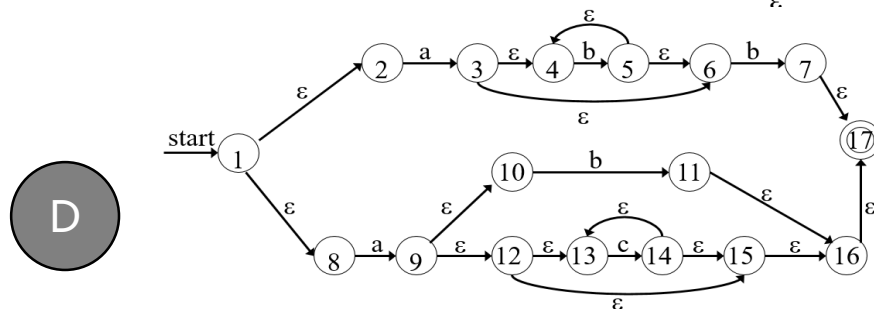
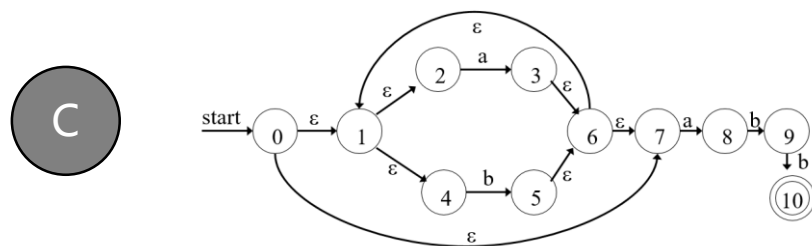
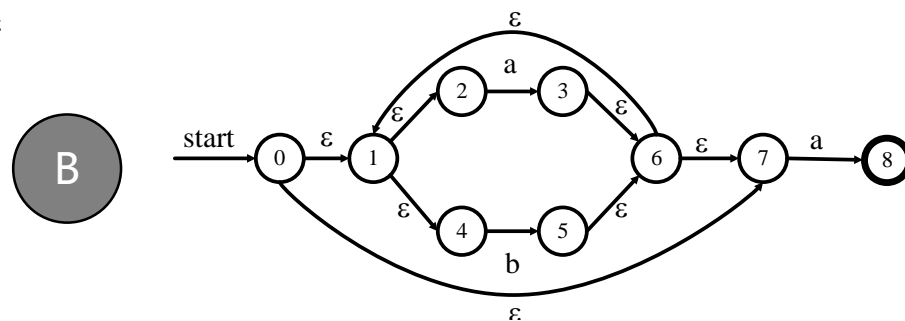
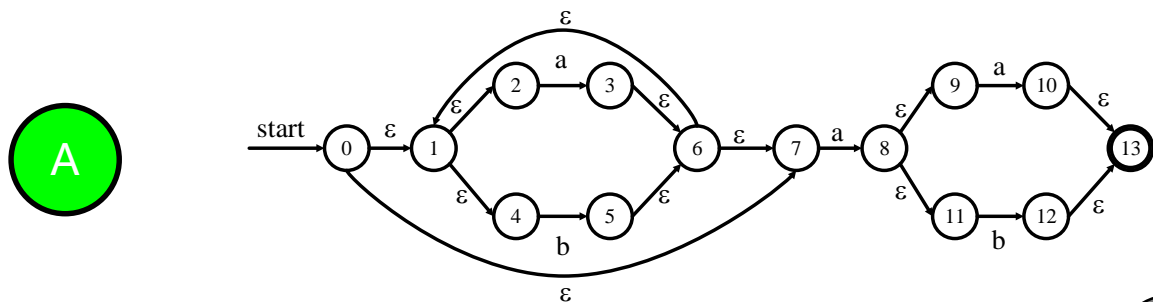


用Thompson方法将正则表达式 $(a|b)^*a(a|b)$ 转换为NFA N , N 为____



第九章 代码优化

优化的主要来源

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 $A[i][j] = A[i][j] + 10;$ 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

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);
}
```

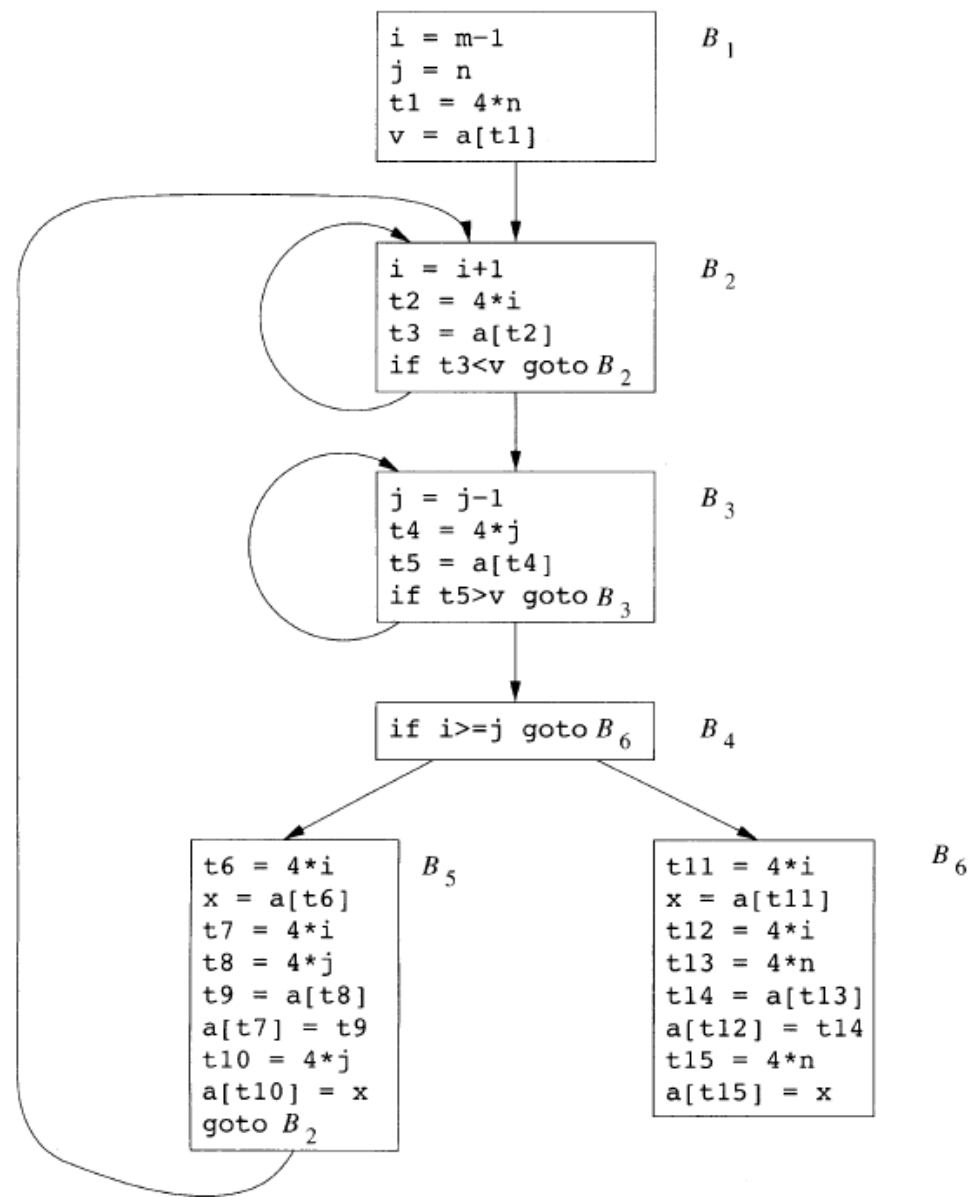
三地址代码

(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



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；

