**重点：**

* 理解3种数据流分析的含义，如何设计类似的算法，如何优化
* 理解种数据流分析的共性与区别
* 理解迭代算法并弄懂算法为什么能停止

## 1.Reaching Definitions Analysis (may analysis)

**问题定义**：给变量v一个定义d（赋值），存在一条路径使得程序点p能够到达q，且在这个过程中不能改变v的赋值。

**应用举例**：检测未定义的变量，若v可达p且v没有被定义，则为未定义的变量。

**抽象表示**：设程序有n条赋值语句，用n位向量来表示能reach与不能reach。

什么是definition？ D: v = x op y 类似于赋值。

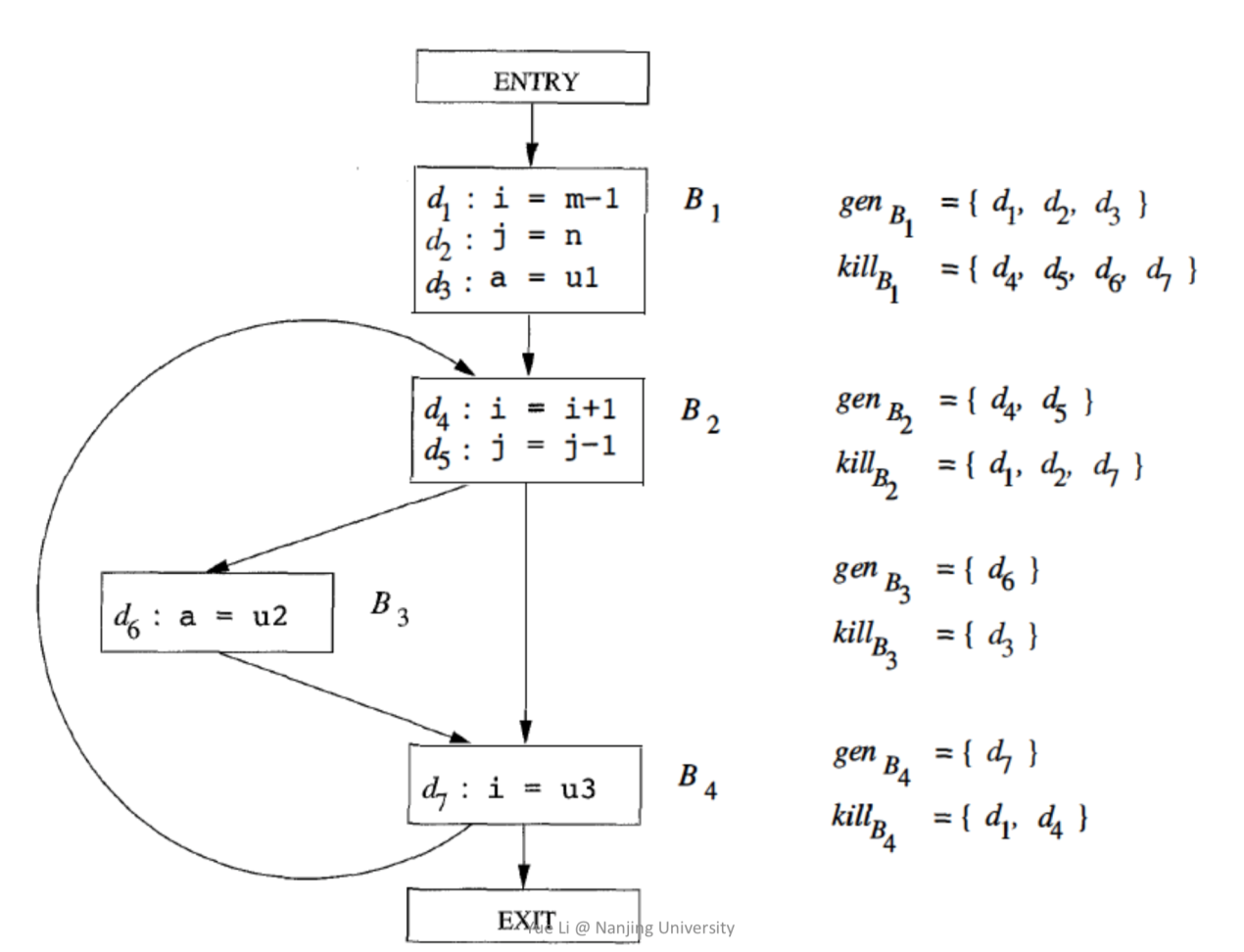
**Transfer Function**：OUT[B] = genB U (IN[B] - killB)

