



软件分析

静态单赋值和稀疏分析

熊英飞

北京大学

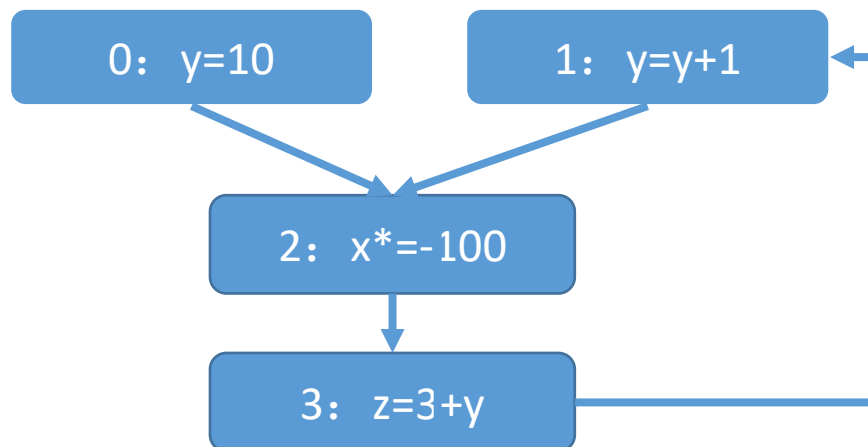


关于变量中保存值的分析

- 大量分析是关于变量中保存了什么值的
 - 符号分析
 - 区间分析
 - 常量传播



数据流分析的问题



- 问题1：每个结点都要保存一份关于 x, y, z 的值
 - 即使结点2和 y 没有关系
- 问题2：当1的转换函数更新 y 的时候，该更新只和3有关，但我们不可避免的要通过2才能到达3



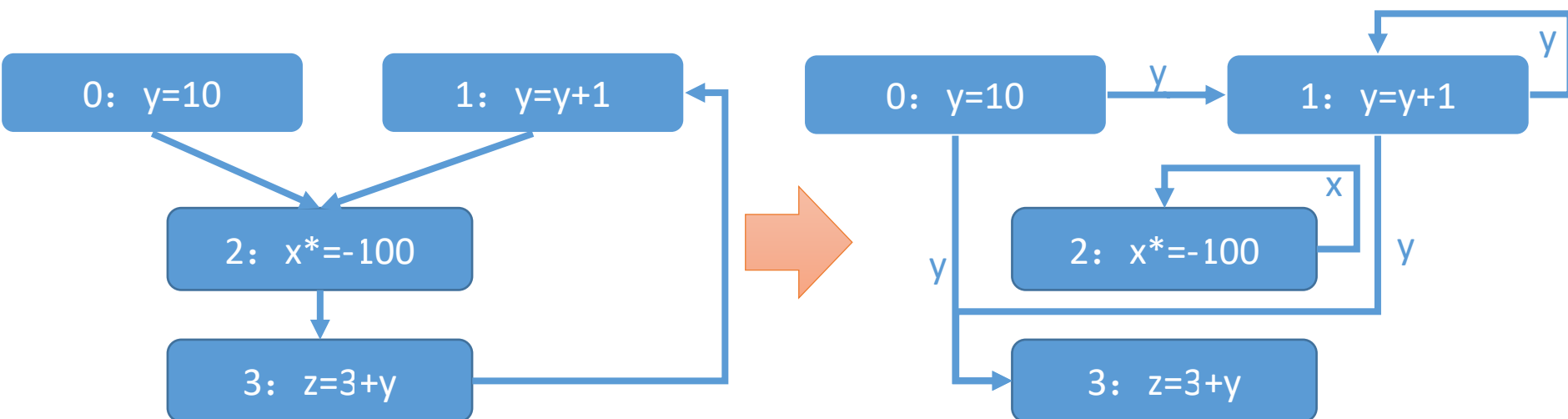
定义-使用关系

Def-Use关系

- 给定变量 x ，如果结点A可能改变 x 的值，结点B可能使用结点A改变后的 x 的值，则结点A和结点B存在定义-使用关系
 - 注意区分声明和定义
 - 声明：说明程序中有某个变量和变量的类型
 - 定义：给变量赋值



稀疏分析



- 每个结点只保存自己定义的变量的抽象值
- 只沿着定义-使用边传递抽象值
- 通常图上的边数大幅减少，图变得稀疏 (sparse)
- 分析速度大大高于原始数据流分析

$$\begin{aligned}y_0 &= f_0() \\y_1 &= f_1(y_0 \sqcup y_1) \\x_2 &= f_2(x_2 \sqcup x_0) \\z_3 &= f_3(y_0 \sqcup y_1)\end{aligned}$$



相关性质

- 假设结果基于集合的May分析，即返回的总是真实结果的超集
- 健壮性Soundness: 用原数据流算法求出来的每一个结果新算法都会求出来
- 准确性Precision: 用新算法求出来的每一个结果原算法都会求出来



获取定义-使用关系： 问题1

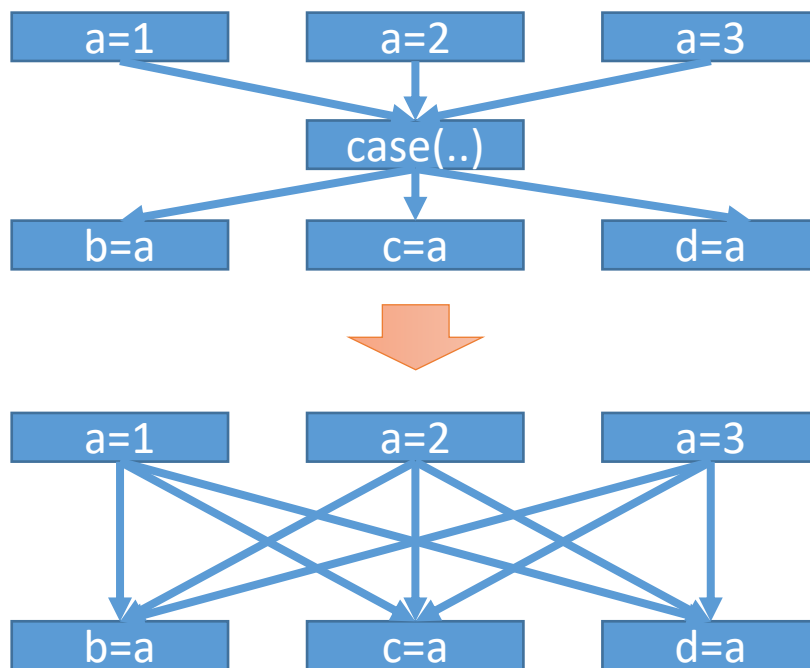
- 如何获取定义-使用关系
 - 可以通过Reaching Definition获取定义-使用关系
- 如何还原原始数据流分析的结果
 - 通过Reaching Definition获取使用变量以外的其他变量的定义
- Reaching Definition的复杂度
 - 程序中赋值语句个数为 m ，控制流图结点为 n
 - 更新单个节点的时间为 $O(m)$ （假设并集和差集的时间复杂度都是 $O(m)$ ）
 - 总共需要更新 $O(mn)$ 次
 - 总时间 $O(nm^2)$
- Reaching Definition本身就不够快，且额外实现比较麻烦



