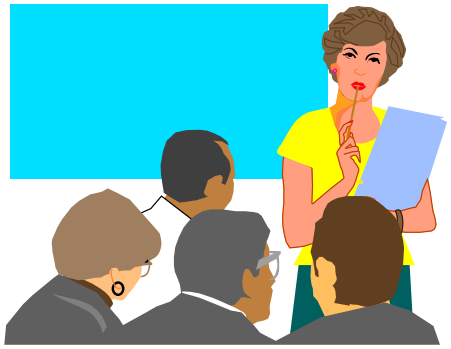
编译技术



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

局部优化：死代码消除 (Dead Code Elimination)

```
a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Source: Stanford CS143 (2012)

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

Source: Stanford CS143 (2012)

- 消除死代码（Dead Code）
- 变量的活性（Liveness of an Variable）

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

```
a = b + c
```

$$L=\{\ldots\}$$

局部优化：一个完整的例子

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Source: Stanford CS143 (2012)

Source: Stanford CS143 (2012)

全局优化：数据流分析

- 消除死代码（Dead Code）
- 变量的活性（Liveness of an Variable）

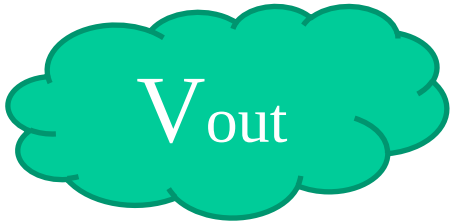
$$(L - \{a\}) \cup \{b,c\}$$

```
a = b + c
```

$$L=\{\ldots\}$$

- 消除死代码（Dead Code）
- 变量的活性（Liveness of an Variable）


```
a = b + c
```



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

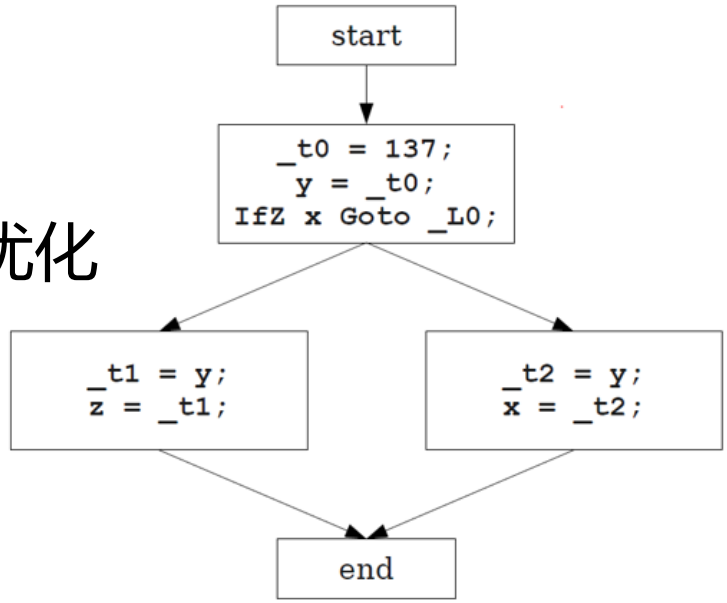
14.3 全局优化

和局部优化的区别：

局部优化在一个基本块内
没有控制流

全局优化处理的是基本块之间的优化
受到控制流的影响

1. 分支
2. 循环



14.3 全局优化

主要手段：数据流分析

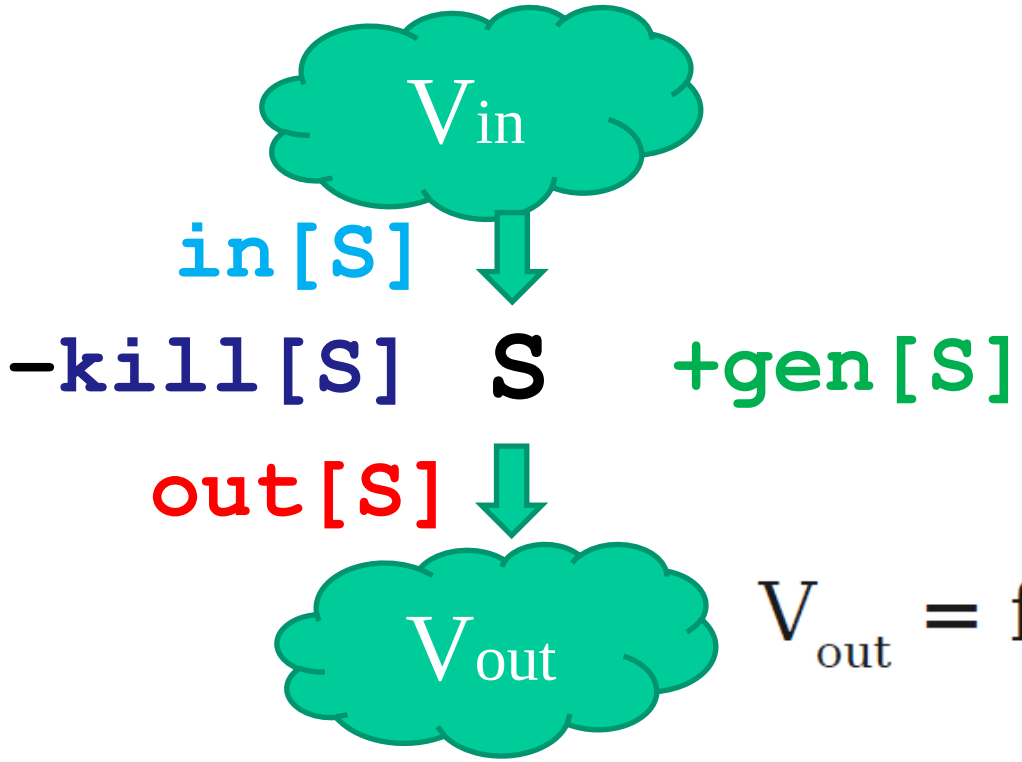
- **用于获取数据在程序执行路径中如何流动的有关信息。**
- 例如：
  - 某个变量在某个特定的执行点（语句前后）是否还“存活”
  - 某个变量的值，是在什么地方定义的
  - 某个变量在某一执行点上被定义的值，可能在哪些其他执行点被使用
- **是全局优化的基础**

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

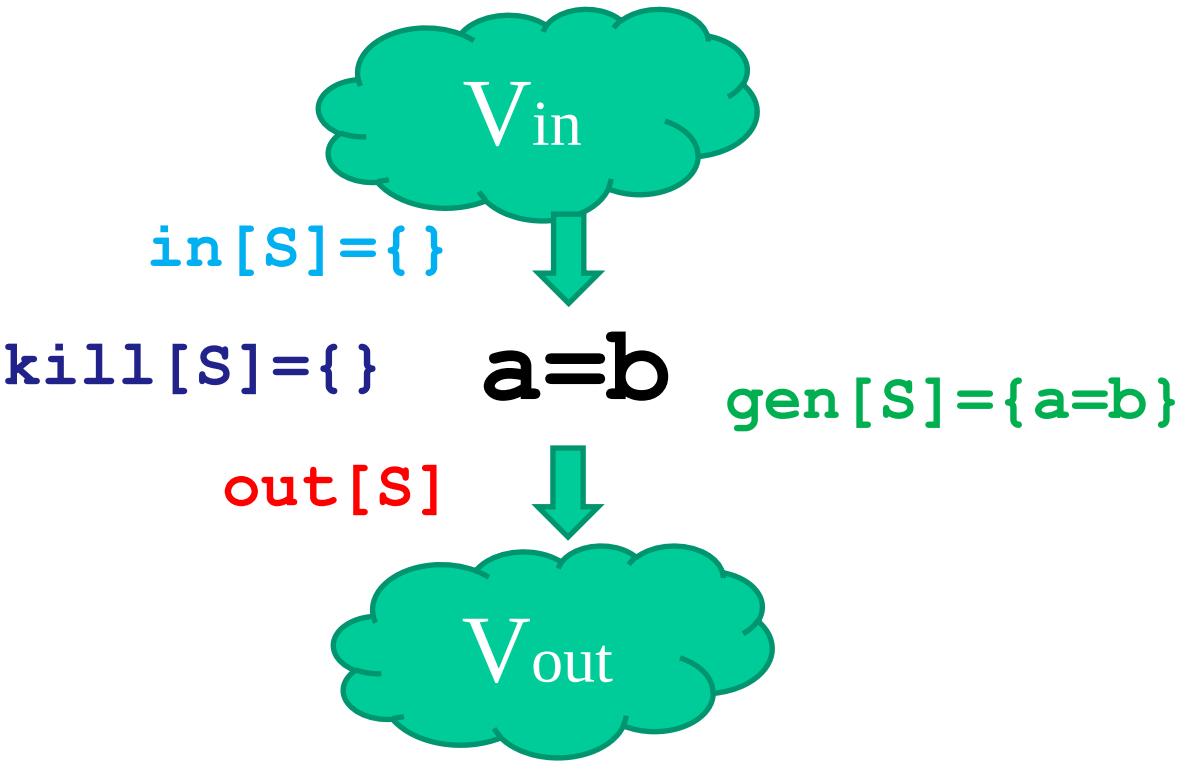
out[S]=gen[S]∪(in[S]-kill[S])



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

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

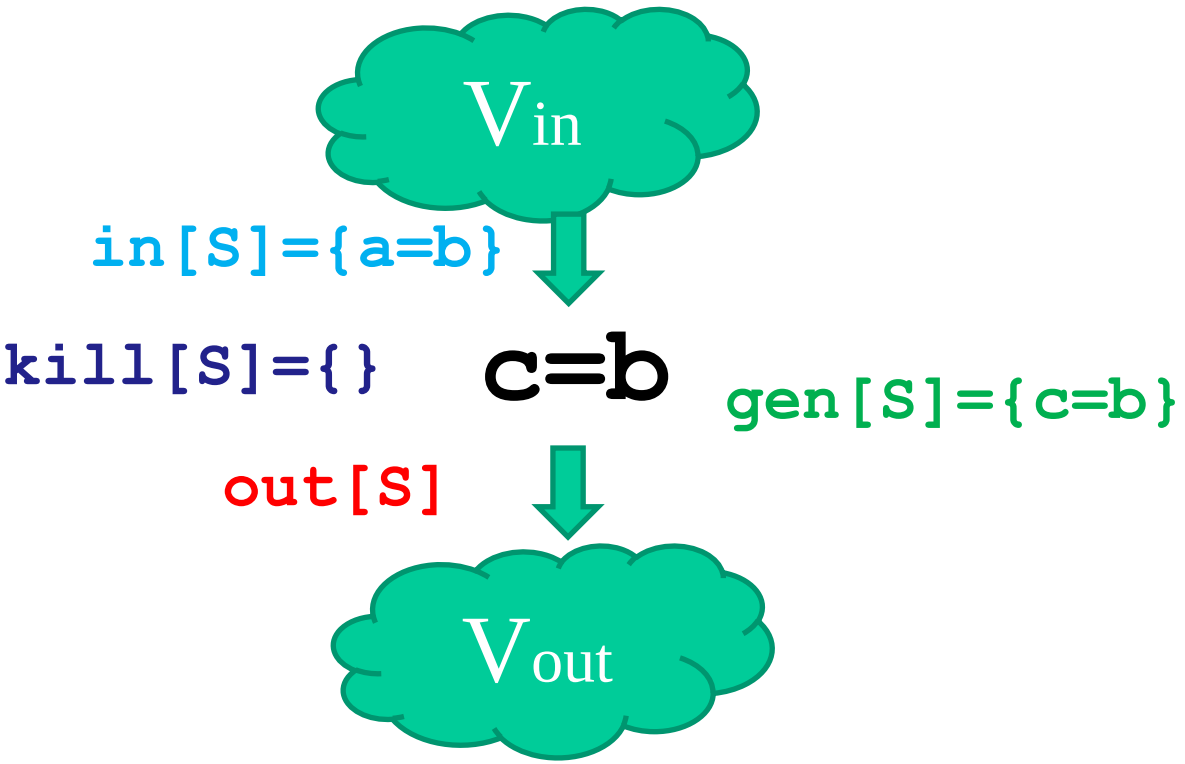
```
a=b;
c=b;
d=a+b;
e=a+b;
d=b;
f=a+b;
```



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

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

