

Lecture 5.1

数据流分析和代码优化

徐 辉

xuh@fudan.edu.cn



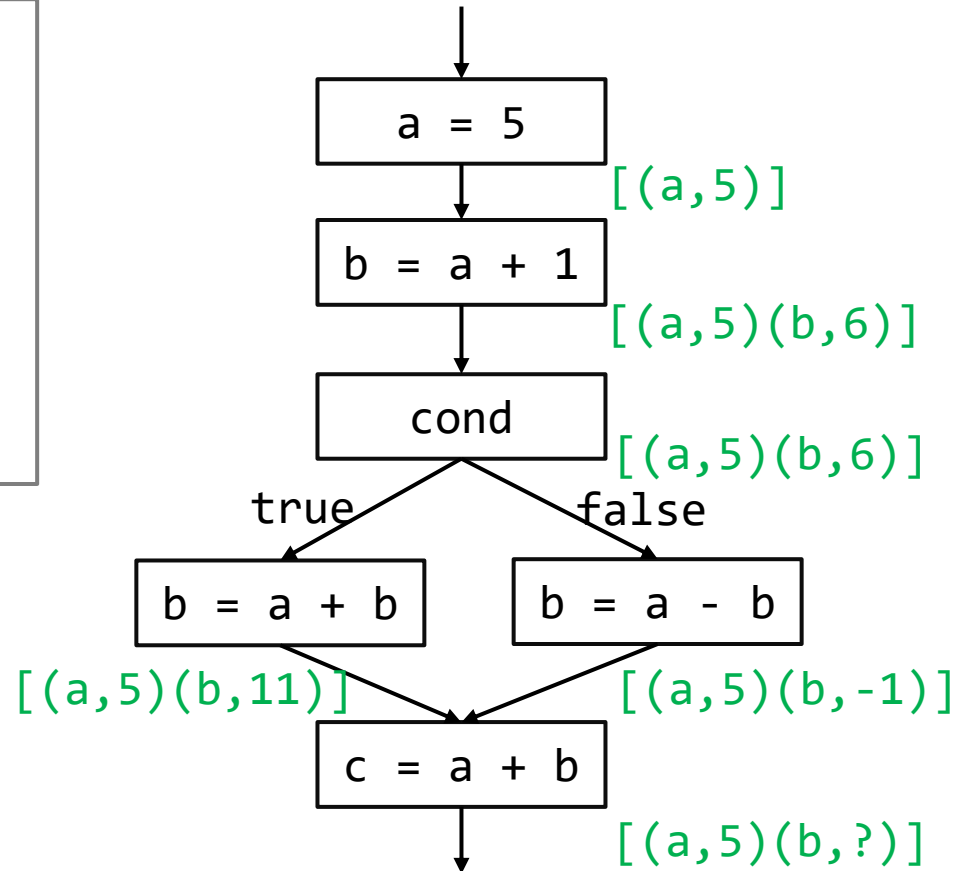
大纲

- 一、常量分析和优化
- 二、可达性分析和数值流图
- 三、静态单赋值形式

常量分析问题

- 分析每一个program point有哪些变量为定值？

```
let a:const int = 5;  
let b = a + 1;  
if(cond) {  
    b = a + b;  
} else {  
    b = a - b;  
}  
let c = a + b;
```

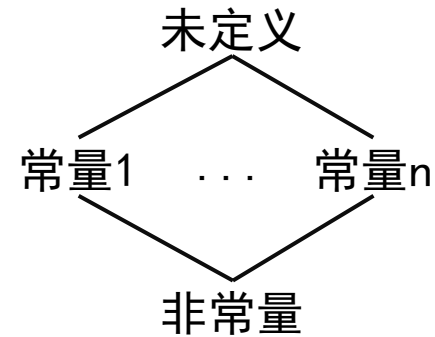


定义Transfer函数

- 对于赋值语句 $x = rvalue$
 - 如果 $rvalue$ 的形式为常量 c ，则 x 的状态为 “常量 c ”
 - 如果 $rvalue$ 的形式为变量 y ，则 x 的状态等同于 y 的状态
 - 如果 $rvalue$ 的形式为 “ $y \text{ op } z$ ”，则 x 的状态为：
 - $y \text{ op } z$ 运算后的值，如果 y 和 z 是常量
 - 非常量，如果 y 或 z 是非常量
 - 未定义，如果 y 或 z 未定义
 - 如果 $rvalue$ 的形式为函数调用，则 x 的状态为非常量
 - x 以外的其它的变量状态不变

基于Lattice的方法

- 使用poset (V, \wedge) 对变量的状态建模
 - partially ordered set
 - 未定义 \geq 常量x
 - 常量x \geq 非常量
 - top element: 未定义
 - bottom element: 非常量
 - Meet运算
 - 未定义 \wedge 常量x = 常量x
 - 常量x \wedge 常量y = 非常量
 - 常量x \wedge 非常量 = 非常量



常量分析和优化

- 常量传播 (constant propagation) : 识别常量并将相应的变量替换为常量
- 常量折叠 (constant folding) : 编译时完成对常量表达式的计算

```
let a:const int = 5;  
let b = a + 1;  
if(cond) {  
    b = a + b;  
} else {  
    b = a - b;  
}  
let c = a + b;
```

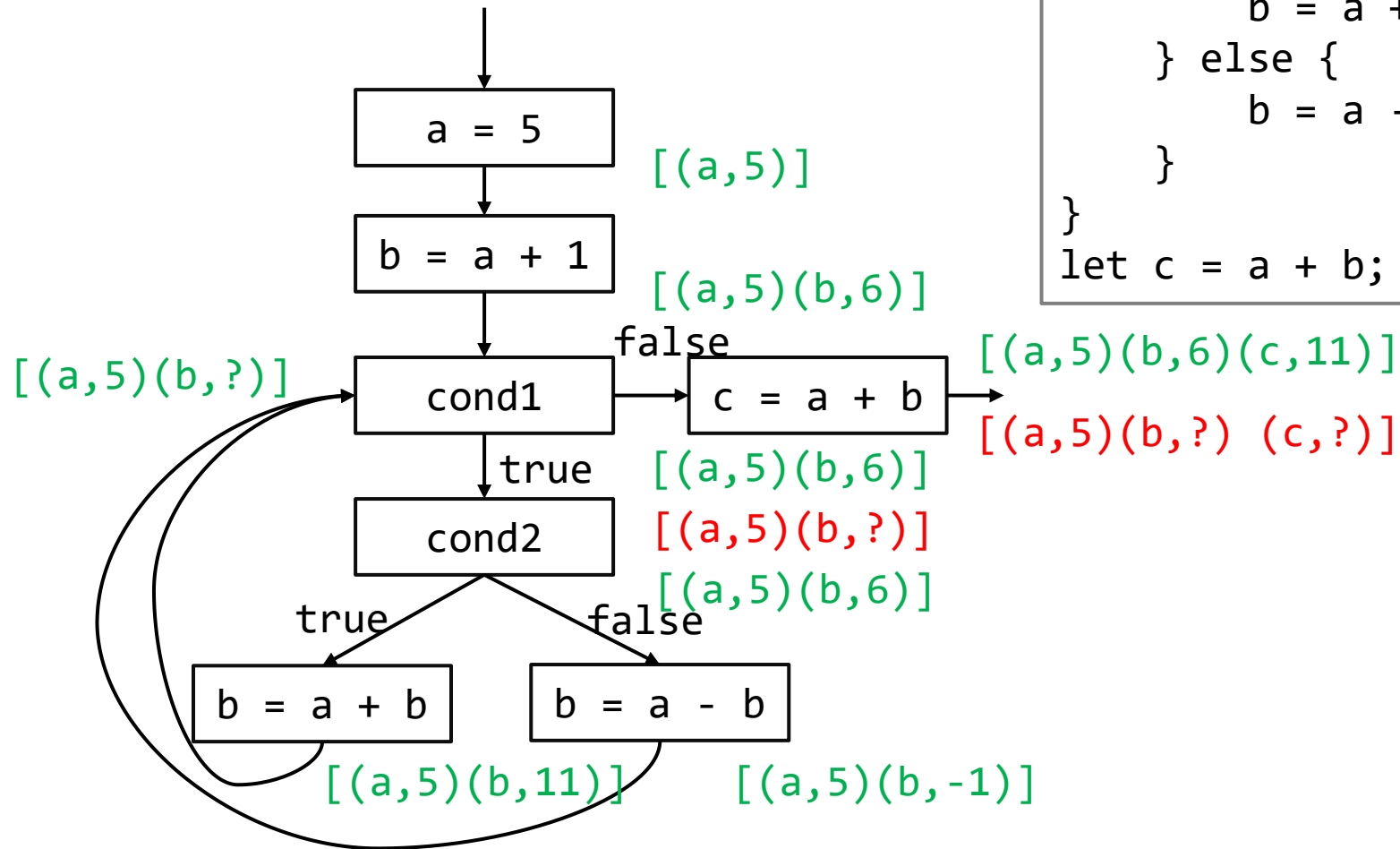


```
let a:const int = 5;  
let b = 6;  
if(cond) {  
    b = 11;  
} else {  
    b = -1;  
}  
let c = 5 + b;
```

有循环的情况

- Chaotic iteration

```
let a:const int = 5;  
let b = a + 1;  
while(cond1){  
    if(cond2) {  
        b = a + b;  
    } else {  
        b = a - b;  
    }  
}  
let c = a + b;
```



算法设计思路

```
For (each node n): 1. 初始化每个变量的状态为未定义
    IN[n] = {<v, undef>: v is a program variable}
    OUT[n] =  $\emptyset$ 
Repeat:
    For(each node n): 2. 遍历控制流图
        For(each n's predecessor p) 3. 如入度>1, 则meet
            IN[n] = IN[n]  $\cap$  OUT[p]
            OUT[n] = TRANSFER(n) 4. 分析当前节点的语义
Until IN[n] and OUT[n] stops changing for all n
```

5. 重复该过程直至结束条件: fixed point

Worklist算法

```
For (each node n):  
    IN[n] = {<v, undef>: v is a program variable}  
    OUT[n] =  $\emptyset$   
Worklist = entry  
While Worklist !=  $\emptyset$   
    n = Worklist.pop()  
    old = OUT[n]  
    IN[n] = IN[n]  $\cap_{p \in \text{predecessor}(n)}$  OUT[p]  
    OUT[n] = TRANSFER(n)  
    if (OUT[n] != old)  
        For (each n's successor s)  
            Worklist.push(s)
```

1.通过worklist记录待分析节点

2.每次pop一个节点

3.如果OUT[n]改变,则将其next节点加入worklist

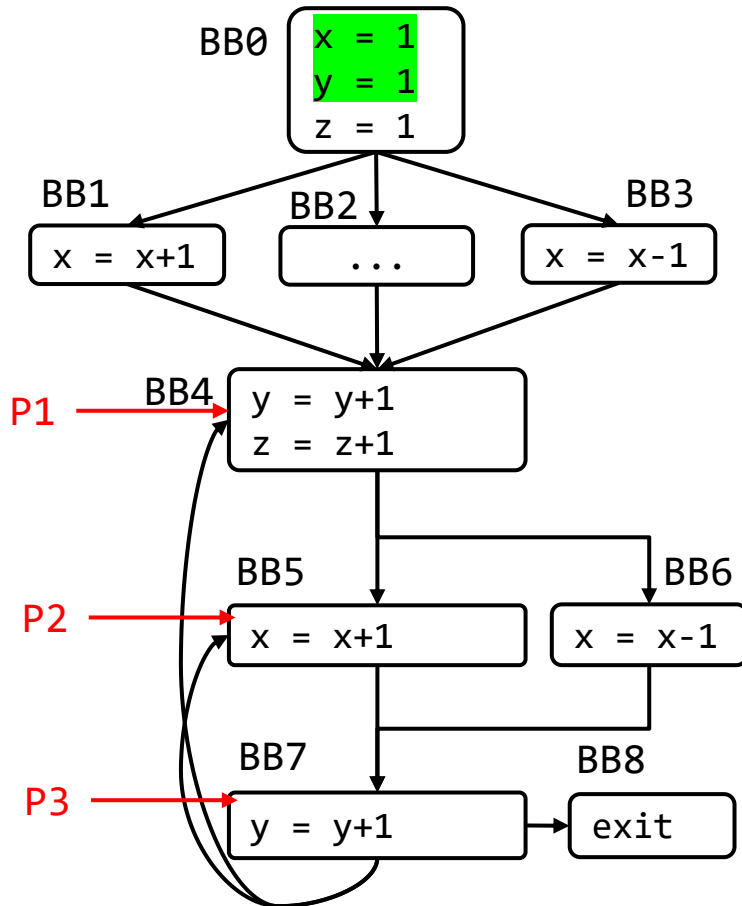
常量分析小节

- 本质：编译时计算
- 主要应用：
 - 常量折叠
 - 谓词常量= \Rightarrow 无用代码删除
 - ...