获取定义-使用关系： 问题2

- 在极端情况下，如果可能的定义较多，程序中的边会大幅增长，分析速度反而变慢

```
case (...) of
  0: a := 1;
  1: a := 2;
  2: a := 3;
end
case (...) of
  0: b := a;
  1: c := a;
  2: d := a;
end
```



a的值在三个位置被重复合并了三次

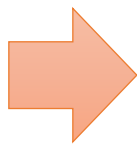


静态单赋值形式

Single Static Assignment

- 每个变量都只被赋值一次

```
x=10;  
y=y+1;  
x=y+x;  
y=y+1;  
z=y;
```



```
x0=10;  
y0=y+1;  
x1=y0+x0;  
y1=y0+1;  
z0=y1;
```

练习：把以下程序转成静态单赋值形式



`x=10;`

`x+=y;`

`if (x>10)`

`z=10;`

`else`

`z=20;`

`x+=z;`

`x0=10;`

`x1=x0+y;`

`if (x1>10)`

`z0=10;`

`else`

`z1=20;`

`x2=x1+z?;`



引入函数 ϕ

$x=10;$

$x+=y;$

if ($x>10$)

$z=10;$

else

$z=20;$

$x+=z;$

函数 ϕ 代表根据不同的控制流选择不同的值

$x0=10;$

$x1=x0+y;$

if ($x1>10$)

$z0=10;$

else

$z1=20;$

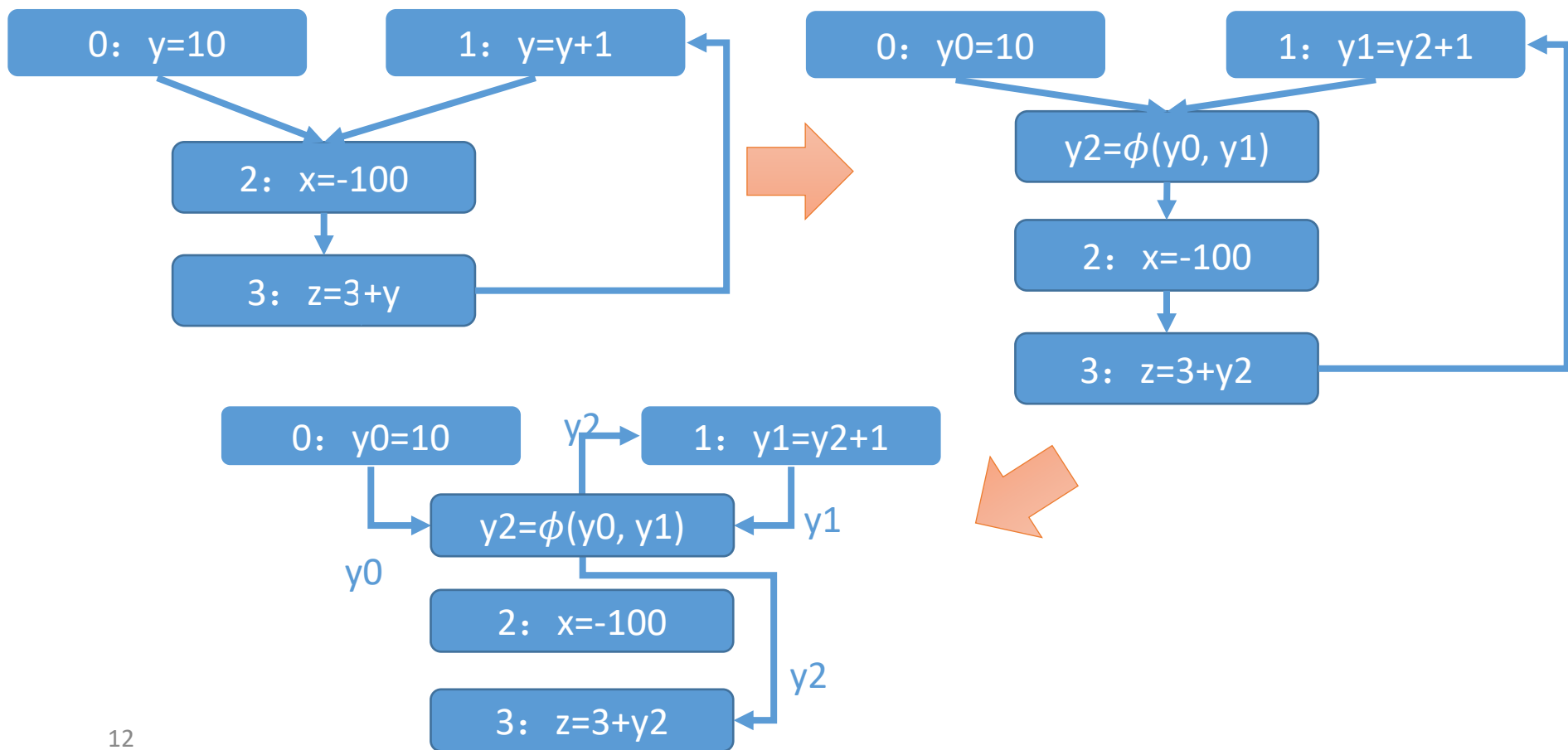
$z2=\phi(z0, z1);$

$x2=x1+z2;$



静态单赋值与数据流分析

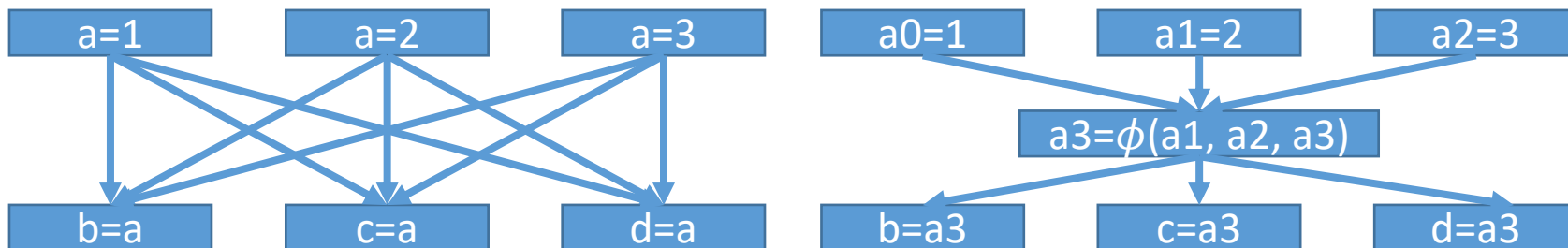
- 静态单赋值直接提供了定义-使用链





静态单赋值的好处

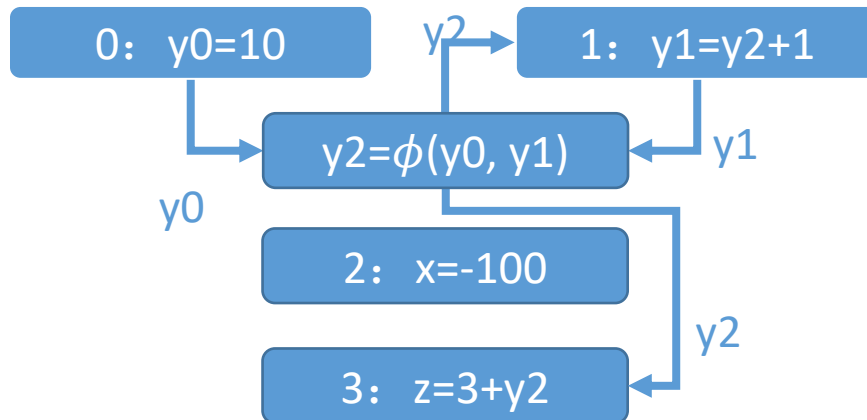
- 静态单赋值存在高效算法，且通常编译器框架都支持静态单赋值形式
- 静态单赋值中的边不会平方增长





