Lecture 5.1

数据流分析和代码优化

徐 辉

xuh@fudan.edu.cn

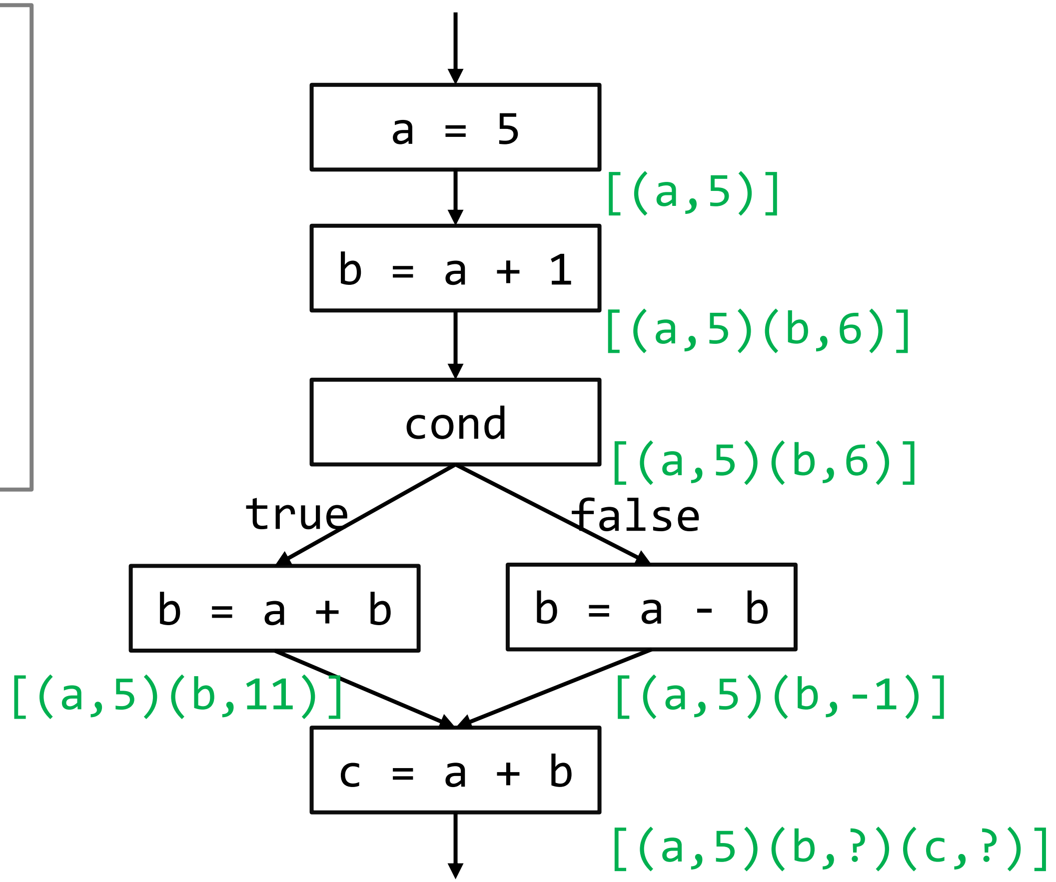
大纲

- 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 = lvalue
  - 如果lvalue的形式为常量c，则x的状态为“常量c”
  - 如果lvalue的形式为变量y，则x的状态等同于y的状态
  - 如果lvalue的形式为“y op z”，则x的状态为：
    - y 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lvalue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

常量分析和优化

- 常量传播（constant propagation）
- 常量折叠（constant folding）

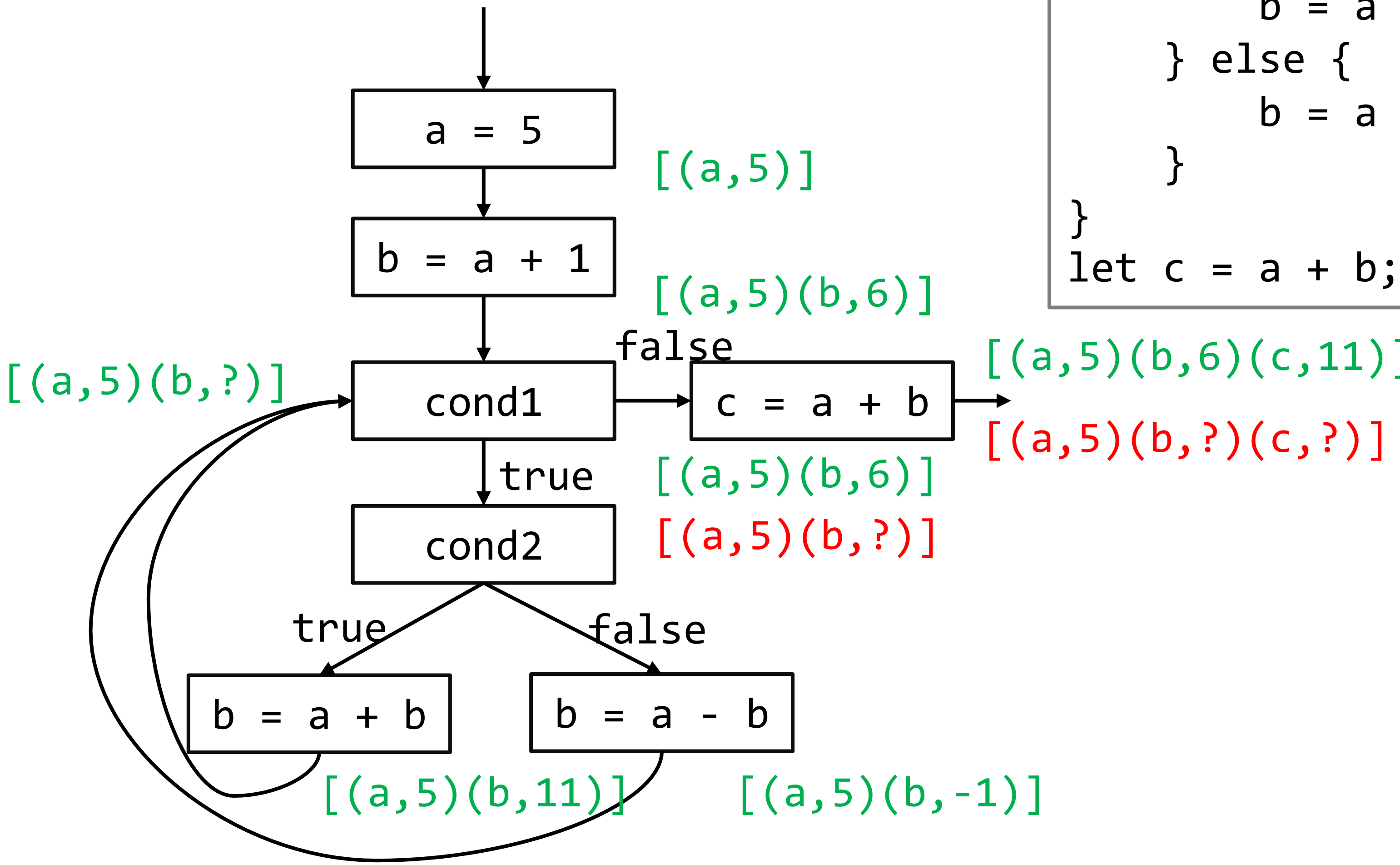
```
let a:const int = 5;
let b = a + 1;
if(cond) {
    b = a + b;
} else {
    b = a - b;
}
let c = a + b;
```

```
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

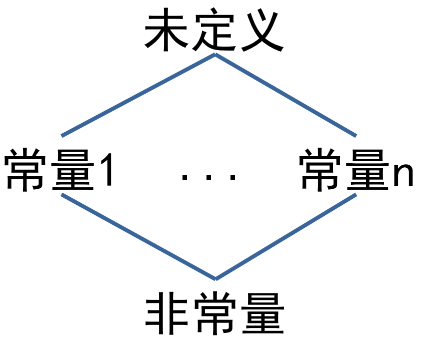
- Chaotic iteration

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



基于Lattice的方法

- 使用poset（V,∧）对变量的状态建模
  - patially ordered set
    - 未定义 ≥ 常量x
    - 常量x ≥ 非常量
  - top element：未定义
  - bottem element：非常量
  - Meet运算
    - 未定义 ∧ 常量x = 常量x
    - 常量x ∧ 常量y = 非常量
    - 常量x ∧ 非常量 = 非常量



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] = Ø
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] ∩ 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. 结束条件：fixed point

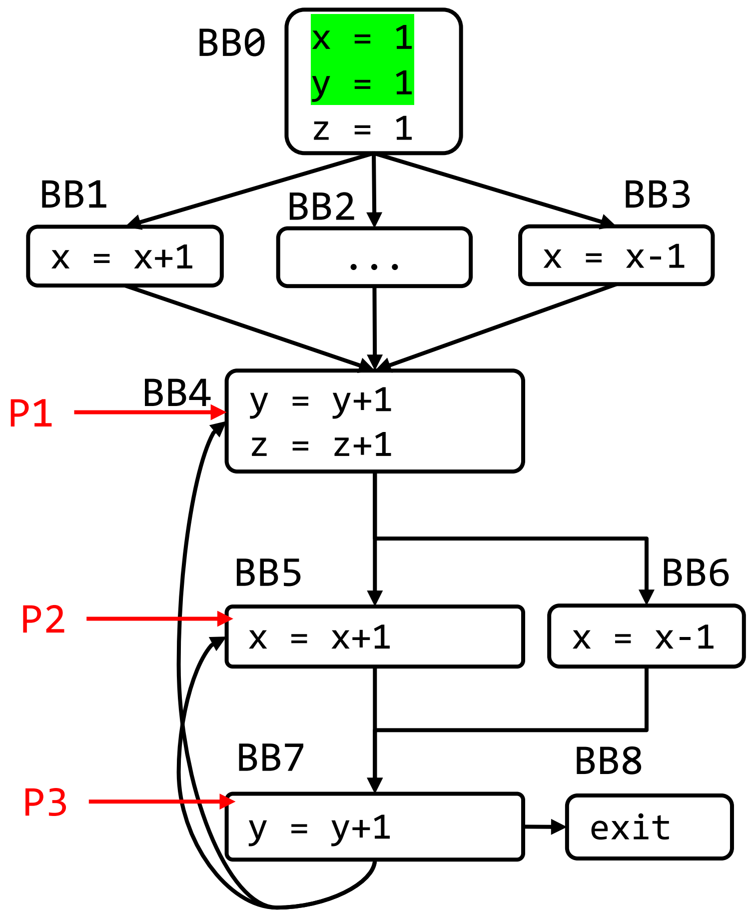
常量分析小节

- 本质：编译时计算
- 应用：
  - 常量折叠
  - 无效代码删除

大纲

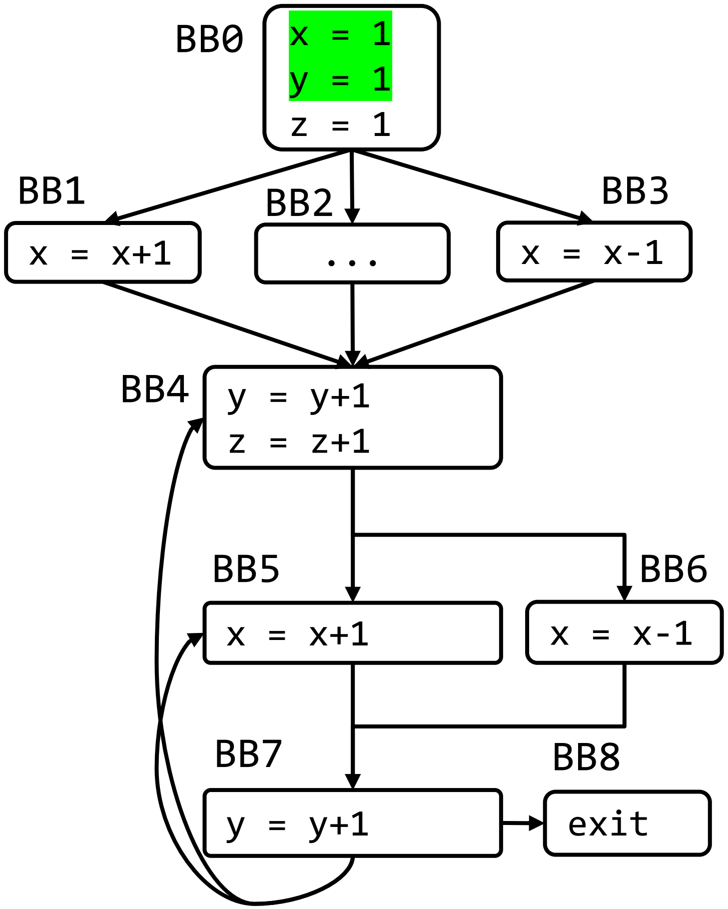
- 一、常量分析和优化
- 二、可达性分析和数值流图
- 三、静态单赋值形式

可达性分析：分析def-use关系



- 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

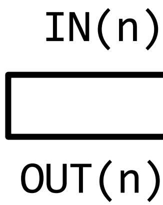
分析方法



- 为每条语句分配一个编号
  - BB0-1: x=1
  - BB0-2: y=1
  - ...
