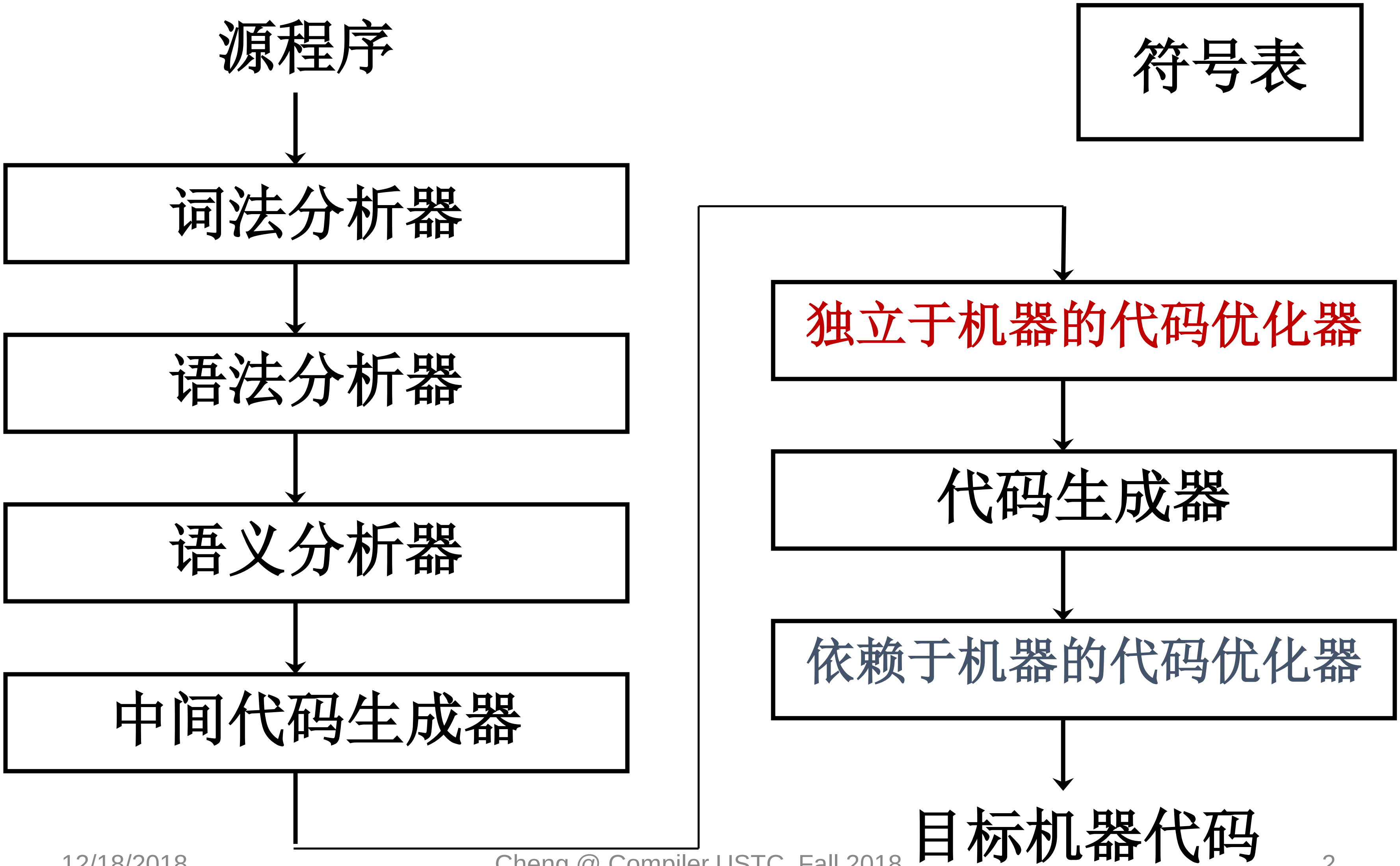
《编译原理与技术》
独立于机器的优化

计算机科学与技术学院
李 诚

16/12/2018

独立于机器的优化


源程序
符号表
词法分析器
语法分析器
语义分析器
中间代码生成器
独立于机器的代码优化器
代码生成器
依赖于机器的代码优化器
目标机器代码

本章内容

❑代码优化

- ❖通过程序变换（局部变换和全局变换）来改进程序，称为优化

❑代码优化的种类

- ❖基本块内优化、全局优化
- ❖公共子表达式删除、复写传播、死代码删除
- ❖循环优化

❑代码优化的实现方式

- ❖数据流分析及其一般框架、循环的识别和分析

优化的源头和主要种类

- 程序中存在许多程序员无法避免的冗余运算

  如A[i][j]和X.f1这样访问数组元素和结构体的域的操作

  - 编译后，这些访问操作展开成多步低级算术运算
  - 对同一个数据结构多次访问导致许多公共低级运算

- 主要种类

  - 公共子表达式删除(common subexpression elimination)
  - 复写传播(copy propagation)
  - 死代码删除(dead code elimination)
  - 代码外提(loop hoisting, code motion)

University of Science and Technology of China

快速排序程序片段如下，

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;
```

```
//B1
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
```

优化举例

快速排序程序片段如下，

```
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;
```

```
//B2
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto （5）
```

优化举例

快速排序程序片段如下，

```
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;
```

```
//B3
(9)   j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
```

优化举例

快速排序程序片段如下，

```
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;
```

```
//B4
(13)  if i >= j goto (23)
```

优化举例

快速排序程序片段如下，

```
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;
```

```
//B5
(14)  t6 := 4 * i
(15)  x := a[t6]
(16)  t7 := 4 * i
(17)  t8 := 4 * j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4 * j
(21)  a[t10] := x
(22)  goto (5)
```

优化举例

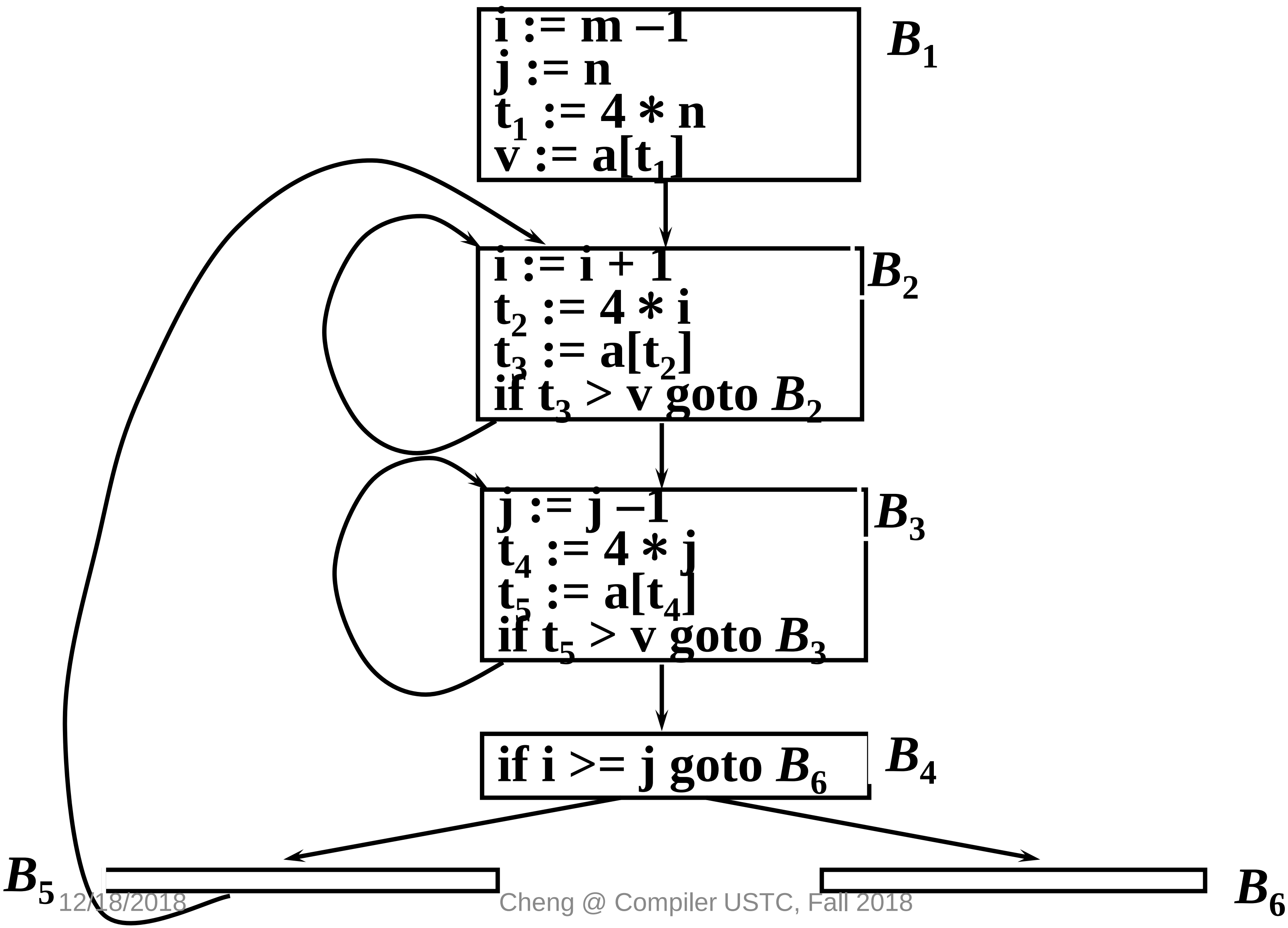
快速排序程序片段如下，

```
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;

```
//B6
(23)  t11 := 4 * i
(24)  x := a[t11]
(25)  t12 := 4 * i
(26)  t13 := 4 * n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4 * n
(30)  a[t15] := x
```

优化举例-流图


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

□公共子表达式删除 - 基本块内

B_5

t_6 := 4 * i
x := a[t_6]
*t_7 := 4 * i*
t_8 := 4 * j
t_9 := a[t_8]
a[t_7] := t_9
*t_{10} := 4 * j*
a[t_{10}] := x
goto B_2

变换

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

❑公共子表达式删除 - 基本块内

B_5

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_6

t_{11} := 4 * i
x := a[t_{11}]
t_{13} := 4 * n
t_{14} := a[t_{13}]
a[t_{11}] := t_{14}
a[t_{13}] := x

❑公共子表达式删除 - 基本块间

$t_2 := 4 * i : B_2 \rightarrow B_5$

$t_2 := 4 * i : B_2 \rightarrow B_6$

$t_4 := 4 * j : B_3 \rightarrow B_5$

$t_1 := 4 * n : B_1 \rightarrow B_6$

B_5

$t_6 := 4 * i$
$x := a[t_6]$
$t_8 := 4 * j$
$t_9 := a[t_8]$
$a[t_6] := t_9$
$a[t_8] := x$
goto B_2

变换

B_6

$t_{11} := 4 * i$
$x := a[t_{11}]$
$t_{13} := 4 * n$
$t_{14} := a[t_{13}]$
$a[t_{11}] := t_{14}$
$a[t_{13}] := x$

优化举例

公共子表达式删除 - 基本块间

t3:=a[t2] : B2 → B5

t3:=a[t2] : B2 → B6

t5:= a[t4] : B3 → B5

B_5

$x := a[t_2]$
$t_9 := a[t_4]$
$a[t_2] := t_9$
$a[t_4] := x$
goto B_2

B_6

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

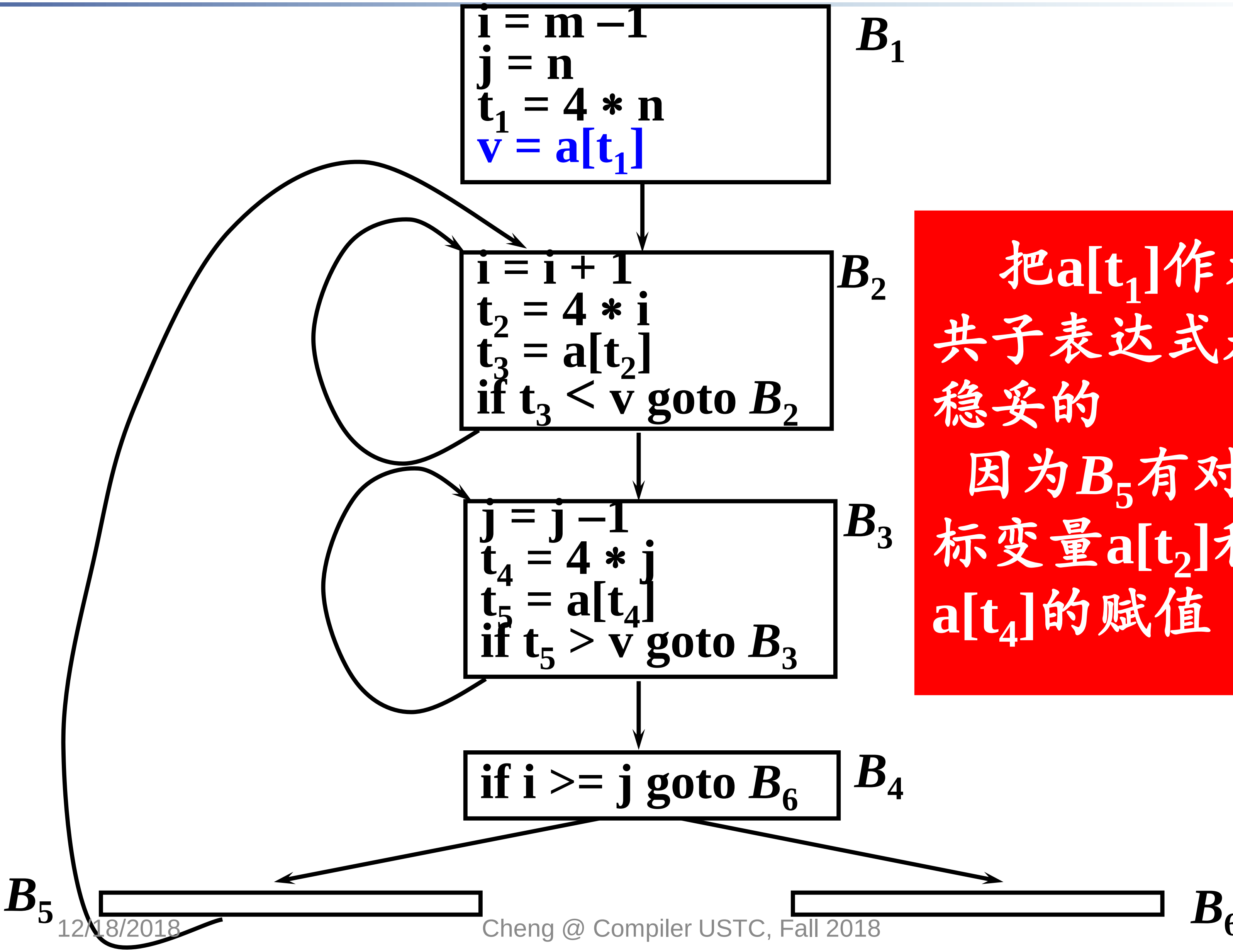
变换

❑公共子表达式删除 - 基本块间

❖B_1中 v := a[t1] 能否作为公共子表达式?

