数据流分析

简介

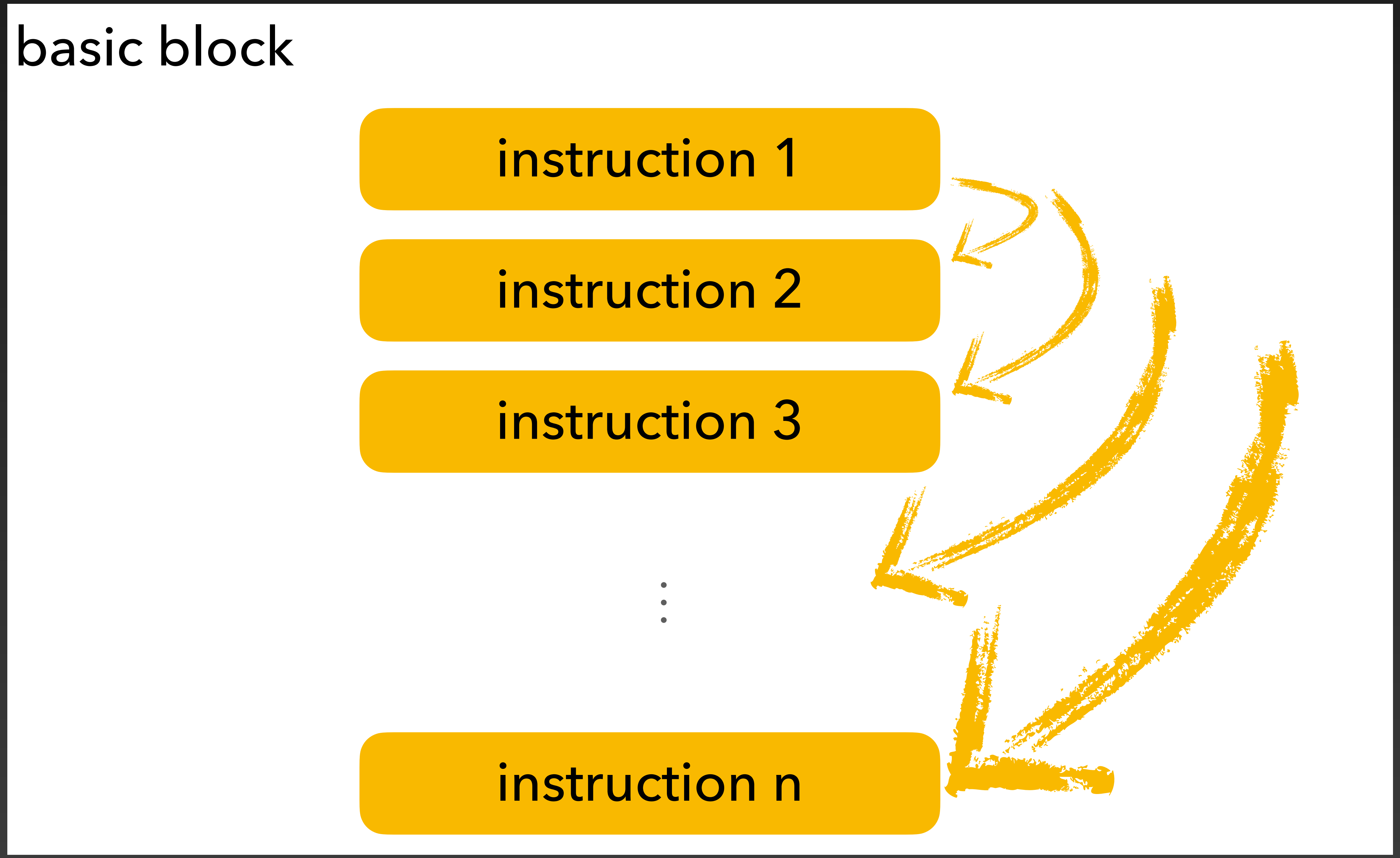
钮鑫涛

什么是数据流分析

- 分析出一个给定的程序如何操纵他的数据的全局信息
    - 不同的优化技术会使用不同的分析的信息
        - 比如变量的活跃范围，常量折叠...
    - 分析不能给出错误的信息！因为错误的分析信息会导致错误的优化！
        - 因此，分析应该给出"保守"的近似结果！（可以不说话，但不撒谎）

什么是数据流分析(Data flow analysis)

- 本地数据流分析（e.g., 值编号）
  - 分析基本块内的每个指令的效果（语义）
  - 复合指令间的效果来获取基本块内每个指令想要的信息（关心的性质，如：会出现除0吗？会出现数组越界吗？…）



什么是数据流分析(Data flow analysis)

- 全局数据流分析 (分析时越过基本块)
  - 分析每个基本块的效果
  - 复合基本块的效果来获取基本块边界的信息（基本块看成整体）
  - 从基本块的边界开始，使用本地分析技术对每个指令的信息进行分析
  - 当然，你可以不总结边界信息，而是完全使用本地分析技术，对所有指令（无论是基本块内，还是基本块外）进行分析
    - 这种做法会加大分析的代价（当然相应的精度会高点）

一个语句的效果

Def-use的角度

- 对一条指令 a = b + c 而言：
  - Uses 两个源变量（b， c）
  - Kills 一个旧的定义（旧的a的定义）
  - Defines 一个新的定义（a）

一个基本块的效果

- 复合多个语句的效果 → 一个基本块的效果
  - 在基本块内的一个**locally exposed use** （来自其他基本块）, 其使用的data item 不能越过一个基本块内的对这个data item的definition
  - 任何一个在基本块内的对data item的定义都会kill所有到达这个基本块（来自其他基本块）的对同样data item的定义
  - 一个**locally generated definition** （**该基本块生成的**） = 基本块内最后的对data item的定义

