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

基本块和流图

中间代码的流图表示法

- 中间代码划分成为基本块(basic block)，其特点是单入口单出口，即：
  - 控制流只能从第一个指令进入
  - 除了基本块最后一个指令，控制流不会跳转/停机
- 流图中结点是基本块，边指明了哪些基本块可以跟在一个基本块之后运行

流图可作为优化的基础

- 它指出了基本块之间的控制流
- 可根据流图了解一个值是否会被使用等信息

划分基本块的算法

输入：三地址指令序列

输出：基本块的列表

方法：

- 确定leader指令（基本块的第一个指令）符合以下任一条：
  - 中间代码的第一个三地址指令
  - 任意一个条件或无条件转移指令的目标指令
  - 紧跟在一个条件/无条件转移指令之后的指令
- 确定基本块
  - 每个首指令对应于一个基本块：从首指令（包含）开始到下一个首指令（不含）

基本块划分举例

第一个指令
- 1

跳转指令的目标
- 3、2、13

跳转指令的下一条指令
- 10、12

基本块：

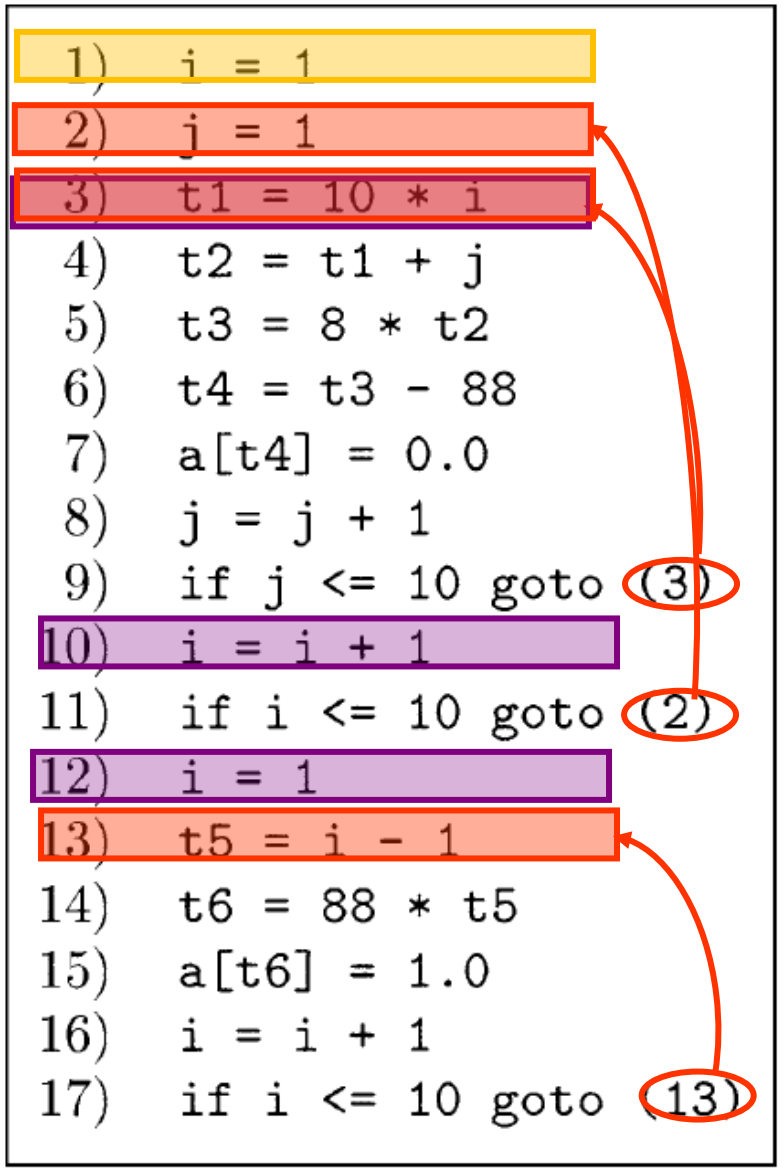
1-1；2-2；

3-9(包括跳转语句)；

10-11(包括跳转语句)；

12-12；

13-17 (包括跳转语句)



流图的构造

流图的顶点是基本块

两个顶点B和C之间有一条有向边，当且仅当基本块C的第一个指令有可能在B的最后一个指令之后执行。原因：

- B的结尾指令是一条跳转到C的开头的条件/无条件语句
- 在原来的序列中，C紧跟在B之后，且B的结尾不是无条件跳转语句
- B是C的前驱，C是B的后继

流图中增加额外的入口，出口结点各一个

- 不对应于实际的中间指令（是增加）
- 入口到第一条指令有一条边
- 从任何可能最后执行的基本块到出口有一条边

流图绘制

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

流图的例子

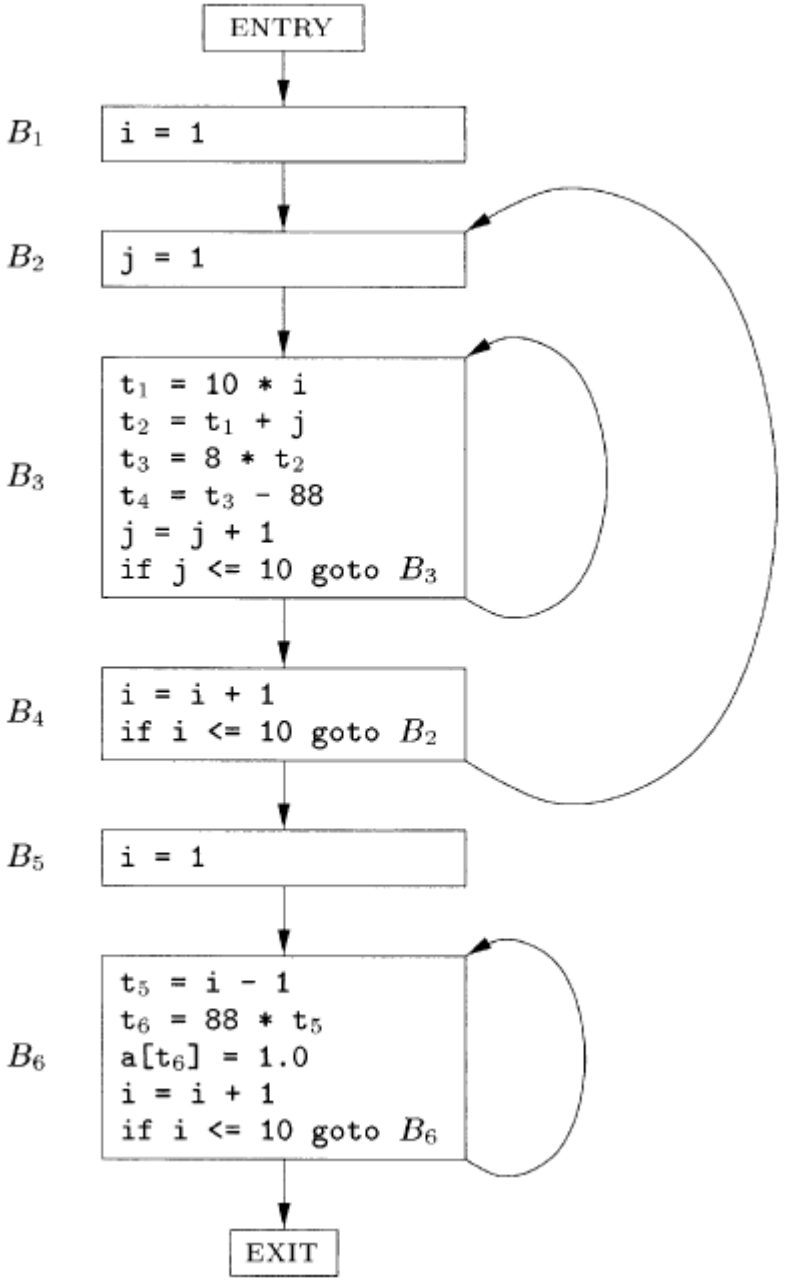
因跳转而生成的边

- B3 → B3
- B4 → B2
- B6 → B6

因为顺序而生成的边

- 其它

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```



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

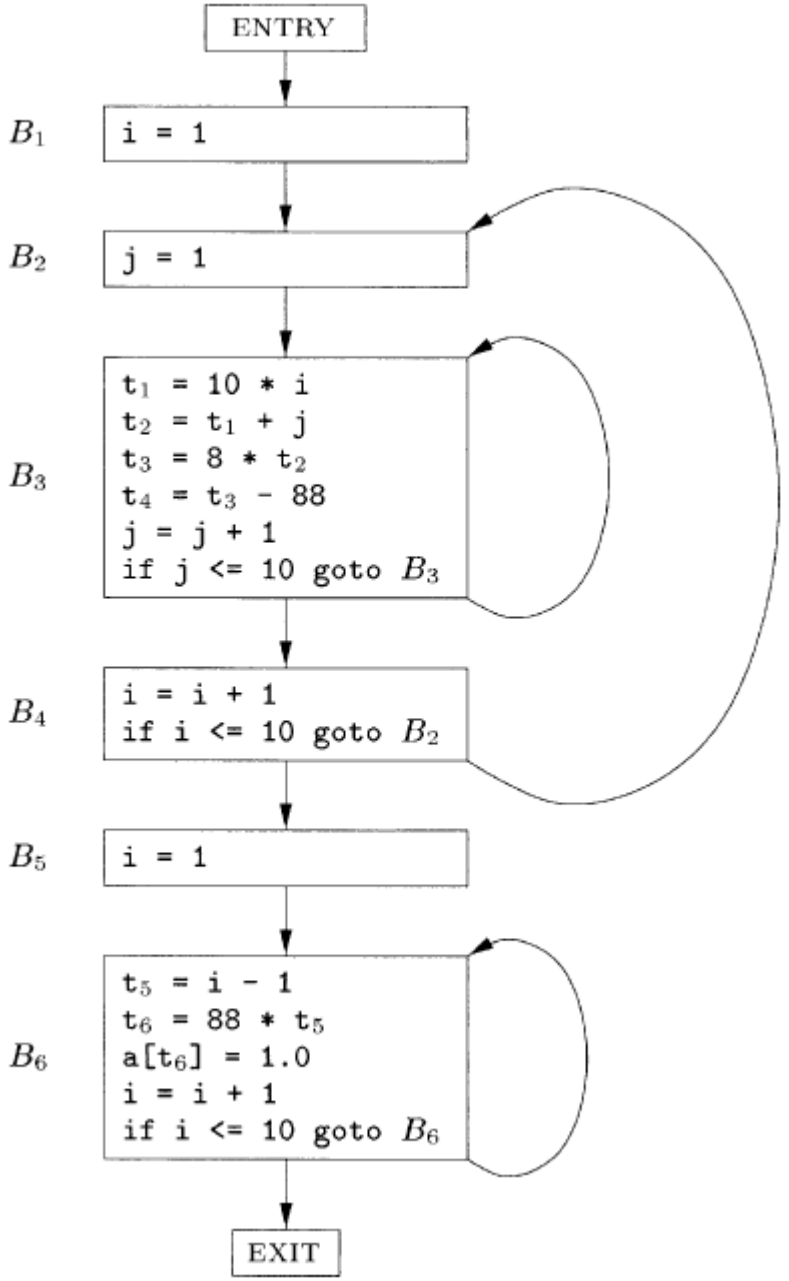
循环的定义

- 循环L是一个结点集合
- 存在一个循环入口（loop entry）节点，其唯一的前驱可以是循环L之外的结点
- 其余结点都存在到达L的入口的非空路径，且路径都在L中。