B_5
x := t_3
a[t_2] := t_5
a[t_4] := x
goto B_2

B_6
x := t_3
t_{14} := a[t_1]
a[t_2] := t_{14}
a[t_1] := x

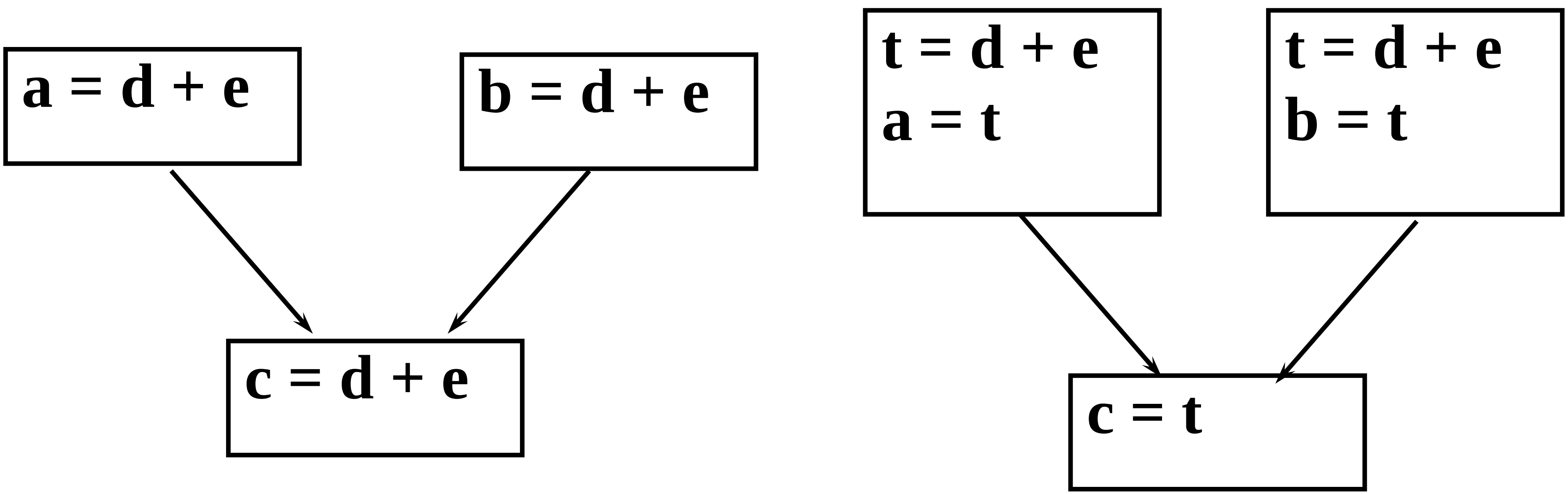


把$a[t_1]$作为公共子表达式是不稳妥的

因为B_5有对下标变量$a[t_2]$和$a[t_4]$的赋值

优化举例-复写传播

- 复写语句：形式为f = g的赋值
- 优化过程中会大量引入复写



删除局部公共子表达式期间引进复写

优化举例-复写传播

- 复写语句：形式为f = g的赋值
- 优化过程中会大量引入复写
- 复写传播变换的做法是在复写语句f = g后，尽可能用g代表f

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

优化举例-复写传播

中国科学技术大学 University of Science and Technology of China

- ❑复写语句：形式为f = g的赋值
- ❑优化过程中会大量引入复写
- ❑复写传播变换的做法是在复写语句f = g后，尽可能用g代表f
- ❑复写传播变换本身并不是优化，但它给其它优化带来机会
  - ❖常量合并（编译时可完成的计算）
  - ❖死代码删除

pi = 3.14

…

y = pi * 5

优化举例-死代码删除

- 死代码是指计算的结果决不被引用的语句
- 一些优化变换可能会引起死代码

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

```
debug = true;                |    debug = false;
. . .                        |    . . .
if (debug) print …           |    if (debug) print …
```

靠优化来保证目标代码中没有该条件语句部分

优化举例-死代码删除

- 死代码是指计算的结果决不被引用的语句
- 一些优化变换可能会引起死代码

例：复写传播可能会引起死代码删除

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

优化举例-循环优化

- 代码外提是循环优化的一种

例：while (i <= limit – 2) …

代码外提后变换成

```
t = limit – 2;
while (i <= t ) …
```

优化举例-循环优化

❑强度削弱和归纳变量删除

- ❖j和t_4的值步伐一致地变化
- ❖这样的变量叫做归纳变量
- ❖在循环中有多个归纳变量时，也许只需要留下一个
- ❖对本例可以先做强度削弱它给删除归纳变量创造机会
  - ➢用廉价运算替换(加替换乘)