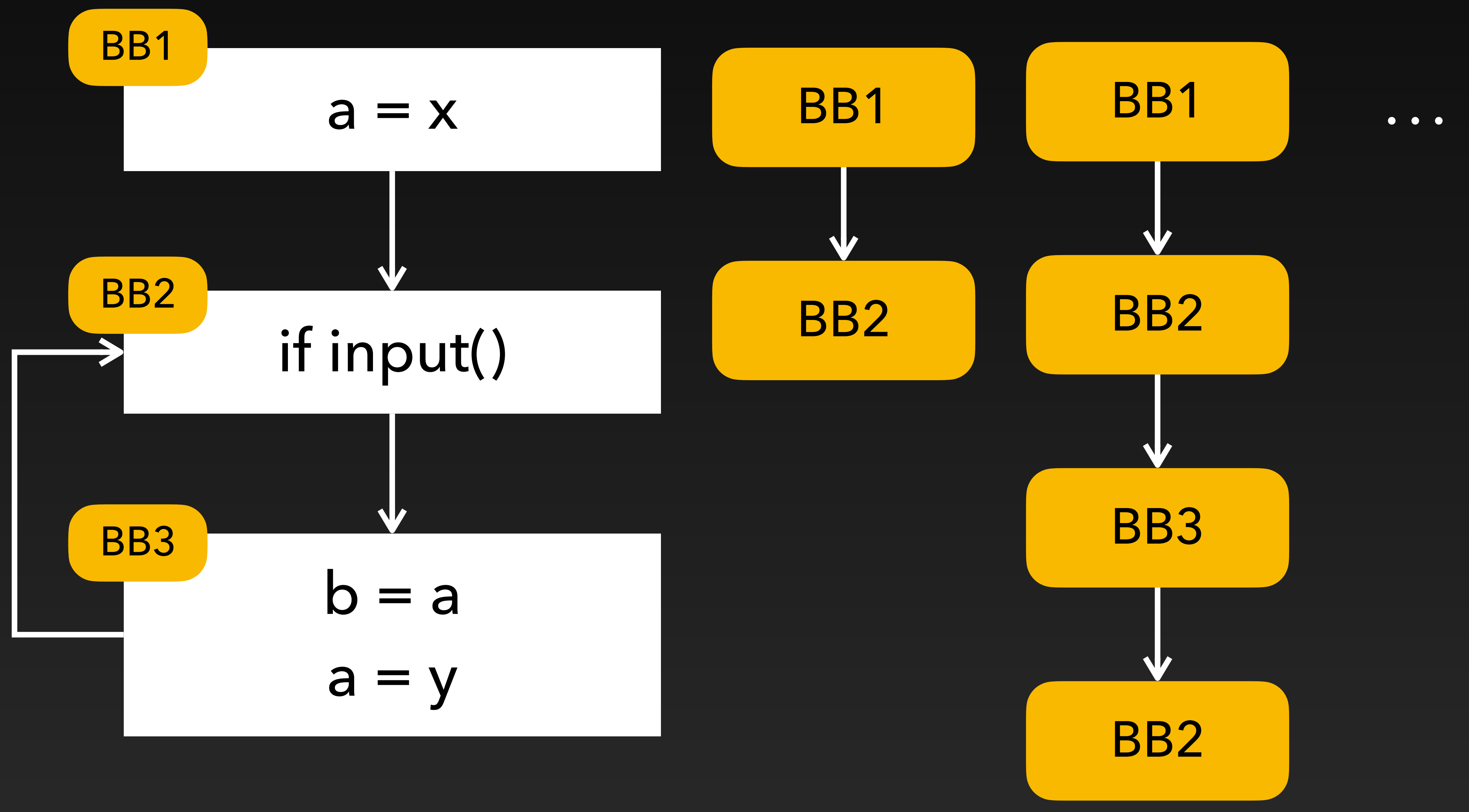
```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Locally exposed use: r1 (Note: r2 has been defined previously in this bb)
Kills definitions: t2
Locally generated definitions: t2 (Note: r1 and r2 will be redefined later)

穿越基本块的复合效果

静态的程序 VS 动态的执行

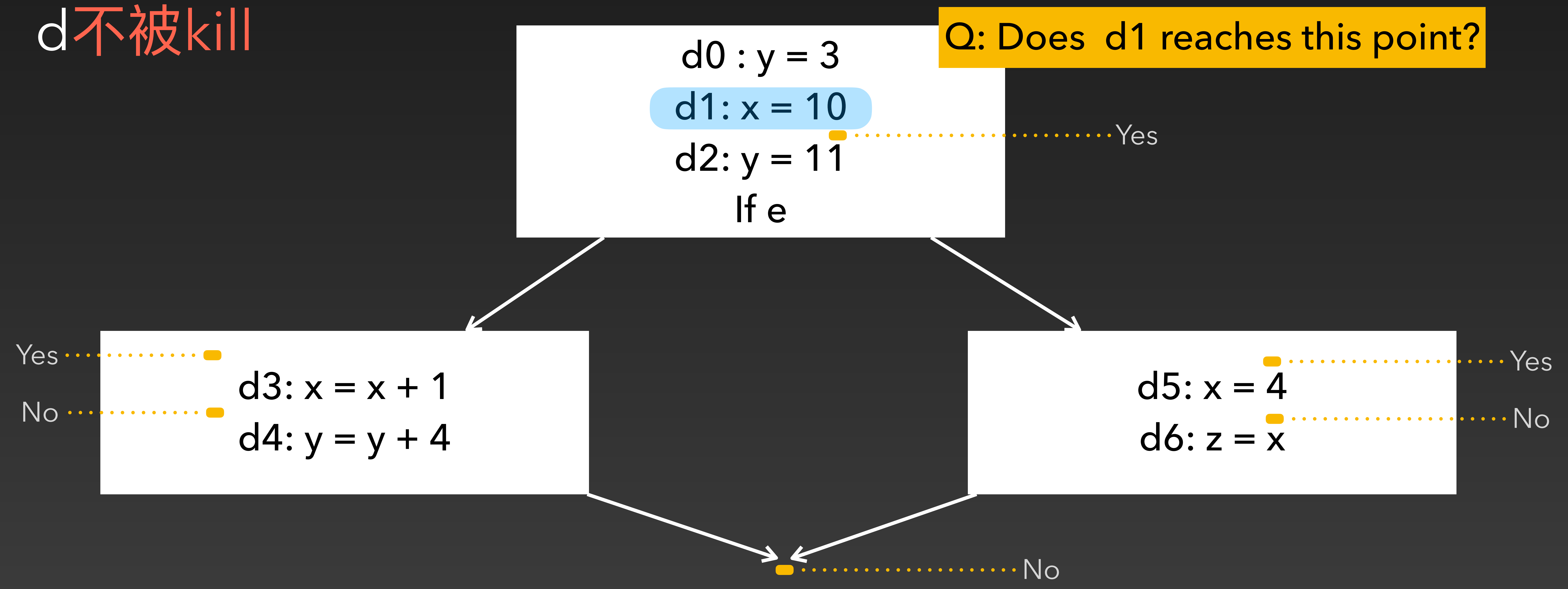
- 静态的角度看：
  - 就是一个有限的字符串
- 动态的角度看：
  - 可以有很多（无数）多种可能的执行路径
- 数据流抽象
  - 不是针对一次执行，而是所有可能的执行
  - 在每个程序的静态点关联一个谓词判定，该判定对于所有动态的执行都为True
    - 对程序的每一个点，分析所有可能执行下这个点的信息，并复合出一个抽象信息（抽象解释！）



定义可达性（Reaching Definitions）

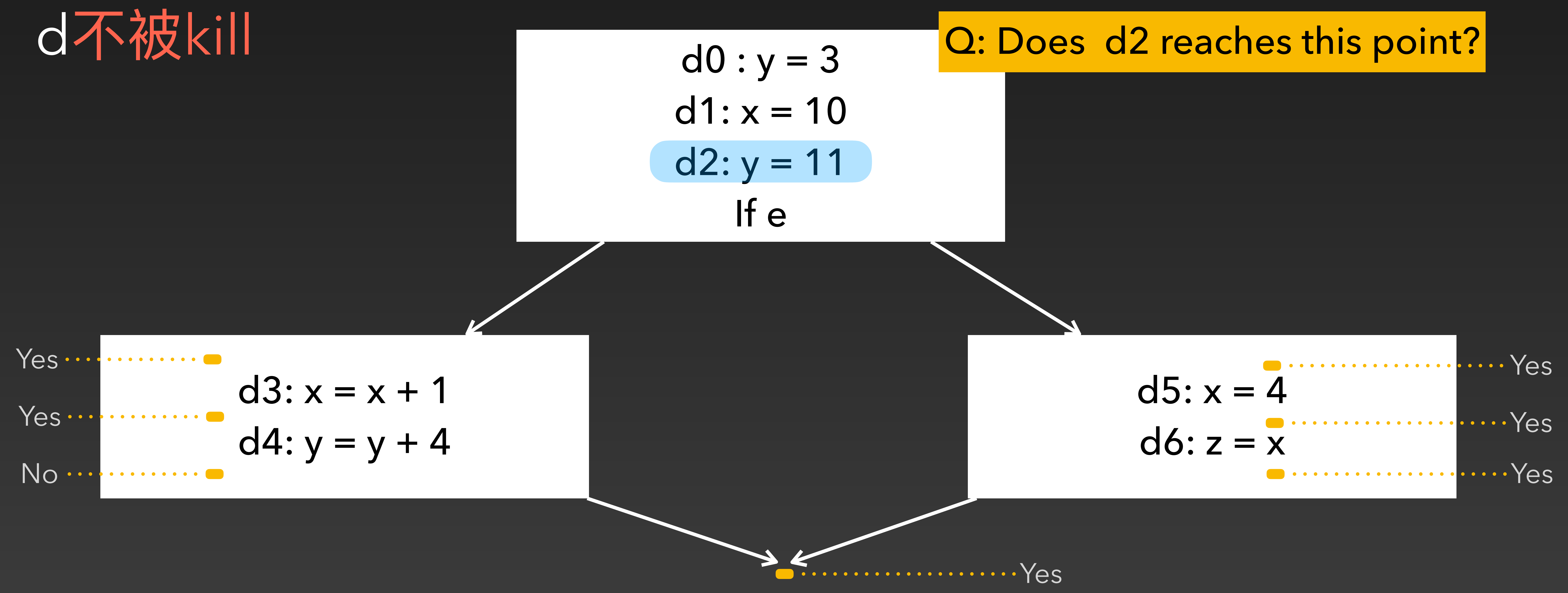
可达的定义（Reaching Definitions）

- 一个变量x的定义（definition），是一个指令对x的赋值（或可能赋值，比如在一个条件判定下）
