

# 第九章 机器无关的优化

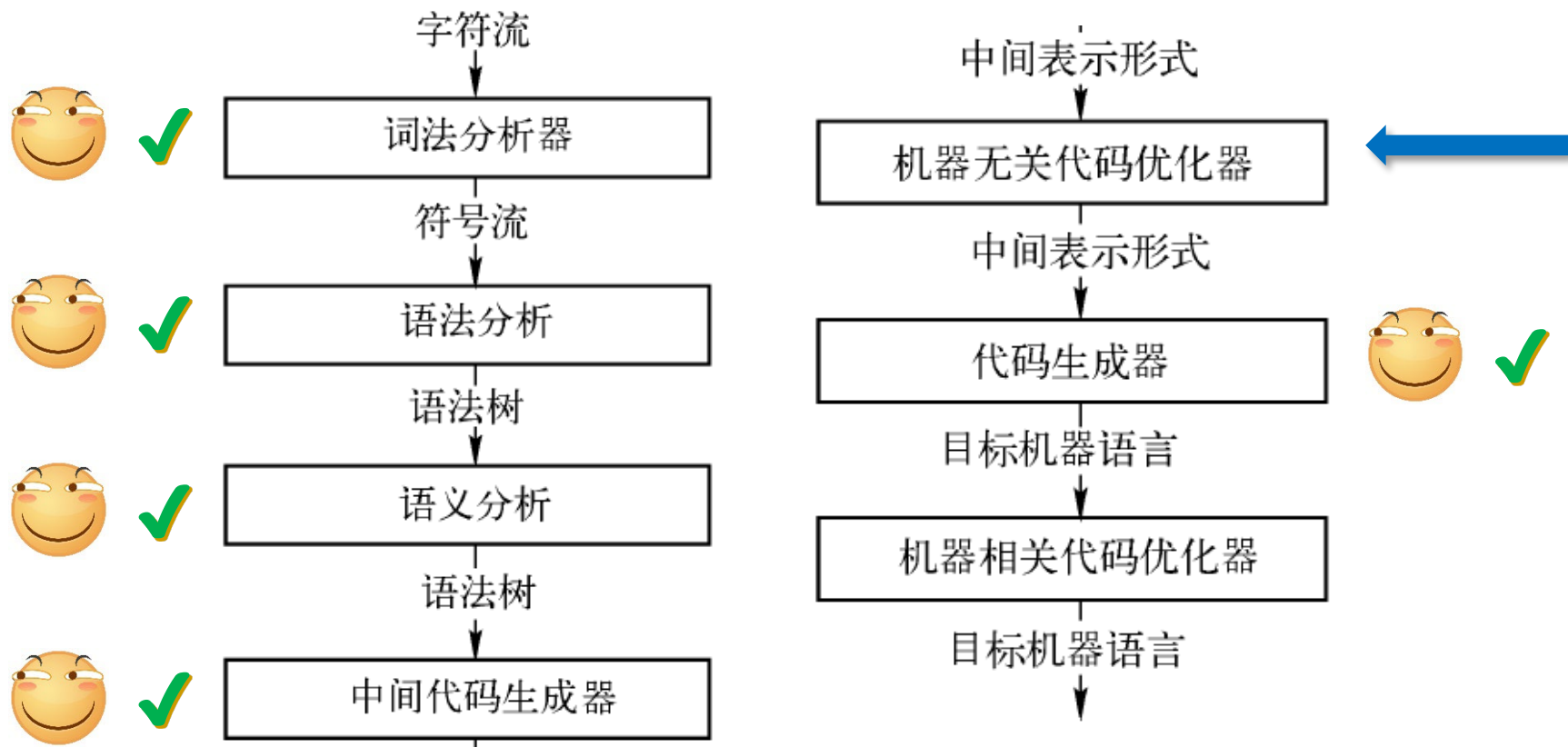
《编译原理》

谭添

南京大学计算机系

2026年春季

# 闯关进度



# 主要内容

---

- 引言
- 优化的来源
- 数据流分析
- 部分冗余消除
- 循环的识别、分析和优化

# 引言

- 代码优化或者代码改进
  - 在目标代码中消除不必要的指令
  - 把一个指令序列替换为一个完成相同功能的更快的指令序列
- 全局优化
  - 具体的优化实现基于数据流分析技术
  - 用以收集程序相关信息的算法

# 优化的主要来源

- 编译器只能通过一些相对低层的语义等价转换来优化代码
  - 冗余运算的原因
    - 源程序中的冗余
    - 高级程序设计语言编程的副产品，如 `A[i][j].f = 0; A[i][j].k = 1;`
  - 语义不变的优化
    - 公共子表达式消除
    - 复制传播
    - 死代码消除
    - 常量折叠

# 优化的例子 (1)

- 快速排序算法

```
void quicksort(int m, int n)
    /* 递归地对 a[m]和a[n]之间的元素排序 */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* 片断由此开始 */
    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; /* 对换a[i]和a[j]*/
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* 对换a[i]和a[n] */
    /* 片断在此结束 */
    quicksort(m, j); quicksort(i+1, n);
}
```

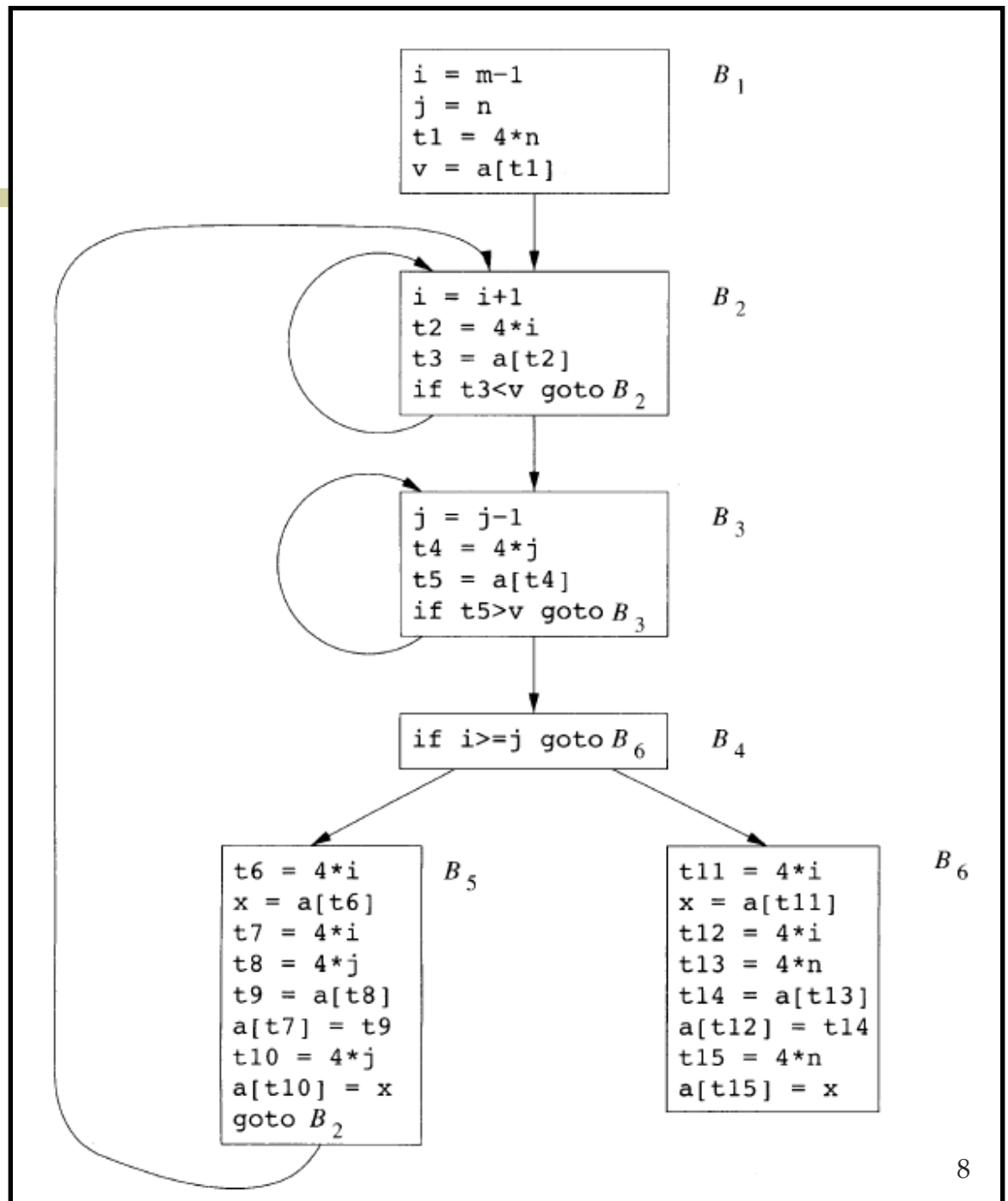
# 优化的例子 (2)

- 三地址代码

(1)	$i = m - 1$	(16)	$t7 = 4 * i$
(2)	$j = n$	(17)	$t8 = 4 * j$
(3)	$t1 = 4 * n$	(18)	$t9 = a[t8]$
(4)	$v = a[t1]$	(19)	$a[t7] = t9$
(5)	$i = i + 1$	(20)	$t10 = 4 * j$
(6)	$t2 = 4 * i$	(21)	$a[t10] = x$
(7)	$t3 = a[t2]$	(22)	goto (5)
(8)	if $t3 < v$ goto (5)	(23)	$t11 = 4 * i$
(9)	$j = j - 1$	(24)	$x = a[t11]$
(10)	$t4 = 4 * j$	(25)	$t12 = 4 * i$
(11)	$t5 = a[t4]$	(26)	$t13 = 4 * n$
(12)	if $t5 > v$ goto (9)	(27)	$t14 = a[t13]$
(13)	if $i \geq j$ goto (23)	(28)	$a[t12] = t14$
(14)	$t6 = 4 * i$	(29)	$t15 = 4 * n$
(15)	$x = a[t6]$	(30)	$a[t15] = x$

# 流图

- 循环
  - $B_2$
  - $B_3$
  - $B_2, B_3, B_4, B_5$



# 控制流图 (Control-Flow Graph)

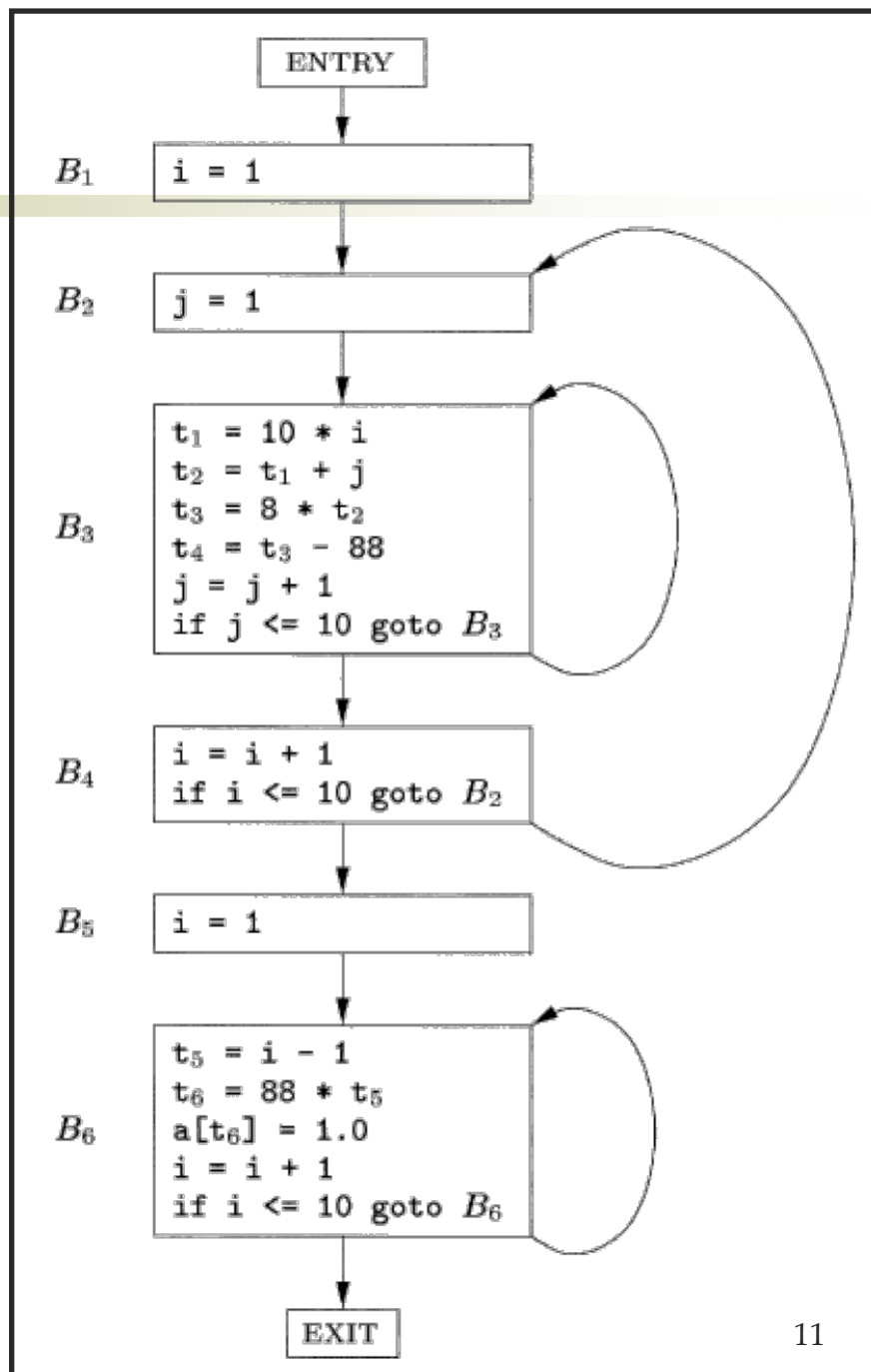
- 表示程序控制流的有向图 (简称流图)
- 流图的结点是基本块
- 流图的边指明了哪些基本块可以跟在一个基本块之后运行
  - 若基本块 $B_i$ 最后一条指令执行后紧跟着执行的指令可能为基本块 $B_j$ , 则图上有一条边 $B_i \rightarrow B_j$
- 图上的路径表示程序执行的路径, 可作为优化与分析的基础
  - 它给出了程序的**控制流**
  - 可以根据流图了解到一个值是否会被使用等信息

# 流图的构造

- 流图的**结点**是基本块
  - 两个结点 $B$ 和 $C$ 之间有一条有向边 iff 基本块 $C$ 的第一个指令**可能**在 $B$ 的最后一个指令之后执行
- 存在边  $B \rightarrow C$  的情况
  - $B$ 的结尾指令是一条跳转到 $C$ 的开头的条件/无条件语句
  - $C$ 紧跟在 $B$ 之后，且 $B$ 的结尾不是无条件跳转语句
  - 我们称 $B$ 是 $C$ 的**前驱**， $C$ 是 $B$ 的**后继**
- **入口/出口结点**
  - 不和任何中间指令对应；入口到第一条指令有一条边；任何可能最后执行的基本块到出口有一条边

# 流图的例子

- 因跳转而生成的边
  - $B_3 \rightarrow B_3$
  - $B_4 \rightarrow B_2$
  - $B_6 \rightarrow B_6$
- 因为顺序而生成的边
  - 其它