**解释**：基本块B的输出 = 块B内的所有变量v的定义（赋值/修改）语句 U （块B的输入 - 程序中其它所有定义了变量v的语句）。本质就是本块与前驱修改变量的语句 作用之和（去掉前驱的重复修改语句）。



怎么理解，就是基于控制流而得到。

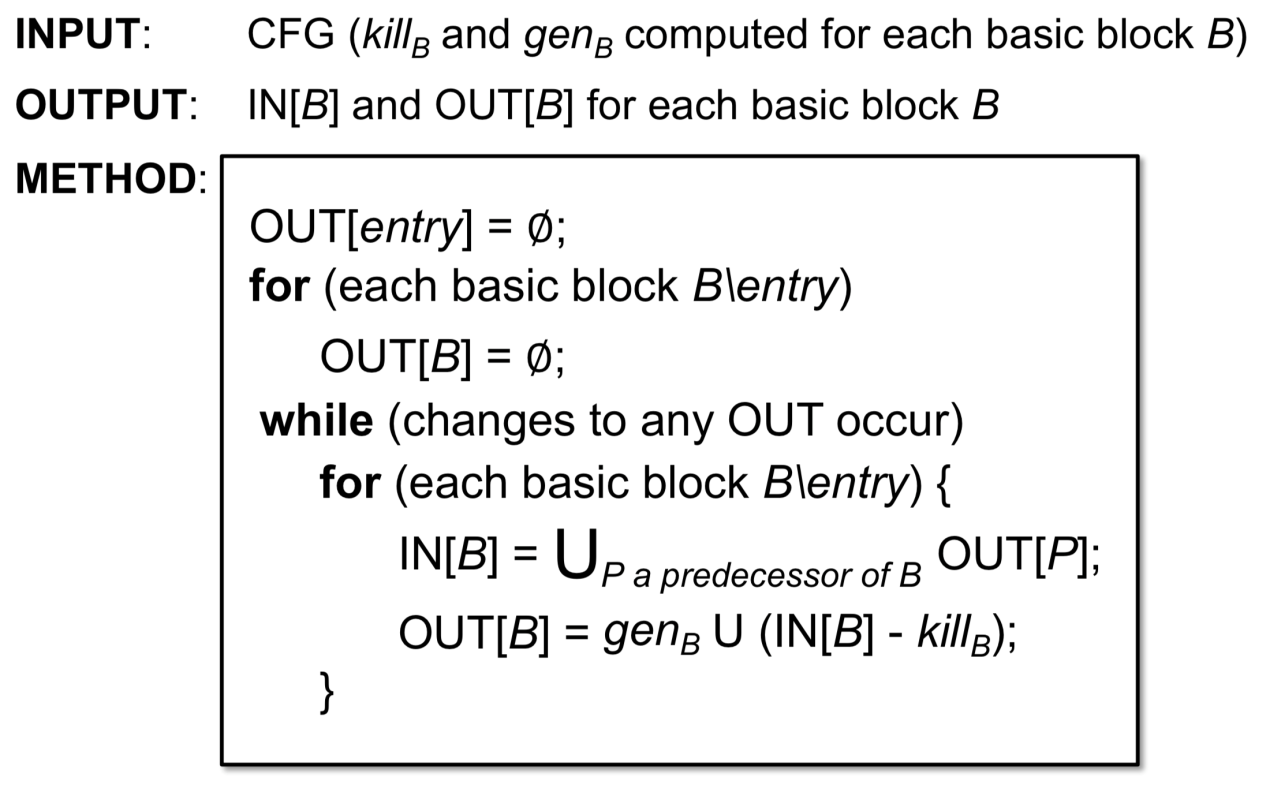
**解释**：基本块B的输入 = 块B所有前驱块P的输出的并集。注意，所有前驱块意味着只要有一条路径能够到达块B，就是它的前驱，包括条件跳转与无条件跳转。



#### （2）算法

**目的**：输入CFG，计算好每个基本块的killB（程序中其它块中定义了变量v的语句）和genB（块B内的所有变量v的定义语句），输出每个基本块的IN[B]和OUT[B]。

**方法**：首先所有基本块的OUT[B]初始化为空。遍历每一个基本块B，按以上两个公式计算块B的IN[B]和OUT[B]，只要这次遍历时有某个块的OUT[B]发生变化，则重新遍历一次（因为程序中有循环存在，只要某块的OUT[B]变了，就意味着后继块的IN[B]变了）。

