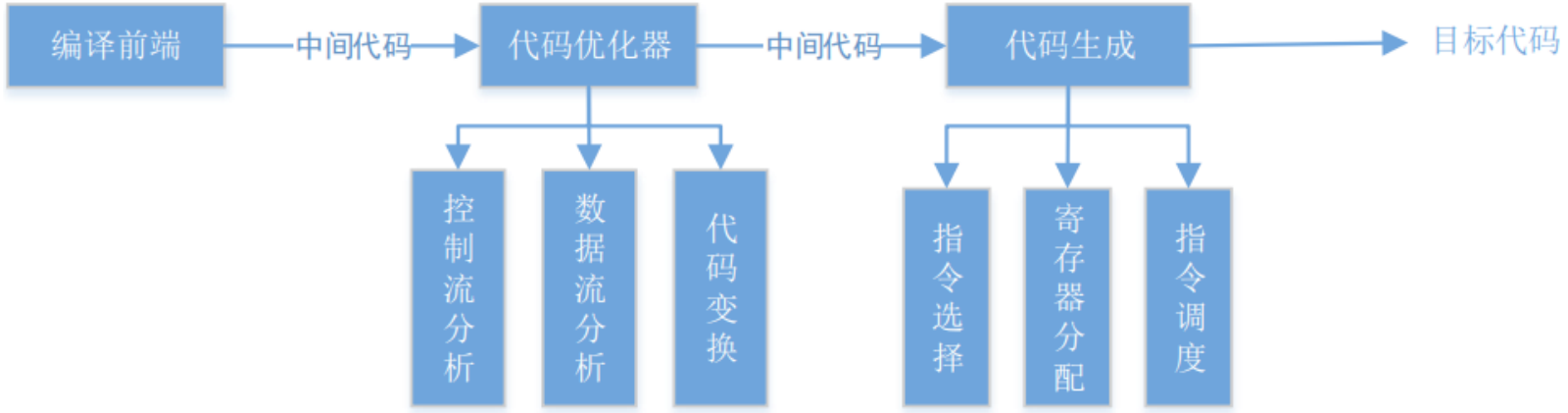
中国人民大学
RENMIN UNIVERSITY OF CHINA

编译原理
--控制流和数据流

刘爽
中国人民大学信息学院

第二部分：编译优化与代码生成



- 优化就是通过分析编译过程中的中间代码，对其进行变换，以便生成更好的目标代码
- 更好：运行更快/代码更小/更安全，等等

问题的引入

- 优化 = 分析 + 转换
- 分析是优化的基础

```
double compute_pi(int n) {
   int  i;
   double  w, x, sum, pi;
   w = 1.0/n;
   sum = 0.0;
   for(i = 1; i<= n;i++){
     x = w * (i- 0.5);
     sum = sum + 4.0 / (1.0 + x*x);
   }
  pi = w * sum;
  return pi;
}
```

辛普森公式计算程序

```
1     double compute_pi(int n) {
2       w = 1.0/n;
3       sum = 0.0;
4       L2: if i > n goto L1
5         t1 = i – 0.5
6         x = w * t1
7         t2 = x*x
8         t3 = 1.0+t2
9         t4 = 4.0/t3
10        sum = sum + t4
11      i=i+1
12    goto L2
13    L1: pi = w * sum;
14    return pi;
}
```

中间代码

问题的引入

- 优化 = 分析 + 转换
- 分析是优化的基础
  - 流分析
    - 控制流分析（Control flow Analysis）
    - 数据流分析（Data flow Analysis）
  - 别名分析（Alias Analysis）
  - 依赖关系分析（Dependence Analysis）

本章目标：理解和掌握控制流分析和数据流分析的基本原理和方法

内容

一．控制流分析

1. 基本块和控制流图
2. 必经关系
3. 循环

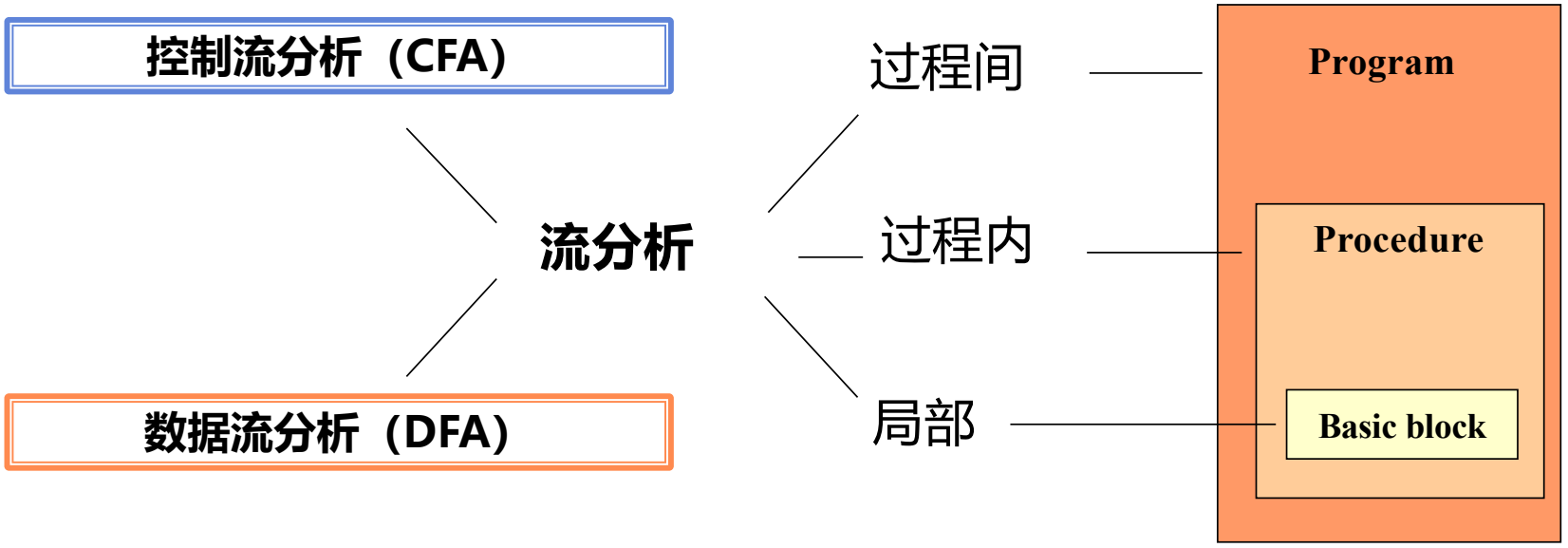
二．数据流分析

1. 数据流分析的基本方法
2. 到达-定值分析
3. 活跃变量分析
4. 迭代的数据流分析框架
5. 其他数据流问题
6. 静态单一赋值

一、 控制流分析

➢ 控制流分析

- 分析判断程序的控制结构，构建程序的**控制流图**（*Control Flow Graph，CFG*）



- 控制流图是后续程序分析和变换的基础

1、基本块和控制流图

- 基本块(Basic Block)
  - 基本块是一段只能从它的开始处进入，结束处离开的顺序代码序列
  - 基本块只有最后一条语句是分支语句，并且只有第一条语句是分支的目标
  - 在基本块内部，除了第一条指令外，每条指令都只有一个前驱；除了最后一条指令外，每条指令只有一个后继

- 使用基本块代替指令，可以减少分析的时间和空间开销

1.1 识别基本块

1. 首先，扫描程序，使用下列规则识别基本块的首语句：

   (规则1) 程序/函数的第一条语句

   (规则2) 分支语句的目标 (对大多数中间代码，
   分支的目标是带有标号的语句)

   (规则3) 任何紧跟在分支或者return语句后面的语句

2. 每个首语句对应的基本块包括了首语句+下一个首语句之间（不包括下一个首语句）的语句

1.1 识别基本块

例：计算自然数n的阶乘的程序。

当n等于0时，fac等于1，n大于0时，fac=n!。

1、首先找出所有的首语句。

```
1          fac = 1;              规则1
2          t1 = n == 0;
3          if t1 goto L1
4      L2: t2 = i>n;             规则2
5          if t2 goto L1;
6          fac = fac * i:        规则3
7          i=i+1;
8          goto L2;
9      L1: return fac;           规则2
```

（规则1） 程序/函数的第一条语句

（规则2） 分支语句的目标 (对大多数中间代码， 分支的目标是带有标号的语句)

（规则3）任何紧跟在分支或者return语句后面的语句

1.1 识别基本块

2、根据首语句，划分基本块

1	fac = 1;	规则1	BB1
2	t1 = n == 0;		
3	if t1 goto L1		
4	L2: t2 = i>n;	规则2	BB2
5	if t2 goto L1;		
6	fac = fac * i:	规则3	BB3
7	i=i+1;		
8	goto L2;		
9	L1: return fac;	规则2	BB4

1.2 控制流图

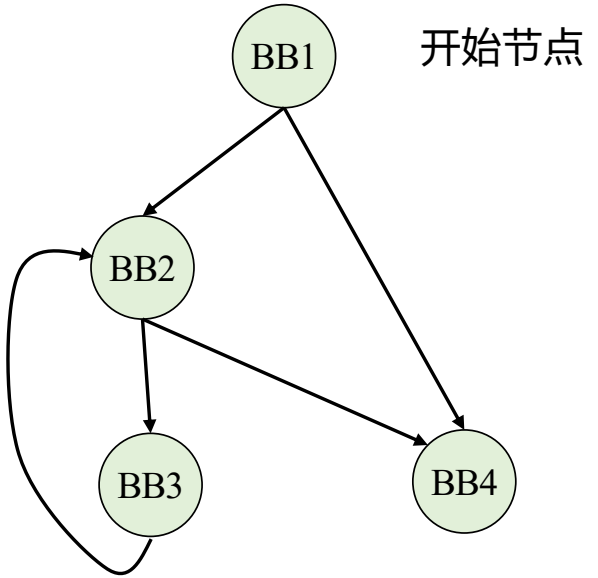
- 控制流图（*control flow graph* ,CFG）是一个有向流图G=(*N*,*E*),
  其中:
  - 节点*N*表示基本块
  - 边*E*表示程序的控制流向
- 首语句是第一条语句的称为开始节点（***start node***）
- CFG中的边只表示程序可能走这条路径

1.2 构建控制流图

- 如果基本块BB1和BB2满足下面的条件之一，则存在一条从BB1到BB2的有向边
  - （1）从BB1的最后一条语句可以跳转到BB2的第一条语句
  - （2）BB2紧跟在BB1之后，并且BB1的最后一条指令不是无条件转移

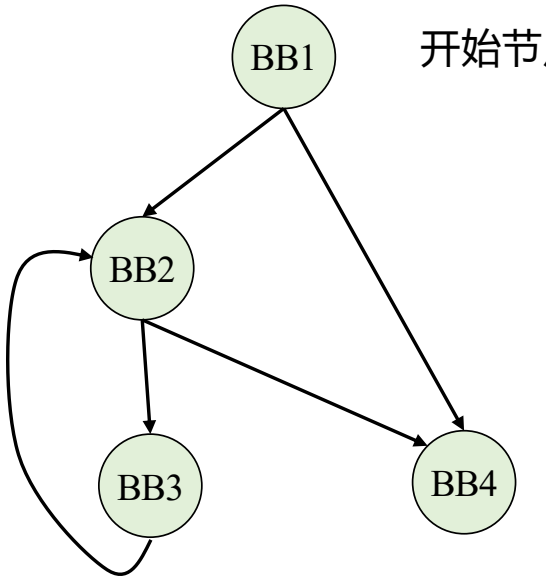
1.2 构建控制流图

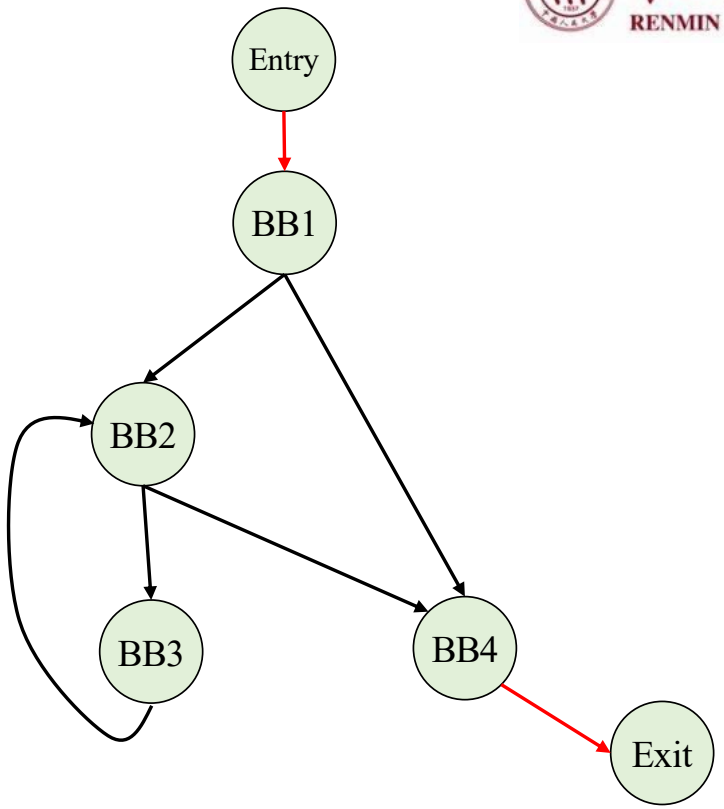
1	fac = 1;	规则1	BB1
2	t1 = n == 0;		
3	if t1 goto L1		
4	L2: t2 = i>n;	规则2	BB2
5	if t2 goto L1;		
6	fac = fac * i:	规则3	BB3
7	i=i+1;		
8	goto L2;		
9	L1: return fac;	规则2	BB4