- 用IN(n)表示节点的入向属性集合。
- 用OUT(n)表示节点的出向属性集合。
- 遍历控制流图并应用Transfer和Join函数计算每一个节点的IN(n)和OUT(n)。
- 直到IN(n)和OUT(n)不再变化。

定义Transfer和Join函数

Transfer



$$\mathrm{OUT(n)} = (\mathrm{IN(n)} - \mathrm{KILL(n)}) \cup \mathrm{Gen(n)}$$

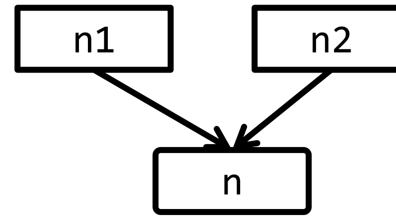
n: $\boxed{\text{cond=?}}$

$$\mathrm{Gen(n)} = \emptyset$$
$$\mathrm{KILL(n)} = \emptyset$$

n: $\boxed{\text{x=a}}$

$$\mathrm{Gen(n)} = \{< x, n >\}$$
$$\mathrm{KILL(n)} = \{< x, m >: m \neq n\}$$

Join



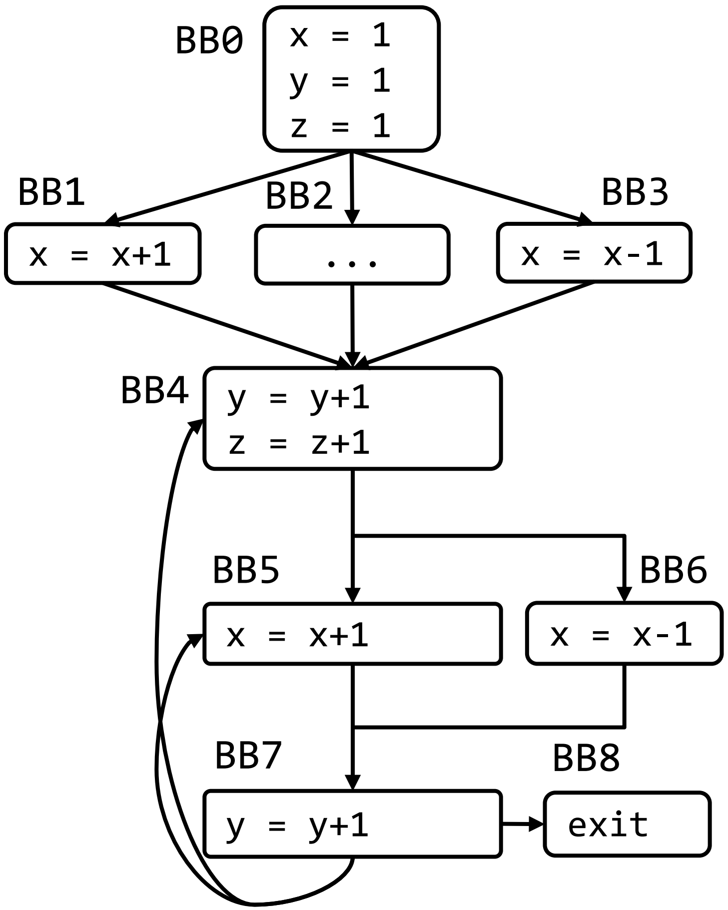
$$\mathrm{IN(n)} = \mathrm{OUT(n1)} \cup \mathrm{OUT(n2)}$$

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

Chaotic Iteration

```
For (each node n):
    IN[n] = OUT[n] = Ø
OUT[entry] = {<v, ?>: v is a program variable}
Repeat:
    For(each node n):
        For(each n's predecessor p)
            IN[n] = IN[n] ∪ OUT[p]
        OUT(n)=(IN[n]-KILL(n)) ∪ Gen(n)
Until IN[n] and OUT[n] stops changing for all n
```

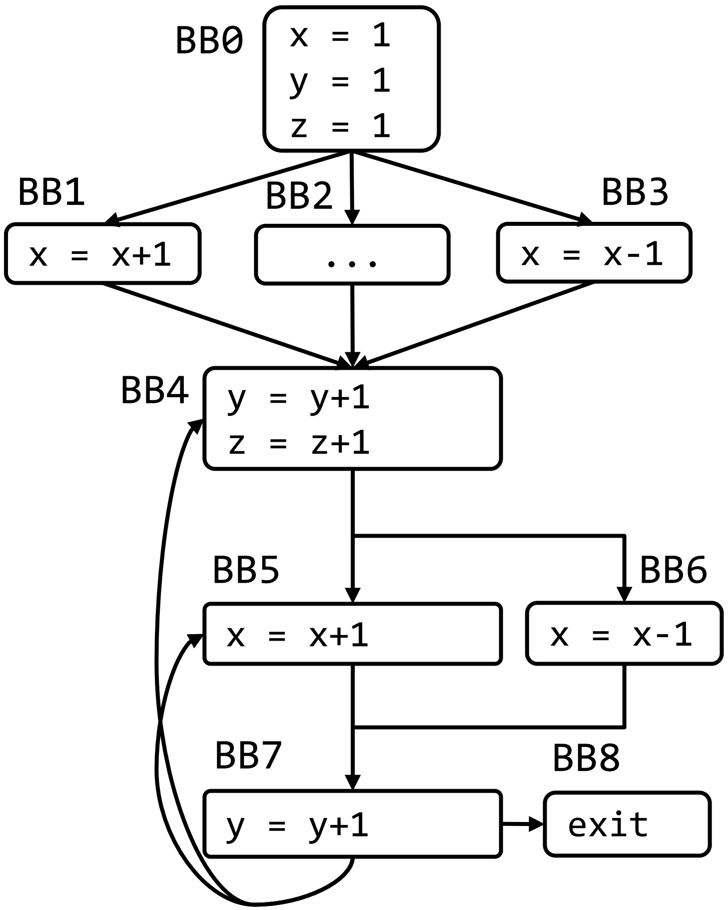
应用：第1轮



<x, n>：表示变量x在第n个节点被赋值

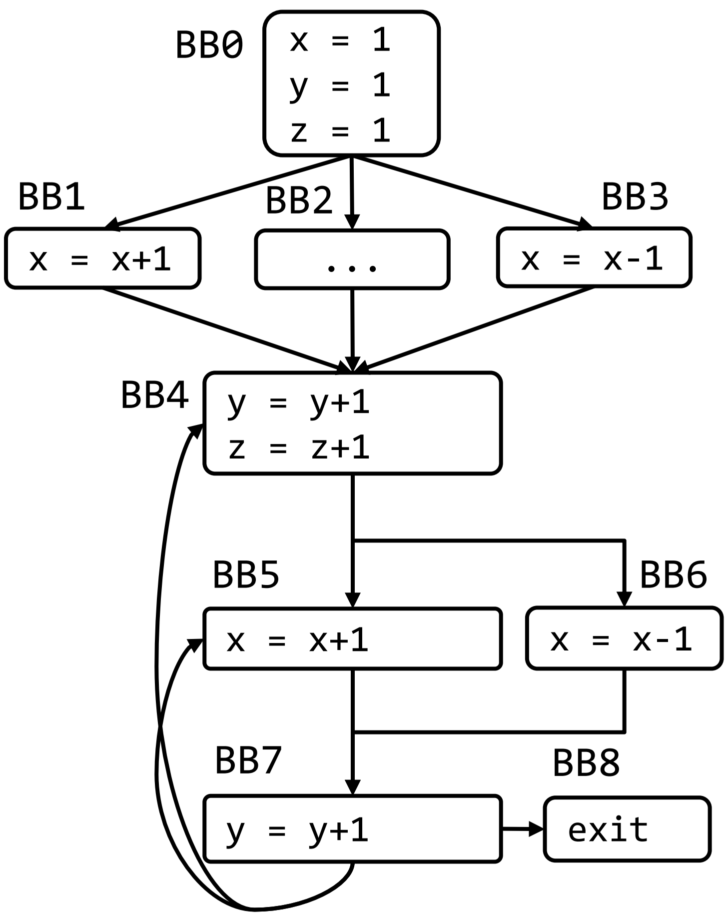
n	IN(n)	OUT(n)
BB0-1	-	{<x, 0-1>}
BB0-2	{<x, 0-1>}	{<x, 0-1><y,0-1>}
