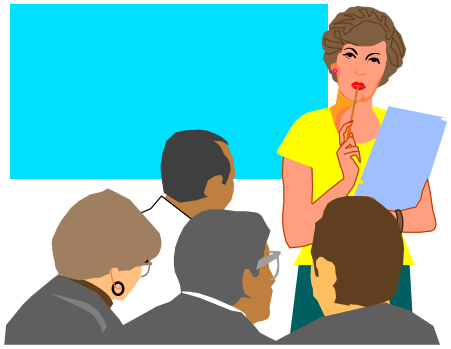
编译技术



胡春明
hucm@buaa.edu.cn

2019.9-2019.12

编译过程

编译过程是指将高级语言程序翻译为等价的目标程序的过程。

习惯上是将编译过程划分为5个基本阶段：

第十四章 代码优化

- 概述
- 优化的基本方法和例子
- 基本块和流图
- 基本块内的优化
- 全局优化

概述

代码优化（code optimization)

指编译程序为了生成高质量的目标程序而做的各种加工和处理。

目的：提高目标代码运行效率
- 时间效率（减少运行时间）
- 空间效率（减少内存容量）
- 能耗使用？（如在手机上）

原则：进行优化必须严格遵循“不能改变原有程序语义”原则。

优化方法的分类2：

- **局部优化技术**
  - 指在基本块内进行的优化
  - 例如，局部公共子表达式删除
- **全局优化技术**
  - 函数/过程内进行的优化
  - 跨越基本块
  - 例如，全局数据流分析
- **跨函数优化技术**
  - 整个程序
  - 例如，跨函数别名分析，逃逸分析 等

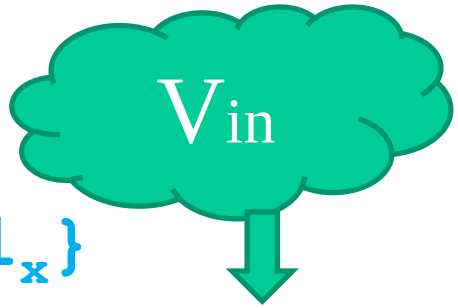
数据流分析方程

考察在程序的某个执行点的数据流信息。

- out[S]=gen[S]∪(in[S]-kill[S])
  - S 代表某条语句（基本块，基本块集合，或语句集合）
  - out[S]代表在该语句**末尾得到**的数据流信息
  - gen[S]代表该语句**本身产生**的数据流信息
  - in[S]代表**进入**该语句时的数据流信息
  - kill[S]代表该语句**注销**的数据流信息

数据流/可达定义分析

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



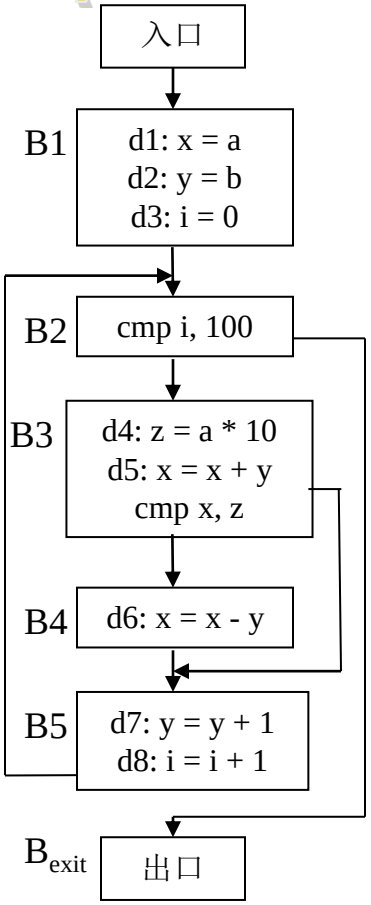
$in[S]=\{d1_x\}$

$-kill[S]=\{\}$ d2 : y=b $+gen[S]=\{d2_y\}$

$out[S]$

$$V_{out} = f_{a = b+c}(V_{in})$$

$$out[S]=\{d2_y\} \cup(\{d1_x\}-\{\})$$
$$= \{d1_x, d2_y\}$$




```
gen[B1] = {d1, d2, d3}
kill[B1] = {d5, d6, d7, d8}

gen[B2] = { }
kill[B2] = { }

gen[B3] = {d4, d5}
kill[B3] = {d1, d6}

gen[B4] = {d6}
kill[B4] = {d1, d5}

gen[B5] = {d7, d8}
kill[B5] = {d2, d3}
```

$in[B] = \cup_{B的前驱基本块P} out[P]$

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

迭代计算：

```
in[B1] = {},
out[B1] = {d1, d2, d3}
```

B2的前驱为B1和B5

```
in[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}
out[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}
```

B3的前驱为B2

```
in[B3] = {d1, d2, d3, d4, d5, d6, d7, d8}
out[B3] = {d2, d3, d4, d5, d7, d8}
```

B4的前驱为B3

```
in[B4] = {d2, d3, d4, d5, d7, d8}
out[B4] = {d2, d3, d4, d6, d7, d8}
```

B5的前驱为B3和B4

```
in[B5] = {d2, d3, d4, d5, d6, d7, d8}
out[B5] = {d4, d5, d6, d7, d8}
```

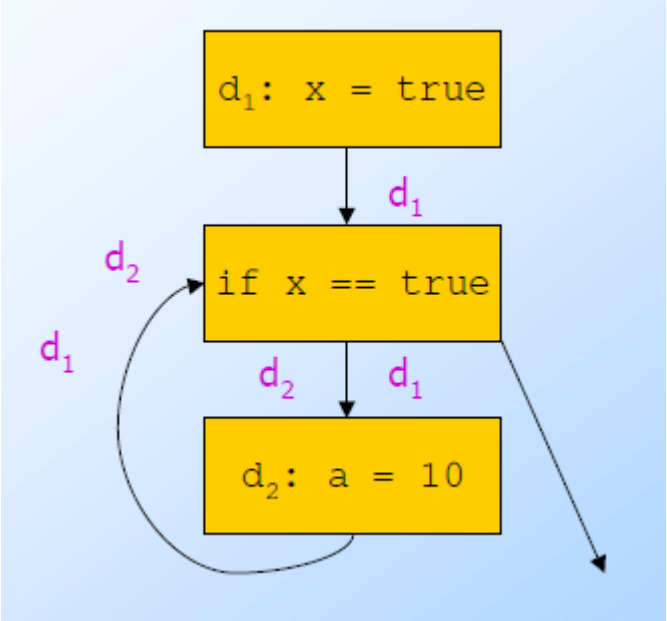
无改变，计算结束

可达定义（Reaching Definition）分析的用途?

```
bool x = true;
while (x) {
    … … // no change to x
}
```

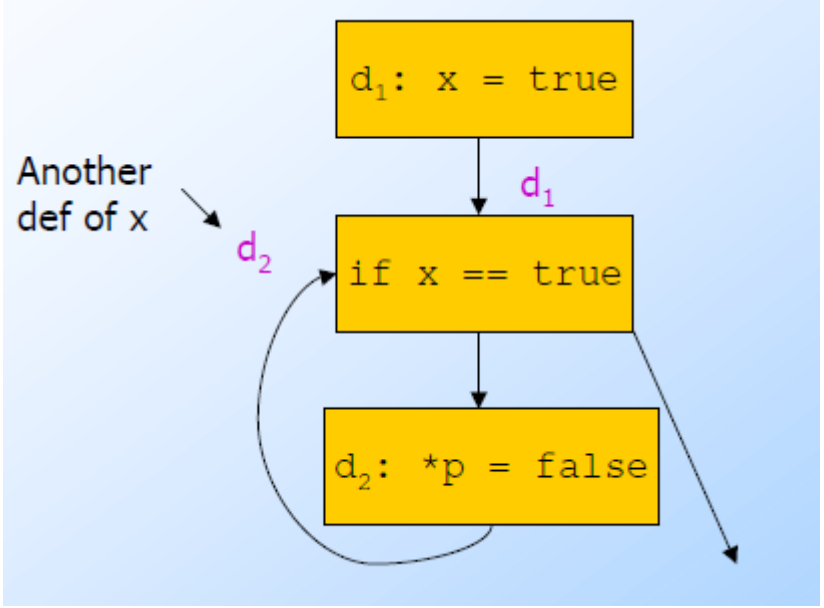
这个程序会死循环吗?

```
bool x = true;
while (x) {
    a=10; // for example
}
```



p 可能指向x吗?

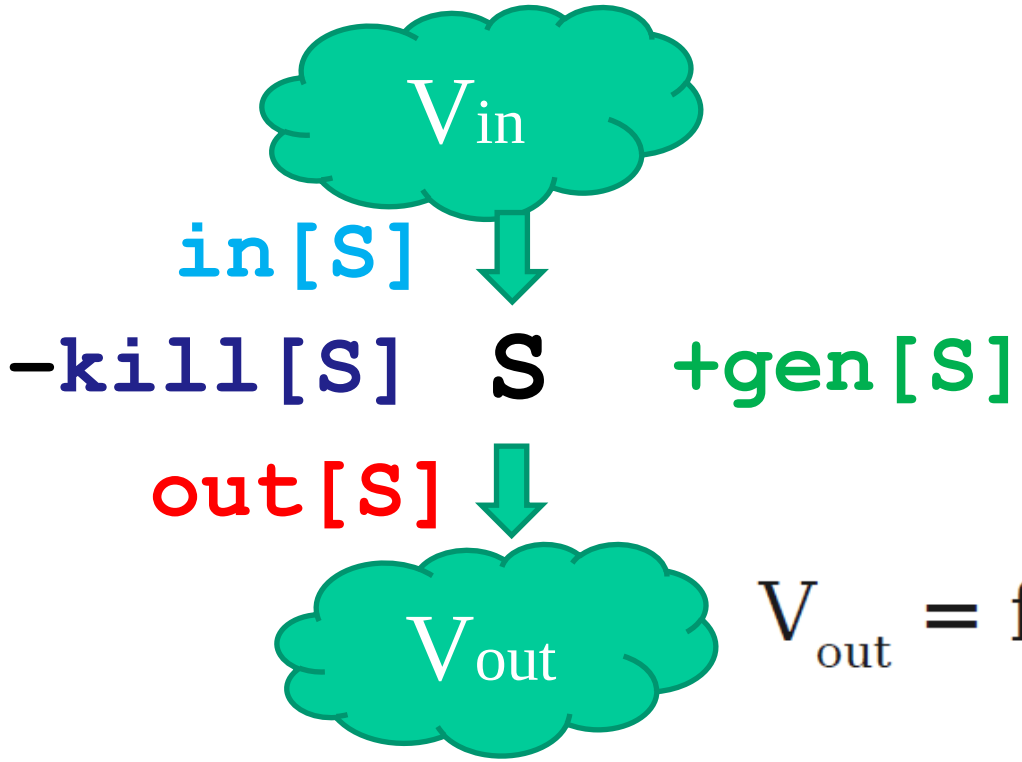
```
bool x = true;
while (x) {
    … …
    *p = false;
    … …
}
```



工程上可以通过两遍处理

数据流/可用表达式：公共子表达式删除

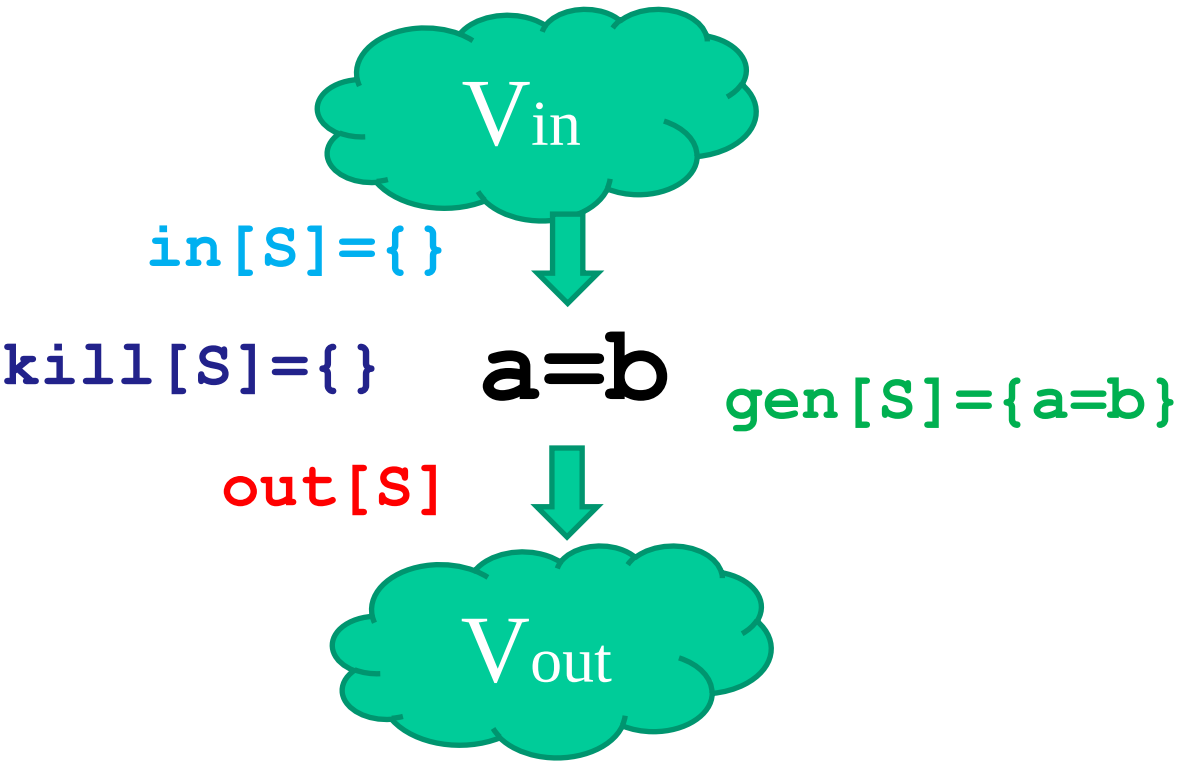
out[S]=gen[S]∪(in[S]-kill[S])



$$V_{out} = f_{a = b+c}(V_{in})$$

out[S]=gen[S]∪(in[S]-kill[S])

