软件分析

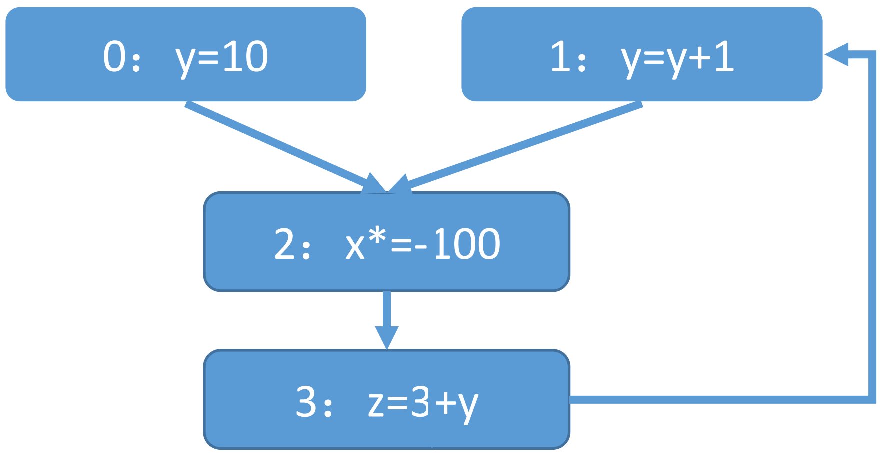
静态单赋值和稀疏分析

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

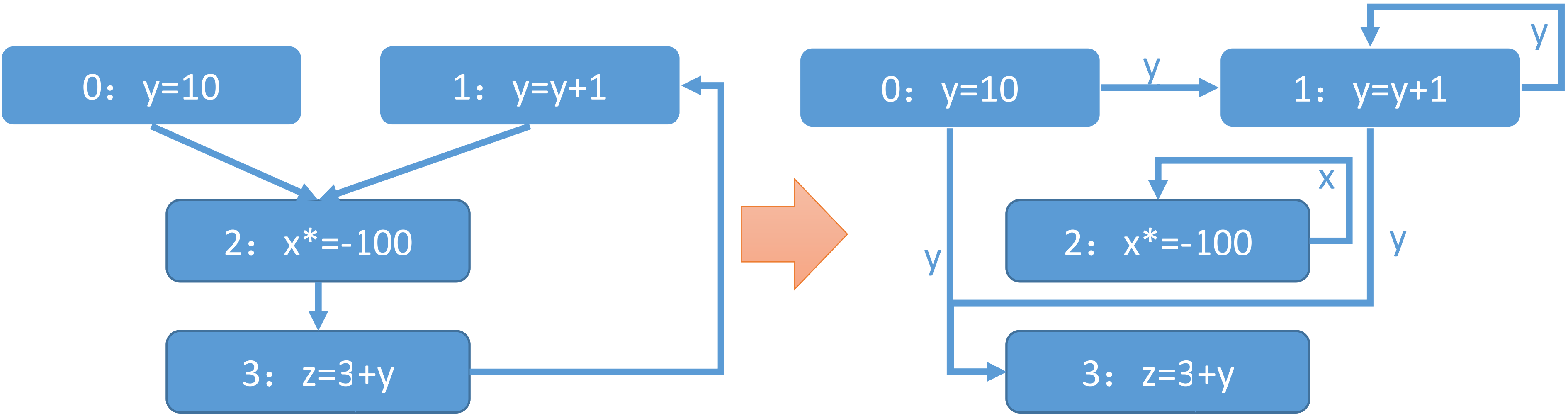


- 问题1： 每个结点都要保存一份关于x,y,z的值
  - 即使结点2和y没有关系
- 问题2： 当1的转换函数更新y的时候，该更新只和3有关，但我们不可避免的要通过2才能到达3

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Def-Use关系

基于Def-Use的数据流分析



- 每个结点只保存自己定义的变量的抽象值
- 只沿着Def-Use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

$$y_0 = f_0(\quad)$$
$$y_1 = f_1(y_0 \sqcap y_1)$$
$$x_2 = f_2(x_2 \sqcap x_0)$$
$$z_3 = f_3(y_0 \sqcap y_1)$$

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

基于Def-Use的数据流分析：问题1

- 如何获取Def-Use关系
  - 可以通过Reaching Definition获取Def-Use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

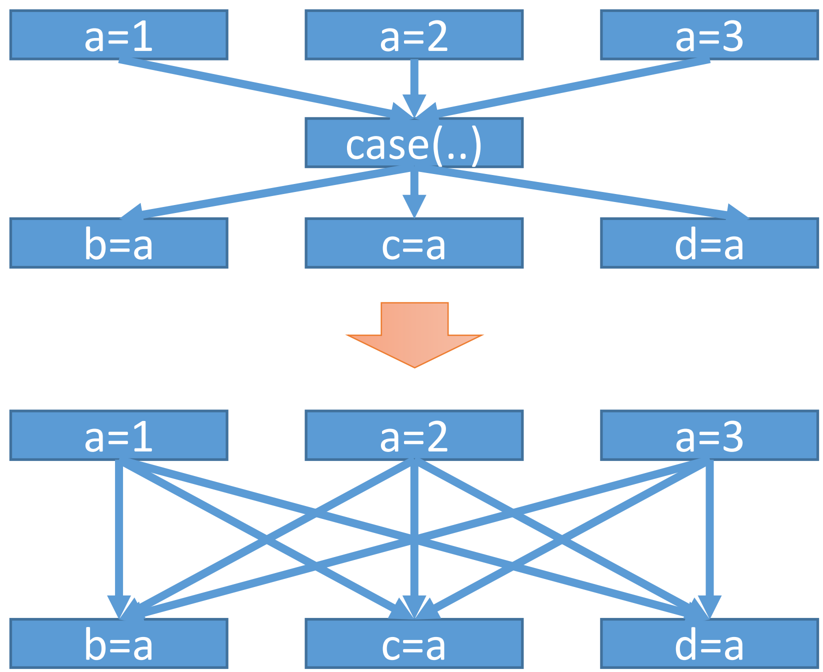
- Reaching Definition的复杂度
  - 程序中赋值语句个数为m，控制流图结点为n
  - 更新单个节点的时间为O(m)（假设并集和差集的时间复杂度都是O(m)）
  - 总共需要更新O(mn)次
  - 总时间$O(nm^2)$