```
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

a) 消除之前

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

B₅

b) 消除之后

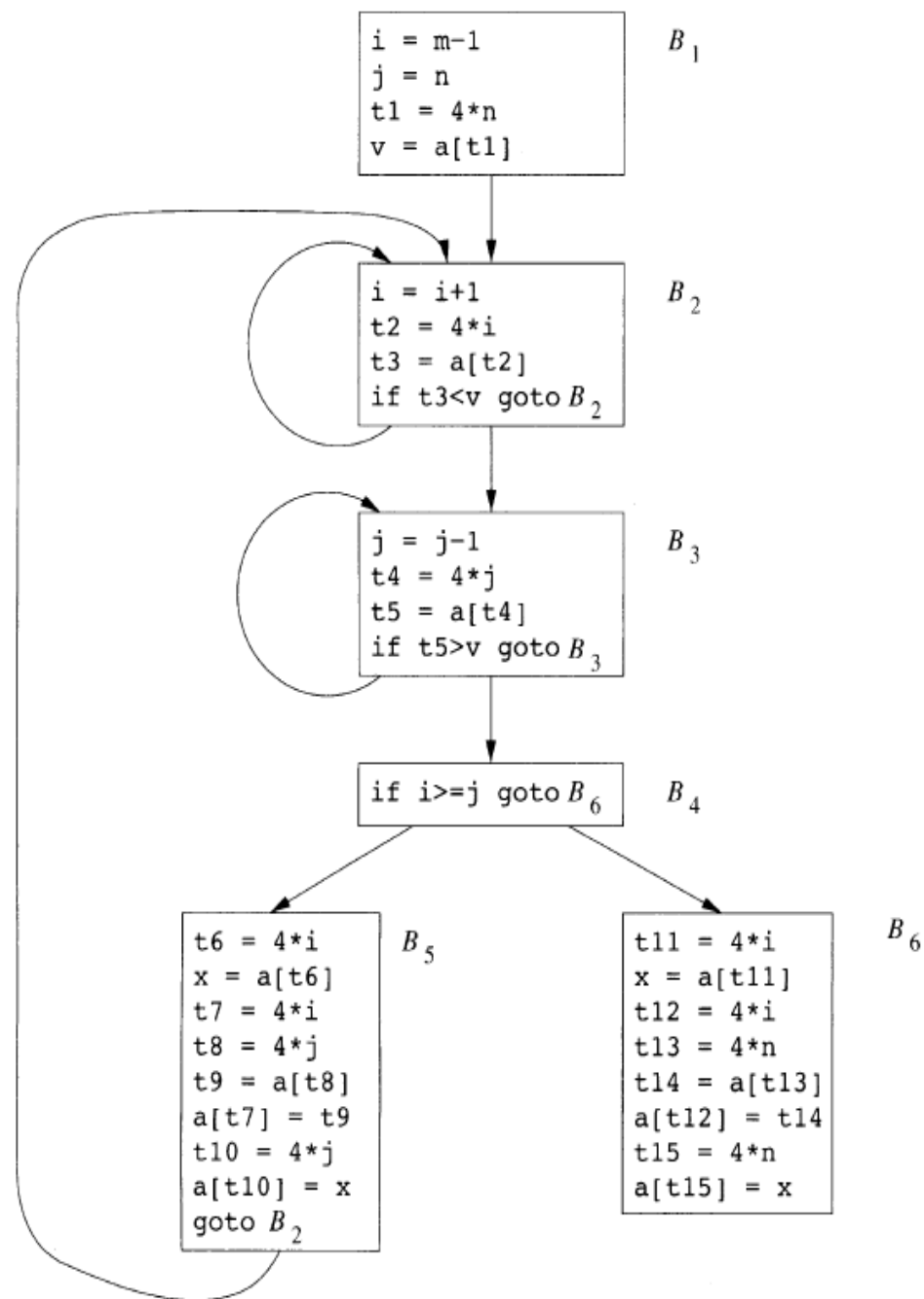
全局公共子表达式的例子

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 ;
- t_2 、 t_4 在赋值之后没有被改变过, 因此 B_5 中可直接使用它们;
- t_4 在替换 t_8 之后, B_5 : $a[t_8]$ 和 B_3 : $a[t_4]$ 又相同;

同样:

- B_5 中赋给 x 的值和 B_2 中赋给 t_3 的值相同;
- B_6 中的 $a[t_{13}]$ 和 B_1 中的 $a[t_1]$ 不同, 因为 B_5 中可能改变 a 的值;