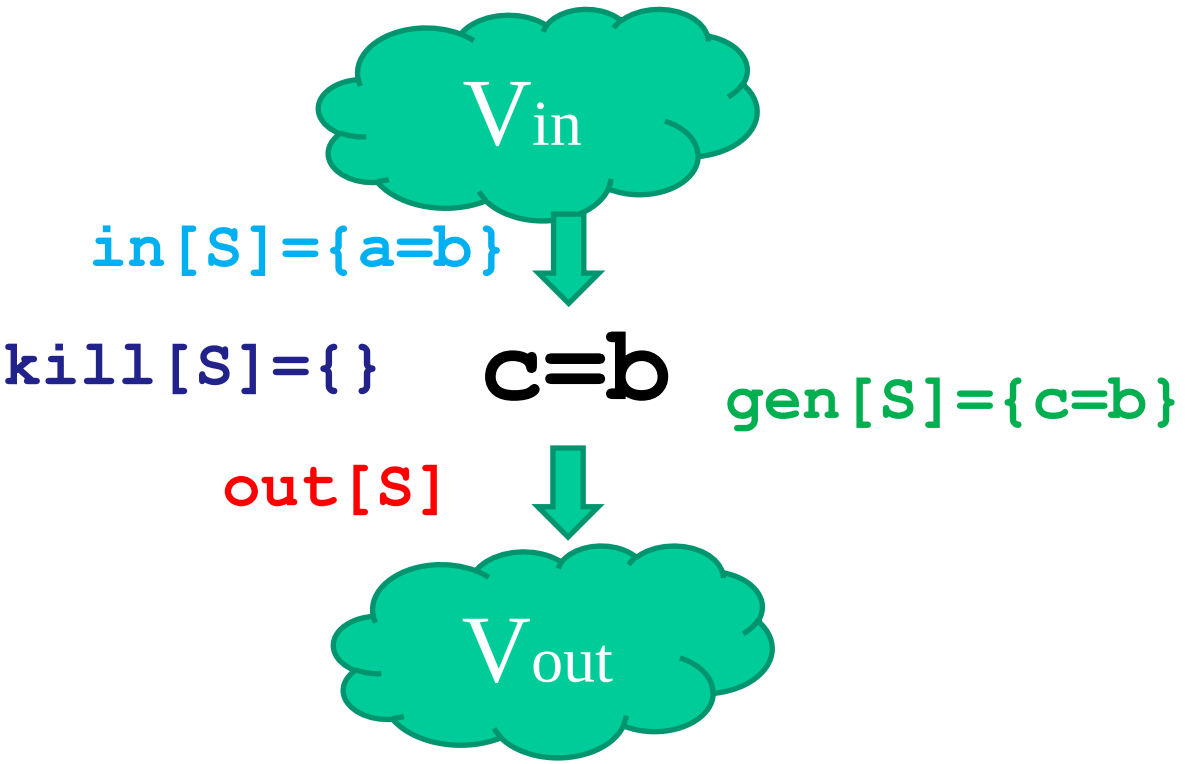
```
 a=b;
  c=b;
d=a+b;
e=a+b;
  d=b;
f=a+b;
```



out[S]={a=b} ∪({}-{})
= {a=b}

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

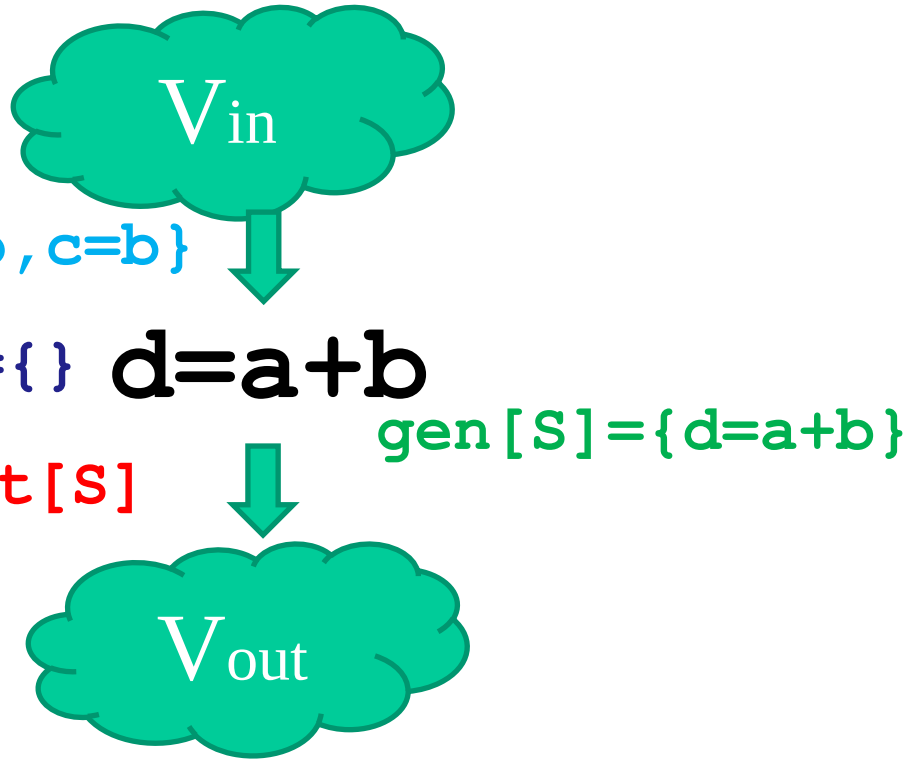
```
a=b;
c=b;
d=a+b;
e=a+b;
d=b;
f=a+b;
```



$$out[S]=\{c=b\}\cup(\{a=b\}-\{\})$$
$$=\{a=b,\ c=b\}$$

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

```
a=b;
c=b;
d=a+b;
e=a+b;
d=b;
f=a+b;
```



