

Dataflow analysis

Dataflow analysis: what is it?

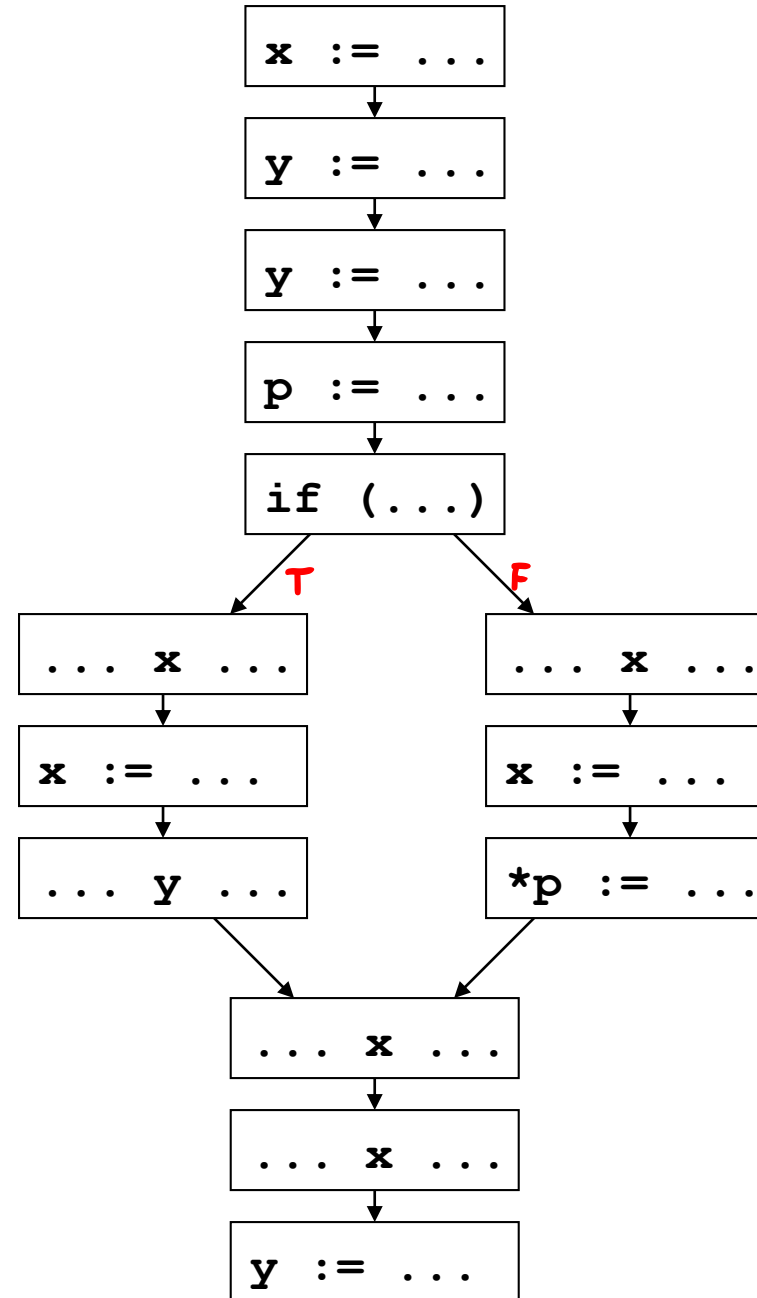
- A common framework for expressing algorithms that compute information about a program
- Why is such a framework useful?
- Provides a common language, which makes it easier to:
 - communicate your analysis to others
 - compare analyses
 - adapt techniques from one analysis to another
 - reuse implementations (eg: dataflow analysis frameworks)

Control Flow Graphs

- For now, we will use a Control Flow Graph representation of programs
 - each statement becomes a node
 - edges between nodes represent control flow
- Later we will see other program representations
 - variations on the CFG (eg CFG with basic blocks)
 - other graph based representations

Example CFG

```
x := ...
y := ...
y := ...
p := ...
if (...) {
  ... x ...
  x := ...
  ... y ...
}
else {
  ... x ...
  x := ...
  *p := ...
}
... x ...
... y ...
y := ...
```



