第九章 机器无关的优化

陈 林

主要内容

- 优化的来源
  - 全局公共子表达式
  - 复制传播
  - 死代码消除
  - 代码移动
  - 归纳变量和强度消减
- 数据流分析
  - 到达定值分析
  - 活跃变量分析
  - 可用表达式分析
- 循环的优化

引言

- 代码优化
  - 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
- 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
    - 比如A[i][j].f = 0; A[i][j].k = 1;中的冗余运算
- 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);
}
```

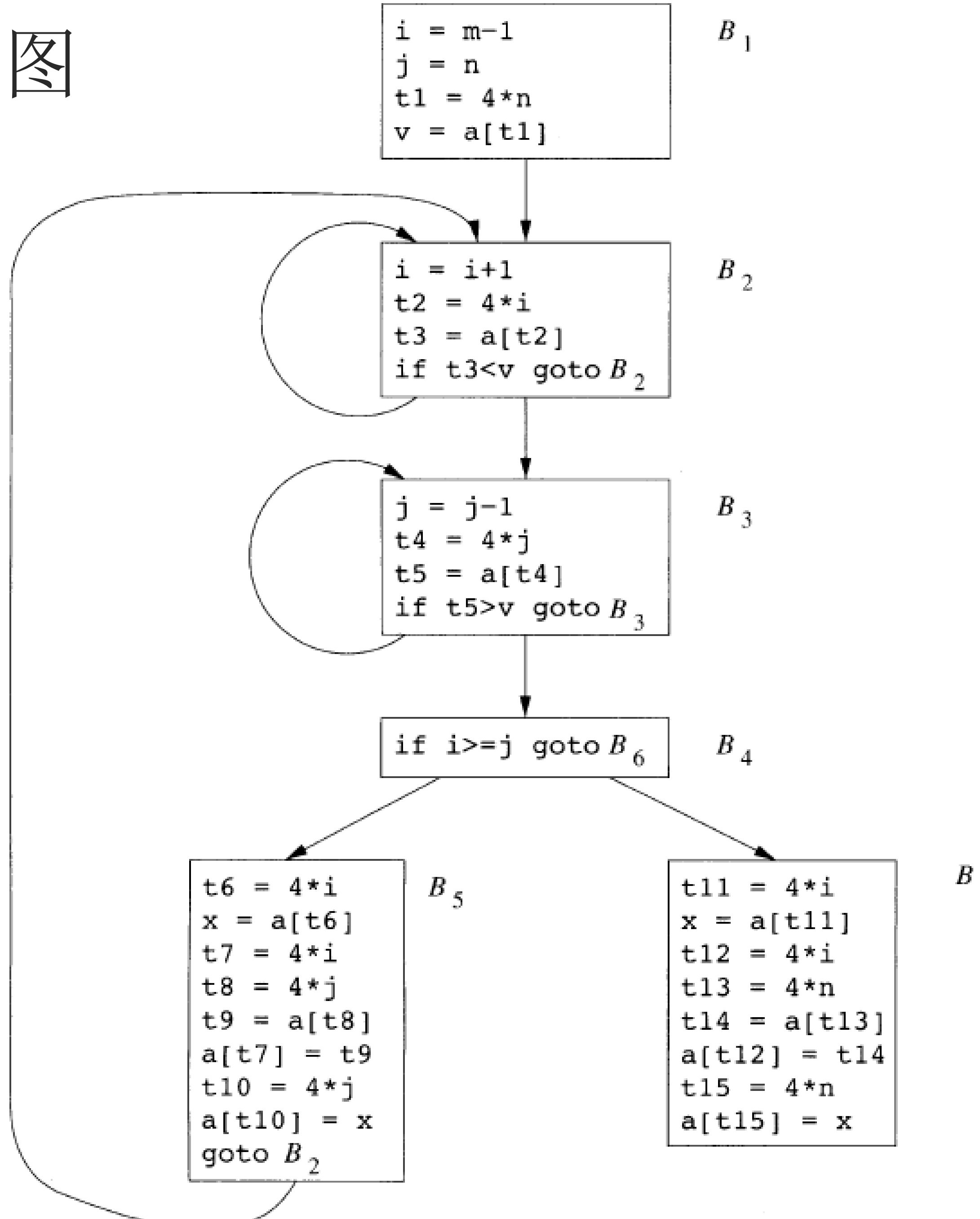
优化的例子（2）

- 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>=j goto (23)      (28)  a[t12] = t14
(14)  t6 = 4*i               (29)  t15 = 4*n
(15)  x = a[t6]              (30)  a[t15] = x
```

Quicksort的流图

- 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经被计算过，且
  - E的分量在该次计算之后一直没有被改变，
- 那么E的本次出现就是一个公共子表达式
- 如果上一次E的值赋给了x，且x的值至今没有被修改过，那么我们就可以使用x，而不需要计算E

B_5

```
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
```

a) 消除之前

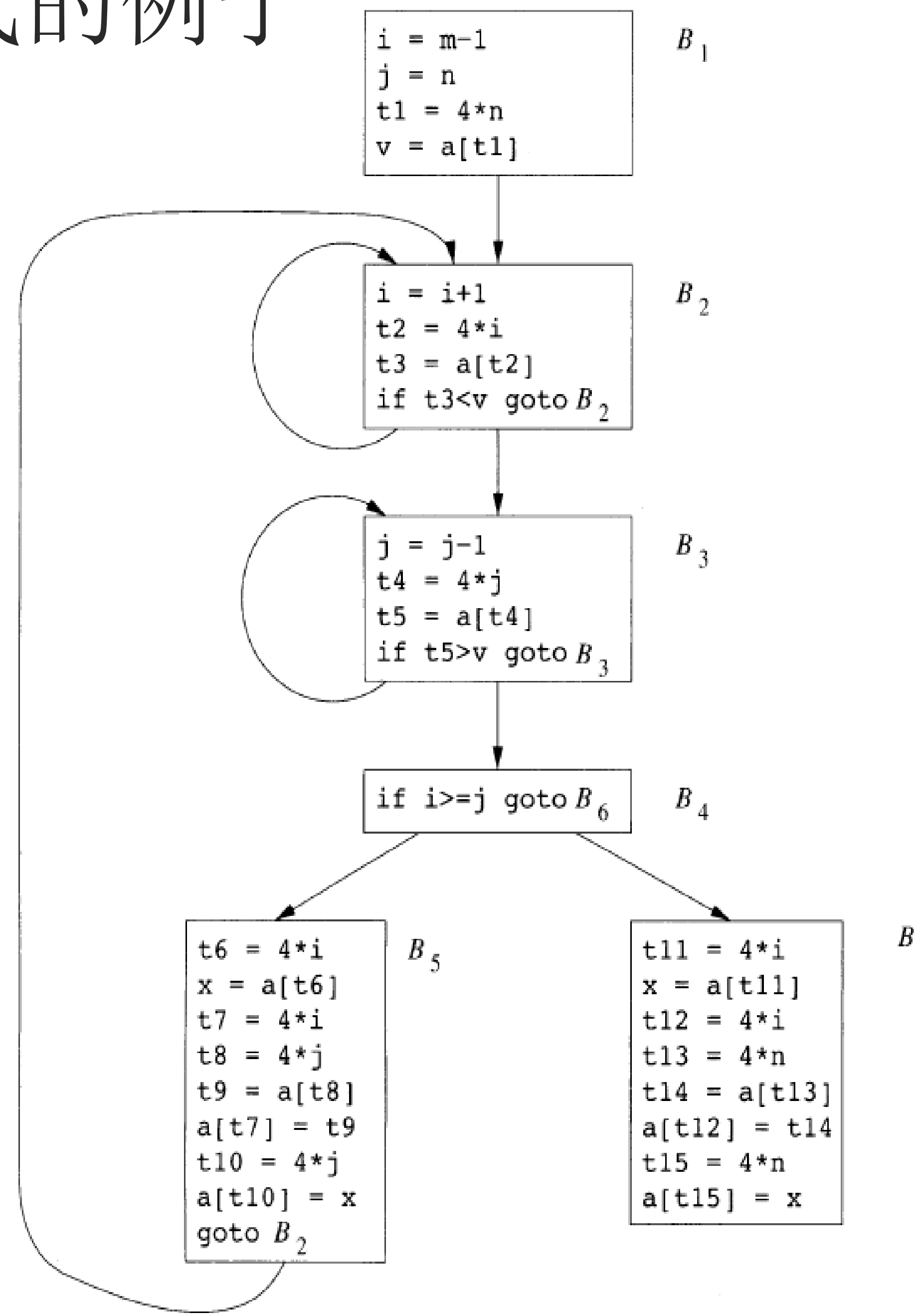
B_5

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

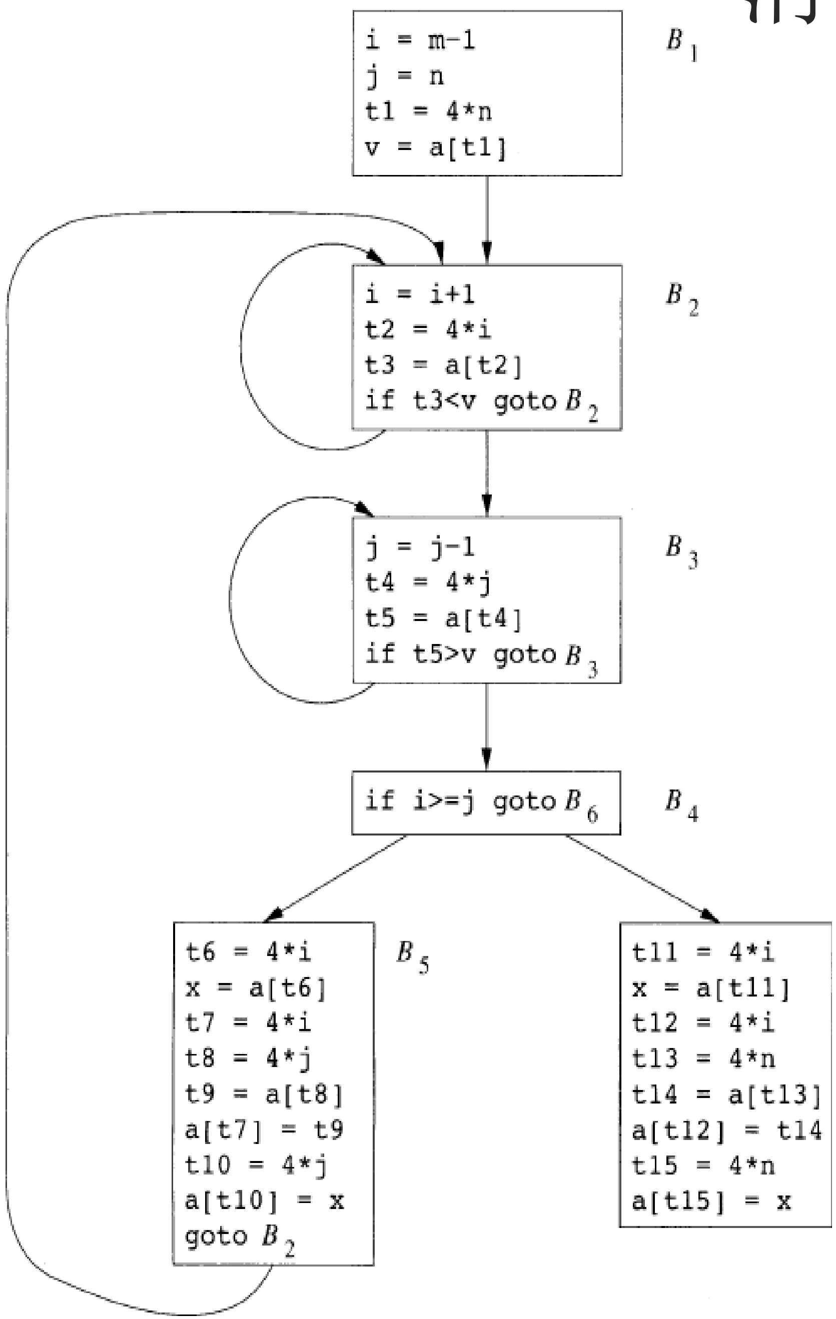
b) 消除之后

全局公共子表达式的例子

- 右图
  - 在B_2、B_3中计算了4*i和4*j
  - 到达B_5之前必然经过B_2、B_3
  - t2、t4在赋值之后没有被改变过，因此B_5中可直接使用它们
  - t4在替换t8之后，B_5：a[t8]和B_3：a[t4]又相同
- 同样：
  - B_5中赋给x的值和B_2中赋给t3的值相同
  - B_6中的a[t13]和B_1中的a[t1]不同，因为B_5中可能改变a的值