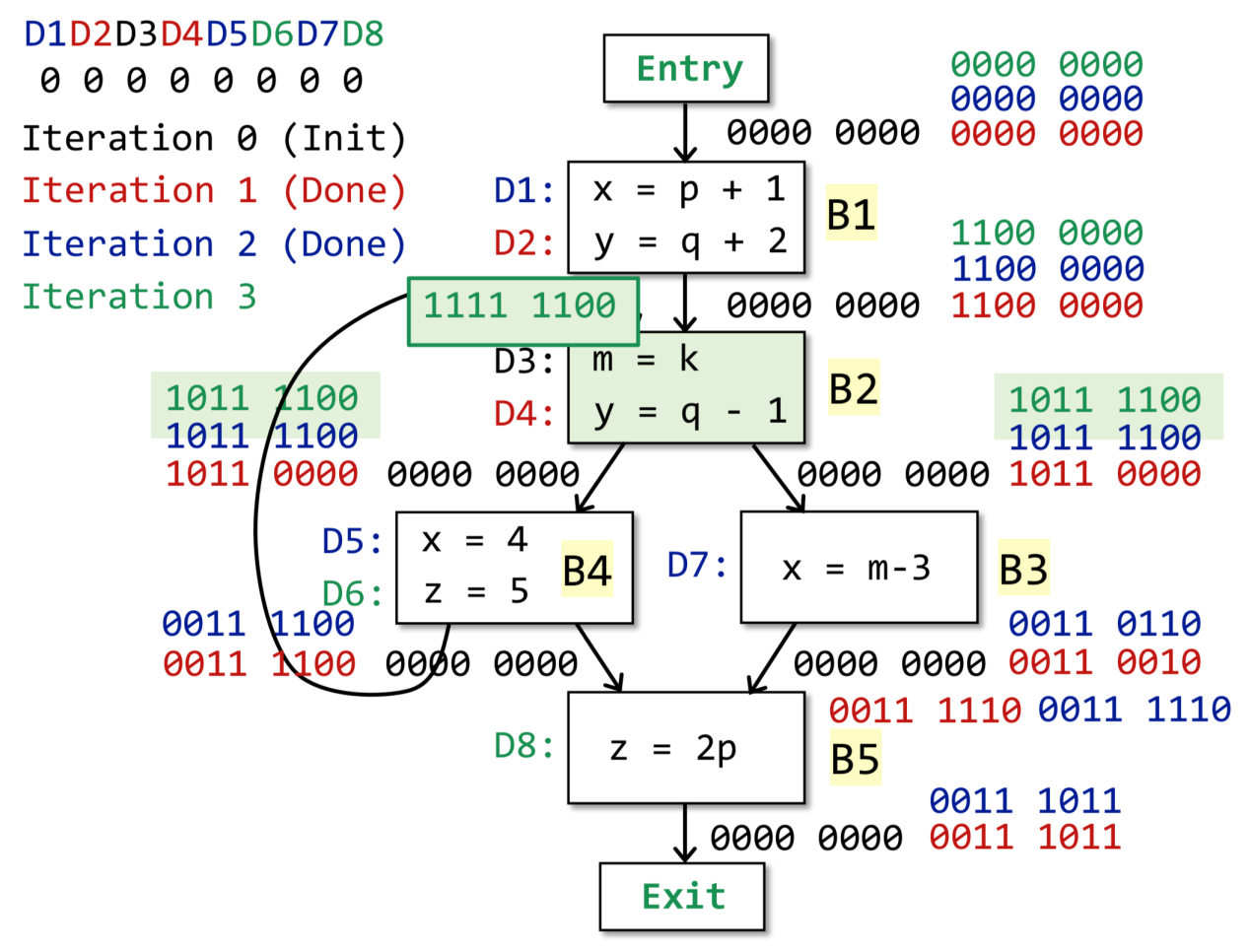


#### （3）实例：

**抽象表示**：设程序有n条赋值语句，用n位向量来表示能reach与不能reach。

**说明**：红色-第1次遍历；蓝色-第2次遍历；绿色-第3次遍历。

**结果**：3次遍历之后，每个基本块的OUT[B]都不再变化。



现在，我们可以回想一下，数据流分析的目标是，最后得到了，每个程序点关联一个数据流值（该点所有可能的程序状态的一个抽象表示，也就是这个n位向量）。在这个过程中，我们对个基本块，不断利用基于转换规则的语义（也就是transfer functions，构成基本块的语句集）-OUT[B]、控制流的约束-IN[B]，最终得到一个稳定的安全的近似约束集。

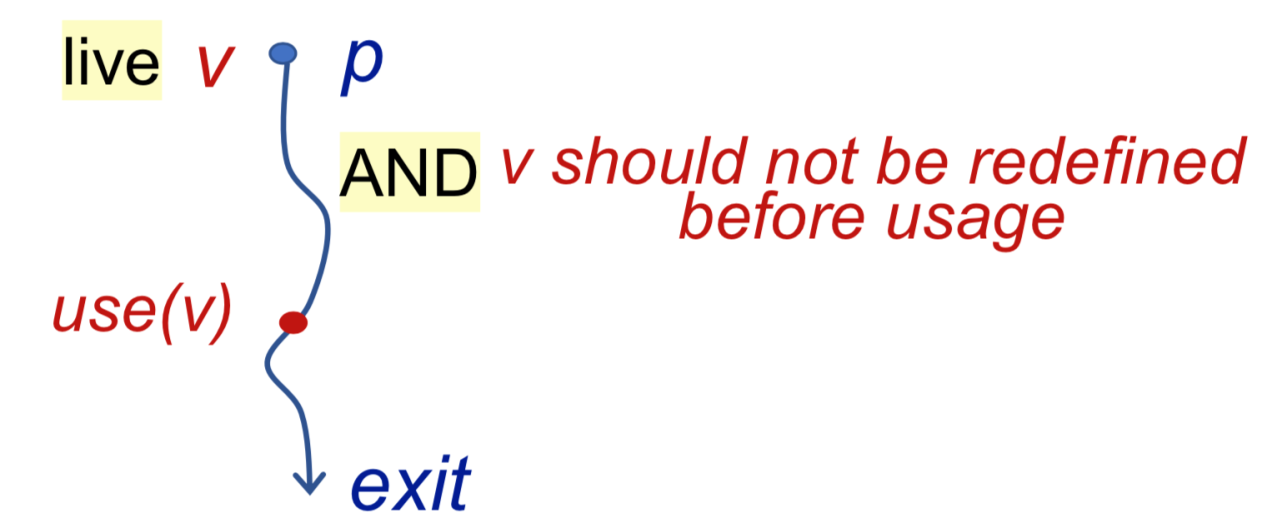
#### （4）算法会停止吗？

OUT[B] = genB U (IN[B] - killB)