```
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]∪(in[S]-kill[S])

a=b;

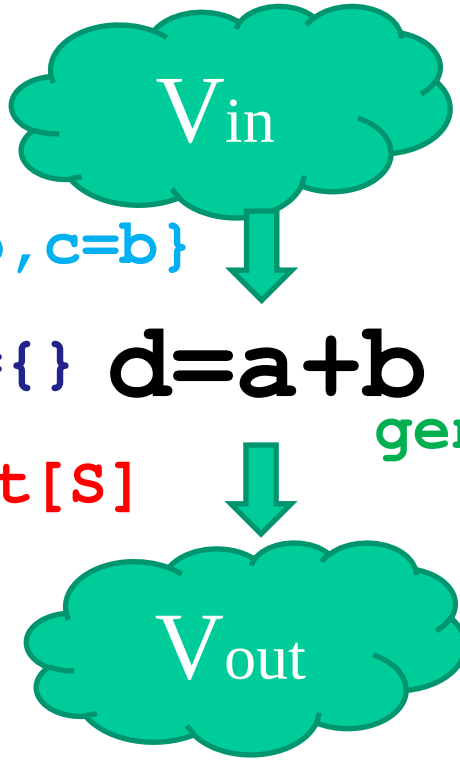
c=b;

d=a+b;

e=a+b;

d=b;

f=a+b;



out[S]={d=a+b} ∪({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)

数据流方程求解：3个关键因素

- 当前语句**产生和注销的信息取决于**需要解决的具体问题：可以由`in[S]`定义`out[S]`，也可以反向定义，由`out[S]`定义`in[S]`
- 由于数据是**沿着程序的执行路径**，因此数据流分析的结果受到**程序控制结构**的影响
- 代码中出现的诸如过程调用、指针访问以及数组成员访问等操作，对定义和求解一个数据流方程都会带来不同程度的困难

程序的状态

- **程序的执行过程：程序状态的变换过程**
  - 程序状态由程序中的变量和其它数据结构组成
  - 每一条执行指令都可能改变程序的状态
- 通过数据流分析，可以了解程序的状态。

全局优化：可达定义分析（reaching def）

到达定义（reaching definition）分析

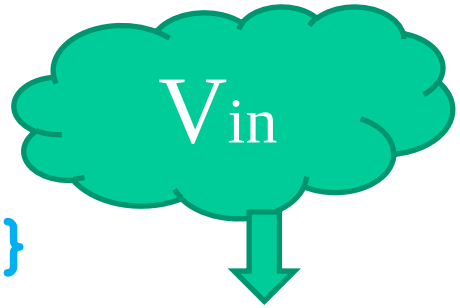
- 通过到达定义分析，希望知道：
  - 在程序的**某个静态点p**（例如某个代码、基本块）执行前后）
  - 某个变量可能出现的值都是在哪里被定义的?

到达定义（reaching definition）分析

- 通过到达定义分析，希望知道：
  - 在程序的**某个静态点p**（例如某个代码、基本块）执行前后）
  - 某个变量可能出现的值都是在哪里被定义的?

- 例如：在p处对变量x的引用是否有效?
  - 如果从定义点d出发，存在一条路径达到p，且在该路径上，不存在对该变量的其他定义语句，则认为“变量x的定义点d到达（可达）静态点p”
  - 如果路径上存在对该变量的其他赋值语句，那么路径上的前一个定义点就被路径上的后一个定义点“杀死（kill）”，或者消除了

out[S]=gen[S]∪(in[S]-kill[S])



in[S]={}

-kill[S]={}

d1:x=a

+gen[S]={$d1_x$}

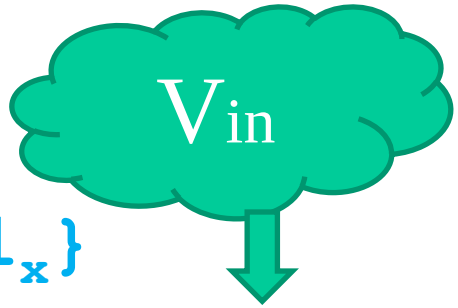
out[S]

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

out[S]={$d1_x$} ∪({}-{})

= {$d1_x$}

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



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

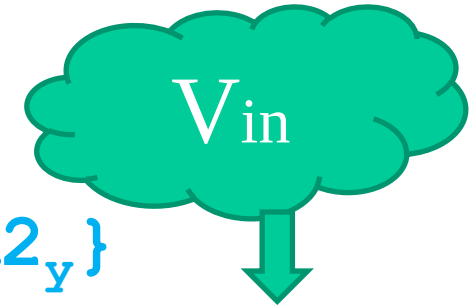
$-kill[S]=\{\}$ d2 : y=b $+gen[S]=\{d2_y\}$

$out[S]$

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

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

out[S]=gen[S]∪(in[S]-kill[S])



in[S]={$d1_x$,$d2_y$}

-kill[S]={$d1_x$} d3:x=0 +gen[S]={$d3_x$}

out[S]

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

out[S]={$d3_x$} ∪({$d1_x$,$d2_y$}-{$d1_x$})