消除公共子表达式后



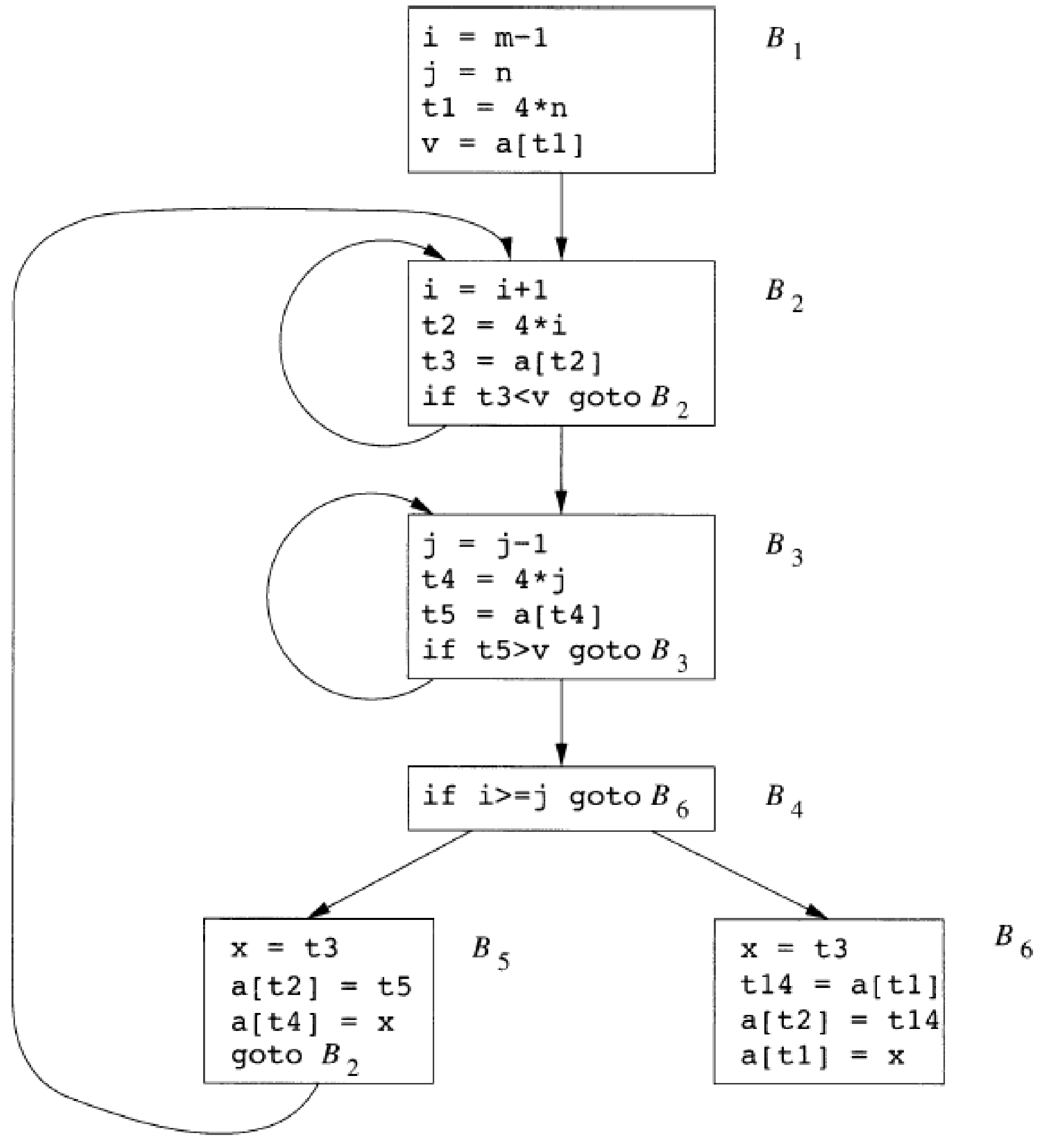
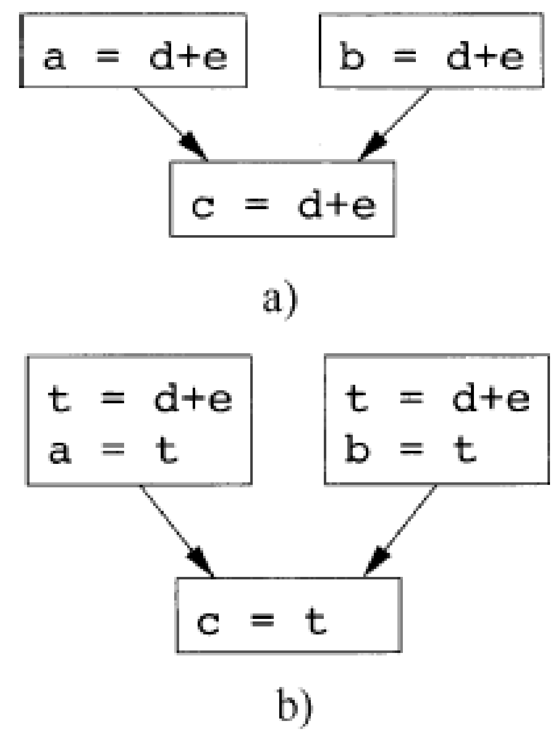


图 9-5　经过公共子表达式消除之后的 B_5 和 B_6

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

B_5

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

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；否则这个变量就是死的，此时对这个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

x=t3
a[t2] = t5
a[t4] = t3
goto B_2

$\Rightarrow$

a[t2] = t5
a[t4] = t3
goto B_2

代码移动

- 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，可以在循环外计算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的赋值都使得x的值增加c，那么x就是归纳变量
  - 把对x的赋值改成增量操作，可消减计算强度，提高效率
  - 如果两个归纳变量步调一致，还可以删除其中的某一个
- 例子
  - 如果在循环开始时刻保持t4=4*j
  - 那么，j=j-1后面的t4=4*j每次赋值使得t4减4
  - 替换为t4 = t4 - 4
  - t2也可以同样处理