静态单赋值vs流非敏感分析

- 静态单赋值形式上的流非敏感分析与流敏感分析结果等价
 - 所有变量只赋值一次，在程序任何位置变量的值都相等
 - 除了赋值位置，其他转换函数都是“传递值”

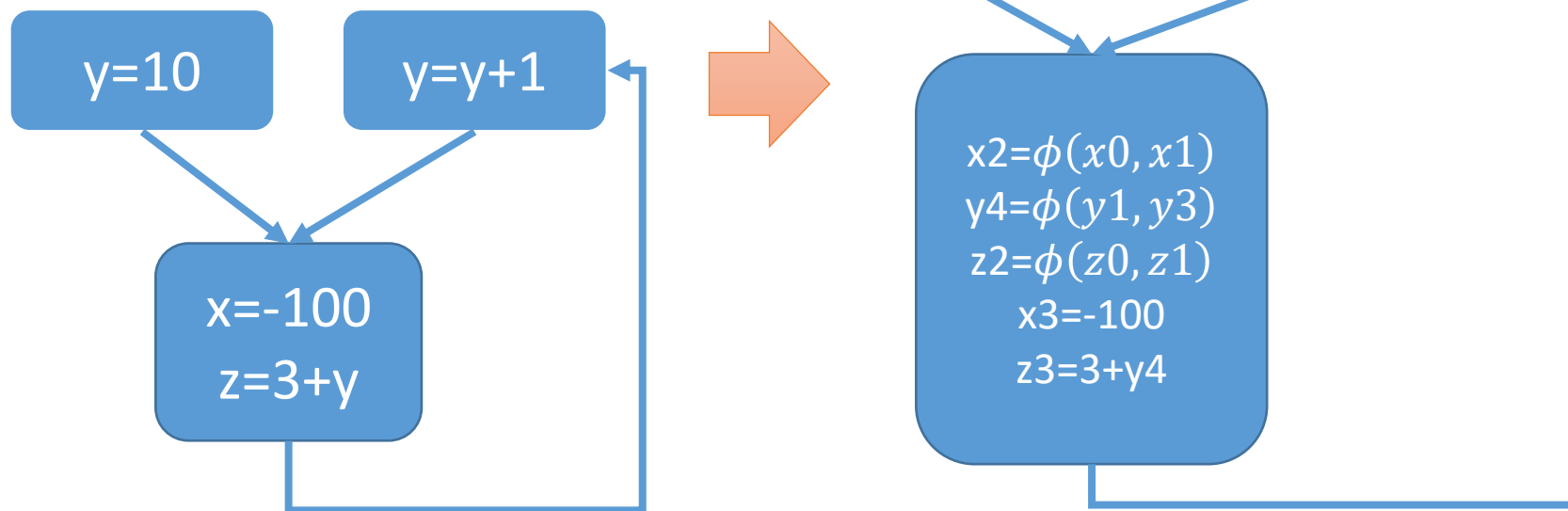


- 静态单赋值形式上的流非敏感分析与稀疏分析完全相同
 - 稀疏分析通常以静态单赋值上的流非敏感分析进行



转换到静态单赋值形式

- 简单算法
 - 每个基本块的头部对所有变量添加 ϕ 函数
 - 替换对应变量的值





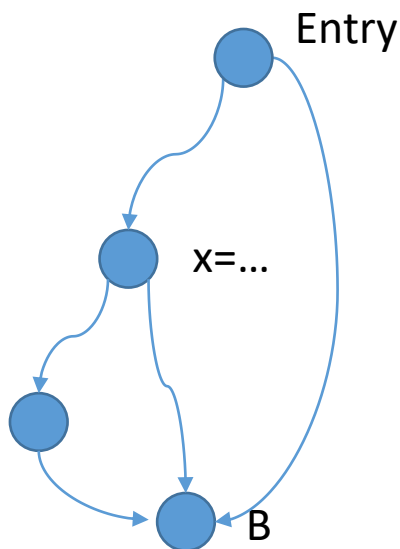
简单算法的问题

- 简单算法引入大量额外 ϕ 函数
 - 控制流图的每个结点会保存所有变量的值
 - 每条控制流图的边都会对每个变量产生定义-使用关系
 - 实际图并没有变得稀疏，反而可能更加稠密
- 希望能尽量减少引入的 ϕ 函数，即产生 ϕ 函数尽量少的静态单赋值形式



加入 ϕ 函数的条件

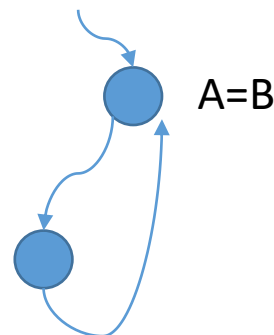
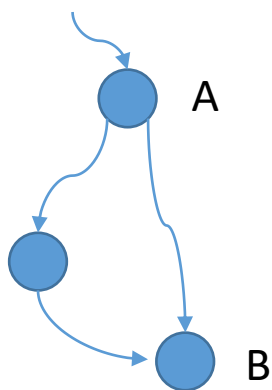
- 至少两条路径在B处汇合
- 其中一条经过了某个赋值语句
- 另外一条没有经过
- 赋值语句和B之间没有别的语句满足上述条件





支配关系

- 结点A支配 (dominate) 结点B: 所有从Entry到B的路径都要通过A

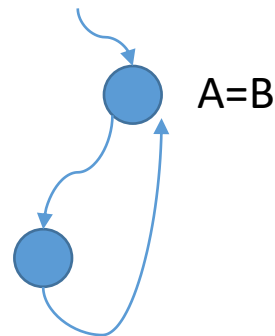
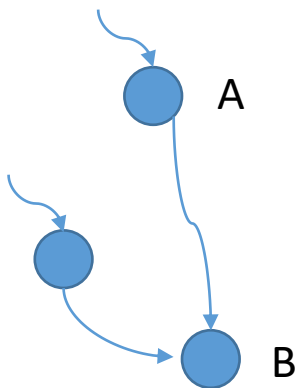


- 结点A严格支配 (Strictly dominate) 结点B: A支配B并且A和B不是一个结点
 - A不严格支配B \Rightarrow 至少存在一条路径, 在到达B之前不经过A