消除公共子表达式后的结果

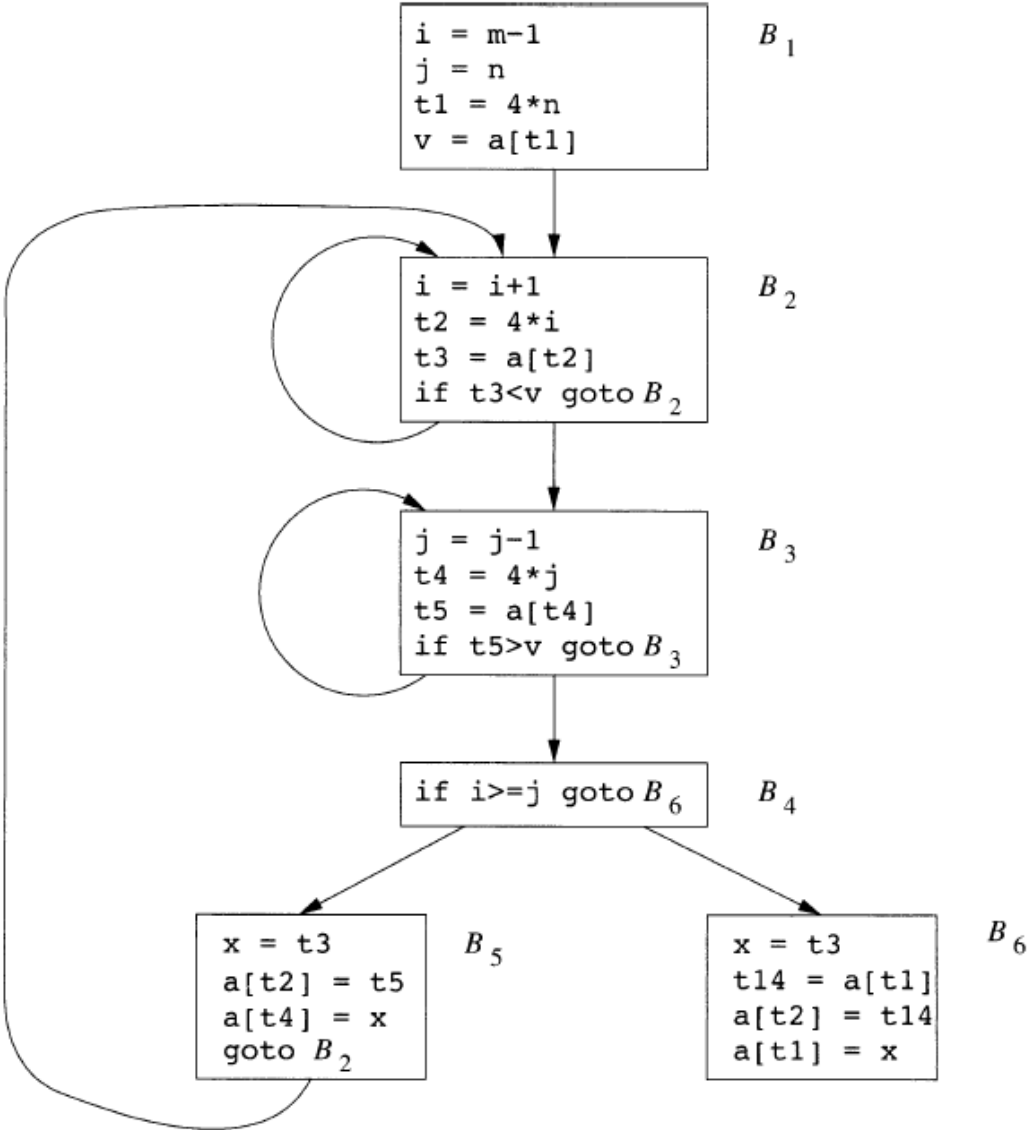
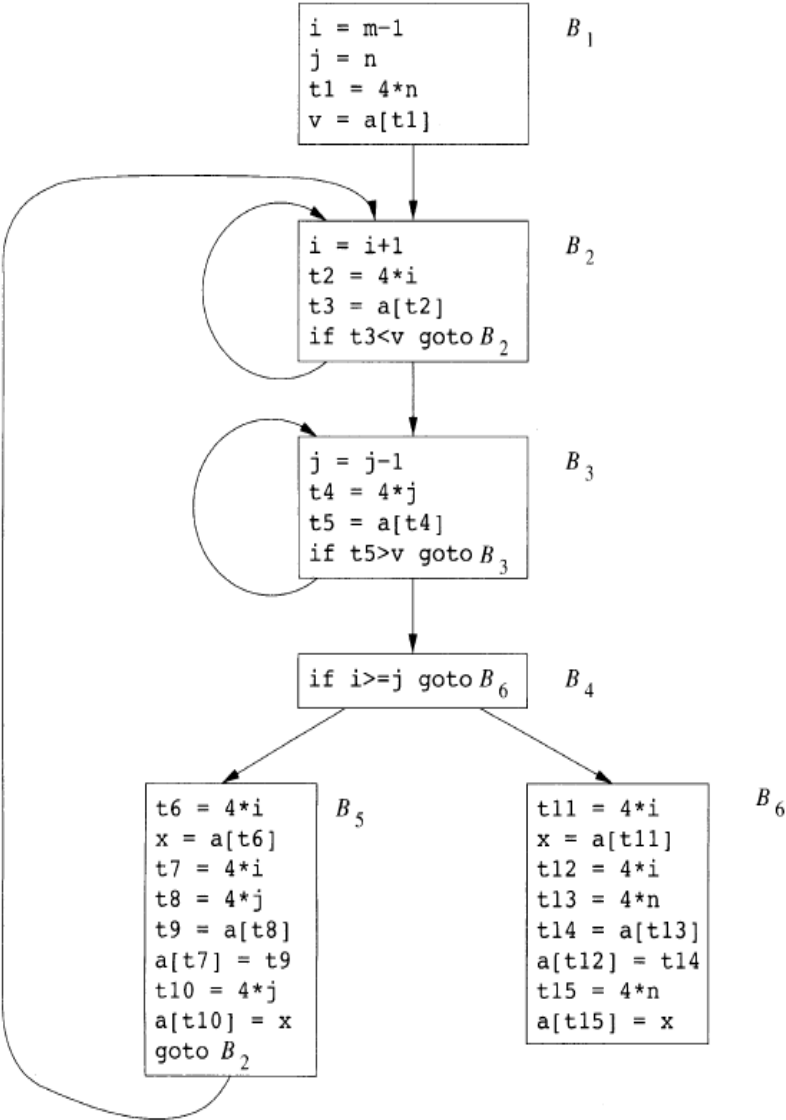


