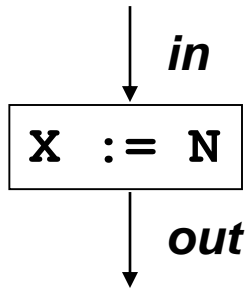


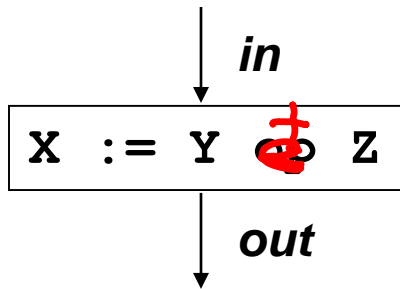
$\{a \rightarrow 5, b \rightarrow 6, c \rightarrow 10\}$

## Another example: constant prop

- Set  $D = \mathcal{P}(\{x \rightarrow c \mid x \in \text{Var}, c \in \text{const}\})$   
 $T = \emptyset$



$$F_{x := N}(\text{in}) = \text{in} - \{x \rightarrow * \} \cup \{x \rightarrow N\}$$

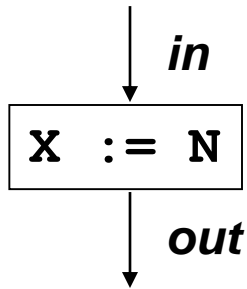


$$F_{x := Y \text{ op } Z}(\text{in}) = \cup \left\{ \begin{array}{l} x \rightarrow N_1 \text{ op } N_2 \\ Y \rightarrow N_1 \in \text{in} \\ Z \rightarrow N_2 \in \text{in} \end{array} \right\}$$

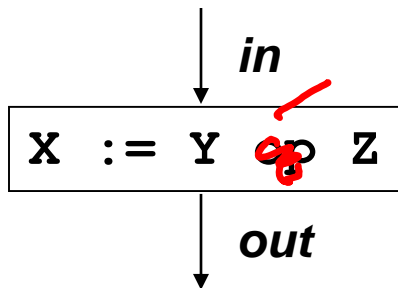
$$x = 10$$
$$x = a + b$$

# Another example: constant prop

- Set  $D = 2 \{ x \rightarrow N \mid x \in \text{Vars} \wedge N \in Z \}$



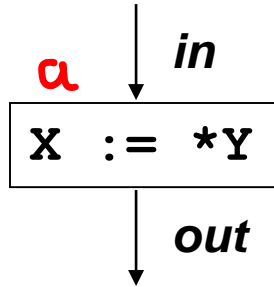
$$F_{X := N}(\text{in}) = \text{in} - \{ X \rightarrow * \} \cup \{ X \rightarrow N \}$$



$$F_{X := Y \text{ op } Z}(\text{in}) = \text{in} - \{ X \rightarrow * \} \cup \\ \{ X \rightarrow N \mid (Y \rightarrow N_1) \in \text{in} \wedge \\ (Z \rightarrow N_2) \in \text{in} \wedge \\ N = \underline{N_1 \text{ op } N_2} \}$$

# Another example: constant prop

$a \rightarrow 5, b \rightarrow 6$

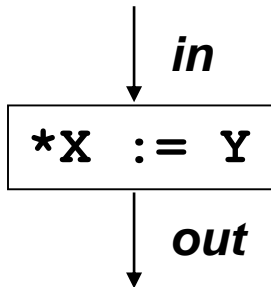


$$F_{x := *y}(\text{in}) = \text{in} - \{x \rightarrow x\} \cup$$

$$\{x \rightarrow N \mid \forall v \in \text{MPT}(y)$$

$$v \rightarrow \# \in \text{in}\}$$

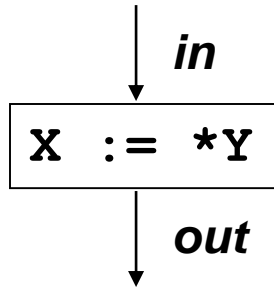
$a \rightarrow 5$   
 $y \rightarrow 5$



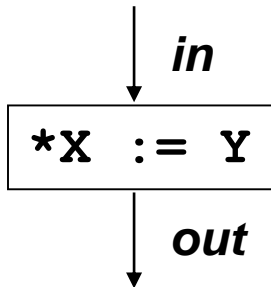
$$F_{*x := y}(\text{in}) = \text{in} - \{v \rightarrow * \mid v \in \text{MPT}(x)\}$$