支配边界Dominance Frontier

- 支配边界：刚好丢失支配的边界节点
- 结点A的支配边界中包括B，当且仅当
 - A支配B的某一个前驱结点
 - A不严格支配B



- 对任意赋值语句 $x=...$ 所在的结点A，所有A的支配边界需要插入 ϕ 函数计算x的值

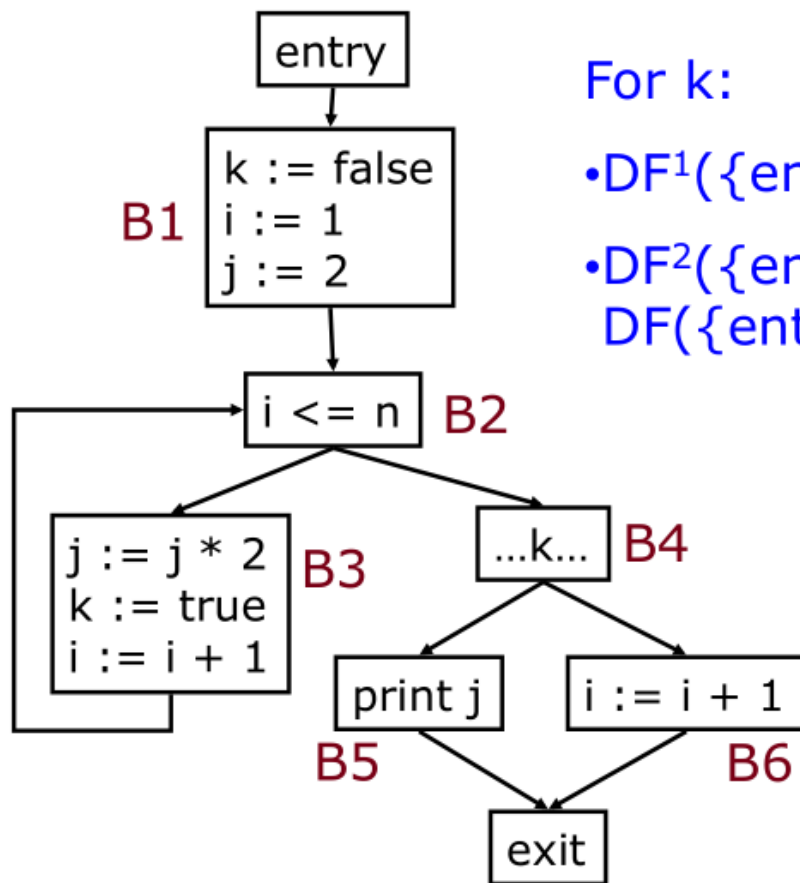


转换到静态单赋值形式

- 令 $DF(a)$ 为 a 的支配边界集合
- 定义
 - $DF(A) = \bigcup_{\{a \in A\}} DF(a)$
 - $DF^+(A) = \lim_{i \rightarrow \infty} DF^i(A)$
 - $DF^1(A) = DF(A)$
 - $DF^{i+1}(A) = DF(\bigcup_{j \leq i} DF^j(A))$
- 对任意变量 i , 令 A 为所有对 i 赋值的结点, $DF^+(A)$ 就是所有需要插入 ϕ 函数的结点



转换示例



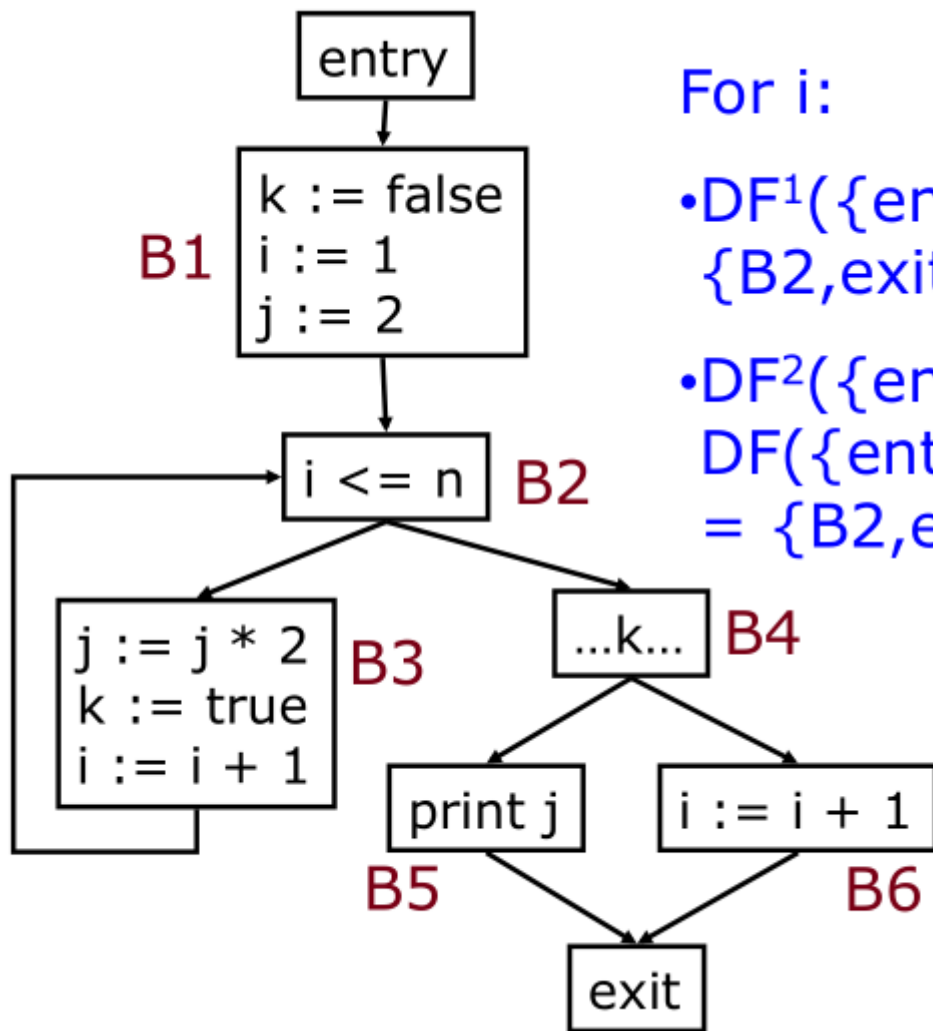
For k:

$$\bullet DF^1(\{\text{entry}, B1, B3\}) = \{B2\}$$

$$\bullet DF^2(\{\text{entry}, B1, B3\}) = DF(\{\text{entry}, B1, B2, B3\}) = \{B2\}$$



转换示例

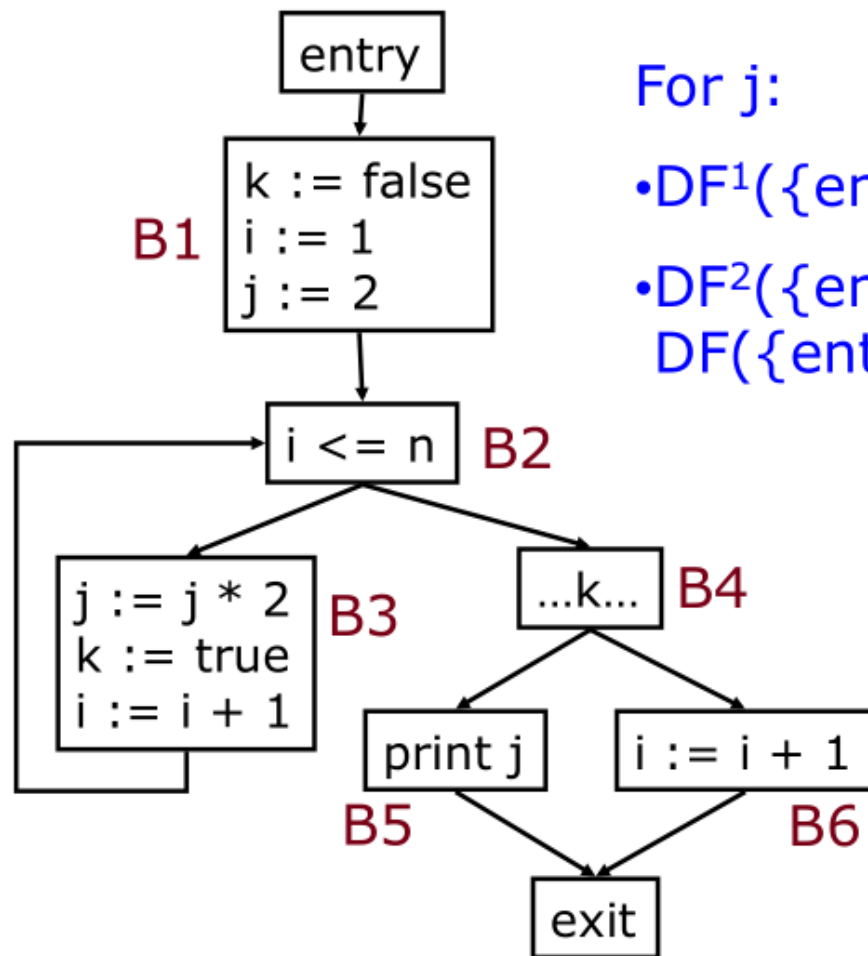


For i:

• $DF^1(\{\text{entry}, B1, B3, B6\}) = \{\text{B2}, \text{exit}\}$

• $DF^2(\{\text{entry}, B1, B3, B6\}) = DF(\{\text{entry}, B1, B2, B3, B6, \text{exit}\}) = \{\text{B2}, \text{exit}\}$

转换示例

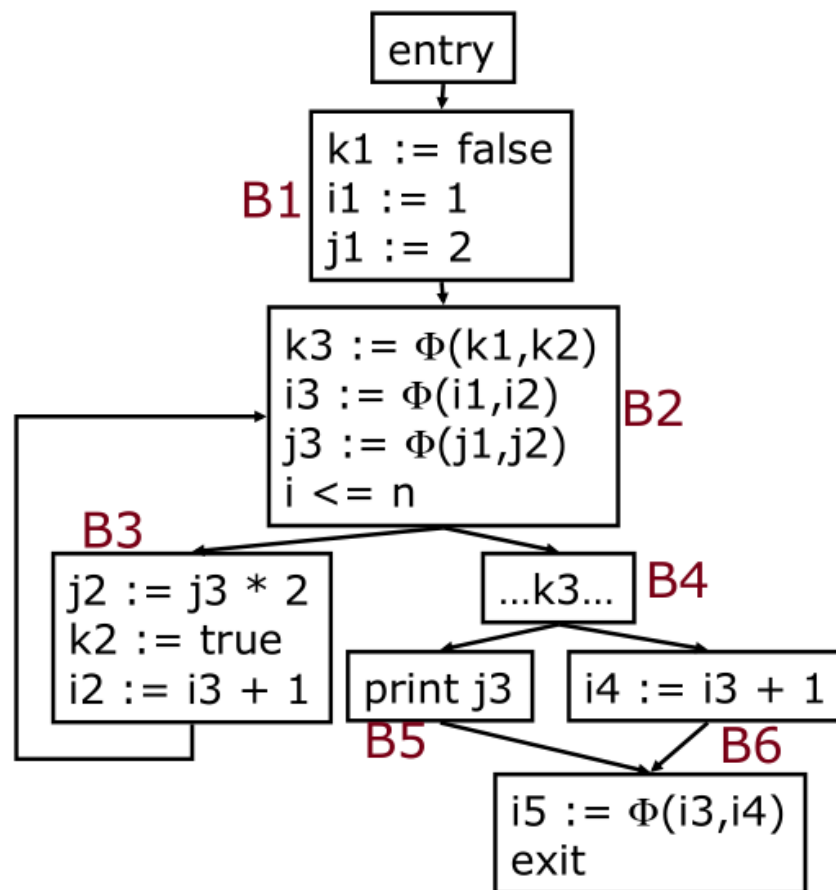


For j:

- $DF^1(\{\text{entry}, B1, B3\}) = \{B2\}$
- $DF^2(\{\text{entry}, B1, B3\}) =$
 $DF(\{\text{entry}, B1, B2, B3\}) = \{B2\}$



转换结果





计算支配边界的算法

- Lengauer and Tarjan算法
 - 复杂度为 $O(E\alpha(E,N))$
 - E 为边数, N 为结点数, α 为Ackerman函数的逆
 - Ackerman函数基本可以认为是常数
- Cooper, Harvey, Kennedy算法, 2001年
 - 复杂度为 $O(N^2)$
 - 在实践中更常见的小控制流图 (1000结点以下) 上在比Lengauer and Tarjan算法要快
 - 下面介绍CHK算法

Cooper, Harvey, Kennedy算法

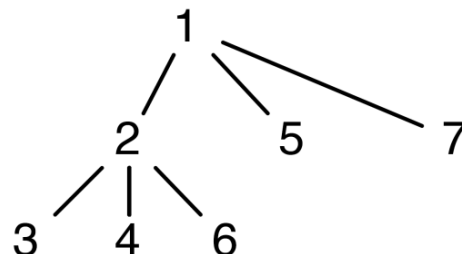
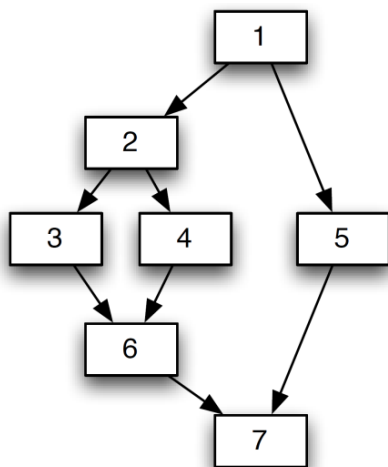


- 分为两步
 - 计算直接支配者
 - 直接支配关系使得很多中间结点可以被跳过，提高效率
 - 计算支配边界



直接支配者

- 直接支配者immediate dominator: 如果a严格支配b, 并且不存在c, a严格支配c且c严格支配b, 则a是b的直接支配者, 记为 $\text{idom}(b)$
- $\text{idom}(a)$ 是a的所有前驱结点在直接支配关系上的最近的公共祖先



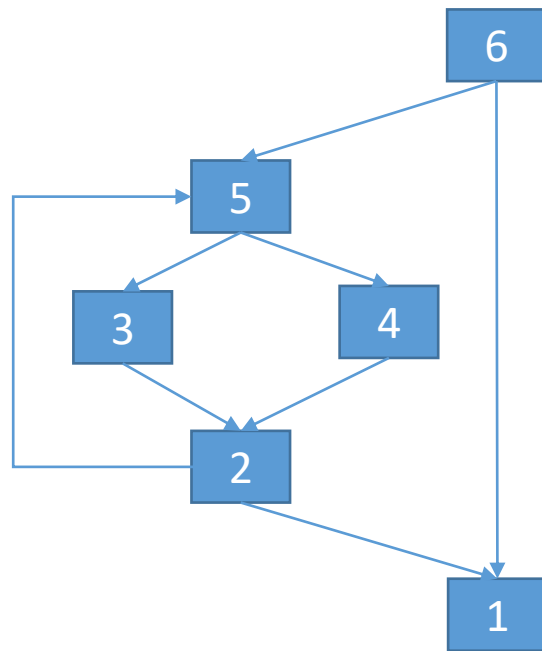


图的后序遍历

- 执行如下深度优先搜索算法，传给visit的结点序列即为后序遍历序列

```
dfs(n:node) {  
    for(s: 所有n的后继结点)  
        if (s没有作为参数传给dfs)  
            dfs(s);  
    visit(n)  
}
```