B_3

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

强度削弱和归纳变量删除

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

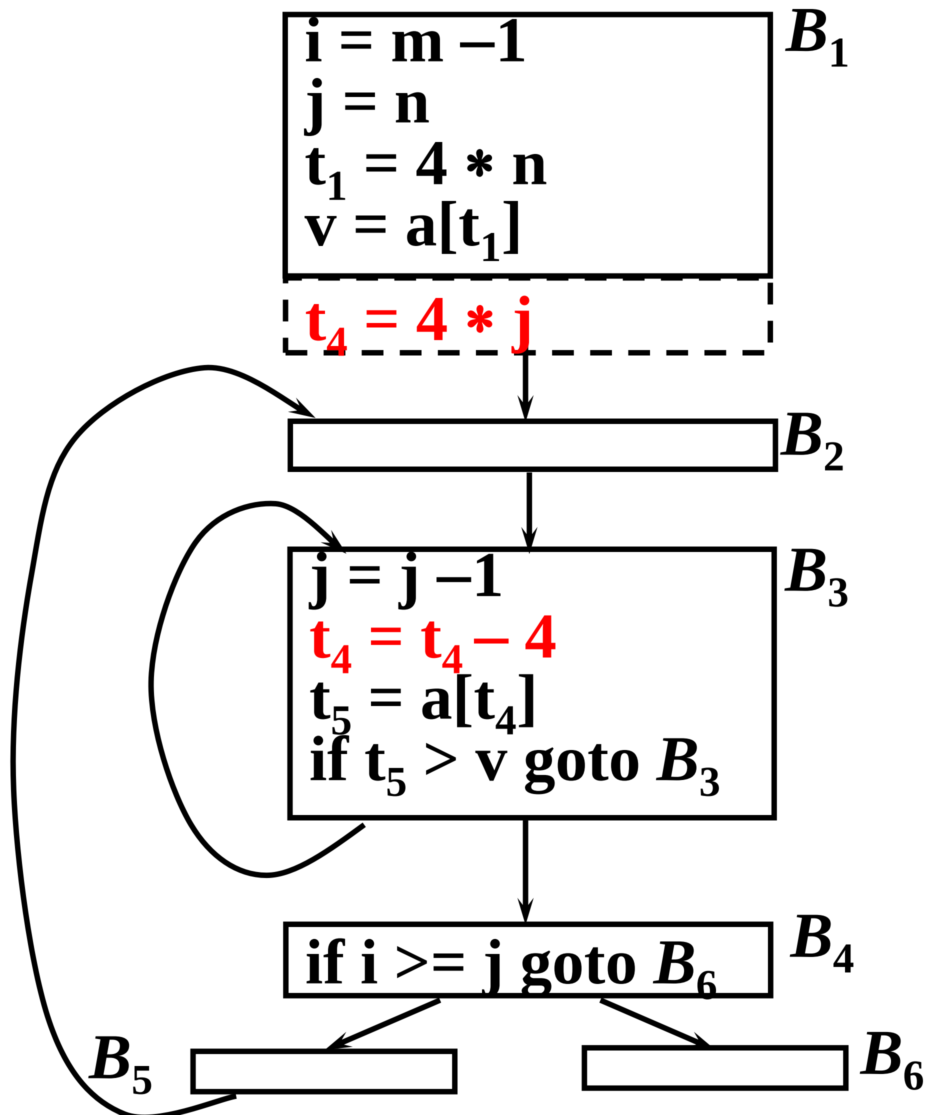
B_3

除第一次外，

$t_4 == 4 * j$在B_3的入口一定保持

在$j = j - 1$后，

关系$t_4 == 4 * j + 4$也保持



9强度削弱和归纳变量删除

B_1
```
i = m –1
j = n
t1 = 4 * n
v = a[t1]
t4 = 4 * j
```

B_2
```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B_3
```
j = j –1
t4 = t4–4
t5 = a[t4]
if t5 > v goto B3
```

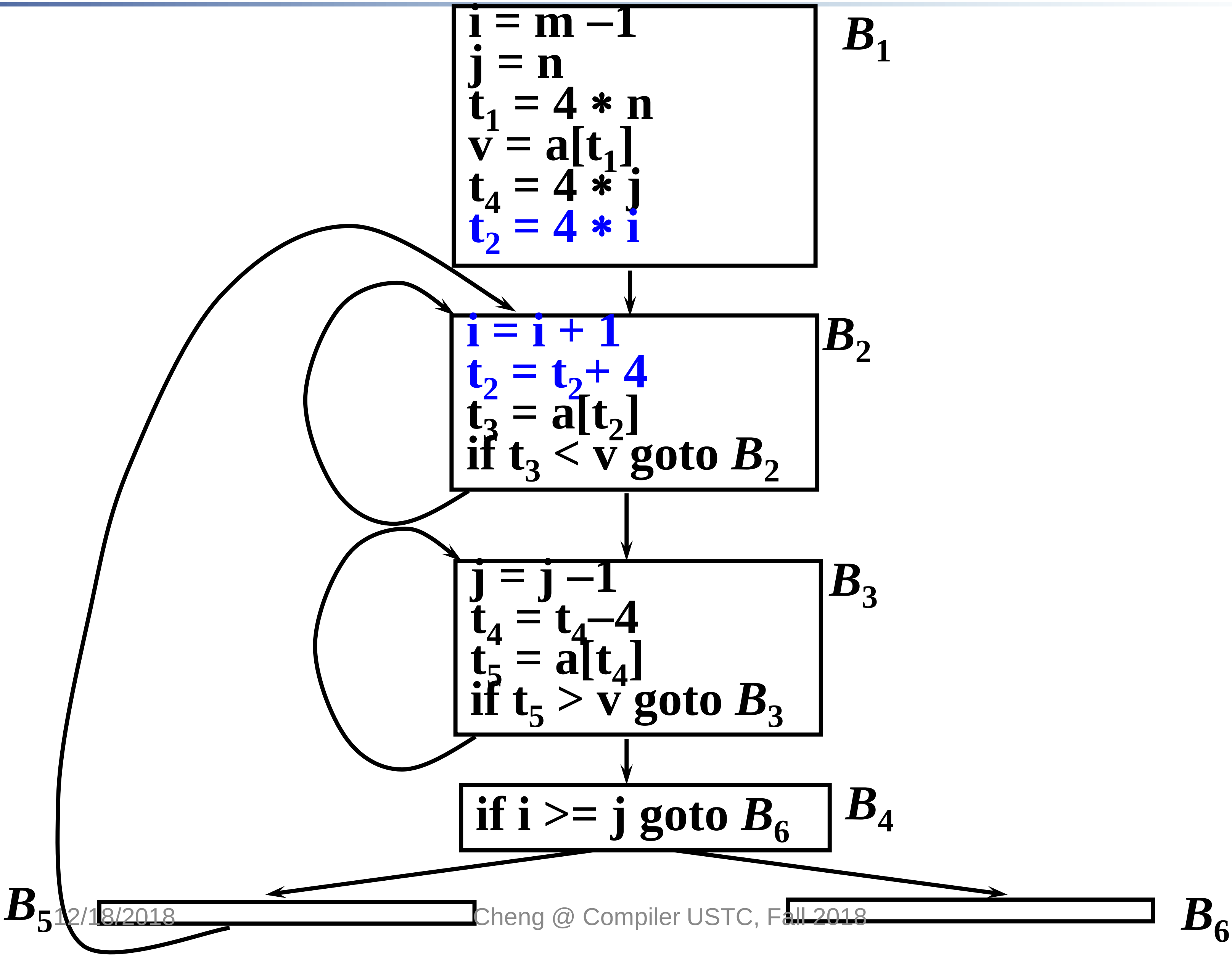
B_4
```
if i >= j goto B6
```

B_5

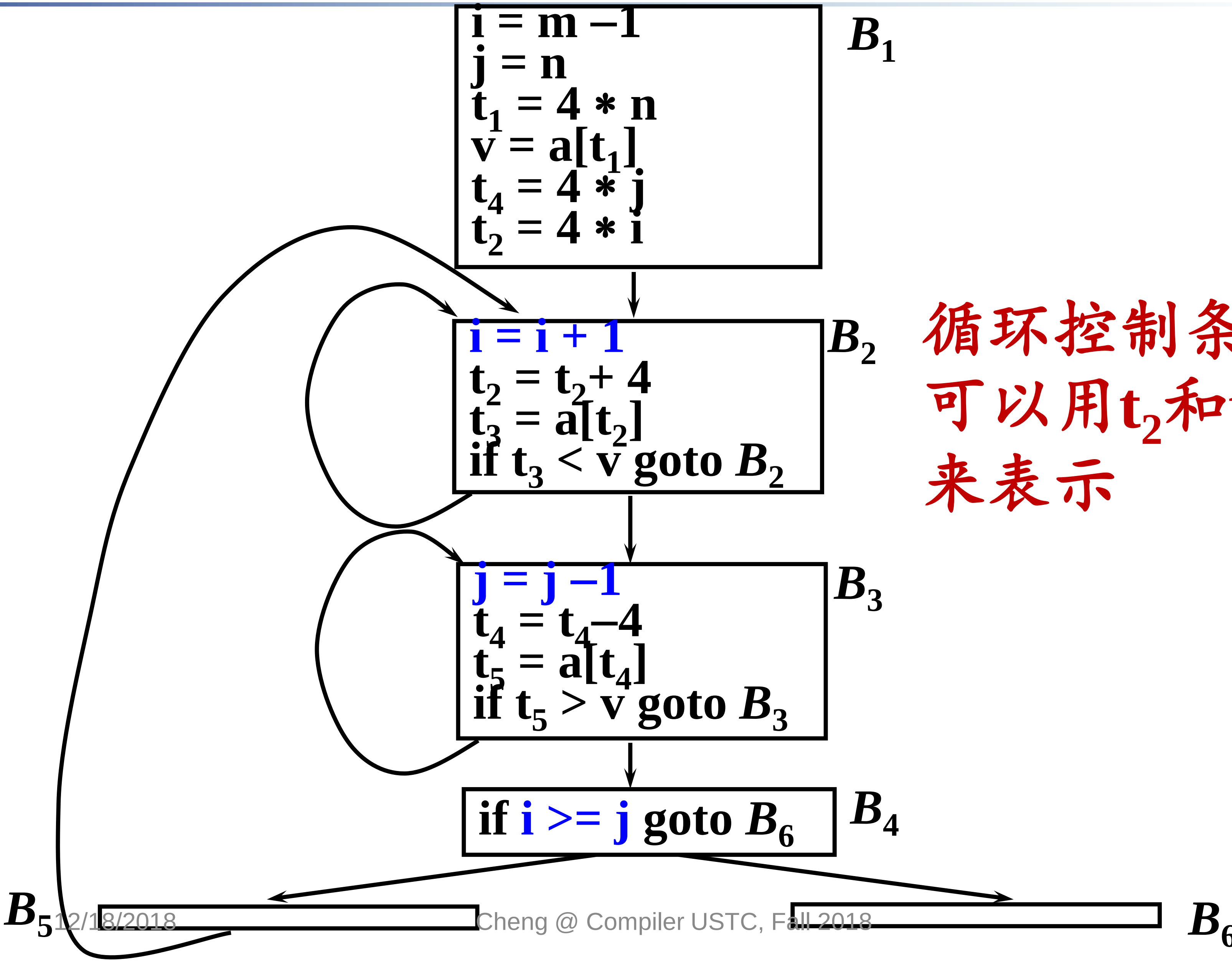
B_6

B_2也可以进行类似的变换

强度削弱和归纳变量删除



强度削弱和归纳变量删除



循环控制条件可以用t_2和t_4来表示

优化后的结果

B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
$t_2 = 4 * i$
$t_4 = 4 * j$

B_2

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

B_3

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

B_4

if $t_2 >= t_4$ goto B_6

B_5

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

B_6

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

本章内容

University of Science and Technology of China

❑代码优化

- ❖通过程序变换（局部变换和全局变换）来改进程序，称为优化

❑代码优化的种类

- ❖基本块内优化、全局优化
- ❖公共子表达式删除、复写传播、死代码删除
- ❖循环优化

❑代码优化的实现方式

- ❖数据流分析及其一般框架、循环的识别和分析

程序分析

- ❑基本块、流图
- ❑控制流分析
- ❑数据流分析

❑流图上的点(程序点)

- ❖基本块中，两个相邻的语句之间为程序的一个点
- ❖基本块的开始点和结束点

❑流图上的路径

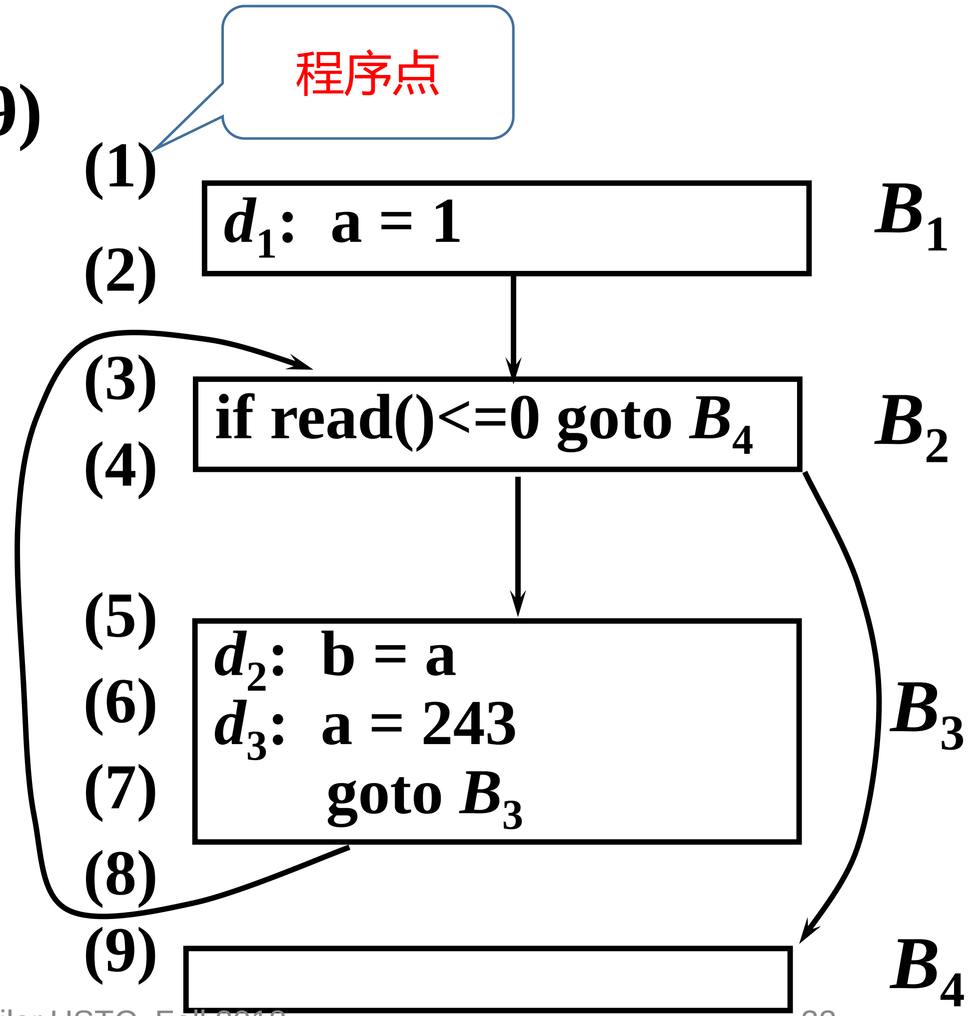
- ❖点序列$p_1, p_2, \ldots, p_n$，对1和n - 1间的每个i，满足

(1) p_i是先于一个语句的点，p_{i+1}是同一块中位于该语句后的点，或者

(2) p_i是某块的结束点，p_{i+1}是后继块的开始点

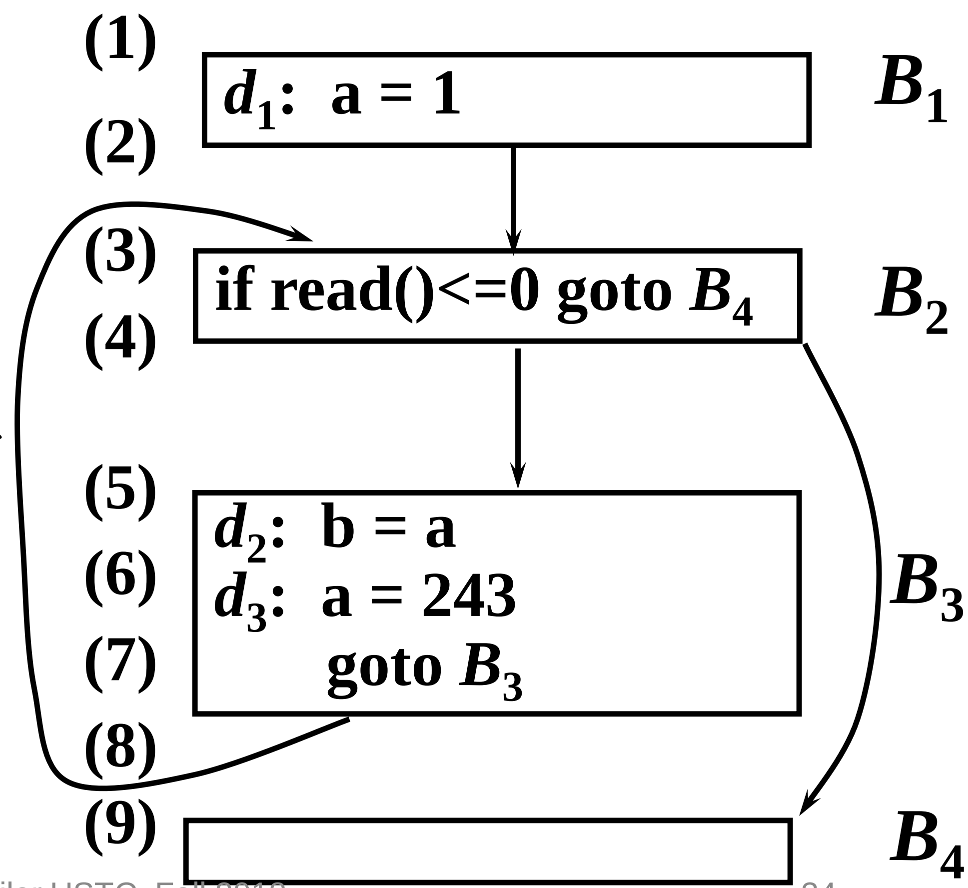
❑流图上路径实例

- (1, 2, 3, 4, 9)
- (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)
- (1, 2, 3, 4, 5, 6, 7, 8,
  3, 4, 5, 6, 7, 8, 3, 4, 9)
- (1, 2, 3, 4, 5, 6, 7, 8,
  3, 4, 5, 6, 7, 8,
  3, 4, 5, 6, 7, 8, …)
  路径长度无限
- **路径数无限**



数据流分析介绍

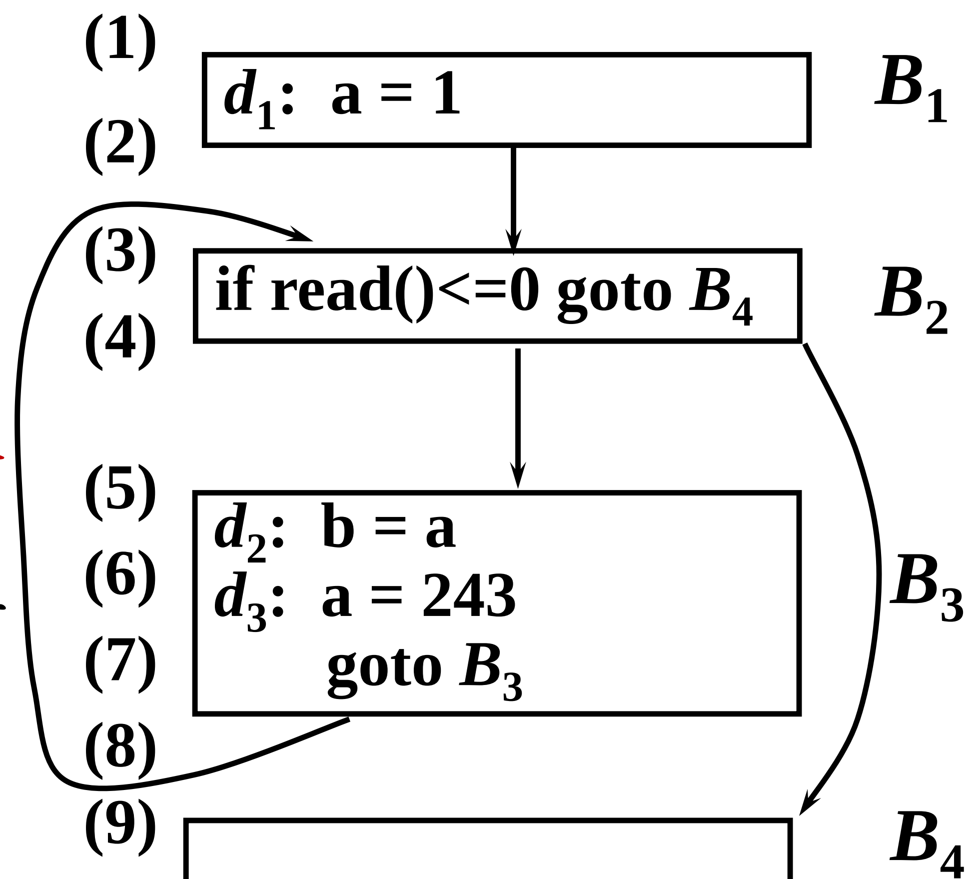
- 分析程序的行为时，必须在其流图上考虑**所有的执行路径**（在调用或返回语句被执行时，还需要考虑执行路径在多个流图之间的跳转）