$a \rightarrow 5$

# Another example: constant prop



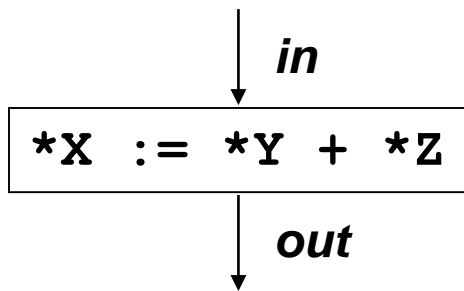
$$F_{X := *Y}(\text{in}) = \text{in} - \{ X \rightarrow * \} \\ \cup \{ X \rightarrow N \mid \forall Z \in \text{may-point-to}(Y) . \\ (Z \rightarrow N) \in \text{in} \}$$



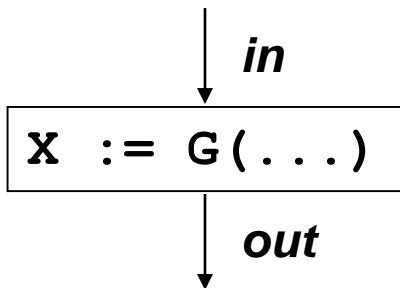
$$F_{*X := Y}(\text{in}) = \text{in} - \{ Z \rightarrow * \mid Z \in \text{may-point}(X) \} \\ \cup \{ Z \rightarrow N \mid Z \in \text{must-point-to}(X) \wedge \\ Y \rightarrow N \in \text{in} \} \\ \cup \{ Z \rightarrow N \mid (Y \rightarrow N) \in \text{in} \wedge \\ (Z \rightarrow N) \in \text{in} \}$$

# Another example: constant prop

---



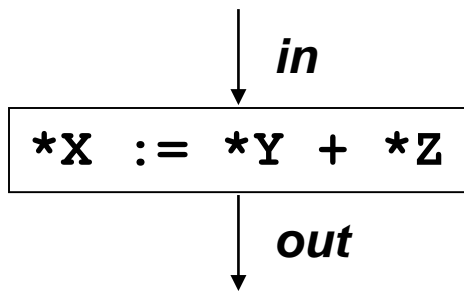
$$F_{*X := *Y + *Z}(\text{in}) =$$



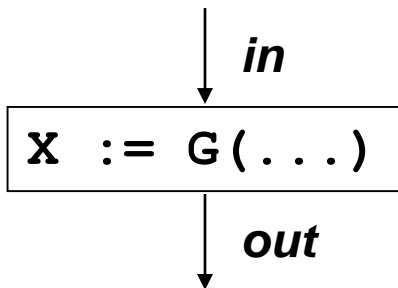
$$F_{X := G(\dots)}(\text{in}) =$$

# Another example: constant prop

---



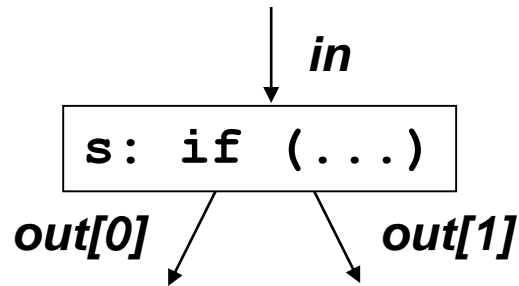
$$F_{*X := *Y + *Z}(\text{in}) = F_{a := *Y; b := *Z; c := a + b; *X := c}(\text{in})$$



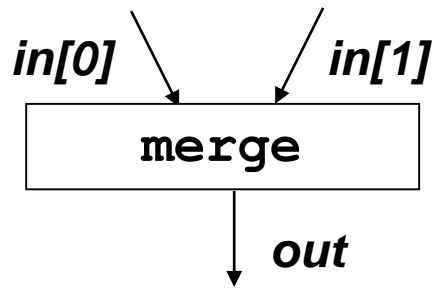
$$F_{X := G(\dots)}(\text{in}) = \emptyset$$

