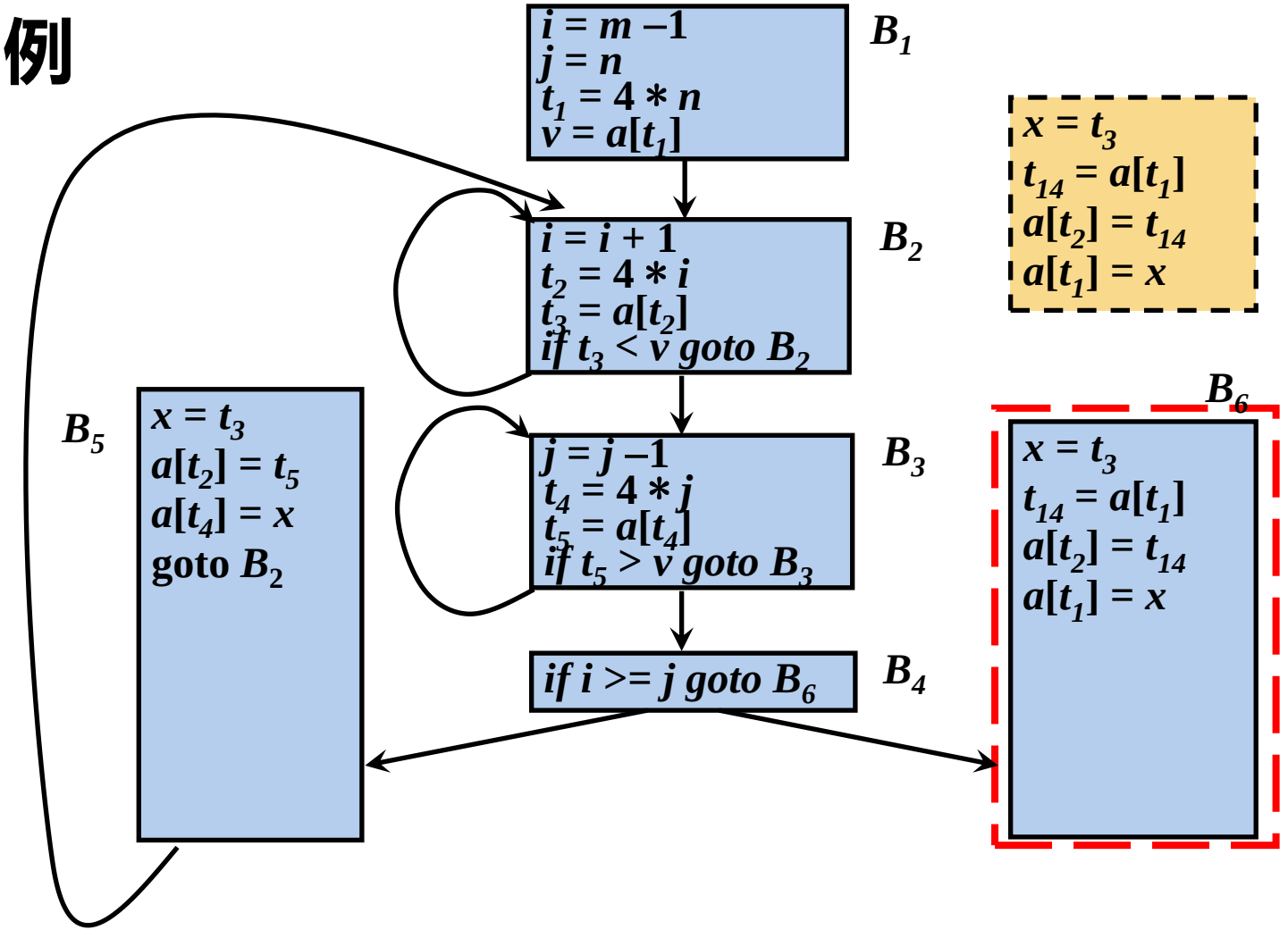
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
B6
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
a[t1]能否作为公共子表达式?
全局公共子表达式
把a[t1]作为公共子表达式是不稳妥的，因为控制离开B1进入B6之前可能进入B5，而B5有对a的赋值

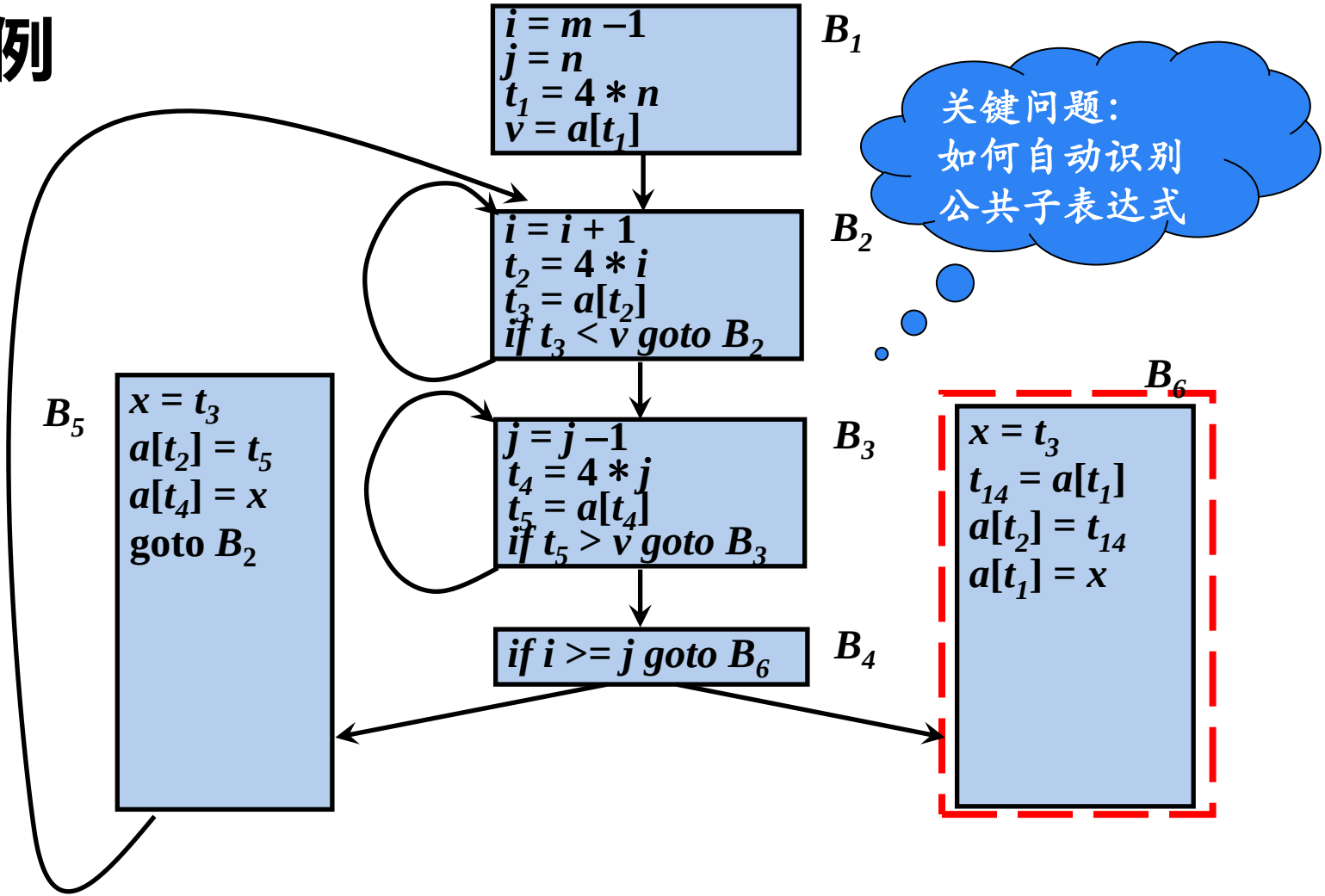
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x

例

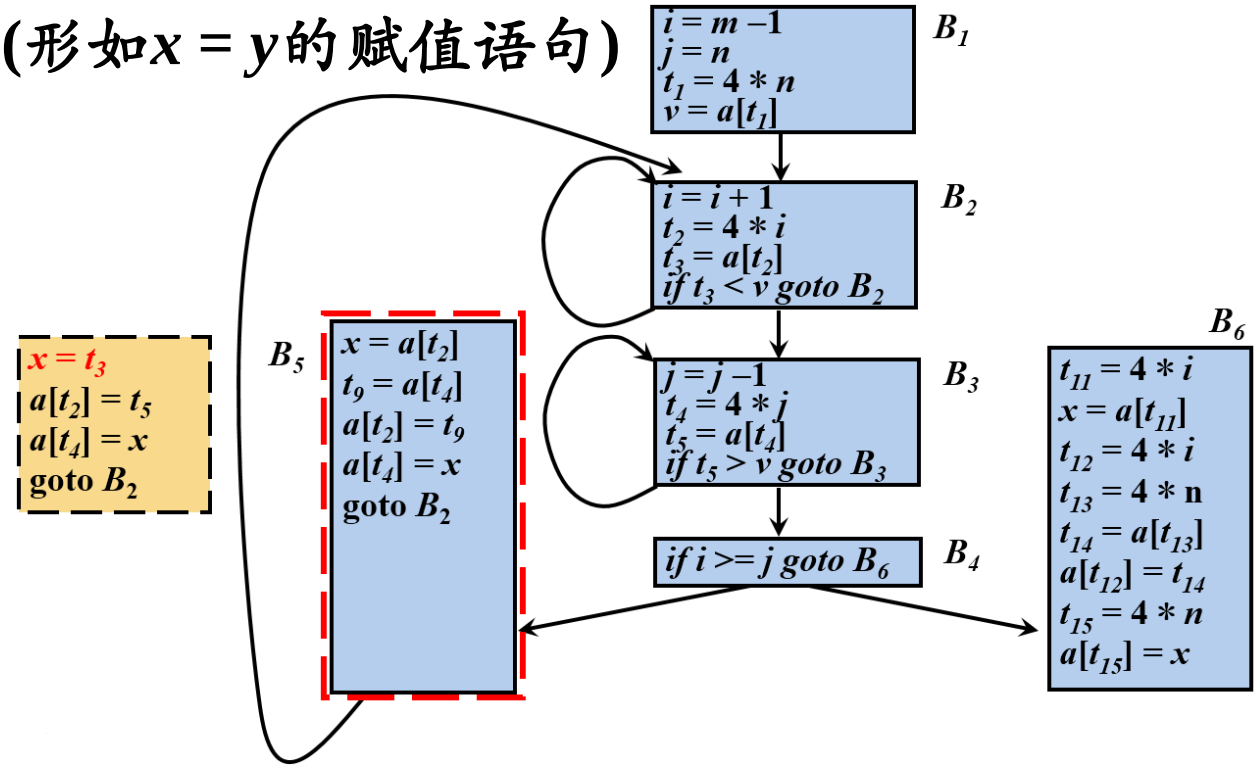

i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x

例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
关键问题：
如何自动识别
公共子表达式

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)

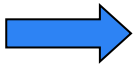


② 删除无用代码

- 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

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2



$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

B_6

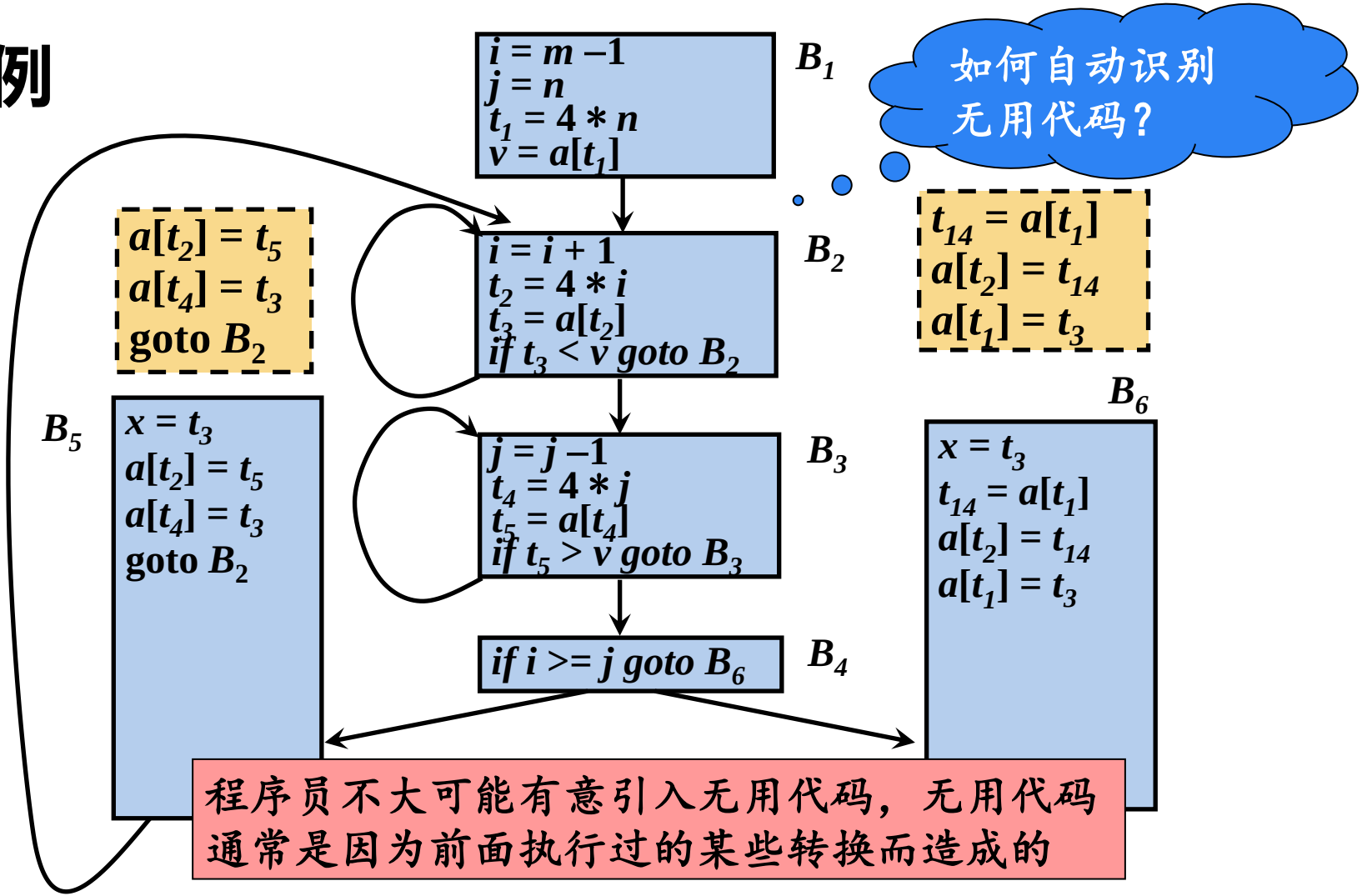
$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = t_3$

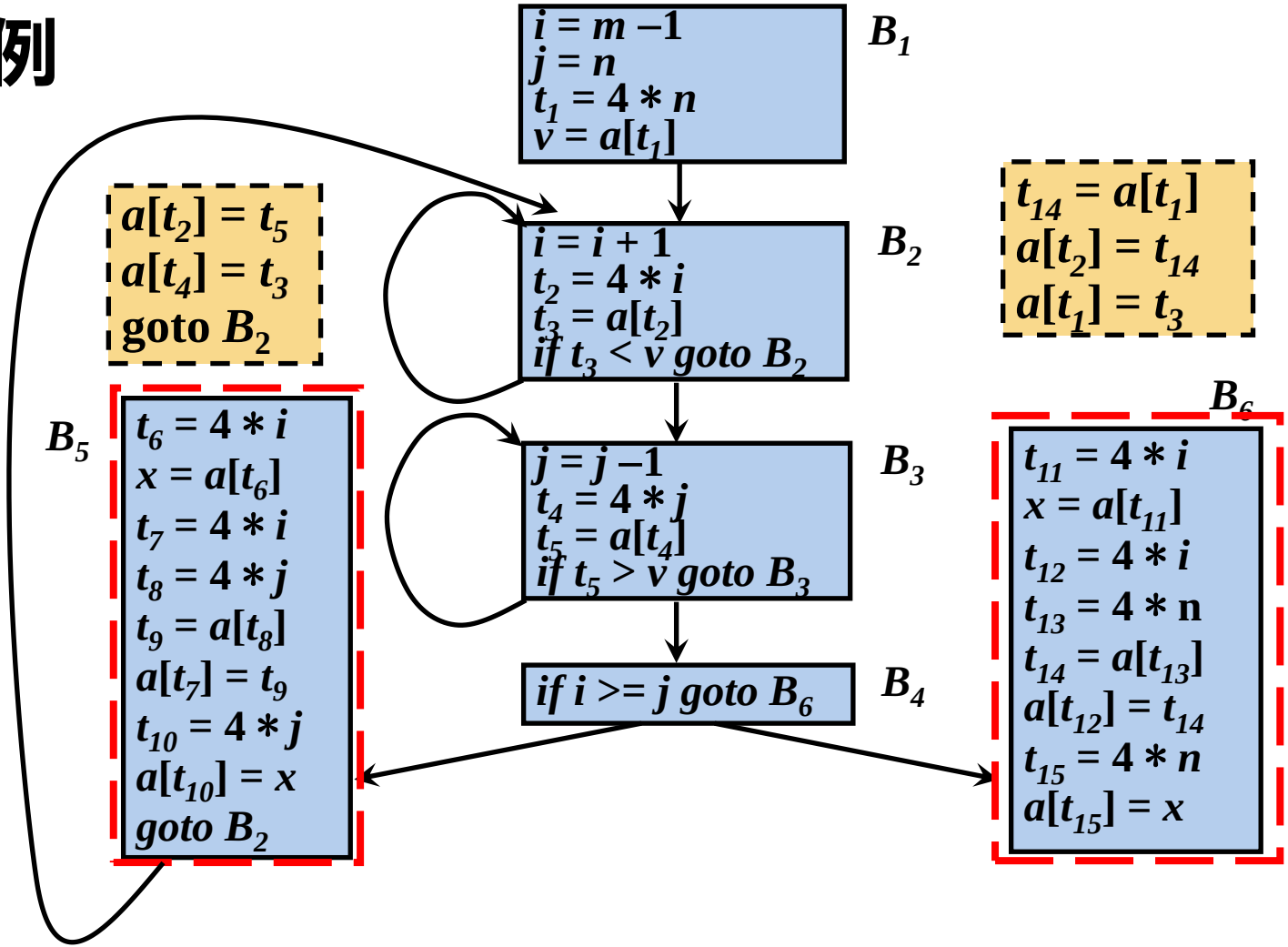
② 删除无用代码

- 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


$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
B_1
如何自动识别无用代码？
$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2
B_2
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = t_3$
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3
B_3
B_5
$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2
if $i >= j$ goto B_6
B_4
B_6
$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = t_3$
程序员不大可能有意引入无用代码，无用代码通常是因为前面执行过的某些转换而造成的

