第十章：代码优化

提纲

10.1 基本块和流图
10.2 常用的代码优化方法
10.3 基本块的优化
10.4 数据流分析

基本块(*Basic Block*)

- 基本块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个指令进入该块。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止

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;
```

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

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*)

- 流图的结点是一些基本块
- 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*)

- 流图的结点是一些基本块
- 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：
    - 有一个从*B*的结尾跳转到*C*的开头的条件或无条件跳转语句
    - 按照原来的三地址语句序列中的顺序，*C*紧跟在之*B*后，且*B*的结尾不存在无条件跳转语句

例

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

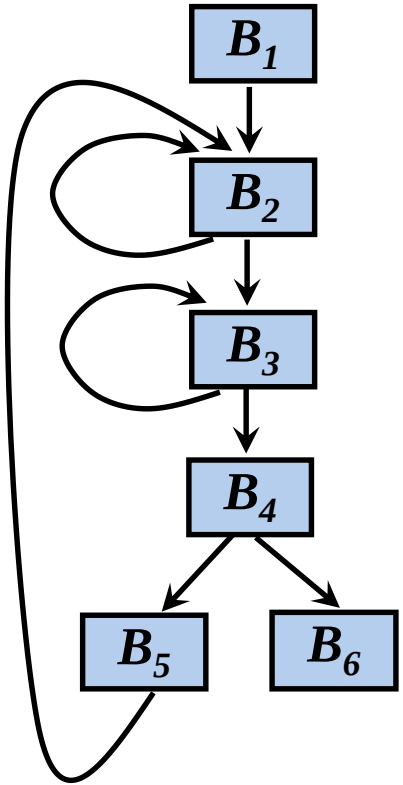
B_2
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) *if* $t_3 > v$ *goto*(5)

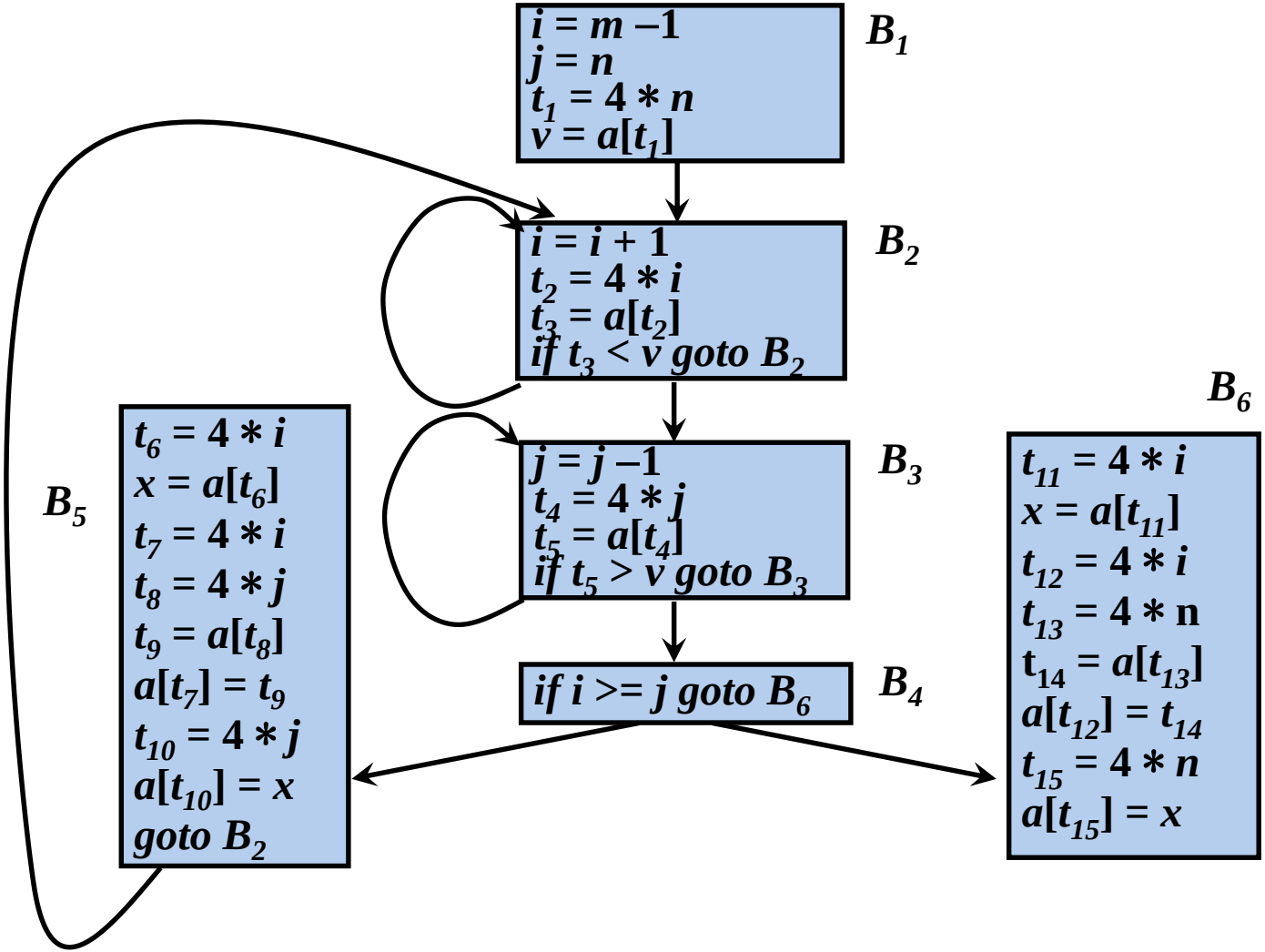
B_3
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) *if* $t_5 > v$ *goto*(9)

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

10.1 基本块和流图
10.2 常用的代码优化方法
10.3 基本块的优化
10.4 数据流分析