# Another example: constant prop

---



*a := 5*  
*if (a == 5)*  
*└*  
*└*



# Lattice

---

- $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) =$



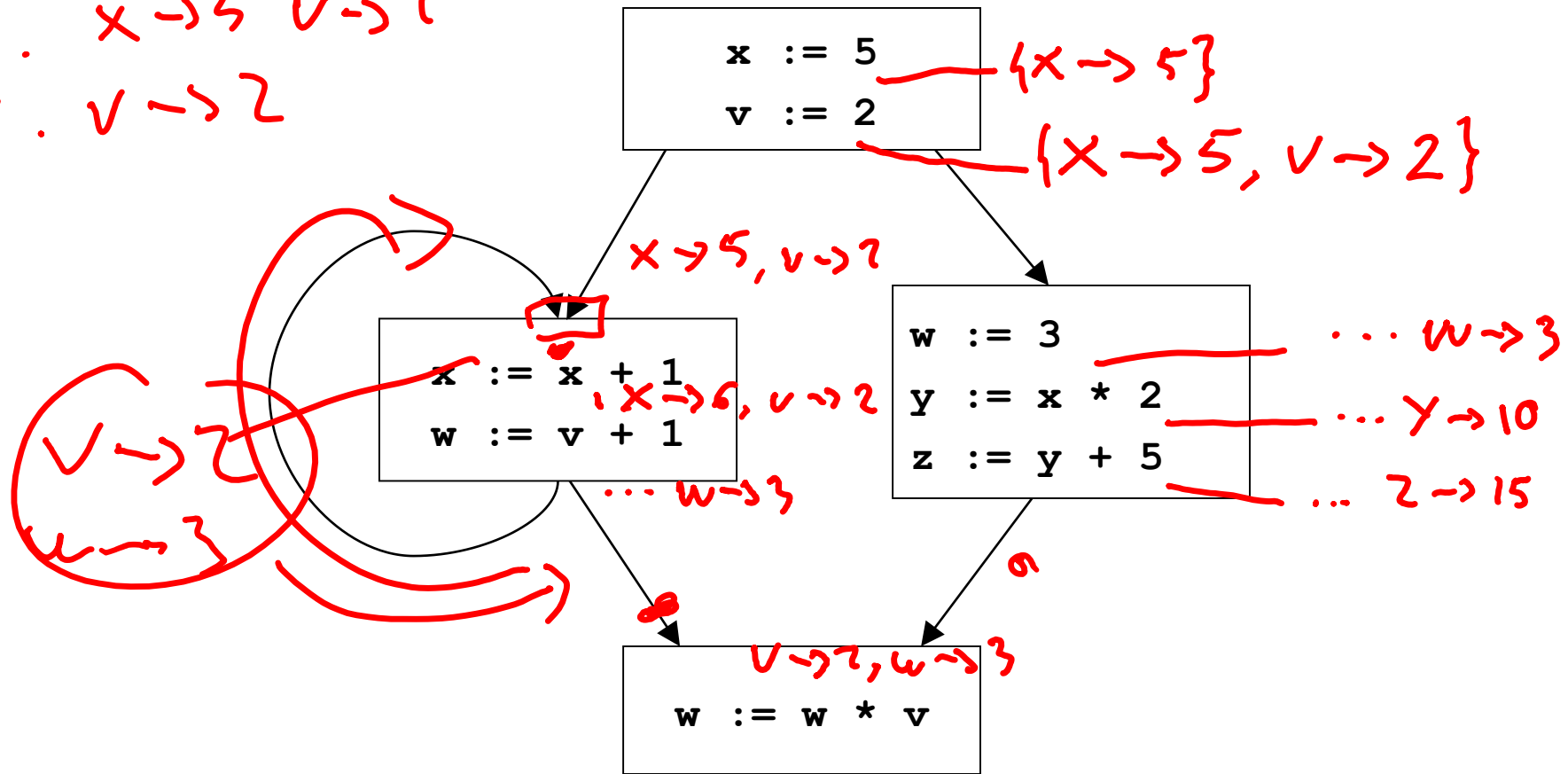
# Lattice

---

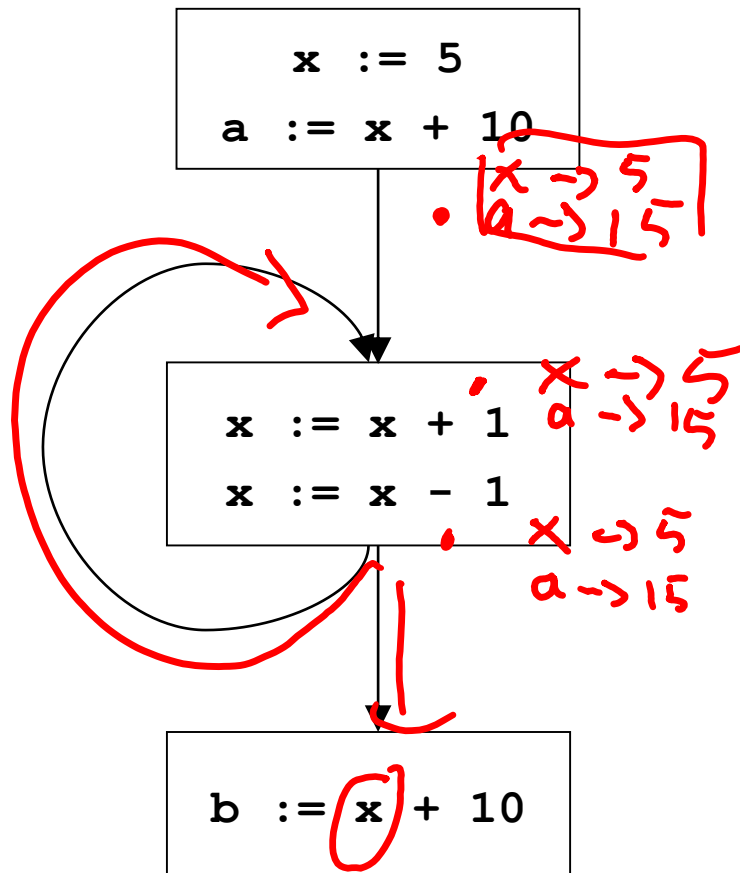
- $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) =$   
 $(2^A, \supseteq, A, \emptyset, \cap, \cup)$   
where  $A = \{x \rightarrow N \mid x \in \text{Vars} \wedge N \in \mathbb{Z}\}$