  - 通常，从流图得到的程序**执行路径数无限**，且执行路径长度没有有限的上界



数据流分析介绍

中国科学技术大学 University of Science and Technology of China

- 分析程序的行为时，必须在其流图上考虑**所有的执行路径**（在调用或返回语句被执行时，还需要考虑执行路径在多个流图之间的跳转）
  - 每个程序点的**不同状态数也可能无限**
  - 程序状态：存储单元到值的映射



数据流分析介绍

- 问题：把握所有执行路径上的所有程序状态一般来说是不可能的
- 解决方案：数据流分析抽取解决特定任务所需信息，以总结出用于该分析目的的一组有限的事实
  - 并且这组事实和到达这个程序点的路径无关，即从任何路径到达该程序点都有这样的事实
  - 分析的目的不同，从程序状态提炼的信息也不同
  - 例如，常量传播分析是力求判定对一个特定变量的所有赋值在某个特定程序点是否总是给定相同的常数值

❑数据流值

- ❖数据流分析总把程序点和数据流值联系起来
- ❖数据流值代表在程序点能观测到的所有可能程序状态集合的一个抽象

- ❖语句*s*前后两点数据流值用IN[*s*]和OUT[*s*]来表示
- ❖数据流问题就是通过基于语句语义的约束（迁移函数）和基于控制流的约束来寻找所有语句*s*的IN[*s*]和OUT[*s*] 的一个解

❑语义约束-迁移函数 f

❖语句前后两点的数据流值受该语句的语义约束

❖若沿执行路径正向传播，则 $\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$

❖若沿执行路径逆向传播，则 $\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$

若基本块 B 由语句 $s_1, s_2, \ldots, s_n$ 依次组成，则

❖$\mathrm{IN}[s_i+1] = \mathrm{OUT}[s_i]$，$i = 1, 2, \ldots, n-1$（逆向…）

❖$f_B = f_n \circ \ldots \circ f_2 \circ f_1$ （逆向 $f_B = f_1 \circ \ldots \circ f_{n-1} \circ f_n$）

❖$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$ （逆向 $\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$）

考虑的是在语句执行后输入输出之间的变化关系

- 控制流约束
  - 正向传播

    $\mathrm{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \mathrm{OUT}[P]$（可能用I）
  - 逆向传播

    $\mathrm{OUT}[B] = \cup_{S\text{是}B\text{的后继}} \mathrm{IN}[S]$（可能用I）
- 约束方程组的解通常不是唯一的
  - 求解的目标是要找到满足这两组约束（控制流约束和迁移约束）的最“精确”解

考虑的是在其他语句或块对于输入的影响和本次执行的输出对其他语句和块的影响

到达-定值分析

- 到达一个程序点的所有定值(gen)
  - 可用来判断一个变量在某程序点是否为常量
  - 可用来判断一个变量在某程序点是否无初值
- 别名给到达-定值的计算带来困难，因此，本章其余部分仅考虑变量无别名的情况
- 定值的注销(kill)
  - 在一条执行路径上，对x的赋值注销先前对x的所有赋值

□点(5)所有程序状态:

- ❖a ∈ {1, 243}
- ❖由$\{d_1, d_3\}$定值

(1) 到达-定值

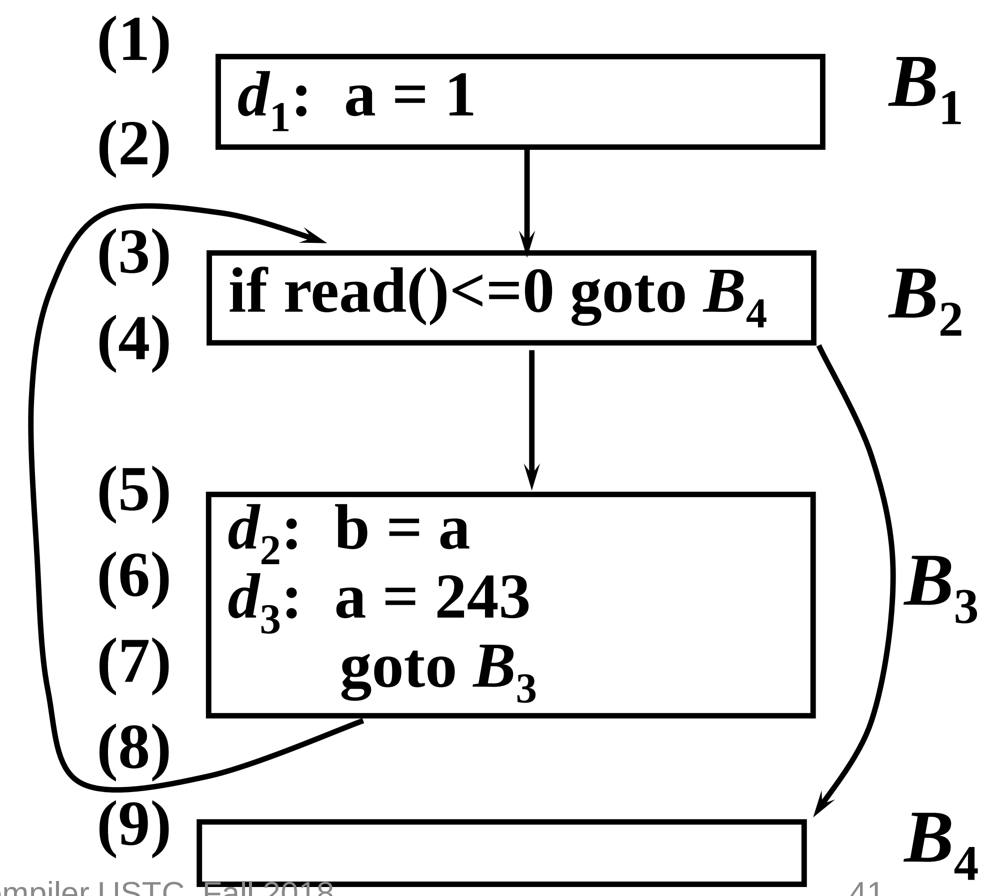
- $\{d_1, d_3\}$的定值

到达点(5)

(2) 常量合并

- a在点(5)不是

常量



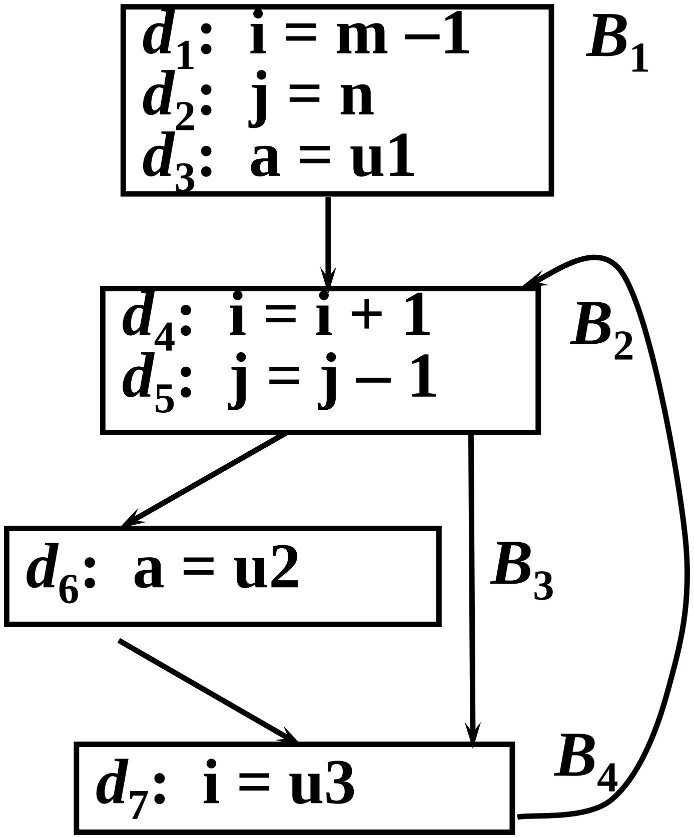
- *gen*和*kill*分别表示一个基本块生成和注销的定值

