第十章 代码优化

重点：代码优化的任务，局部优化、循环优化、全局优化的基本方法。
难点：控制流分析，数据流分析。

第10章 代码优化

第10章 代码优化

- 代码优化就是为了提高目标程序的效率，对程序进行等价变换，亦即在保持功能不变的前提下，对源程序进行合理的变换，使得目标代码具有更高的时间效率和/或空间效率。
- 空间效率和时间效率有时是一对矛盾，有时不能兼顾。
- 优化的要求：
  - 必须是等价变换(保持功能)
  - 为优化的努力必须是值得的。
  - 有时优化后的代码的效率反而会下降。

代码优化程序的结构

- 控制流分析的主要目的是分析出程序的循环结构。循环结构中的代码的效率是整个程序的效率的关键。
- 数据流分析进行数据流信息的收集，主要是变量的定义和引用情况的数据流信息。
  - 到达-定义分析；活跃变量分析；可用表达式分析.
- 代码变换：根据上面的分析,对中间代码进行等价变换.

10.1 优化的种类

机器相关性

- 机器相关优化：寄存器优化，多处理器优化，特殊指令优化，无用指令消除等。
- 机器无关优化：

优化范围

- 局部优化：单个基本块范围内的优化，常量合并优化，公共子表达式删除，计算强度削弱和无用代码删除。
- 全局优化：主要是基于循环的优化：循环不变优化，归纳变量删除，计算强度削减。

优化语言级

- 优化语言级：针对中间代码，针对机器语言。

程序例子

本节所用的例子

```
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;
```

(1) $i := m - 1$

(2) $j := n$

(3) $t_1 := 4 * n$

(4) $v := a[t_1]$

(5) $i := i + 1$

(6) $t_2 := 4 * i$

(7) $t_3 := a[t_2]$

(8) if $t_3 < v$ goto(5)

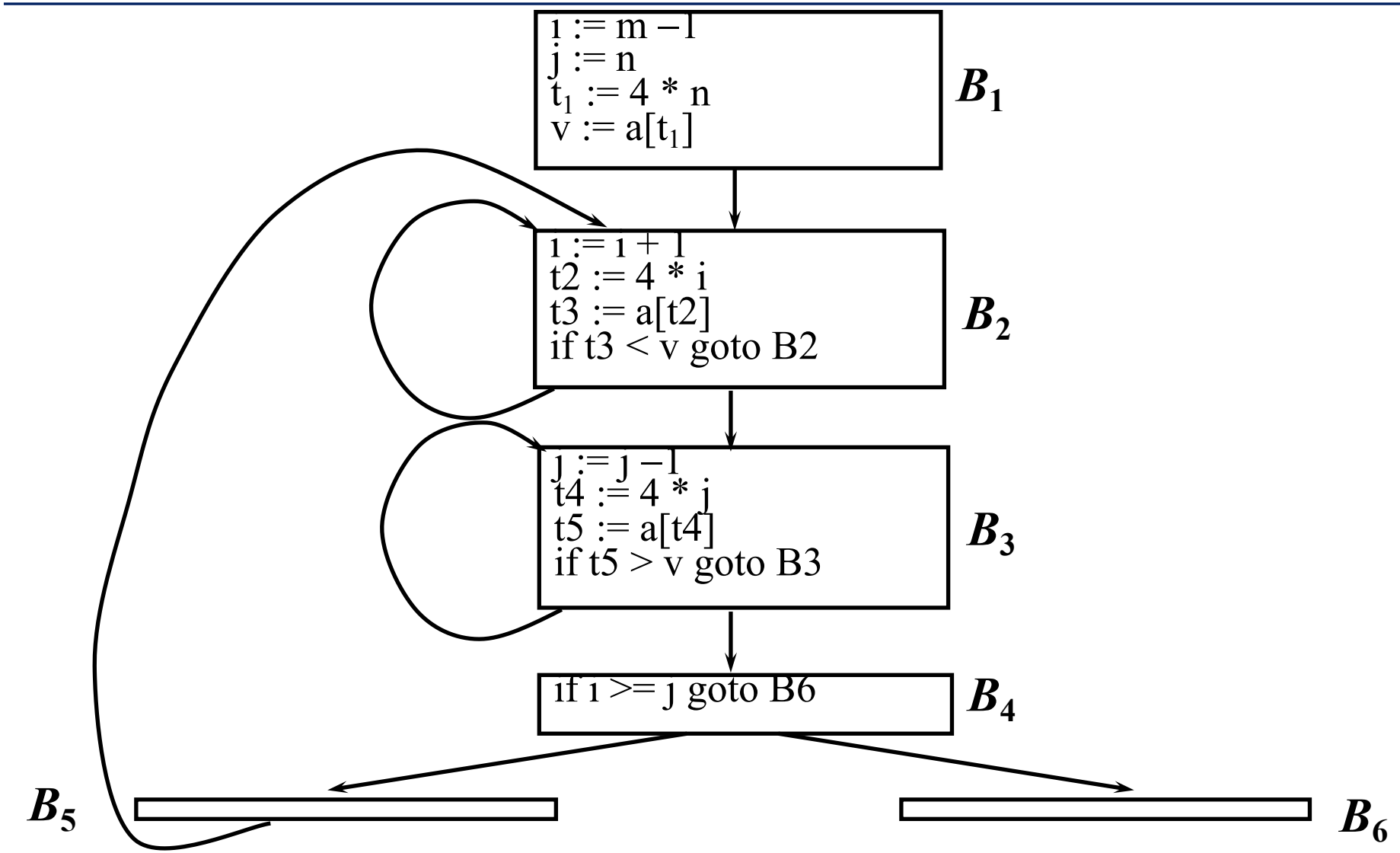
程序例子

本节所用的例子

```
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;
```

```
(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]
 . . .
```

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

10.1.1公共子表达式删除

- 局部公共子表达式

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

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

10.1.1公共子表达式删除

- 局部公共子表达式

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

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

10.1.1公共子表达式删除

- 局部公共子表达式

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

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

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

10.1.1公共子表达式删除

- 全局公共子表达式

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

```
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
```

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

10.1.1公共子表达式删除

全局公共子表达式

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

```
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
```

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

```
x := a[t2]
t9 := a[t4]
a[t2] := t9
a[t4] := x
goto B2
```

10.1.1公共子表达式删除

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

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

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

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

10.1.1公共子表达式删除

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

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

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

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

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

10.1.1公共子表达式删除

B_6 x = a[i]; a[i] = a[n]; a[n] = x;

$t_{11} := 4 * i$
$x := a[t_{11}]$
$t_{12} := 4 * i$
$t_{13} := 4 * n$
$t_{14} := a[t_{13}]$
$a[t_{12}] := t_{14}$
$t_{15} := 4 * n$
$a[t_{15}] := x$

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

10.1.1公共子表达式删除

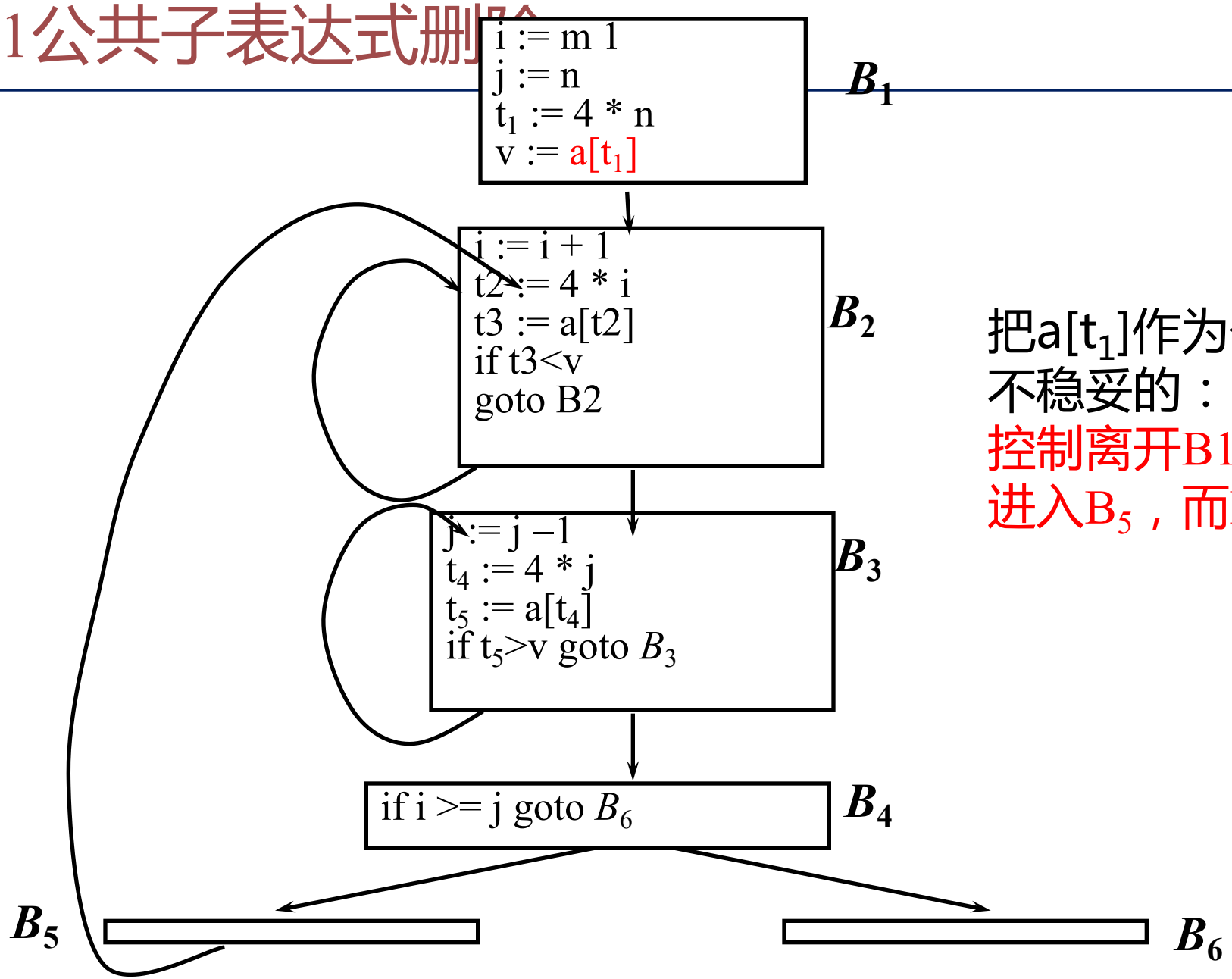
B_6 x = a[i]; a[i] = a[n]; a[n] = x;