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

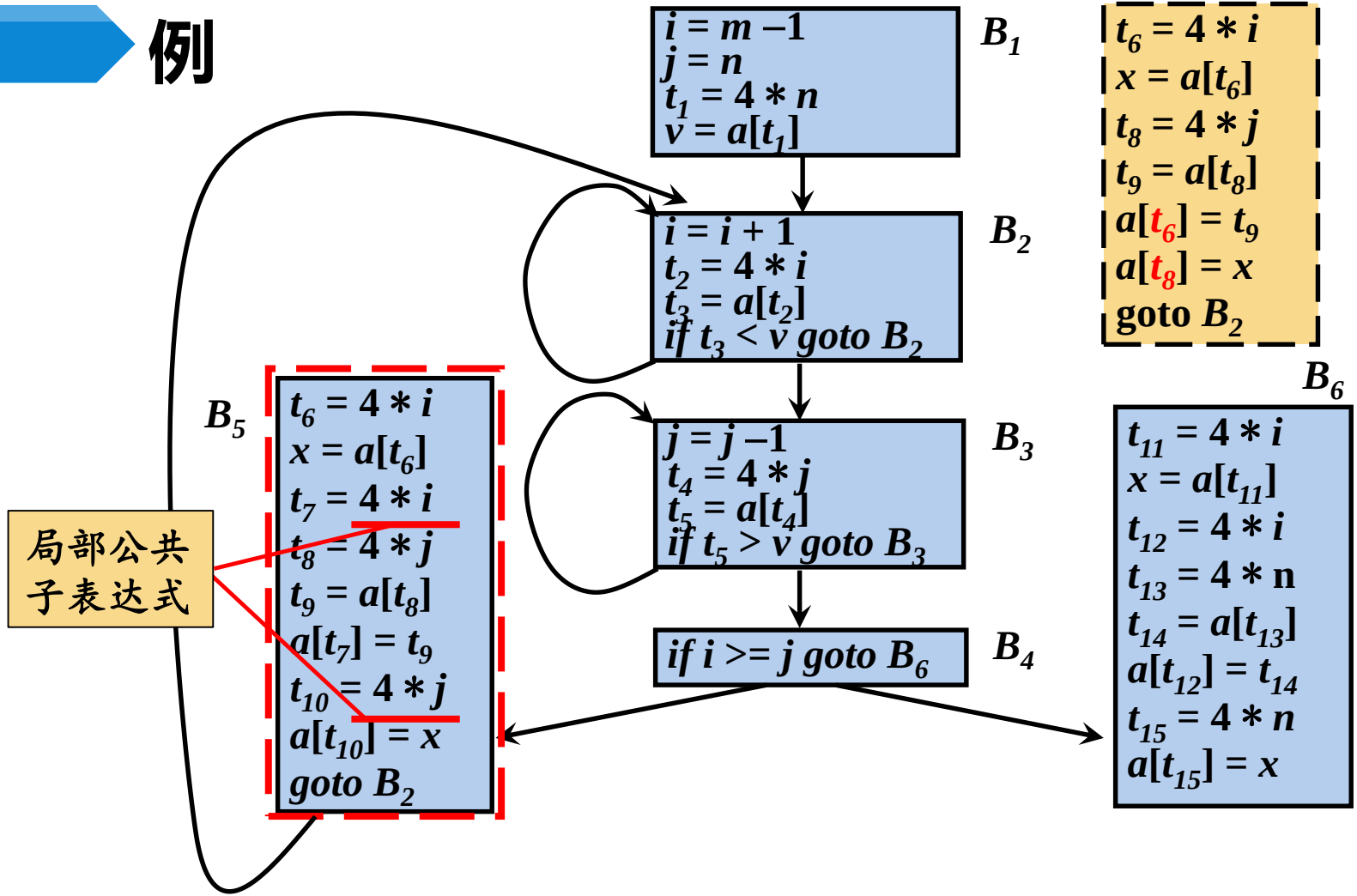
常用的优化方法

- 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

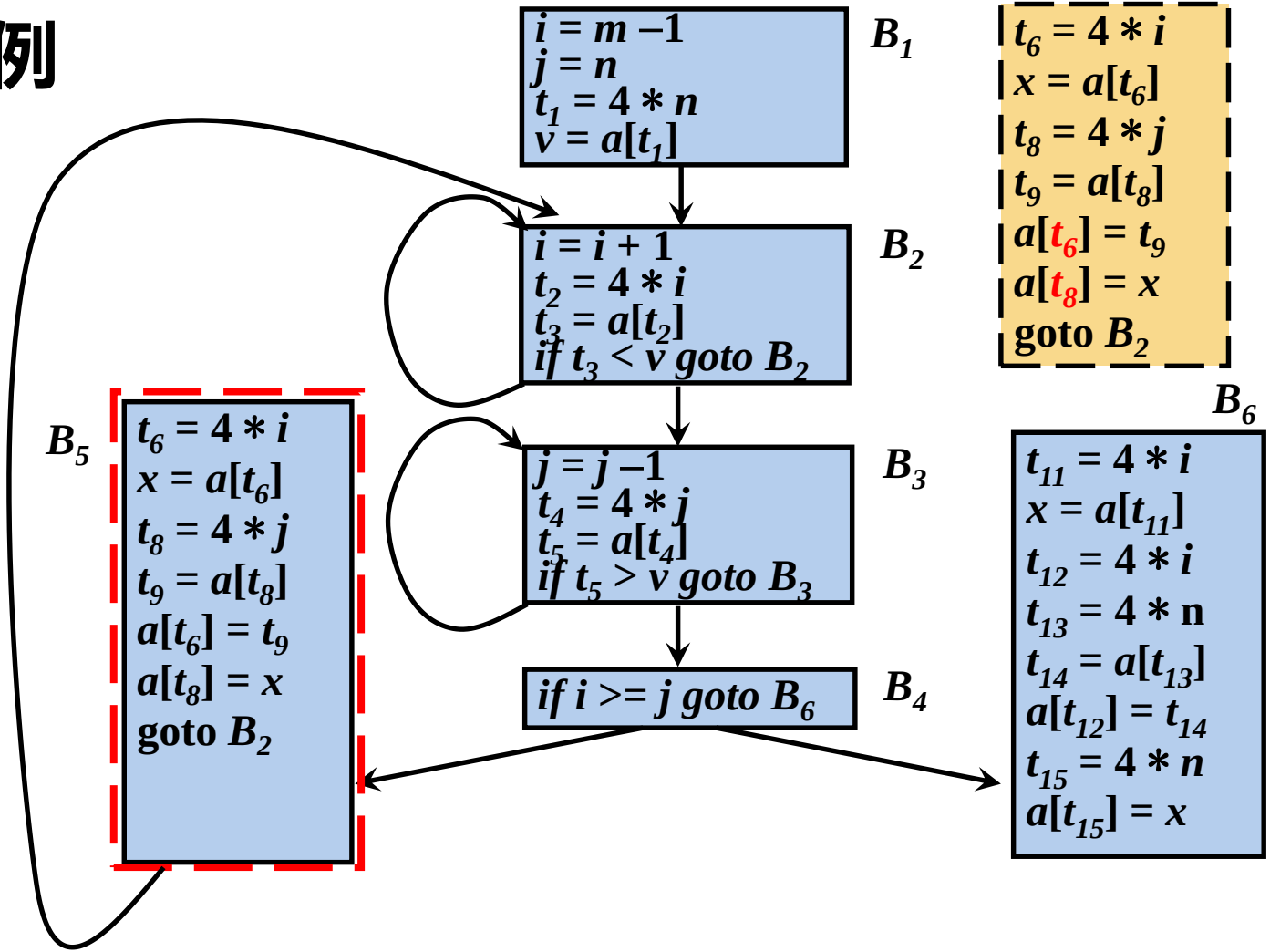
① 删除公共子表达式

- 公共子表达式
  - 如果表达式x op y之前已被计算过，并且从之前的计算到现在，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*)

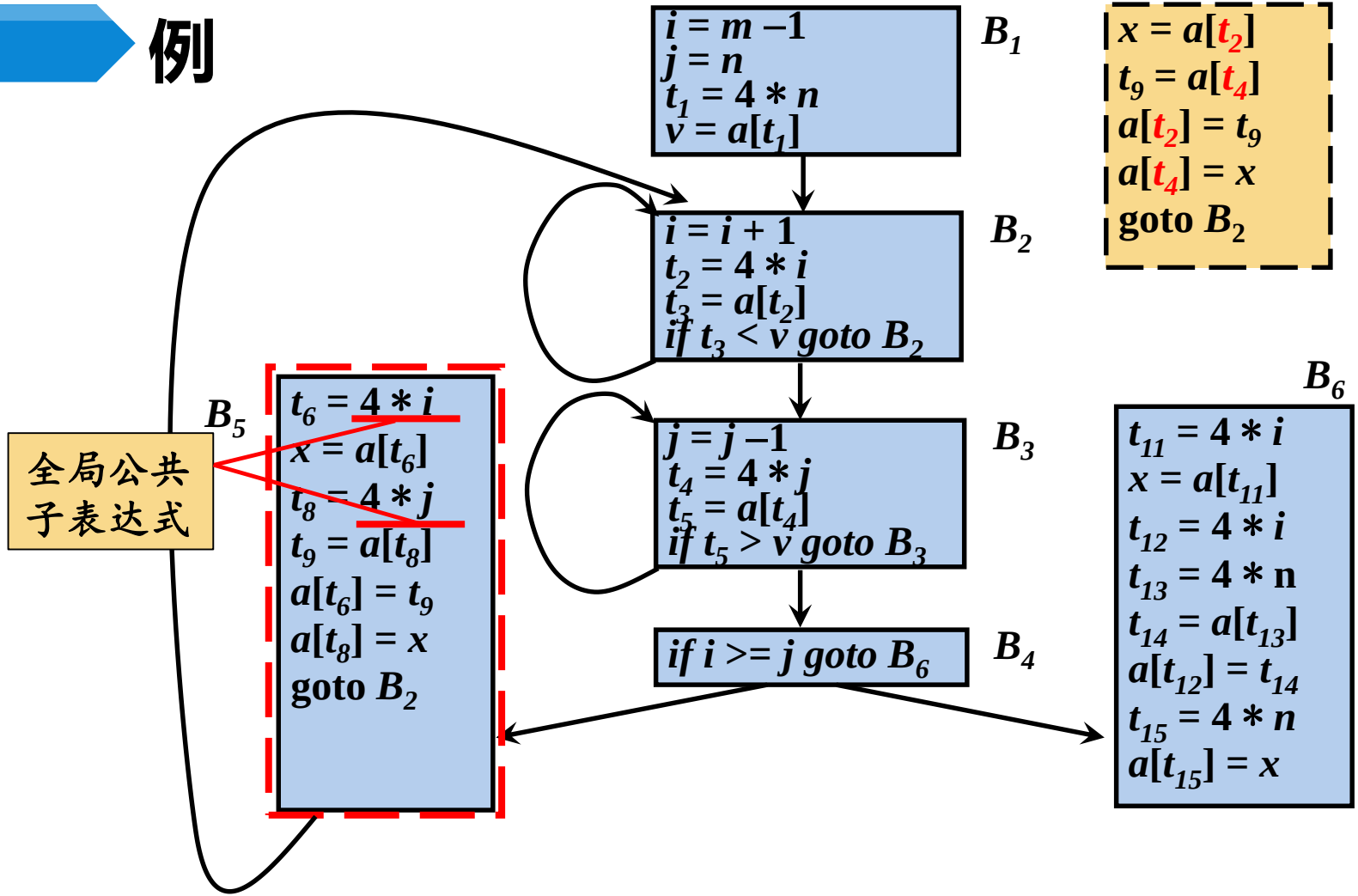
例


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

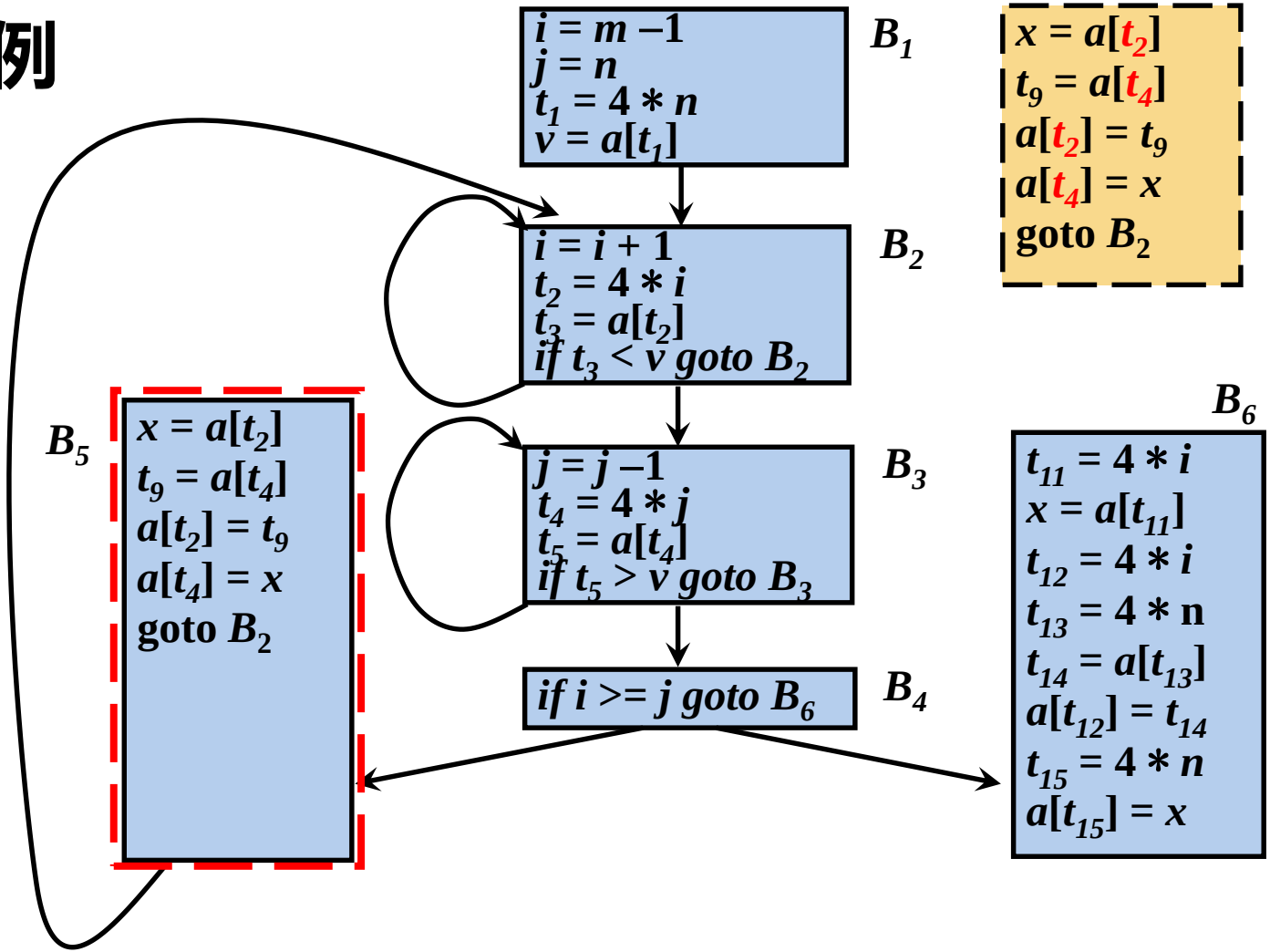
例


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

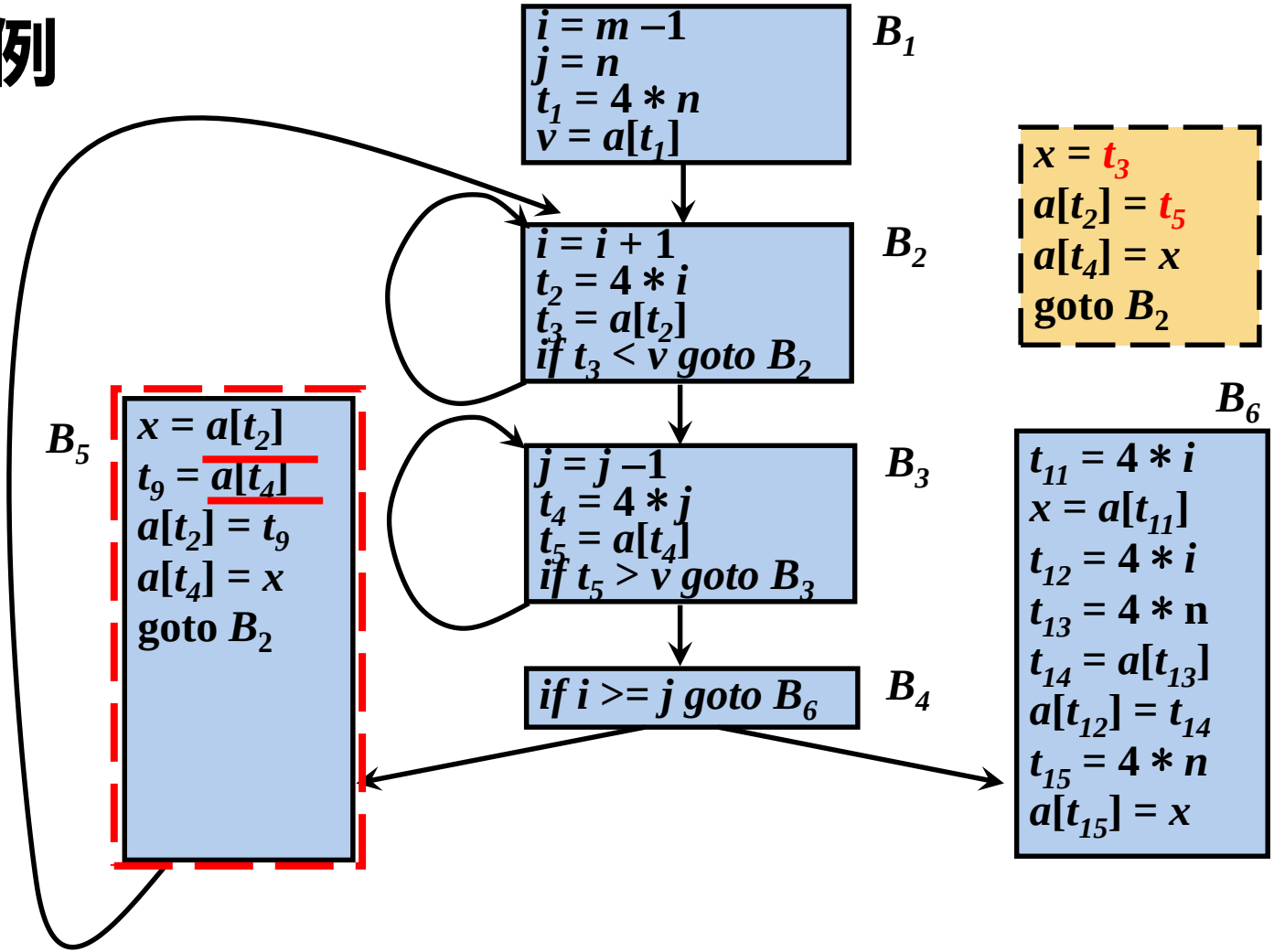
例


i = m –1
j = n
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]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
全局公共
子表达式
x = a[t2]
t9 = a[t4]
a[t2] = t9
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

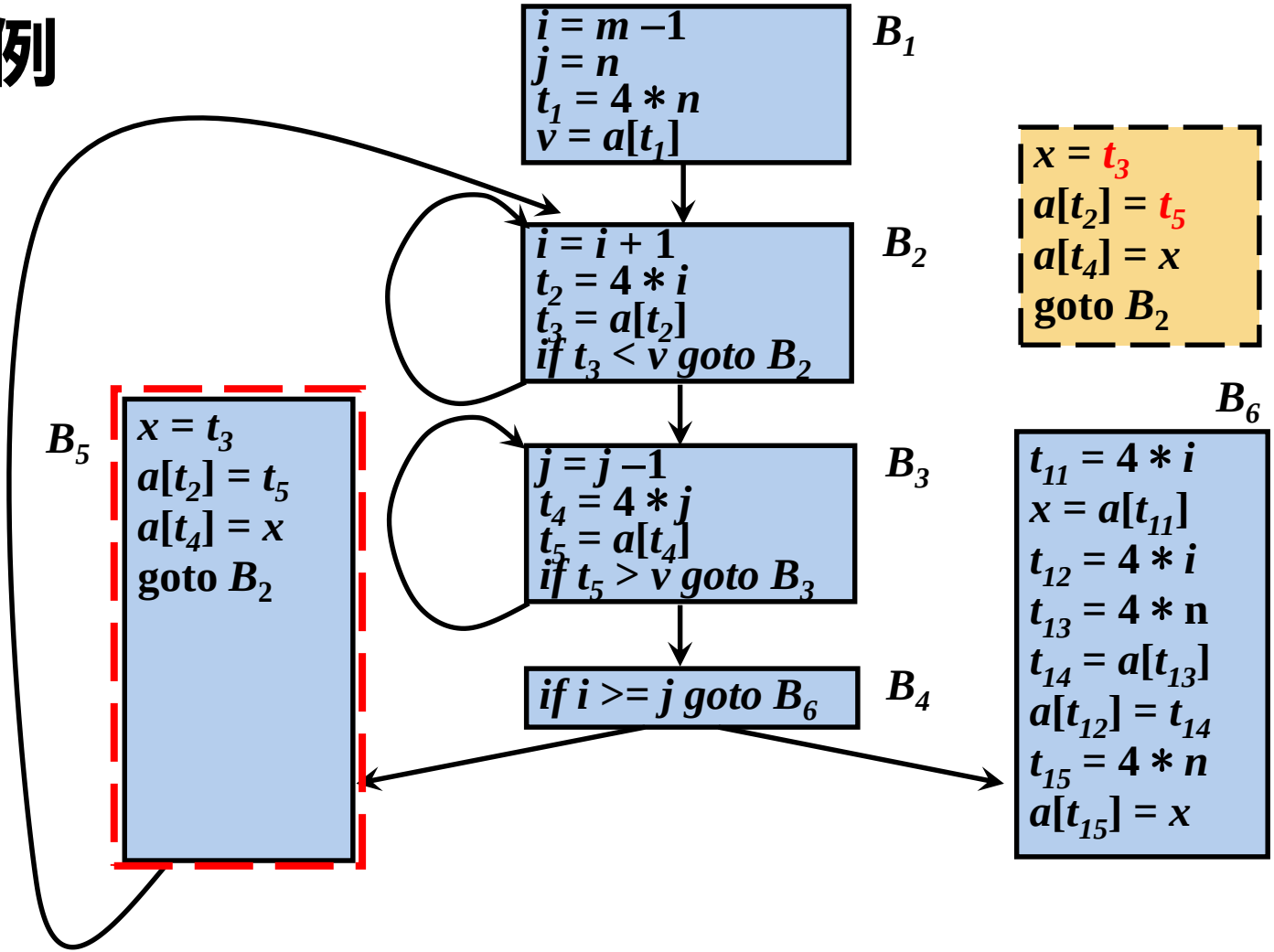
例


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

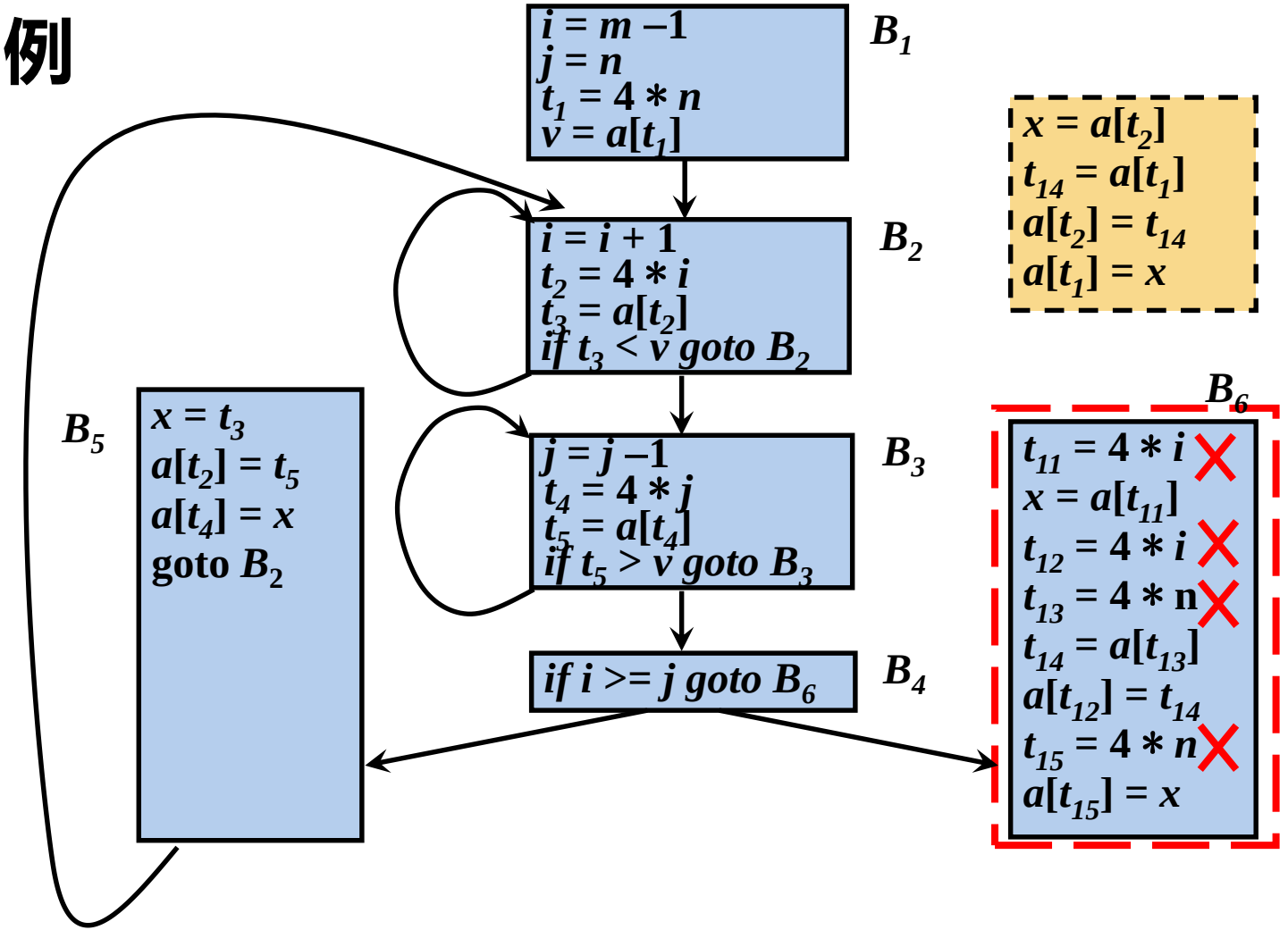
例


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
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