$$out[S]=\{d=a+b\}\cup(\{a=b,c=b\}-\{\})$$
$$=\{a=b,c=b,d=a+b\}$$

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Source: Stanford CS143 (2012)

```
                    { }
                  a = b;
                { a = b }
                  c = b; c=a;
              { a = b, c = b }
                d = a + b;
            { a = b, c = b, d = a + b }
                e = a + b;  e=d;
        { a = b, c = b, d = a + b, e = a + b }
                d = b; d=a;
          { a = b, c = b, d = b, e = a + b }
                f = a + b; f=e;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Source: Stanford CS143 (2012)

全局优化/数据流：活跃变量分析

活跃变量分析 (Live-variable Analysis)

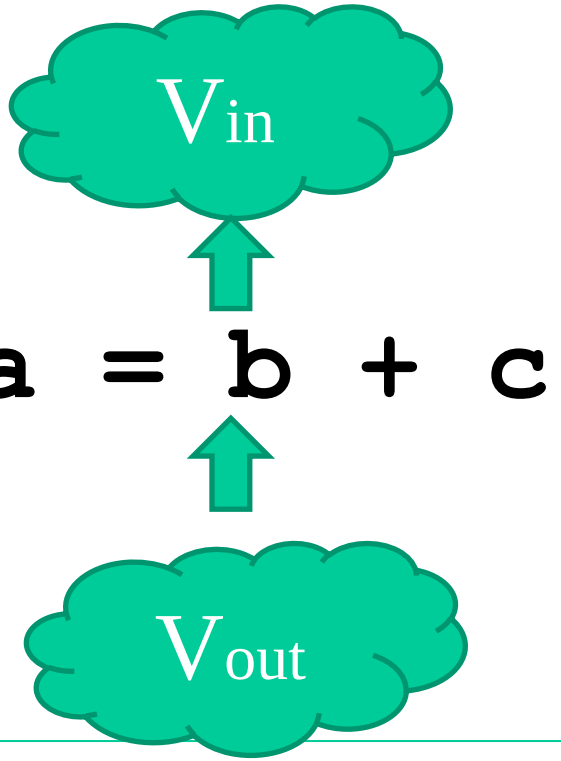
- 活跃变量信息对于寄存器分配，不论是全局寄存器分配还是临时寄存器分配都有重要意义。
  - 如果拥有寄存器的变量x在p点开始的任何路径上不再活跃，可以释放寄存器
  - 如果两个变量的活跃范围不重合，则可以共享同一个寄存器

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

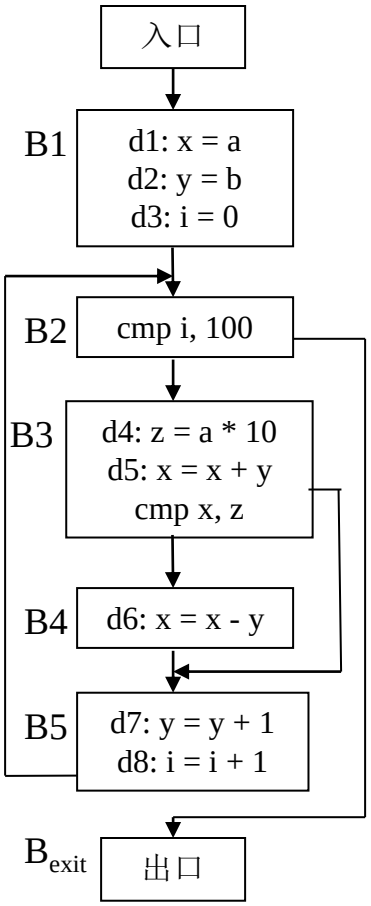
$$in[S] = use[S] \cup (out[S] - def[S])$$

引用变量会产生新的数据流　　赋值会删除数据流

$$(L - \{a\}) \cup \{b,c\}$$

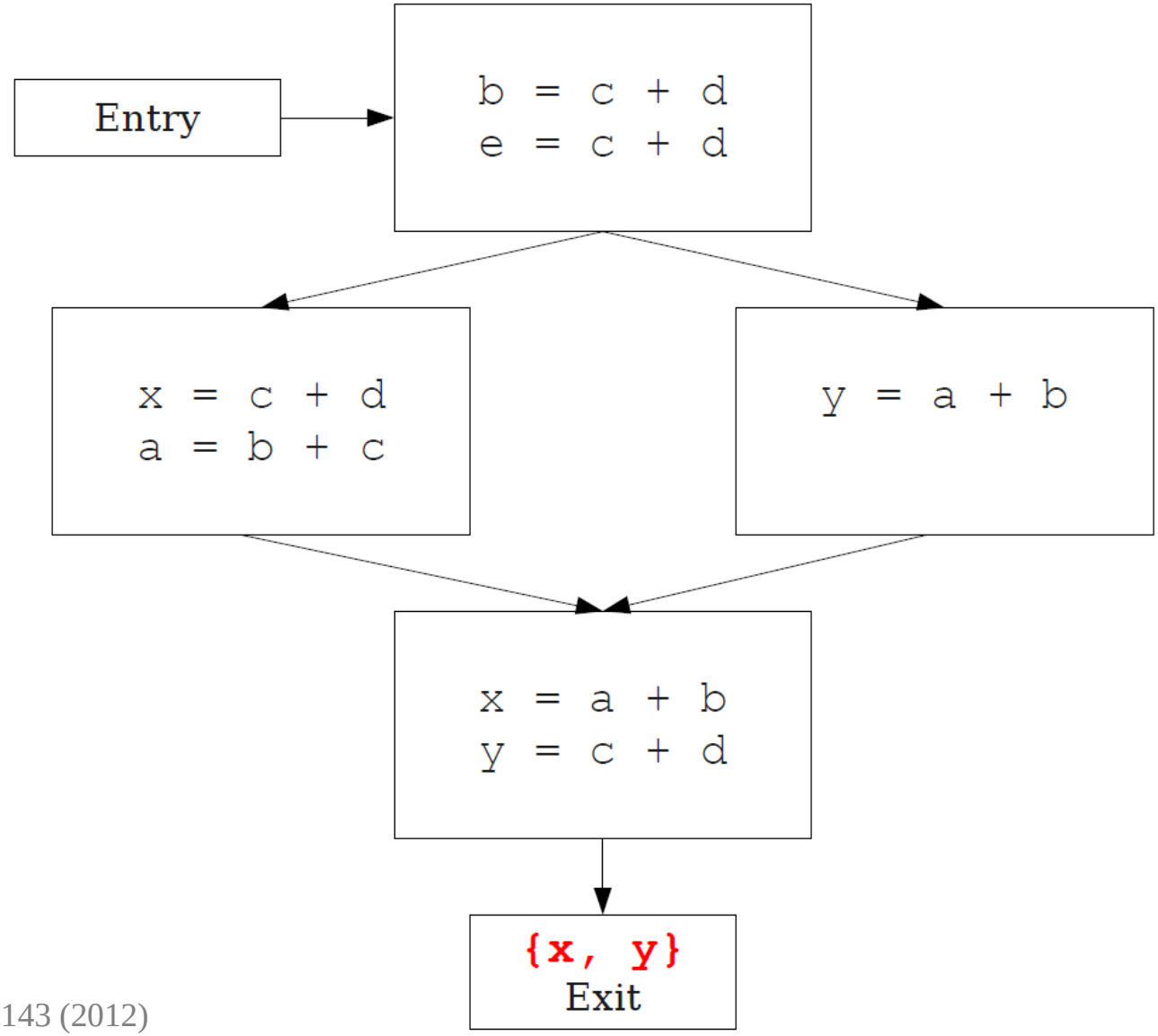


$L=\{\ldots\}$

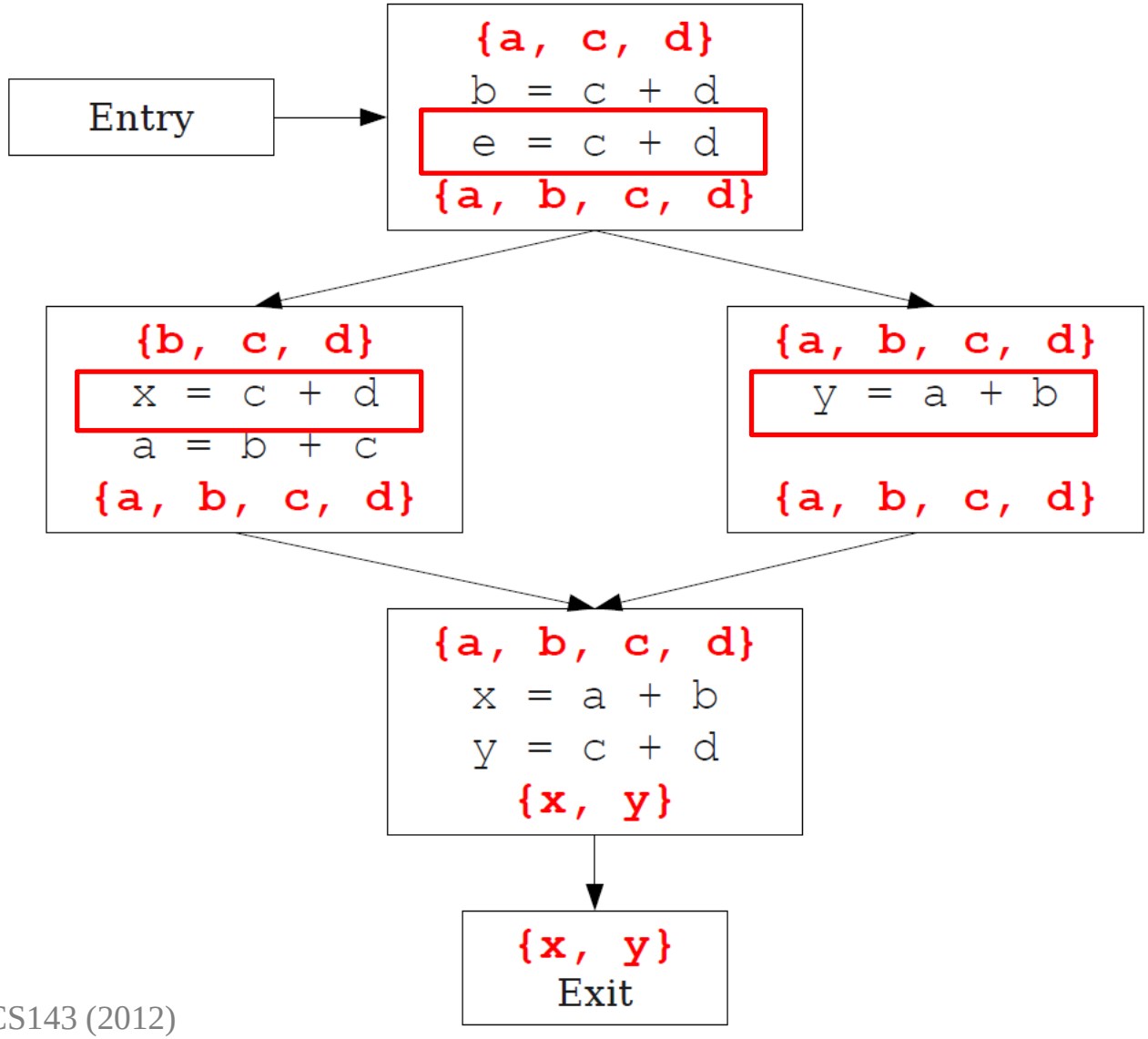


流图	def[B]	use[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
B1	x, y, i	a, b	a, b	a,x,y,i	a, b	a,x,y,i	a, b	a,x,y,i
B2	∅	i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B3	z	a, x, y	a,x,y,i	x, y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B4	∅	x, y	x, y, i	y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B5	∅	y, i	y, i	∅	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B_{exit}			∅	∅	∅	∅	∅	∅