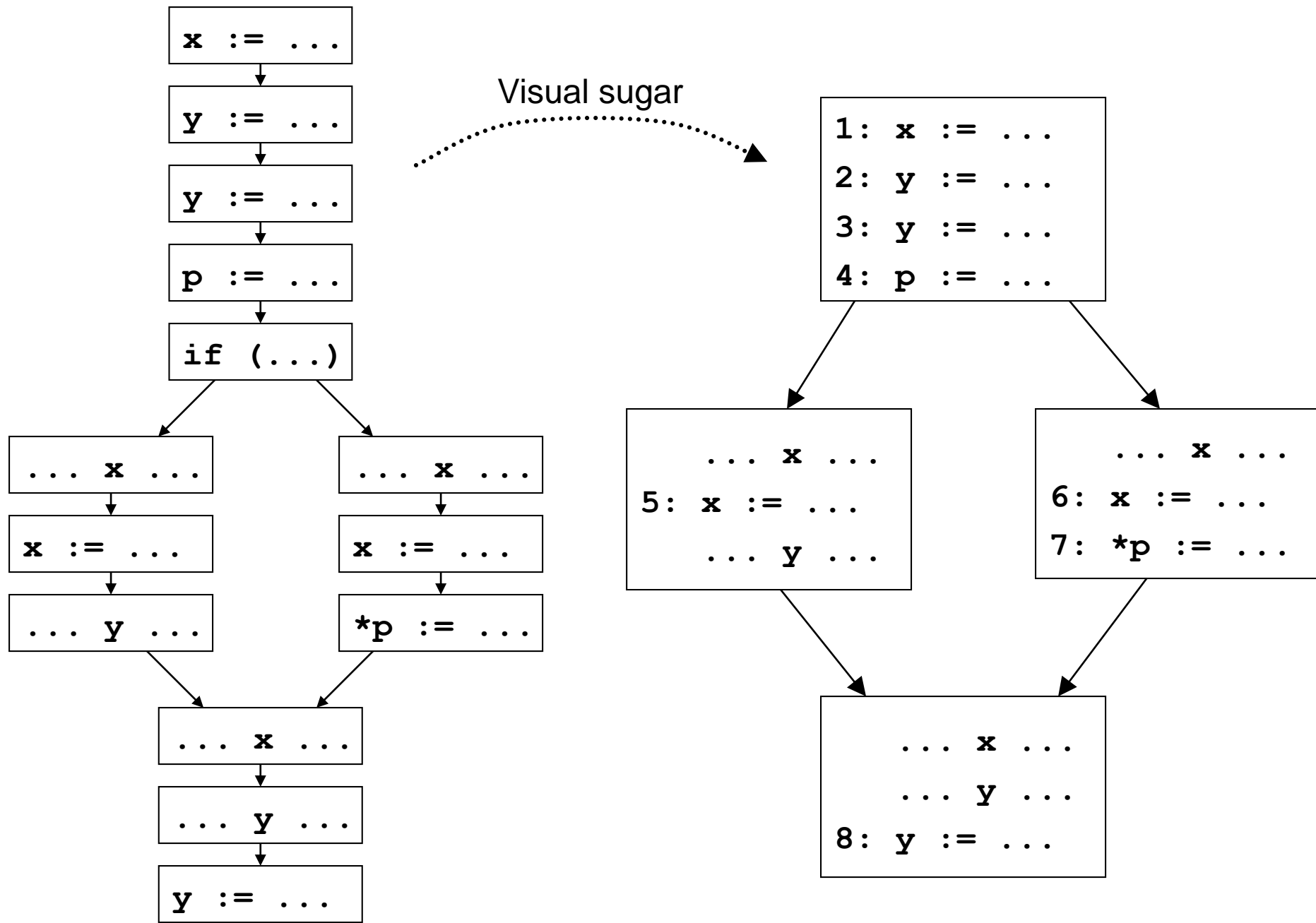
An example DFA: reaching definitions

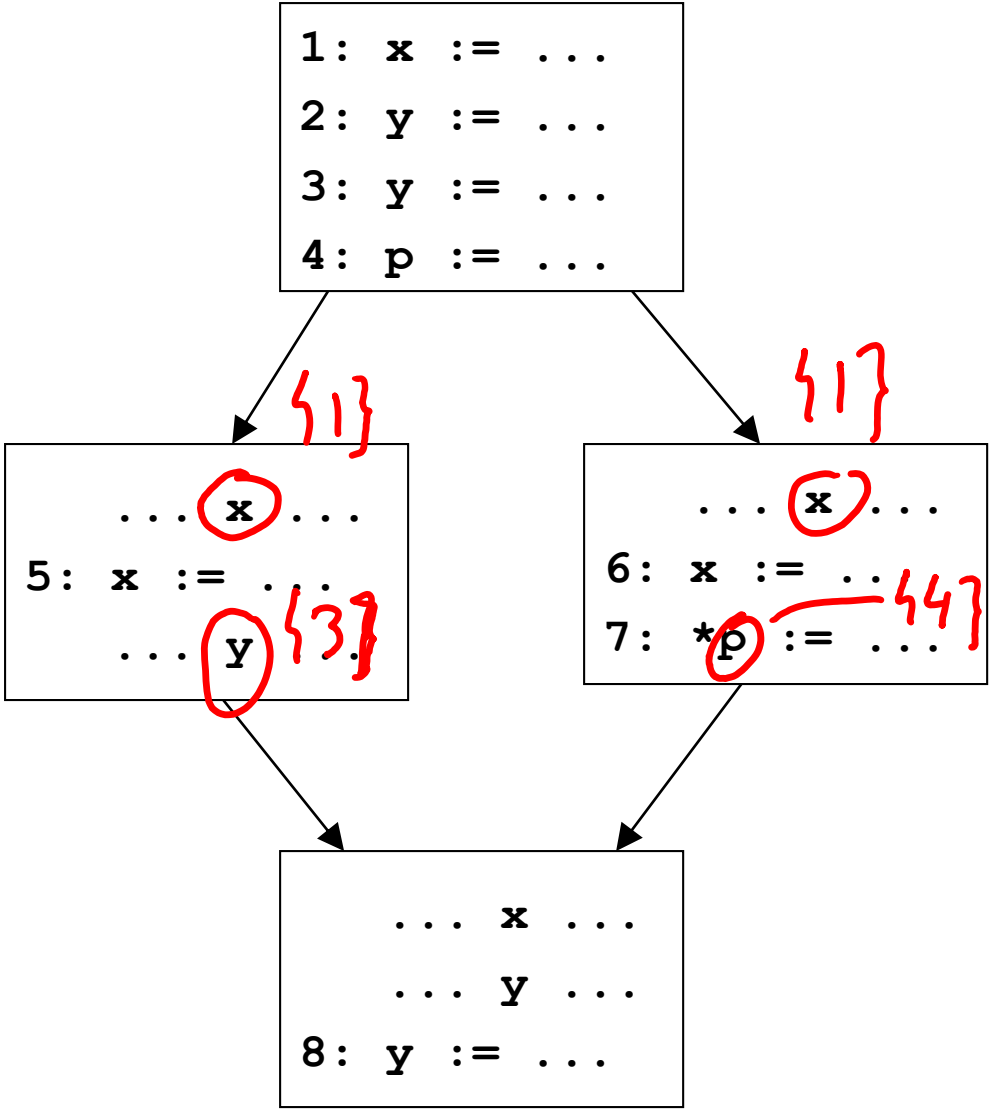
- For each use of a variable, determine what assignments could have set the value being read from the variable
- Information useful for:
 - performing constant and copy prop
 - detecting references to undefined variables
 - presenting “def/use chains” to the programmer
 - building other representations, like the DFG
- Let’s try this out on an example