控制流图

1.2 构建控制流图





入口节点（Entry node）：一条entry到开始节点的边

出口节点（Exit Node）：CFG中，每一个没有后续的基本块都有一条边转向exit

1.3 前驱和后继

定义: 控制流图 $G = (N, E)$，其中：

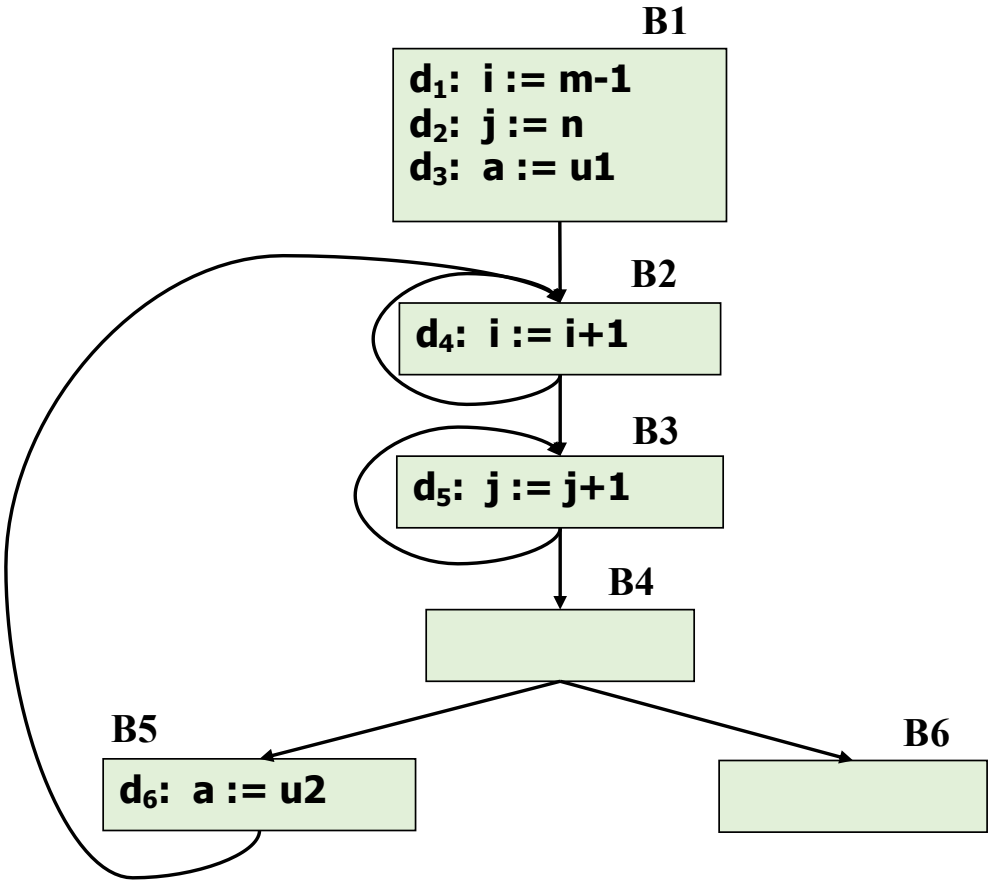
N: 节点集合

E: 边的集合

有 $a \in N, b \in N$

- $pred(b) = \{a \in N \mid \exists e \in E, e = a \rightarrow b\}$
- $succ\{(b) = \{a \in N \mid \exists e \in E, e = b \rightarrow a\}$
- a 是分支节点 $if\ succ(a) > 1$
- a 是汇合节点 $if\ pred(a) > 1$

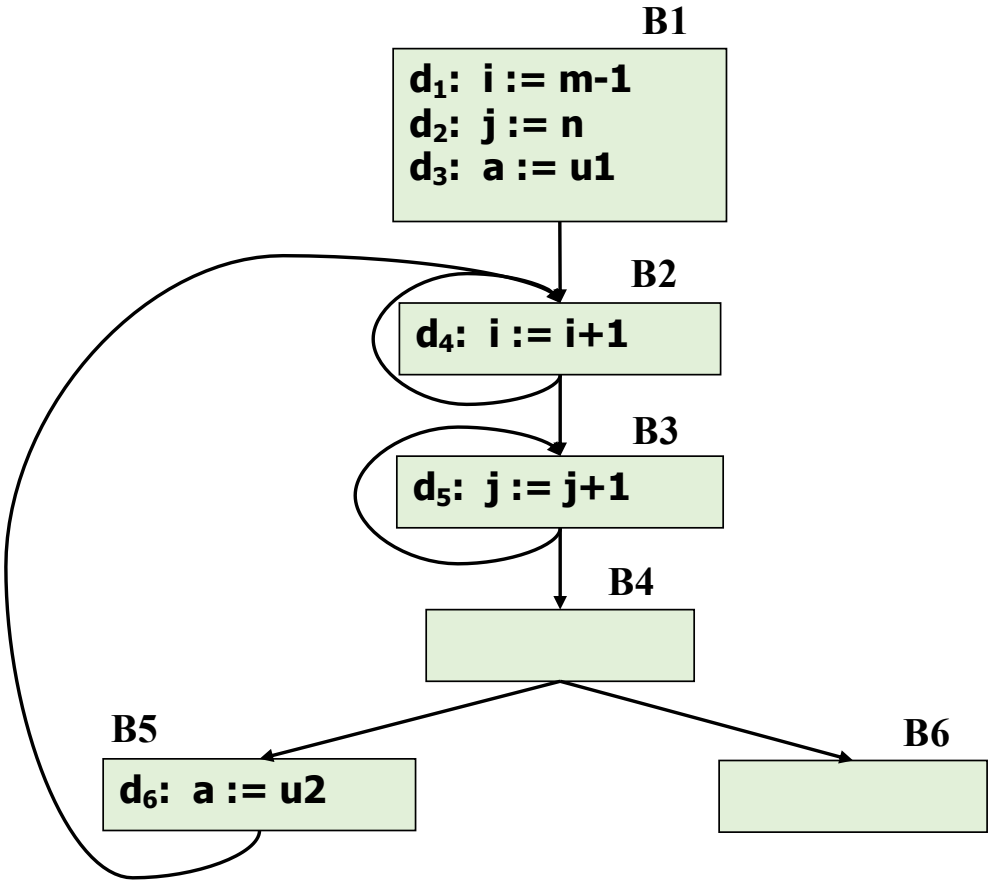
1.4 点（Point）和路径（Path）



一个基本块内的点：
- 语句之间
- 第一条语句之前
- 最后一条语句之后

第一条语句之前：基本块入口点
最后一条语句之后：基本块出口点

1.4 点（Point）和路径（Path）



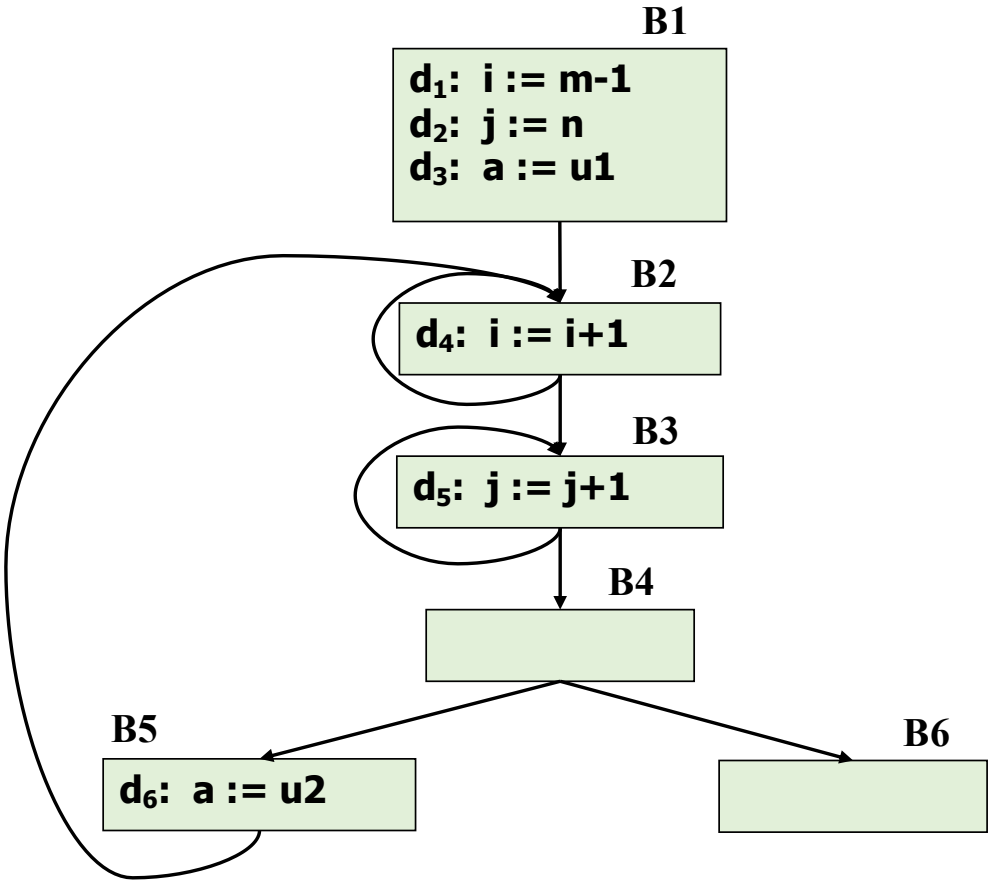
一个基本块内的点：

- 语句之间
- 第一条语句之前
- 最后一条语句之后

基本块 B1, B2, B3和B5 分别有多少个点?