循环的例子

循环

- {B3}
- {B6}
- {B2,B3,B4}

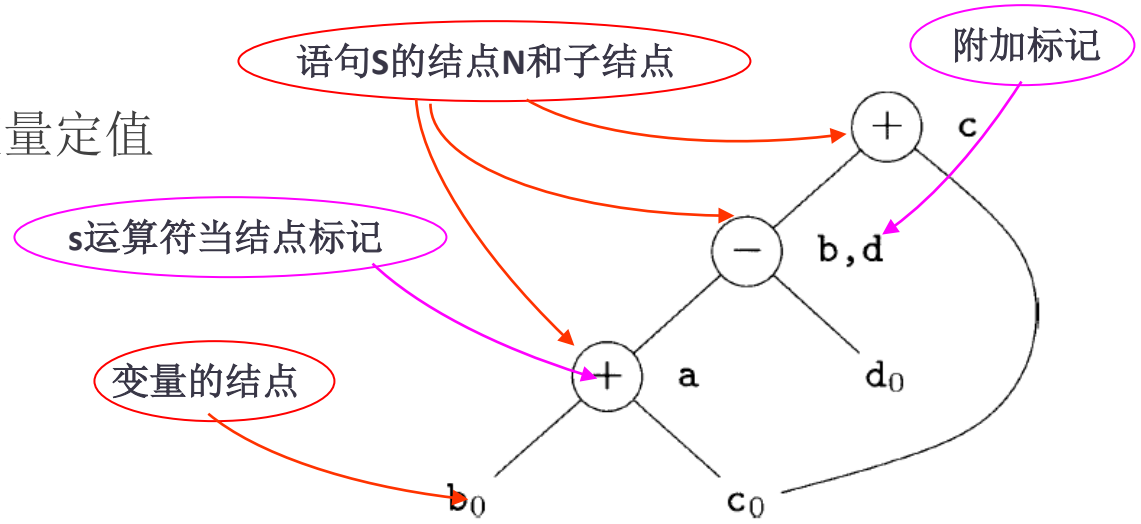


基本块的几种优化

一个基本块可用一个DAG图表示(基本的DAG图)

- 每个变量对应于一个结点，表示初始值
- 每条语句s有一个相关结点N,具有
  - 子结点：对应于其它语句,是在s之前，最后一个对s所使用的某个运算分量进行定值的语句。
  - 标记：s的运算符
  - 附加标记：一组变量，表明s是在此基本块内最晚对该变量定值
- 某些输出结点：结点对应的变量在基本块出口处活跃

（出口活跃属于全局数据流分析）



DAG示例

DAG图的构造

为基本块中出现的每个变量建立结点（表示基本值）

顺序扫描各个三地址指令

- 如果指令为x=y op z
  - 为这个指令建立结点N，标号为op；
  - N的子结点为y、z当前关联的结点；
  - x和N关联；
- 如果指令为x=y；
  - 不建立新结点；
  - 设y关联到N，那么x现在也关联到N

扫描结束后，对于所有在出口处活跃的变量x，将x所关联的结点设置为输出结点

DAG的作用

DAG图描述了基本块运行时各个值之间的关系。

可以DAG为基础，对代码进行转换

- 消除局部公共子表达式
- 消除死代码
- 对语句重新排序
- 重新排序运算分量的顺序

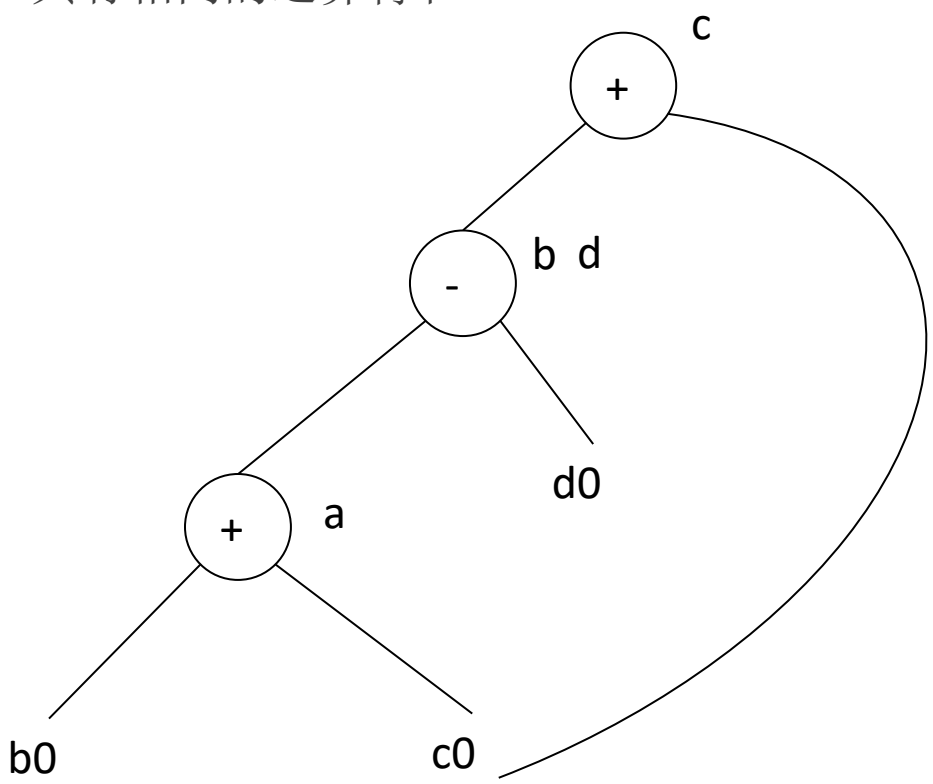
局部公共子表达式

局部公共子表达式的发现

- 建立某个结点M之前，首先检查是否存在一个结点N，它和M具有相同的运算符和子结点（顺序相同）。
- 如果存在，则不需要生成新的结点，用N代表M；

例如：

- a=b+c
- b=a-d
- c=b+c
- d=a-d
- 找出公共的表达式？



注意：两个b+c实际上并不是公共子表达式

消除死代码

死代码：是指在程序操作过程中永远不可能被执行到的代码。

在DAG图上消除没有附加活跃变量的根结点（没有父结点的结点），即可消除死代码

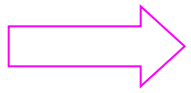
如果图中c、e不是活跃变量，则可以删除标号为e、c的结点。

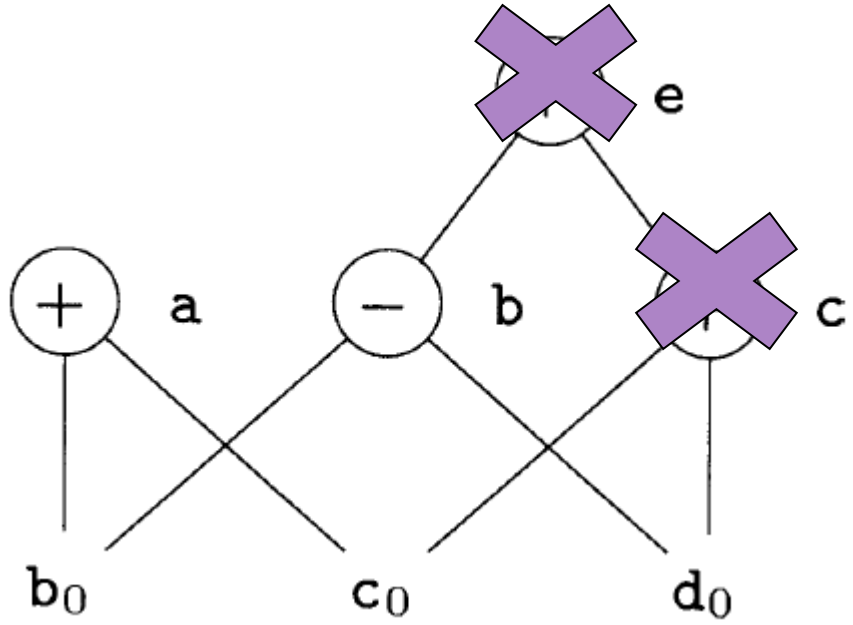
a = b + c

b = b - d

c = c + d

e = b + c





DAG方法的不足

a = b + c
b = b – d
c = c + d
e = b + c

为了计算第四式的e值，而有的2,3式实际上是冗余的，也就是第一式和第四式等价。该代码本可优化，但发现不了

b + c = (b – d) + (c + d)