例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
a[t2] = t5
a[t4] = t3
goto B2
B5
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
t14 = a[t1]
a[t2] = t14
a[t1] = t3
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

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“常量传播”）
- 例： $l = 2*3.14*r$

$t_1 = 2 * 3.14$
$t_2 = t_1 * r$
$l = t_2$

$t_1 = 2 * 3.14$
$t_2 = 6.28 * r$
$l = t_2$

常量合并也能给删除无用代码带来机会

③ 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 );
}
```

- 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 );
}
```

循环不变计算

(1) $n = 10$
(2) if n>=360 goto(21)
(3) goto (7)
(4) $t_1 = n + 1$
(5) $n = t_1$
(6) goto (2)
(7) $t_2 = 1 / 360$
(8) $t_3 = t_2 * pi$
(9) $t_4 = t_3 * r$
(10) $t_5 = t_4 * r$
(11) $t_6 = t_5 * n$
(12) $S = t_6$
…
(20) goto (4)
(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; ... }
```

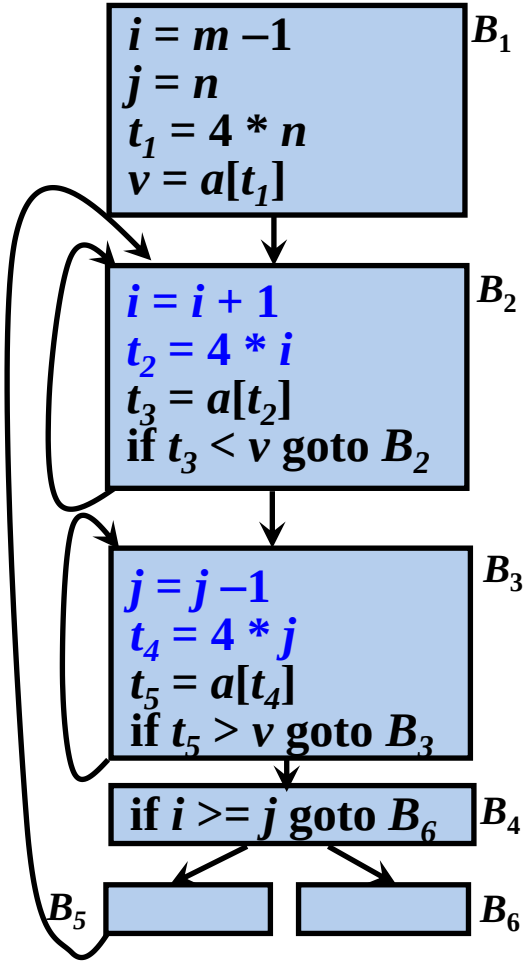
④ 强度削弱(*Strength Reduction*)

- 强度削弱
  - 用较快的操作代替较慢的操作，如用加代替乘
  - 例
    - $2*x$ 或 $2.0*x$ $\Rightarrow$ $x+x$
    - $x/2$ $\Rightarrow$ $x*0.5$
    - x^2 $\Rightarrow$ $x*x$
    - $a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$ $\Rightarrow((\ldots(a_nx+a_{n-1})x+a_{n-2})\ldots)x+a_1)x+a_0$

循环中的强度削弱

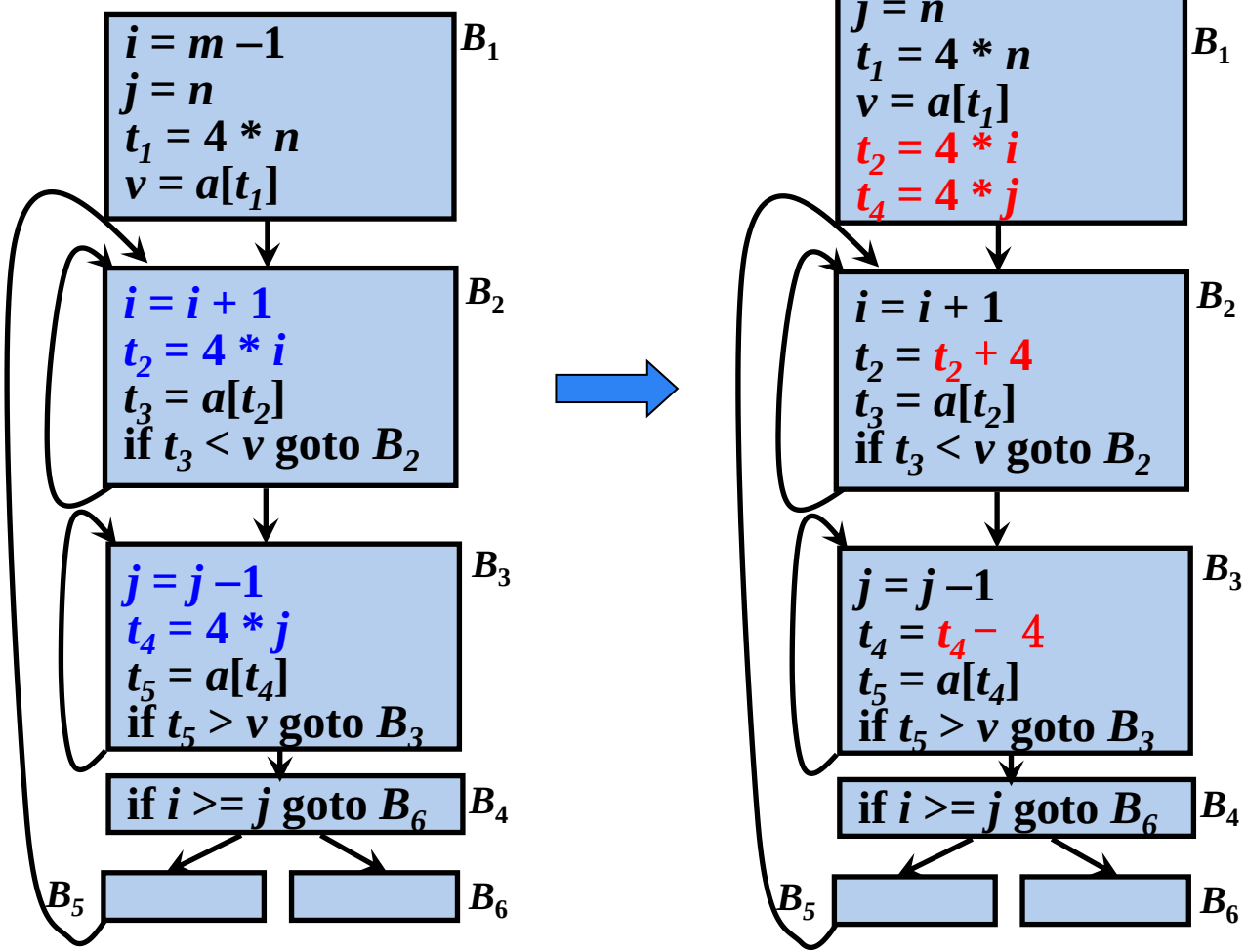
- 归纳变量
  - 对于一个变量x，如果存在一个正的或负的常数c使得每次x被赋值时它的值总增加c，那么x就称为归纳变量(*Induction Variable*)

例

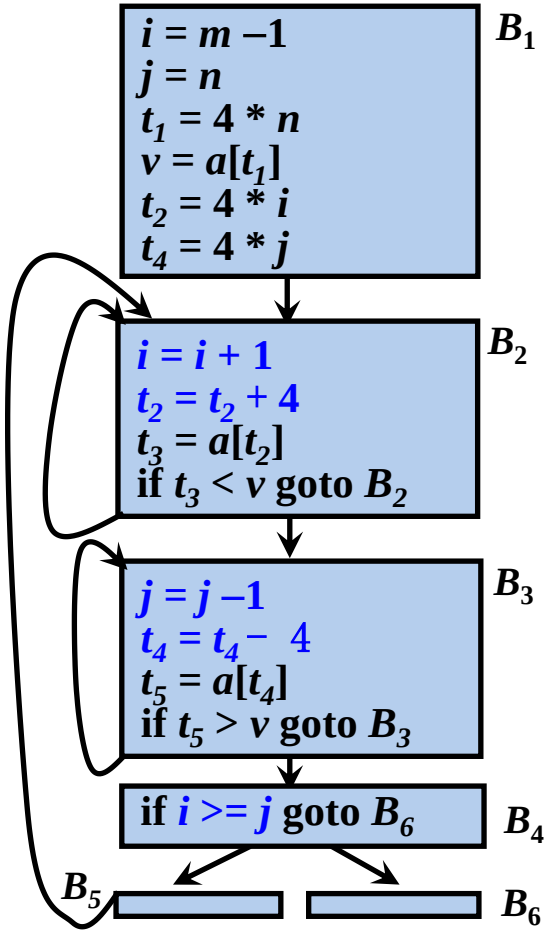


归纳变量可以通过在每次循环迭代中进行一次简单的增量运算(加法或减法)来计算

例

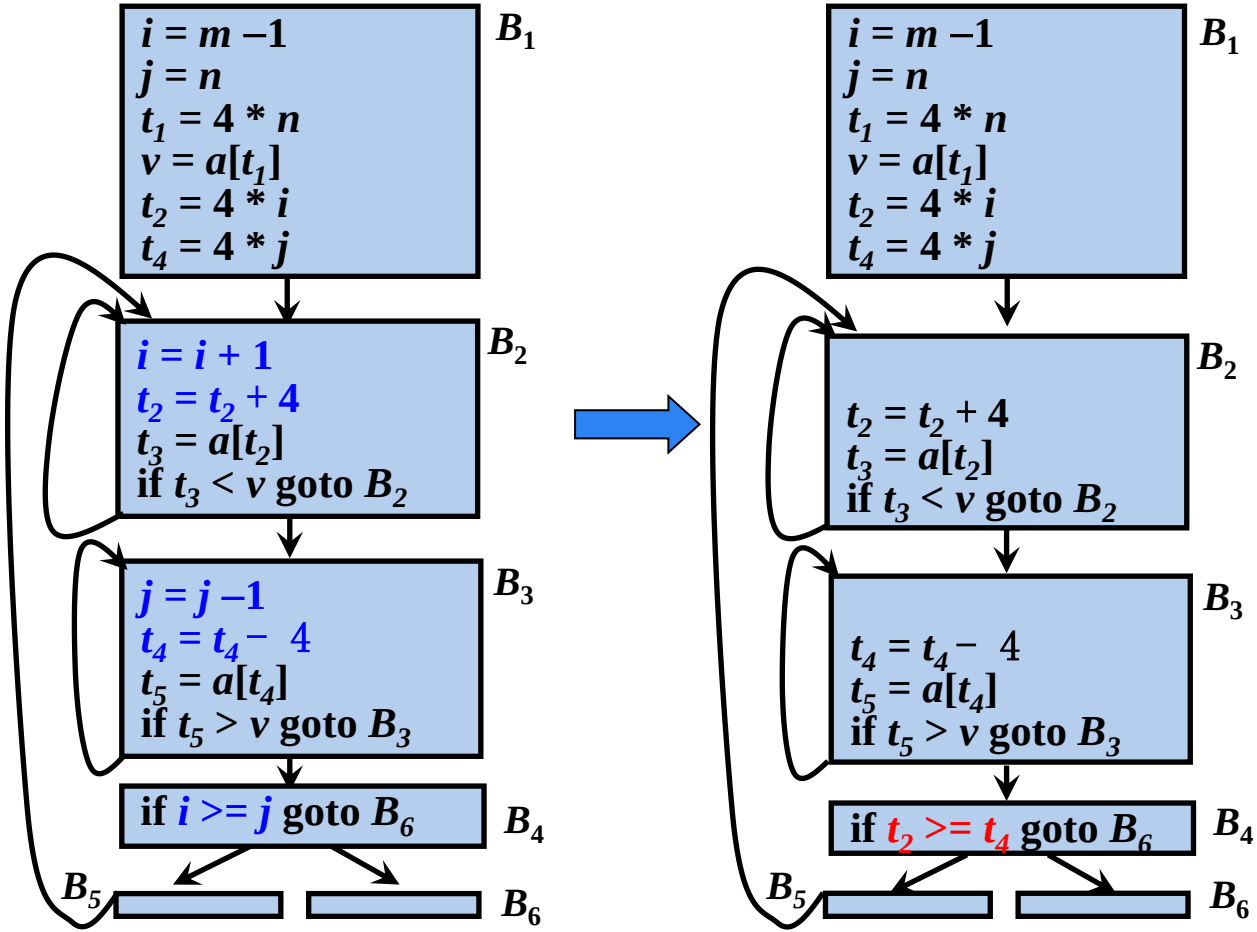

i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6
i = m –1
j = n
t1 = 4 * n
v = a[t1]
t2 = 4 * i
t4 = 4 * j
B1
i = i + 1
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = t4 – 4
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

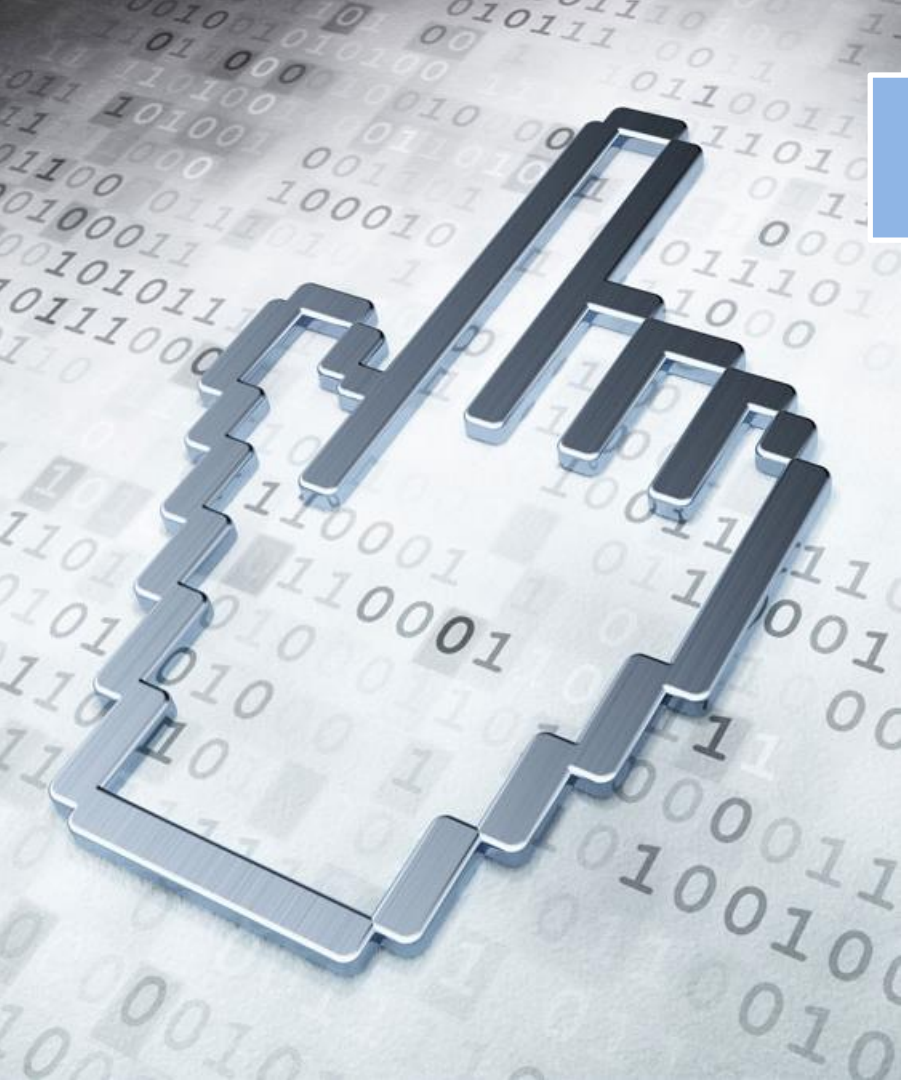
⑤ 删除归纳变量



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

⑤ 删除归纳变量


i = m –1
j = n
t1 = 4 * n
v = a[t1]
t2 = 4 * i
t4 = 4 * j
B1
i = i + 1
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = t4 – 4
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6
i = m –1
j = n
t1 = 4 * n
v = a[t1]
t2 = 4 * i
t4 = 4 * j
B1
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2
B2
t4 = t4 – 4
t5 = a[t4]
if t5 > v goto B3
B3
if t2 >= t4 goto B6
B4
B5
B6



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

- 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*)

基本块的 *DAG* 表示

局部公共子表达式

- 例

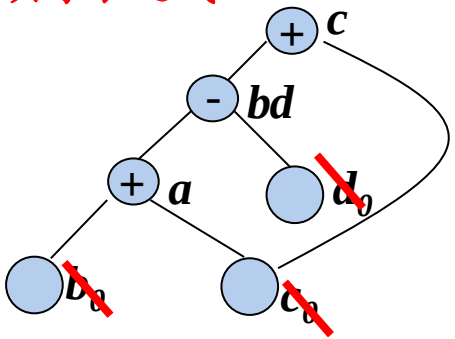
  $a = b + c$

  $b = a - d$

  $c = b + c$

  $d = a - d$

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，就不往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



- 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；同时还有一个定值变量表被关联到N，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，则这个运算分量对应的子结点就是代表该运算分量初始值的叶结点(为区别起见，叶节点的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

基于基本块的 *DAG* 删除无用代码

- 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

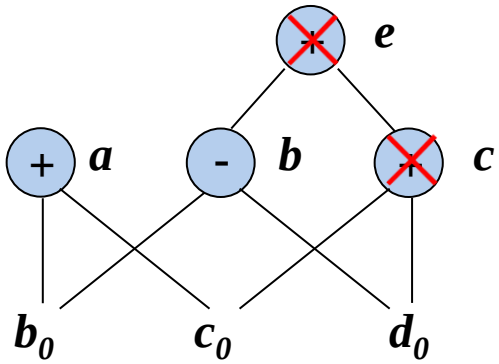
- 例

  $a = b + c$

  $b = b - d$

  ~~$c = c + d$~~

  ~~$e = b + c$~~



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

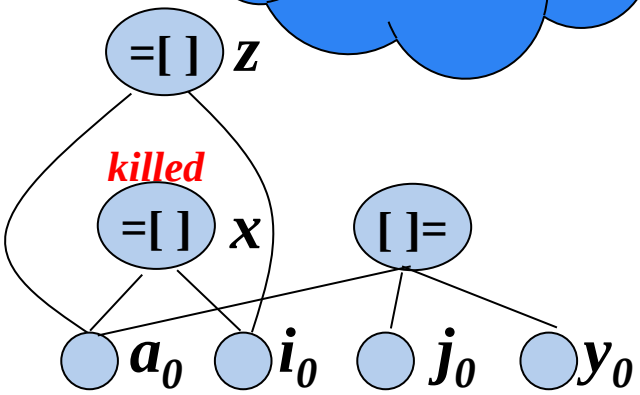
- 例

$x=a[i]$

$a[j]=y$

$z=a[i]$

在构造DAG时，如何防止系统将$a[i]$误判为公共子表达式？

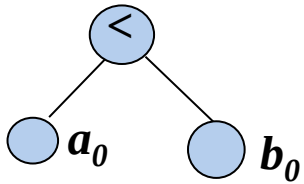


- 对于形如$a[j]=y$的三地址指令，创建一个运算符为“[]=”的结点，这个结点有3个子结点，分别表示a、j和y
- 该结点没有定值变量表
- 该结点的创建将杀死所有已经建立的、其值依赖于a的结点
- 一个被杀死的结点不能再获得任何定值变量，也就是说，它不可能成为一个公共子表达式

跳转指令的表示

- 例

if a <b goto n



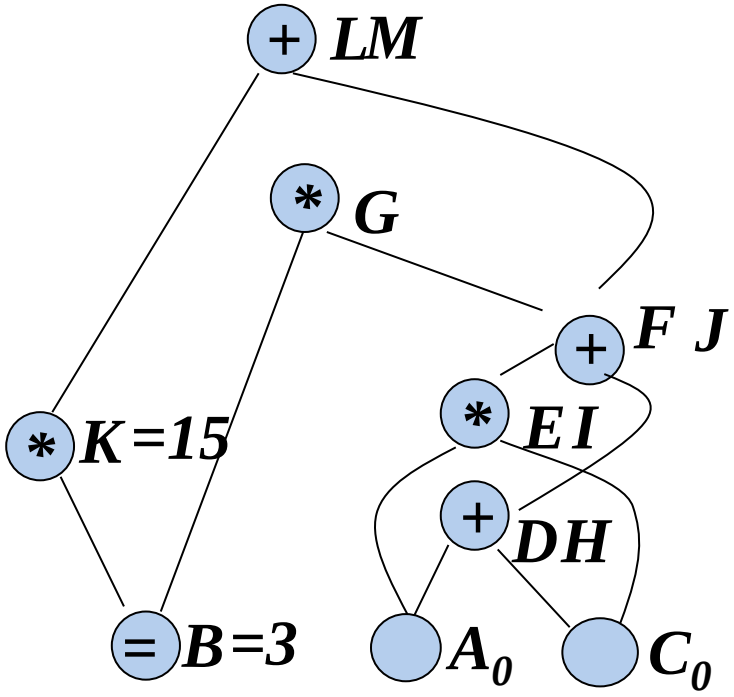
从 *DAG* 到基本块的重组

- 对每个具有若干定值变量的节点，构造一个三地址语句来计算其中某个变量的值
  - 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，但是不包含编译器为处理表达式而生成的临时变量)
  - 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

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 $M = L$

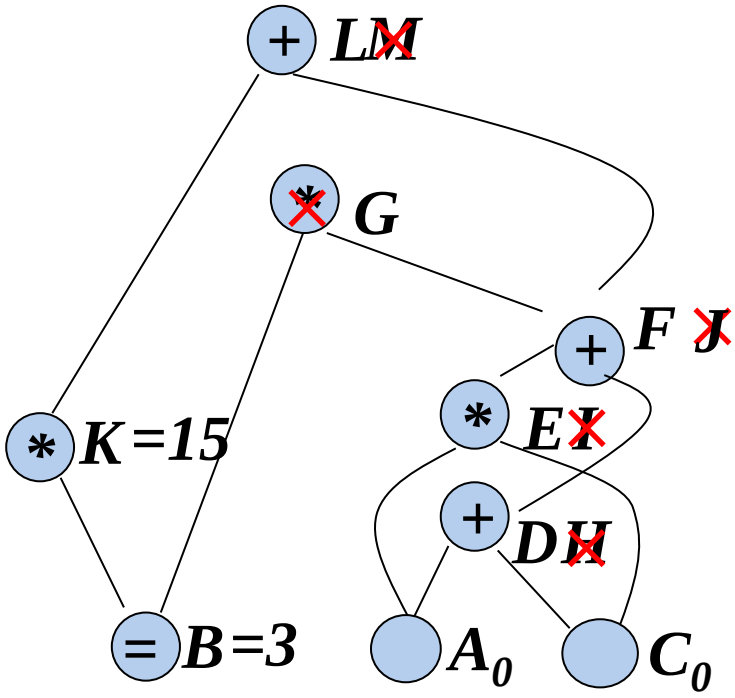


假设：仅变量L在基本块出口之后活跃

例

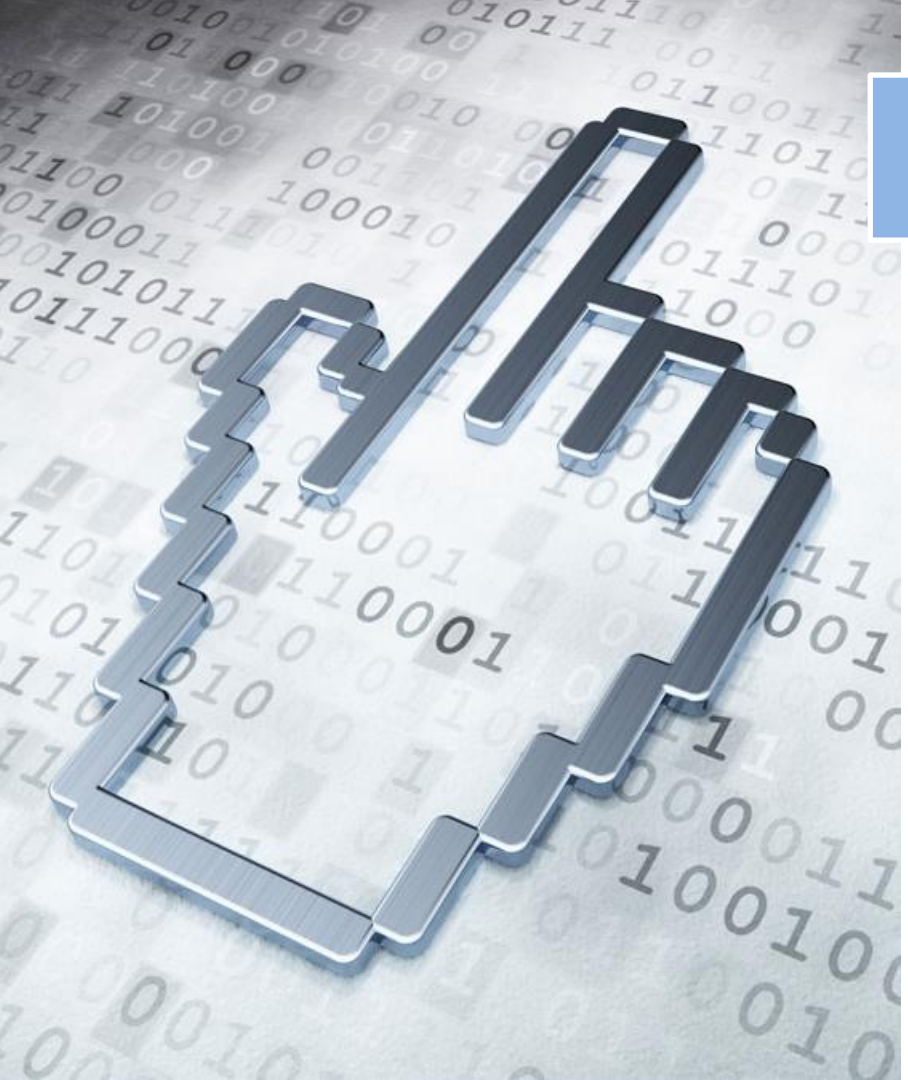
- 给定一个基本块

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 $M = L$



$D = A + C$
$E = A * C$
$F = E + D$
$L = 15 + F$

假设：仅变量L在基本块出口之后活跃



提纲

8.1 流图

8.2 优化的分类

8.3 基本块的优化

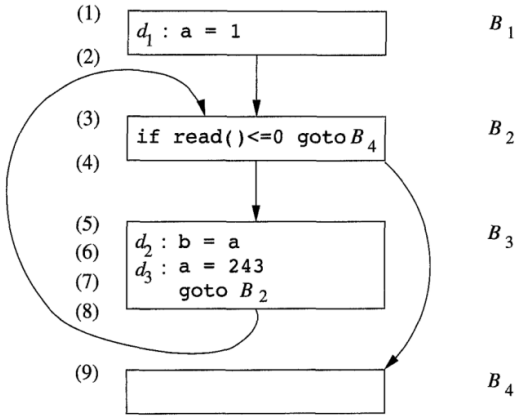
8.4 数据流分析

8.5 流图中的循环

8.6 全局优化

8.4 数据流分析(data-flow analysis)

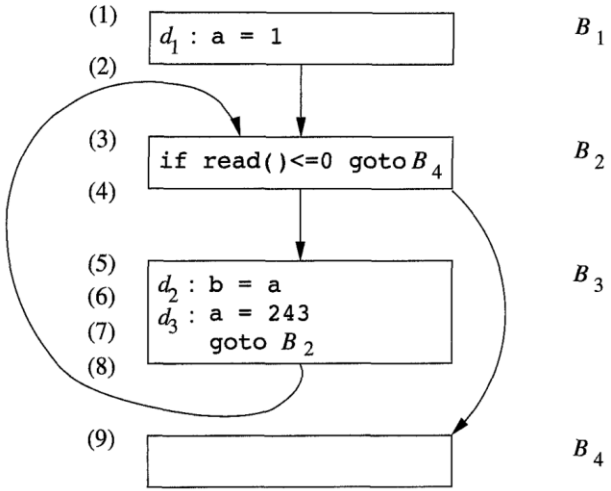
- 数据流分析
  - 一组用来获取有关数据如何沿着程序执行路径流动的相关信息的技术
- 在每一种数据流分析应用中，都会把每个程序点和一个数据流值关联起来
  - 程序点：流图基本块中的位置，包括
    - 第一个语句之前
    - 两个相邻语句之间
    - 最后一个语句之后



- 如果有一个从基本块B_1到基本块B_2的边，那么B_2的第一个语句之前的程序点可能紧跟在B_1的最后一个语句的程序点之后

8.4 数据流分析(data-flow analysis)

- 数据流分析
  - 一组用来获取有关数据如何沿着程序执行路径流动的相关信息的技术
- 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



假设所关心的数据流值为：在每个程序点，变量a可能有哪些值

- 程序点(6)：{ 1, 243 }
- 程序点(7)：{ 243 }

数据流分析的主要应用

- 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

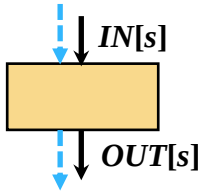
- 语句的数据流模式
  - $IN[s]$: 语句s之前的数据流值

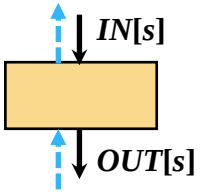
    $OUT[s]$: 语句s之后的数据流值
  - f_s: 语句s的传递函数(transfer function)
    - 一个赋值语句s之前和之后的数据流值的关系
    - 传递函数的两种风格
      - 信息沿执行路径前向传播(前向数据流问题)

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