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

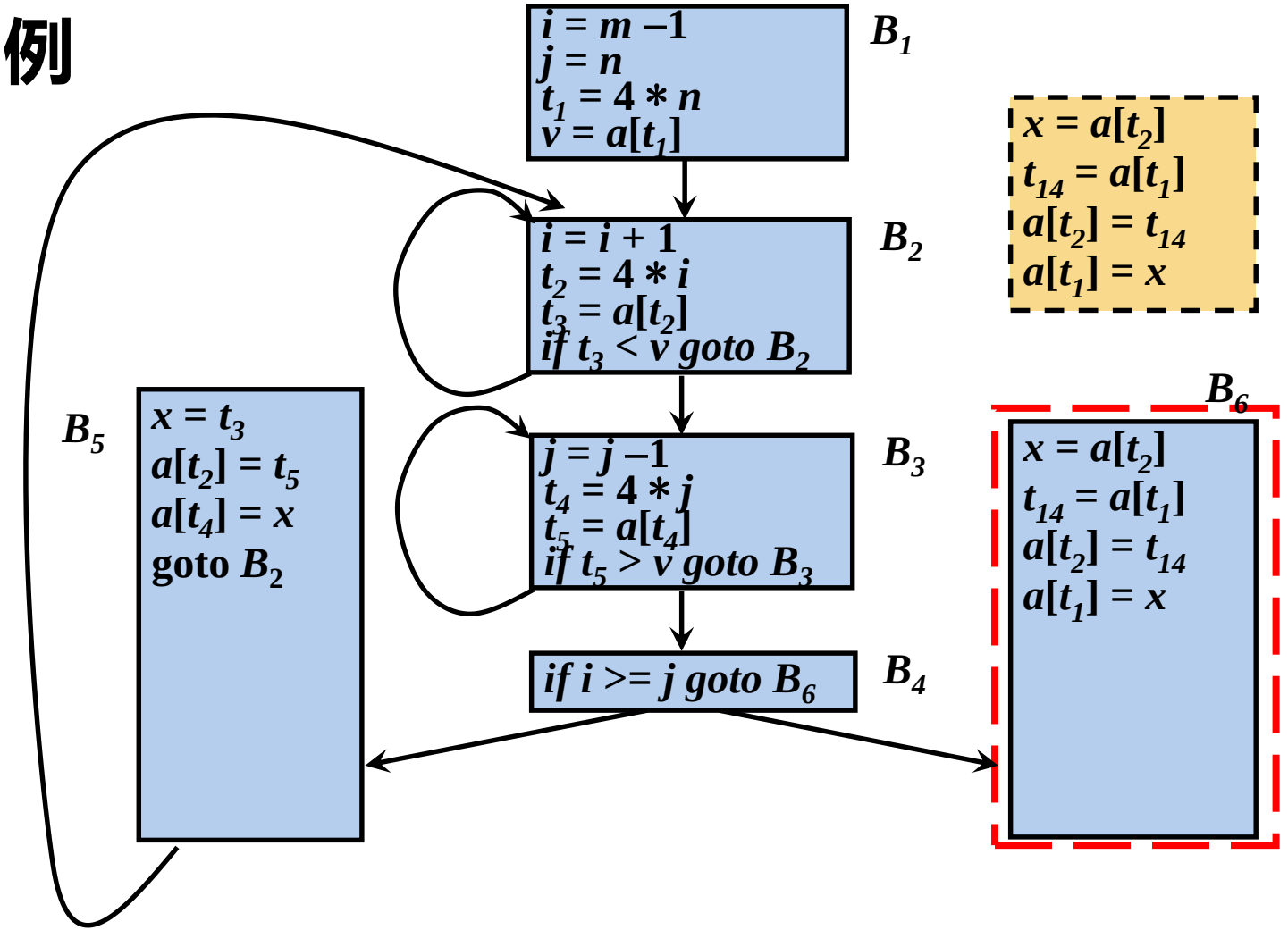
例


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
x = t3
a[t2] = t5
a[t4] = x
goto B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
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

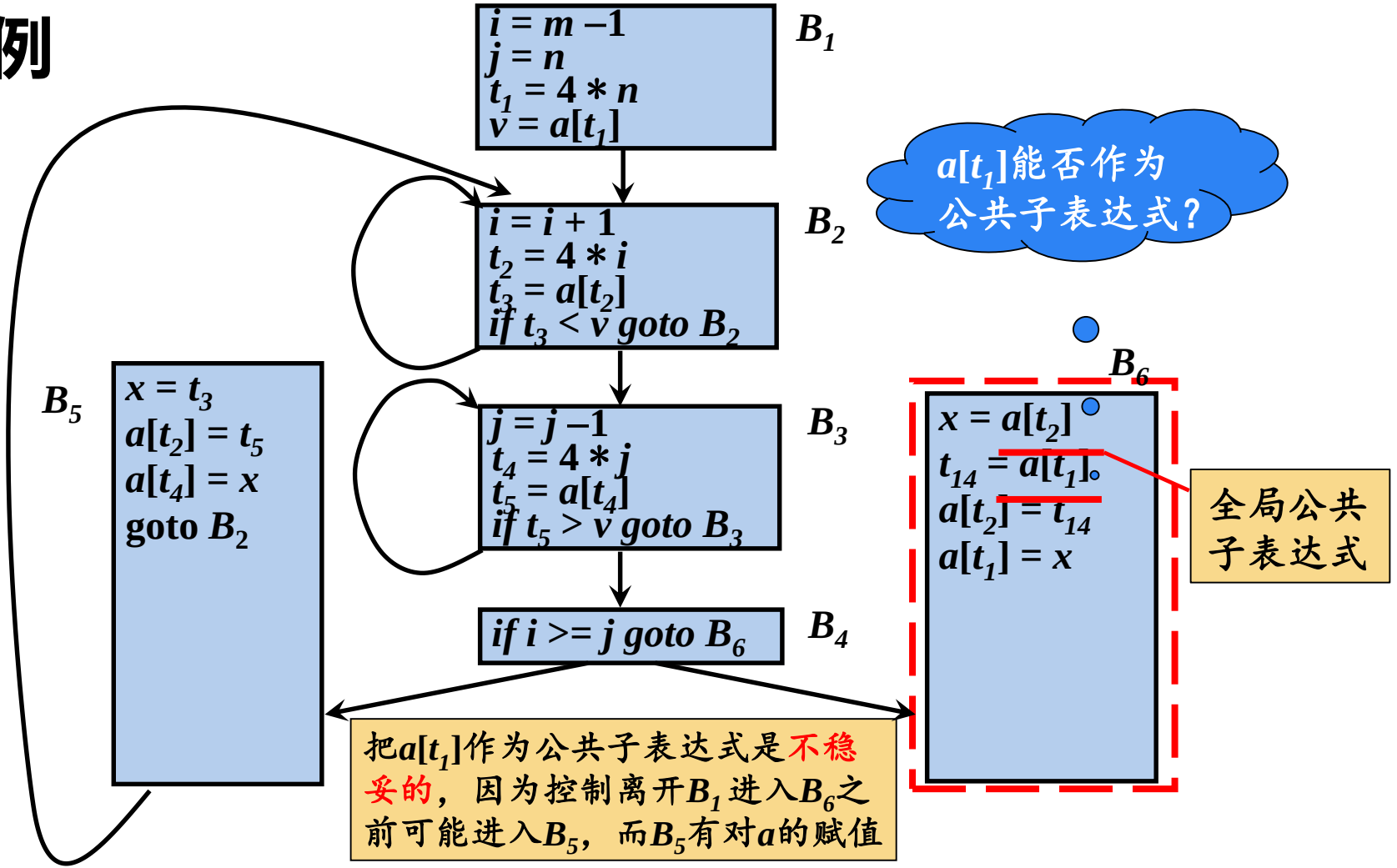
例


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

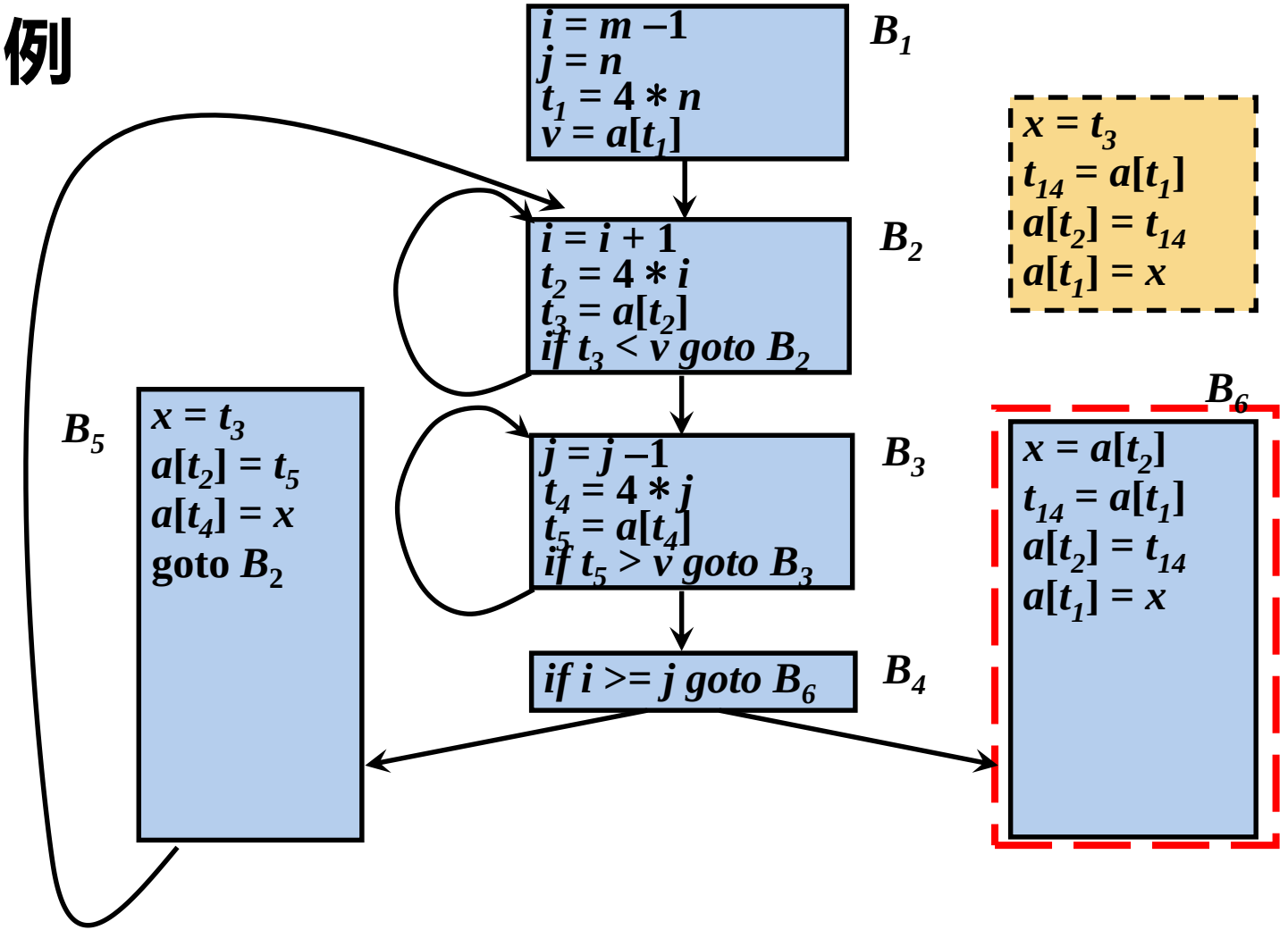
例


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

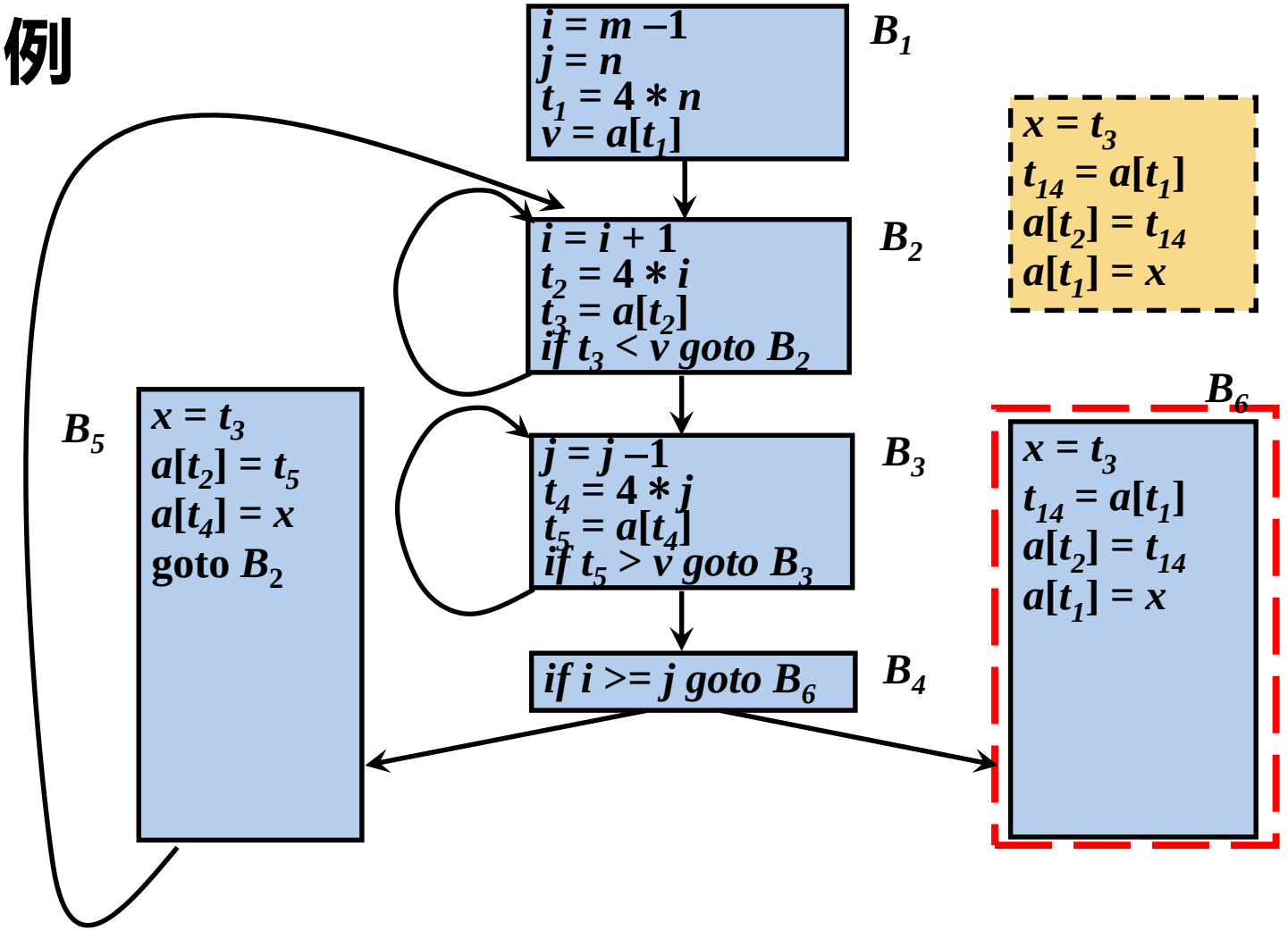
例


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
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
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

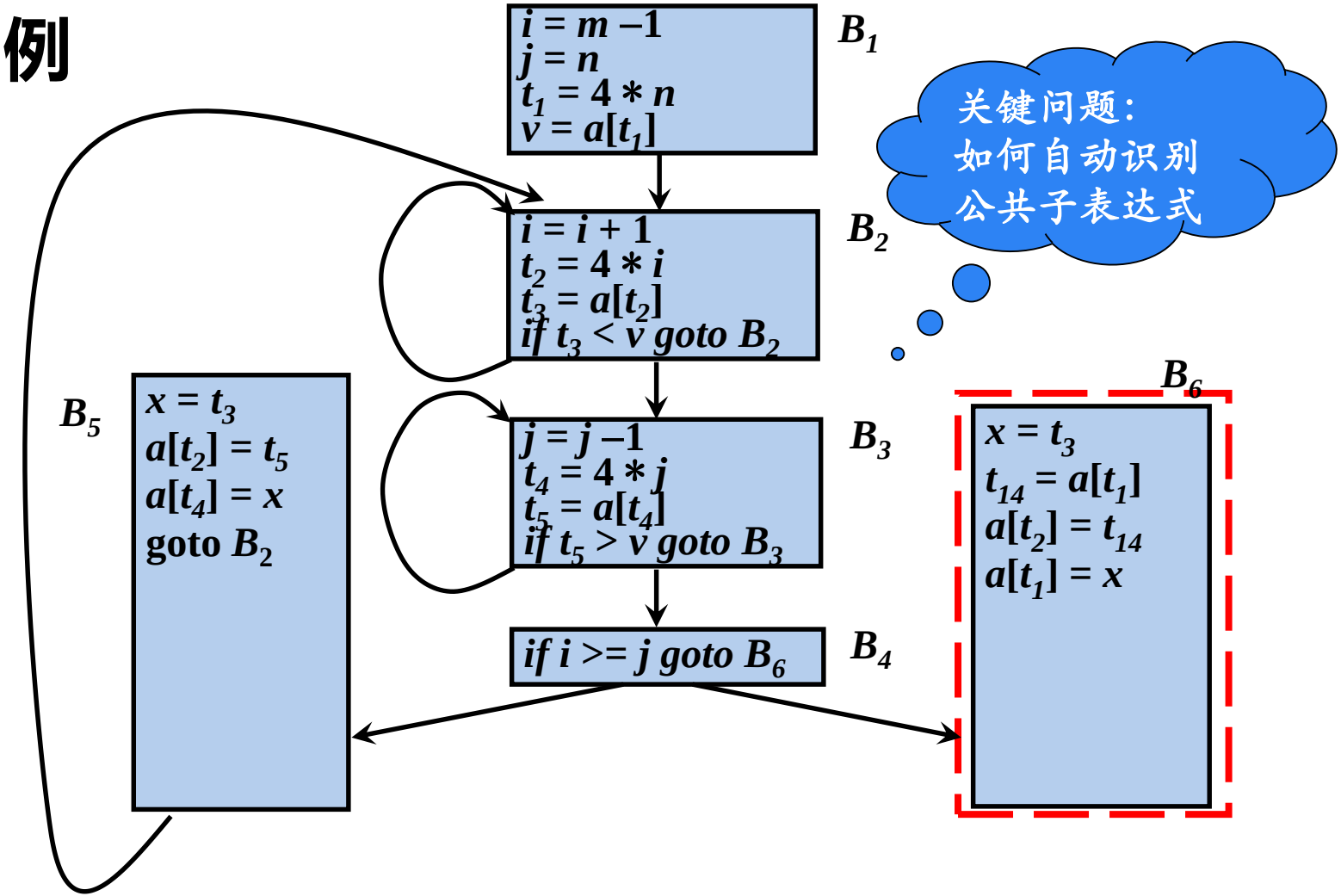
例


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
x = a[t2]
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6

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

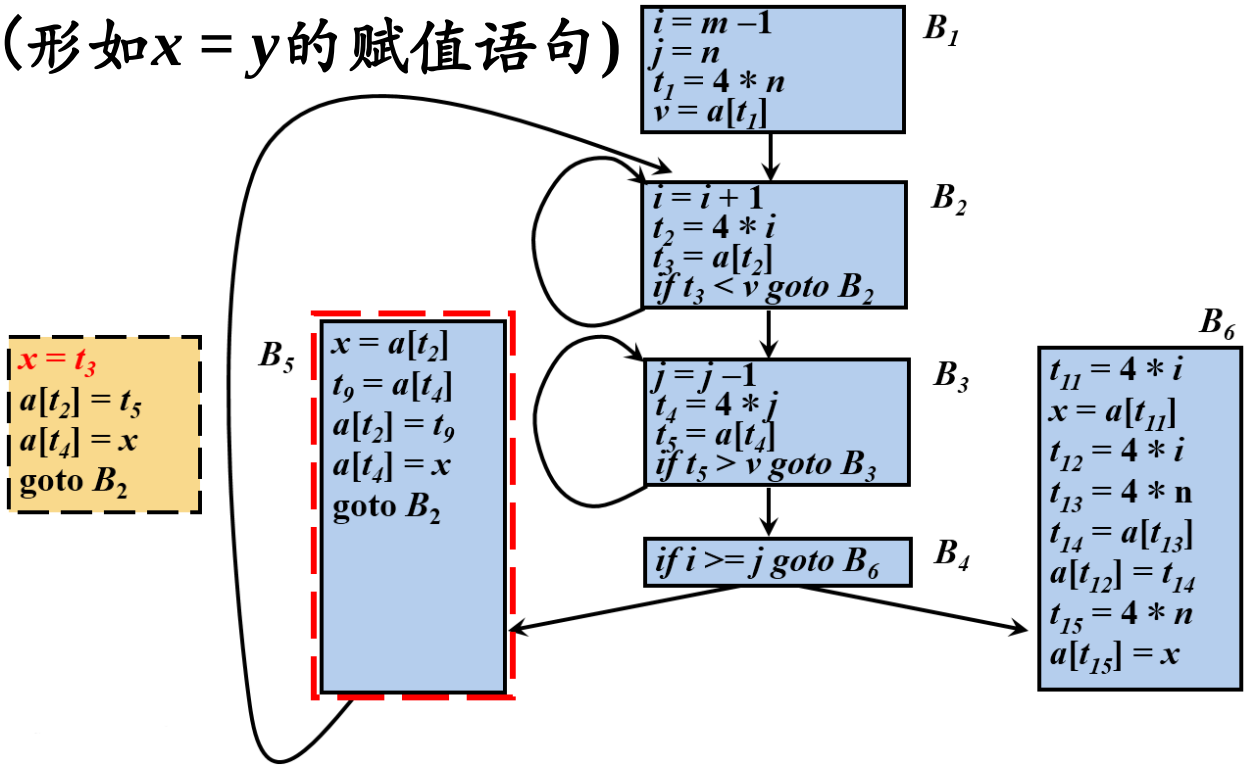

i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
关键问题：
如何自动识别
公共子表达式