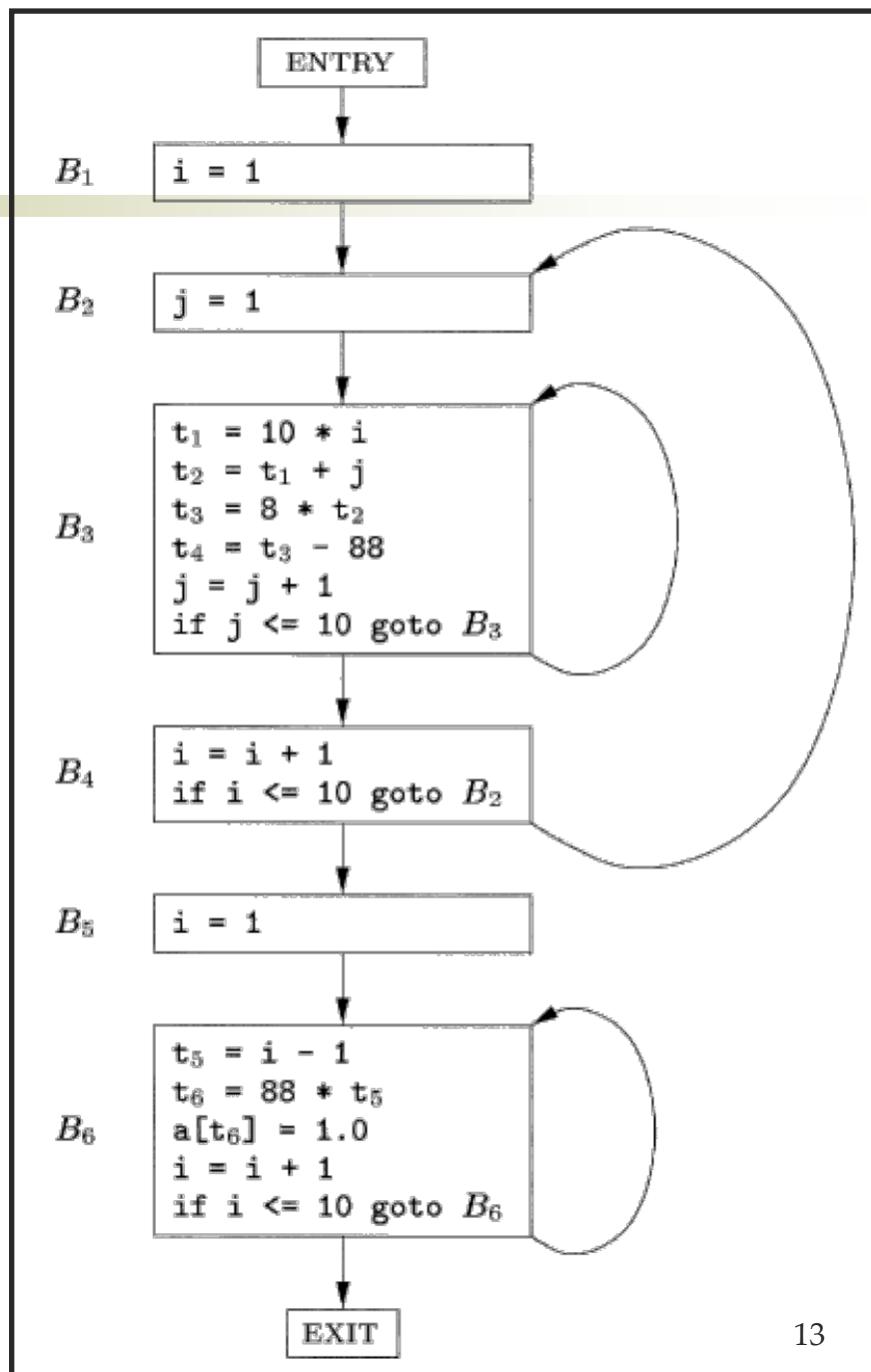


# 循环

- 程序的大部分运行时间花费在循环上，因此循环是识别的重点 (优化的目标)
- 循环的定义
  - 循环 $L$ 是一个结点集合
  - 存在一个循环入口 (Loop entry) 结点，是唯一的、前驱可以在循环 $L$ 之外的结点，到达其余结点的路径必然先经过这个入口结点
  - 其余结点都存在到达入口结点的非空路径，且路径都在 $L$ 中

# 循环的例子

- 循环
  - $\{ B_3 \}$
  - $\{ B_2, B_3, B_4 \}$
  - $\{ B_6 \}$
- 对于  $\{ B_2, B_3, B_4 \}$  的解释
  - $B_2$  为入口结点
  - $B_1, B_5, B_6$  不在循环内
    - 到达  $B_1$  可不经过  $B_2$
    - $B_5, B_6$  没有到达  $B_2$  的结点



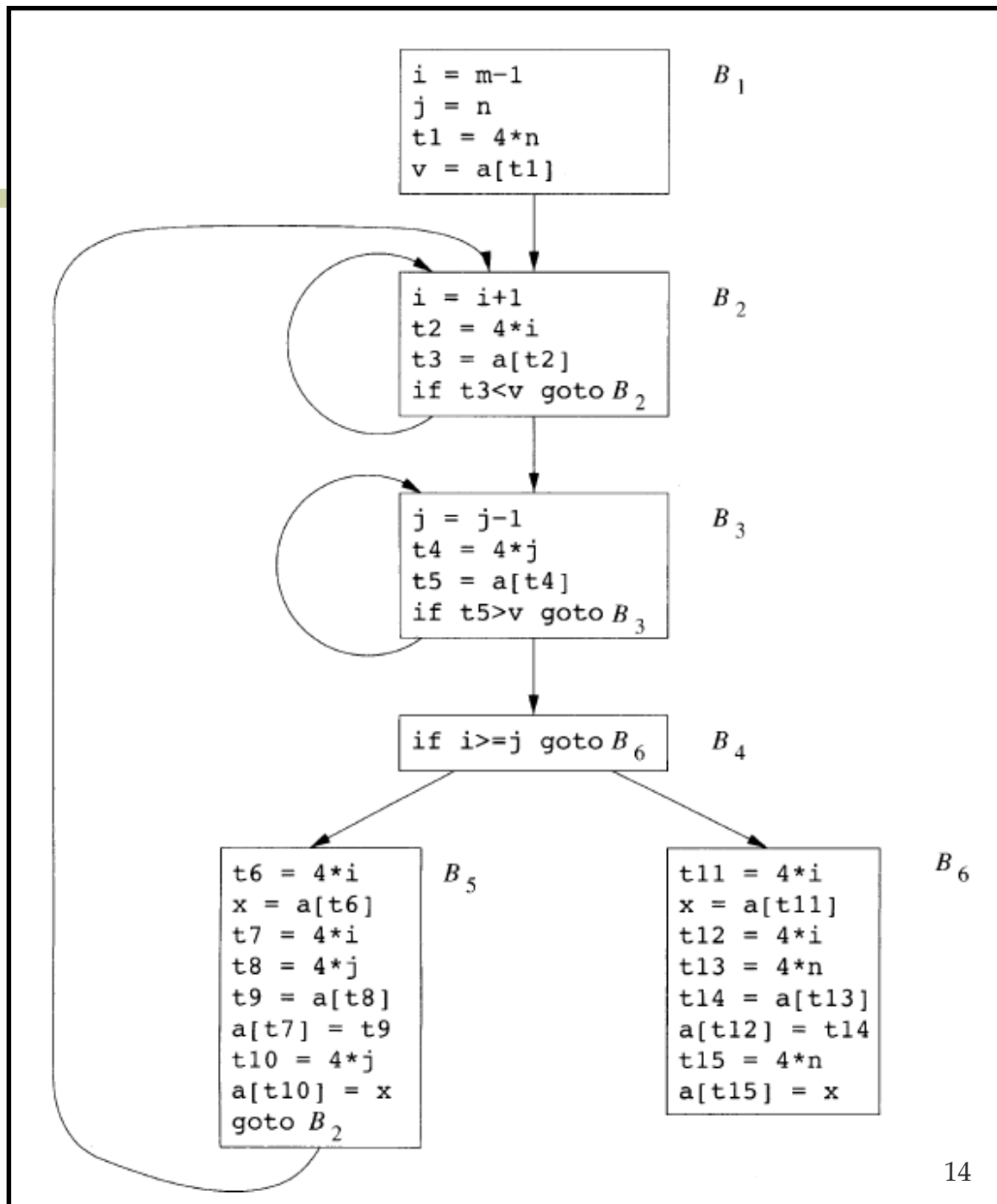
# 流图

- 循环

- $B_2$

- $B_3$

- $B_2, B_3, B_4, B_5$



# 全局公共子表达式

- 如果 $E$ 
  - 在某次出现之前**必然**已被计算过，且
  - $E$ 的运算**分量**在该次计算之后没有被改变
  - 那么， $E$ 的本次出现就是一个**公共子表达式**
- 如果上次 $E$ 值赋给了 $x$ ，且 $x$ 值没有被修改过，那么我们可以使用 $x$ ，而无需计算 $E$

<pre>t6 = <u>4*i</u> x = a[t6] t7 = <u>4*j</u> t8 = <u>4*j</u> t9 = a[t8] a[t7] = t9 t10 = <u>4*j</u> a[t10] = x goto B<sub>2</sub></pre>	$B_5$
a) 消除之前	
<pre>t6 = <u>4*i</u> x = a[t6] t8 = <u>4*j</u> t9 = a[t8] <u>a[t6]</u> = t9 <u>a[t8]</u> = x goto B<sub>2</sub></pre>	$B_5$
b) 消除之后	

$t7 \Rightarrow t6$   
 $t10 \Rightarrow t8$

例子：

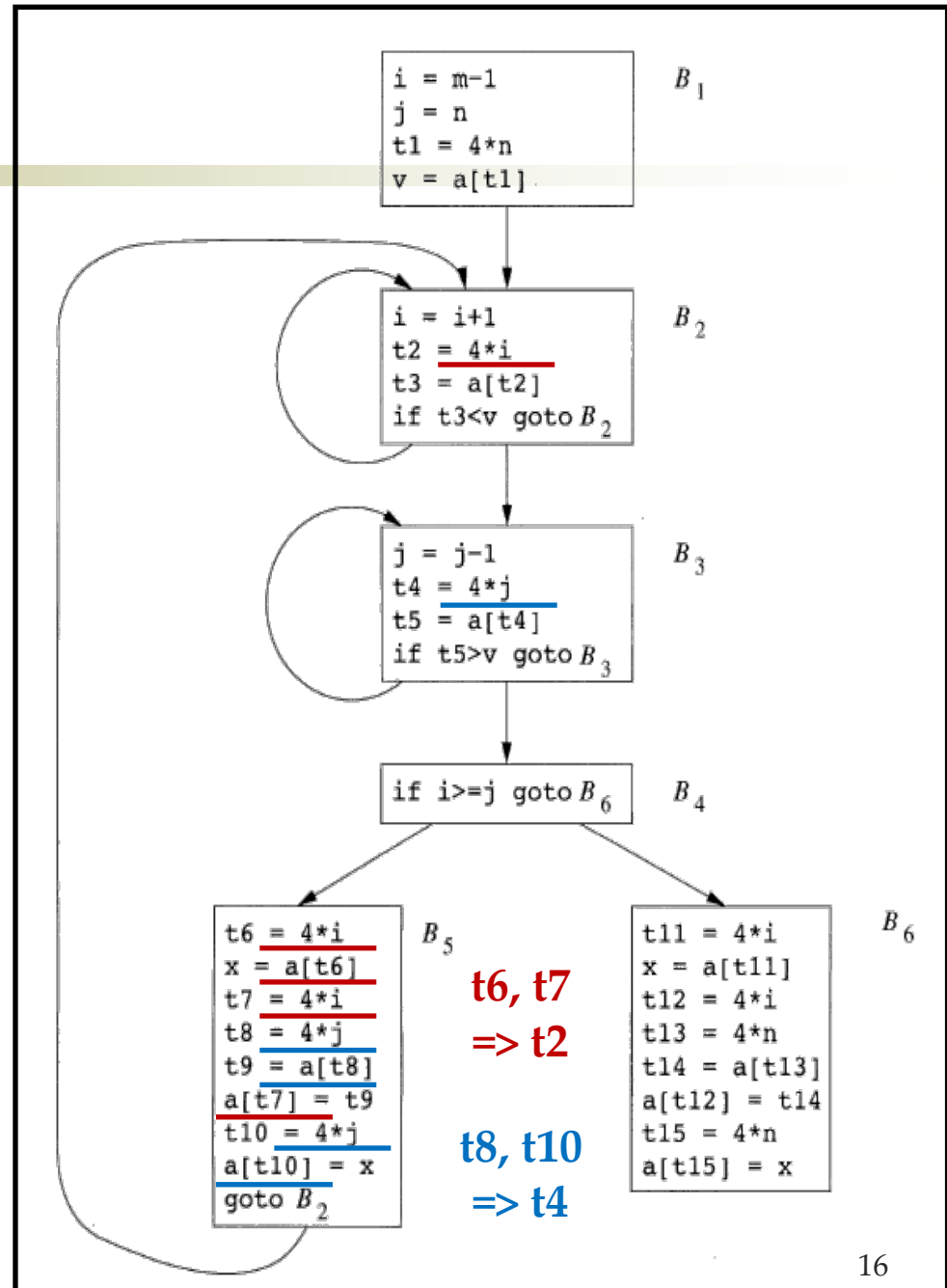
$$t7 = 4 * i$$

$$t10 = 4 * j$$

不需要重新计算

# 例子

- $B_2, B_3$ 中计算了 $4 * i, 4 * j$ , 且到达 $B_5$ 之前必然经过 $B_2, B_3$
- $t_2, t_4$ 在赋值后没有被改变过, 因此 $B_5$ 中可直接使用它们
- $B_5$ 中赋给 $x$ 的值 ( $a[t_6]$ ) 和 $B_2$ 中赋给 $t_3$ 的值 ( $a[t_2]$ ) 相同
- $t_4$ 替换 $t_8$ 后,  $B_5$ 中 $a[t_8]$  和 $B_3$ 中 $a[t_4]$ 又相同
- $B_6$ 中的 $a[t_{13}]$ 和 $B_1$ 中的 $a[t_1]$ 不同, 因为 $B_5$ 可能改变 $a$ 的值