a[t_1]能否作为公共子表达式？

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

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

10.1.1公共子表达式删除

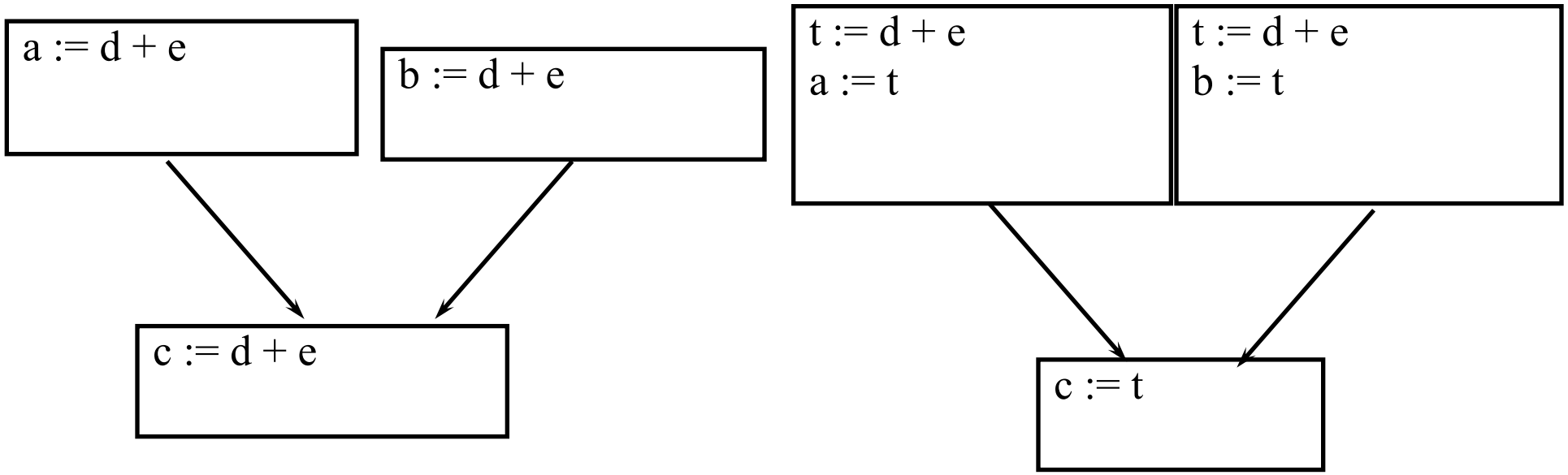


把a[t_1]作为公共子表达式是不稳妥的：

控制离开B1进入B6之前可能进入B_5，而B5有对a的赋值

10.1.2 复制传播

- 形如f := g的赋值语句叫做复制语句
- 优化过程中会大量引入复制



删除局部公共子表达式期间引进复制

10.1.2 复制传播

- 复制传播变换的思想是在复制语句f := g之后尽可能用g代替f
- 复制传播变换本身并不是优化，但它给其它优化带来机会
  - 无用代码删除

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

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

10.1.3 无用代码删除

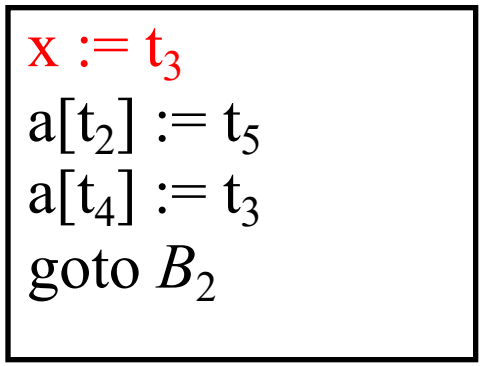
- 无用代码是指计算结果以后不被引用的语句
- 一些优化变换可能会引入无用代码
- 例：

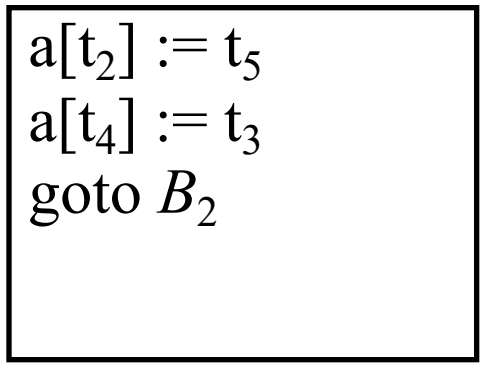
```
debug := true;                    debug := false;
    . . .          测试后改成         . . .
if(debug)print …                  if(debug)print …
```

10.1.3 无用代码删除

- 无用代码是指计算结果以后不被引用的语句
- 一些优化变换可能会引入无用代码

例：复制传播可能会引入无用代码





10.1.4 代码外提

- 结果独立于循环执行次数的表达式称为循环不变计算。如果将循环不变计算从循环中移出到循环的前面，将会减少循环执行的代码总数，大大提高代码的执行效率。这种与循环有关的优化方法称为代码外提。
- 例如，下面的while语句中，limit-2就是循环不变计算。
  - while(i <=limit-2) {/*假设循环体中的语句不改变limit的值*/}
  - 代码外提将生成如下的等价语句：
  - t := limit-2;
  - while (i <= t) {/*假设循环体中的语句不改变limit或t*/}

10.1.5 强度削弱

- 实现同样的运算可以有多种方式。用计算较快的运算代替较慢的运算。
- x^2 变成 x*x。
- 2*x或2.0*x 变成 x+x
- x/2 变成 x*0.5
- $a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$ 变成

  $((\ldots(a_nx+a_{n-1})x+ a_{n-2})\ldots)x+a_1)x+a_0$

10.1.5 强度削弱

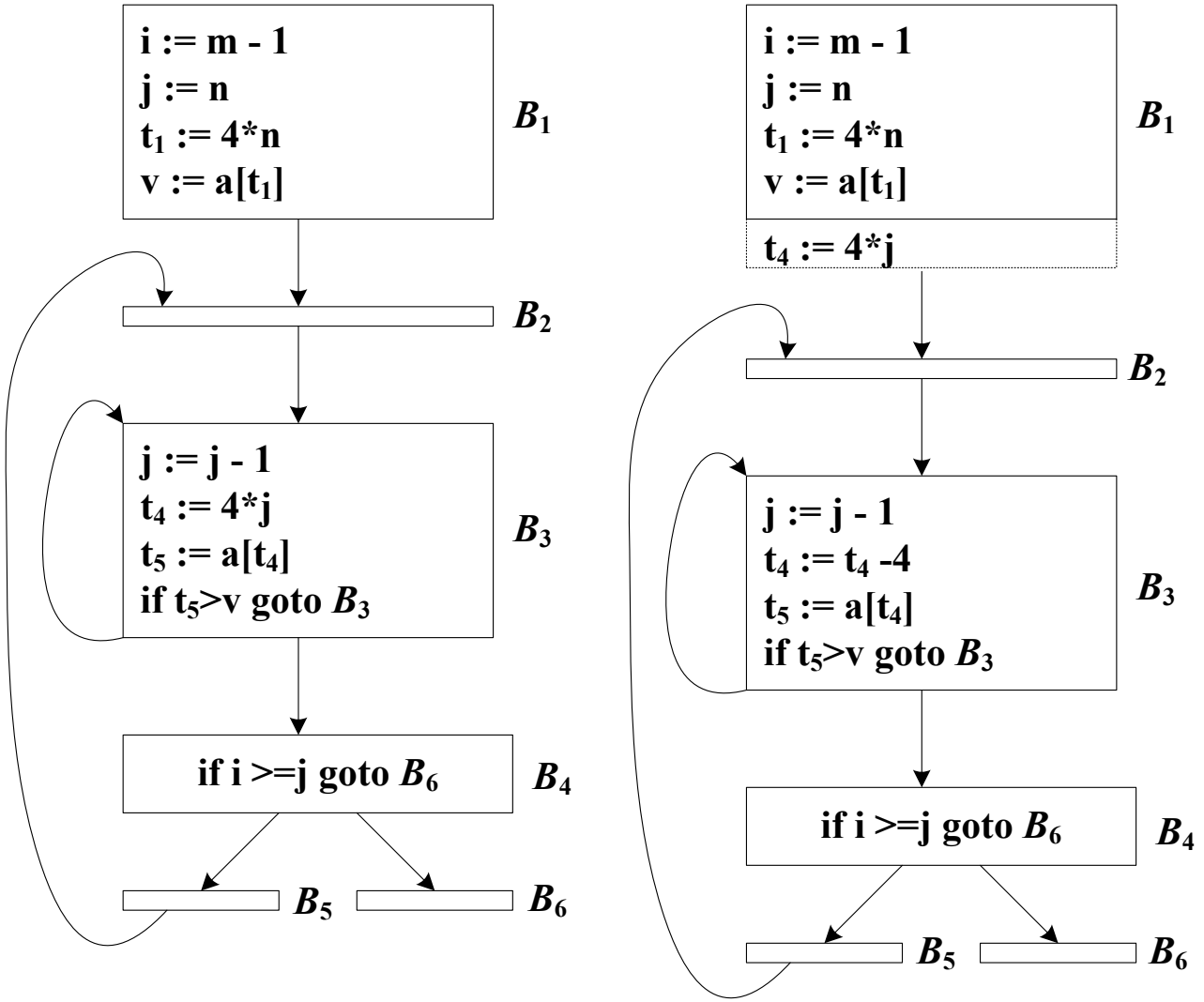


图10.6 将强度削弱应用到块B_3中的4*j

10.1.5 归纳变量删除

- 在围绕B_3的循环的每次迭代中，j和t_4的值总是同步变化，每次j的值减1，t_4的值就减4，这是因为4*j赋给了t_4，我们将这样的变量称为归纳变量
- 如果循环中存在两个或更多的归纳变量，也许可以只保留一个，而去掉其余的，以便提高程序的运行效率。

