第八章 代码优化1

南京信息工程大学
凌妙根
2024年冬

NUIST

主要内容

- 基本块和流图
- 常用的优化方法
- 基本块的优化
- 数据流分析

什么是代码优化?

- 对代码进行等价的程序变换

采用什么工具?

- 优化时，通常需要分析程序的控制流程。用来描述代码优化的工具叫流图。流图的每一个节点是一个基本块。

8.1 流图

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机

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跳转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跳转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) $v = a[t_1]$

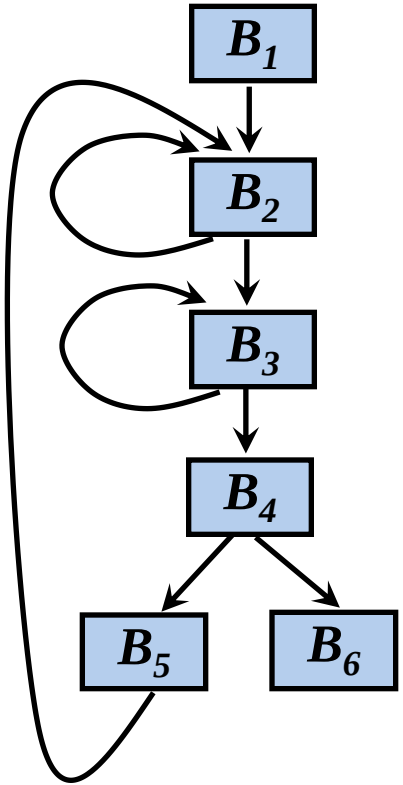
B_2
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) *if* $t_3 > v$ *goto*(5)

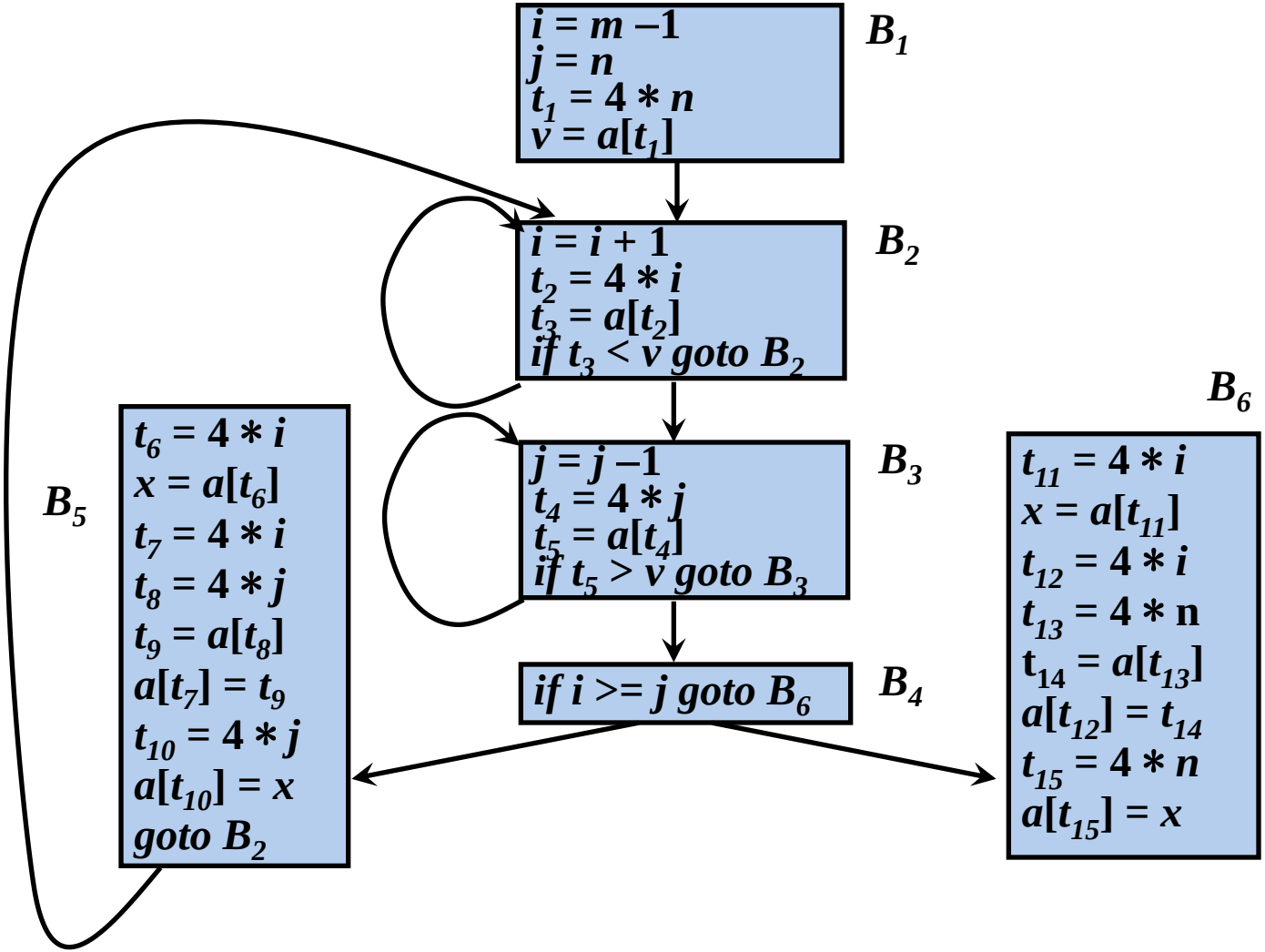
B_3
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) *if* $t_5 > v$ *goto*(9)