BB0-3	{<x,0-1><y,0-1>}	{<x, 0-1><y,0-1><z,0-3>}
BB1-1	{<x,0-1><y,0-1><z,0-3>}	{<x,1-1><y,0-1><z,0-3>}
BB4-1	{<x,1-1><y,0-1><z,0-3>}	{<x,1-1><y,4-1><z,0-3>}
BB4-2	{<x,1-1><y,4-1><z,0-3>}	{<x,1-1><y,4-1><z,4-2>}
BB5-1	{<x,1-1><y,4-1>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

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

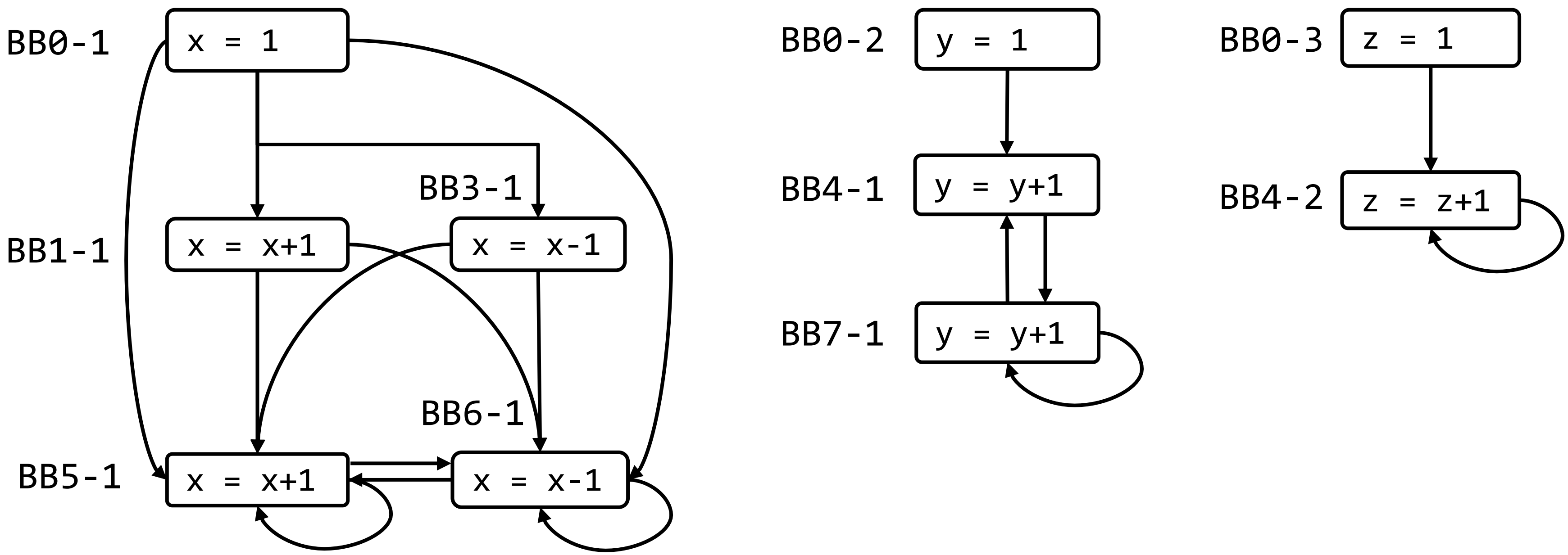


n	IN(n)	OUT(n)
BB4-1	{<x,0-1,1-1,3-1,5-1,6-1> <y,0-1,7-1> <z,0-3,4-2>}	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,0-3,4-2>}
BB4-2	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,0-3,4-2>}	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,4-2>}
BB5-1	{<x,0-1,1-1,3-1,5-1,6-1> <y,4-1> <z,4-2>} {<x,5-1,6-1> <y, 7-1> <z,4-2>}	{<x,5-1> <y,4-1,7-1>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

算法是否一定会终止？

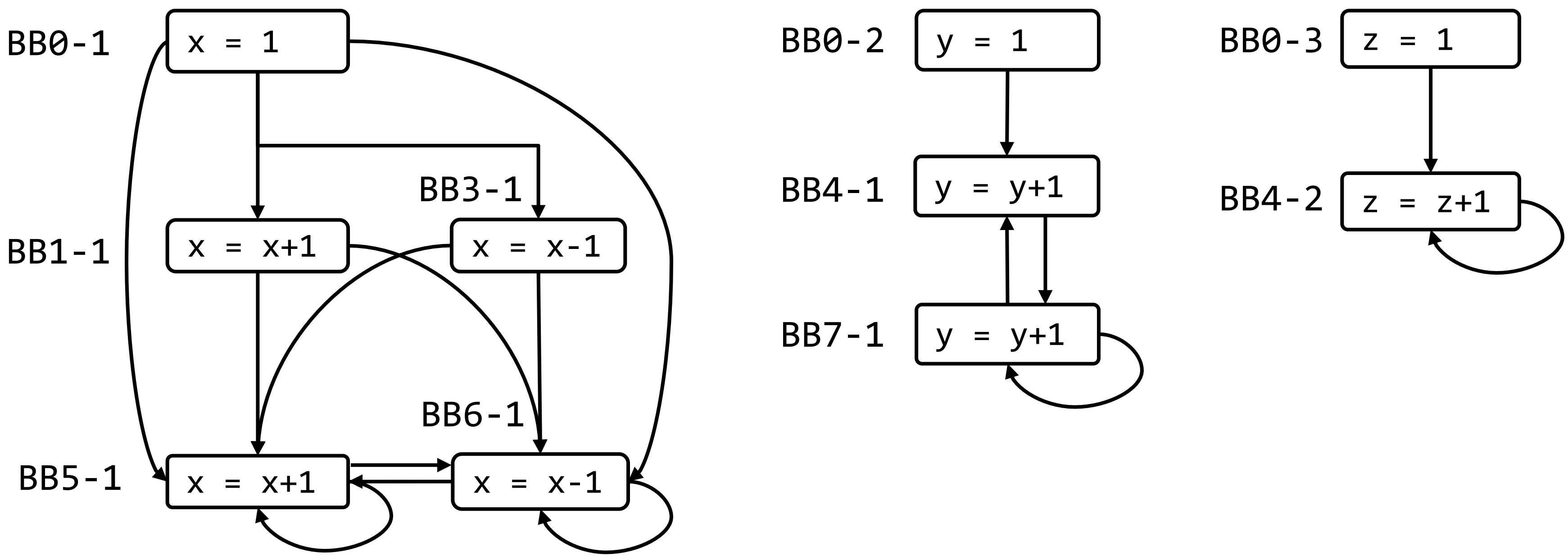
- 可达定义分析的迭代算法一定会终止
  - Join和Transfer的两个函数是单调的（monotonic）
    - IN和OUT集合元素数目只会增加，不会减少。
  - IN集合OUT不可能无限扩大，最大是程序中所有定义语句的集合。
  - IN和OUT一定会在某一轮迭代后停止改变。

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

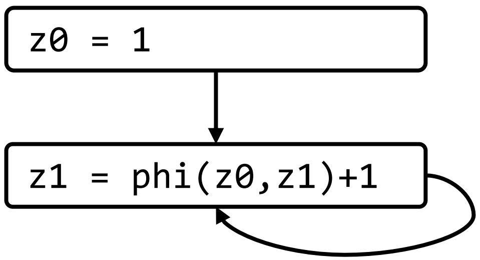
- 代码优化：
  - 代码移动（如公共表达式提取）
  - 延迟计算
  - ...