10.1.5 归纳变量删除

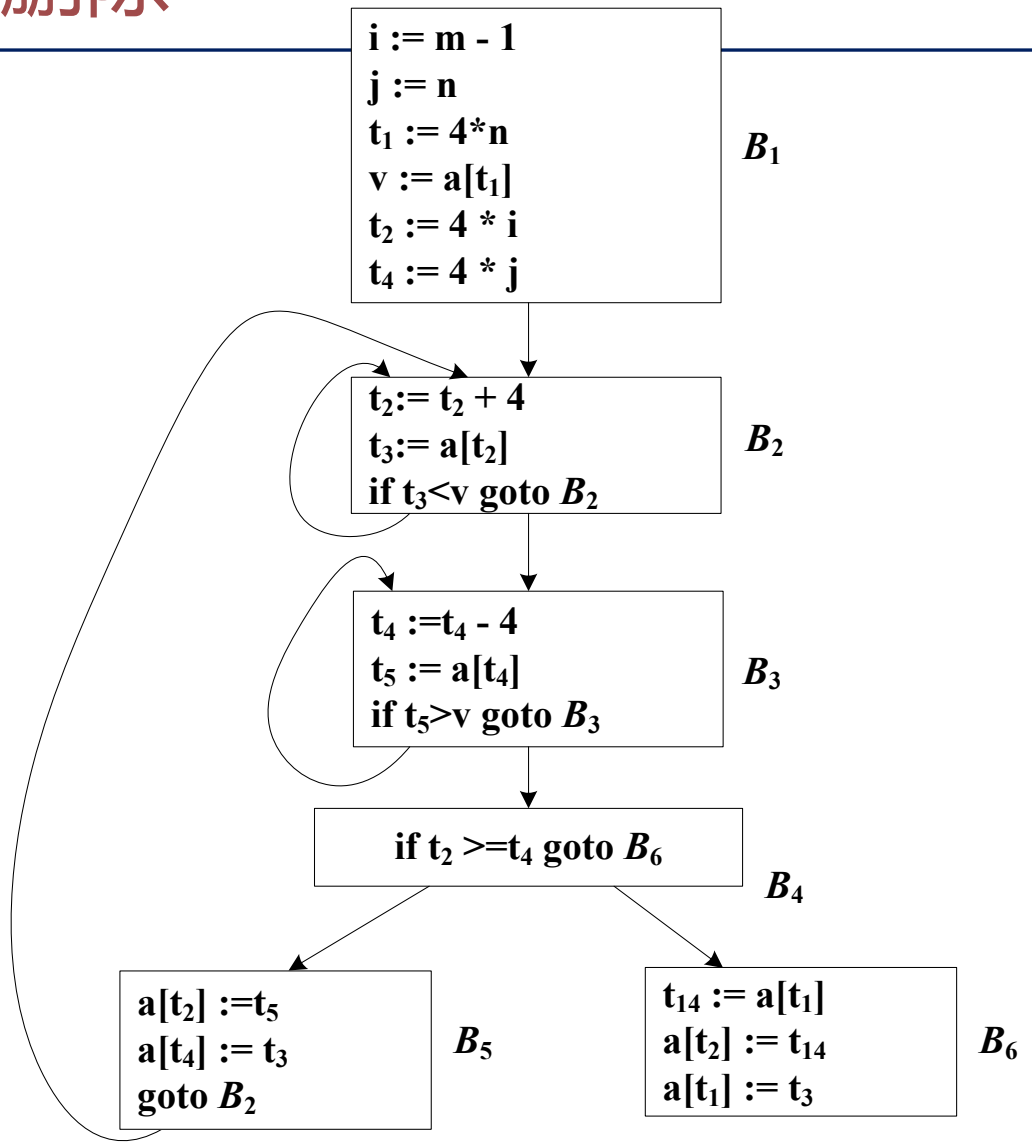


图10.7 归纳变量删除后的流图

10.2 控制流分析

- 为了对程序进行优化，尤其是对循环进行优化，必须首先分析程序中的控制流程，以便找出程序中的循环结构，这是控制流分析的主要任务。
- 为此，首先需要将程序划分为基本块集合并转换成流图，然后再从流图中找出循环。

10.2.1 基本块

- 基本块(basic block)是一个连续的语句序列，控制流从它的开始进入，并从它的末尾离开，中间不存在中断或分支(末尾除外)。下面的三地址码序列就形成了一个基本块：

  - $t_1 := a*a$
  - $t_2 := a*b$
  - $t_3 := 2*t_2$
  - $t_4 := t_1+t_3$
  - $t_5 := b*b$
  - $t_6 := t_4+t_5$

基本块的划分算法

算法10.1 基本块的划分

- 输入：一个三地址码序列；
- 输出：一个基本块列表，其中每个三地址码仅在一个块中；
- 步骤：

1. 首先确定所有的入口(leader)语句，即基本块的第一个语句。确定入口的规则如下：
   a) 第一个语句是入口语句；
   b) 任何能由条件转移语句或无条件转移语句转移到的语句都是入口语句；
   c) 紧跟在转移语句或条件转移后面的语句是入口语句。
2. 对于每个入口语句，其基本块由它和直到下一个入口语句(但不含该入口语句)或程序结束为止的所有语句组成。

10.2.2 流图

- 程序的控制流信息可以用流图表示，流图是一个节点为基本块的有向图。
- 以程序的第一个语句作为入口语句的节点称为初始节点。
- 如果在某个执行序列中B_2跟随在B_1之后，则从B_1到B_2有一条有向边。
- 如果从B_1的最后一条语句有条件或无条件转移到B_2的第一个语句；或者按程序正文的次序B_2紧跟在B_1之后，并且B_1不是结束于无条件转移，则称B_1是B_2的前驱，而B_2是B_1的后继。

10.2.3 循环

- 流图中的循环就是具有唯一入口的强连通子图，而且从循环外进入循环内，必须首先经过循环的入口节点。
- 如果从流图的初始节点到节点n的每条路径都要经过节点d，则说节点d支配(dominate)节点n，记作$d\ dom\ n$，d又称为n的必经节点。
- 根据该定义，每个节点都支配它自身，而循环的入口节点则支配循环中的所有节点。

支配节点集计算

算法10.2 支配节点集计算

- 输入：流图G，其节点集为N，边集为E，初始节点为n_0
- 输出：关系dom；
- 步骤：

1．$D(n_0) := \{n_0\}$;

2．for $N-\{n_0\}$中的n do $D(n) := \varnothing$;

/* 初始化完毕 */

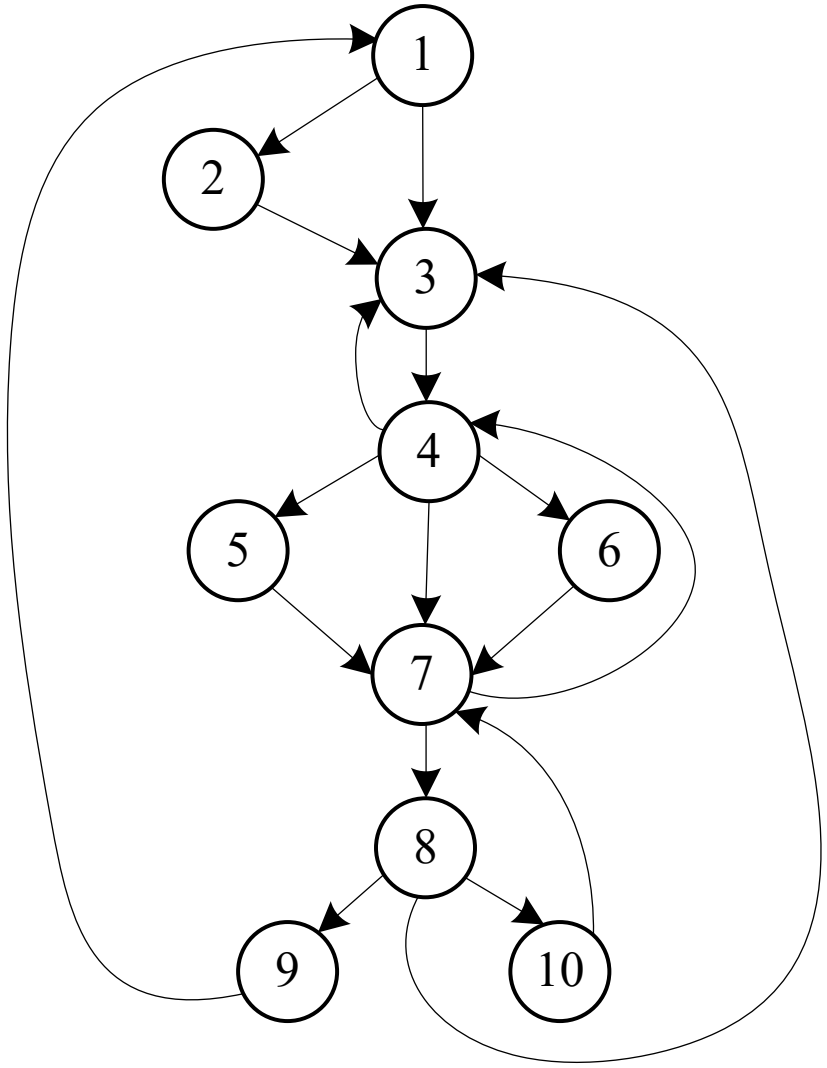
3．while $D(n)$发生变化 do

4． for $N-\{n_0\}$中的n do

5． $D(n) := \{n\} \cup \bigcap_{n\text{的前驱}p} D(p)$;

例10.6

- $D(1)=\{1\}$
- $D(2)=\{1,2\}$
- $D(3)=\{1,3\}$
- $D(4)=\{1,3,4\}$
- $D(5)=\{1,3,4,5\}$
- $D(6)=\{1,3,4,6\}$
- $D(7)=\{1,3,4,7\}$
- $D(8)=\{1,3,4,7,8\}$
- $D(9)=\{1,3,4,7,8,9\}$
- $D(10)=\{1,3,4,7,8,10\}$