      - 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
  - f_s: 语句s的传递函数(transfer function)
    - 一个赋值语句s之前和之后的数据流值的关系
  - 基本块中相邻两个语句之间的数据流值的关系
    - 设基本块B由语句$s_1, s_2, \ldots, s_n$顺序组成，则

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

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

  $OUT[B]$: 紧随基本块B之后的数据流值

- 设基本块B由语句$s_1,s_2,\ldots,s_n$ 顺序组成，则
  - $IN[B] = IN[s_1]$
  - $OUT[B] = OUT[s_n]$
- f_B: 基本块B的传递函数
  - 前向数据流问题: $OUT[B] = f_B(IN[B])$

$$f_B = f_{sn}\cdot\ldots\cdot f_{s2}\cdot f_{s1}$$

$$\begin{aligned}
&OUT[B]\\
&= OUT[s_n]\\
&= f_{sn}(IN[s_n])\\
&= f_{sn}(OUT[s_{n-1}])\\
&= f_{sn}\cdot f_{s(n-1)}(IN[s_{n-1}])\\
&= f_{sn}\cdot f_{s(n-1)}(OUT[s_{n-2}])\\
&\ldots\\
&= f_{sn}\cdot f_{s(n-1)}\cdot\ldots\cdot f_{s2}(OUT[s_1])\\
&= f_{sn}\cdot f_{s(n-1)}\cdot\ldots\cdot f_{s2}\cdot f_{s1}(IN[s_1])\\
&= f_{sn}\cdot f_{s(n-1)}\cdot\ldots\cdot f_{s2}\cdot f_{s1}(IN[B])
\end{aligned}$$

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

- 设基本块B由语句$s_1,s_2,\ldots,s_n$顺序组成，则
  - $IN[B] = IN[s_1]$
  - $OUT[B] = OUT[s_n]$
- f_B：基本块B的传递函数
  - 前向数据流问题：$OUT[B] = f_B(IN[B])$

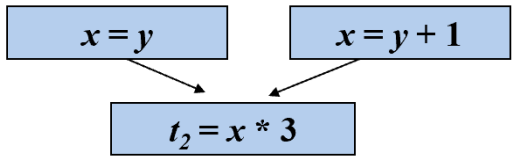
    $f_B = f_{sn}\cdot\ldots\cdot f_{s2}\cdot f_{s1}$
  - 逆向数据流问题：$IN[B] = f_B(OUT[B])$

    $f_B = f_{s1}\cdot f_{s2}\cdot\ldots\cdot f_{sn}$

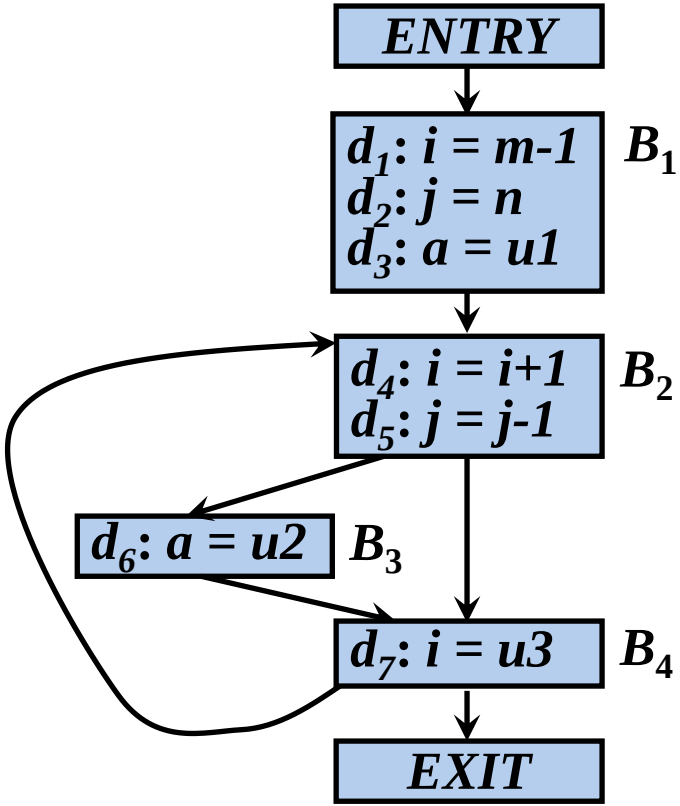
$$
\begin{aligned}
&IN[B]\\
&= IN[s_1]\\
&= f_{s1}(OUT[s_1])\\
&= f_{s1}(IN[s_2])\\
&= f_{s1}\cdot f_{s2}(OUT[s_2])\\
&= f_{s1}\cdot f_{s2}(IN[s_3])\\
&\ldots\\
&= f_{s1}\cdot f_{s2}\cdot\ldots\cdot f_{s(n-1)}(IN[s_n])\\
&= f_{s1}\cdot f_{s2}\cdot\ldots\cdot f_{s(n-1)}\cdot f_{sn}(OUT[s_n])\\
&= f_{s1}\cdot f_{s2}\cdot\ldots\cdot f_{s(n-1)}\cdot f_{sn}(OUT[B])
\end{aligned}
$$

8.4.1 到达定值分析

- 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d没有被“杀死”(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'“杀死”了)，则称定值d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
- 常量传播
  - 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
- 常量传播
- 判定变量x在p点上是否未经定值就被引用

“生成”与“杀死”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“杀死”了程序中其它对u的定值

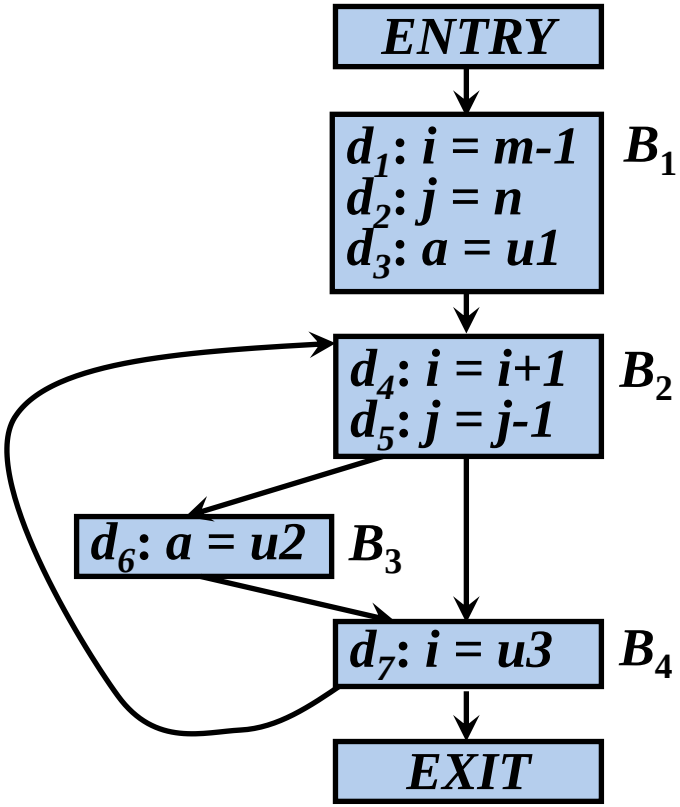
到达定值的传递函数

- f_d：定值d: $u = v + w$的传递函数
  - $f_d(x) = gen_d \cup (x\text{-}kill_d)$ —— 生成–杀死形式
    - gen_d：由语句d生成的定值的集合　　$gen_d = \{d\}$
    - $kill_d$：由语句d杀死的定值的集合（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
  - $f_d(x) = gen_d \cup (x - kill_d)$
- f_B：基本块B的传递函数
  - $f_B(x) = gen_B \cup (x - kill_B)$
    - $kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$
      - 被基本块B中各个语句杀死的定值的集合
    - $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \ldots \cup (gen_1 - kill_2 - kill_3 - \ldots - kill_n)$
      - 基本块中没有被块中各语句“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

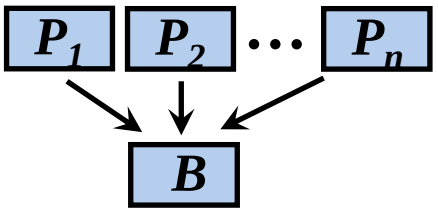
  $OUT[B]$：到达流图中基本块B的出口处的定值的集合

- 方程
  - $OUT[ENRTY]=\Phi$
  - $OUT[B]=f_B(IN[B])$ （$B\neq ENTRY$）
    - $f_B(x)=gen_B \cup (x\text{-}kill_B)$

  $OUT[B]=gen_B \cup (IN[B]\text{-}kill_B)$

  - $IN[B]=\cup_{P是B的一个前驱} OUT[P]$ （$B\neq ENTRY$）

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

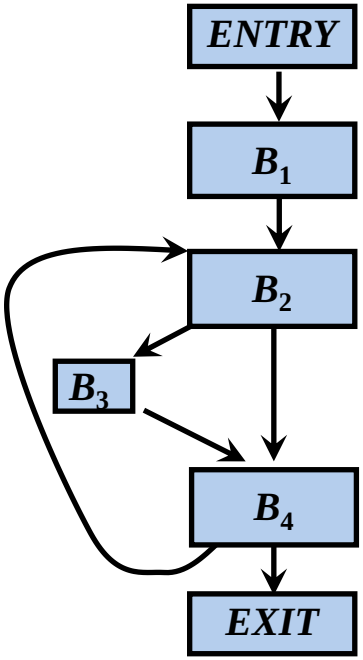


计算到达定值的迭代算法

- 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) {
        $IN[B] = \cup_{P是B的一个前驱} OUT[P]$;
        $OUT[B] = gen_B \cup (IN[B] - kill_B)$
    }

例

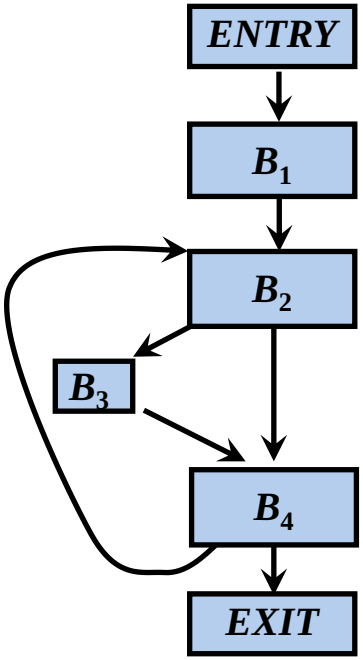


gen_{B1}={ d_1, d_2, d_3 }
$kill_{B1}$={ d_4, d_5, d_6, d_7 }
gen_{B2}={ d_4, d_5 }
$kill_{B2}$={ d_1, d_2, d_7 }
gen_{B3}={ d_6 }
$kill_{B3}$={ d_3 }
gen_{B4}={ d_7 }
$kill_{B4}$={ d_1, d_4 }

$OUT[ENTRY] = \Phi$;
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*) {
    $IN[B] = \cup_{P是B的一个前驱} OUT[P]$;
    $OUT[B] = gen_B \cup (IN[B] - kill_B)$
  }

B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$	$IN[B]^3$	$OUT[B]^3$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111	001 0111	001 0111

例



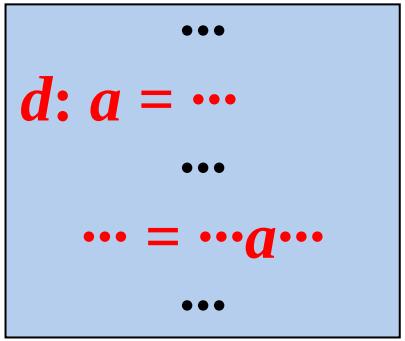
$gen_{B1}=\{ d_1, d_2, d_3 \}$
$kill_{B1}=\{ d_4, d_5, d_6, d_7 \}$
$gen_{B2}=\{ d_4, d_5 \}$
$kill_{B2}=\{ d_1, d_2, d_7 \}$
$gen_{B3}=\{ d_6 \}$
$kill_{B3}=\{ d_3 \}$
$gen_{B4}=\{ d_7 \}$
$kill_{B4}=\{ d_1, d_4 \}$

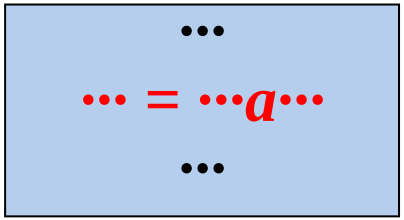
IN[B]	B_2	B_3	B_4
d_1	√	×	×
d_2	√	×	×
d_3	√	√	√
d_4	×	√	√
d_5	√	√	√
d_6	√	√	√
d_7	√	×	×

B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$	$IN[B]^3$	$OUT[B]^3$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111	001 0111	001 0111

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

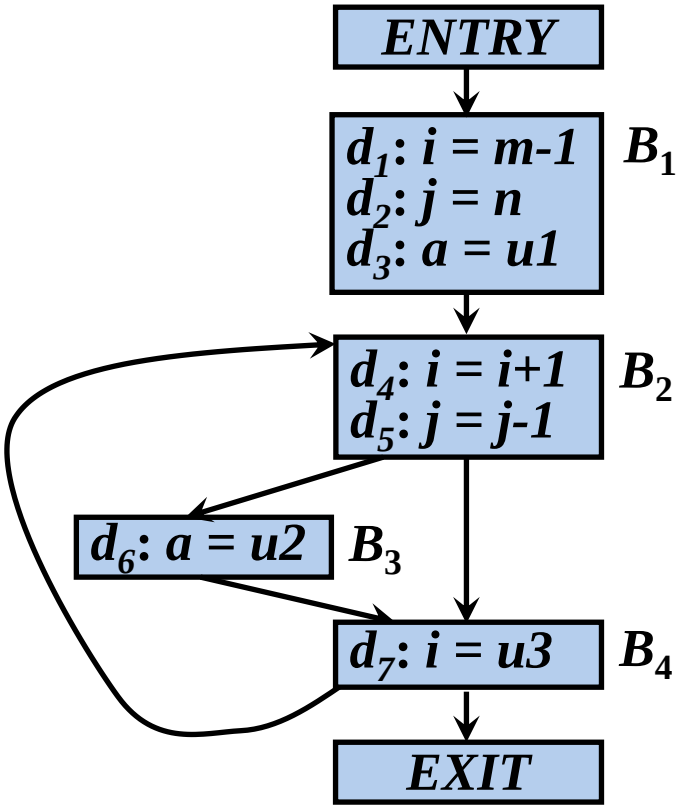




8.4.2 活跃变量分析

- 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



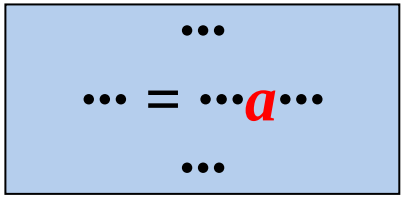
OUT[B]	B_1	B_2	B_3	B_4
a	×	×	×	×
i	√	×	×	√
j	√	√	√	√
m	×	×	×	×
n	×	×	×	×
u1	×	×	×	×
u2	√	√	√	√
u3	√	√	√	√

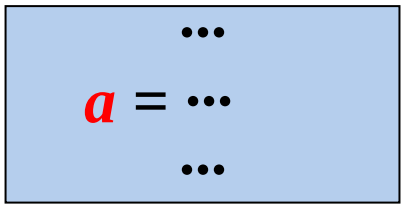
活跃变量信息的主要用途

- 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

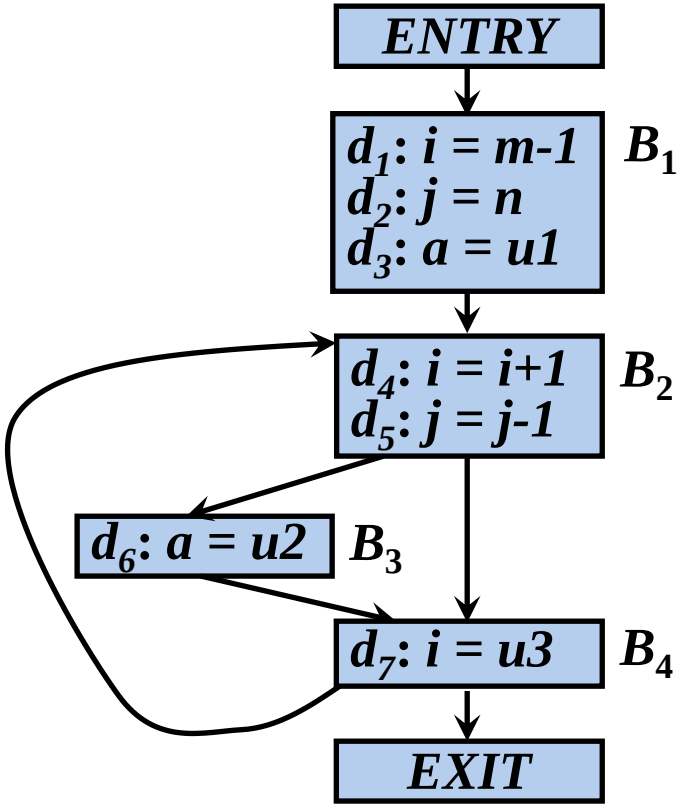
活跃变量的传递函数

- 逆向数据流问题