# 消除以后

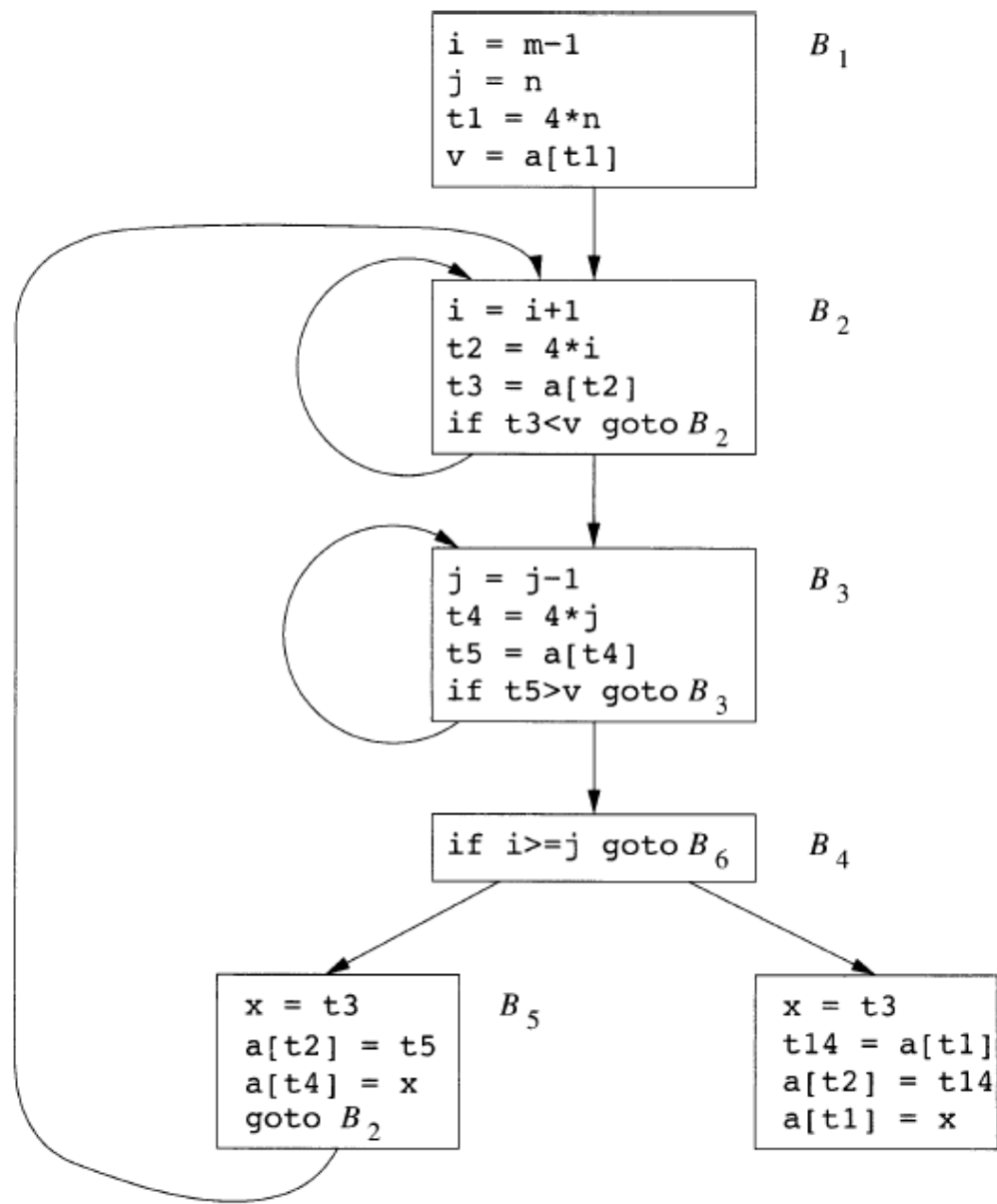
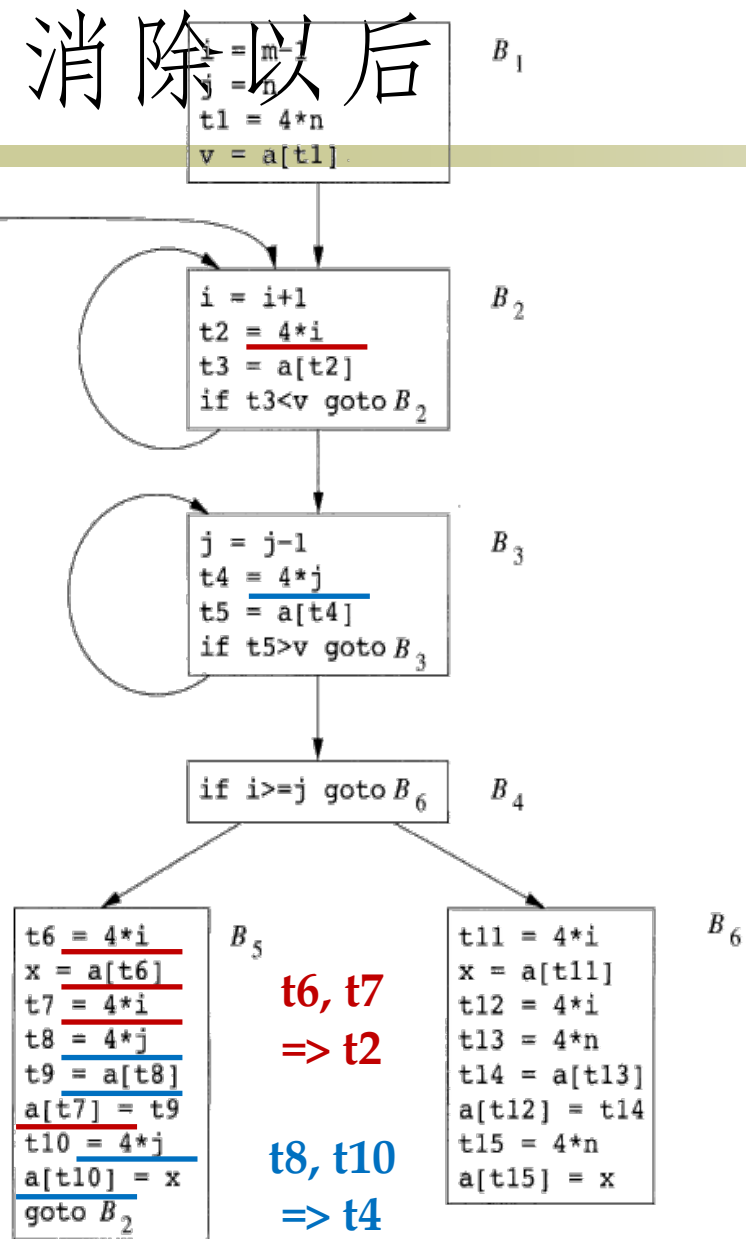
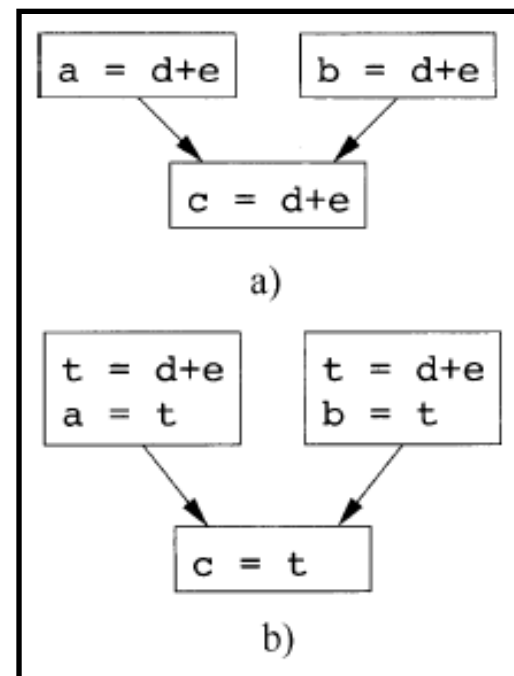


图9-5 经过公共子表达式消除之后的  $B_5$  和  $B_6$

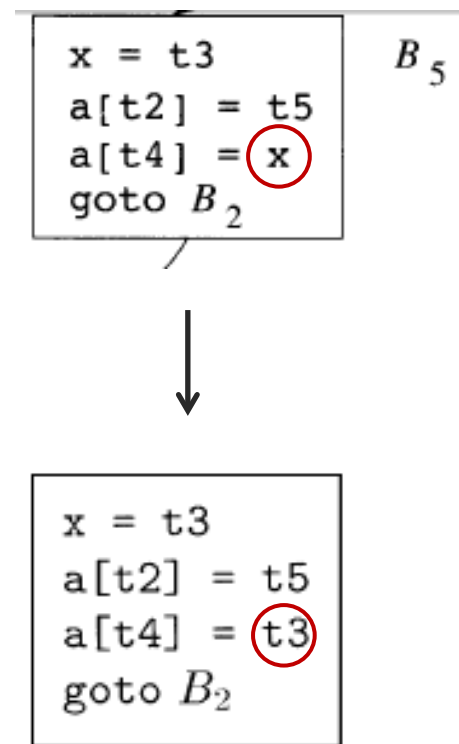
# 复制传播

- 形如 $u = v$ 的复制语句使得语句后面的程序点上， $u$ 的值等于 $v$ 的值
  - 如果在某个位置上 $u$ 一定等于 $v$ ，那么可以把 $u$ 替换为 $v$
  - 有时可以彻底消除对 $u$ 的使用，从而消除对 $u$ 的赋值语句
- 右图所示，消除公共子表达式时引入了复制语句
- 如果尽可能用 $t$ 来替换 $c$ ，可能就不需要 $c = t$ 这个语句了



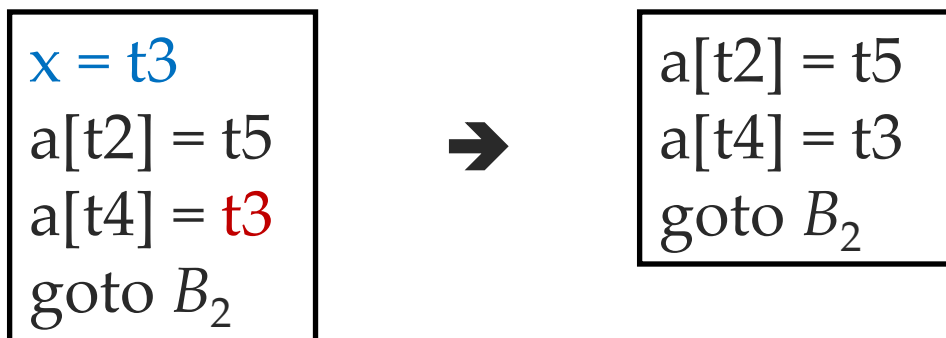
# 复制传播的例子

- 右图显示了对 $B_5$ 进行复制传播处理的情况
  - 可能消除所有对 $x$ 的使用



# 死代码消除

- 如果一个变量在某个程序点上的值可能会在之后被使用，那么这个变量在这个点上**活跃的**；否则这个变量就是**死的**，此时对该变量的赋值就是没有用的**死代码**
- 死代码多半是因为前面的优化而形成的
- 比如， $B_5$ 中的 $x = t3$ 就是死代码
- 消除后得到



# 代码移动

- 循环中的代码会被执行很多次
  - 循环不变表达式：循环的同一次运行的不同迭代中，表达式的值不变
- 把循环不变表达式移动到循环入口之前计算可以提高效率
  - 循环入口：进入循环的跳转都以这个入口为目标
- `while (i <= limit - 2) ...`
  - 如果循环体不改变`limit`的值，可在循环外计算`limit - 2`  
`t = limit - 2`  
`while (i <= t) ...`

# 归纳变量和强度消减

- 归纳变量

- 每次对 $x$ 赋值都使 $x$ 增加 $c$
- 把赋值改为**增量操作**，可消减计算强度
- 两个归纳变量步调一致，可删除一个
- 例子

- 循环开始保持 $t4 = 4 * j$
- $j = j - 1$ 后面的 $t4 = 4 * j$ 每次赋值使 $t4$ 减4
- 可替换为 $t4 = t4 - 4$
- $t2$ 也可同样处理

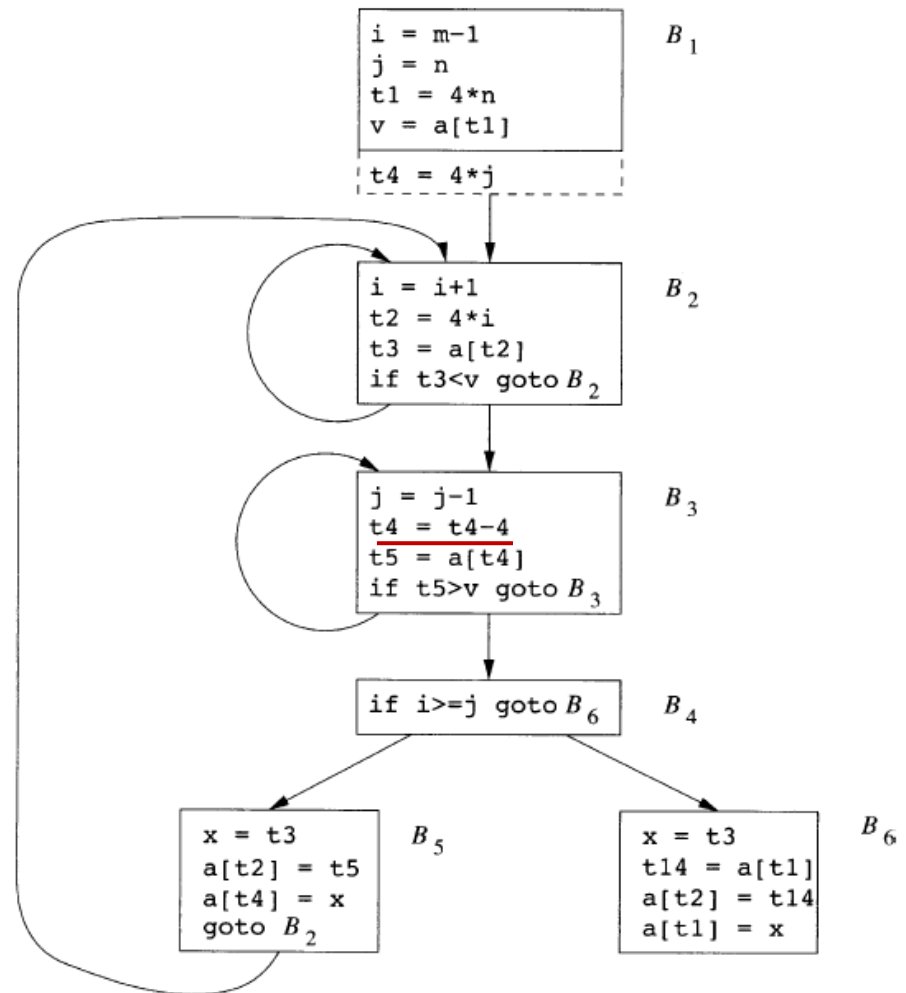


图 9-8 对基本块  $B_3$  中的  $4 * j$  应用强度消减优化

# 数据流分析

- 数据流分析

- 用于获取数据沿着程序执行路径流动信息的相关技术
- 是优化的基础

- 例如

- 两个表达式是否一定计算得到相同的值？(可用表达式→公共子表达式)
- 一个语句的计算结果是否可能被后续语句使用？(活跃变量分析→死代码消除)

# 数据流抽象 (1)

- 程序点

- 三地址语句之前或之后的位置
- 基本块内部：一个语句之后的程序点等于下一个语句之前的程序点
- 如果流图中有 $B_1$ 到 $B_2$ 的边，那么 $B_2$ 的第一个语句之前的点可能紧跟在 $B_1$ 的最后语句之后的点后面执行

- 从 $p_1$ 到 $p_n$ 的执行路径： $p_1, p_2, \dots, p_n$

- 要么 $p_i$ 是一个语句之前的点，且 $p_{i+1}$ 是该语句之后的点
- 要么 $p_i$ 是某个基本块的结尾，且 $p_{i+1}$ 是该基本块的某个后继的开头

# 数据流抽象 (2)

- 出现在某个程序点的**程序状态**
  - 在某个运行时刻，当指令指针 (PC) 指向这个程序点时，各个变量和动态内存中存放的**值**
  - 指令指针可能多次指向同一个程序点，因此一个程序点可能对应**多个**程序状态
- 数据流分析把可能出现在某个程序点上的程序状态集合总结为一些**特性**
  - 不管程序怎么运行，当它到达某个程序点时，程序状态**总是满足**分析得到的特性
  - 不同的分析技术关心不同的信息

# 例子

- 路径
  - 1, 2, 3, 4, 9
  - 1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9
- 第一次到达(5),  $a = 1$
- 第二次到达(5),  $a = 243$ 
  - 之后都是243
- 我们可以说
  - (5)具有特性 $a = 1$ 或 $a = 243$
  - 表示成为 $\langle a, \{1, 243\} \rangle$

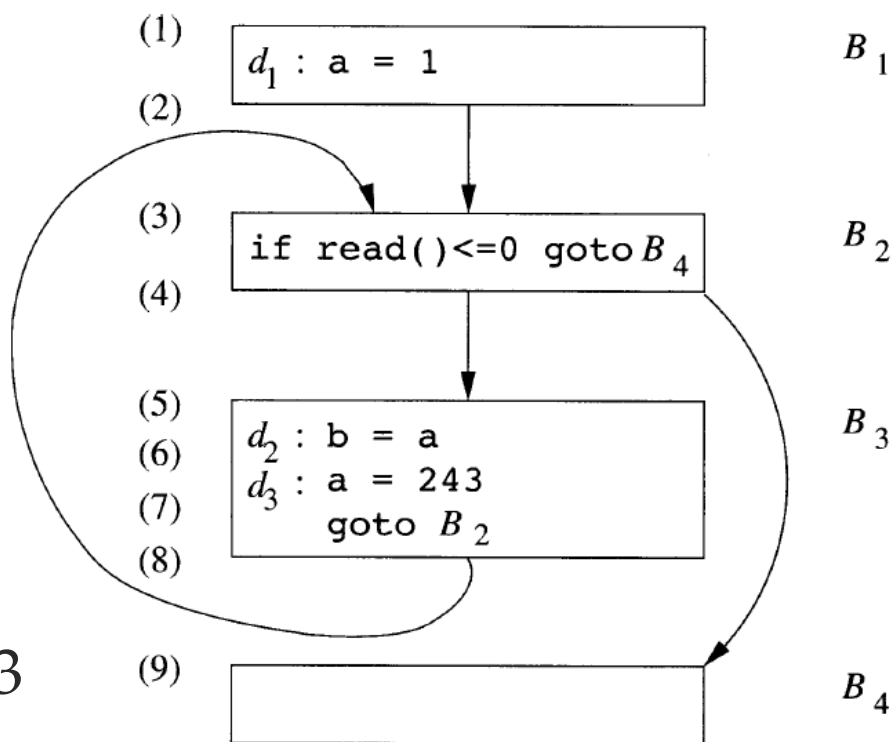


图 9-12 说明数据流抽象的例子程序

# 性质和算法

- 根据不同的需要来设置不同的性质集合，然后设计分析算法
  - 程序点上的性质被表示成为**数据流值**，求解这些数据流值实际上就是推导这些性质的过程
- 例子
  - 如果要求出变量在某个点上的值可能在哪儿定值，可以使用**到达定值 (Reaching Definition)**
    - 性质形式： $x$ 由 $d_1$ 定值
  - 如果希望实现**常量折叠优化**，我们关心的是某个点上变量 $x$ 的值是否总是由某个常量赋值语句赋予
    - 性质形式： $x = c$ ，以及 $x = \text{NAC}$

# 数据流分析模式

- 数据流分析中，程序点和数据流值关联起来
  - 数据流值表示了程序点具有的性质
  - 和某个程序点关联的数据流值：程序运行中经过这个点时必然满足的条件(安全)
- 域
  - 某个数据流所有可能值的集合称为该数据流值的域
  - 不同的应用选用不同的域，比如到达定值
    - 目标是分析在某个点上，各个变量的值由哪些语句定值
    - 因此其数据流值是定值(即三地址语句)的集合

# 数据流分析

- 对一组约束求解，得到各个点上的数据流值
  - 两类约束：基于**语句语义**和基于**控制流**
- **基于语句语义的约束**
  - 一个语句之前和之后的数据流值受到其语义的约束
  - 语句语义通常用**转换函数**表示，它把一个数据流值映射为另一个数据流值
    - $OUT[s] = f_s(IN[s])$  // 正向       $IN[s] = f_s(OUT[s])$  // 逆向
- **基于控制流的约束**
  - 在基本块内部，一个语句的输出 = 下一语句的输入
  - 流图的控制流边也对应新的约束

# 例子

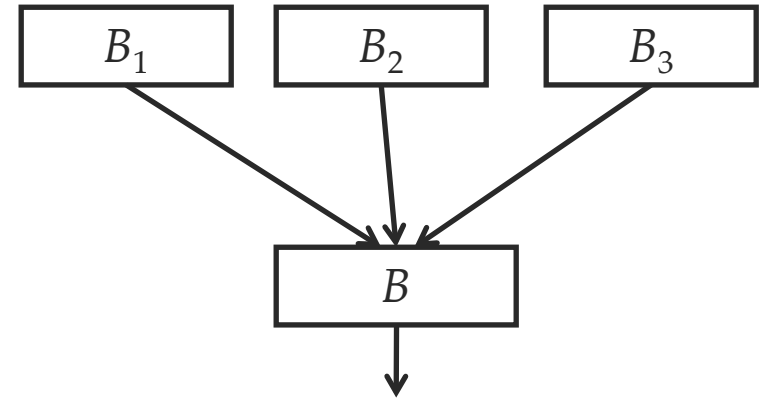
- 考虑各个变量在某个程序点上是否为常量
  - $s$ 是语句 $x = 3$
  - 考虑变量 $x, y, z$
  - 如果IN[ $s$ ]:  $x: \text{NAC}; y: 7; z: 3$
  - 那么OUT[ $s$ ]:  $x: 3; y: 7; z: 3$
- 如果
  - $s$ 是 $x = y + z$ , 那么OUT[ $s$ ]是?
  - $s$ 是 $x = x + y$ , 那么OUT[ $s$ ]是?