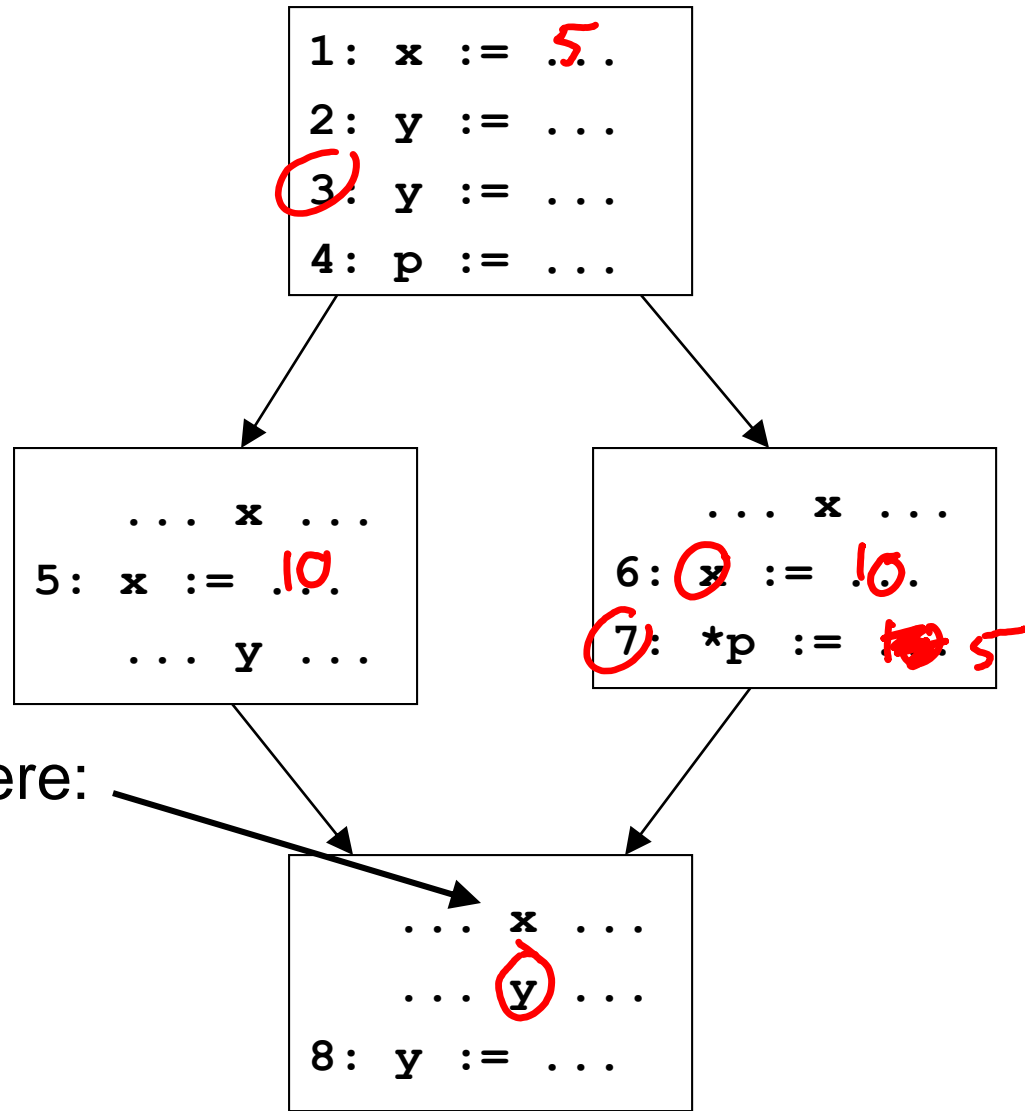
A. Yes

B. No

Be Honest

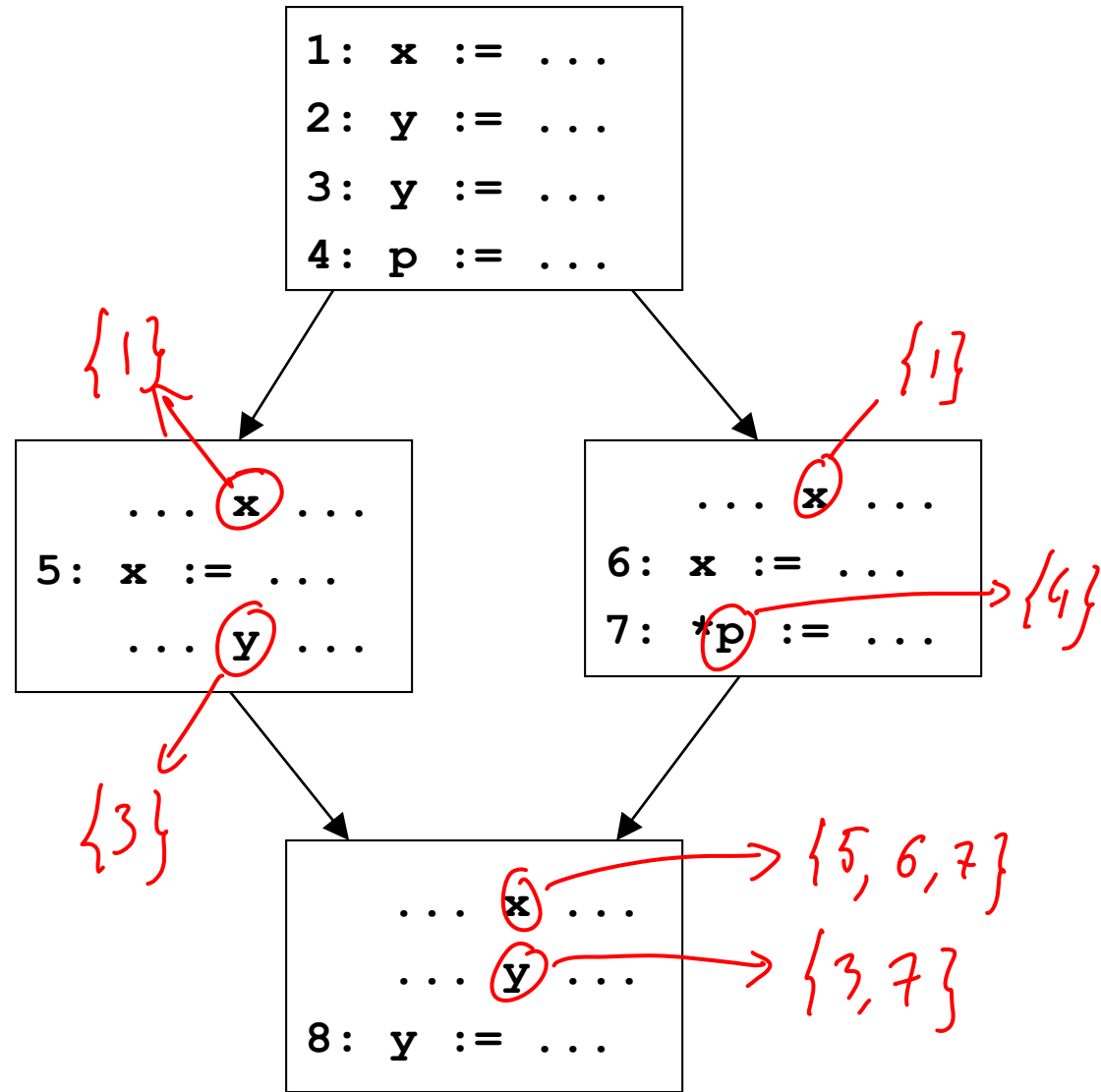






What is reaching defn set for x here:

- A. { 5 }
- B. { 6 }
- C. { 5, 6 }
- D. { 5, 6, 7 }
- E. { 5, 6, 7, 1 }



Safety

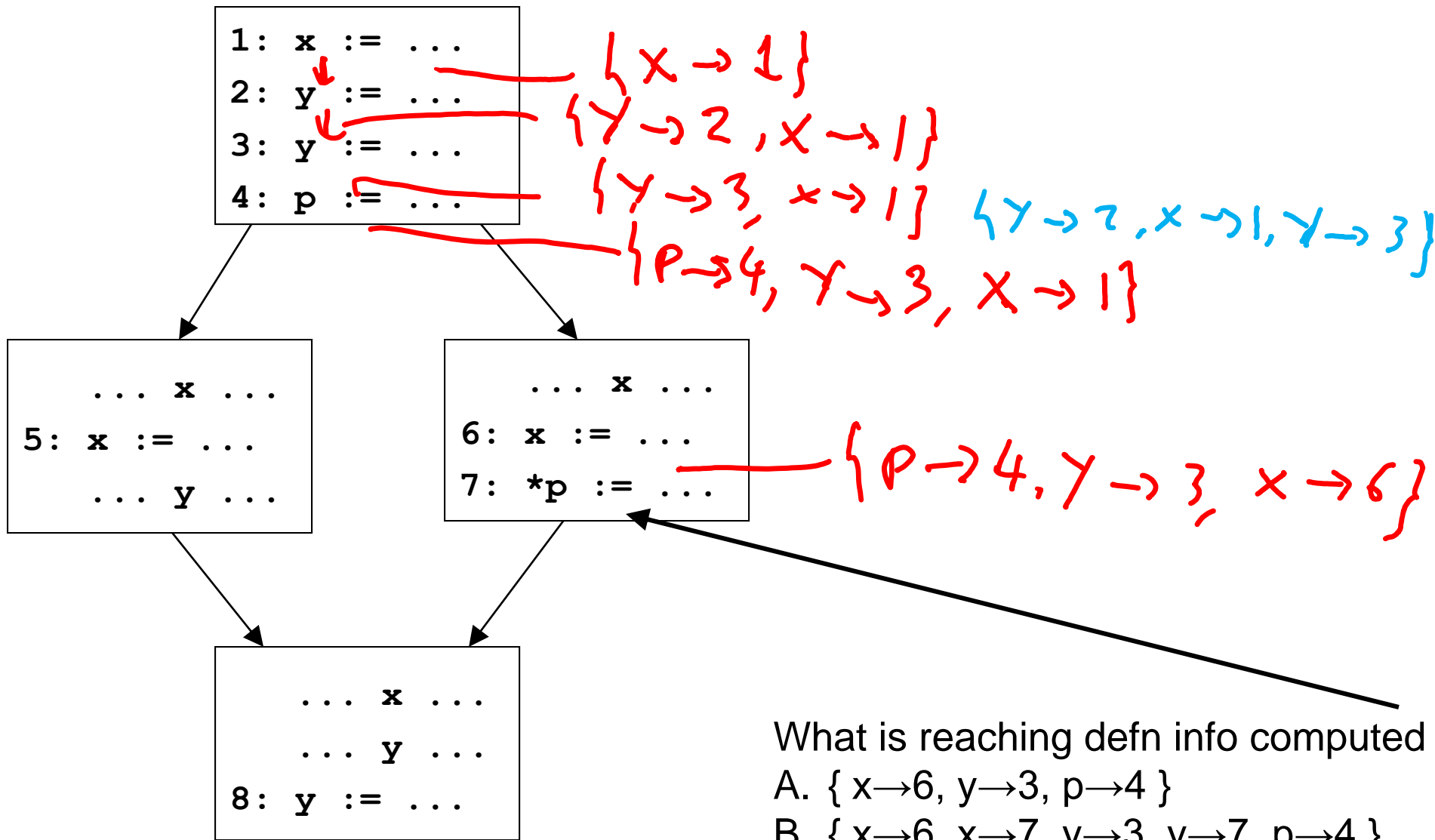
- When is computed info safe?
- Recall intended use of this info:
 - performing constant and copy prop
 - detecting references to undefined variables
 - presenting “def/use chains” to the programmer
 - building other representations, like the DFG
- Safety:
 - can have more bindings than the “true” answer, but can’t miss any

Reaching definitions generalized

- DFA framework geared to computing information at each program point (edge) in the CFG
 - So generalize problem by stating what should be computed at each program point
- For each program point in the CFG, compute the set of definitions (statements) that may reach that point
- Notion of safety remains the same

Reaching definitions generalized

- Computed information at a program point is a set of var \rightarrow stmt bindings
 - eg: $\{ x \rightarrow s_1, x \rightarrow s_2, y \rightarrow s_3 \}$
- How do we get the previous info we wanted?
 - if a var x is used in a stmt whose incoming info is in ,
 - then: $\{ s \mid (x \rightarrow s) \in in \}$
- This is a common pattern
 - generalize the problem to define what information should be computed at each program point
 - use the computed information at the program points to get the original info we wanted

