



Iterative Data Flow Analysis

Dr. William Harrison

harrisonwl@missouri.edu

CS 4430 Compilers I



Iterative Data Flow Analysis

- Generally, an analysis of data dependency within a program
 - Like liveness
 - Liveness for programs with loops is solved with IDFA
- First, **attributes** are associated with each node/basic block and given initial values
- Second, relationships between these attributes are specified as **data flow equations**
- Third, a solution to these equations is solved by **iteration**



Example: Reaching Definitions

- A **definition** is an assignment of some value to a variable
- A particular definition is said to **reach** a given point within a procedure if
 - There is an execution path from the definition to that point s.t. the variable may have the value given in the definition
- **Reaching analysis** asks, for each variable, which definitions of it may apply
 - Just like use-def



Reaching Definitions Analysis

```
int g(int m, int i);  
int f(n)  
{  
    int i=0, j;  
    if (n==1) i=2;  
    while (n>0) {  
        j = i+1;  
        n = g(n,i);  
    }  
    return j;  
}
```

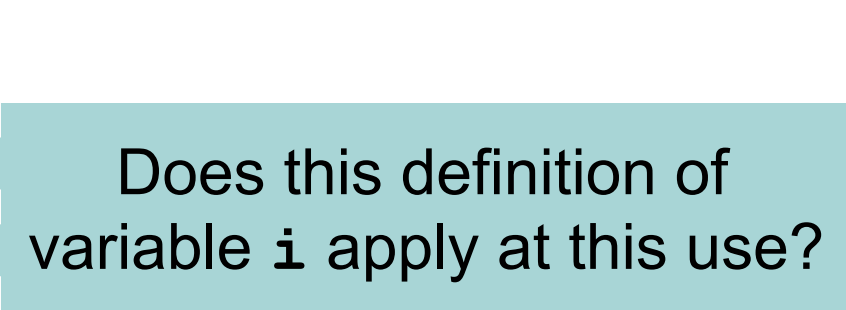
...asks the question: at which points further in a program do particular definitions reach?



Reaching Definitions Analysis

```
int g(int m, int i);
```

```
int f(n)
{
    int i=0, j;
    if (n==1) i=2;
    while (n>0) {
        j = i+1;
        n = g(n,i);
    }
    return j;
}
```



Does this definition of variable `i` apply at this use?

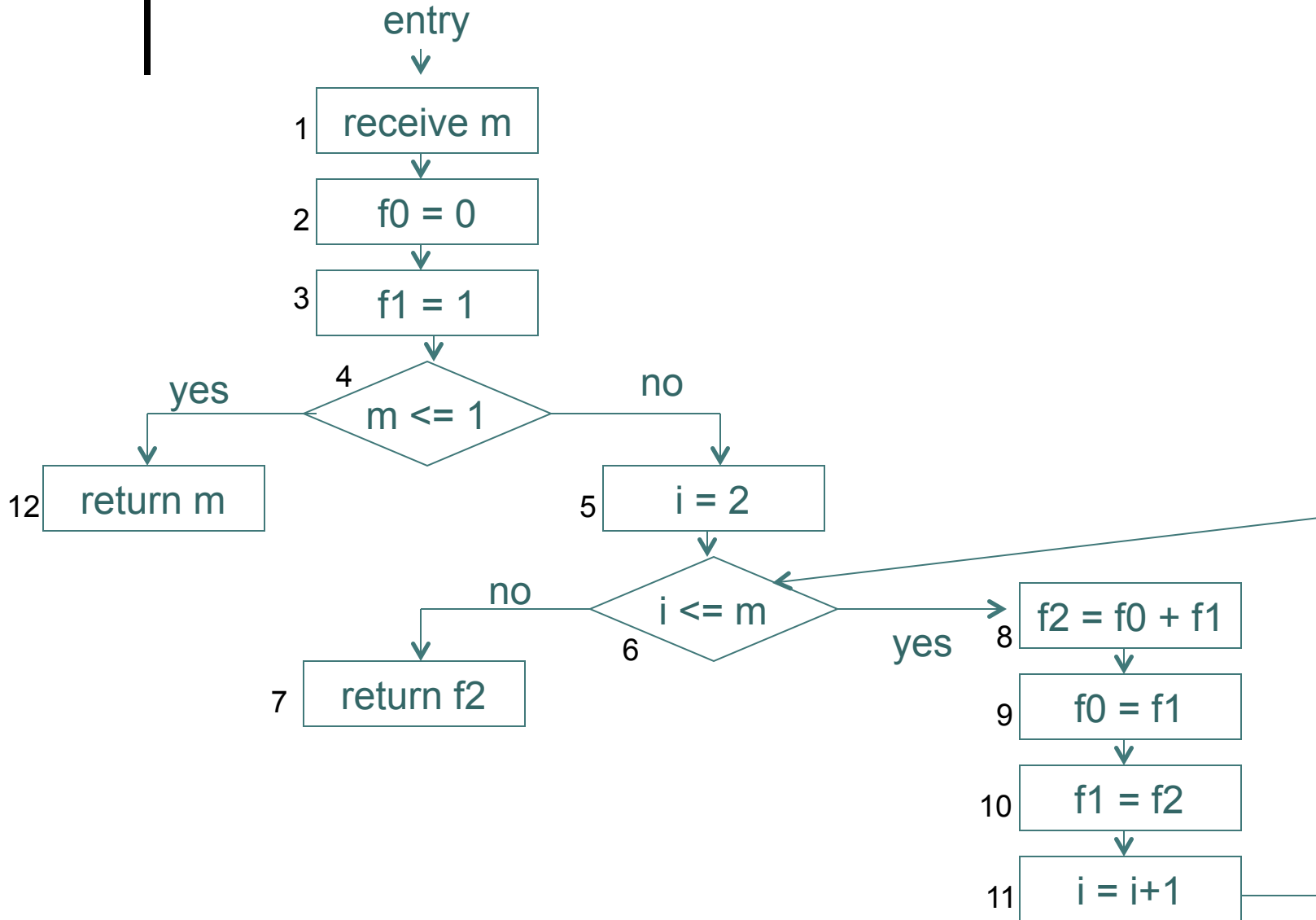
Reaching, like liveness, can generally only be **estimated**



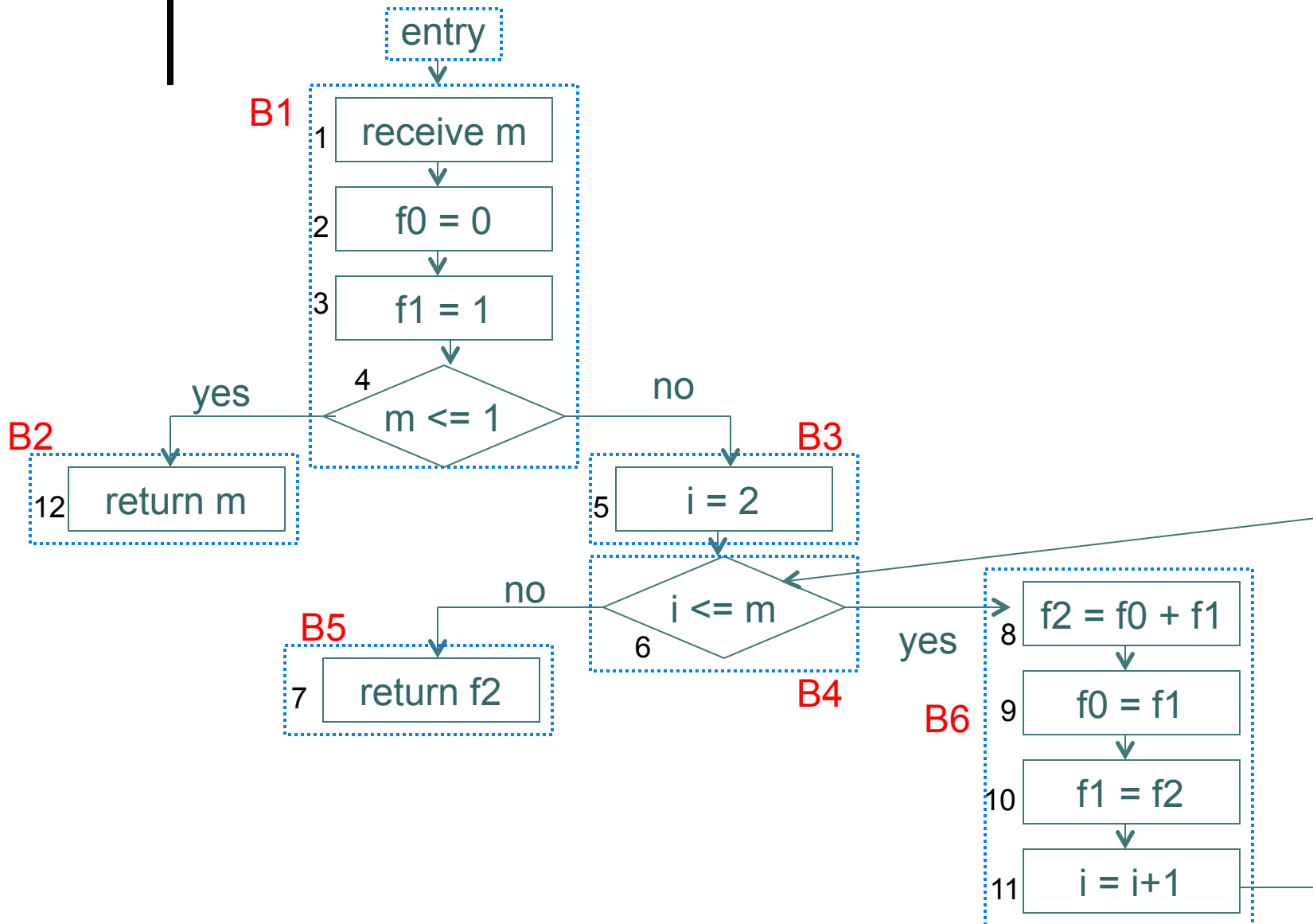
An Example

```
int fib(int m)
{   int f0=0, f1=1, f2, i;
    if (m<=1) {
        return m;
    } else {
        for (i=2; i<=m; i++) {
            f2 = f0 + f1;
            f0 = f1;
            f1 = f2;
        }
        return f2;
    }
}
```