大纲

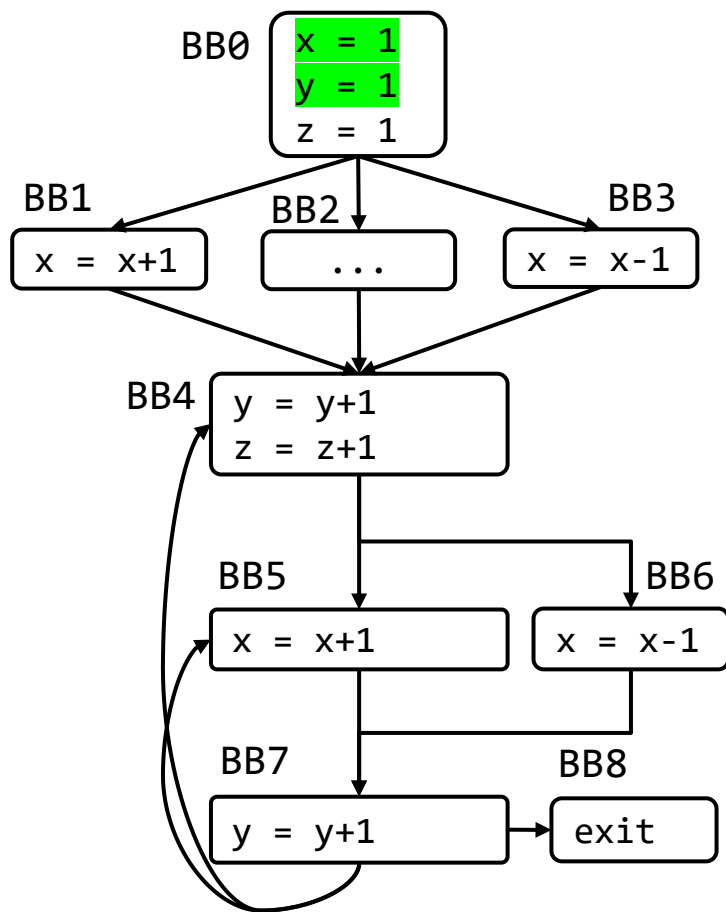
- 一、常量分析和优化
- 二、可达性分析和数值流图
- 三、静态单赋值形式

可达性分析：分析def-use关系



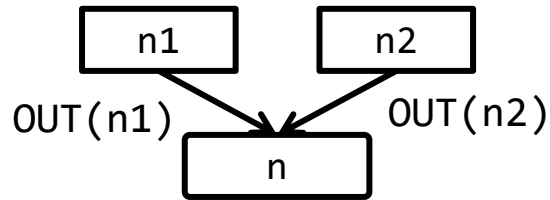
- def-use: 广义的常量分析问题
- 为每一个程序点 (program point) 分析某一变量的某个定义/赋值语句是否可达
 - `x = 1`可以到达: P1, P2
 - `y = 1`可以到达: P1

分析方法



- 为每条语句分配一个编号
 - BB0-1: $x = 1$
 - BB0-2: $y = 1$
 - ...
- 结果保存:
 - $IN(n)$: 节点的入向属性集合
 - $OUT(n)$: 节点的出向属性集合
- 正向遍历控制流图
 - 如遇到指令: $x = \text{load } a$
 - $Gen(n) = \{ \langle x, n \rangle \}$
 - $KILL(n) = \{ \langle x, m \rangle : m \neq n \}$
 - ...
 - $OUT(n) = (IN(n) - KILL(n)) \cup Gen(n)$

控制流和环的处理



$$IN(n) = \bigcup_{n' \in \text{predecessor}(n)} OUT(n')$$

For (each node n):

$IN[n] = OUT[n] = \emptyset$

$OUT[\text{entry}] = \{ \langle v, ? \rangle : v \text{ is a program variable} \}$

Repeat:

For(each node n):

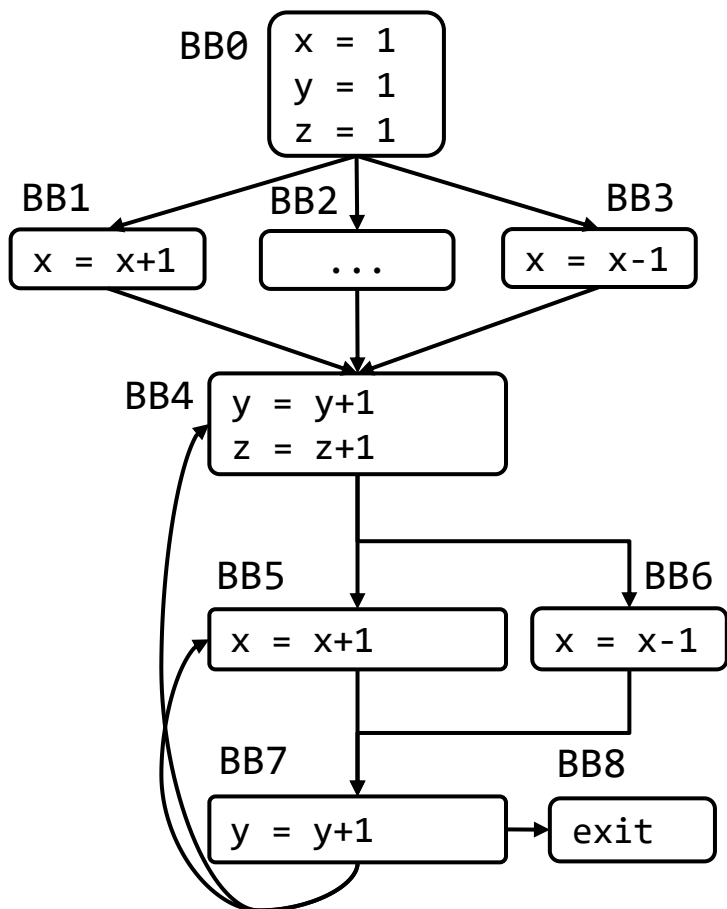
For(each n's predecessor p)

$IN[n] = IN[n] \cup OUT[p]$

$OUT[n] = (IN[n] - KILL(n)) \cup Gen(n)$

Until $IN[n]$ and $OUT[n]$ stops changing for all n

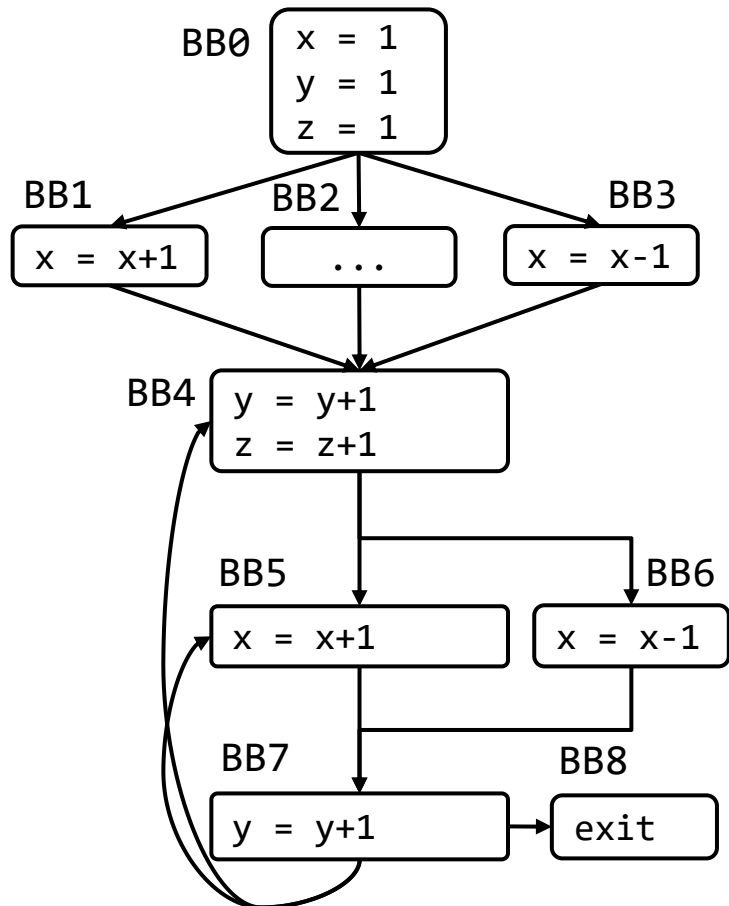
应用：第1轮



$\langle x, n \rangle$: 表示变量x在第n个节点被赋值

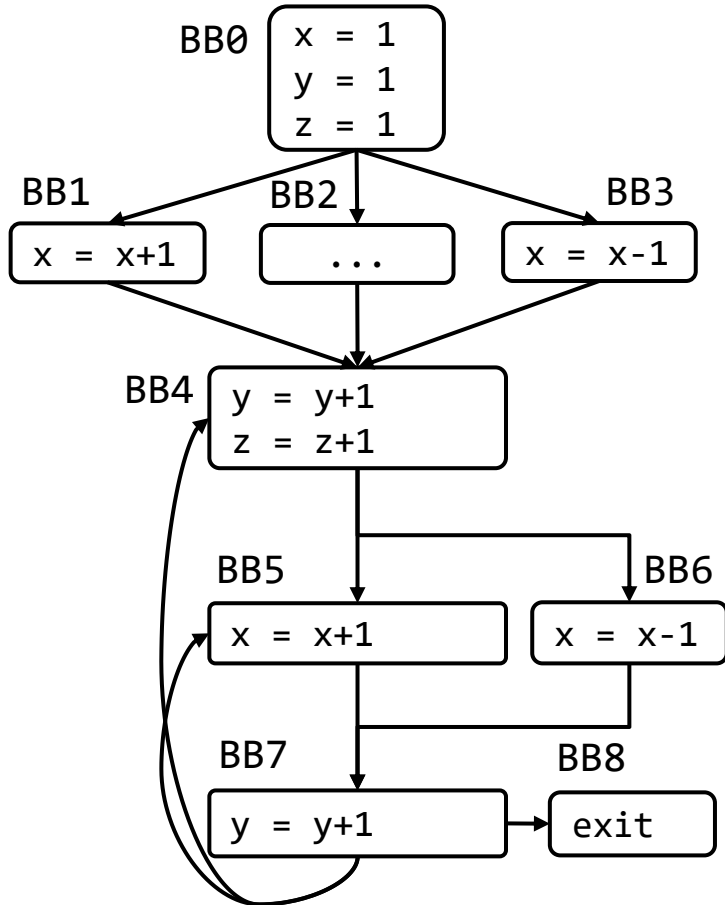
n	IN(n)	OUT(n)
BB0-1	-	$\{\langle x, 0-1 \rangle\}$
BB0-2	$\{\langle x, 0-1 \rangle\}$	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle\}$
BB0-3	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle\}$	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$
BB1-1	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$	$\{\langle x, 1-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$
BB4-1	$\{\langle x, 1-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$	$\{\langle x, 1-1 \rangle \langle y, 4-1 \rangle \langle z, 0-3 \rangle\}$
BB4-2	$\{\langle x, 1-1 \rangle \langle y, 4-1 \rangle \langle z, 0-3 \rangle\}$	$\{\langle x, 1-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$
BB5-1	$\{\langle x, 1-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$	$\{\langle x, 5-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$
BB7-1	$\{\langle x, 5-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$	$\{\langle x, 5-1 \rangle \langle y, 7-1 \rangle \langle z, 4-2 \rangle\}$
BB8-1	$\{\langle x, 5-1 \rangle \langle y, 7-1 \rangle \langle z, 4-2 \rangle\}$	-
BB4-1	$\{\langle x, 1-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$ $\{\langle x, 5-1 \rangle \langle y, 7-1 \rangle \langle z, 4-2 \rangle\}$	
BB5-1	$\{\langle x, 1-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$ $\{\langle x, 5-1 \rangle \langle y, 7-1 \rangle \langle z, 4-2 \rangle\}$	
BB6-1	$\{\langle x, 1-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$	$\{\langle x, 6-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$
BB7-1	$\{\langle x, 5-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$ $\{\langle x, 6-1 \rangle \langle y, 4-1 \rangle \langle z, 4-2 \rangle\}$	
BB2-1	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$
BB4-1	$\{\langle x, 1-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$ $\{\langle x, 5-1 \rangle \langle y, 7-1 \rangle \langle z, 4-2 \rangle\}$ $\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$	
BB3-1	$\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$	$\{\langle x, 3-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$
BB4-1	$\{\langle x, 1-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$ $\{\langle x, 5-1 \rangle \langle y, 7-1 \rangle \langle z, 4-2 \rangle\}$ $\{\langle x, 0-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$ $\{\langle x, 3-1 \rangle \langle y, 0-1 \rangle \langle z, 0-3 \rangle\}$	