**大致理解**：genB和 killB是不变的，只有IN[B]在变化，所以说OUT[B]只会增加不会减少，n向量长度是有限的，所以最终肯定会停止。具体涉及到不动点证明，后续课程会讲解。

## 4.Live Variables Analysis (may analysis)

**问题定义**：某程序点p处的变量v，从p开始到exit块的CFG中是否有某条路径用到了v，如果用到了v，则v在p点为live，否则为dead。其中有一个隐含条件，在点p和引用点之间不能重定义v。



**应用场景**：可用于寄存器分配，如果寄存器满了，就需要替换掉不会被用到的变量。

**抽象表示**：程序中的n个变量用长度为n bit的向量来表示，对应bit为1，则该变量为live，反之为0则为dead。

#### （1）公式分析

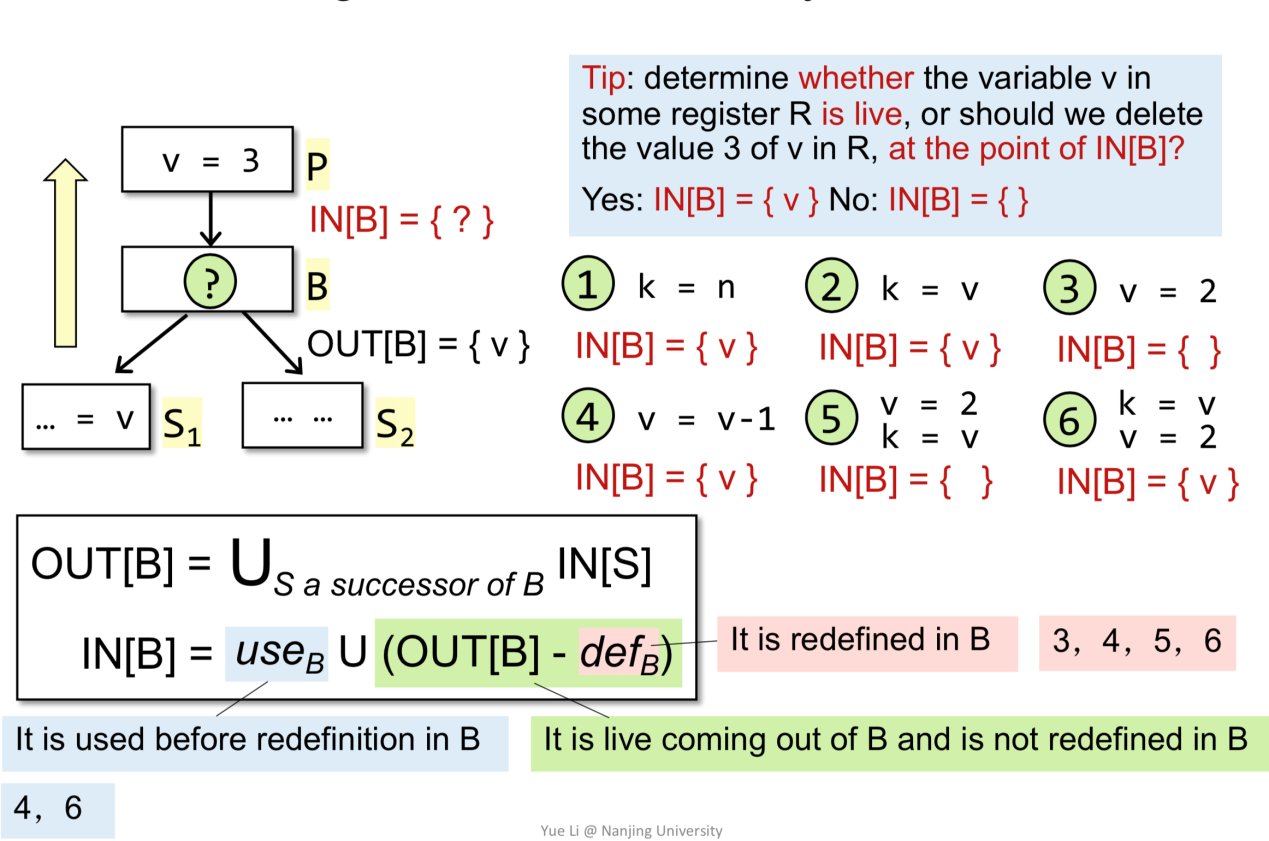