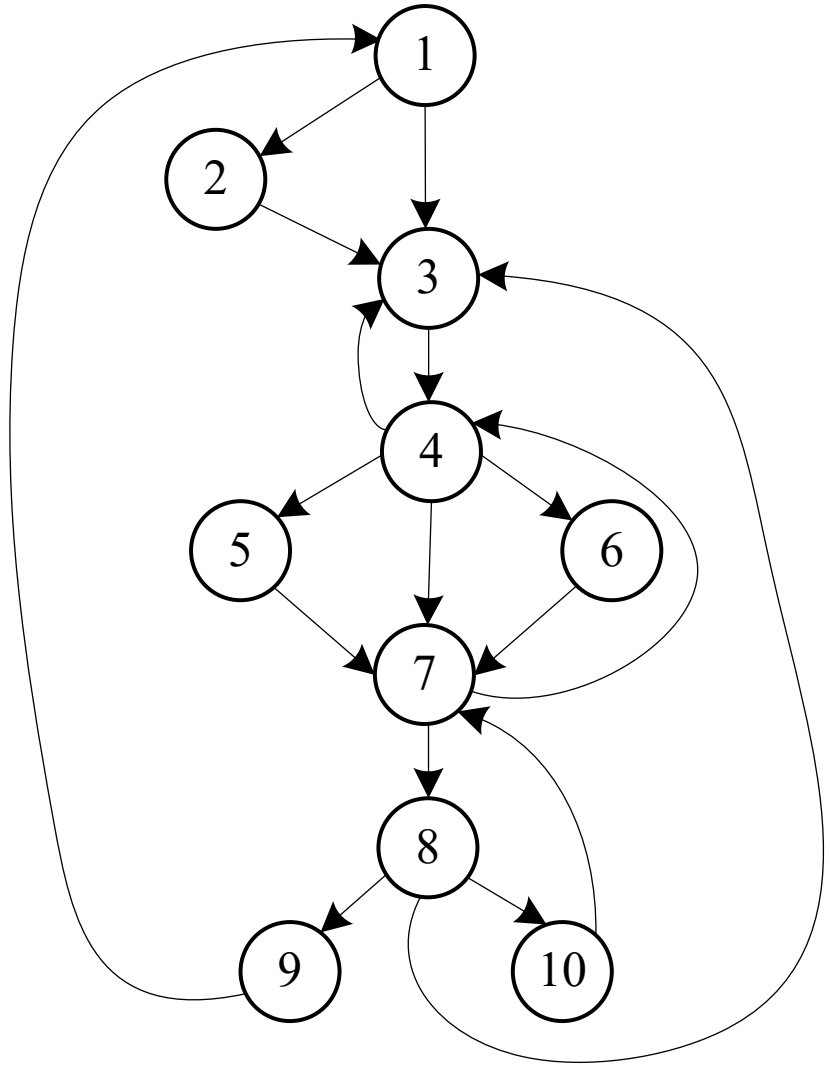


循环

- 易于被优化的(自然)循环的性质：
  - 循环必须有唯一的入口点，称为首节点(header)。首节点支配循环中的所有节点。
  - 循环至少迭代一次，亦即至少有一条返回首节点的路径。
- 为了寻找流图中的循环，必须寻找可以返回到循环入口节点的有向边，这种边叫做回边(back edge)，如果$b\ dom\ a$，则边$a \rightarrow b$称为回边。 利用支配节点集可以求出流图中的所有回边。
- 给定一条回边$n \rightarrow d$，定义该边的自然循环(natural loop)为d加上所有不经过d而能到达n的节点集合。d是该循环的首节点。

例10.6

- 3 dom 4
  - 回边4→3
- 4 dom 7
  - 回边7→4
- 10→7的自然循环{7，8，10}
- 7→4的自然循环{4,5,6,7,8,10}
- 4→3，8→3的自然循环{3,4,5,6,7,8,10}



自然循环的构造

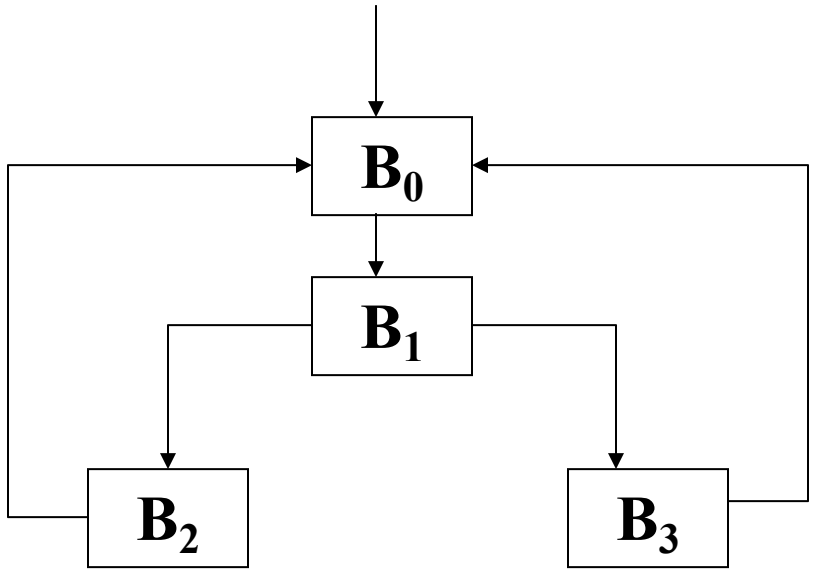
算法10.3 构造回边的自然循环

- 输入：流图G和回边$n \rightarrow d$；
- 输出：由$n \rightarrow d$的自然循环中所有节点构成的集合$loop$；
- 步骤：

```
procedure insert(m);
if m不在loop中 then begin    loop := loop ∪ {m};
将m压入栈stack    end;
/* 下面是主程序 */
stack := 空;    loop := {d};    insert(n);
while stack非空 do begin
从stack弹出第一个元素m;
for m的每个前驱p do insert(p)
end
```

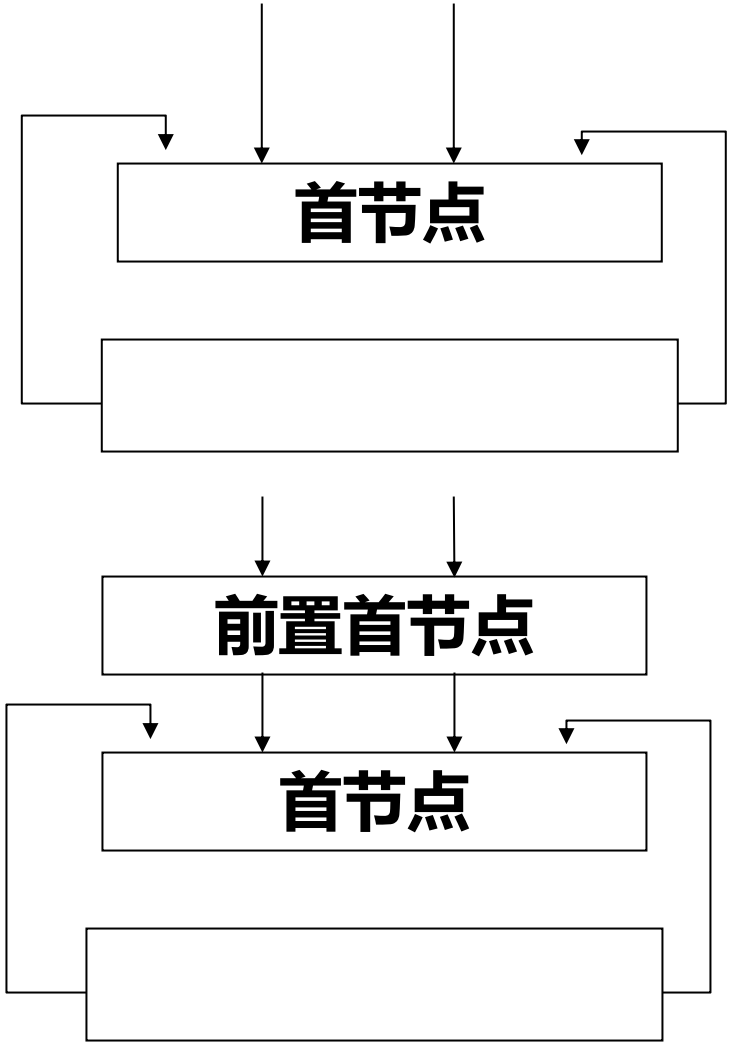
循环

- 如果我们只将自然循环当作“循环”，则循环或者互不相交，或者一个在另外一个里面。
- 内循环：不包含其他循环的循环称为内循环。
- 如果两个循环具有相同的首节点，那么很难说一个包含另外一个。此时把两个循环合并。



循环

- 某些变换要求我们将循环中的某些语句移到首节点的前面。因此，在开始处理循环L之前，往往需要先创建一个称为前置首节点的新块。前置首节点的唯一后继是L的首节点，原来从L外到达L首节点的边都将改成进入前置首节点，从循环L里面到达首节点的边不改变。
- 初始时前置首节点为空，但对L的变换可能会将一些语句放到该节点中。



可归约流图

- 可归约流图：一个流图*G*是可约的，当且仅当可以把它的边分成两个不相交的组，其中一组仅含回边，另一组的边都不是回边，这些边被称为前向边，所有前向边形成一个无环有向图，在该图中，每个节点都可以从*G*的初始节点到达。
- 循环的可约流图具有一个重要的性质，就是当流图中的一些节点被看作是循环的节点集合时，这些节点之间一定包含一条回边。于是，为了找出流图可约程序中的所有循环，只要检查回边的自然循环即可。

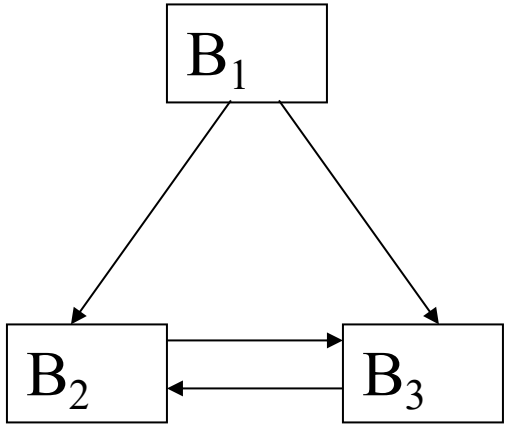


图10.10 一个不可约流图

10.3 数据流分析

- 为了进行代码优化，编译器必须掌握程序中变量的定义和引用情况，有了这些信息才能对源程序进行合适的等价变换。
  - 例如，了解每个基本块的出口处哪些变量是活跃的可以改进寄存器的利用率，执行常量合并和无用代码删除则需要利用变量的“到达-定义”信息。
- 对程序中变量的定义和引用关系的分析称为数据流分析

数据流分析的种类

- 到达-定义分析
- 活跃变量分析
- 可用表达式分析

10.3.1 数据流方程的一般形式

- 下面的方程叫做数据流方程：

  - $out[S] = gen[S] \cup (in[S] - kill[S])$ (10.3)

    该方程的含义是，当控制流通过一个语句S时，在S末尾得到的信息($out[S]$)或者是在S中产生的信息($gen[S]$)，或者是进入S开始点时携带的、并且没有被S注销的那些信息($in[S]$表示进入S开始点时携带的信息，$kill[S]$表示被S注销的信息)。

  - 也可以根据$out[S]$来定义$in[S]$

    $in[S] = (out[S] - kill[S]) \cup gen[S]$ (10.4)

10.3.1 数据流方程的一般形式

- 不同的问题方程的意义可能有所不同，主要可由以下两点来区别：
  - 信息流向问题。根据信息流向可以将数据流分析问题分为正向和反向两类，正向的含义是根据*in*集合来计算*out*集合，反向则是从*out*集合来计算*in*集合。
  - 聚合操作问题。所谓聚合操作，是指当有多条边进入某一基本块B时，由B的前驱节点的*out*集计算$in[B]$时采用的集合操作(并或交)。到达-定义等方程采用并操作，全局可用表达式采用的则是交操作。

10.3.1 数据流方程的一般形式

- 在基本块中，将相邻语句间的位置称为点，第一个语句前和最后一个语句后的位置也称为点。从点p_1到点p_n的路径是这样的点序列p_1，p_2,…，p_n,对于$\forall i(1\leq i\leq n\text{-}1)$下列条件之一成立：
  - p_i是紧接在一个语句前面的点，p_{i+1}是同一块中紧跟在该语句后面的点；
  - p_i是某基本块的结束点，而p_{i+1}是后继块的开始点。
- 为简单起见，假设控制只能从一个开始点进入基本块，而当基本块结束时控制只能从一个结束点离开。