第2轮



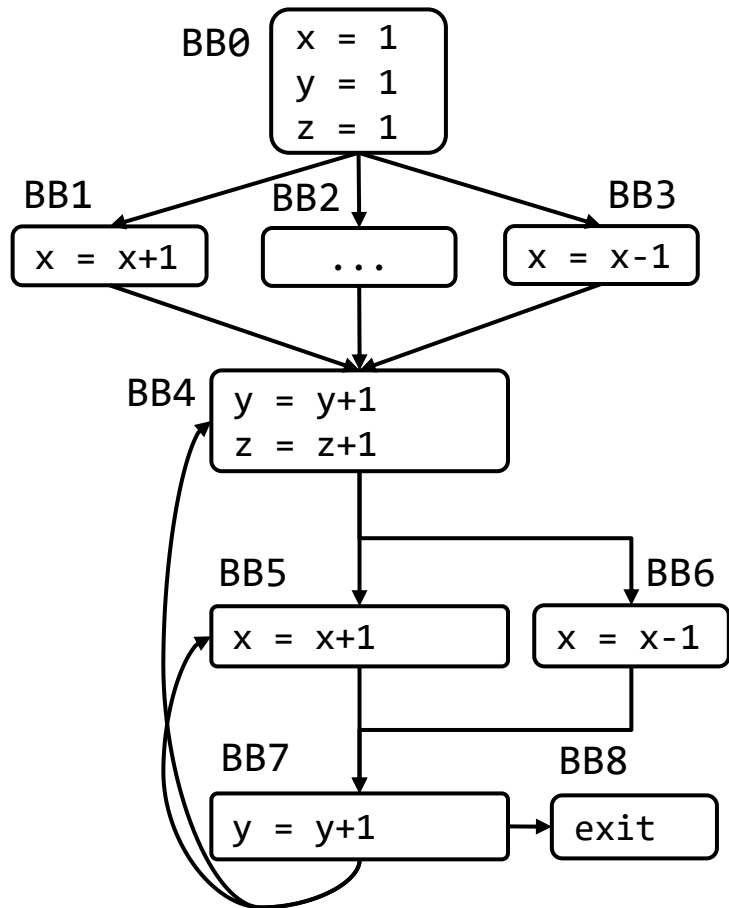
n	IN(n)	OUT(n)
BB4 - 1	{<x,0-1,1-1,3-1,5-1> <y,0-1,7-1> <z,0-3,4-2>}	
BB4 - 2		
BB5 - 1	{<x,1-1,5-1><y,4-1,7-1><z,4-2>}	
BB7 - 1	{<x,5-1,6-1><y,4-1><z,4-2>}	
BB8 - 1		
BB4 - 1		
BB5 - 1		
BB6 - 1		
BB7 - 1		

第2轮



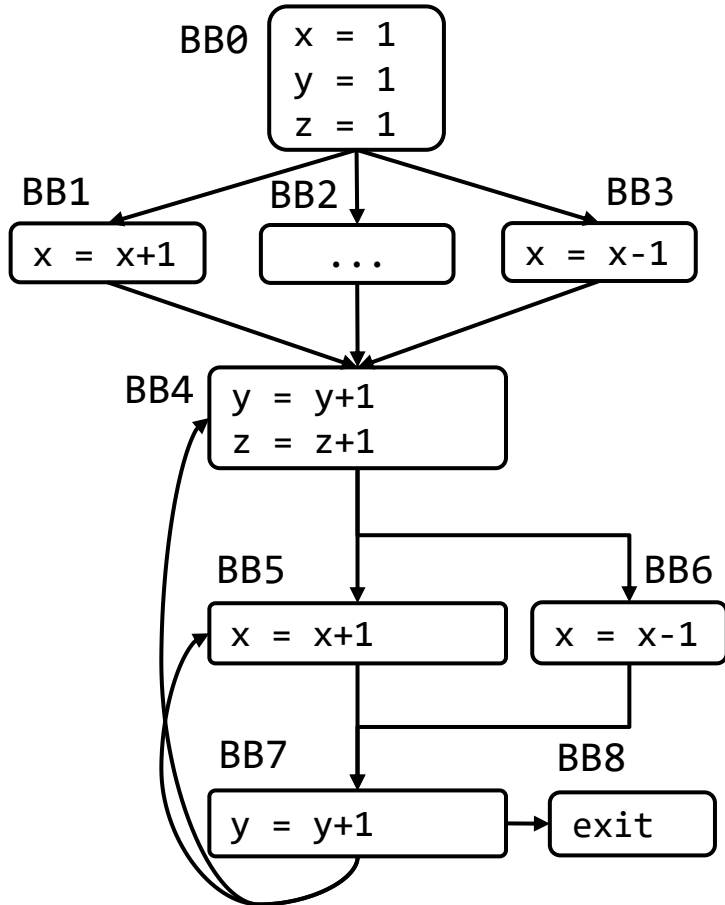
n	IN(n)	OUT(n)
BB4-1	{<x,0-1,1-1,3-1,5-1> <y,0-1,7-1> <z,0-3,4-2>}	<x,0-1,1-1,3-1,5-1> <y,4-1> <z,0-3,4-2>}
BB4-2	{<x,0-1,1-1,3-1,5-1> <y,4-1> <z,0-3,4-2>}	{<x,0-1,1-1,3-1,5-1> <y,4-1> <z,4-2>}
BB5-1	{<x,0-1,1-1,3-1,5-1> <y,4-1> <z,4-2>} {<x,5-1><y,7-1><z,4-2>}	{<x,5-1> <y,4-1,7-1> <z,4-2>}
BB7-1	{<x,5-1> <y,4-1,7-1> <z,4-2>} {<x,6-1><y,4-1><z,4-2>}	{<x,5-1,6-1> <y, 7-1> <z,4-2>}
BB8-1	{<x,5-1,6-1> <y, 7-1> <z,4-2>}	-
BB4-1	{<x,0-1,1-1,3-1,5-1> <y,0-1,7-1> <z,0-3,4-2>} {<x,5-1,6-1> <y, 7-1> <z,4-2>}	
BB5-1	{<x,0-1,1-1,3-1,5-1> <y,4-1, 7-1> <z,4-2>} {<x,5-1,6-1> <y, 7-1> <z,4-2>}	
BB6-1	{<x,0-1,1-1,3-1,5-1> <y,4-1> <z,4-2>}	{<x,6-1> <y,4-1> <z,4-2>}
BB7-1		

第3轮



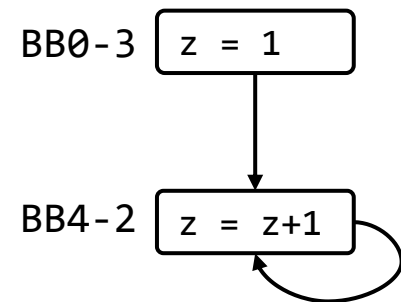
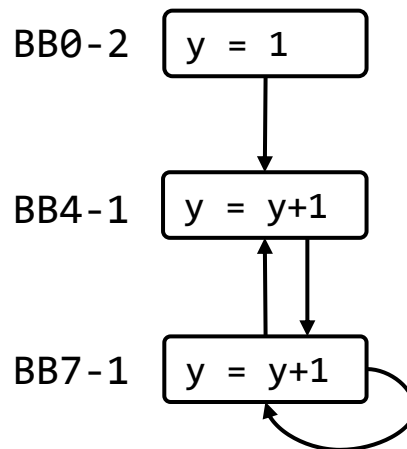
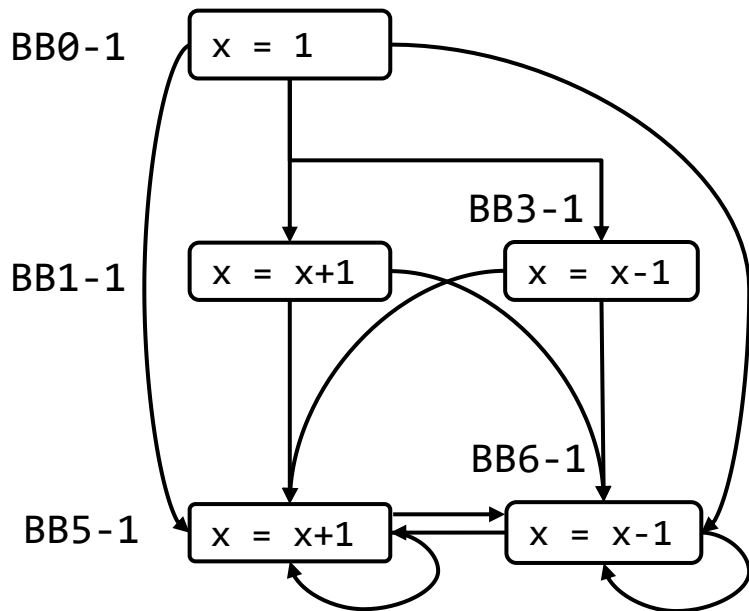
n	IN(n)	OUT(n)
BB4-1	{<x,0-1,1-1,3-1,5-1,6-1> <y,0-1,7-1> <z,0-3,4-2>}	
BB4-2		
BB5-1	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1,7-1> <z,4-2>}	
BB7-1		
BB8-1		
BB4-1		
BB5-1		
BB6-1		
BB7-1		

第3轮



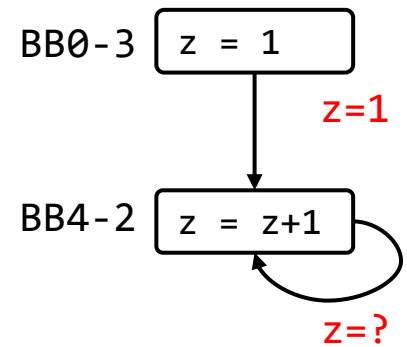
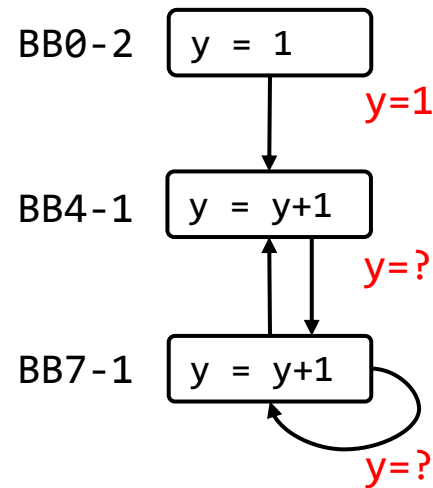
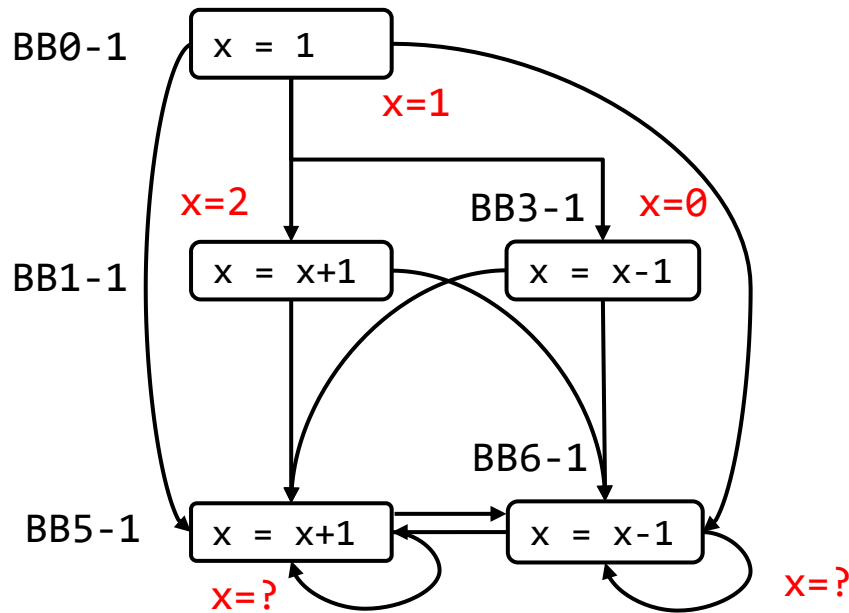
n	IN(n)	OUT(n)
BB4 - 1	{<x,0-1,1-1,3-1,5-1,6-1> <y,0-1,7-1> <z,0-3,4-2>}	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,0-3,4-2>}
BB4 - 2	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,0-3,4-2>}	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,4-2>}
BB5 - 1	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,4-2> {<x,5-1,6-1> <y, 7-1> <z,4-2>}	{<x,5-1> <y,4-1,7-1> <z,4-2>}
BB7 - 1	{<x,5-1> <y,4-1,7-1> <z,4-2> {<x,6-1><y,4-1><z,4-2>}	{<x,5-1,6-1> <y, 7-1> <z,4-2>}
BB8 - 1	{<x,5-1,6-1> <y, 7-1> <z,4-2>}	-
BB4 - 1	{<x,0-1,1-1,3-1,5-1,6-1> <y,0-1,7-1> <z,0-3,4-2> {<x,5-1,6-1> <y, 7-1> <z,4-2>}	
BB5 - 1	{<x,0-1,1-1,3-1,5-1,6-1> <y,0-1,7-1> <z,0-3,4-2> {<x,5-1,6-1> <y, 7-1> <z,4-2>}	
BB6 - 1	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,4-2>}	{<x,6-1> <y,4-1> <z,4-2>}
BB7 - 1		