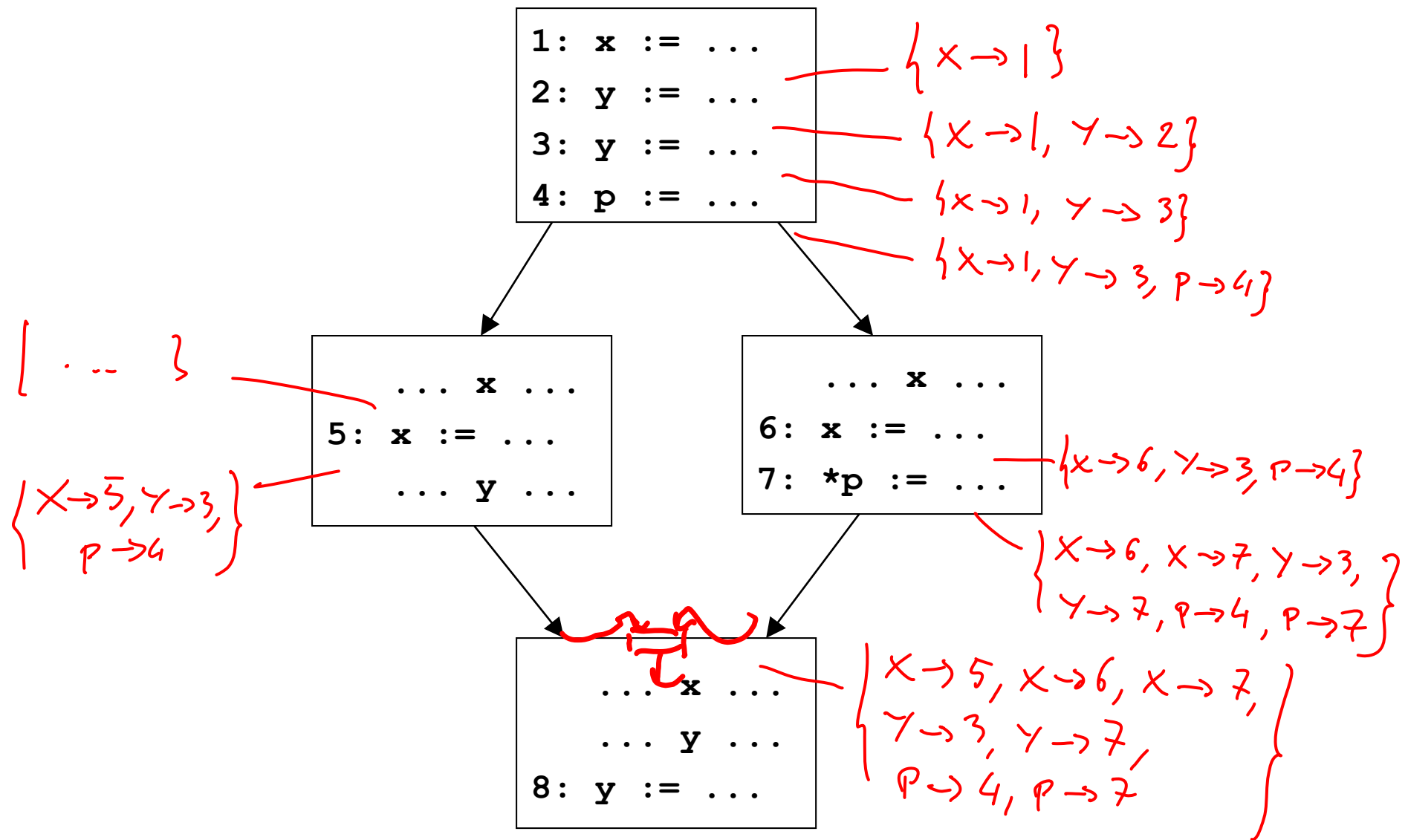


What is reaching defn info computed here

A. $\{x \rightarrow 6, y \rightarrow 3, p \rightarrow 4\}$

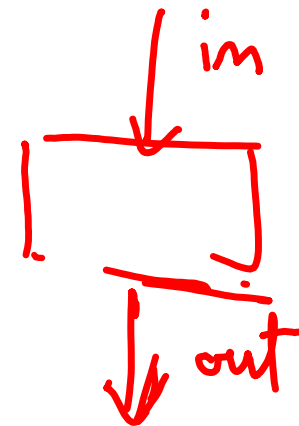
B. $\{x \rightarrow 6, x \rightarrow 7, y \rightarrow 3, y \rightarrow 7, p \rightarrow 4\}$

C. None of the above $p \rightarrow 7$

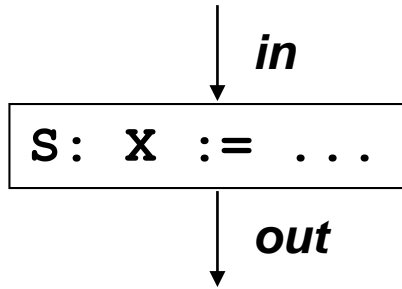


Using constraints to formalize DFA

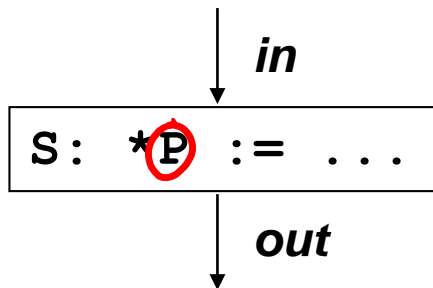
- Now that we've gone through some examples, let's try to precisely express the algorithms for computing dataflow information
- We'll model DFA as solving a system of constraints
- Each node in the CFG will impose constraints relating information at predecessor and successor points
- Solution to constraints is result of analysis



Constraints for reaching definitions



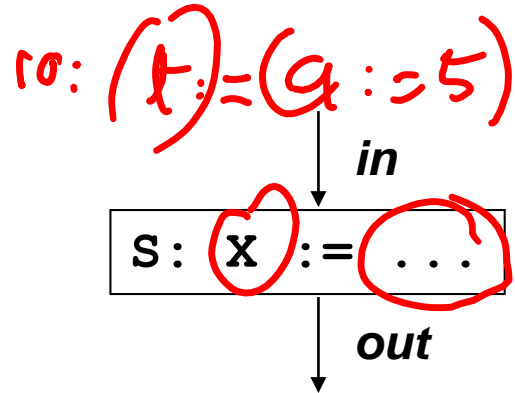
$$out = in - \{x \rightarrow *\} \cup \{x \rightarrow S\}$$



$modPt(P)$

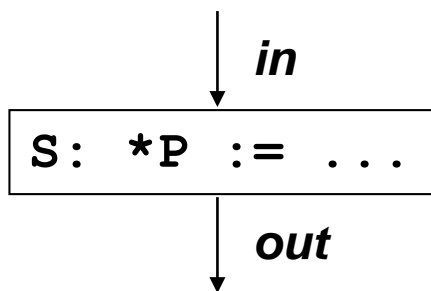
$$out = in \cup \{x \rightarrow S \mid x \in modPt(P)\}$$

Constraints for reaching definitions



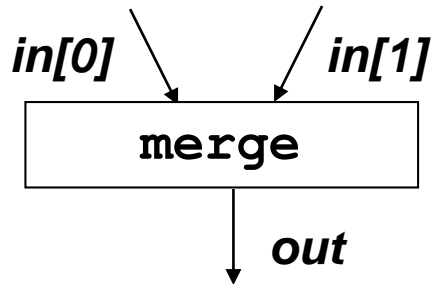
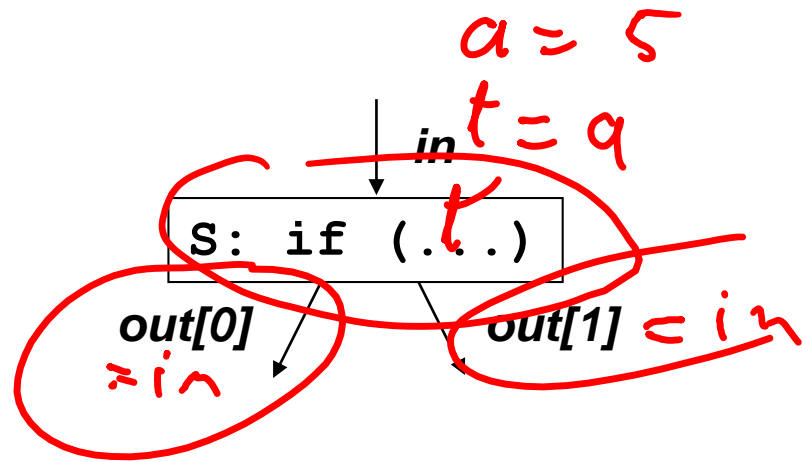
$\{t \rightarrow 10\}$