- 缺陷检测：
  - 为初始化的变量、指针
  - 除数是否可能为0

大纲

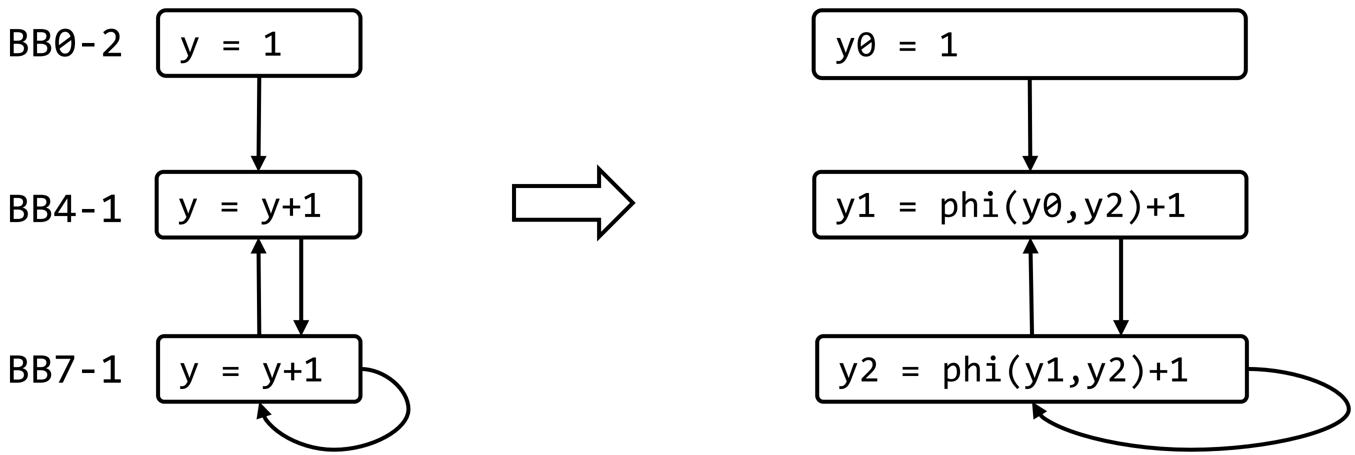
- 一、常量分析和优化
- 二、可达性分析和数值流图
- 三、静态单赋值形式

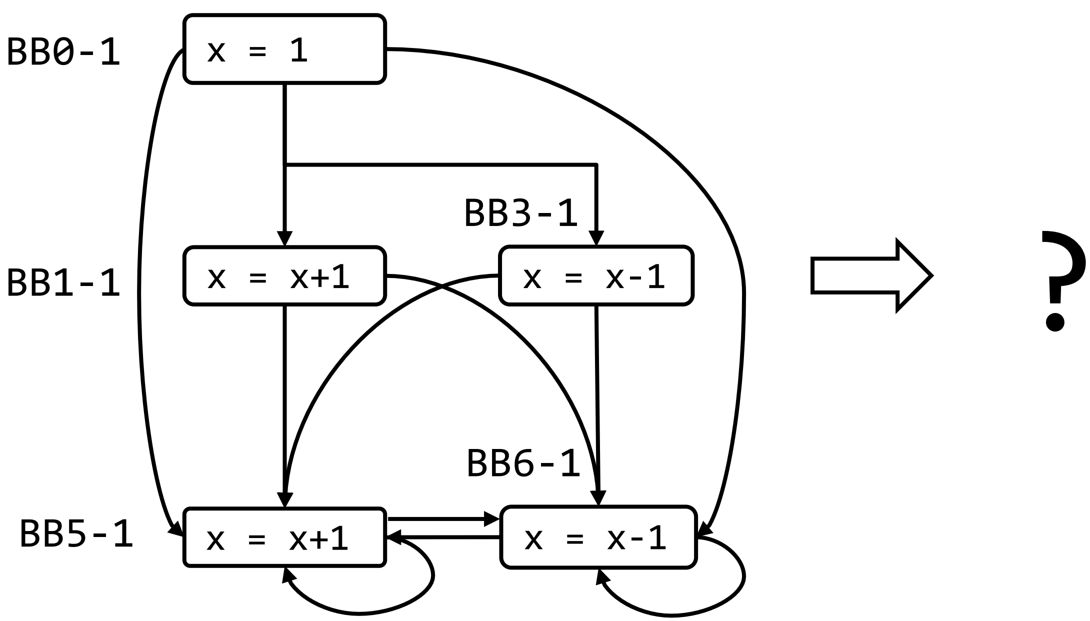
静态单赋值形式

- 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函数解决控制流带来的(def_1,def_2)-use问题
    - 如%3 = phi(%1,%2)
  - 分析数据流关系无需再考虑CFG



VFG=>单赋值形式

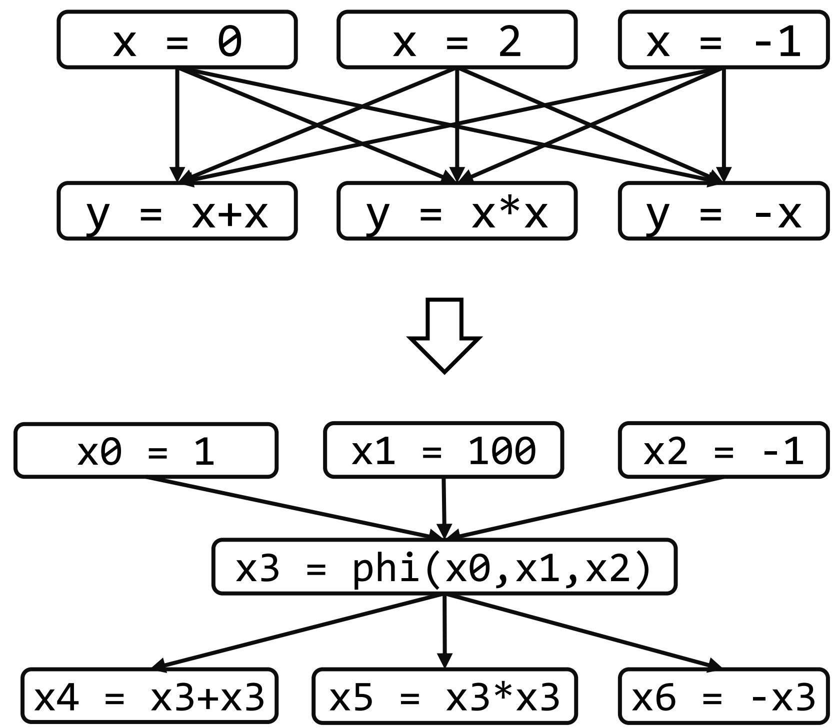

BB0-2
y = 1
BB4-1
y = y+1
BB7-1
y = y+1
y0 = 1
y1 = phi(y0,y2)+1
y2 = phi(y1,y2)+1


BB0-1
x = 1
BB1-1
x = x+1
BB3-1
x = x-1
BB6-1
BB5-1
x = x+1
x = x-1
?

SSA简化def-use关系

- 原始程序的def-use关系数量是$O(n^2)$；
- SSA的def-use数量减少为$O(n)$。

```
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函数