构建数值流图Value-Flow Graph



基于Value-Flow进行常量分析

- Sparse value-flow analysis



VFG的应用

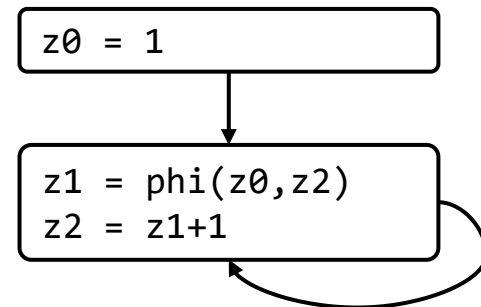
- 代码优化：
 - 代码移动（如公共表达式提取）
 - 延迟计算
 - ...
- 缺陷检测：
 - 为初始化的变量、指针
 - 除数是否可能为0

大纲

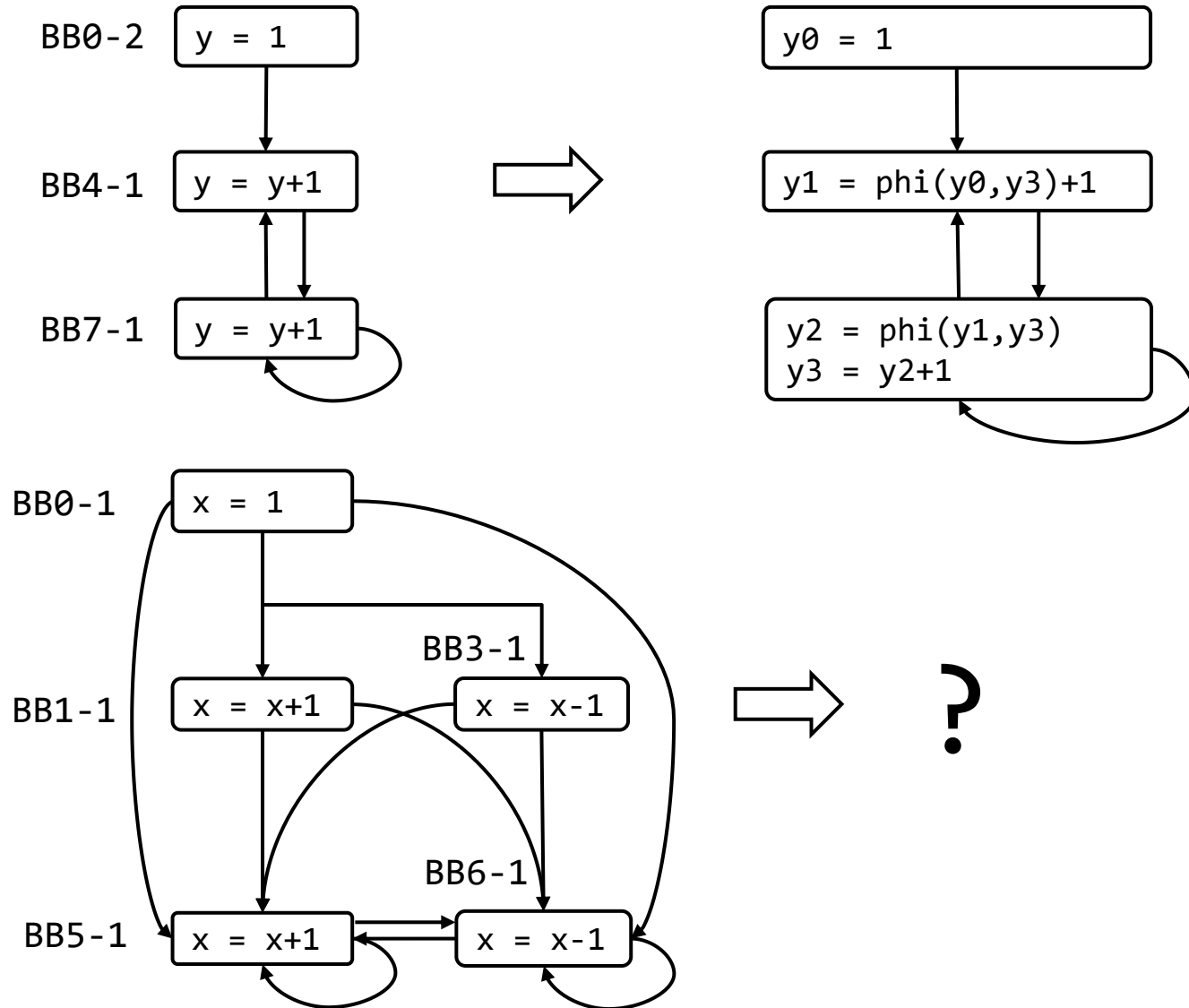
- 一、常量分析和优化
- 二、可达性分析和数值流图
- 三、静态单赋值形式

静态单赋值形式

- 传统数据流分析需要考虑所有指令，效率低
- VFG需要另外构建图
- 将def-use关系融入到中间代码中？
 - SSA (Static Single Assignment)
 - 1988年Barry K. Rosen等人提出SSA
- 提取变量的def-use关系，简化数据流分析过程
 - 每个变量仅被赋值1次
 - 使用phi指令解决控制流带来的 (def_1, def_2) -use问题
 - 如 $\%3 = \text{phi}(\%1, \%2)$
 - 分析数据流关系无需再考虑CFG



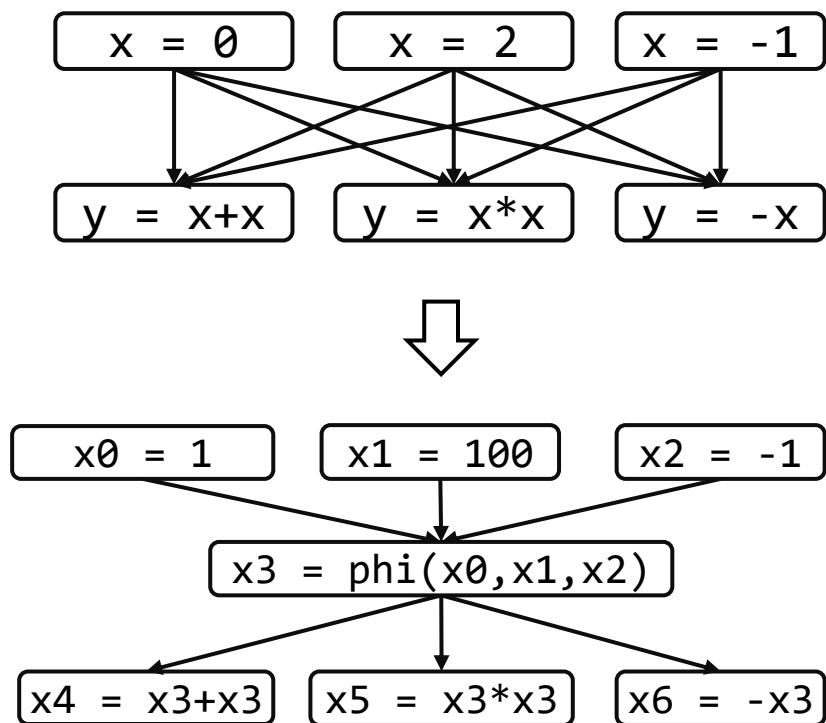
VFG=>单赋值形式



SSA简化def-use关系

- 原始程序的def-use关系数量是 $O(m \times n)$;
- SSA的def-use数量减少为 $O(m + n)$ 。

```
match v1:
  0 => { x = 0; }
  1 => { x = 1; }
  _ => { x = -1; }
...
match v2:
  0 => { x = x + x; }
  1 => { x = x * x; }
  _ => { x = -x; }
```



IR=>SSA

- 解决局部变量load-store的问题，跨代码块使用临时变量
- 关键问题：
 - 分析def-use关系
 - 插入phi指令

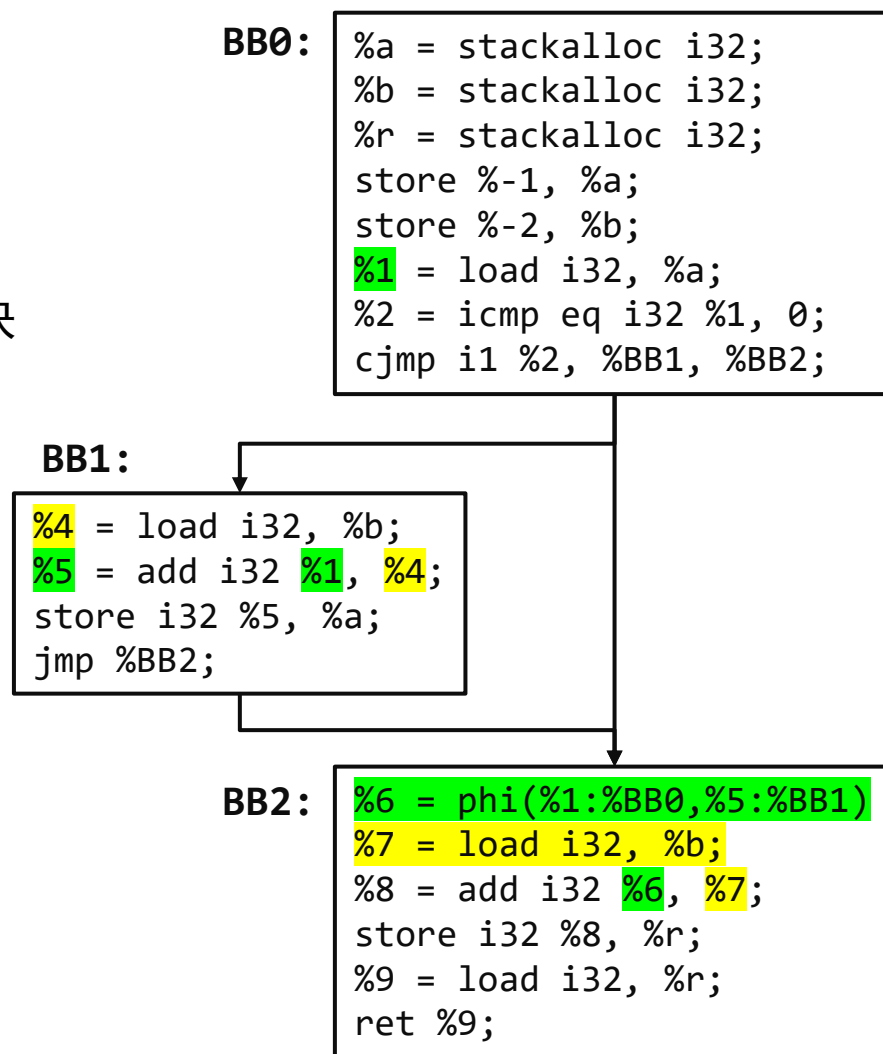
```
fn foo(a:int, b:int)->int {  
    if(a==0)  
        a = a + b;  
    let r:int = a + b;  
    return r;  
}
```



```
define fn i32 foo(i32 %a, i32 %b){  
%BB0:  
    %a = stackalloc i32;  
    %b = stackalloc i32;  
    %r = stackalloc i32;  
    store %-1, %a;  
    store %-2, %b;  
    %1 = load i32, %a;  
    %2 = icmp eq i32 %1, 0;  
    cjmp i1 %2, %BB1, %BB2;  
%BB1:  
    %3 = load i32, %a;  
    %4 = load i32, %b;  
    %5 = add i32 %3, %4;  
    store i32 %5, %a;  
    jmp %BB2;  
%BB2:  
    %6 = load i32, %a;  
    %7 = load i32, %b;  
    %8 = add i32 %6, %7;  
    store i32 %8, %r;  
    %9 = load i32, %r;  
    ret %9;  
}
```