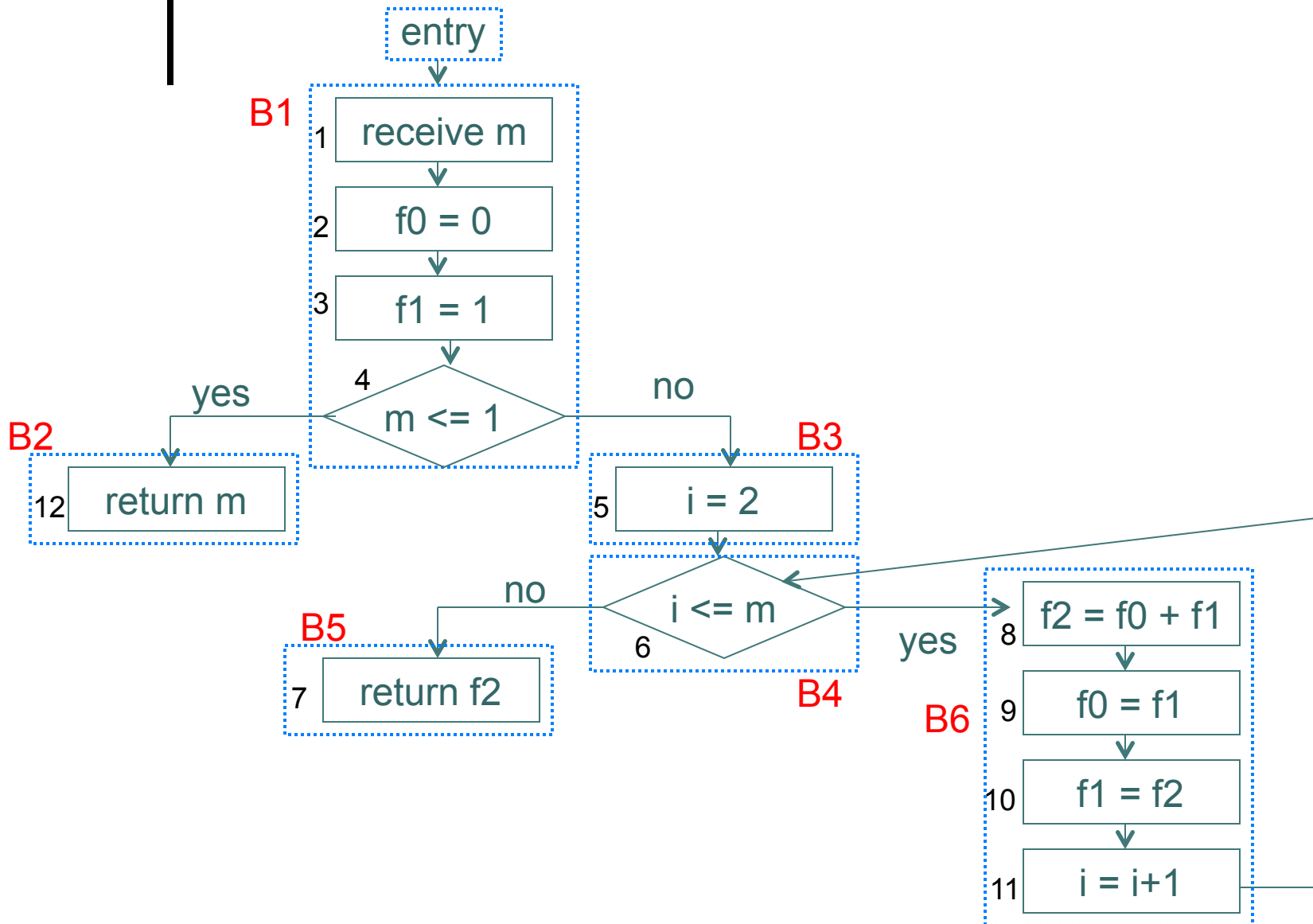
Control Flow Graph for Fibonacci



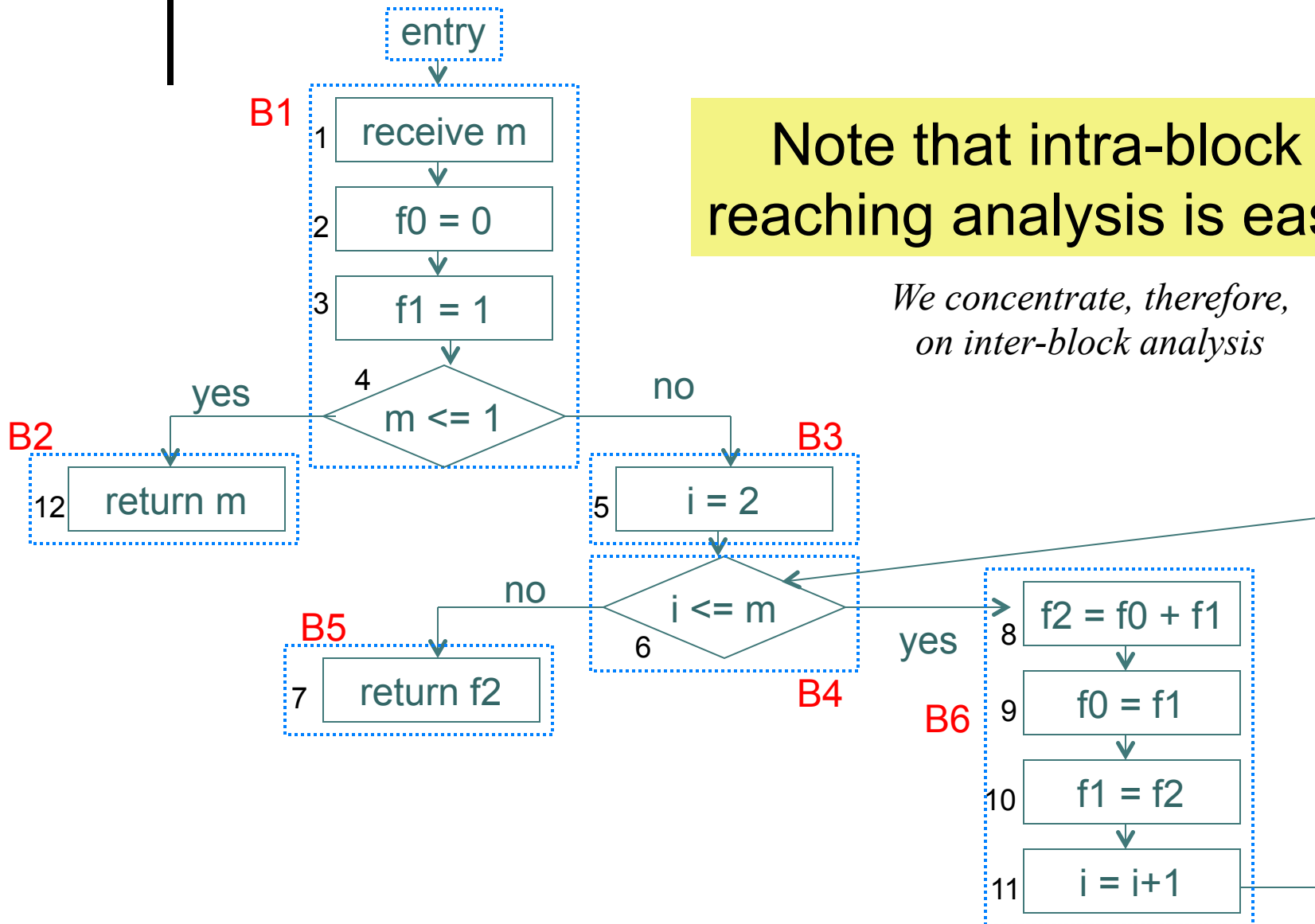
Control Flow Graph for Fibonacci with Basic Blocks



Nodes 1, 2, 3, 5, 8, 9, 10, 11 contain “Definitions”