$out = in - \{X \rightarrow S' \mid S' \in \text{stmts}\} \cup \{X \rightarrow S\}$



- Using may-point-to information:
 $out = in \cup \{X \rightarrow S \mid X \in \text{may-point-to}(P)\}$
- Using must-point-to as well:
 $out = in - \{X \rightarrow S' \mid X \in \text{must-point-to}(P) \wedge S' \in \text{stmts}\} \cup \{X \rightarrow S \mid X \in \text{may-point-to}(P)\}$

Constraints for reaching definitions

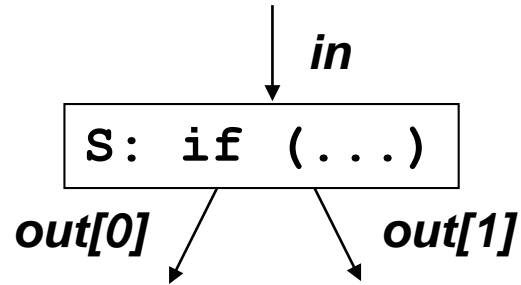


A. \cup

B. \cap

C. Neither

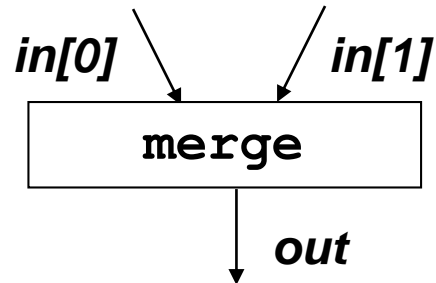
Constraints for reaching definitions



$$out[0] = in \wedge$$

$$out[1] = in$$

more generally: $\forall i. out[i] = in$



$$out = in[0] \cup in[1]$$

more generally: $out = \bigcup_i in[i]$

Flow functions

- The constraint for a statement kind s often have the form: $out = F_s(in)$
- F_s is called a flow function
 - other names for it: dataflow function, transfer function
- Given information in before statement s , $F_s(in)$ returns information after statement s
- Other formulations have the statement s as an explicit parameter to F : given a statement s and some information in , $F(s, in)$ returns the outgoing information after statement s

Flow functions, some issues

- Issue: what does one do when there are multiple input edges to a node?
- Issue: what does one do when there are multiple outgoing edges to a node?

Flow functions, some issues

- Issue: what does one do when there are multiple input edges to a node?
 - the flow function takes as input a tuple of values, one value for each incoming edge
- Issue: what does one do when there are multiple outgoing edges to a node?
 - the flow function returns a tuple of values, one value for each outgoing edge
 - can also have one flow function per outgoing edge

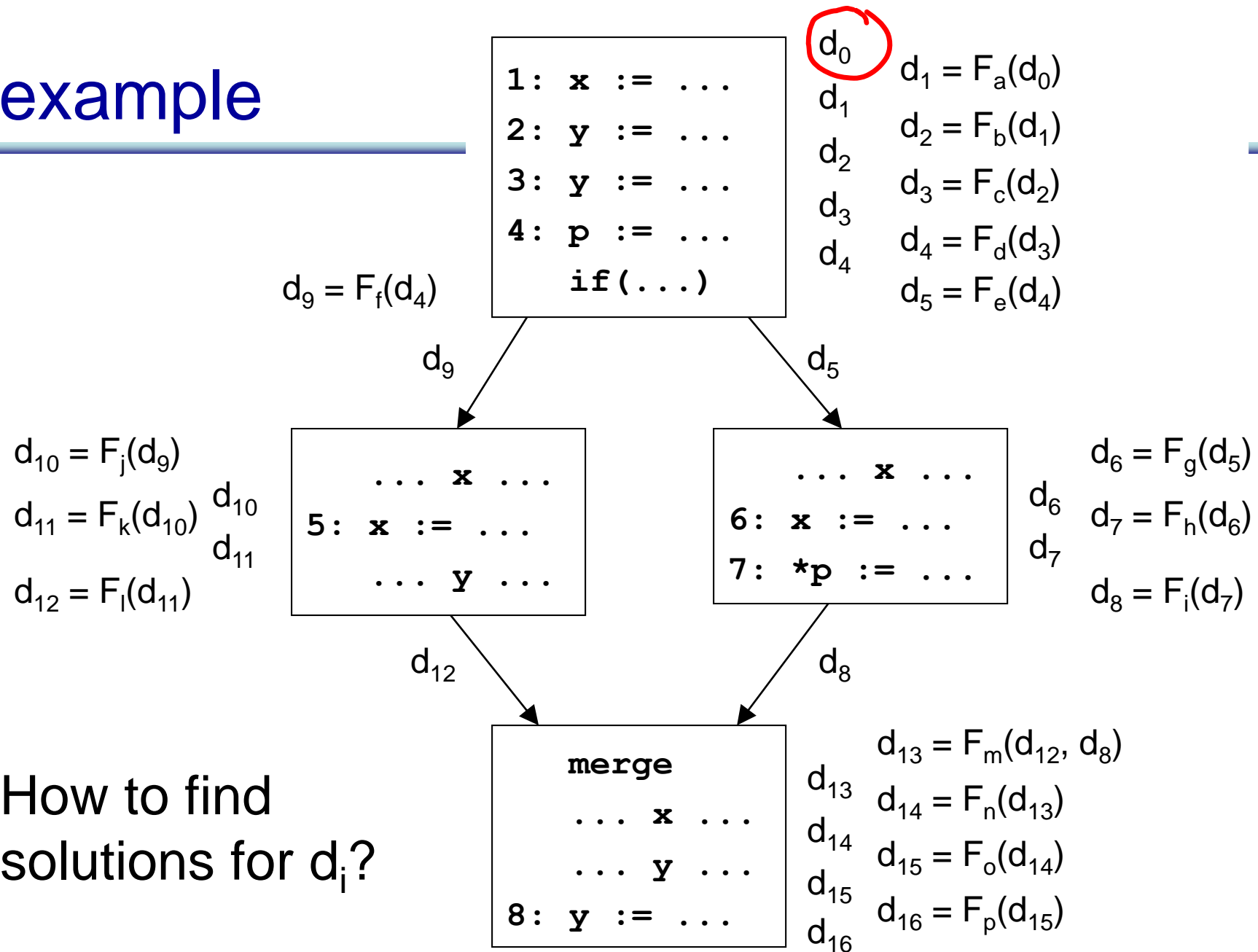
Flow functions

- Flow functions are a central component of a dataflow analysis
- They state constraints on the information flowing into and out of a statement
- This version of the flow functions is local
 - it applies to a particular statement kind
 - we'll see global flow functions shortly...