= {$d3_x$,$d2_y$}

回忆：全局优化

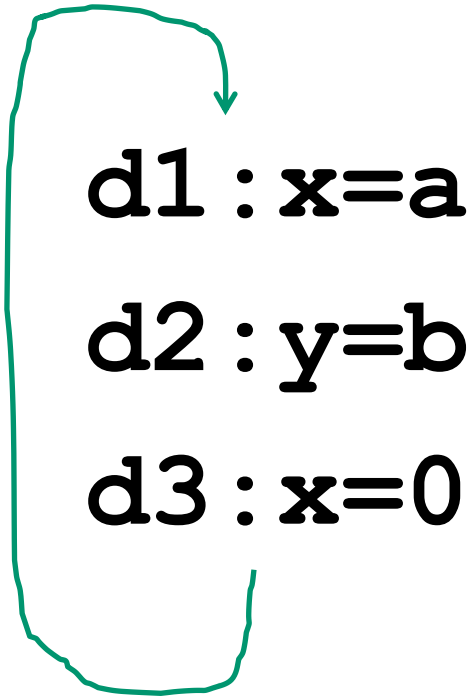
和局部优化的区别：

局部优化在一个基本块内
没有控制流

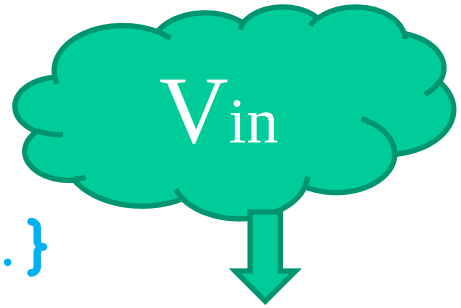
全局优化处理的是基本块之间的优化
受到控制流的影响

1. 分支

2. 循环



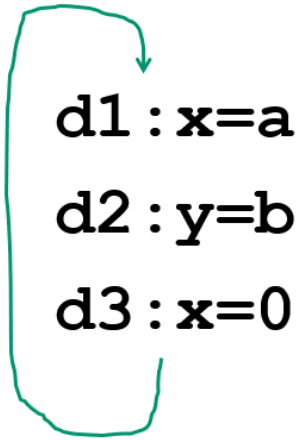
$$out[S]=gen[S]\cup(in[S]-kill[S])$$



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

$-kill[S]=\{d3_x\}$ **d1 : x=a** $+gen[S]=\{d1_x\}$

$out[S]$



$$out[S]=\{d1_x\}\cup(\{\dots\}-\{d3_x\})$$

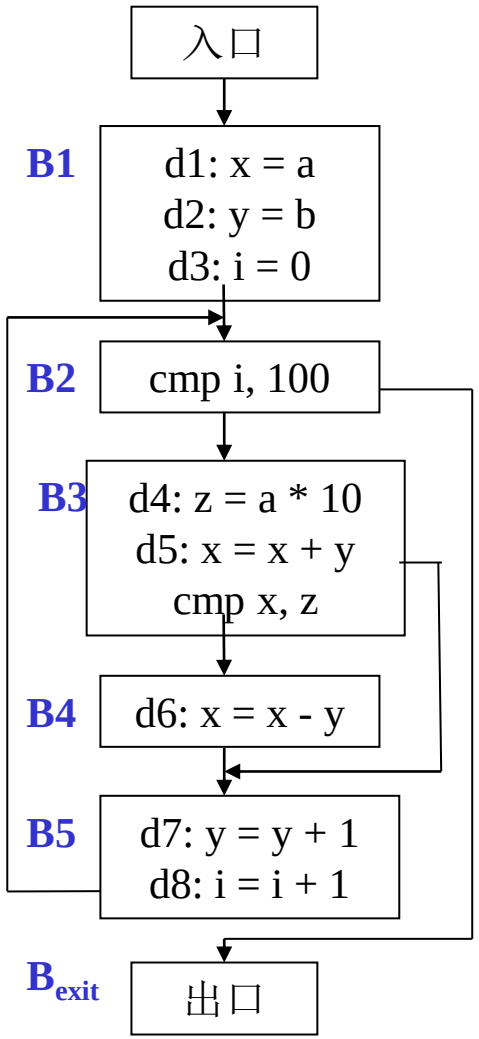
- 对于基本块中的某一条中间代码：

  d1: u = v op w,　　v和w为变量，op为操作符

- 代码对应的到达定义数据流方程是

  out[d1]=gen[d1]∪(in[d1]-kill[d1])

- 其中
  - **gen[d1] = {d1}**，表明该语句产生了一个定义点（定义点 d1 定义了变量 u）
  - **kill[d1]** 是程序中所有对变量u定义的其他定义点的集合**（包括d1之前或之后的定义点）**
  - 对于该代码在同一基本块中紧邻的后继代码，假设其为d2，**in[d2]**等价于**out[d1]**




```
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}
```

d1~d8 八个定义点

基本块B的到达定义数据流方程

out[B]=gen[B]∪(in[B]-kill[B])

- **in[B]** 为进入基本块时的数据流信息
- **kill[B]** = kill[d1] ∪ kill[d2]…∪ kill[dn]，d1~dn依次为基本块中的语句
- **gen[B]** = gen[dn] ∪(gen[d(n-1)] – kill[dn]) ∪(gen[d(n-2)] – kill[d(n-1)] – kill[dn])…∪(gen[d1] - kill[d2] – kill[d3]…-kill[dn])

例：

- out[B] = gen[B] ∪ (in[B] – kill[B])
  - in[B]为进入基本块时的数据流信息
  - kill[B] = kill[d1] ∪ kill[d2]…∪ kill[dn]，d1~dn依次为基本块中的语句
  - gen[B] = gen[dn] ∪(gen[d(n-1)] – kill[dn]) ∪(gen[d(n-2)] – kill[d(n-1)] – kill[dn])…∪(gen[d1] - kill[d2] – kill[d3]…- kill[dn])

```
d1: a = b + 1
d2: a = b + 2
```

kill[B] = kill[d1] ∪ kill[d2] = {d2} ∪ {d1} = {d1, d2}

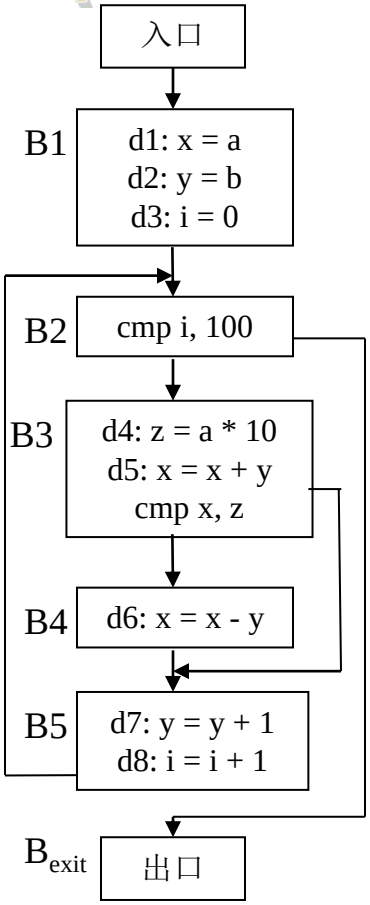
gen[B] = gen[d2] ∪ (gen[d1] – kill[d2]) =
{d2} ∪ ({d1} – {d1}) = {d2}

out[B] = gen[B] ∪ (in[B] – kill[B]) =
{d2} ∪ (in[B] - {d1, d2})

算法14.5 基本块的到达定义数据流分析（迭代思想）

- **输入**：程序流图，且基本块的kill[]和gen[] 已经计算完毕
- **输出**：每个基本块入口和出口处的in和out集合，即in[B]和out[B]

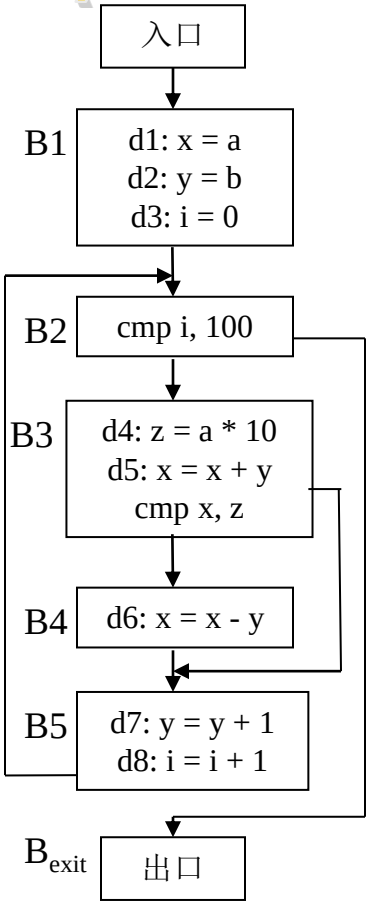
1. 将包括代表流图出口基本块B_{exit}的所有基本块的out集合，初始化为空集。
2. 根据方程**$in[B] = \cup_{B的前驱基本块P} out[P]$**，$out[B] = gen[B] \cup (in[B] - kill[B])$，为每个基本块B依次计算集合in[B]和out[B]。
3. 如果某个基本块计算得到的out[B]与该基本块此前计算得出的out[B]不同，**则循环执行步骤2**，直到所有基本块的out[B]集合不再产生变化为止。




```
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}
```

$in[B] = \cup_{B\text{的前驱基本块}P} out[P]$

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




```
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}
```