10.3.2 到达-定义分析

- 变量x的定义是一条赋值或可能赋值给x的语句。最普通的定义是对x的赋值或从I/O设备读一个值并赋给x的语句，这些语句比较明确地为x定义了一个值，称为x的明确定义。也有一些语句只是可能为x定义一个值，称为x的含糊定义，其常见形式有：
  - 1．以x为参数的过程调用(传值方式除外)或者可能访问x的过程；
  - 2．通过可能指向x的指针对x赋值。

10.3.2 到达-定义分析

- 对于定义*d*，如果存在一条从紧跟*d*的点到*p*的路径，并且在这条路径上*d*没有被“注销”，则称定义*d*到达(reach)点*p*。
- 如果沿着这条路径的某两点间存在a的其它定义，则将注销(kill)变量a的那个定义。注意，只有a的明确定义才能注销a的其它定义。
- 到达定义信息可以用引用-定义链(即ud-链)来保存，它是一个链表，对于变量的每次引用，到达该引用的所有定义都保存在该链表中。

10.3.2 到达-定义分析

- 如果块B中在变量a的引用之前没有任何a的明确定义，那么a的这次引用的ud-链为$in[B]$中a的定义的集合。如果B中在a的这次引用之前存在a的明确定义，那么只有a的最后一次定义会在ud-链中，而$in[B]$不能放在ud-链中
- 另外，如果存在a的含糊定义，那么所有那些在该定义和a的这次引用之间没有a的明确定义的定义都将被放在a的这次引用的ud-链中
- 利用ud-链可以求出循环中的所有循环不变计算，常量传播也需要用到ud-链信息

到达定义数据流方程(记号)

- in[B]: 表示基本块B的入口点处各个变量的定义集合。
  - 如果B中点p之前有x的定义d，且这个定义能够到达p，则点p处x的ud链是$\{d\}$。
  - 否则，点p处x的ud链就是in[B]中x的定义集合。
- $P[B]$：B的所有前驱基本块的集合。

到达定义数据流方程(记号)

- gen[B]：各个变量在B内定义，并能够到达B的出口点的所有定义的集合。
- out[B]：各个变量的能够到达基本块B的出口点的所有定义的集合。
- kill[B]：各个变量在基本块B中重新定义，即在此块内部被注销的定义点的集合。

到达定义数据流方程

- in[B] = ∪out[p] where p is in P[B]
- out[B] = gen[B] ∪ (in[B]-kill[B])
- 其中：
  - gen[B]可以从基本块中求出：使用DAG图就可以得到。
  - kill[B]中，对于整个流图中的所有x的定义点，如果B中有对x的定义，那么该定义点在kill[B]中。

方程求解算法

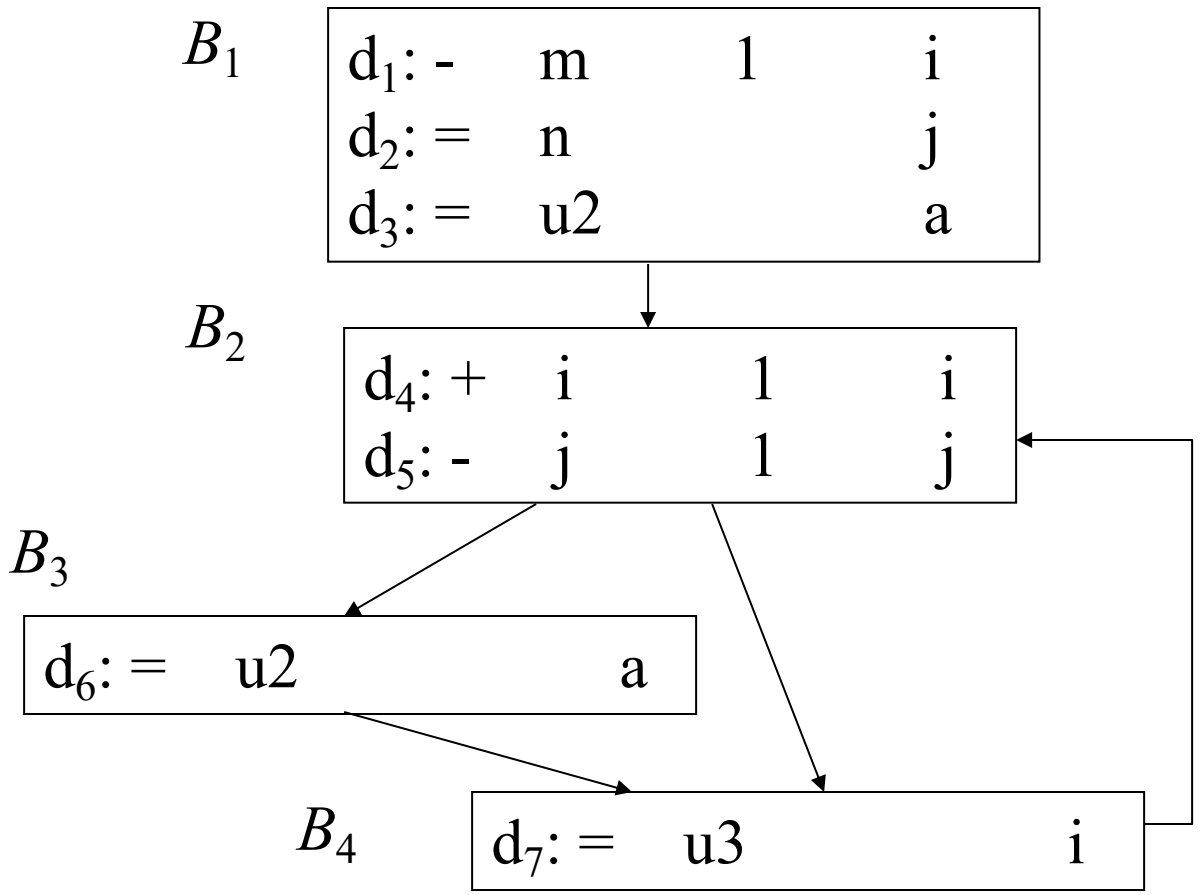
- 使用迭代方法。

初始值设置为：in[B_i]=空；out[B]=gen[B_i];

```
change = true;
while(change)
{  change = false;
   for each B do
          {in[B] = ∪ out[p] where p is in P[B];
           out[B] = gen[B] ∪ (in[B]-kill[B]);
           oldout = out[B];
           if(out[B] != oldout) change = true;}
}
```

算法例子

- 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}



计算过程

初始化：

- in[B_1] =in[B_2] =in[B_3] = in[B_4] =空
- out[B_1]={d_1,d_2,d_3}, out[B_2]={d_4,d_5}
- out[B_3]={d_6}, out[B_4]={d_7}.

第一次循环：

- in[B_1]=空; in[B_2] ={d_1,d_2,d_3,d_7};
  in[B_3]={d_4,d_5};
- in[B_4]={d_4,d_5,d_6}; out[B_1]={d_1,d_2,d_3};
- out[B_2]={d_3,d_4,d_5}…

结果：

- in[B1]=空； out [B1]={d1,d2,d3};
- in[B2]={d1,d2,d3,d5,d6,d7}; out[B2]={d3,d4,d5,d6};
- in[B3]={d3,d4,d5,d6}; out [B3]={d4,d5,d6};
- in[B4]={d3,d4,d5,d6}; out[B4]={d3,d5,d6,d7};

10.3.3 活跃变量分析

- 对于变量x和点p，在流图中沿从p开始的某条路径，是否可以引用x在p点的值。如果可以则称x在p点是活跃的，否则，x在p点就是无用的。
- 活跃变量信息在目标代码生成时具有重要的作用。当我们在寄存器中计算一个值之后，通常假设在某个块中还要引用它，如果它在该块的末尾是无用的，则不需要存储该值。
- 消除复制四元式的依据也是对活跃变量的分析。如果某个变量的值在以后不被引用，那么该复制四元式可以被消除。

记号

- in[B]: 基本块B的入口点的活跃变量集合。
- out[B]: 是在基本块B的出口点的活跃变量集。
- def[B]: 是在基本块b内的定义，但是定义前在B中没有被引用的变量的集合。
- use[B]: 表示在基本块中引用，但是引用前在B中没有被定义的变量集合。
- 其中，def[B]和use[B]是可以从基本块B中直接求得的量，因此在方程中作为已知量。

活跃变量数据流方程

- in[B] = use[B] ∪(out[B]-def[B])
- out[B] = ∪ in[S]，其中S为B的所有后继。
- 变量在某点活跃，表示变量在该点的值在以后会被使用。
- 第一个方程表示：
  - 一个变量在进入块B时是活跃的，如果它在该块中于定义前被引用
  - 一个变量在离开块B时是活跃的，而且在该块中没有被重新定义。
- 第二个方程表示：
  - 一个变量在离开块B时是活跃的，当且仅当它在进入该块的某个后继时是活跃的。