# 基本块内的数据流模式

- 基本块内的控制流非常简单
  - 从头到尾不会中断
  - 没有分支
- 基本块的效果就是各个语句的效果的**复合**
- 可以预先处理基本块内部的数据流关系，给出基本块对应的**转换函数**
  - $\text{OUT}[B] = f_B(\text{IN}[B])$  或  $\text{IN}[B] = f_B(\text{OUT}[B])$
- 设基本块包含语句  $s_1, s_2, \dots, s_n$ 
  - $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$

# 基本块之间的控制流约束

- 前向数据流问题
  - $B$ 的传递函数根据 $IN[B]$ 计算得到 $OUT[B]$
  - $IN[B]$ 和 $B$ 的各前驱基本块的 $OUT$ 值之间具有约束关系
- 逆向数据流问题
  - $B$ 的传递函数根据 $OUT[B]$ 计算得到 $IN[B]$
  - $OUT[B]$ 和 $B$ 的各后继基本块的 $IN$ 值之间具有约束关系



前向数据流的例子

假如

$OUT[B_1]: x: 3; y: 4; z: NAC$

$OUT[B_2]: x: 3; y: 5; z: 7$

$OUT[B_3]: x: 3; y: 4; z: 7$

则

$IN[B]: x: 3; y: NAC; z: NAC$

# 数据流方程解的精确性和安全性

- 数据流方程通常没有唯一解
- 目标是寻找一个最“精确”且满足约束的解
  - 精确：能够进行更多的改进
  - 满足约束：根据分析结果来改进代码是安全的

# 到达定值 (1)

- 到达定值

- 假定 $x$ 有定值 $d$ ，如果存在一个路径，从紧随 $d$ 的点到达某点 $p$ ，并且此路径上面没有 $x$ 的其他定值点，则称 $x$ 的定值 $d$ 到达 $p$
- 如果在这条路径上有对 $x$ 的其它定值，我们说变量 $x$ 的这个定值 $d$ 被杀死了

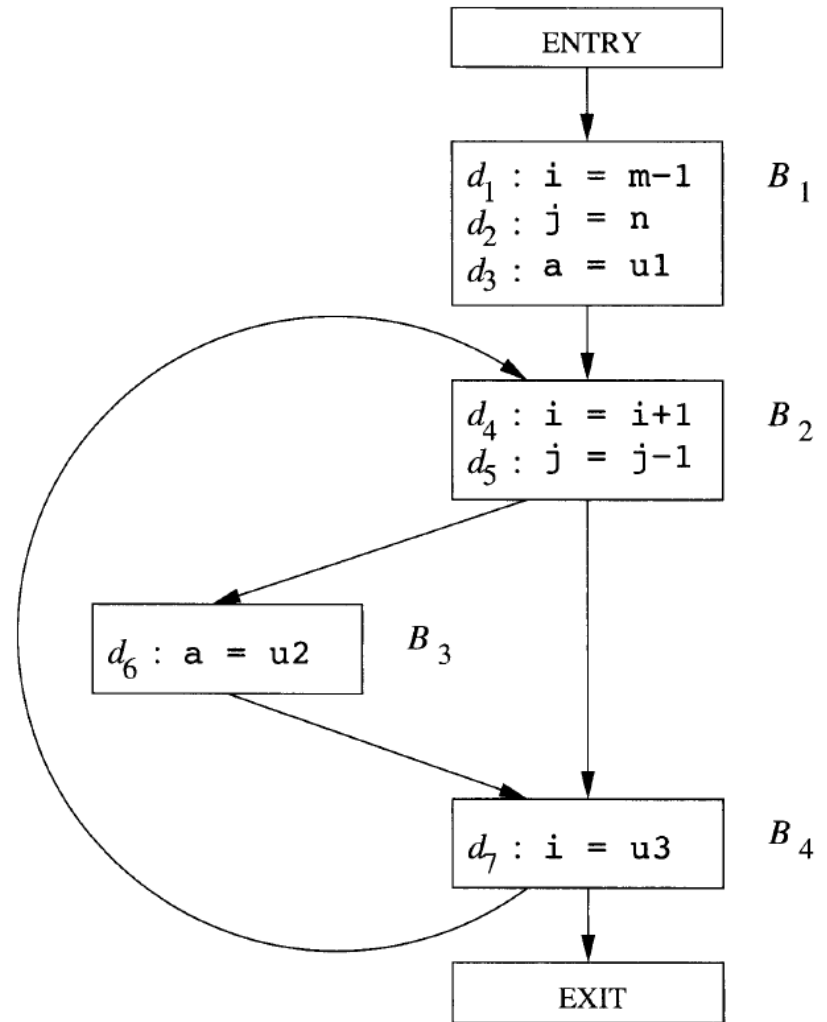
- 如果某个变量 $x$ 的一个定值 $d$ 到达了点 $p$ ，在 $p$ 点使用变量 $x$ 的时候， $x$ 的值是由 $d$ 最后定值的

# 到达定值 (2)

- 到达定值的解允许不精确，但必须是安全的
  - 分析得到的到达定值可能实际上不会到达
  - 但是实际到达的一定被分析出来，否则不安全
- 比如确定 $x$ 在 $p$ 点是否为常量
  - 忽略实际的到达定值使得变化的值被误认为常量，将这些值替换为常量会引起错误，不安全
  - 过多估计则相反

# 到达定值的例子

- $B_1$ 全部定值到达 $B_2$ 的开头
- $d_5$ 到达 $B_2$ 的开头 (循环)
- $d_2$ 被 $d_5$ 杀死, 不能到达 $B_3$ 、 $B_4$ 的开头
- $d_4$ 不能到达 $B_2$ 的开头, 因为被 $d_7$ 杀死
- $d_6$ 到达 $B_2$ 的开头



# 语句/基本块的转换方程 (1)

- 定值  $d: u = v + w$ 
  - 生成了对变量  $u$  的定值  $d$ ，杀死其它对  $u$  的定值
  - 生成-杀死形式:  $f_d(x) = gen_d \cup (x - kill_d)$
  - $gen_d = \{ d \}$ ,  $kill_d = \{ \text{程序中其它对 } u \text{ 的定值} \}$
- 生成-杀死形式的函数复合仍具有该形式
  - $f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2)$   
 $= (gen_2 \cup (gen_1 - kill_2)) \cup (x - kill_1 \cup kill_2)$
  - 生成的定值: 由第二部分生成、以及由第一部分生成且没有被第二部分杀死
  - 杀死的定值: 被第一部分杀死、以及被第二部分杀死的定值

## 语句/基本块的转换方程 (2)

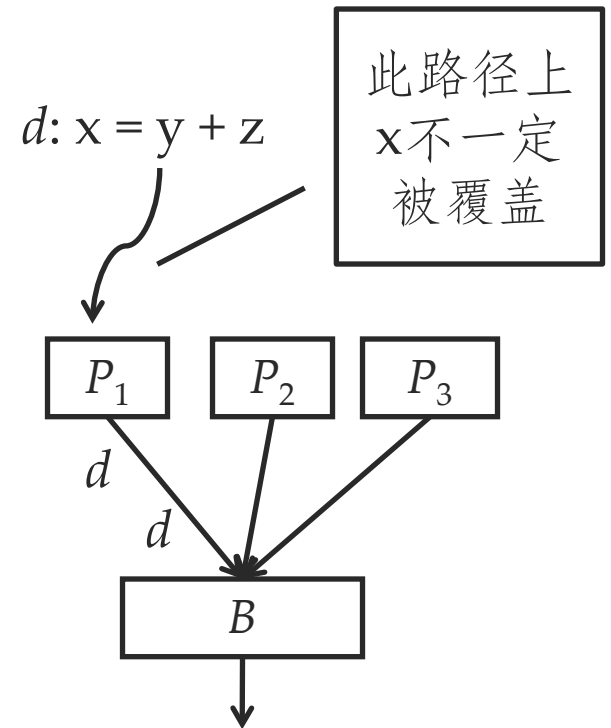
- 设 $B$ 有 $n$ 个语句，第 $i$ 个语句的转换函数为 $f_i$
- $f_B(x) = gen_B \cup (x - kill_B)$ 
  - $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup \dots \cup (gen_1 - kill_2 - kill_3 - \dots - kill_n)$
  - $kill_B = kill_1 \cup kill_2 \cup \dots \cup kill_n$
  - $gen_B$ 是被第 $i$ 个语句生成，且没有被其后的句子杀死的定值的集合
  - $kill_B$ 为被 $B$ 各个语句杀死的定值的并集

# *gen*和*kill*的例子

- 基本块
  - $d_1: a = 3$
  - $d_2: a = 4$
- *gen*集合:  $\{d_2\}$
- *kill*集合:  $\{\text{流图中所有针对}a\text{的定值}\}$

# 到达定值的控制流方程

- 只要一个定值能够沿某条路径到达一个程序点，这个定值就是到达定值
- $IN[B] = \bigcup_{P \text{ 是 } B \text{ 的前驱基本块}} OUT[P]$ 
  - 如果从基本块 $P$ 到 $B$ 有一条控制流边，那么 $OUT[P]$ 在 $IN[B]$ 中
  - 一个定值必然先在某个前驱的 $OUT$ 值中，才能出现在 $B$ 的 $IN$ 中
- $\cup$ 称为到达定值的交汇运算符



# 控制流方程的迭代解法 (1)

- ENTRY基本块的传递函数是常函数
  - $OUT[ENTRY] = \text{空集}$
- 其他基本块
  - $OUT[B] = gen_B \cup (IN[B] - kill_B)$  // 基本块内部
  - $IN[B] = \cup_{P \text{ 是 } B \text{ 的前驱基本块}} OUT[P]$  // 基本块之间
- 迭代解法
  - 首先求出各基本块的 $gen_B$ 和 $kill_B$
  - 令所有的 $OUT[B]$ 都是空集，然后不停迭代，得到最小不动点的解

## 控制流方程的迭代解法 (2)

- 输入：流图、各基本块的 $kill$ 和 $gen$ 集合
- 输出： $IN[B]$ 和 $OUT[B]$
- 方法

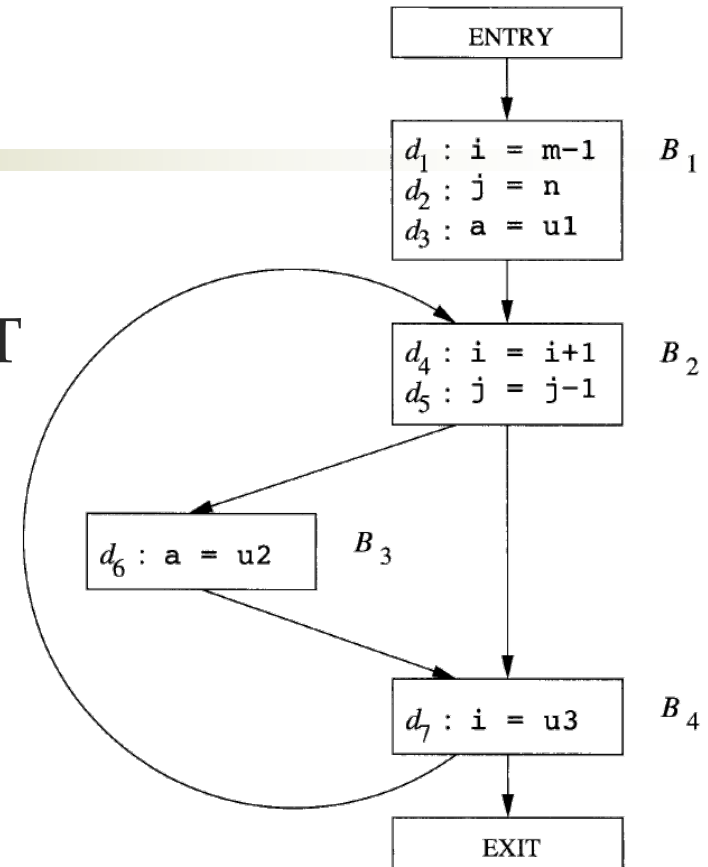
```
1)   $OUT[ENTRY] = \emptyset;$   
2)  for (除  $ENTRY$  之外的每个基本块  $B$ )  $OUT[B] = \emptyset;$   
3)  while (某个  $OUT$  值发生了改变)  
4)      for (除  $ENTRY$  之外的每个基本块  $B$ ) {  
5)           $IN[B] = \bigcup_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P];$   
6)           $OUT[B] = gen_B \cup (IN[B] - kill_B);$   
      }
```

# 控制流方程的迭代解法 (3)

- 解法的正确性
  - 不断向前传递各个定值，直到该定值被杀死为止
- 为什么不会出现死循环？
  - 各个OUT[B]在算法执行过程中不会变小
  - 且OUT[B]显然有有穷的上界
  - 只有一次迭代之后增大了某个OUT[B]的值，算法才会进行下一次迭代
- 最大的迭代次数是流图的结点数 $n$ 
  - 定值经过 $n$ 步必然已经到达所有可能到达的结点
- 结束时，各个OUT/IN值必然满足数据流方程

# 到达定值求解的例子

- 7个bit从左到右表示 $d_1, d_2, \dots, d_7$
- for循环时依次遍历 $B_1, B_2, B_3, B_4, \text{EXIT}$
- 每一列表示一次迭代计算
- $B_1$ 生成 $d_1, d_2, d_3$ , 杀死 $d_4, d_5, d_6, d_7$
- $B_2$ 生成 $d_4, d_5$ , 杀死 $d_1, d_2, d_7$
- $B_3$ 生成 $d_6$ , 杀死 $d_3$
- $B_4$ 生成 $d_7$ , 杀死 $d_1, d_4$



Block $B$	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
$B_1$	000 0000	000 0000	111 0000	000 0000	111 0000
$B_2$	000 0000	111 0000	001 1100	111 0111	001 1110
$B_3$	000 0000	001 1100	000 1110	001 1110	000 1110
$B_4$	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

# 活跃变量分析

- 活跃变量分析
  - $x$ 在 $p$ 上的值是否会在某条从 $p$ 出发的路径中使用
  - 变量 $x$ 在 $p$ 上活跃，当且仅当存在一条从 $p$ 开始的路径，该路径的末端使用了 $x$ ，且路径上没有对 $x$ 进行定值
- 用途
  - 寄存器分配/死代码删除/...
- 数据流值
  - (活跃)变量的集合