**理解**：我们是前向分析，只要有一条子路是live，父节点就是live。

**Transfer Function**：IN[B] = useB U (OUT[B] - defB)

**理解**：IN[B] = 本块中use出现在define之前的变量 U （OUT[B]出口的live情况 - 本块中出现了define的变量）。define指的是定义/赋值。

**特例分析**：如以下图所示，第4种情况，v=v-1，实际上use出现在define之前，v是使用的。

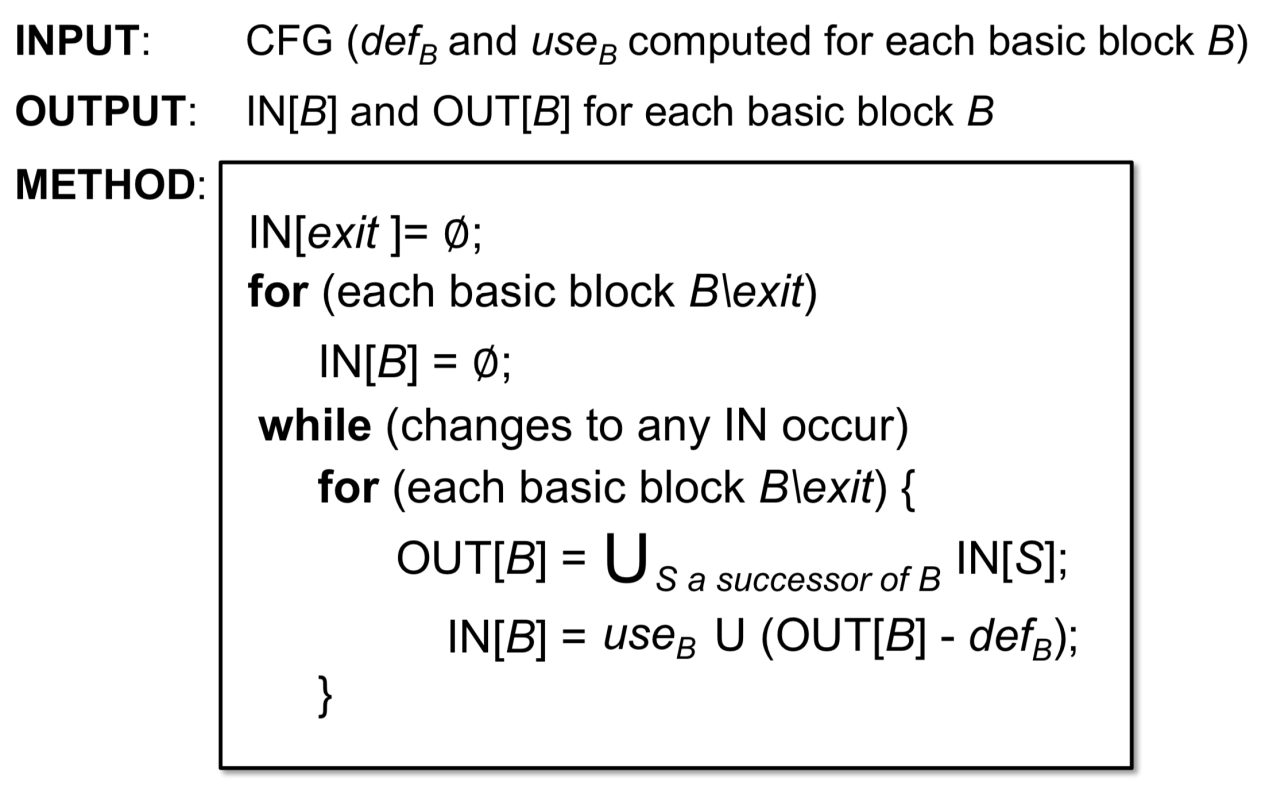


#### （2）算法

**目的**：输入CFG，计算好每个基本块中的defB（重定义）和useB（出现在重定义之前的使用）。输出每个基本块的IN[B]和OUT[B]。

**方法**：首先初始化每个基本块的IN[B]为空集。遍历每一个基本块B，按以上两个公式计算块B的OUT[B]和IN[B]，只要这次遍历时有某个块的IN[B]发生变化，则重新遍历一次（因为有循环，只要某块的IN[B]变了，就意味前驱块的OUT[B]变了）。