$in[B] = \cup_{B的前驱基本块P} out[P]$

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

步骤2：

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

B2的前驱为B1和B5

```
in[B2] = {d1, d2, d3}
out[B2] = {d1, d2, d3}
```

B3的前驱为B2

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

B4的前驱为B3

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

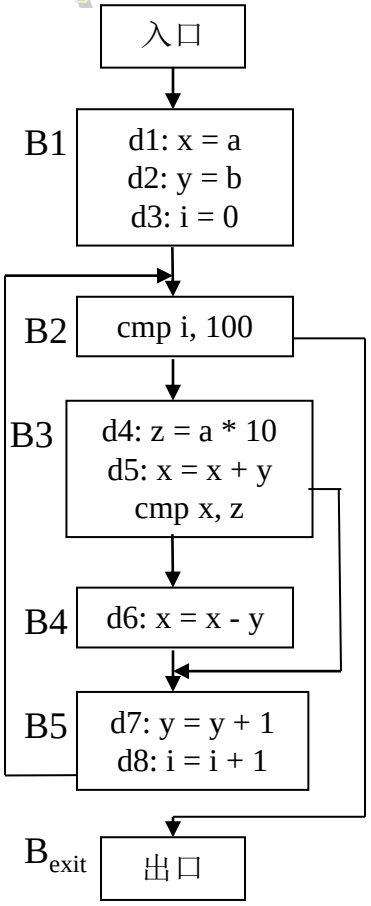
B5的前驱为B3和B4

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

B5的修改会导致B2修改

Bexit的前驱为B2，

$in[B_{exit}] = \{d1, d2, d3\}$




```
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}
```

$in[B] = \cup_{B的前驱基本块P} out[P]$

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

B5的前驱为B3和B4

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

迭代计算：

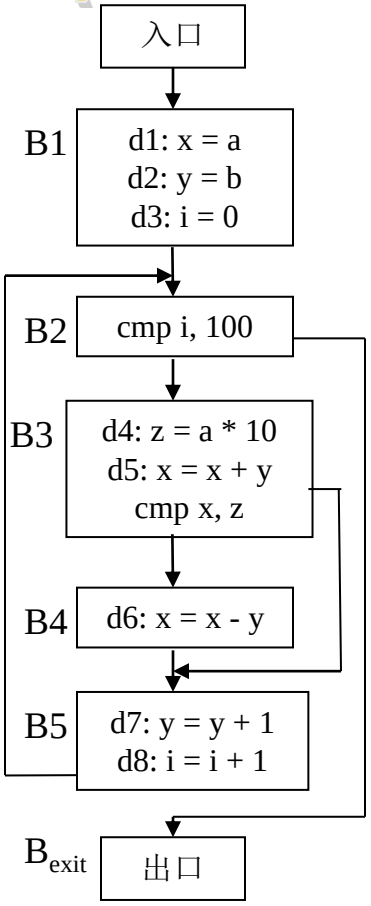
```
in[B1] = {},
out[B1] = {d1, d2, d3}
```

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}
out[B3] = {d2, d3, d4, d5}
```




```
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：

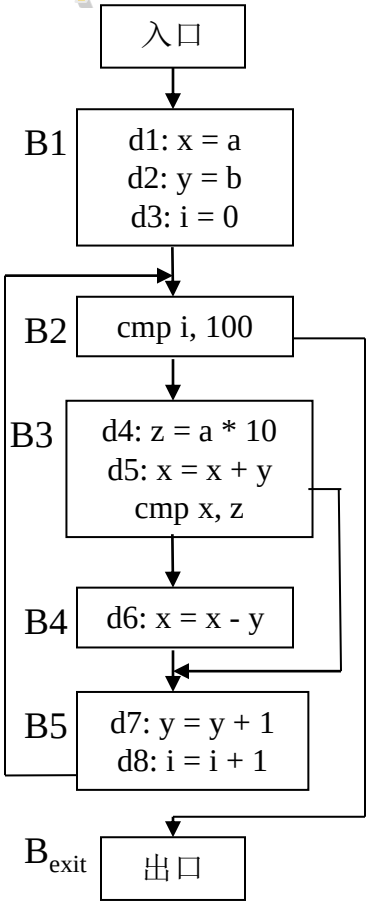
```
in[B1] = {},
out[B1] = {d1, d2, d3}
```

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}
```




```
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

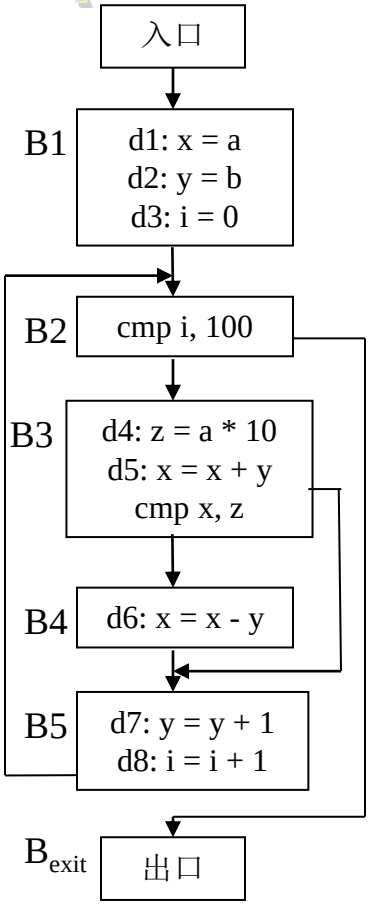
```
in[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}
out[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}
```

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}
```




```
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

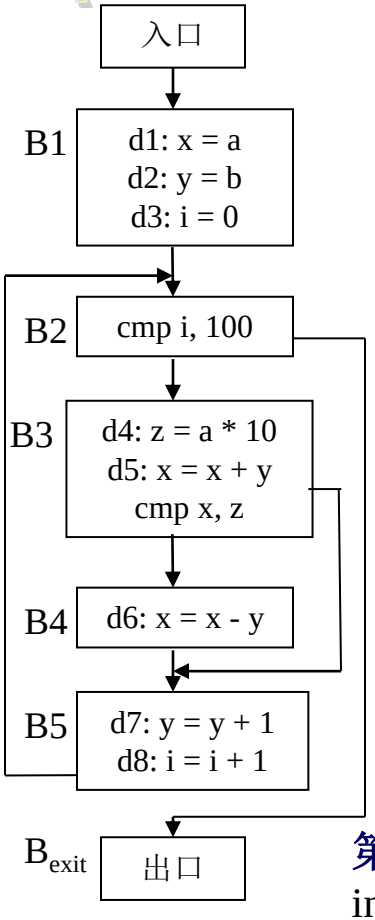
```
in[B4] = {d2, d3, d4, d5, d7, d8}
out[B4] = {d2, d3, d4, d6, d7, d8}
```

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

所有迭代计算过程



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}

$in[B] = \cup_{B的前驱基本块P} out[P]$
$out[B] = gen[B] \cup (in[B] - kill[B])$

步骤2：

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

B2的前驱为B1和B5

in[B2] = {d1, d2, d3}, out[B2] = {d1, d2, d3}

B3的前驱为B2

in[B3] = {d1, d2, d3}, out[B3] = { d2, d3, d4, d5}

B4的前驱为B3

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

B5的前驱为B3和B4

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

Bexit的前驱为B2， $in[B_{exit}]$ = {d1, d2, d3}

第二次步骤2

in[B1] = {}, out[B1] = {d1, d2, d3}
in[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}, out[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}
in[B3] = {d1, d2, d3, d4, d5, d6, d7, d8}, out[B3] = {d2, d3, d4, d5, d7, d8}
in[B4] = {d2, d3, d4, d5, d7, d8}, out[B4] = {d2, d3, d4, d6, d7, d8}
in[B5] = {d2, d3, d4, d5, d6, d7, d8}, out[B5] = {d4, d5, d6, d7, d8}
$in[B_{exit}]$ = {d1, d2, d3, d4, d5, d6, d7, d8}

算法实现

- 集合 “∪”和“-”运算：可以采用位向量（Bit Vector）的方式完成。
- 将集合中的每个定义点，根据其下标映射为一个无限位二进制数的某一位，例如，可以将d1映射为第1位，d3映射为第3位，以此类推。
  - 例如，`out[B3] = { d2, d3, d4, d5, d7, d8}`，其对应的二进制位向量为 `11011110`，该位向量从低位到高位依次对应d1~d8。
  - 基于这样的设定，集合之间的“∪”运算等价于位向量之间的或运算，集合之间的“-”运算等价于将后者取补（取反加一）后，和前者进行按位与运算。
- 在数据流分析方法的实现中，位向量是常用的手段之一。

到达定义（reaching definition）分析

- 特别说明
  - 变量的定义：赋值语句、过程参数、指针引用等多种形式
  - 不能判断时：保守处理

全局优化：活跃变量分析（Liveness）

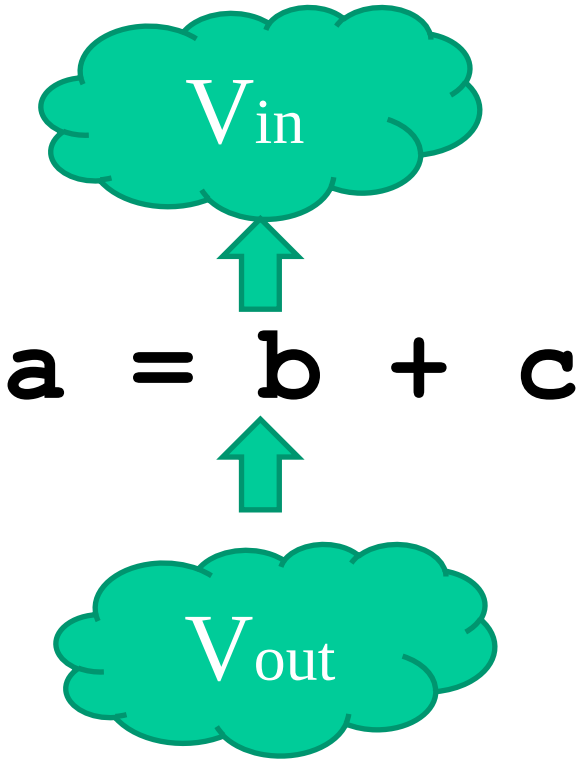
- 消除死代码（Dead Code）
- 变量的活性（Liveness of an Variable）