B1 有4个点,
B2, B3, B5 各2个点

1.4 点（Point）和路径（Path）

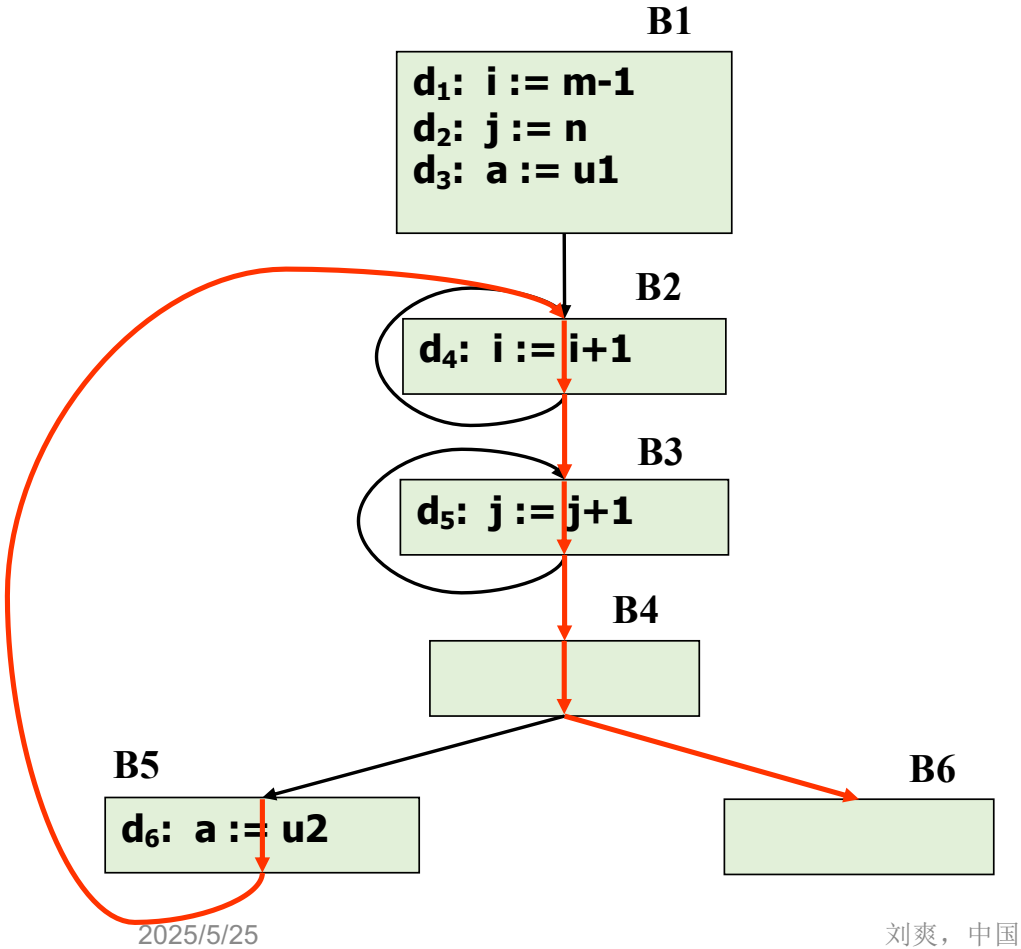


路径是点$p_1, p_2, \cdots, p_n$ 组成的序列，满足：

(i) 如果p_i 紧邻在语句S之前，p_{i+1} 紧跟在S之后

(ii) 或者p_i 是一个基本块的出口点，p_{i+1} 是后续基本块的入口点

1.4 点（Point）和路径（Path）



B5的入口点到 B6的入口点是否存在一条路径?

存在。从B5入口点开始，经过B5所有点，到B5的出口点，到B2, B3和B4所有点，到达B6入口点

2 必经节点

- 也称为支配节点

CFG中，如果从入口节点到节点b的每一条路径都经过节点a，则称a是b的必经节点。

b的所有必经节点构成b的必经节点集合$\mathrm{dom}(b)$

必经节点关系是一种偏序关系：

1. 自反的，$a \in \mathrm{dom}(a)$
2. 传递的，if $a\ dom\ b$ & $b\ dom\ c$，则$a\ dom\ c$
3. 反对称的，if $a\ dom\ b$ & $b\ dom\ a$，则$a = b$

2.1 必经关系

1. 如果从入口节点s到b的每一条路径，都包括a，

   a 是b的必经节点，记作 $a \le b$ 或 $a\ dom(b)$

2. 如果$a \le b$ 且 $a \ne b$

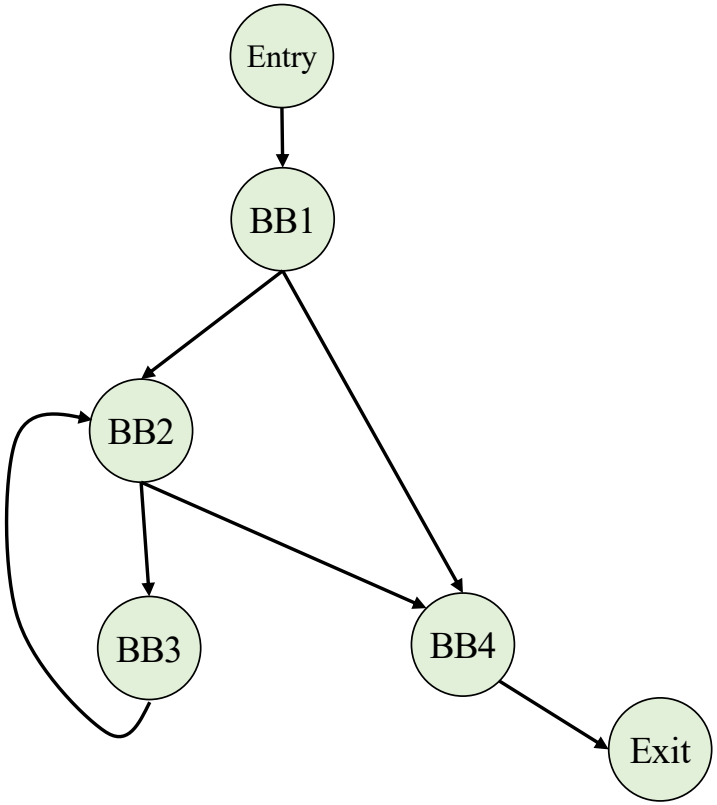
   a是b的严格必经节点， $a < b$ 或 $a\ sdom(b)$

3. 如果$a<b$ 且不存在一个节点 c $\in$N，满足 a<c< b

   a是b的直接必经节点, $a <_i b$ ， 也记作$a\ idom\ (b)$

4. 直接必经节点是唯一的

2.1 必经关系



dom(Entry) = {Entry}

dom(BB1) = {Entry,BB1}

dom(BB2) = {Entry, BB1,BB2}

dom(BB3) = {Entry, BB1, BB2, BB3}

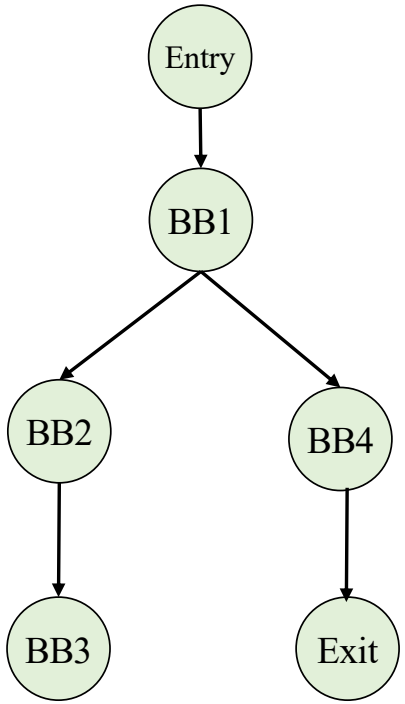
dom(BB4) = {Entry, BB1, BB4}

dom(Exit） = {Entry, BB1,BB4,Exit}

2.2 必经节点树

- 必经节点树包含CFG所有节点，每个节点*a*的直接必经节点到*a*有一条有向边

  - 入口节点Entry是根
  - 每个节点是其后代的必经节点



必经节点树

2.3 计算必经节点集合

- 基本思想
  - 节点b是b的必经节点。
  - 如果a是b的唯一前驱，则a是b的必经节点
  - 如果a是b所有前驱的必经节点，则a是b的必经节点

$$\forall p \in pred(b), \text{if } a \le p, \text{则 } a \le b$$

2.3 计算必经节点集合

- 求节点a的必经节点集合dom(a)
  - 对于入口节点，dom(s) = {s}
  - 节点$a \neq s$，dom(a) = {a} ∪ (∩$p \in pred(a)$ dom(p))

```
dom(s) = { s }
for n ∈ N –{ s } do
  dom(n) = N
repeat
  changed = false
  for n ∈ N – {s} {
    olddom = dom(n)
    dom(n) = {n} ∪ ∩p∈pred(n) dom(p)
     if dom(n) ≠ olddom then changed = true
  }
until (changed = false)
```

计算复杂性 : O(N^2)

2.4 计算直接必经节点

- 求节点a的直接必经节点idom(*a*)

```
for (a ∈ N)
  idom (a) = dom(a) – {a}

for a ∈ N –{Entry} {
        for d ∈ idom(a) {
                for s∈ idom(a)-{d} {
                if ( s∈ idom(d) )
                        idom(a) = idom(a) – {s}
                }
        }
 }
```

3 循环

- 构建必经节点目的之一:找出循环
  - 循环占据了SPEC CPU程序90%以上的执行时间

- 什么是自然循环?
  - 在控制流图中，自然循环是满足下列条件的节点结合S:
    - S包括一个头节点h
    - 对S中任意节点a，都有一条有向边组成的路径从a到h
    - 从h到S中任意节点，都存在一条路径
    - 从S之外的任意节点到达S中的节点，都必须经过h

3 循环

- CFG中的强连通部分

  循环是控制流图$G = (N, E)$的强连通子图$G' = (N', E')$，在这个子图中，每个节点都可以通过一条路径到达另一个任意节点

- 循环的另一个定义

  循环是控制流图$G = (N, E)$的强连通子图$G' = (N', E', h)$，其中，h是所有N'中节点的必经节点。

3.1 找出循环

- 利用回边发现循环
  - CFG中，如果节点a <b，边$b \rightarrow a$称为回边
- 循环
  - 给定一条回边$b \rightarrow a$，和它关联的循环是指以a为必经节点且能够到达b的所有节点构成的集合，a为头节点
- 找出循环过程