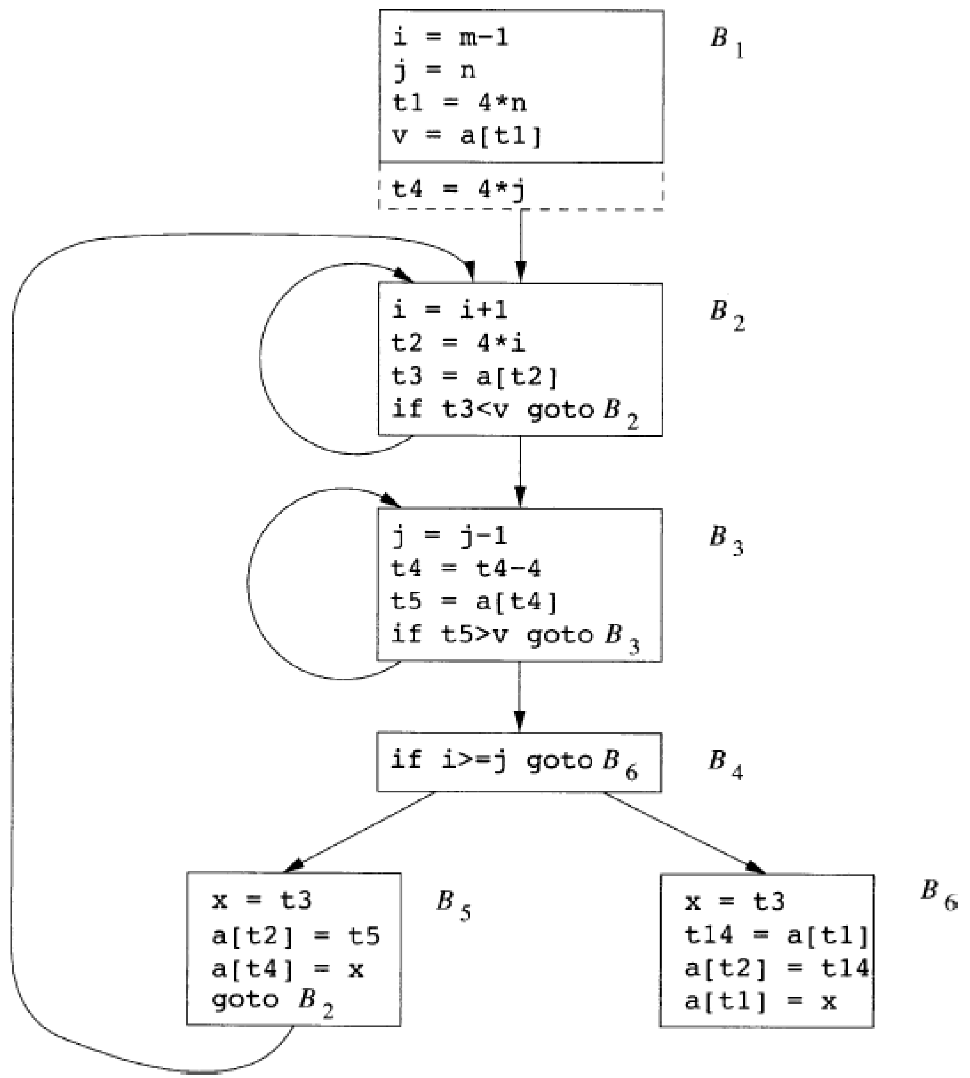


图 9-8　对基本块 B_3 中的 $4*j$ 应用强度消减优化

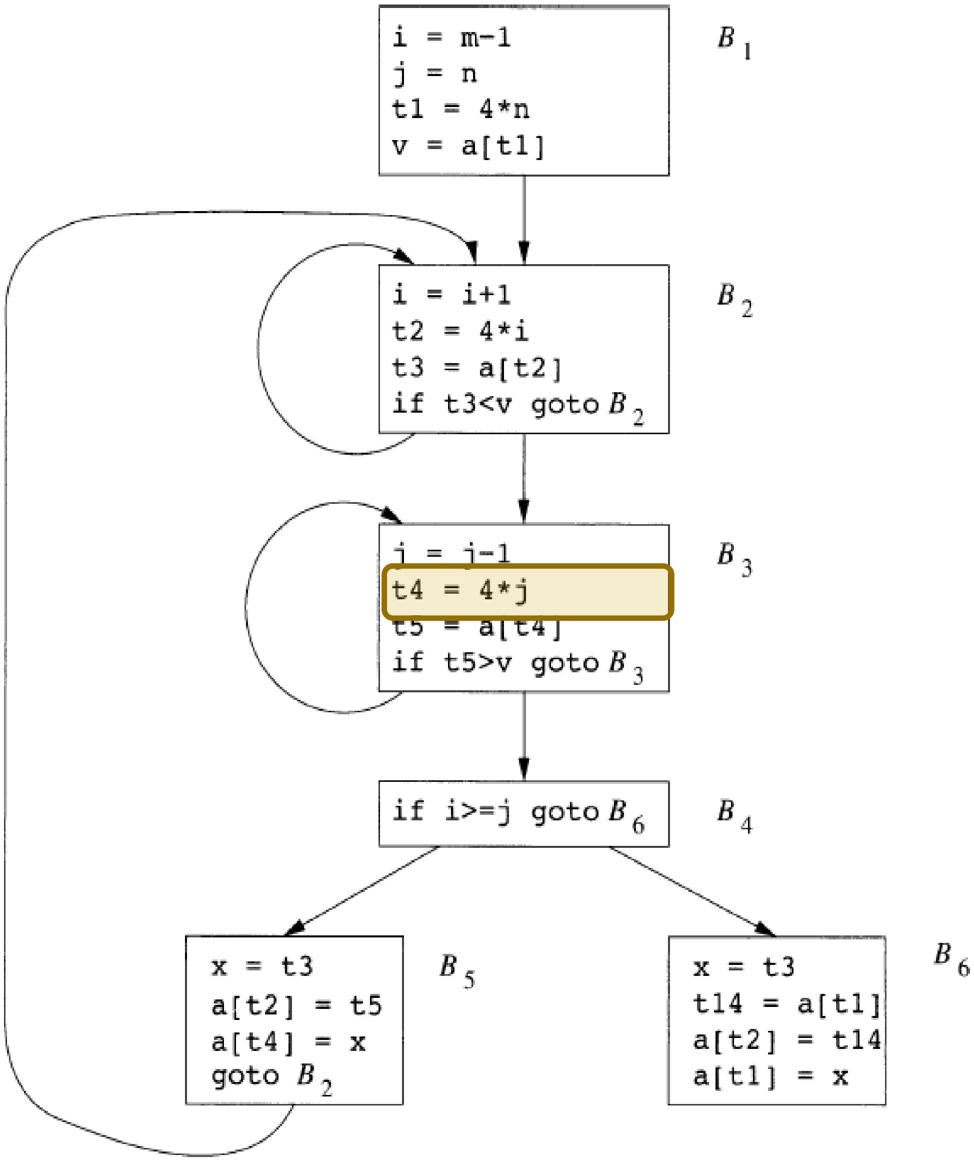


图 9-5　经过公共子表达式消除之后的 B_5 和 B_6

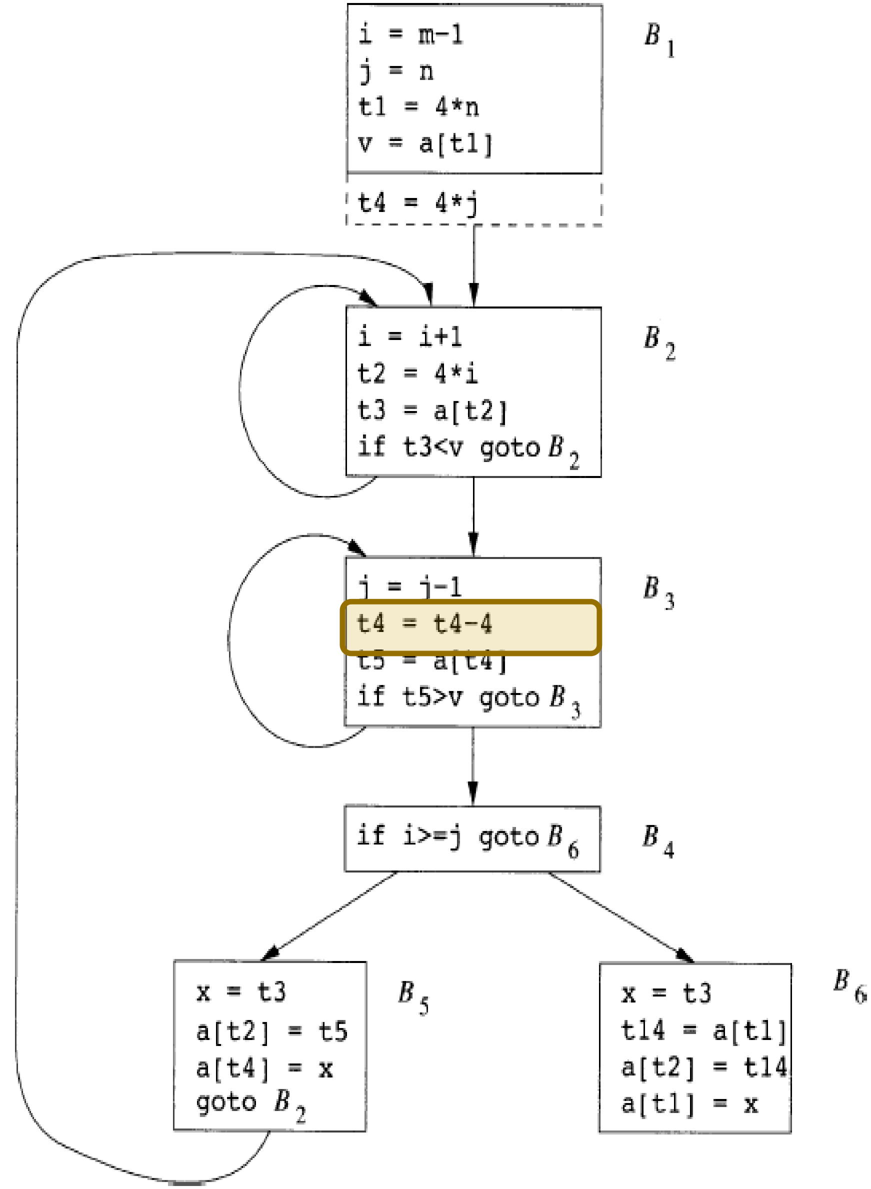


图 9-8　对基本块 B_3 中的 $4*j$ 应用强度消减优化

继续优化

- 对t2强度消减
- B_4中对i和j的测试可以替换为对t2, t4的测试

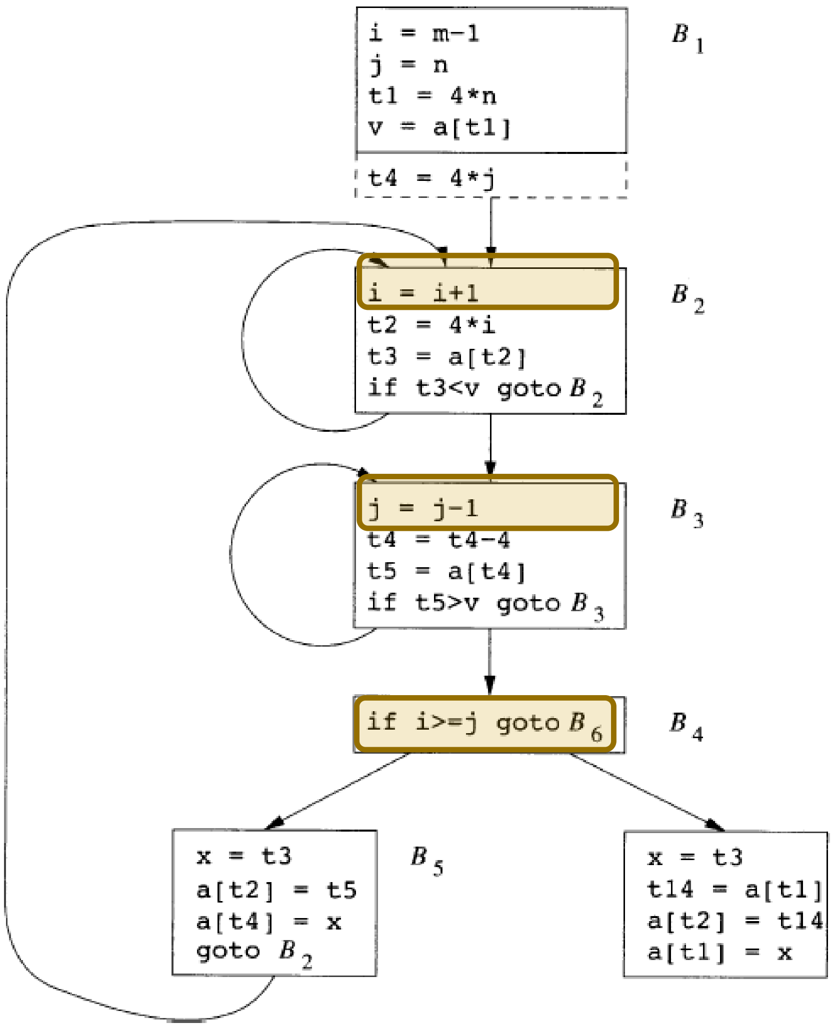
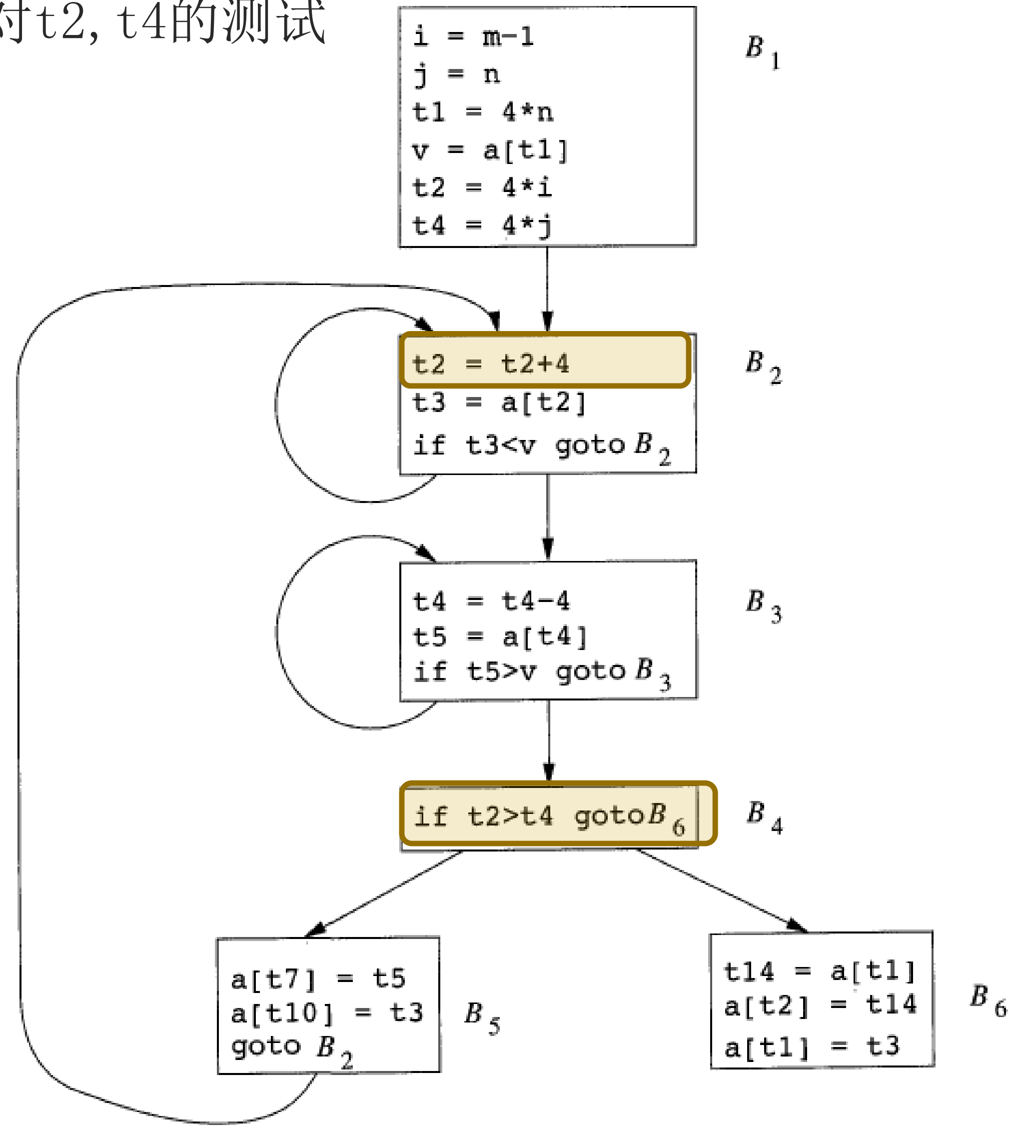