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

```
a = b + c
```

$$L=\{\ldots\}$$

(L - {a}) U {b,c}



L={...}

活跃变量分析 (Live-variable Analysis)

- 达到定义分析是沿着流图路径的，有的数据流分析是方向计算的
- **活跃变量分析：了解变量x在某个执行点p是活跃的**
  - 变量x的值在p点或沿着从p出发的某条路经中会被使用，则称x在p点是活跃的。

  - 了解到某个变量x在程序的某个点上是否活跃，或者从该点出发的某条路径上是否会被使用
  - 如果存在被使用的可能，x在该程序点上便是活跃的，否则就是非活跃，或者死的。

活跃变量分析 (Live-variable Analysis)

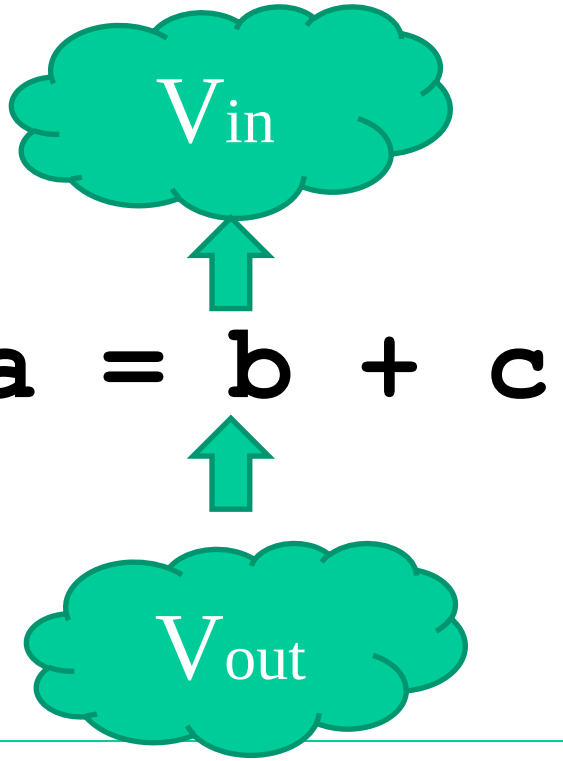
- 活跃变量信息对于寄存器分配，不论是全局寄存器分配还是临时寄存器分配都有重要意义。
  - 如果拥有寄存器的变量x在p点开始的任何路径上不再活跃，可以释放寄存器
  - 如果两个变量的活跃范围不重合，则可以共享同一个寄存器

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

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

引用变量会产生新的数据流　　赋值会删除数据流

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



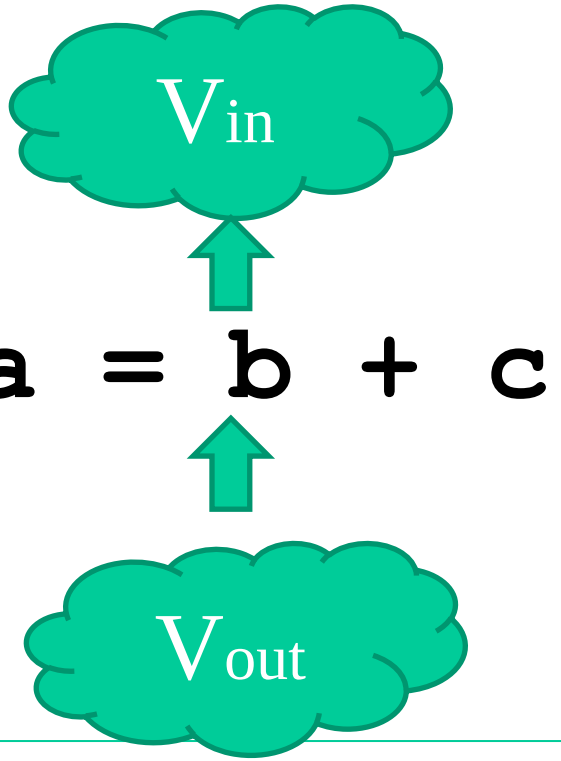
$L=\{\ldots\}$

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

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

引用变量会产生新的数据流　　赋值会删除数据流

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



$L=\{\ldots\}$

活跃变量分析 (Live-variable Analysis)

- 数据流方程如下：

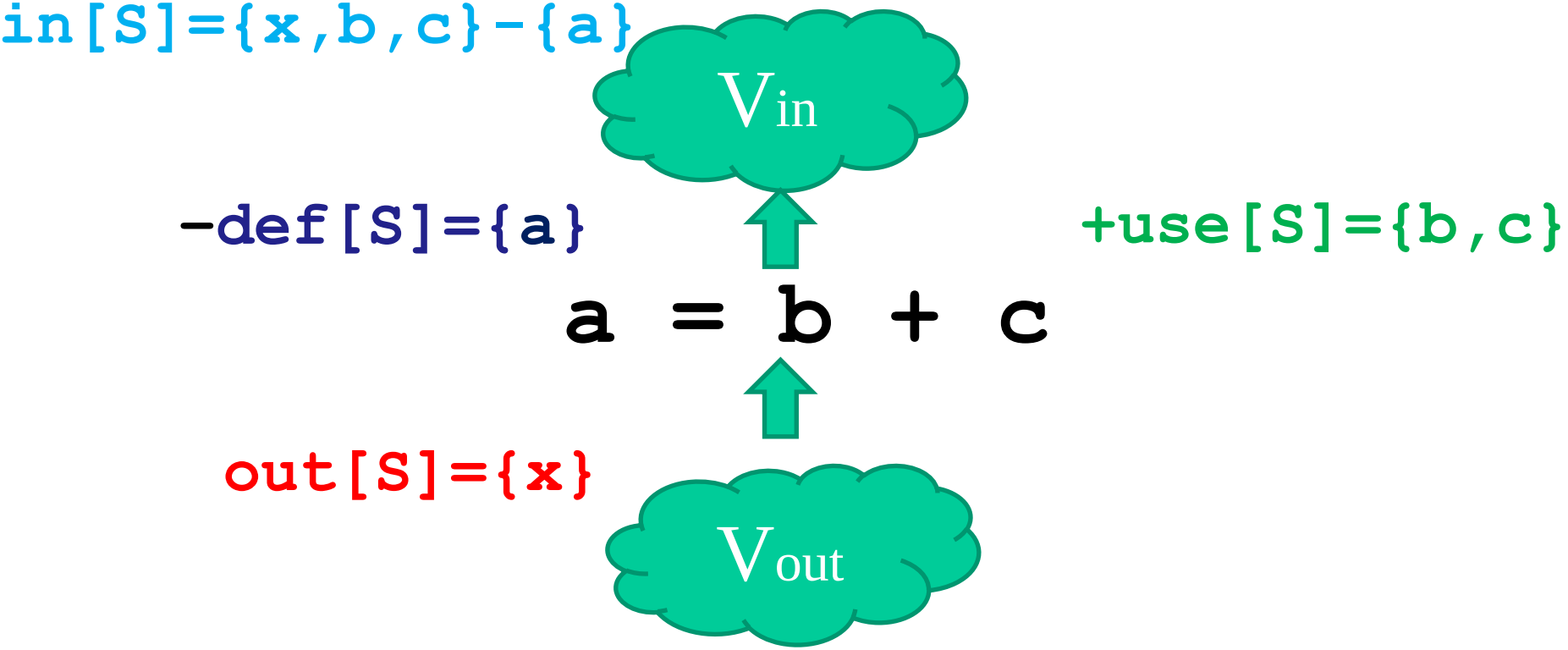
  $in[S]=use[S]\cup(out[S]-def[S])$

  - $out[B] = \cup_{B的后继基本块P} in[P]$
  - $def[B]$：变量在B中**被定义（赋值）先于**任何对它们的使用
  - $use[B]$：变量在B中**被使用先于**任何对它们的定义