  - $IN[B] = f_B(OUT\ [B])$
- $f_B(x) = use_B \cup (x\text{-}def_B)$
  - 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
  - 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





例：各基本块B的use_B和def_B



- use_{B1} ={ m, n, u1 }
- def_{B1} ={ i, j, a }
- use_{B2} ={ i, j }
- def_{B2}= Φ
- use_{B3} ={ u2 }
- def_{B3} ={ a }
- use_{B4} ={ u3 }
- def_{B4} ={ 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

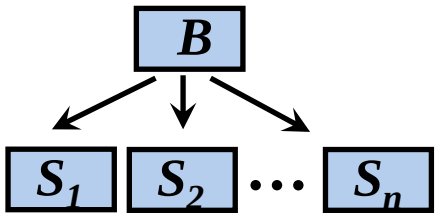
  $OUT[B]$：在基本块B的出口处的活跃变量集合

- 方程
  - $IN[EXIT] = \Phi$
  - $IN[B] = f_B(OUT[B])$ $(B \neq EXIT)$
    - $f_B(x) = use_B \cup (x - def_B)$

  $$IN[B] = use_B \cup (OUT[B] - def_B)$$

  - $OUT[B] = \cup_{S\text{是}B\text{的一个后继}} IN[S]$ $(B \neq EXIT)$

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)

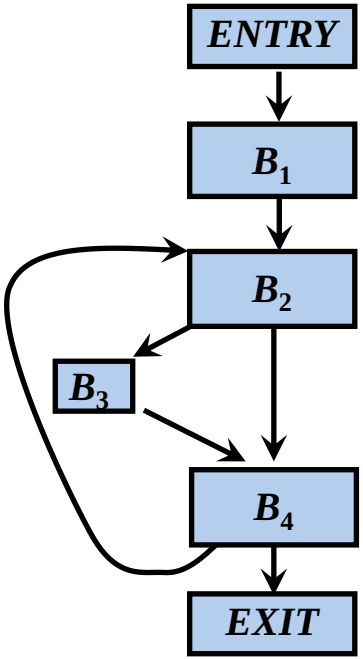
  for (除$EXIT$之外的每个基本块B) {

    $OUT[B] = \cup_{S是B的一个后继} IN[S]$;

    $IN[B] = use_B \cup (OUT[B] - def_B)$;

  }

例

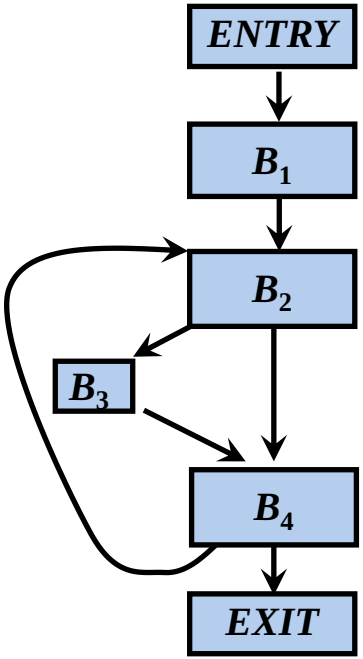


use_{B1} ={ m, n, u1 }
def_{B1} ={ i, j, a }
use_{B2} ={ i, j }
def_{B2}= Φ
use_{B3} ={ u2 }
def_{B3} ={ a }
use_{B4} ={ u3 }
def_{B4} ={ i }

$IN[EXIT] = \Phi$;
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*) {
    $OUT[B] = \cup_{S是B的一个后继} IN[S]$;
    $IN[B] = use_B \cup (OUT[B] - def_B)$;
  }

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4		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



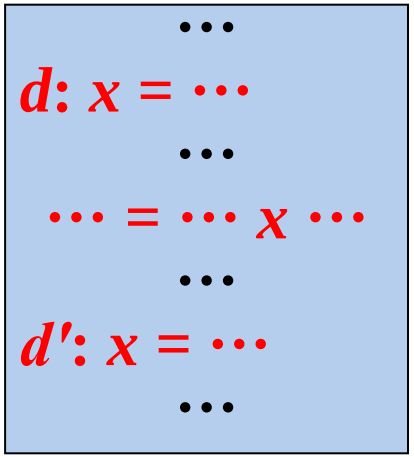
use_{B1} ={ m, n, u1 }
def_{B1} ={ i, j, a }
use_{B2} ={ i, j }
def_{B2}= Φ
use_{B3} ={ u2 }
def_{B3} ={ a }
use_{B4} ={ u3 }
def_{B4} ={ i }

OUT[B]	B_1	B_2	B_3	B_4
a	×	×	×	×
i	√	×	×	√
j	√	√	√	√
m	×	×	×	×
n	×	×	×	×
u_1	×	×	×	×
u_2	√	√	√	√
u_3	√	√	√	√

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^2$	$IN[B]^2$
B_4		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

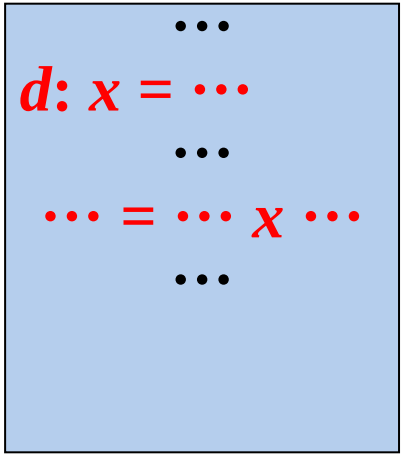
定值-引用链 (*Definition-Use Chains*)

- 定值-引用链：设变量x有一个定值d，该定值所有能够到达的引用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，该定值所有能够到达的引用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

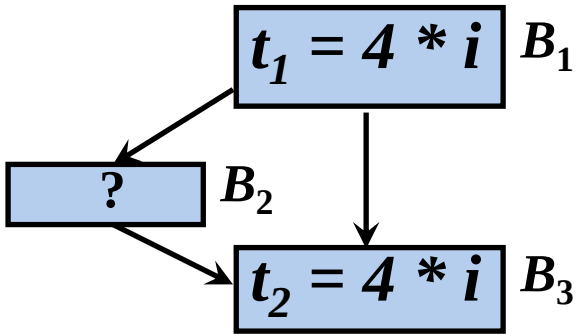


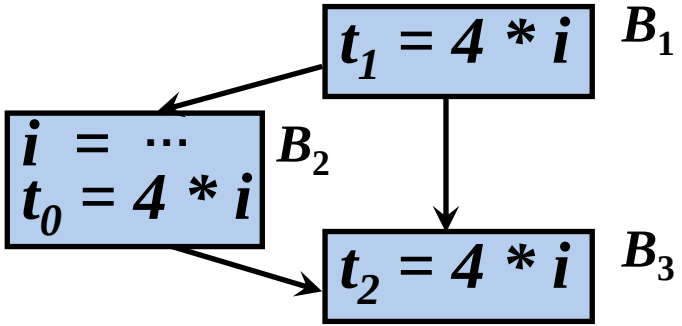
8.4.3 可用表达式分析

- 可用表达式
  - 如果从流图的首节点到达程序点 p 的每条路径都对表达式 x 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 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 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

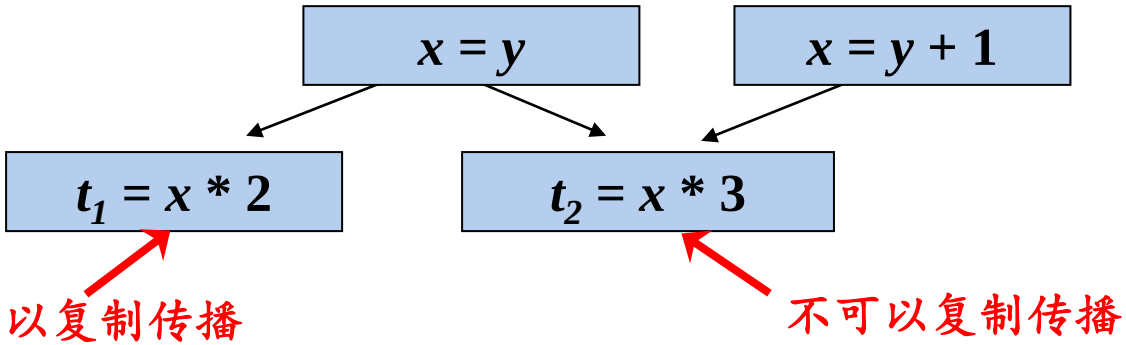




如果i在B_2中没有被赋予新值，或者在B_2中，对i赋值后又重新计算了$4*i$

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

从流图的首节点到达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

- 对于可用表达式数据流模式而言，如果基本块B对x 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 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 op y，则称B杀死表达式x op y
- $f_B(x)= e\_gen_B \cup (x- e\_kill_B)$
  - 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所杀死的U中的可用表达式的集合
    - U：所有出现在程序中一个或多个语句的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
例：$x = x \text{ op } y$

语句	可用表达式
……	Ø
a : = b+c	
……	{ b+c }
b : = a-d	
……	{ a-d }
c : = b+c	
……	{ a-d }
d : = a-d	
……	Ø

e_kill_B的计算

- 初始化：$e\_kill_B = \Phi$
- 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

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

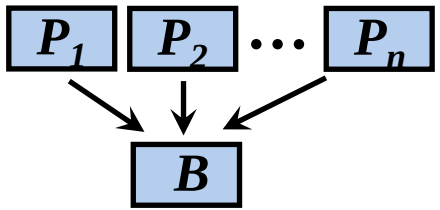
  $OUT[B]$：在B的出口处可用的U中的表达式集合

- 方程
  - $OUT[ENTRY]=\Phi$
  - $OUT[B]=f_B(IN[B])$　$(B\neq ENTRY)$
    - $f_B(x)=e\_gen_B \cup (x-e\_kill_B)$

  $$OUT[B]=e\_gen_B \cup (IN[B]-e\_kill_B)$$

  - $IN[B]=\cap_{P是B的一个前驱} OUT[P]$　　$(B\neq ENTRY)$

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



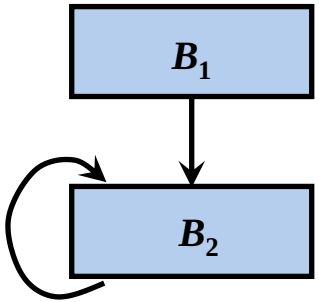
计算可用表达式的迭代算法

- 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) {
    $IN[B] = \cap_{P是B的一个前驱} OUT[P]$
    $OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$;
    }

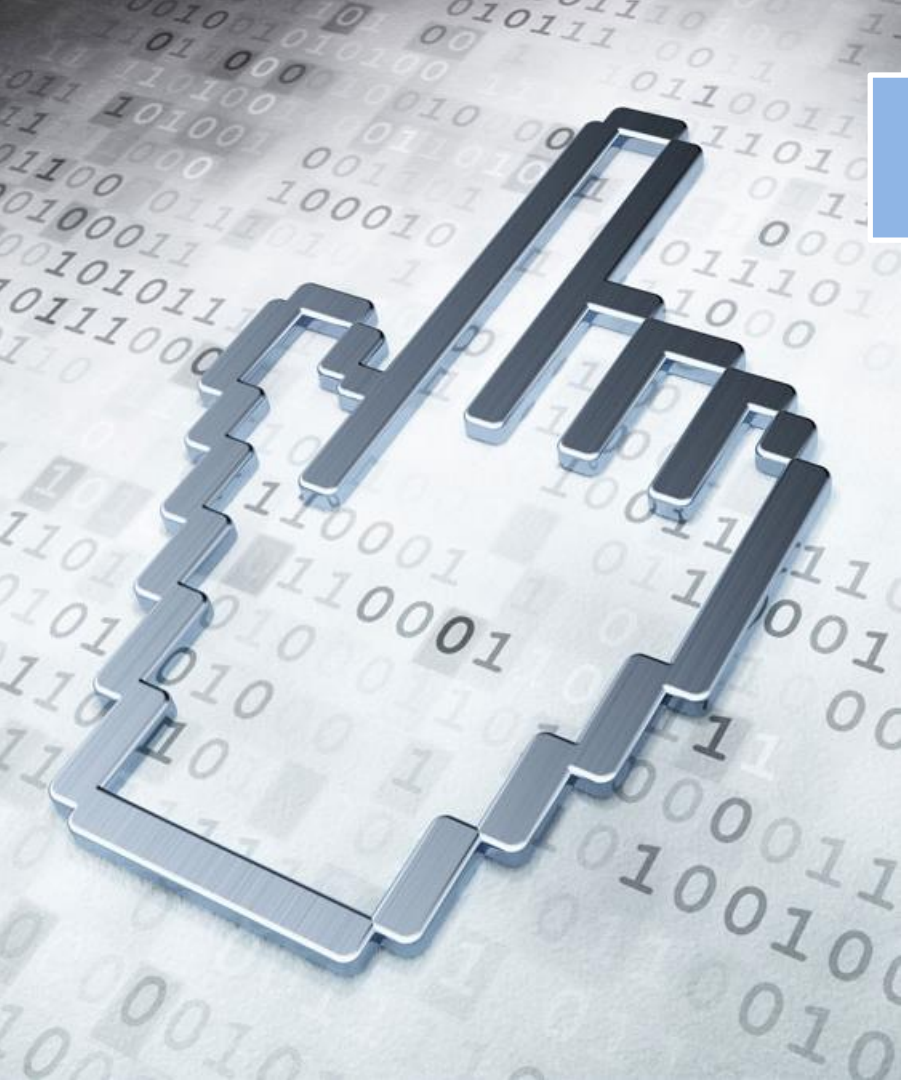
为什么将$OUT[B]$集合初始化为U?

- 将OUT集合初始化为Φ局限性太大
  - 例



$$IN[B_2]^1 = OUT[B_1]^1 \cap OUT[B_2]^0$$

$$= \begin{cases} \Phi & \text{，如果 } OUT[B_2]^0 = \Phi \\ OUT[B_1]^1 & \text{，如果 } OUT[B_2]^0 = U \end{cases}$$



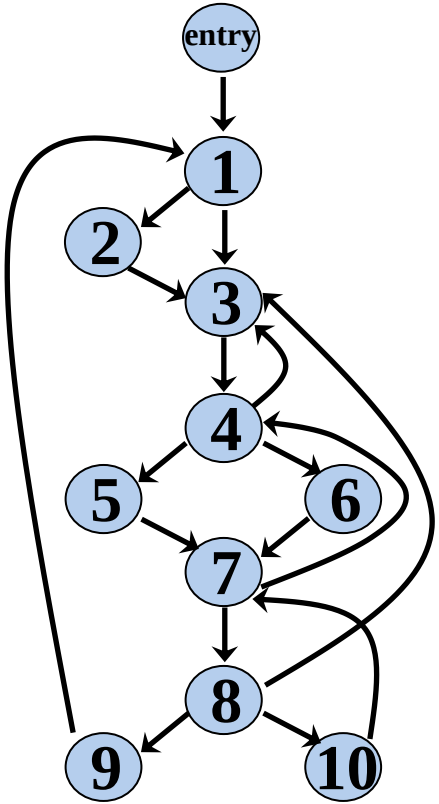
提纲

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 dom n*

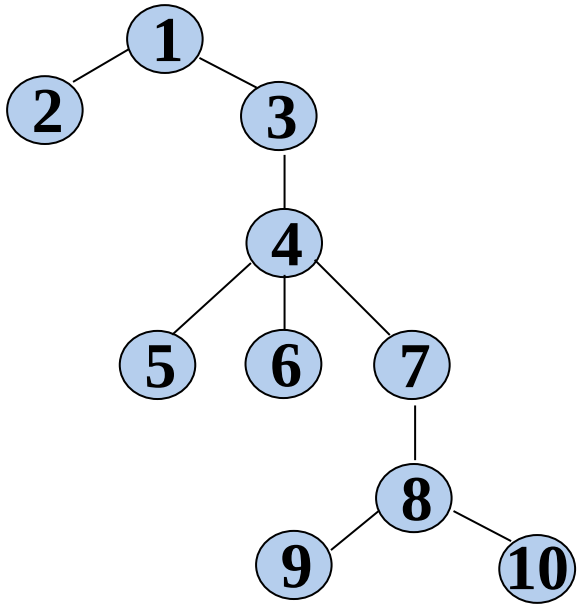
每个结点都支配它自己

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*)

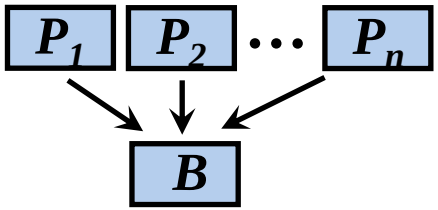


每个结点只支配它和它的后代结点

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
    - $OUT[ENTRY] = \{ ENTRY \}$
    - $OUT[B] = f_B(IN[B])$ $(B \neq ENTRY)$
      - $f_B(x) = x \cup \{ B \}$
    - $IN[B] = \cap_{P是B的一个前驱} OUT[P]$ $(B \neq ENTRY)$



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

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
- 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)

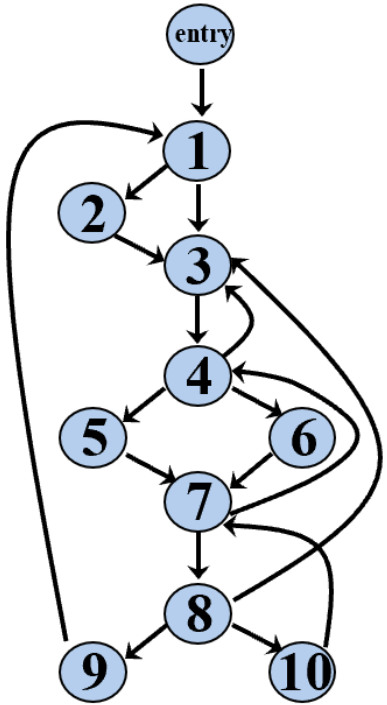
    { $IN[B]=\cap_{P是B的一个前驱} OUT[P]$

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

    }

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]=∩P是B的一个前驱 OUT[P]
      OUT[B]=IN[B]∪{B}
    }
```