**问题**：遍历基本块的顺序有要求吗？ 没有要求，但是会影响遍历的次数。



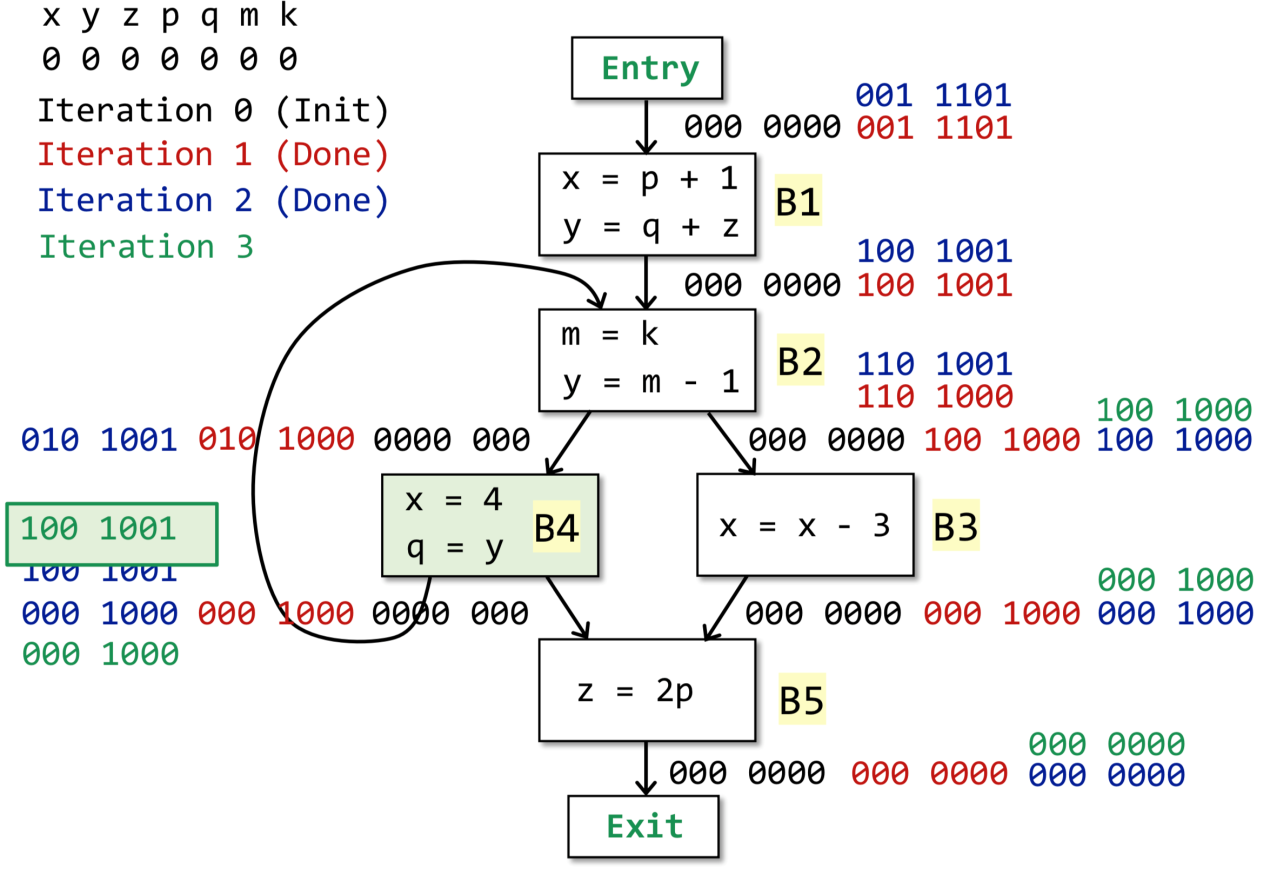
**初始化规律**：一般情况下，may analysis 全部初始化为空，must analysis全部初始化为all。

#### （3）实例

**抽象表示**：程序中的n个变量用长度为n bit的向量来表示，对应bit为1，则该变量为live，反之为0则为dead。

**说明**：从下往上遍历基本块，黑色-初始化；红色-第1次；蓝色-第2次；绿色-第3次。

**结果**：3次遍历后，IN[B]不再变化，遍历结束。



## 5.Available Expressions Analysis (must analysis)

**问题定义**：程序点p处的表达式x op y可用需满足2个条件，一是从entry到p点必须经过x op y，二是最后一次使用x op y之后，没有重定义操作数x、y。（如果重定义了x 或 y，如x = a op2 b，则原来的表达式x op y中的x或y就会被替代）。

**应用场景**：用于优化，检测全局公共子表达式。

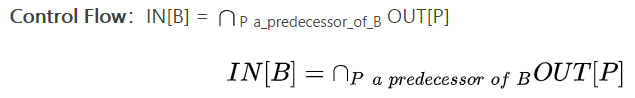
**抽象表示**：程序中的n个表达式，用长度为n bit的向量来表示，1表示可用，0表示不可用。