# *use* 与 *def*

- 语句的转换函数
  - $s: x = y + z$
  - $use_s = \{ y, z \}$
  - $def_s = \{ x \}$
  
- 假设基本块中包含语句  $s_1, s_2, \dots, s_n$ , 那么
  - $use_B = use_1 \cup (use_2 - def_1) \cup (use_3 - def_1 - def_2) \cup \dots \cup (use_n - def_1 - def_2 \dots - def_{n-1})$
  - $def_B = def_1 \cup def_2 \cup def_3 \cup \dots \cup def_n$

# 基本块内的数据流方程

- 基本块的转换函数仍然是生成-杀死形式，但是从OUT值计算出IN值 (逆向)
  - $use_B$ : 在 $B$ 中先于定值被使用
  - $def_B$ : 在 $B$ 中被定值
- 例子
  - 基本块 $B_2$ :  
 $i = i + 1;$   
 $j = j - 1;$ 
    - $use = \{ i, j \}$
    - $def = \{ i, j \}$

# 活跃变量数据流方程

- 任何变量在程序出口处不再活跃
  - $IN[EXIT] = \text{空集}$
- 对于所有非EXIT基本块
  - $IN[B] = use_B \cup (OUT[B] - def_B)$  // 基本块内部
  - $OUT[B] = \cup_{S \text{ 是 } B \text{ 的后继基本块}} IN[S]$  // 基本块之间
- 和到达定值相比较
  - 都使用并集运算 $\cup$ 作为交汇运算
  - 数据流值传递方向相反，因此初始化的值 (IN) 不一样

# 活跃变量分析的迭代方法

- 这个算法中 $IN[B]$ 总是越来越大，且 $IN[B]$ 都有上界，因此必然会停机

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

# 可用表达式分析

- $x + y$ 在 $p$ 点可用的条件
  - 从流图入口结点到达 $p$ 的**每条路径**都对 $x + y$ 求值，且在最后一次求值之后再没有对 $x$ 或 $y$ 赋值
- 主要用途
  - 寻找**全局公共子表达式**
- 生成-杀死
  - **生成**：基本块求值 $x + y$ ，且之后没有对 $x$ 或 $y$ 赋值，那么它生成了 $x + y$
  - **杀死**：基本块对 $x$ 或 $y$ 赋值，且没有重新计算 $x + y$ ，那么它杀死了 $x + y$

# 计算基本块生成的表达式

- 初始化  $S = \{\}$
- 从头到尾逐个处理基本块中的指令  $x = y + z$ 
  - 把  $y + z$  添加到  $S$  中
  - 从  $S$  中删除任何涉及变量  $x$  的表达式
- 遍历结束时得到基本块 **生成** 的表达式集合
- **杀死** 的表达式集合
  - 表达式的某个分量在基本块中被定值，并且该表达式没有被再次生成

# 基本块生成/杀死的表达式的例子

语 句	可用表达式
	$\emptyset$
$a = b + c$	$\{b + c\}$
$b = a - d$	$\{a - d\}$
$c = b + c$	$\{a - d\}$
$d = a - d$	$\emptyset$

# 可用表达式的数据流方程

- ENTRY结点的出口处没有可用表达式
  - $OUT[ENTRY] = \text{空集}$
- 其他基本块的方程
  - $OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$  // 基本块内部
  - $IN[B] = \bigcap_{P \text{ 是 } B \text{ 的前驱基本块}} OUT[P]$  // 基本块之间
- 和其他方程类比
  - 前向传播
  - 交汇运算是交集运算

# 可用表达式分析的迭代方法

- 注意：OUT值的初始化值是全集

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

# 三种数据流方程的总结

	到达定值	活跃变量	可用表达式
域	Sets of definitions	Sets of variables	Sets of expressions
方向	Forwards	Backwards	Forwards
传递函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
边界条件	$OUT[ENTRY] = \emptyset$	$IN[EXIT] = \emptyset$	$OUT[ENTRY] = \emptyset$
交汇运算( $\wedge$ )	$\cup$	$\cup$	$\cap$
方程组	$OUT[B] = f_B(IN[B])$ $IN[B] =$ $\bigwedge_{P, pred(B)} OUT[P]$	$IN[B] = f_B(OUT[B])$ $OUT[B] =$ $\bigwedge_{S, succ(B)} IN[S]$	$OUT[B] = f_B(IN[B])$ $IN[B] =$ $\bigwedge_{P, pred(B)} OUT[P]$
初始值	$OUT[B] = \emptyset$	$IN[B] = \emptyset$	$OUT[B] = U$

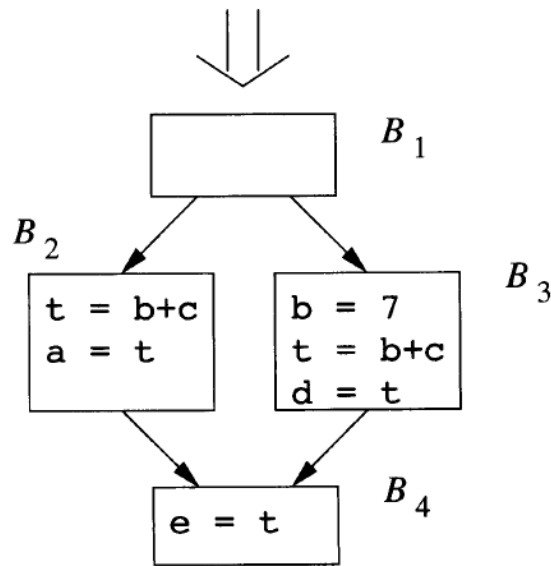
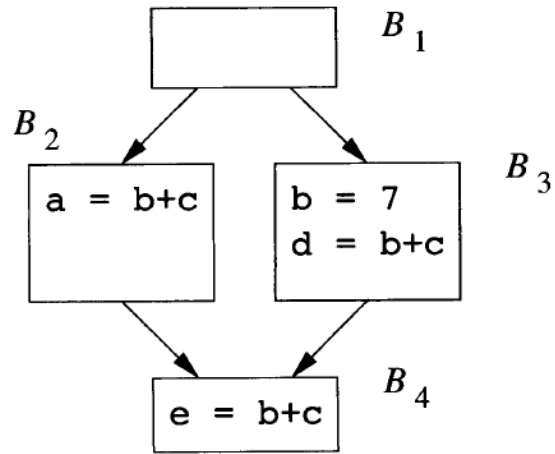
# 数据流分析的理论基础

- 问题
  - 数据流分析中的迭代算法在什么情况下正确?
  - 得到的解有多精确?
  - 迭代算法是否收敛?
  - 方程组的解的含义是什么?
- 通过定义一个数据流分析框架，利用离散数学中半格、偏序等概念和性质，给出数据流分析算法的理论依据

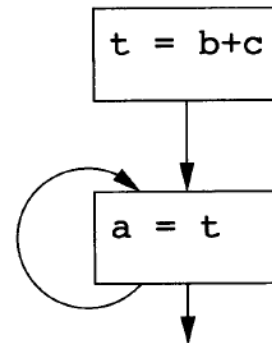
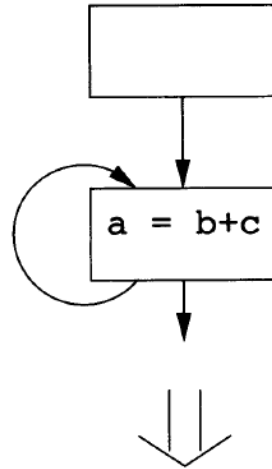
# 消除冗余计算

- 目标：尽量减少表达式求值的次数
- 对于表达式  $x + y$ 
  - 全局公共子表达式：如果对  $x + y$  求值前的程序点上  $x + y$  可用，那么不需要再对  $x + y$  求值
  - 循环不变表达式：循环中的表达式  $x + y$  的值不变，可以只计算一次
  - 部分冗余：在程序按照某些路径到达这个点的时候  $x + y$  已经被计算过，但沿着另外一些路径到达时， $x + y$  尚未计算过
- 需要更复杂的数据流分析技术

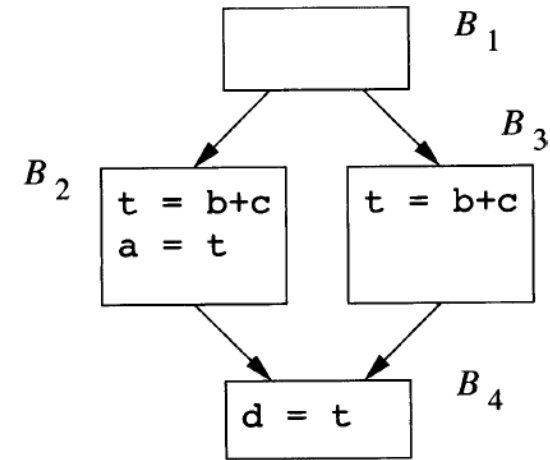
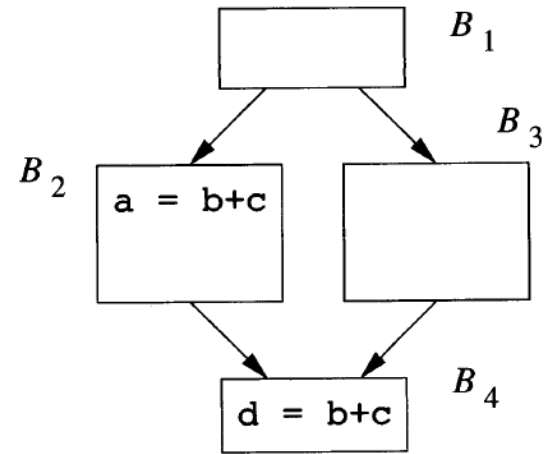
# 冗余的例子



a) 公共子表达式



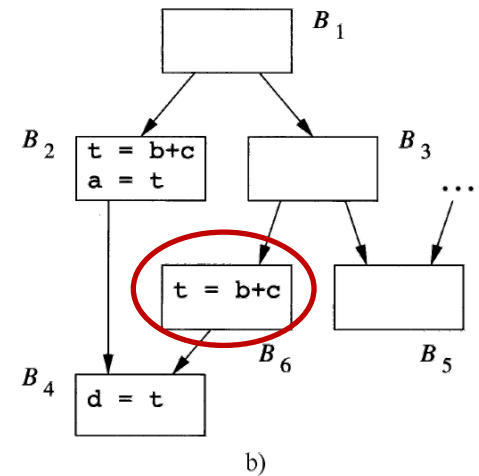
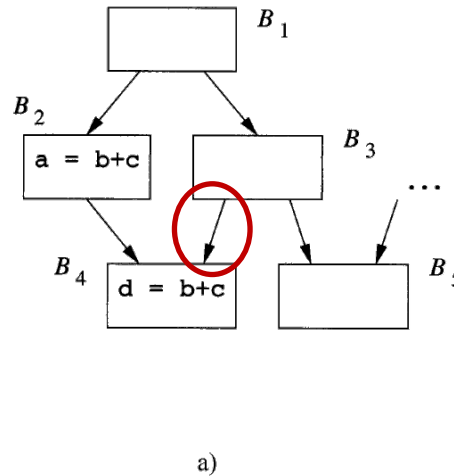
b) 循环不变代码移动



c) 部分冗余消除

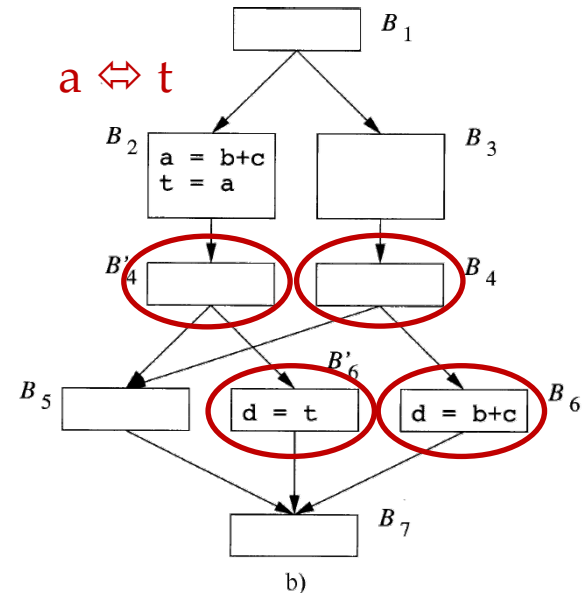
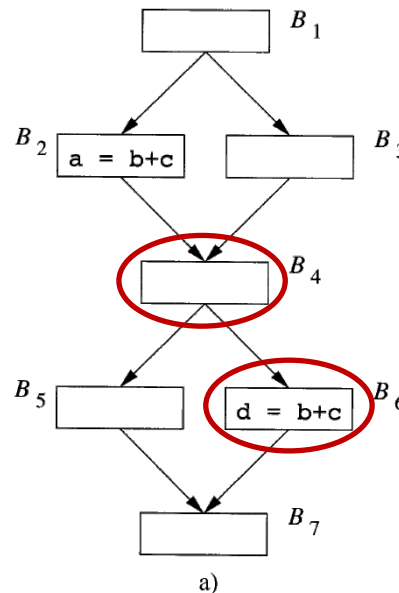
# 需要添加基本块来消除的冗余

- 进行两种操作
  - 在关键边上增加基本块
  - 进行代码复制



## 关键边

- 从具有多个后继的结点到达具有多个前驱的结点



# 懒惰代码移动

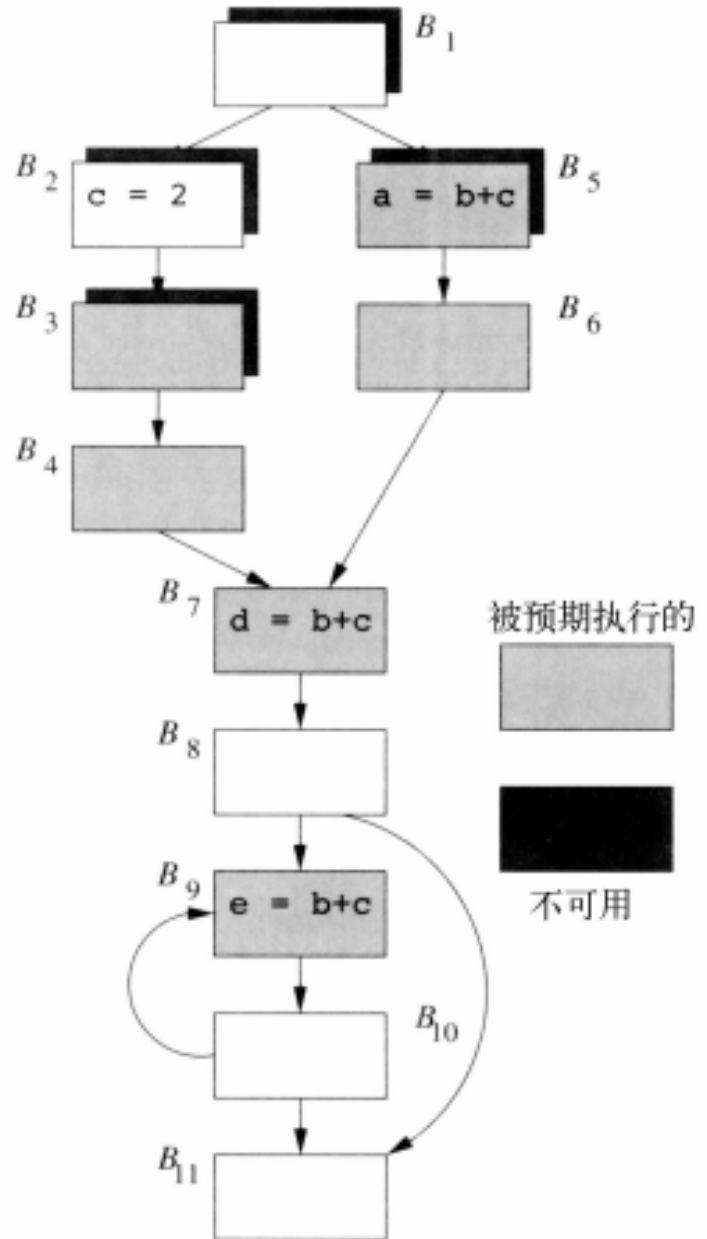
- 目标
  - 所有不复制代码就可消除的冗余计算都被消除
  - 优化后的代码不会执行原程序中不执行的任何计算
  - 表达式的计算应该**尽量靠后**，以利于寄存器的分配
- 冗余消除
  - 完全冗余
  - 部分冗余：在流图中放置表达式 $x + y$ 的拷贝，使得某处的 $x + y$ 成为完全冗余，从而删除