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$



到达-定值分析

❑基本块的*gen*和*kill*是怎样计算的

❖对三地址指令 *d*: u = v + w，它的状态迁移函数是

$$f_d(x) = gen_d \cup (x - kill_d)$$

❖若： $f_1(x) = gen_1 \cup (x - kill_1)$, $f_2(x) = gen_2 \cup (x - kill_2)$

则： $f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2)$

$= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2))$

❖若基本块*B*有*n*条三地址指令

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

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

到达-定值分析

□到达-定值的数据流等式

- ❖ gen_B：B中能到达B的结束点的定值语句
- ❖ $kill_B$：整个程序中决不会到达B结束点的定值
- ❖ IN[B]：能到达B的开始点的定值集合
- ❖ OUT[B]：能到达B的结束点的定值集合

两组等式（根据*gen*和*kill*定义IN和OUT）

- ❖ $\mathrm{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \mathrm{OUT}[P]$
- ❖ $\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$
- ❖ $\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$

□到达-定值方程组的迭代求解，最终到达不动点

到达-定值的迭代计算算法

// 正向数据流分析

(1) OUT[ENTRY] = Ø;

(2) for (除了ENTRY以外的每个块B) OUT[B] = Ø;

(3) while (任何一个OUT出现变化)

(4) for (除了ENTRY以外的每个块B) {

(5) $IN[B] = \cup_{P是B的前驱} OUT[P]$;

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

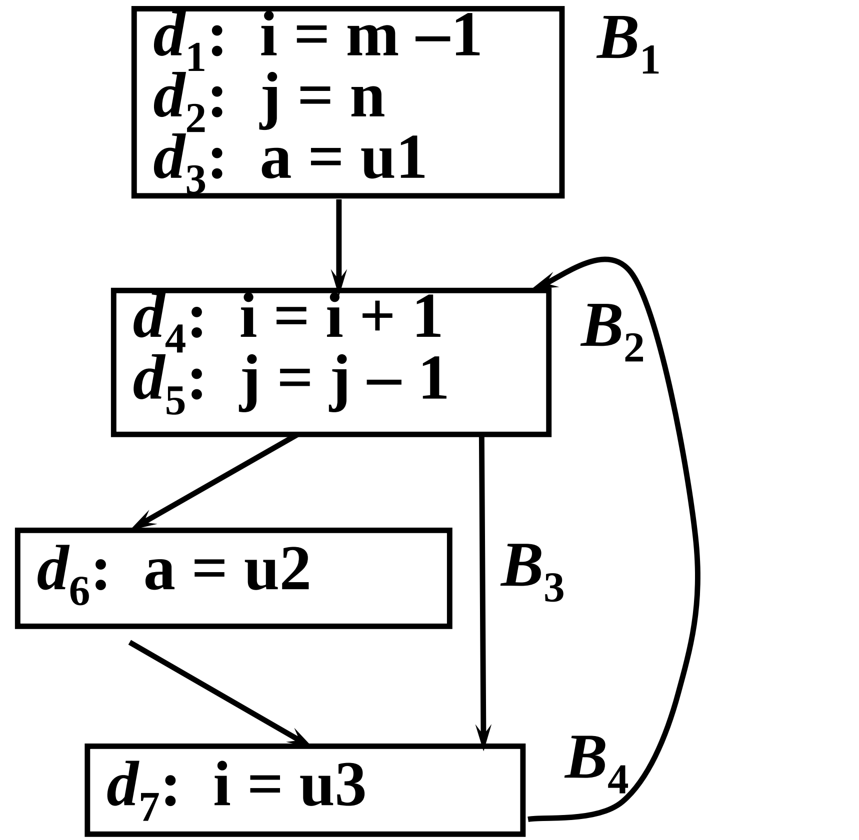
(7) $OUT[B] = f_B(IN[B])$;}

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1		000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$

$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$

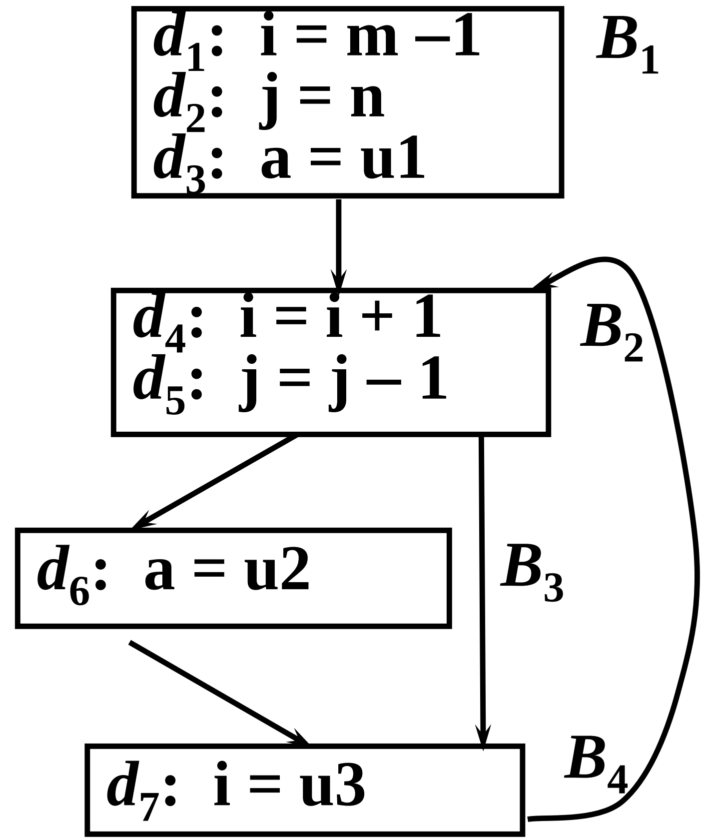
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$

$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3] = \{d_3\}$

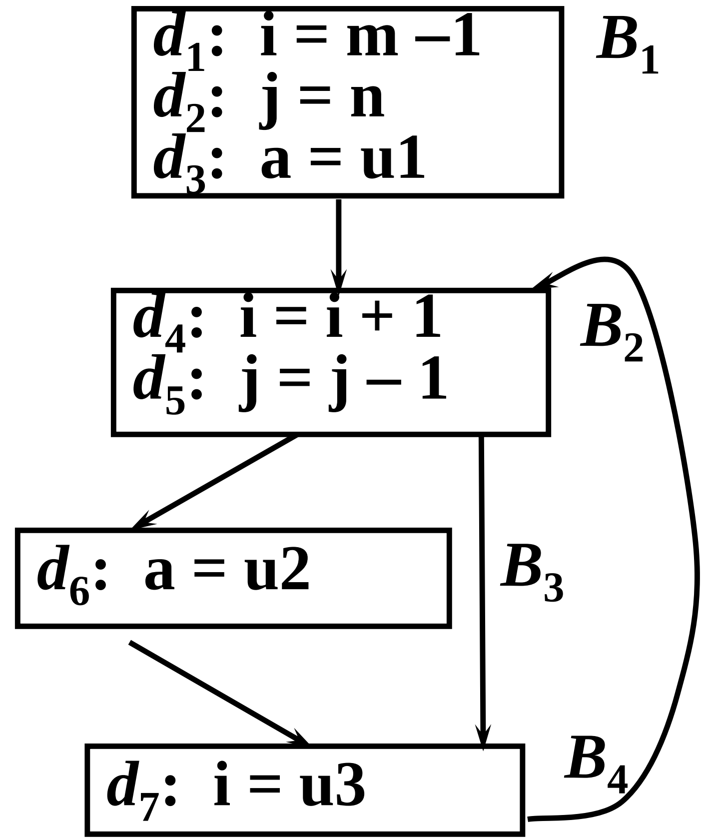
$gen\ [B_4] = \{d_7\}$

$kill\ [B_4] = \{d_1, d_4\}$

$\mathrm{IN}[B] = \cup_{P是B的前驱}\ \mathrm{OUT}[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

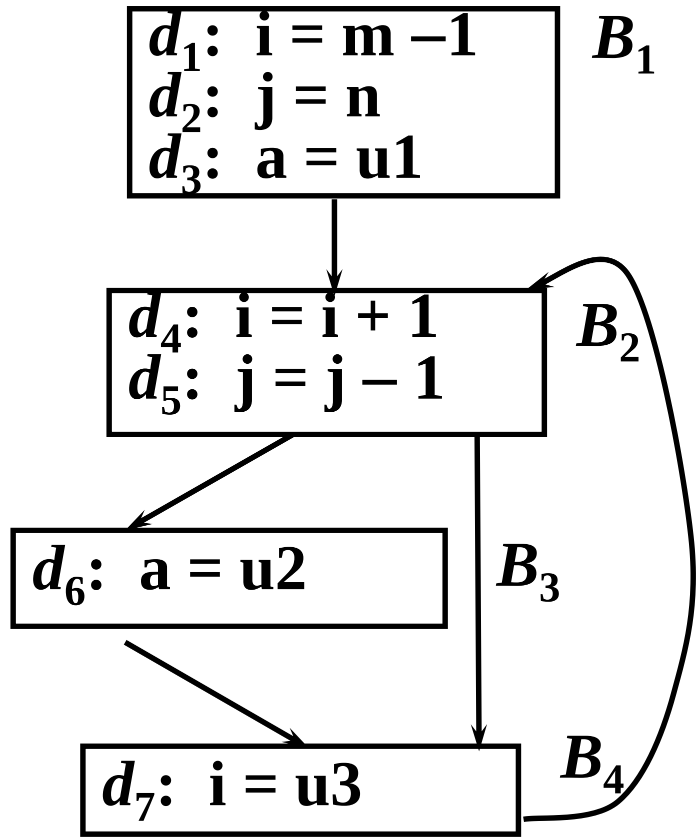
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	000 0000