**说明**：属于forward分析。

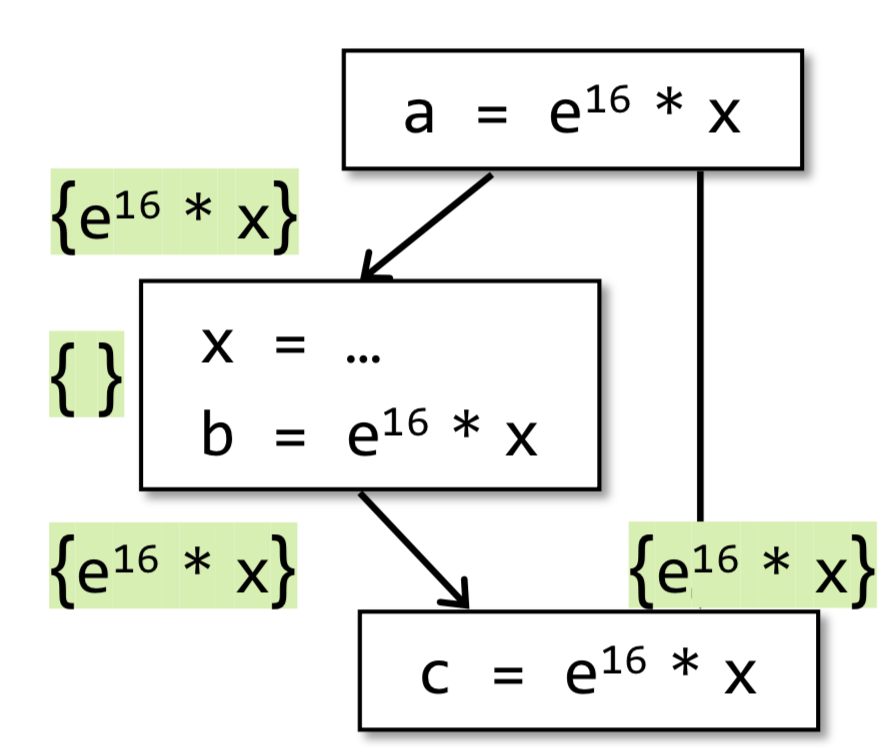
#### （1）公式分析

**Transfer Function**：OUT[B] = genB U (IN[B] - killB)

**理解**：genB—基本块B中所有新的表达式（并且在这个表达式之后，不能对表达式中出现的变量进行重定义）-->加入到OUT；killB—从IN中删除变量被重新定义的表达式。



**理解**：从entry到p点的所有路径都必须经过该表达式。

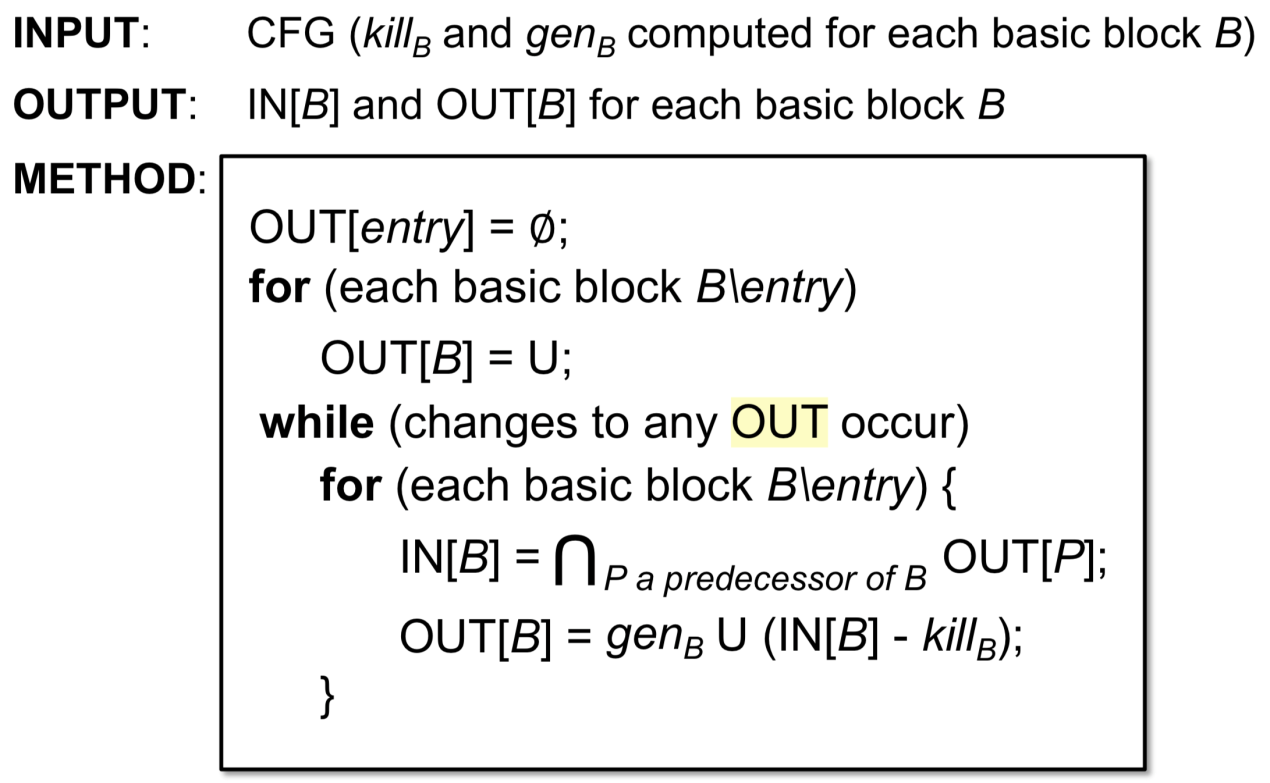


**问题**：该分析为什么属于must analysis呢？因为我们允许有漏报，不能有误报，比如以上示例中，改为x=3，去掉 b=e16\*x，该公式会把该表达式识别为不可用。但事实是可用的，因为把x=3替换到表达式中并不影响该表达式的形式。这里虽然漏报了，但是不影响程序分析结果的正确性。

#### （2）算法

**目的**：输入CFG，提前计算好genB和killB。

**方法**：首先将OUT[entry]初始化为空，所有基本块的OUT[B]**初始化为1...1**。遍历每一个基本块B，按以上两个公式计算块B的IN[B]和OUT[B]，只要这次遍历时有某个块的OUT[B]发生变化，则重新遍历一次（因为有循环，只要某块的OUT[B]变了，就意味后继块的IN[B]变了）。