# Example

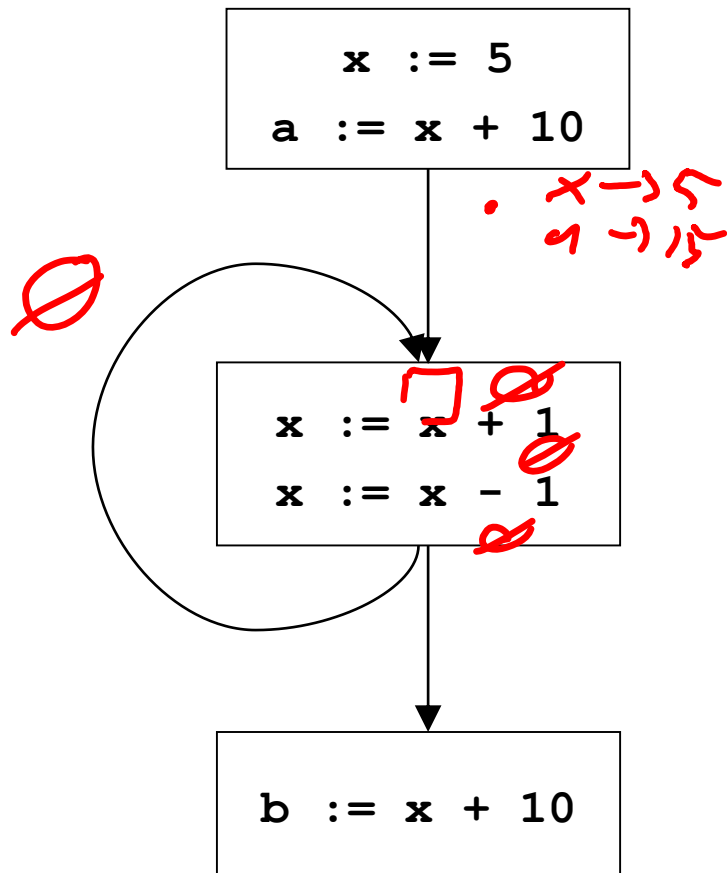
1.  $x \rightarrow 5$   $v \rightarrow 2$
2.  $v \rightarrow 2$



# Another Example



# Another Example starting at top



# Back to lattice

---

- $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) =$   
 $(2^A, \supseteq, A, \emptyset, \cap, \cup)$   
where  $A = \{ x \rightarrow N \mid x \in \text{Vars} \wedge N \in \mathbb{Z} \}$
- What's the problem with this lattice?