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

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)

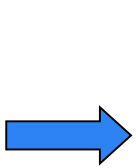


② 删除无用代码

- 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

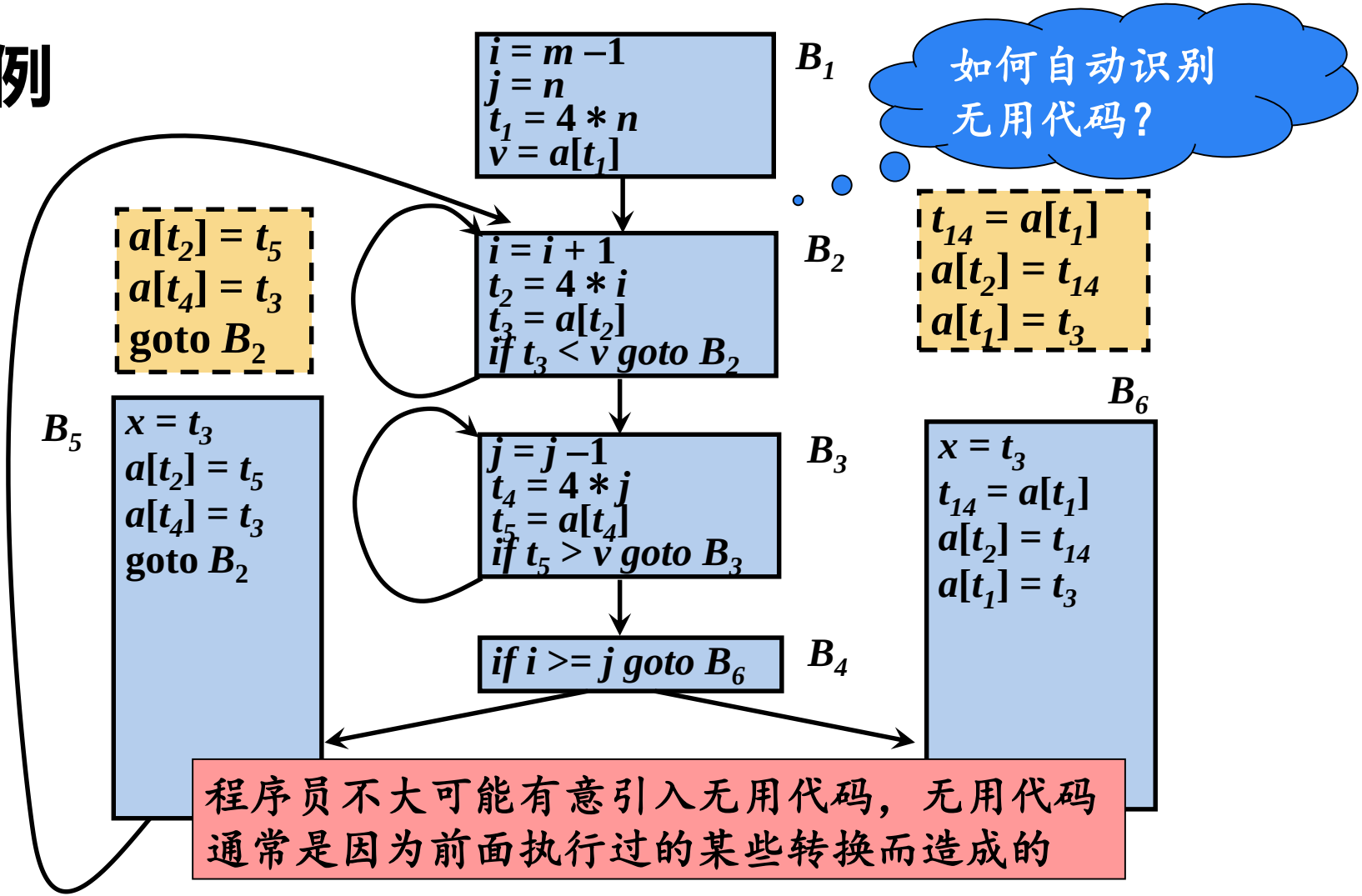
$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

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

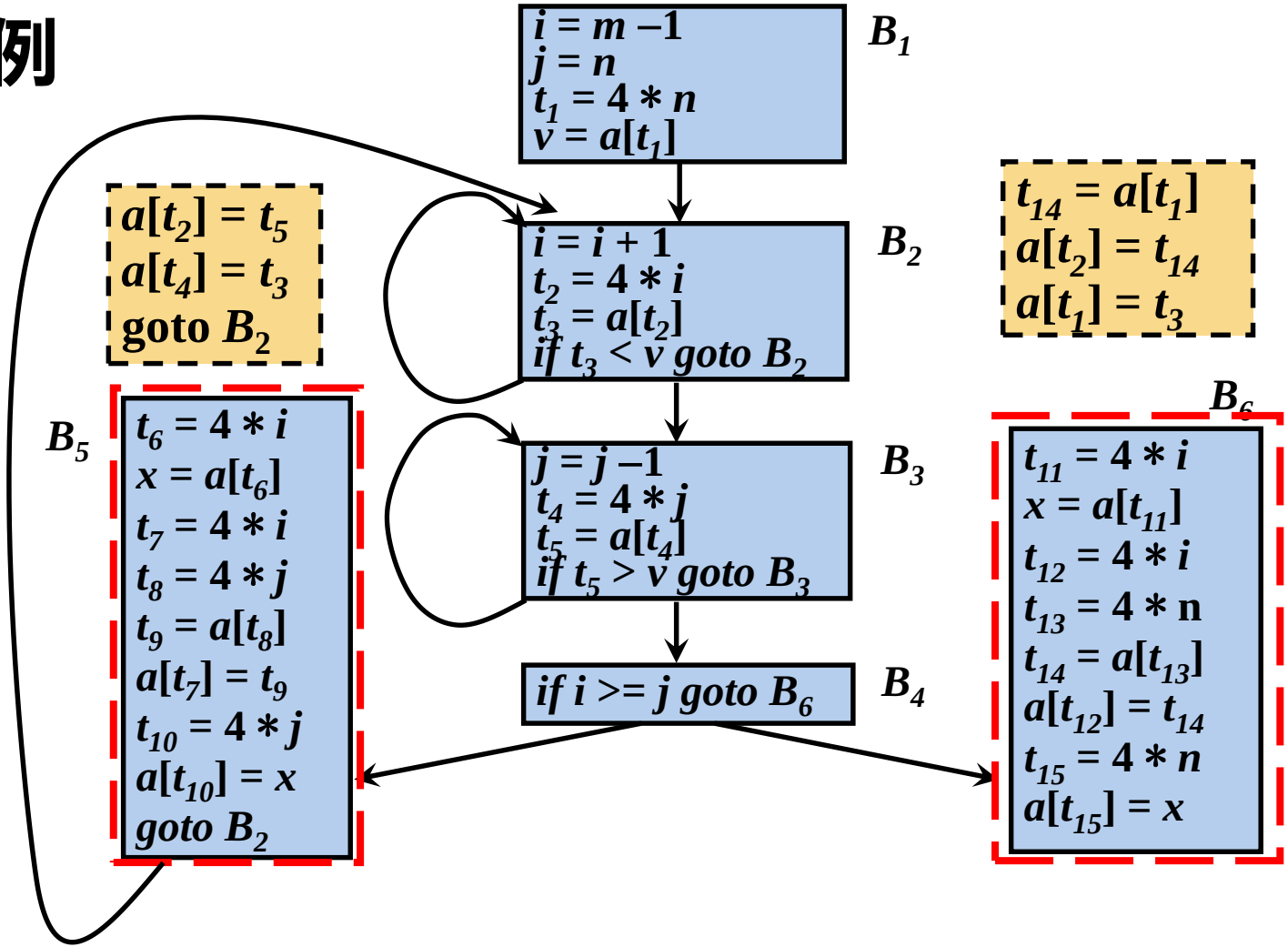
② 删除无用代码

- 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
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = t_3$
B_5
$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3
B_3
B_6
$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = t_3$
if $i >= j$ goto B_6
B_4
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

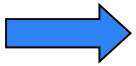
常用的优化方法

- 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

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
  - 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

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 );
}
```