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

8.2 优化的分类

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

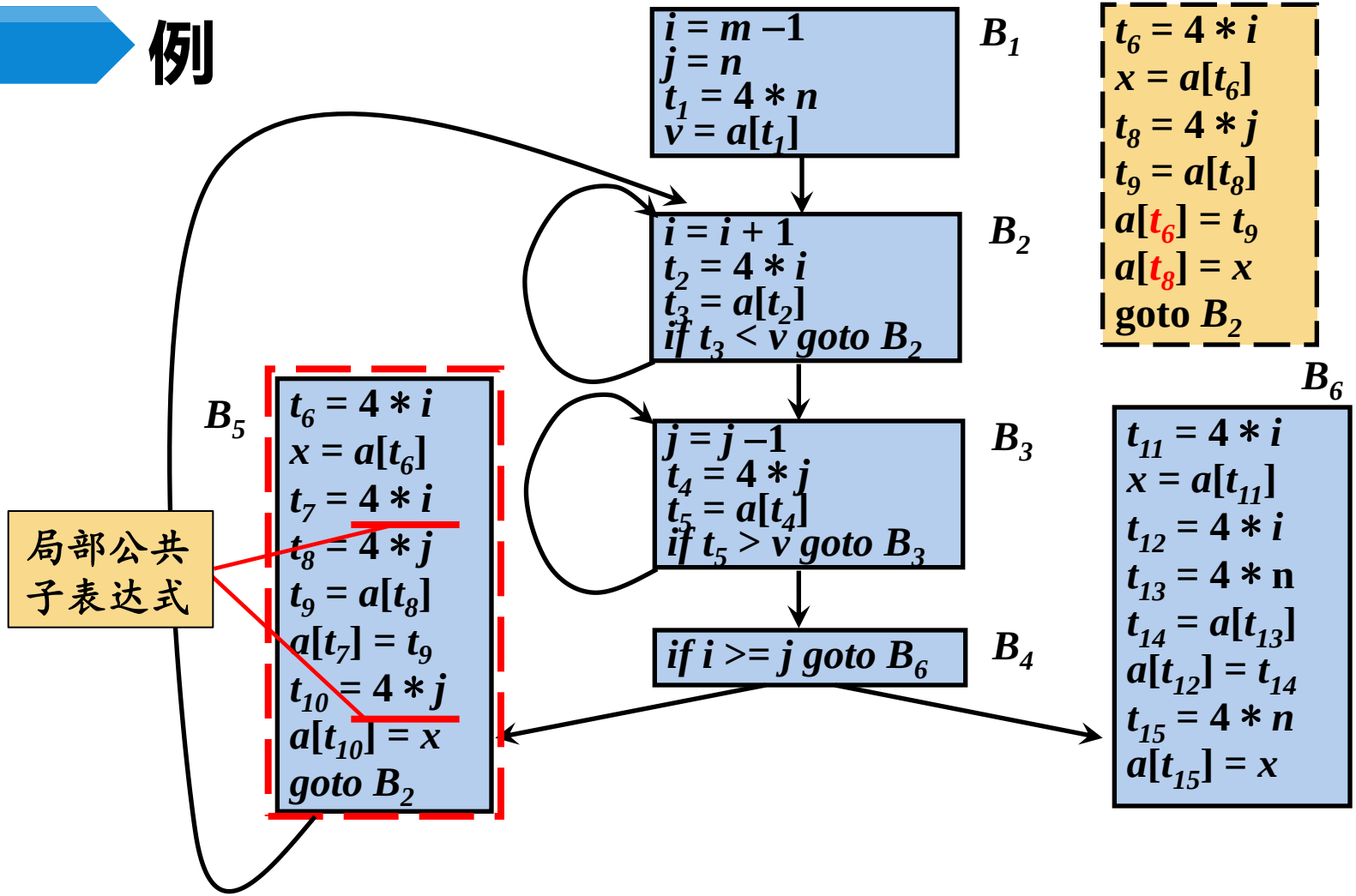
常用的优化方法

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

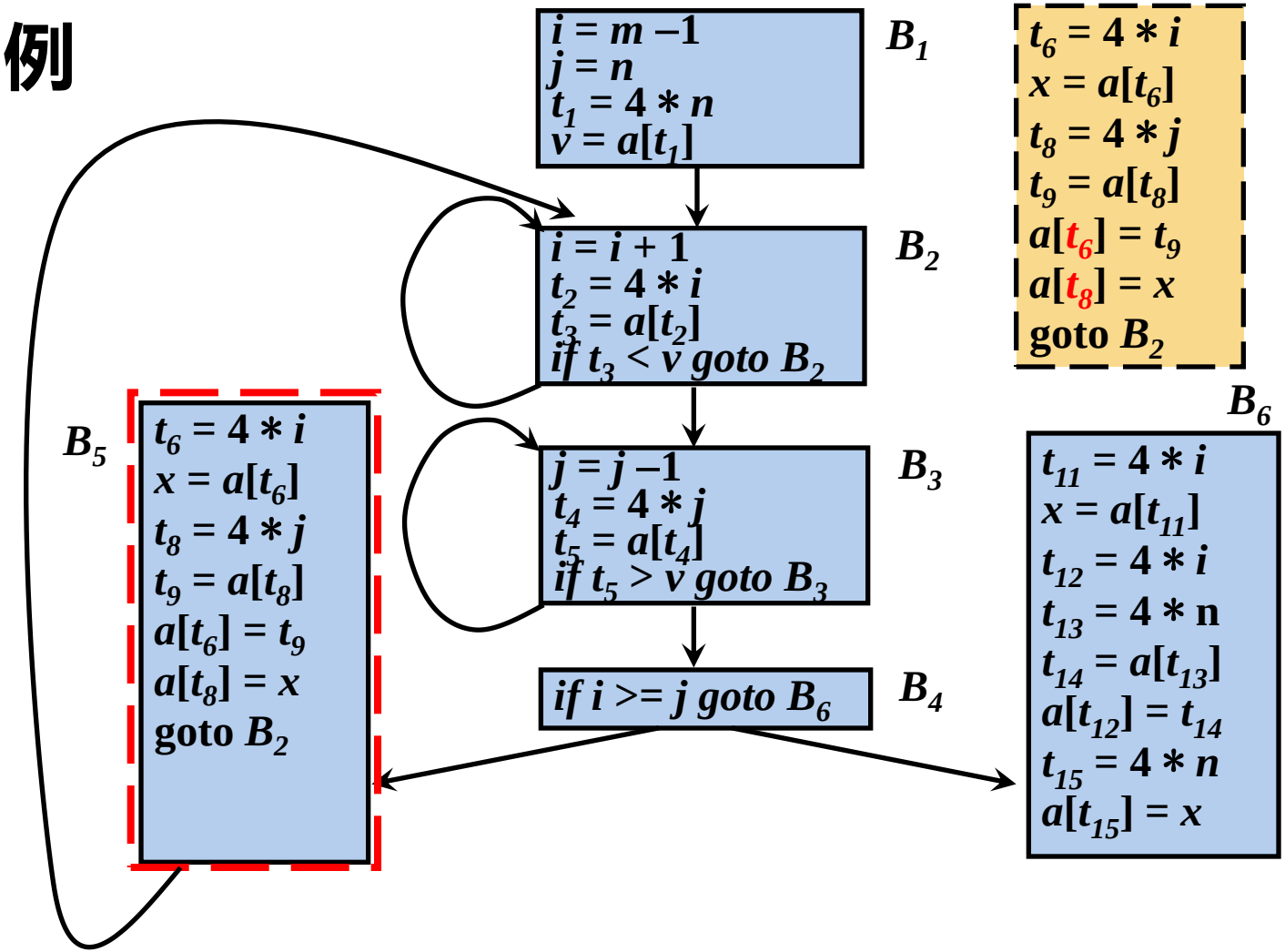
① 删除公共子表达式

- 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*)