# Back to lattice

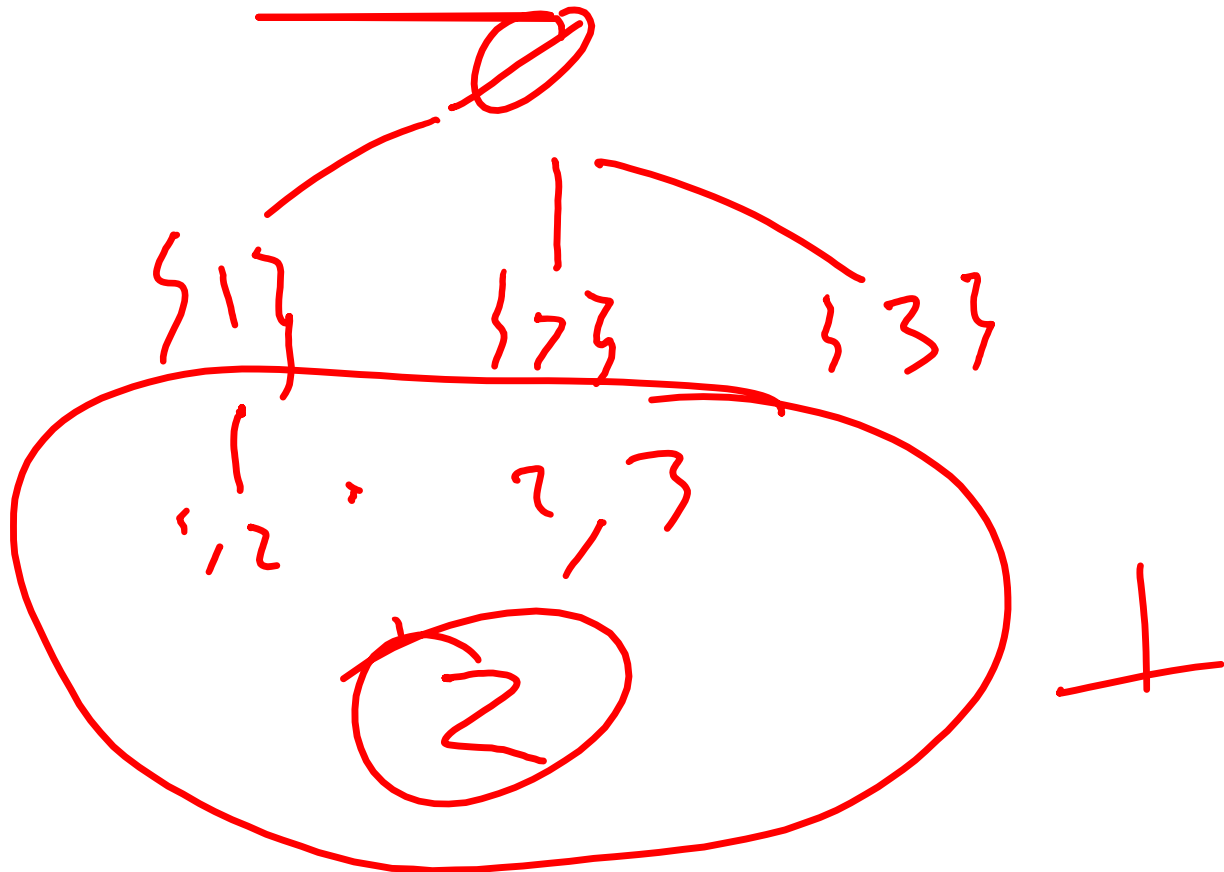
---

- $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) =$   
 $(2^A, \supseteq, A, \emptyset, \cap, \cup)$   
where  $A = \{ x \rightarrow N \mid x \in \text{Vars} \wedge N \in \mathbb{Z} \}$
- What's the problem with this lattice?
- Lattice is infinitely high, which means we can't guarantee termination

# Better lattice

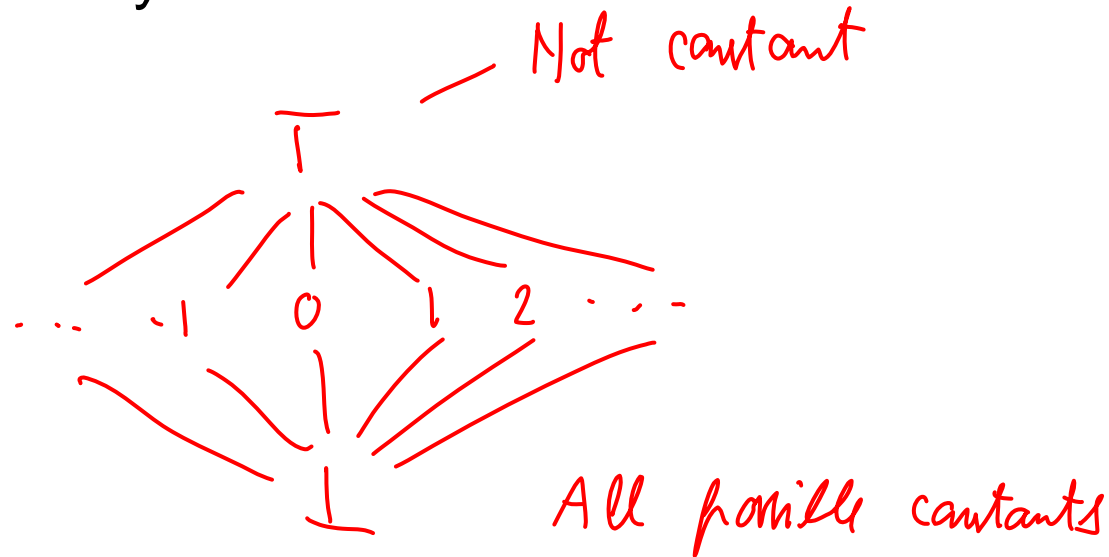
---

- Suppose we only had one variable



# Better lattice

- Suppose we only had one variable



- $D = \{\perp, \top\} \cup \mathbb{Z}$
- $\forall i \in \mathbb{Z}. \perp \sqsubseteq i \wedge i \sqsubseteq \top$
- height = 3