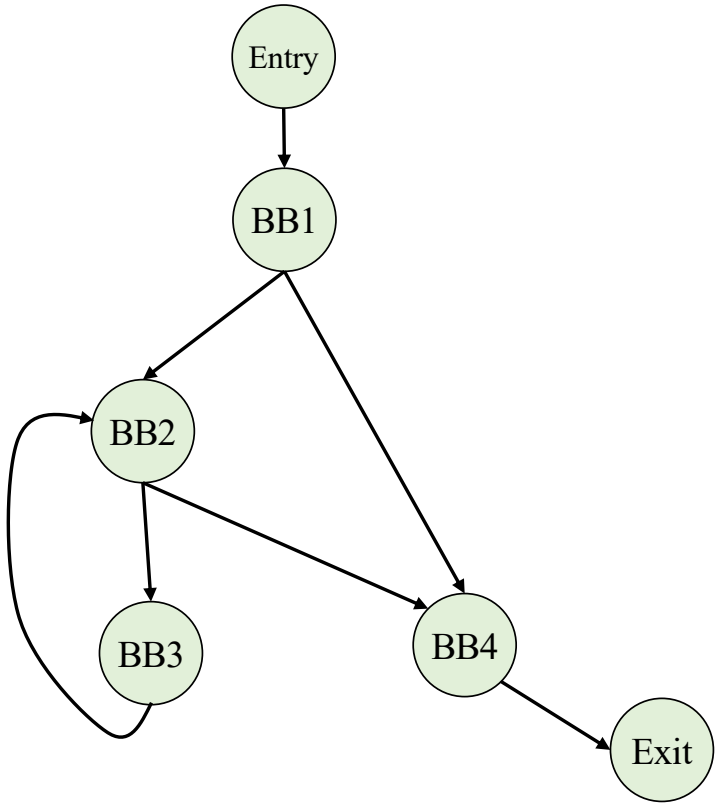
1) 找出回边$b \rightarrow a$

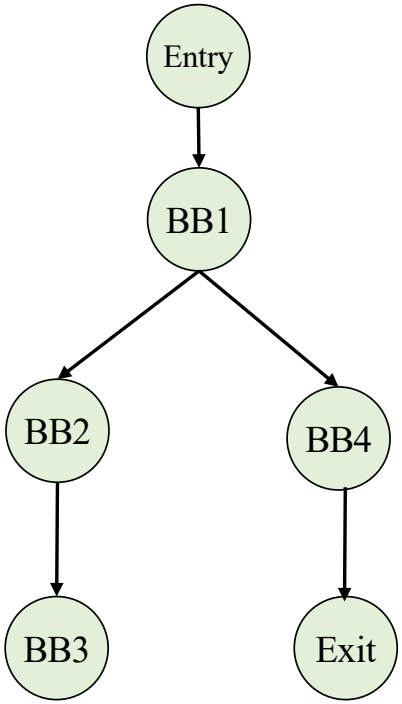
2) 找出以a为严格必经节点的所有节点

3) 从2)选出能够到达b的节点

3.1 找出循环

找出回边*BB3→BB2*





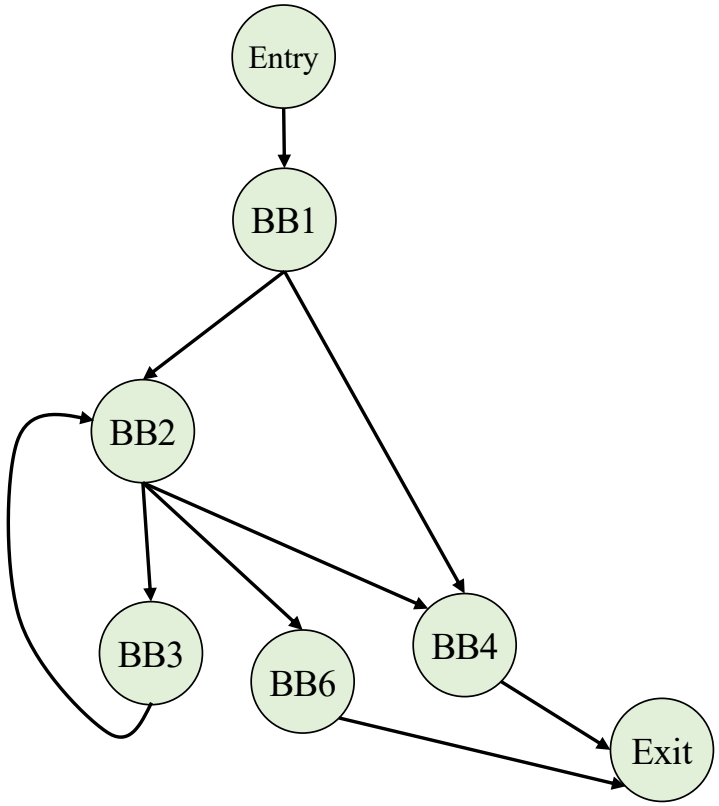
必经节点树

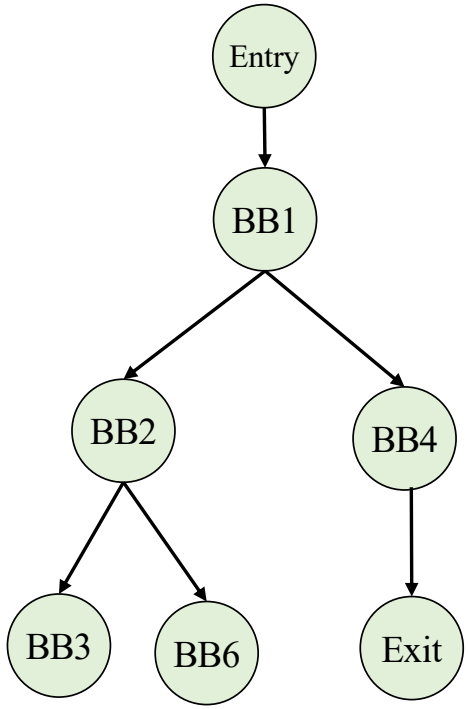
以BB2为严格必经节点的节点：
BB3 （以BB2为根的子树）

循环L={BB2，BB3}

3.1 找出循环

找出回边*BB3→BB2*





必经节点树

以BB2为严格必经节点的节点：BB3、BB6 （以BB2为根的子树）

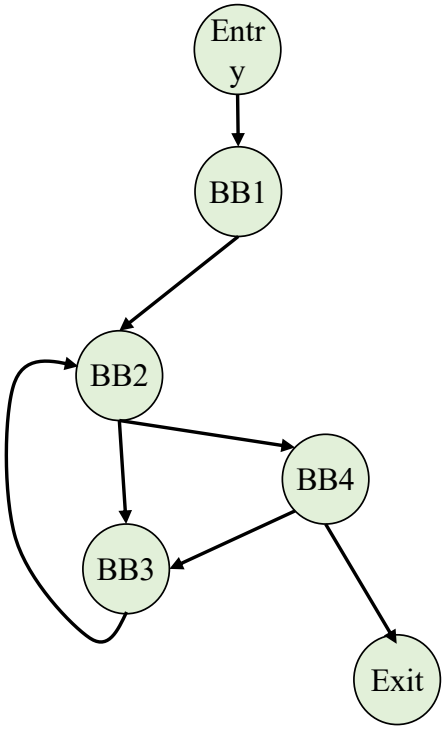
但是从BB6没有路径到达BB3

循环L={BB2，BB3}

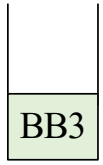
3.1 找出循环

```
procedure insert(d){
  if(d ∉ loop) {
    loop = loop ∪ {d};
    将d压入stack中;
  }
}
```

```
回边b→a
loop = {a};     // a加入到循环，它是头节点
insert(b);      //将b加入到循环
while (stack 非空){
  从stack弹出第一个元素m;
  for m的每个前驱p;
    insert(p); 将p加入到循环
}
```

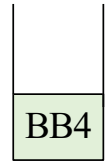


回边：BB3→BB2
loop: {BB2}
insert(BB3), loop={BB2,BB3}



stack

弹出BB3
BB3前驱BB2，BB4
insert(BB4) loop={BB2,BB3,BB4}

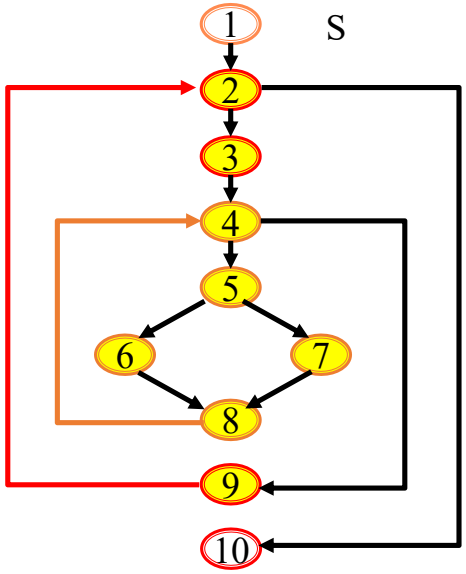


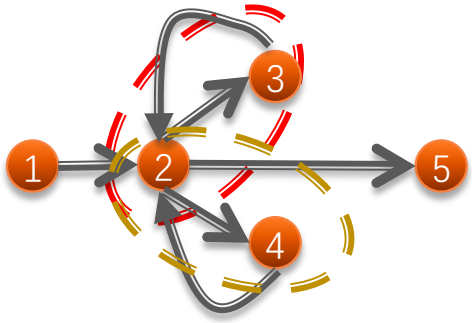
stack

弹出BB4
BB4前驱BB2
loop={BB2,BB3,BB4}

3.2 嵌套循环

- 假设A 和 B是循环，头节点分别为 a、b
  - $a \neq b$
  - $b \in$ A
- 则
  - B是A的真子集
  - B是A的嵌套循环，即B是A的内层循环
- 思考?



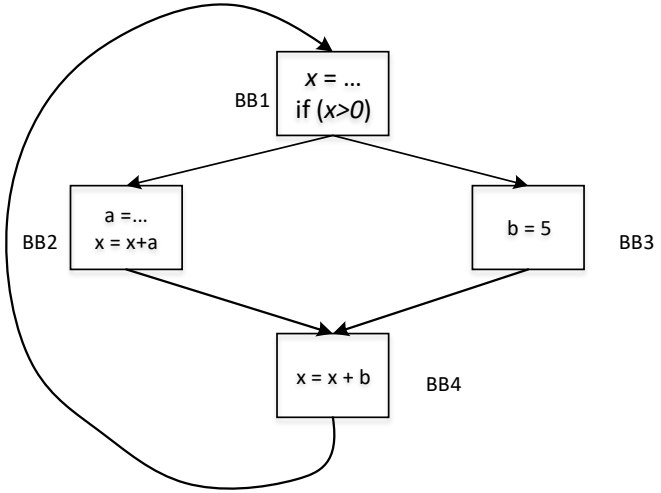


哪一个是内层循环?如何处理

优化方法之一: 循环合并

二、数据流分析

- 控制流图（CFG）中没有包含程序数据信息



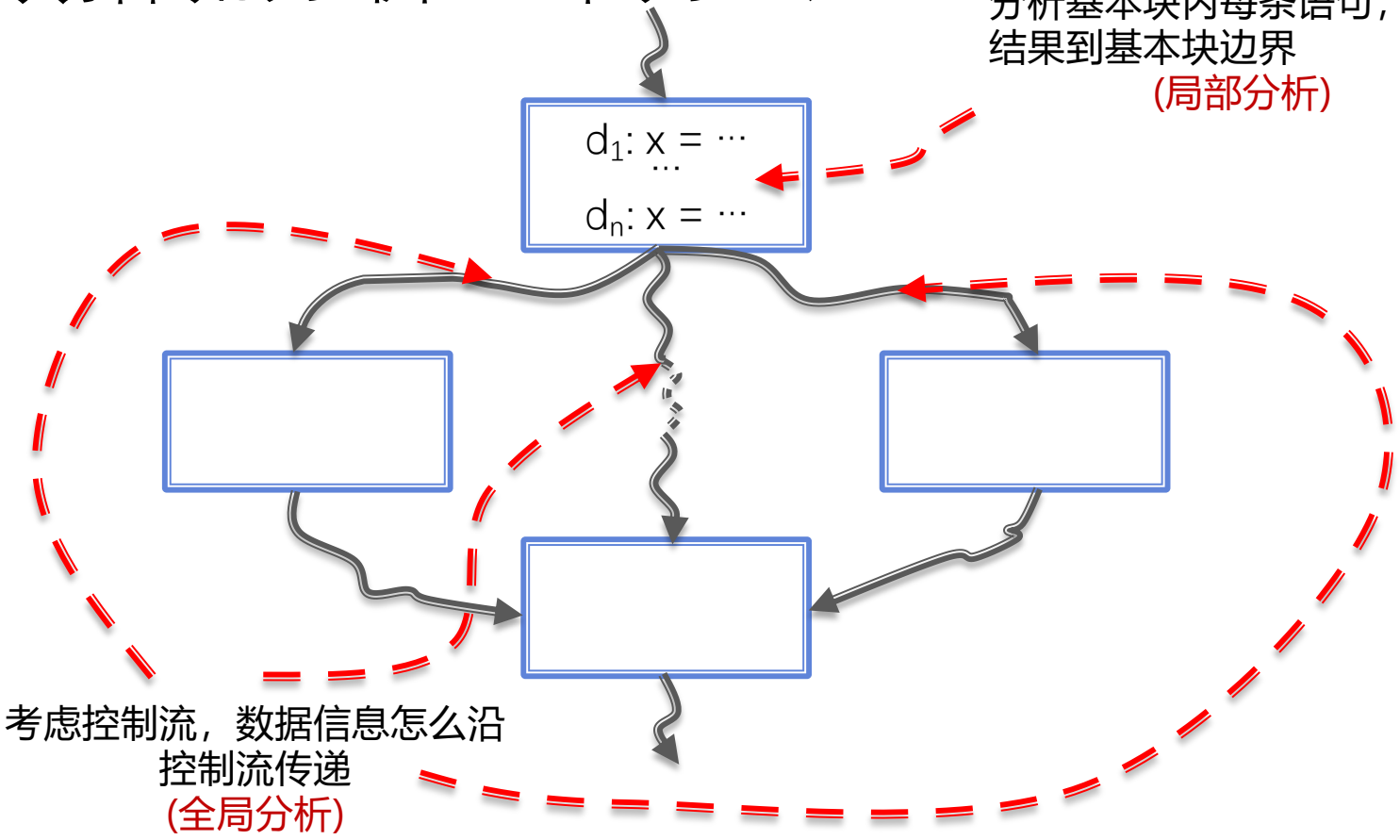
- 基本块BB2定值了变量a，除了BB2，a是否还会被其它基本块使用?
- 在BB4入口b是否是一个常数?