B_3		000 0000
B_4		000 0000



$gen[B_1] = \{d_1, d_2, d_3\}$
$kill[B_1]=\{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4, d_5\}$
$kill[B_2] = \{d_1, d_2, d_7\}$

$gen[B_3] = \{d_6\}$
$kill[B_3] = \{d_3\}$

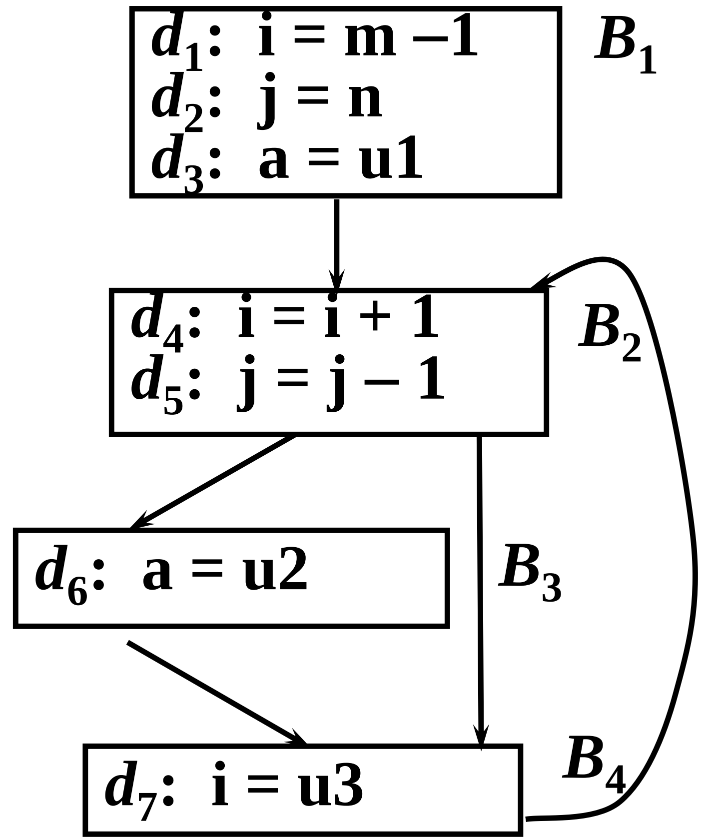
$gen[B_4] = \{d_7\}$
$kill[B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

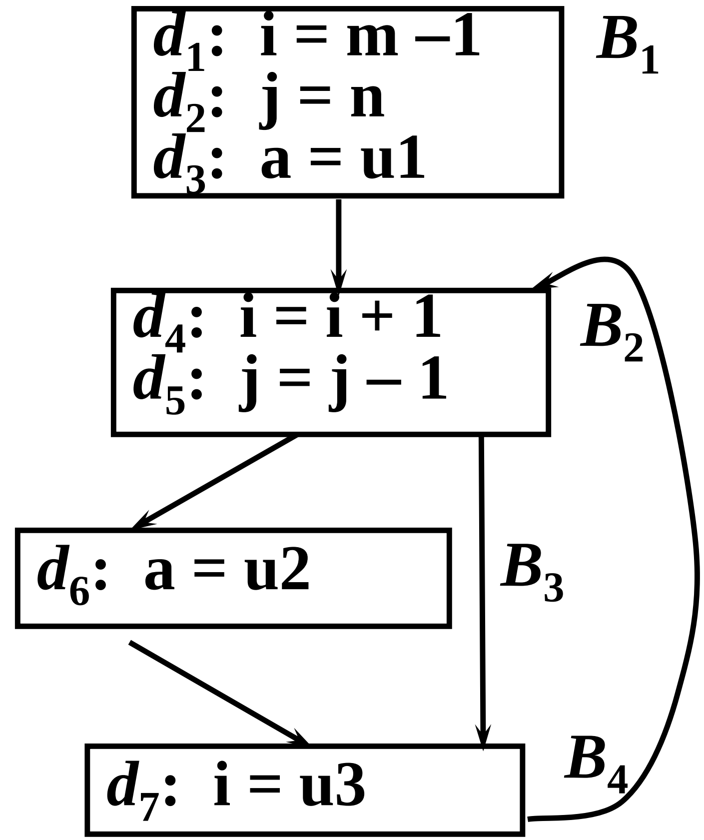
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$$IN[B] = \cup_{P是B的前驱} OUT[P]$$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

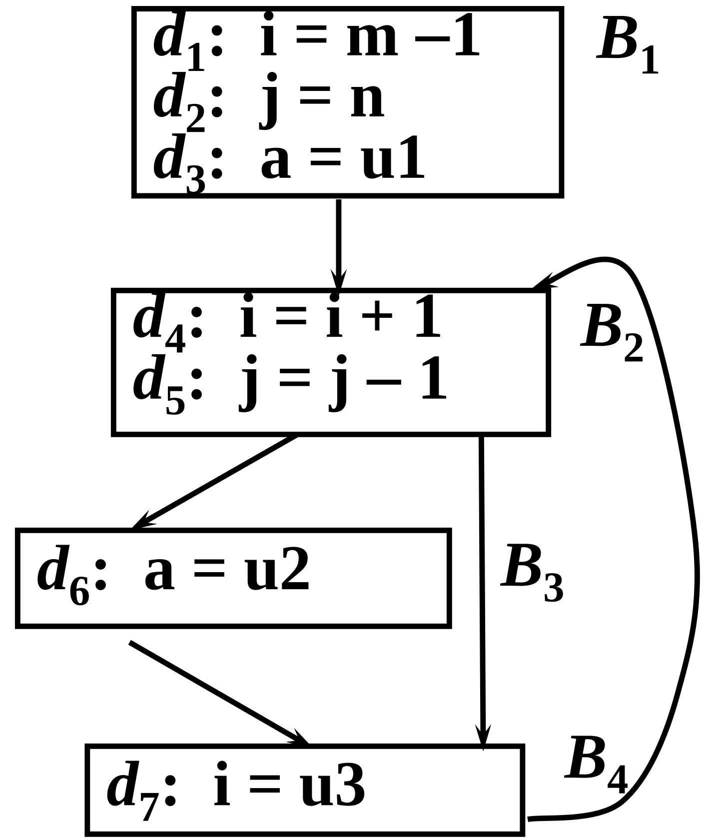
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

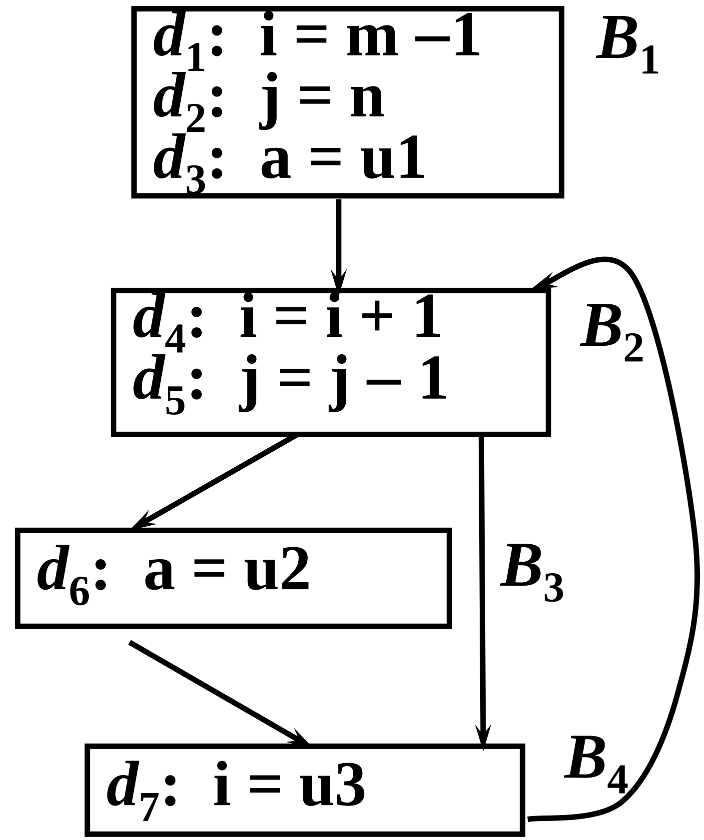
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$\text{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	000 0000



$gen[B_1] = \{d_1, d_2, d_3\}$
$kill[B_1] = \{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4, d_5\}$
$kill[B_2] = \{d_1, d_2, d_7\}$

$gen[B_3] = \{d_6\}$
$kill[B_3] = \{d_3\}$

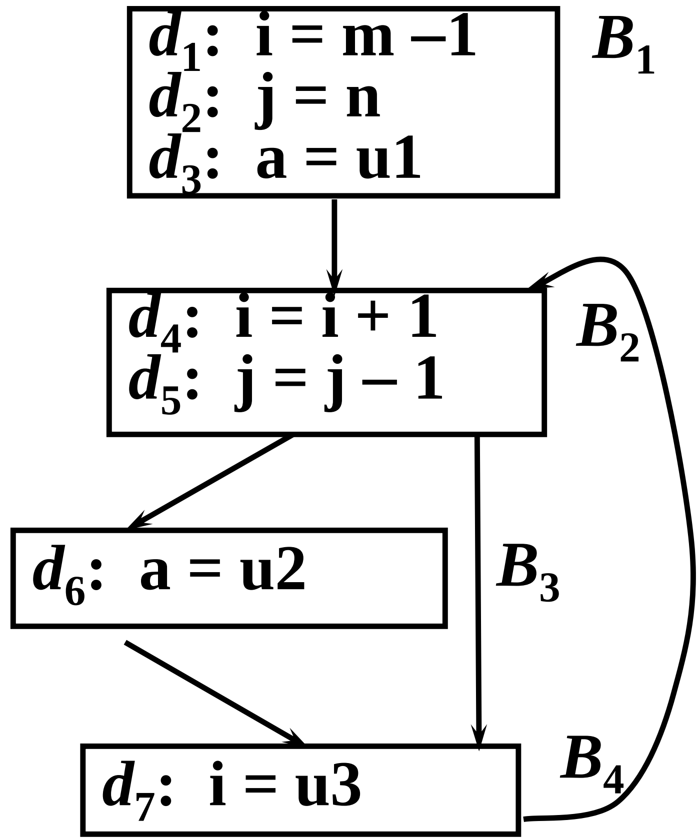
$gen[B_4] = \{d_7\}$
$kill[B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	001 0111



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

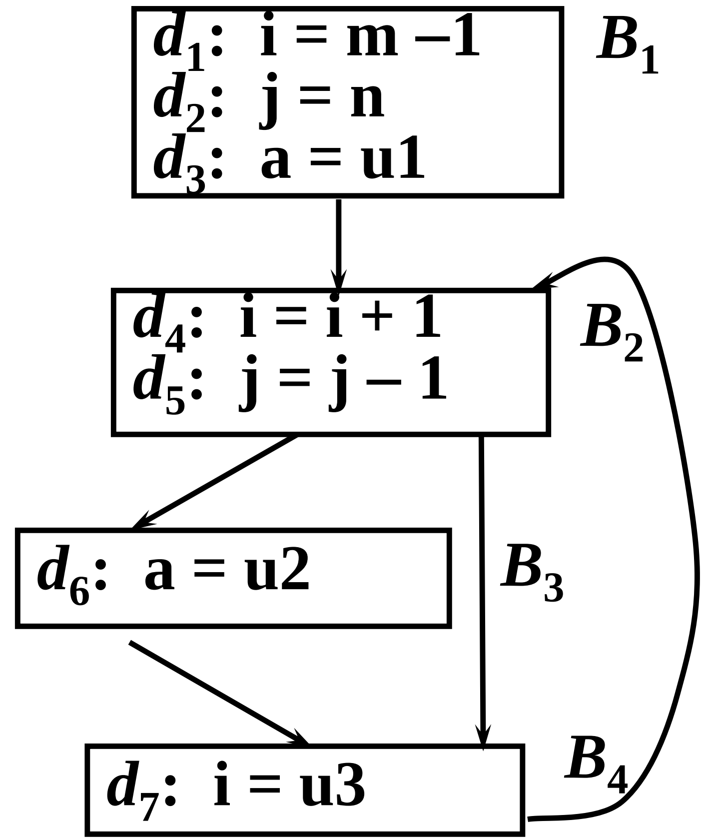
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P\text{是}B\text{的前驱}} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1100
B_3	001 1100	000 1110
B_4	001 1110	001 0111



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

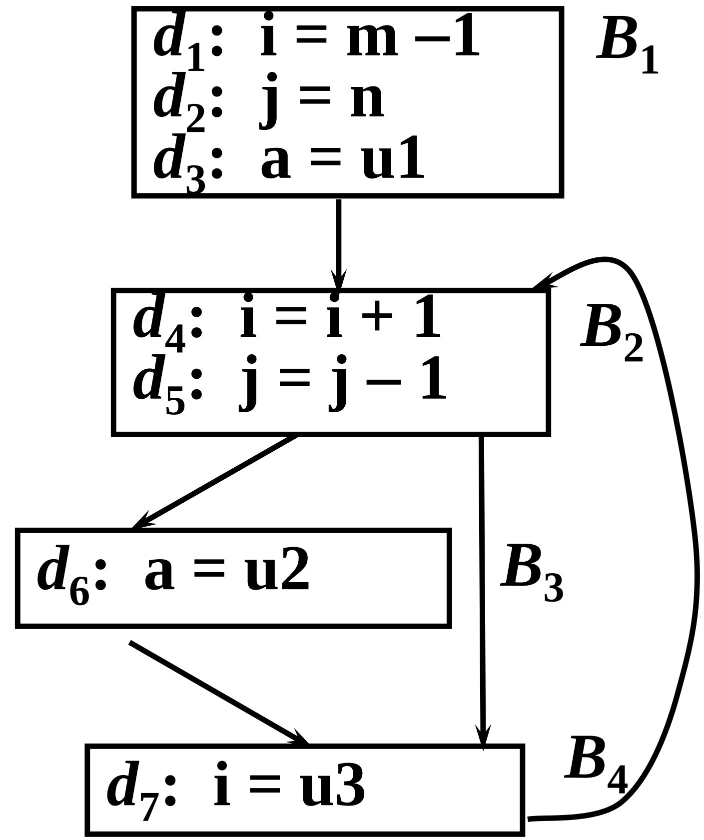
到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1110
B_3	001 1100	000 1110
B_4	001 1110	001 0111

不再继续演示迭代计算



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析非向量计算方法

- **迭代计算**
  - 计算次序，深度优先序，即 B1 -> B2 -> B3 -> B4
  - 初始值：for all B: IN[B] = ∅; OUT[B] = GEN[B]
  - 第一次迭代：

IN[B1] = ∅; // B1 无前驱结点

OUT[B1] = GEN[B1] ∪(IN[B1]-KILL[B1]) = GEN[B1] = { d1, d2, d3 }