Intra- vs. Inter-block Reaching Analysis





Attributes

- Reaching Definitions

$RCHin(block)$ = set of definitions that reach *block*

- Preserved Definitions

$PRSV(block)$ = set of definitions preserved by *block*

- Generated Definitions

$GEN(block)$ = set of definitions in *block* not
subsequently killed in *block*

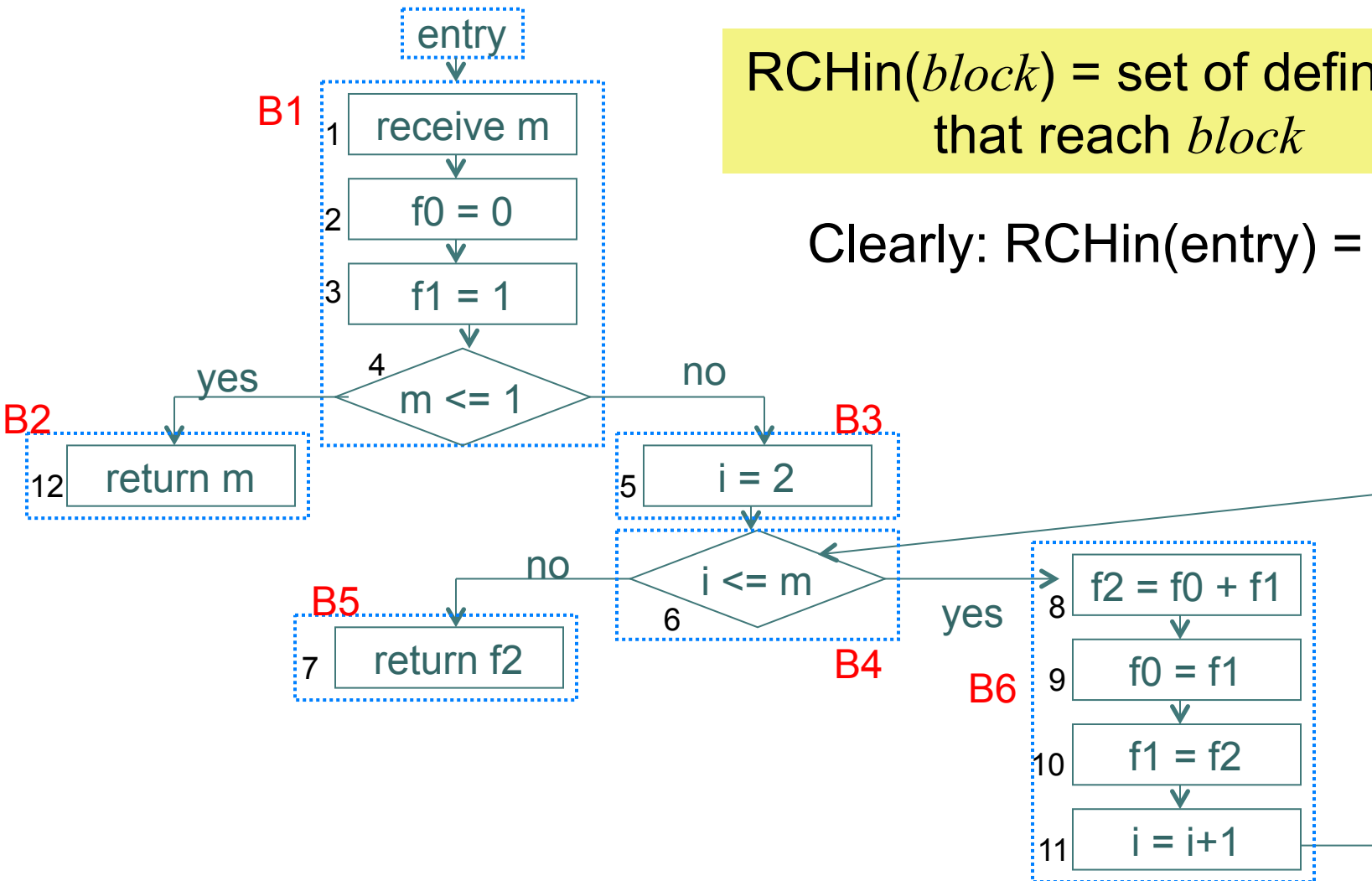
- Out-reaching Definitions

$RCHout(block)$ = set of definitions reaching end of *block*

Reaching Definitions: $RCHin(node)$

$RCHin(block) = \text{set of definitions that reach } block$

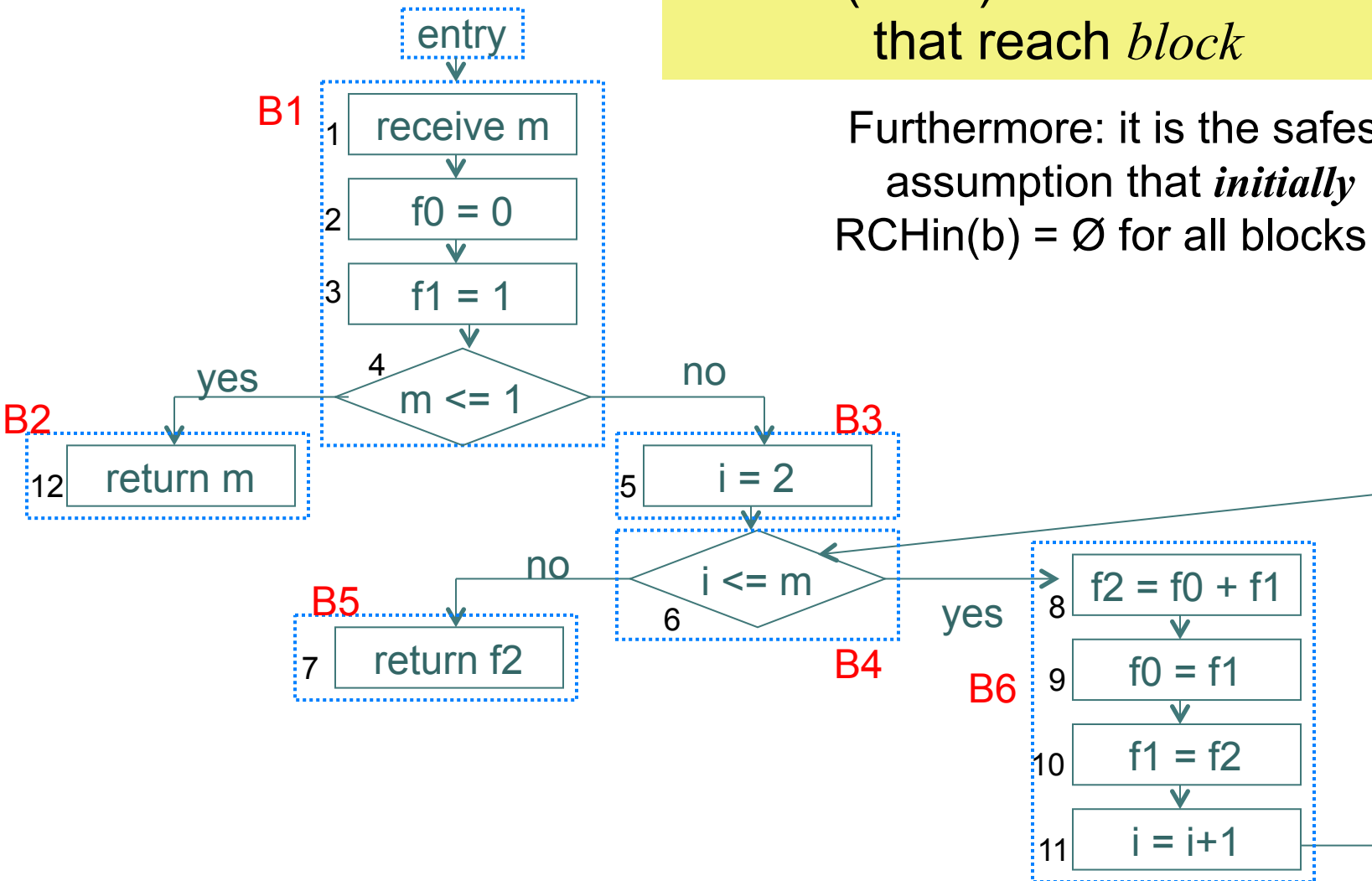
Clearly: $RCHin(entry) = \emptyset$



Reaching Definitions: $RCHin(node)$

$RCHin(block) = \text{set of definitions that reach } block$

Furthermore: it is the safest assumption that *initially* $RCHin(b) = \emptyset$ for all blocks b