图 9-8　对基本块 B_3 中的 $4*j$ 应用强度消减优化



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的信息的相关技术
  - 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？（公共子表达式）
  - 一个语句的计算结果有没有可能被后续语句使用？（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_2的执行路径：p_1，p_2，…，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
- 为了高效、自动地进行数据流分析，通常要求这些特性能够被高效地表示和求解

例子（1）

- 路径
  - 1, 2, 3, 4, 9
  - 1, 2, 3, 4, 5, 6, 7, 8, 9
  - …
- 第一次到达(5)， a=1；第二次到达(5)，a=243；且之后都是243
- 我们可以说：
  - 点(5)具有特性a=1 or a=243
  - 表示成为<a, {1, 243}>

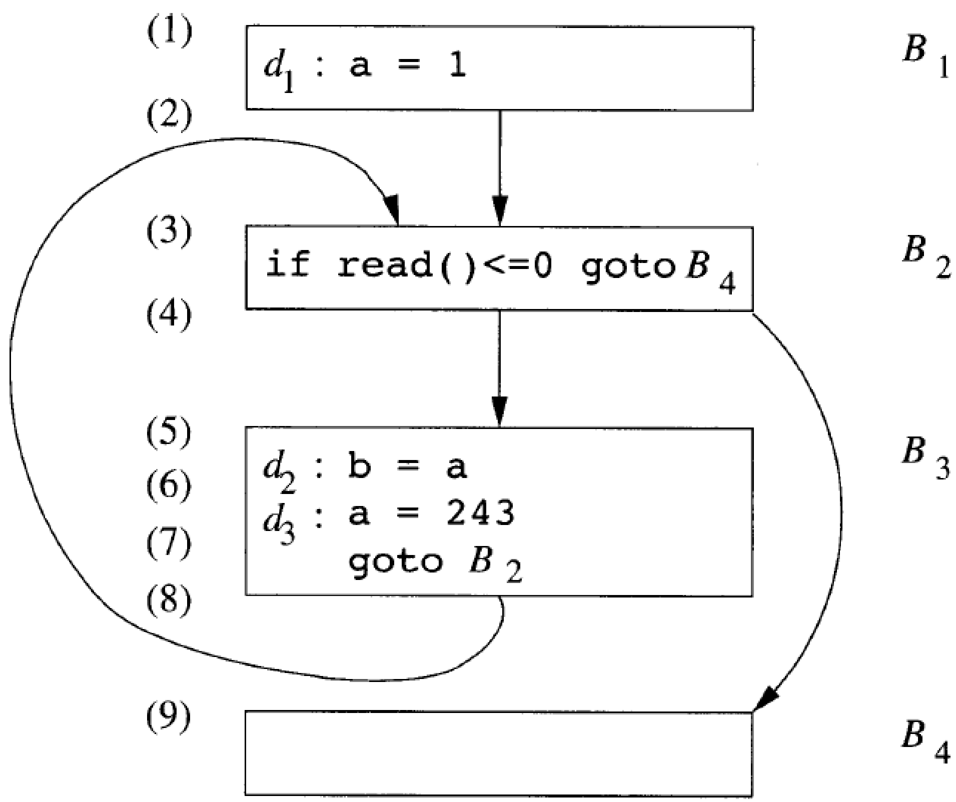


图 9-12　说明数据流抽象的例子程序

性质和算法

- 不同的需求对应不同的的性质集合与算法

有可能做激
进的优化吗？

- 分析得到的性质集合应该是一个安全的估计值
    - 即根据这些性质进行优化不会改变程序的语义

数据流分析模式

- 数据流值
  - 某个程序点所有可能的状态集合的抽象表示
  - 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这个条件
- 域
  - 所有可能的数据流值的集合
- 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
  - 因此数据流值是定值（即三地址语句）的集合，表明集合中的定值对某个变量定值了

数据流分析

- 数据流分析：对一组约束求解
  - IN[s]和OUT[s]
- 基于语句语义的约束(传递函数)
  - $IN[s]=f_s(OUT[s])$　　（逆向）
  - $OUT[s]=f_s(IN[s])$　　（正向）
- 基于控制流的约束
  - $IN[s_{i+1}]=OUT[s_i]$

例子（1）

- 假设我们考虑各个变量在某个程序点上是否常量

IN[s]：x:NAC;　　y:7;　　z:3

s: x=3

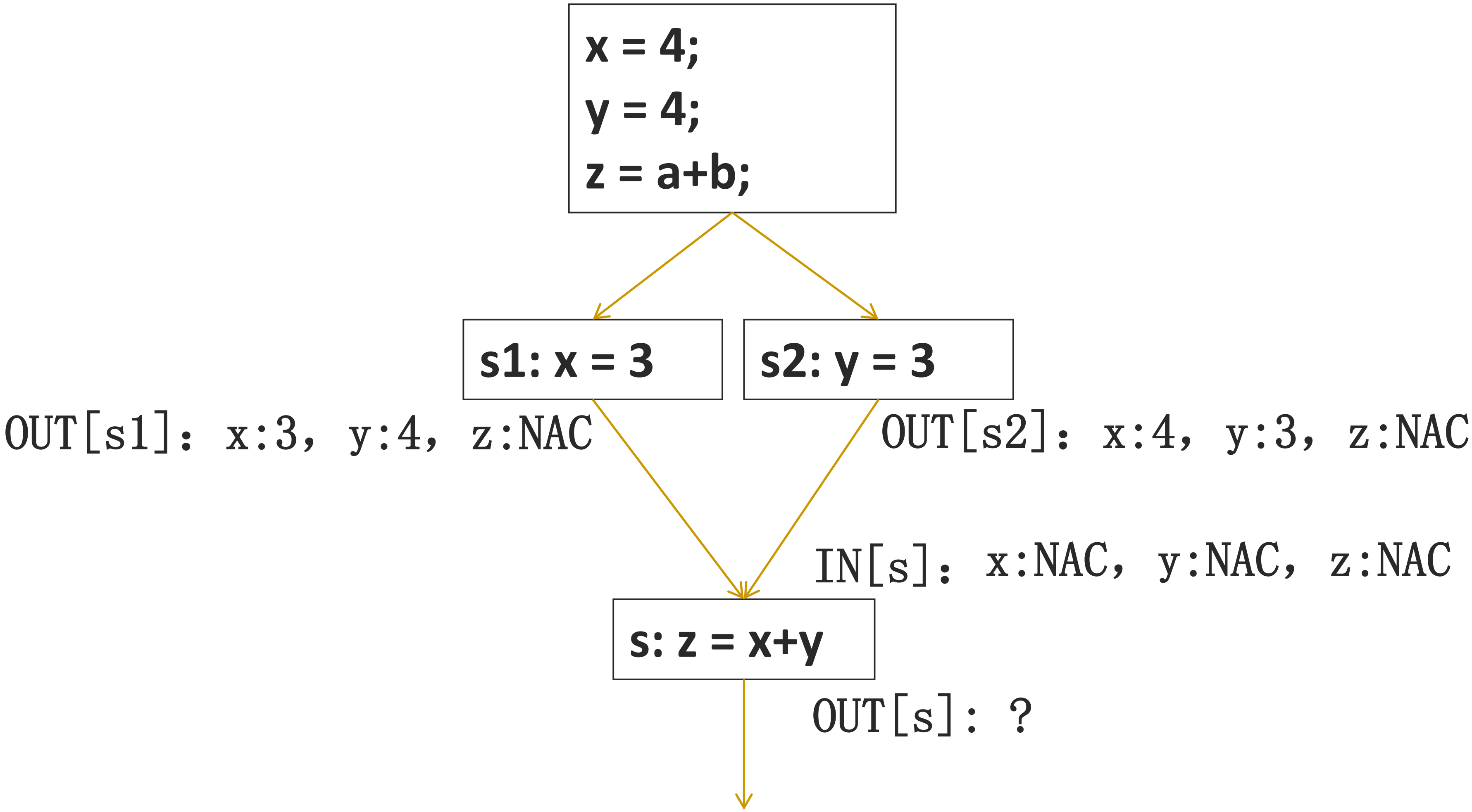
OUT[s]：x:3;　　y:7;　　z:3

IN[s]：x:NAC;　　y:7;　　z:3

s: x=y+z

OUT[s]：x:10;　　y:7;　　z:3

例子（2）


x = 4;
y = 4;
z = a+b;
s1: x = 3
s2: y = 3
OUT[s1]： x:3， y:4， z:NAC
OUT[s2]： x:4， y:3， z:NAC
IN[s]： x:NAC， y:NAC， z:NAC
s: z = x+y
OUT[s]： ?

基本块上的数据流模式

- 基本块的控制流非常简单
  - 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，给出基本块对应的传递函数；

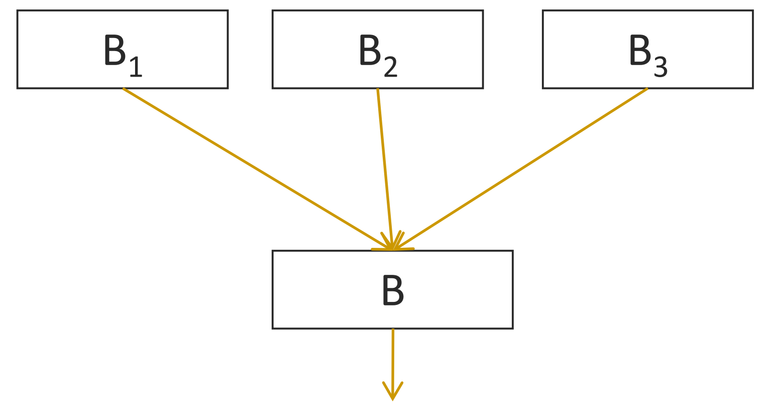
  $IN[B] = f_B(OUT[B])$ 或者

  $OUT[B] = f_B(IN[B])$
- 设基本块包含语句s1, s2, …, sn

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

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：

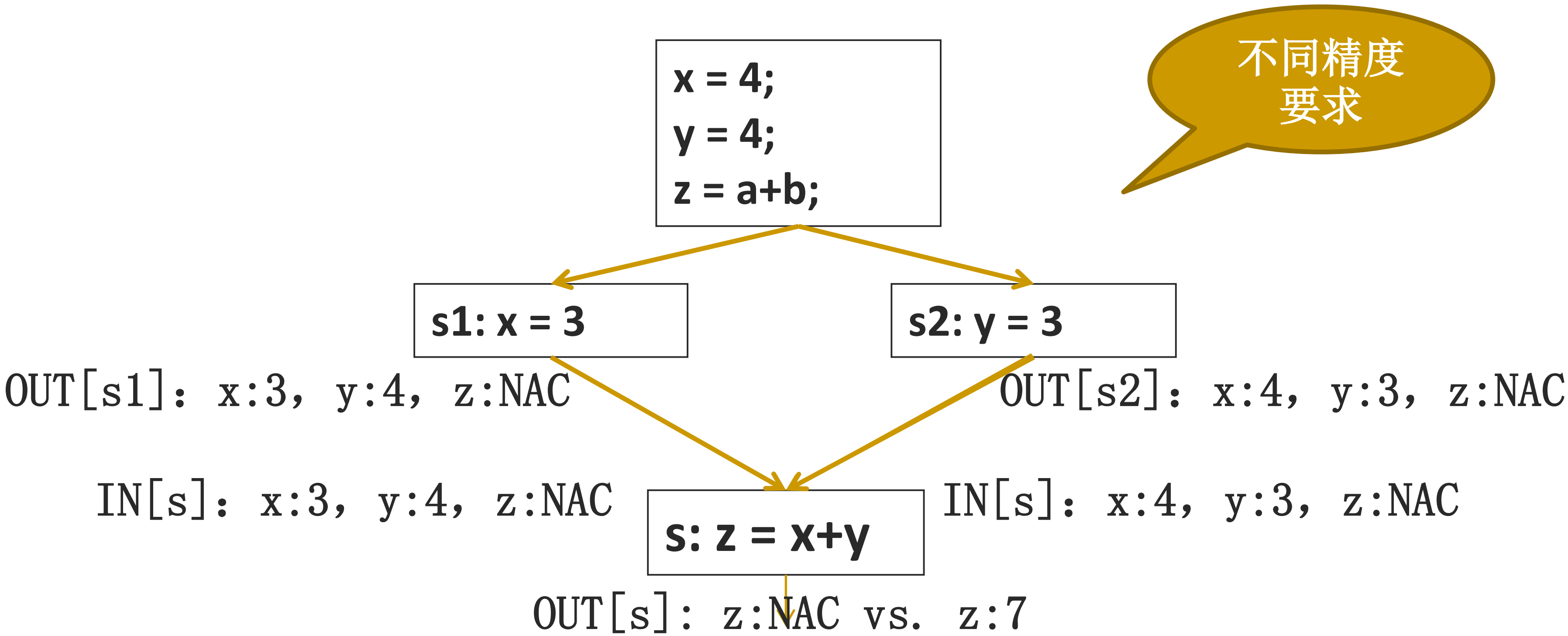
OUT[B1]：	**x:3**	**y:4**	**z:NAC**
OUT[B2]：	**x:3**	**y:5**	**z:7**
OUT[B3]：	**x:3**	**y:4**	**z:7**