- 无环的时候，后继节点一定更小

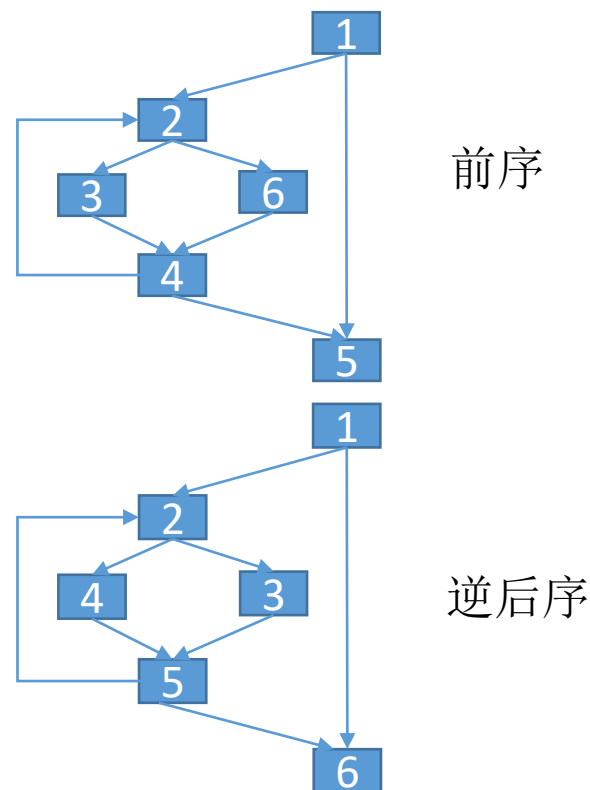




后序不是前序的逆

- 执行如下深度优先搜索算法，传给visit的结点序列即为前序遍历序列

```
dfs(n:node) {  
    visit(n)  
    for(s: 所有n的后继结点)  
        if (s没有作为参数传给dfs)  
            dfs(s);  
}
```





直接支配者计算算法

```
对所有结点n, idom(n)=null;
idom(entry)=entry;
do {
    逆后序遍历entry以外的结点b {
        idomb=任意idom不为空的前驱;
        对其他idom不为空的前驱p {
            idomb=公共祖先(p, idomb);
        }
        idom(b)=idomb;
    }
} while(idom有修改);
```

```
node 公共祖先(p1, p2) {
    while (p1 ≠ p2) {
        while (p1 < p2) p1 = idom(p1);
        while (p2 < p1) p2 = idom(p2);
    }
    return p1;
}
```

假设结点都按后序遍历编号



直接支配者计算例子

对所有结点 n , $idom(n)=null$;

$idom(entry)=entry$;

do {

逆后序遍历 $entry$ 以外的结点 b {

$idomb$ =任意 $idom$ 不为空的前驱;

对其他 $idom$ 不为空的前驱 p {

$idomb$ =公共祖先(p , $idomb$); }

$idom(b)=idomb$; }

} while($idom$ 有修改);

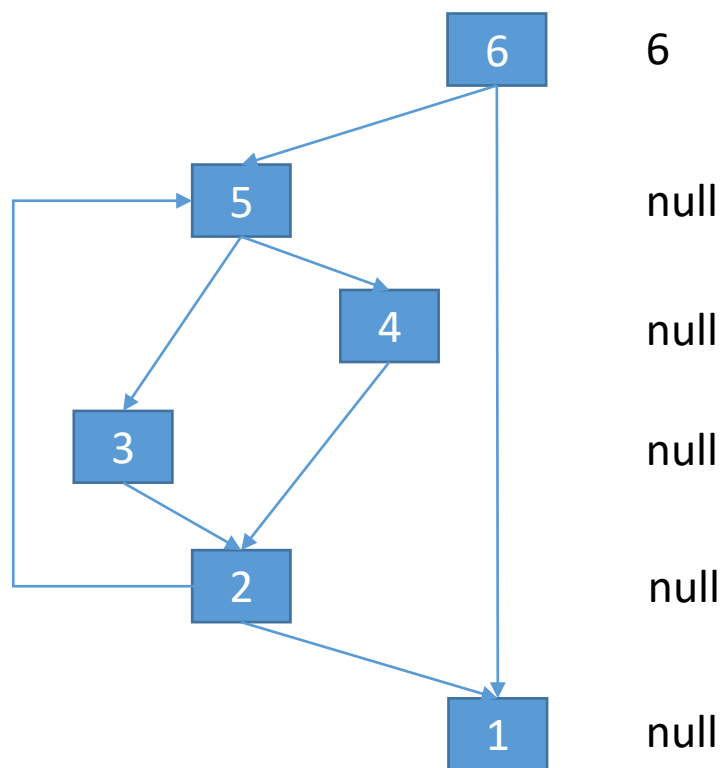
node 公共祖先($p1$, $p2$) {

while ($p1 \neq p2$) {

while ($p1 < p2$) $p1 = idom(p1)$;

while ($p2 < p1$) $p2 = idom(p2)$; }

return $p1$; }





直接支配者计算例子

对所有结点 n , $idom(n)=null$;

$idom(entry)=entry$;

do {

逆后序遍历 $entry$ 以外的结点 b {

$idomb$ =任意 $idom$ 不为空的前驱;

对其他 $idom$ 不为空的前驱 p {

$idomb$ =公共祖先($p, idomb$); }

$idom(b)=idomb$; }

} while($idom$ 有修改);

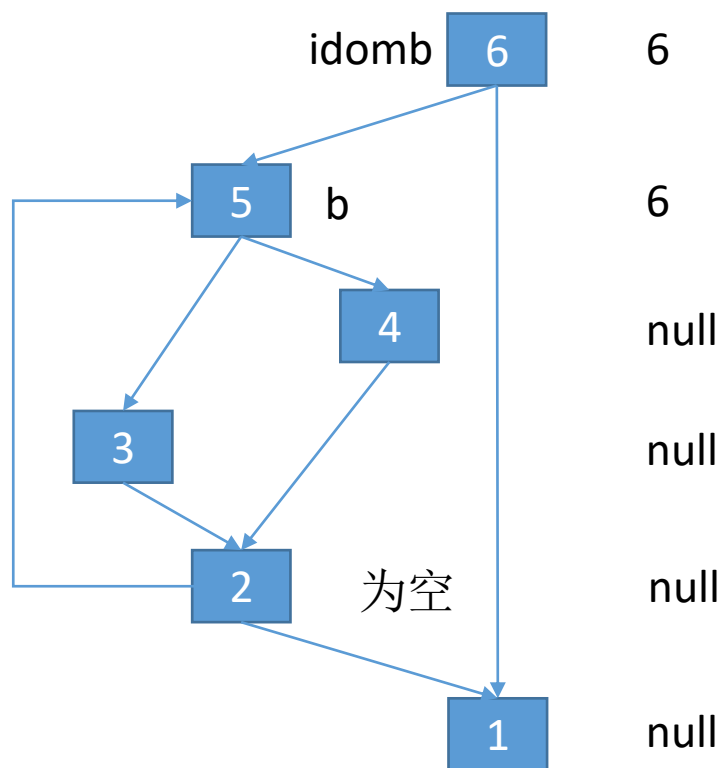
node 公共祖先($p1, p2$) {

while ($p1 \neq p2$) {

while ($p1 < p2$) $p1 = idom(p1)$;

while ($p2 < p1$) $p2 = idom(p2)$; }

return $p1$; }





直接支配者计算例子

对所有结点n, $\text{idom}(n) = \text{null}$;

$\text{idom}(\text{entry}) = \text{entry}$;

do {

逆后序遍历entry以外的结点b {

$\text{idomb} = \text{任意idom不为空的前驱}$;

对其他idom不为空的前驱p {

$\text{idomb} = \text{公共祖先}(p, \text{idomb})$; }

$\text{idom}(b) = \text{idomb}$; }

} while(idom有修改);