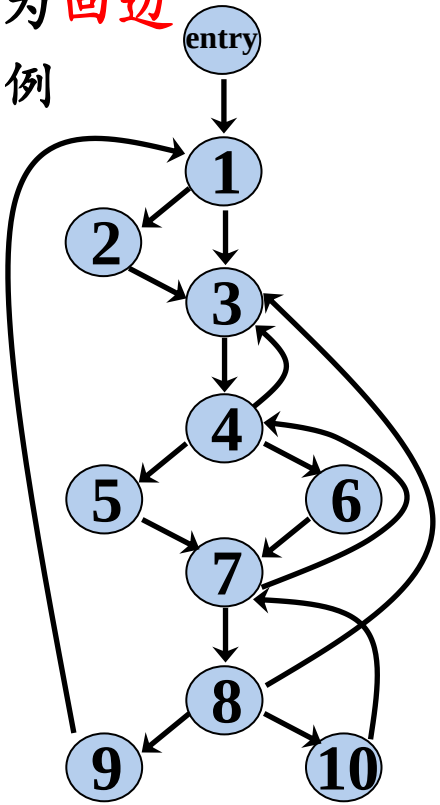


	$OUT^0[B]$	$IN^1[B]$	$OUT^1[B]$
E	{ E }		
①	N	{ E }	{ E ① }
②	N	{ E ① }	{ E ① ② }
③	N	{ E ① }	{ E ① ③ }
④	N	{ E ① ③ }	{ E ① ③ ④ }
⑤	N	{ E ① ③ ④ }	{ E ① ③ ④ ⑤ }
⑥	N	{ E ① ③ ④ }	{ E ① ③ ④ ⑥ }
⑦	N	{ E ① ③ ④ }	{ E ① ③ ④ ⑦ }
⑧	N	{ E ① ③ ④ ⑦ }	{ E ① ③ ④ ⑦ ⑧ }
⑨	N	{ E ① ③ ④ ⑦ ⑧ }	{ E ① ③ ④ ⑦ ⑧ ⑨ }
⑩	N	{ E ①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

回边 (*Back Edges*)

~~假定流图中存在两个结点d和n满足d dom n。~~如果存在从结点n到d的有向边$n \to d$，那么这条边称为回边

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



n　　d

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：

4->3

7->4

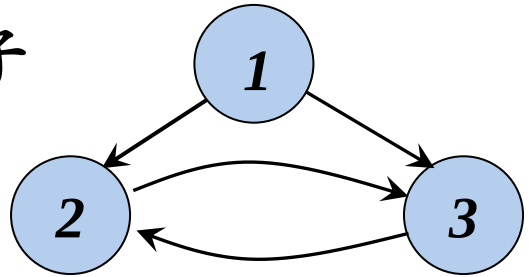
8->3

9->1

10->7

自然循环 (*Natural Loops*)