- **可达定义分析**的数据流：沿流图中的控制流方向计算
- **活跃变量分析的**数据流：沿流图中控制流的**反方向**计算

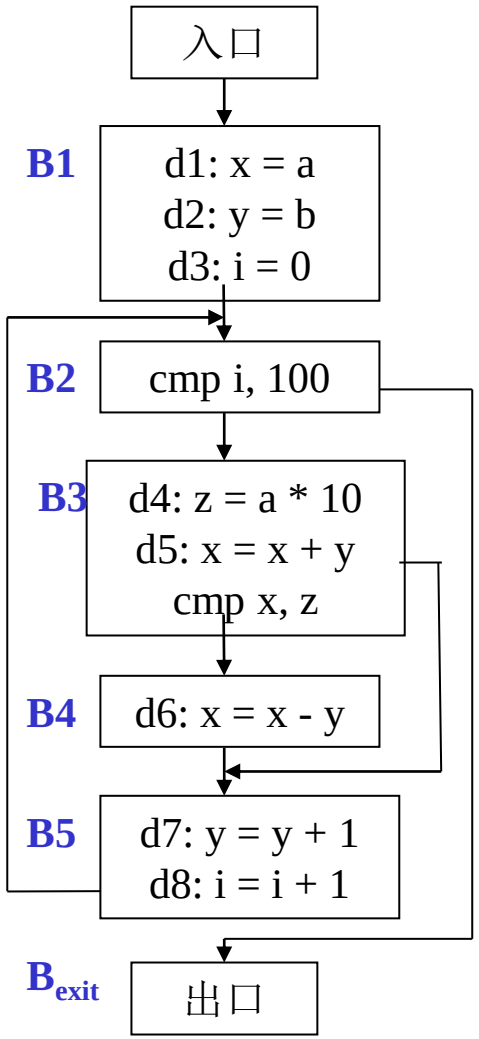
in[S]=gen[S]∪(out[S]-kill[S])

in[S]=use[S]∪(out[S]-def[S])



in[S]={b,c} ∪({x}-{a})

活跃变量分析



- **def[B]** ：变量在B中**被定义（赋值）先于**任何对它们的使用
- **use[B]** ：变量在B中**被使用先于**任何对它们的定义

```
use[B3] = {a, x, y}
def[B3] = {z}
```

d1~d8 八个定义点

基本块的活跃变量数据流分析 (基本思路：迭代)

输入：程序流图，且基本块的use集和def集已计算完毕

输出：每个基本块入口和出口处的in[B]和out[B]

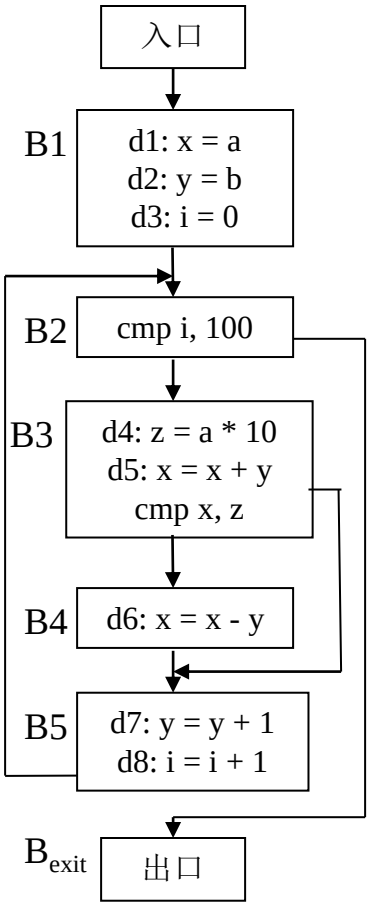
方法：

1. 将包括代表流图出口基本块B_{exit}在内的所有基本块的in集合，初始化为空集。
2. 根据方程 $out[B] = \cup_{B的后继基本块P} in[P]$

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

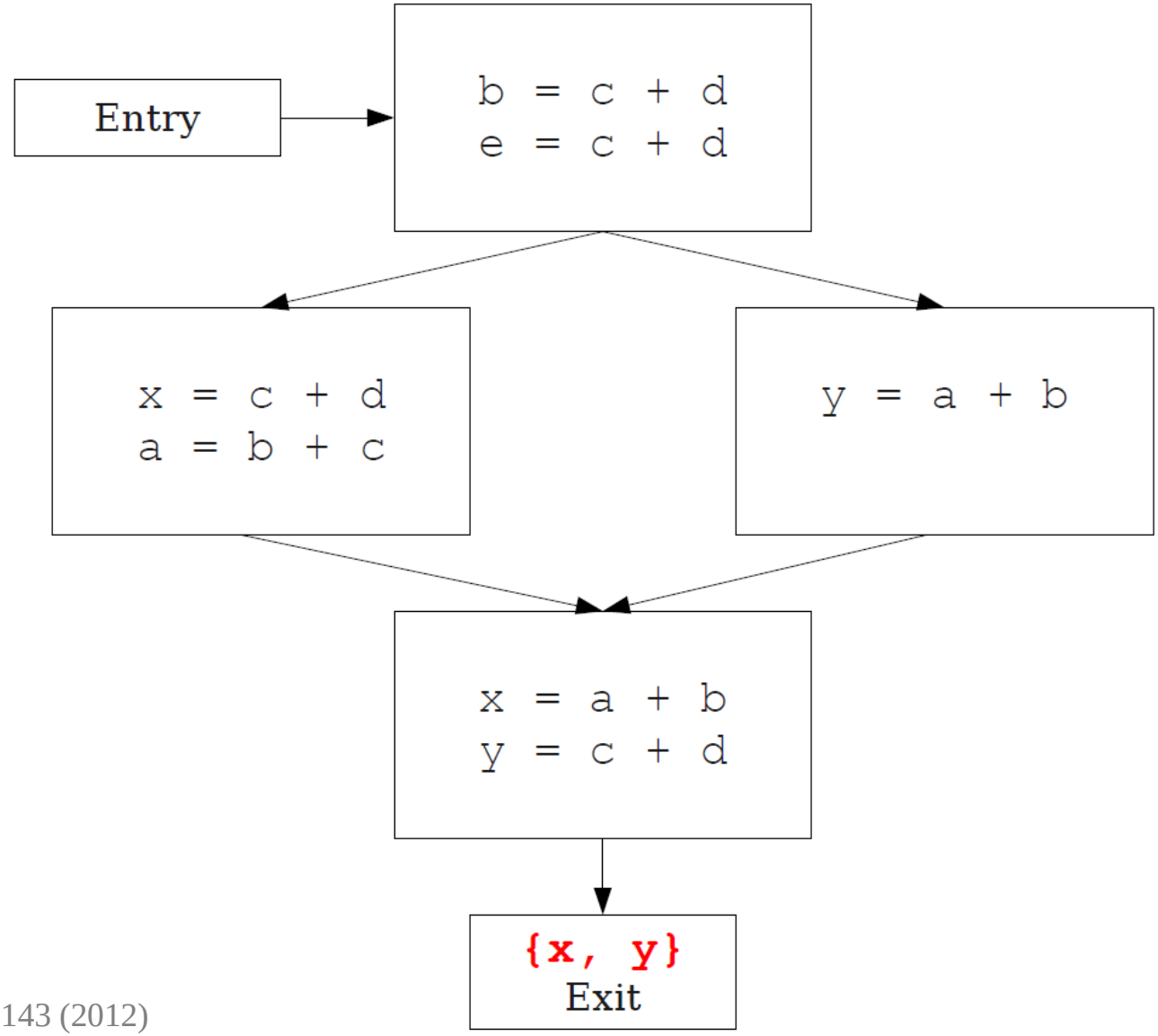
   为每个基本块B依次计算集合 out[B] 和 in[B]
3. 如果计算得到in[B]与此前计算得出的in[B]不同，则循环执行步骤2，直到所有基本块的in[B]集合不再产生变化为止

```
for each B do in[B] = {};

while 集合in发生变化 do
    for each B do begin
        out[B] = ∪B的后继基本块P in[P]
        in[B]=use[B]∪(out[B]-def[B])
    end
```