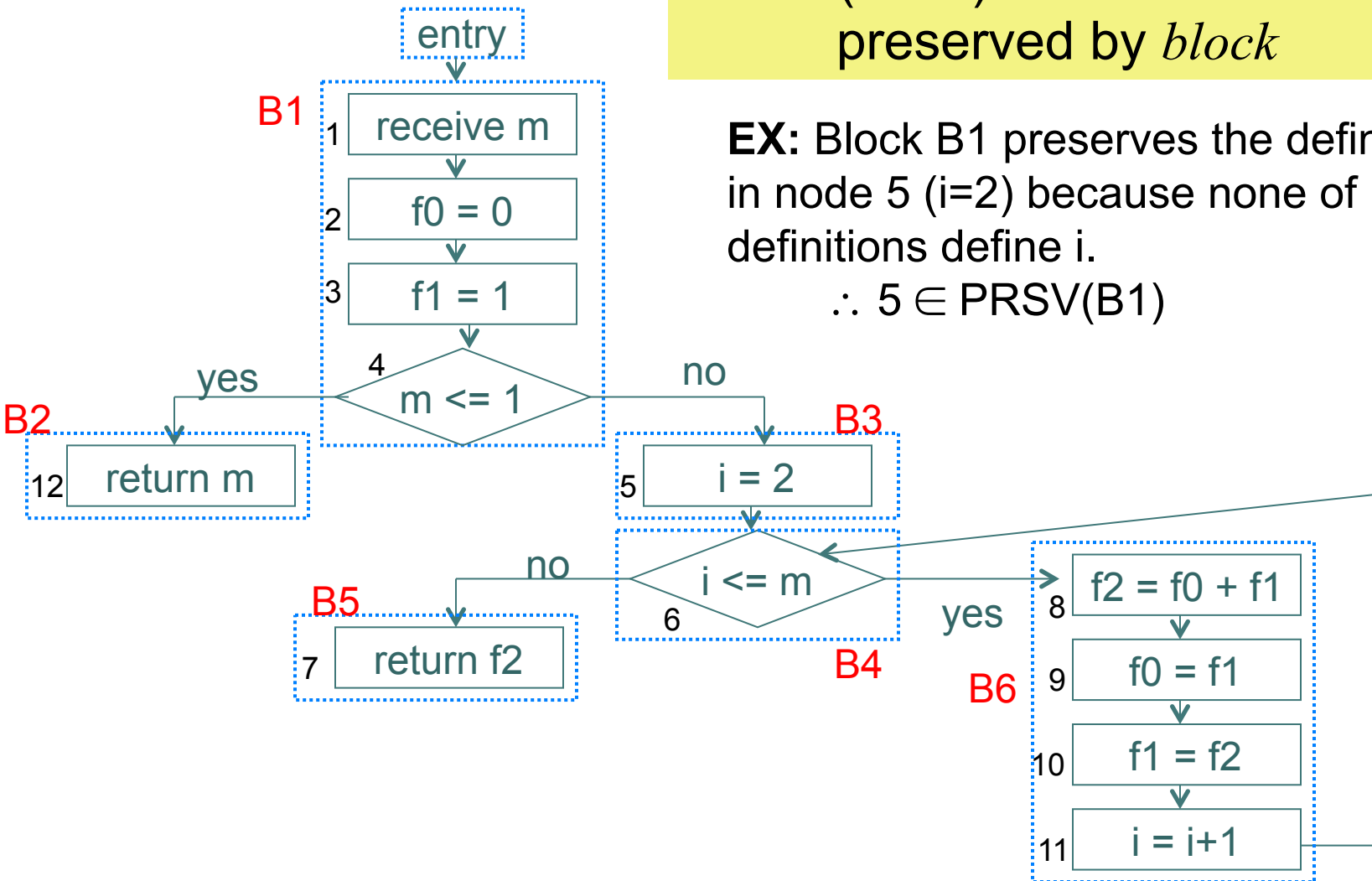


Preserved Definitions: $\text{PRSV}(\text{block})$

$\text{PRSV}(\text{block}) = \text{set of definitions preserved by block}$

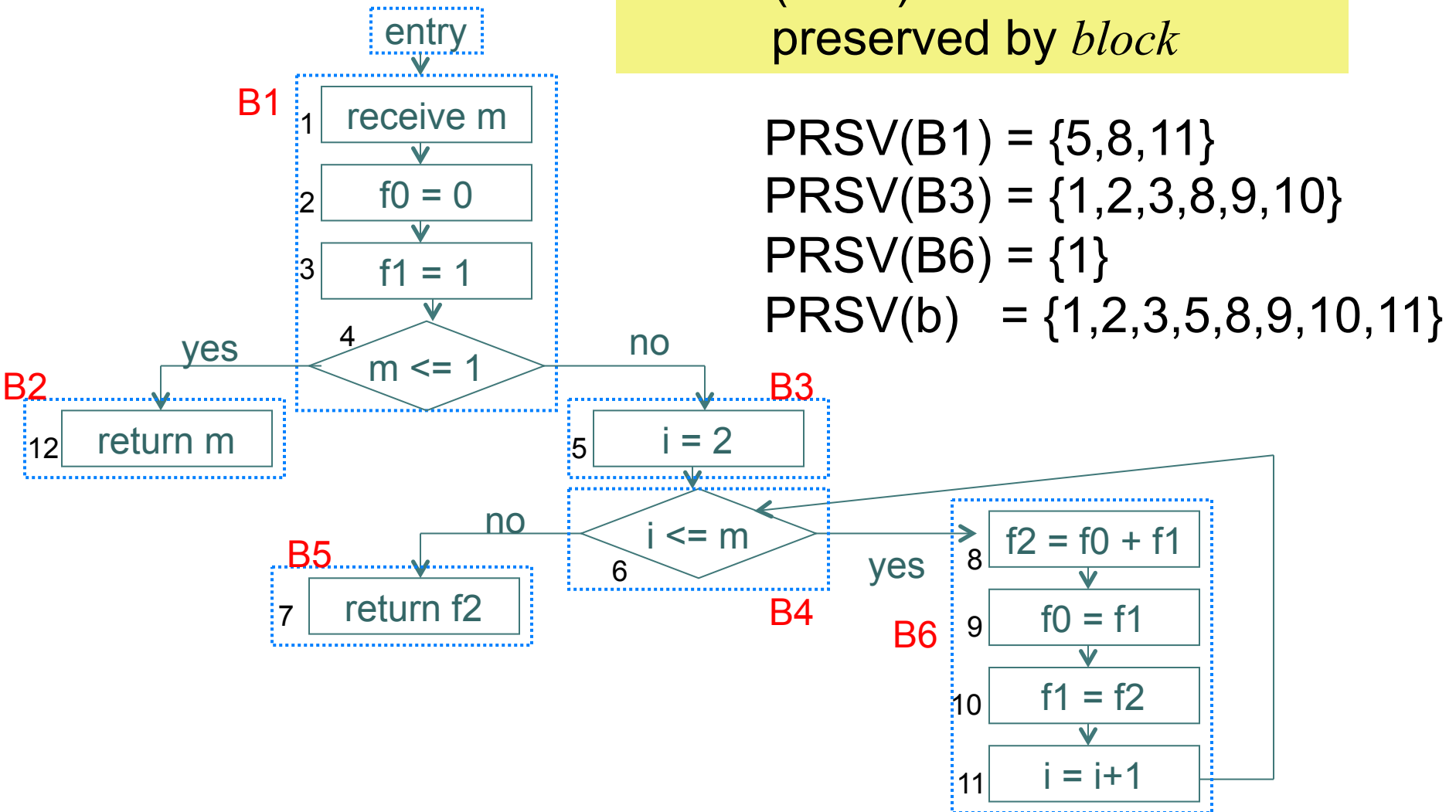
EX: Block B1 preserves the definition in node 5 ($i=2$) because none of B1's definitions define i .