# For all variables

---

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices  $(D_1, \sqsubseteq_1, \perp_1, \top_1, \sqcup_1, \sqcap_1) \dots (D_n, \sqsubseteq_n, \perp_n, \top_n, \sqcup_n, \sqcap_n)$  create:

tuple lattice  $D^n =$

# For all variables

---

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices  $(D_1, \sqsubseteq_1, \perp_1, \top_1, \sqcup_1, \sqcap_1) \dots (D_n, \sqsubseteq_n, \perp_n, \top_n, \sqcup_n, \sqcap_n)$  create:

tuple lattice  $D^n = ((D_1 \times \dots \times D_n), \sqsubseteq, \perp, \top, \sqcup, \sqcap)$  where

$$\perp = (\perp_1, \dots, \perp_n)$$

$$\top = (\top_1, \dots, \top_n)$$

$$(a_1, \dots, a_n) \sqcup (b_1, \dots, b_n) = (a_1 \sqcup_1 b_1, \dots, a_n \sqcup_n b_n)$$

$$(a_1, \dots, a_n) \sqcap (b_1, \dots, b_n) = (a_1 \sqcap_1 b_1, \dots, a_n \sqcap_n b_n)$$

$$\text{height} = \text{height}(D_1) + \dots + \text{height}(D_n)$$

# For all variables

---

- Option 2: Map from variables to single lattice
- Given lattice  $(D, \sqsubseteq_1, \perp_1, \top_1, \sqcup_1, \sqcap_1)$  and a set  $V$ , create:

map lattice  $V \rightarrow D = (V \rightarrow D, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$

$$\perp = \lambda v \rightarrow \perp_1$$

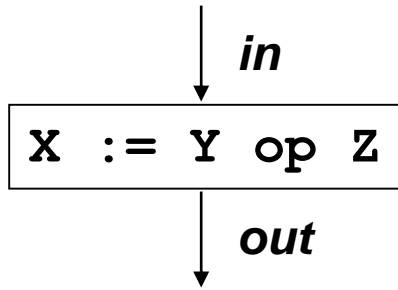
$$\top = \lambda v \rightarrow \top_1$$

$$m_1 \sqsubseteq m_2 = \lambda v \rightarrow m_1(v) \sqsubseteq_1 m_2(v)$$

$$m_1 \sqsubseteq m_2 \iff \forall v. m_1(v) \sqsubseteq_1 m_2(v)$$

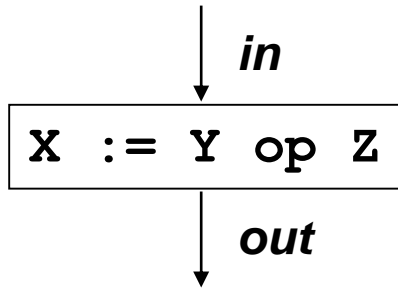
# Back to example

---



$$F_{X := Y \text{ op } Z}(\text{in}) = \text{in}[x \hat{=} \text{in}(Y) \hat{\text{op}} \text{in}(Z)]$$

# Back to example



$$F_{X := Y \text{ op } Z}(\text{in}) = \text{in} [ X \mapsto \text{in}(Y) \hat{\text{op}} \text{in}(Z) ]$$

where  $a \hat{\text{op}} b =$

+	1	2
3	4	5
4	5	6

$\hat{\text{op}}$	⊥	$d_1$	⊤
⊥	⊥	⊥	⊤
$d_2$	⊥	$d_1 \text{ op } d_2$	⊤
⊤	⊤	⊤	⊤

# General approach to domain design

---

- Simple lattices:
  - boolean logic lattice
  - powerset lattice
  - incomparable set: set of incomparable values, plus top and bottom (eg const prop lattice)
  - two point lattice: just top and bottom
- Use combinators to create more complicated lattices
  - tuple lattice constructor
  - map lattice constructor

# May vs Must

---

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

- Has to do with definition of computed info
- Set of  $x \rightarrow y$  must-point-to pairs
  - if we compute  $x \rightarrow y$ , then, then during program execution,  $x$  must point to  $y$
- Set of  $x \rightarrow y$  may-point-to pairs
  - if during program execution, it is possible for  $x$  to point to  $y$ , then we must compute  $x \rightarrow y$

# May vs must

---

	May	Must
most optimistic (bottom)	$\emptyset$	FS
most conservative (top)	FS	$\emptyset$
safe		
merge	$\cup$	$\cap$