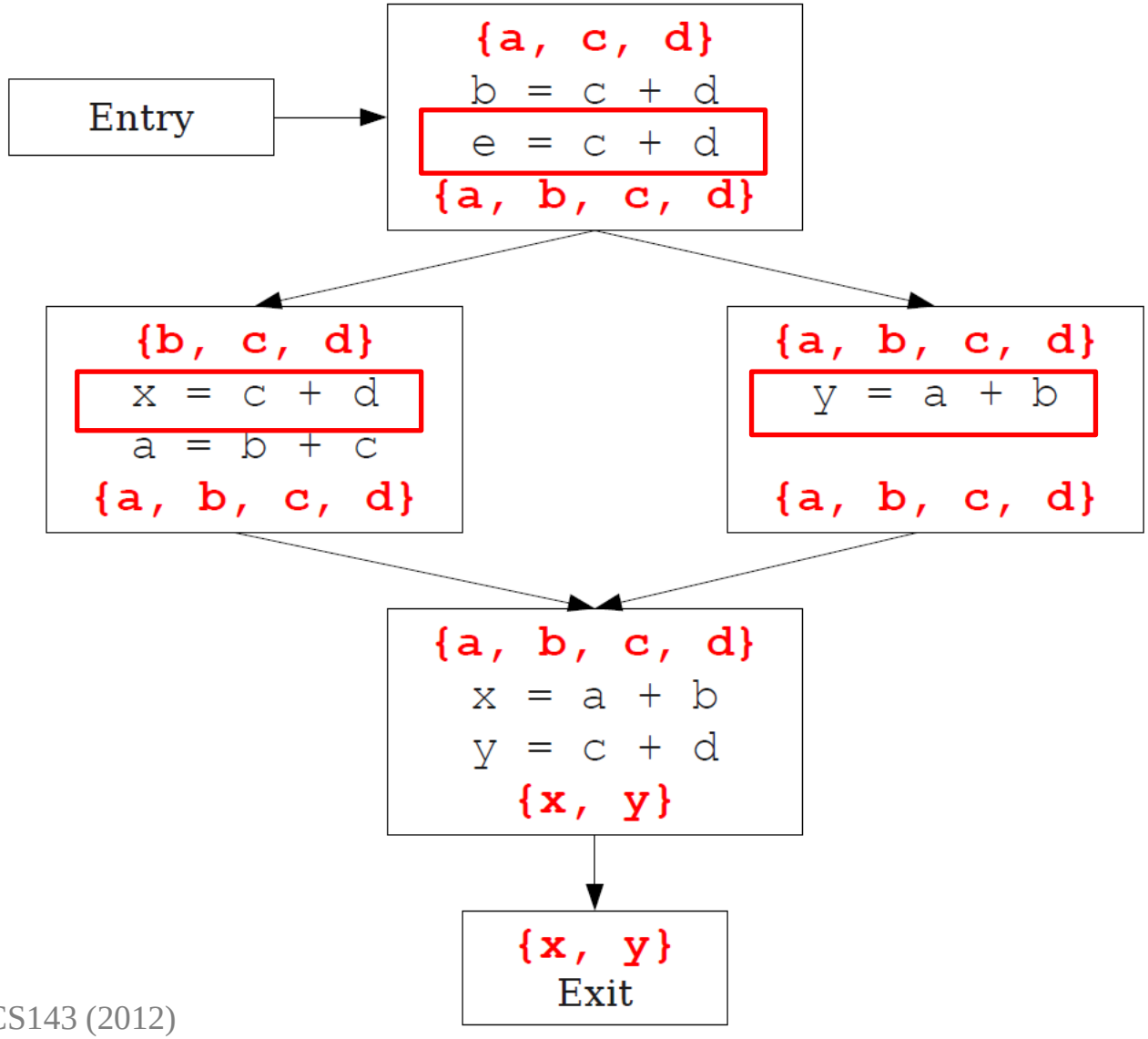


流图	def[B]	use[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
B1	x, y, i	a, b	a, b	a,x,y,i	a, b	a,x,y,i	a, b	a,x,y,i
B2	∅	i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B3	z	a, x, y	a,x,y,i	x, y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B4	∅	x, y	x, y, i	y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B5	∅	y, i	y, i	∅	a,x,y,i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

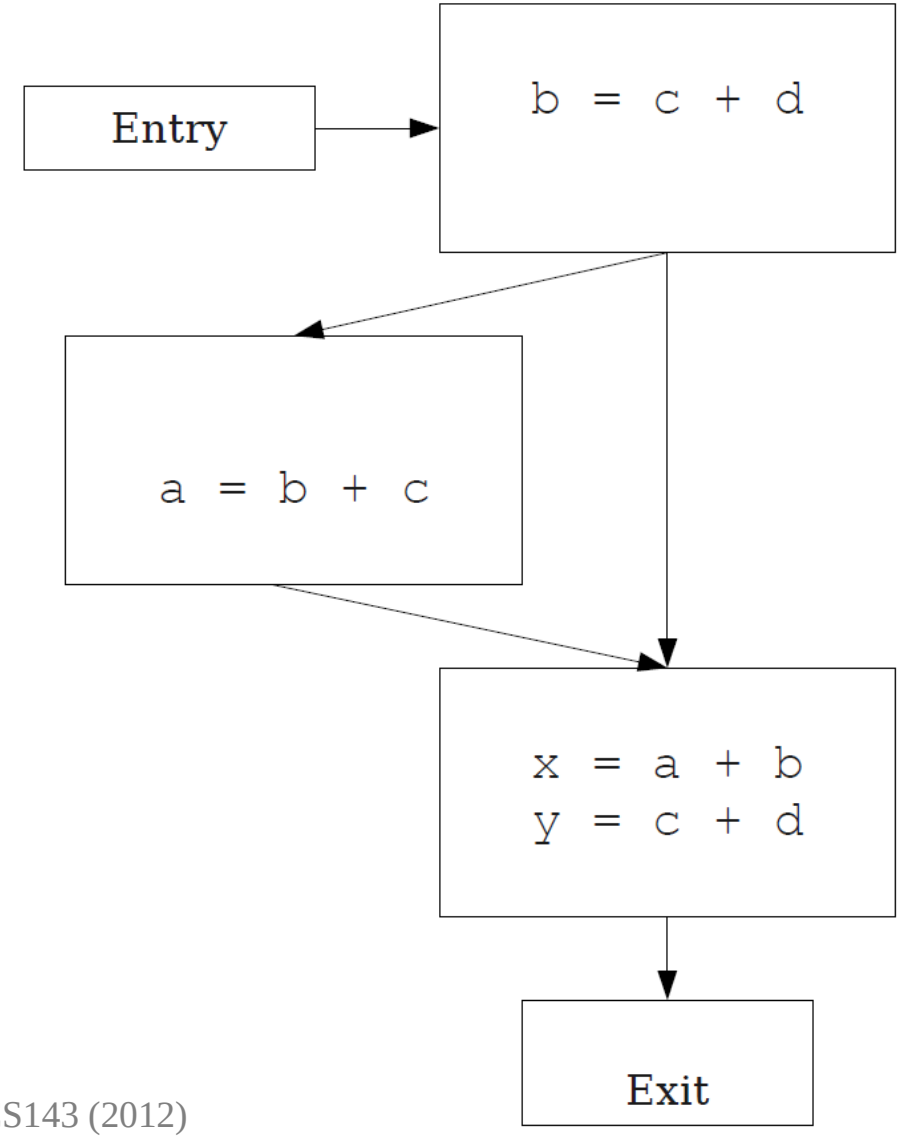
全局死代码消除



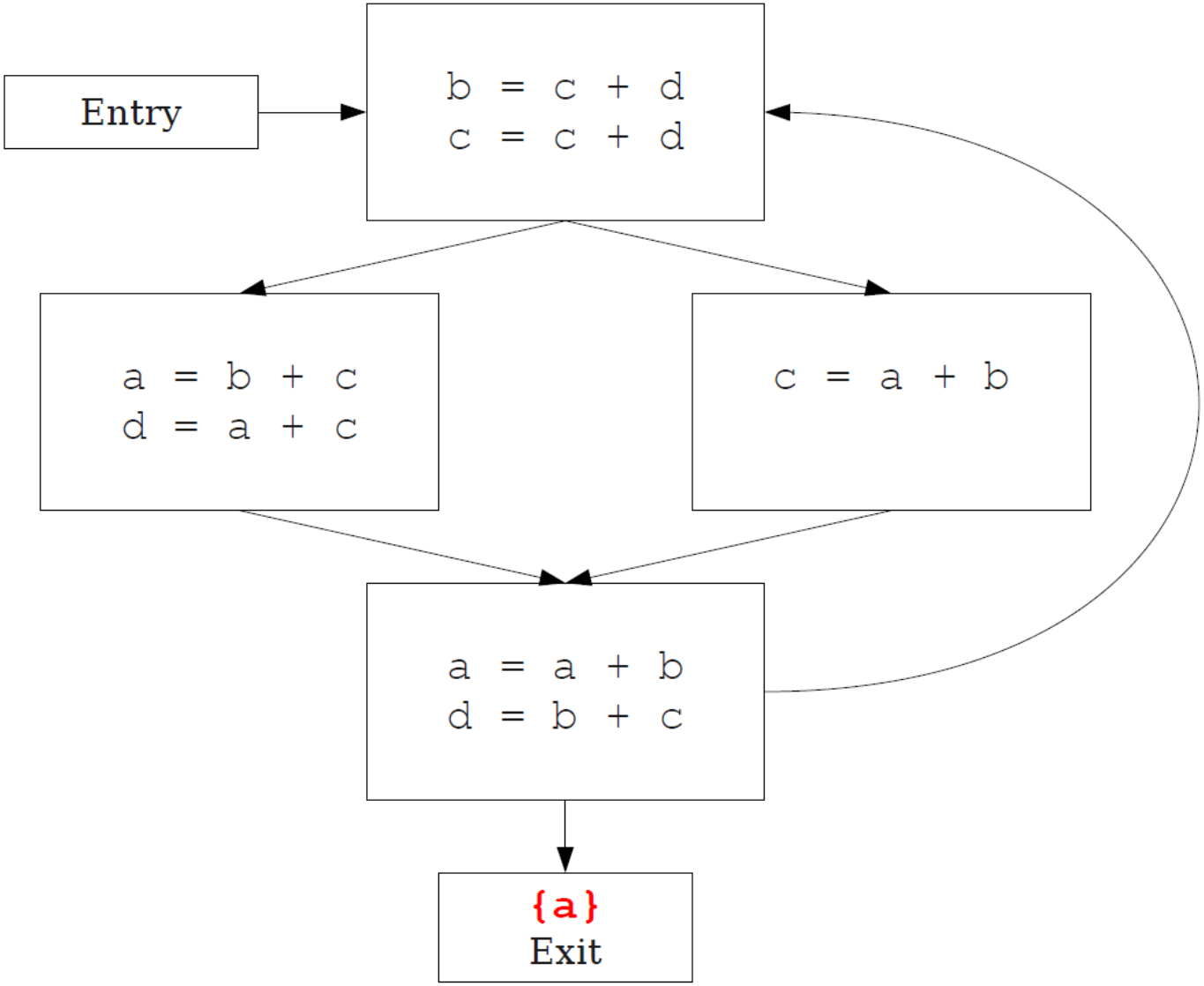
Source: Stanford CS143 (2012)