  - 一个定义d到达（reaches）一个点p，当存在一个路径可以从d到p，在这个路径上，d不被kill



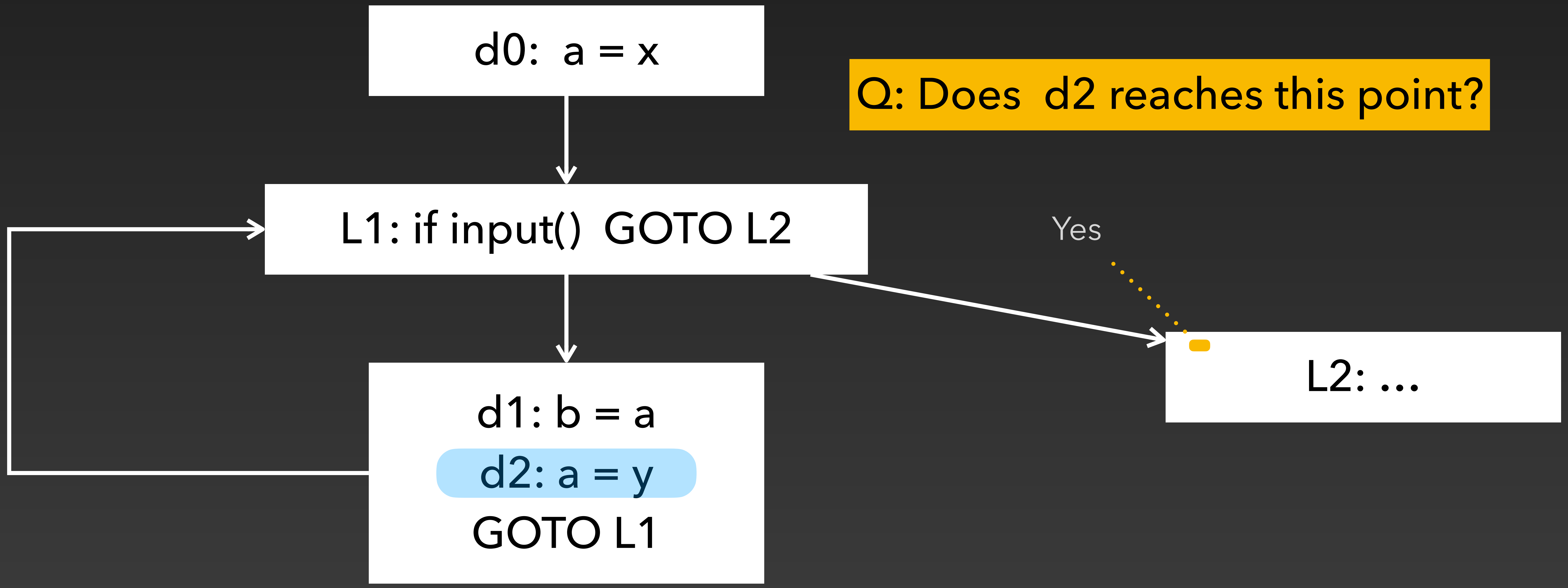
可达的定义（Reaching Definitions）

- 一个变量x的定义（definition），是一个指令对x的赋值（或可能赋值，比如在一个条件判定下）
  - 一个定义d到达（reaches）一个点p，当存在一个路径可以从d到p，在这个路径上，d不被kill



可达的定义（Reaching Definitions）

- 一个变量x的定义（definition），是一个指令对x的赋值（或可能赋值，比如在一个条件判定下）
  - 一个定义d到达（reaches）一个点p，当存在一个路径可以从d到p，在这个路径上，d不被kill



定义可达的作用

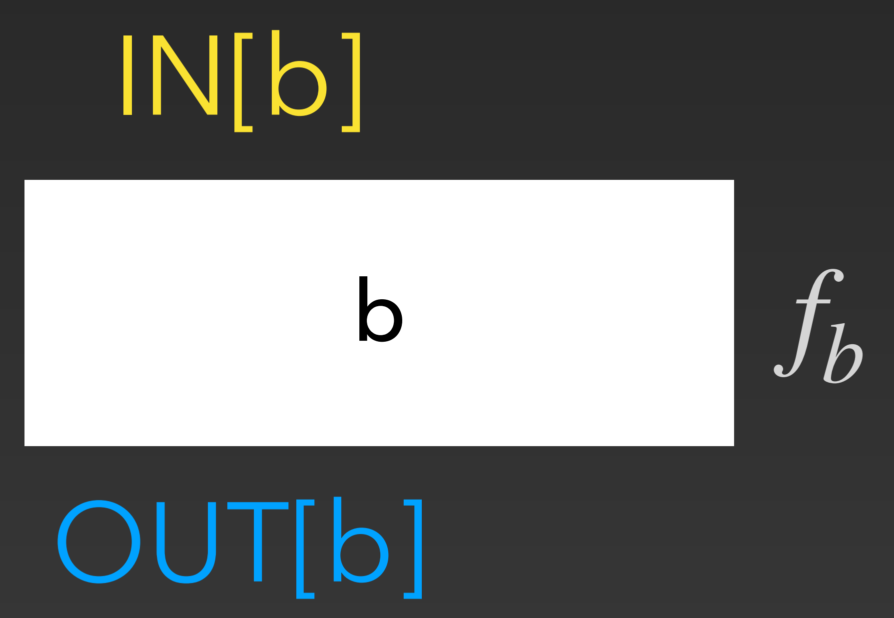
- 定义可达可以用来侦测 “未定义变量的使用” 这样的行为
  - 可以为每个变量在开始都创建一个假定义点
  - 然后看看这些假定义点会不会在使用点可达
    - 如果可达，那么就是一个未定义行为！

可达定义的分析

- 问题描述
  - 对程序中的每个基本块b，判定程序的每个定义是否会到达 b：
    - 我们关心一个基本块在开始的上述可达定义的信息（叫做IN[b]），和结束的可达定义信息（叫做OUT[b]）
  - 具体表示：
    - IN[b], OUT[b] : 就是一个bit数组，每个元素代表一个定义
  - 一个block b可以影响这个计算：