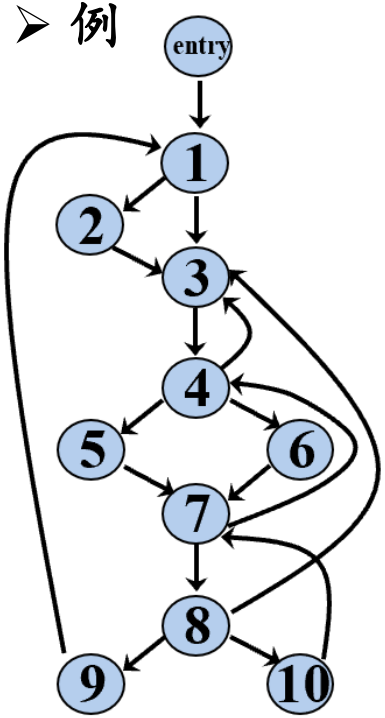
- 从程序分析的角度来看，循环在代码中以什么形式出现并不重要，重要的是它是否具有易于优化的性质
- 自然循环是一种适合于优化的循环，它满足以下性质
  - 有唯一的入口结点，称为首结点(*header*)
    - 首结点支配循环中的所有结点
  - 循环中至少有一条返回首结点的路径，否则，控制就不可能从“循环”中直接回到循环头，也就无法构成循环
- 非自然循环的例子



自然循环的识别

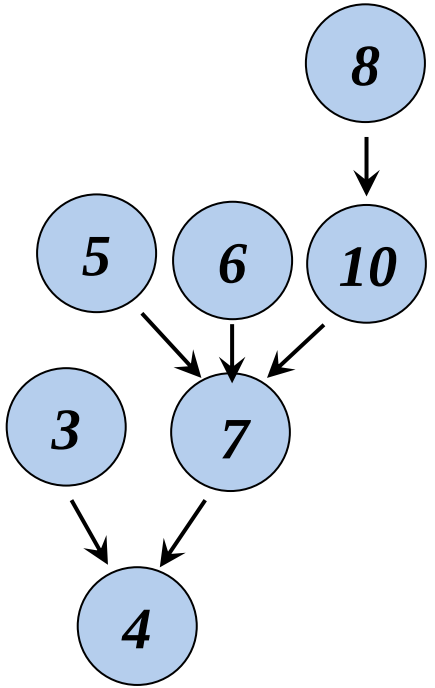
- 给定一个回边$n \rightarrow d$，该回边的自然循环为：d，以及所有可以不经过d而到达n的结点。d为该循环的首结点。

- 例



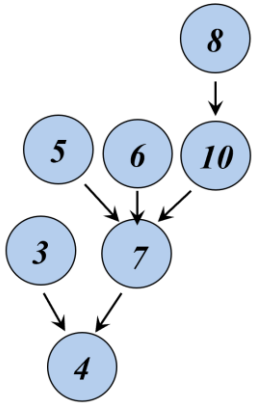
n d

回边	自然循环
4→3	③④⑤⑥⑦⑧⑩
7→4	④⑤⑥⑦⑧⑩
8→3	③④⑤⑥⑦⑧⑩
9→1	① ~ ⑩
10→7	⑦⑧⑩



算法：构造一条回边的自然循环

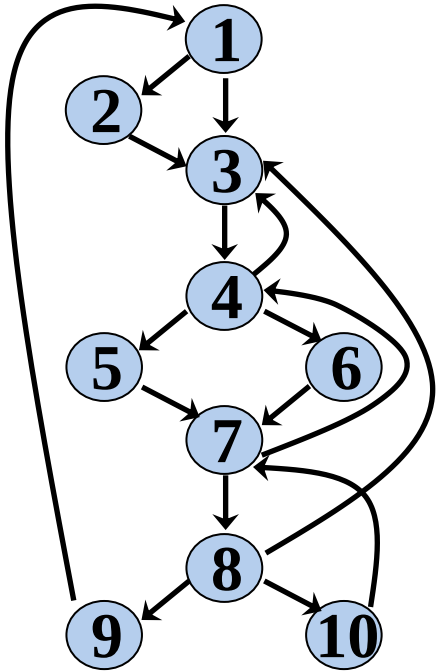
- 输入：流图 G和回边$n \to d$
- 输出：由回边$n \to d$的自然循环中的所有结点组成的集合
- 方法：




```
stack= Φ;  loop={ n, d };  push(stack, 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 ∪{ p}; push(stack,p); }
    }
}
```

结点d在初始时刻已经在$loop$中，不会把它放入栈中，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。

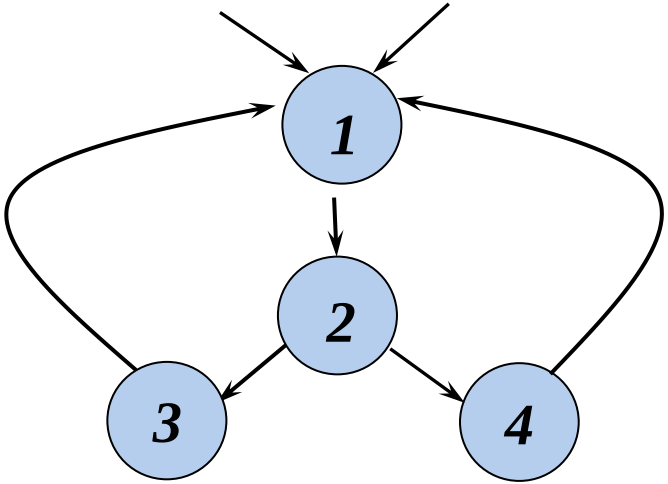
- 自然循环的一个重要性质
  - 如果两个自然循环的首结点不相同，则这两个循环要么互不相交，要么一个完全包含(嵌入)在另外一个里面
    - 例



回边	自然循环
4->3	③④⑤⑥⑦⑧⑩
7->4	④⑤⑥⑦⑧⑩
8->3	③④⑤⑥⑦⑧⑩
9->1	① ~ ⑩
10->7	⑦⑧⑩

最内循环 (*Innermost Loops*):
不包含其它循环的循环

- 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
  - 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



数据流分析技术小结

- 数据流分析技术及其主要应用
  - 到达定值分析
  - 活跃变量分析
  - 可用表达式分析
  - 支配结点分析

数据流分析的主要应用

- 到达定值　∃
  - 定义：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d没有被“杀死”(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'“杀死”了)，则称定值d到达程序点p
  - 分析任务：程序点p处使用的x的值可能是在哪里定值的
  - 分析结果的表示：ud链(引用-定值链)
  - 主要用途
    - 循环不变计算的检测
    - 检测“变量未经定值就被引用”
    - 常量传播

数据流分析的主要应用

- 到达定值

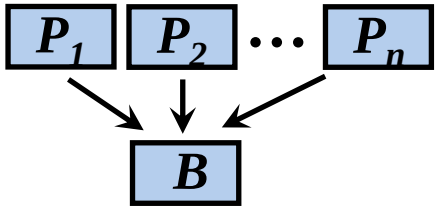
∃ 正向

$OUT[ENTRY] = \Phi$

$OUT[B] = \Phi$

$IN[B] = \cup_{P是B的一个前驱} OUT[P]$

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



数据流分析的主要应用

- 到达定值　∃　正向

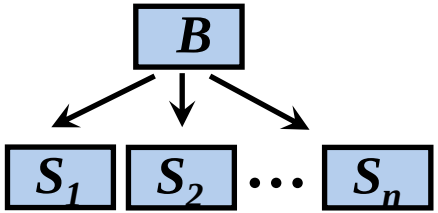
$$OUT[ENTRY] = \Phi \qquad IN[B] = \cup_{P\text{是}B\text{的一个前驱}} OUT[P]$$
$$OUT[B] = \Phi \qquad OUT[B] = gen_B \cup (IN[B] - kill_B)$$

- 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*)
  - 分析任务：*x*在程序点*p*处的定值在哪里会被使用
  - 分析结果的表示：*du*链 (定值-引用链)
  - 主要用途
    - 删除无用赋值
    - 寄存器分配

数据流分析的主要应用

- 到达定值　∃　正向
- 活跃变量　∃　逆向

$OUT[ENTRY] = \Phi$	$IN[B] = \cup_{P是B的一个前驱} OUT[P]$
$OUT[B] = \Phi$	$OUT[B] = gen_B \cup (IN[B] - kill_B)$
$IN[EXIT] = \Phi$	$OUT[B] = \cup_{S是B的一个后继} IN[S]$
$IN[B] = \Phi$	$IN[B] = use_B \cup (OUT[B] - def_B)$



数据流分析的主要应用

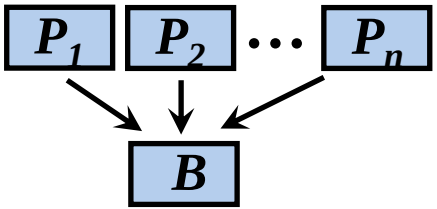
- 到达定值 ∃ 正向
- 活跃变量 ∃ 逆向
- 可用表达式 ∀
  - 如果从流图的首节点到达程序点 p 的每条路径都对表达式 x op y 进行计算，并且从最后一个这样的计算到点 p 之间没有再次对 x 或 y 定值，那么表达式 x op y 在点 p 是可用的(*available*)
  - 主要用途
    - 消除全局公共子表达式
    - 复制传播

$OUT[ENTRY] = \Phi$	$IN[B] = \cup_{P是B的一个前驱} OUT[P]$
$OUT[B] = \Phi$	$OUT[B] = gen_B \cup (IN[B] - kill_B)$
$IN[EXIT] = \Phi$	$OUT[B] = \cup_{S是B的一个后继} IN[S]$
$IN[B] = \Phi$	$IN[B] = use_B \cup (OUT[B] - def_B)$

数据流分析的主要应用

- 到达定值 ∃ 正向
- 活跃变量 ∃ 逆向
- 可用表达式 ∀ 正向

$OUT[ENTRY] = \Phi$ $OUT[B] = \Phi$	$IN[B] = \cup_{P是B的一个前驱} OUT[P]$ $OUT[B] = gen_B \cup (IN[B] - kill_B)$
$IN[EXIT] = \Phi$ $IN[B] = \Phi$	$OUT[B] = \cup_{S是B的一个后继} IN[S]$ $IN[B] = use_B \cup (OUT[B] - def_B)$
$OUT[ENTRY] = \Phi;$ $OUT[B] = U;$	$IN[B] = \cap_{P是B的一个前驱} OUT[P]$ $OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$



数据流分析的主要应用

- 到达定值 ∃ 正向
- 活跃变量 ∃ 逆向
- 可用表达式 ∀ 正向
- 支配结点 ∀

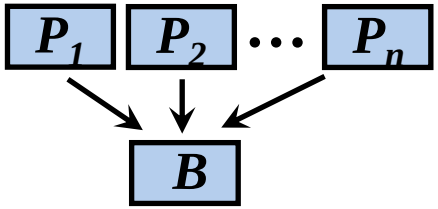
$OUT[ENTRY] = \Phi$ $OUT[B] = \Phi$	$IN[B] = \cup_{P是B的一个前驱} OUT[P]$ $OUT[B] = gen_B \cup (IN[B] - kill_B)$
$IN[EXIT] = \Phi$ $IN[B] = \Phi$	$OUT[B] = \cup_{S是B的一个后继} IN[S]$ $IN[B] = use_B \cup (OUT[B] - def_B)$
$OUT[ENTRY] = \Phi;$ $OUT[B] = U;$	$IN[B] = \cap_{P是B的一个前驱} OUT[P]$ $OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$

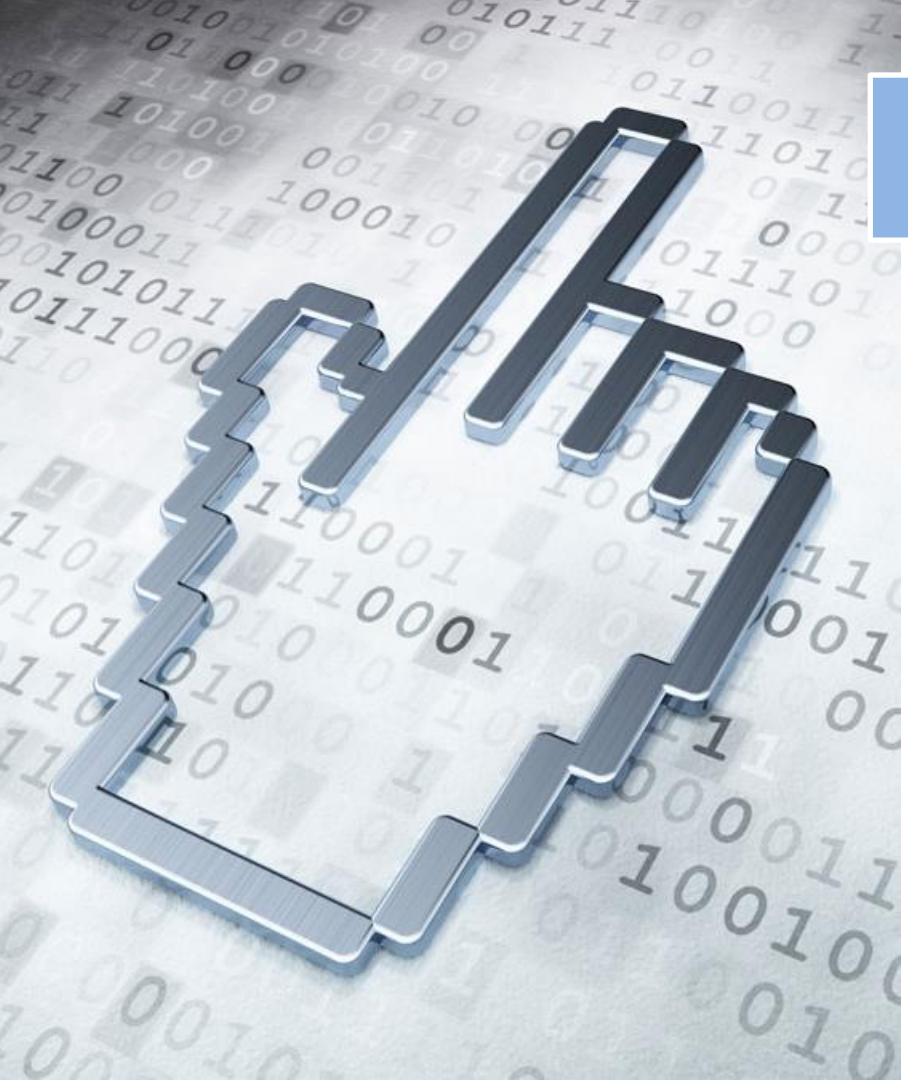
- 如果从流图的入口结点到结点n的每条路径都经过结点d，则称结点d支配(*dominate*)结点n，记为$d\ dom\ n$

数据流分析的主要应用

- 到达定值 ∃ 正向
- 活跃变量 ∃ 逆向
- 可用表达式 ∀ 正向
- 支配结点 ∀ 正向

$OUT[ENTRY] = \Phi$	$IN[B] = \cup_{P是B的一个前驱} OUT[P]$
$OUT[B] = \Phi$	$OUT[B] = gen_B \cup (IN[B] - kill_B)$
$IN[EXIT] = \Phi$	$OUT[B] = \cup_{S是B的一个后继} IN[S]$
$IN[B] = \Phi$	$IN[B] = use_B \cup (OUT[B] - def_B)$
$OUT[ENTRY] = \Phi;$	$IN[B] = \cap_{P是B的一个前驱} OUT[P]$
$OUT[B] = U;$	$OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$
$OUT[ENTRY] = \{ENTRY\}$	$IN[B] = \cap_{P是B的一个前驱} OUT[P]$
$OUT[B] = N$	$OUT[B] = IN[B] \cup \{B\}$





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

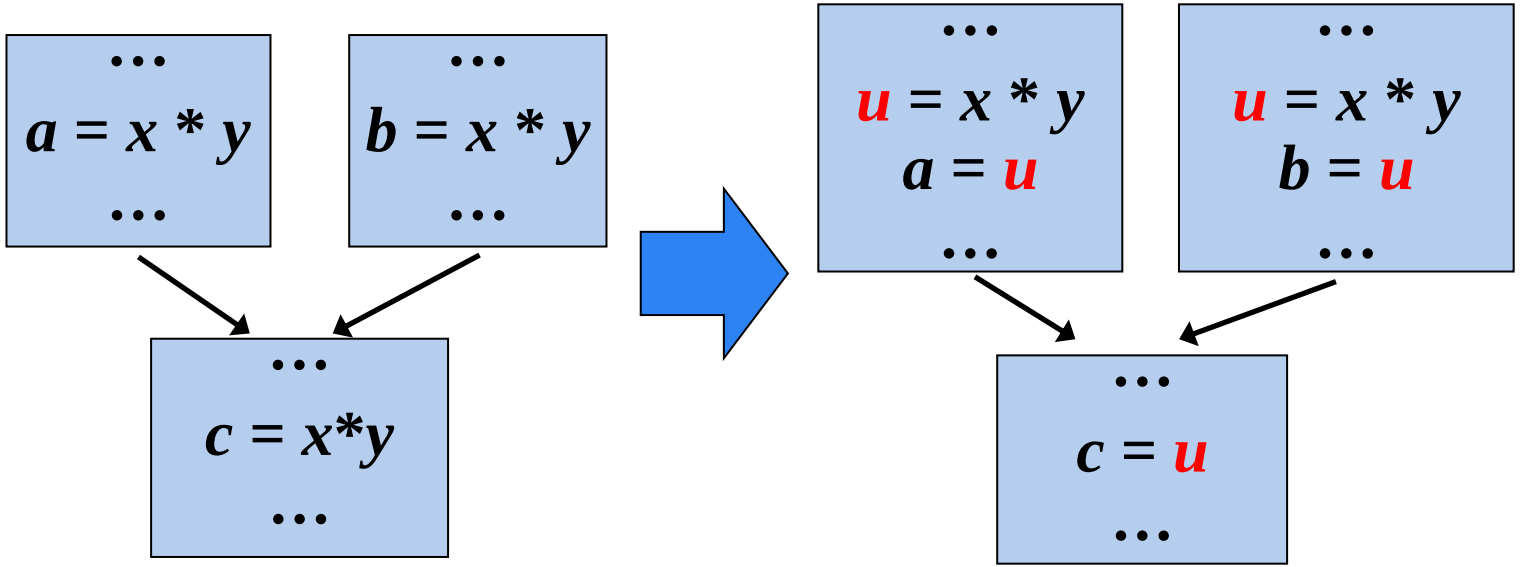
8.6 全局优化

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

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：$z = x\ \mathrm{op}\ y$，如果$x\ \mathrm{op}\ y$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$x\ \mathrm{op}\ y$的块，找到所有离s最近的计算了$x\ \mathrm{op}\ y$的语句