在DAG上应用代数恒等式的优化

消除计算步骤

- x+0=0+x=x　　x-0=x
- x*1=1*x=x　　x/1=x

降低计算强度

- x^2=x*x　　2*x=x+x

常量合并

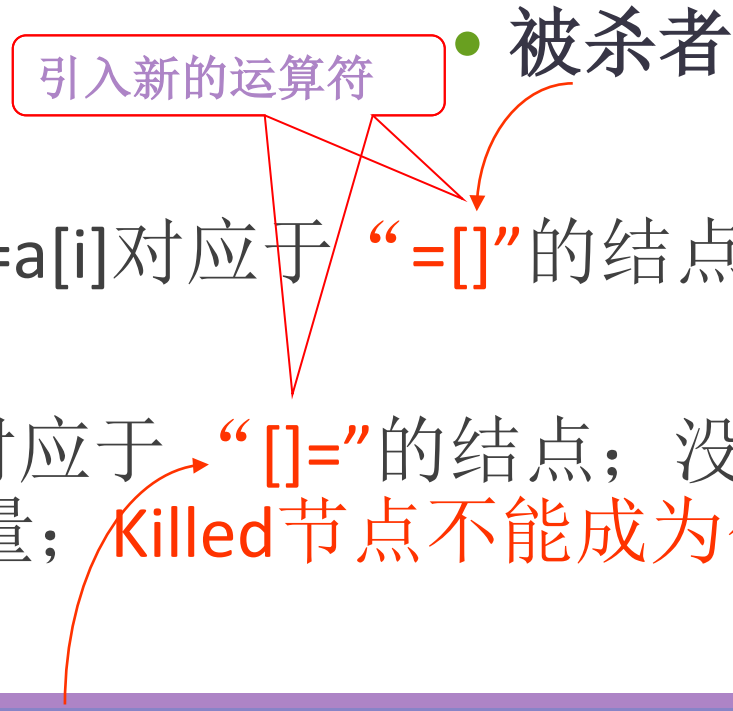
- 2*3.14可以用6.28替换

实现这些优化时，只需要在DAG图上寻找特定的模式

数组引用—避免误优化

注意：a[j]可能改变a[i]的值，因此不能和普通的运算符一样构造相应的结点

- x=a[i]
- a[j]=y
- z=a[i]



从数组取值的运算x=a[i]对应于“=[]”的结点，x作为这个结点的标号之一；

对数组赋值的运算对应于“[]=”的结点；没有关联的变量、且杀死所有依赖于a的变量；Killed节点不能成为公共子表达式

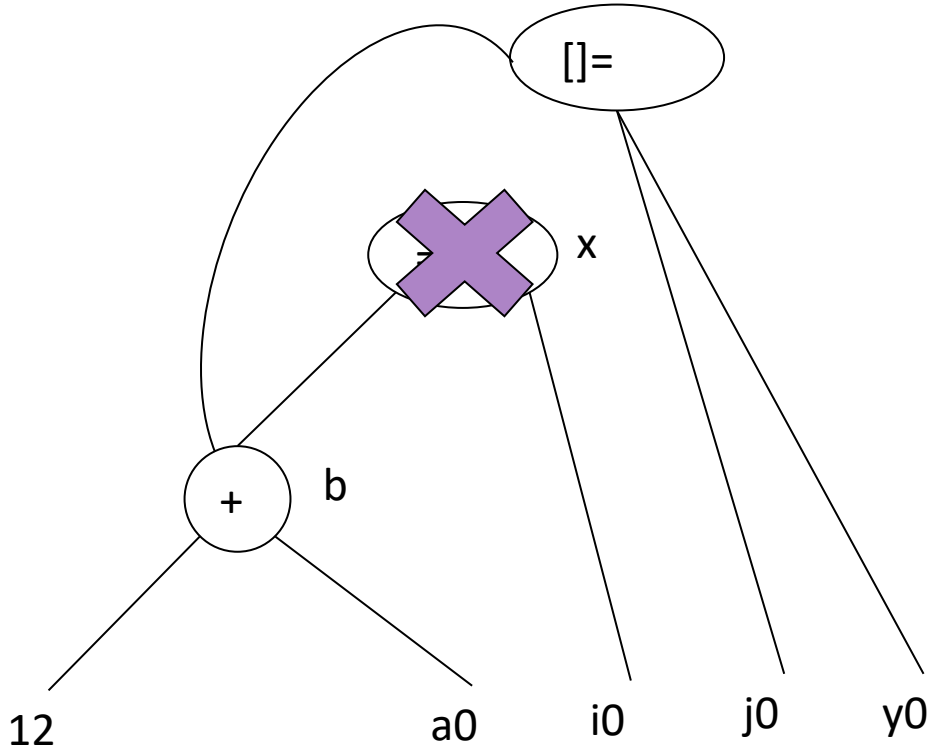
数组引用的DAG的例子

b=12+a

x=b[i]

b[j]=y

注意a是被杀死结点的孙结点。



- 如果没有杀手的出现则被杀者本可以象前述一样进行优化。
- 杀死的意思——指不能参加优化

指针赋值/过程调用

通过指针进行取值/赋值：x=*p *q=y。最粗略地估计：

- x使用了任意变量，因此无法消除死代码
- 而*q=y对任意变量赋值，因此杀死了全部其他结点(可类似引出新的运算符“=*”与“*=”帮助分析)

杀的范围过大。可以通过（全局/局部）指针分析,缩小范围；比如针对

- p=&x
- *p=y可以只杀死那些以x为附加变量的结点

过程调用也类似，为了安全：

- 必须假设它使用了访问范围内所有变量
- 假设修改了访问范围内的所有变量 。杀谁？
- 全杀 !!

从DAG到基本块

重构的方法

- 对每个结点构造一个三地址语句来计算对应的值
- 结果应该尽量赋给一个活跃的变量
  - 一般为出口活跃，如果不确定则假设所有非临时变量都出口活跃
- 如果结点有多个关联的变量，则需要用复制语句进行赋值。

重组基本块的例子

原三地址码

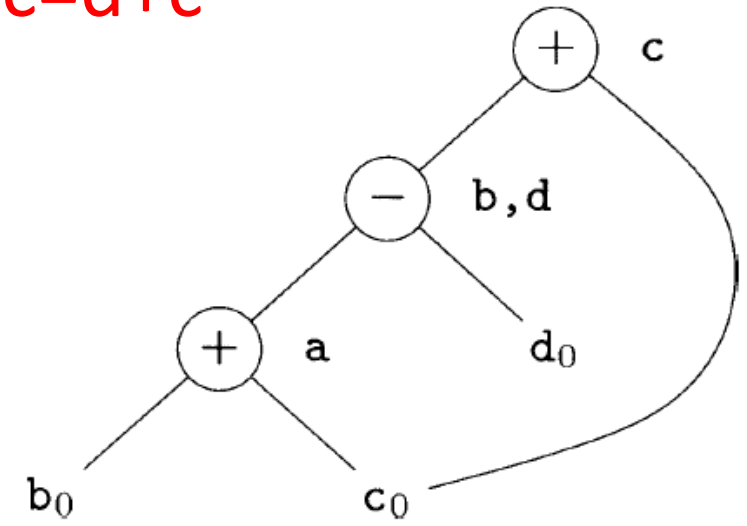
a=b+c

b=a-d

c=b+c

d=a-d

- 根据DAG构造是结点产生的顺序
- a=b+c
- d=a-d
- b=d
- c=d+c



重组的规则

重组时应该注意求值的顺序

- 指令的顺序必须遵守DAG中结点的顺序
- 对数组的赋值必须跟在所有原来在它之前的赋值/求值操作之后。
- 对数组元素的求值必须跟在所有原来在它之前的赋值指令之后
- 对变量的使用必须跟在所有原来在它之前的过程调用和指针间接赋值之后
- 任何过程调用或者指针间接赋值必须跟在原来在它之前的变量求值之后。

如果两个指令之间可能相互影响，那么它们的顺序就不应该改变。

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);
}
```

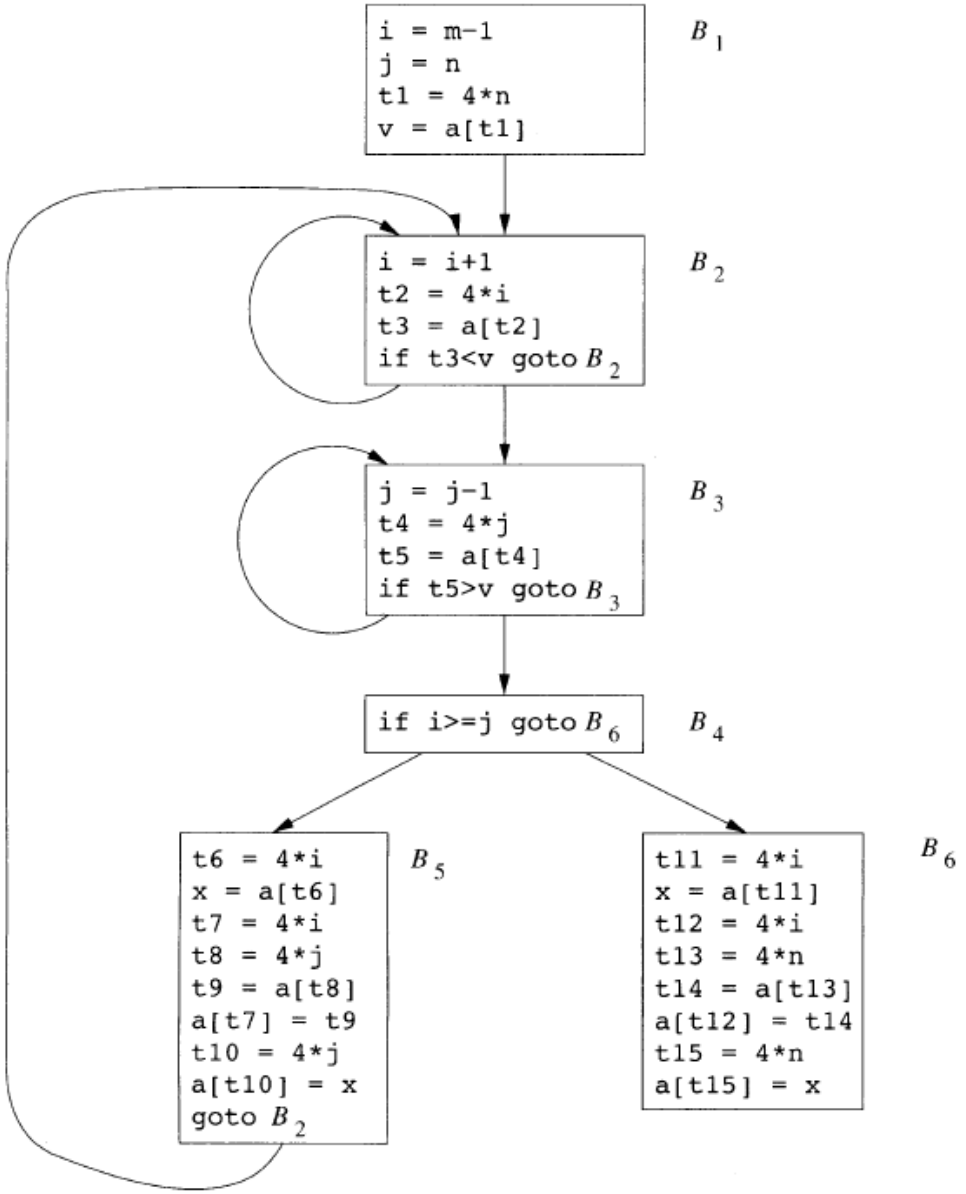
三地址代码