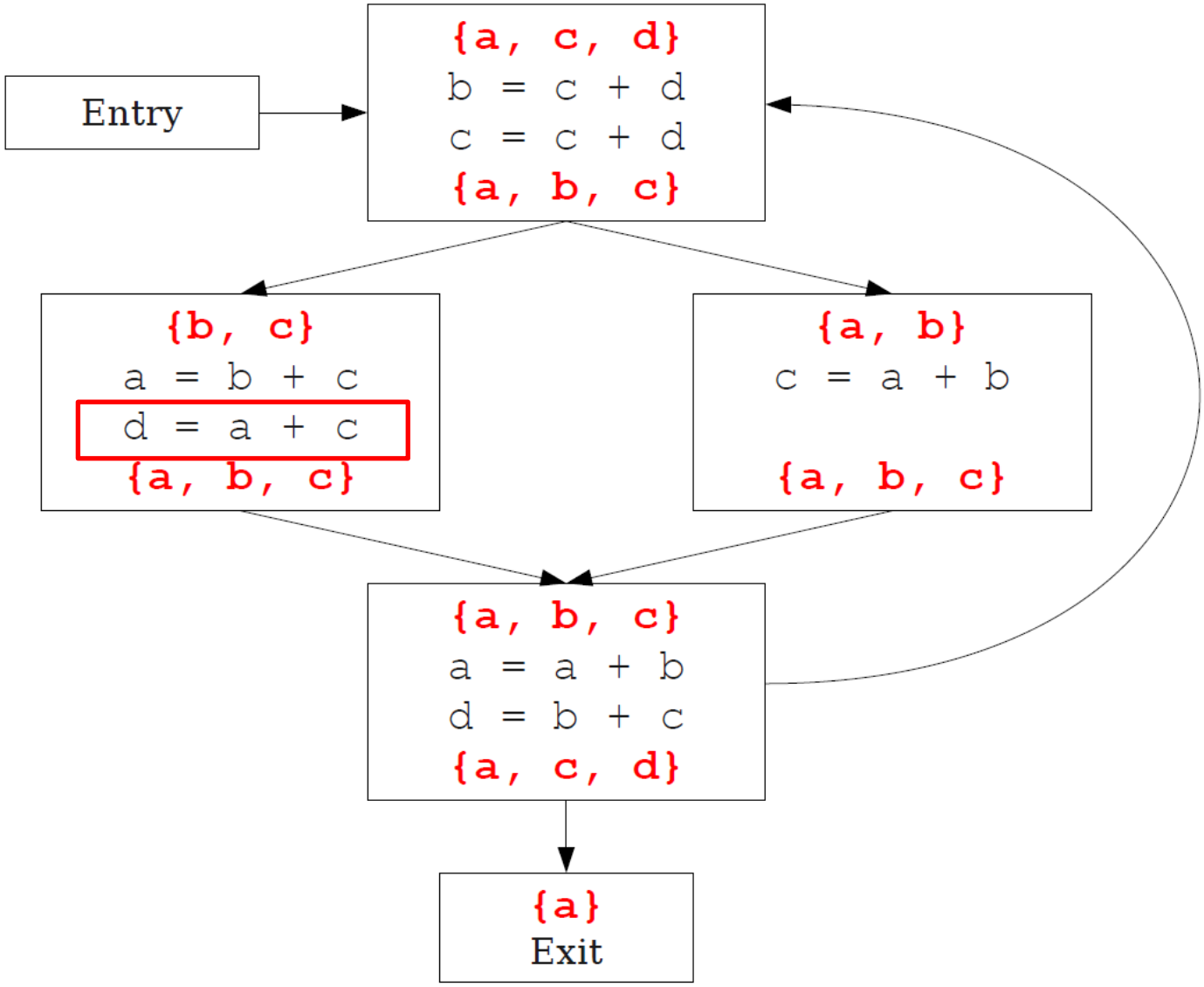


Source: Stanford CS143 (2012)

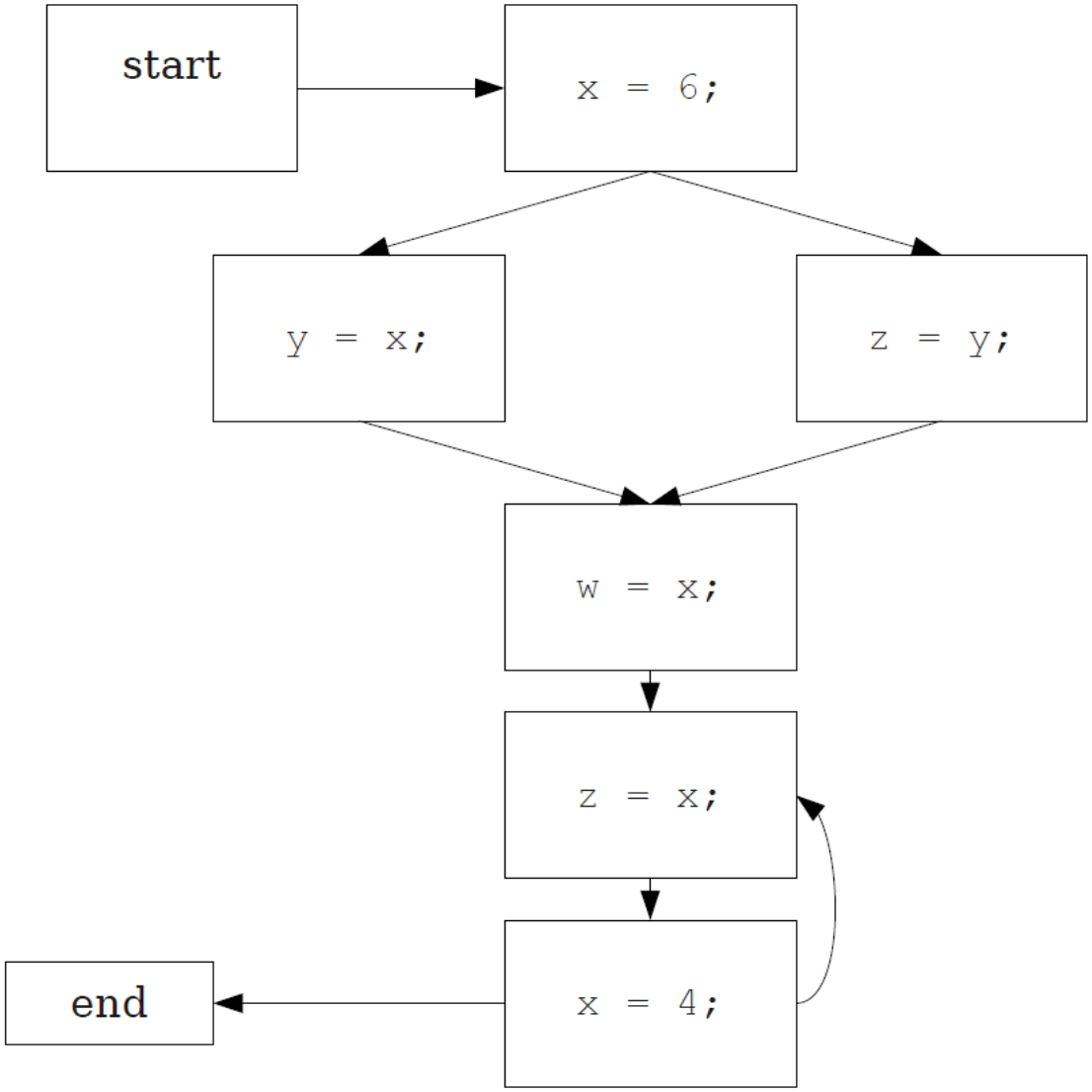


Source: Stanford CS143 (2012)


Entry
b = c + d
c = c + d
a = b + c
d = a + c
c = a + b
a = a + b
d = b + c
{a}
Exit


Entry
{a, c, d}
b = c + d
c = c + d
{a, b, c}
{b, c}
a = b + c
d = a + c
{a, b, c}
{a, b}
c = a + b
{a, b, c}
{a, b, c}
a = a + b
d = b + c
{a, c, d}
{a}
Exit

全局复制传播（常量传播）


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

作业： 14章 3-5题

谢谢！