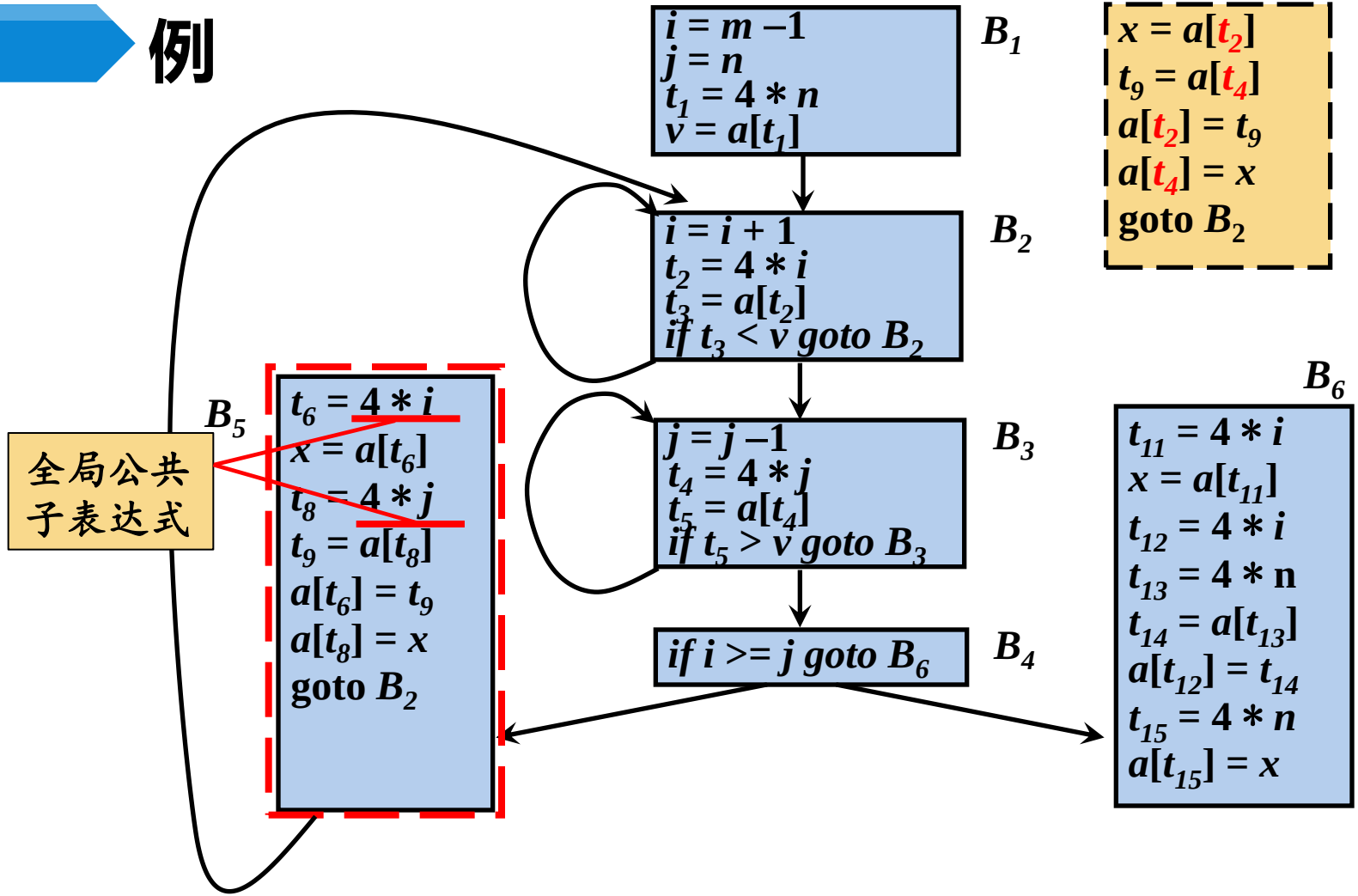
例


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

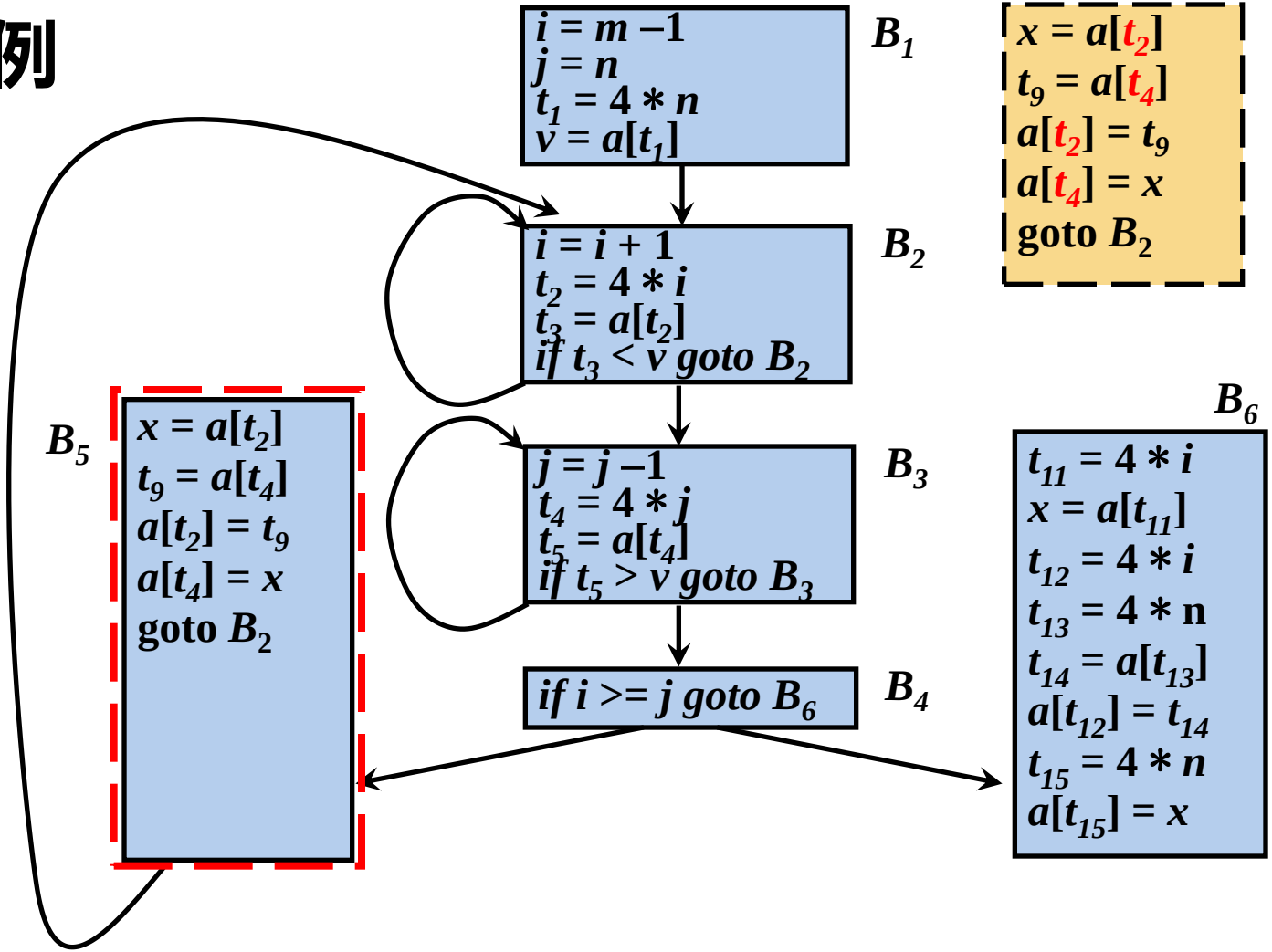
例


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

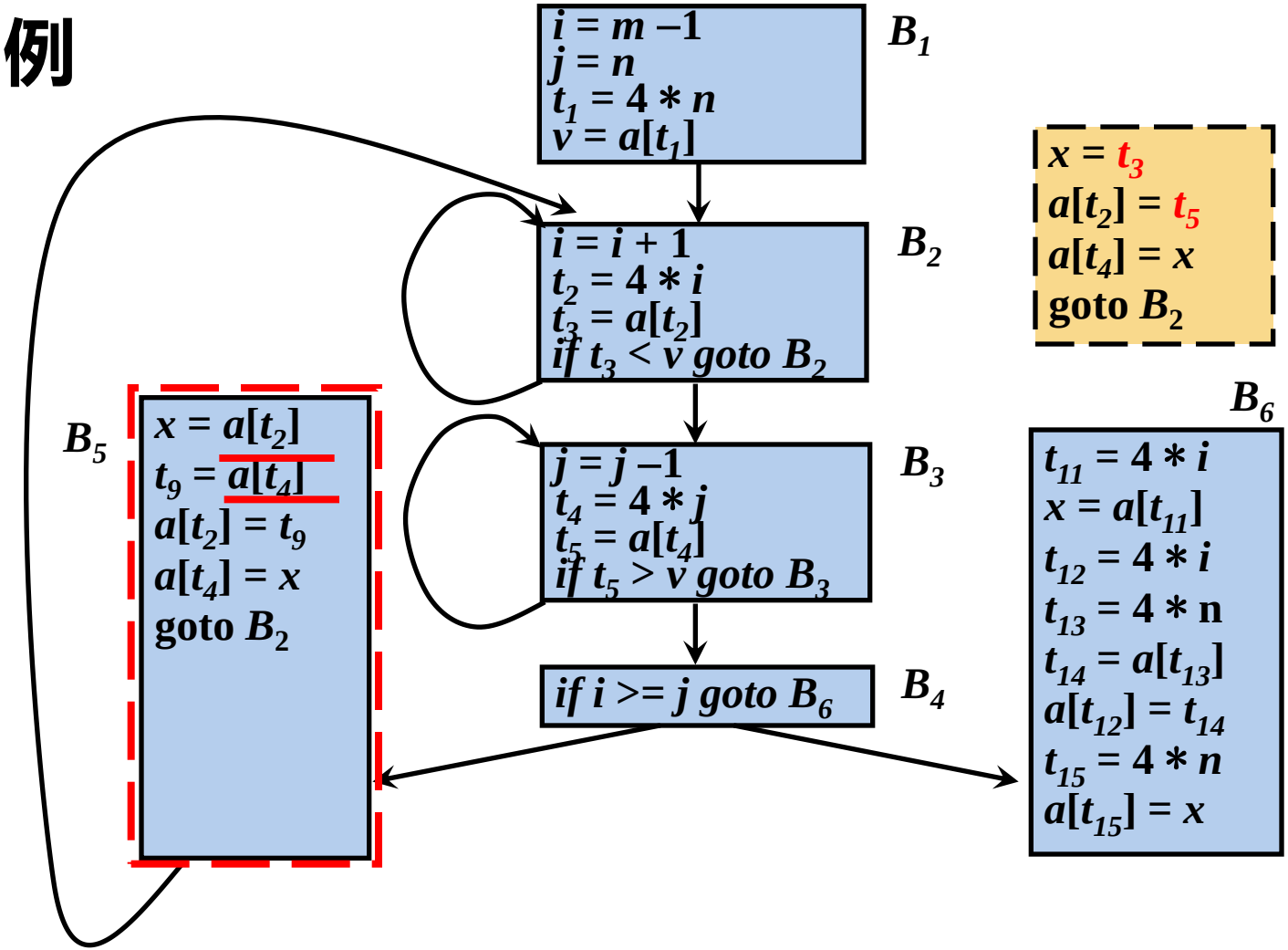
例


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

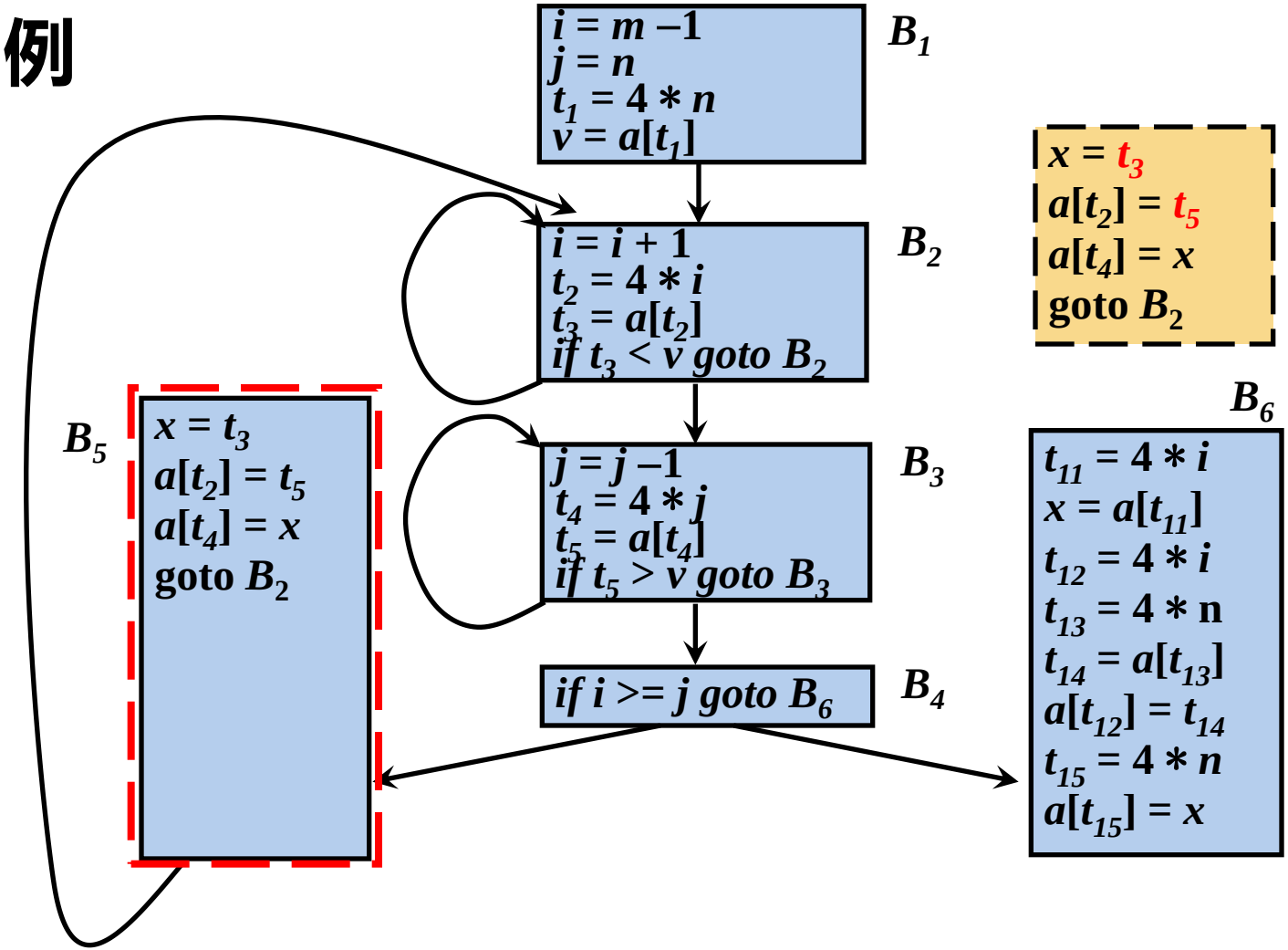
例


i = m –1
j = n
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
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x
B6
x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2