    - 我们可以从IN[b]得到OUT[b]吗？或者，我们可以从OUT[b]逆推回IN[b]吗？

一个基本块的作用

- 前向问题(Forward problem): 给定IN[b]， 计算OUT[b]
- 本质上给出一个基本块b的转换函数f_b （this transfer function abstracts the execution with respect to the problem of interest）
  - $OUT[b] = f_b(IN[b])$
  - 从输入的可达定义 → 出去的可达定义

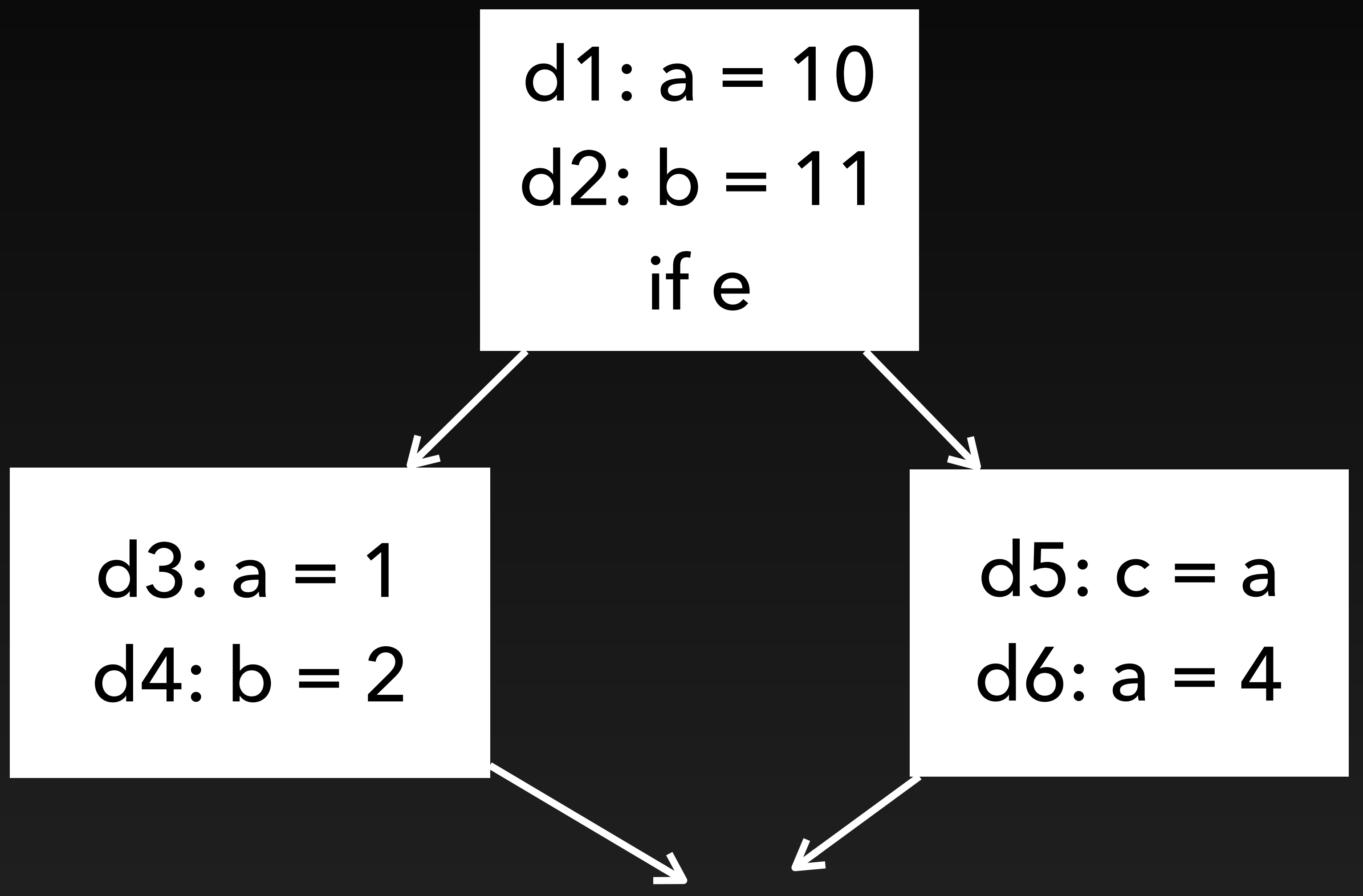


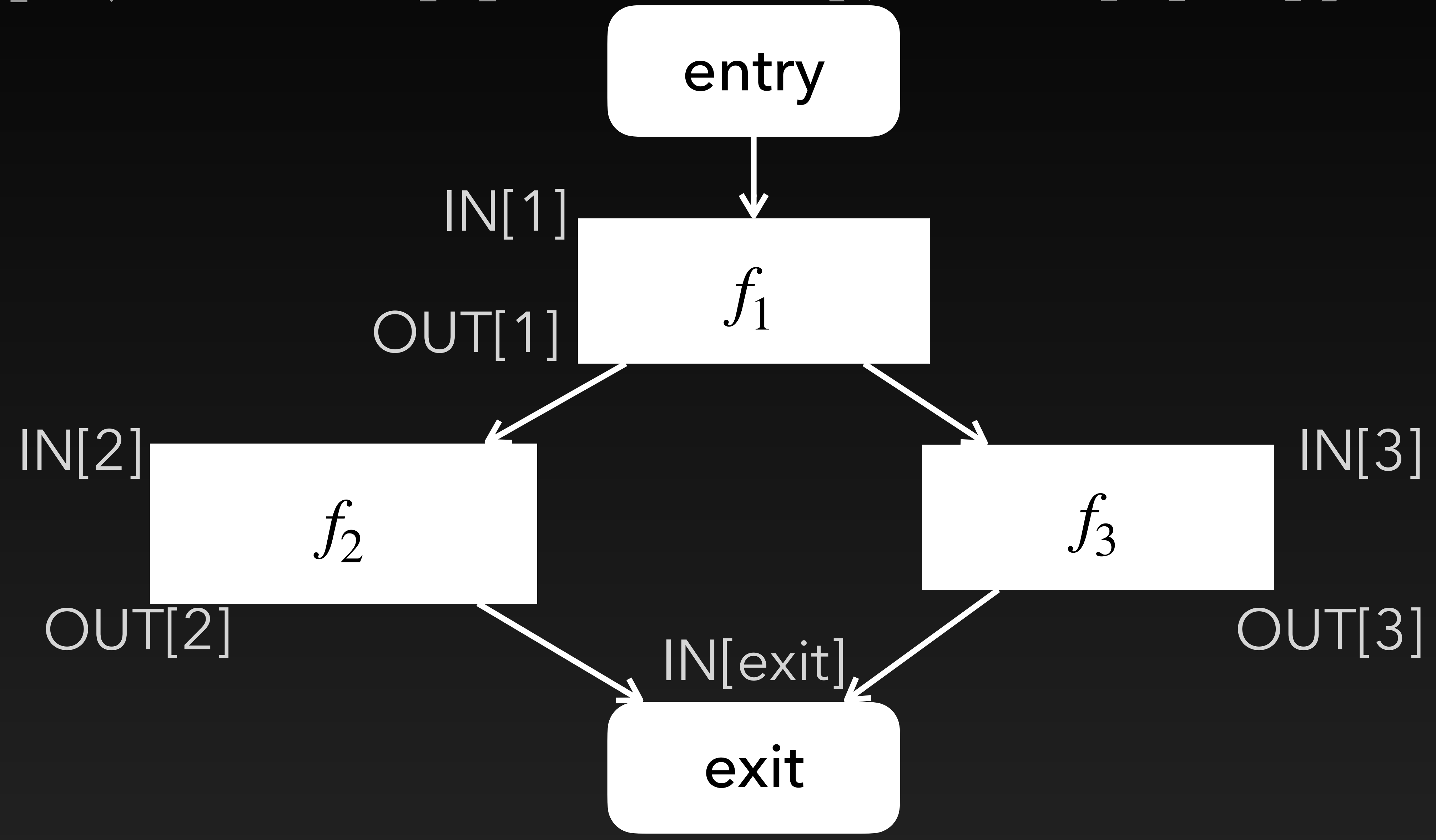
描述一个基本块的作用

- 一个基本块b
  - generates definitions $GEN[b]$: 基本块b本地产生的定义
  - propagate definitions $IN[b] - KILL[b]$: $KILL[b]$是在基本块内的本地定义消除所有定义

$$OUT[b] = GEN[b] \cup (IN[b] - KILL[b])$$

控制流中边的作用（无环图）

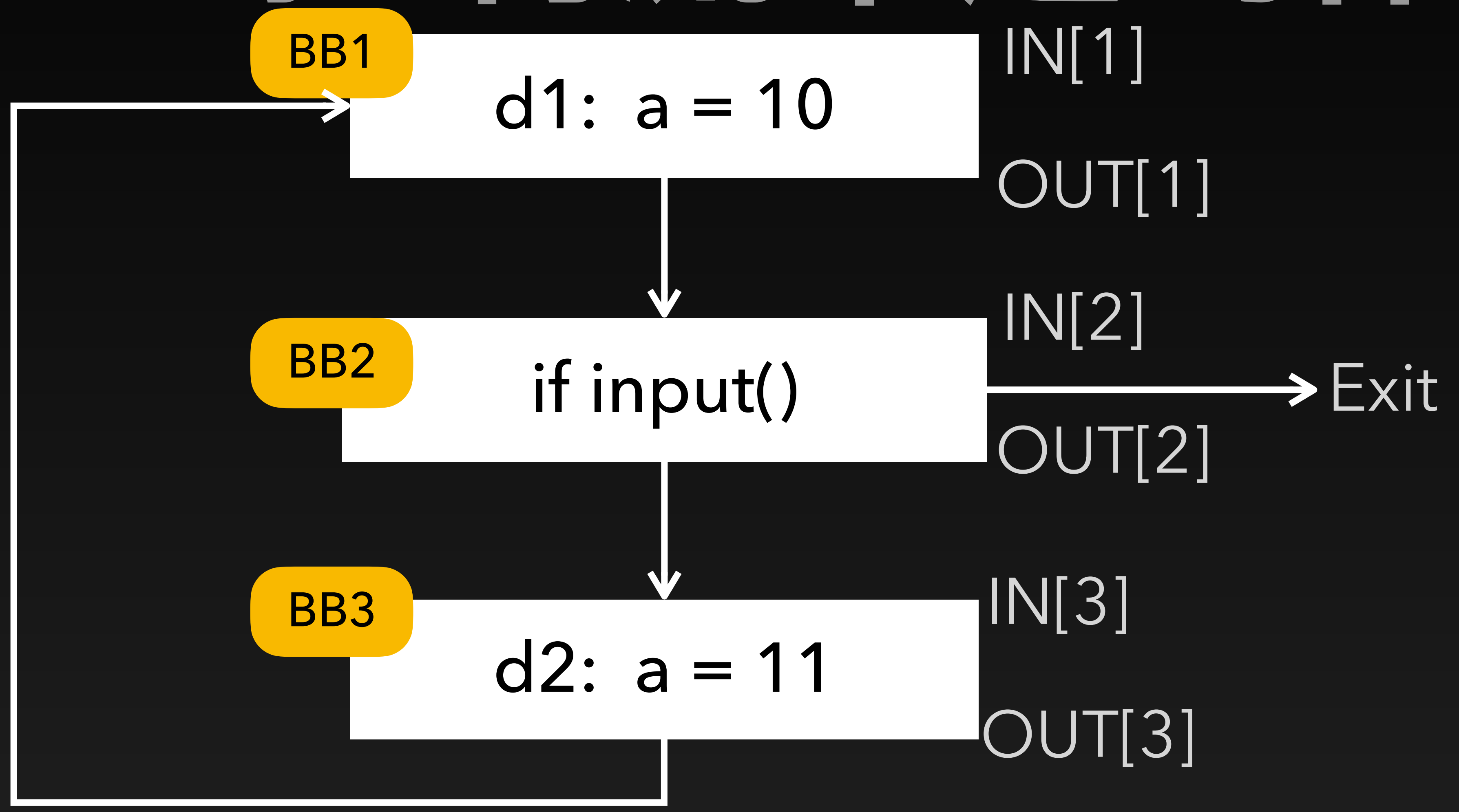




	GEN	KILL
1	{1, 2}	{3, 4, 6}
2	{3, 4}	{1, 2, 6}
3	{5, 6}	{1, 3}