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“精确的”、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

数据流分析

- 到达定值分析
- 活跃变量分析
- 可用表达式分析

到达定值

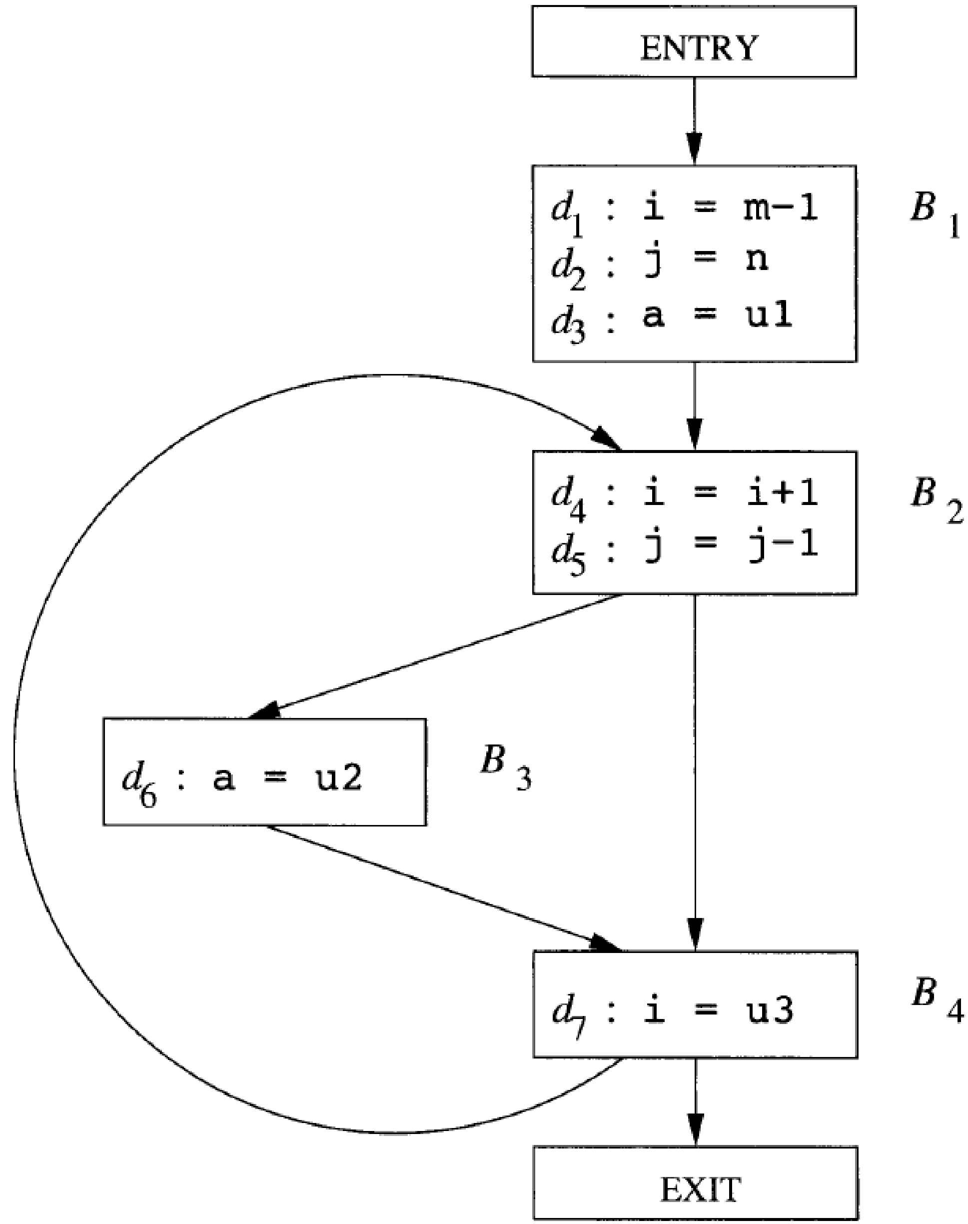
- 到达定值
  - 如果存在一条从定值d后面的程序点到达某个点p的路径，且这条路径上d没有被杀死，那么定值d到达p
- 杀死：路径上对x的其他定值杀死了之前对x的定值

- 直观含义
  - 如果d到达p，那么在p点使用的值就可能是由d定值的

- 思考：不确定是否赋值该怎么办？
  - *p = 3 （不确定p的指向）
  - 过程参数、数组、指针等等

到达定值的例子

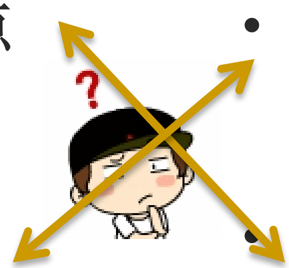
- B_1中全部定值到达B_2的开头
- d_5到达B_2的开头（循环）
- d_2被d_5杀死，不能到达B_3、B_4的开头
- d_4不能到达B_2的开头，因为被d_7杀死
- d_6到达B_2的开头



到达定值

- 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

- 对于可能定值的情况，怎么做才是安全的？
  - 确定变量x在某个程序点是否常量
  - 假设所有可能定值都不能到达



  - 确定变量是否先使用后定值（未初始化就使用）
  - 假设所有可能定值都能到达

语句/基本块的传递方程（1）

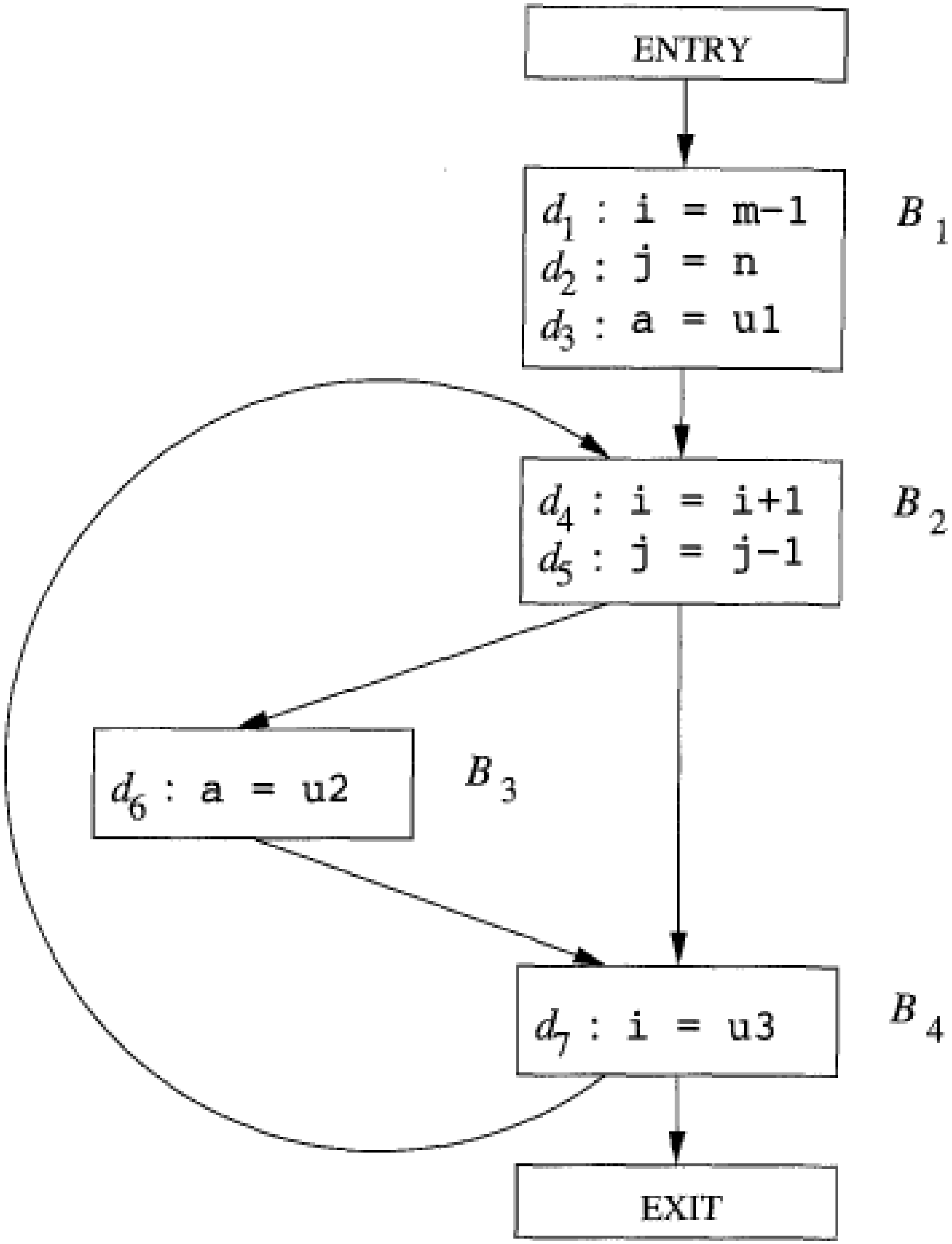
- 定值　d: u=v+w
  - 生成了对变量u的定值d，杀死了其他对u的定值
  - 生成-杀死：$f_d(s)=gen_d \cup (x-kill_d)$
    - 其中$gen_d=\{d\}$，$kill_d=\{$程序中其他对u的定值$\}$
- 生成-杀死形式的函数的并置（复合）
  - $f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x-kill_1) - kill_2)$
    $= (gen_2 \cup (gen_1-kill_2)) \cup (x-kill_1 \cup kill_2))$
  - 生成的定值：由第二部分生成、以及由第一个部分生成且没有被第二部分杀死
  - 杀死的定值：被第一部分杀死的定值、以及被第二部分杀死的定值

语句/基本块的传递方程（2）

- 设B有n个语句，第i个语句的传递函数为f_i
- $f_B(s)=gen_B \cup (x-kill_B)$
- $gen_B = gen_n \cup (gen_{n-1}-kill_n) \cup (gen_{n-2}-kill_{n-1}-kill_n)$
  $\cup (gen_1-kill_2-kill_3 \ldots -kill_n)$
- $kill_B= kill_1 \cup kill_2 \cup \ldots \cup kill_n$