    ② 建立新的临时变量u

    ③ 把步骤①中找到的语句$w = x\ \mathrm{op}\ y$用下列语句代替：

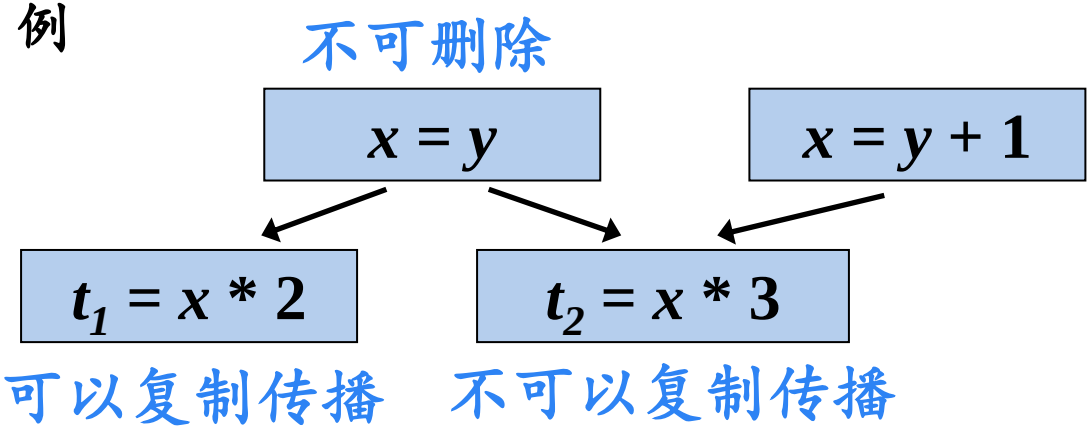
    $$u = x\ \mathrm{op}\ y$$

    $$w = u$$

    ④ 用$z = u$替代s

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



- 在x的引用点u用y代替x (复制传播)的条件
  - 复制语句s: x=y在u点“可用”

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

    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

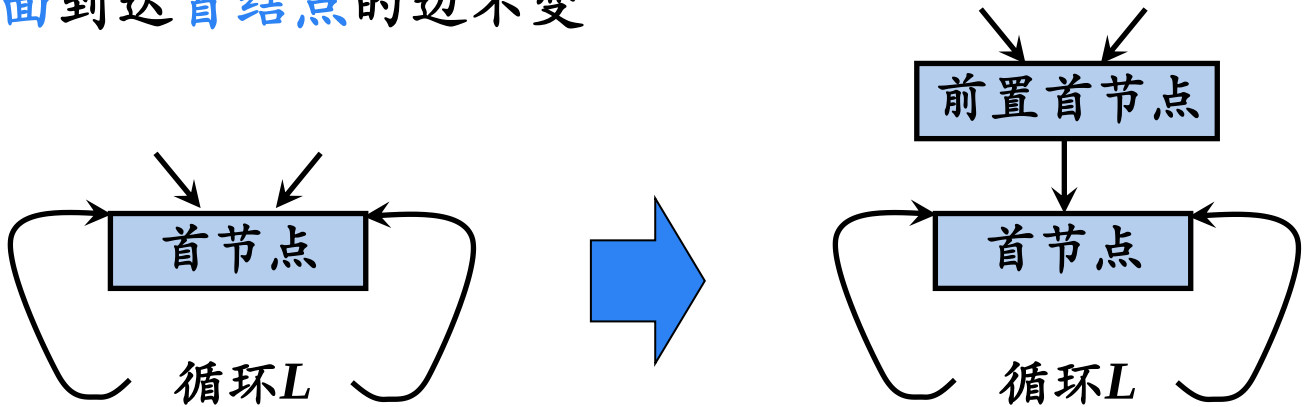
循环不变计算检测算法

- 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

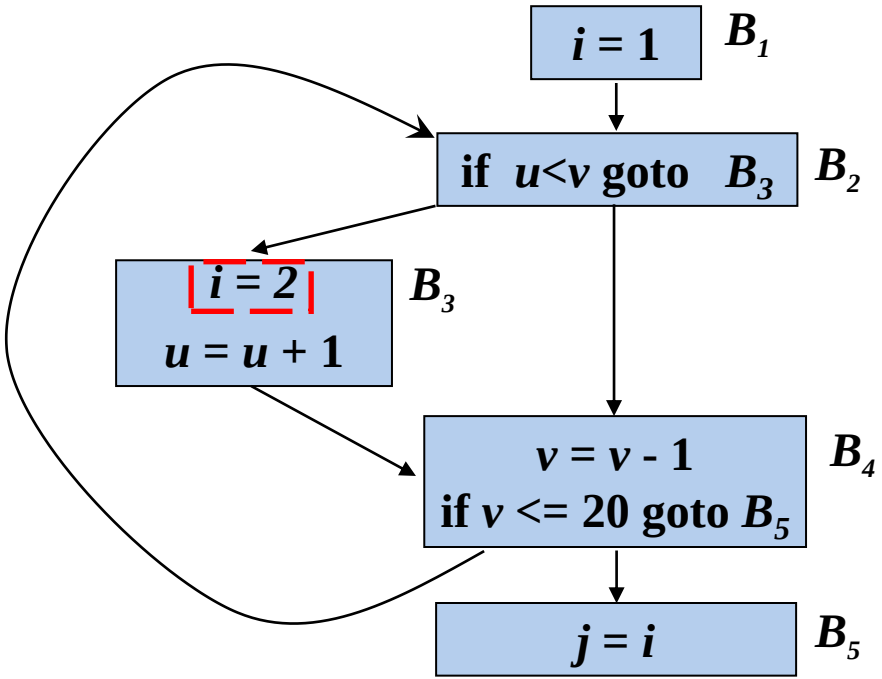
代码外提

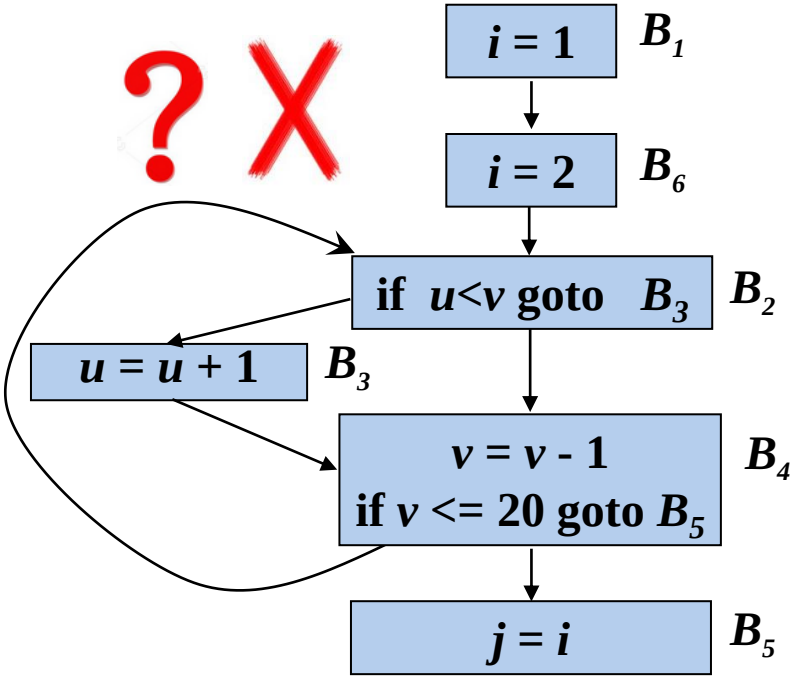
- 前置首结点 (*preheader*)
  - 循环不变计算将被移至首结点之前，为此创建一个称为前置首结点的新块。前置首结点 的唯一后继是首结点，并且原来从循环*L*外到达*L*首结点的边都改成进入前置首结点。 从循环*L*里面到达首结点的边不变



循环不变计算语句 $s: x = y + z$ 移动的条件

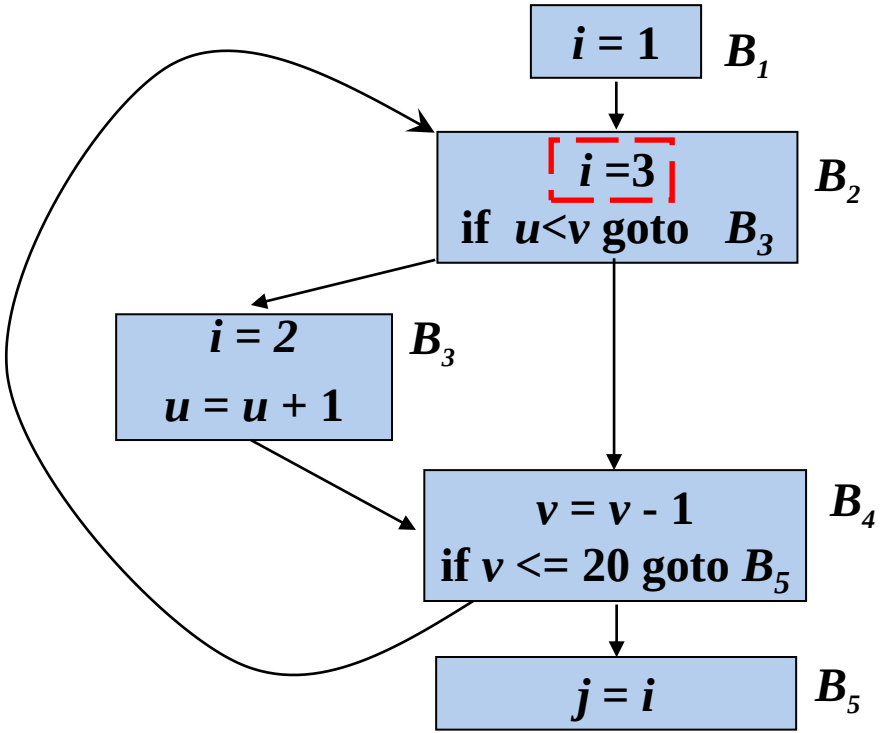
(1) s所在的基本块是循环所有出口结点(有后继结点在循环外的结点)的支配结点





循环不变计算语句 $s: x = y + z$ 移动的条件

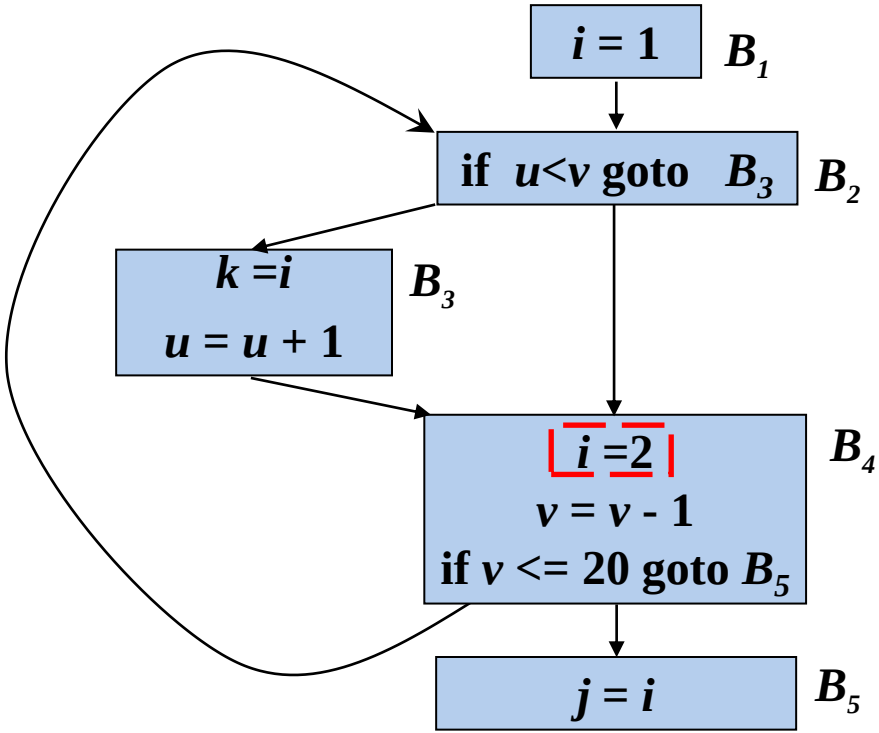
(2) 循环中没有其它语句对x赋值



- 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



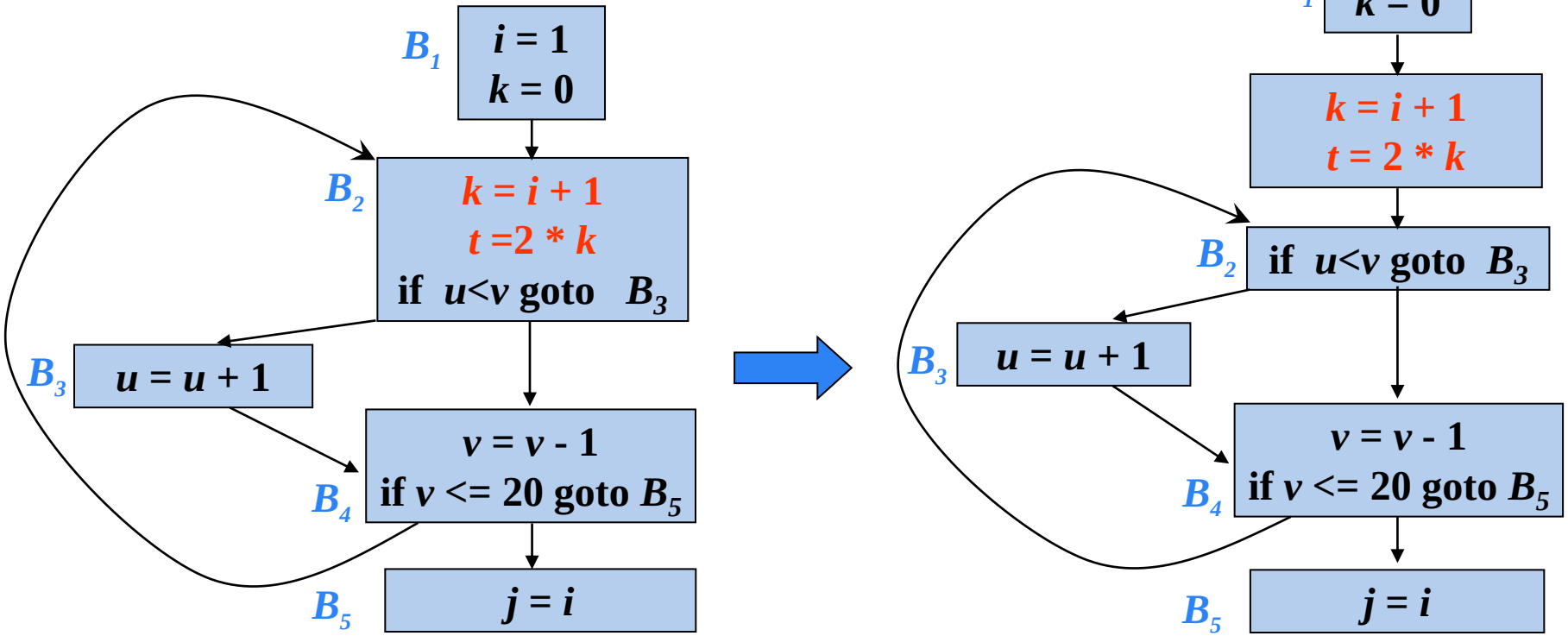
- 外提前
  - k的值有可能等于1，也可能等于2
- 外提后
  - k的值只能等于2

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

例


B1
i = 1
k = 0
B2
k = i + 1
t =2 * k
if u<v goto B3
B3
u = u + 1
B4
v = v - 1
if v <= 20 goto B5
B5
j = i
B1
i = 1
k = 0
k = i + 1
t = 2 * k
B2
if u<v goto B3
B3
u = u + 1
B4
v = v - 1
if v <= 20 goto B5
B5
j = i