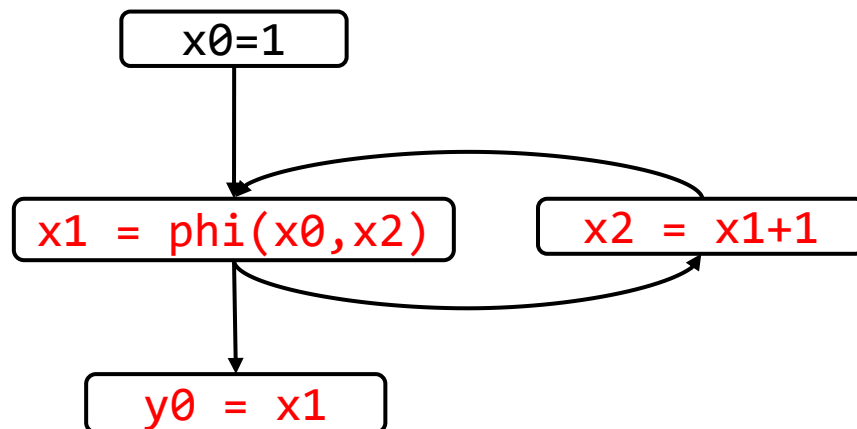
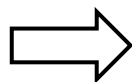
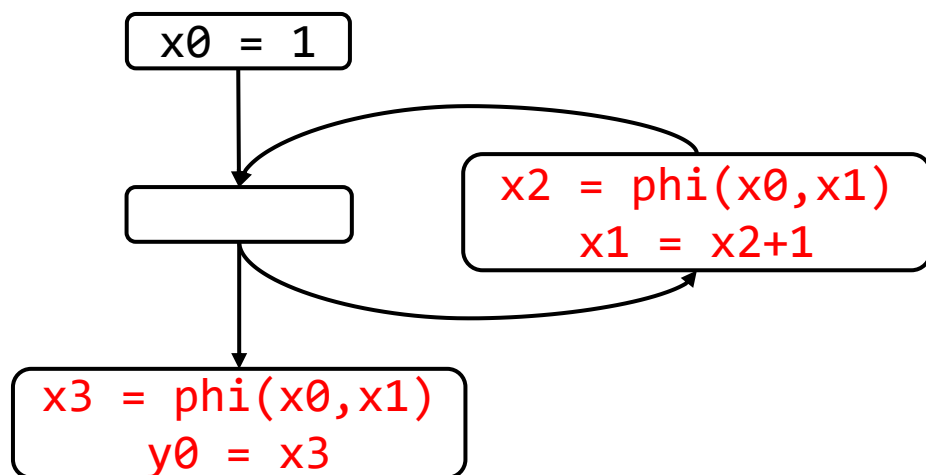
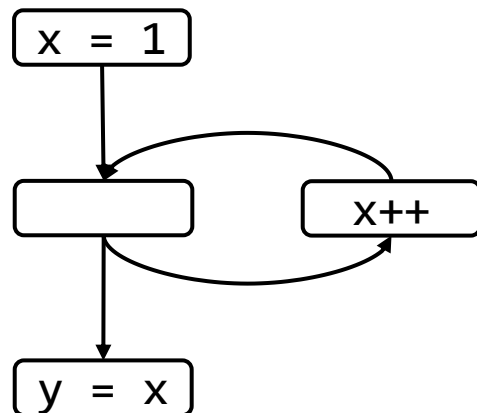
举例

- 数据流分析确定def-use
- Phi指令放置思路：
 - 思路一：
 - 该代码块use(x)
 - 多个def(x)可到达该代码块
 - 思路二：入度>1的节点
 - 根据支配边界优化



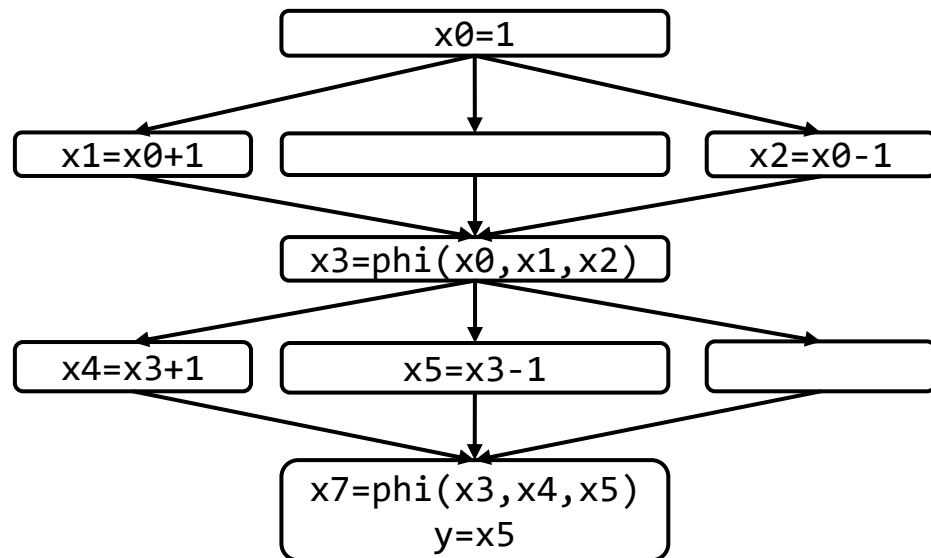
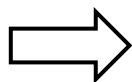
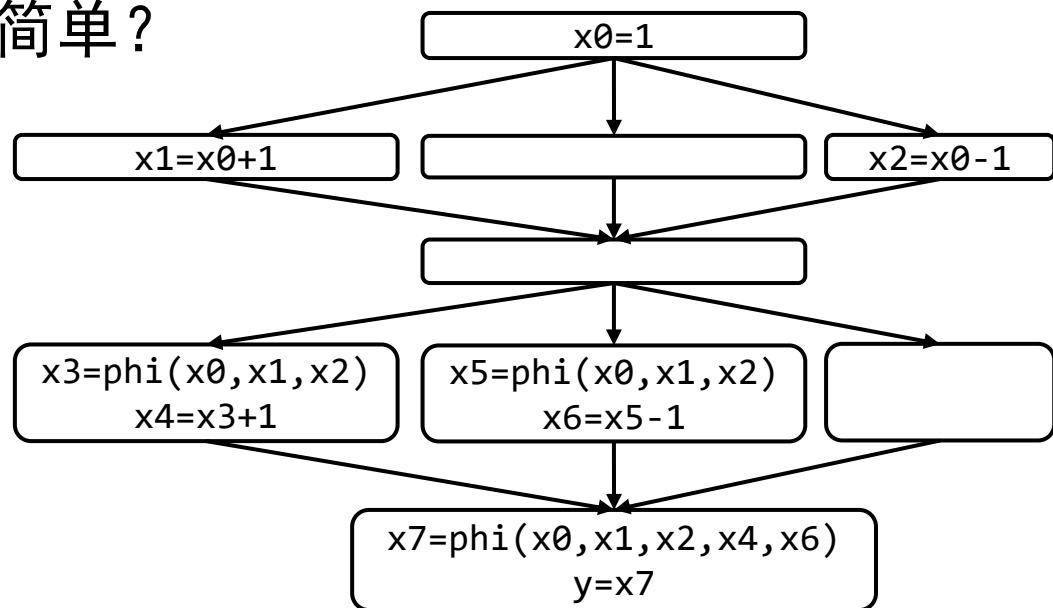
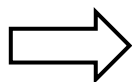
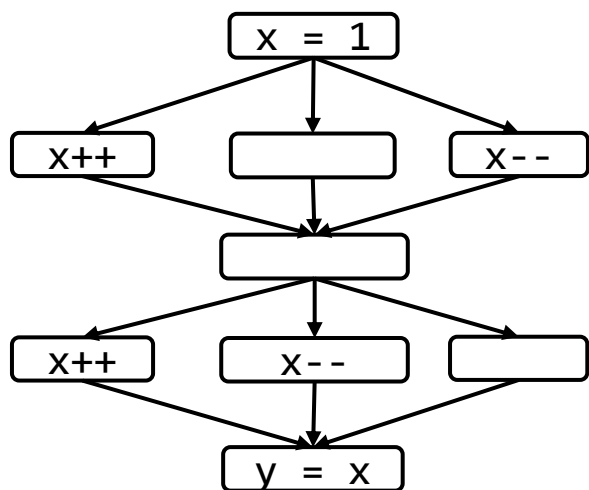
案例对比1