Summary of flow functions

- Flow functions: Given information *in* before statement *s*, $F_s(in)$ returns information after statement *s*
- Flow functions are a central component of a dataflow analysis
- They state constraints on the information flowing into and out of a statement

Back to example

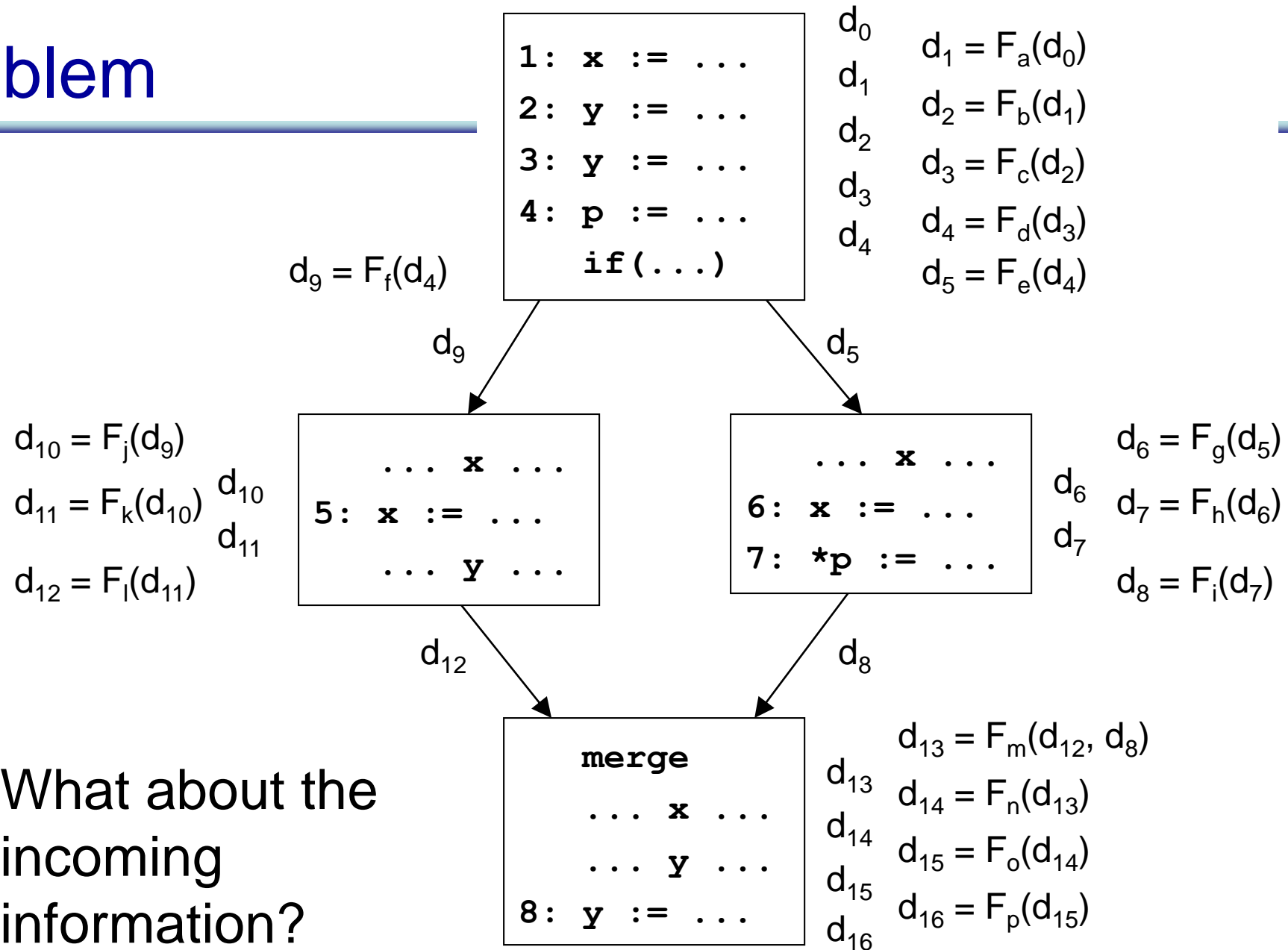


How to find solutions for d_i ?

How to find solutions for d_i ?

- This is a forward problem
 - given information flowing *in* to a node, can determine using the flow function the info flow *out* of the node
- To solve, simply propagate information forward through the control flow graph, using the flow functions
- What are the problems with this approach?

First problem

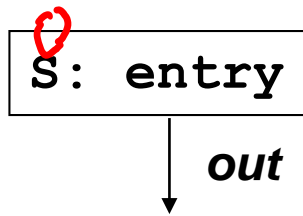


What about the incoming information?

First problem

- What about the incoming information?
 - d_0 is not constrained
 - so where do we start?
- Need to constrain d_0
- Two options:
 - explicitly state entry information
 - have an entry node whose flow function sets the information on entry (doesn't matter if entry node has an incoming edge, its flow function ignores any input)