# 基本步骤

- 按照如下四个步骤进行处理
  - 1. 找出各程序点上**预期执行**的所有表达式
  - 2. 在表达式被预期执行但是**不可用**的程序点上，放置表达式的计算
  - 3. 把表达式尽量**后延**到某个程序点，在到达这个点的所有路径上，这个表达式在这个程序点之前被预期执行，但是还没有使用这个值
  - 4. 消除只使用一次的临时变量

# 1. 预期执行表达式 (1)

- 被预期执行
  - 如果从程序点 $p$ 出发的所有路径都会计算表达式 $b + c$ 的值，并且 $b$ 和 $c$ 在那时的值就是它们在点 $p$ 的值，那么表达式 $b + c$ 在点 $p$ 上被预期执行
- 例子
  - 表达式 $b + c$ 在 $B_3$ 、 $B_4$ 、 $B_5$ 、 $B_6$ 、 $B_7$ 和 $B_9$ 的入口处被预期执行



# 1. 预期执行表达式 (2)

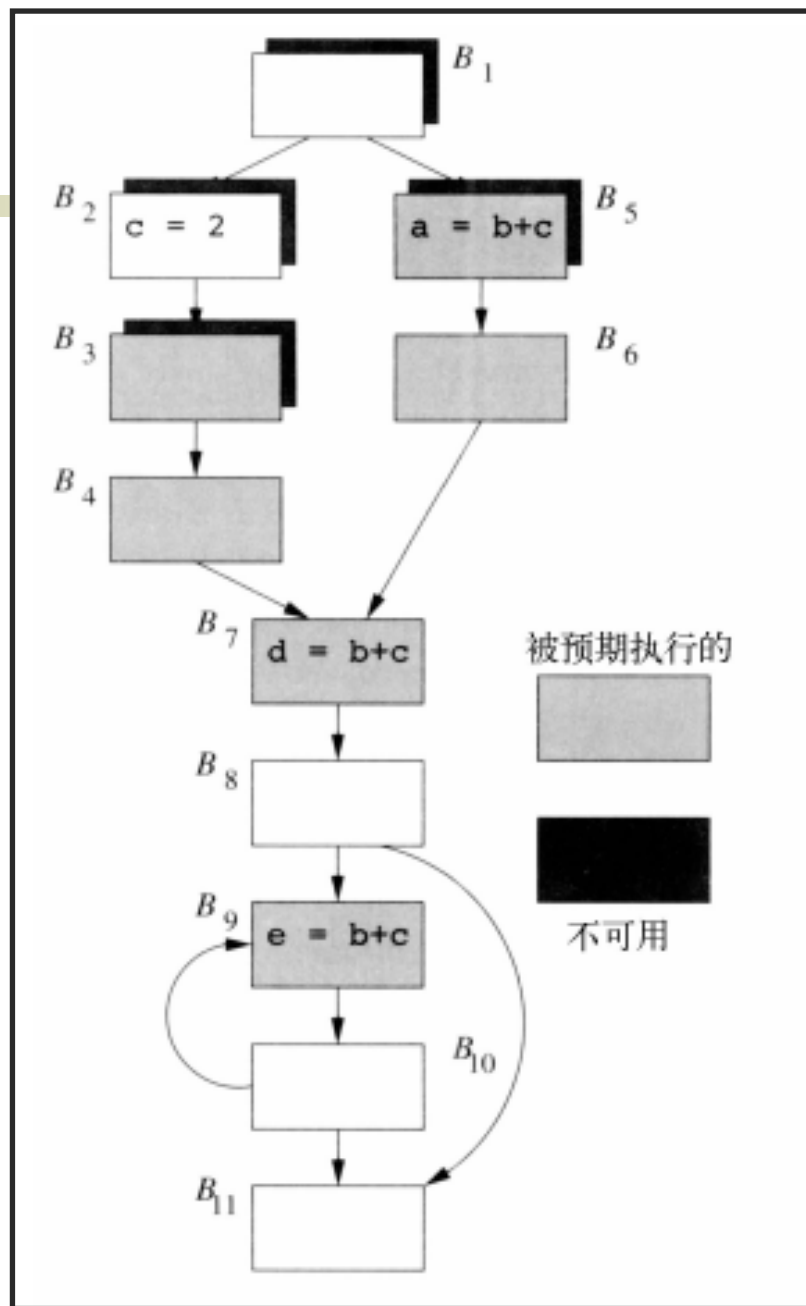
- 数据流分析框架
  - 逆向分析
  - 基本块内部：当表达式在 $B$ 出口处被预期执行，且它没有被 $B$ 杀死，那么它在 $B$ 入口处也被预期执行
  - 基本块之间：当在 $B$ 的所有后继基本块的入口处都被预期执行，那么表达式在 $B$ 出口处被预期执行
  - 在整个程序的出口处，没有表达式被预期执行
- 求出预期执行点之后，表达式被放置到首次被预期执行的程序点上
  - 此时一些表达式变得完全冗余

## 2. 可用表达式

- 和前面的可用表达式类似，但假设代码已经被复制到了预期执行点上
- 表达式在基本块的**出口处可用**的条件
  - 在基本块的入口处可用，或在基本块的入口处的预期执行表达式中
  - 且没有被这个基本块杀死

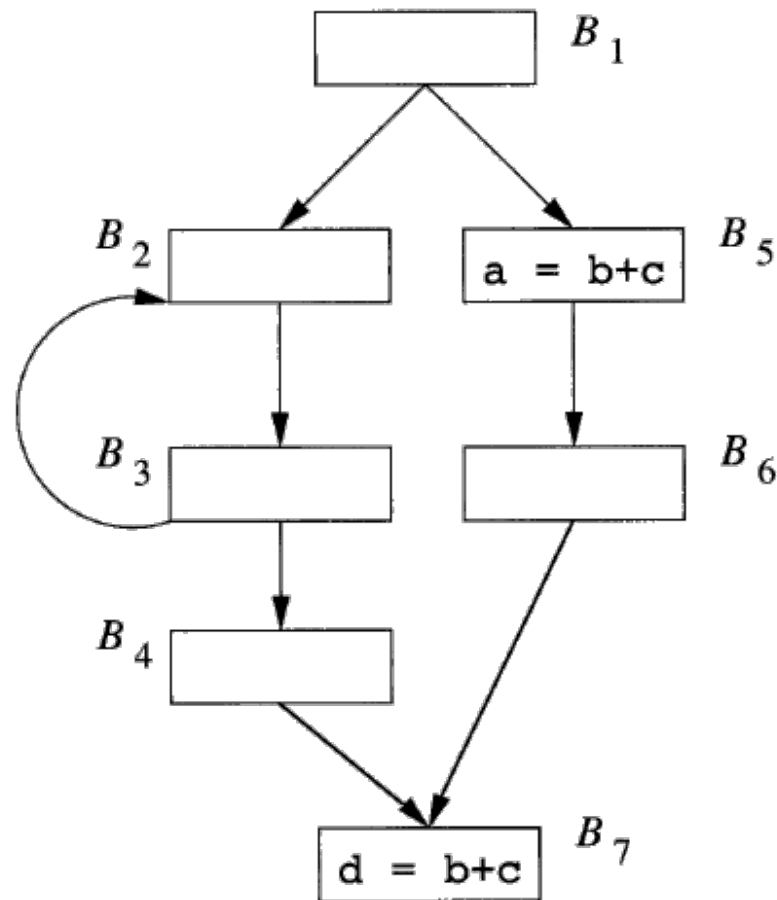
## 2. 例子

- 表达式  $b + c$  在  $B_1$ 、 $B_2$ 、 $B_3$  和  $B_5$  不可用



### 3. 可后延表达式 (1)

- 在保持程序语义的情况下，尽可能延后计算表达式
- $x + y$ 可后延到 $p$ 的条件
  - 所有从程序入口到达 $p$ 的路径中都会碰到一个位置较前的 $x + y$ ，且在最后一个这样的位置到 $p$ 之间没有使用 $x + y$
- 右边的例子
  - $b + c$ 在 $B_1$ 被预期执行
  - $b + c$ 可后延到 $B_4 \rightarrow B_7$ 的边上

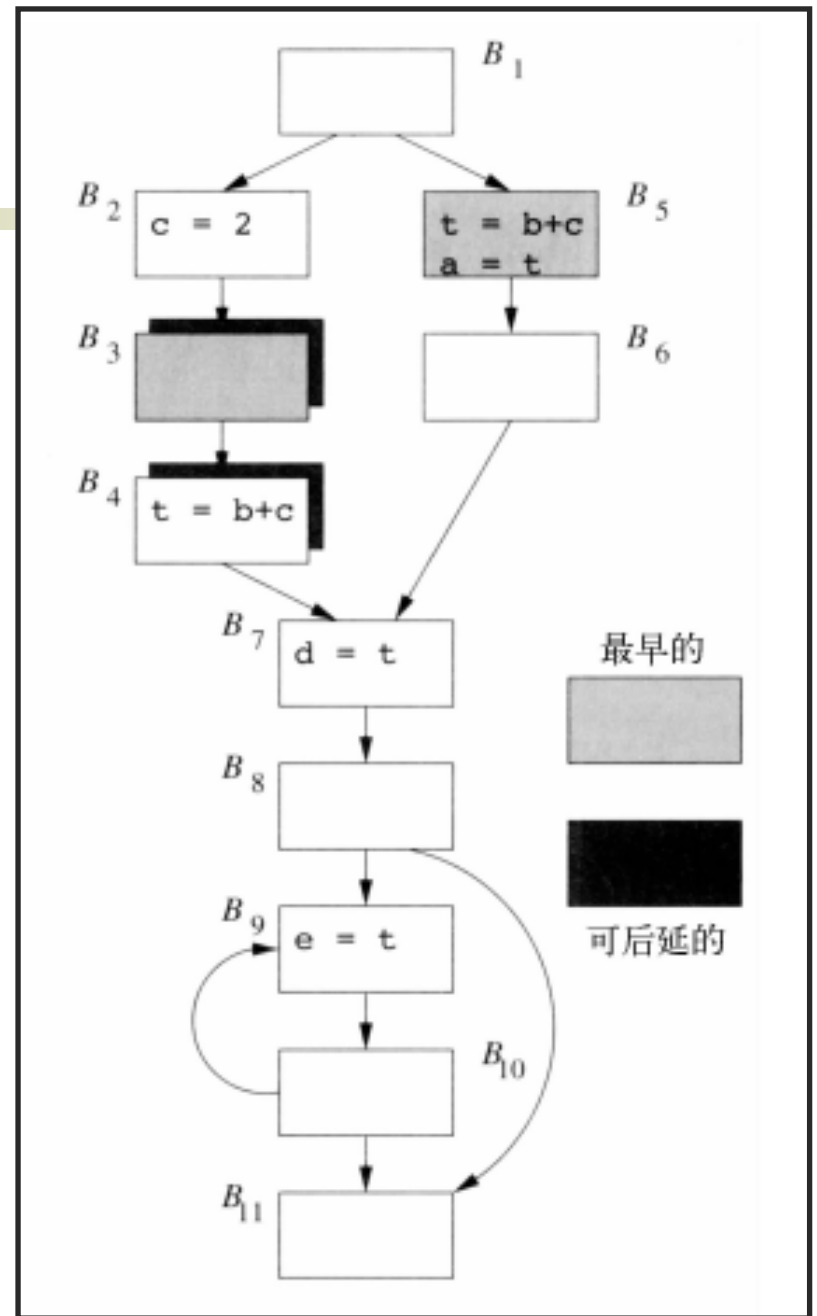


### 3. 可后延表达式 (2)

- 粗略地说，一个表达式将被放置在**边界**上，即一个表达式从**可后延**变成**不可后延**的地方

# 3. 例子

- 可放置 $b + c$ 的两个最早的点是 $B_3$ 和 $B_5$ 
  - 不能从 $B_5$ 后延到 $B_6$
  - 但可以从 $B_3$ 后延到 $B_4$



## 4. 被使用的表达式

- 确定一个被引入的临时变量是否在其所在基本块之外的其它地方使用
  - 对表达式的活跃性分析
  - 如果从程序点 $p$ 出发的一条路径在表达式被重新求值之前使用了该表达式，那么该表达式在点 $p$ 上使用

	a) 被预期执行的表达式	b) 可用表达式
域	Sets of expressions	Sets of expressions
方向	逆向	Forwards
传递函数	$f_B(x) = e\_use_B \cup (x - e\_kill_B)$	$f_B(x) = (anticipated[B].in \cup x) - e\_kill_B$
边界条件	$IN[EXIT] = \emptyset$	$OUT[ENTRY] = \emptyset$
交汇运算( $\wedge$ )	$\cap$	$\cap$
方程组	$IN[B] = f_B(OUT[B])$ $OUT[B] = \bigwedge_{S, succ(B)} IN[S]$	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P, pred(B)} OUT[P]$
初始化	$IN[B] = U$	$OUT[B] = U$
	c) 可后延表达式	d) 被使用的表达式
域	Sets of expressions	Sets of expressions
方向	前向	Backwards
传递函数	$f_B(x) = (earliest[B] \cup x) - e\_use_B$	$f_B(x) = (e\_use_B \cup x) - latest[B]$
边界条件	$OUT[ENTRY] = \emptyset$	$IN[EXIT] = \emptyset$
交汇运算( $\wedge$ )	$\cap$	$\cup$
方程组	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P, pred(B)} OUT[P]$	$IN[B] = f_B(OUT[B])$ $OUT[B] = \bigwedge_{S, succ(B)} IN[S]$
初始化	$OUT[B] = U$	$IN[B] = \emptyset$
$earliest[B] = anticipated[B].in - available[B].in$ $latest[B] = (earliest[B] \cup postponable[B].in) \cap (e\_use_B \cup \neg(\bigcap_{S, succ[B]} (earliest[S] \cup postponable[S].in)))$		

# 流图中的循环

- 循环的重要性
  - 程序的大部分执行时间都花在循环上
  - 也是数据流分析需要经过若干次迭代的原因
- 相关概念
  - 支配结点
  - 深度优先排序
  - 回边
  - 图的深度
  - 可归约性

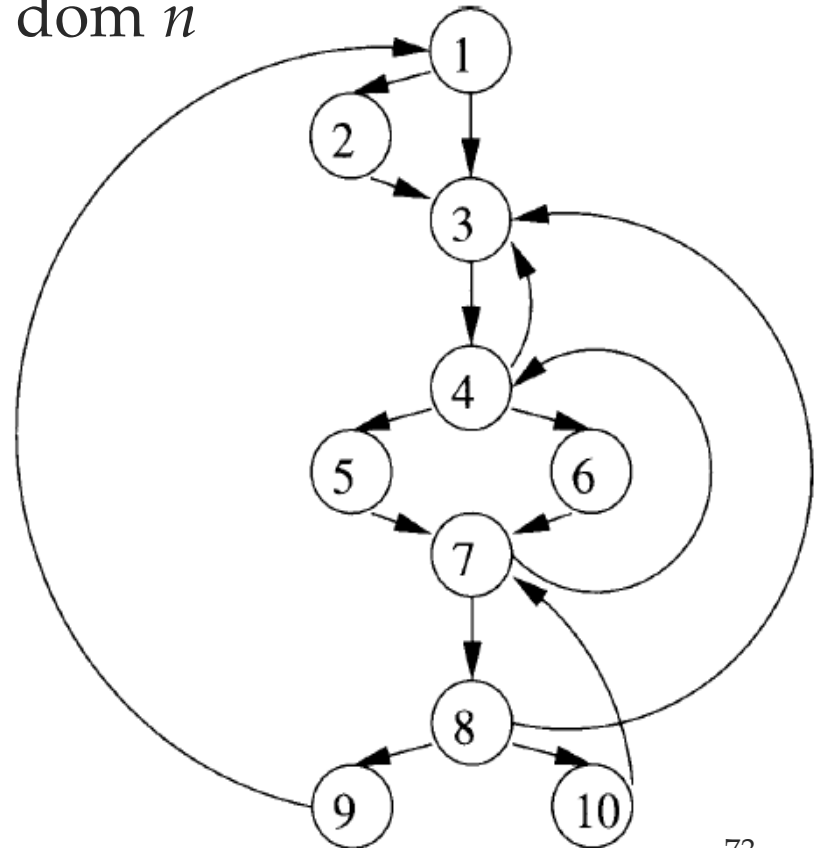
# 支配结点

- 支配 (Dominate)

- 如果每条从入口结点到达 $n$ 的路径都经过 $d$ ，那么 $d$ 支配 $n$ ，记为 $d \text{ dom } n$

- 例子

- 1支配所有结点
- 2只支配自己
- 3支配除了1、2外的其它结点
- 4支配1、2、3外的其它结点
- 5、6只支配自身
- 7支配7、8、9、10
- 8支配8、9、10
- 9、10只支配自身