- Reaching Definition本身就不够快

基于Def-Use的数据流分析：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
```



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

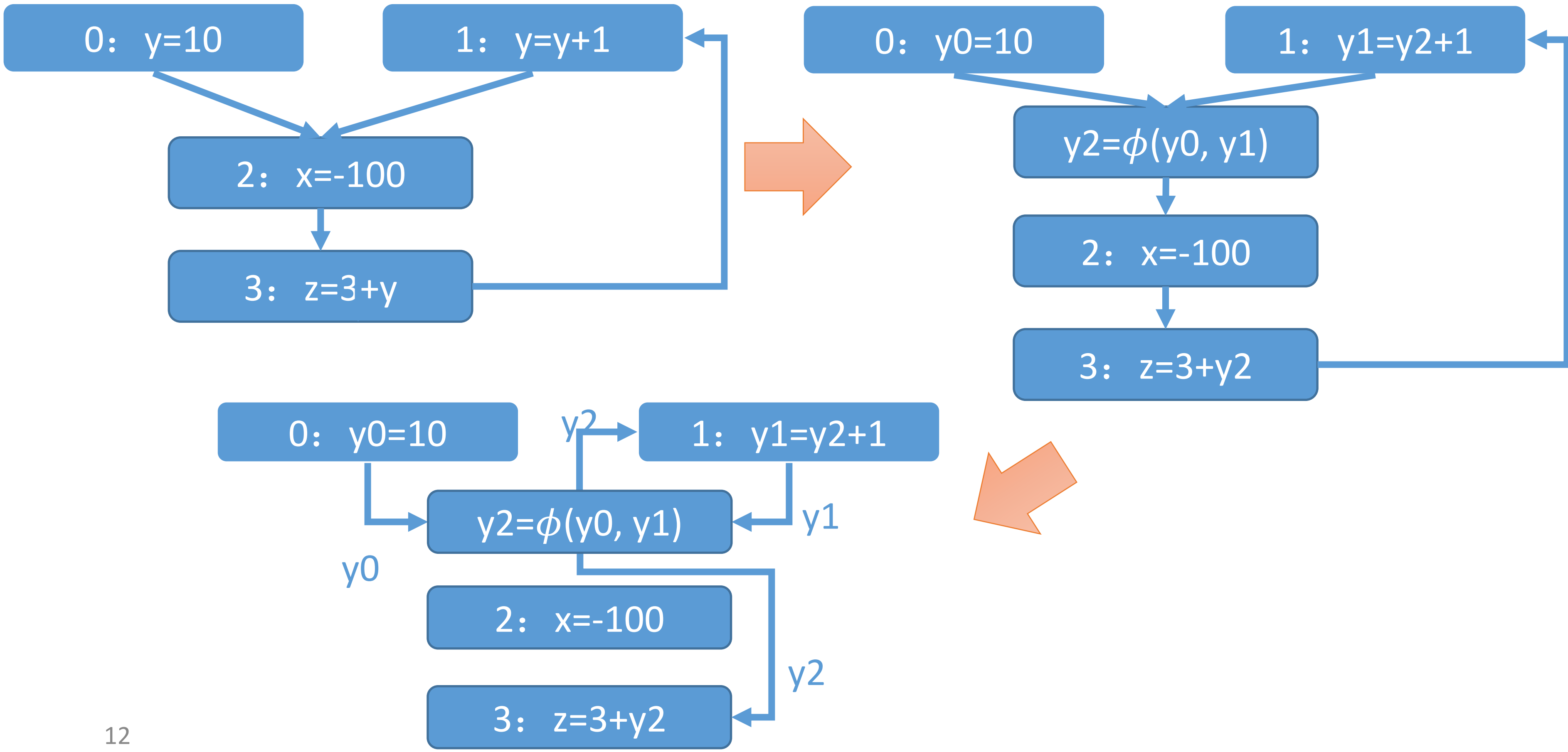
```
x=10;
x+=y;
if (x>10)
  z=10;
else
  z=20;
x+=z;
```

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=φ(z0, z1);
x2=x1+z2;
```