Entry node



out = { X → S | X ∈ Formals }

$\{ a \rightarrow 0, b \rightarrow 0 \}$

$f(a, b) \{$

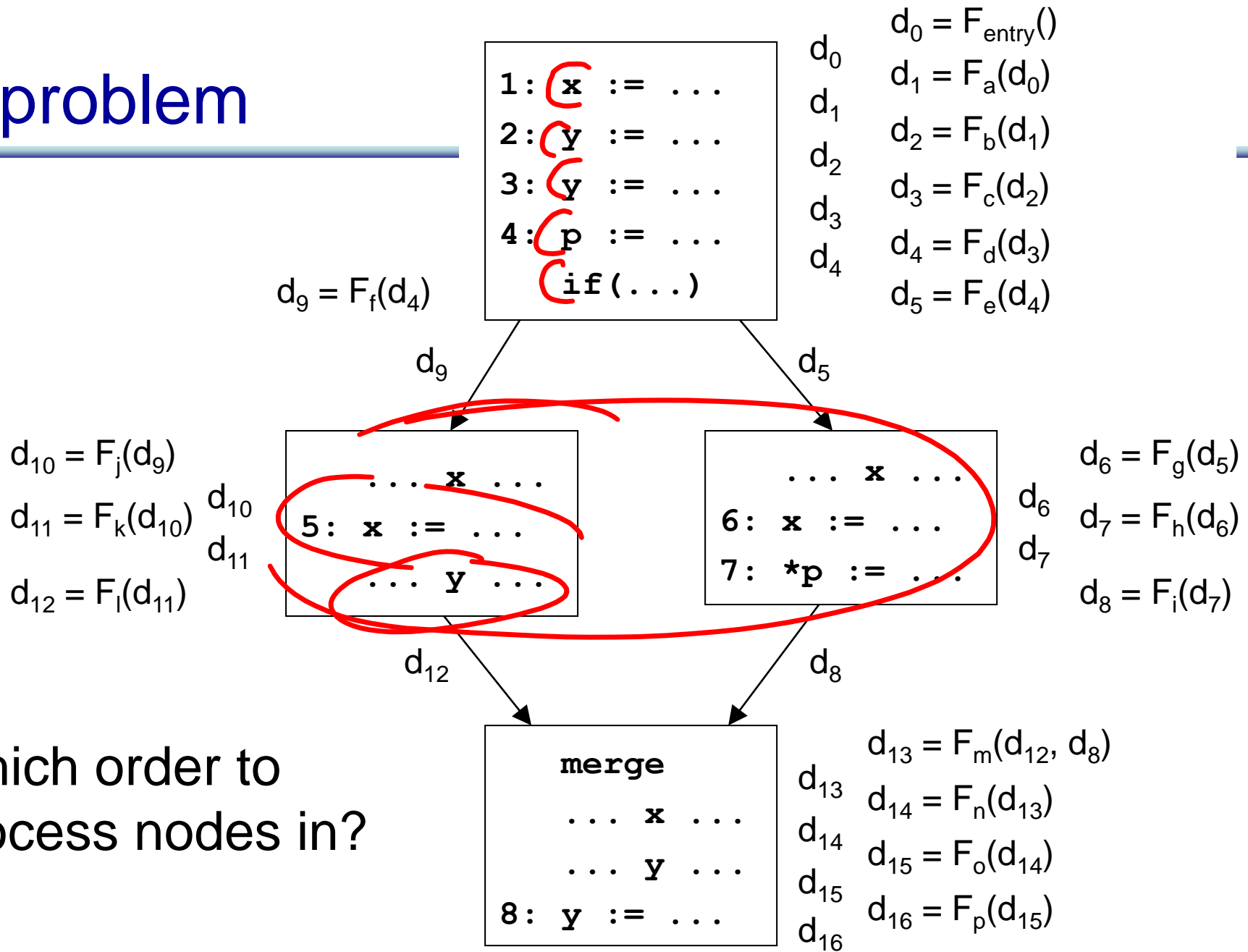
$\}$

Second problem

A. DFS

B. BFS

C. Neither



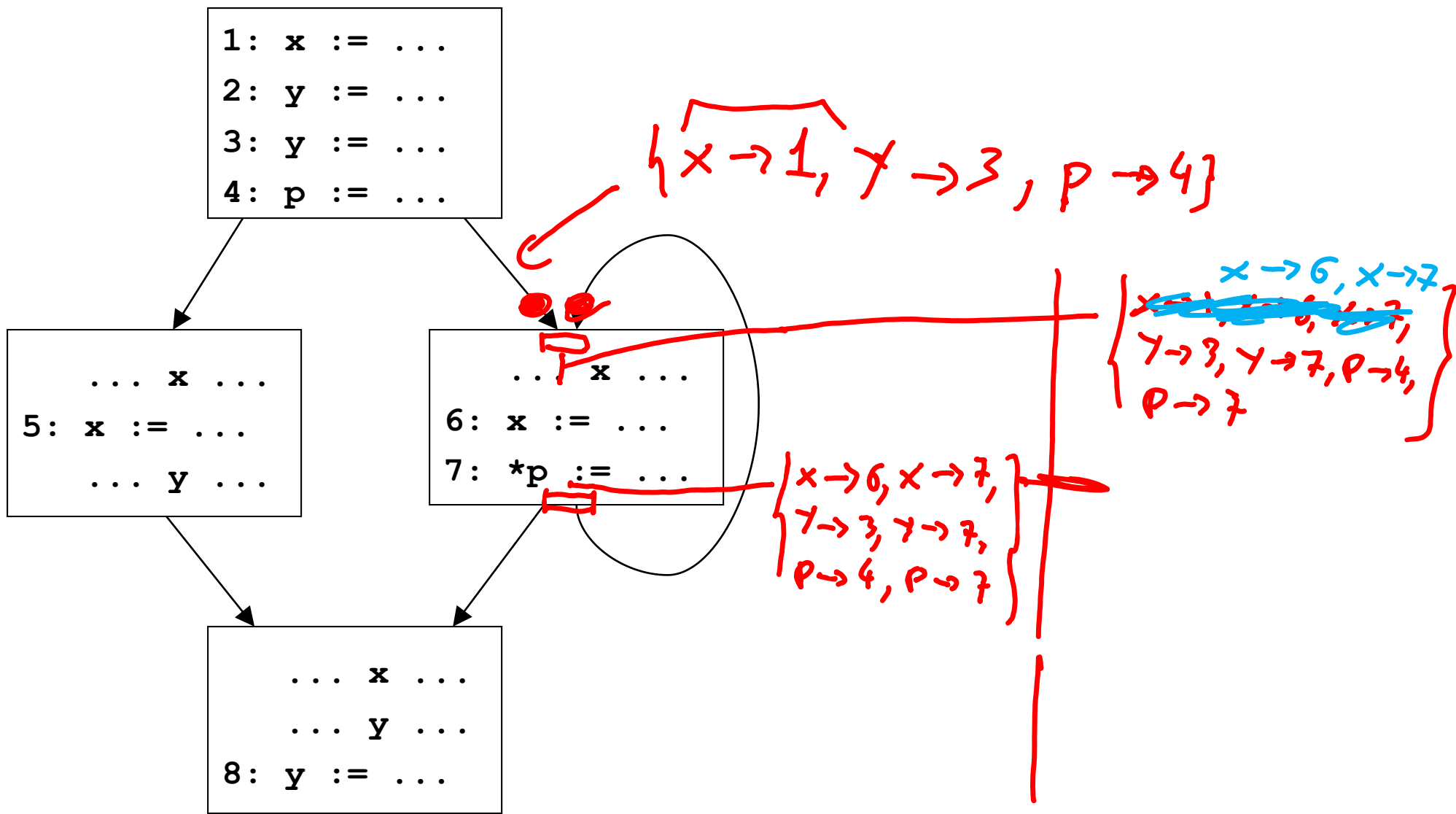
Which order to process nodes in?

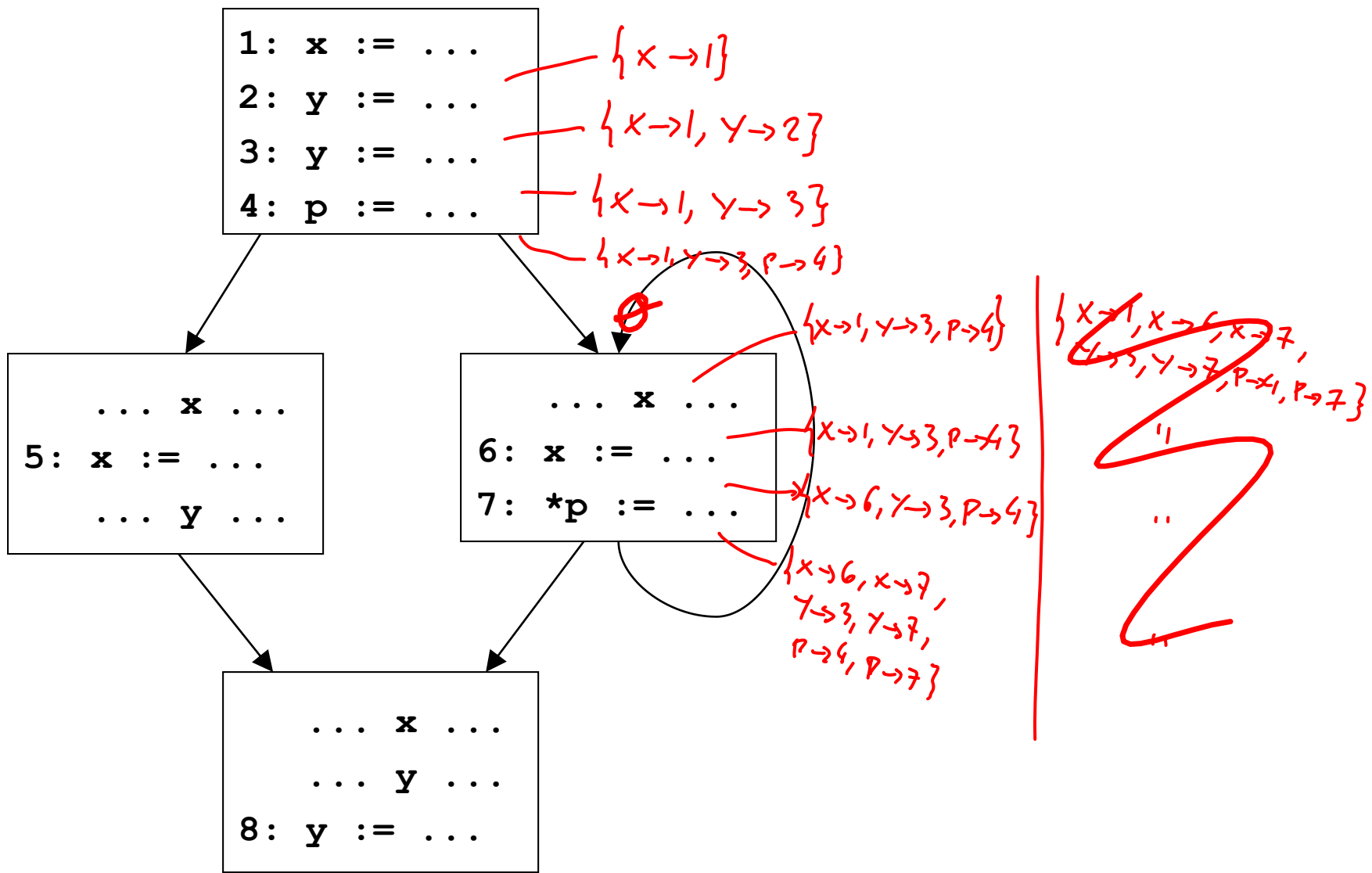
Second problem

- Which order to process nodes in?
- Sort nodes in topological order
 - each node appears in the order after all of its predecessors
- Just run the flow functions for each of the nodes in the topological order
- What's the problem now?