- 我们已经知道如何从IN[b]计算OUT[b]，但这个IN怎么来呢？这个IN是由这个基本块的前驱块的OUT决定，如果前驱只有一个块，那么就是简单的传播（相同）
- 但如果是多个前驱块呢？我们需要复合他们
  - 我们需要一个**meet**操作来join多个前驱节点的数据，应该怎么做呢？
    - 对于个可达定义这个问题而言：$IN[b] = OUT[p_1] \cup OUT[p_2] \cup \ldots OUT[p_n]$, 其中$p_1, p_2, \ldots p_n$都是b的前驱

控制流中边的作用（有环图）



	GEN	KILL
1	{1}	{2}
2		
3	{2}	{1}

- 之前的等式依然成立
  - $OUT[b] = GEN[b] \cup (IN[b] - KILL[b])$
  - $IN[b] = OUT[p_1] \cup OUT[p_2] \cup ... OUT[p_n]$
- 任何解决此方程的解：不动点
- 具体解法：需要反复的使用这个方程，直到不动点，即Worklist 算法

可达定义的Worklist算法：

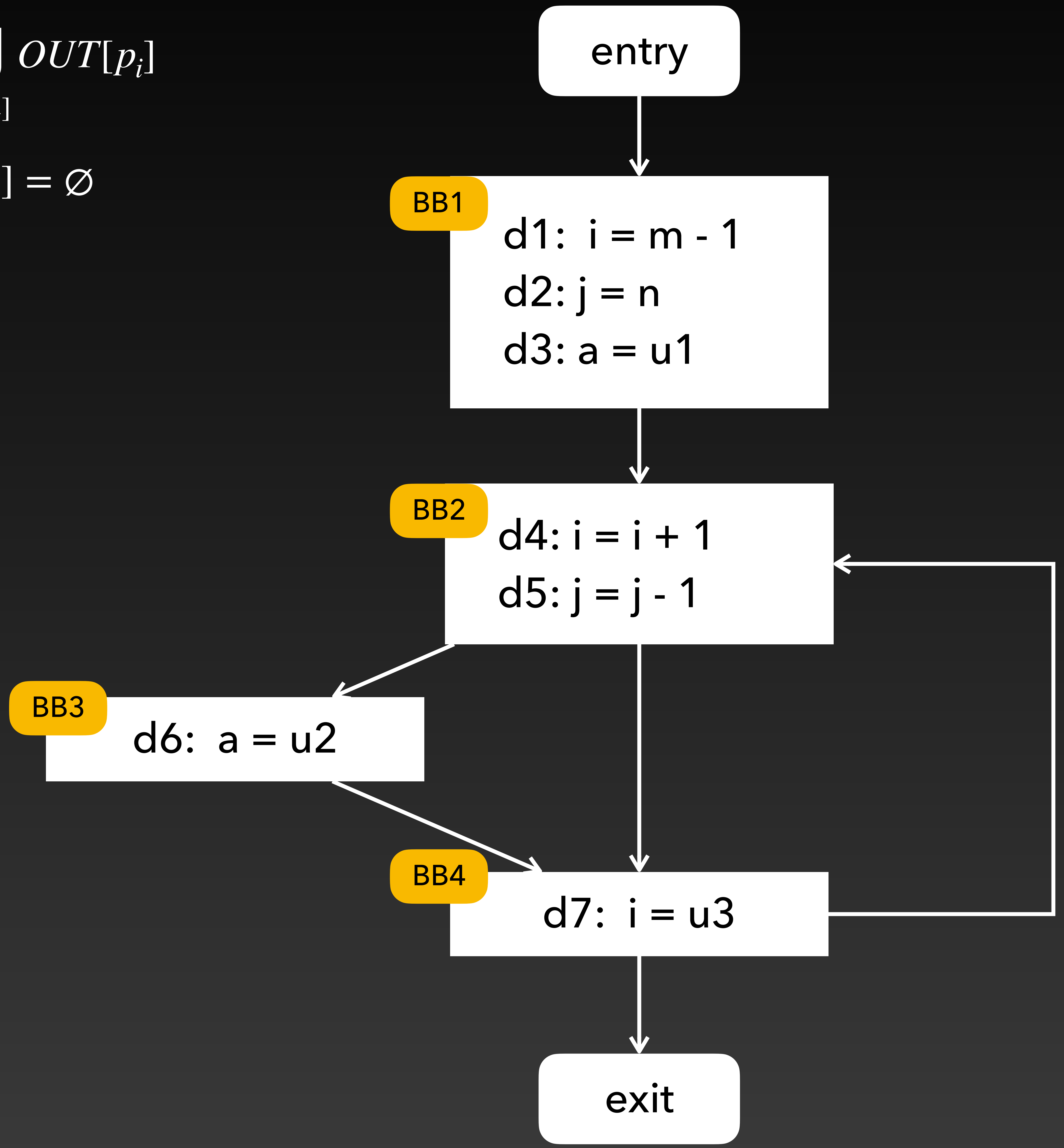
```
Input: Control Flow Graph CFG = (N, E, Entry, Exit)
  /* Initialize */
  OUT[Entry] = { }  /* could set OUT[Entry]  to special defs, if reaching then undefined use! */