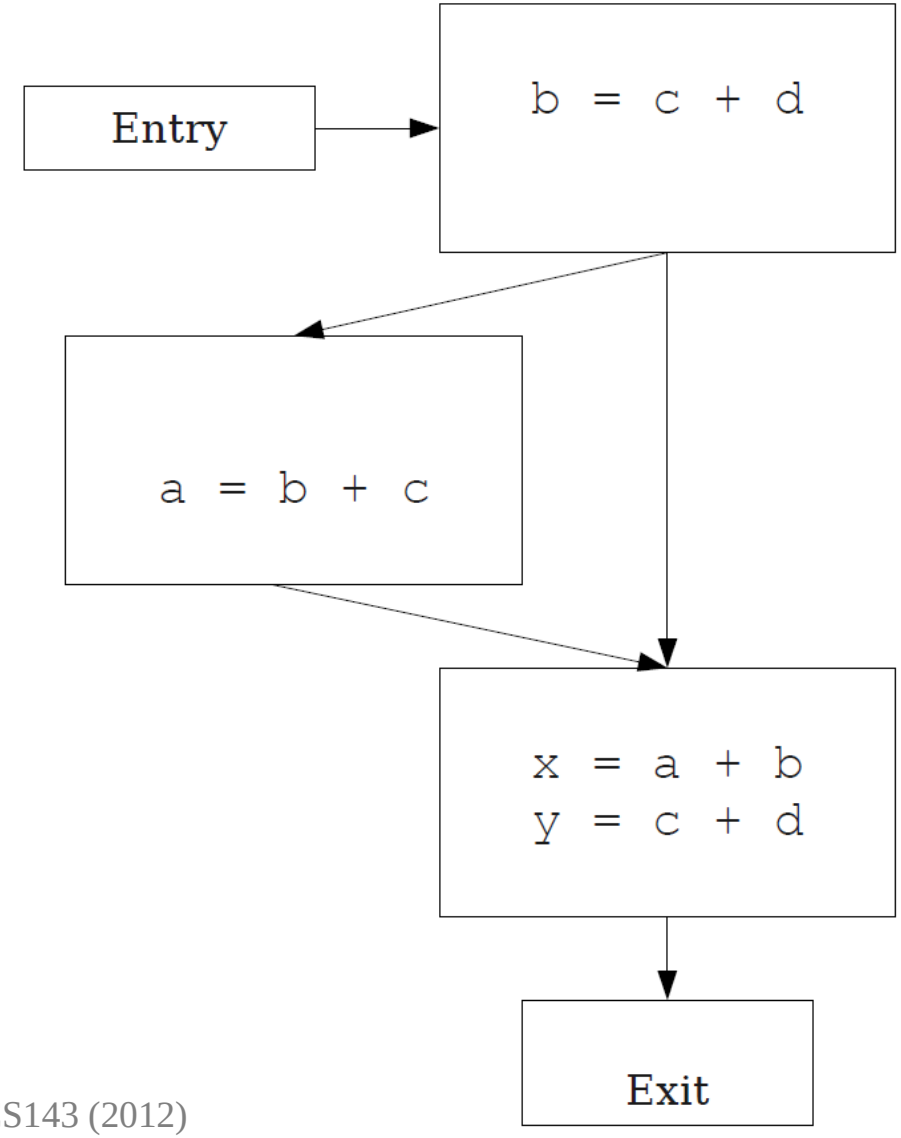
数据流/活性：死代码消除



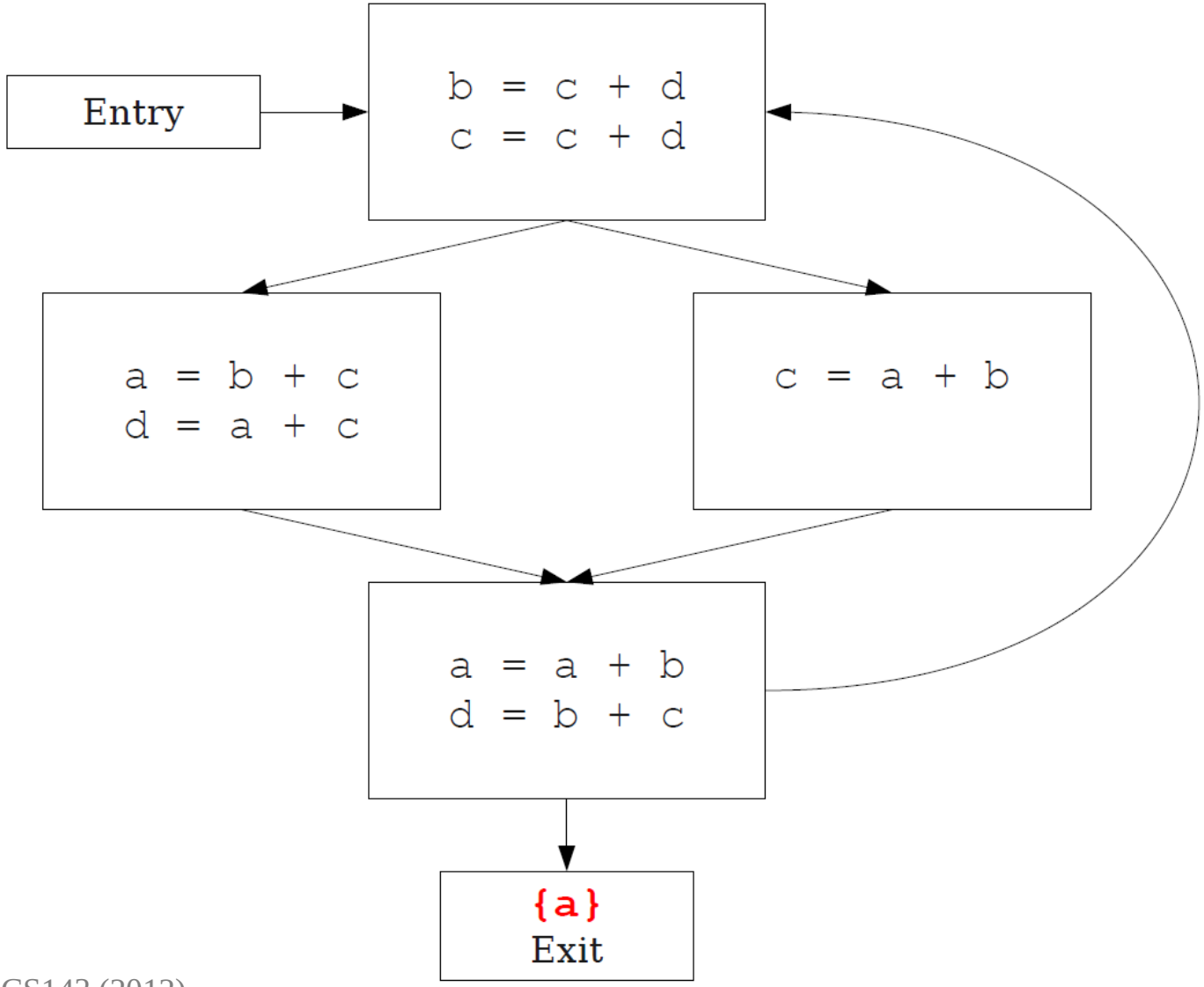
Source: Stanford CS143 (2012)



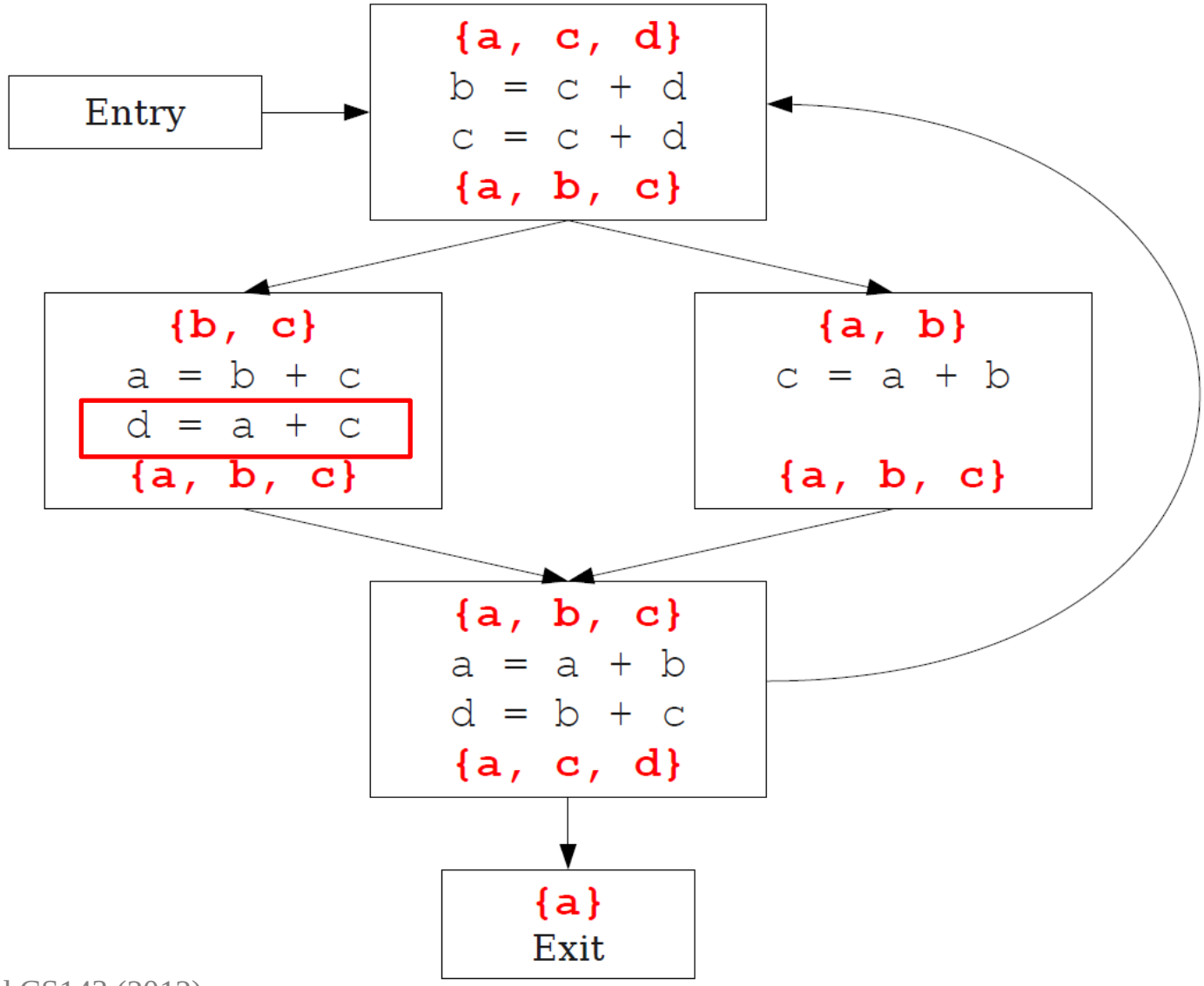
Source: Stanford CS143 (2012)



Source: Stanford CS143 (2012)



Source: Stanford CS143 (2012)



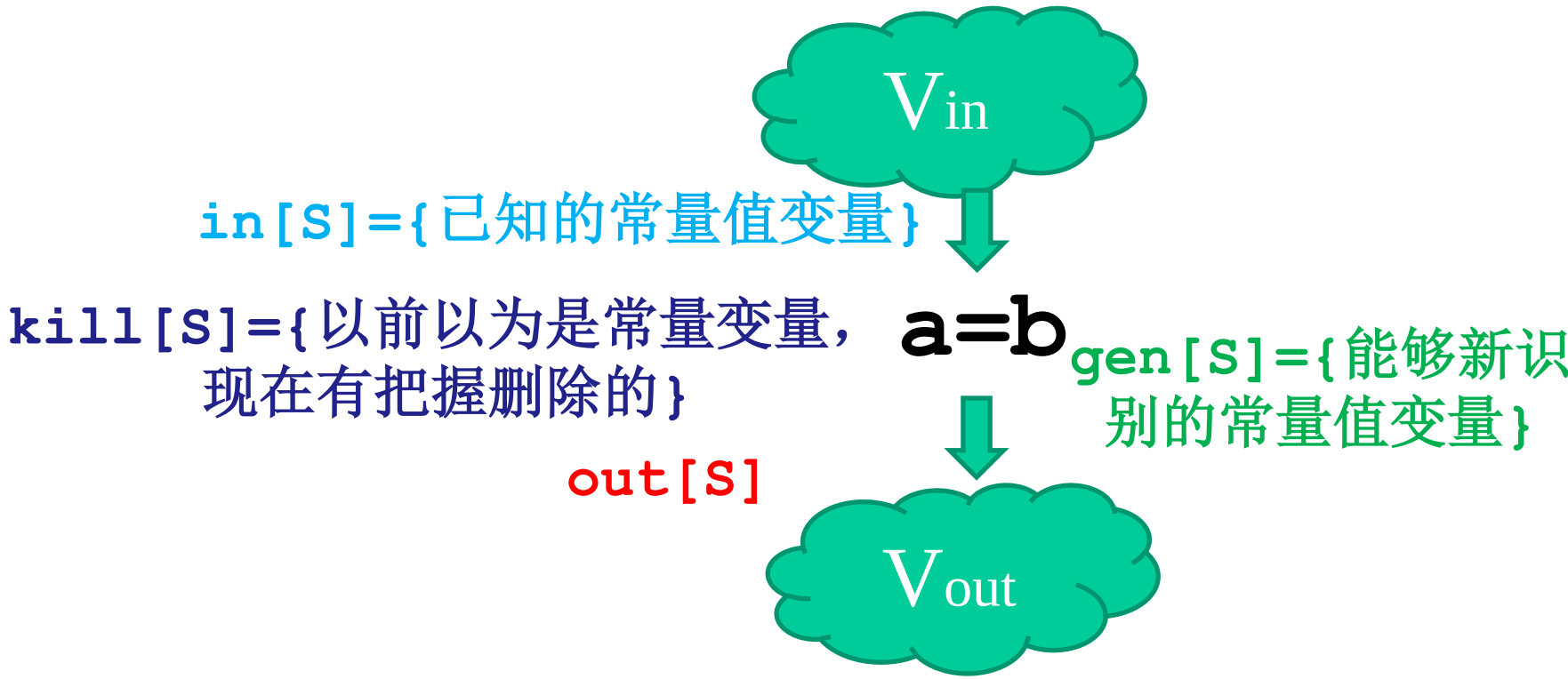
Source: Stanford CS143 (2012)

全局复制传播（常量传播）

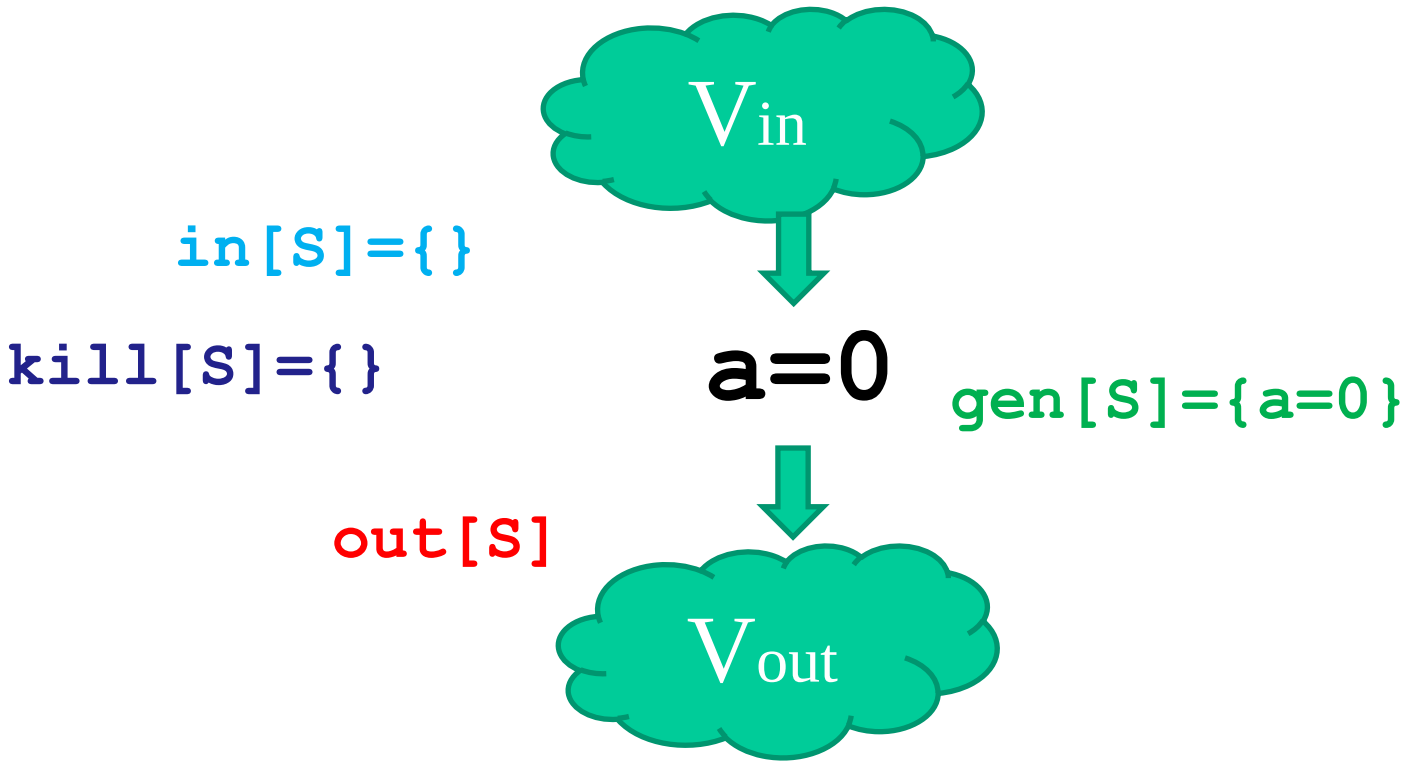
全局常量传播 (Global Constant Progagtion)

- **目的**：寻找所有可以被替换成常量的变量。
  - 思考：a=b; 能替换 a=<常量>的条件?
  - 需要设计怎样的状态集?
  - 分析顺序？从前往后，还是从后往前?
  - 控制流的跳转会带来什么变化?

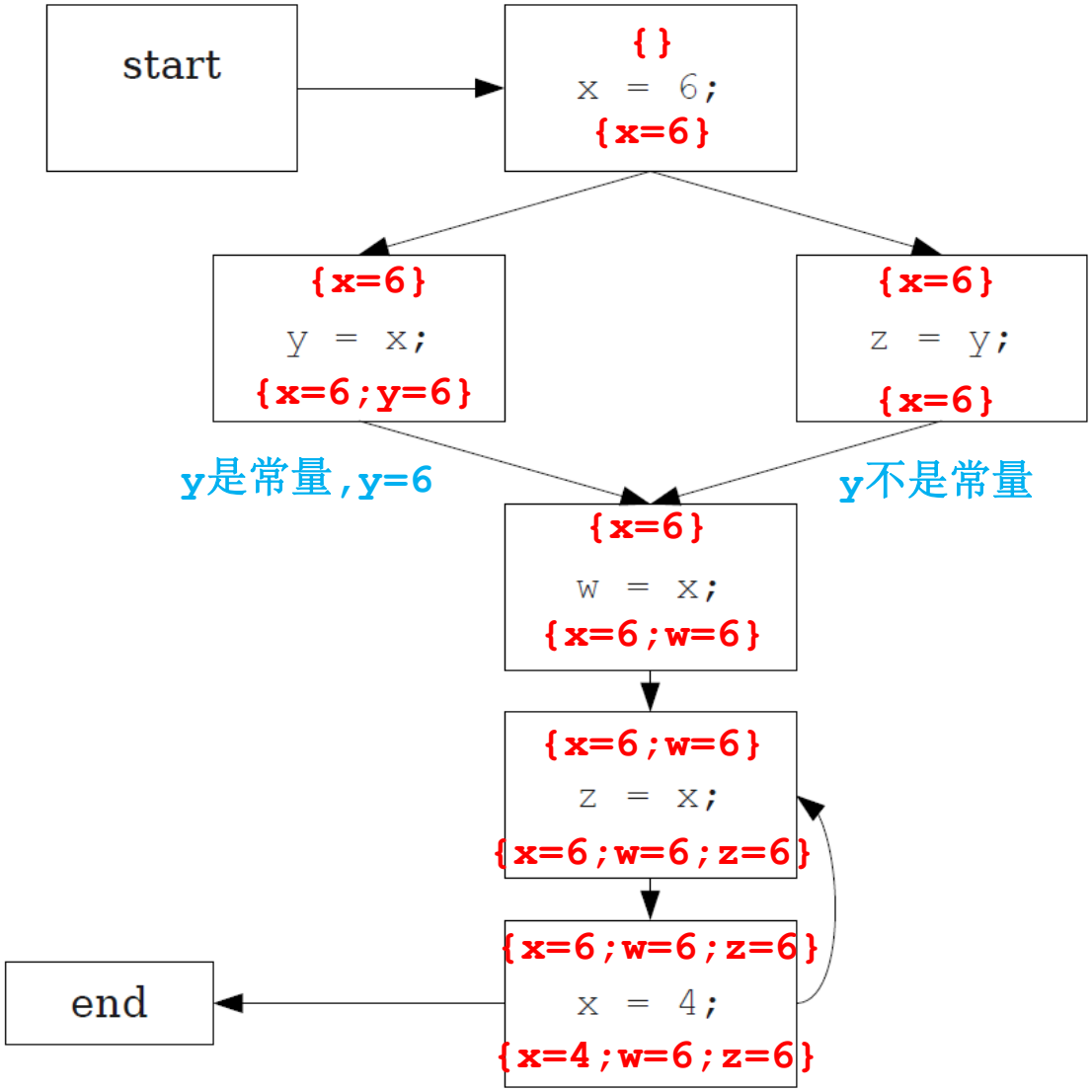
$$out[S]=gen[S]\cup(in[S]-kill[S])$$

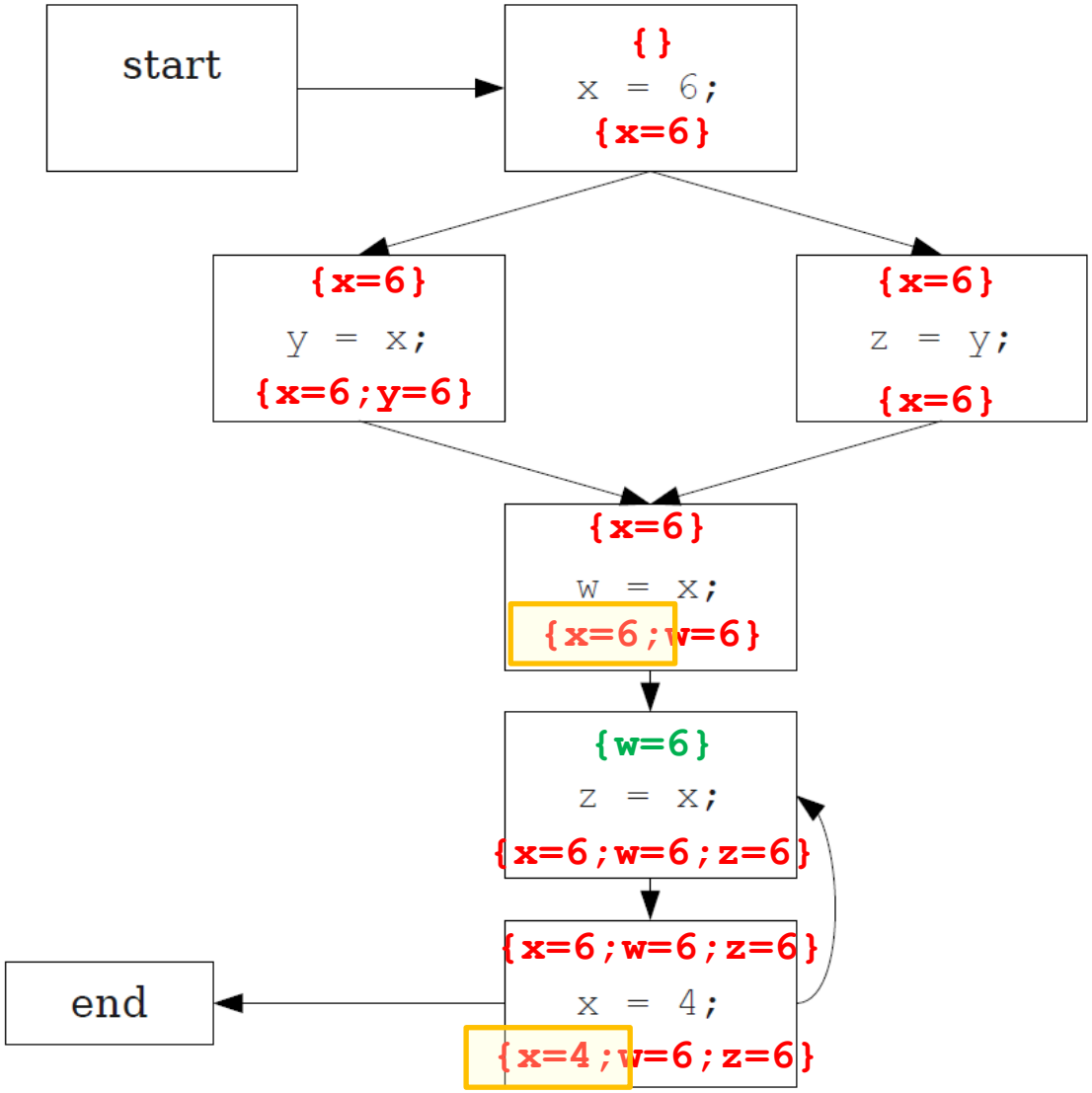


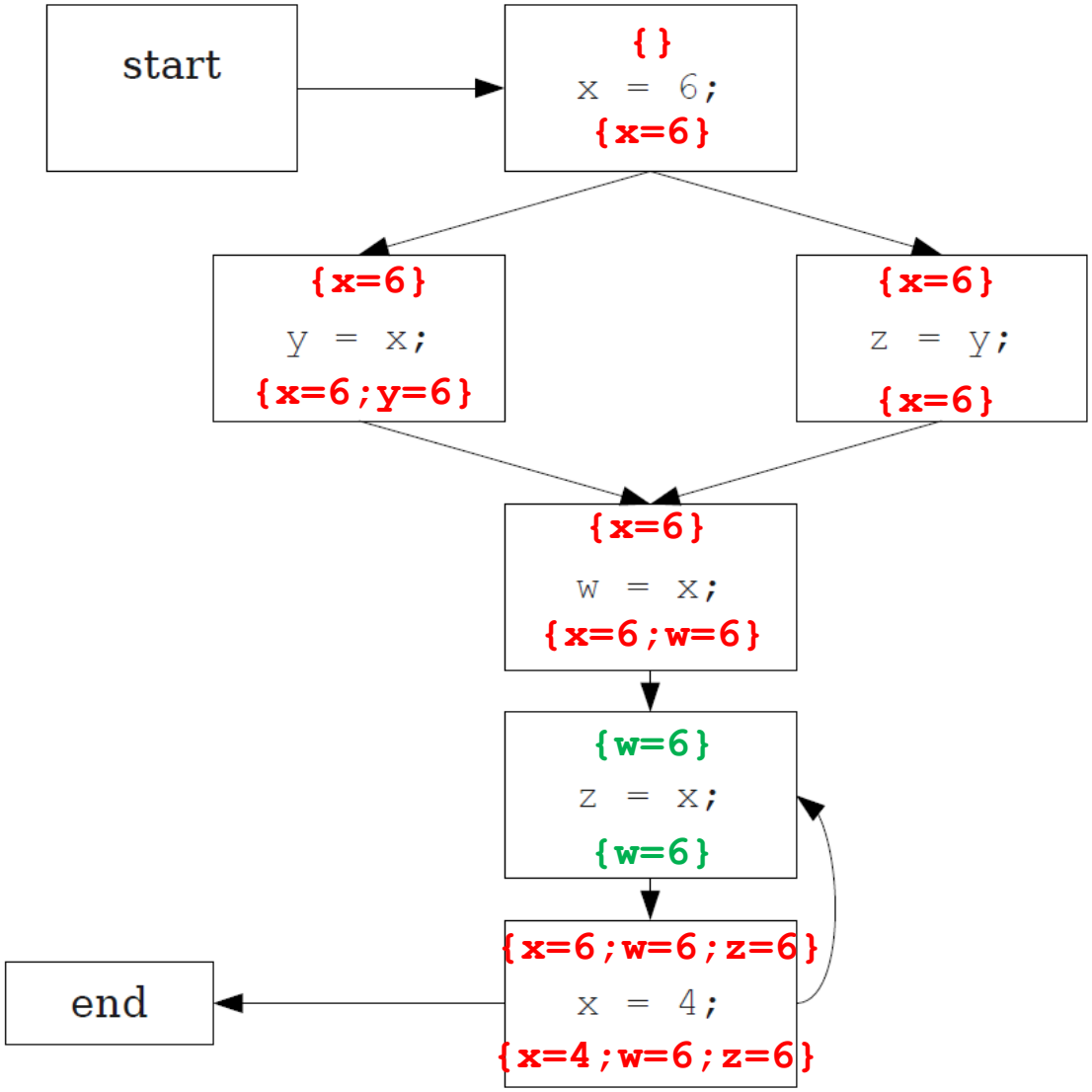
$$out[S]=gen[S]\cup(in[S]-kill[S])$$

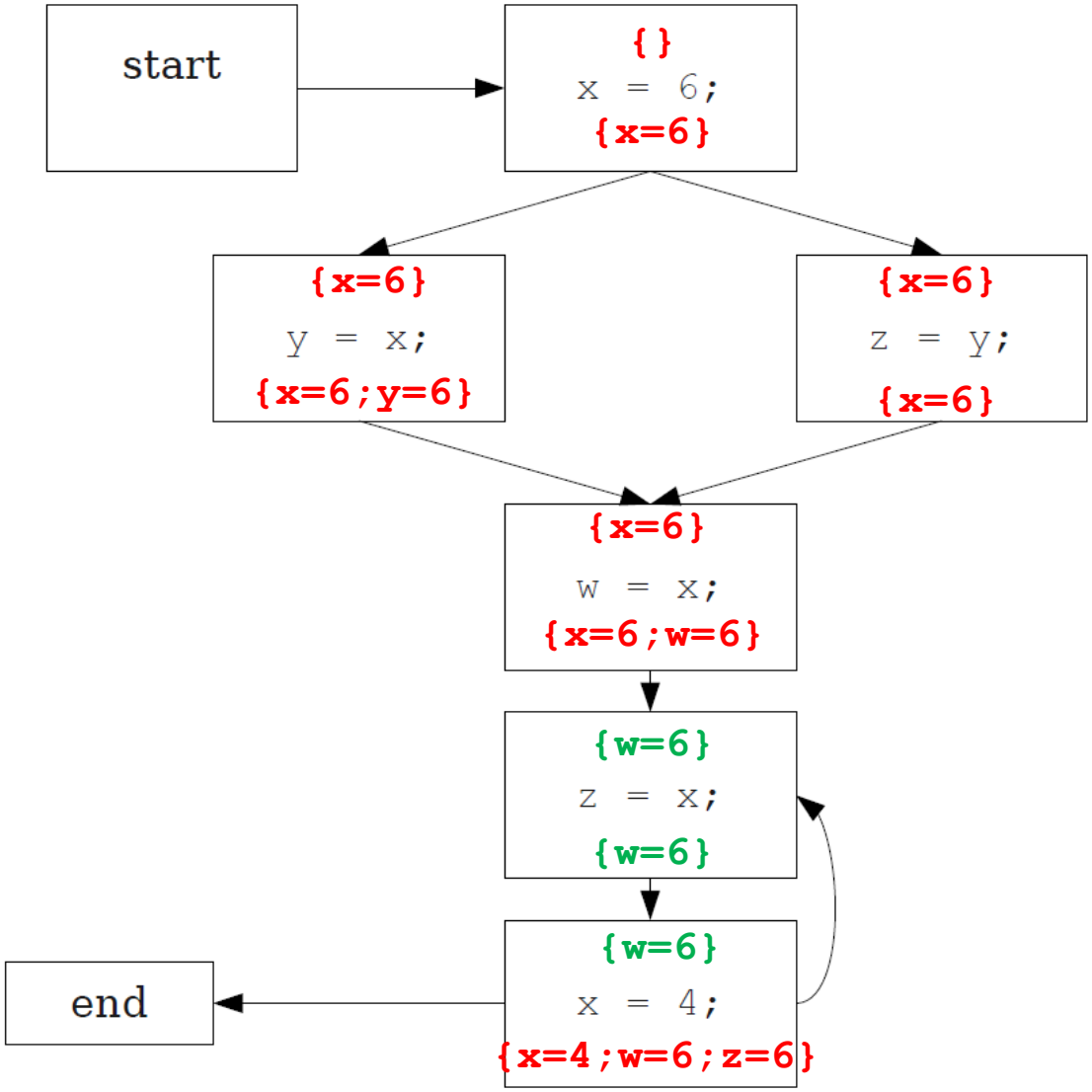


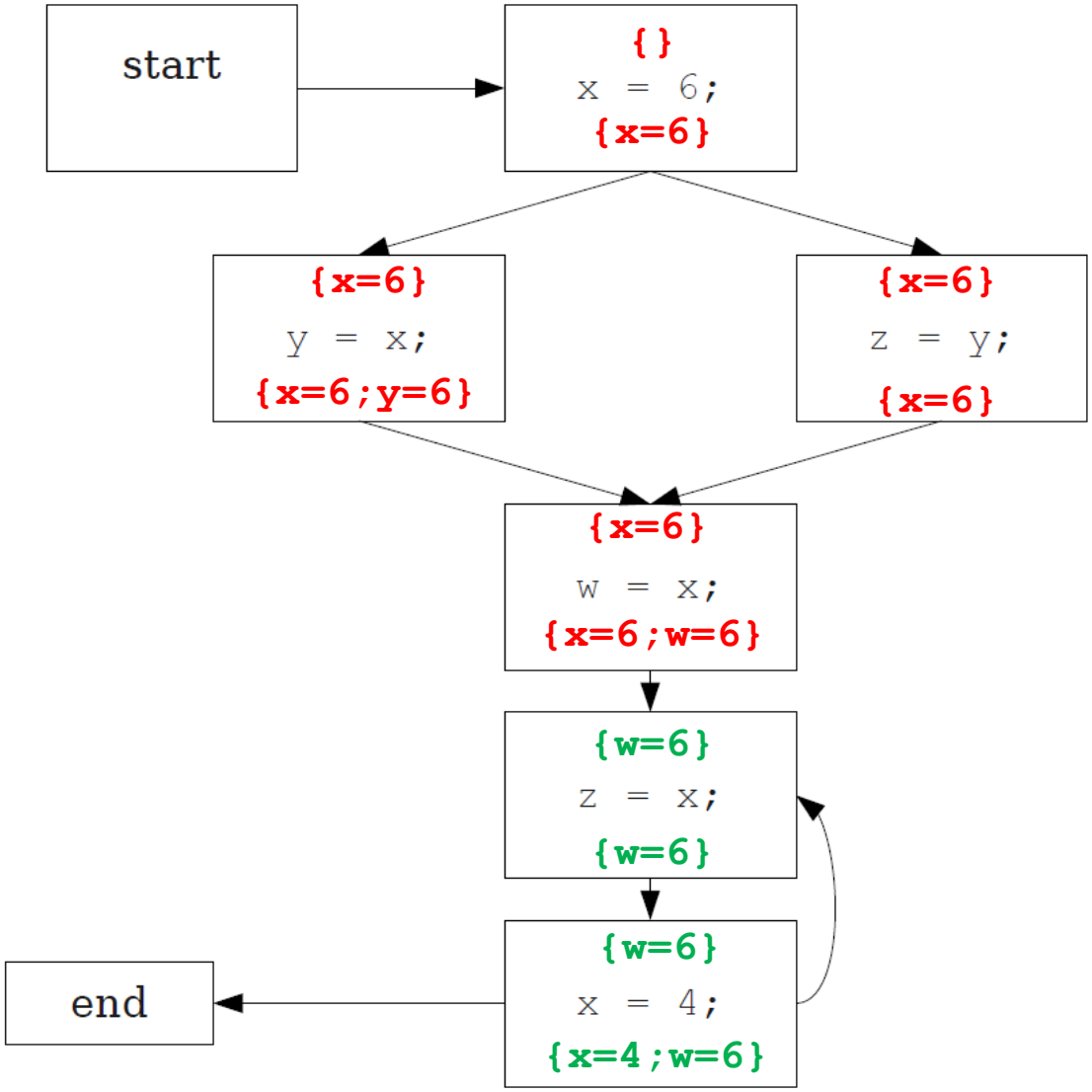
$$out[S]=\{a=0\}\cup(\{\}-\{\})$$
$$=\{a=0\}$$

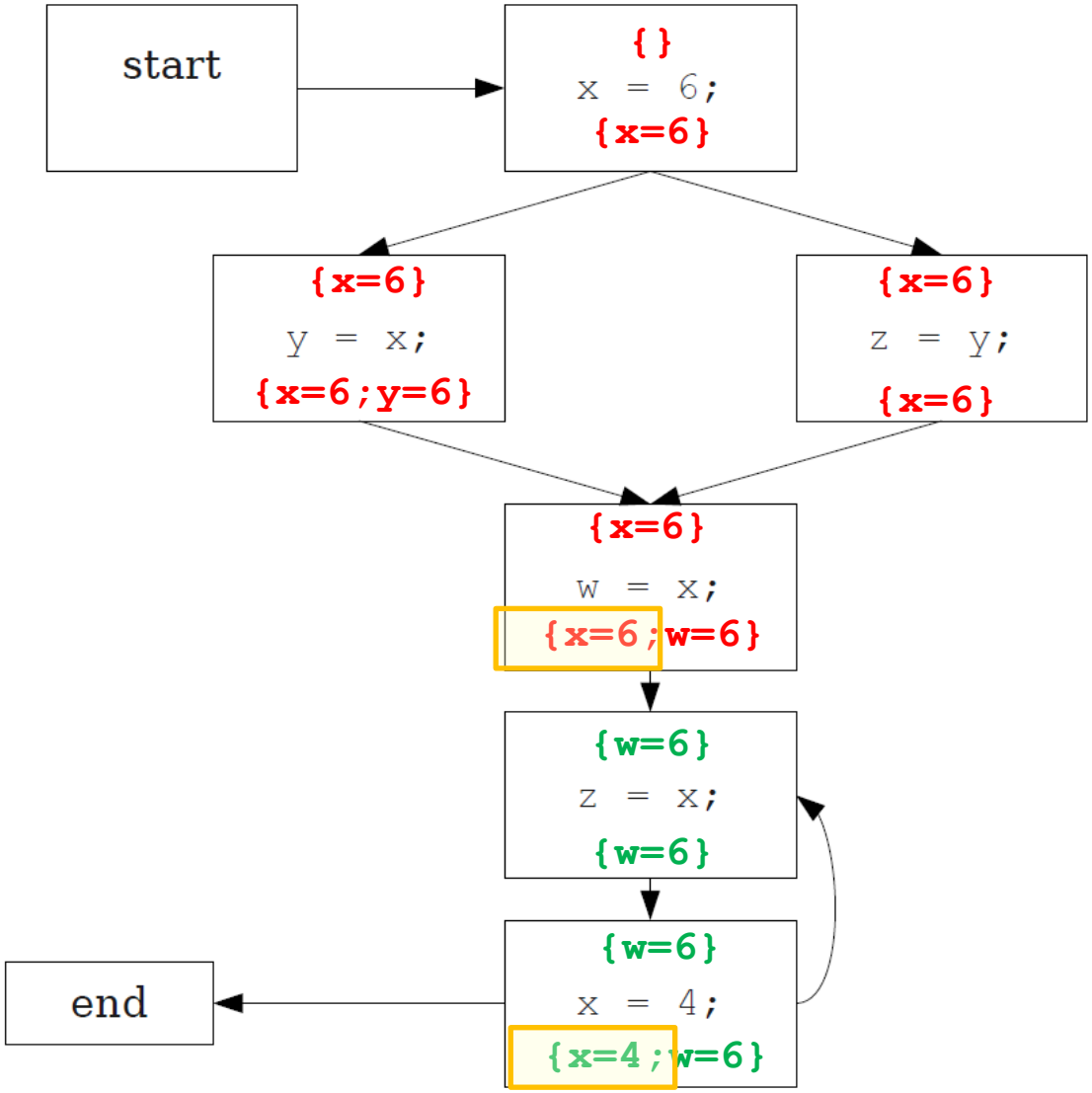

start
{ }
x = 6;
{x=6}
{x=6}
y = x;
{x=6;y=6}
{x=6}
z = y;
{x=6}
y是常量,y=6
y不是常量
{x=6}
w = x;
{x=6;w=6}
{x=6;w=6}
z = x;
{x=6;w=6;z=6}
{x=6;w=6;z=6}
x = 4;
{x=4;w=6;z=6}
end

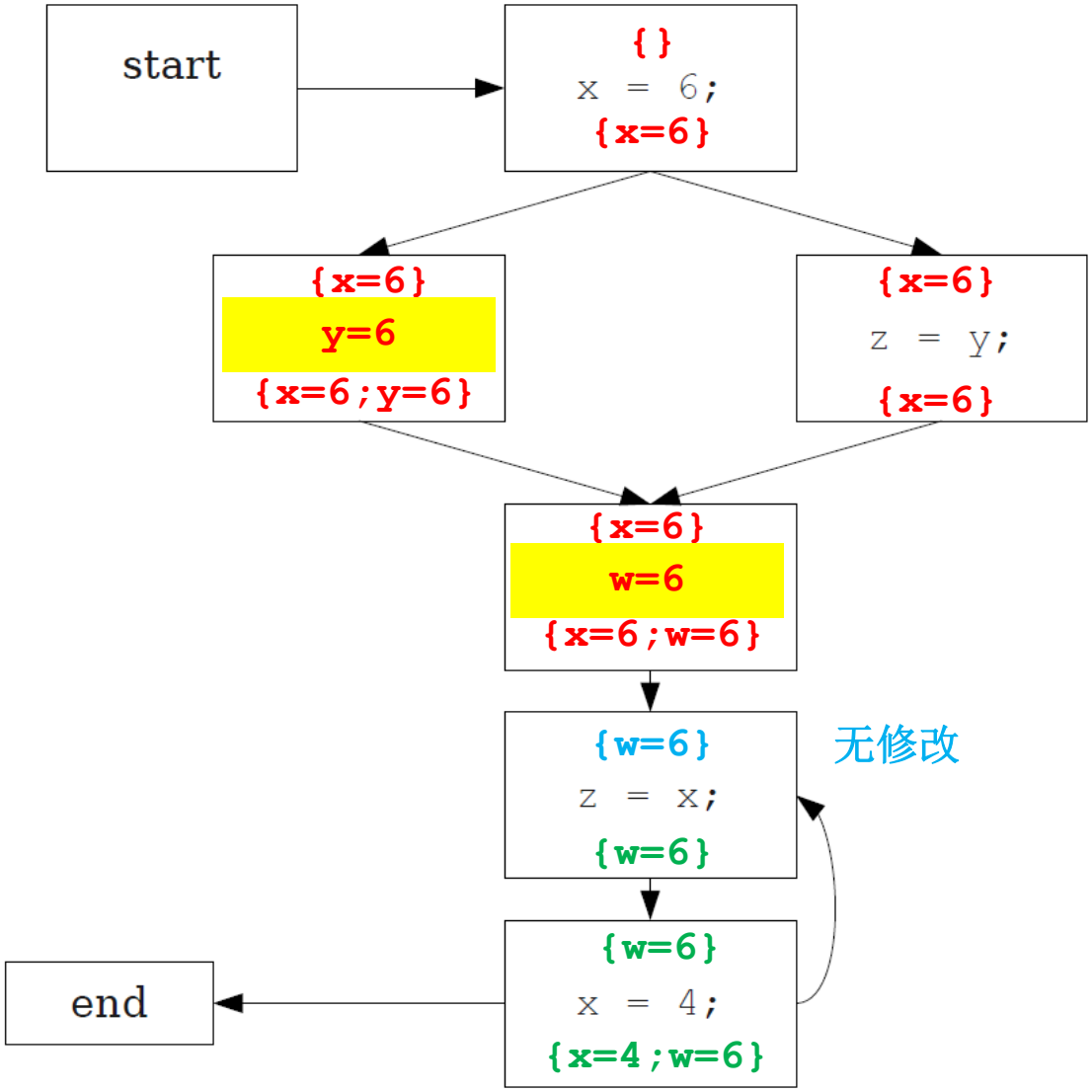

start
{ }
x = 6;
{x=6}
{x=6}
y = x;
{x=6;y=6}
{x=6}
z = y;
{x=6}
{x=6}
w = x;
{x=6;w=6}
{w=6}
z = x;
{x=6;w=6;z=6}
{x=6;w=6;z=6}
x = 4;