$\therefore 5 \in \text{PRSV}(\text{B1})$



Preserved Definitions: $\text{PRSV}(\text{block})$

$\text{PRSV}(\text{block}) = \text{set of definitions preserved by block}$





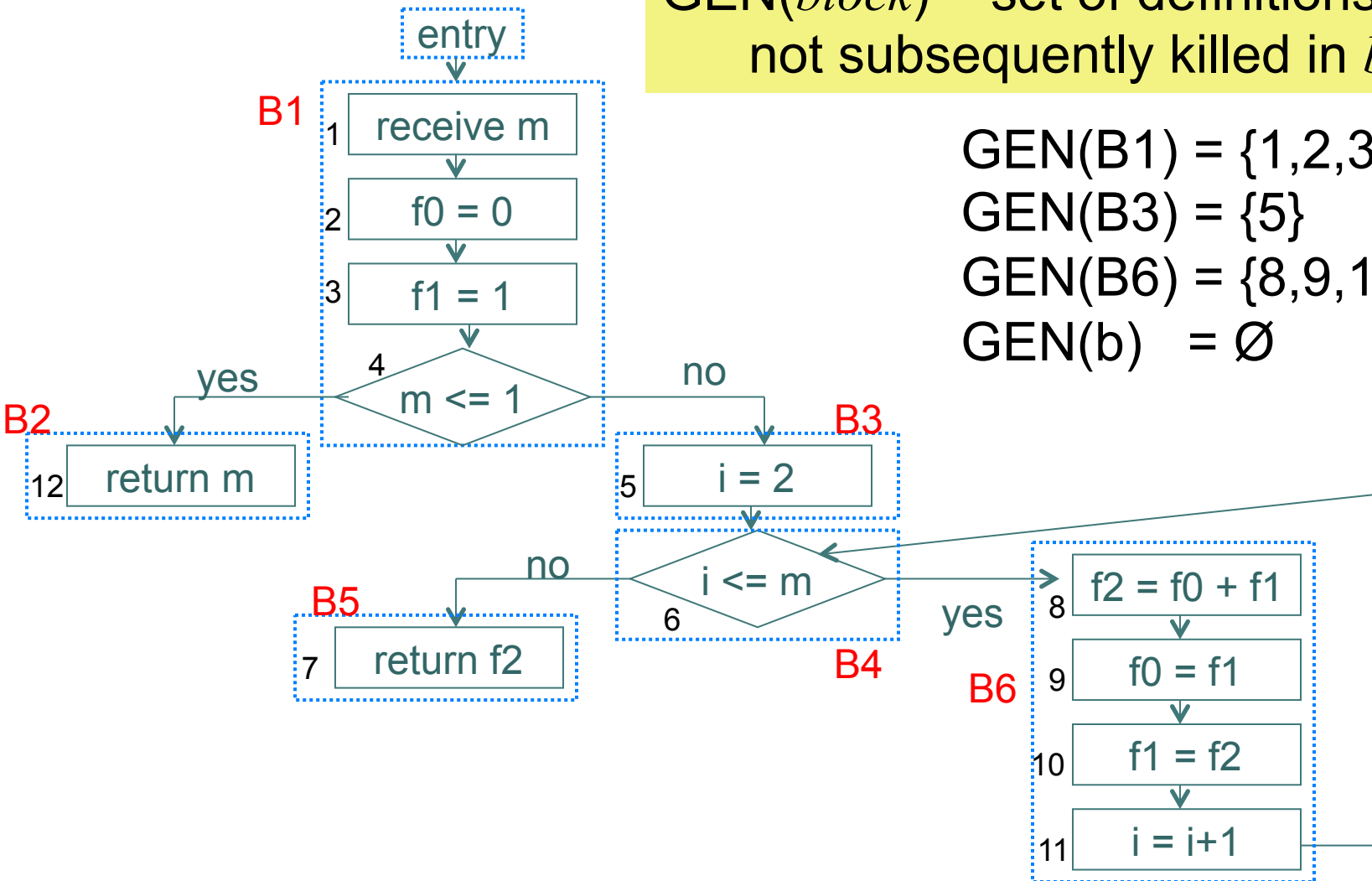
“Killed” Definitions

A definition is said to “kill” another definition if they write to the same location

EX: “ $x = y * z$ ” kills “ $x = 2 + u$ ”

Generated Definitions: $\text{GEN}(\text{block})$

$\text{GEN}(\text{block})$ = set of definitions in *block*
not subsequently killed in *block*



$$\text{GEN}(B1) = \{1, 2, 3\}$$

$$\text{GEN}(B3) = \{5\}$$

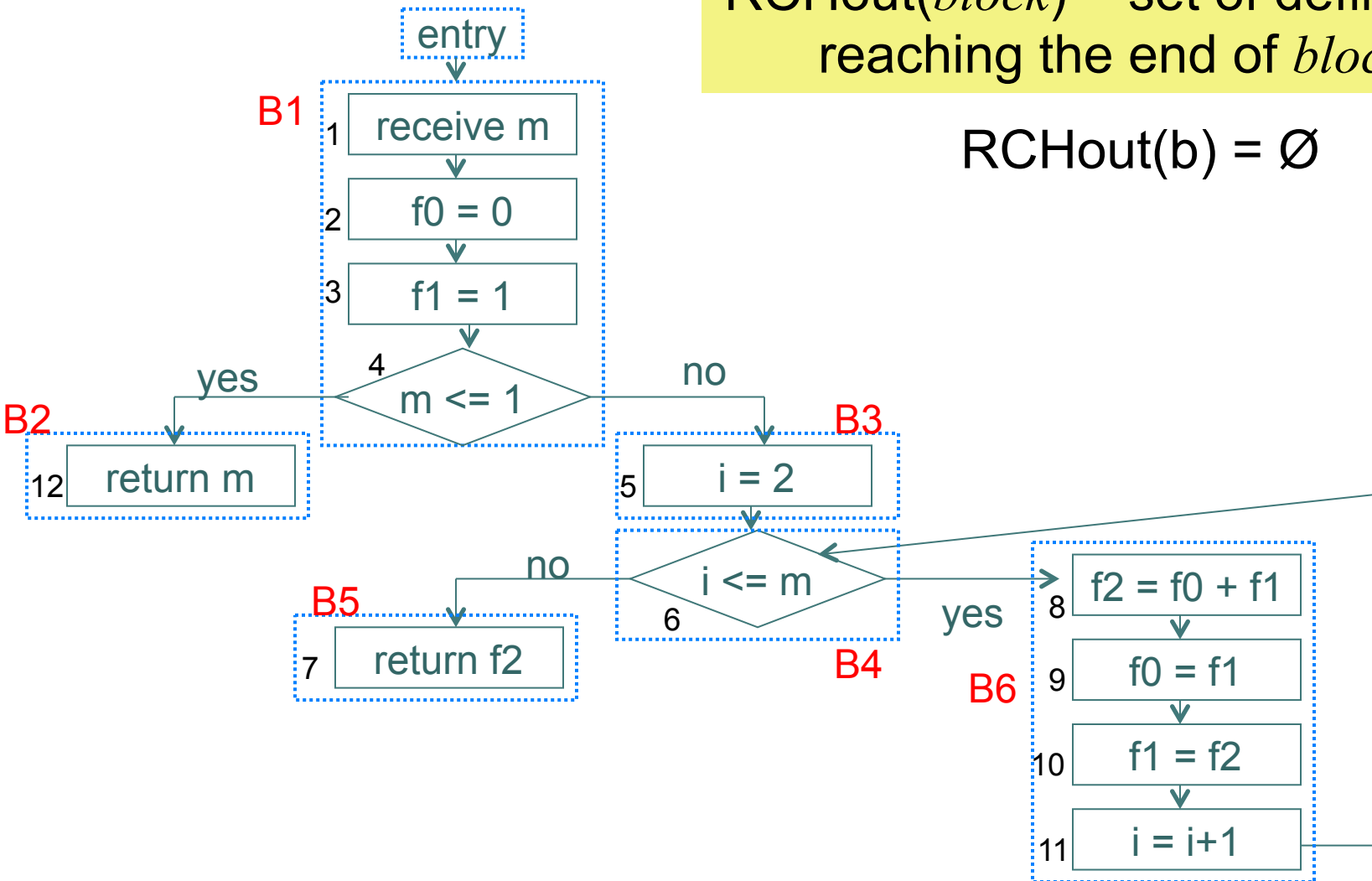
$$\text{GEN}(B6) = \{8, 9, 10, 11\}$$

$$\text{GEN}(b) = \emptyset$$

Out Reaching Definitions: $RCHout(block)$

$RCHout(block)$ = set of definitions reaching the end of *block*

$$RCHout(b) = \emptyset$$





Data Flow Equations

The definitions out of a block are:

- those generated by it and
- those reaching it that are preserved

$$\text{RCHout}(b) = \text{GEN}(b) \cup (\text{RCHin}(b) \cap \text{PRSV}(b)) \text{ for all } b$$

The definitions reaching a block are those out-reaching from its predecessors

$$\text{RCHin}(b) = \bigcup_{p \in \text{Pred}(b)} \text{RCHout}(p) \text{ for all } b$$



Solving Data Flow Equations Iteratively

1. Initialize the attributes
2. Treat the data flow equations as assignments
3. If there has been a change to the computed attributes, go to 2 otherwise halt



Solving Data Flow Equations

Here is a code outline

Initialization code

repeat

$\text{RCHout}(b) := \text{GEN}(b) \cup (\text{RCHin}(b) \cap \text{PRSV}(b))$ *for all* b

$\text{RCHin}(b) := \bigcup_{p \in \text{Pred}(b)} \text{RCHout}(p)$ *for all* b

until

no change to RCHin/RCHout



Next time

- Justifying Iterative Solution
 - I.e., why does this give us a solution?
- Liveness as iterative data flow analysis



Iterative Data Flow Analysis

- Generally, an analysis of data dependency within a program
 - Like liveness
 - Liveness for programs with loops is solved with IDFA
- First, **attributes** are associated with each node/basic block and given initial values
- Second, relationships between these attributes are specified as **data flow equations**
- Third, a solution to these equations is solved by **iteration**



Liveness Analysis

- ...determines when the value within a virtual register may still be used
 - a.k.a. its value is “live”
- ...and when it definitely won't
 - a.k.a. its value is “dead”
- This property, “liveness”, may be **approximated** statically

More Precise Definition of Liveness

Definition

- **assignment** of a value to a variable
- $\text{def}[v]$ = set of nodes that define variable v
- $\text{def}[n]$ = set of variables defined at node n

$a \leftarrow 0$

Use

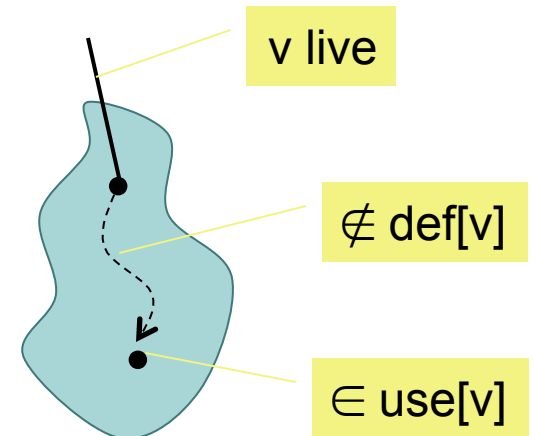
- **reading** the value to a variable
- $\text{use}[v/n]$ = analogous to $\text{def}[v/n]$