图 9-5 经过公共子表达式消除之后的 B_5 和 B_6

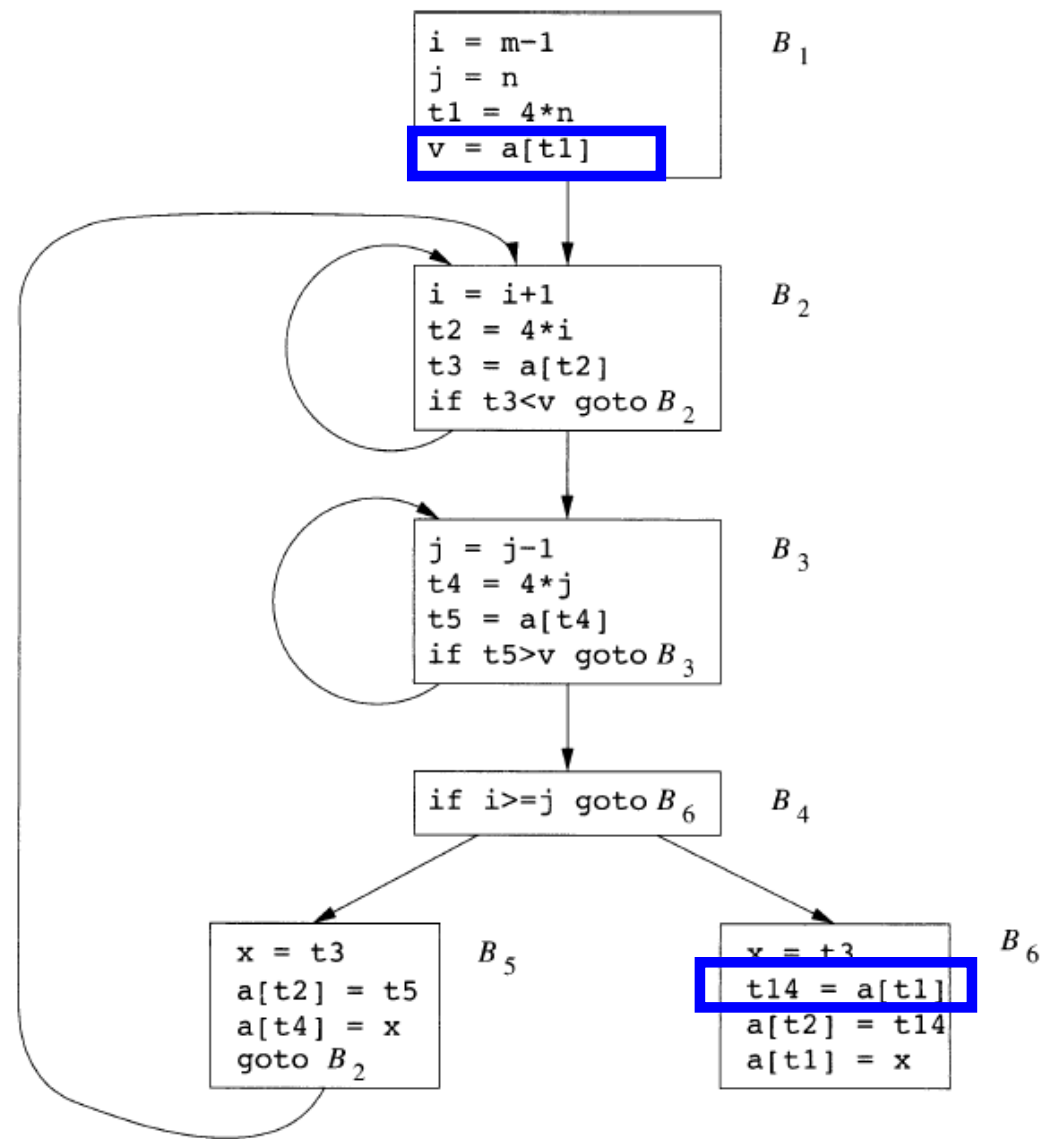
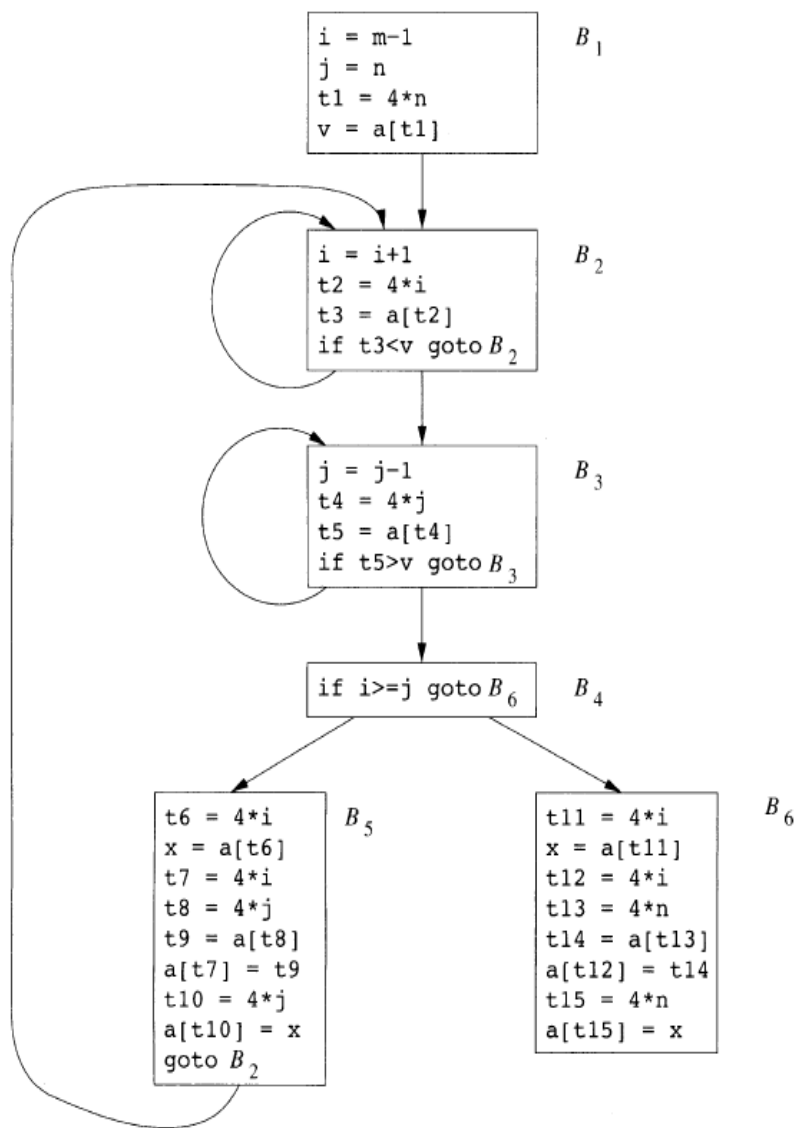