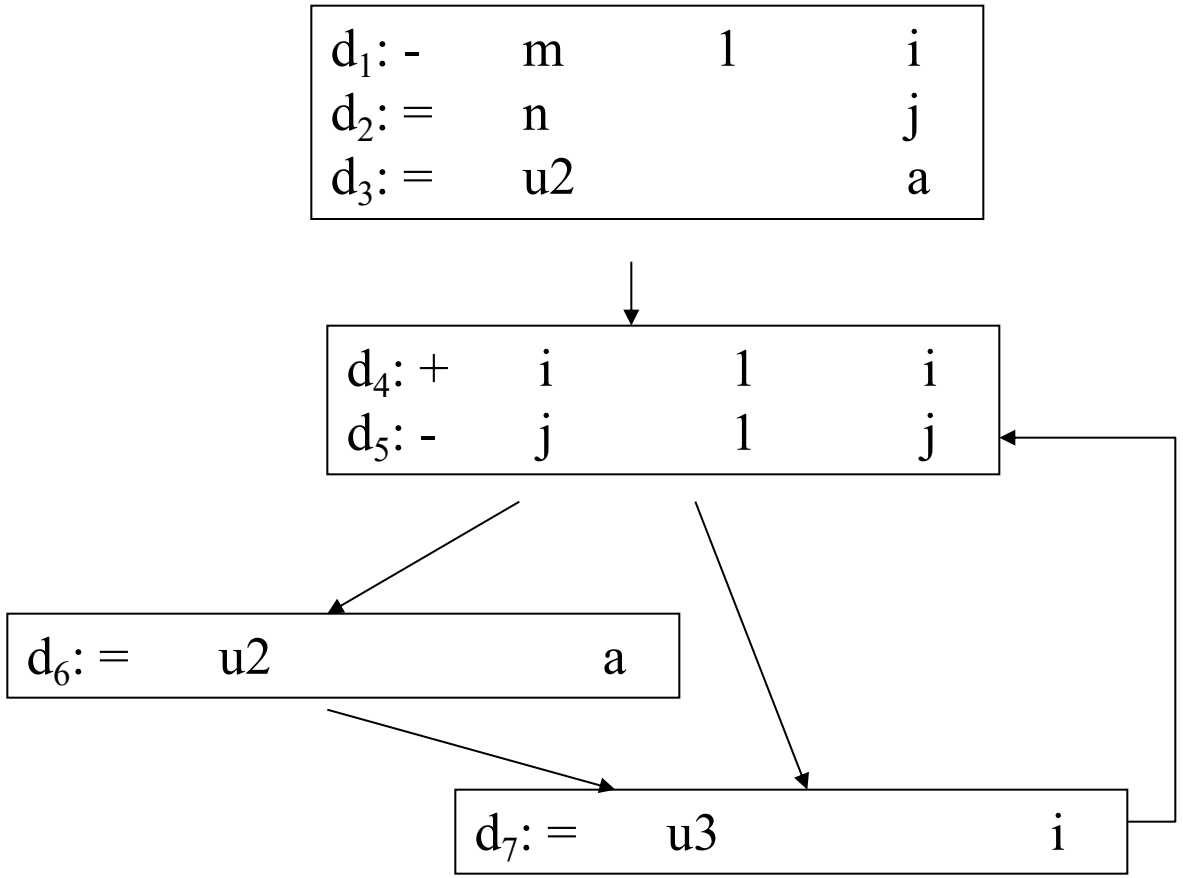
活跃变量数据流方程求解

- 设置初值：in[B_i]=空;
- 重复执行以下步骤直到in[B_i]不再改变：

```
for(i=1; i<n; i++)
{
    out[B_i]= ∪ in[s]; s是B_i的后继；
    in[B_i]=use[B_i] ∪ (out[B_i]-def[B_i]);
}
```

活跃变量数据流方程例子

- def[B_1]={i,j,a}
- use[B_1]={m,n, u1}
- def[B_2]=空
- use[B_2]={i,j}
- def[B_3]={a}
- use[B_3]={u2}
- def[B_4]={i}
- use[B_4]={u3}



迭代过程

- 第一次循环：
  - out[B_1]=空　in[B_1]={m,n,u1}
  - out[B_2]=空　in [B_2]={i,j}
  - out[B_3]={i,j}　in [B_3]={i,j,u2}
  - out[B_4]={i,j,u2}　in[B_4]={j,u2,u3}
- 第二次循环：
  - out[B_1]={i,j,u2,u3}　　in [B_1]={m,n,u1,u2,u3}
  - out[B_2]={i,j,u2,u3}　　in [B_2]={i,j,u2,u3}
  - out[B_3]={i,j,u2,u3}　　in [B_3]={i,j,u2,u3}
  - out[B_4]={i,j,u2,u3}　　in [B_4]={j,u2,u3}
- 第三次循环各个值不再改变，完成求解。

10.3.3 活跃变量分析

- 定义-引用链

  - 定义-引用链(简称du-链)是一种和活跃变量分析方式相同的数据流信息。定义-引用链的计算与引用-定义链的计算正好相反，它是从变量x的某个定义点p出发，计算该定义可以到达的所有引用s的集合，所谓可以到达是指从p到s有一条没有重新定义x的路径。

- 在代码优化过程中，无用代码删除、强度削弱、循环中的代码外提以及目标代码生成过程中的寄存器分配都要用到du-链信息。

10.3.4 可用表达式分析

- 如果从初始节点到*p*的每一条路径(不必是无环路的)都要计算x op y，而且在到达*p*的这些路径上没有对x或y的赋值，则称表达式x op y在*p*点是可用的。
- 对表达式的注销：如果基本块*B*中含有对x或y的赋值(或可能赋值)，而且后来没有重新计算x op y，则称*B*注销了表达式x op y。
- 表达式的生成：如果基本块*B*明确地计算了x op y，并且后来没有重新定义x或y，则称*B*生成了表达式x op y。
- 可用表达式信息的主要用途是检测公共子表达式。

记号

- out[B]：在基本块出口处的可用表达式集合。
- in[B]：在基本块入口处的可用表达式集合。
- e_gen[B]：基本块B生成的可用表达式的集合。
- e_kill[B]：基本块B注销的可用表达式的集合。
- e_gen[B]和e_kill[B]的值可以直接从流图计算出来，因此在数据流方程中，可以将e_gen[B]和e_kill[B]当作已知量看待。

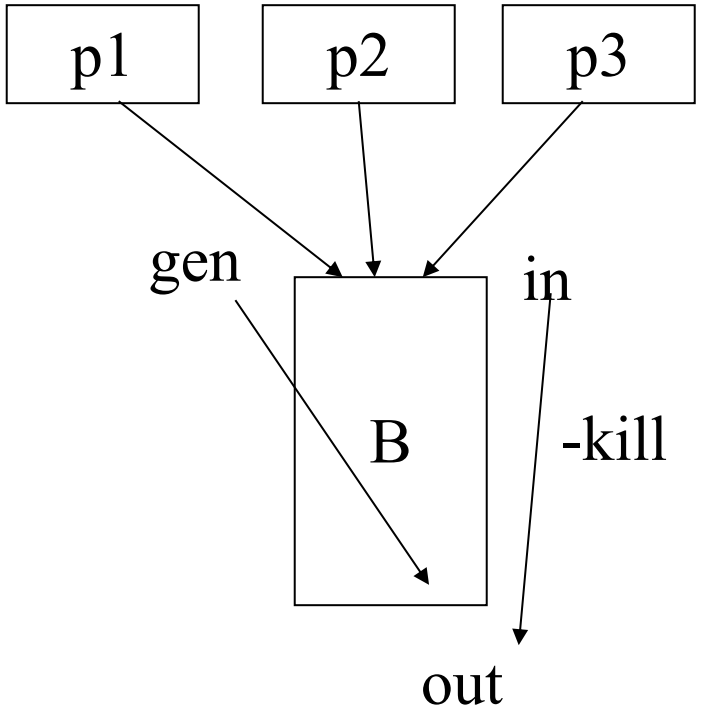
e_gen[B]的计算

- 对于一个基本块B，e_gen[B]的计算过程为：
- 初始设置：e_gen[B]=空；
- 顺序扫描每个四元式：
  - 对于四元式op x y z，把x op y加入e_gen[B]，
  - 从gen[B]中删除和z相关的表达式。
- 最后的e_gen[B]就是相应的集合。

e_kill[B]的计算

- 设流图的表达式全集为E;
- 初始设置：E_K = 空；
- 顺序扫描基本块的每个四元式：
  - 对于四元式op x y z，把表达式x op y从E_K中消除；
  - 把E中所有和z相关的四元式加入到E_K中。
- 扫描完所有的四元式之后，E_K就是所求的e_kill[*B*]。

数据流方程



1. out[B]=e_gen[B]∪(in[B]-e_kill[B])
2. in[B]= ∩out[p] B!=B_1, p是B的前驱。
3. in[B_1]=空集

- 说明：
  - 在程序开始的时候，无可用表达式。(3)
  - 一个表达式在某个基本块的入口点可用，必须要求它在所有前驱的出口点也可用。（2）
  - 一个表达式在某个基本块的出口点可用，或者该表达式是由它的产生的；或者该表达式在入口点可用，且没有被注销掉。（1）

方程求解算法

- 迭代算法
- 初始化：in[B_1]=空; out[B_1]=e_gen[B_1];out[B_i]=U-e_kill[B_i](i>=2)
- 重复执行下列算法直到out稳定:

```
for（ i=2; i<=n ;i++) {
    in[Bi]= ∩out[p], p是Bi的前驱；
    out[Bi]=e_gen[Bi] ∪(in[Bi]-e_kill[Bi]);
}
```

算法说明

- 初始化值和前面的两个算法不同。out[B_i]的初值大于实际的值。
- 在迭代的过程种，out[B_i]的值逐渐缩小直到稳定。
- U表示四元式的全集，就是四元式序列中所有表达式x op y的集合。

10.4 局部优化

- 基本块的功能实际上就是计算一组表达式，这些表达式是在基本块出口活跃的变量的值。如果两个基本块计算一组同样的表达式，则称它们是等价的。
- 可以对基本块应用很多变换而不改变它所计算的表达式集合，许多这样的变换对改进最终由某基本块生成的代码的质量很有用。
- 利用基本块的dag表示可以实现一些常用的对基本块的变换。

10.4.1 基本块的dag表示

- dag的构造方法

(1) 基本块中出现的每个变量都有一个dag节点表示其初始值。

(2) 基本块中的每个语句s都有一个dag节点n与之相关联。n的子节点是那些在s之前、最后一次对s中用到的运算对象进行定义的语句所对应的节点。

(3) 节点n由s中用到的运算符来标记，节点n还附加了一组变量，这些变量在基本块中都是由s最后定义的。

(4) 如果有的话，还要记下那些其值在块的出口是活跃的节点，它们是输出节点。流图的另一个基本块以后可能会用到这些变量的值。

利用dag进行的基本块变换

(1) 局部公共子表达式删除。

(2) 无用代码删除。

(3) 交换两个独立的相邻语句的次序，以便减少某个临时值需要保存在寄存器中的时间。

(4) 使用代数规则重新排列三地址码的运算对象的顺序，以便简化计算过程。

10.4.2 局部公共子表达式删除

- 例10.12

  a := b+c

  b := a-d

  c := b+c

  d := a-d (10.8)