$a > 9$

Liveness

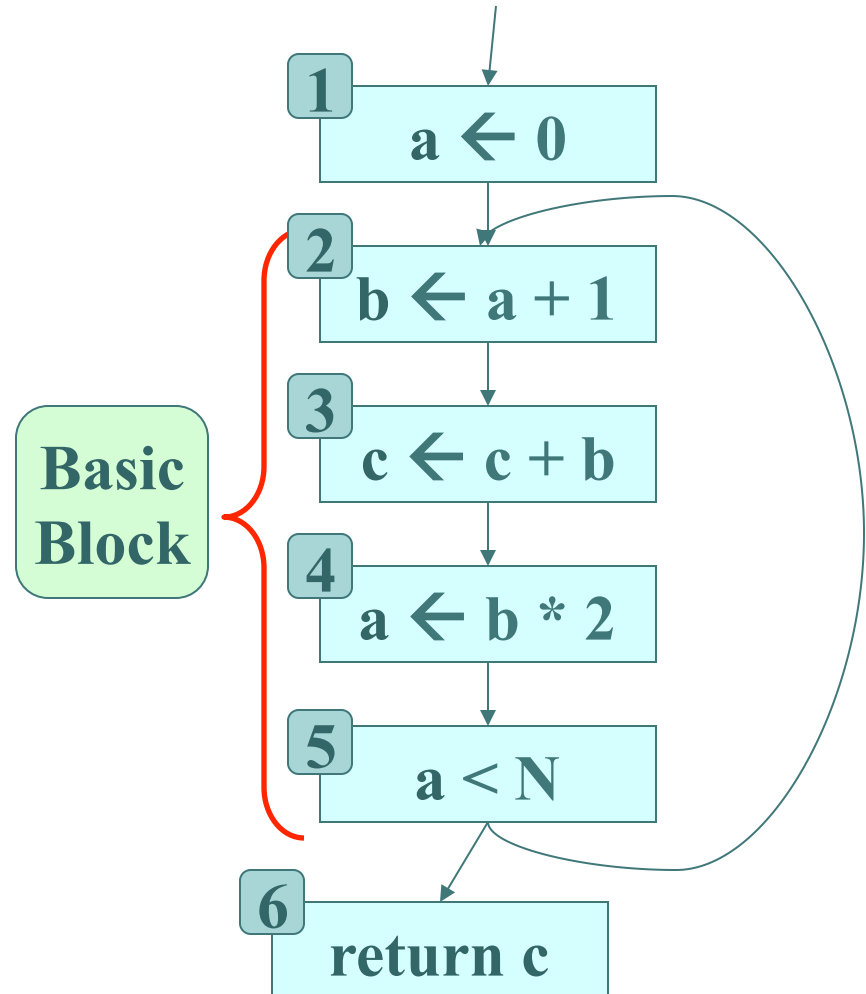
v is **live on a CFG edge** if

1. \exists a directed path from that edge to a use of v
2. The path does not go through any defn of v



Small Control Flow Graph

$a \leftarrow 0$
L: $b \leftarrow a + 1$
 $c \leftarrow c + b$
 $a \leftarrow b * 2$
if ($a < N$) goto L
return c



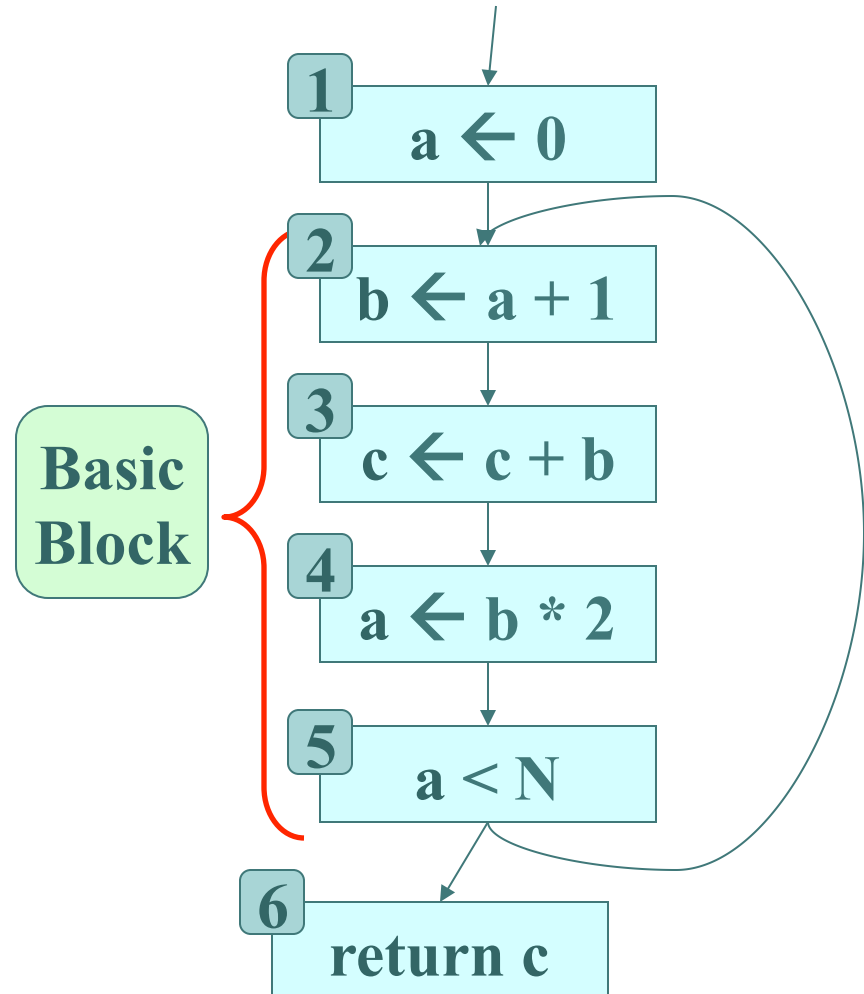
The Flow of Liveness

Data-flow

- Liveness flows through the edges of the CFG

Direction of Flow

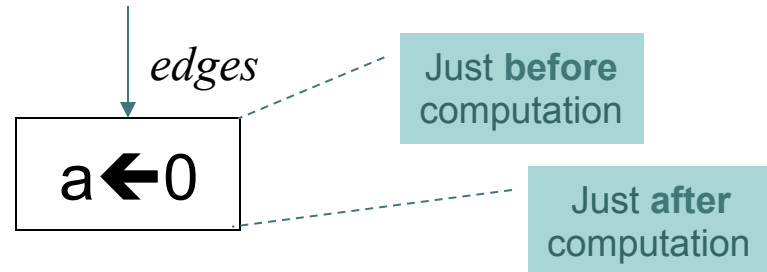
- Liveness flows backwards because
 - Behavior at future nodes determines liveness at given node
- Consider a or b
- “Forward” properties exist (e.g., reaching)



Liveness at Nodes

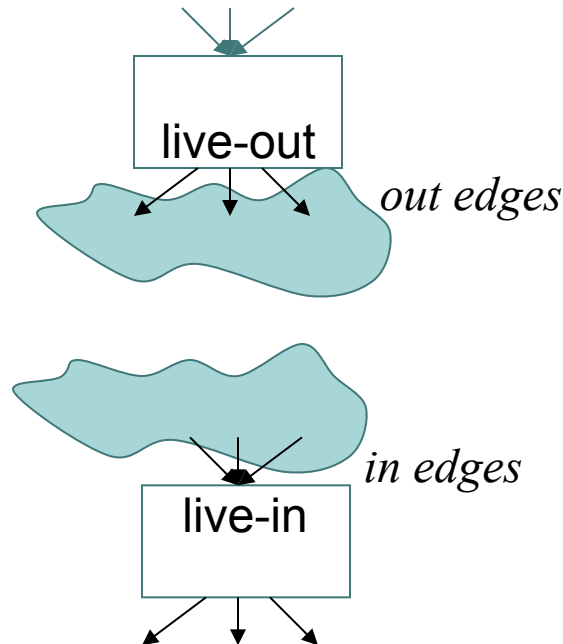
We have liveness at edges

- How do we talk about liveness at nodes?