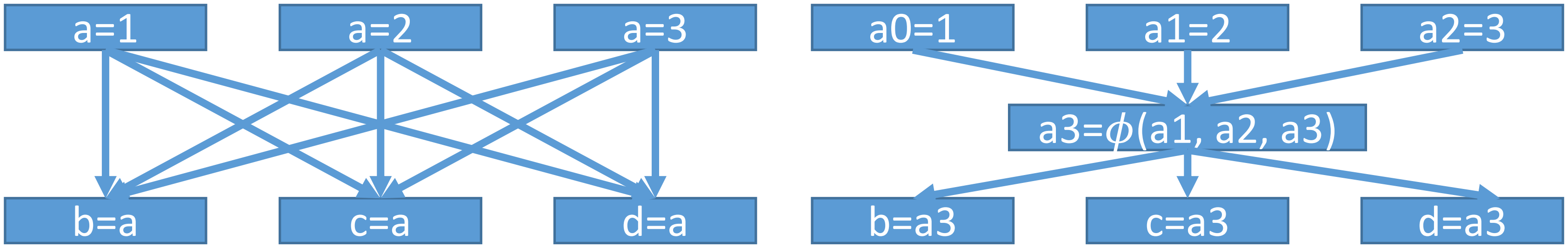
静态单赋值与数据流分析

- 静态单赋值直接提供了Def-Use链



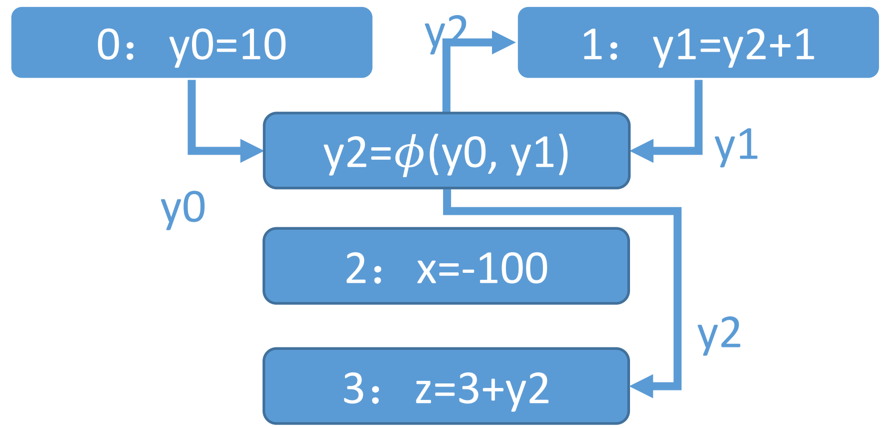
静态单赋值的好处

- 静态单赋值存在高效算法
- 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等价

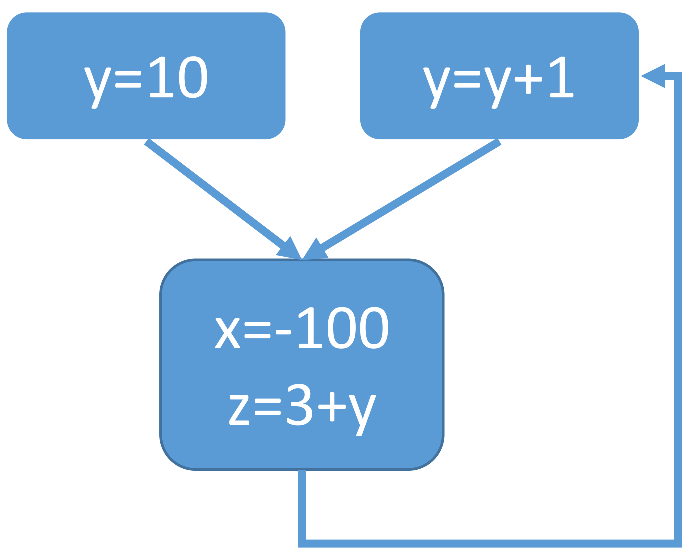


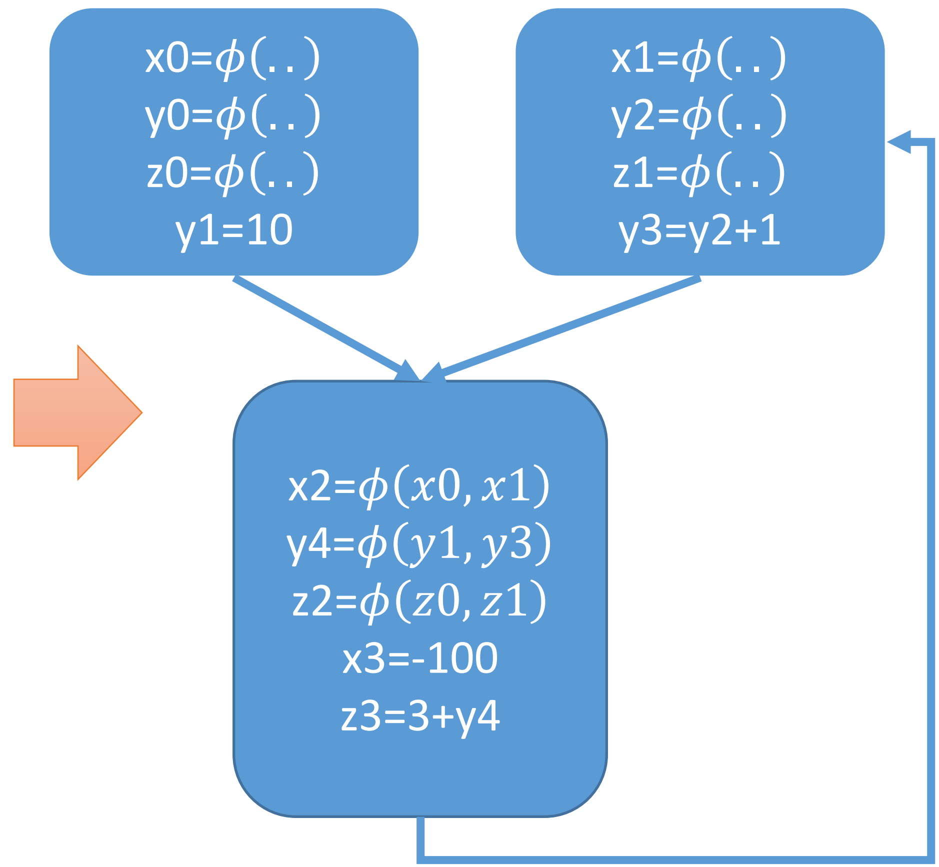
基于静态单赋值形式的分析通常又称为稀疏分析Sparse Analysis

- 流非敏感分析：
  - 每次有值变化时，挑选受影响的转换函数重新执行
  - 全局只保存一份抽象数据
- 流敏感分析：
  - 每次有值变化时，沿着边寻找后继节点的转换函数重新执行
  - 每个结点保存一份抽象数据

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



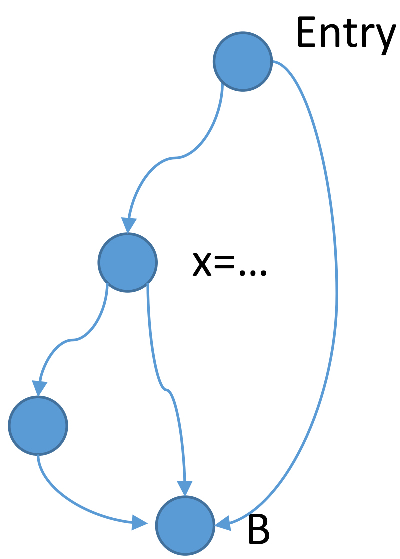


简单算法的问题

- 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Def-Use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

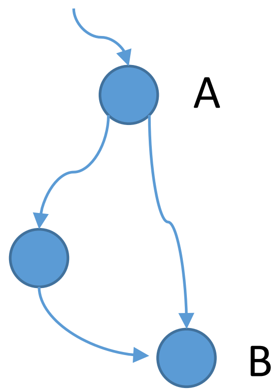
加入ϕ函数的条件

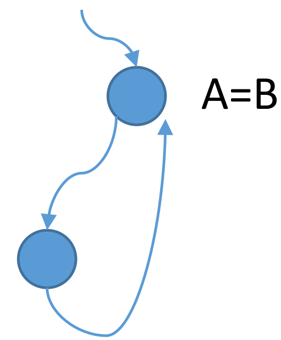
- 至少两条路径在B处汇合
- 其中一条经过了某个赋值语句
- 另外一条没有经过
- 赋值语句和B之间没有别的语句满足上述条件



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

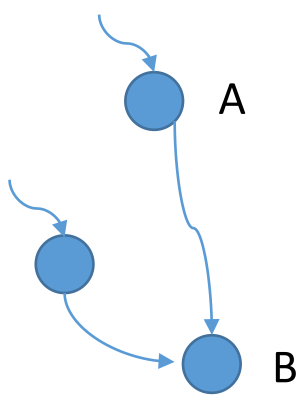


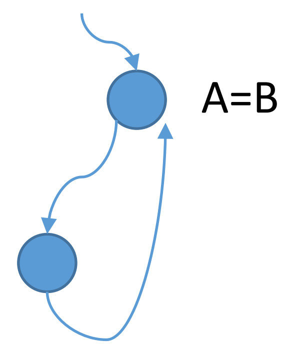


- 结点A严格支配（Strictly dominate）结点B：A支配B并且A和B不是一个结点
  - A不严格支配B => 至少存在一条路径，在到达B之前不经过A