```
(1)     i = m-1
(2)     j = n
(3)     t1 = 4*n
(4)     v = a[t1]
(5)     i = i+1
(6)     t2 = 4*i
(7)     t3 = a[t2]
(8)     if t3<v goto (5)
(9)     j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(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)
(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
```

流图

循环：

- B_2
- B_3
- B_2、B_3、B_4、B_5



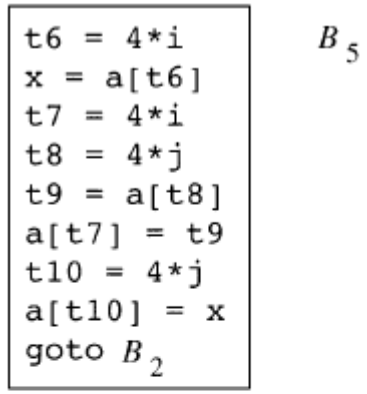
全局公共子表达式

如果E

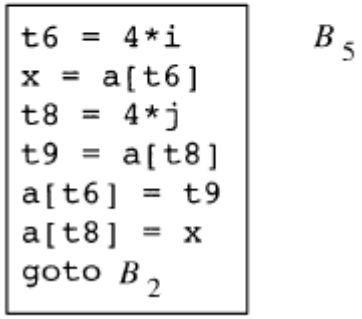
- 在某次出现之前必然已经被计算过，且
- E的分量在该次计算之后一直没有被改变，

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；



a) 消除之前



b) 消除之后

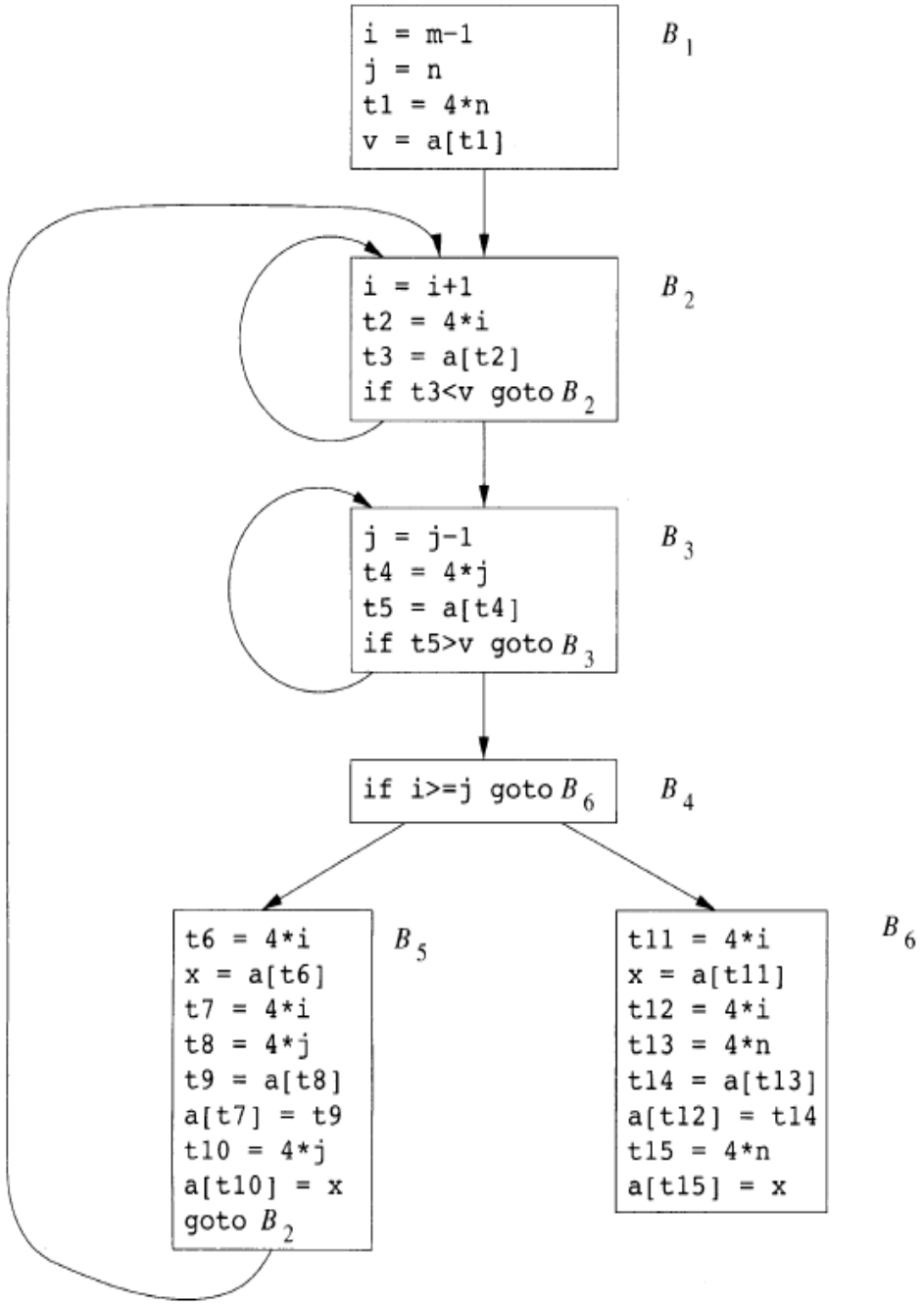
全局公共子表达式的例子

右图

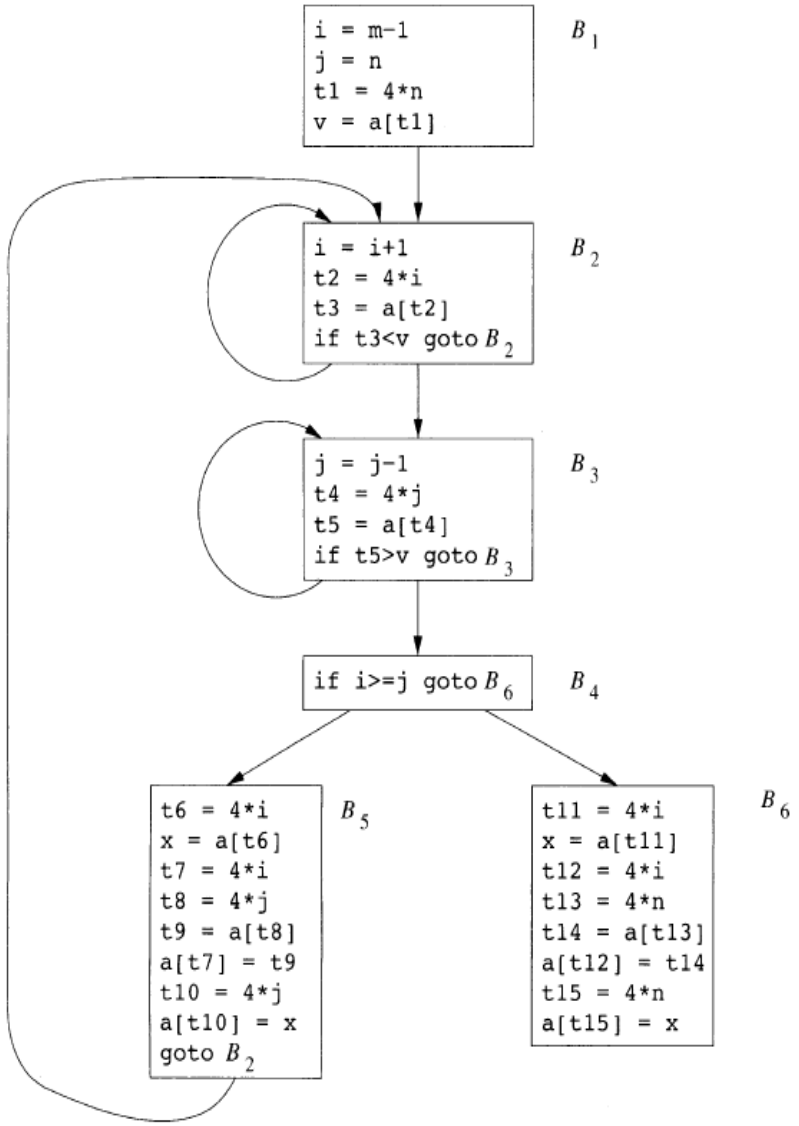
- 在B_2、B_3中计算了4*i和4*j