```
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函数放置思路:
  - 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

%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:

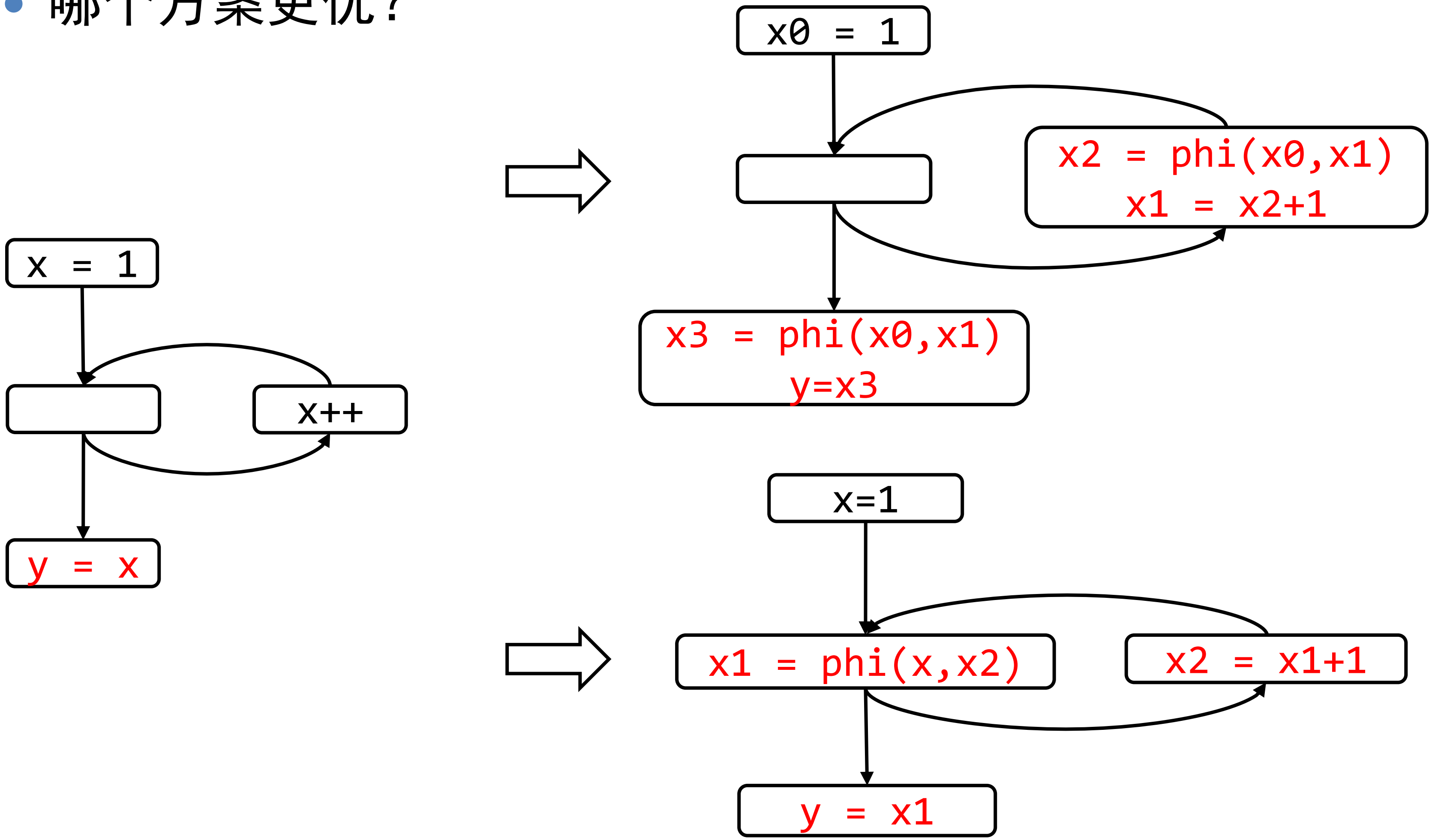
~~%3 = load i32, %a;~~
%4 = load i32, %b;
%5 = add i32 ~~%3~~ %1, %4;
store i32 %5, %a;
jmp %BB2;

%BB2:

~~%6 = load i32, %a;~~
%6 = phi(%1:%BB0,%5:%BB1)
~~%7 = load i32, %b;~~
%8 = add i32 %6, ~~%7~~ %4;
store i32 %8, %r;
%9 = load i32, %r;
ret %9;

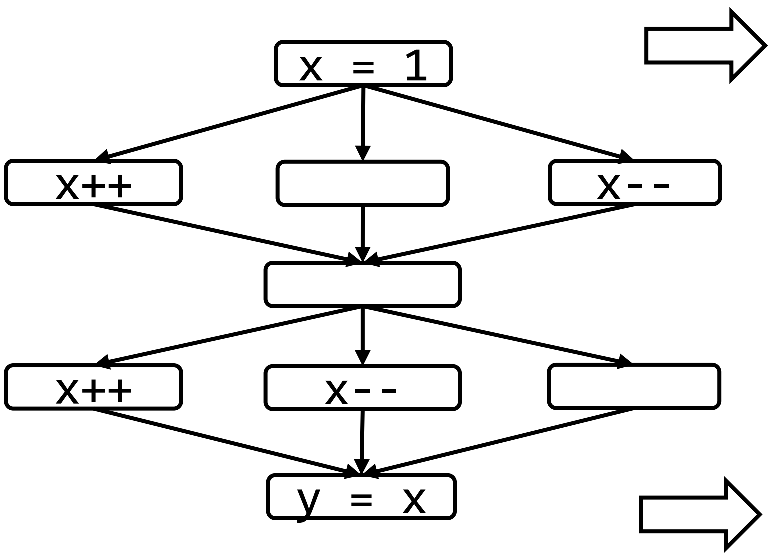
案例对比1

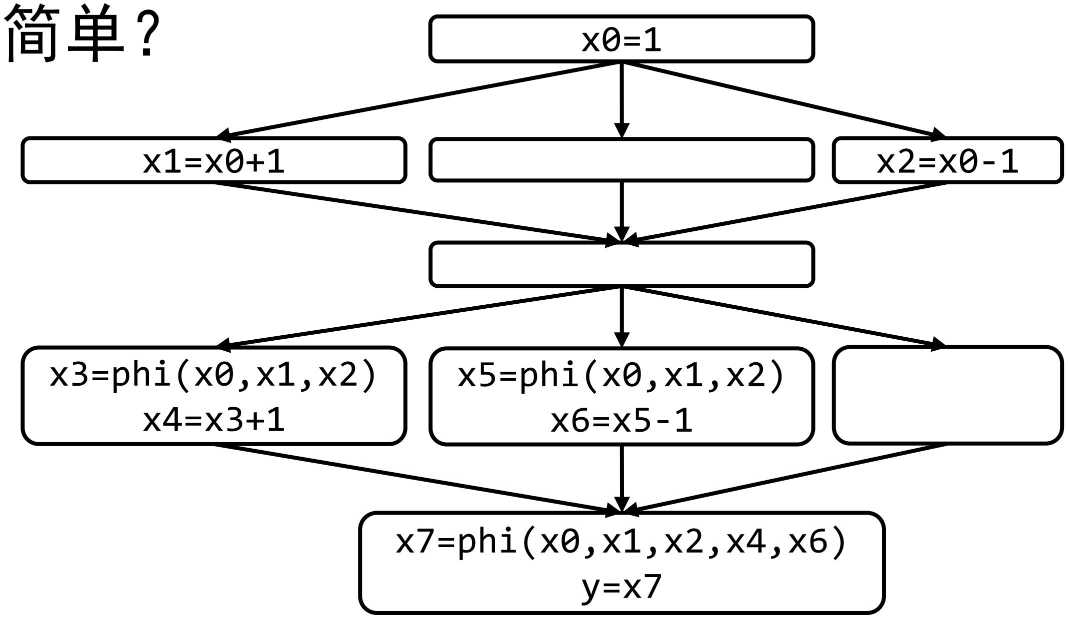
- 哪个方案更优？

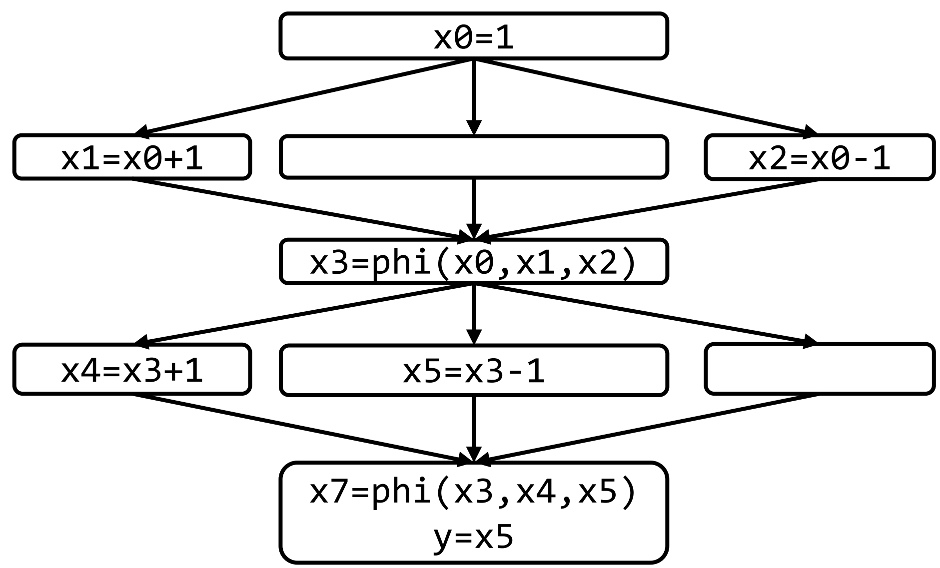