- 数据流分析（Data Flow Analysis，DFA）
  - 获取和分析程序中数据的相关信息，获取数据流信息

二、数据流分析

- 数据流分析（Data Flow Analysis，DFA）
  - 和控制流一样，是后续程序分析和编译优化的基础
- 数据信息和程序的控制转移相关
  - 流-敏感的分析
- 基于数据流分析，完成
  - 冗余优化
    - 公用表达式删除
    - 无用代码删除 ……
  - 寄存器分配

2.1 数据流分析基本方法



2.1 数据流分析基本方法

- 数据流分析主要有三种分析方法
  - 迭代数据流分析
  - 基于区间的数据流分析
  - 结构分析

2.2 到达-定值分析

- 定值和使用
  - 每个赋值就是一次定值

S_k: $V_1 = V_2 + V_3$

- S_k 对变量V_1进行定值
- S_k 使用 V_2 and V_3

2.2.1 到达-定值

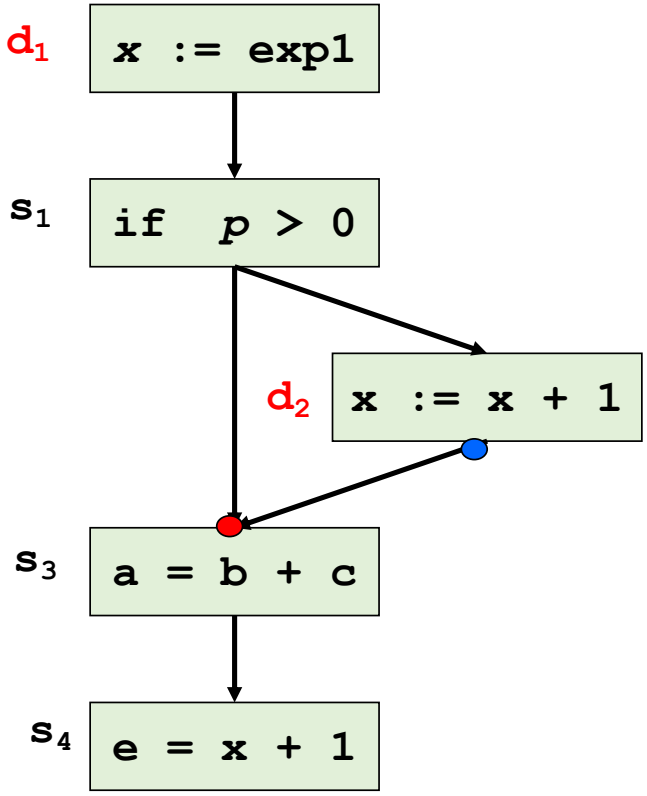
- 到达-定值（Reaching Definitions）
  - 变量*x*有一次定值d，如果存在一条从定值点开始的路径到达点p，并且沿着这条路径，定值d没有被其他定值杀死，那么就说定值d达到点p

- **到达-定值分析**
  - 对每个基本块，分析求解到达基本块边界（入口点和出口点）的所有定值

2.2.1 到达-定值

d_1 到达 ●

d_1 不能到达 ●，被定值d_2杀死



2.2.2 到达-定值分析方法

- **gen(B)**：基本块B内的定值集合，并且集合内定值能够到达基本块B的出口点

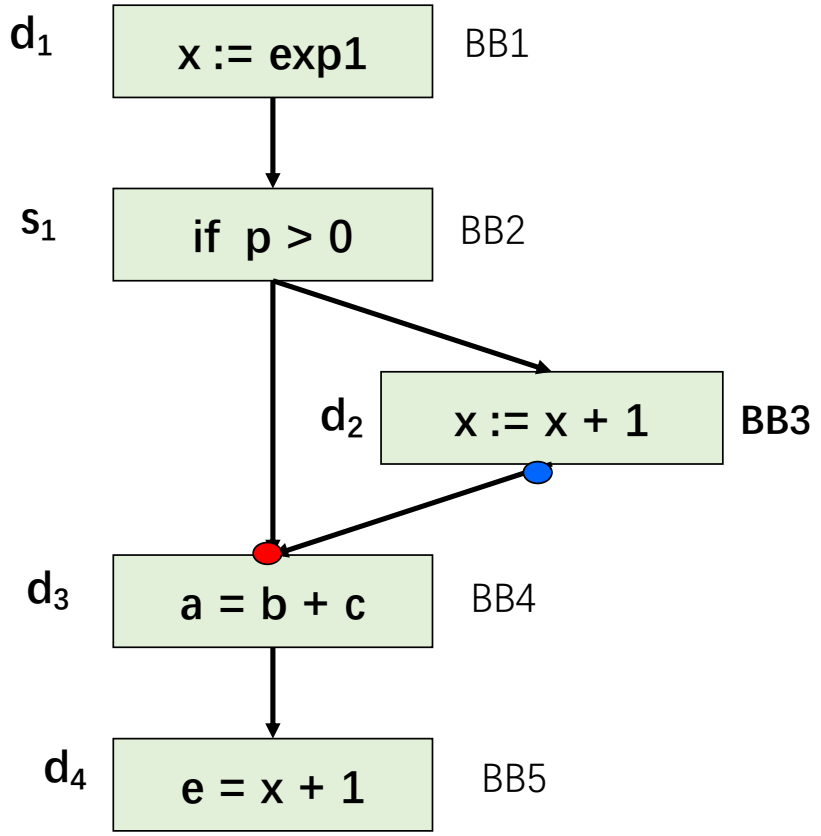
$$gen[B1]=\{d_1\}$$

$$gen[B4]=\{d_3\}$$

- **kill(B)**：被基本块B杀死的在其他基本块内的定值集合
  - 每一个在B内定值的变量*v*，杀死的定值包括所有其他基本块内对 *v* 的定值

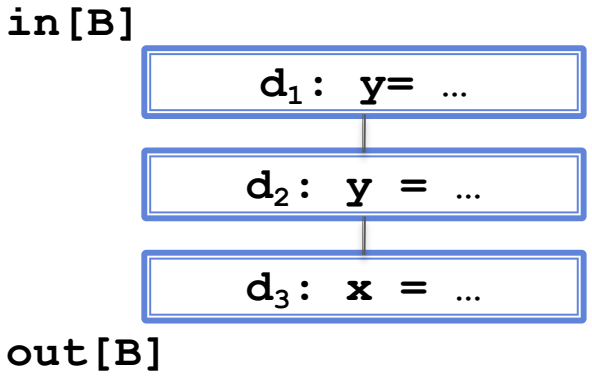
$$kill[B3]=\{d_1\}$$

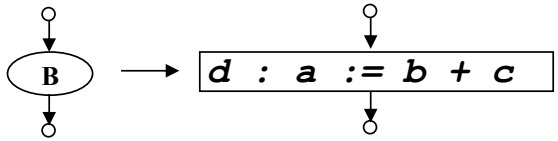
$$Kill[B4]=\varnothing$$



2.2.2 到达-定值分析方法

- 第一步，建立数据流方程
  - 基本块内（局部分析）

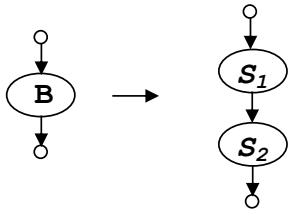




$$out[B] = f_s(in[B])$$
$$= gen[B] \cup (in[B] - kill[B])$$

2.2.2 到达-定值分析方法

局部分析



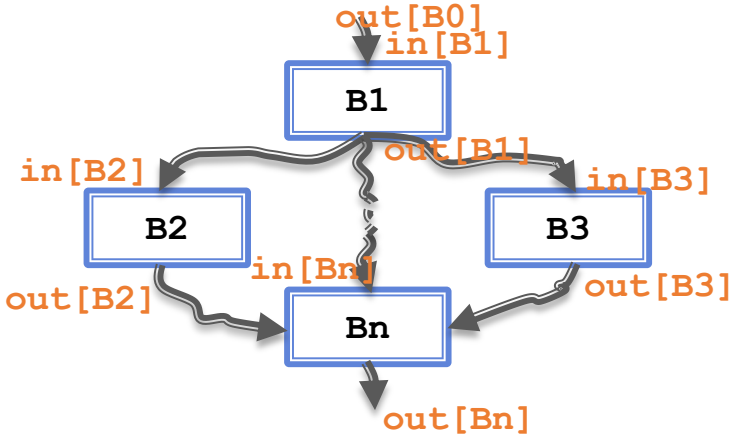
$$in[S_1] = in[B]$$
$$in[S_2] = out[S_1]$$
$$out[B] = out[S_2]$$

$$\begin{aligned} out[B] &= f_B(in[B]) \\ &= f_{S2}(f_{S1}(in[B])) \\ &= gen[B] \cup (in[B] - kill[B]) \end{aligned}$$

基本块入口点的到达定值集合为in[B]，经过该基本块后，到达基本块出口点的定值包括基本块生成的定值，以及属于in[B]并且没有被基本块B杀死的定值

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

2.2.2 到达-定值分析方法




```
in[B2] = out[B1]
out[B2] = gen[B2]U(in[B2]- kill[B2] )
```

```
in[Bn] = out[B1]Uout[B2] U … =
```

$$\bigcup_{P_i \in pred[B_n]} out[P_i]$$

```
out[Bn] = gen[Bn]U(in[Bn]-kill[Bn])
```

2.2.2 到达-定值分析方法

$$in[B] = \bigcup_{P_i \in pred[B_n]} out[P_i]$$

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

控制流图中，特殊的两个空结点：**Entry、Exit**

显然有，**out[Entry] = ∅**

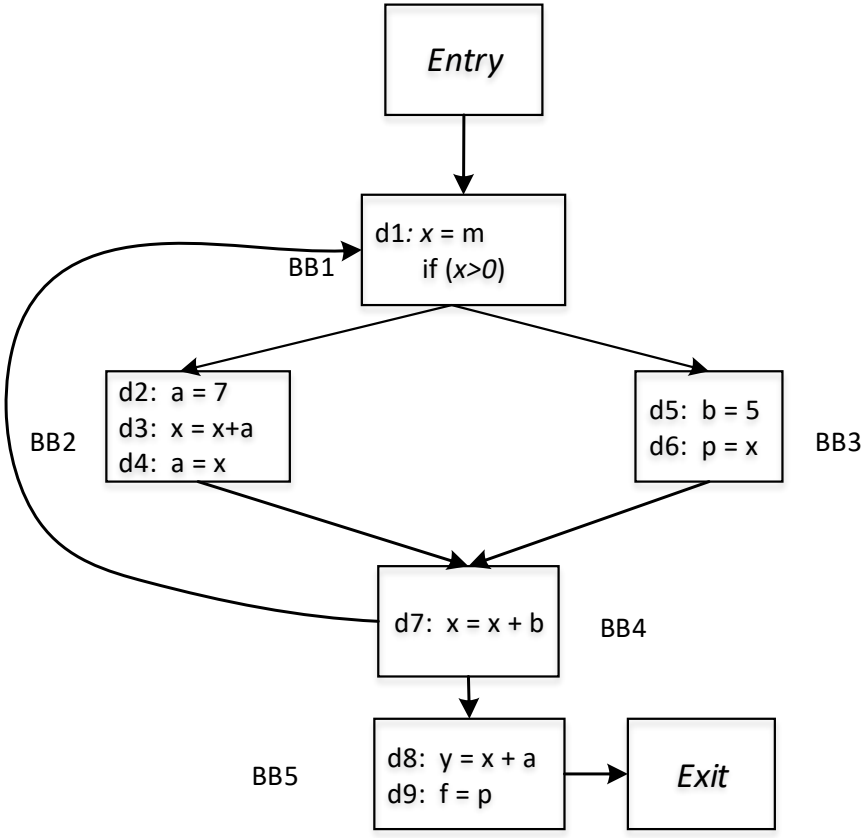
求解方程组方法：不动点方法

2.2.3 到达-定值分析方法

```
out[Entry]=∅
每个基本块B∈N-{Entry}
        out[B]=∅
change = true
while(changes) { //循环，直到所有基本块的out都不再发生变化
        change = false
        对每个基本块B∈N-{Entry} {
                oldout=out[B]
                in[B]= ⋃_{P∈pred[B]} out[P]
                out[B]= gen[B]U(in[B]-kill[B])
                if(out[B]≠oldout) change=true;
        }
}
```