- 到达B_5之前必然经过B_2、B_3；
- t2、t4在赋值之后没有被改变过，因此B_5中可直接使用它们；
- t4在替换t8之后，B_5：a[t8]和B_3：a[t4]又相同；

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



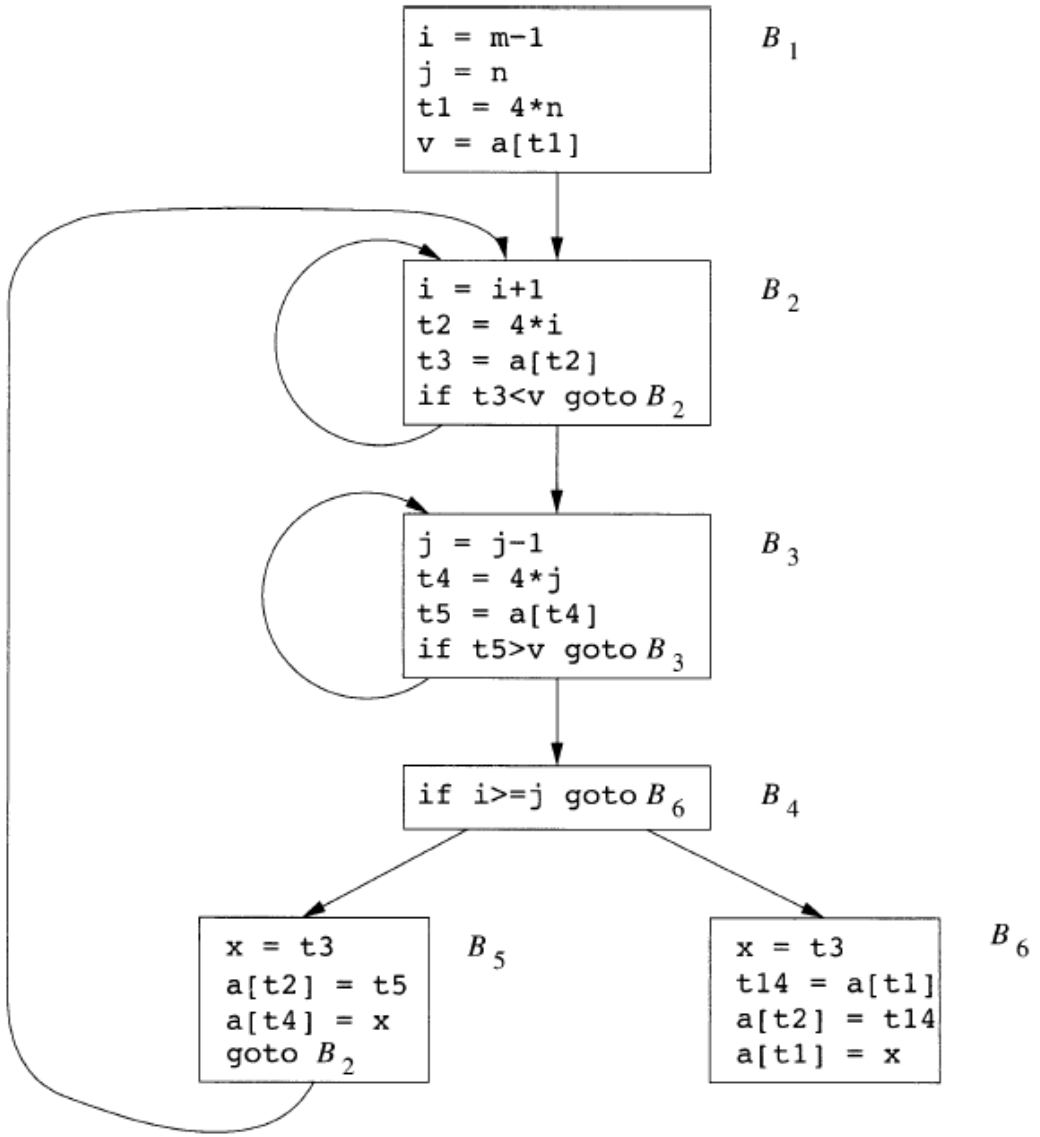
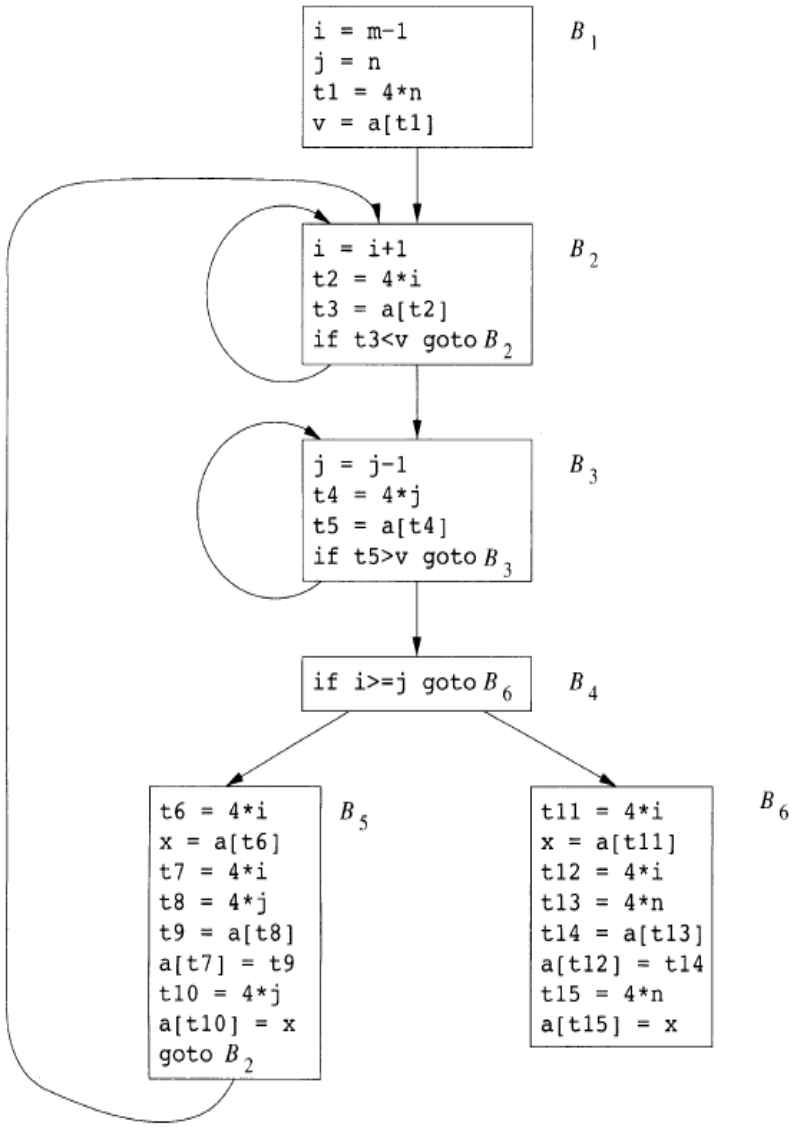


图 9-5　经过公共子表达式消除之后的 B_5 和 B_6



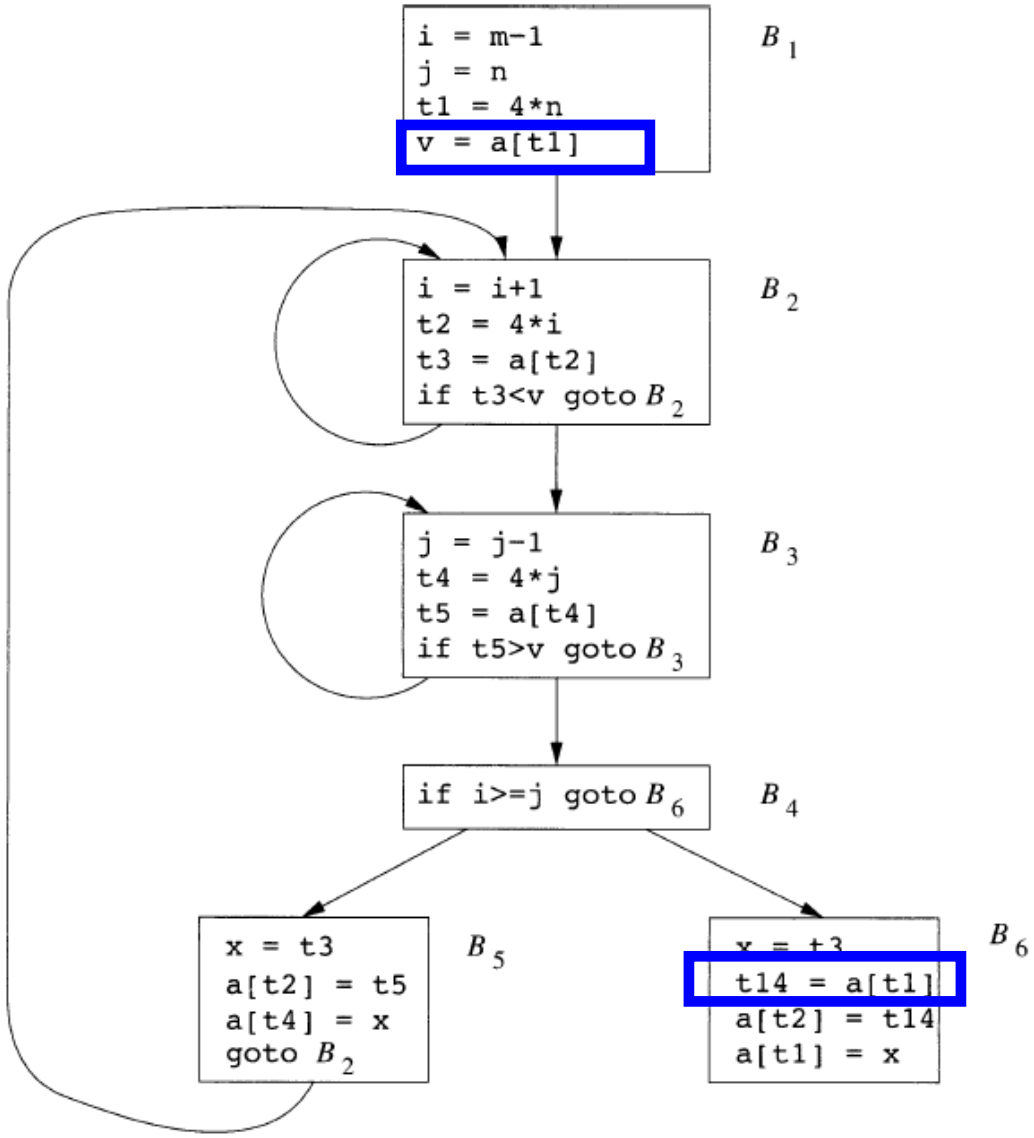


图 9-5　经过公共子表达式消除之后的 B_5 和 B_6

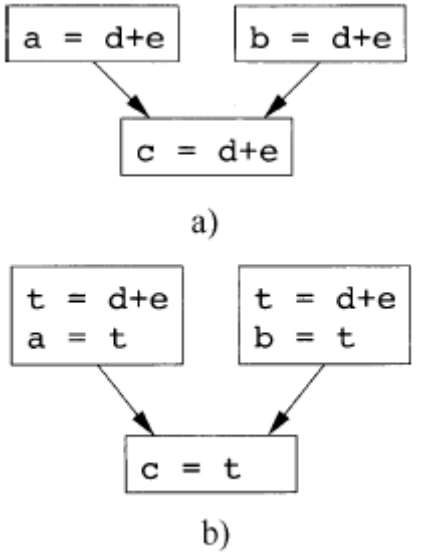
复制传播

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；

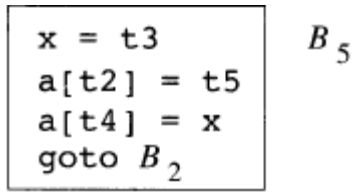
如果尽可能用t来替换c，可能就不需要c=t这个语句了。



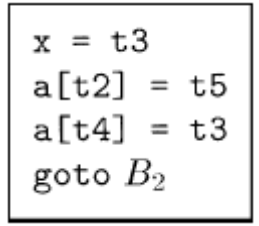
复制传播的例子

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



B_5



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                    goto B2
goto B2
```