支配边界Dominance Frontier

- 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





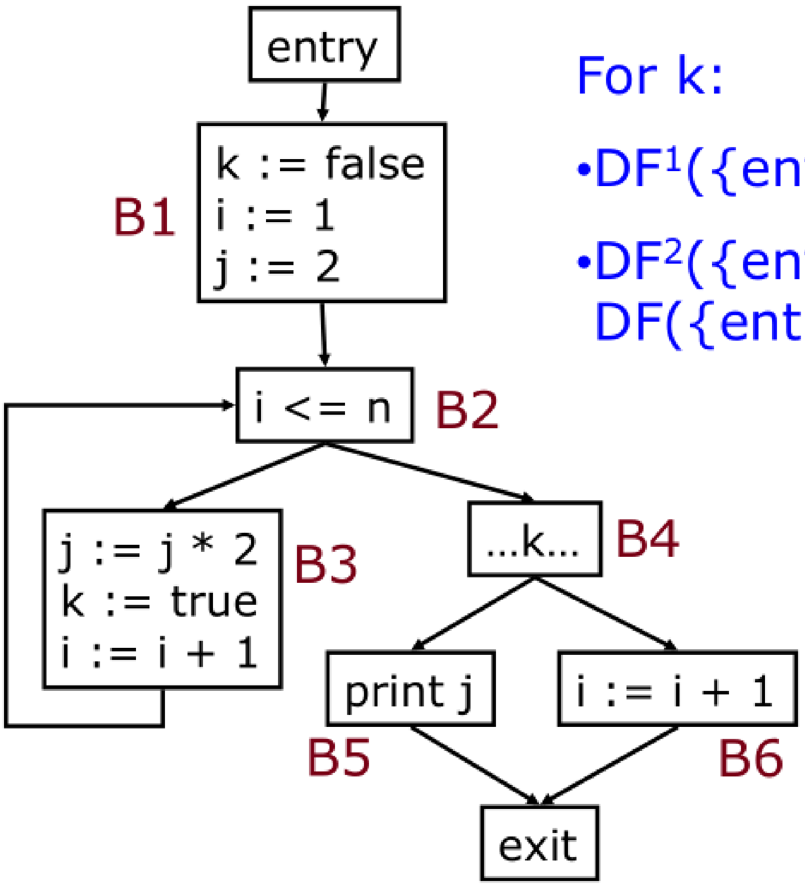
- 对任意赋值语句x=...所在的结点A，所有A的支配边界需要插入ϕ函数计算x的值

转换到静态单赋值形式

- 令DF(a)为a的支配边界集合
- 定义
  - $\mathrm{DF}(\mathrm{A}) = \bigcup_{\{\mathrm{a} \in \mathrm{A}\}} \mathrm{DF}(\mathrm{a})$
  - $\mathrm{DF}^{+}(\mathrm{A}) = \lim_{i \to \infty} DF^{i}(A)$
    - $DF^{1}(A) = DF(A)$
    - $DF^{i+1}(A) = DF\left(\bigcup_{j \leq i} DF^{j}(A)\right)$

- 对任意变量i，令A为所有对i赋值的结点，$\mathrm{DF}^{+}(\mathrm{A})$就是所有需要插入$\phi$函数的结点

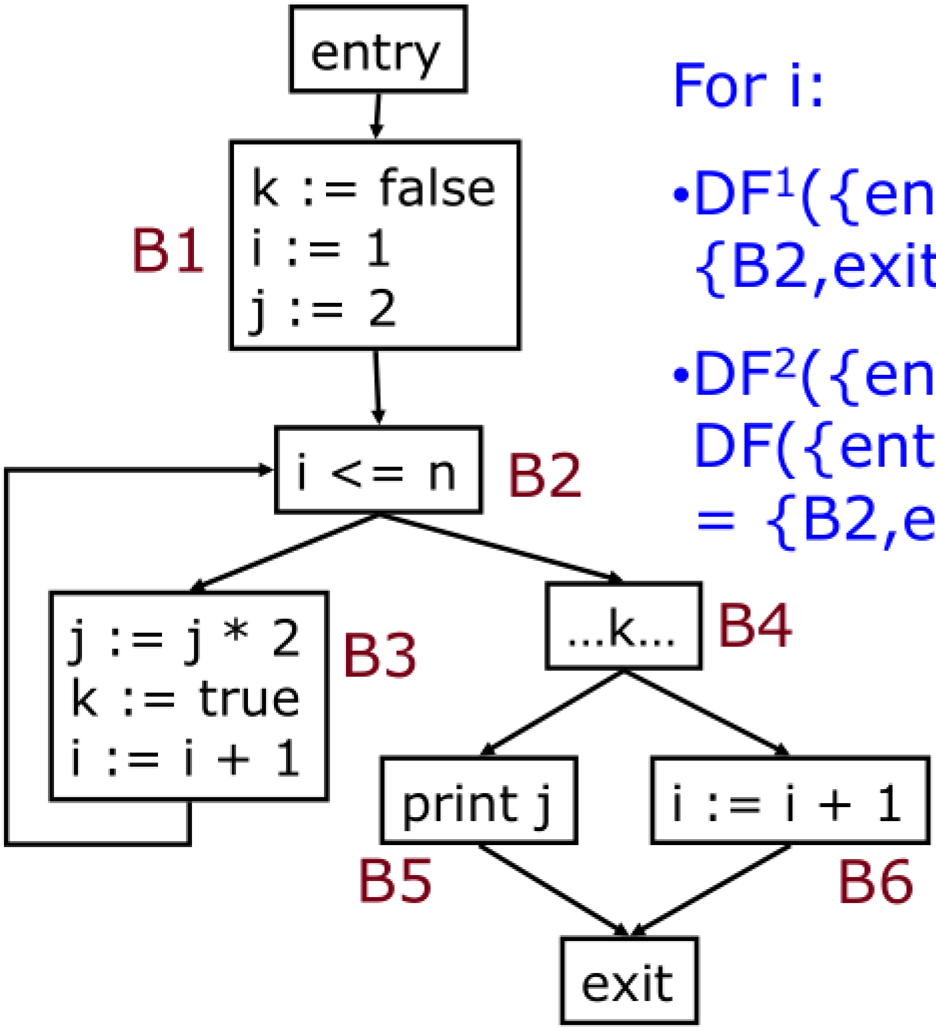
转换示例



For k:

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

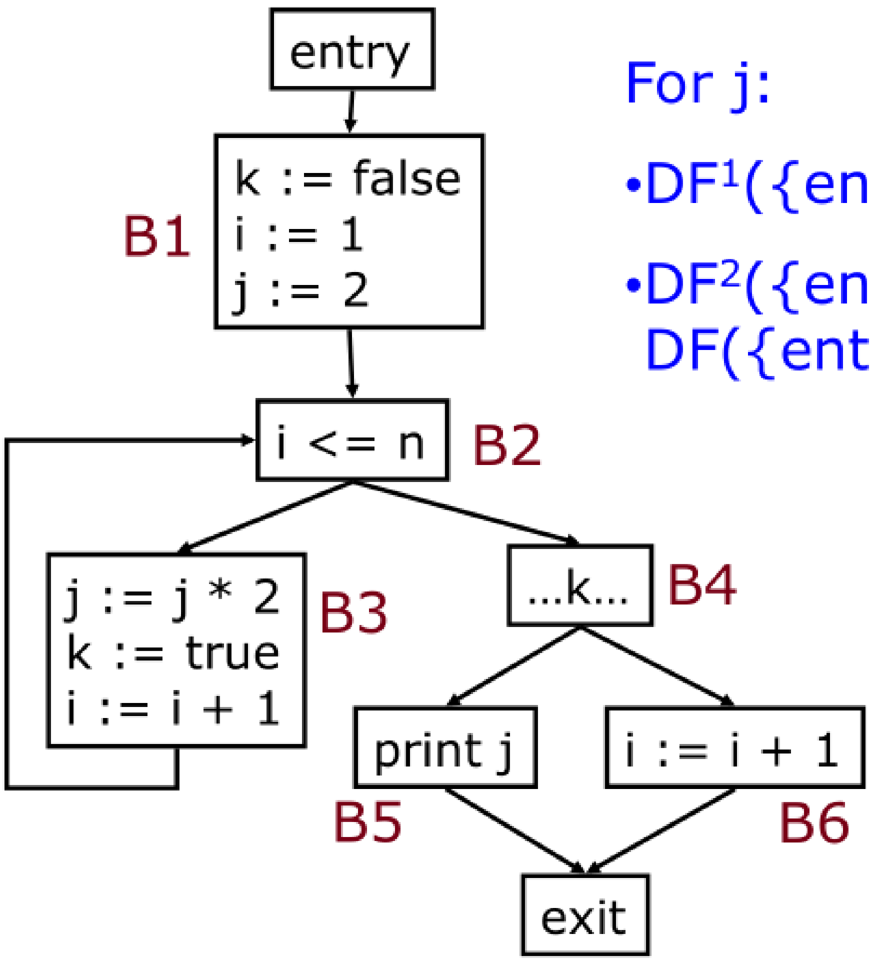
转换示例



For i:

- $DF^1(\{entry, B1, B3, B6\}) = \{B2, exit\}$
- $DF^2(\{entry, B1, B3, B6\}) = DF(\{entry, B1, B2, B3, B6, exit\}) = \{B2, exit\}$

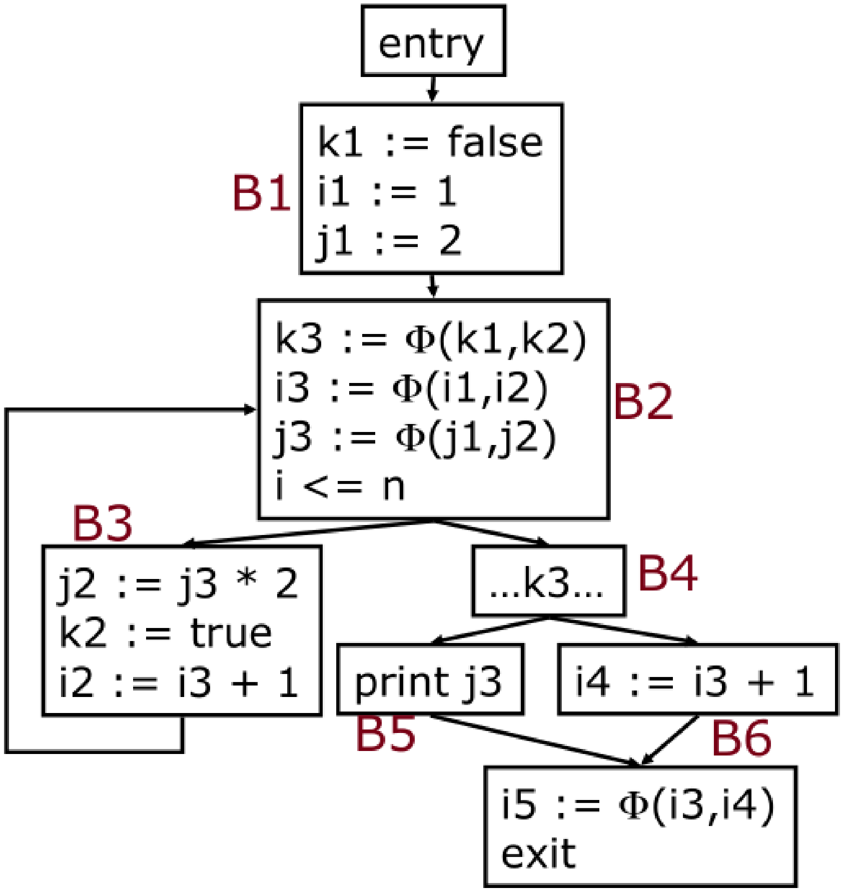
转换示例



For j:

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

转换结果


entry
B1
k1 := false
i1 := 1
j1 := 2
k3 := Φ(k1,k2)
i3 := Φ(i1,i2)
j3 := Φ(j1,j2)
i <= n
B2
B3
j2 := j3 * 2
k2 := true
i2 := i3 + 1
...k3...
B4
print j3
i4 := i3 + 1
B5
B6
i5 := Φ(i3,i4)
exit