求解不动点

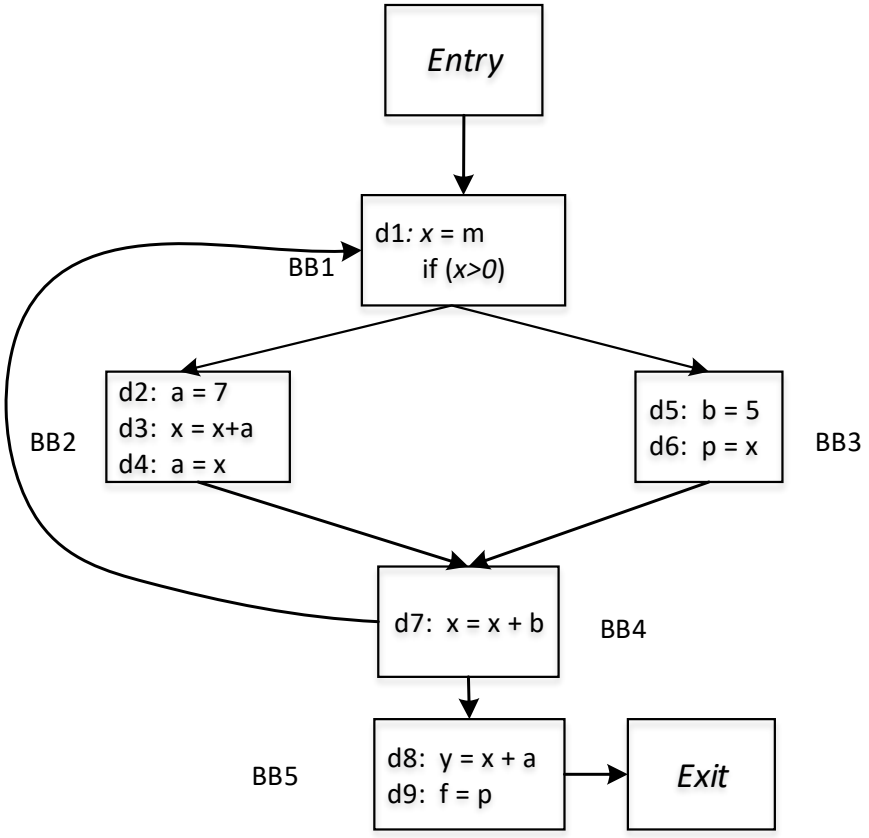
2.2.3 到达-定值分析示例



第一步: 计算每个基本块的gen和 kill集合

	gen	kill
BB1	{d1}	{d3, d7}
BB2	{d3, d4}	{d1,d2,d7}
BB3	{d5, d6}	∅
BB4	{d7}	{d1, d3}
BB5	{d8, d9}	∅

2.2.3 到达-定值分析示例



第二步：对每个基本块B，置初值

$out[B] = \varnothing$

第一次迭代：

$in[BB1] = \varnothing$

$out[BB1] = \{d_1\}$

$in[BB2] = \{d_1\}$

$out[BB2] = \{d_3, d_4\}$

$in[BB3] = \{d_1\}$

$out[BB3] = \{d_1, d_5, d_6\}$

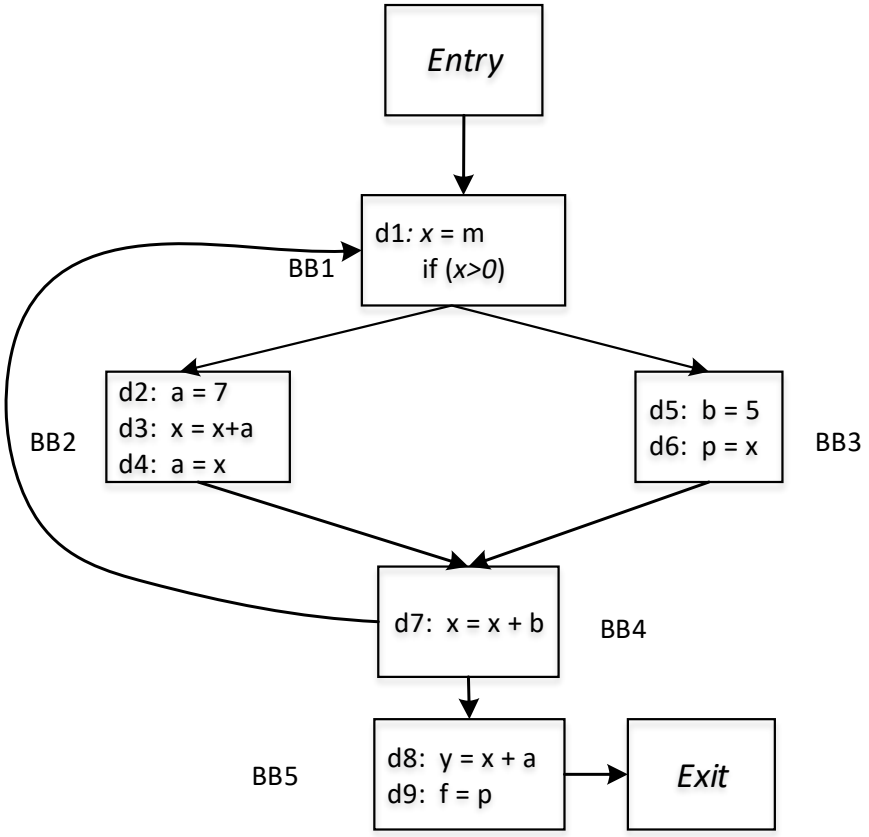
$in[BB4] = \{d_1, d_3, d_4, d_5, d_6\}$

$out[BB4] = \{d_4, d_5, d_6, d_7\}$

$in[BB5] = \{d_4, d_5, d_6, d_7\}$

$Out[BB5] = \{d_4, d_5, d_6, d_7, d_8, d_9\}$

2.2.4 到达-定值分析示例



循环，直到找到不动点：

$in[B]=\cup out[P]$　　P是B的前驱

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

$in[B1]=\{d_4,d_5,d_6,d_7\}$

$out[B1]=\{d_1,d_4,d_5,d_6\}$

$in[B2]=\{d_1,d_4,d_5,d_6\}$

$out[B2]=\{d_3,d_4,d_5,d_6\}$

$in[BB3]=\{d_1,d_4,d_5,d_6\}$

$out[BB3]=\{d_1,d_4,d_5,d_6\}$

$in[BB4]=\{d_1,d_3,d_4,d_5,d_6\}$

$out[BB4]=\{d_4,d_5,d_6,d_7\}$

$in[BB5]=\{d_4,d_5,d_6,d_7\}$

$Out[BB5]=\{d_4,d_5,d_6,d_7,d_8,d_9\}$

2.2.4 到达-定值信息存储

- 位串表示
  - 一位表示一次定值
  - 位串长度=程序中定值次数
  - 集合的操作可以高效使用“位与”或者“位或”操作实现

	第一次迭代		第二次迭代		第三次迭代	
	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
BB1	000 000 000	100 000 000	000 111 100	100 111 000	000 111 100	100 111 000
BB2	100 000 000	001 100 000	100 111 000	001 111 000	100 111 000	001 111 000
BB3	100 000 000	100 011 000	100 111 000	100 111 000	100 111 000	100 111 000
BB4	101 111 000	000 111 100	101 111 000	000 111 100	101 111 000	000 111 100
BB5	000 111 100	000 111 111	000 111 100	000 111 111	000 111 100	000 111 111

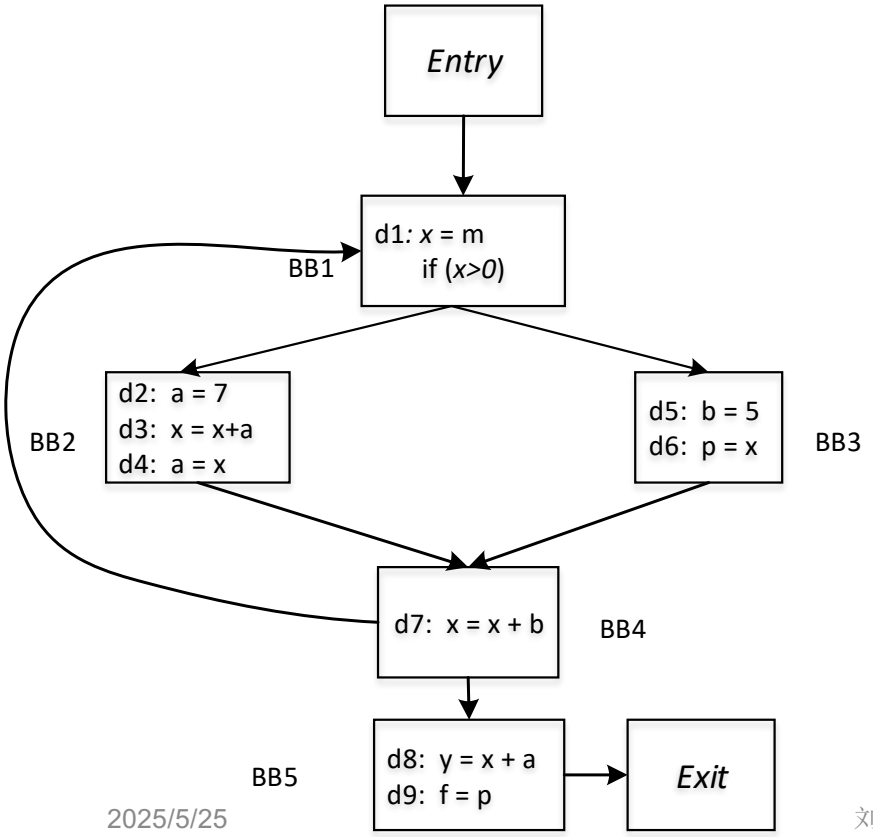
2.2.4 到达-定值信息存储

- 使用-定值链(Use-Definition Chain,UD链)
  - 一种稀疏的表示方式
  - 对变量的每一次使用，到达该使用的所有定值都保存在链表中

  如果使用在基本块B中
    - 如果B中在使用之前没有对v的定值，那么该次使用的UD链表包括in[B]中对v的所有定值
    - 否则，链表只包含一个元素，即在该次使用之前，对v的最后一次定值

2.2.4 到达-定值信息存储

- 使用-定值链(Use-Definition Chain，UD链)



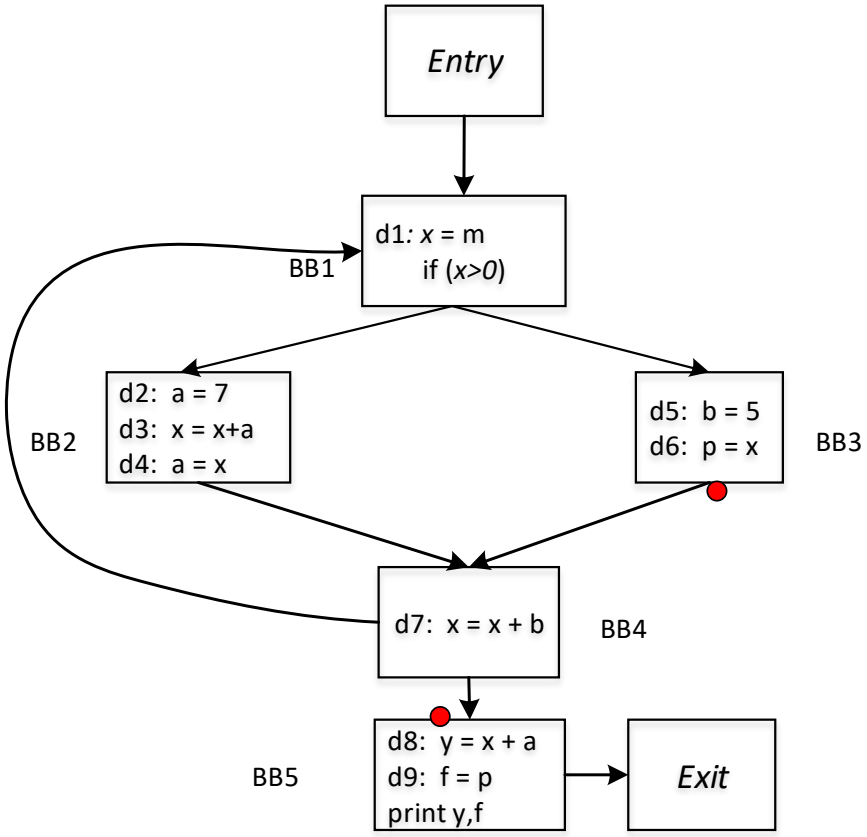
- 基本块BB4中语句d7: x = x +b，到达该x的使用的定值为d1和d3
- BB2中语句d3: x=x+a，到达a的使用的定值只有d2

3 活跃变量

- 如果一个变量 *v* 在点p开始的某条路径上被使用，那么变量*v*在点p活跃（*live*）
- 否则，变量*v*在点p是死变量

- 活跃变量分析
  - 对每个基本块B，分析获取B边界处（入口和出口点）活跃变量集合

3.1 活跃变量

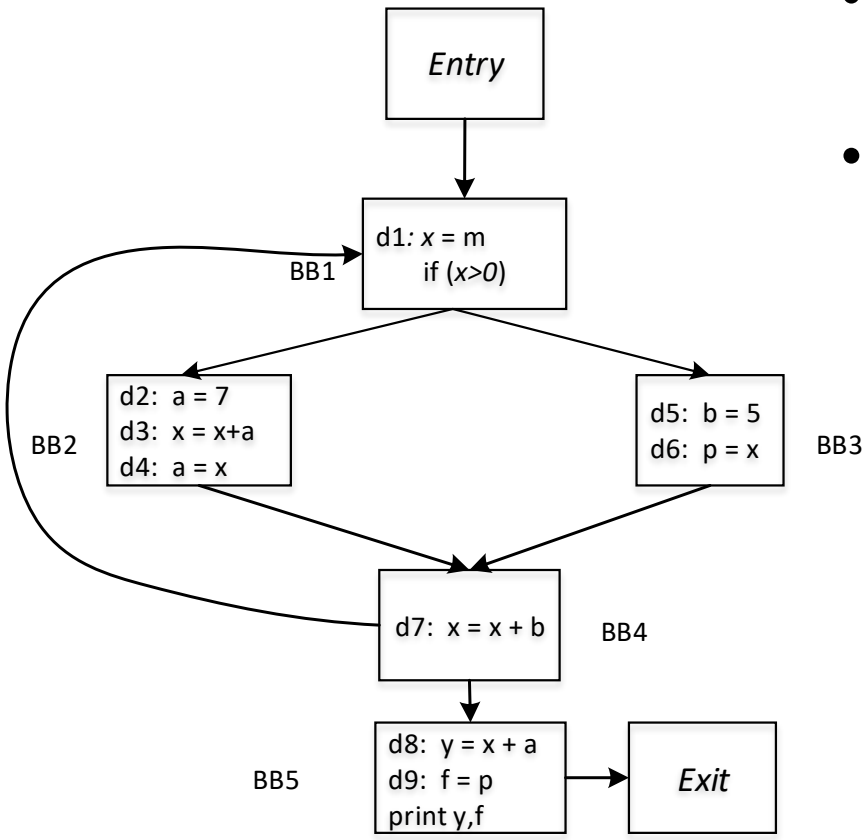


- 在BB5入口点，哪些变量是活跃的?