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

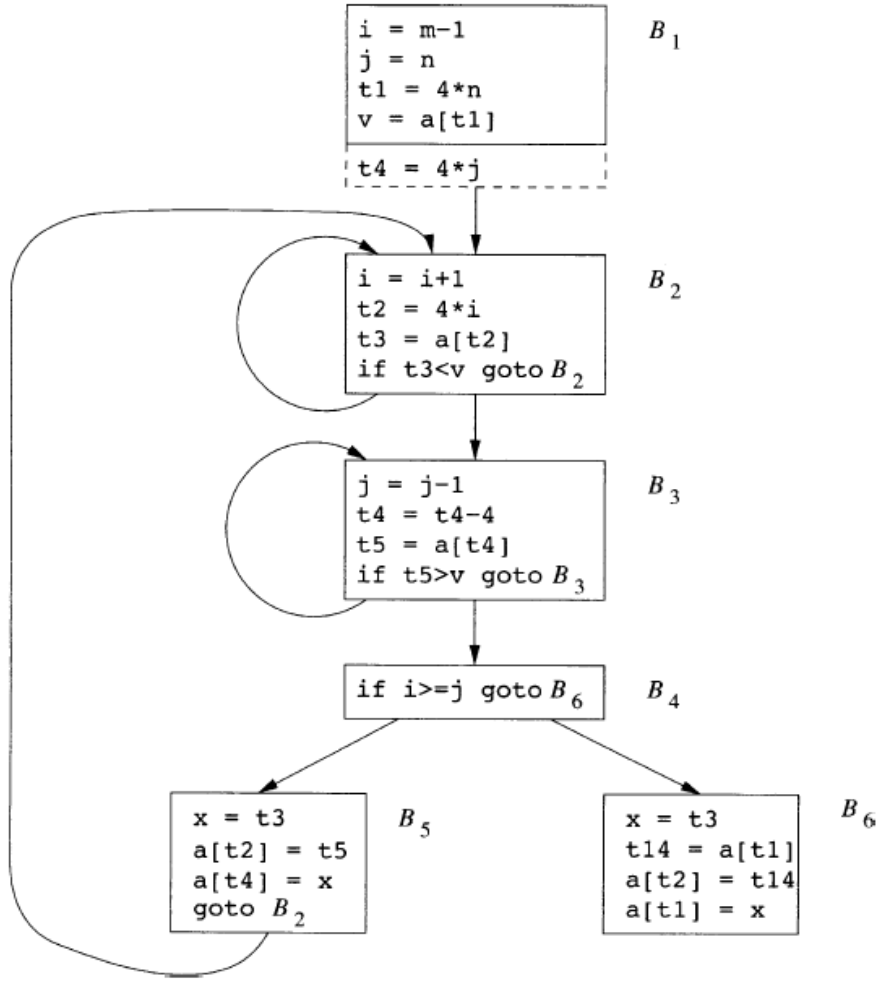


图 9-8　对基本块 B_3 中的 $4 * j$ 应用强度消减优化

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

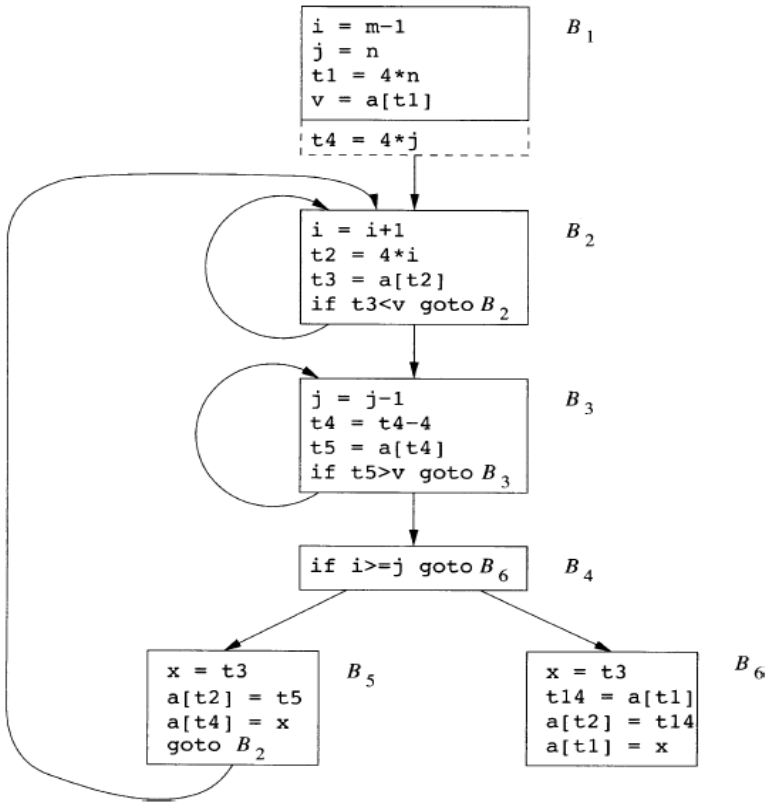


图 9-8　对基本块 B_3 中的 $4 * j$ 应用强度消减优化

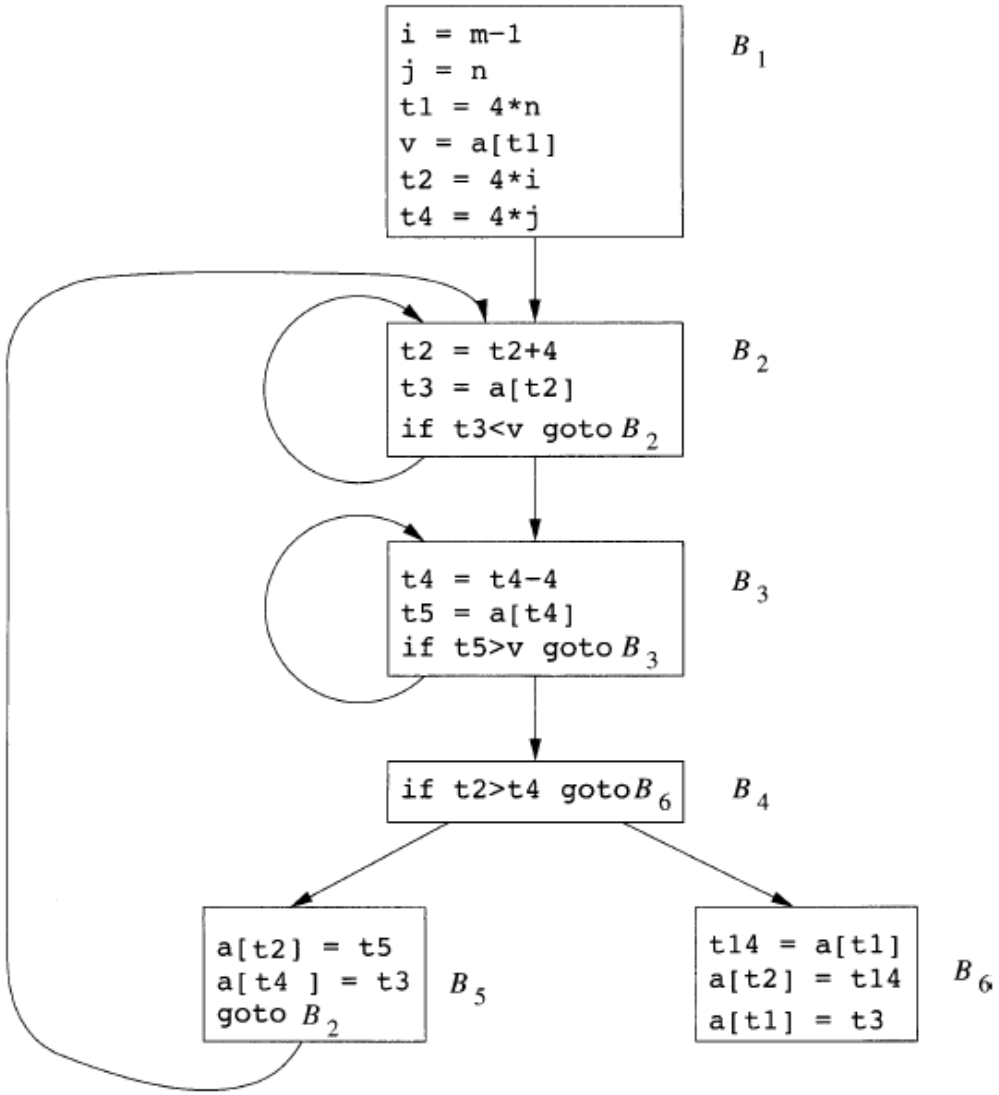
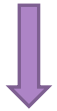