Two more definitions

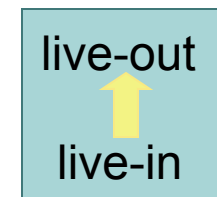
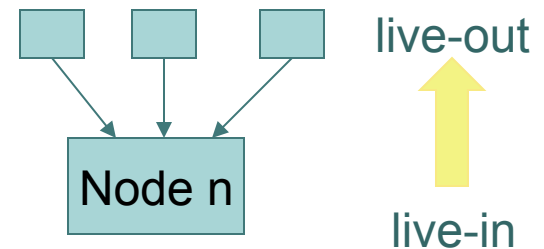
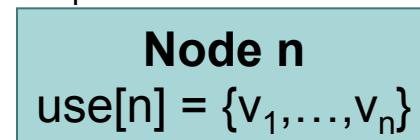
- A variable is **live-out** at a node if it is live on **any** of that node's out-edges
- A variable is **live-in** at a node if it is live on **any** of the node's in-edges



Computing Liveness

1. **Generate liveness:** if a variable is in $\text{use}[n]$, then it is in $\text{live-in}[n]$
2. **Push liveness across edges:** if a variable is in $\text{live-in}[n]$, then it is in live-out for all nodes in $\text{pred}[n]$
3. **Push liveness across nodes:** if a variable is in $\text{live-out}[n]$ and not in $\text{def}[n]$ then the variable is in $\text{live-in}[n]$

v_i are live-in for n





Step 1. The Attributes

1. **Generate liveness:** if a variable is in $\text{use}[n]$, then it is in $\text{live-in}[n]$
2. **Push liveness across edges:** if a variable is in $\text{live-in}[n]$, then it is in live-out for all nodes in $\text{pred}[n]$
3. **Push liveness across nodes:** if a variable is in $\text{live-out}[n]$ and not in $\text{def}[n]$ then the variable is in $\text{live-in}[n]$

$\text{live-in}[n]$	=	} Initial values?
$\text{live-out}[n]$	=	
$\text{use}[n]$	=	
$\text{def}[n]$	=	



Step 1. The Attributes

1. **Generate liveness:** if a variable is in $\text{use}[n]$, then it is in $\text{live-in}[n]$
2. **Push liveness across edges:** if a variable is in $\text{live-in}[n]$, then it is in live-out for all nodes in $\text{pred}[n]$
3. **Push liveness across nodes:** if a variable is in $\text{live-out}[n]$ and not in $\text{def}[n]$ then the variable is in $\text{live-in}[n]$

$\text{live-in}[n] = \emptyset$

$\text{live-out}[n] = \emptyset$

$\text{use}[n] =$

$\text{def}[n] =$

} constant, determined by node n



Step 2. Data Flow Equations

1. **Generate liveness:** if a variable is in $\text{use}[n]$, then it is in $\text{live-in}[n]$
2. **Push liveness across edges:** if a variable is in $\text{live-in}[n]$, then it is in live-out for all nodes in $\text{pred}[n]$
3. **Push liveness across nodes:** if a variable is in $\text{live-out}[n]$ and not in $\text{def}[n]$ then the variable is in $\text{live-in}[n]$

$\text{live-in}[n] = ?$

$\text{live-out}[n] = ?$



Step 2. Data Flow Equations

1. **Generate liveness:** if a variable is in $\text{use}[n]$, then it is in $\text{live-in}[n]$
2. **Push liveness across edges:** if a variable is in $\text{live-in}[n]$, then it is in live-out for all nodes in $\text{pred}[n]$
3. **Push liveness across nodes:** if a variable is in $\text{live-out}[n]$ and not in $\text{def}[n]$ then the variable is in $\text{live-in}[n]$

$$\text{live-in}[n] = \text{use}[n] \cup (\text{live-out}[n] - \text{def}[n])$$

$$\text{live-out}[n] = \bigcup_{s \in \text{succ}[n]} \text{live-in}[s]$$

Step 3. Solving DFE Iteratively

foreach node n in CFG

$in[n] = \emptyset ; out[n] = \emptyset$

initialization

repeat

foreach node n in CFG

$in'[n] = in[n]$

$out'[n] = out[n]$

$in[n] = use[n] \cup (out[n] - def[n])$

$out[n] = \bigcup_{s \in succ[n]} in[s]$

save current results

iterative solution

until $in'[n] == in[n] \ \& \ out'[n] == out[n]$ for all nodes n

convergence
test

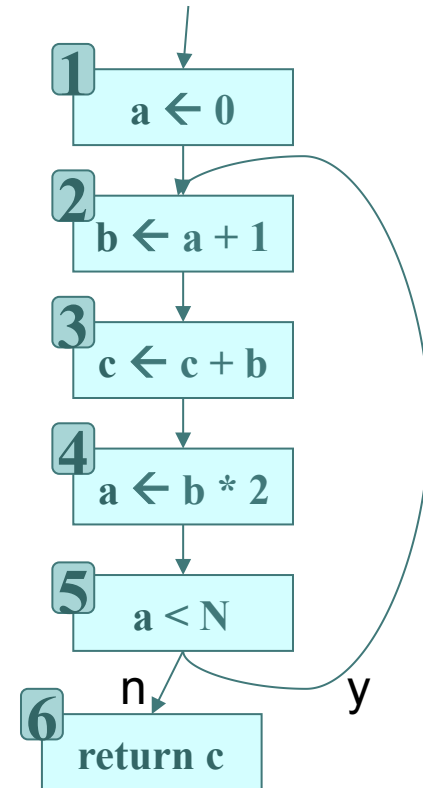
Example

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

node	use	def	iteration step													
			1	2	3	4	5	6	7							
1	a			a	a	ac	c	ac	c	ac	c	ac				
2	a	b	a	a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc	
3	bc	c	bc	bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc	
4	b	a	b	b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac	
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	
6	c		c	c	c	c	c	c	c	c	c	c	c	c	c	

in = red
out = blue



Example (cont' d)

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

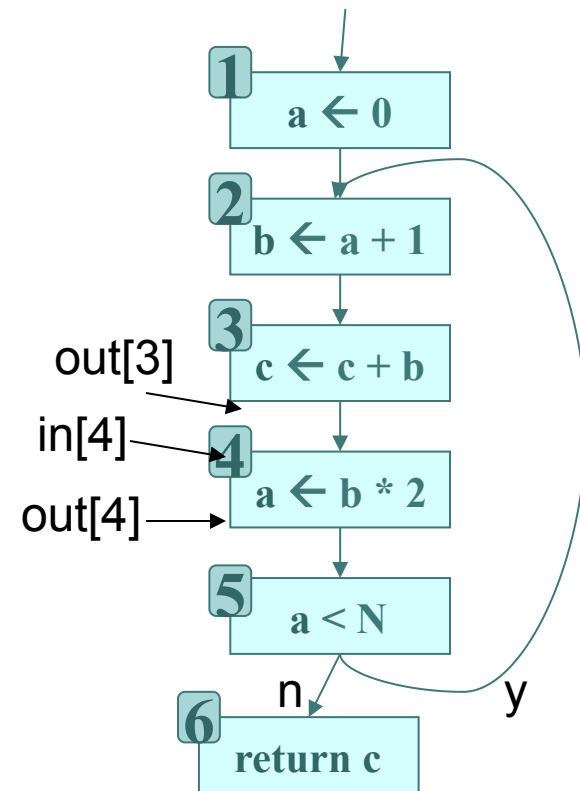
$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

Improving Performance

consider the (3→4) edge in the graph:

out[4] is used to compute in[4],
in[4] is used to compute out[3],...

So, we should compute in the
order: out[4], in[4], out[3], in[3],...



Order of computation should follow flow direction

Step 3. Solving DFE Iteratively revisited

foreach node n in CFG

$\text{in}[n] = \emptyset ; \text{out}[n] = \emptyset$

} initialization

repeat

foreach node n in CFG in **reverse topological sort order**

$\text{in}'[n] = \text{in}[n]$

$\text{out}'[n] = \text{out}[n]$

$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$

$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$

} save current results

} Note the change in order

until $\text{in}'[n] == \text{in}[n] \ \& \ \text{out}'[n] == \text{out}[n]$ for all nodes n

} convergence
test

Example

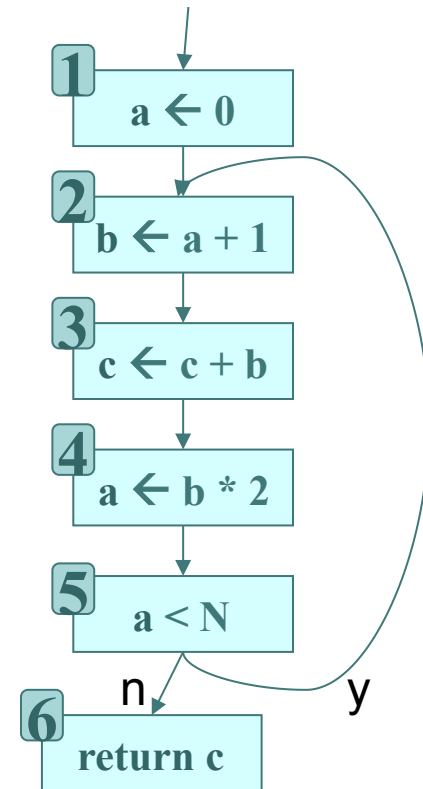
$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

node	use	def	1	2	3	4	5	6	7
6	c		c	c	c				
5	a		c ac	ac	ac	ac	ac		
4	b	a	ac bc	ac bc	ac bc	bc			
3	bc	c	bc bc	bc bc	bc bc	bc	bc		
2	a	b	bc ac	bc ac	bc ac	bc	ac		
1	a		ac c	ac c	ac c	c			

out = red
in = blue

Note the change in order!