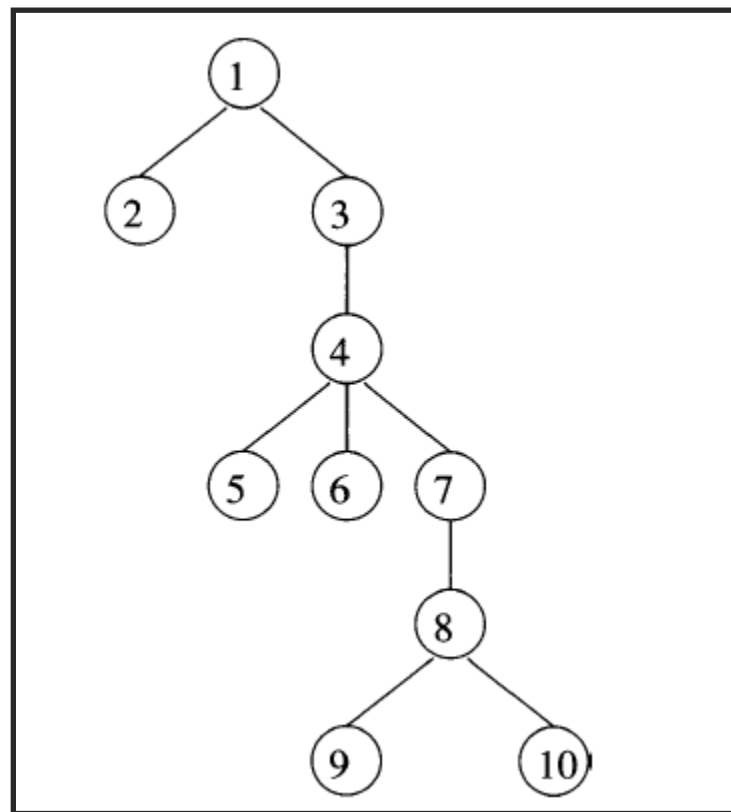
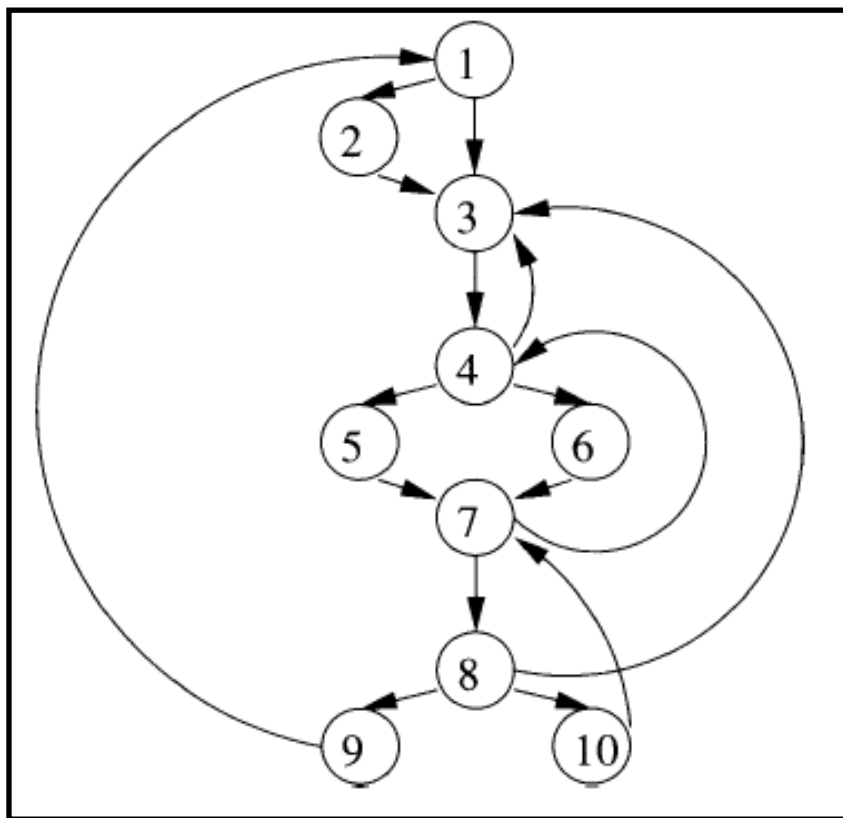


# 支配结点树 (1)

- 支配结点树可以表示支配关系
  - 根结点：入口结点
  - 每个结点 $d$ 支配且只支配树中的后代结点
- 直接支配结点
  - 从入口结点到达 $n$ 的任何路径 (不含 $n$ ) 中，它是路径中最后一个支配 $n$ 的结点
  - 前面的例子：1直接支配3，3直接支配4
  - $n$ 的直接支配结点 $m$ 具有如下性质：如果 $d \neq n$ 且 $d \text{ dom } n$ ，那么 $d \text{ dom } m$

# 支配结点树 (2)

- 例子



# 寻找支配结点算法 (1)

- 计算流图中各个结点 $n$ 的所有支配结点
  - $p_1, p_2, \dots, p_k$ 是 $n$ 的所有前驱且 $d \neq n$ , 那么 $d \text{ dom } n$ 当且仅当 $d \text{ dom } p_i (1 \leq i \leq k)$
- 一个结点的支配结点集合是它的所有前驱的支配结点集合的交集 (加上它自己)
- 前向数据流分析问题

# 寻找支配结点算法 (2)

- 求解如图所示的数据流方程组，可以得到各结点对应的支配结点集合
- $D(n) = \text{OUT}[n]$

	支配结点
域	The power set of $N$
方向	Forwards
传递函数	$f_B(x) = x \cup \{B\}$
边界条件	$\text{OUT}[\text{ENTRY}] = \{\text{ENTRY}\}$
交汇运算( $\wedge$ )	$\cap$
方程式	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P, \text{pred}(B)} \text{OUT}[P]$
初始化设置	$\text{OUT}[B] = N$

图 9-40 一个计算支配结点的数据流算法

# 例子

```
1)  $OUT[ENTRY] = v_{ENTRY};$   
2) for (除 ENTRY 之外的每个基本块  $B$ )  $OUT[B] = \top;$   
3) while (某个 OUT 值发生了改变)  
4)   for (除 ENTRY 之外的每个基本块  $B$ ) {  
5)      $IN[B] = \bigwedge_{P \text{ 是 } B \text{ 的一个前驱}} OUT[P];$   
6)      $OUT[B] = f_B(IN[B]);$   
   }
```

$$D(1) = \{1\}$$

$$D(2) = \{2\} \cup D(1) = \{1, 2\}$$

$$D(3) = \{3\} \cup (\{1\} \cap \{1, 2\} \cap \{1, 2, \dots, 10\} \cap \{1, 2, \dots, 10\}) = \{1, 3\}$$

$$D(4) = \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1, 3\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4\}$$

$$D(5) = \{5\} \cup D(4) = \{5\} \cup \{1, 3, 4\} = \{1, 3, 4, 5\}$$

$$D(6) = \{6\} \cup D(4) = \{6\} \cup \{1, 3, 4\} = \{1, 3, 4, 6\}$$

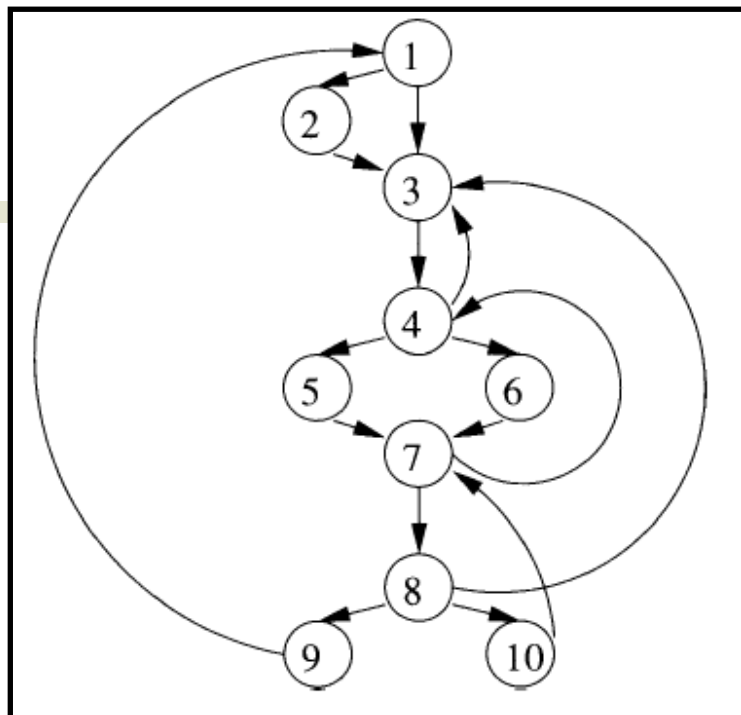
$$D(7) = \{7\} \cup (D(5) \cap D(6) \cap D(10))$$

$$= \{7\} \cup (\{1, 3, 4, 5\} \cap \{1, 3, 4, 6\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4, 7\}$$

$$D(8) = \{8\} \cup D(7) = \{8\} \cup \{1, 3, 4, 7\} = \{1, 3, 4, 7, 8\}$$

$$D(9) = \{9\} \cup D(8) = \{9\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 9\}$$

$$D(10) = \{10\} \cup D(8) = \{10\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 10\}$$

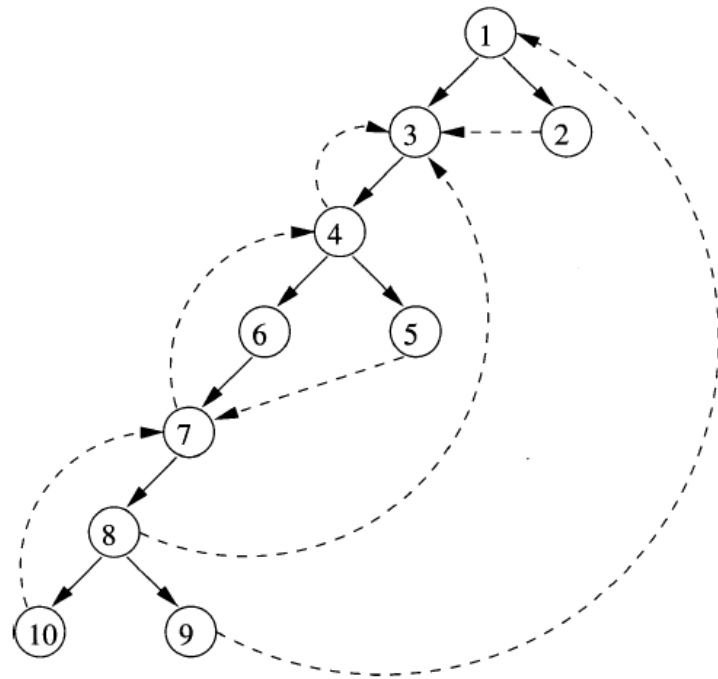
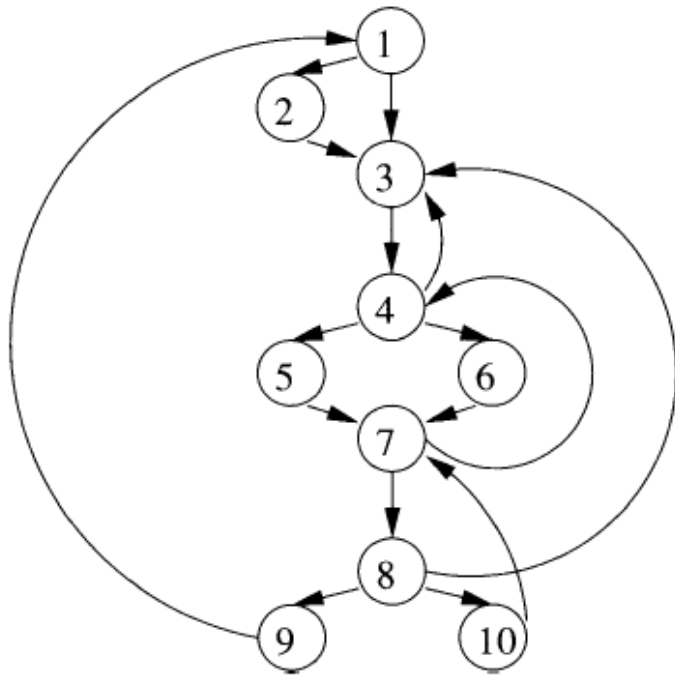


# 深度优先生成树

- 深度优先搜索 (Depth-First Search / DFS)
  - 搜索过程从入口结点开始，并首先访问已知离入口结点最远的结点
- 深度优先生成树
  - 一个深度优先过程中的搜索路线形成了一个深度优先生成树 (Depth-First Spanning Tree / DFST)

# 例子

- 左边：流图，右边：深度优先生成树
  - 实线边形成了这棵树，虚线边是流图中其它的边
  - 深度优先搜索：1-3-4-6-7-8-10-8-9-8-7-.....



# 深度优先排序

- 前序遍历

- 先访问一个结点，然后从左到右递归地访问该结点的子结点

- 后序遍历

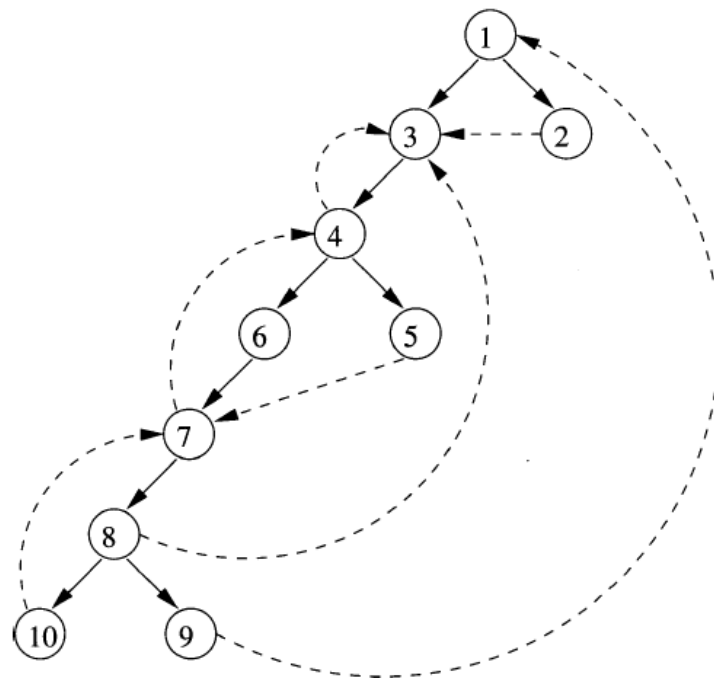
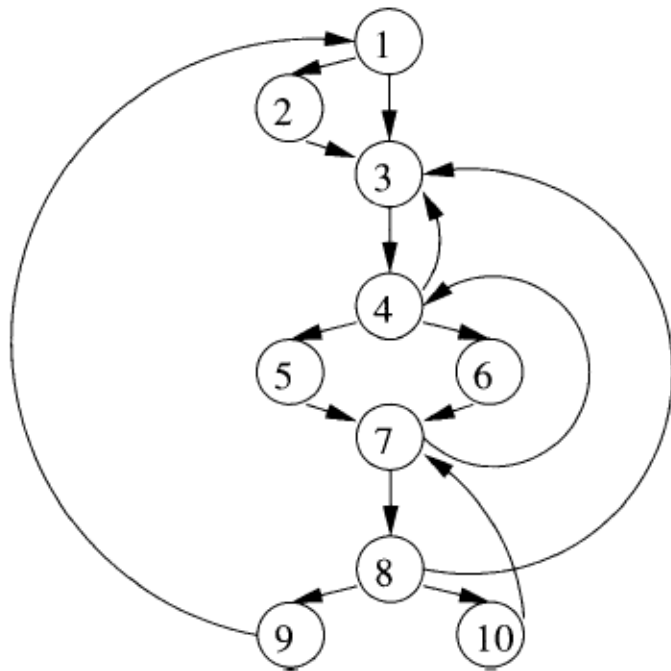
- 首先递归地从左到右访问一个结点的子结点，然后访问该结点

- 深度优先排序

- 首先访问一个结点，然后递归访问该结点的最右子结点，再递归访问这个子结点左边的子结点，依次类推 (与后序遍历的顺序相反)

# 例子

- 前序遍历：1, 3, 4, 6, 7, 8, 10, 9, 5, 2
- 后序遍历：10, 9, 8, 7, 6, 5, 4, 3, 2, 1
- 深度优先排序：1, 2, 3, 4, 5, 6, 7, 8, 9, 10