IN[B2] = OUT[B1] ∪OUT[B4] = {d1, d2, d3 } ∪{ d7 } = { d1, d2, d3, d7 }

OUT[B2] = GEN[B2] ∪(IN[B2]-KILL[B2]) = { d4, d5 } ∪{ d3 } = { d3, d4, d5 }

IN[B3] = OUT[B2] = { d3, d4, d5 }

OUT[B3] = { d6 } ∪ ({ d3, d4, d5 } – { d3 }) = { d4, d5, d6 }

IN[B4] = OUT[B3] ∪ OUT[B2] = { d3, d4, d5, d6 }

OUT[B4] = { d7 } ∪ ({ d3, d4, d5, d6 } – { d1, d4 }) = { d3, d5, d6, d7 }

—第二次迭代

IN[B1] = ∅; // B1 无前驱结点

OUT[B1] = GEN[B1] ∪ (IN[B1]-KILL[B1]) = GEN[B1] = { d1, d2, d3 }

IN[B2] = OUT[B1] ∪ OUT[B4] = { d1,d2,d3 } ∪ { d3, d5, d6, d7 } = {d1, d2, d3, d5, d6, d7}

OUT[B2] = GEN[B2] ∪ (IN[B2]-KILL[B2]) = { d4, d5 } ∪ { d3, d5, d6 } = { d3, d4, d5, d6 }

IN[B3] = OUT[B2] = { d3, d4, d5, d6 }

OUT[B3] = { d6 } ∪ ({ d3, d4, d5, d6 } – { d3 }) = { d4, d5, d6 }

IN[B4] = OUT[B3] ∪ OUT[B2] = { d3, d4, d5, d6 }

OUT[B4] = { d7 } ∪ ({ d3, d4, d5, d6 } – { d1, d4 }) = { d3, d5, d6, d7 }

经过第二次迭代后，IN[B]和OUT[B] 不再变化。

到达-定值分析

- 到达-定值数据流等式是正向的方程

  $$\text{OUT}[B] = gen[B] \cup (\text{IN}[B] - kill[B])$$

  $$\text{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$$

  某些数据流等式是反向的

- 到达-定值数据流等式的合流运算是求并集

  $$\text{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$$

  某些数据流等式的合流运算是求交集

- 对到达-定值数据流方程，迭代求它的最小解

  某些数据流方程可能需要求最大解

❑定义

- ❖ x的值在*p*点开始的某条执行路径上被引用，则说x在*p*点活跃，否则称x在*p*点已经死亡
- ❖ IN[*B*]：块*B*开始点的活跃变量集合
- ❖ OUT[*B*]：块*B*结束点的活跃变量集合
- ❖ use_B：块*B*中有引用且在引用前无定值的变量集
- ❖ def_B：块*B*中有定值的变量集

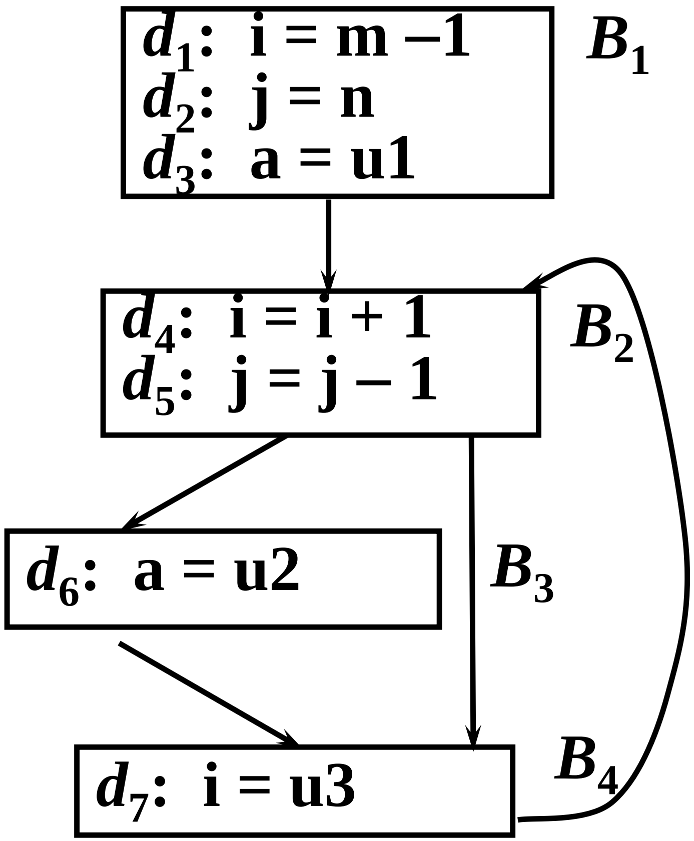
❑应用

- ❖ 一种重要应用就是基本块的寄存器分配

❑例

❖ $use[B_2] = \{ i, j \}$

❖ $def[B_2] = \{ i, j \}$



活跃变量分析

活跃变量数据流等式

- $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$
- $\text{OUT}[B] = \cup_{S\text{是}B\text{的后继}} \text{IN}[S]$
- $\text{IN}[\text{EXIT}] = \varnothing$

和到达-定值等式之间的联系与区别

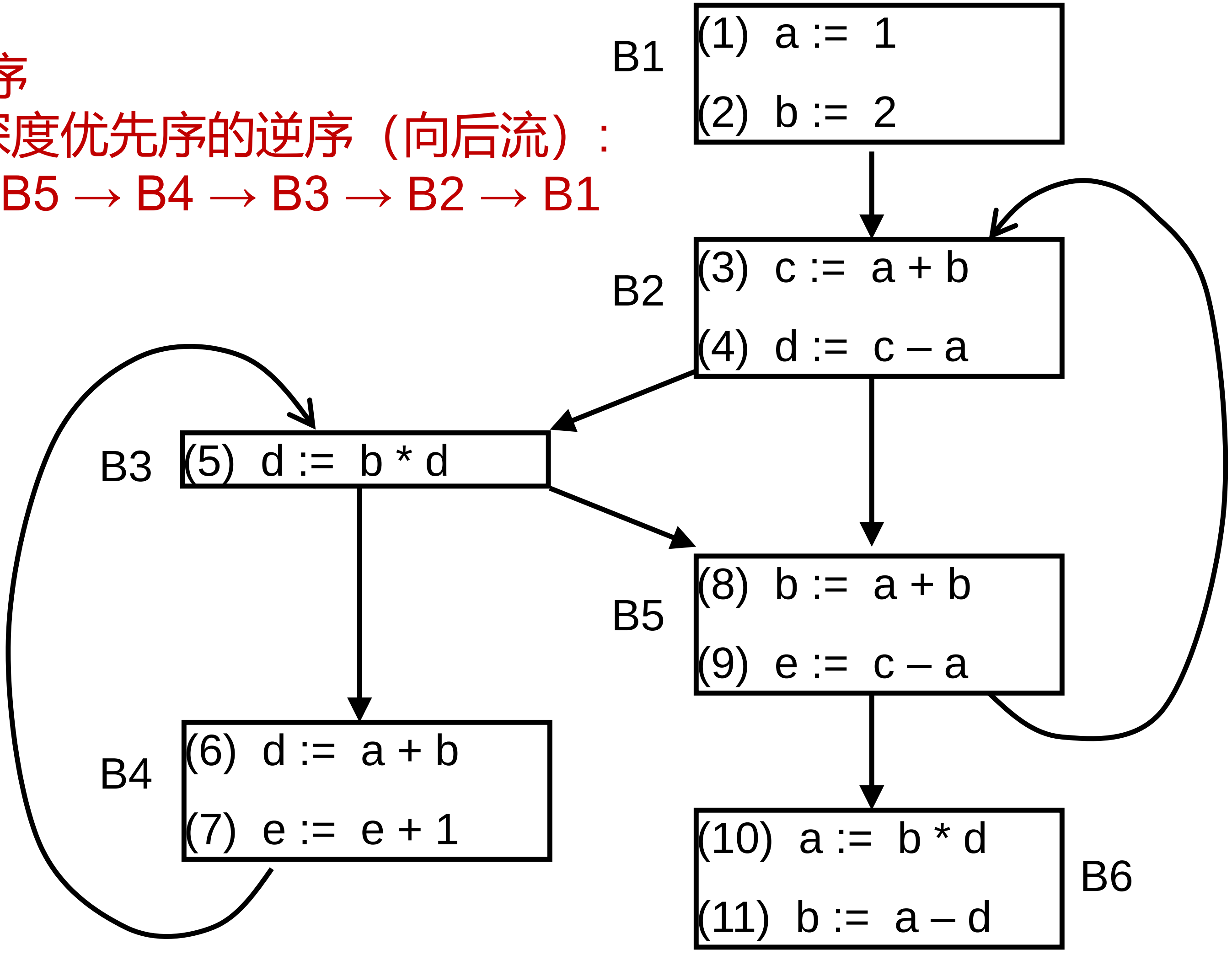
- 都以集合并算符作为它们的汇合算符
- 信息流动方向相反，IN和OUT的作用相互交换