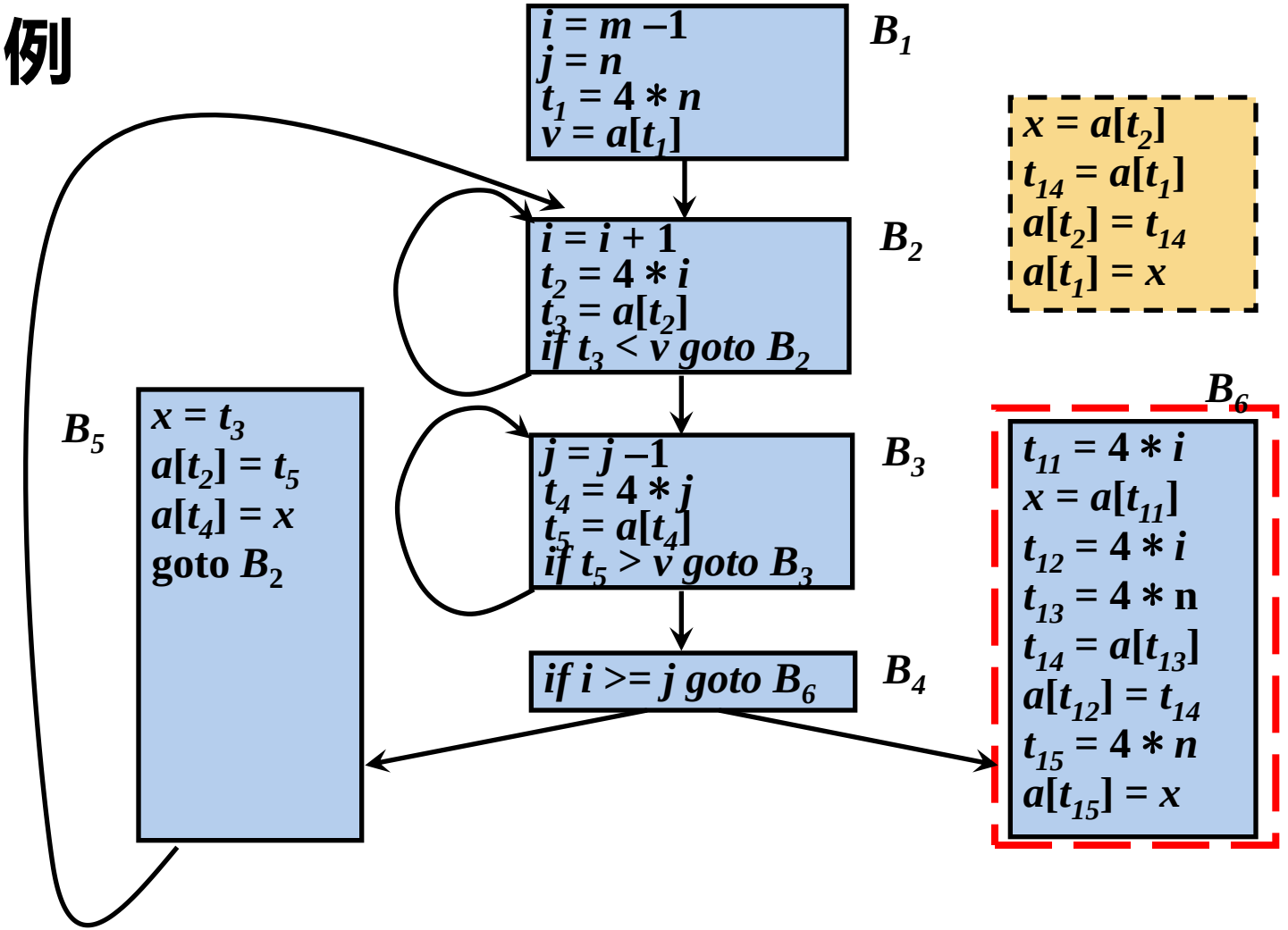
例


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

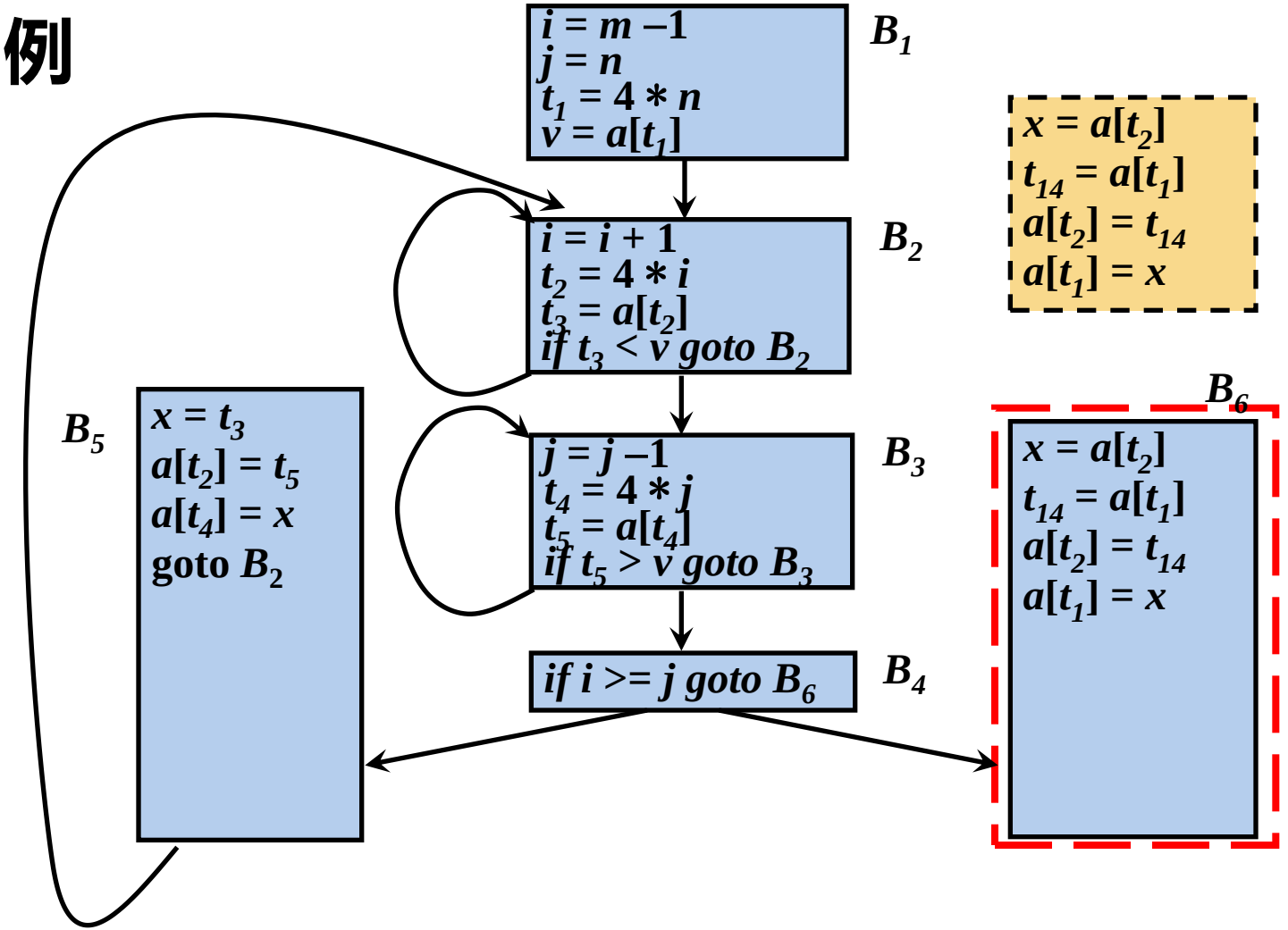
例


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

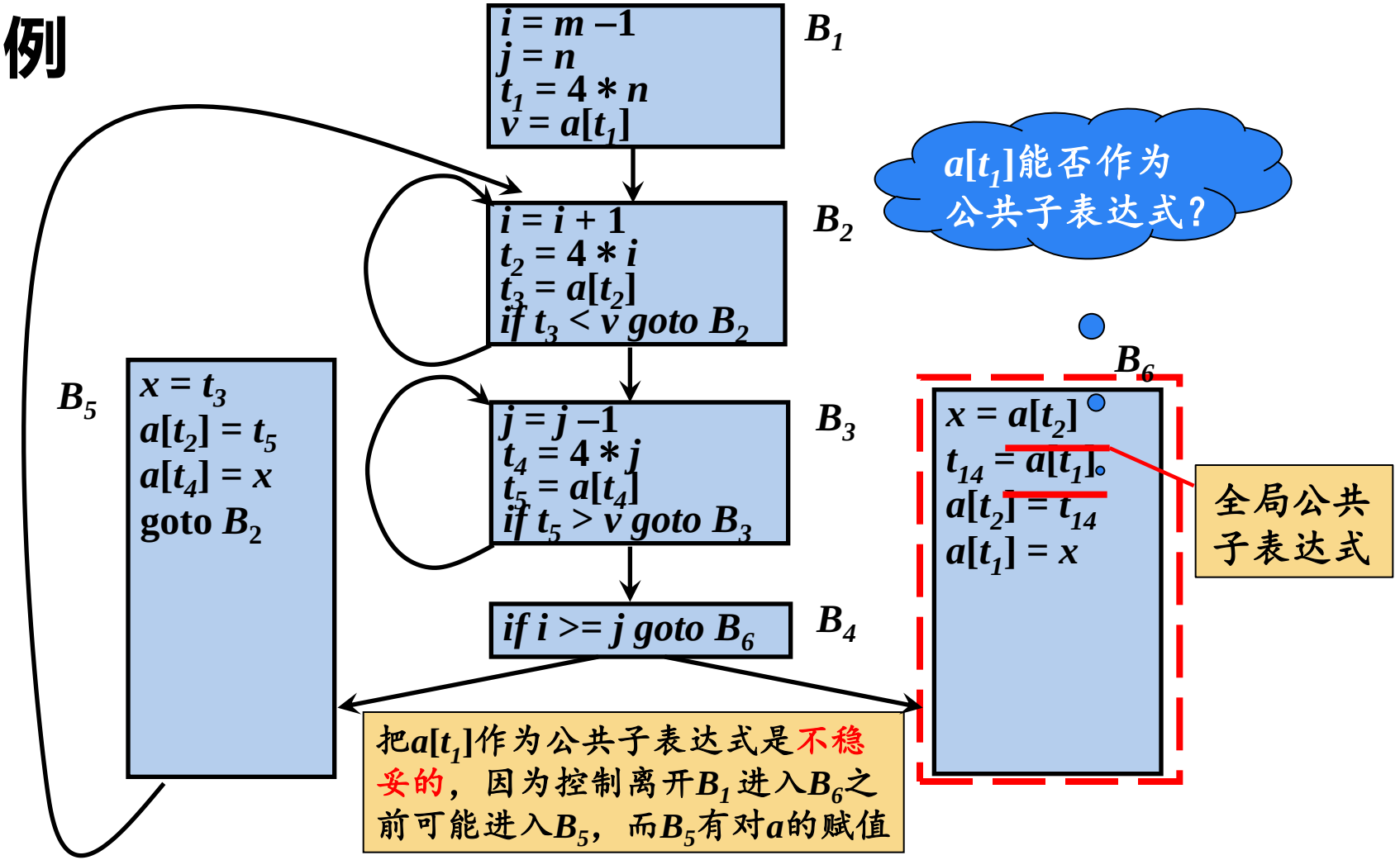
例


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

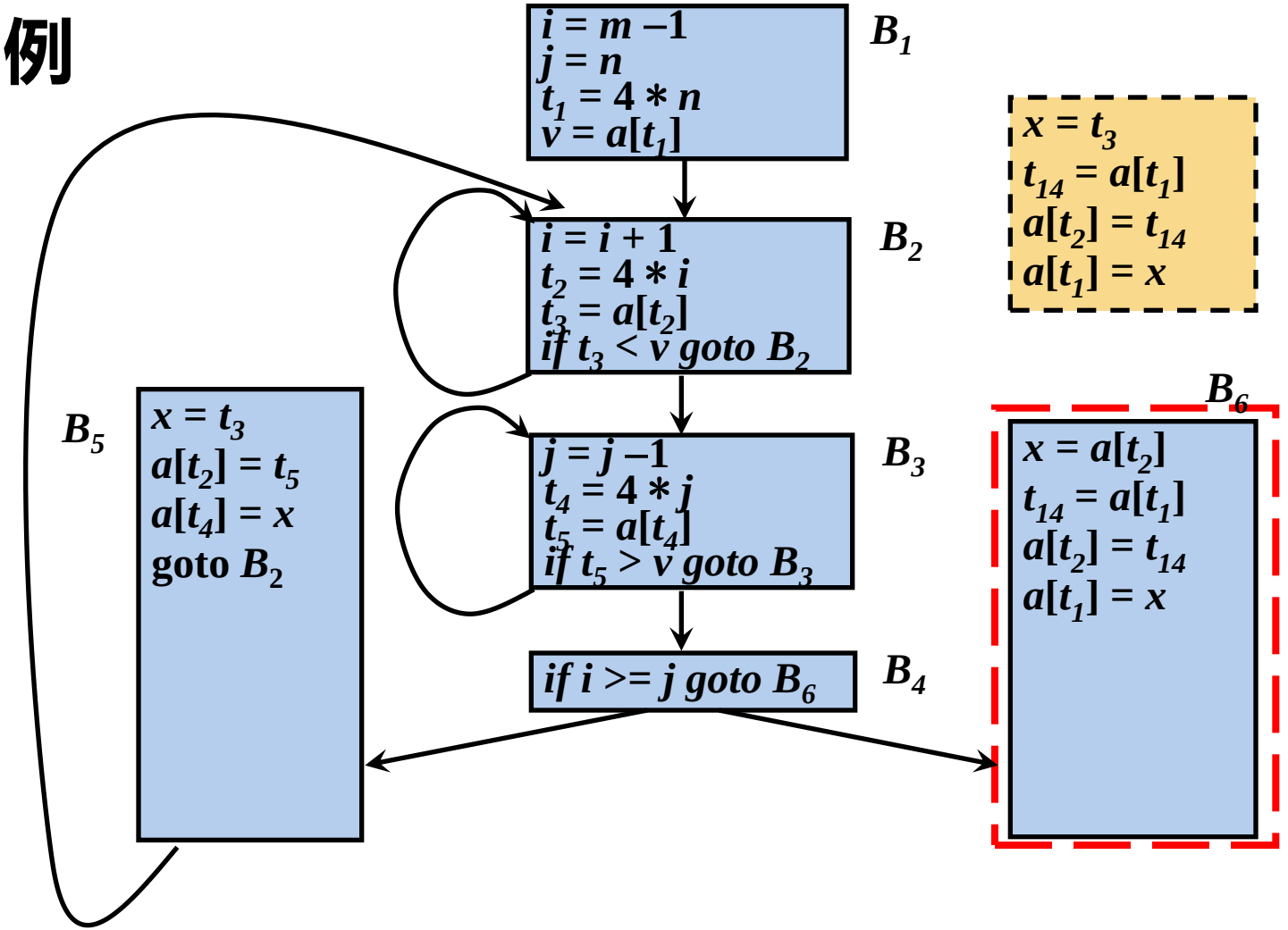
例


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

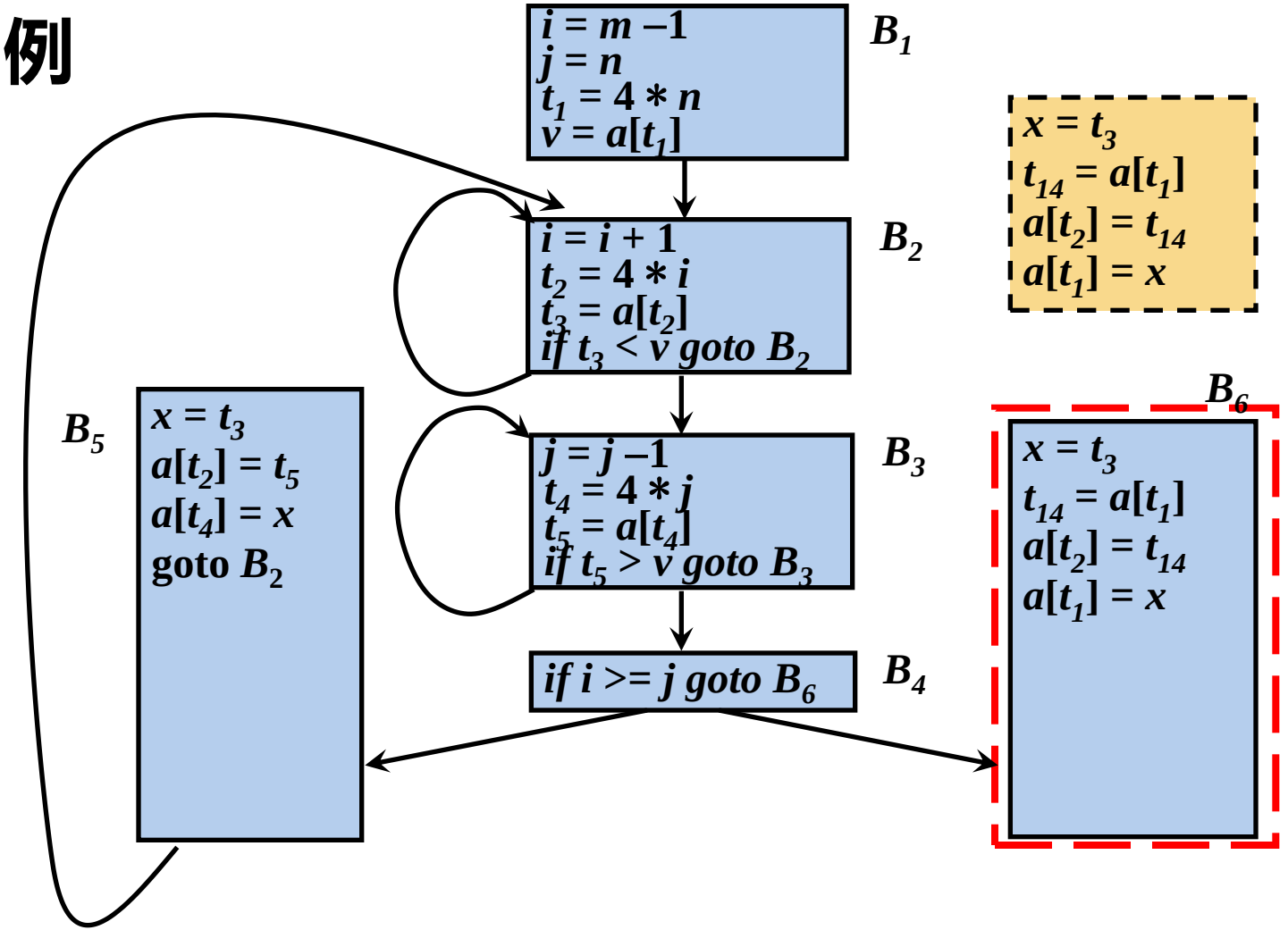
例


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
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