计算支配边界的算法

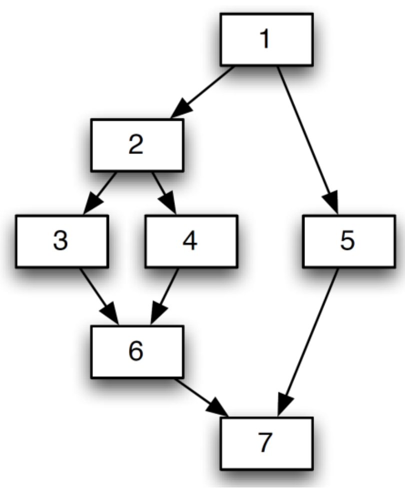
- Lengauer and Tarjan算法
  - 复杂度为O(Eα(E,N))
  - E为边数，N为结点数，α为Ackerman函数的逆
  - Ackerman函数基本可以认为是常数
- Cooper, Harvey, Kennedy算法，2001年
  - 复杂度为$\mathrm{O}(\mathrm{N}^2)$
  - 在实践中更常见的小控制流图（1000结点以下）上在比Lenauer and Tarjan算法要快
  - 下面介绍CHK算法

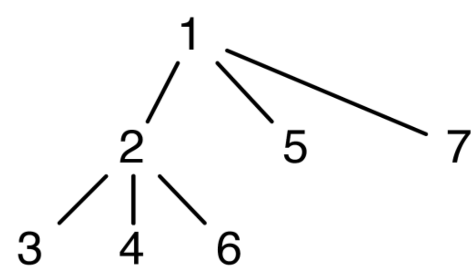
Cooper, Harvey, Kennedy算法

- 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idom(b)
- 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

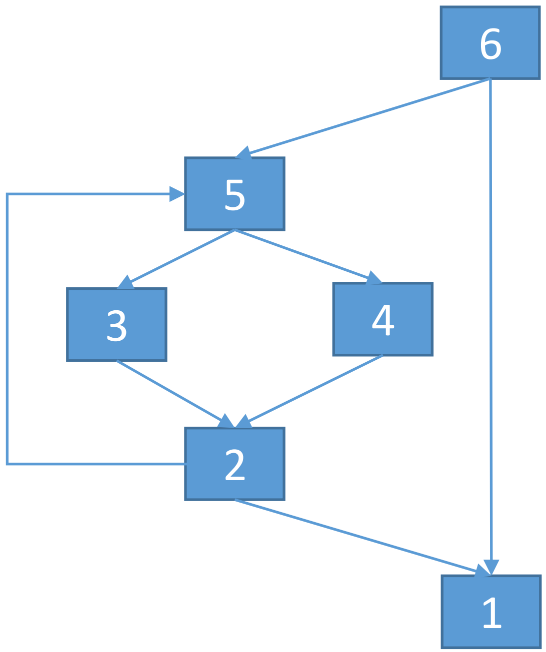




图的后序遍历

- 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)
}
```



直接支配者计算算法

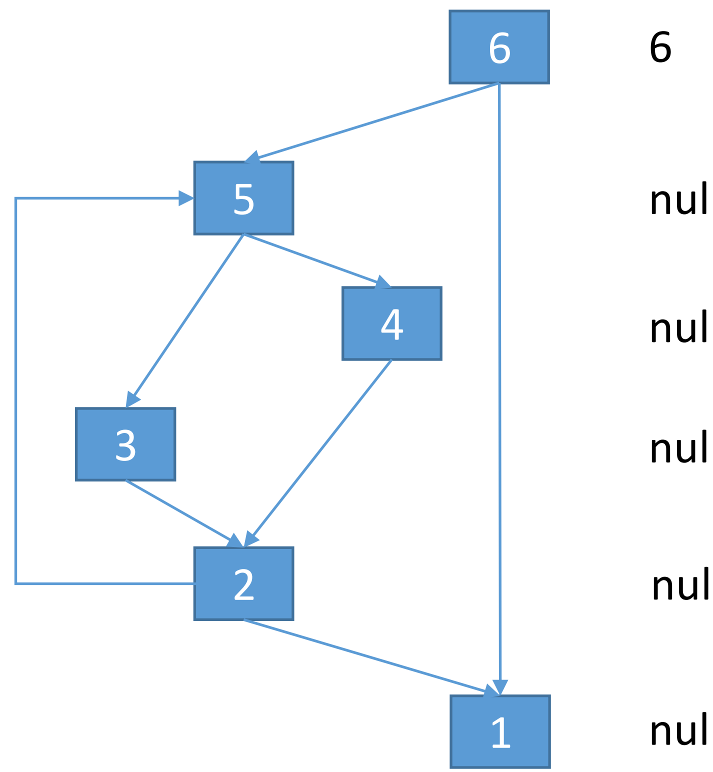
```
对所有结点n，idom(n)=null;
idom(entry)=entry;
do {
  逆后序遍历entry以外的结点b {
    idomb=任意idom不为空的前驱;
    对其他idom不为空的前驱p {
      idomb=公共祖先(p, idomb);
    }
    idom(b)=idomb;
  }
} while(idom有修改);
```