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

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

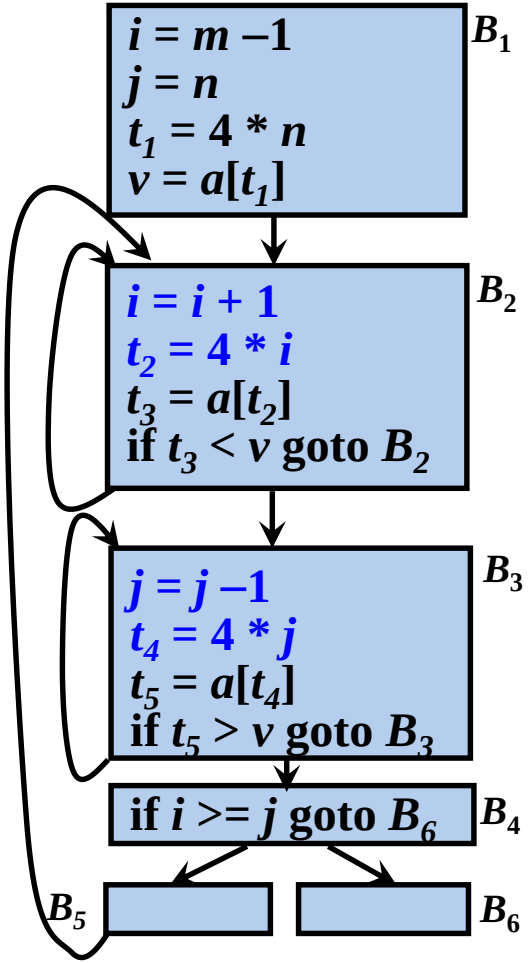
⑤ 强度削弱(*Strength Reduction*)

- 强度削弱
  - 用较快的操作代替较慢的操作，如用加代替乘
  - 例
    - $2*x$ 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

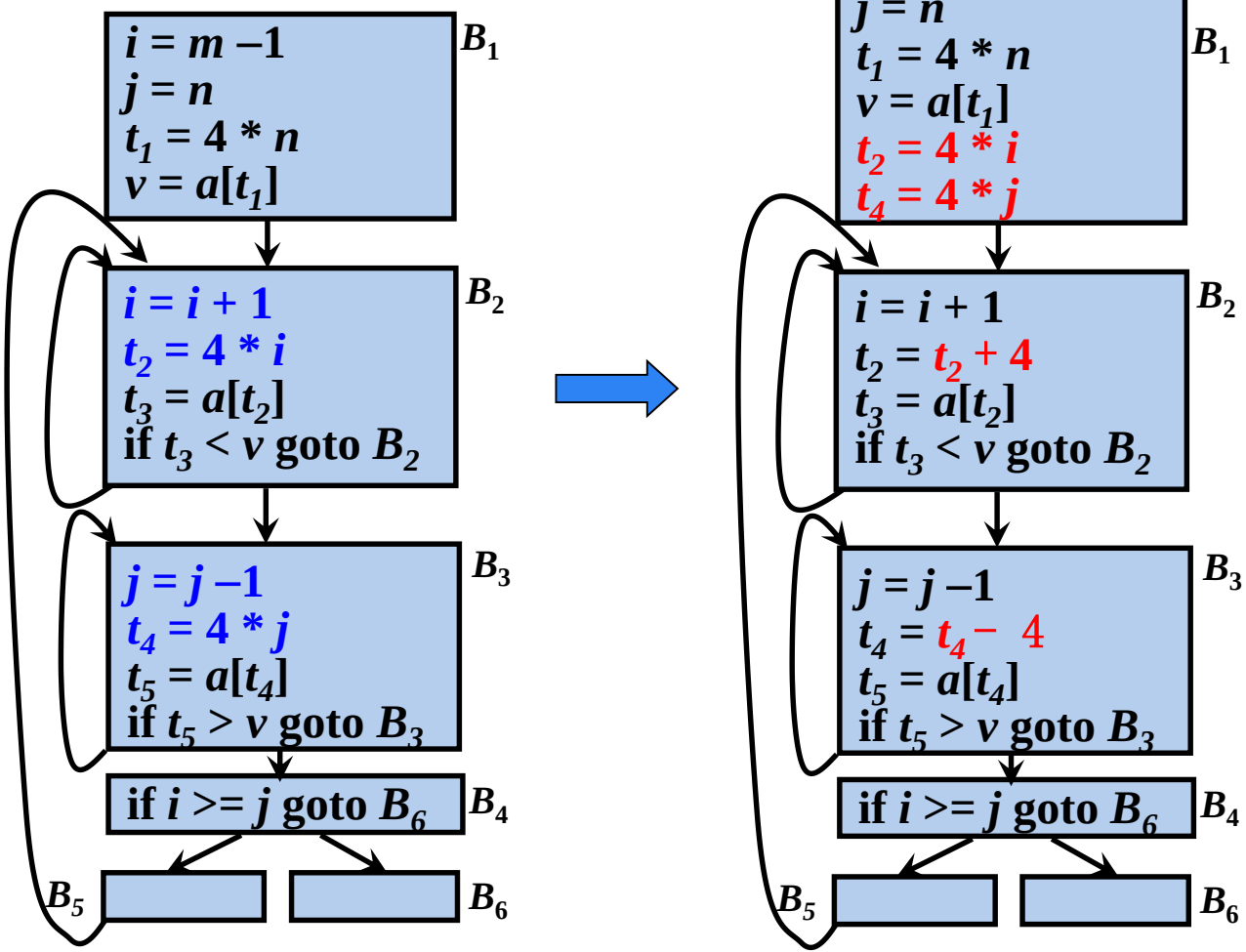
- 归纳变量
  - 对于一个变量x，如果存在一个正的或负的常数c使得每次x被赋值时它的值总增加c，那么x就称为归纳变量(*Induction Variable*)