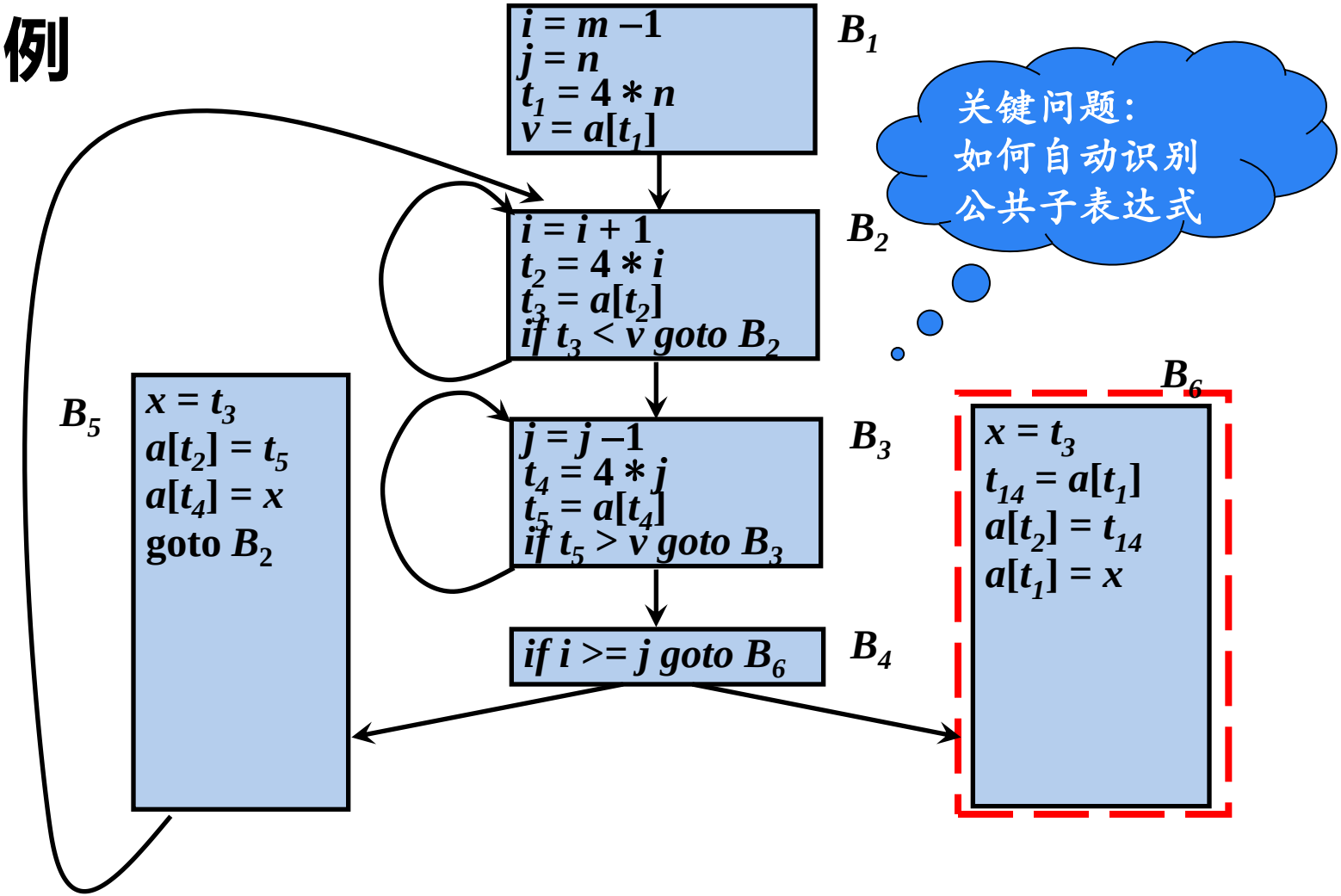
例


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

例

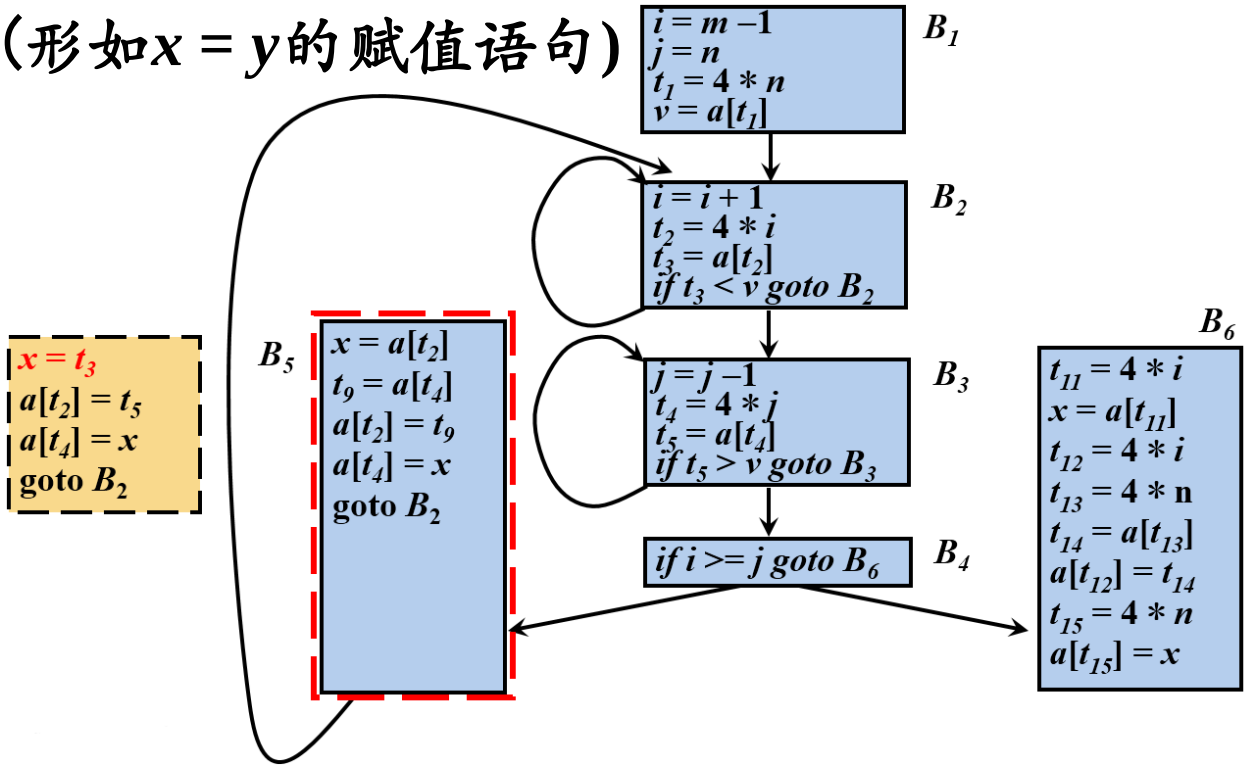

i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x

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

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

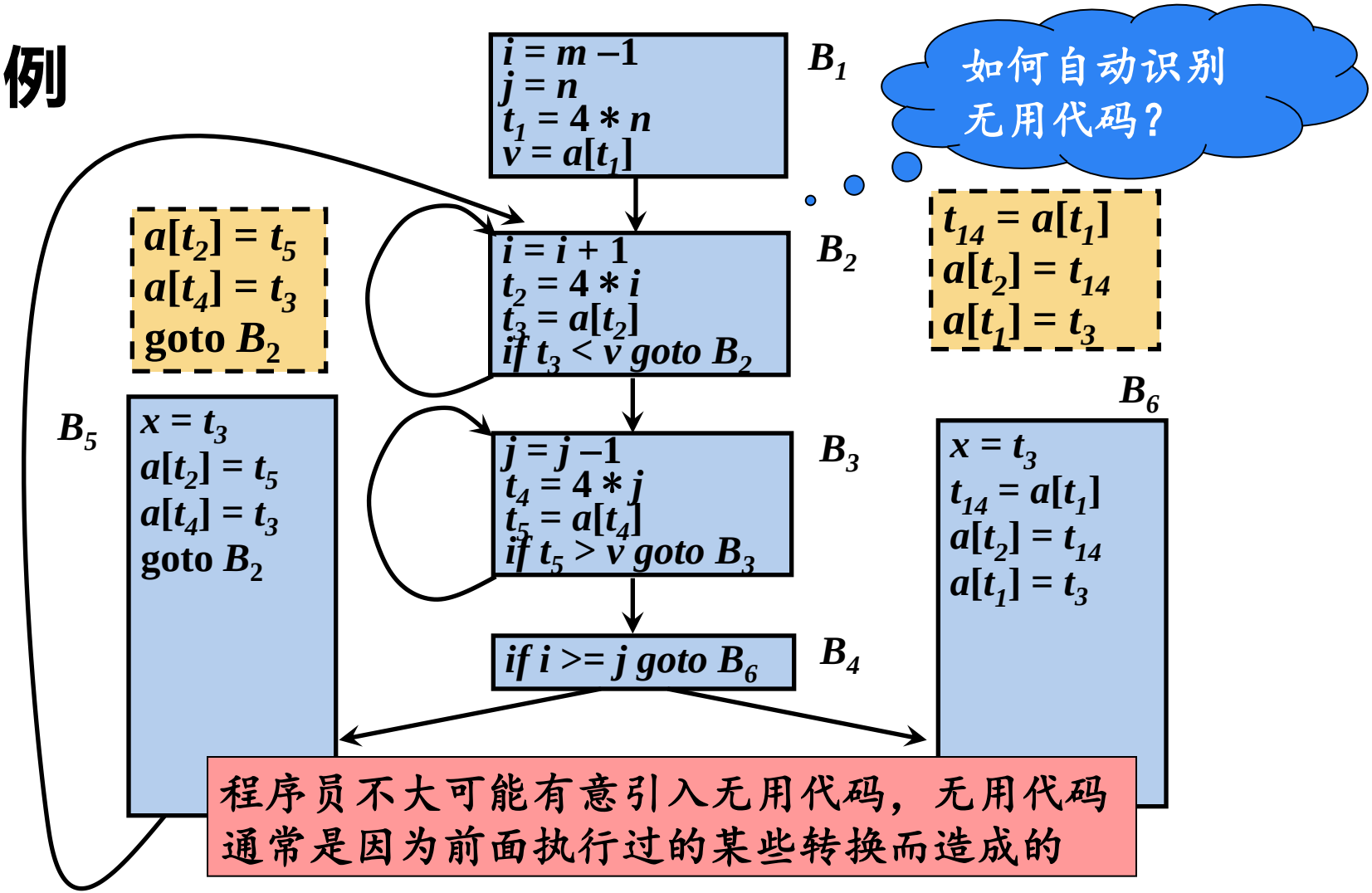
→

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

② 删除无用代码

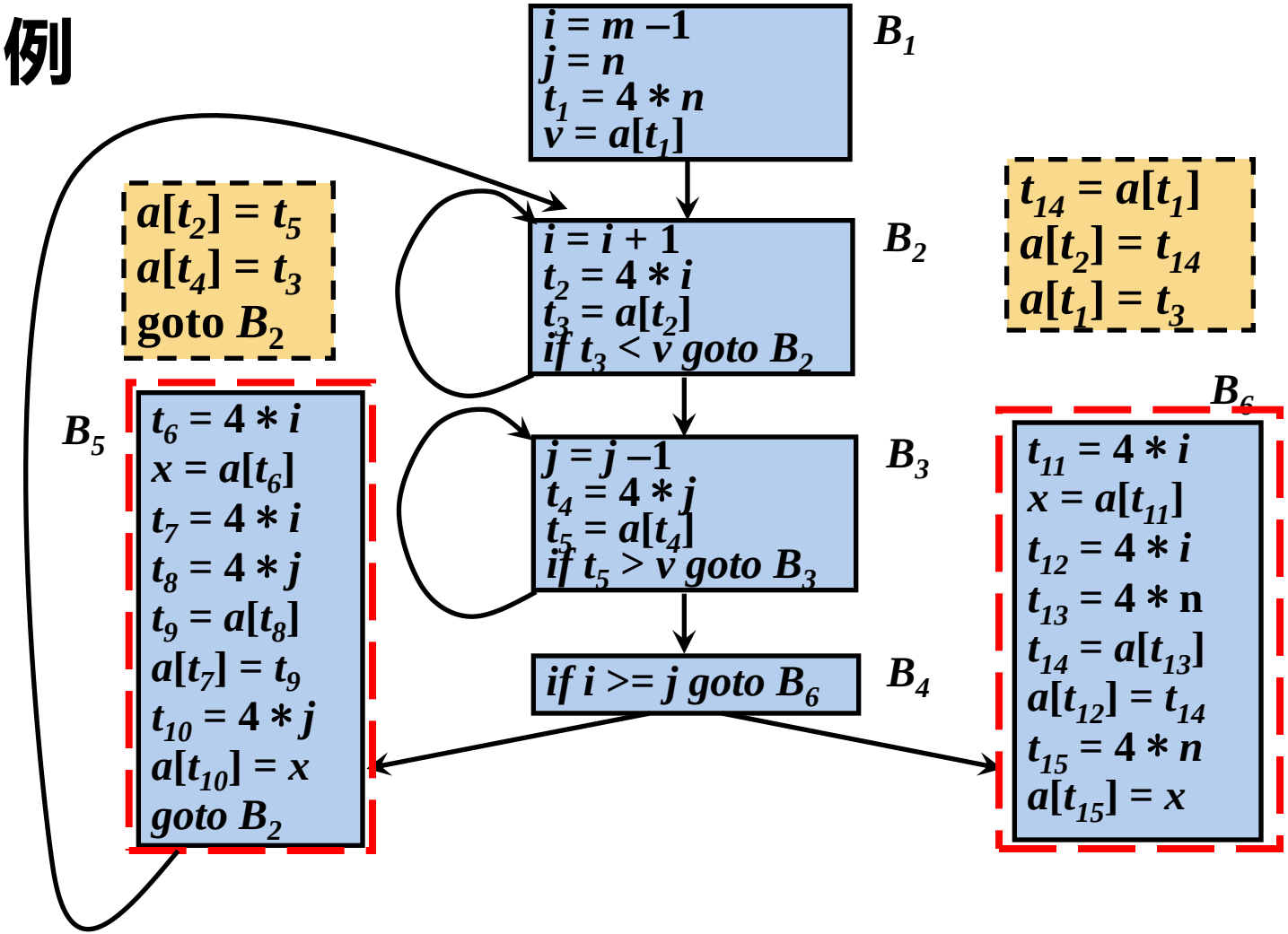
- 复制传播
  - 常用的公共子表达式消除算法和其它一些优化算法会引入一些复制语句(形如$x = y$的赋值语句)
  - 复制传播：在复制语句$x = y$之后尽可能地用y代替x
    - 复制传播给删除无用代码带来机会
- 无用代码(死代码*Dead-Code*)：其计算结果永远不会被使用的语句

例



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

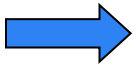
例


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
a[t2] = t5
a[t4] = t3
goto B2
t14 = a[t1]
a[t2] = t14
a[t1] = t3
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
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
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