图 9-5 经过公共子表达式消除之后的 B_5 和 B_6

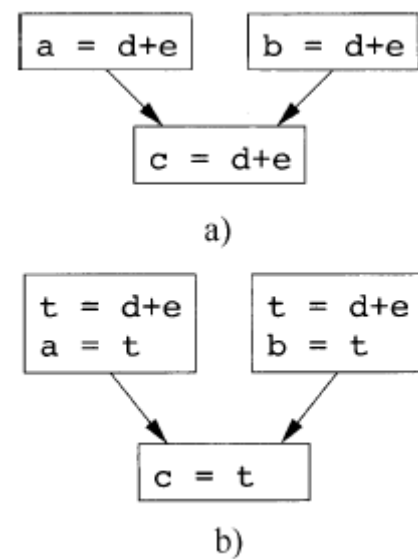
复制传播

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

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

B_5

```
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

$x=t3$		$a[t2] = t5$
$a[t2] = t5$		$a[t4] = t3$
$a[t4] = t3$	\implies	goto B_2
goto B_2		

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

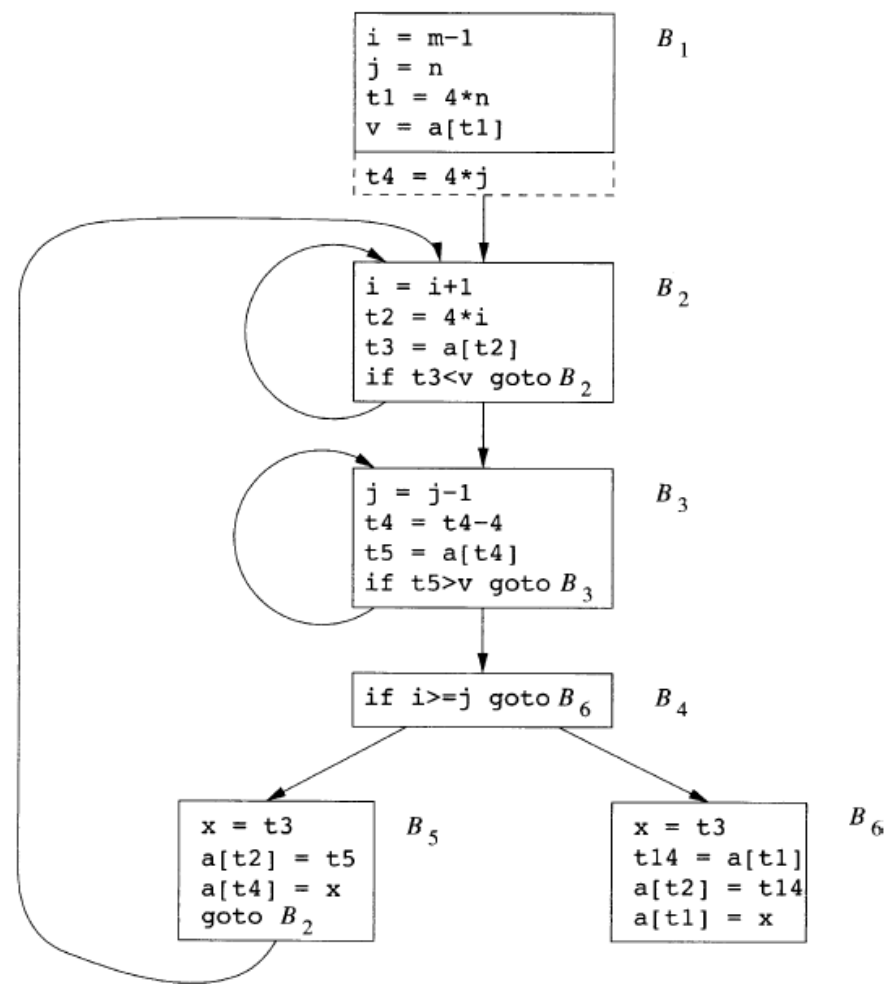


图 9-8 对基本块 B_3 中的 $4*j$ 应用强度消减优化

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

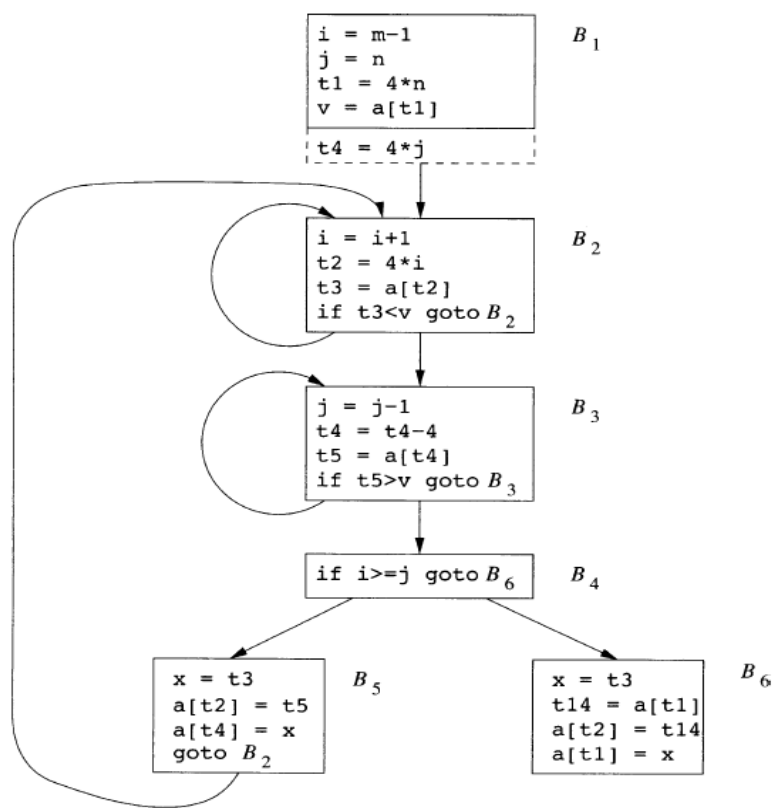


图 9-8 对基本块 B_3 中的 $4*j$ 应用强度消减优化

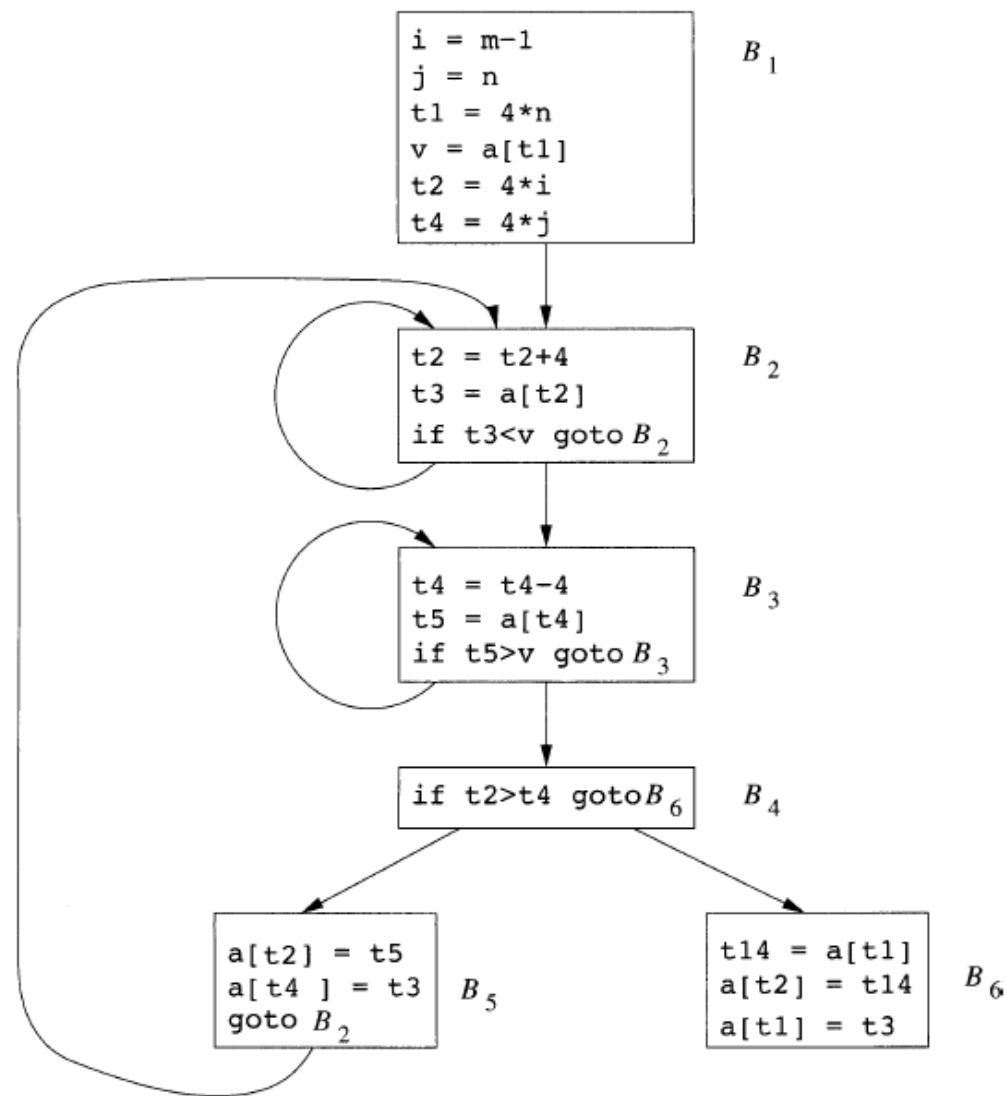
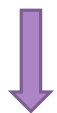


图 9-9 归纳变量消除之后的流图

常量折叠

$A = 3 * 5$



$A = 15$

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_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
- 1,2,3,4,5,6,7,8,3,4,9
- ...

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 $\langle a, \{1, 243\} \rangle$

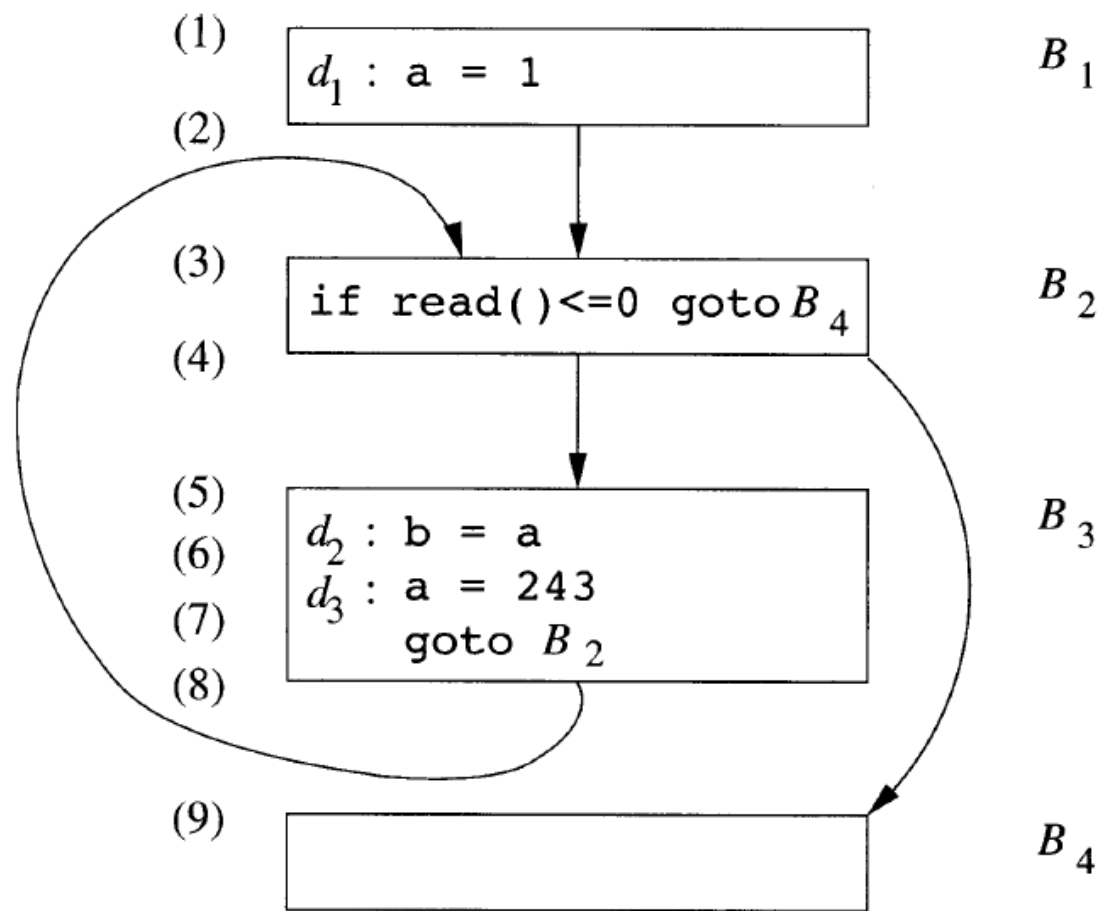


图 9-12 说明数据流抽象的例子程序

例子 (1)

点(5)所有程序状态:

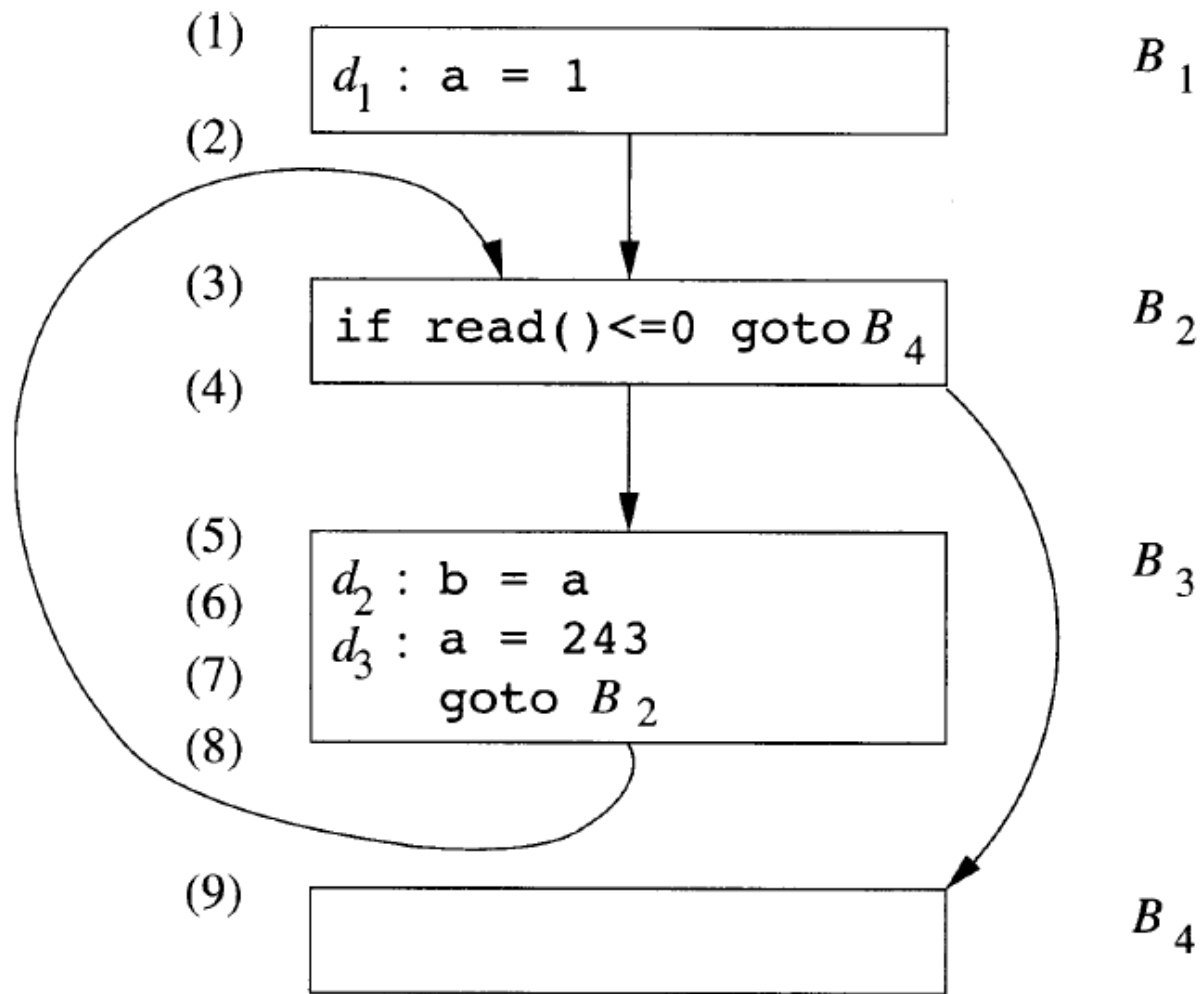
- $a \in \{1, 243\}$
- 由 $\{d_1, d_3\}$ 定值

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



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；求解这些数据流值的过程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里定值，可以使用到达定值
 - 性质形式：x由d1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特定的常量赋值语句赋予的。