Time Complexity (rough)

- Consider a program of size N
 - $N = \max(\text{nodes in CFG, number of vars})$
 - \therefore each live-in, live-out set has at most N elements
 - Each set union takes $O(N)$ time

Step 3. Solving DFE Iteratively revisited

foreach node n in CFG

$in[n] = \emptyset ; out[n] = \emptyset$

} $O(N)$

repeat

foreach node n in CFG in reverse topological sort order

$in'[n] = in[n]$

$out'[n] = out[n]$

$out[n] = \bigcup_{s \in succ[n]} in[s]$

$in[n] = use[n] \cup (out[n] - def[n])$

} $O(N^2)$

} $O(N^2)$

until $in'[n] == in[n] \ \& \ out'[n] == out[n]$ for all nodes n } $O(N)$

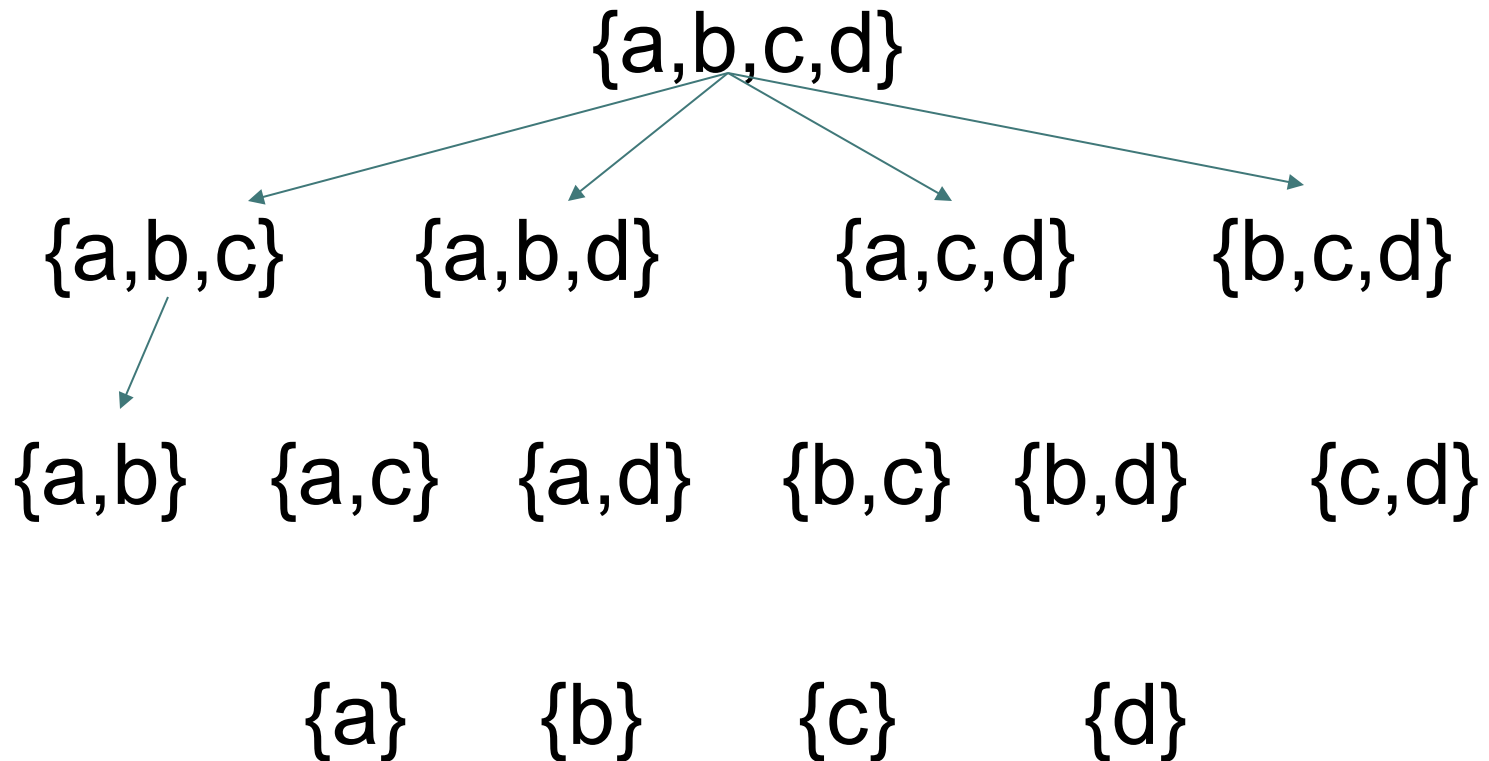
\therefore Worst case is $O(N^2 \times N^2) = O(N^4)$



More Performance Considerations

- Use basic blocks instead of nodes
 - Merge nodes into basic blocks to decrease size of CFG
- Representation of sets
 - For dense sets, use a bit vector representation
 - This can reduce the cost of set operations to $O(1)$
 - For sparse sets, use sorted (linked) lists
- Typical Case: 2 to 3 iterations with good ordering and sparse sets
 - In the $O(N)$ to $O(N^2)$ range for average

Termination Guarantees: possible values of live-in (for example)



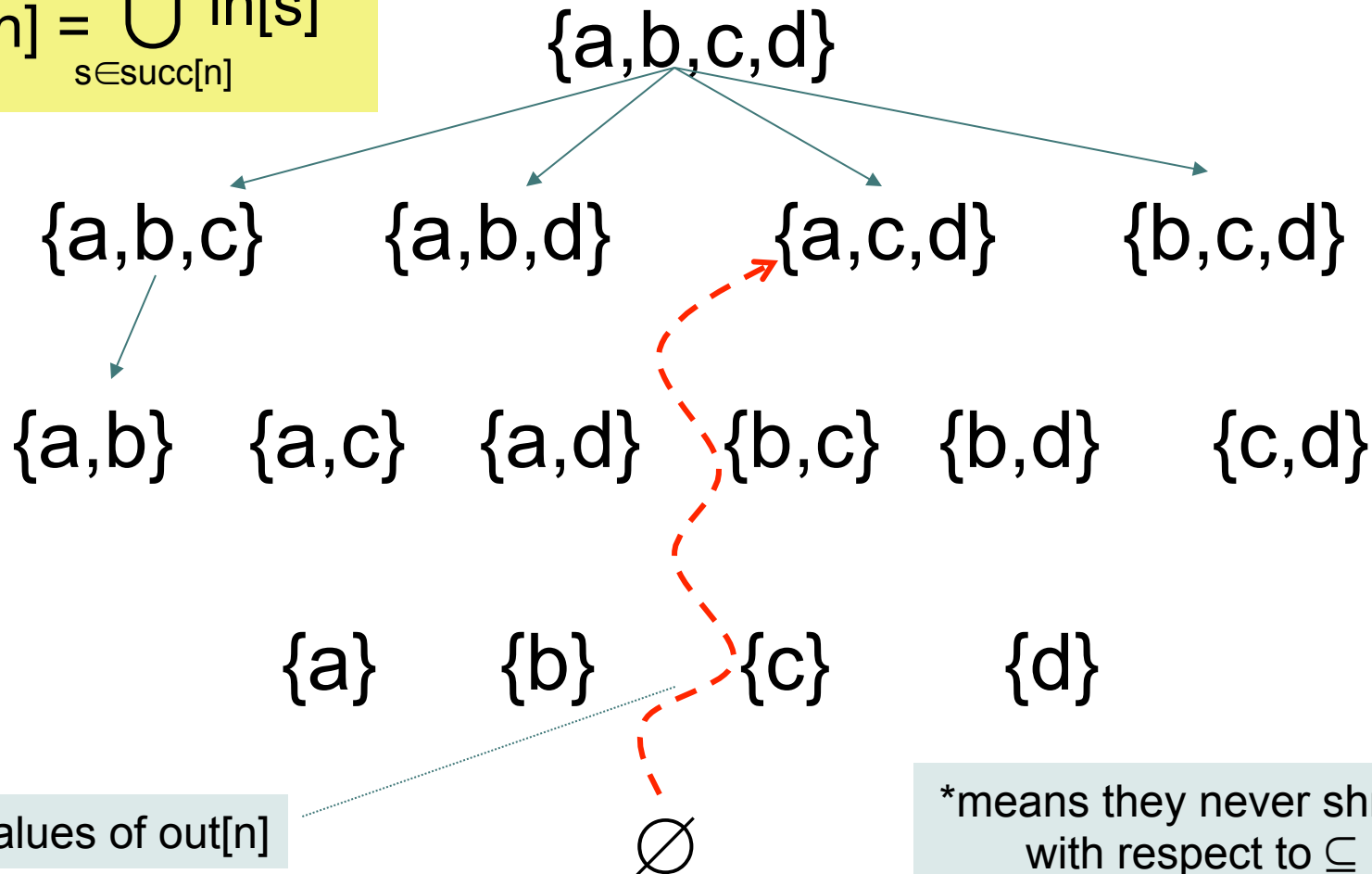
*Not all links shown



→ means is a “subset of”

Attribute values are **monotonically increasing***

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$



*means they never shrink
with respect to \subseteq