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： $l = 2*3.14*r$

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



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

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; ... }
```

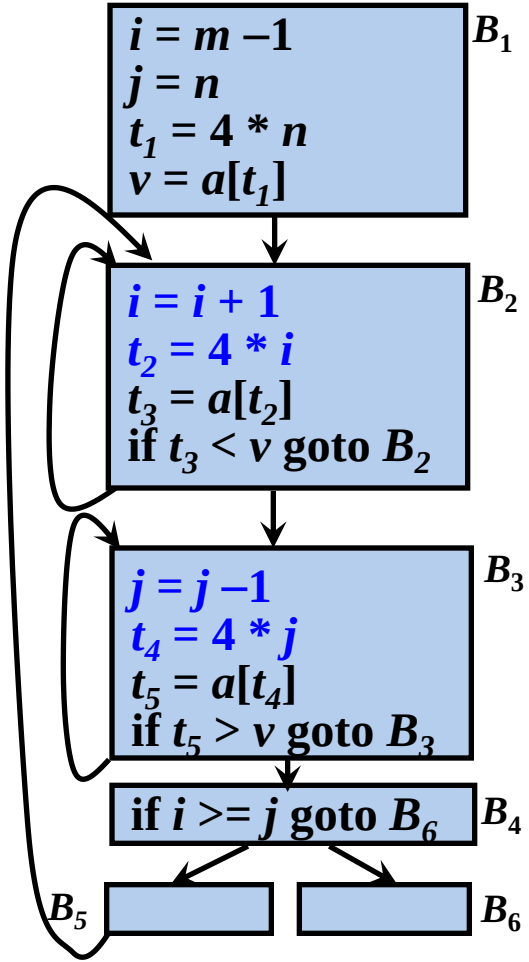
⑤ 强度削弱(*Strength Reduction*)

- 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

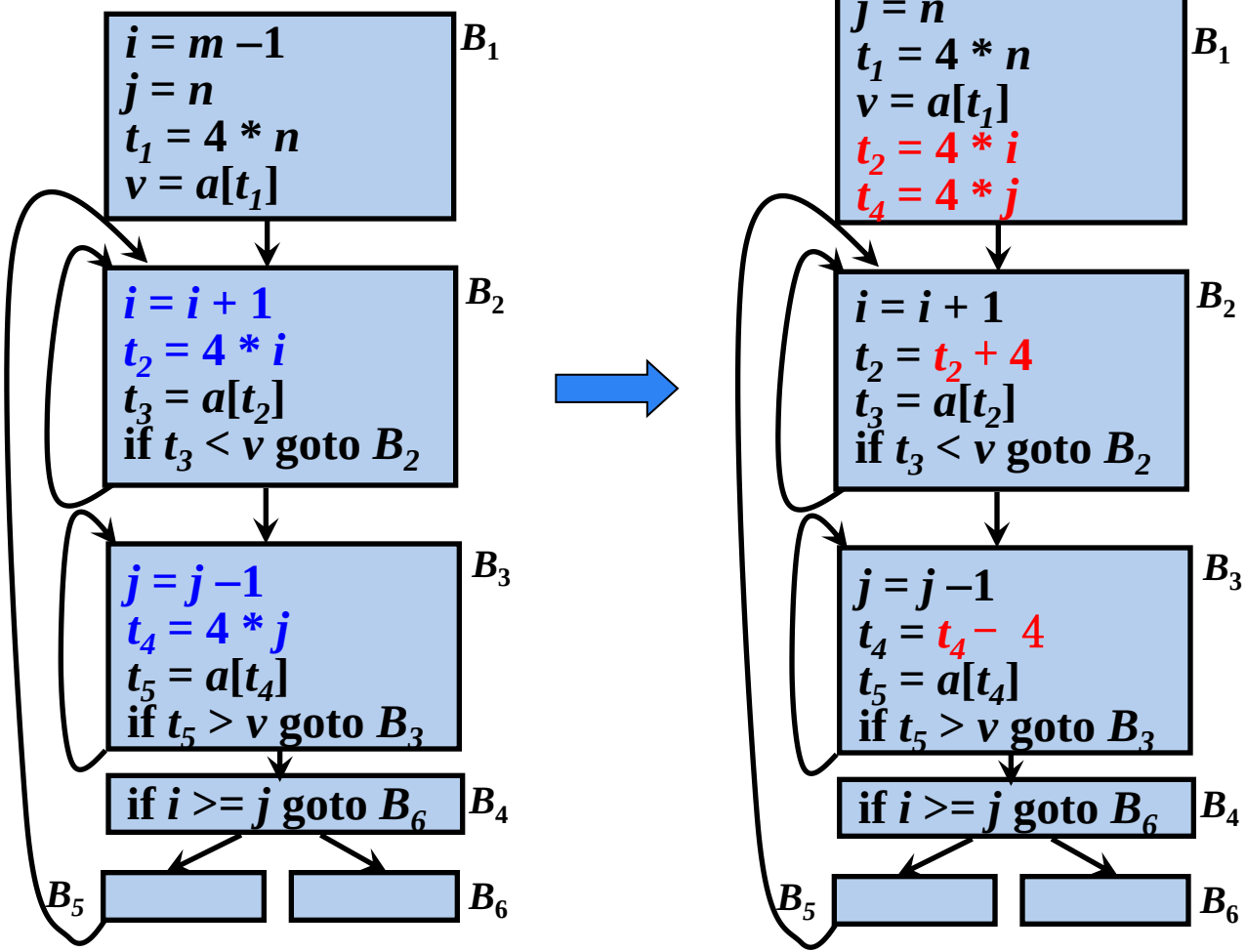
- 归纳变量
  - 对于一个变量x，如果存在一个正的或负的常数c使得每次x被赋值时它的值总增加c，那么x就称为归纳变量(*Induction Variable*)

例

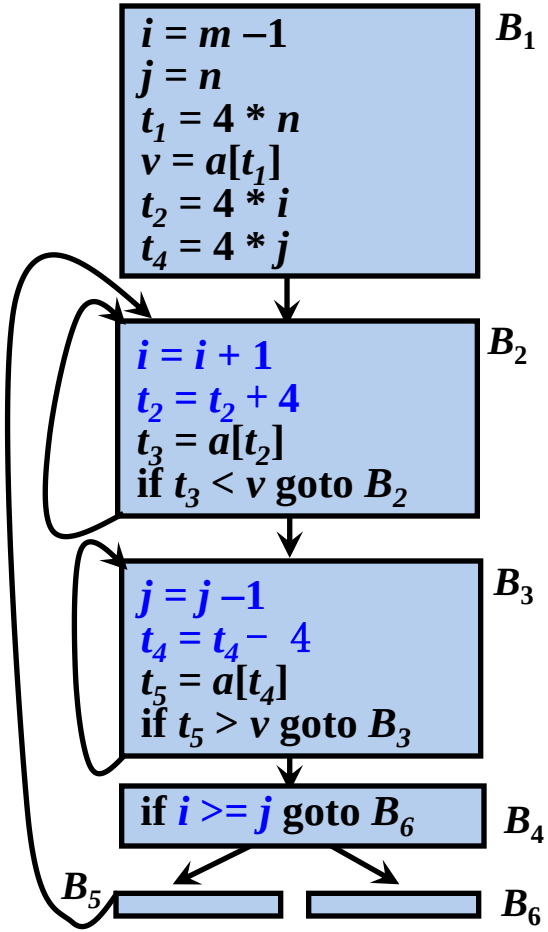


归纳变量可以通过在每次循环迭代中进行一次简单的增量运算(加法或减法)来计算

例

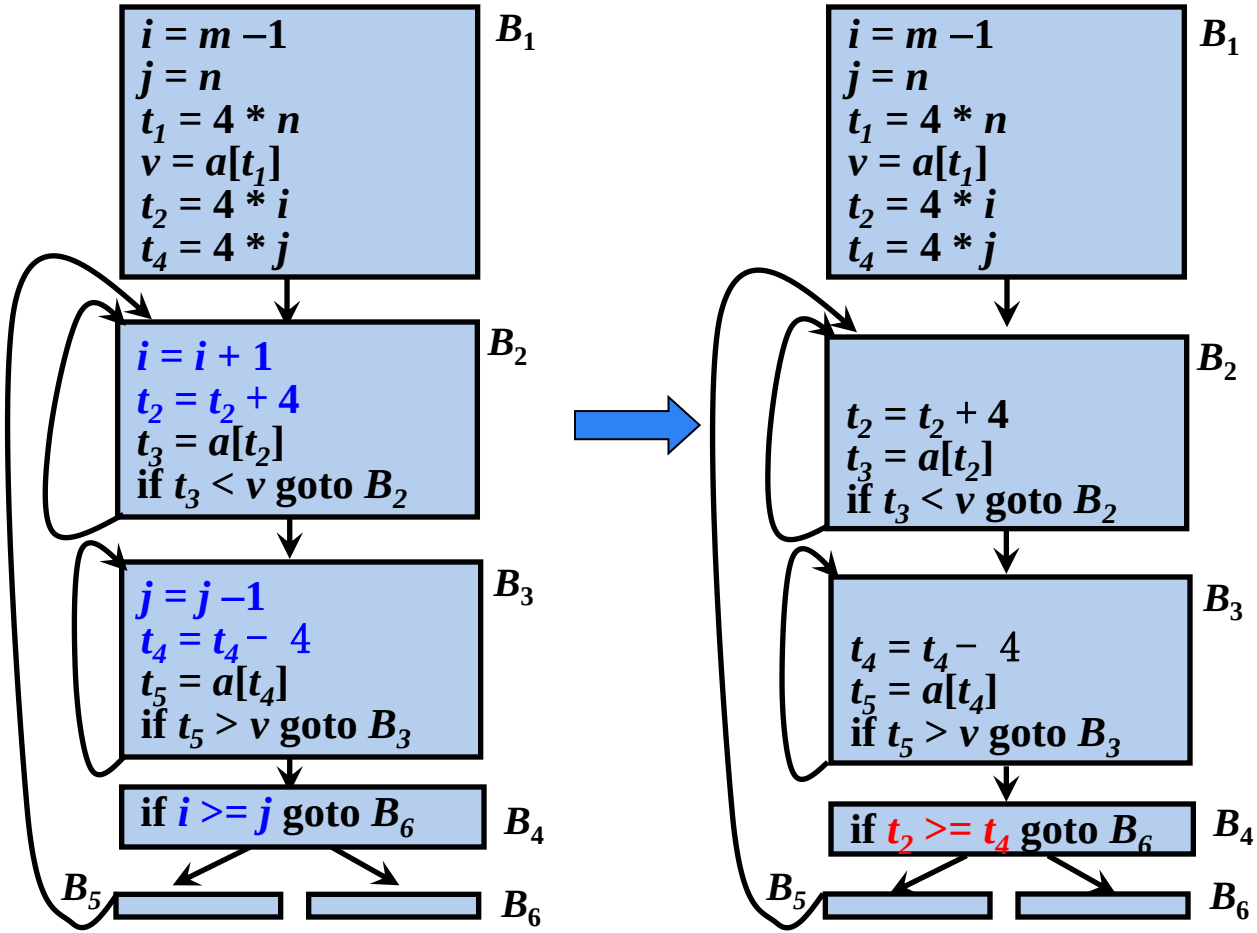

i = m –1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6
i = m –1
j = n
t1 = 4 * n
v = a[t1]
t2 = 4 * i
t4 = 4 * j
B1
i = i + 1
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2
B2
j = j –1
t4 = t4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

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

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

- 例

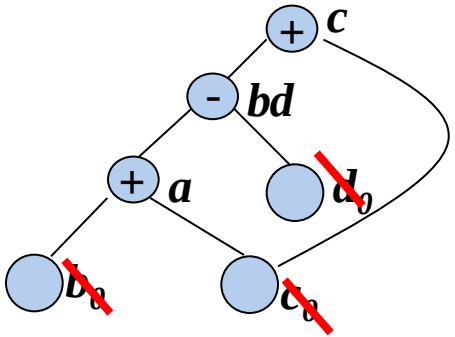
  $a = b + c$

  $b = a - d$

  $c = b + c$

  $d = a - d$

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



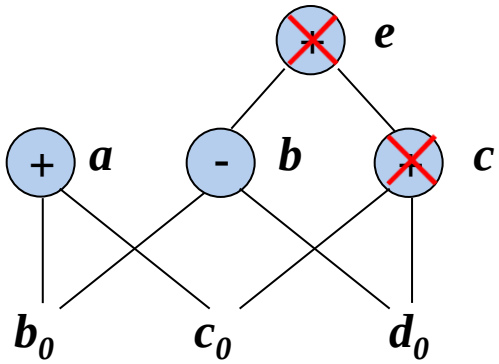
- 基本块中的每个语句s都对应一个内部结点N 和相应的子节点
  - 结点N的标号是s中的运算符；同时还有一个定值变量（表）被关联到N，表示s是在此基本块内最晚对表中变量进行定值的语句
  - N的子结点是基本块中在s之前、最后一个对s所使用的运算分量进行定值的语句对应的结点。如果s的某个运算分量在基本块内没有在s之前被定值，则这个运算分量对应的子结点就是代表该运算分量初始值的叶结点(为区别起见，叶节点的定值变量表中的变量加上下脚标0)
  - 在为语句x=y+z构造结点N的时候，如果x已经在某结点M的定值变量表中，则从M的定值变量表中删除变量x

基于基本块的 DAG 删除无用代码

- 从一个DAG上删除所有不包含活跃变量（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

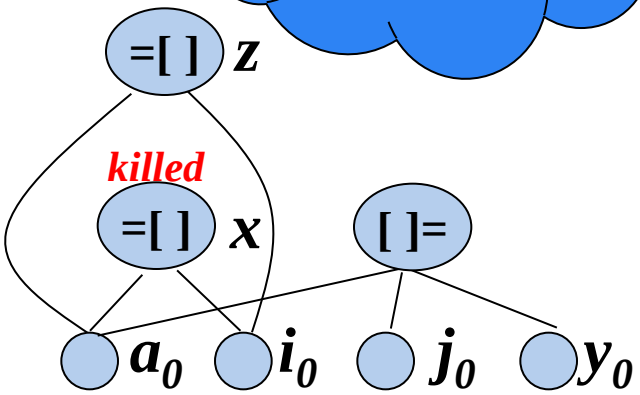
- 例

$x=a[i]$

$a[j]=y$

$z=a[i]$

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

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，没有被杀死

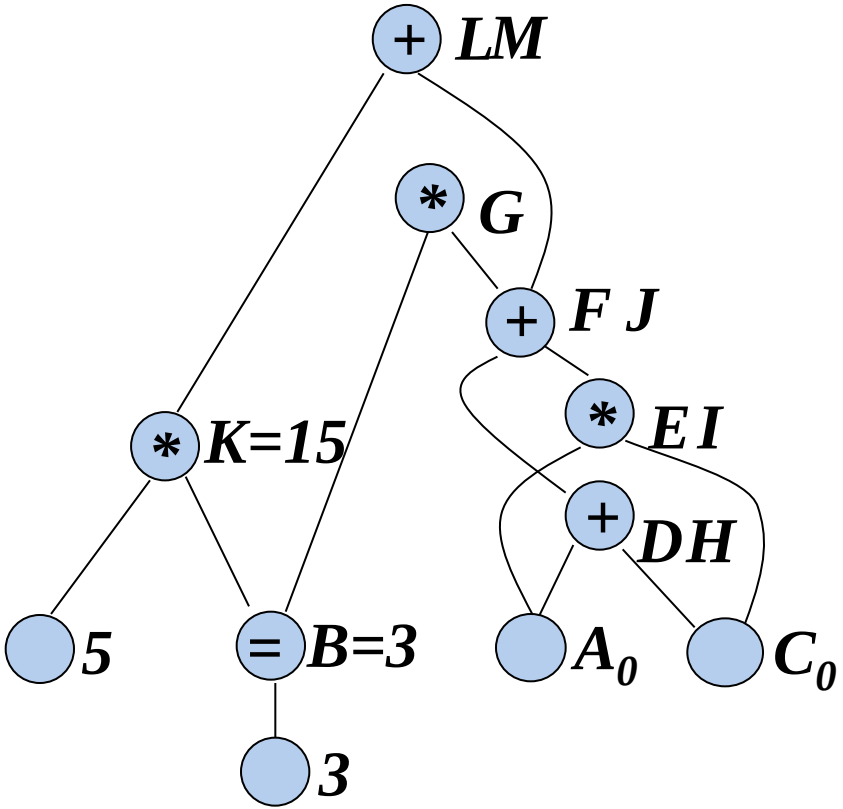
从 *DAG* 到基本块的重组

- 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