- 如果b在出口处不是活跃的：

  a := b+c

  d := a-d

  c := d+c

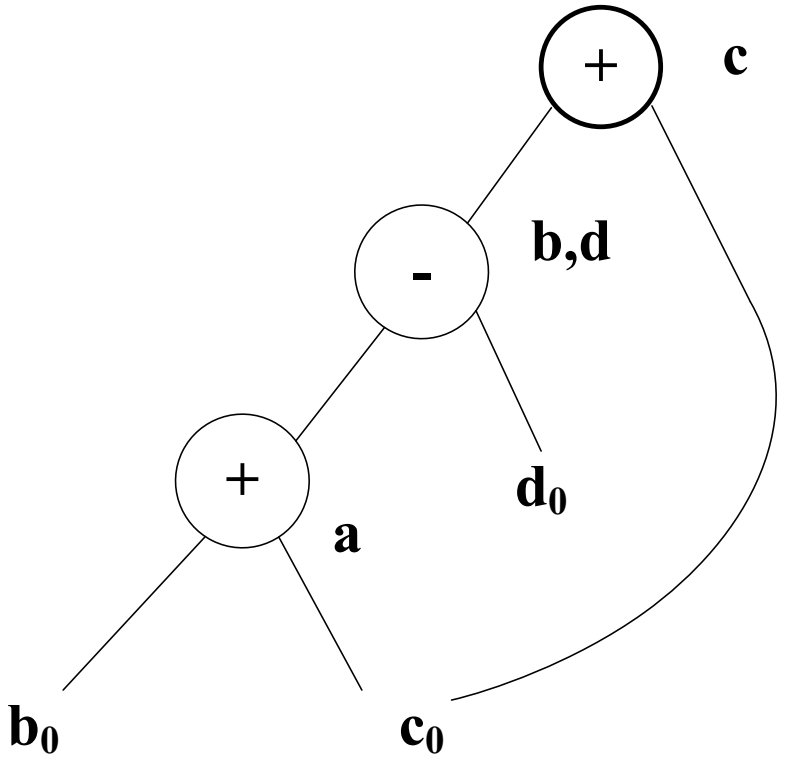


图10.13 基本块(10.8)的dag

10.4.3 无用代码删除

- 在dag上删除无用代码的方法很简单：只要从dag上删除所有没有附加活跃变量的根节点(即没有父节点的节点)即可。重复进行这样的处理即可从dag中删除所有与无用代码相对应的节点。

```
a := b+c
b := b-d
c := c+d
e := b+c
```

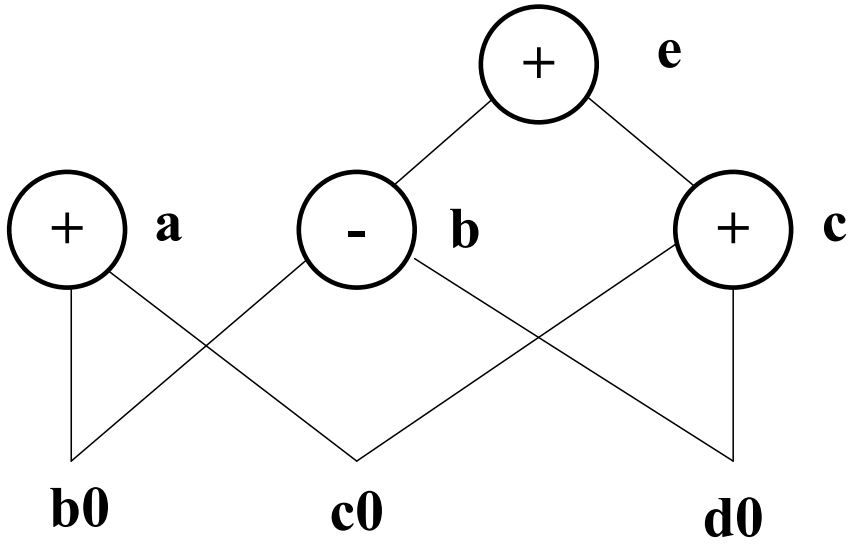


图10.14基本块(10.9)的dag

- 图10.14中的a和b是活跃变量，而c和e不是，我们可以立即删除标记为e的根节点。随后，标记为c的节点变成了根节点，下一步也可以被删除。

10.4.4代数恒等式的使用

- 代数恒等式代表基本块上另一类重要的优化方法，下面是一些常见的代数恒等式：
- x+0 = 0+x = x　　x–0 = x　　x*1 = 1*x = x　　x/1 = x
- 另外一类代数优化是局部强度削弱，即用较快的运算符取代较慢的运算符，例如：
- x**2 = x*x　　2.0*x = x+x　　x/2 = x*0.5

10.4.4代数恒等式的使用

- 第三类相关的优化技术是常量合并。即在编译时对常量表达式进行计算，并利用它们的值取代常量表达式。例如，表达式2*3.14可以替换为6.28。
- dag构造过程可以帮助我们应用上述和更多其它的通用代数变换，比如交换律和结合律。

10.4.5 数组引用的dag表示

在dag中表示数组访问的正确方法为：

- 将数组元素赋给其他变量的运算(如x = a[i])用一个新创建的运算符为=[]的节点表示。该节点的左右子节点分别代表数组初始值(本例中为a0)和下标i。变量x则是该节点的附加标记之一。
- 对数组元素的赋值(如a[j]=y)则用一个新创建的运算符为[]=的节点来表示。该节点的三个子节点分别表示a0、j和y。该节点不带任何附加标记。这是因为该节点的创建注销了所有当前已经创建的、其值依赖于a0的节点，而一个被注销的节点不可能再获得任何标记。也就是说，它不可能成为一个公共子表达式。

10.4.5 数组引用的dag表示

- 例10.14 基本块

  x = a[i]

  a[j] = y

  z = a[i]

的dag如图10.15所示。

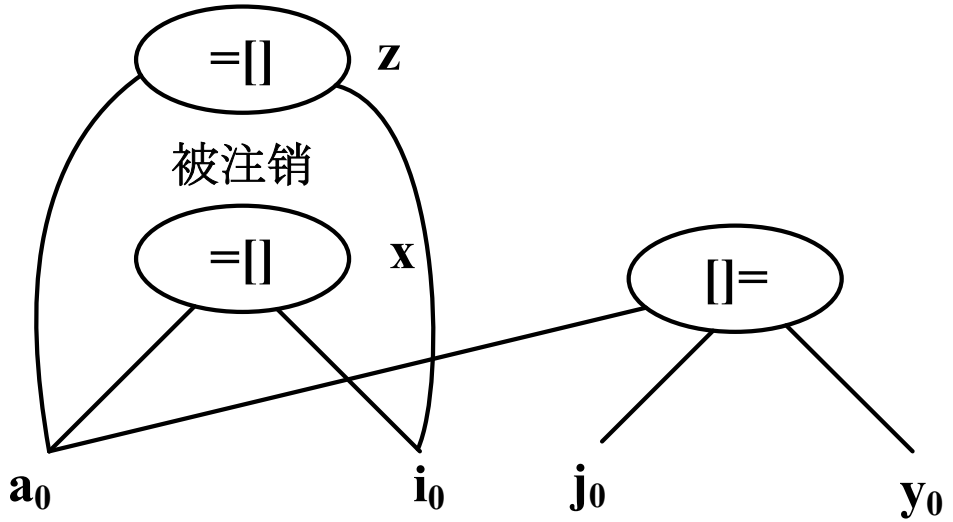


图10.15 将数组元素赋值给其他变量的语句序列的dag

10.4.6 指针赋值和过程调用的dag表示

- 当像语句x=*p或*q=y那样通过指针进行间接赋值时，并不知道p和q指向哪里。
- x=*p可能是对任意某个变量的引用，而*q=y则可能是对任意某个变量的赋值。因而，运算符=*必须把当前所有带有附加标识符的节点当作其参数。但这么做将会影响无用代码的删除。更为严重的是，*=运算符将把所有迄今为止构造出来的dag中的其他节点全部注销。
- 可以进行一些全局指针分析，以便把一个指针在代码中某个位置上可能指向的变量限制在一个较小的的子集内。

10.4.7 从dag到基本块的重组

- 对每个具有一个或多个附加变量的节点，构造一个三地址码来计算其中某个变量的值。尽量把计算得到的结果赋给一个在基本块出口处活跃的变量，如果没有全局活跃变量的信息，则假设程序的所有变量在基本块的出口处都是活跃的(临时变量除外)。

- 如果节点具有多个附加的活跃变量，则必须引入复制语句，以便为每一个变量都赋予正确的值。当然，通过全局优化技术可以消除这些复制语句。

10.4.7 从dag到基本块的重组

- 当从dag重构基本块时，不仅要关心用哪些变量来存放dag中节点的值，还要关心计算不同节点值的语句顺序。下面是应遵循的规则：

  (1) 语句的顺序必须遵守dag中节点的顺序。也就是说，只有在计算出某个节点的各个子节点的值之后，才能计算该节点的值。

  (2) 对数组的赋值必须跟在所有(按照原基本块中的语句顺序)在它之前的对同一数组的赋值或引用之后

10.4.7 从dag到基本块的重组

(3) 对数组元素的引用必须跟在所有(在原基本块中)在它之前的对同一数组的赋值语句之后。对同一数组的两次引用可以交换顺序，前提是交换时它们都没有越过某个对同一数组的赋值运算。

(4) 对某个变量的引用必须跟在所有(在原基本块中)在它之前的过程调用或指针赋值运算之后。

(5) 任何过程调用或指针赋值都必须跟在所有(在原基本块中)在它之前的对任何变量的引用之后。

10.5 循环优化

- 循环不变计算的检测
- 代码外提
- 归纳变量删除和强度削弱

10.5.1循环不变计算的检测

算法10.7 循环不变计算检测

- 输入：由一组基本块构成的循环L，每个基本块包括一系列的三地址码，且每个三地址码的ud-链均可用；
- 输出：从控制进入循环L一直到离开L，每次都计算同样值的三地址码；
- 步骤：
  1. 将如下语句标记为“不变”：它们的运算对象或者是常数，或者它们的所有到达-定义都在循环L的外面。
  2. 重复步骤(3),直到某次重复没有新的语句可标记为“不变”为止
  3. 将如下语句标记为“不变”：它们先前没有被标记，而且它们的所有运算对象或者是常数，或者其到达定义都在循环L之外，或者只有一个到达定义，这个定义是循环L中已标记为“不变”的语句。

10.5.2 代码外提

- 将语句*s*：x:= y+z外提的条件：

1．含有语句*s*的块是循环中所有出口节点的支配节点，出口节点指的是其后继节点不在循环中的节点

2．循环中没有其它语句对x赋值。如果x是只赋值一次的临时变量，该条件肯定满足，因此不必检查。

3．循环中x的引用仅由*s*到达，如果x是临时变量，该条件一般也可以满足。

10.5.2 代码外提

算法10.8　代码外提

- 输入：带有ud-链和支配节点信息的循环L；
- 输出：循环的修正版本，增加了前置首节点，且(可能)有一些语句外提到前置首节点中；
- 步骤：

1. 应用算法10.7寻找循环不变语句。
2. 对(1)中找到的每个定义x的语句s，检查是否满足下列条件：
   a) s所在的块支配L的所有出口；
   b) x在L的其它地方没有被定义，而且
   c) L中所有x的引用只能由s中x的定义到达。
3. 按算法10.7找出的次序，把(1)中找出的满足(2)中3个条件的每个语句移到新的前置首节点。但是，若s的运算对象在L中被定义(由算法10.7的(3)找出这种s)，那么只有这些对象的定义语句外提到前置首节点后，才能外提s。

10.5.3 归纳变量删除和强度削弱

算法10.9　归纳变量检测

- 输入：带有到达定义信息和循环不变计算信息(由算法10.7得到)的循环L；
- 输出：一组归纳变量，以及与每个归纳变量j相关联的三元组(i, c, d)，其中i是基本归纳变量，c和d是常量，在j的定义点，j的值由c*i+d给出。称j属于i族，基本归纳变量i也属于它自己的族；
- 步骤：