Second problem, prime

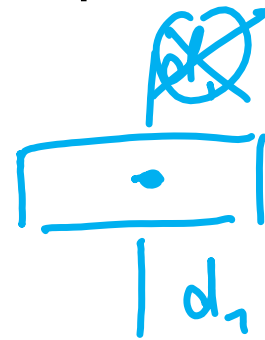
- When there are loops, there is no topological order!
- What to do?
- Let's try and see what we can do





Worklist algorithm

- Initialize all d_i to the empty set
- Store all nodes onto a worklist
- while worklist is not empty:
 - remove node n from worklist
 - apply flow function for node n
 - update the appropriate d_i , and add nodes whose inputs have changed back onto worklist



Worklist algorithm

```
let m: map from edge to computed value at edge
```

```
let worklist: work list of nodes
```

```
for each edge e in CFG do
```

```
  m(e) :=  $\emptyset$ 
```

```
for each node n do
```

```
  worklist.add(n)
```

```
while (worklist.empty.not) do
```

```
  let n := worklist.remove_any;
```

```
  let info_in := m(n.incoming_edges);
```

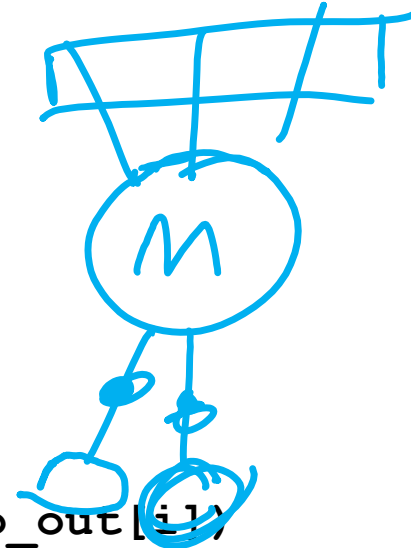
```
  let info_out := F(n, info_in);
```

```
  for i := 0 .. info_out.length-1 do
```

```
    if (m(n.outgoing_edges[i])  $\neq$  info_out[i])
```

```
      m(n.outgoing_edges[i]) := info_out[i];
```

```
      worklist.add(n.outgoing_edges[i].dst);
```



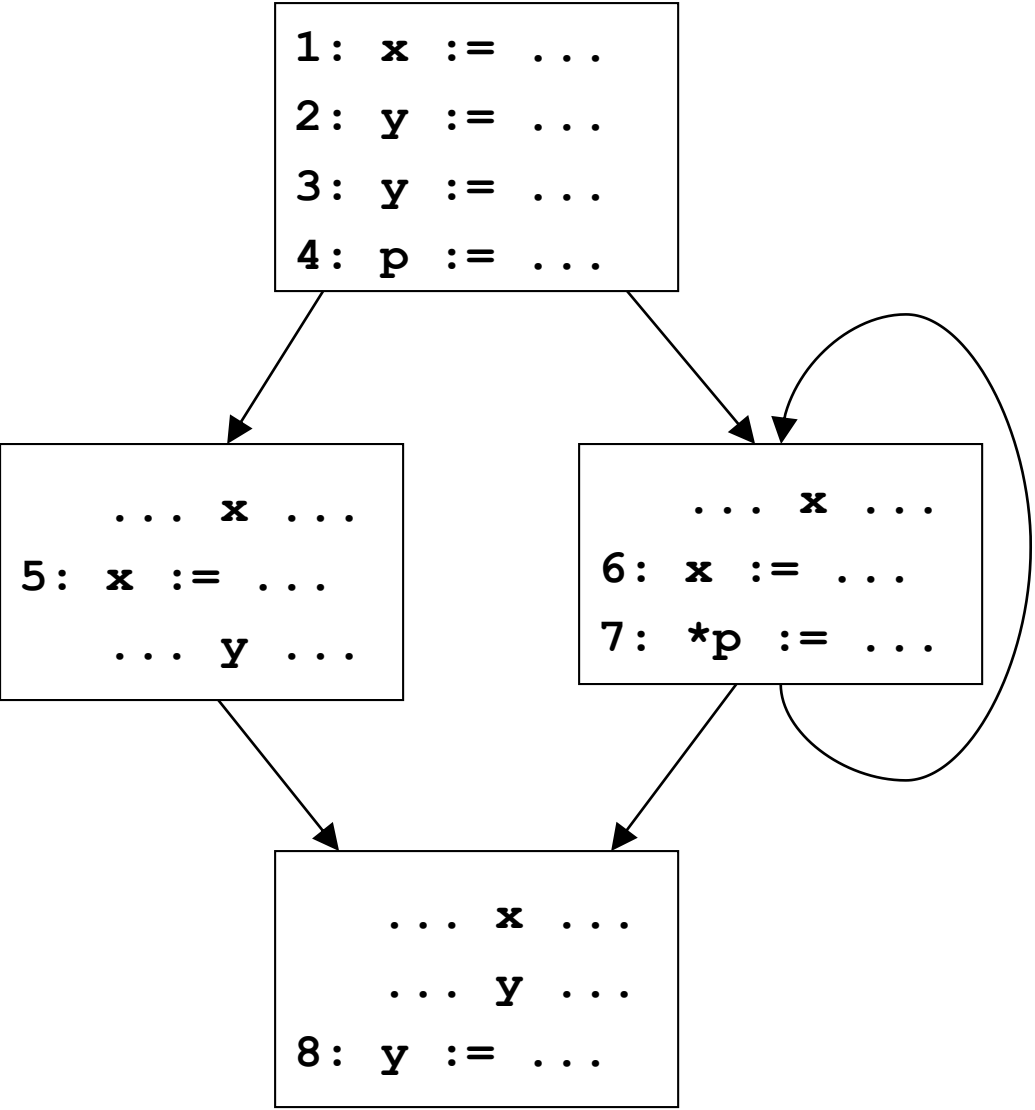
Order of nodes

- Topological order assuming back-edges have been removed
- Reverse depth-first post-order
- Use an ordered worklist

A. Yes

B. No

C. Not sure



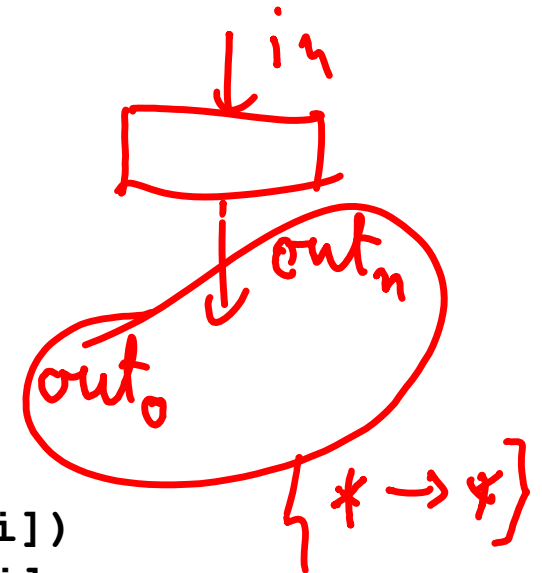
Termination

- Is termination important? In other words, can we stop the algorithm at any point?
 - A. Yes
 - B. No

Termination

- Assuming we're doing reaching defs, let's try to guarantee that the worklist loop terminates, regardless of what the flow function F does

```
while (worklist.empty.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length-1 do
    if (m(n.outgoing_edges[i]) ≠ info_out[i])
      m(n.outgoing_edges[i]) := info_out[i];
      worklist.add(n.outgoing_edges[i].dst);
```



Termination

- Assuming we're doing reaching defs, let's try to guarantee that the worklist loop terminates, regardless of what the flow function F does

```
while (worklist.empty.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length-1 do
    let new_info := m(n.outgoing_edges[i]) U
                  info_out[i];
    if (m(n.outgoing_edges[i]) ≠ new_info)
      m(n.outgoing_edges[i]) := new_info;
      worklist.add(n.outgoing_edges[i].dst);
```

Structure of the domain

- We're using the structure of the domain outside of the flow functions
- In general, it's useful to have a framework that formalizes this structure
- We will use lattices