- *use*和*def*分别取代*gen*和*kill*
- 仍然需要最小解

活跃变量分析

计算次序

* 结点深度优先序的逆序（向后流）：

* B6 → B5 → B4 → B3 → B2 → B1



基本块出口活跃变量

- 各基本块USE和DEF如下，

  USE[B1] = { } ; DEF[B1] = { a, b }

  USE[B2] = { a, b } ; DEF[B2] = { c, d }

  USE[B3] = { b, d } ; DEF[B3] = { }

  USE[B4] = { a, b, e } ; DEF[B4] = { d }

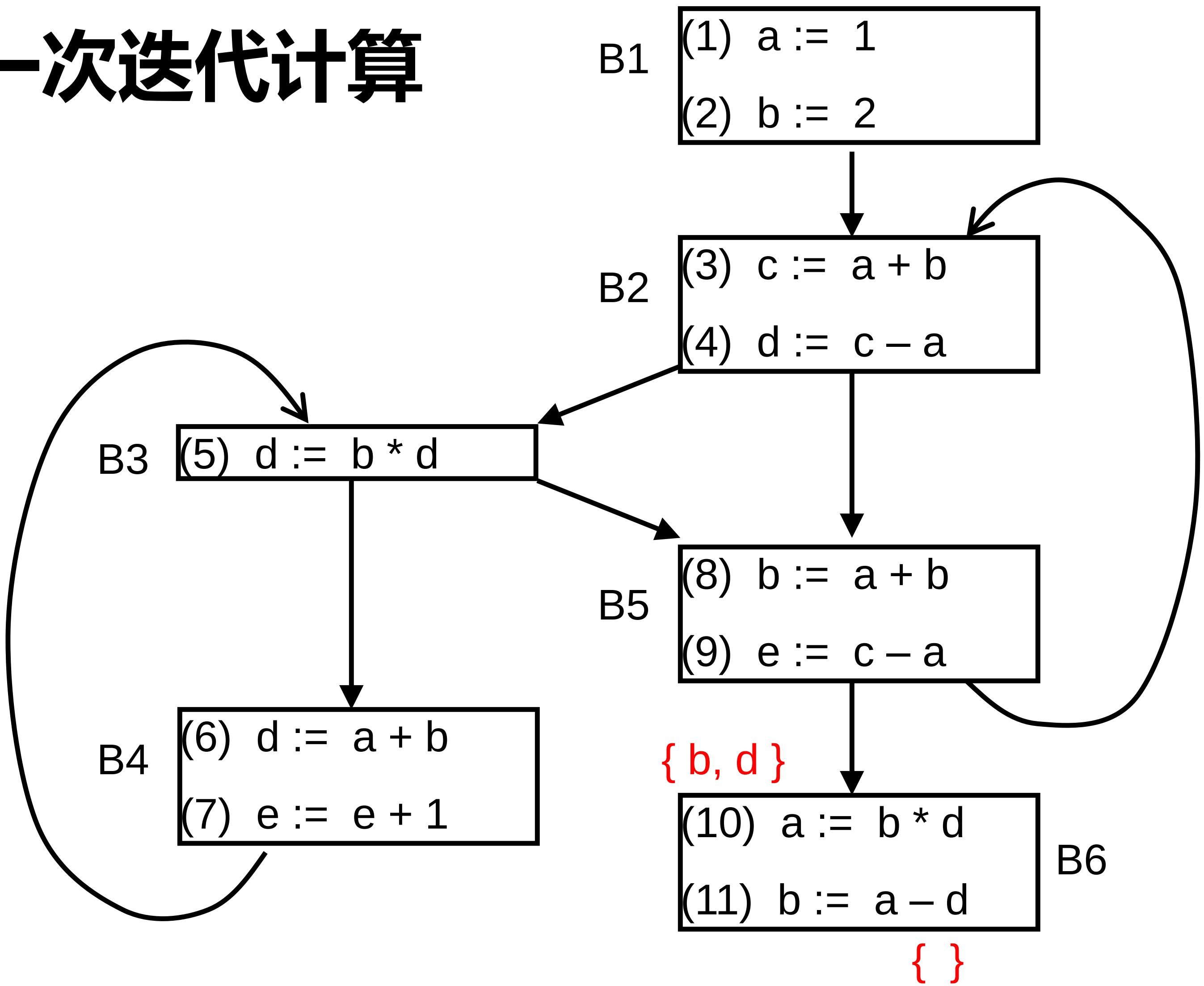
  USE[B5] = { a, b, c } ; DEF[B5] = { e }

  USE[B6] = { b, d } ; DEF[B6] = { a }

- 初始值，all B, IN[B] = { },

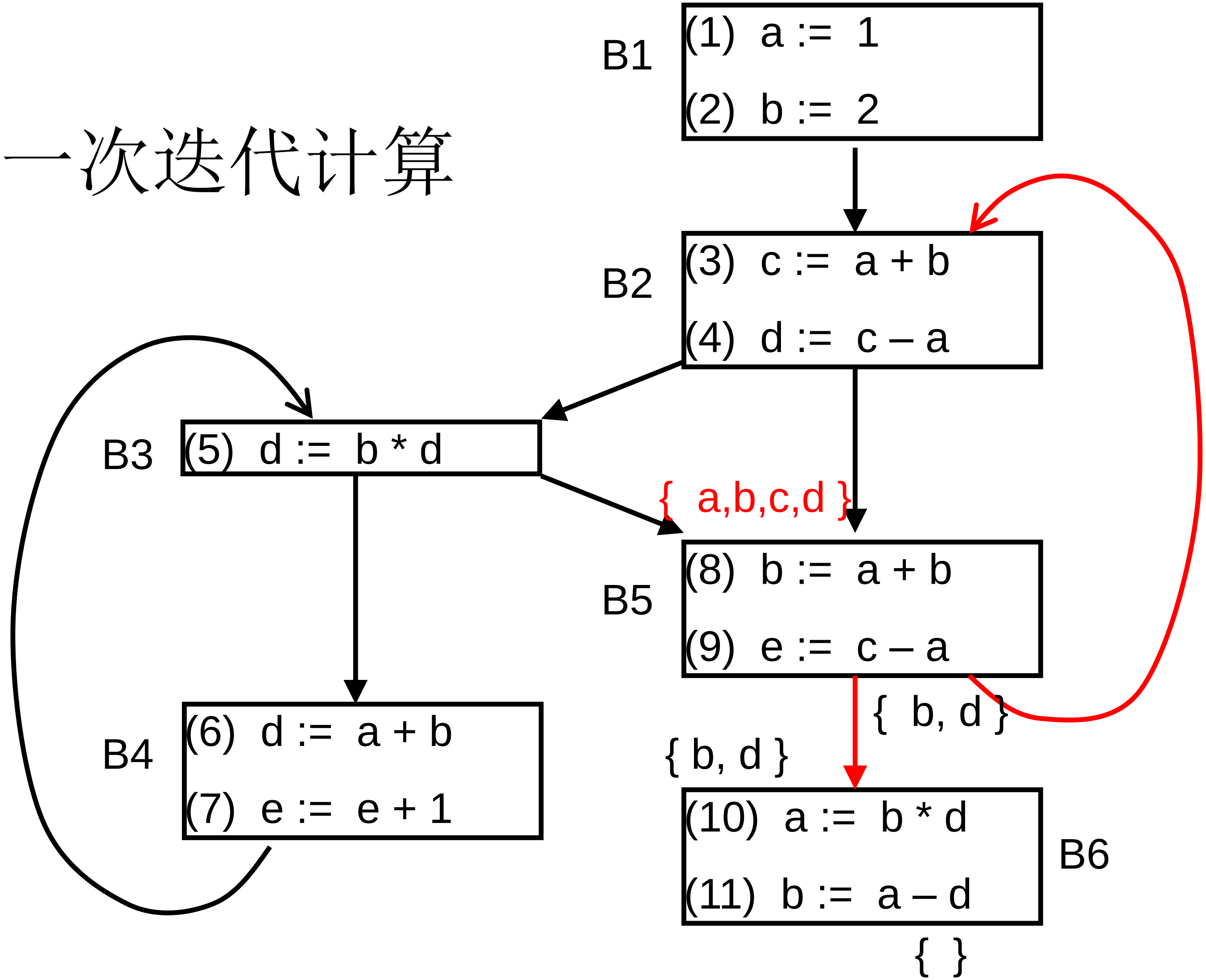
  OUT[B6]={ }//出口块

□第一次迭代计算


B1
(1) a := 1
(2) b := 2
B2
(3) c := a + b
(4) d := c – a
B3
(5) d := b * d
B4
(6) d := a + b
(7) e := e + 1
B5
(8) b := a + b
(9) e := c – a
{ b, d }
(10) a := b * d
(11) b := a – d
B6
{ }

基本块出口活跃变量

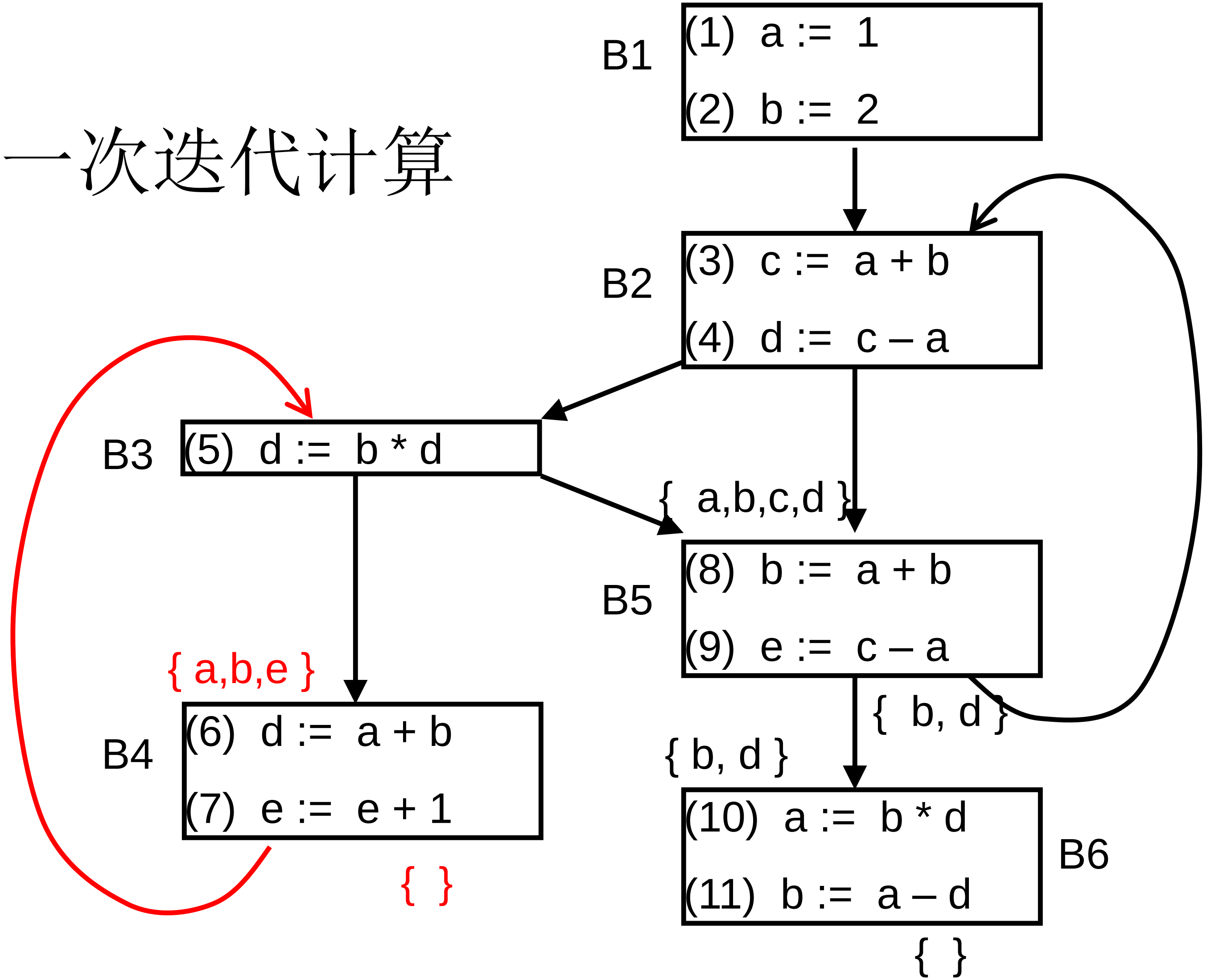
- 第一次迭代计算



基本块出口活跃变量

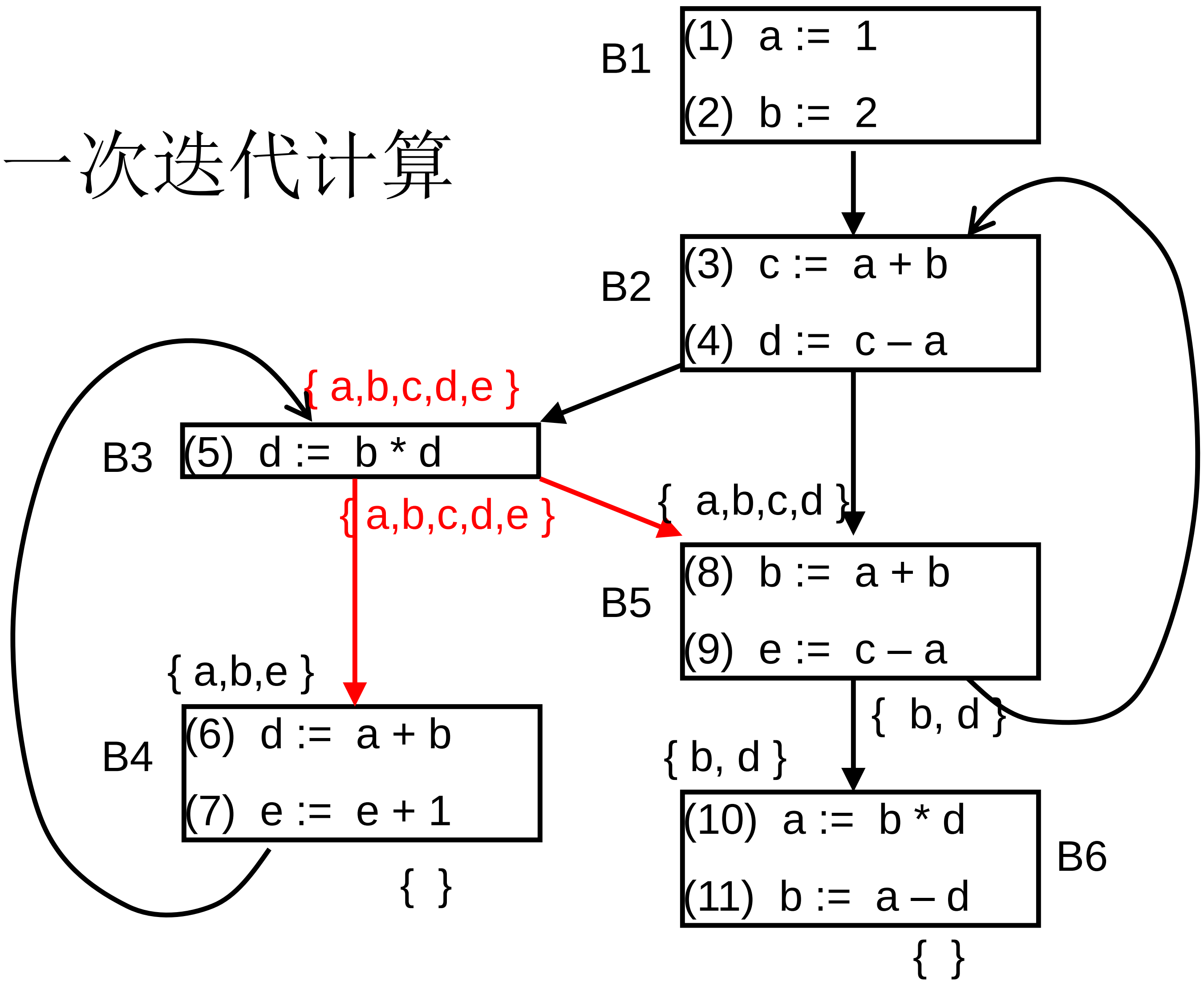
中国科学技术大学 University of Science and Technology of China

- 第一次迭代计算



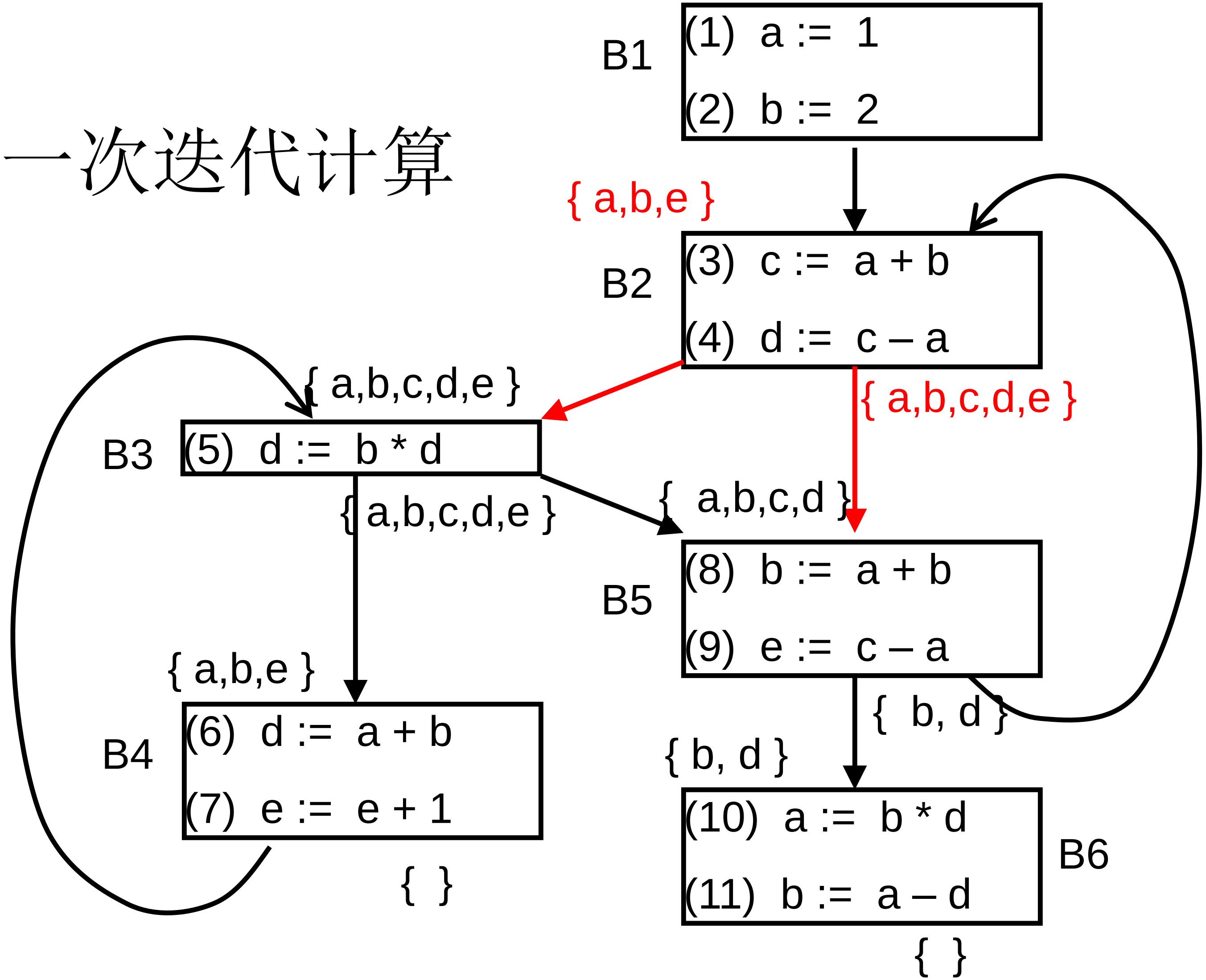
基本块出口活跃变量

- 第一次迭代计算



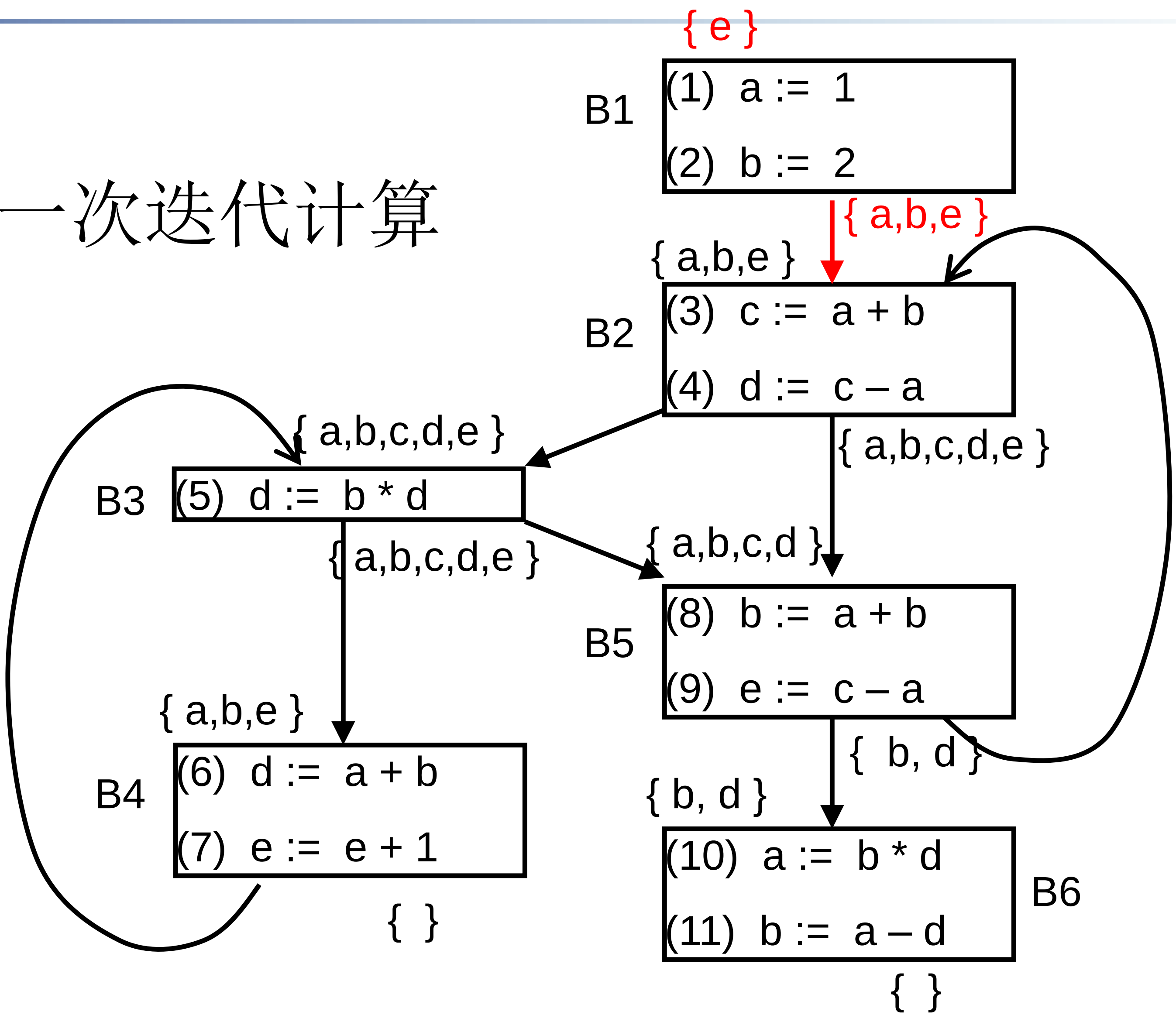
基本块出口活跃变量

- 第一次迭代计算



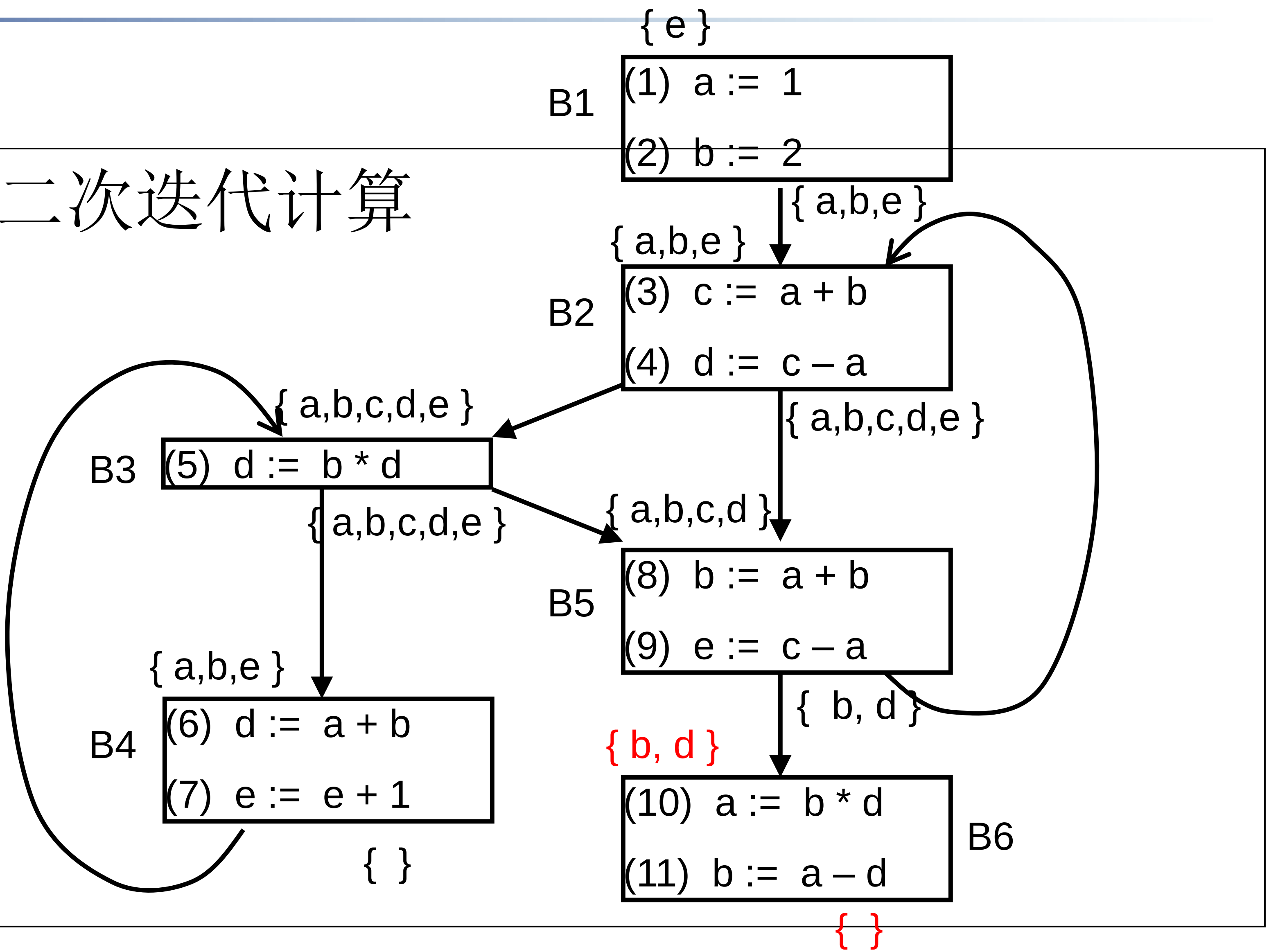
基本块出口活跃变量

- 第一次迭代计算



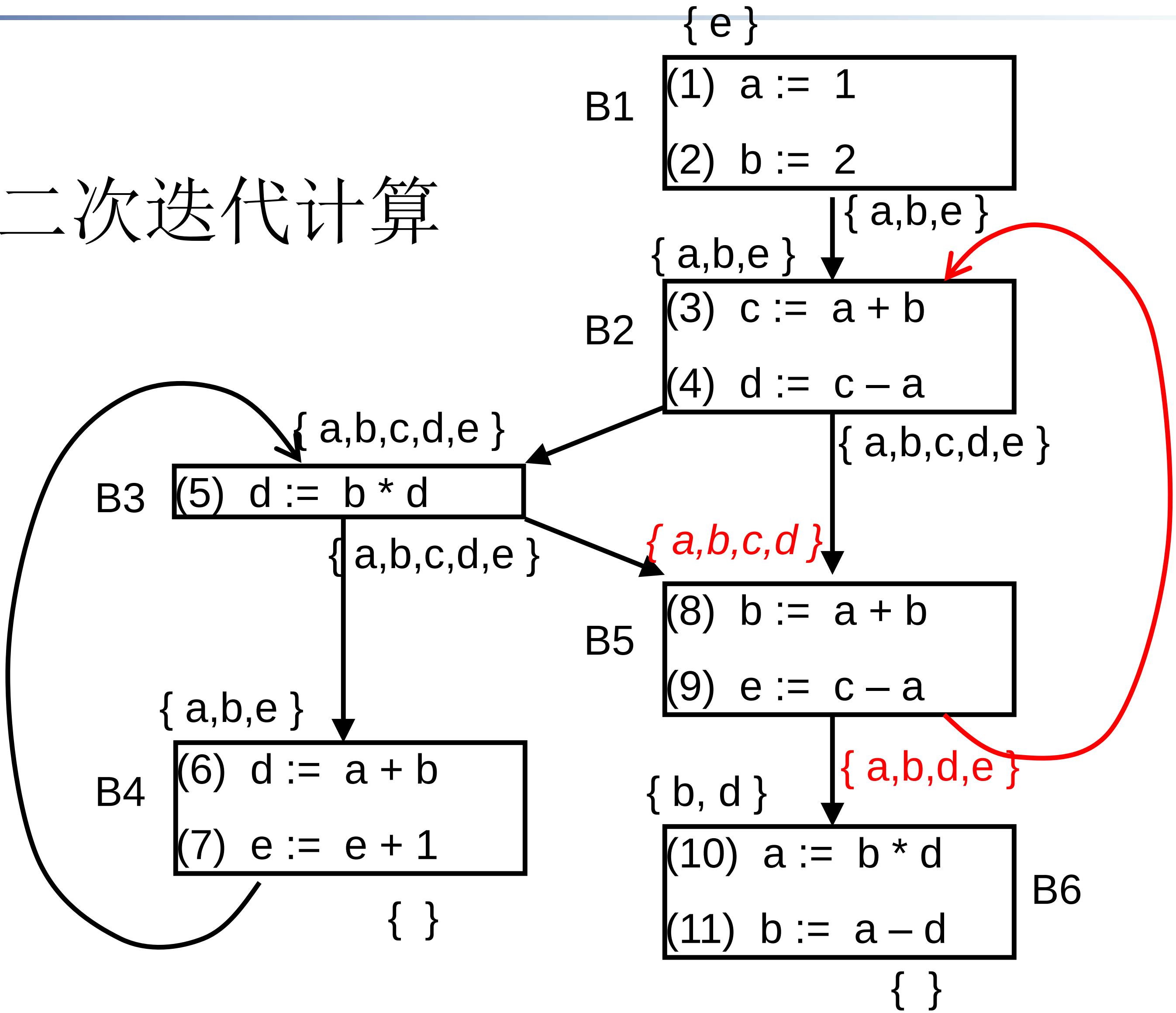
基本块出口活跃变量

- 第二次迭代计算



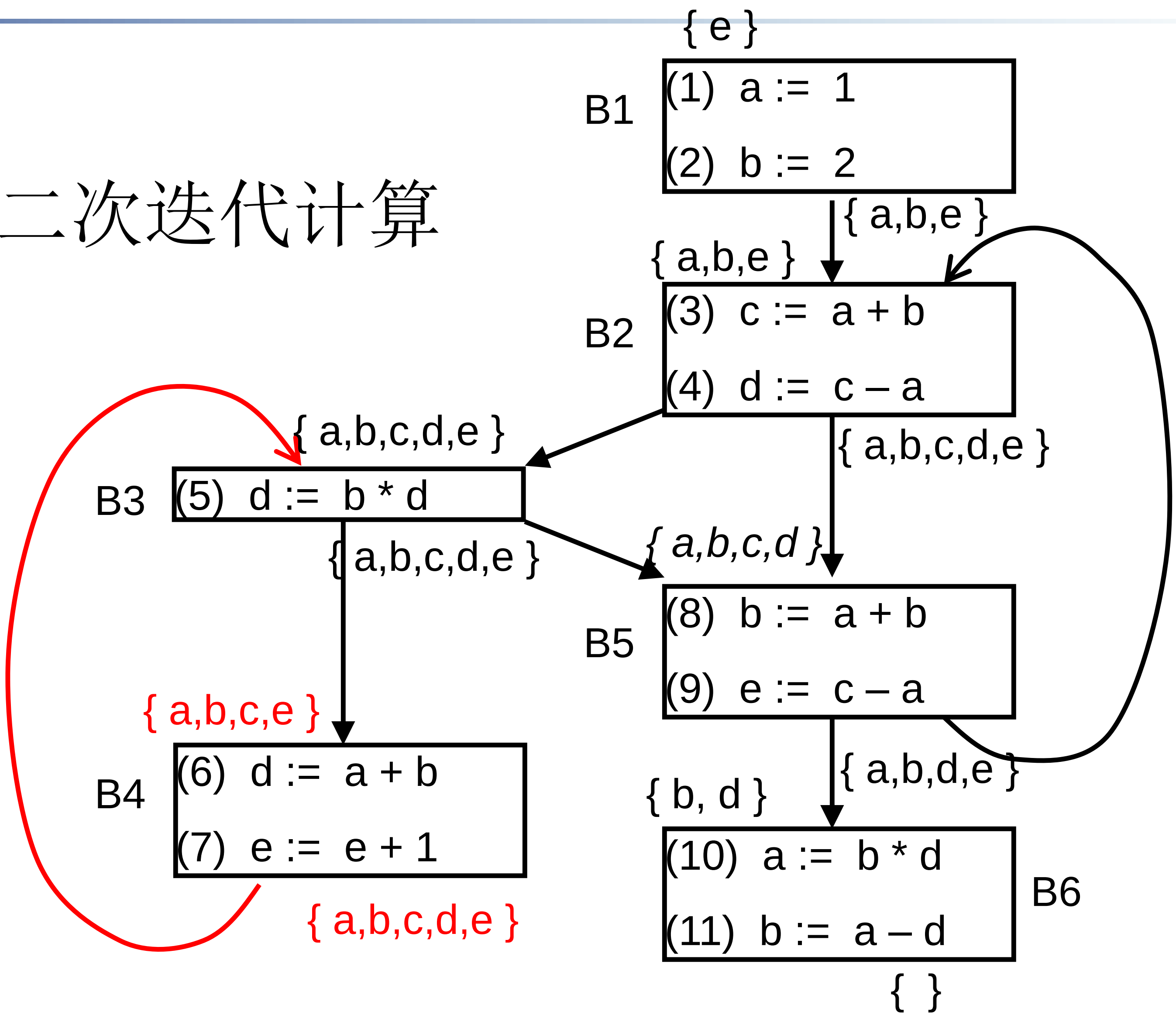
基本块出口活跃变量

- 第二次迭代计算



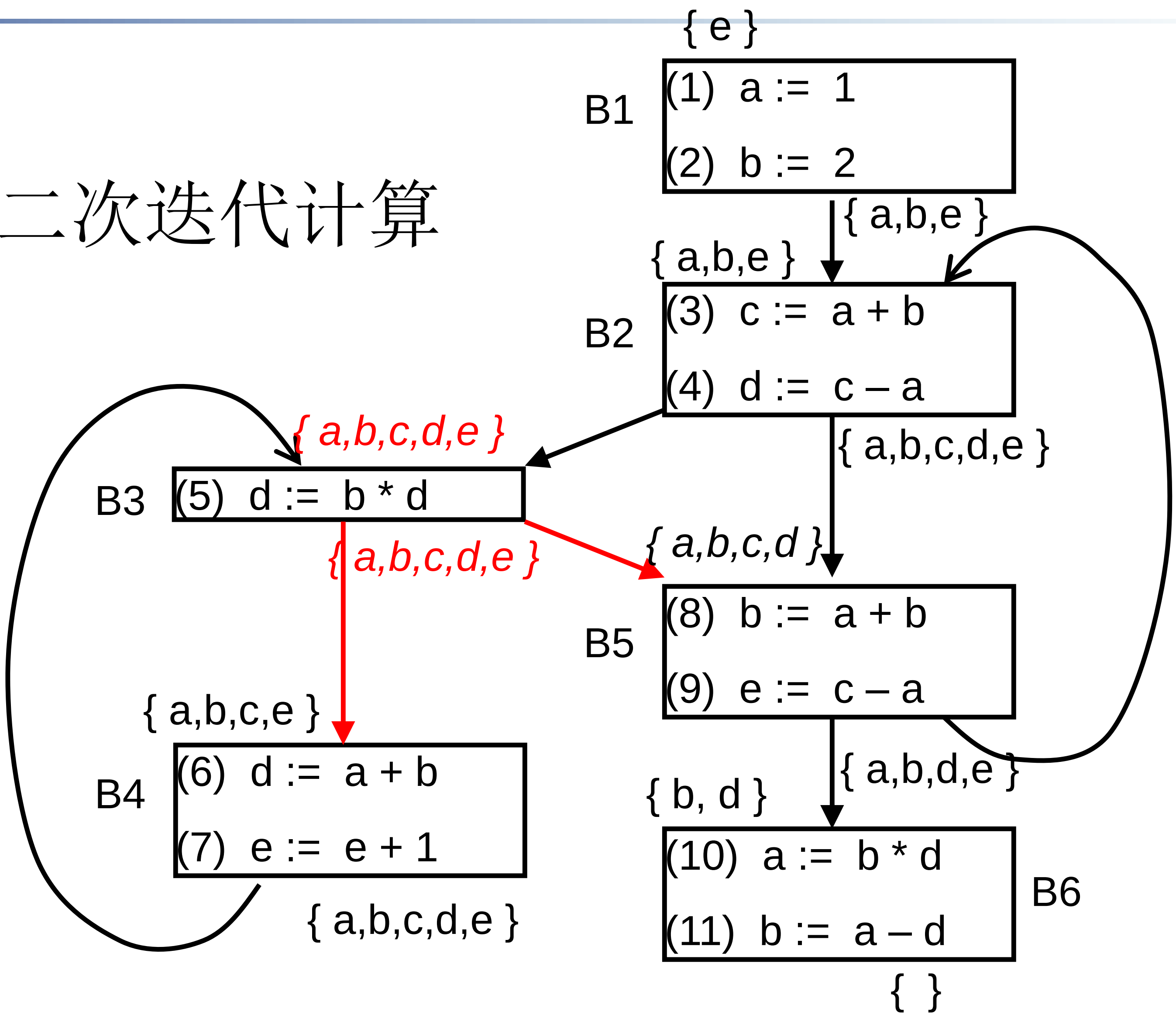
基本块出口活跃变量

- 第二次迭代计算



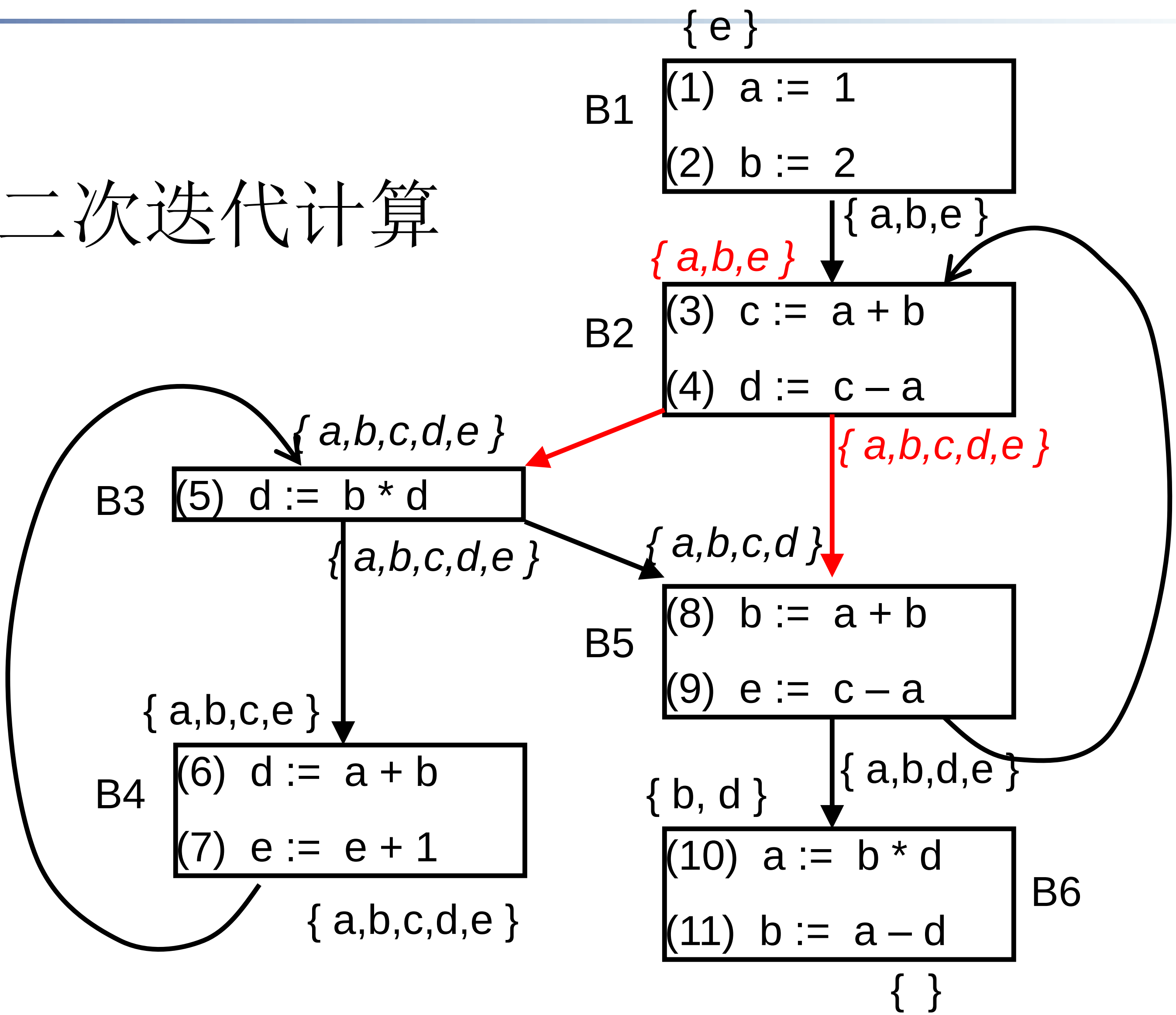
基本块出口活跃变量

- 第二次迭代计算



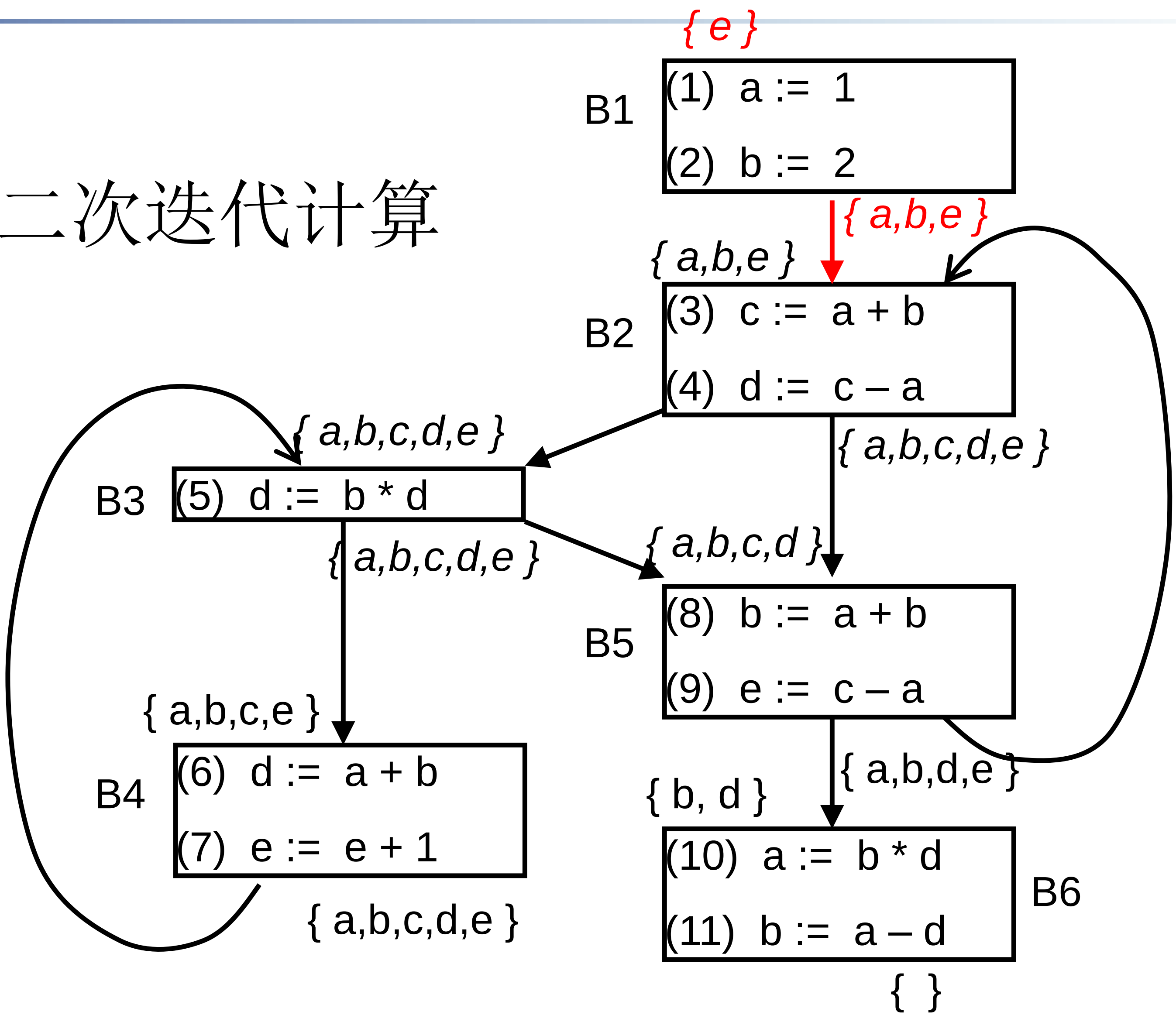
基本块出口活跃变量

- 第二次迭代计算



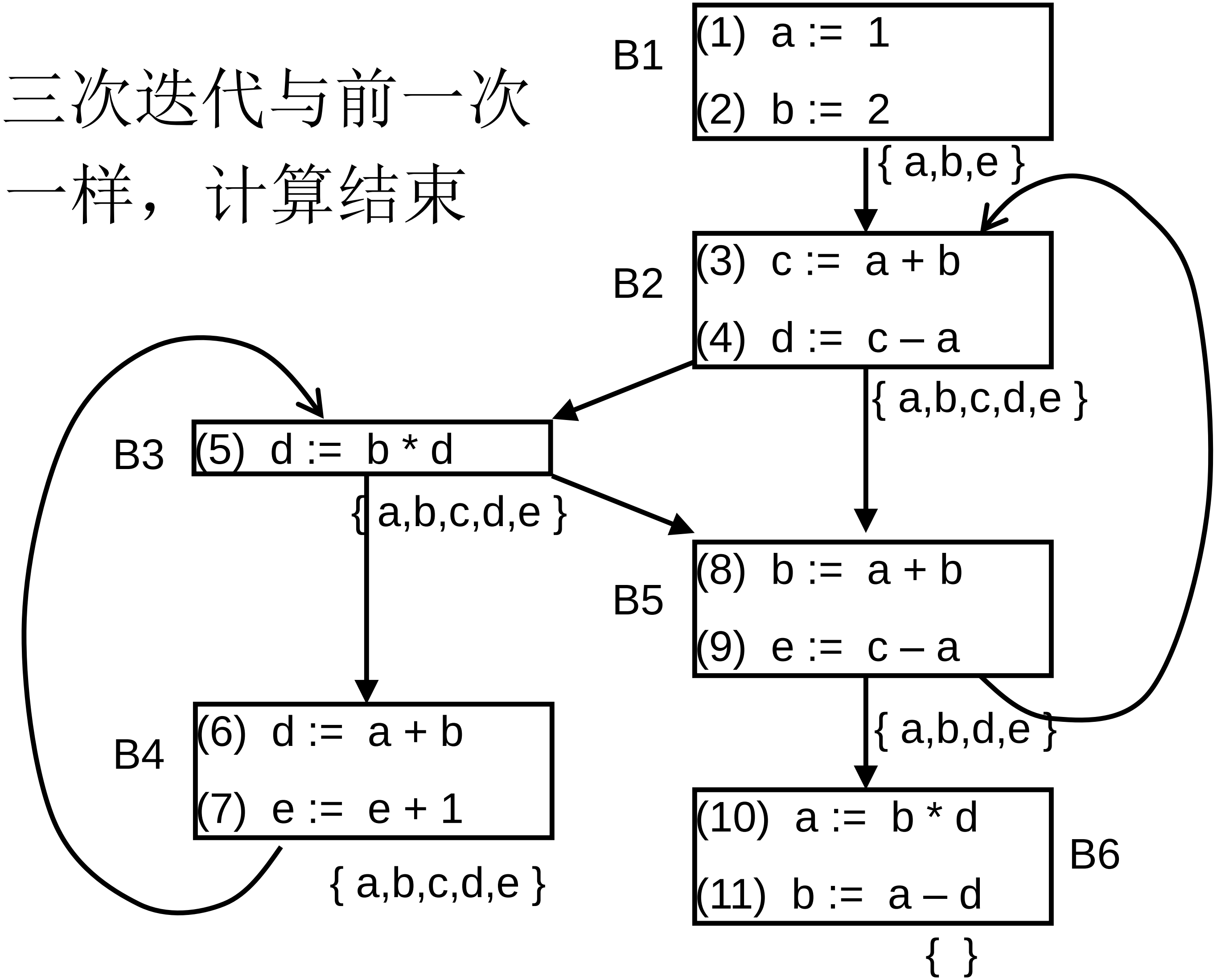
基本块出口活跃变量

- 第二次迭代计算



基本块出口活跃变量

- 第三次迭代与前一次结果一样，计算结束



中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
独立于机器的优化

At the end, if you fail, at least you did something interesting, rather than doing something boring and also failing. Or doing something boring and then forgetting how to do something interesting.

—— Barbara Liskov (Turing Award 2008)