- $kill_B$ 为被B各个语句杀死的定值的并集
- 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

gen和kill的例子



$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

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
- $IN[B] = \cup_{P是B的前驱基本块} OUT[P]$
  - 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
- ∪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

$$OUT[ENTRY] = 空集$$

- 其他基本块

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

$$IN[B] = \cup_{P是B的前驱基本块} OUT[P]$$

- 迭代解法
  - 首先求出各基本块的gen和kill
  - 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：

$\mathrm{OUT}[\mathrm{ENTRY}] = \emptyset;$

for（除 ENTRY 之外的每个基本块 B） $\mathrm{OUT}[B] = \emptyset;$

while（某个 OUT 值发生了改变）

  for（除 ENTRY 之外的每个基本块 B）{

    $\mathrm{IN}[B] = \bigcup_{P\text{是}B\text{的一个前驱}} \mathrm{OUT}[P];$

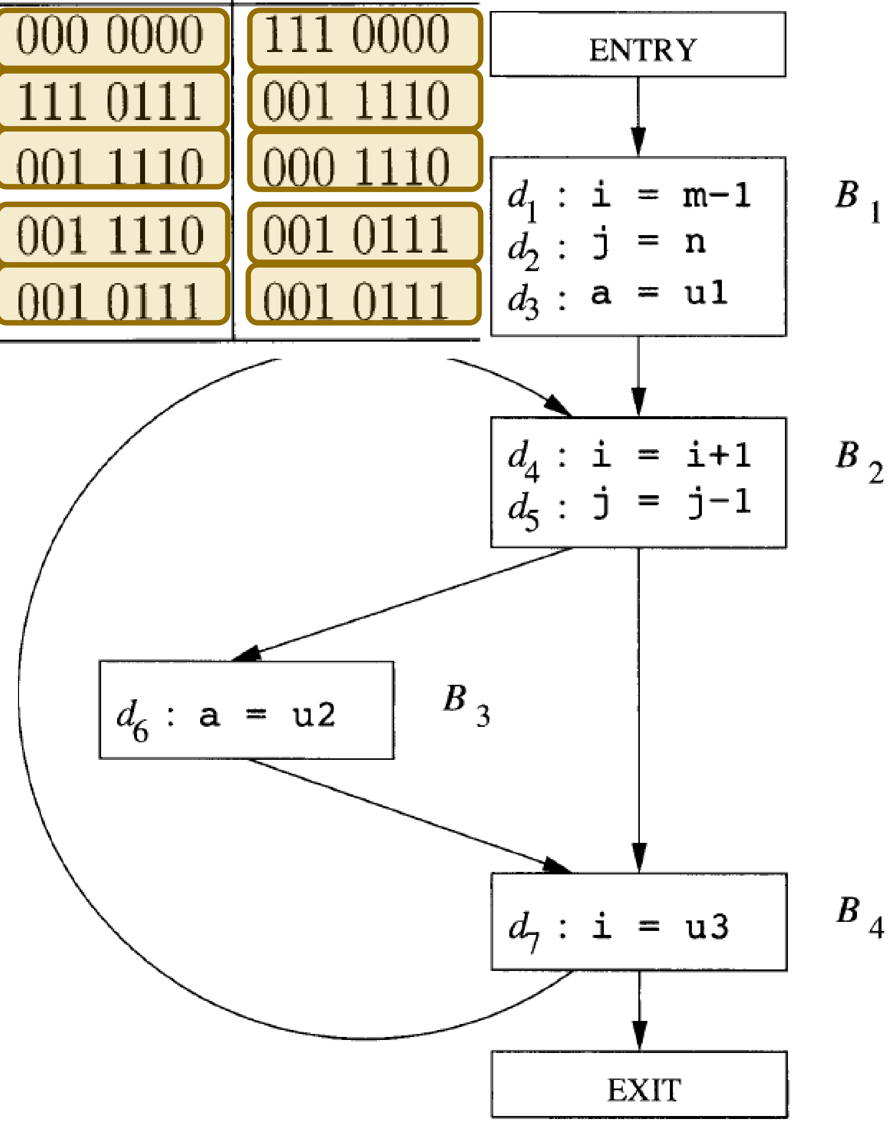
    $\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B);$

  }

到达定值求解的例子

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

- 7个bit从左到右表示$d_1,d_2,\ldots,d_n$
- for循环时依次遍历 B_1,B_2,B_3,B_4,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



控制流方程的迭代解法（3）

- 解法的正确性
  - 直观解释：不断向前传递各个定值，直到该定值被杀死为止
- 算法为什么能终止？
  - 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
- 最大迭代次数是流图的结点个数n
  - 定值经过n步必然已经到达所有可能到达的结点
- 算法结束时，各个OUT/IN值必然满足数据流方程

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
  - 一个变量x在p上活跃，当且仅当存在一条从p点开始的路径，该路径的末端使用了x，且路径上没有对x进行覆盖
- 用途
  - 寄存器分配/死代码删除/…
- 数据流值
  - (活跃)变量的集合

基本块内的数据流方程

- 基本块的传递函数仍然是生成-杀死形式，但是从OUT值计算出IN值（逆向）
  - use_B，可能在B中先于定值被使用（GEN）
  - def_B，在B中一定先于使用被定值（KILL）

- 例子：
  - 基本块

    i=i+1

    j=j-1

  - use {i,j}
  - def { }

USEB和DEFB的用法

- 语句的传递函数
  - s：x=y+z
  - $use_s = \{y, z\}$
  - $def_s=\{x\}-\{y, z\}$ //x=y+z是模板，y、z可能和x相同
- 假设基本块中包含语句s1, s2, …, sn，那么
- $use_B = use_1 \cup (use_2-def_1) \cup (use_3-def_1-def_2)$
  $\cup (use_n-def_1-def_2 \cdots -def_{n-1})$
- $def_B=def_1 \cup def_2 \cup \cdots \cup def_n$

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] = 空集
- 对于所有不等于EXIT的基本块
  - $IN[B] = use_B \cup (OUT[B] - def_B)$
  - $OUT[B] = \cup_{S是B的后继基本块} IN[S]$
- 和到达定值相比较
  - 都使用并集运算∪作为交汇运算
  - 数据流值的传递方向相反：因此初始化的值不一样

活跃变量分析的迭代方法

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

- 这个算法中IN[B]总是越来越大，且IN[B]都有上界，因此必然会终止

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或者y赋值
- 主要用途
  - 寻找全局公共子表达式
- 生成-杀死
  - 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
  - 生成：基本块求值x+y，且之后没有对x或者y赋值，那么它生成了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定值，且没有被再次生成

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]={ }
- 其他基本块的方程
  - $OUT[B] = e\_gen_B \cup (IN[B]-e\_kill_B)$
  - $IN[B] = \cap_{P是B的前驱基本块} OUT[P]$
- 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
  - 这样的初始集合可以求得更有用的解

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

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	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
交汇运算(∧)	$\cup$	$\cup$	$\cap$
方程组	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
初始值	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

主要内容

- 优化的来源
  - 全局公共子表达式
  - 复制传播
  - 死代码消除
  - 代码移动
  - 归纳变量和强度消减
- 数据流分析
  - 到达定值分析
  - 活跃变量分析
  - 可用表达式分析
  - 常量传播
- 循环的优化

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？
  - 迭代算法是否收敛？
  - 方程组的解的含义是什么？
  - 得到的解有多精确？

  - 正确性问题
  - 精度问题

数据流框架的通用算法

- 前向

1) $\text{OUT}[\text{ENTRY}] = v_{\text{ENTRY}}$;
2) **for** (除 ENTRY 之外的每个基本块 B) $\text{OUT}[B] = \top$;
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) $\text{IN}[B] = \bigwedge_{P\text{是}B\text{的一个前驱}} \text{OUT}[P]$;
6) $\text{OUT}[B] = f_B(\text{IN}[B])$;
}

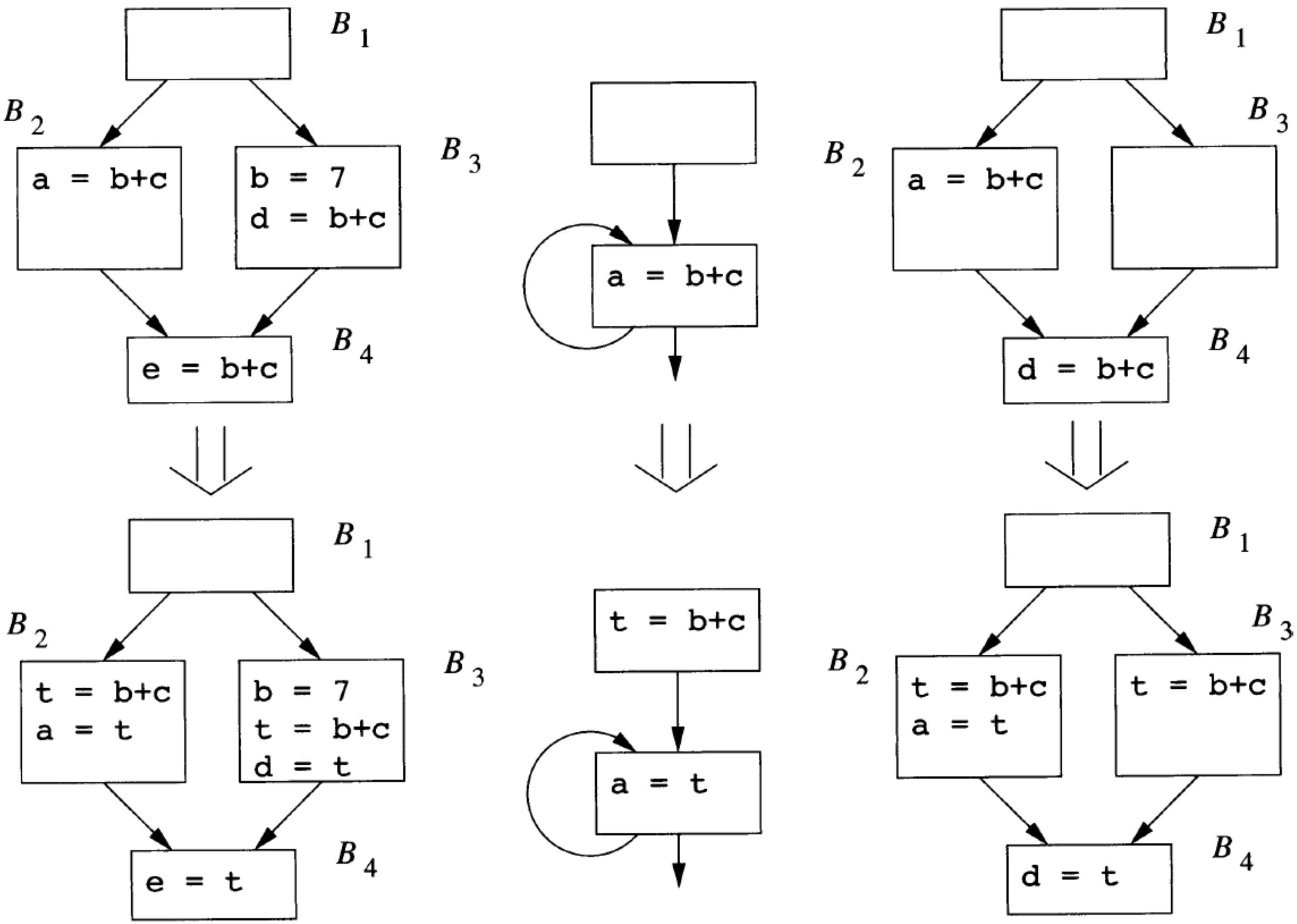
- 逆向

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

部分冗余消除

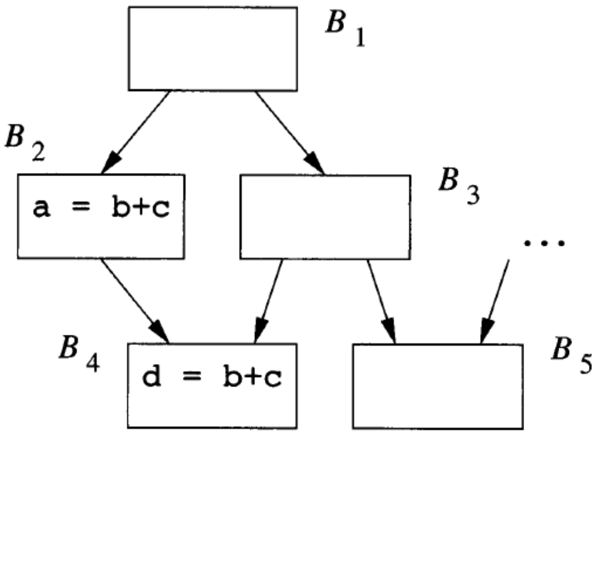
- 目标：尽量减少表达式求值的次数
- 对于表达式x+y
  - 公共表达式：如果对x+y求值前的程序点上x+y可用，那么我们不需要再对x+y求值；
  - 循环不变表达式：循环中的表达式x+y的值不变，可以只计算一次
  - 部分冗余：在程序按照某些路径到达这个点的时候x+y已经被计算过，但是沿着另外一些路径到达时，x+y尚未计算过
  - 处理方法：移动对x+y求值的位置
- 需要使用四个数据流方程来达到优化的目的