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

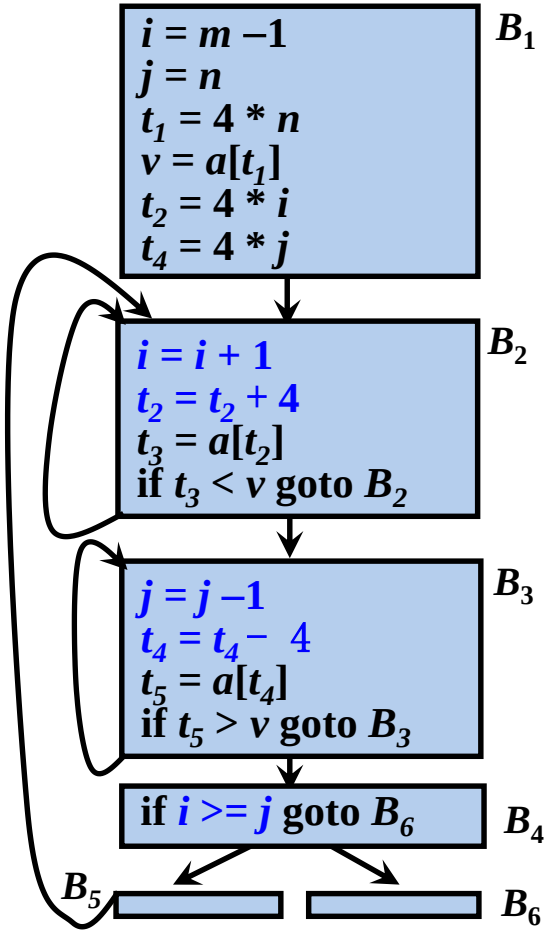
例


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

常用的优化方法

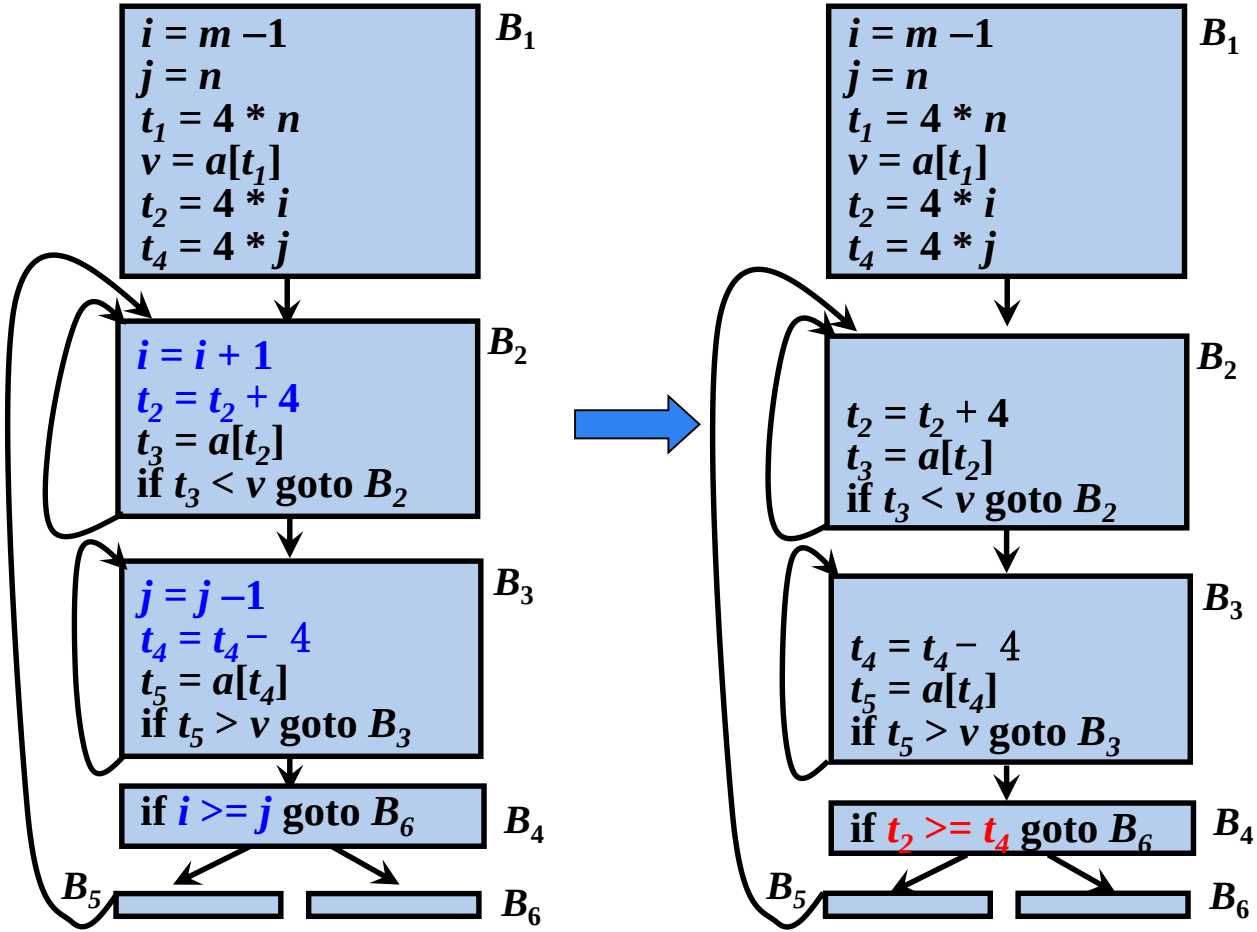
- 删除公共子表达式
- 删除无用代码
- 常量合并
- 代码移动
- 强度削弱
- 删除归纳变量

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

⑥ 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

10.1 基本块和流图
10.2 常用的代码优化方法
10.3 基本块的优化
10.4 数据流分析

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

- 例

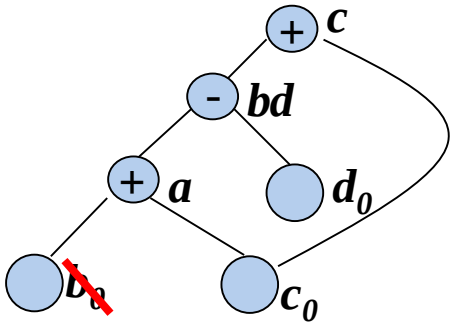
  $a = b + c$

  $b = a - d$

  $c = b + c$

  $d = a - d$

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

基本块的 *DAG* 表示

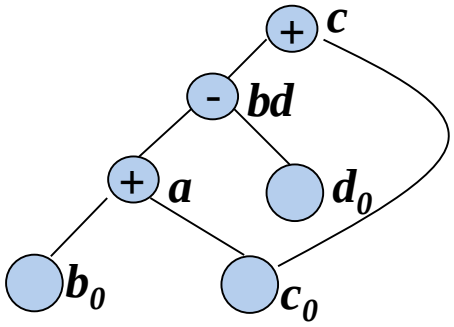
- 例

  $a = b + c$

  $b = a - d$

  $c = b + c$

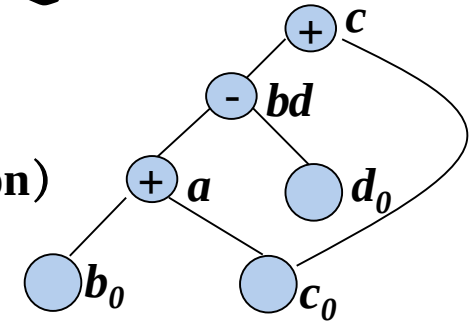
  $d = a - d$



- 基本块中的某些结点被指明为输出结点（output node），这些结点的变量在基本块的出口处活跃。也就是说，这些变量的值可能以后会在流图的另一个基本块中被使用到。计算得到这些活跃变量是全局数据流分析的问题。

基本块的 *DAG* 表示可以改进代码质量

- 基本块的DAG表示使我们可以对基本块所代表的代码进行一些转换，以改进代码质量。
  - 可以消除局部公共子表达式（local common subexpression）
  - 可以消除死代码（dead code）
  - 可以对相互独立的语句进行重新排序，从而降低一个临时值需要保持在寄存器中的时间
  - 可以使用代数规则重排三地址指令的运算分量，有时可以简化计算过程。



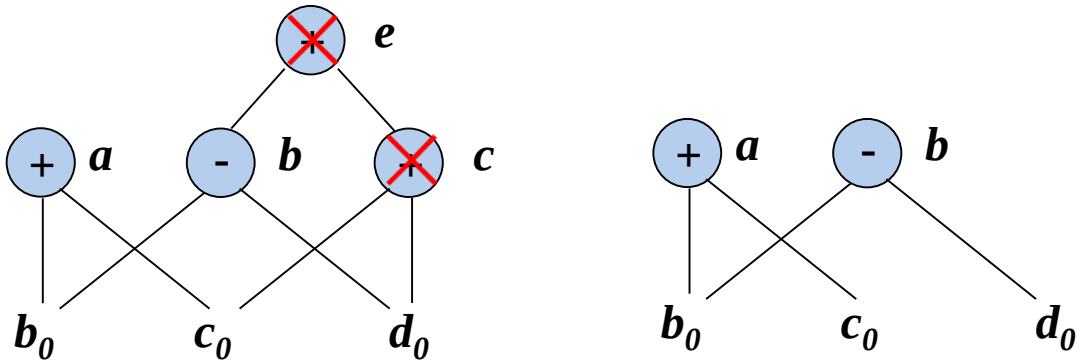
$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$

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中消除所有对应于死代码的结点

- 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

利用代数规则对基本块优化

- 例

$x + 0 = 0 + x = x$　　$x - 0 = x$　(恒等式消除计算步骤）

$x \times 1 = 1 \times x = x$　　$x/1 = x$

$2 \times x = x + x$　　(强度消减）

$a = b + c$

$e = c + d + b$

应用加法交换律和结合律，改为：

$a = b + c$

$t = c + d$

$e = t + b$

如果t没有在基本块之外使用，可改为：

$a = b + c$

$e = a + d$

数组元素赋值指令的表示

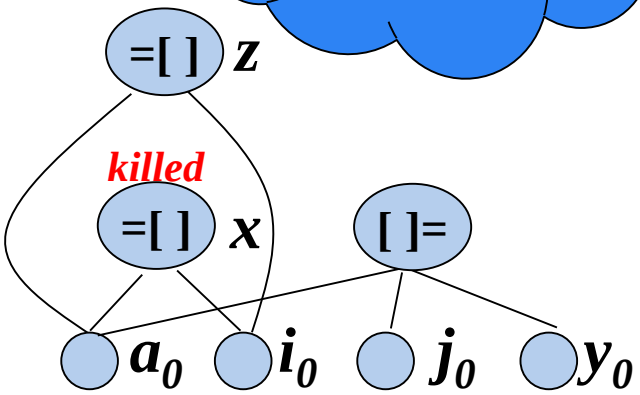
- 例

$x=a[i]$

$a[j]=y$

$z=a[i]$

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

根据基本块的*DAG*可以获得一些非常有用的信息

- 确定哪些变量的值在该基本块中赋值前被引用过
  - 在*DAG*中创建了叶结点的那些变量
- 确定哪些语句计算的值可以在基本块外被引用
  - 在*DAG*构造过程中为语句*s*（该语句为变量*x*定值）创建的节点*N*，在*DAG*构造结束时*x*仍然是*N*的定值变量