案例对比2

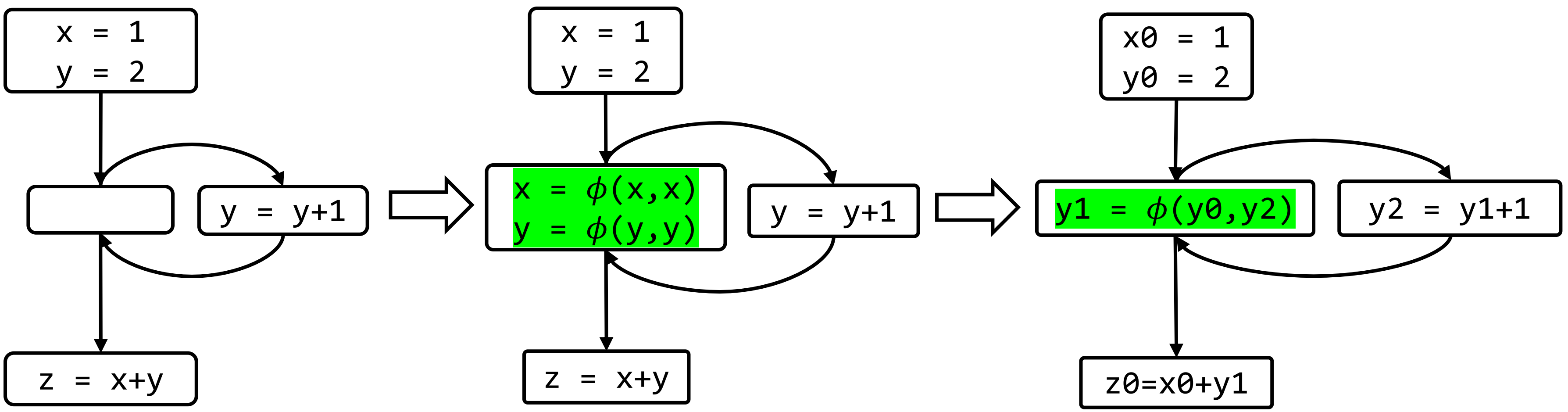
- 哪个方案def-use更简单？



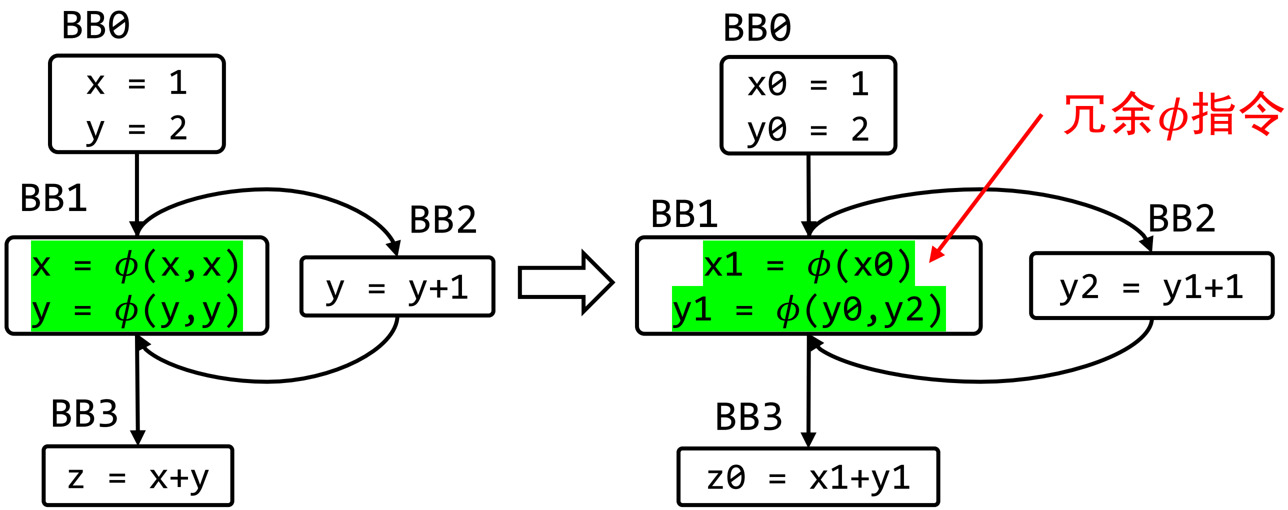




多个变量的情况

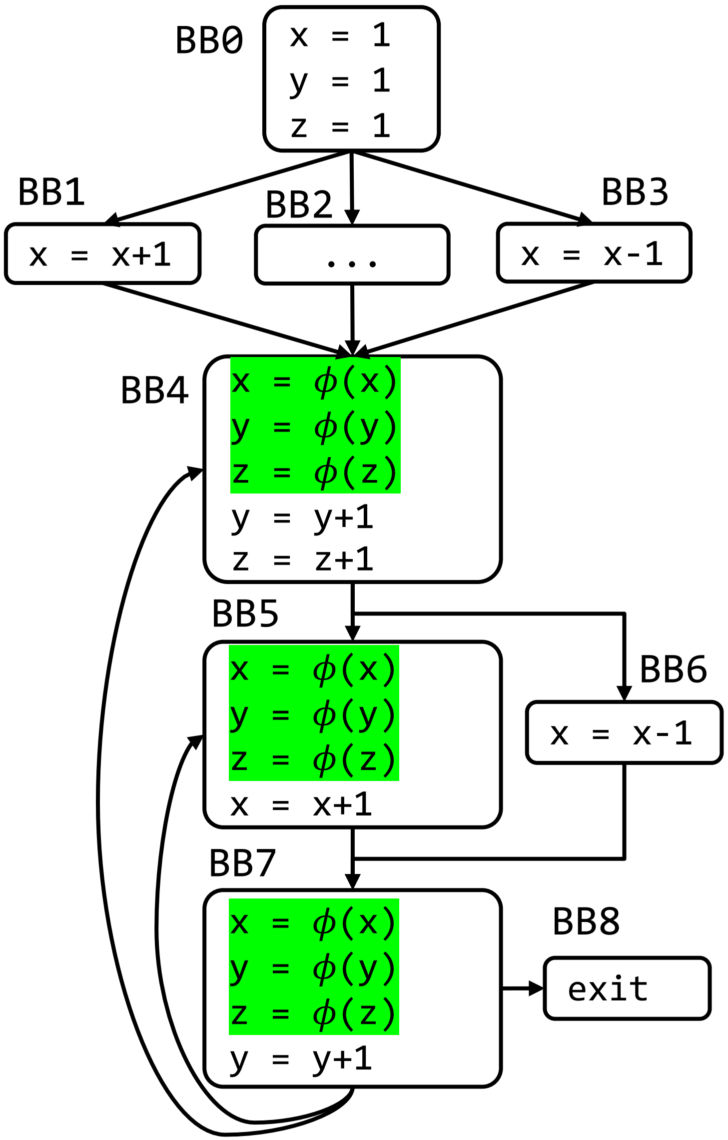


遍历控制流图构建SSA



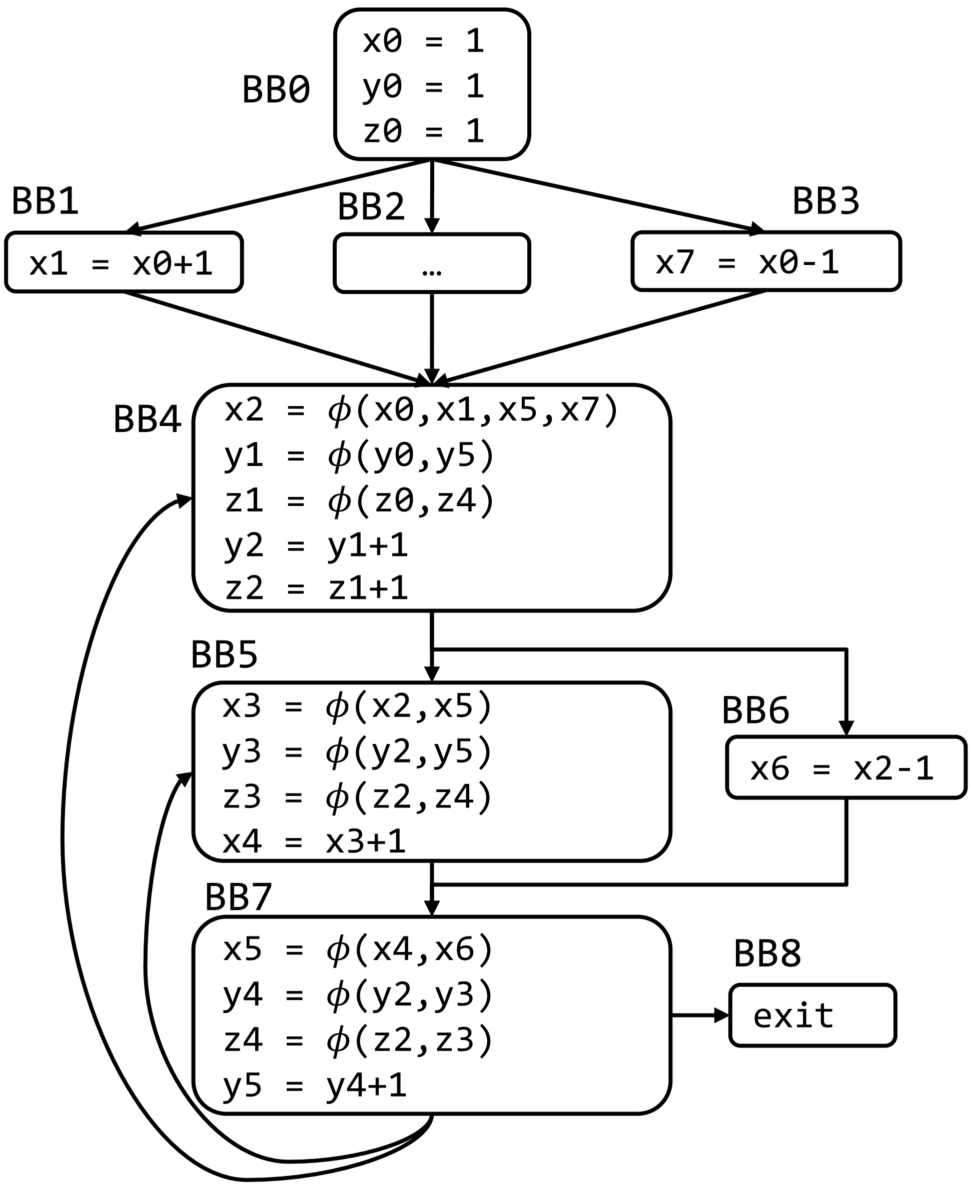
- DFS遍历控制流图构建SSA，顺序：BB0->BB1->BB2->BB1->BB3
  - BB0：x0=1，y0=100
  - BB1：x1=ϕ(x0)，y1=ϕ(y0)
  - BB2：y2=y1/2
  - BB1：x1=ϕ(x0)，y1=ϕ(y0,y2)
  - BB3：z0=x1+y1
- 开销：
  - 每个节点需要更新次数为其入度
  - $\forall bb_i \rightarrow bb_j \in CFG, \mathrm{Update}(bb_j)$

练习：遍历控制流图构建SSA



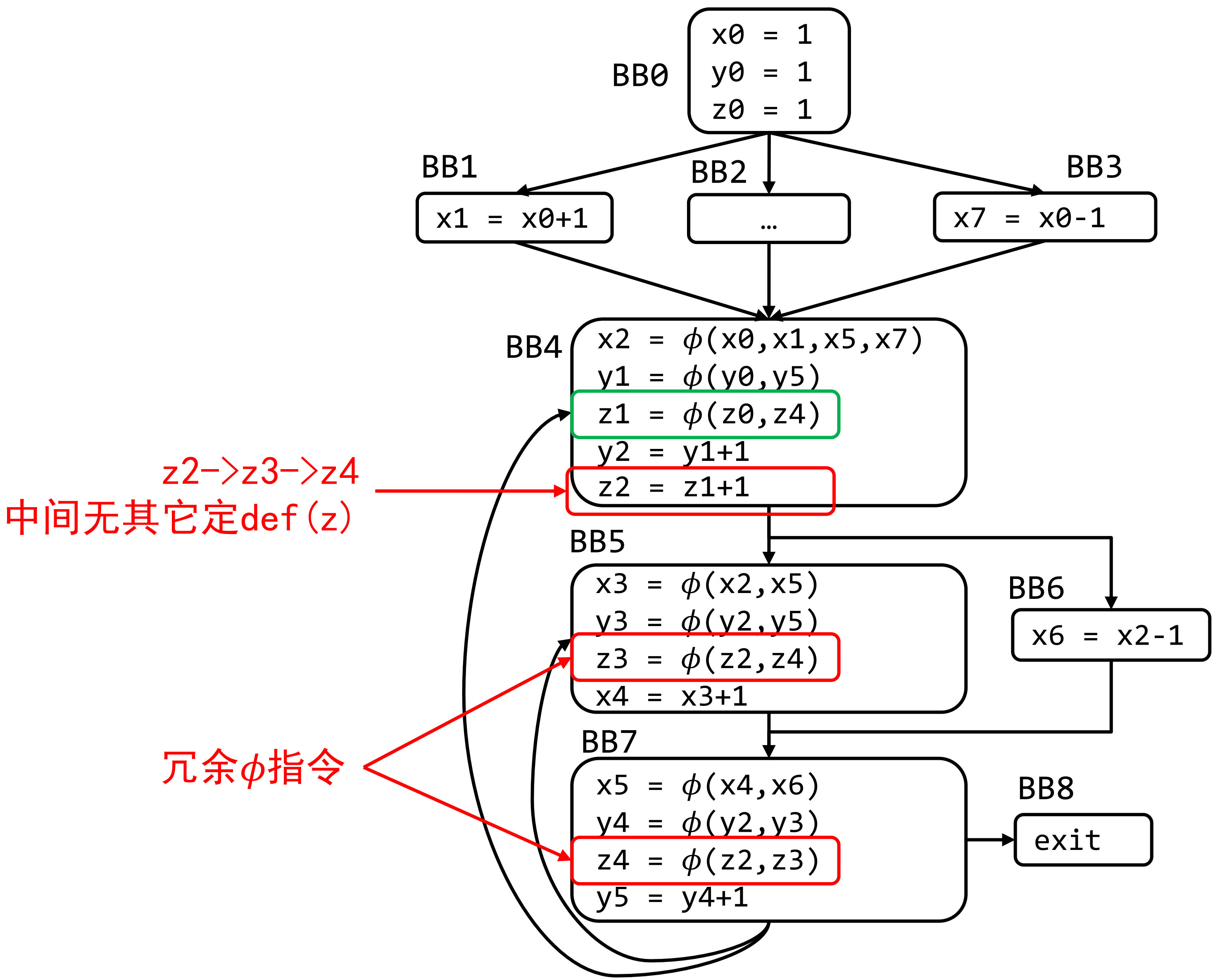
- DFS顺序：BB0->BB1->BB4->BB5->BB7->BB8->BB6->BB2->BB3
  - BB0：x0=1，y0=1，z0=1
  - BB1：...