- 哪个方案更优？



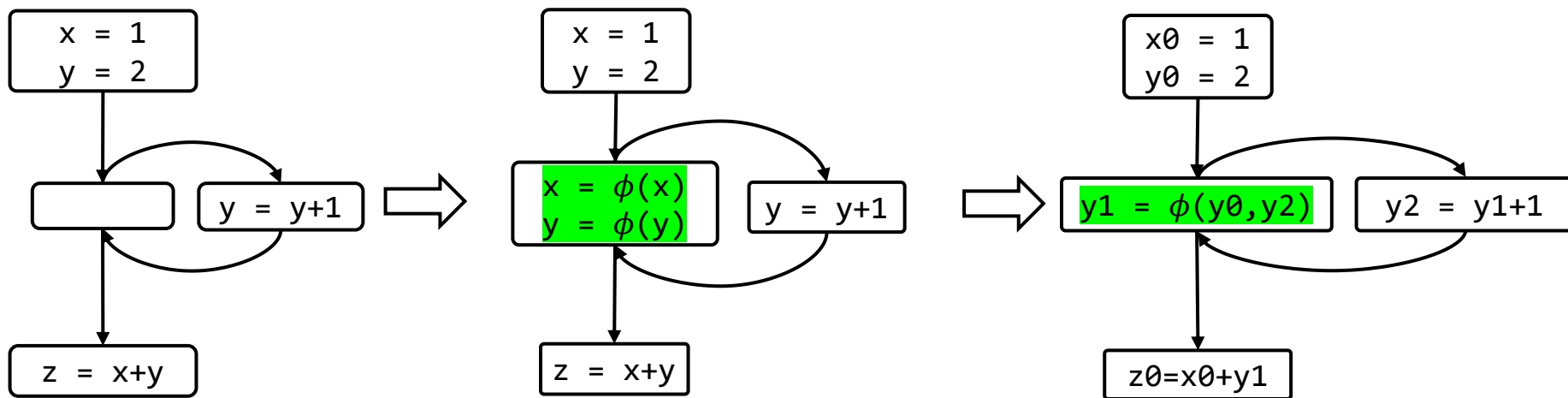
案例对比2

- 哪个方案def-use更简单？

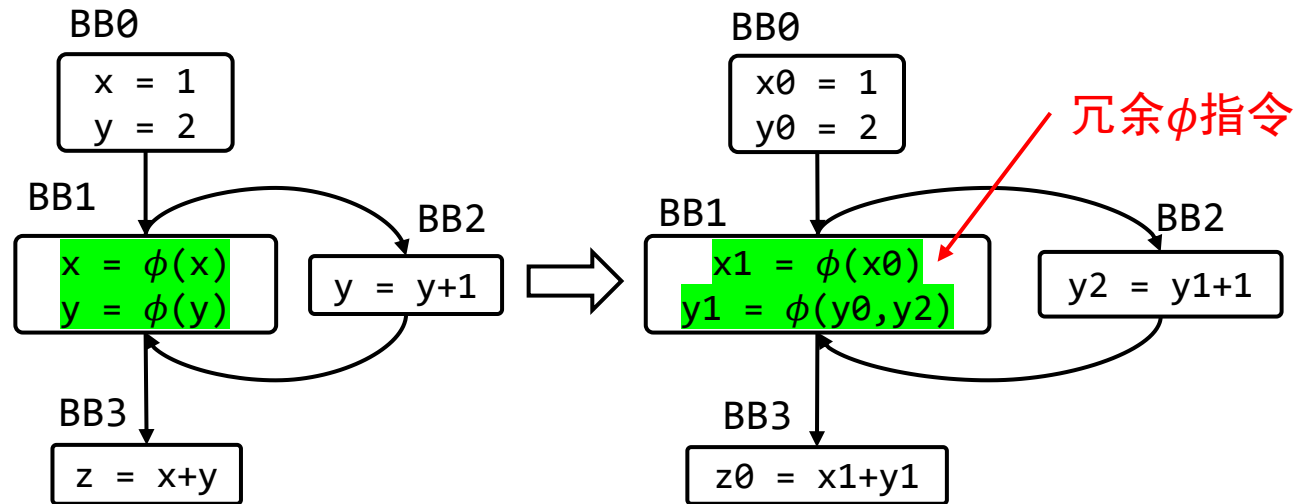


多个变量的情况

- 先插入phi节点, 然后通过def-use分析编号
- 问题: 可能存在冗余phi节点

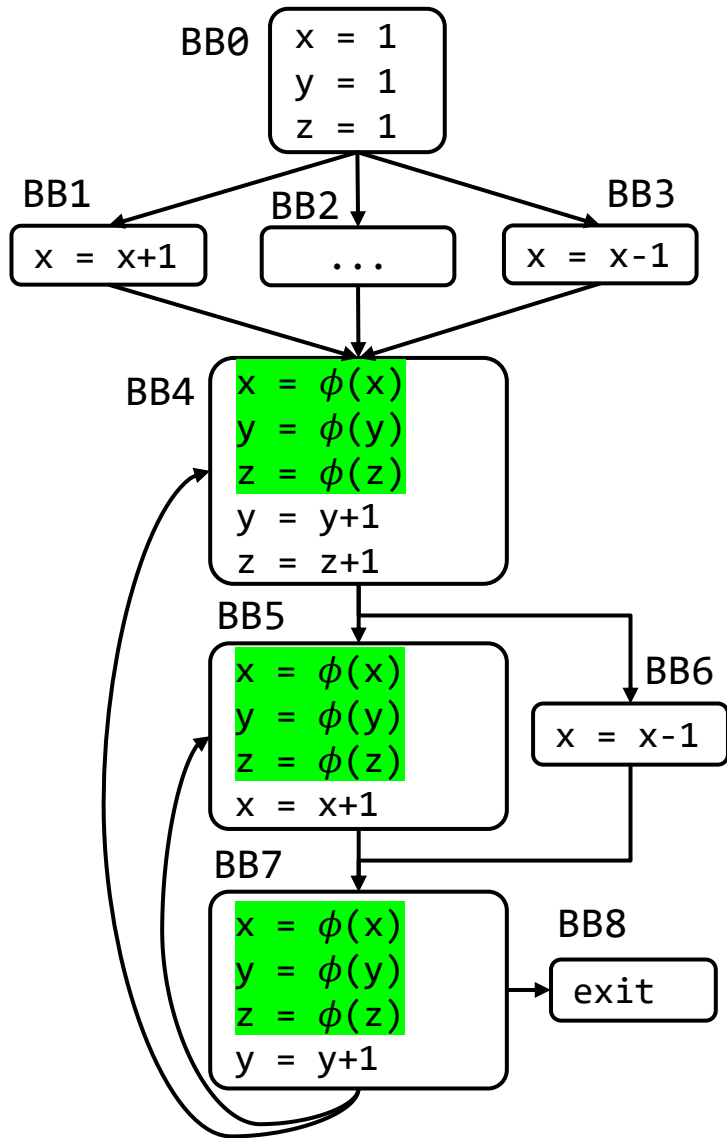


遍历控制流图构建SSA



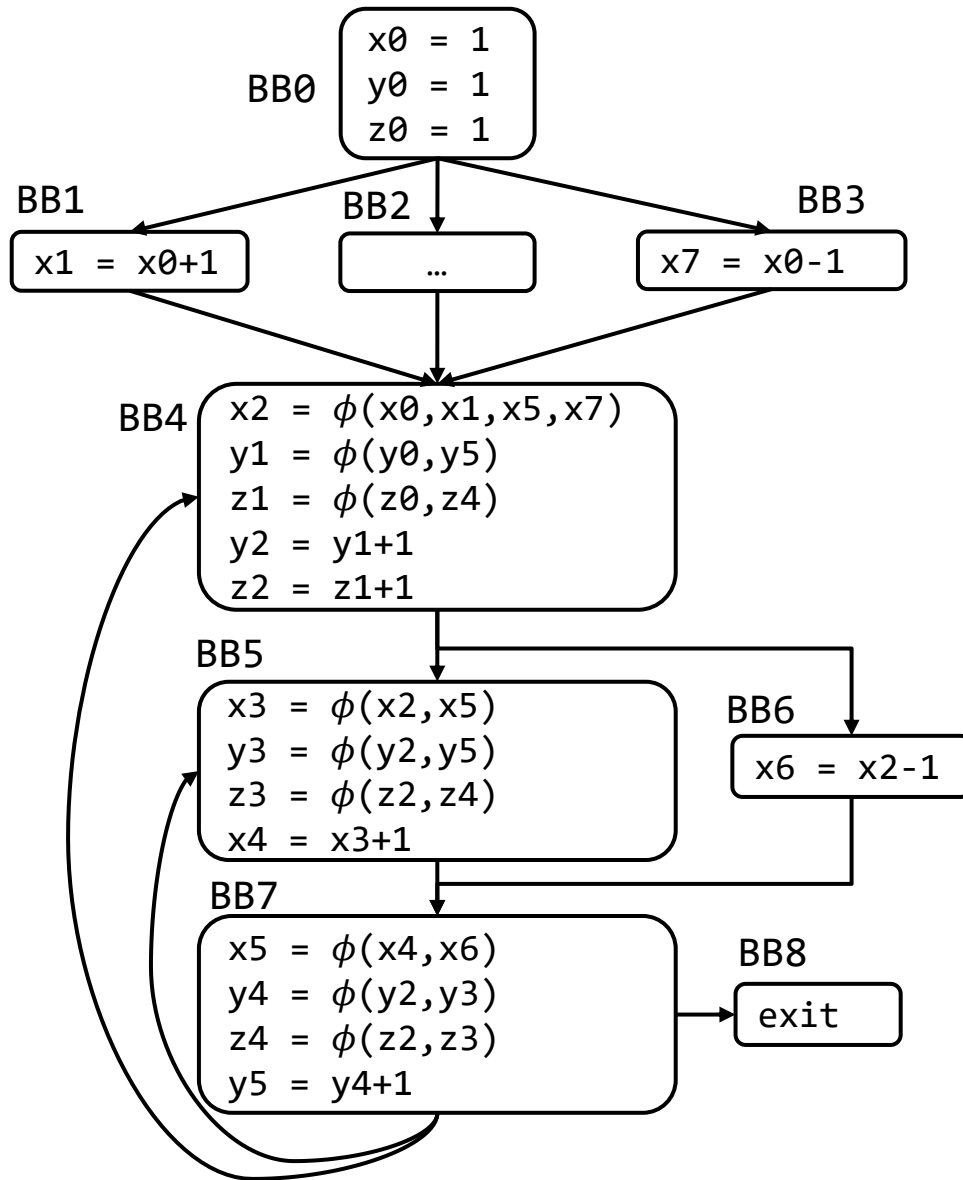
- DFS遍历控制流图构建SSA，顺序：BB0- \rightarrow BB1- \rightarrow BB2- \rightarrow BB1- \rightarrow BB3
 - BB0: $x_0=1, y_0=2$
 - BB1: $x_1=\phi(x_0), y_1=\phi(y_0)$
 - BB2: $y_2=y_1+2$
 - BB1: $x_1=\phi(x_0), y_1=\phi(y_0, y_2)$
 - BB3: $z_0=x_1+y_1$
- Worklist算法开销：
 - 每个节点需要更新次数为其入度
 - $\forall bb_i \rightarrow bb_j \in CFG, \text{Update}(bb_j)$

练习：遍历控制流图构建SSA



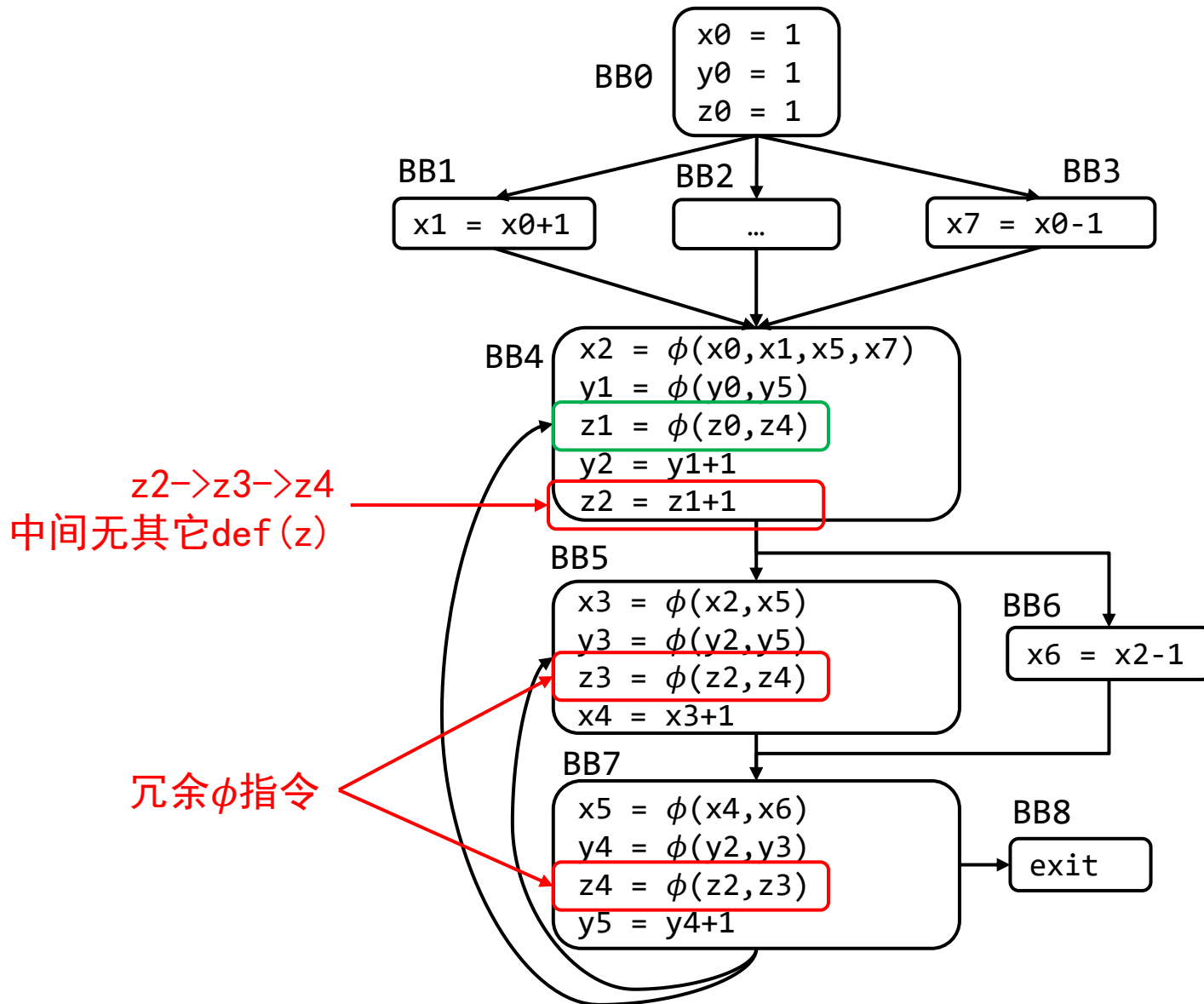
- DFS顺序：BB0->BB1->BB4->BB5->BB7->BB8->BB6->BB2->BB3
 - BB0: $x_0=1$, $y_0=1$, $z_0=1$
 - BB1: ...