{x=4;w=6;z=6}
end

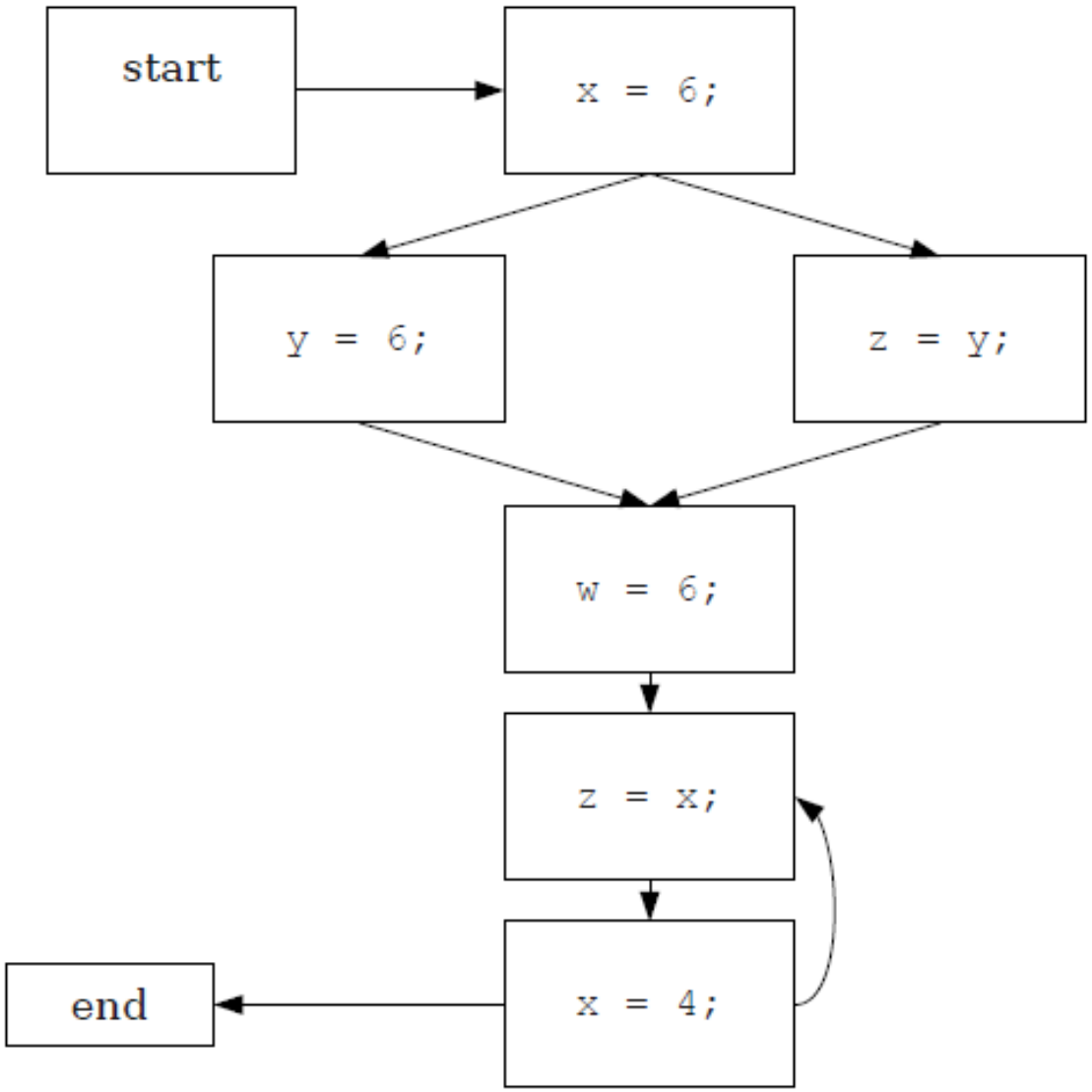

start
{ }
x = 6;
{x=6}
{x=6}
y = x;
{x=6;y=6}
{x=6}
z = y;
{x=6}
{x=6}
w = x;
{x=6;w=6}
{w=6}
z = x;
{w=6}
{x=6;w=6;z=6}
x = 4;
{x=4;w=6;z=6}
end


start
{ }
x = 6;
{x=6}
{x=6}
y = x;
{x=6;y=6}
{x=6}
z = y;
{x=6}
{x=6}
w = x;
{x=6;w=6}
{w=6}
z = x;
{w=6}
{w=6}
x = 4;
{x=4;w=6;z=6}
end


start
{ }
x = 6;
{x=6}
{x=6}
y = x;
{x=6;y=6}
{x=6}
z = y;
{x=6}
{x=6}
w = x;
{x=6;w=6}
{w=6}
z = x;
{w=6}
{w=6}
x = 4;
{x=4;w=6}
end


start
{ }
x = 6;
{x=6}
{x=6}
y = x;
{x=6;y=6}
{x=6}
z = y;
{x=6}
{x=6}
w = x;
{x=6;w=6}
{w=6}
z = x;
{w=6}
{w=6}
x = 4;
{x=4;w=6}
end

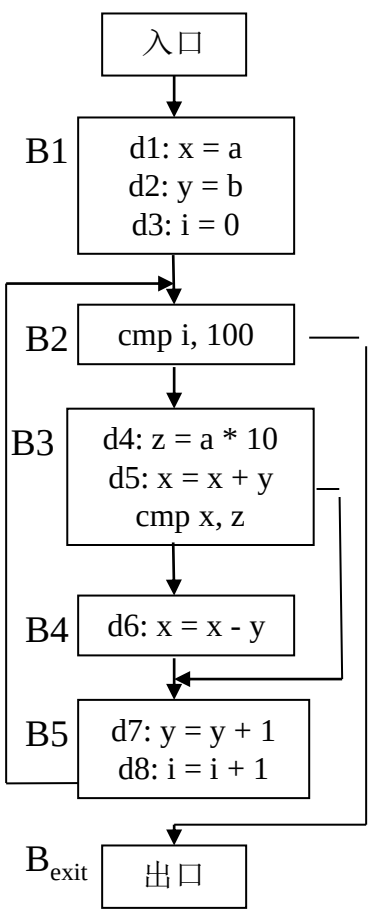

start
{ }
x = 6;
{x=6}
{x=6}
y=6
{x=6;y=6}
{x=6}
z = y;
{x=6}
{x=6}
w=6
{x=6;w=6}
{w=6}
z = x;
{w=6}
无修改
{w=6}
x = 4;
{x=4;w=6}
end


start
x = 6;
y = 6;
z = y;
w = 6;
z = x;
x = 4;
end

代码生成与优化：寄存器

从活跃变量到冲突图

流图



	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B_{exit}	∅	∅

- 变量x, y, i：均定义于B1，在B2~B5入口处均活跃。

注意，x在B3、B4中都被重新定义过，但x被定义前均被使用过，因此其在同一基本块中发生在使用之前的定义仅余B1。

变量y和i的情况类似。

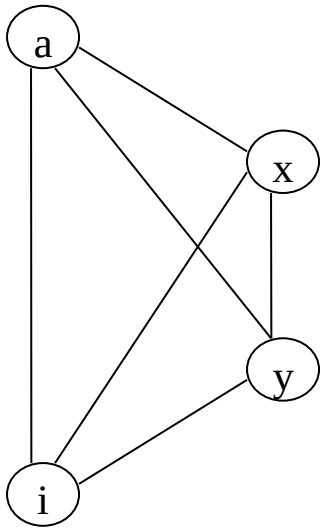
- 变量a：在流图中无定义点，在B1~B5入口处均活跃
- 变量b：在流图中无定义点，在B1入口处活跃
- 变量z：定义于B3，且仅在B3中被使用

冲突图

	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B_{exit}	∅	∅

假设只有跨越基本块活跃的变量才能分配到全局寄存器

并且**活跃范围重合**的变量之间无法共享全局寄存器

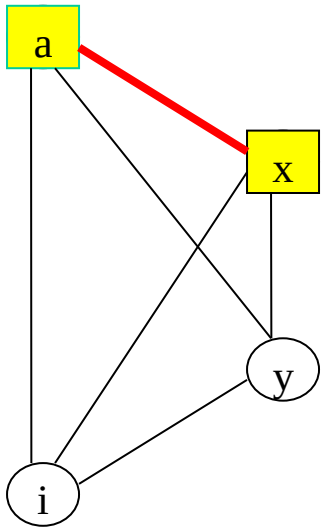


冲突图

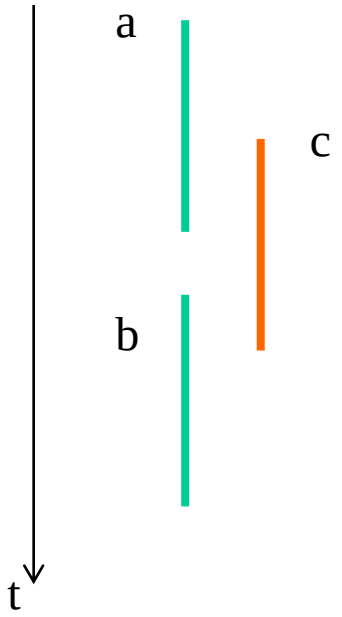