图 9-9　归纳变量消除之后的流图

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：$p_1, p_2, \ldots, p_n$

- 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<a,{1,243}>

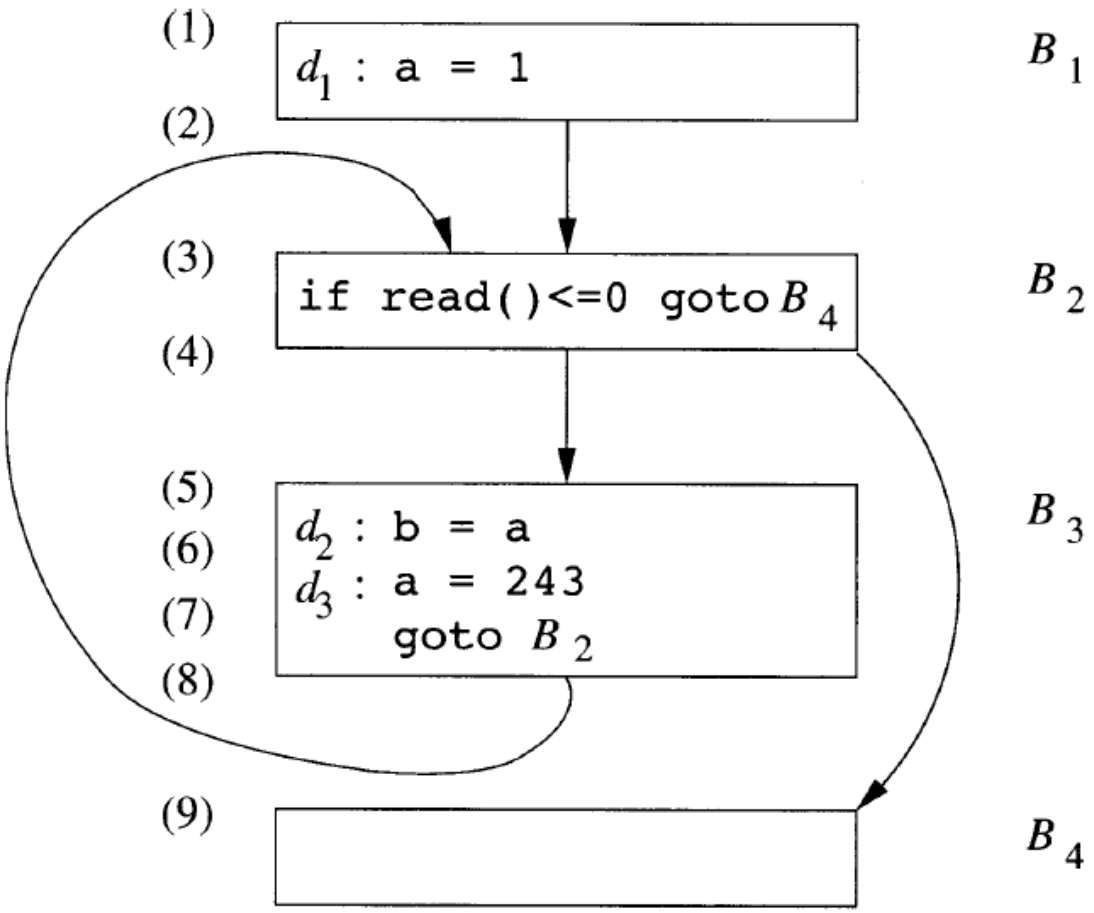


图 9-12　说明数据流抽象的例子程序

例子（1）

点(5)所有程序状态：

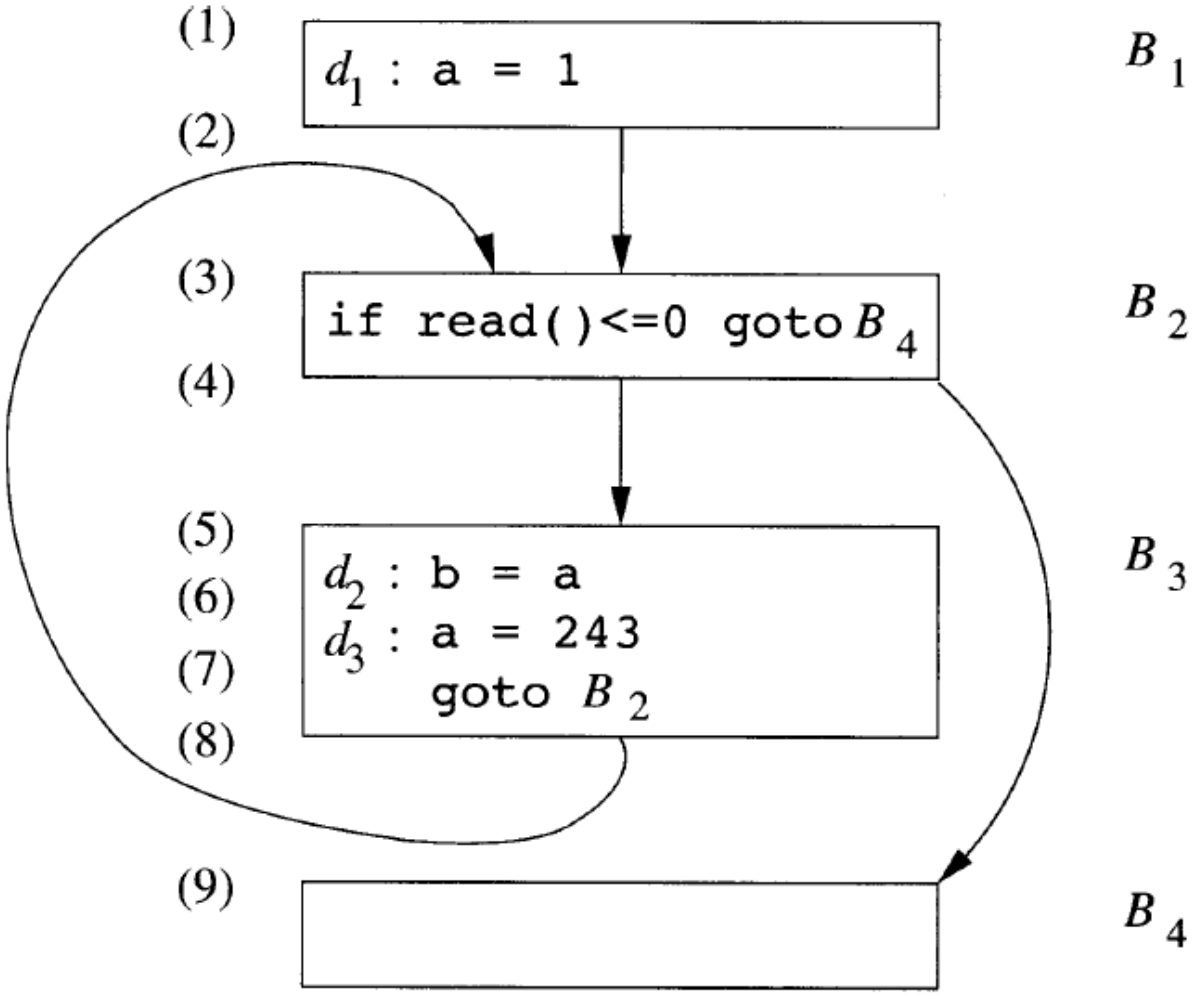
- a∈ {1, 243}
- 由{d_1, d_3}定值

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；

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

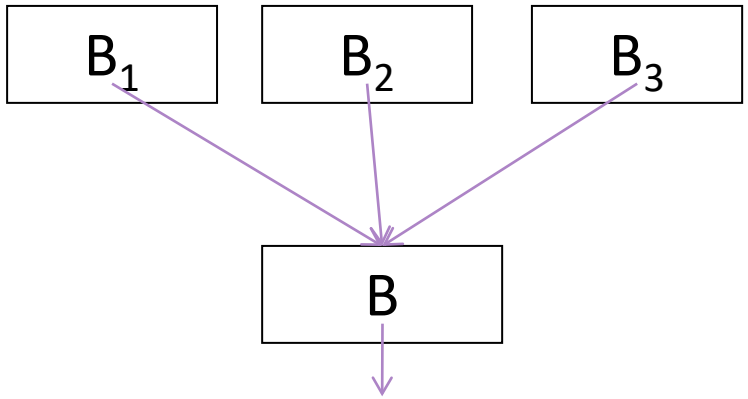
设基本块包含语句s1,s2,…,sn；

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

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

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的值就可能是由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