结果



- DFS顺序: BB0-→BB1-→BB4-→BB5-→BB7-→BB8-→BB6-→BB2-→BB3
- BB0: $x_0=1$, $y_0=1$, $z_0=1$
- BB1: $x_1=x_0+1$
- BB4: $x_2=\phi(x_1)$, $y_1=\phi(y_0)$, $z_1=\phi(z_0)$, $y_2=y_1+1$, $z_2=z_1+1$
- BB5: $x_3=\phi(x_2)$, $y_3=\phi(y_2)$, $z_3=\phi(z_2)$, $x_4=x_3+1$
- BB7: $x_5=\phi(x_4)$, $y_4=\phi(y_3)$, $z_4=\phi(z_3)$, $y_5=y_4+1$
- BB8:
- BB4: $x_2=\phi(x_1, x_5)$, $y_1=\phi(y_0, y_5)$, $z_1=\phi(z_0, z_4)$
- BB5: $x_3=\phi(x_2, x_5)$, $y_3=\phi(y_2, y_5)$, $z_3=\phi(z_2, z_4)$,
- BB6: $x_6 = x_2 - 1$
- BB7: $x_5=\phi(x_4, x_6)$, $y_4=\phi(y_2, y_3)$, $z_4=\phi(z_2, z_3)$
- BB2:
- BB4: $x_2=\phi(x_0, x_1, x_5)$
- BB3: $x_7=x_0-1$
- BB4: $x_2 = \phi(x_0, x_1, x_5, x_7)$

冗余Phi指令

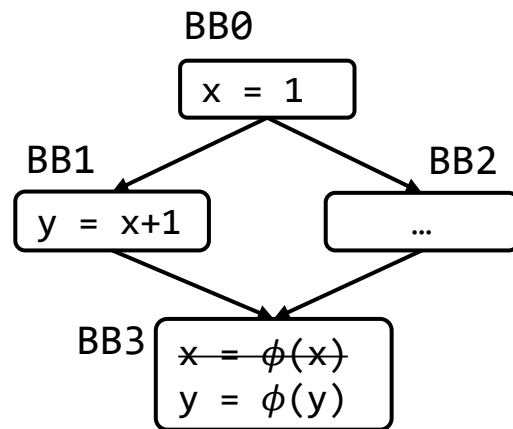


Phi指令放置思路

- Phi指令放置思路：
 - 思路一：该代码块 $use(x)$ ，且多个 $def(x)$ 可到达该代码块
 - 和VFG类似，不能简化 $def-use$ 关系数量
 - 不是最优方案
 - 思路二：入度 >1 的节点
 - 缺点：引入冗余的Phi指令
 - 根据支配边界优化

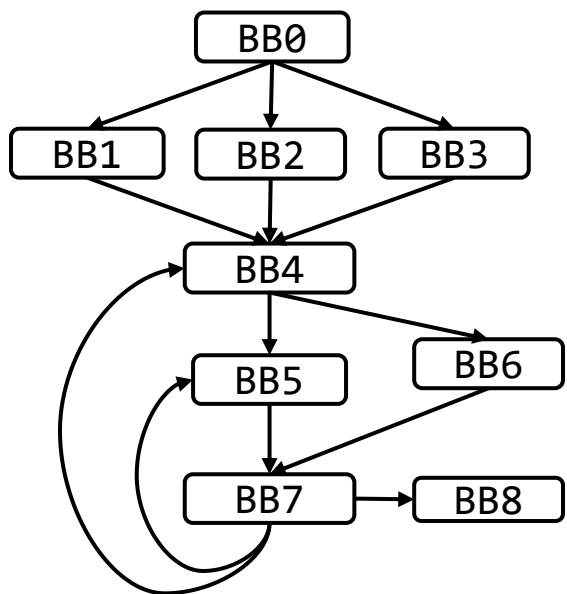
基于支配边界优化Phi指令的设置

- BB0支配BB2，BB1和BB2的支配边界都是BB3
- 如果BB1和BB2中都没有def(x)，BB3不需要phi(x)，可直接使用BB0中的def(x)。
- 如果BB1中有def(y)，BB3中一定需要phi(y)
 - 为什么？



支配的基本概念

- 给定有向图 $G(V, E)$ 与起点 v_0 ，如果从 v_0 到某个点 v_j 均需要经过点 v_i ，则称 v_i 支配 v_j 或 v_i 是 v_j 的一个支配点。
 - $v_i \in Dom(v_j)$
- 如果 $v_i \neq v_j$ ，则称 v_i 严格支配 v_j 。



控制流图

$$Dom(bb_0) = \{bb_0\}$$

$$Dom(bb_1) = \{bb_0, bb_1\}$$

$$Dom(bb_2) = \{bb_1, bb_2\}$$

$$Dom(bb_3) = \{bb_0, bb_3\}$$

$$Dom(bb_4) = \{bb_0, bb_4\}$$

$$Dom(bb_5) = \{bb_0, bb_4, bb_5\}$$

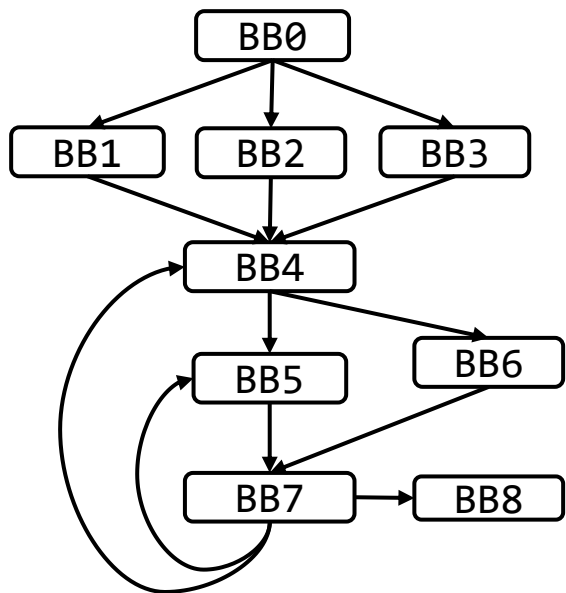
$$Dom(bb_6) = \{bb_0, bb_4, bb_6\}$$

$$Dom(bb_7) = \{bb_0, bb_4\}$$

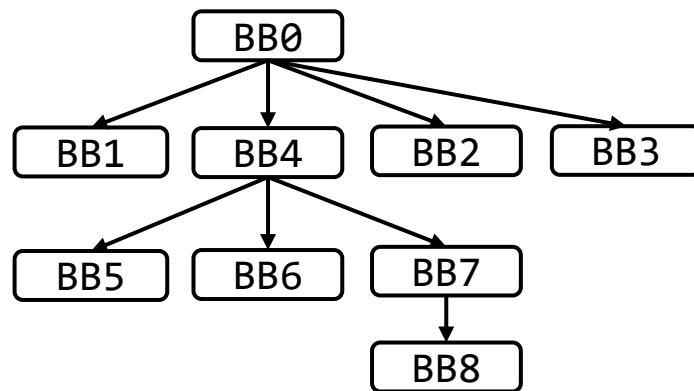
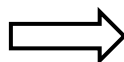
$$Dom(bb_8) = \{bb_0, bb_4, bb_7\}$$

支配树的基本概念

- 所有 v_j 的严格支配点中与 v_j 最接近的点成为 v_j 的最近支配点。
 - $Idom(v_j) = v_i$, v_j 的其它严格支配点均严格支配 v_i 。
- 连接接所有的最近支配关系, 形成一棵支配树。
 - 根节点外的每一点均存在唯一的最近支配点。



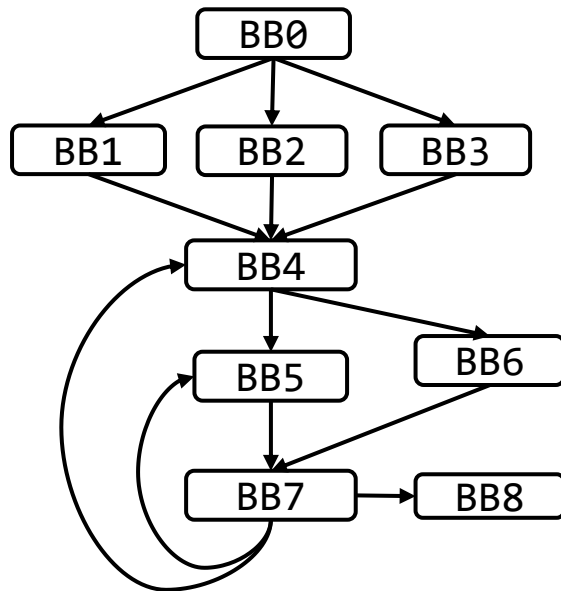
控制流图



支配树

支配边界Dominance Frontier

- v_i 的支配边界是所有满足条件的 v_j 的集合
 - v_i 支配 v_j 的一个前序节点
 - v_i 并不严格支配 v_j



控制流图

$$\begin{aligned} DF(bb_0) &= \{\} \\ DF(bb_1) &= \{bb_4\} \\ DF(bb_2) &= \{bb_4\} \\ DF(bb_3) &= \{bb_4\} \\ DF(bb_4) &= \{bb_4\} \\ DF(bb_5) &= \{bb_7\} \\ DF(bb_6) &= \{bb_7\} \\ DF(bb_7) &= \{bb_4, bb_5\} \\ DF(bb_8) &= \{\} \end{aligned}$$

利用支配边界设置Phi指令

- 初始化：枚举所有变量的def-sites

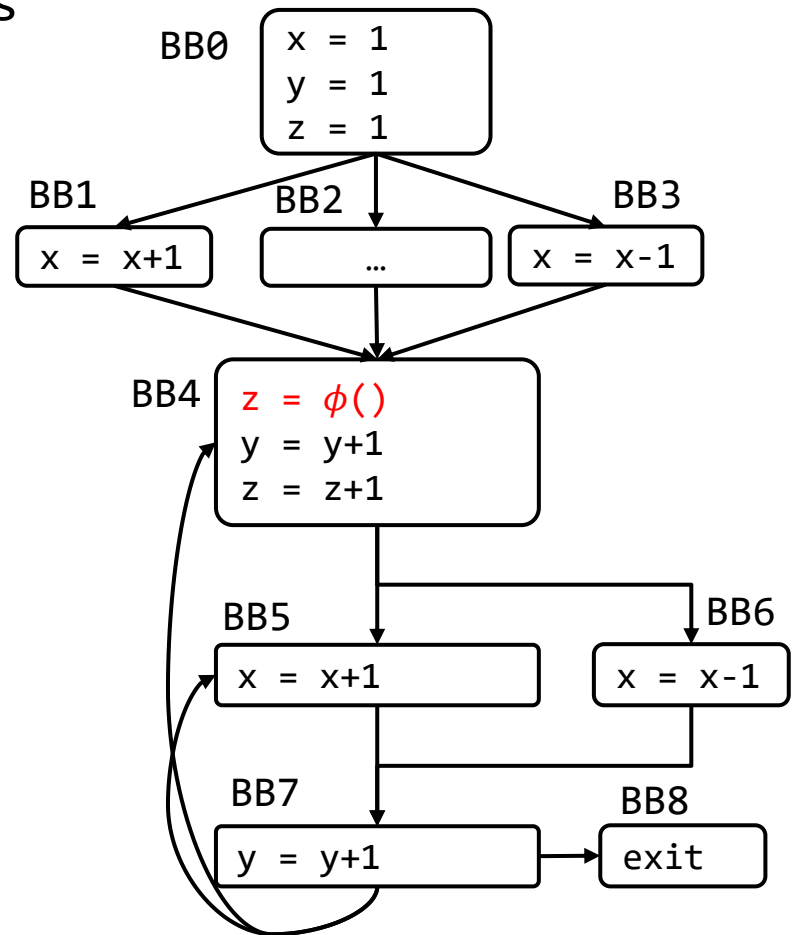
- $\text{def-sites}(x) = \{\text{BB0}, \text{BB1}, \text{BB3}, \text{BB5}, \text{BB6}\}$
- $\text{def-sites}(y) = \{\text{BB0}, \text{BB4}, \text{BB7}\}$
- $\text{def-sites}(z) = \{\text{BB0}, \text{BB4}\}$

- 为每个变量在 BB_j 增加phi节点：

- $\text{BB}_i \in \text{def-sites}(x)$
- $\text{BB}_j \in \text{DF}(\text{BB}_i)$

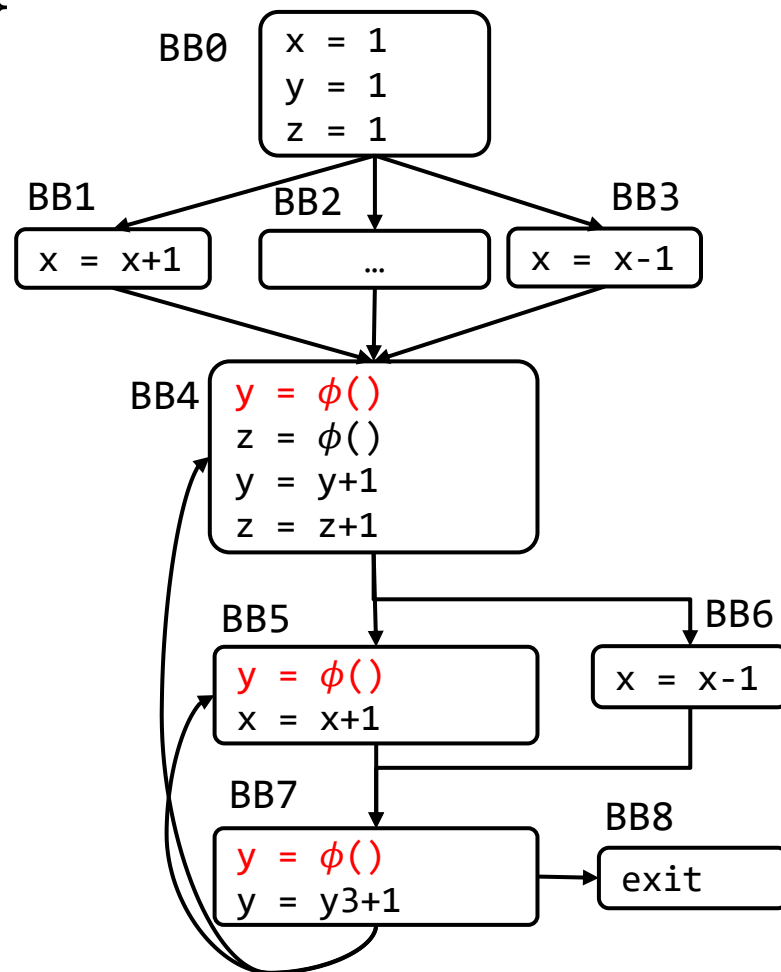
- 以变量 z 为例：

- $\text{def-sites}(z) = \{\text{BB0}, \text{BB4}\}$
 - $\text{DF}(\text{BB}_0) = \{\}$
 - $\text{DF}(\text{BB}_4) = \{\text{BB}_4\}$
 - 在 BB_4 增加phi指令的 $\text{phi}(z)$