# 深度优先生成树和深度优先排序算法

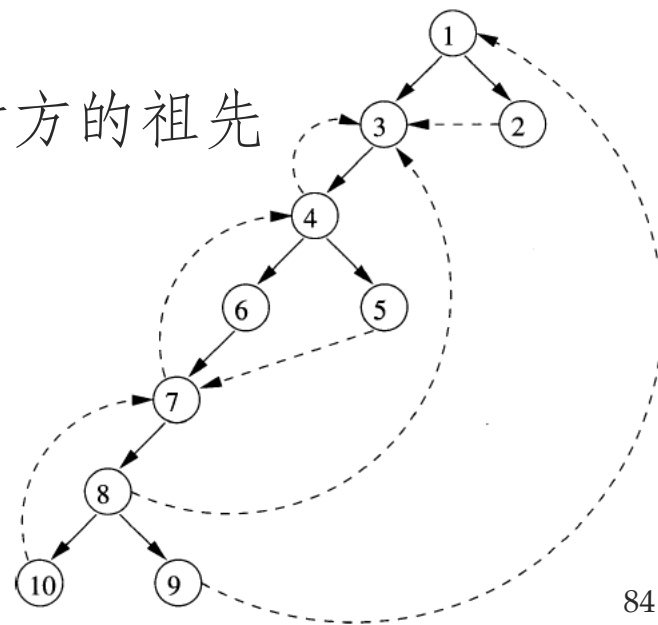
- $T$ 中记录了深度优先生成树的边集合
- $dfn[n]$ 表示 $n$ 的深度优先编号
- $c$ 的值从 $n$ 逐步递减到1

```
void search(n) {
    将  $n$  标记为“visited”;
    for ( $n$  的各个后继  $s$ )
        if ( $s$  标记为“unvisited”) {
            将边  $n \rightarrow s$  加入到  $T$  中;
            search( $s$ );
        }
     $dfn[n] = c$ ;
     $c = c - 1$ ;
}

main() {
     $T = \emptyset$ ; /* 边集 */
    for ( $G$  的各个结点  $n$ )
        把  $n$  标记为“unvisited”;
     $c = G$  的结点个数;
    search( $n_0$ );
}
```

# 流图中边的分类

- 为一个流图构造出DFST之后，流图中的边可以分为三类
  - **前进边**：从结点 $m$ 到达 $m$ 在DFST树中的一个**真后代**结点的边 (DFST中的所有边都是前进边)
  - **后退边**：从 $m$ 到达 $m$ 在DFST树中的某个**祖先** (包括 $m$ ) 的边
  - **交叉边**：边的 $src$ 和 $dest$ 都不是对方的祖先
  - 考虑： $3 \rightarrow 4$ ,  $10 \rightarrow 7$ ,  $5 \rightarrow 7$

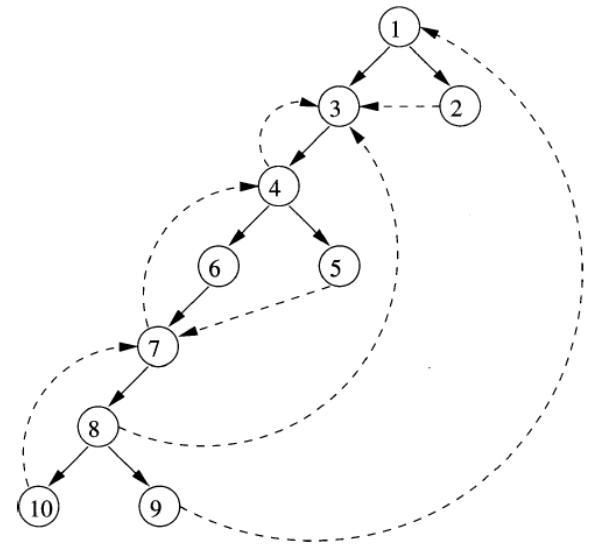
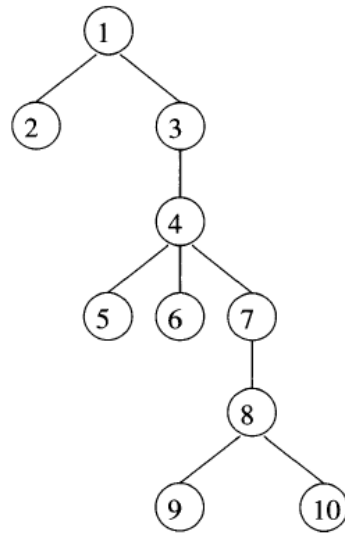
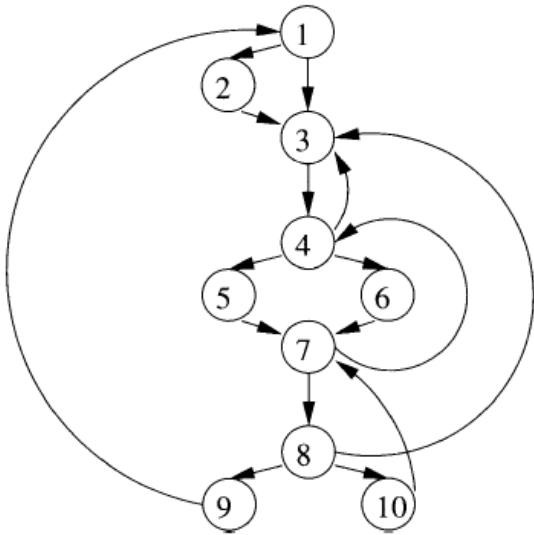


# 回边和可归约性 (1)

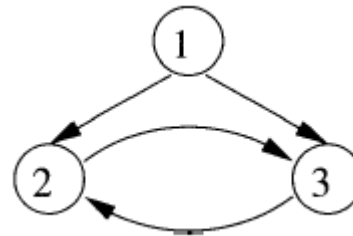
- 回边
  - 边  $a \rightarrow b$ , 头  $b$  支配了尾  $a$
  - 每条回边都是后退边, 但不是所有后退边都是回边
- 如果一个流图的任何深度优先生成树中的所有后退边都是回边, 那么该流图就是可归约的
  - 可归约流图的DFST的后退边集合就是回边集合
  - 不可归约流图的DFST中可能有一些后退边不是回边

# 回边和可归约性 (2)

- 现实中出现的流图基本都是可归约的
  - 可归约的例子



- 不可归约的例子



# 流图的深度

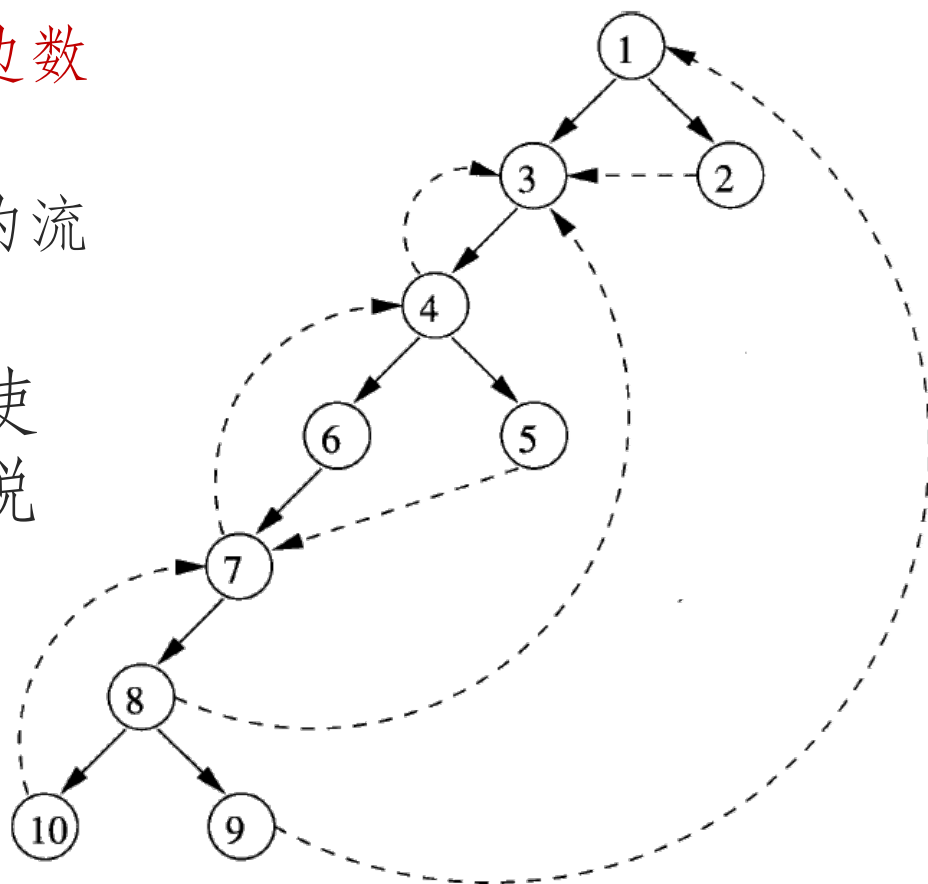
- 流图相对于DFST的深度

- 各条无环路径上后退边数中的最大值
- 不会大于直观上所说的流图中的循环嵌套深度

- 对可归约的流图，可使用回边来定义，且可说是流图的深度

- 右边的流图深度为3

- $10 \rightarrow 7 \rightarrow 4 \rightarrow 3$



# 自然循环

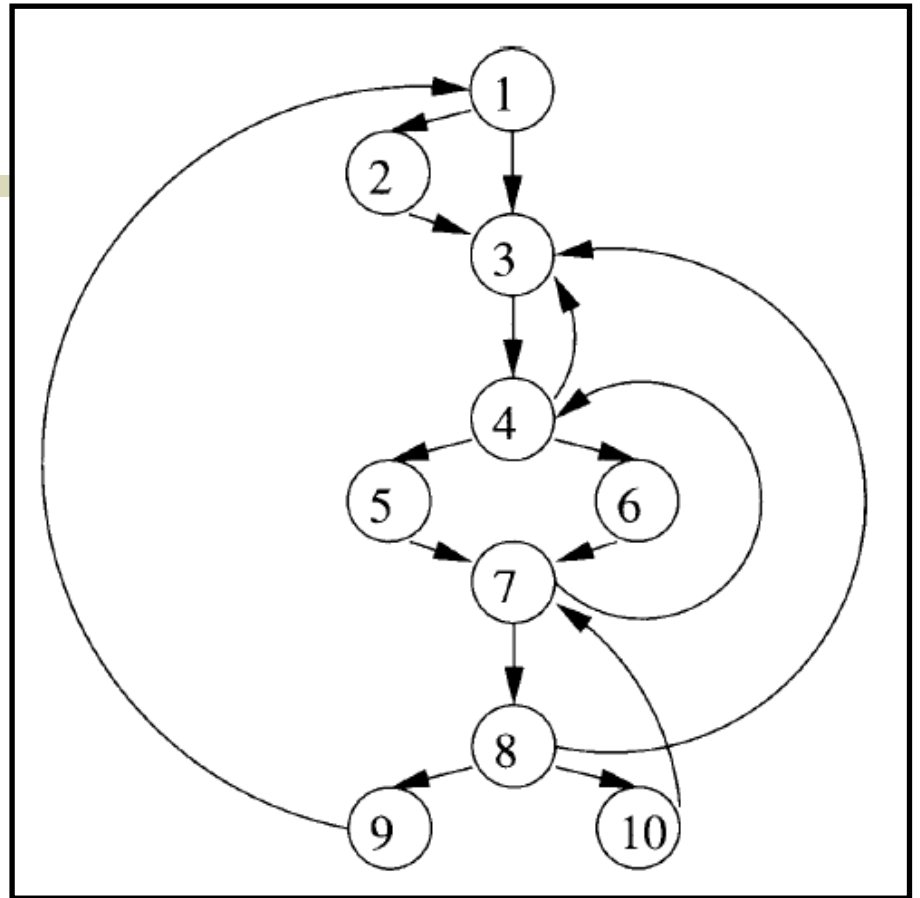
- 自然循环的性质
  - 有一个**唯一**的入口结点 (循环头Header), 这个结点**支配**循环中的所有结点
  - 必然存在进入循环头的**回边**
- 自然循环的定义
  - 给定回边  $n \rightarrow d$  的自然循环是  $d$ , 加上不经过  $d$  就能够到达  $n$  的结点的集合
  - $d$  是这个循环的头

# 自然循环构造算法

- 输入：流图 $G$ 和回边 $n \rightarrow d$
- 输出：给定回边 $n \rightarrow d$ 的自然循环中的所有结点的集合 $loop$
- 方法
  - $loop = \{ n, d \}$ ， $d$ 标记为visited
  - 从 $n$ 开始，**逆向**对流图进行深度优先搜索，把所有访问到的结点都加入 $loop$ ，加入 $loop$ 的结点都标记为visited
  - 搜索过程中，**不越过**标记为visited的结点

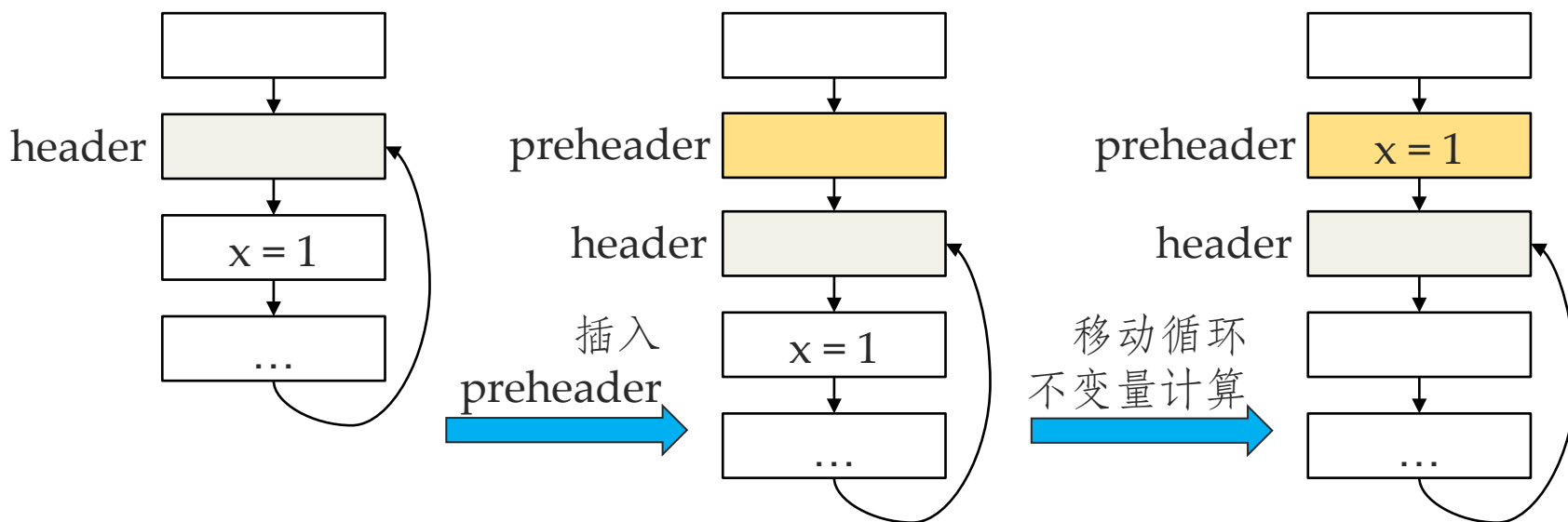
# 自然循环的例子

- 回边： $10 \rightarrow 7$ 
  - $\{7, 8, 10\}$
- 回边： $7 \rightarrow 4$ 
  - $\{4, 5, 6, 7, 8, 10\}$
  - 包含了前面的循环
- 回边  $4 \rightarrow 3$  ( $8 \rightarrow 3$ )
  - 同样的头
  - 同样的结点集合  $\{3, 4, 5, 6, 7, 8, 10\}$
- 回边  $9 \rightarrow 1$ 
  - 整个流图



# 循环头上头 (Preheader)

- 每个自然循环都有一个循环头header
- 一些循环相关的优化 (如循环不变量移动) 需要在循环头之前插入代码
- 常见做法是在循环头之前插入一个基本块 (称为preheader), 用于存放相关的代码



# 总结

- 优化的来源
  - 冗余计算
- 基本优化技术
  - 公共子表达式消除、复制传播、死代码消除、常量折叠
- 数据流分析
  - 控制流图
  - 到达定值分析、活跃变量分析、可用表达式分析
- 部分冗余消除
  - 预期执行表达式、懒惰代码移动
- 循环的识别
  - 支配节点、深度优先生成树(回边)、自然循环