```
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

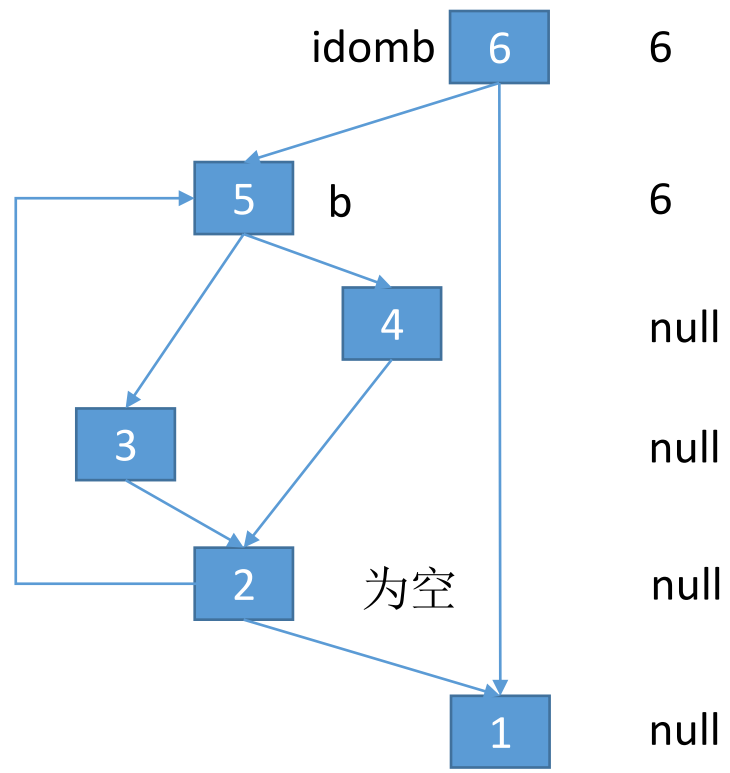
```
对所有结点n，idom(n)=null;
idom(entry)=entry;
do {
 逆后序遍历entry以外的结点b {
   idomb=任意idom不为空的前驱;
   对其他idom不为空的前驱p {
     idomb=公共祖先(p, idomb); }
   idom(b)=idomb; }
} while(idom有修改);

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



直接支配者计算例子

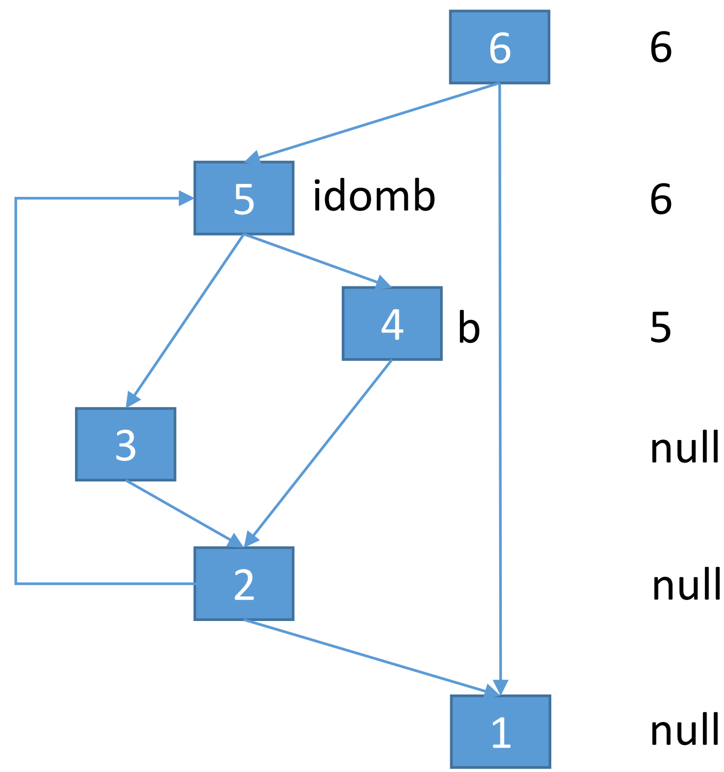
对所有结点n，idom(n)=null;
idom(entry)=entry;
do {
  逆后序遍历entry以外的结点b {
    idomb=任意idom不为空的前驱;
    对其他idom不为空的前驱p {
      idomb=公共祖先(p, idomb); }
    idom(b)=idomb; }
} while(idom有修改);

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



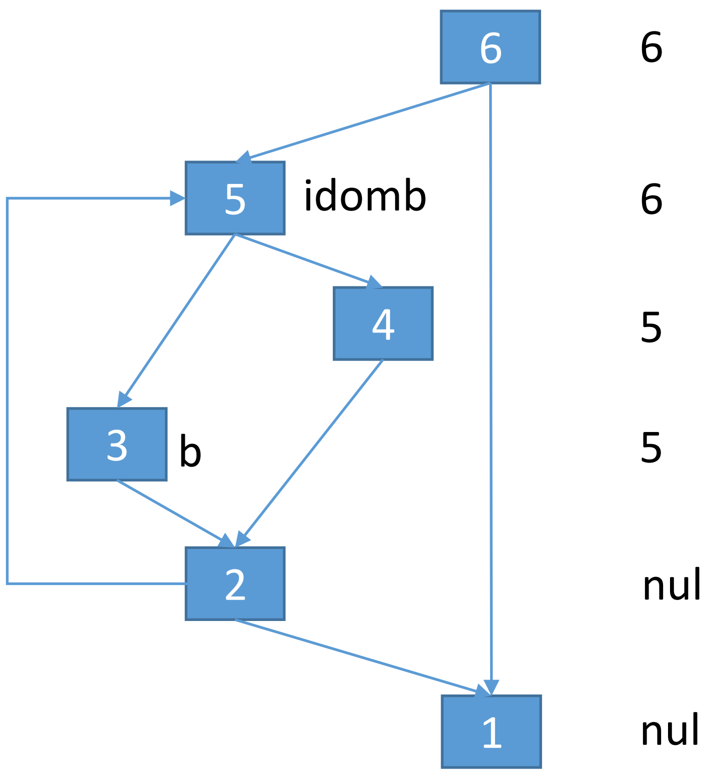
直接支配者计算例子

```
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 ≠ p2) {
    while (p1 < p2) p1 = idom(p1);
    while (p2 < p1) p2 = idom(p2);  }
  return p1; }
```



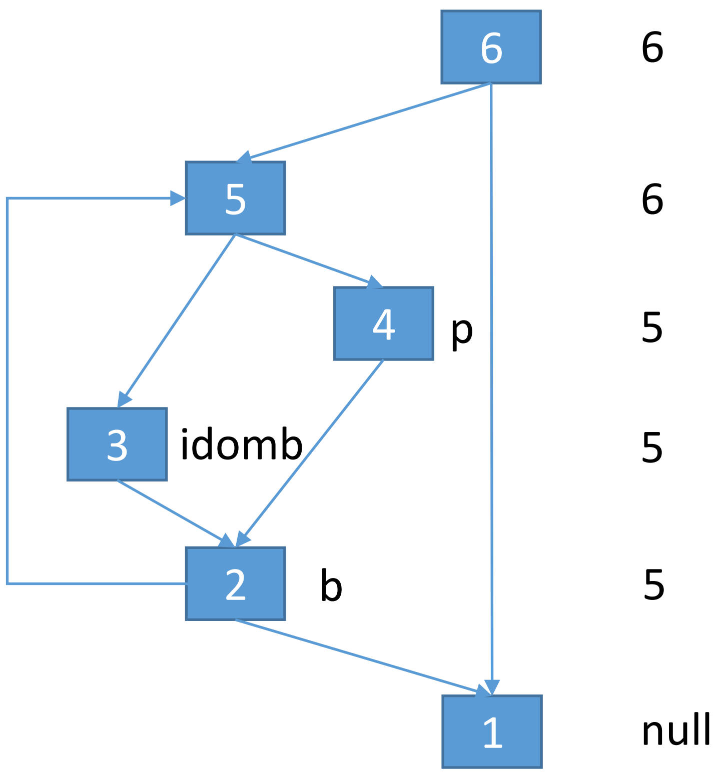
直接支配者计算例子