 - 性质形式：x=c，以及x=NAC

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这个条件；

域

- 所有可能的数据流值的集合称为这个数据流值的域
- 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

对一组约束求解，得到各个点上数据流值。

约束分成两类：基于语句和基于控制流

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该语句语义的约束；
- 语句语义通常用传递函数表示，它把一个数据流值映射为另一个数据流值；
 - $IN[s]=f_s(OUT[s])$ （逆向）
 - $OUT[s]=f_s(IN[s])$ （正向）

基于控制流的约束

- 在基本块内部，一个语句的输出和下一语句的输入相同；
- 流图的控制流边也对应新的约束

例子 (1)

假设我们考虑各个变量在某个程序点上是否常量

- s 是语句 $x=3$;
- 考虑变量 x, y, z ;
- $IN[s]: x:NAC; y:7; z:3$

那么 $OUT[s]$ 就是: $x:3; y:7; z:3$

如果

- s 是 $x=y+z$, $OUT[s]$ 是?
- s 是 $x=x+y$, $OUT[s]$ 是?

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；

$$IN[B] = f_B(OUT[B]) \quad \text{或者} \quad OUT[B] = f_B(IN[B])$$

设基本块包含语句 s_1, s_2, \dots, s_n ；

$$f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$$

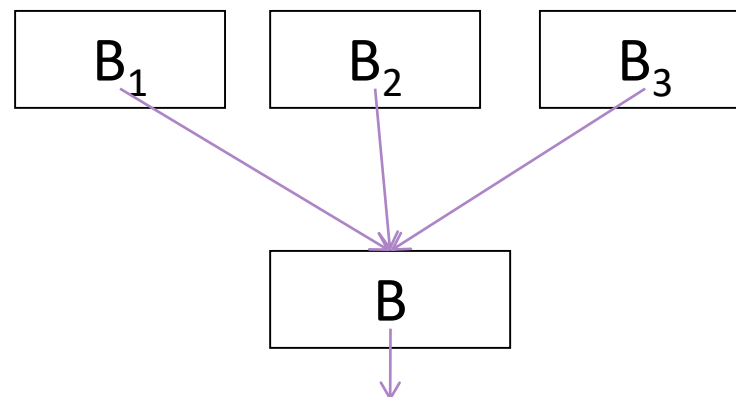
基本块之间的控制流约束

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 $IN[B]$;
- $OUT[B]$ 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

到达定值 (0)

到达程序点的所有定值

- 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

到达定值 (1)

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；

- 考虑到间接赋值，如果定值可能不是对 x 进行复制，则不能杀死这个定值

直观含义

- 如果 d 到达 p ，那么在 p 点使用的 x 的值就可能由 d 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

用途：

- 确定 x 在 p 点是否常量
 - 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
- 确定变量是否先使用后定值

到达定值的例子