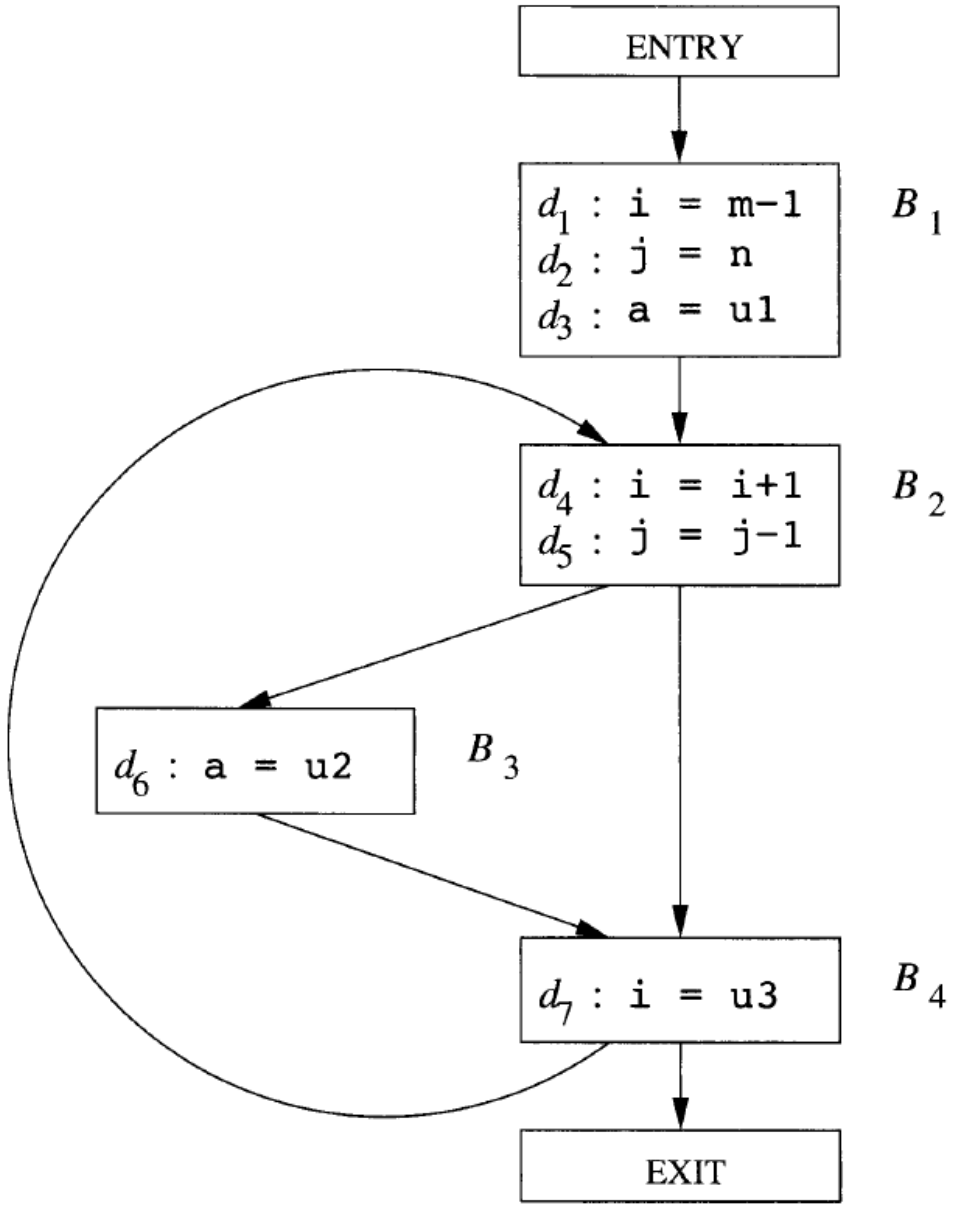
B_1中全部定值到达B_2的开头；

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}，$kill_d$={程序中其他对u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

$f_B(x)=gen_B \cup (x-kill_B)$

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

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

$kill_B$ 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

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；

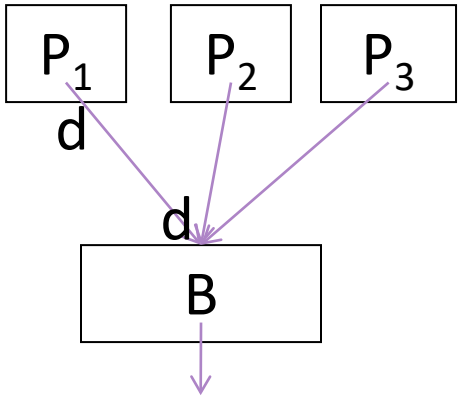
$IN[B] = \cup_{P是B的前驱基本块} OUT[P]$

- 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

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；

$$OUT[ENTRY] = 空集$$

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

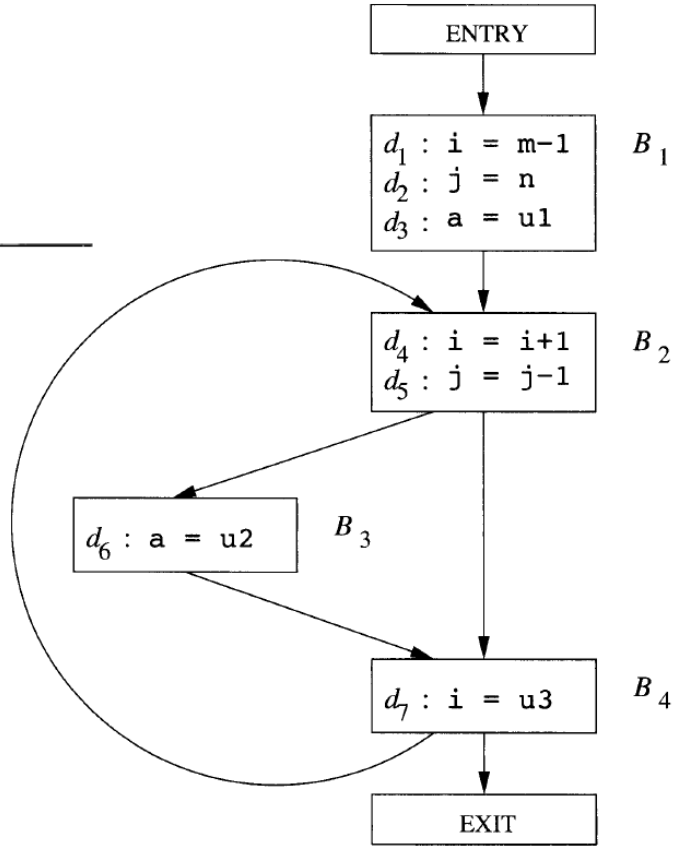
输出：IN[B]和OUT[B]

方法：

```
1)  OUT[ENTRY] = ∅;
2)  for (除 ENTRY 之外的每个基本块 B) OUT[B] = ∅;
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] = ⋃_{P是B的一个前驱} OUT[P];
6)          OUT[B] = gen_B ∪ (IN[B] − 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$d_1,d_2,\ldots,d_n$

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

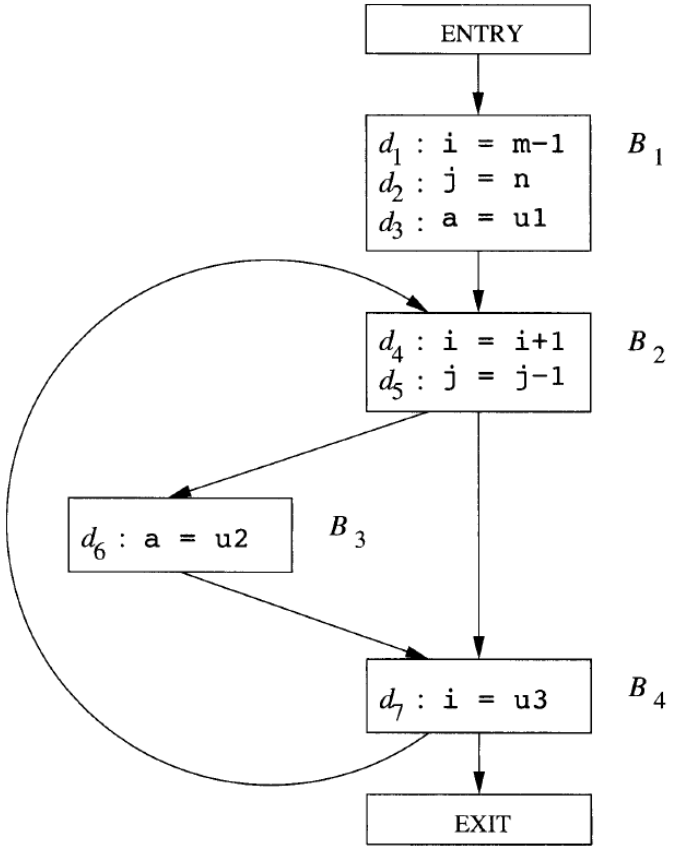
B_2生成d_4,d_5，杀死d_1,d_2,d_7

B_3生成d_6，杀死d_3

B_4生成d_7，杀死d_1,d_4

到达定值求解的例子

Block B	OUT$[B]^0$	IN$[B]^1$	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$d_1,d_2,\ldots,d_n$

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

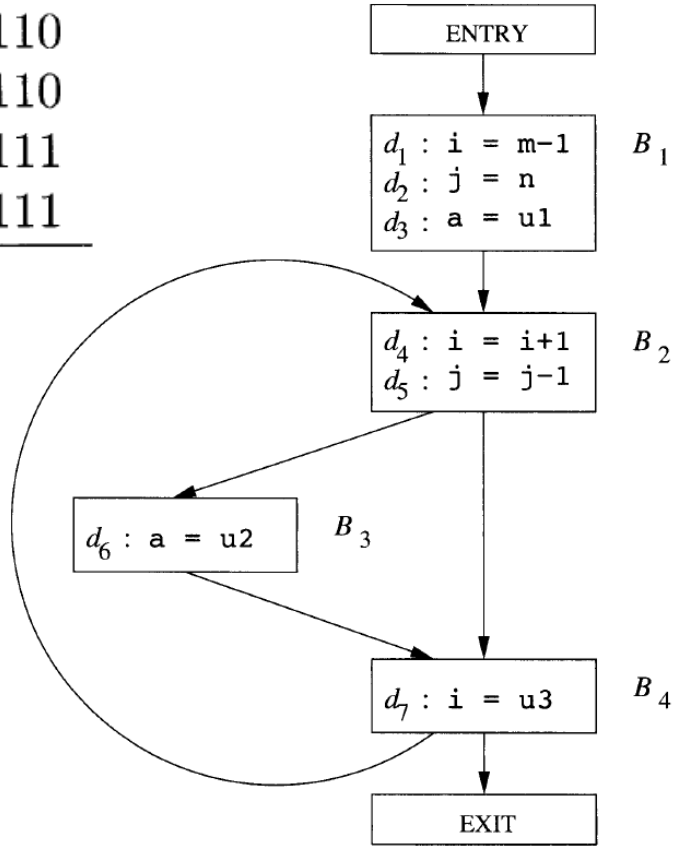
B_2生成d_4,d_5，杀死d_1,d_2,d_7

B_3生成d_6，杀死d_3

B_4生成d_7，杀死d_1,d_4

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$d_1,d_2,\ldots,d_n$

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

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

$use_B = use_1 \cup (use_2 - def_1) \cup (use_3 - def_1 - def_2) \cup (use_n - def_1 - def_2 \ldots - def_{n-1})$

$def_B = def_1 \cup def_2 \cup \ldots \cup def_n$

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

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

和到达定值相比较

- 都使用并集运算∪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

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

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($\wedge$)	$\cup$	$\cup$	$\cap$
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	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

总结

- 基本块优化
- 流图优化
  - 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）