	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B_{exit}	∅	∅

节点a：待分配全局寄存器的变量a
节点x：待分配全局寄存器的变量x

边a-x：变量 a 在变量 x 定义（赋值）处是活跃的

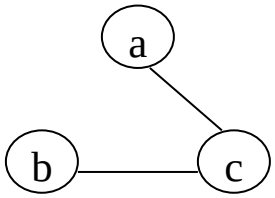


活跃变量冲突的定义



- 变量a和变量b不冲突
- 变量a和变量c冲突
- 变量b和变量c冲突

可以着色

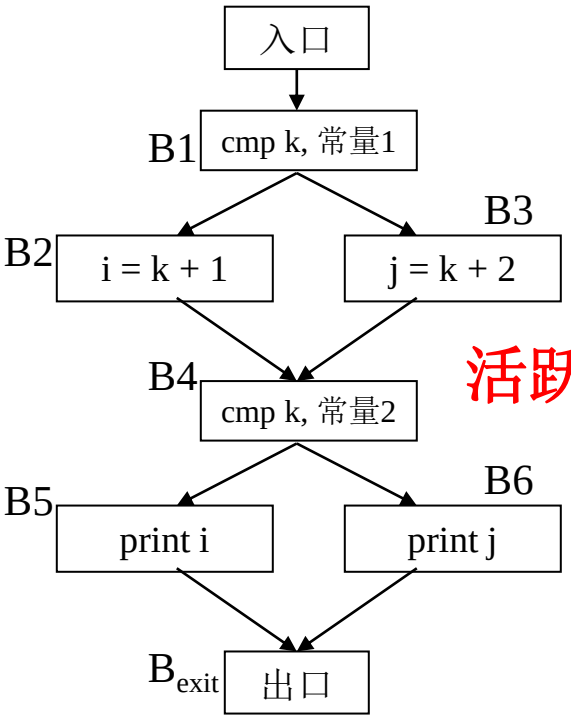


(b)

a和c使用不同的寄存器
b和c使用不同的寄存器

冲突图：连线多画了，不影响程序的正确性；
少连线了，会影响程序的正确性。

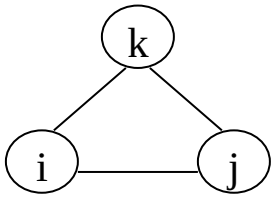
活跃变量冲突的不同定义



(a)

活跃变量：i, j, k

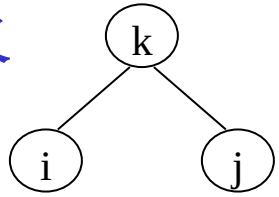
- 变量i和变量j在B2和B3中被分别定义，并在B5和B6中被分别使用
- 根据活跃变量分析结果，i和j一定同时在B4的入口处活跃



(b)

但即使i和j使用同一寄存器，程序运行结果仍符合语义

冲突图中两个节点（变量）间存在边的条件约束为：
其中一个变量在另一个变量的定义点处活跃

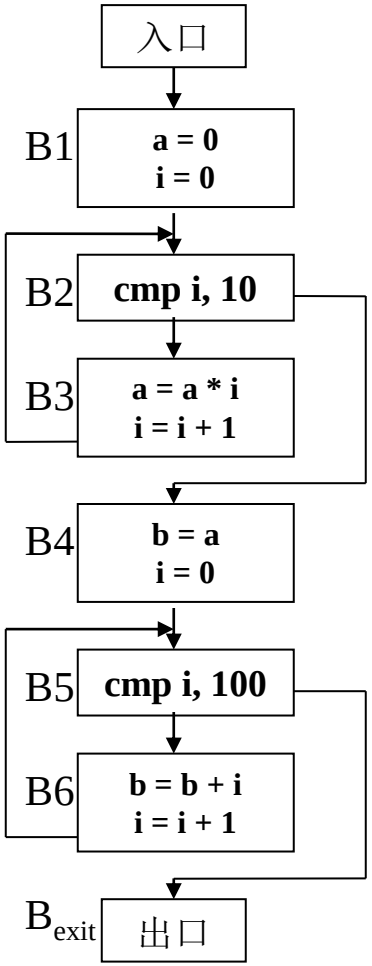


(c)

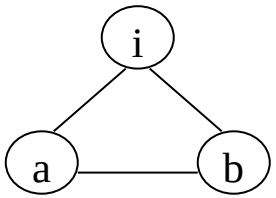
关于变量冲突的判断

- **两个变量中的一个变量在另一个变量定义（赋值）处是活跃的**，它们就是冲突的。
  - 算法一：在每一个变量的定义点计算活跃变量
  - 算法二：计算基本块入口处的活跃变量（in的集合），这些变量在该基本块中的定义点活跃，因而冲突。之后，在基本块内部，进一步计算每个定义点的活跃变量（基本块范围内计算）
  - 基本块内是线性的，可降低计算复杂度

被多次定义的变量和冲突图



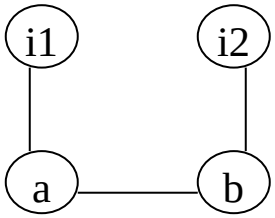
(a)



(b)

变量i在第一个循环中被定义和使用，执行第二个循环前，i被重新定义和使用

（三色)



(c)

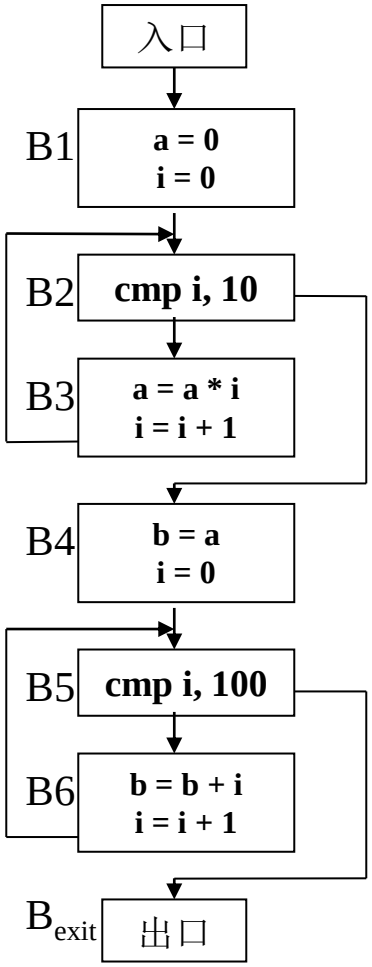
变量a或变量b伴随着变量i一同使用

（两色)

变量i在第一个循环和第二个循环中，是否可以使用不同的全局寄存器?

Define-use链

- **变量的定义-使用链（Define-Use链）**，变量的某一定义点，以及所有可能使用该定义点所定义变量值的使用点所组成的一个链



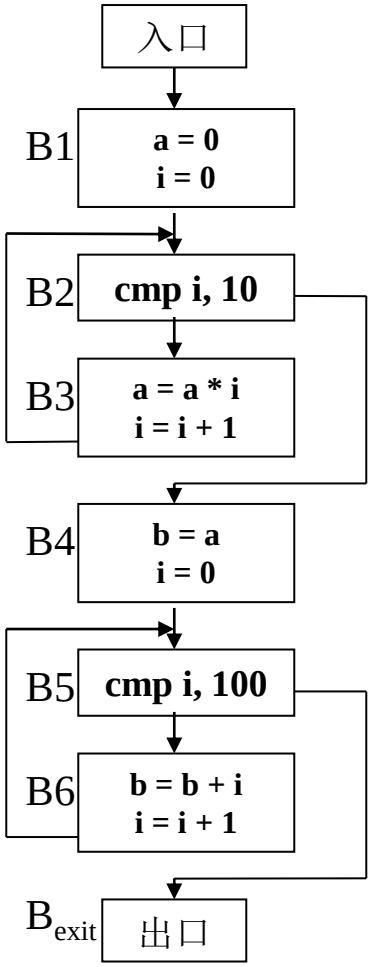
变量a： L1 {<B1, 1>, <B3, 1>, <B4, 1>}
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b： L3 {<B4, 1>, <B6, 1>}
L4 {<B6, 1>, <B6, 1>}

变量i： L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}