结果



- DFS顺序：BB0->BB1->BB4->BB5->BB7->BB8->BB6->BB2->BB3
  - BB0: x0=1, y0=1, z0=1
  - BB1: x1=x0+1
  - BB4: x2=ϕ(x1), y1=ϕ(y0), z1=ϕ(z0), y2=y1+1, z2=z1+1
  - BB5: x3=ϕ(x2), y3=ϕ(y2), z3=ϕ(z2), x4=x3+1
  - BB7: x5=ϕ(x4), y4=ϕ(y3), z4=ϕ(z3), y5=y4+1
  - BB8:
  - BB4:x2=ϕ(x1,x5), y1=ϕ(y0,y5), z1=ϕ(z0,z4)
  - BB5:x3=ϕ(x2,x5), y3=ϕ(y2,y5), z3=ϕ(z2,z4),
  - BB6: x6 = x2-1
  - BB7: x5=ϕ(x4,x6), y4=ϕ(y2,y3), z4=ϕ(z2,z3)
  - BB2:
  - BB4: x2=ϕ(x0,x1,x5)
  - BB3: x7=x0-1
  - BB4: x2 = ϕ(x0,x1,x5,x7)

冗余Phi指令



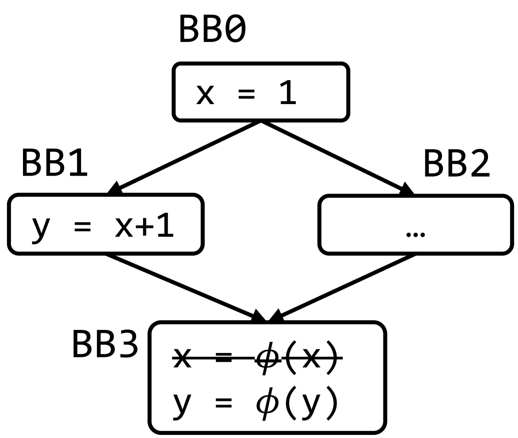
BB0
x0 = 1
y0 = 1
z0 = 1
BB1
x1 = x0+1
BB2
…
BB3
x7 = x0-1
BB4
x2 = ϕ(x0,x1,x5,x7)
y1 = ϕ(y0,y5)
z1 = ϕ(z0,z4)
y2 = y1+1
z2 = z1+1
z2->z3->z4
中间无其它定def(z)
BB5
x3 = ϕ(x2,x5)
y3 = ϕ(y2,y5)
z3 = ϕ(z2,z4)
x4 = x3+1
BB6
x6 = x2-1
BB7
x5 = ϕ(x4,x6)
y4 = ϕ(y2,y3)
z4 = ϕ(z2,z3)
y5 = y4+1
冗余ϕ指令
BB8
exit

phi函数放置思路

- phi函数放置思路:
  - 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

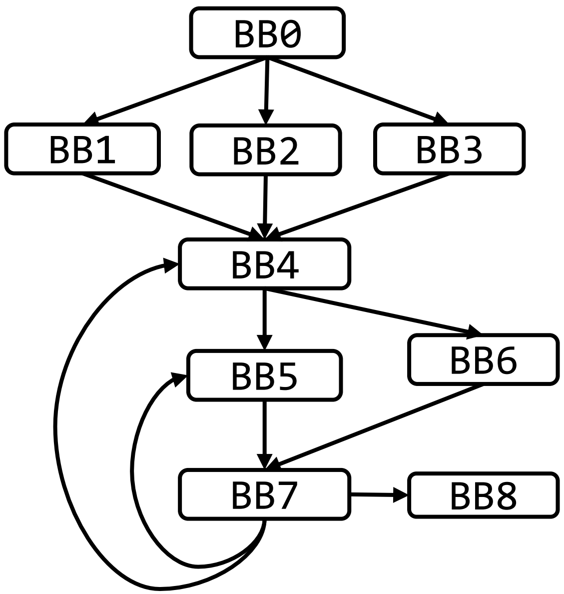
基于支配边界优化phi函数的设置

- 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中很可能需要phi(y)，
  - 有可能是false positive。



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。

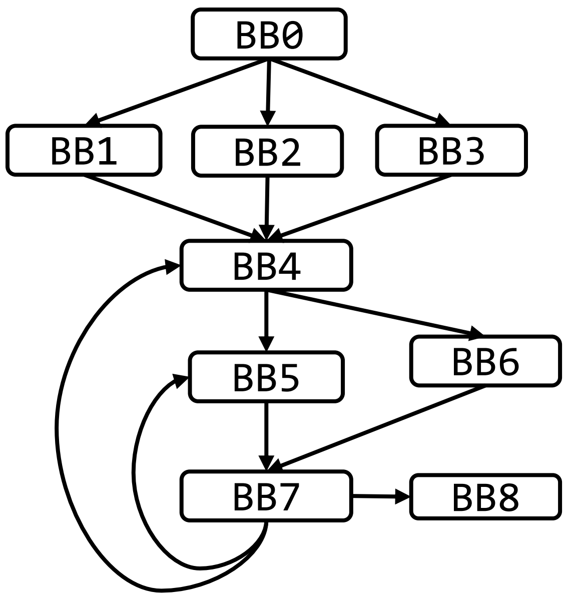


控制流图

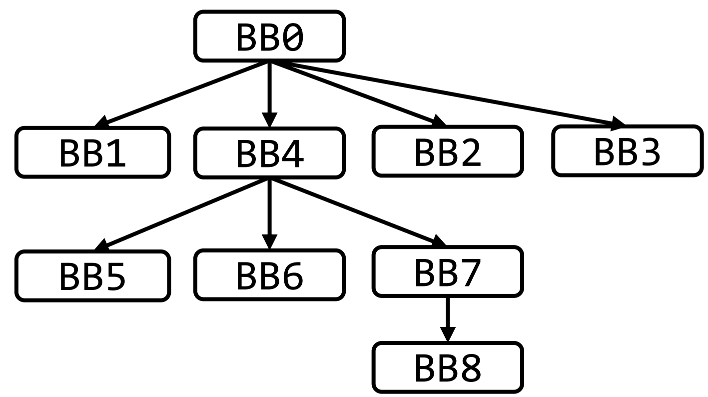
$$Dom(bb_0) = \{bb_0\}$$
$$Dom(bb_1) = \{bb_0, bb_1\}$$
$$Dom(bb_2) = \{bb_1, bb_2\}$$
$$Dom(bb_3) = \{bb_0, bb_3\}$$
$$Dom(bb_4) = \{bb_0, bb_4\}$$
$$Dom(bb_5) = \{bb_0, bb_4, bb_5\}$$
$$Dom(bb_6) = \{bb_0, bb_4, bb_6\}$$
$$Dom(bb_7) = \{bb_0, bb_4\}$$
$$Dom(bb_8) = \{bb_0, bb_7\}$$

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。



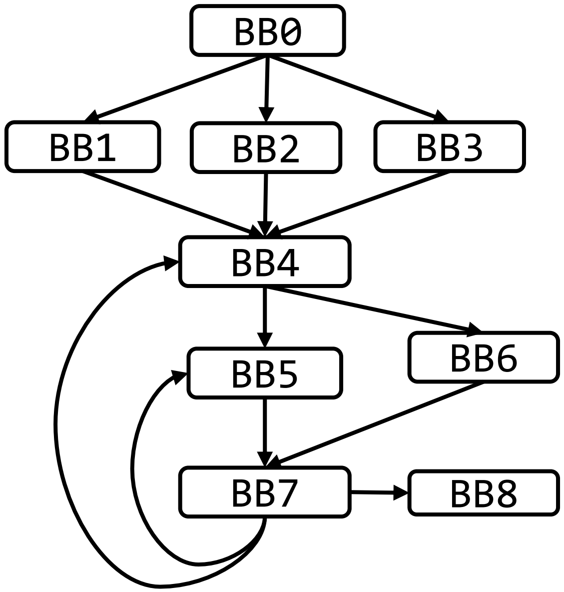
控制流图



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

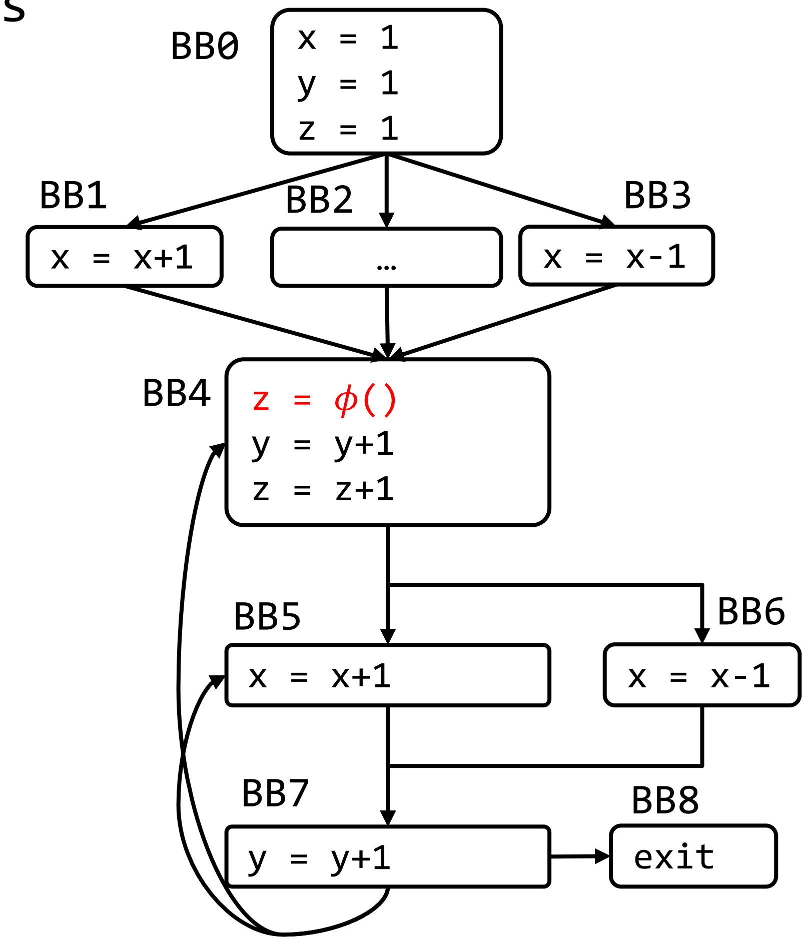


控制流图

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