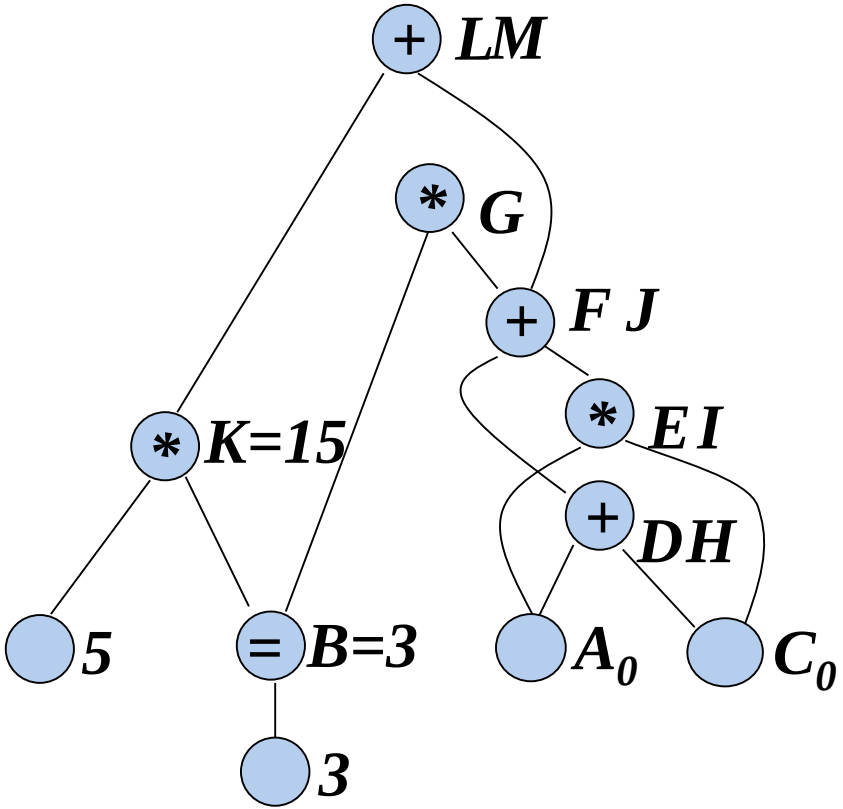
从 *DAG* 到基本块的重组

- 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 $M = L$

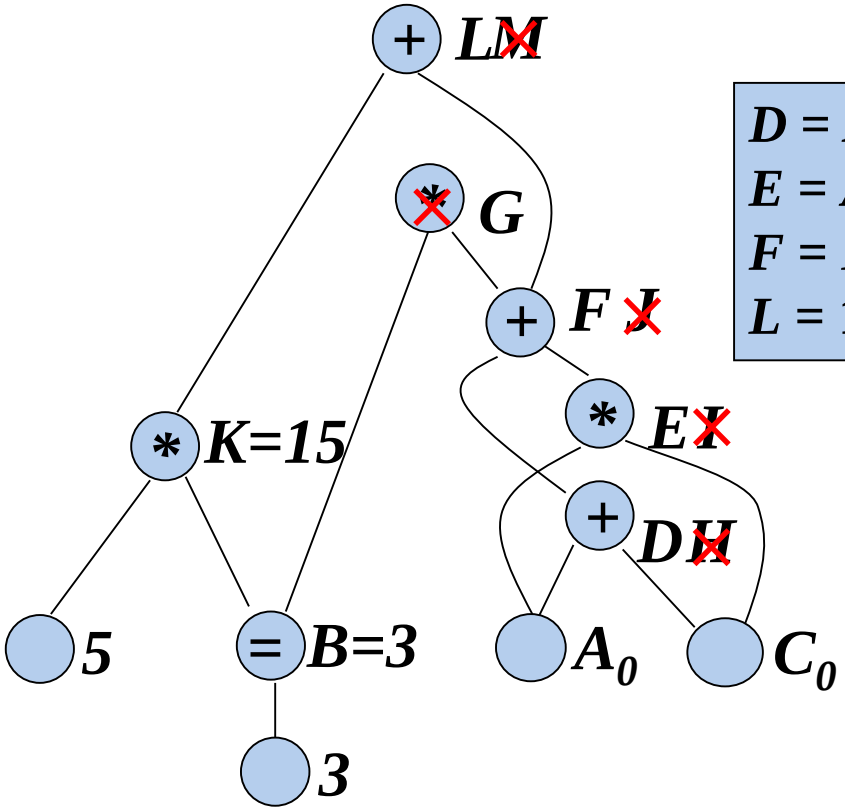


假设：仅变量L在基本块出口之后活跃

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

例

* for i from 1 to 10 do
* for j form 1 to 10 do
* a[i, j]=0;
* for i from 1 to 10 do
* a[i, j]= 1.0

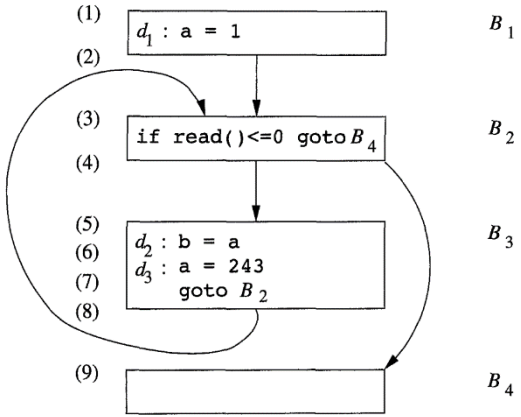
1 : i = 1
2 : j = 1
3 : t1 = 10 * i
4 : t2 = t1 + j
5 : t3 = 8 * t2
6 : t4 = t3 – 88
7 : a[t4] = 0.0
8 : j = j + 1
9: if j<=10 goto 3
10: i = i+1
11: if i<=10 goto 2
12: i = 1
13: t5 = i -1
14: t6 = 88 * t5
15: a[t6] = 1.0
16: i = i+1
17: if i <= 10 goto 13

提纲

10.1 基本块和流图
10.2 常用的代码优化方法
10.3 基本块的优化
10.4 数据流分析

8.4 数据流分析(data-flow analysis)

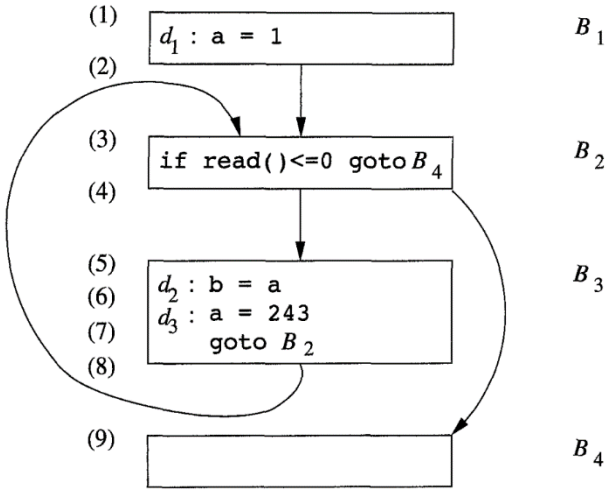
- 数据流分析
  - 一组用以收集程序相关信息的算法
- 在每一种数据流分析应用中，都会把每个程序点和一个数据流值关联起来
  - 程序点：流图基本块中的位置，包括
    - 第一个语句之前
    - 两个相邻语句之间
    - 最后一个语句之后



- 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(data-flow analysis)

- 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*)

数据流分析模式

数据流问题就是要对一组约束求解，约束分为两种：
- 基于语句语义（传递函数）的约束
- 基于控制流的约束

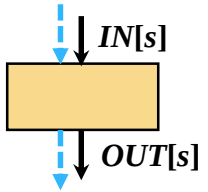
- 语句的数据流模式
  - *IN*[*s*]: 语句*s*之前的数据流值

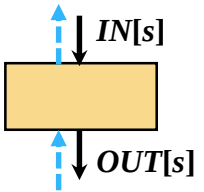
    OUT[*s*]: 语句*s*之后的数据流值
  - f_s: 语句*s*的传递函数(transfer function)
    - 定义：一个赋值语句*s*之前和之后的数据流值的关系
    - 传递函数的两种风格
      - 信息沿执行路径前向传播(前向数据流问题)

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

      - 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

数据流问题就是要对一组约束求解，约束分为两种：
- 基于语句语义（传递函数）的约束
- 基于控制流的约束

- 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: $OUT[B] = f_B(IN[B])$

$$f_B = f_{sn} \cdots f_{s2}\ f_{s1}$$

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

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：$OUT[B] = f_B(IN[B])$

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

    $f_B = f_{s1}\ f_{s2} \cdots f_{sn}$

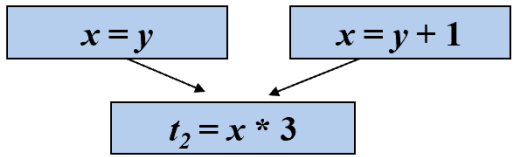
$$
\begin{aligned}
& IN[B] \\
&= IN[s_1] \\
&= f_{s1}(OUT[s_1]) \\
&= f_{s1}(IN[s_2]) \\
&= f_{s1}\ f_{s2}\ (OUT[s_2]) \\
&= f_{s1}\ f_{s2}(IN[s_3]) \\
&\cdots \\
&= f_{s1}\ f_{s2} \cdots f_{s(n-1)}(IN[s_n]) \\
&= f_{s1}\ f_{s2} \cdots\ f_{s(n-1)}\ f_{sn}(OUT[s_n]) \\
&= f_{s1}\ f_{s2} \cdots\ f_{s(n-1)}\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

数据流分析(data-flow analysis)

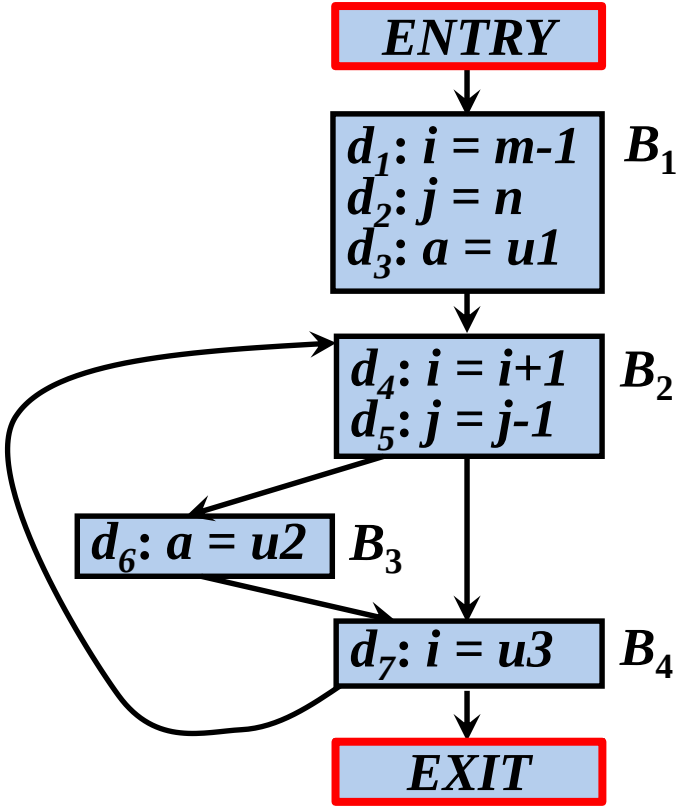
- 数据流分析应用
  - 到达-定值分析 (*Reaching-Definition Analysis*)
  - 活跃变量分析 (*Live-Variable Analysis*)
  - 可用表达式分析 (*Available-Expression Analysis*)

8.4.1 到达定值分析-最常见且有用

- 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	√	×	×

如果存在一条从紧跟在定值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'，则称变量x被这个定值d'“杀死”了)，则称定值d到达程序点p

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
- 常量合并
- 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）

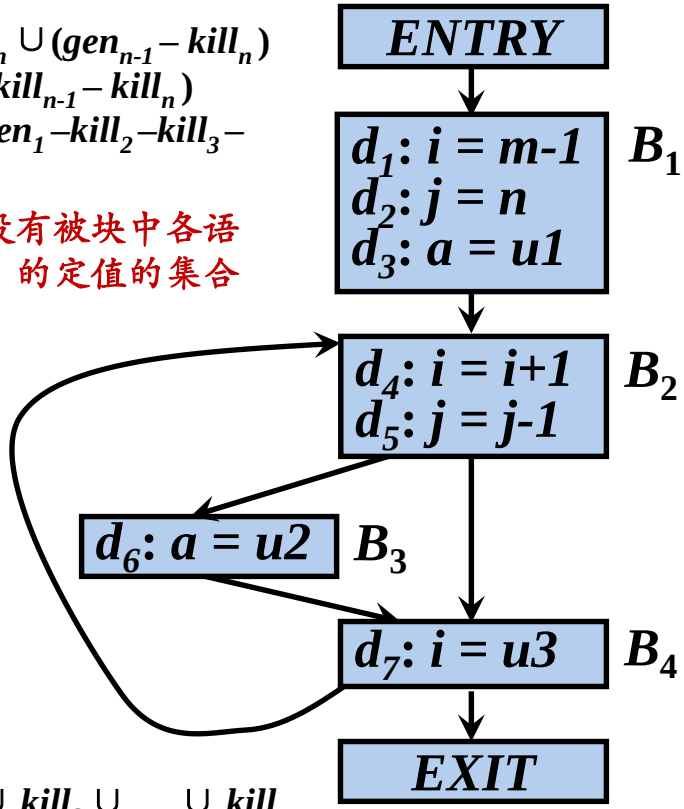
到达定值的传递函数

- 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$kill_B$

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

基本块中没有被块中各语句“杀死”的定值的集合



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

$kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$

被基本块B中各个语句杀死的定值的集合　$kill_d$：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