  入口点开始的路径，使用的变量

  x、a、p

3.2 活跃变量分析方法



- **use[B]：**基本块B内，定值之前使用的变量集合
- **def[B]：**被基本块B定值的变量集合

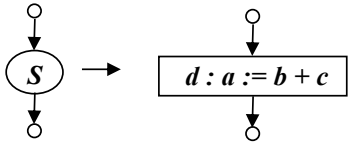
```
use[BB5] = {x, a, p}

def[BB5] = {y, f}
```

活跃变量分析计算的是从点p出发的路径对变量的使用，use[B]是使用在定值之前的变量集合，因此是后向数据流问题。

—— 后向分析

3.2 活跃变量分析方法

局部分析



后向分析：in[S] = f_S(out[S])

f_S：out[S]→in[S]

对于语句S：a = b+c

产生的活跃变量：use[S] = {b, c}

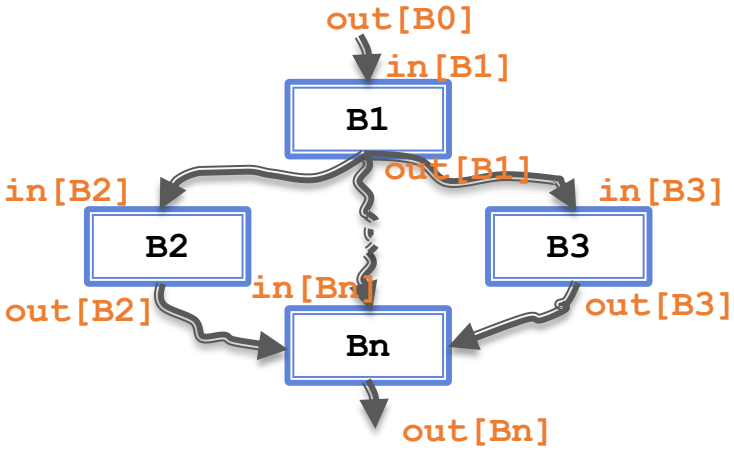
传播：out[S]-def[S]，def[S]=a

in[S]=use[S]U(out[S]-def[S])

若变量x在基本块B的出口点活跃，那么它在基本块B的入口点也活跃的条件是：要么它在基本块内没有被重新定值，要么它在基本块内先使用再定值。

in[B]=f_B(out[B])=use[B]U(out[B]-def[B])

3.2 活跃变量分析方法




```
out[B2]=in[Bn]
in[B2]=use[B2]U(out[B2]-def[B2] )
```

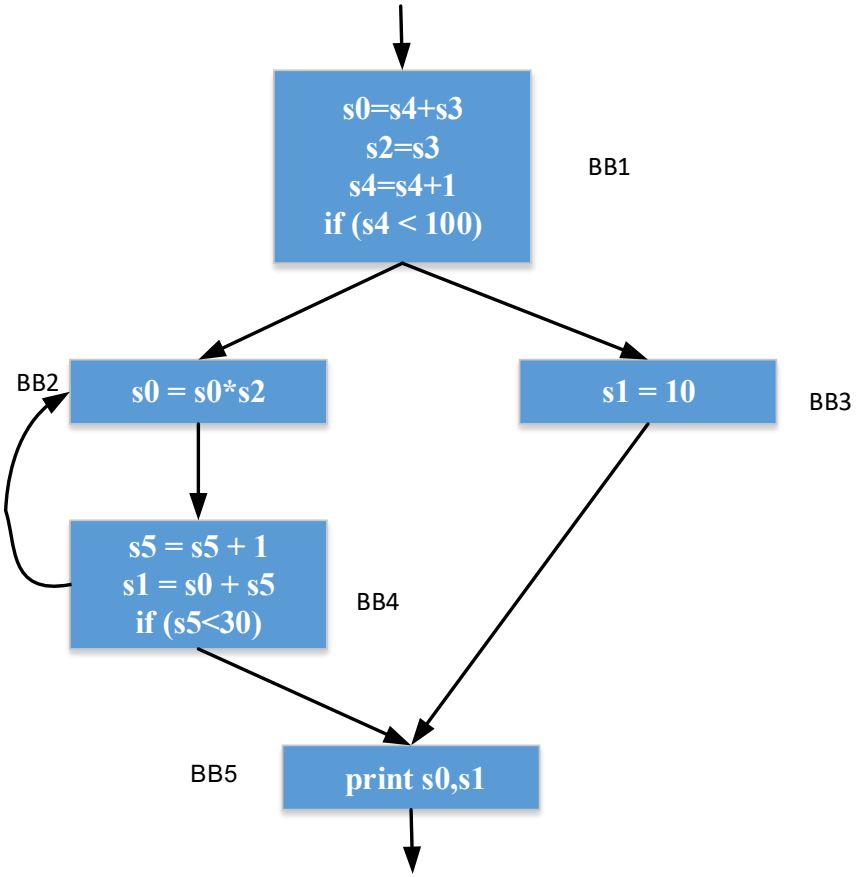
```
out[B1]=in[B1]Uin[B2]U … = ⋃ in[Si]
                           Si∈succ[B1]
in[B1]=use[B1]U(out[B1]-def[B1])
```

3.2 活跃变量分析方法

```
in[Exit]=∅
对每个基本块B∈ N - {Exit}
        in[B]=∅
change=true
while(changes) {
        change = false
        对每个基本块B∈N -{Exit}{
            oldin=in[B]
            out[B]= ⋃_{S∈succ[B]} in[S]
            in[B]=use[B]U(out[B] - def[B])
            if(in[B]≠oldin) change=true;
        }
}
```

求解不动点

3.3 活跃变量分析示例



Basic Block	use(B)	def[B]
1	{s3,s4}	{s0,s2,s4}
2	{s0,s2}	{s0}
3	∅	{s1}
4	{s5,s0}	{s1,s5}
5	{s0,s1}	∅

Basic Block	in(B)	out[B]
1	{s3,s4,s5}	{s0,s2,s5}
2	{s0,s2,s5}	{s0,s2,s5}
3	{s0}	{s0,s1}
4	{s0,s2,s5}	{s0,s1,s2,s5}
5	{s0,s1}	∅

3.4 总结

- 到达-定值（Reaching Definitions）
  - 变量*x*有一次定值d，如果存在一条从定值点开始的路径到达点p，并且沿着这条路径，定值d没有被其他定值杀死，那么就说定值d达到点p

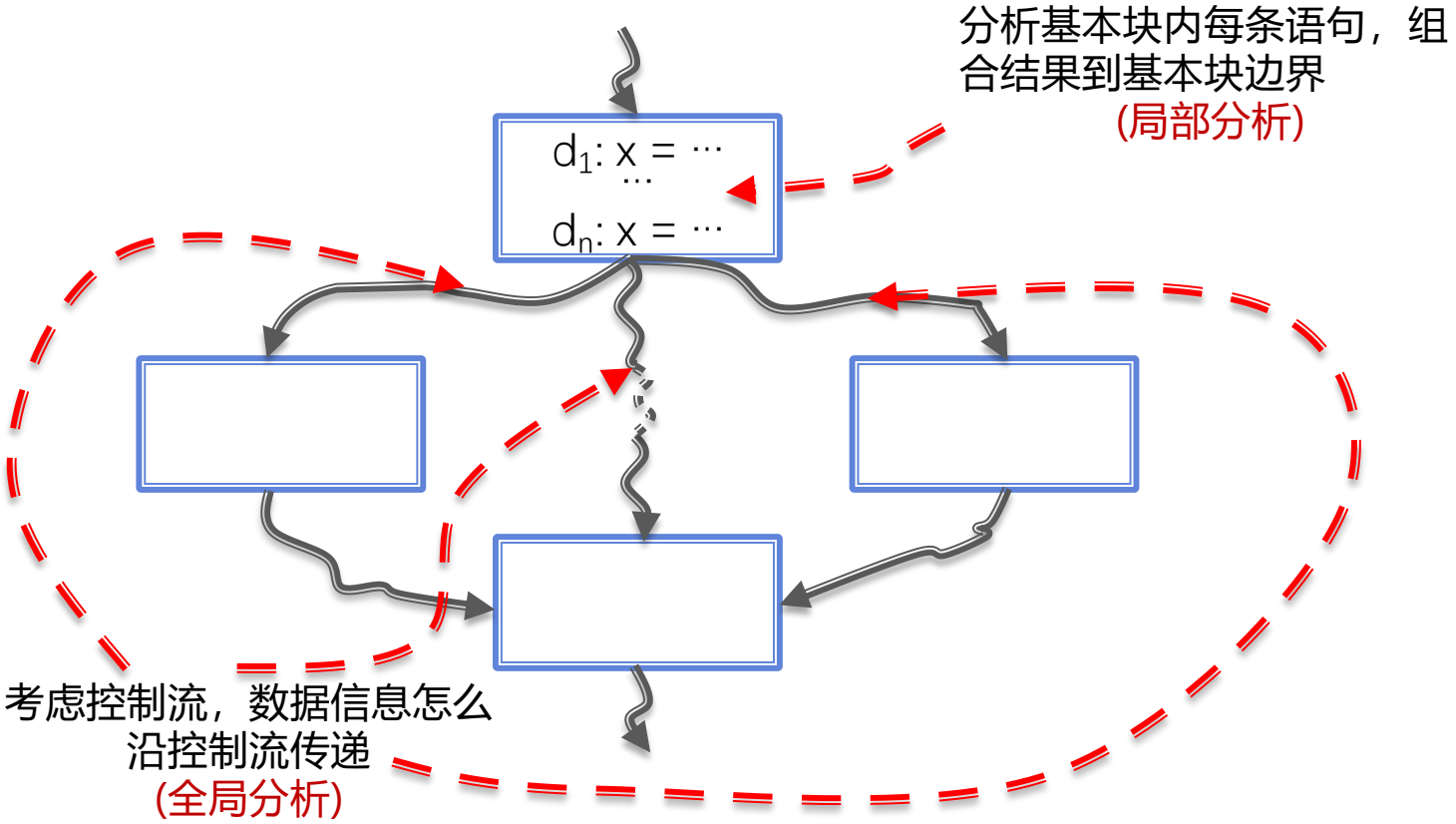
- 活跃变量（Live Variables）
  - 如果一个变量 *v* 在点p开始的某条路径上被使用，那么变量*v*在点p是活跃（*live*）