# May vs must

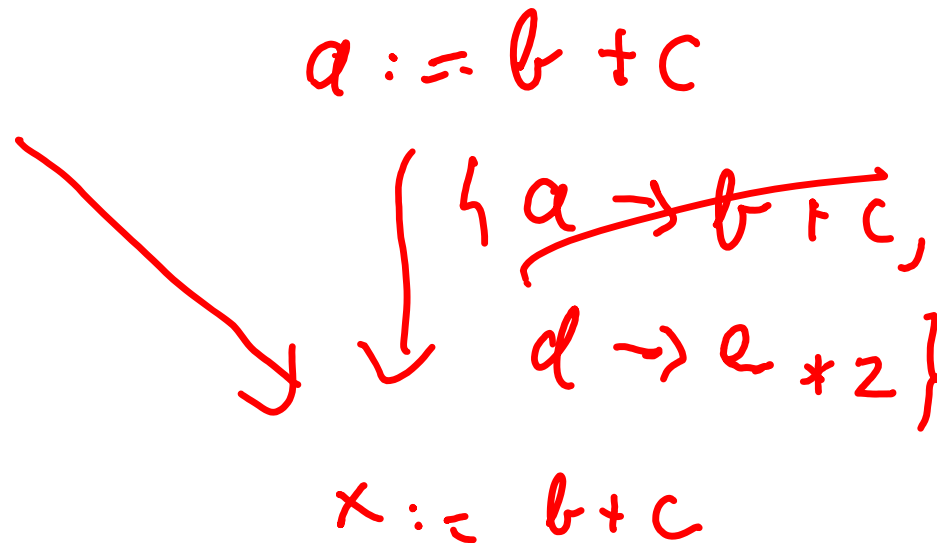
---

	May	Must
most optimistic (bottom)	empty set	full set
most conservative (top)	full set	empty set
safe	overly big	overly small
merge	$\cup$	$\cap$

# Common Sub-expression Elim

---

- Want to compute when an expression is available in a var
- Domain:



# Common Sub-expression Elim

---

- Want to compute when an expression is available in a var
- Domain:

$$S = \{x \rightarrow E \mid x \in \text{Var}, E \in \text{Expr}\}$$

$$P = 2^S$$

$$f = S$$

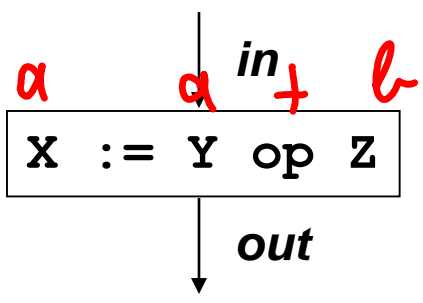
$$T = \emptyset$$

$$u = \Lambda$$

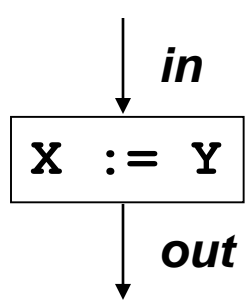
# Flow functions

$$\textcircled{1} \quad a := a + b \quad \{a \rightarrow a + b\}$$

$$\left[ \begin{array}{l} a := a + b \\ a := a + b \end{array} \right] \quad \{a \rightarrow a + b\}$$



$$F_{X := Y \text{ op } Z}(\text{in}) = \text{in} - \left\{ \begin{array}{l} X \rightarrow * \\ * \rightarrow \dots X \dots \end{array} \right\} \cup \{X \rightarrow Y \text{ op } Z\}$$