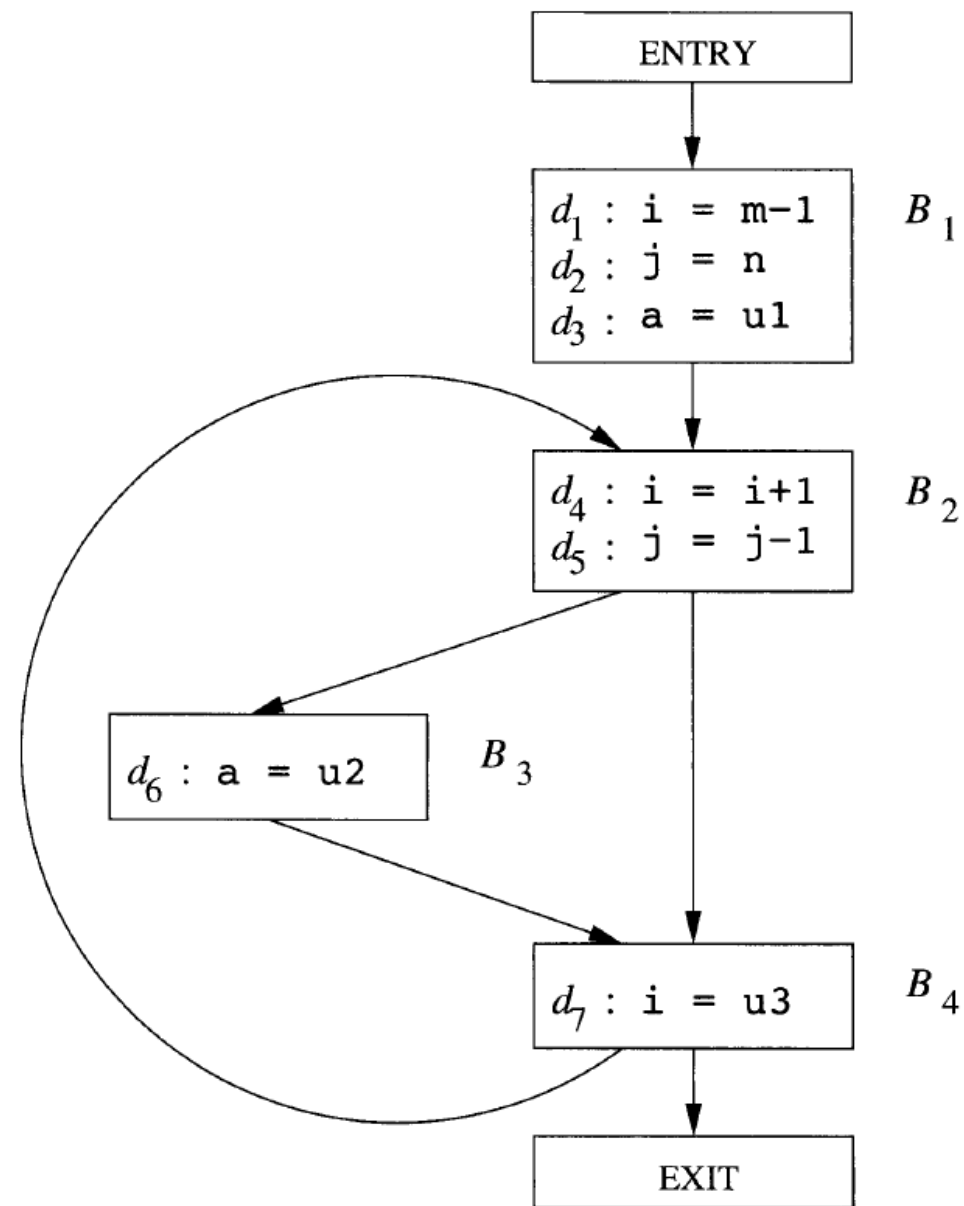
B_1 中全部定值到达 B_2 的开头;

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



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 ；杀死其他对 u 的定值；
- 生成-杀死形式： $f_d(x)=gen_d \cup (x-kill_d)$
- $gen_d=\{d\}$ ， $kill_d=\{\text{程序中其他对}u\text{的定值}\}$

生成-杀死形式的函数的并置（复合）仍具有这个形式

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

$$f_B(x) = \text{gen}_B \cup (x - \text{kill}_B)$$

$$\text{gen}_B = \text{gen}_n \cup (\text{gen}_{n-1} - \text{kill}_n) \cup (\text{gen}_{n-2} - \text{kill}_{n-1} - \text{kill}_n) \cup (\text{gen}_1 - \text{kill}_2 - \text{kill}_3 \dots - \text{kill}_n)$$

$$\text{kill}_B = \text{kill}_1 \cup \text{kill}_2 \cup \dots \cup \text{kill}_n$$

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

基本块:

- $d_1: a = 3$
- $d_2: a = 4$

gen集合: $\{d_2\}$

kill集合: 流图中所有针对a的定值

到达定值的控制流方程

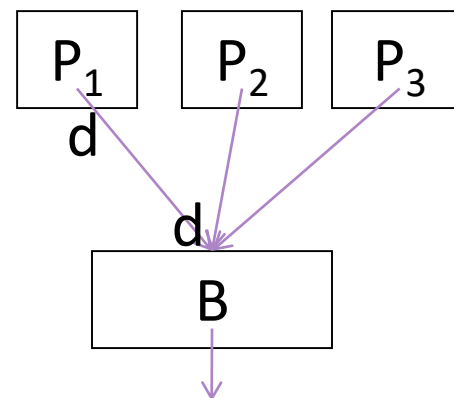
只要一个定值能够沿某条路径到达一个程序点，这个定值就是到达定值；

$$IN[B] = \bigcup_{P \text{ 是 } B \text{ 的前驱基本块}} OUT[P]$$

- 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

U称为到达定值的交汇运算符

d:
 $x=y+z$



控制流方程的迭代解法（1）

ENTRY基本块的传递函数是常函数；

$$\text{OUT}[\text{ENTRY}] = \text{空集}$$

其他基本块

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

$$\text{IN}[B] = \bigcup_{P \text{ 是 } B \text{ 的前驱基本块}} \text{OUT}[P]$$

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]

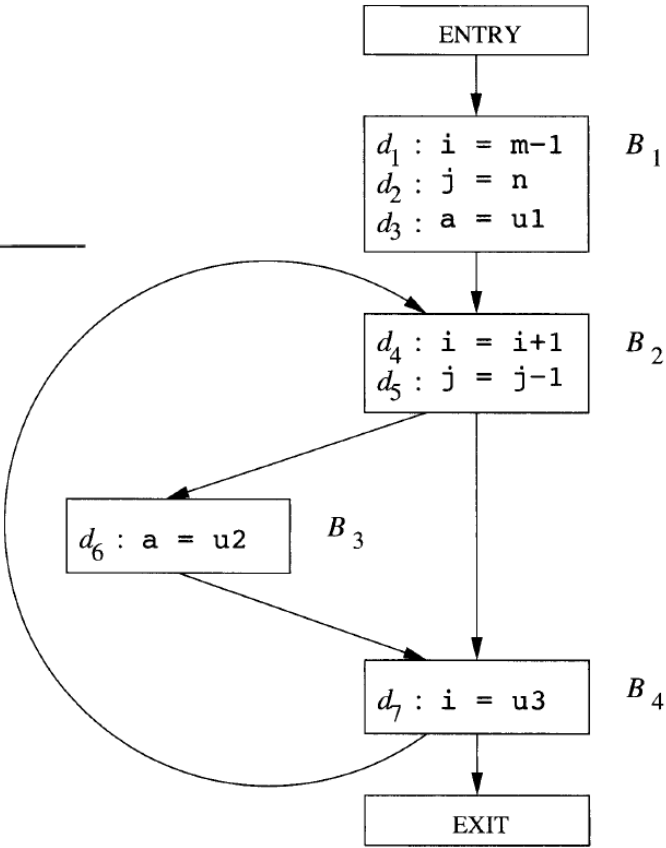
方法：