L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

Define-use链

- **变量的定义-使用链（Define-Use链）**，变量的某一定义点，以及所有可能使用该定义点所定义变量值的使用点所组成的一个链



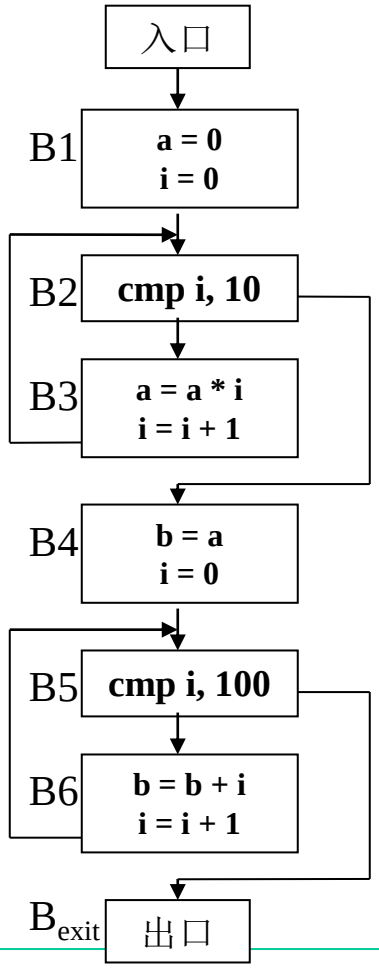
变量a：L1 {<B1, 1>, <B3, 1>, <B4, 1>}
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b：L3 {<B4, 1>, <B6, 1>}
L4 {<B6, 1>, <B6, 1>}

变量i：L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}
L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

可以发现：L5、L6和L7、L8是没有关系的。
后面的网，可以发现同一个变量的定义使用链分裂了，是两个网

- 同一变量的多个定义-使用链，如果它们拥有某个同样的使用点，则合并为同一个网



(a)

变量a: L1 {<B1, 1>, <B3, 1>, <B4, 1>}
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b: L3 {<B4, 1>, <B6, 1>}
L4 {<B6, 1>, <B6, 1>}

变量i: L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}
L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

变量a: W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b: W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

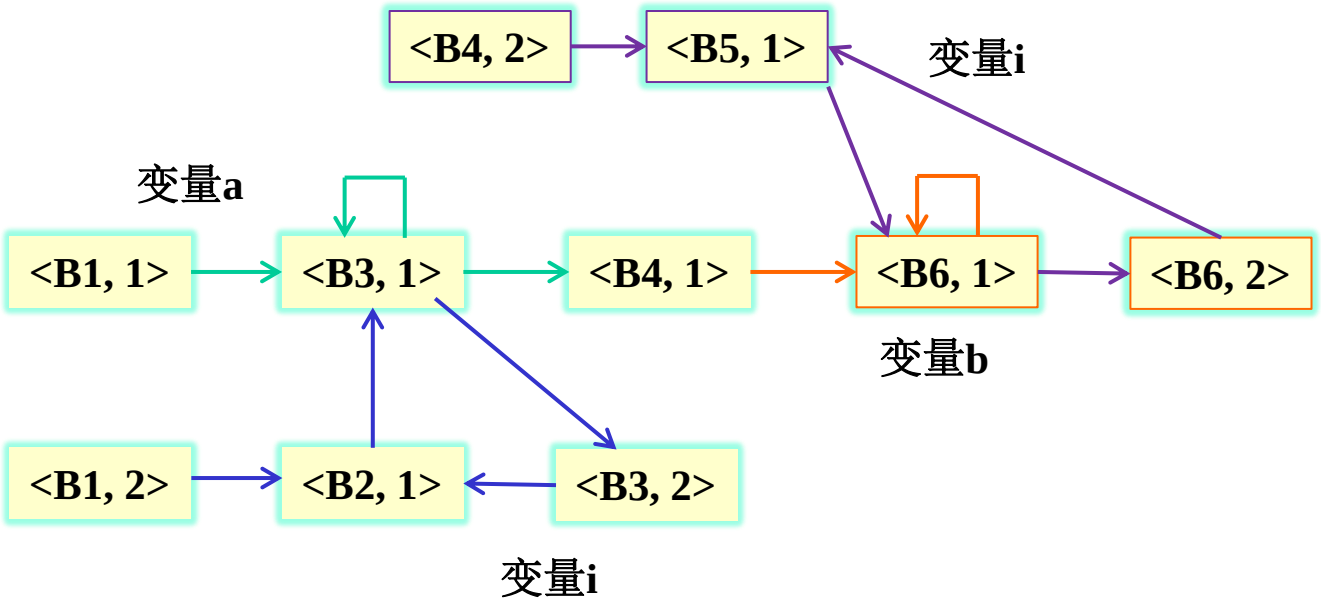
变量i: W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}
W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}

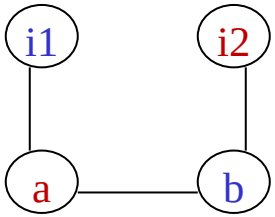
变量a： W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b： W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

变量i： W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}

W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}





循环优化

循环优化

80/20经验规则： “程序运行时间的80%是由仅占源程序20%的部分执行的”。这20%的源程序就是循环部分，特别是多重循环的最内层的循环部分。