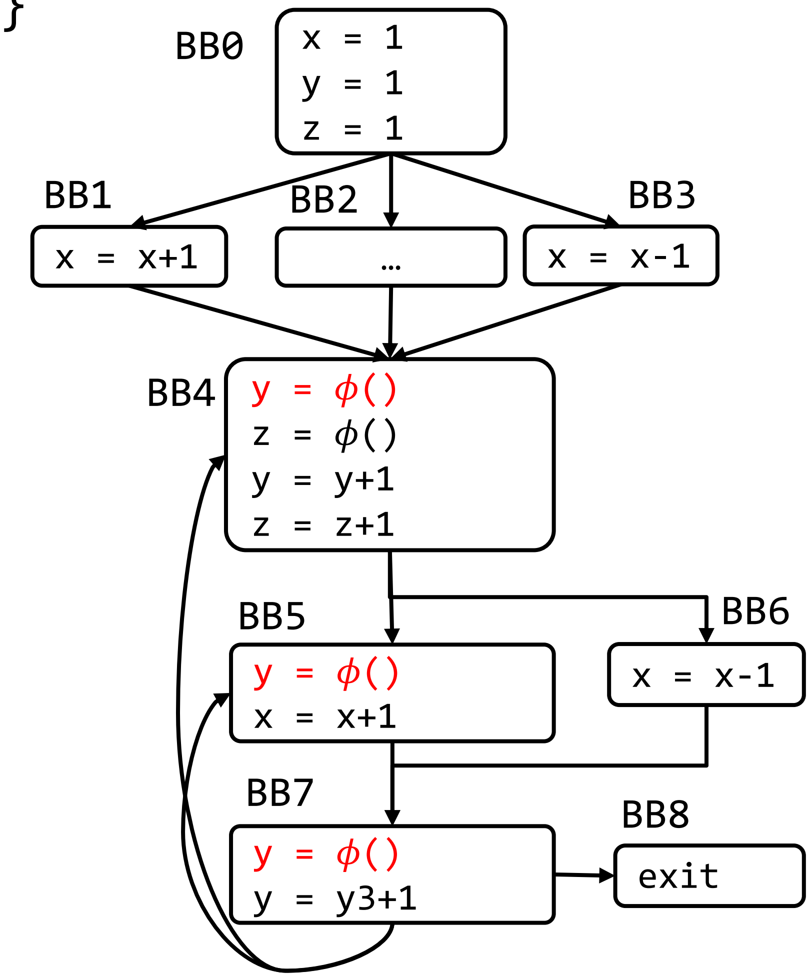
利用支配边界计算def

- 初始化：枚举所有变量的def-sites
  - def-sites(x) = {BB0,BB1,BB3,BB5,BB6}
  - def-sites(y) = {BB0,BB4,BB7}
  - def-sites(z) = {BB0,BB4}
- 为每个变量在BB_j增加phi节点：
  - $BB_i \in$ def-sites(x)
  - $BB_j \in DF(BB_i)$
- 以变量z为例：
  - def-sites(z) = {BB0,BB4}
    - $DF(BB_0) = \{\}$
    - $DF(BB_4) = \{BB_4\}$
    - 在BB_4增加phi函数的phi(z)



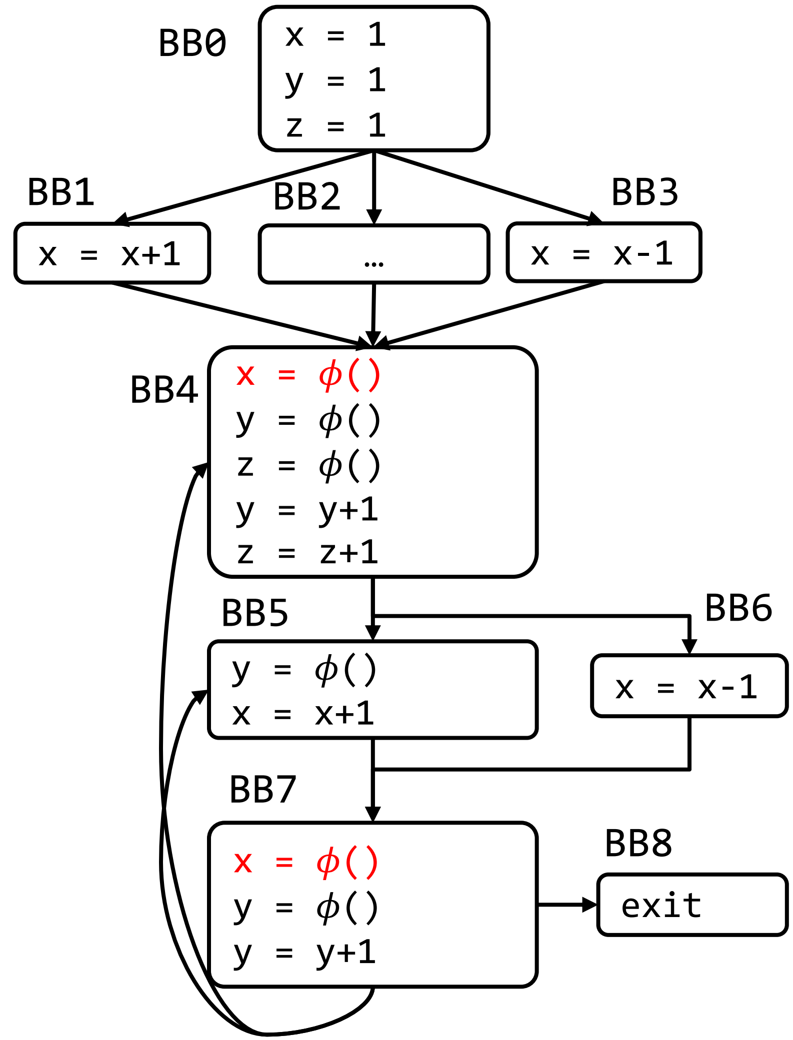
为变量y的插入phi指令

- def-sites(y) = {BB0,BB4,BB7}
  - $DF(BB_0)$ = {}
  - $DF(BB_4)$ = $\{BB_4\}$
    - 在BB_4增加phi函数的phi(y)
  - $DF(BB_7)$ = $\{BB_4, BB_5\}$
    - 在BB_5增加phi函数的phi(y)
    - 将BB5到def-sites(y)
  - $DF(BB_5)$ = $\{BB_7\}$
    - 在BB7增加phi函数的phi(y)

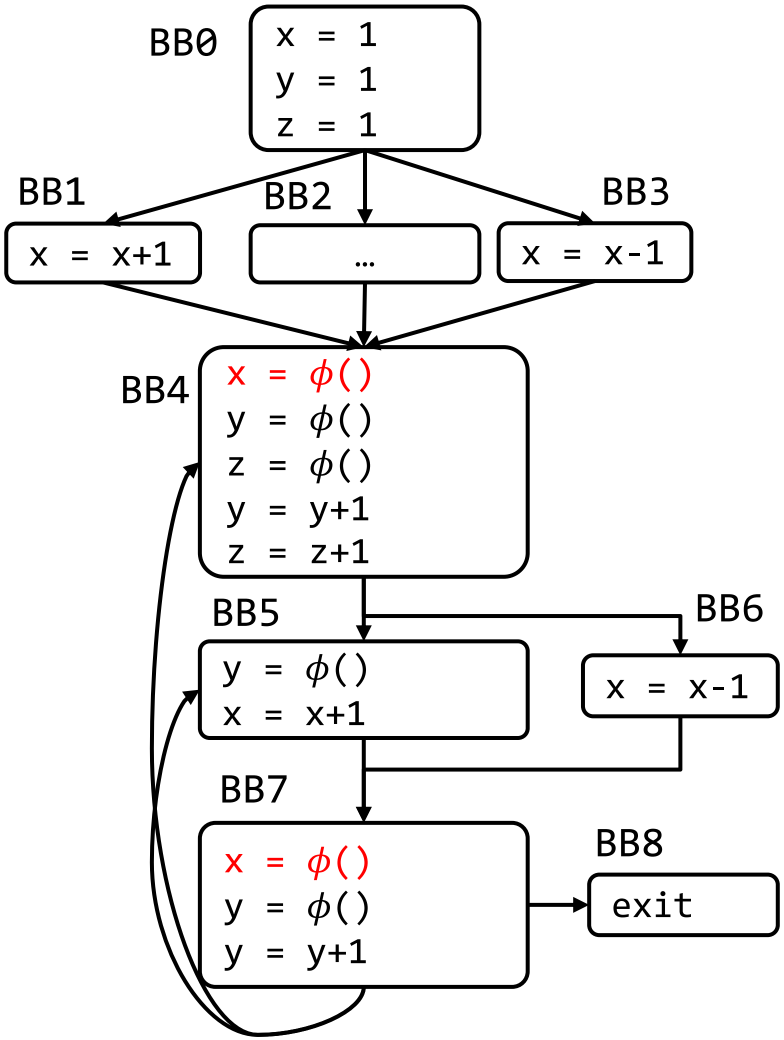


为变量x的插入phi指令

- def-sites(x) = {BB0,BB1,BB3,BB5,BB6}
  - $DF(BB_0)$ = {}
  - $DF(BB_4)$ = $\{BB_4\}$
    - 在BB_4增加phi函数的phi(x)
  - $DF(BB_3)$ = $\{BB_4\}$
  - $DF(BB_5)$ = $\{BB_7\}$
    - 在BB4增加phi函数的phi(x)
  - $DF(BB_6)$ = $\{BB_7\}$



遍历控制流图构建SSA

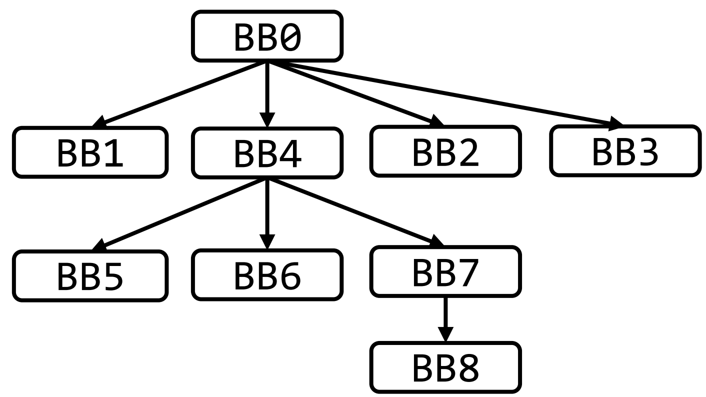


- DFS顺序：BB0->BB1->BB4->BB5->BB7->BB8->BB6->BB2->BB3
  - BB0: x0=1, y0=1, z0=1
  - BB1: ...

如何构建支配树：主要思路

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

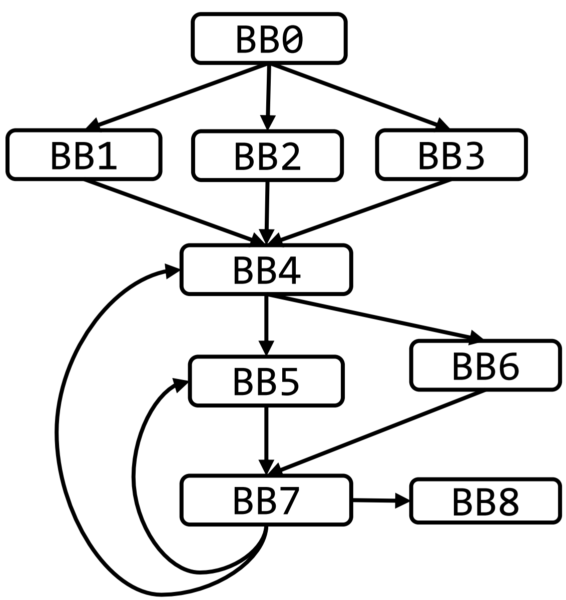
$Dom(bb_0) = \{bb_0\}$
$Dom(bb_1) = \{bb_0, bb_1\}$
$Dom(bb_2) = \{bb_1, bb_2\}$
$Dom(bb_3) = \{bb_0, bb_3\}$
$Dom(bb_4) = \{bb_0, bb_4\}$
$Dom(bb_5) = \{bb_0, bb_4, bb_5\}$
$Dom(bb_6) = \{bb_0, bb_4, bb_6\}$
$Dom(bb_7) = \{bb_0, bb_4\}$
$Dom(bb_8) = \{bb_0, bb_7\}$



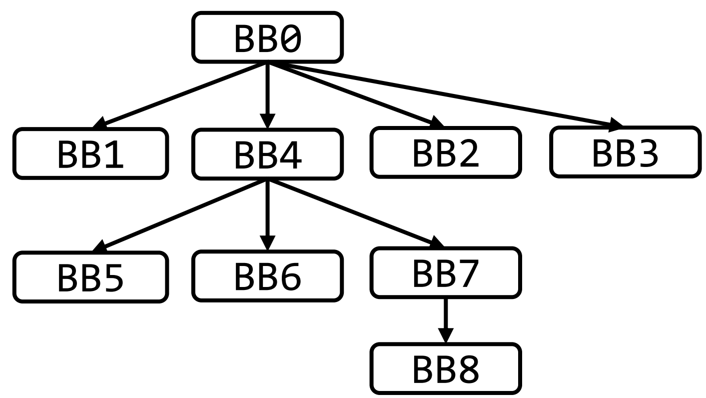
支配树

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$\bigcup_{v_p \in Pred(v)-IDom(v)} IDom(v_p)$
  - 迭代下去直到遇到v的直接支配节点$IDom(v)$



控制流图



支配树

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

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;
}
```