```
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[ $B$ ] =  $\bigcup_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P]$ ;  
6)          OUT[ $B$ ] =  $\text{gen}_B \cup (\text{IN}[B] - \text{kill}_B)$ ;  
          }  
}
```

到达定值求解的例子

Block B	OUT[B] ⁰	
B_1	000 0000	
B_2	000 0000	
B_3	000 0000	
B_4	000 0000	
EXIT	000 0000	

7个bit从左到右表示 d_1, d_2, \dots, d_n
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



到达定值求解的例子

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

7个bit从左到右表示 d_1, d_2, \dots, d_n

for循环时依次遍历 $B_1, B_2, B_3, B_4, \text{EXIT}$

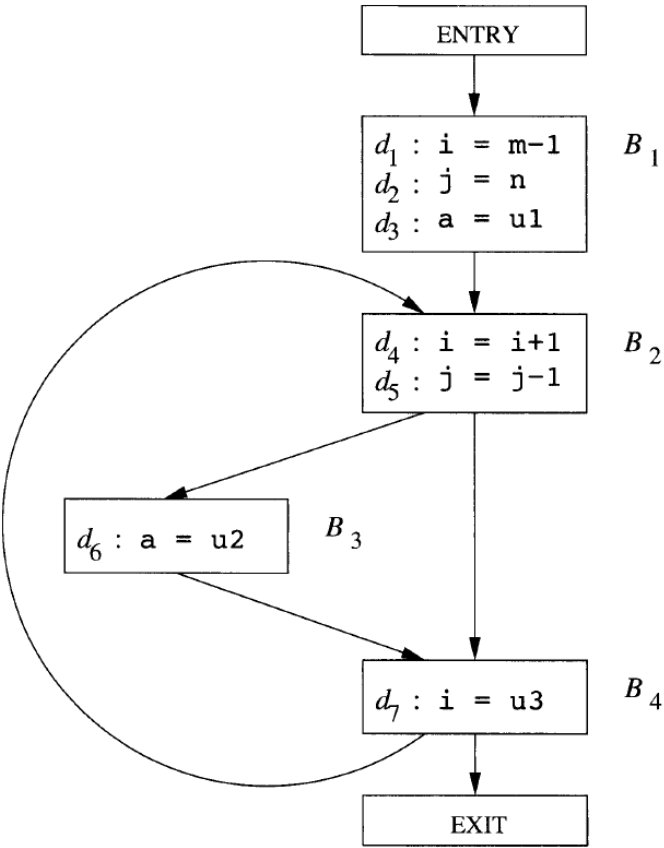
每一列表示一次迭代计算；

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



到达定值求解的例子

Block B	OUT[B] ⁰	IN[B] ¹	OUT[B] ¹	IN[B] ²	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, \dots, d_n

for循环时依次遍历 $B_1, B_2, B_3, B_4, \text{EXIT}$

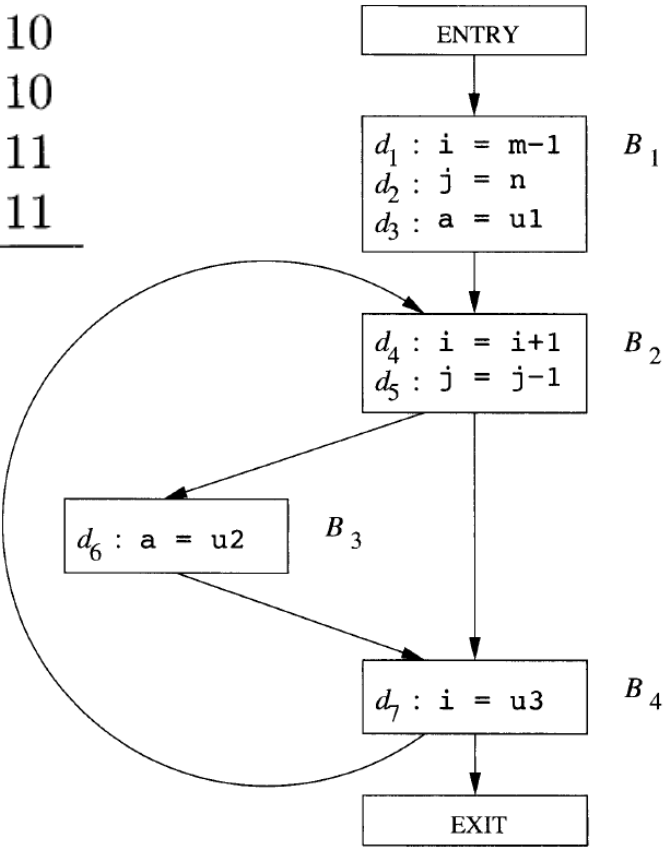
每一列表示一次迭代计算；

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



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，当前仅当存在一条从 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： $i=i+1;$ $j=j-1$
- $use \{i,j\};$ $def \{ \}$

USE_B和DEF_B的用法

假设基本块中包含语句s1,s2,...,sn， 那么

$$\text{use}_B = \text{use}_1 \cup (\text{use}_2 - \text{def}_1) \cup (\text{use}_3 - \text{def}_1 - \text{def}_2) \cup (\text{use}_n - \text{def}_1 - \text{def}_2 \dots - \text{def}_{n-1})$$

$$\text{def}_B = \text{def}_1 \cup \text{def}_2 \cup \dots \cup \text{def}_n$$

活跃变量数据流方程

任何变量在程序出口处不再活跃

- $IN[EXIT] = \text{空集}$

对于所有不等于EXIT的基本块

- $OUT[B] = \bigcup_{S \text{ 是 } B \text{ 的后继基本块}} IN[S]$
- $IN[B] = use_B \cup (OUT[B] - def_B)$

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

```
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[ $B$ ] =  $\bigcup_{S \text{ 是 } B \text{ 的一个后继}} \text{IN}[S]$ ;  
        IN[ $B$ ] =  $use_B \cup (\text{OUT}[B] - def_B)$ ;  
    }
```

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

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

- $OUT[B] = e_gen_B \cup (IN[B] - e_kill_B)$
- $IN[B] = \cap_{P \text{ 是 } B \text{ 的前驱基本块}} OUT[P]$

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

```
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	$OUT[B] = \emptyset$	$IN[B] = \emptyset$	$OUT[B] = U$

总结

全局公共子表达式

复制传播

代码移动

归纳变量

数据流分析（到达定值、活跃变量、可用表达式）