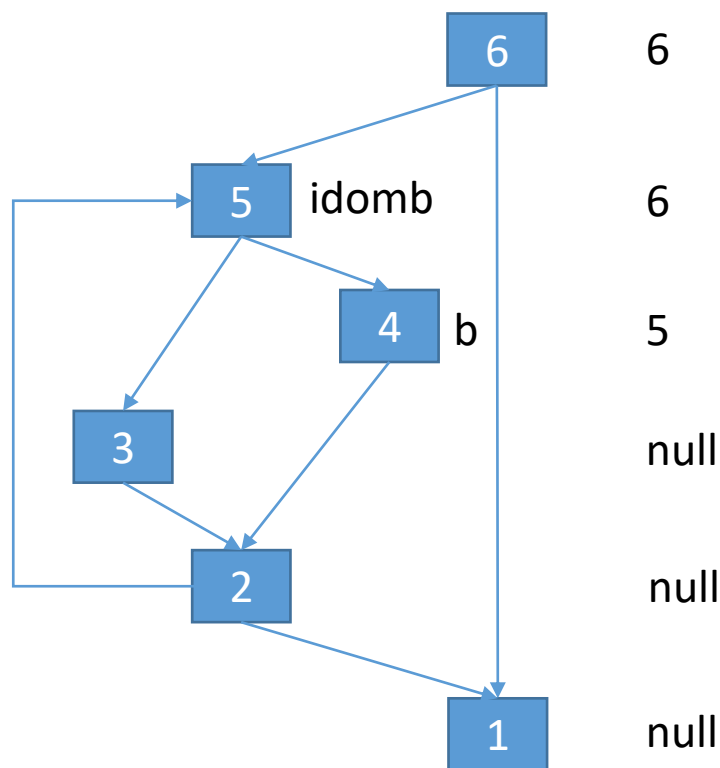
node 公共祖先(p1, p2) {

while (p1 \neq p2) {

while (p1 < p2) p1 = $\text{idom}(p1)$;

while (p2 < p1) p2 = $\text{idom}(p2)$; }

return p1; }





直接支配者计算例子

对所有结点 n , $idom(n)=null$;

$idom(entry)=entry$;

do {

逆后序遍历 $entry$ 以外的结点 b {

$idomb$ =任意 $idom$ 不为空的前驱;

对其他 $idom$ 不为空的前驱 p {

$idomb$ =公共祖先($p, idomb$); }

$idom(b)=idomb$; }

} while($idom$ 有修改);

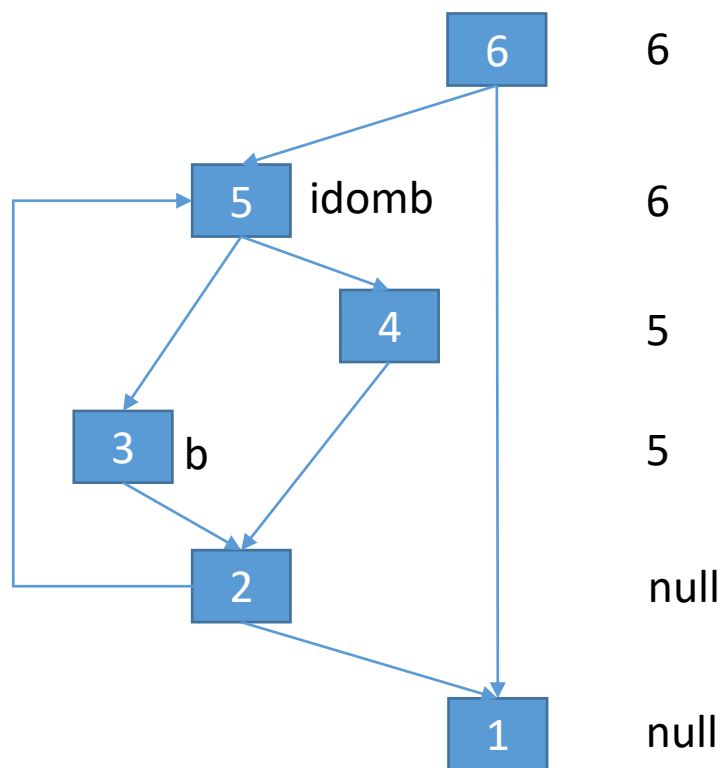
node 公共祖先($p1, p2$) {

while ($p1 \neq p2$) {

while ($p1 < p2$) $p1 = idom(p1)$;

while ($p2 < p1$) $p2 = idom(p2)$; }

return $p1$; }





直接支配者计算例子

对所有结点n, $\text{idom}(n)=\text{null}$;

$\text{idom}(\text{entry})=\text{entry}$;

do {

 逆后序遍历entry以外的结点b {

$\text{idomb}=\text{任意idom不为空的前驱}$;

 对其他 idom 不为空的前驱p {

$\text{idomb}=\text{公共祖先}(p, \text{idomb})$; }

$\text{idom}(b)=\text{idomb}$; }

 } while(idom 有修改);

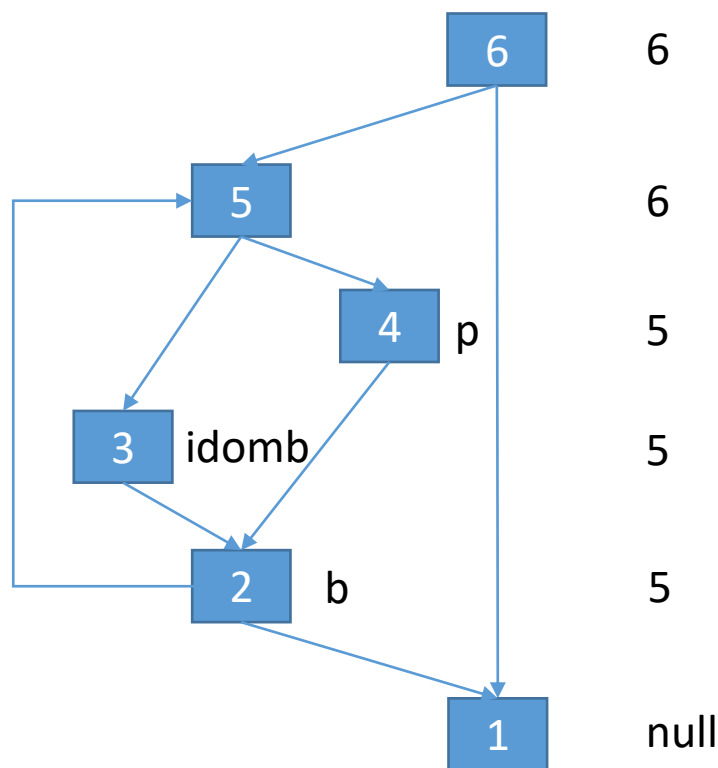
node 公共祖先(p_1, p_2) {

 while ($p_1 \neq p_2$) {

 while ($p_1 < p_2$) $p_1 = \text{idom}(p_1)$;

 while ($p_2 < p_1$) $p_2 = \text{idom}(p_2)$; }

 return p_1 ; }





直接支配者计算例子

对所有结点 n , $idom(n)=null$;

$idom(entry)=entry$;

do {

 逆后序遍历 $entry$ 以外的结点 b {

$idomb$ =任意 $idom$ 不为空的前驱;

 对其他 $idom$ 不为空的前驱 p {

$idomb$ =公共祖先($p, idomb$); }

$idom(b)=idomb$; }

} while($idom$ 有修改);