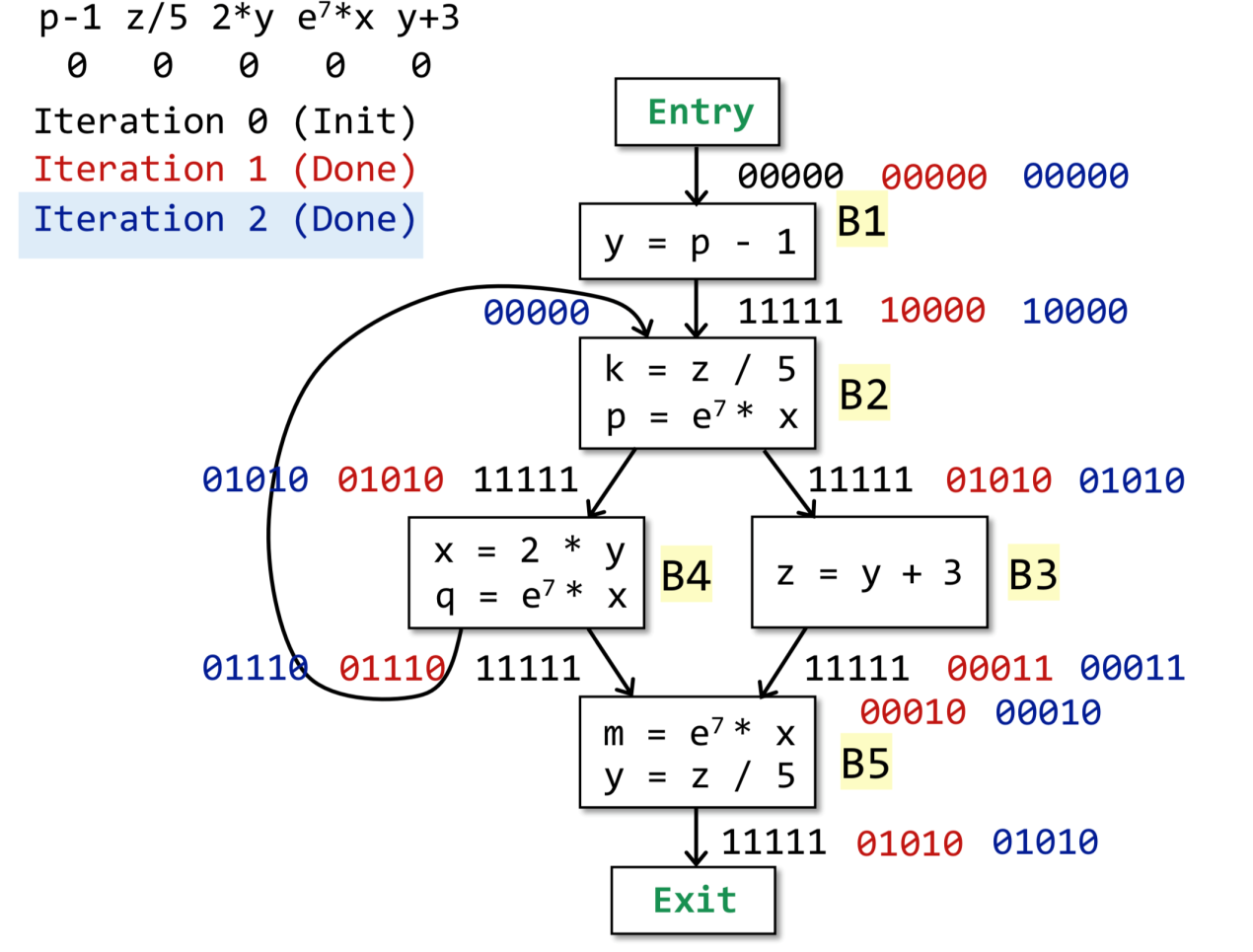


#### （3）实例

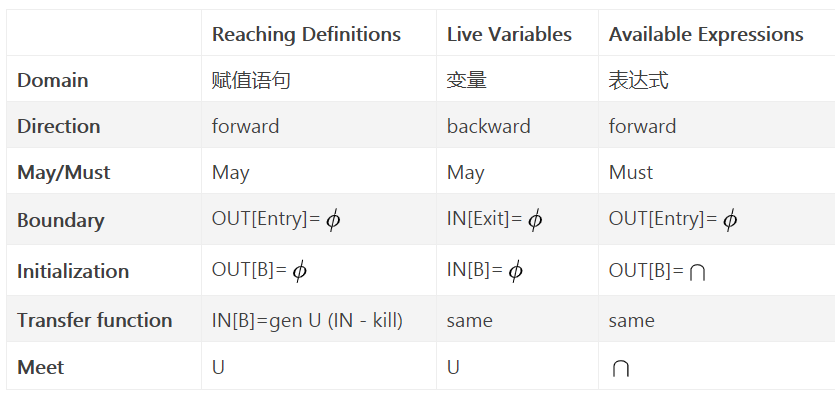
**抽象表示**：程序中的n个表达式，用长度为n bit的向量来表示，1表示可用，0表示不可用。

**说明**：黑色-初始化；红色-第1次；蓝色-第2次。

**结果**：2次遍历后，OUT[B]不再变化，遍历结束。



## 6.三种分析技术对比



**问题**：怎样判断是May还是Must？

Reaching Definitions表示只要从赋值语句到点p**存在1条路径**，则为reaching，结果不一定正确；Live Variables表示只要从点p到Exit**存在1条路径**使用了变量v，则为live，结果不一定正确；Available Expressions表示从Entry到点p的**每一条路径**都经过了该表达式，则为available，结果肯定正确。