3.4 总结： 前向分析和后向分析

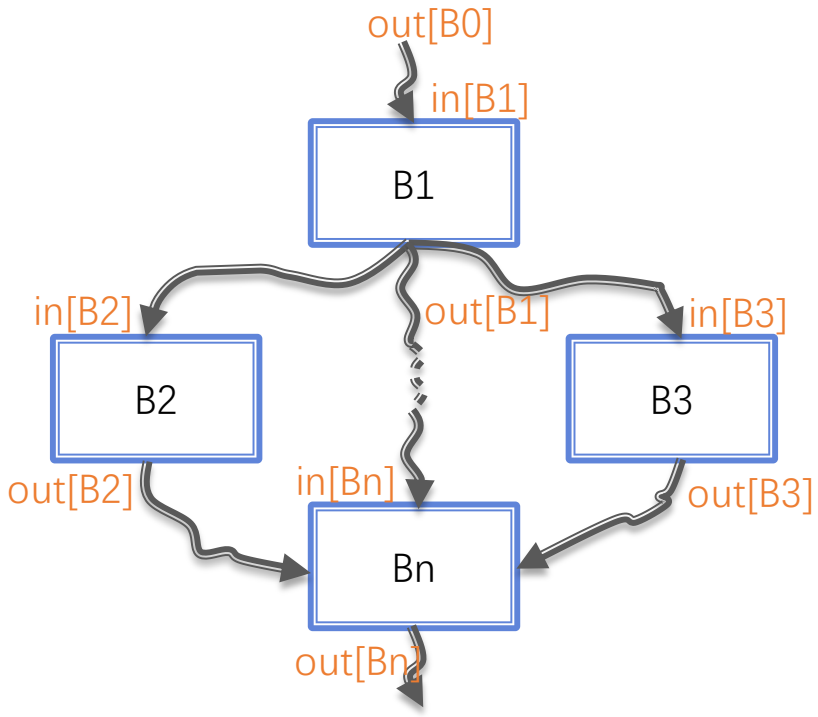
- 前向分析： 在点p计算过去的行为产生的数据流信息
  - 计算CFG中前驱
  - 例如 : 到达-定值分析

- 后向分析： 在点p计算未来行为的数据流信息
  - 计算 CFG中的后继
  - 例如: 活跃变量分析

4. 迭代的数据流分析框架



4、迭代的数据流分析框架



数据流方程的建立依赖于程序语句对求解的数据问题的影响

- 建立一组方程，对于每个基本块，求解出口点数据流信息集合out[B]和入口点数据流信息集合in[B]：
  - 基本块内语句（代码）对求解的数据流问题的影响　转移函数

  f_B: in[B]→out[B]/out[B] → in[B]
  - 控制流作用：

    如果B1、B2有边连接，out[B1]和in[B2]之间的关系
- 寻找方法求解方程组（迭代求不动点）

4、迭代的数据流分析框架

- 迭代的通用数据流分析框架:
  - 一个数据流图，包含两个特殊的结点Entry和Exit
  - 数据流方向D
  - 一个值集V
  - 一个meet操作∧
  - 一个传递函数的集合F，fB表示基本块B的传递函数
  - V的一个常量值Ventry或Vexit，分别表示前向和逆向框架的边界条件
- 输出: 每个基本块边界处V的值

4、迭代的数据流分析框架

- 前向的数据流分析

```
out[Entry] = V_entry;
for (each B∈N-{Entry})
    out[B] = T //初值
while(change to any out occur){
    for(each B∈N-Entry){
        in[B] = ∧_{p∈pred[B]} out[P];
        out[B] = f_B(in[B]);
    }
}
```

4、迭代的数据流分析框架

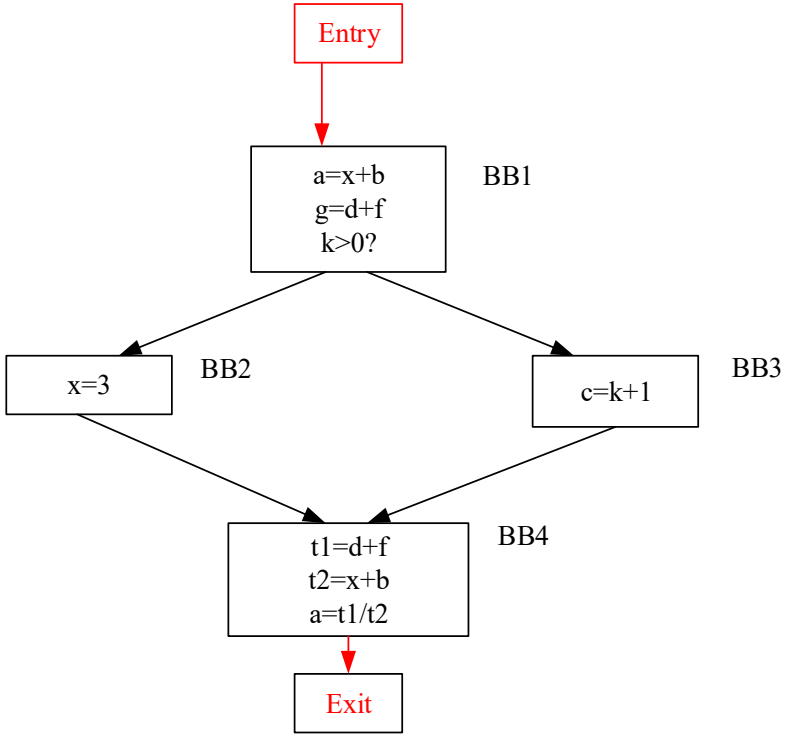
- 后向的数据流分析

```
in[Exit] = Vexit;
for (each B∈N-{Exit})
    in[B] = T //初值
while(change to any in occur){
    for(each B∈N-Exit){
       out[B] = ∧s∈succ[B]in[S];
       in[B] = f_B(out[B]);
    }
}
```

5.1 可用表达式

- 如果一个表达式x⊗y在点p满足下面的条件，我们就说该表达式在点p可用：
  - 从entry到p的每一条路径都计算x⊗y
  - 在到达p之前最后一次计算x⊗y后，没有对x或者y的赋值
- 可用表达式分析就是找出每个基本块B的边界处所有可用的表达式
- 可用表达式的结果用于公用子表达式的删除。

5.1 可用表达式

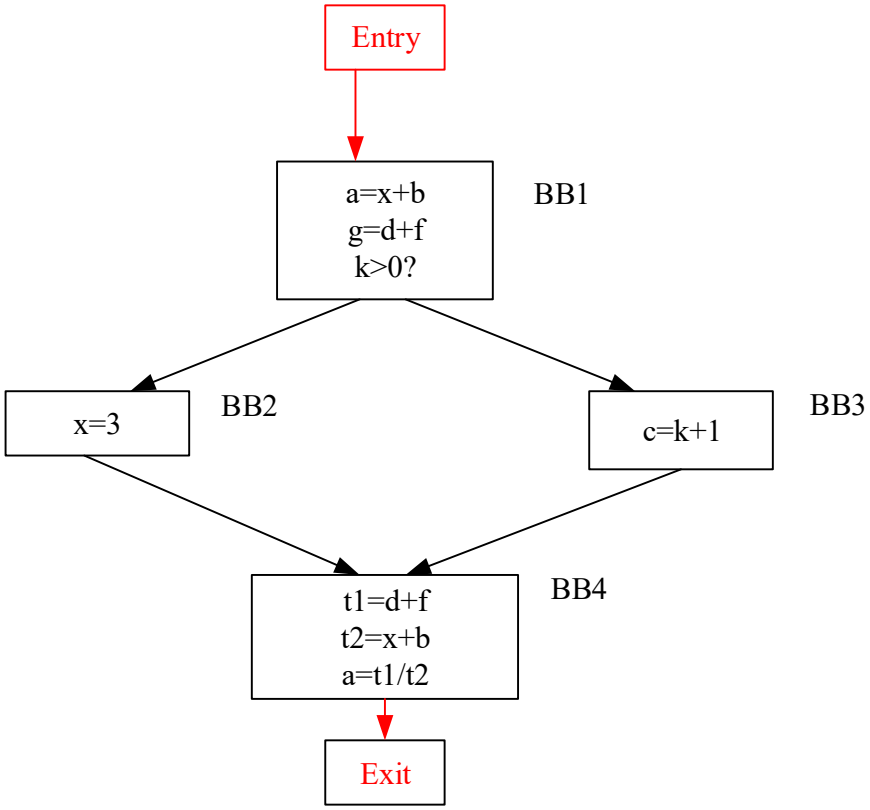


- 可用表达式分析需要定义基本块的两个集合:
  - eval(B):在基本块中计算的、并且在基本块的出口点仍可用的表达式集合
  - kill(B):被基本块B杀死的表达式集合

$$in[B] = \bigcap_{p \in pred[B]} out[p]$$

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

5.1 可用表达式



- in[B]：B的入口点的可用表达式的集合
- out[B]：B的出口点可用表达式的集合
- 可用表达式寻找的是从Start结点到点p的可用表达式，因此是一种前向数据流分析问题。

$$in[B] = \bigcap_{p \in pred[B]} out[p]$$

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

6. 静态单一赋值

- 静态单一赋值，
  - 每一个变量在程序的上下文中只有一次赋值

```
i = j;
m=i+y;
i=p;
K=i+j;
```

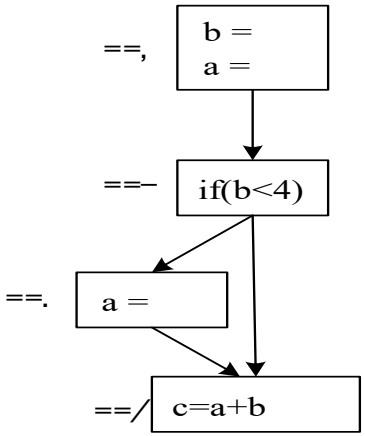
```
i1 = j;
m=i1+y;
i2=p;
K=i2+j;
```

  - 在程序执行过程中，变量可以被动态地定值多次。例如，在循环中。因此，称为静态单一赋值。

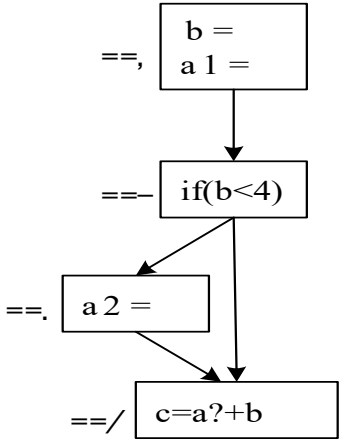
6. 静态单一赋值

- 和UD链相比，SSA具有以下优势：
  - 变量只有一次定值，数据流分析和优化算法更加简单；
  - 定值-使用关系更清晰
    - 源程序中不相关的使用在SSA中变成了不同变量，删除了它们之间不必要的联系
  - SSA比稀疏的链更紧凑
    - 一个变量的UD链需要的空间定值次数乘以使用次数
    - 程序SSA的大小几乎和原程序成线性关系；
  - SSA和控制流图的必经节点有着密切联系
    - “定值（节点）是使用（节点）的必经节点”，能够简化编译分析和优化

6. 静态单一赋值

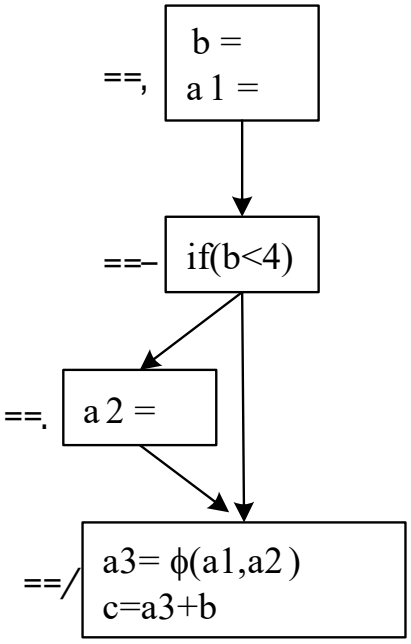


- 我们是否可以把它转换为SSA形式？应该如何转换？



- 当两条控制流边汇合在一起时，在汇合点（BB4），变量a到底使用的那个版本呢？

6. 静态单一赋值



- 引入一个虚构符号ϕ(函数），ϕ函数的参数是到达该使用的变量版本，函数根据到达的路径，返回路径上对应的变量版本。

$$\phi(a1, a2) = \begin{cases} a1 & \text{如果从B2达到B4} \\ a2 & \text{如果从B3到达B4} \end{cases}$$

6. 静态单一赋值

- 如何插入ϕ函数
  - 路径汇合准则
- 必经节点边界准则（支配边界）
- 进一步的阅读
  - Efficiently computing static single assignment form and the control dependence graph
  - Practical Improvements to the Construction and Destruction of Static Single Assignment Form