```
for (i=1;i<=10;i++) {
  for (j=1;j<=100;j++) {
    x := x+0;
    y := 5+7+x;
  }
}
```

优化一条，少10*100次运算

除了对循环体进行优化，还有专用于循环的优化

（1）循环不变式的代码外提

不变表达式：

不随循环控制变量改变而改变的表达式或子表达式。

如：

```
for(i=E1; i<=E3 ; i+=E2) {
    S=0.2*3.1416*R;
    P=0.35*i;
    V=S*P;
}
```

可以外提

不能外提

如　　while　　...　　do

x := ...　　(b*b - 4.0*a*c) ...

若a,b,c的值在该循环中不改变时，则可将循环不变式移到循环之外，即变为：

t_1 := b*b - 4.0*a*c

while　　...　　do

x:= ...(t_1) ...

从而减少计算次数——也称为频度削弱

（2）循环展开

将构成循环体的代码（不包括控制循环的测试和转移部分），**重复产生许多次**（这可在编译时确定），**而不仅仅是一次，以空间换时间。**

例 PL/1中的初始化循环

```
DO    I = 1    TO    30
      A[ I ] = 0.0
END
```

展开

```
      I := 1
L1:   IF  I > 30  THEN
      GOTO    L2
      A[ I ] = 0.0
      I = I+1
      GOTO    L1
L2:
```

代码5条语句

共执行5*30

条语句

```
A[1] = 0.0
A[2] = 0.0
……
A[30] = 0.0
```

30条语句

（指令）执行

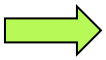
也是30条语句

- 循环一次执行5条语句才给一个变量赋初值。展开后，一条语句就能赋一个值，运行效率高。
- 优化在生成代码时进行，并不是修改源程序。
- 必须知道循环的终值，初值及步长。
- 但并不是所有展开都是合适的。如上例中循环展开后节省执行了转移和测试语句：2*30=60语句 (其实，还不止节省60条）。

∴增加29条省60条

但若循环体中不是一条而是40条语句，则展开后将有40*30条=1200，但省的仍是60条，就不算优化了。

∴判断准则：1. 主存资源丰富
处理机时间昂贵
2. 循环体语句越少越好



循环展开有利
（大型机）

```
DO    I = 1    TO    30
          A[ I ] = 0.0
END
```

实现步骤：

1. 识别循环结构，确定循环的初值，终值和步长。
2. 判断。以空间换时间是否合算来决定是否展开。
3. 展开。重复产生循环体所需的代码个数。

比较复杂：

∵在对空间与时间进行权衡时，还可以考虑一种折衷的办法，即部分展开循环。如上例展为：

```
DO    I = 1   TO    30    BY    3
        A[I] = 0.0
        A[I+1] = 0.0
        A[I+2] = 0.0
END;
```

空间只多二条，但省了20次测试时间（只循环10次）

(3) 归纳变量的优化和条件判断的替换

在每一次执行循环迭代的过程中，若某变量的值固定增加（或减少）一个常量值，则称该变量为**归纳变量(induction variable)**。

即若当前执行循环的第j次迭代。归纳变量的值应为c*j+c′，这里c和c′都是**循环不变式**。

```
For i := 1 to 10 do
    a[i] := b[i] + c[i]
```

```
for   i:= 1      to      10      do
a[i] := b[i]   +  c[i]
```

```
1)        i := 1
2)        labb:
3)        if  i > 10      goto   labe
4)        t1 :=  4*i
5)        t2 :=  b [ t1]
6)        t3 :=  4*i
7)        t4 :=  c [ t3]
8)        t5 := t2 +  t4
9)        t6 :=  4*i
10)       a[t6] := t5
11)       i := i+1
12)       goto      labb
13) labe:
```

优化：

```
1)        u := 4
2)        labb:
3)        if u >40   goto    labe
4)        tb := b [u]
5)        tc := c [u]
6)        t  :=  tb + tc
7)        a [ u] := t
8)        u := u+4
9)        goto          labb
10)       labe :
```

中间变量t1， t3， t6 都是归纳变量　　t1 := 4*i， t3 := 4*i， t6 := 4*i

(4) 其它循环优化方法

- 把多重嵌套的循环变成单层循环。
- 把n个相同形式的循环合成一个循环等。

对于循环优化的效果是很明显的。

(5) in_line 展开

把过程（或函数）调用改为in_line展开可节省许多处理过程（函数）调用所花费的开销。

```
procedure m(i,j:integer;  max:integer);
  begin if i > j then max:=i else max:=j end;
```

若有过程调用 `m(k,0,max);`

则内置展开后为：

```
if k>0 then max:=k else max:=0;
```

省去了函数调用时参数压栈，保存返回地址等指令。

这也仅仅限于简单的函数。

作业： 14章 6题

谢谢！