  for all nodes i
     OUT[i] = { }             /* could optimize by out[i]=gen[i] */

  ChangeNodes = N

  /* Iterate */
  while ChangeNodes != { } {
       remove i from ChangeNodes
       IN[i] = ∪ (OUT[p]), for all predecessors p of i
       oldout = OUT[i]
       OUT[i] = f_i (IN[i])   /* OUT[i] = GEN[i] ∪ (IN[i] – KILL[i]) */
       if (oldout != OUT[i]) {
          for all successors s of i
             add s to ChangeNodes
       }
  }
```

可达定义例子

$OUT[b] = GEN[b] \cup (IN[b] - KILL[b])$

$IN[b] = \bigcup_{i \in [n]} OUT[p_i]$

Init: $OUT[b] = \varnothing$



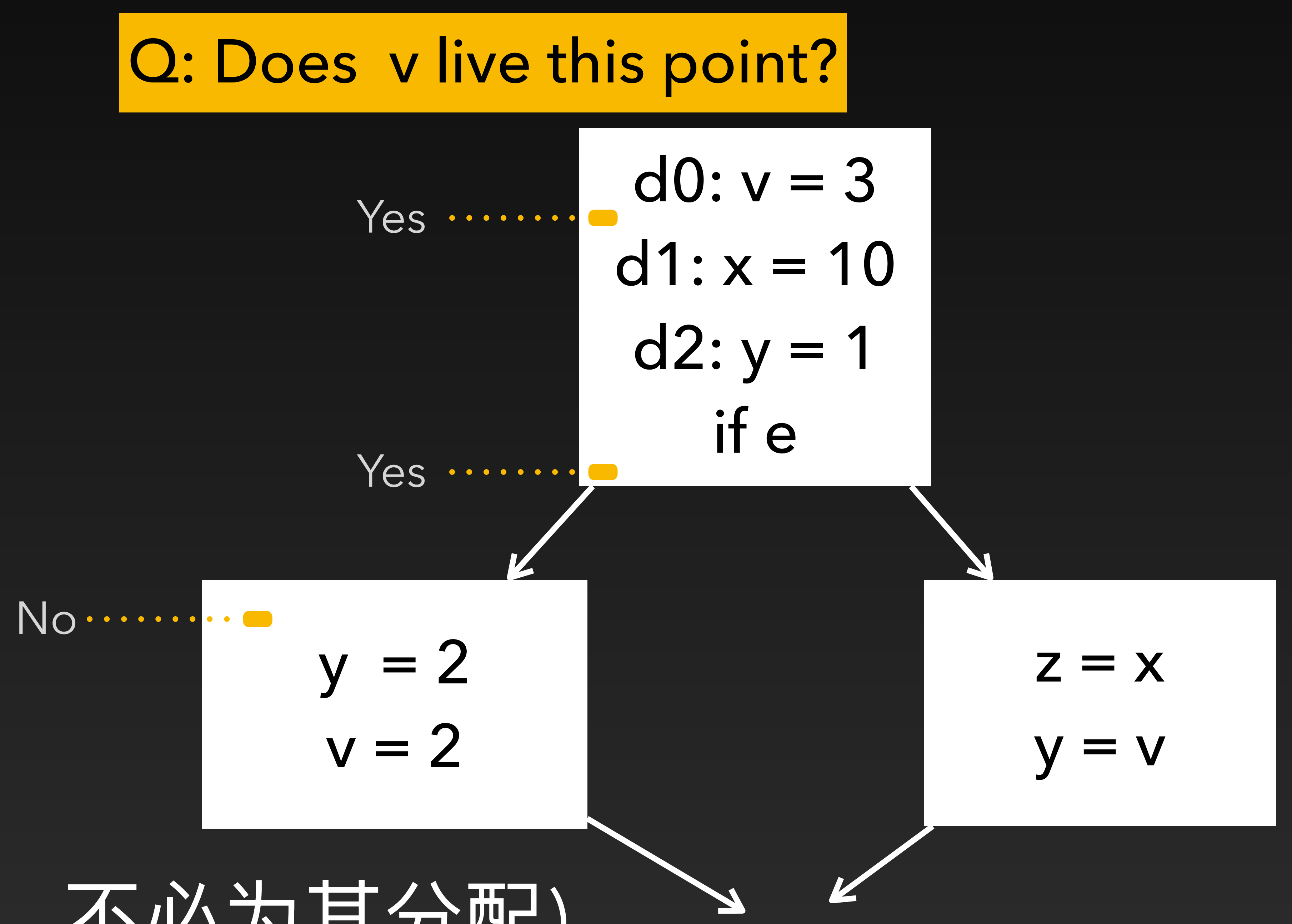
	First Pass							Send Pass						
IN[1]	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OUT[1]	1	1	1	0	0	0	0	1	1	1	0	0	0	0
IN[2]	1	1	1	0	0	0	0	1	1	1	0	1	1	1
OUT[2]	0	0	1	1	1	0	0	0	0	1	1	1	1	0
IN[3]	0	0	1	1	1	0	0	0	0	1	1	1	1	0
OUT[3]	0	0	0	1	1	1	0	0	0	0	1	1	1	0
IN[4]	0	0	1	1	1	1	0	0	0	1	1	1	1	0
OUT[4]	0	0	1	0	1	1	1	0	0	1	0	1	1	1
IN[exit]	0	0	1	0	1	1	1	0	0	1	0	1	1	1

Fixed Point

活跃变量分析