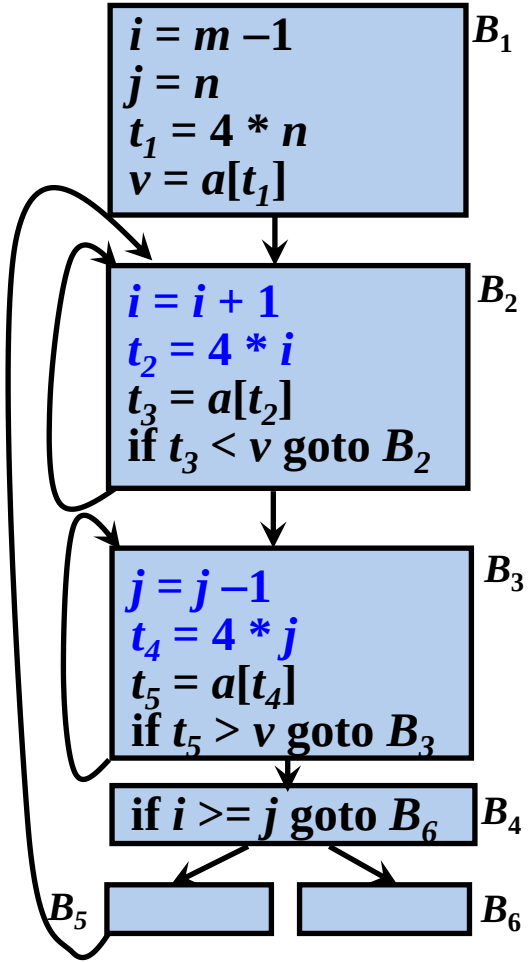
④ 作用于归纳变量的强度削弱

- 对于一个变量x，如果存在一个正的或负的常量c，使得每次x被赋值时，它的值总是增加c，则称x为归纳变量
- 如果循环L中的变量i只有形如$i=i+c$的定值(c是常量)，则称i为循环L的基本归纳变量
- 如果$j=c\times i+d$，其中i是基本归纳变量，c和d是常量，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

④ 作用于归纳变量的强度削弱

- 对于一个变量x，如果存在一个正的或负的常量c，使得每次x被赋值时，它的值总是增加c，则称x为归纳变量
- 如果循环L中的变量i只有形如$i=i+c$的定值(c是常量)，则称i为循环L的基本归纳变量
- 如果$j=c\times i+d$，其中i是基本归纳变量，c和d是常量，则j也是一个归纳变量，称j属于i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



i是基本归纳变量

t_2是i族归纳变量，可以表示为$(i, 4, 0)$

j是基本归纳变量

t_4是j族归纳变量，可以表示为$(j, 4, 0)$

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：

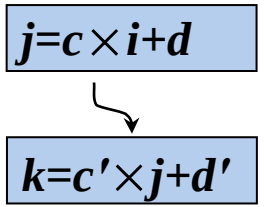
1. 扫描L的语句，找出所有基本归纳变量。在此要用到循环不变计算信息。与每个基本归纳变量i相关联的三元组是$(i, 1, 0)$

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，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
- 如果j不是基本归纳变量，假设其属于i族，k的三元组可以通过j的三元组和k的定值语句来计算，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进行赋值的时候，j当时的值一定等于c*(i当时的值)+d



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

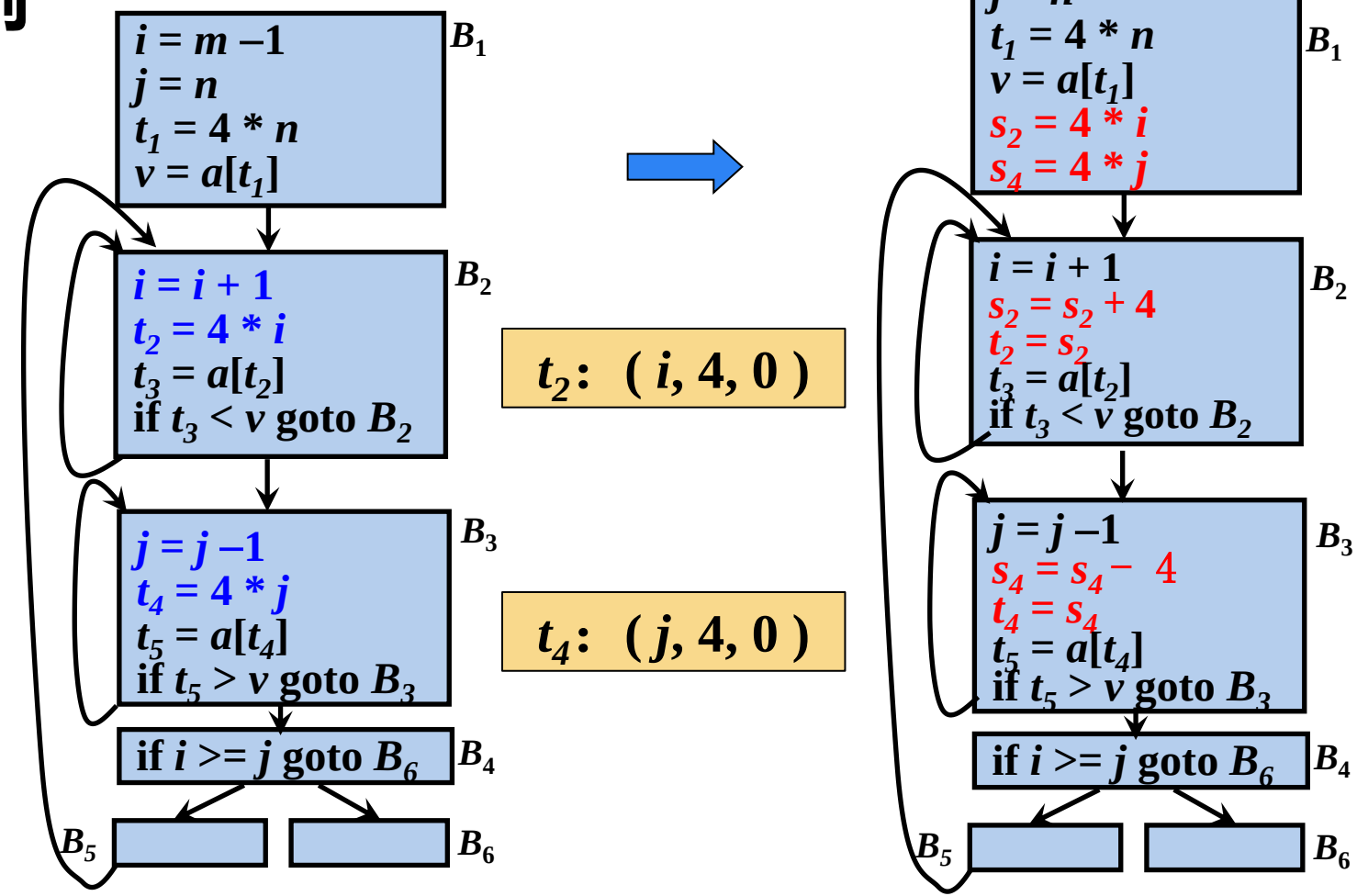
  1.建立新的临时变量t。如果变量j_1和j_2具有相同的三元组，则只为它们建立一个新变量

  2.在前置节点的末尾，添加语句$t=c*i$和$t=t+d$，使得在循环开始的时候$t=c*i+d=j$

  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

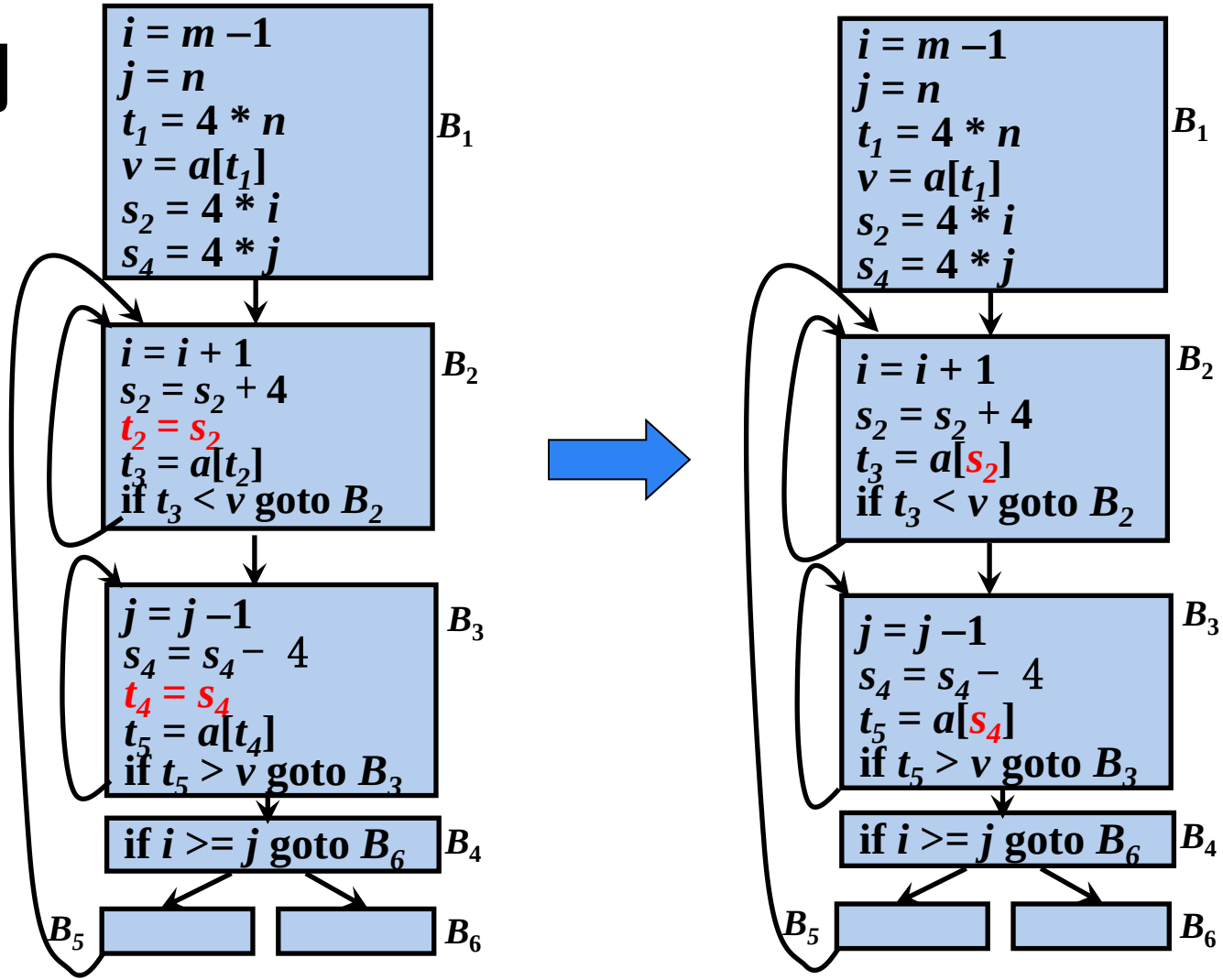
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6
t2: (i, 4, 0)
t4: (j, 4, 0)
i = m –1
j = n
t1 = 4 * n
v = a[t1]
s2 = 4 * i
s4 = 4 * j
B1
i = i + 1
s2 = s2 + 4
t2 = s2
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
s4 = s4 – 4
t4 = s4
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

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$j=t$

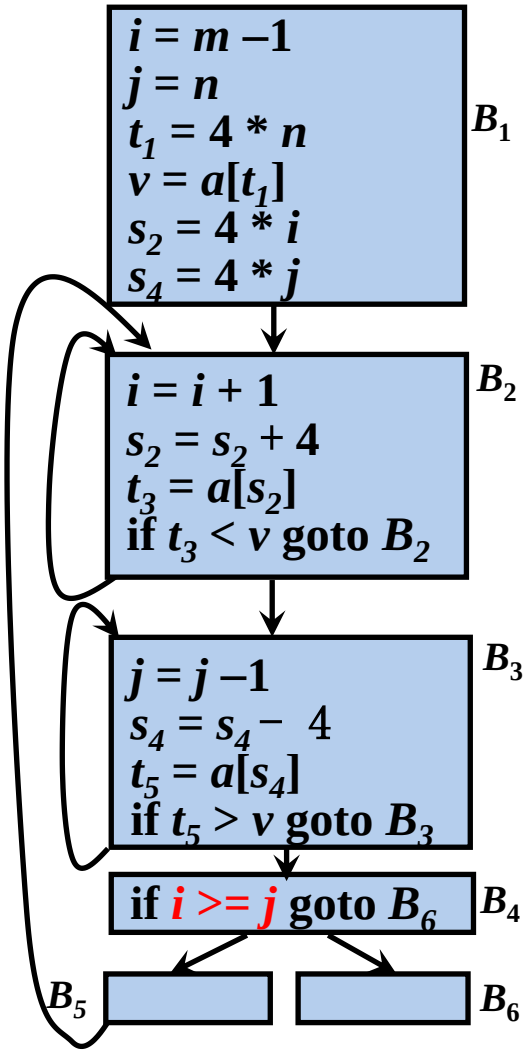
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
s2 = 4 * i
s4 = 4 * j
B1
i = i + 1
s2 = s2 + 4
t2 = s2
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
s4 = s4 – 4
t4 = s4
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6
i = m –1
j = n
t1 = 4 * n
v = a[t1]
s2 = 4 * i
s4 = 4 * j
B1
i = i + 1
s2 = s2 + 4
t3 = a[s2]
if t3 < v goto B2
B2
j = j –1
s4 = s4 – 4
t5 = a[s4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

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

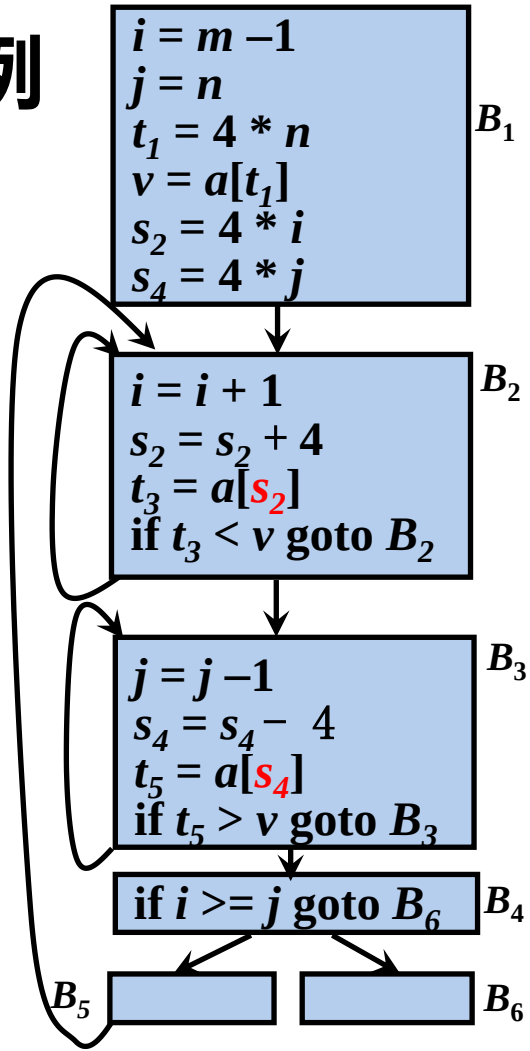
例


i = m –1
j = n
t1 = 4 * n
v = a[t1]
s2 = 4 * i
s4 = 4 * j
B1
i = i + 1
s2 = s2 + 4
t3 = a[s2]
if t3 < v goto B2
B2
j = j –1
s4 = s4 – 4
t5 = a[s4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

删除仅用于测试的归纳变量

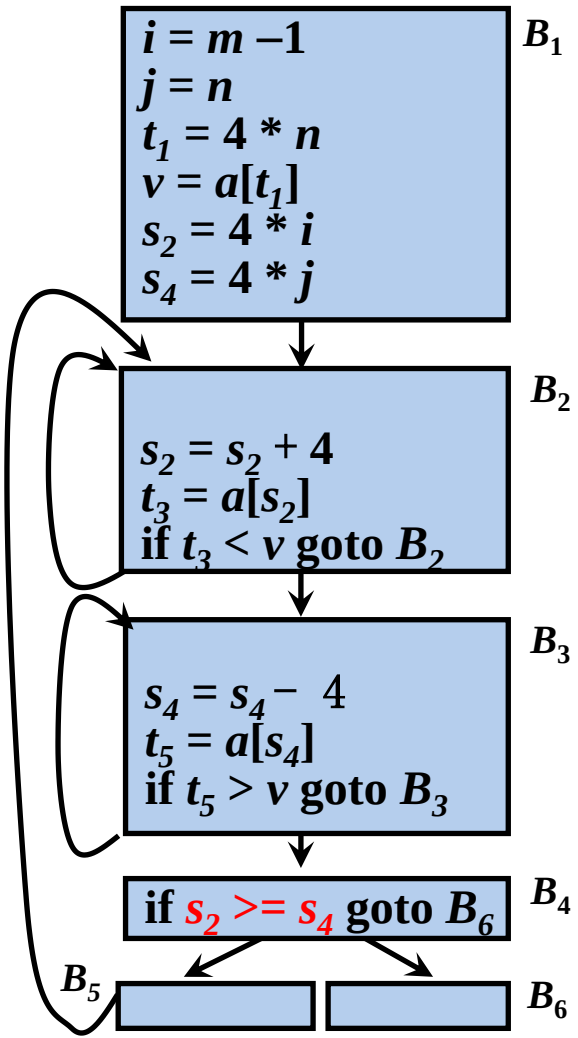
- 对于仅用于测试的基本归纳变量i，取i族的某个归纳变量j(尽量使得c、d简单，即$c=1$或$d=0$的情况)。把每个对i的测试替换成为对j的测试
  - (*relop* i x B)替换为(*relop* j $c*x+d$ B)，其中x不是归纳变量，并假设$c>0$
  - (*relop* i_1 i_2 B)，如果能够找到三元组j_1 (i_1, c, d)和j_2 (i_2, c, d)，那么可以将其替换为(*relop* j_1 j_2 B) (假设$c>0$)。否则，测试的替换可能是没有价值的
- 如果归纳变量i不再被引用，那么可以删除和它相关的指令

例



s_2: (i, 4, 0)

s_4: (j, 4, 0)



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

结束