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 $M = L$

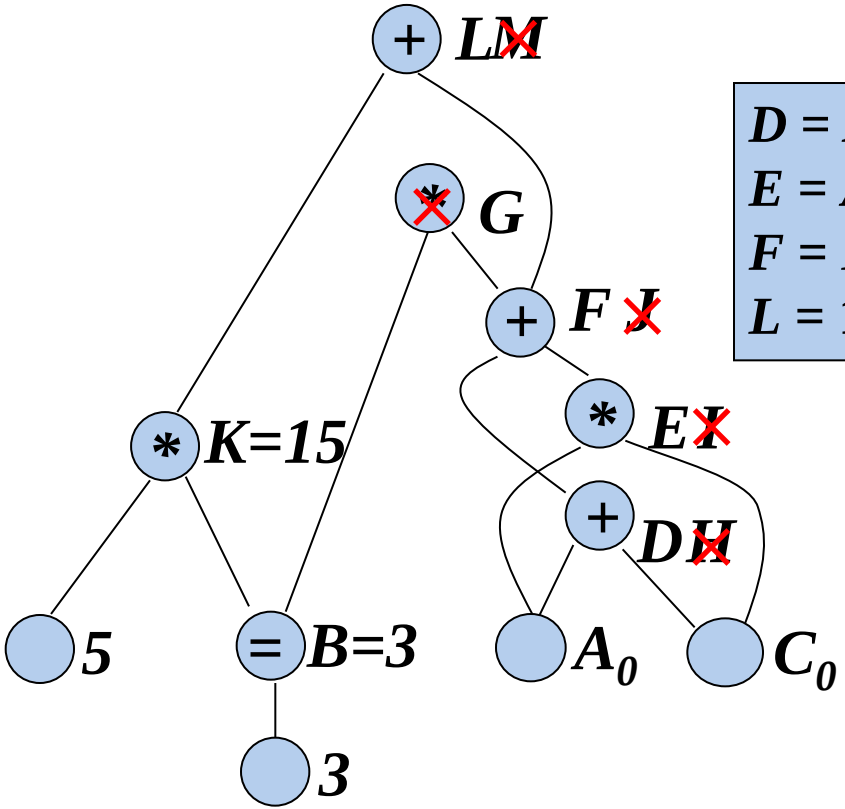


假设：仅变量L在基本块出口之后活跃

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

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

8.4 数据流分析(data-flow analysis)

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

数据流分析模式

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

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

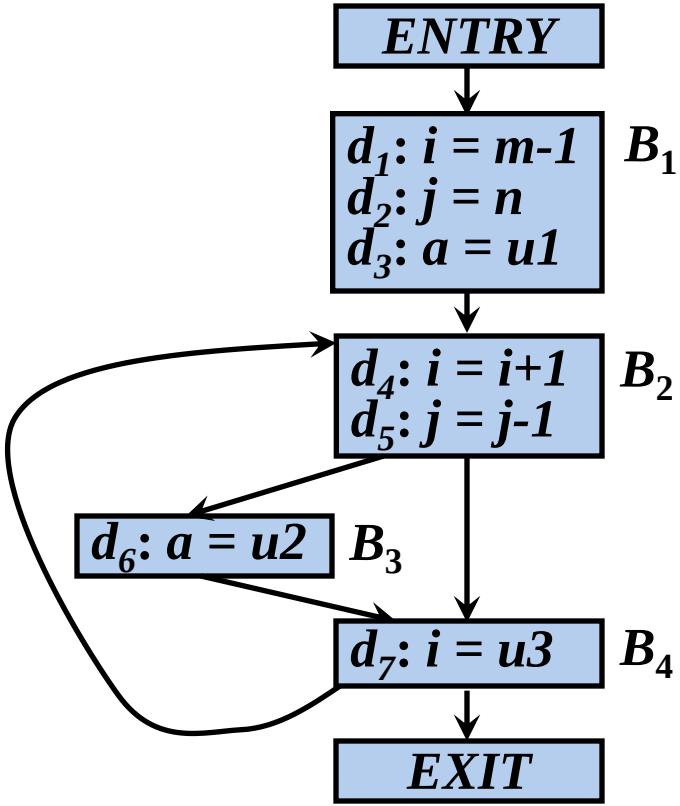
    $f_B = f_{s1}\cdot f_{s2}\cdot\ldots\cdot f_{sn}$

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

8.5 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
  - 直观地讲，如果某个变量x的一个定值d到达点p，在点p处使用的x的值可能就是由d最后赋予的

例：可以到达各基本块的入口处的定值



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

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

到达定值分析的主要用途

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

到达定值分析的主要用途

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

到达定值分析的主要用途

- 循环不变计算的检测
- 常量合并
- 判定变量x在p点上是否未经定值就被引用

“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

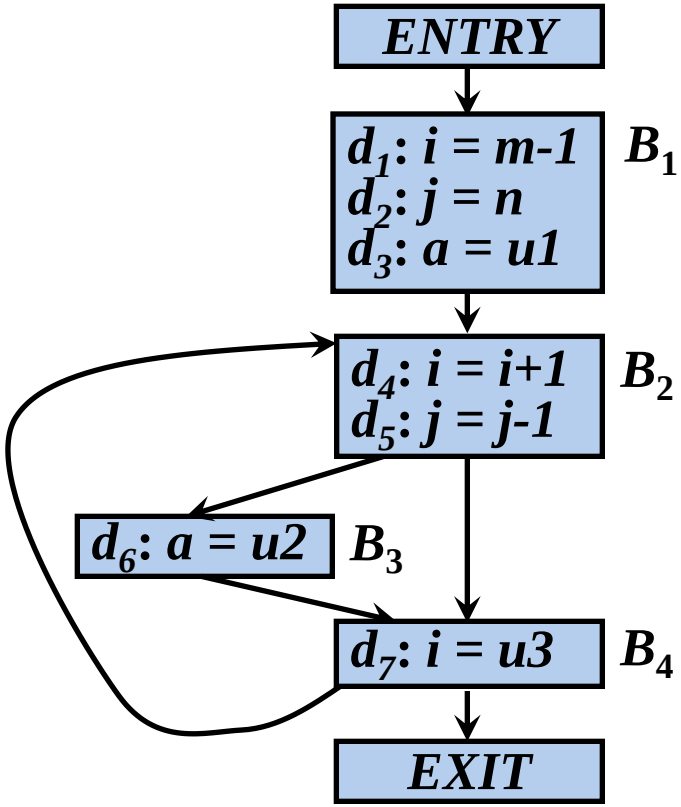
到达定值的传递函数

- 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

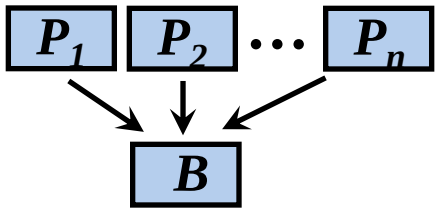
  $OUT[B]$：到达流图中基本块B的出口处的定值的集合

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

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

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

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

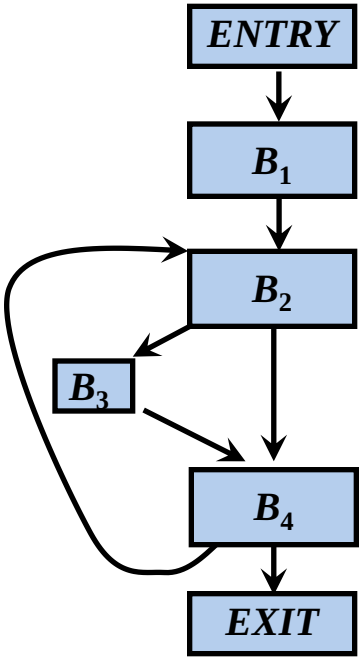


8.6 计算到达定值的迭代算法

- 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

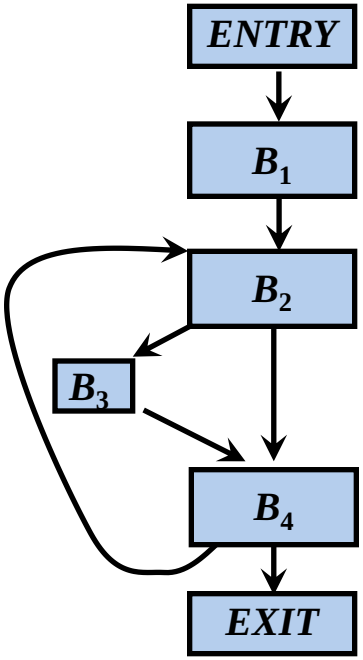


gen_{B1}={ d_1, d_2, d_3 }
$kill_{B1}$={ d_4, d_5, d_6, d_7 }
gen_{B2}={ d_4, d_5 }
$kill_{B2}$={ d_1, d_2, d_7 }
gen_{B3}={ d_6 }
$kill_{B3}$={ d_3 }
gen_{B4}={ d_7 }
$kill_{B4}$={ d_1, d_4 }