活跃变量(Live variable)分析

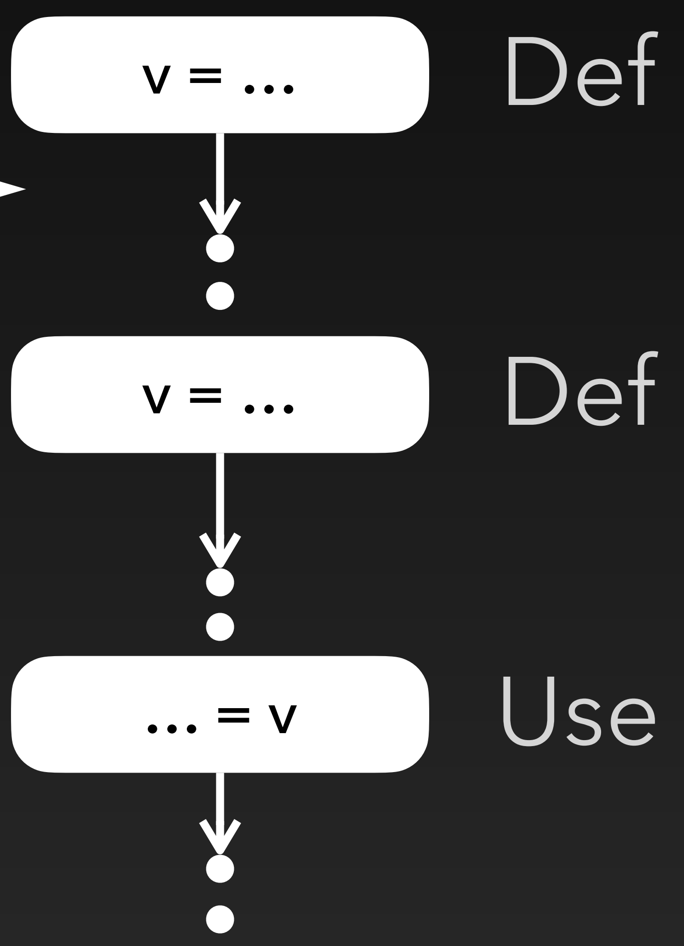
- 定义：
  - 一个变量v在某个点p是活跃的当
    - 从p点开始，v的值未来沿着某条path还会被使用
  - 否则，这个变量就是dead
- 用途：
  - 寄存器的分配 (寄存器是有限的，如果某个变量不再活跃，不必为其分配)
- 问题描述：
  - 对于每个基本块，判定每一个变量是否还在这个基本块的边界活跃（IN[b], OUT[b])
  - 所有变量的判定值形成一个bit数组



活跃变量：基本块的效果（转换函数）

- 后向问题(Backward problem): 给定OUT[b]， 计算IN[b]

一条执行路径



v是活跃的，其在未来被使用，其期间没有被redefine!
从后往前分析比较简单！

控制流

$$IN[b] = f_b(OUT[b])$$

f_b

$OUT[b]$

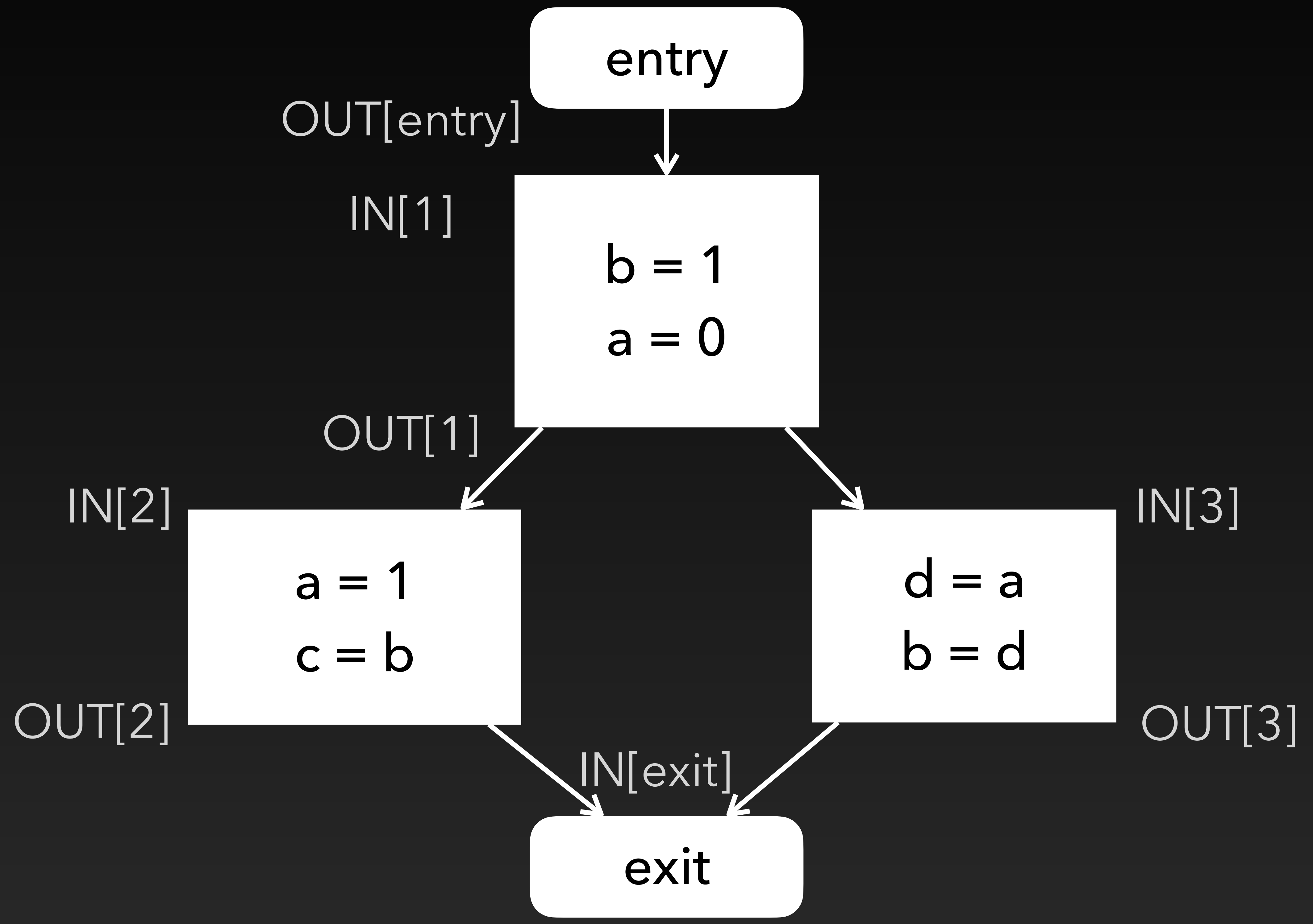
基本块内使用，那么前驱就是活跃的

- 一个基本块b：
  - 产生（Generate） 活跃变量 （对于前驱而言）： Use[b]， 一个会被b本地使用的集合
  - 传播（Propagate） 活跃变量（从后向前）： OUT[b] - Def[b]， 这里Def[b]是在b里定义的变量集合

本来是活跃的，在基本块内redefine了，对于前驱而言，就不活跃了

转换函数： $IN[b] = Use[b] \cup (OUT[b] - Def[b])$

边的影响（在无环图中）



	Use	Def
1	{}	{a, b}
2	{b}	{a, c}
3	{a}	{b, d}

Note: here *d* is not a locally exposed use

- $IN[b] = f_b(OUT[b])$
- join node： a node with multiple successors
- Meet operator
    - $OUT[b] = IN[s_1] \cup IN[s_2] \cup \ldots IN[s_n]$, 其中$s_1, s_2, \ldots s_n$都是b的后继

活跃变量：Worklist 算法