1. 在循环L中找出所有的基本归纳变量，此处需要用到循环不变计算的信息。与每个基本归纳变量i相关联的三元组为(i, 1, 0)。

10.5.3 归纳变量删除和强度削弱

2．在L中寻找具有下列形式之一且只被赋值一次的变量k：

k: = j*b，　k: = b*j，　k: = j/b，　k: = j±b，　k: = b±j

其中，b是常数，j是基本的或非基本的归纳变量。

(1)如果j是基本归纳变量，则k在j族中。k的三元组依赖于定义它的语句。例如，如果k是由k: = j*b定义的，则k的三元组为(j, b, 0)。类似地可以定义其他情况的三元组。

(2)如果j不是基本归纳变量，假设j属于i族，则j还要满足如下要求：

a) 在循环L中对j的唯一赋值和对k的赋值之间没有对i的赋值。

b) 循环L外没有j的定义可到达k。

此时利用j的三元组(i, c, d)和定义k的语句即可计算k的三元组。例如，定义k: = b*j导致k的三元组为(i, b*c, b*d)。注意，b*c和b*d可以在分析过程中完成计算，因为b，c和d都是常数。

10.5.3 归纳变量删除和强度削弱

算法10.10　用于归纳变量的强度削弱。

- 输入：循环L，附带有到达定义信息和由算法10.9算出的归纳变量族；
- 输出：修正后的循环；
- 步骤：依次考察每个基本归纳变量i。对每个三元组为(i, c, d)的i族归纳变量j，执行如下步骤：

1. 建立新变量s，但如果变量j1和j2具有同样的三元组，则只为它们建立一个新变量。
2. 用j: = s代替对j的赋值。

10.5.3 归纳变量删除和强度削弱

3．在L中紧跟在每个赋值语句i: = i+n之后(n是常数)，添加上如下语句：

s := s + c*n

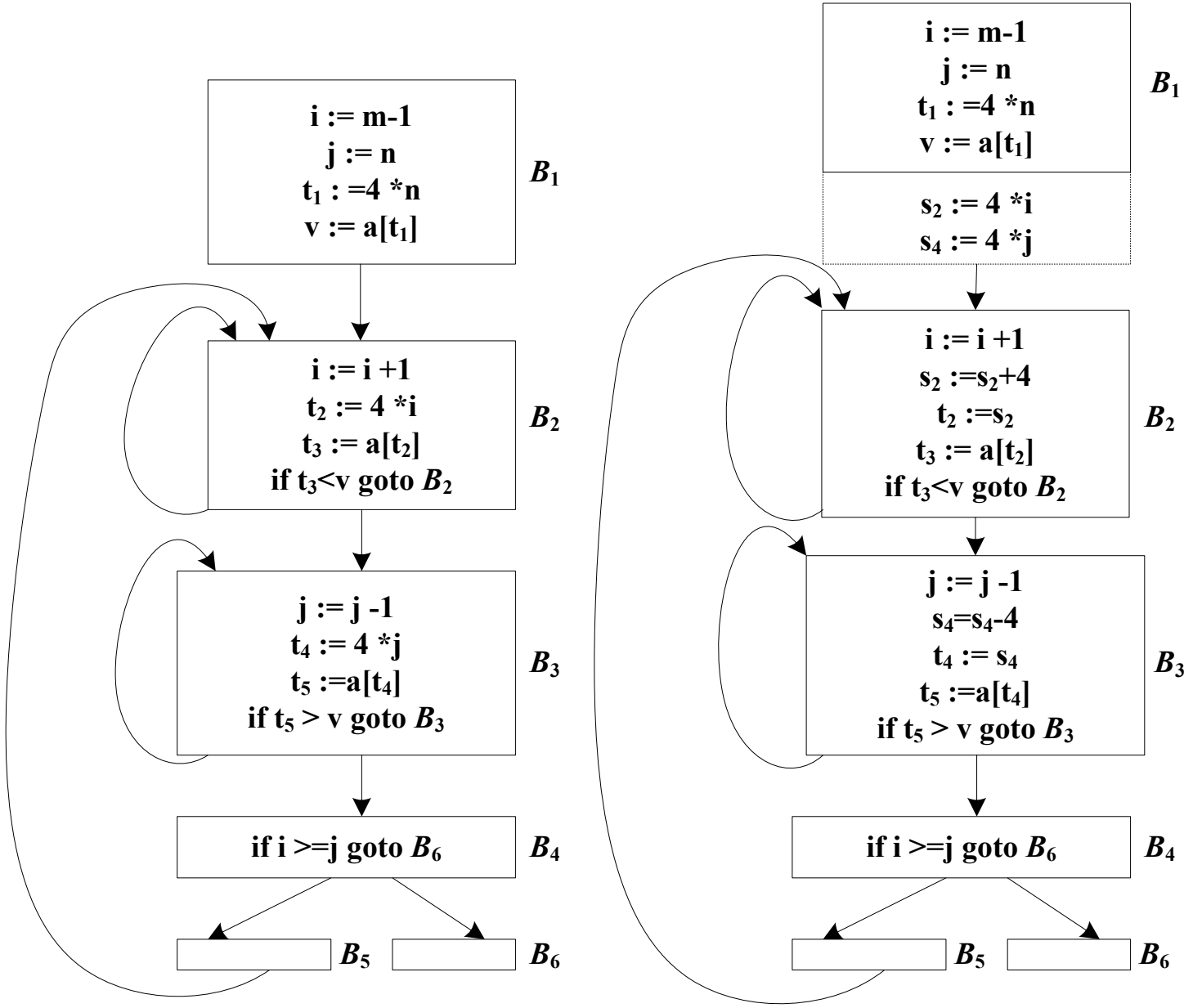
其中，表达式c*n的计算结果为一个常数，因为c和n都是常数。将s放入i族，其三元组为(i, c, d)。

4．必须保证在循环的入口处s的初值为c*i+d，该初始化可以放在前置首节点的末尾，由下面两个语句组成：

s : = c*i　　/ * 如果c为1，则为s:=i * /

s : = s+d　　/ * 如果d为0则省略 * /

注意，s是i族的归纳变量。



(a) 变换前　　(b) 变换后

图10.20 强度削弱

10.5.3 归纳变量删除和强度削弱

算法10.11　归纳变量删除

- 输入：带有到达-定义信息、循环不变计算信息(利用算法10.7求得)和活跃变量信息的循环L；
- 输出：修正后的循环；
- 步骤：

10.5.3 归纳变量删除和强度削弱

1．考虑每个仅用于计算同族中其它归纳变量和条件分支的基本归纳变量i。取i族的某个j，优先取其三元组(i, *c*, *d*)中的*c*和*d*尽可能简单的j(即优先考虑*c* = l和*d* = 0的情况)，把每个包含i的测试语句改成用j代替i。假定下面的*c*是正的。用下面的语句来代替形如if i relop x goto *B*的测试语句，其中x不是归纳变量。

```
r: = c*x        / * 如果c等于1，则为r: = x * /

r: = r+d        / * 如果d为0则省略 * /

if i relop x goto B
```

最后，因为被删除的归纳变量已经没有什么用处，所以从循环中删除所有对它们的赋值。

10.5.3 归纳变量删除和强度削弱

2．再考虑由算法10.10为其引入语句j:=s的每个归纳变量j。首先检查在引入的j: = s和任何j的引用之间有没有对s的赋值，应该是没有。一般情况下，j在定义它的块中被引用，这可以简化该检查；否则，需要用到达定义信息，并加上一些对图的分析来实现这种检查。然后用对s的引用代替所有对j的引用，并删除语句j: = s。

10.6 全局优化

- 全局公共子表达式的删除
- 复制传播

10.6.1 全局公共子表达式删除

算法10.12 全局公共子表达式删除

- 输入：带有可用表达式信息的流图；
- 输出：修正后的流图；
- 步骤：对每个形如x:=y+z的语句*s*，如果y+z在*s*所在块的开始点是可用的，且该块中在语句*s*之前没有对y或z的定义，则执行下面的步骤：

1. 为寻找到达*s*所在块的y+z的计算，只需沿流图的边从该块开始反向搜索，但不穿过任何计算y+z的块。在遇到的每个块中，对y+z的最后一次计算将是到达*s*的y+z的计算。
2. 建立新变量u。
3. 把(1)中找到的每个语句w:=y+z用如下语句代替：

```
u := y+z
w := u
```

4. 用x:=u代替语句*s*。

10.6.2 复制传播

- 算法10.12、归纳变量删除算法和其它一些算法都会引入形如x: = y的代码。
- 如果能找出复制代码*s*：x:=y中定义x的所有引用点，并用y代替x，则可以删除该复制语句。前提是x的每个引用*u*必须满足下列条件：

1．*s*必须是到达*u*的x的唯一定义(即引用*u*的ud-链只包含*s*)。

2．在从*s*到*u*的每条路径，包括穿过*u*若干次(但没有第二次穿过*s*)的路径上，没有对y的赋值。

10.6.2 复制传播

算法10.13复制传播。

- 输入：流图G；到达每个块B的定义的ud-链；c_in[B]，即沿着每条路径到达块B的复制语句x:=y的集合，在这些路径上x: = y的最后一次出现之后没有再对x或y进行赋值；每个定义的引用的du-链；
- 输出：修正后的流图；
- 步骤：对每个复制s：x:=y执行下列步骤：
  1. 确定由该x的定义所能到达的那些x的引用。
  2. 对(1)中找到的每个x的引用，确定s是否在c_in[B]中，其中块B是含有x的本次引用的基本块，而且块B中该引用的前面没有x或y的定义。回想一下，如果s在c_in[B]中，那么s是唯一的到达块B的x的定义。
  3. 如果s满足(2)中的条件，则删掉s，且把(1)中找出的所有x的引用用y来代替。

本章小结

- 代码优化就是对程序进行等价变换，以提高目标程序的效率，通常只对中间代码进行优化。通常包括控制流分析、数据流分析和变换三部分。
- 以程序的基本块为基础，基本块内的优化叫局部优化，跨基本块的优化为全局优化，循环优化是针对循环进行的优化，是全局优化的一部分。
- 公共子表达式的删除、复制传播、无用代码删除、代码外提、强度削弱和归纳变量删除等都是一些常用的针对局部或者全局的代码优化方法。

本章小结

- 划分基本块、构造并分析表示程序控制流信息的流图是进行控制流分析的基础工作。在对循环的分析中，用到支配节点、回边、自然循环、前置首节点、流图的可约等重要概念。
- 对程序中变量的定义和引用关系的分析称为数据流分析，用数据流方程表达变量的定义、引用等，具体进行到达-定义分析、定义-引用分析、可用表达式分析。