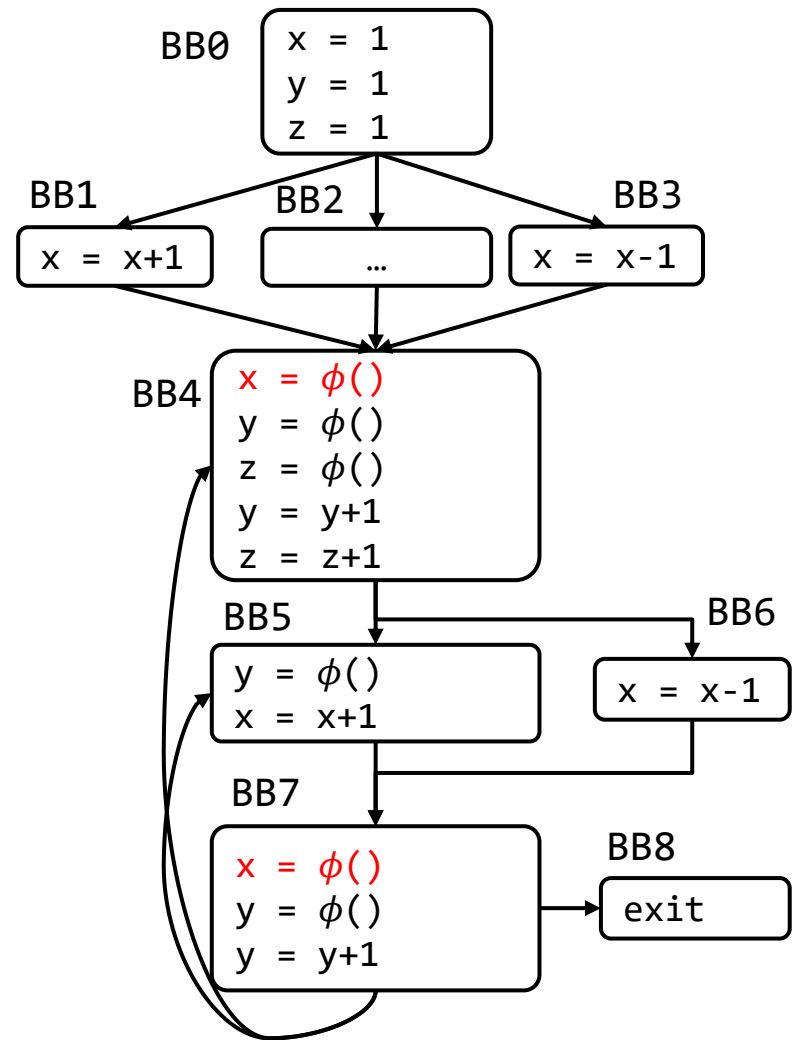
为变量y的插入Phi指令

- $\text{def-sites}(y) = \{\text{BB}_0, \text{BB}_4, \text{BB}_7\}$
 - $\text{DF}(\text{BB}_0) = \{\}$
 - $\text{DF}(\text{BB}_4) = \{\text{BB}_4\}$
 - 在 BB_4 增加phi指令的 $\phi(y)$
 - $\text{DF}(\text{BB}_7) = \{\text{BB}_4, \text{BB}_5\}$
 - 在 BB_5 增加phi指令的 $\phi(y)$
 - 将 BB_5 到 $\text{def-sites}(y)$
 - $\text{DF}(\text{BB}_5) = \{\text{BB}_7\}$
 - 在 BB_7 增加phi指令的 $\phi(y)$



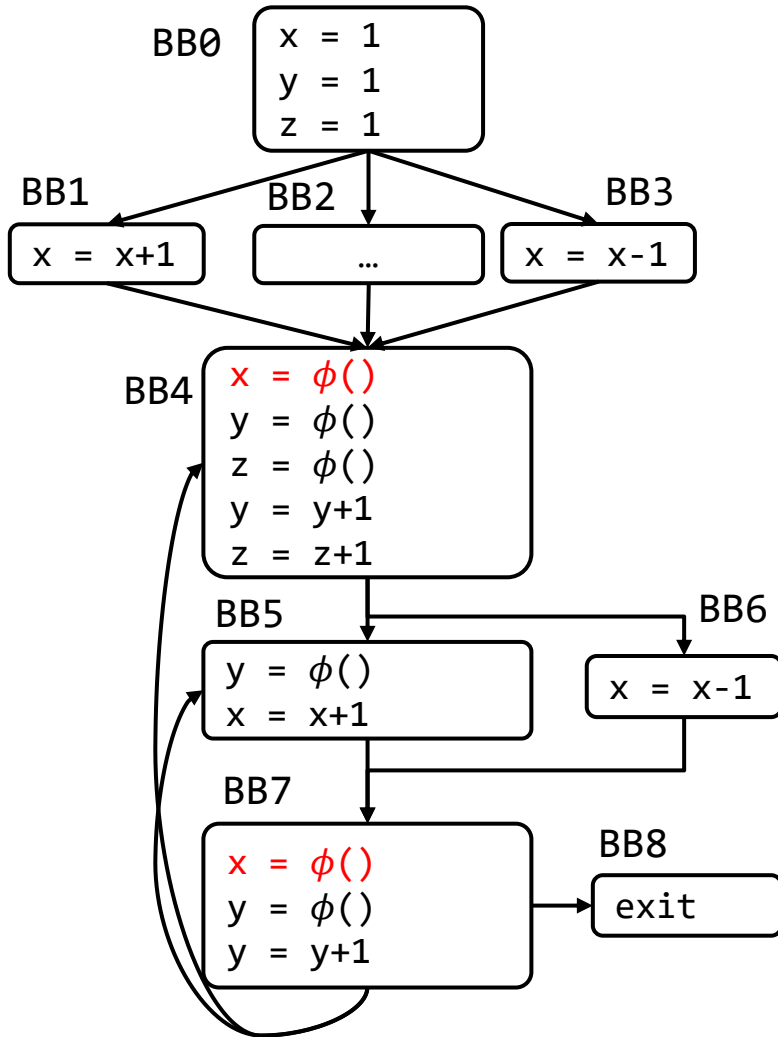
为变量x的插入Phi指令

- $\text{def-sites}(x) = \{BB_0, BB_1, BB_3, BB_5, BB_6\}$
 - $DF(BB_0) = \{\}$
 - $DF(BB_4) = \{BB_4\}$
 - 在 BB_4 增加phi指令的 $\phi(x)$
 - $DF(BB_3) = \{BB_4\}$
 - $DF(BB_5) = \{BB_7\}$
 - 在 BB_4 增加phi指令的 $\phi(x)$
 - $DF(BB_6) = \{BB_7\}$



遍历控制流图构建SSA

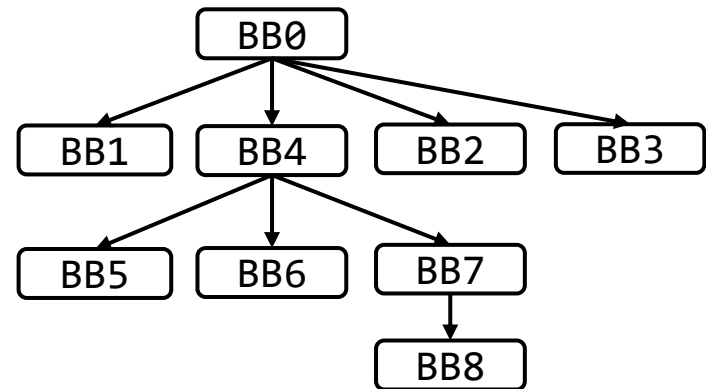
- DFS顺序: BB0->BB1->BB4->BB5->BB7->BB8->BB6->BB2->BB3
 - BB0: $x_0=1, y_0=1, z_0=1$
 - BB1: ...



如何构建支配树：主要思路

$$Dom(v) = \begin{cases} \{v\}, & \text{if } v = v_0 \\ \{v\} \cup \left(\bigcap_{p \in pred(v)} Dom(p) \right), & \text{if } v \neq v_0 \end{cases}$$

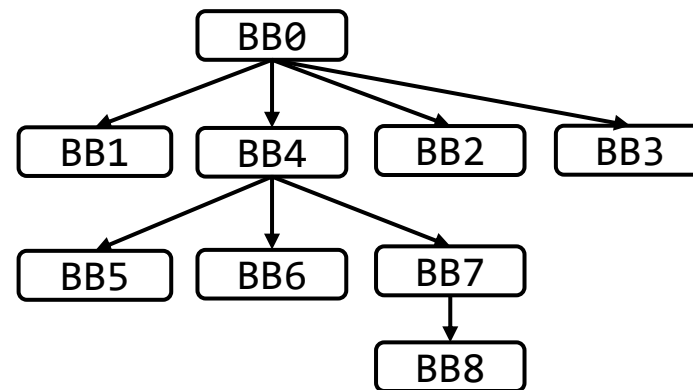
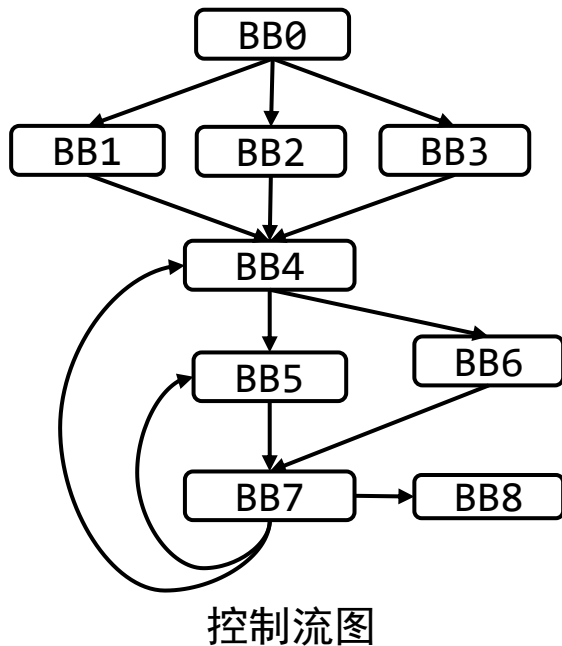
$Dom(bb_0) = \{bb_0\}$
 $Dom(bb_1) = \{bb_0, bb_1\}$
 $Dom(bb_2) = \{bb_1, bb_2\}$
 $Dom(bb_3) = \{bb_0, bb_3\}$
 $Dom(bb_4) = \{bb_0, bb_4\}$
 $Dom(bb_5) = \{bb_0, bb_4, bb_5\}$
 $Dom(bb_6) = \{bb_0, bb_4, bb_6\}$
 $Dom(bb_7) = \{bb_0, bb_4\}$
 $Dom(bb_8) = \{bb_0, bb_4, bb_7\}$



支配树

如何求支配边界：主要思路

- 什么节点会成为支配边界？
 - 入度 >1
- 节点 v 是谁的支配边界？
 - v 的所有前序节点，非支配节点： $Pred(v) - IDom(v)$
 - 所有前序节点的直接支配节点 $\bigcup_{v_p \in Pred(v) - IDom(v)} IDom(v_p) - IDom(v)$
 - 迭代下去直到遇到 v 的直接支配节点 $IDom(v)$



$$\begin{aligned} IDF(bb_4) &= \{bb_1, bb_2, bb_3, bb_7, bb_4\} \\ IDF(bb_5) &= \{bb_7\} \\ IDF(bb_7) &= \{bb_5, bb_6\} \end{aligned}$$

SSA的应用

- 作用和VFG类似
 - 代码优化：常量传播、代码移动...
 - 缺陷检测：为初始化的变量或指针、除数为0、...
- 在中间代码层获得广泛应用
 - 是很多编译器优化算法的基础

思考：SSA是否可能出错？

```
fn foo(int x) -> int{  
    let a:int = 0;  
    let c = &a;  
    c = x + 1;  
    let r = a+1;  
    return r;  
}
```


总结

- 利用常量传播和折叠进行优化的思想
- 可达性分析
 - 常量分析
- 通过数据流分析方法构建value-flow graph
 - def-use关系
- 静态单赋值形式
 - phi指令插入
 - 基于支配边界
 - def-use分析