```
Input: Control Flow Graph CFG = (N, E, Entry, Exit)
  /* Initialize */
  in[Exit] = { }

  for all nodes i
    IN[i] = { }

  ChangeNodes = N

  /* Iterate */
  while ChangeNodes != { } {
        remove i from ChangeNodes
        OUT[i] = ∪ (IN[s]), for all successors s of i
        oldin = IN[i]
        IN[i] = f_i (OUT[i])   /* IN[i] = USE[i] ∪ (OUT[i] – DEF[i]) */
        if (oldin != IN[i]) {
           for all predecessors p of i
              add p to ChangeNodes
        }
  }
```

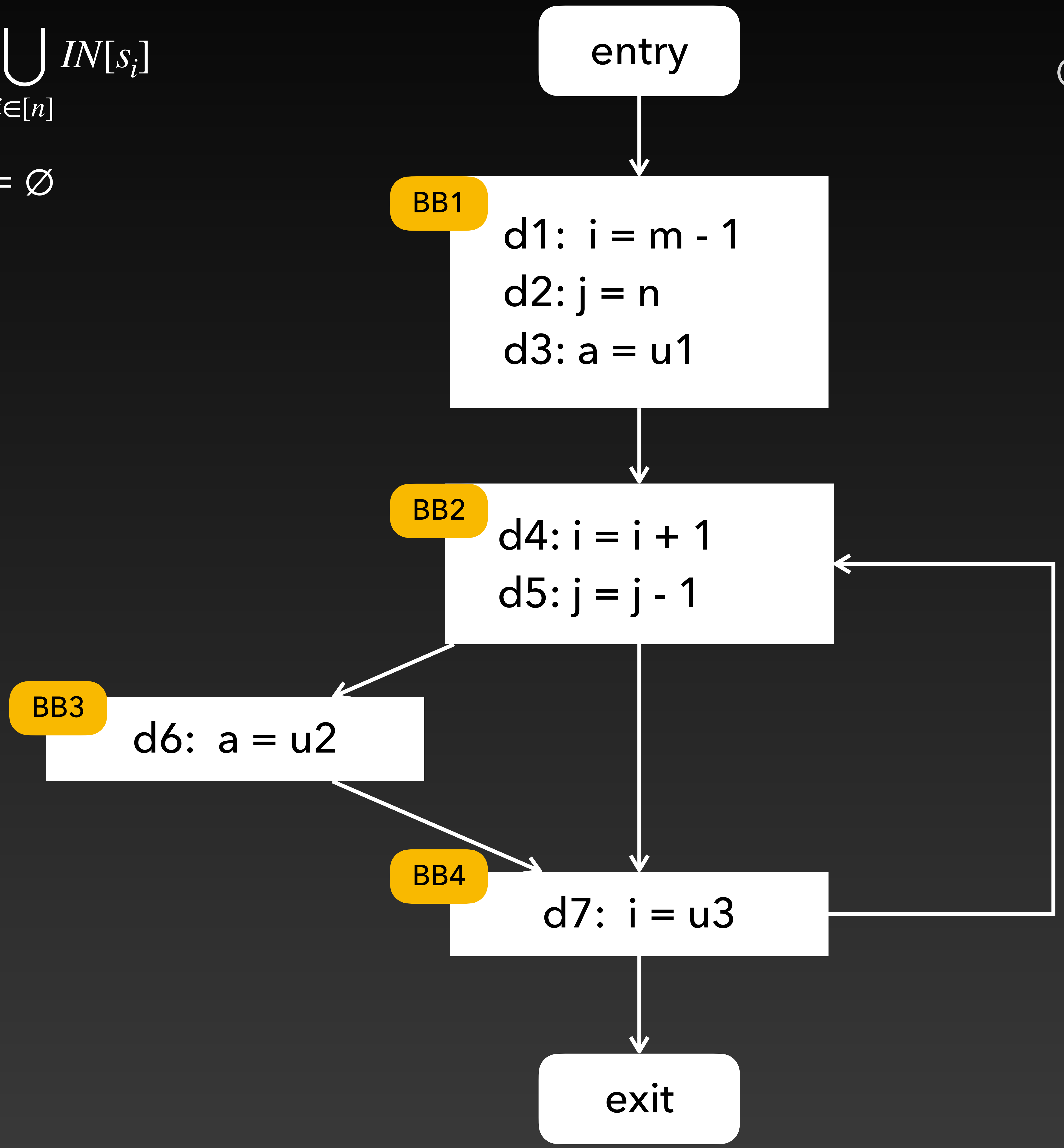
活跃变量例子

1	2	3	4	5	6	7	8
m	n	u1	u2	u3	i	j	a

$IN[b] = Use[b] \cup (OUT[b] - Def[b])$

$OUT[b] = \bigcup_{i \in [n]} IN[s_i]$

Init: $IN[b] = \varnothing$



	First Pass	Send Pass
OUT[entry]	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 0
IN[1]	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 0
OUT[1]	0 0 0 1 1 1 1 0	0 0 0 1 1 1 1 0
IN[2]	0 0 0 1 1 1 1 0	0 0 0 1 1 1 1 0
OUT[2]	0 0 0 1 1 0 0 0	0 0 0 1 1 0 1 0
IN[3]	0 0 0 1 1 0 0 0	0 0 0 1 1 0 1 0
OUT[3]	0 0 0 0 1 0 0 0	0 0 0 1 1 0 1 0
IN[4]	0 0 0 0 1 0 0 0	0 0 0 1 1 0 1 0
OUT[4]	0 0 0 0 0 0 0 0	0 0 0 1 1 1 1 0
IN[exit]	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

Fixed Point

数据流分析框架

	Reaching Definitions	Live Varaibles
Domain	Sets of definitions	Sets of variables
Direction	Forward: $OUT[b] = f_b(IN[b])$ $IN[b] = \wedge\ OUT[pred(b)]$	Backward: $IN[b] = f_b(OUT[b])$ $OUT[b] = \wedge\ IN[succ(b)]$
Transfer function	$f_b(IN[b]) = Gen(b) \cup (IN[b] - Kill(b))$	$f_b(OUT[b]) = Use(b) \cup (OUT[b] - Def(b))$
Meet operation ($\wedge$)	U	U
Boundary condition	$OUT[entry] = \varnothing$	$IN[exit] = \varnothing$
Initial interior points	$OUT[b] = \varnothing$	$IN[b] = \varnothing$

Other Data Flow Analysis problems fit into this general framework, e.g., Available Expressions

一些问题

- 正确性
  - 如果算法终止，那么约束方程（safe-approximation directed constraints）可以满足
    - 约束来自两方面：1. 基本块内的语句本身的语义，2. 控制流（边）的语义
- 精度：
  - 这个答案只是所有可能执行的并吗？可以更加精确吗？
- 收敛性：分析会终止吗？
  - 或者节点集合不断变化？
- 性能：多快速度收敛
  - 需要访问每个节点多少次？


Q&A