```
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]= ∪ OUT[P];   (P是B的一个前驱)
        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



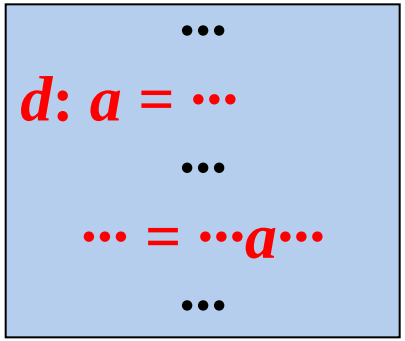
$gen_{B1}=\{ d_1, d_2, d_3 \}$
$kill_{B1}=\{ d_4, d_5, d_6, d_7 \}$
$gen_{B2}=\{ d_4, d_5 \}$
$kill_{B2}=\{ d_1, d_2, d_7 \}$
$gen_{B3}=\{ d_6 \}$
$kill_{B3}=\{ d_3 \}$
$gen_{B4}=\{ d_7 \}$
$kill_{B4}=\{ d_1, d_4 \}$

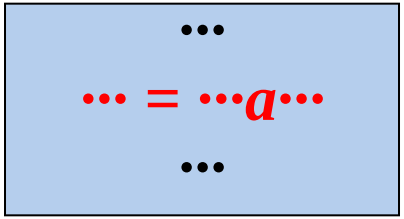
IN[B]	B_2	B_3	B_4
d_1	√	×	×
d_2	√	×	×
d_3	√	√	√
d_4	×	√	√
d_5	√	√	√
d_6	√	√	√
d_7	√	×	×

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

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





第八章 代码优化

到达定值方程的计算

哈尔滨工业大学 陈鄞