冗余的例子

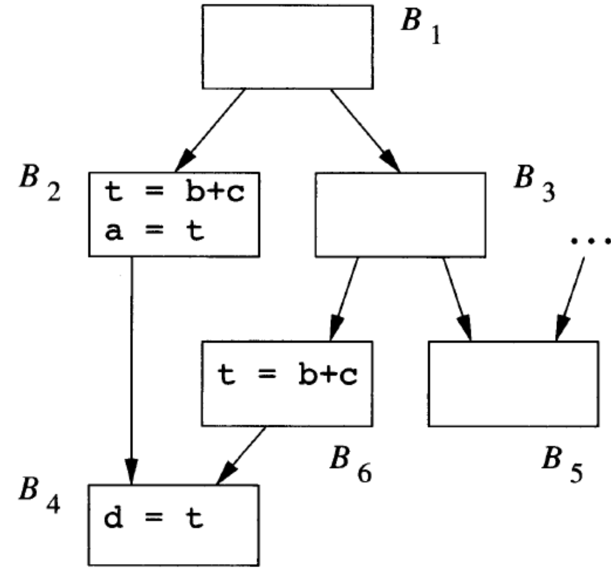


a) 公共子表达式　b) 循环不变代码移动　c) 部分冗余消除

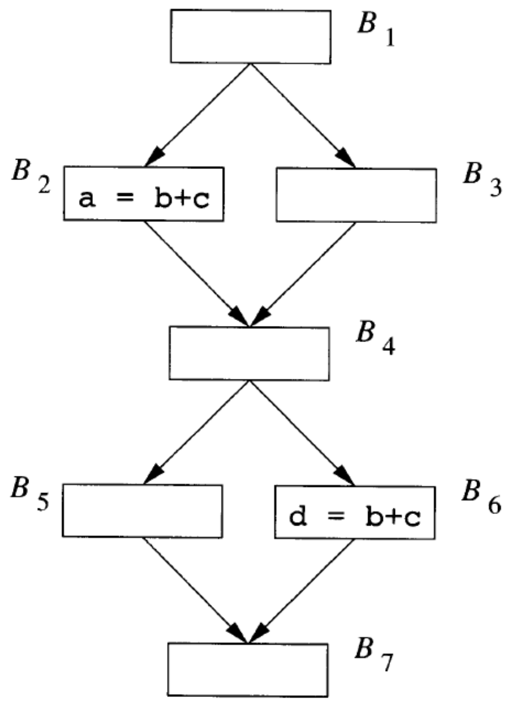
- 允许进行两种操作
  - 在关键边上增加基本块；
  - 进行代码复制
- 关键边：
  - 从具有多个后继的结点到达具有多个前驱的结点



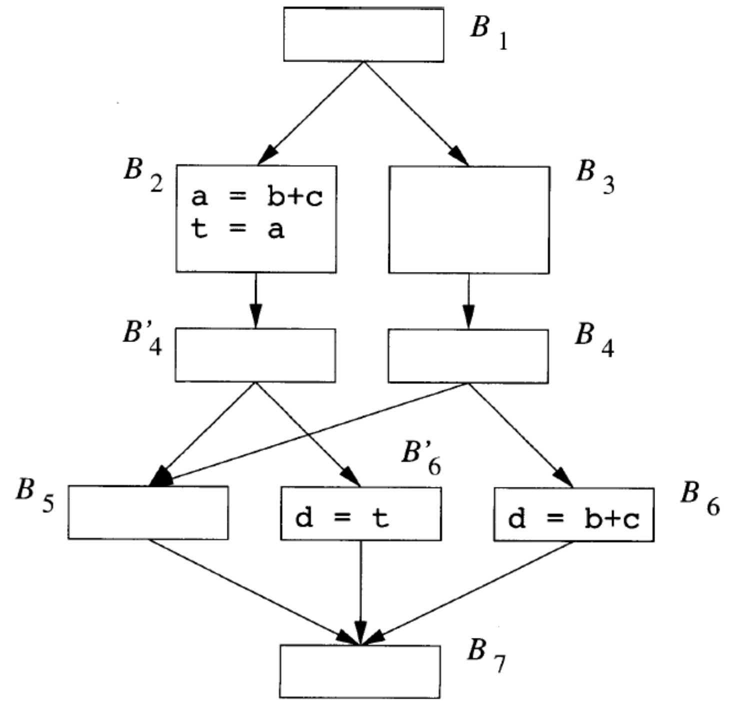
a)



b)



a)



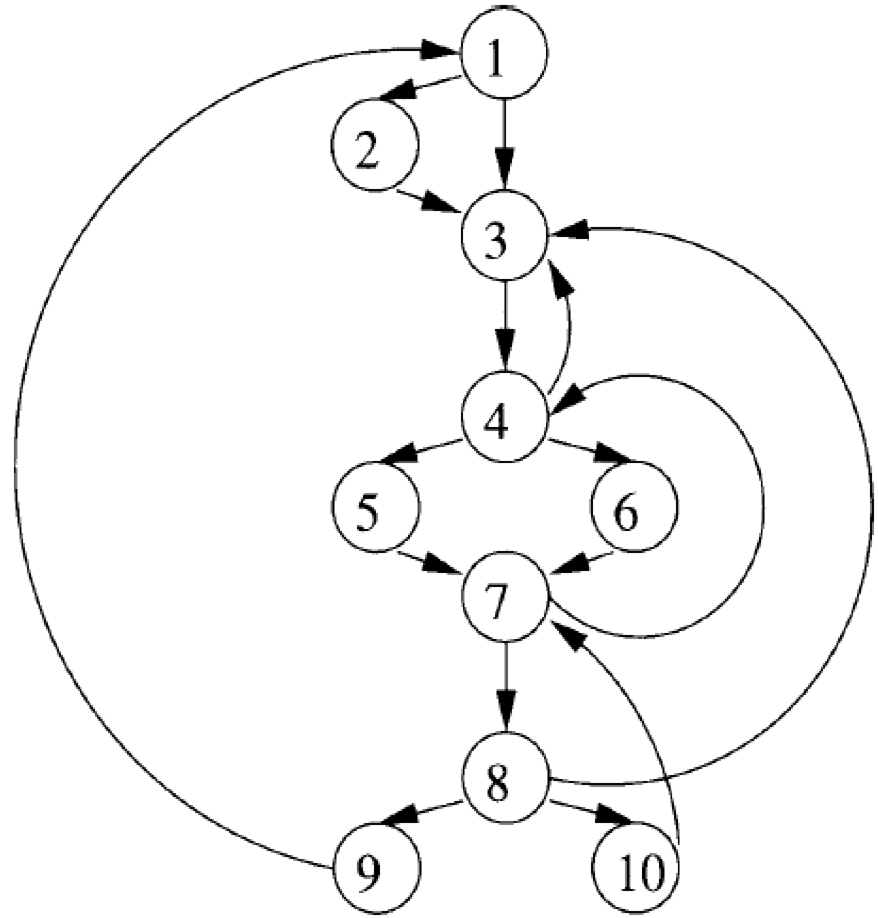
b)

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
- 相关概念
  - 支配结点
  - 深度优先排序
  - 回边
  - 图的深度
  - 可归约性
- 两个重要问题
  - 如何寻找循环
  - 数据流分析的收敛速度

支配结点(必经节点)

- 如果每一条从入口结点到达n的路径都经过d，我们就说d支配(dominate)n，记为d dom n
- 右图：
  - 2只支配自己
  - 3支配除了1，2之外的其它所有结点
  - 4支配1、2、3之外的其它结点
  - 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

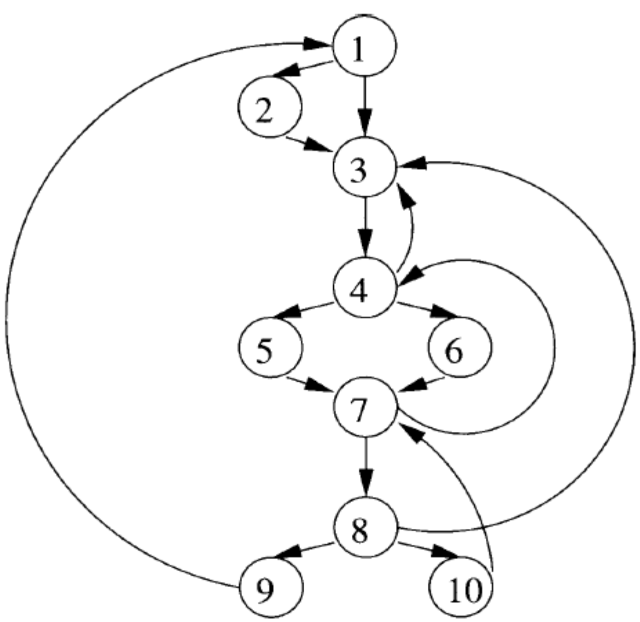


支配结点树

- 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 dom n，那么d dom m

寻找支配结点

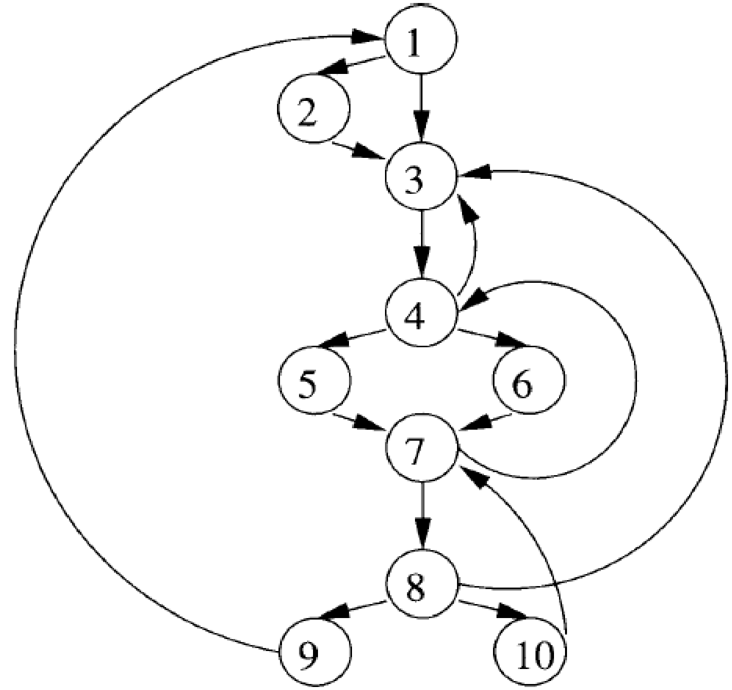
- 求解如右图所示的数据流方程组，就可以得到各个结点对应的支配结点
- D(n) = OUT[n]