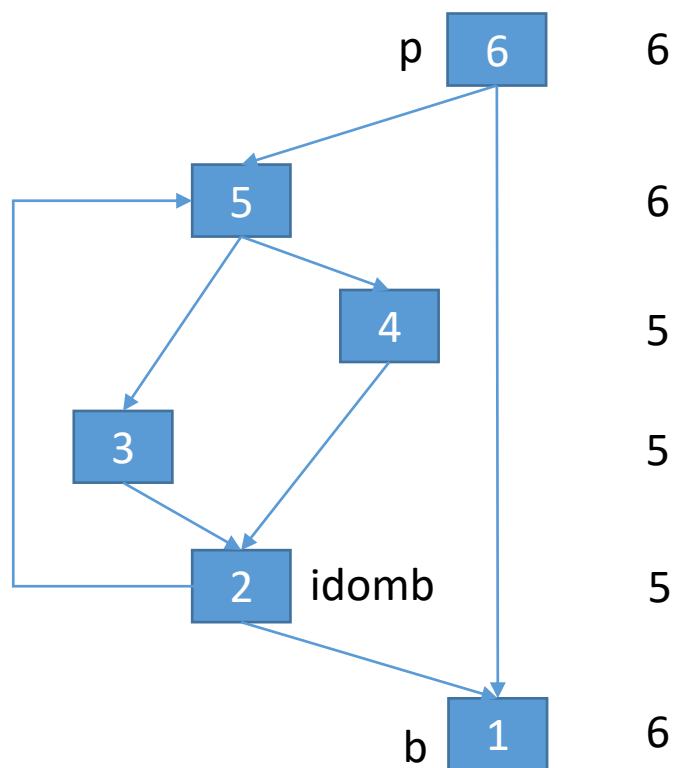
node 公共祖先($p1, p2$) {

 while ($p1 \neq p2$) {

 while ($p1 < p2$) $p1 = idom(p1)$;

 while ($p2 < p1$) $p2 = idom(p2)$; }

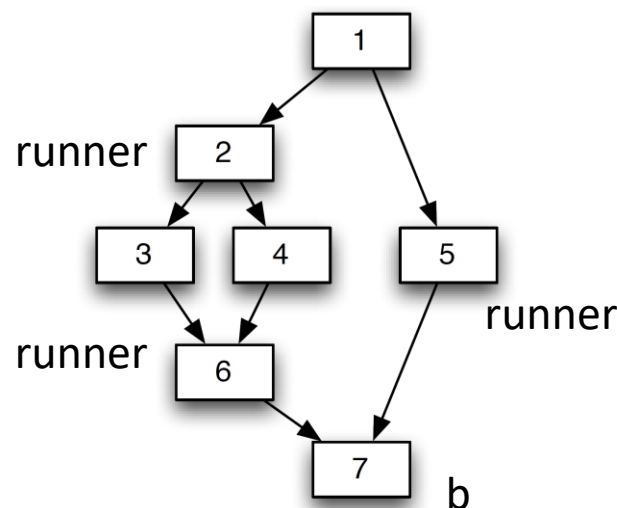
 return $p1$; }





计算支配边界

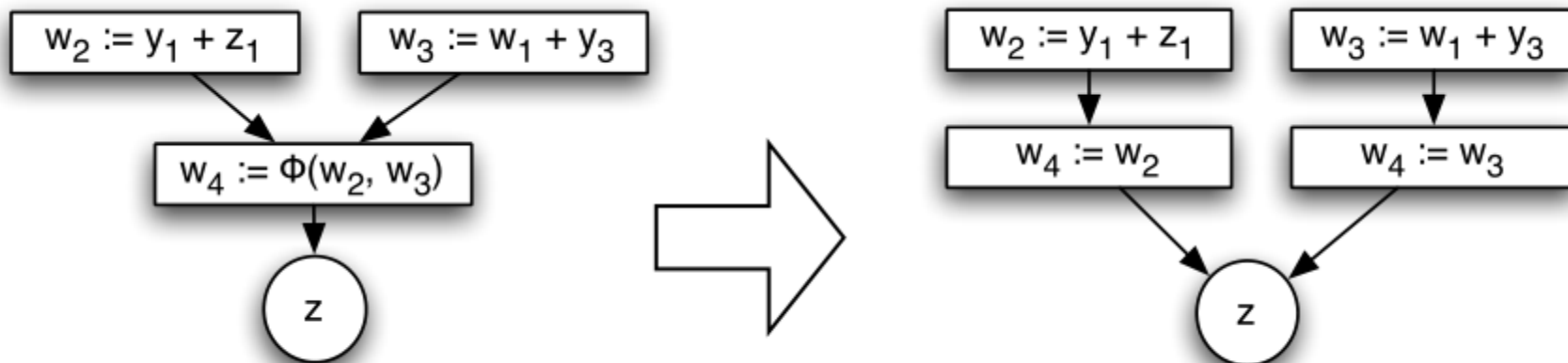
```
for(每个结点b)
  if b的前驱结点数  $\geq 2$ 
    for(每个b的前驱p)
      runner := p
      while runner  $\neq$  idom(b)
        将b加入runner的支配边界
      runner := idom(runner)
```





转换回标准型

- 有些分析任务中我们需要再从静态单赋值转换回标准型
 - 程序优化
- 转换过程就是删除掉静态单赋值中的 ϕ 函数





实践中的静态单赋值形式

- 静态单赋值要求每个变量只被赋值一次
- 基于静态单赋值优化数据流分析的条件：
 - 需要分析的每一个内存位置一旦赋值都不会发生改变。
- 这个条件总能成立吗？

C:

```
a=10;  
i=&a;  
*i=10;
```

Java:

```
a.f=10;  
y=a.f;  
a.f=20;  
y=a.f;
```



解决方案： 部分SSA

- 把内存位置分成两组，转换SSA的时候只转换能转换的组，并只对转换的组做优化
- Java的情况： 栈上的变量为优化组，堆上的变量为不优化组
- C的情况： 把变量分成address-taken和top-level的两组
 - address-taken: 曾经被&取过地址的变量
 - top-level: 从没被&取过地址的变量



C的情况的例子

```
int a, b, *c, *d;
```

a-d均为address-taken变量

```
int* w = &a;
```

$w_1 = \text{ALLOC}_a$

```
int* x = &b;
```

$x_1 = \text{ALLOC}_b$

```
int** y = &c;
```

$y_1 = \text{ALLOC}_c$

```
int** z = y;
```

$z_1 = y_1$

```
    c = 0;
```

STORE 0 y_1

```
    *y = w;
```

STORE w_1 y_1

```
    *z = x;
```

STORE x_1 z_1

```
    y = &d;
```

$y_2 = \text{ALLOC}_d$

```
    z = y;
```

$z_2 = y_2$

```
    *y = w;
```

STORE w_1 y_2

```
    *z = x;
```

STORE x_1 z_2

LLVM IR所采用的SSA形式



作业

- 采用你喜欢的任意程序分析框架（如太阿、LLVM、SOOT）
- 构造一个无法完全转换成SSA的程序，用所选框架转换成部分SSA
- 提交原始代码和转换后的代码，并说明哪些地方没有做到赋值后内存位置不变。



参考资料

- 静态单赋值转换算法
 - 《编译原理》相应章节
 - Cooper et al. A Simple, Fast Dominance Algorithm.
- 稀疏分析相关论文
 - 用 “sparse program analysis” 为关键字进行搜索