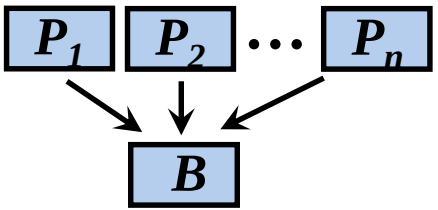
  $OUT[B]$：到达流图中基本块B的出口处的定值的集合

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

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

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

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

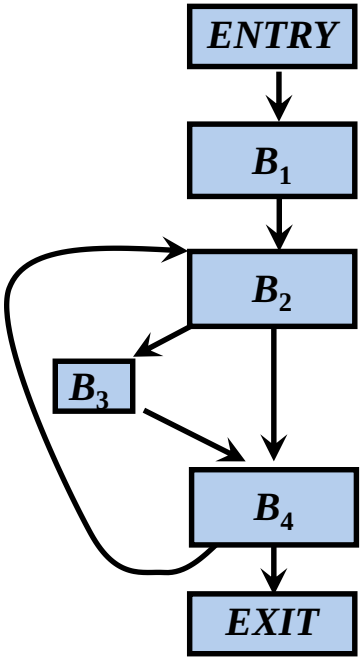


计算到达定值的迭代算法

- 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

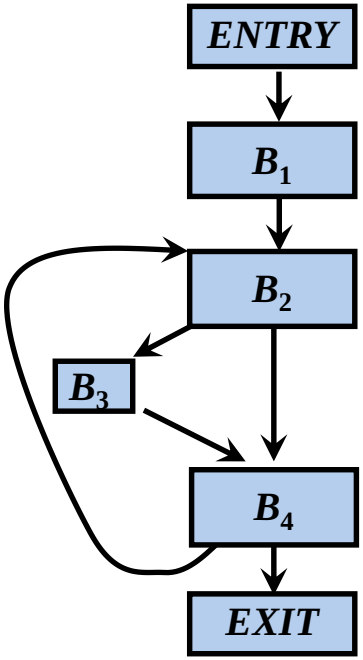


gen_{B1}={ d_1, d_2, d_3 }
$kill_{B1}$={ d_4, d_5, d_6, d_7 }
gen_{B2}={ d_4, d_5 }
$kill_{B2}$={ d_1, d_2, d_7 }
gen_{B3}={ d_6 }
$kill_{B3}$={ d_3 }
gen_{B4}={ d_7 }
$kill_{B4}$={ d_1, d_4 }

```
OUT[ENTRY] = Φ;
for (除ENTRY之外的每个基本块B) OUT[B] = Φ;
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] = gen_B ∪ (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

数据流分析(data-flow analysis)

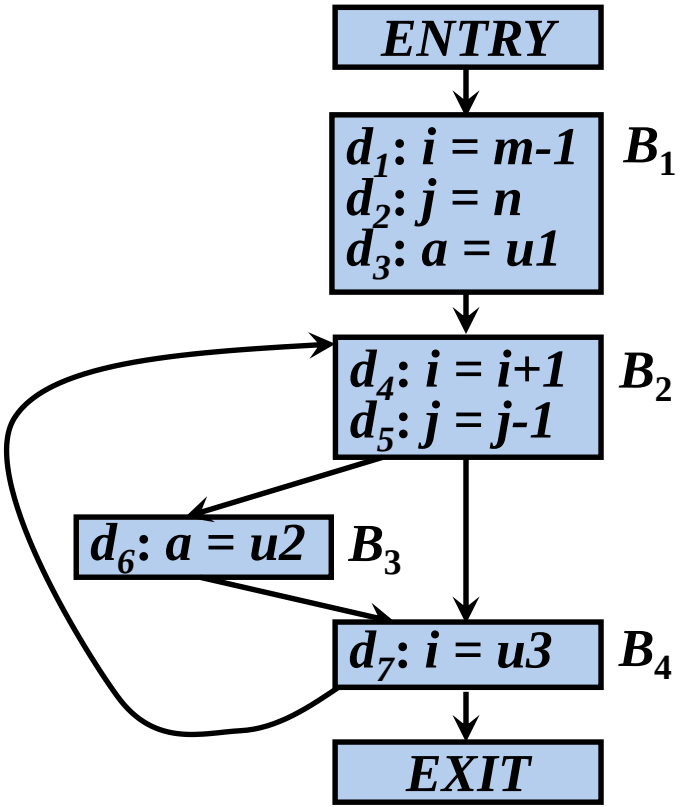
- 数据流分析应用
  - 到达-定值分析 (*Reaching-Definition Analysis*)
  - 活跃变量分析 (*Live-Variable Analysis*)
  - 可用表达式分析 (*Available-Expression Analysis*)

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

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是活跃的，否则称变量x在点p不活跃



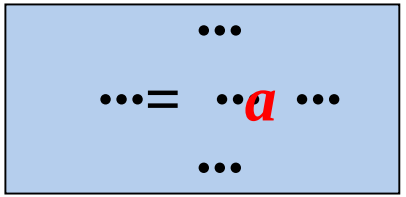
OUT[B]	B_1	B_2	B_3	B_4
a	×	×	×	×
i	√	×	×	√
j	√	√	√	√
m	×	×	×	×
n	×	×	×	×
u1	×	×	×	×
u2	√	√	√	√
u3	√	√	√	√

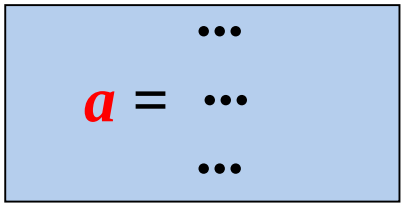
活跃变量信息的主要用途

- 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

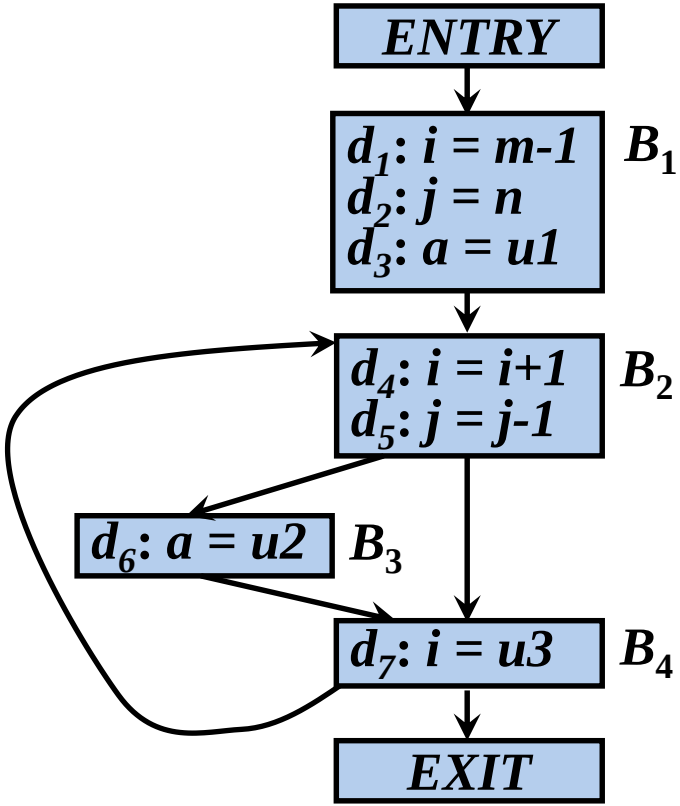




例：各基本块B的use_B和def_B

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

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



- 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

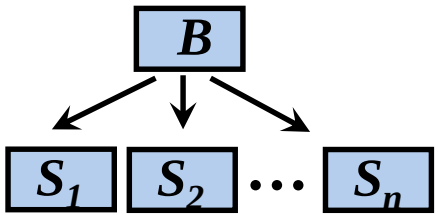
- $IN[B]$：在基本块B的入口处的活跃变量集合

  $OUT[B]$：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)

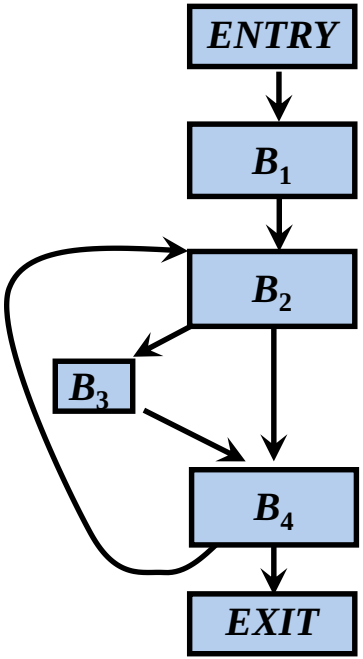
  for (除$EXIT$之外的每个基本块B) {

    $OUT[B] = \cup_{S是B的一个后继} IN[S]$;

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

  }

例

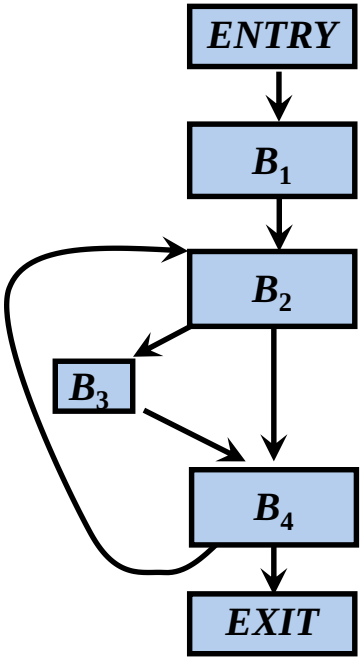


use_{B1} ={ m, n, u1 }
def_{B1} ={ i, j, a }
use_{B2} ={ i, j }
def_{B2}= Φ
use_{B3} ={ u2 }
def_{B3} ={ a }
use_{B4} ={ u3 }
def_{B4} ={ i }

$IN[EXIT] = \Phi$;
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

数据流分析(data-flow analysis)

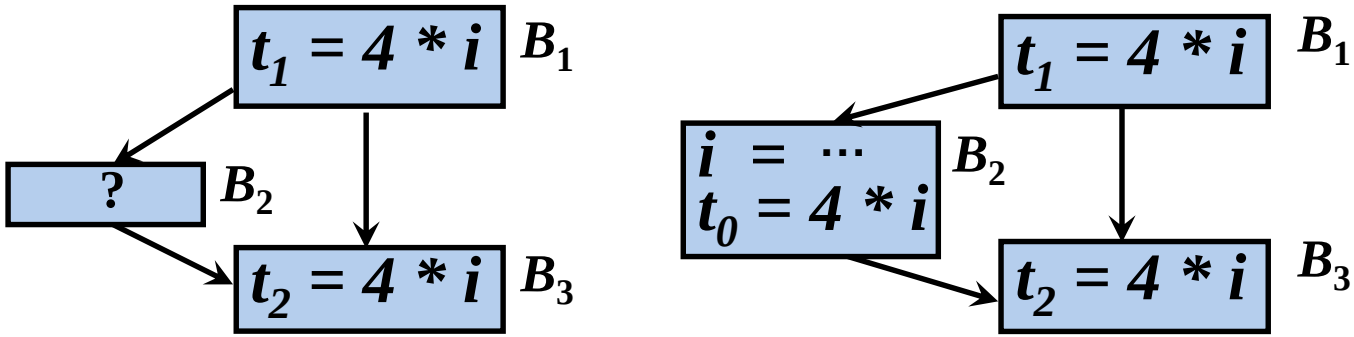
- 数据流分析应用
  - 到达-定值分析 (*Reaching-Definition Analysis*)
  - 活跃变量分析 (*Live-Variable Analysis*)
  - 可用表达式分析 (*Available-Expression Analysis*)

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

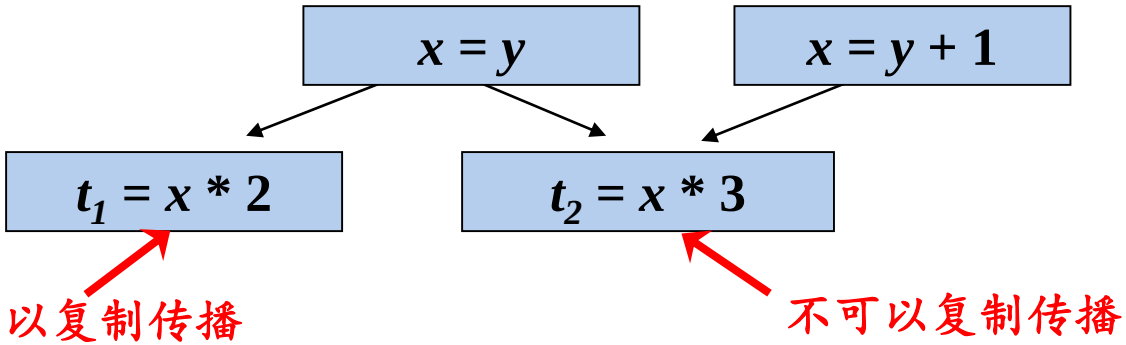


如果i在B_2中没有被赋予新值，或者在B_2中，对i赋值后又重新计算了$4*i$

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

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

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

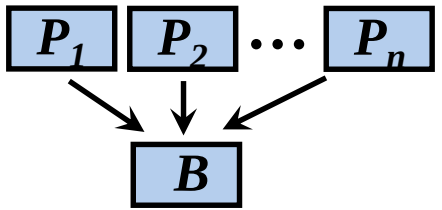
  $OUT[B]$：在B的出口处可用的U中的表达式集合

- 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



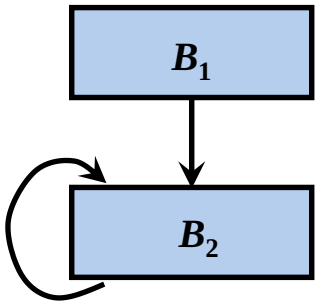
计算可用表达式的迭代算法

- 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



- 如果 $OUT[B_2]^0 = \Phi$

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

- 如果$OUT[B_2]^0 = U$

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