	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	$\mathrm{OUT}[\mathrm{ENTRY}] = \{\mathrm{ENTRY}\}$
交汇运算(∧)	$\cap$
方程式	$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$ $\mathrm{IN}[B] = \bigwedge_{P,pred(B)} \mathrm{OUT}[P]$
初始化设置	$\mathrm{OUT}[B] = N$

图 9-40 一个计算支配结点的数据流算法

```
OUT[ENTRY] = v_ENTRY;
for (除 ENTRY 之外的每个基本块 B) OUT[B] = ⊤;
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] = f_B(IN[B]);
    }
```



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

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

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

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

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

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

$$\begin{aligned} D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\ &= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\} \end{aligned}$$

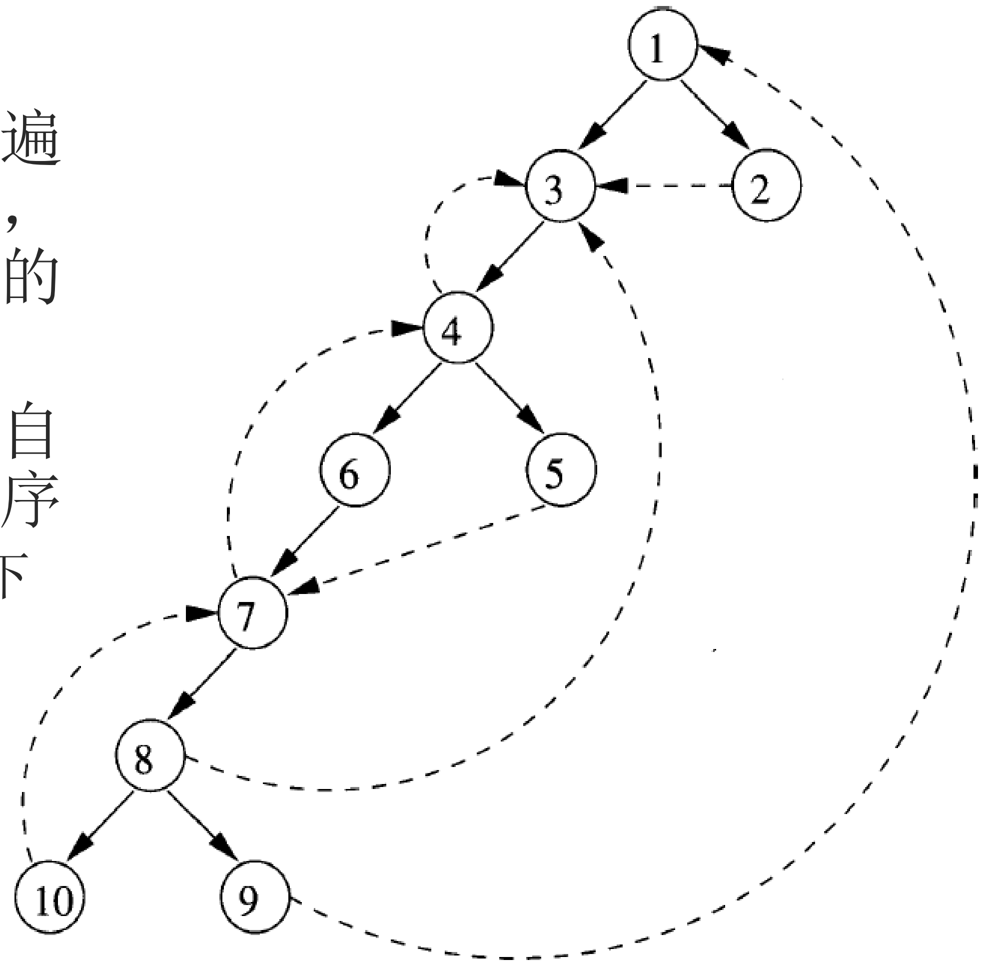
$$D(8) = \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\}$$

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

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

深度优先排序

- 深度优先排序
  - 先访问一个结点，然后遍历该结点的最右子结点，再遍历这个子结点左边的子结点，依此类推
  - 具体访问时，我们可以自己设定各个子结点的顺序
    - 哪个是最右的，哪个是下一个子结点等
- 例子见右图：
  - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



深度优先生成树和排序算法

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

```
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 → s 加入到 T 中;
            search(s);
        }
    dfn[n] = c;
    c = c - 1;
}

main() {
    T = ∅; /* 边集 */
    for (G的各个结点n)
        把 n 标记为 “unvisited”;
    c = G的结点个数;
    search(n0);
}
```

流图的边的分类

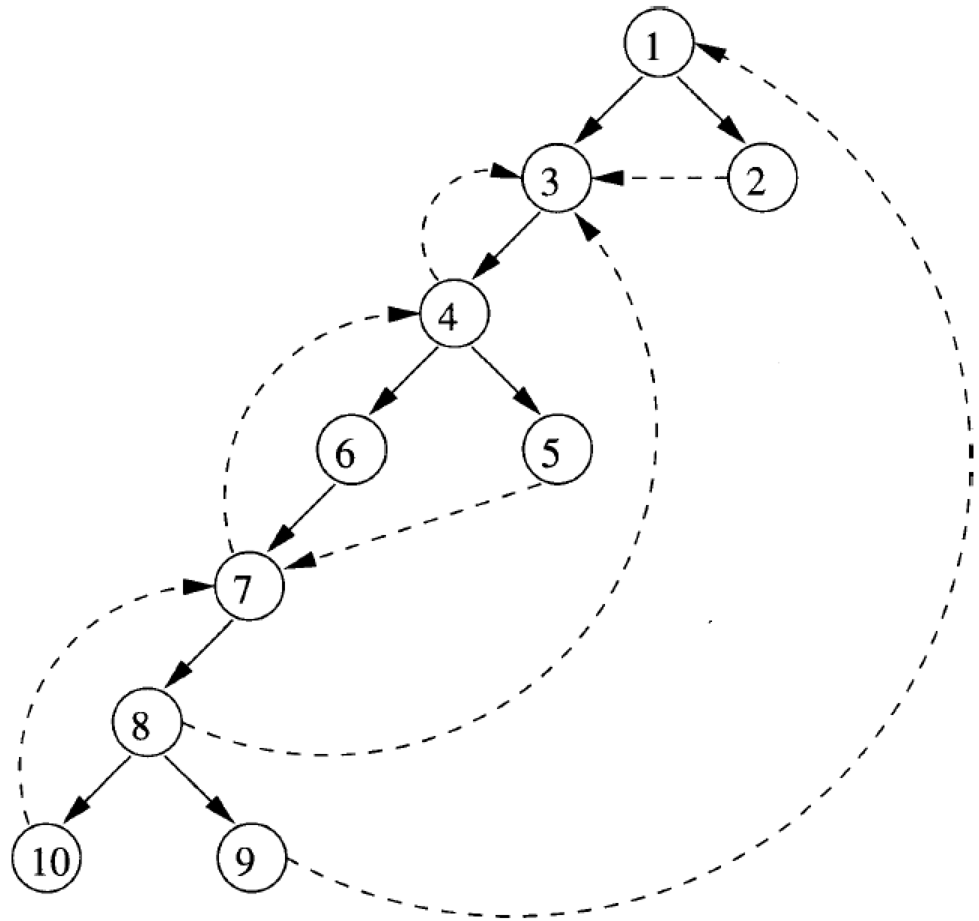
- 前进边
  - 从结点m到达m在DFST树中的一个真后代的边
  - DFST中的所有边都是前进边
- 后退边
  - 从m到达m在DFST树中的某个祖先的边
- 交叉边
  - 边的src和dest都不是对方的祖先
  - 一个结点的子结点按照它们被加入到树中的顺序从左到右排列，那么所有的交叉边都是从右到左的

回边和可归约性

- 回边
  - 边a→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，但是所有的回边仍然都是后退边
- 实践中出现的流图基本都是可归约的

流图的深度

- 一个流图，相对于一棵DFST，的深度
  - 各条无环路径上后退边数中的最大值
  - 这个深度不会大于直观上所说的流图中的循环嵌套深度。
- 对于可归约的流图，我们可以使用“回边”来定义，而且可以说是“流图的深度”
- 右边的流图深度为3
  - 10→7→4→3



自然循环

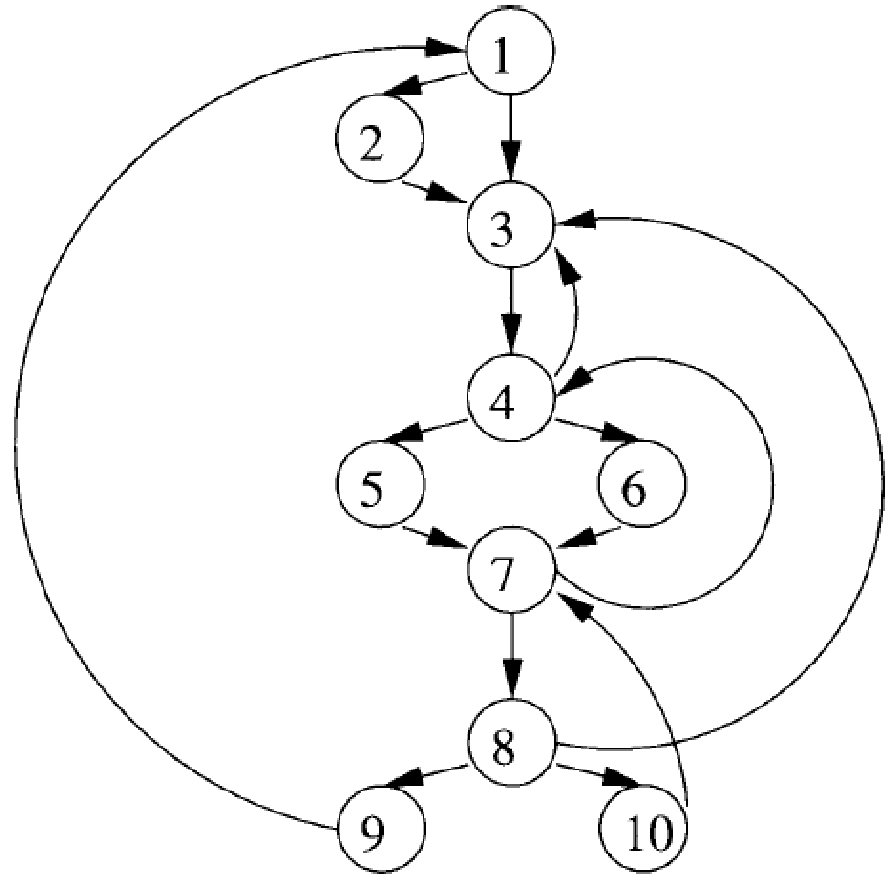
- 自然循环的性质
  - 有一个唯一的入口结点（循环头 header）。这个结点支配循环中的所有结点
  - 必然存在进入循环头的回边
- 自然循环的定义
  - 给定回边n→d的自然循环是d加上不经过d就能够到达n的结点的集合
  - d是这个循环的头

自然循环构造算法

- 输入：流图G和回边n→d;
- 输出：回边n→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，逆向地对数据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→7
  - {7, 8, 10}
- 回边：7→4
  - {4, 5, 6, 7, 8, 10}
  - 包含了前面的循环
- 回边4→3, 8→3
  - 同样的头
  - 同样的结点集合{3, 4, 5, 6, 7, 8, 10}
- 回边9→1
  - 整个流图



自然循环的性质

- 除非两个循环具有同样的循环头，他们
  - 要么互不相交（分离的）
  - 要么一个嵌套于另一个中
- 最内层循环
  - 不包含其它循环的循环
  - 通常是最需要进行优化的地方

课程结束了

谢 谢！