```
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 ≠ p2) {
    while (p1 < p2) p1 = idom(p1);
    while (p2 < p1) p2 = idom(p2); }
  return p1; }
```



直接支配者计算例子

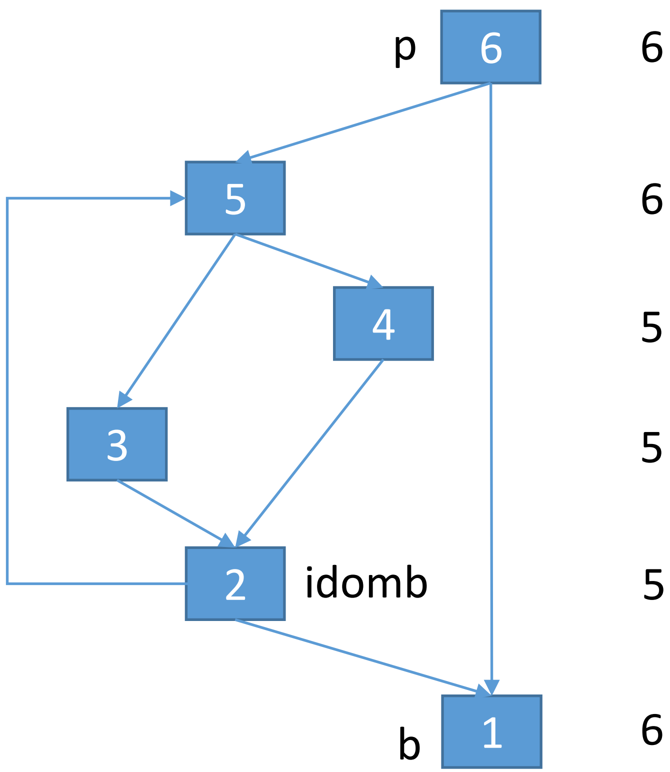
```
对所有结点n，idom(n)=null;
idom(entry)=entry;
do {
 逆后序遍历entry以外的结点b {
   idomb=任意idom不为空的前驱;
   对其他idom不为空的前驱p {
     idomb=公共祖先(p, idomb); }
   idom(b)=idomb; }
} while(idom有修改);

node 公共祖先(p1, p2) {
 while (p1 ≠ 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; }



计算支配边界

for(每个结点b)

  if b的前驱结点数 ≥ 2

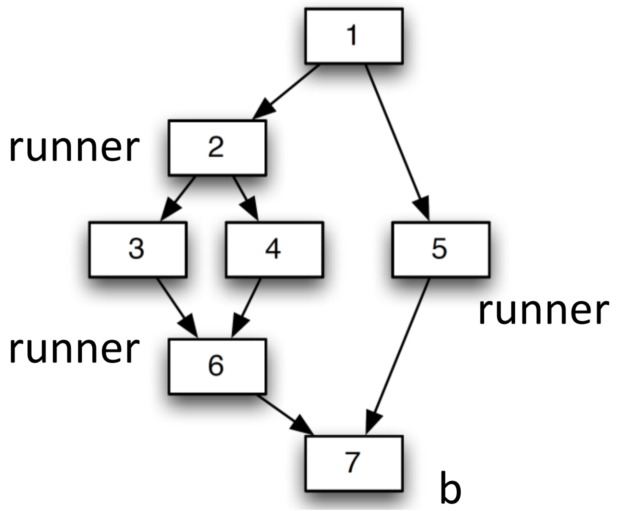
    for(每个b的前驱p)

      runner := p

      while runner ≠ idom(b)

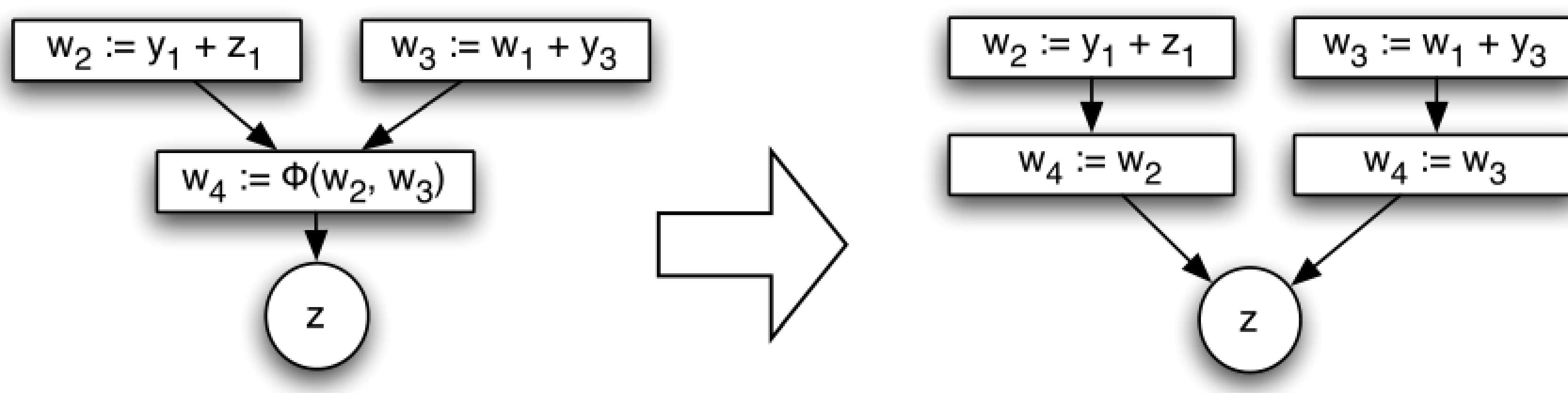
        将b加入runner的支配边界

        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;

int*  w = &a;
int*  x = &b;
int** y = &c;
int** z =  y;
      c =  0;
     *y =  w;
     *z =  x;
      y = &d;
      z =  y;
     *y =  w;
     *z =  x;
```

a-d均为address-taken变量

$w_1 = \mathrm{ALLOC}_a$
$x_1 = \mathrm{ALLOC}_b$
$y_1 = \mathrm{ALLOC}_c$
$z_1 = y_1$
STORE 0 y_1
STORE w_1 y_1
STORE x_1 z_1
$y_2 = \mathrm{ALLOC}_d$
$z_2 = y_2$
STORE w_1 y_2
STORE x_1 z_2

LLVM IR所采用的SSA形式

参考资料

- 静态单赋值转换算法
  - 《编译原理》相应章节
- 稀疏分析相关论文
  - 用“sparse program analysis”为关键字进行搜索