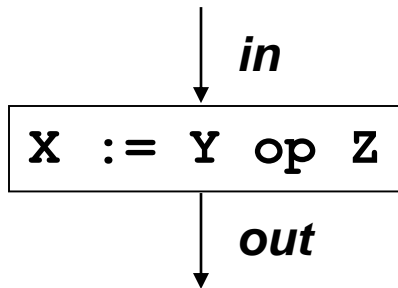


$$F_{X := Y}(\text{in}) = \{ \dots \gamma \rightarrow E, \dots \}$$

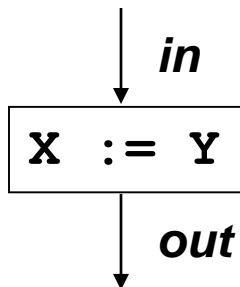
$$X := \cancel{E} Y$$

# Flow functions

---

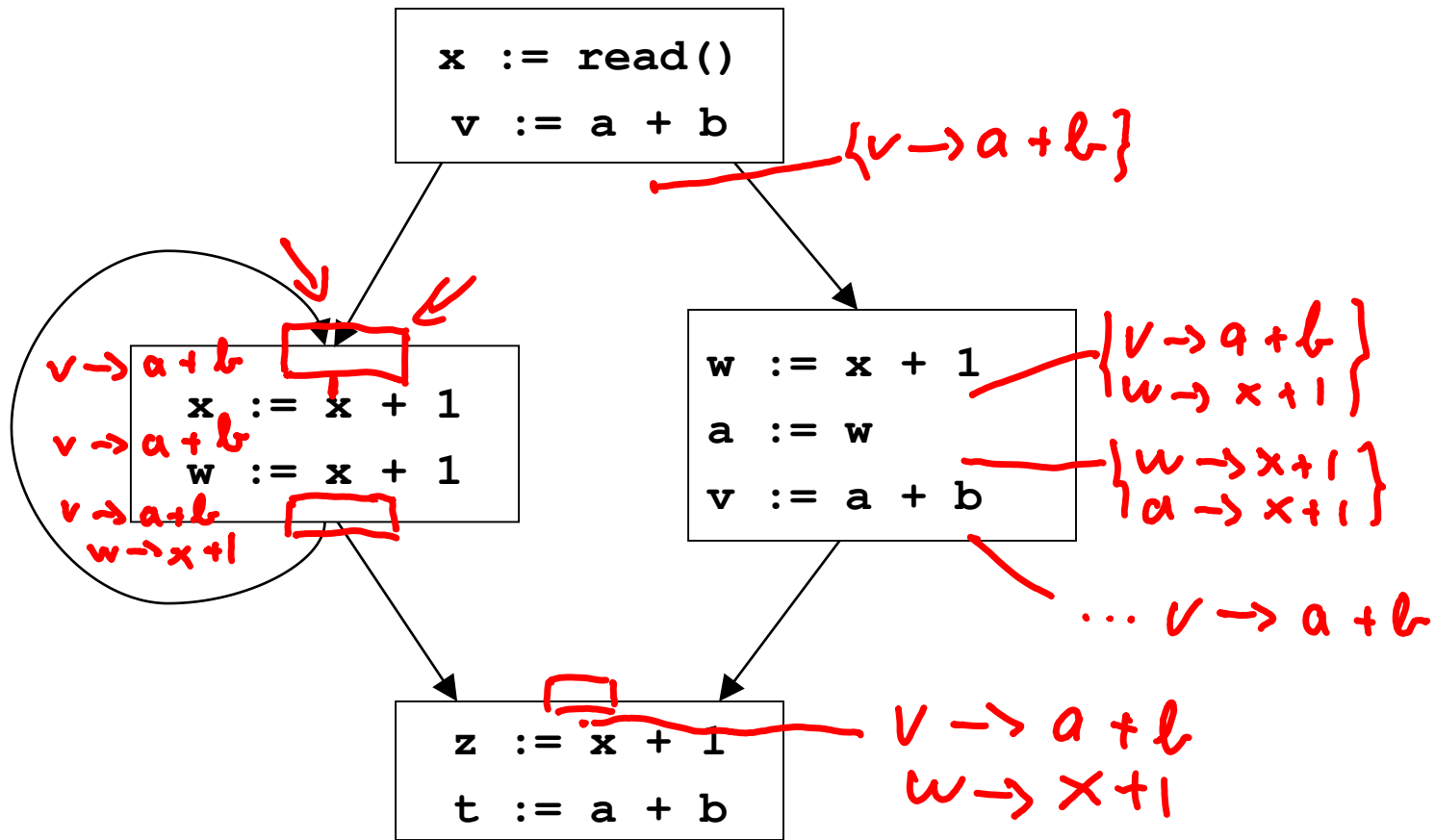


$$F_{X := Y \text{ op } Z}(\text{in}) = \text{in} - \{ X \rightarrow * \} \\ - \{ * \rightarrow \dots X \dots \} \cup \\ \{ X \rightarrow Y \text{ op } Z \mid X \neq Y \wedge X \neq Z \}$$



$$F_{X := Y}(\text{in}) = \text{in} - \{ X \rightarrow * \} \\ - \{ * \rightarrow \dots X \dots \} \cup \\ \{ X \rightarrow E \mid Y \rightarrow E \in \text{in} \}$$

# Example



# Direction of analysis

---

- Although constraints are not directional, flow functions are
- All flow functions we have seen so far are in the forward direction
- In some cases, the constraints are of the form  
$$\text{in} = F(\text{out})$$
- These are called backward problems.
- Example: live variables
  - compute the set of variables that may be live

# Live Variables

---

- A variable is live at a program point if it will be used before being redefined
- A variable is dead at a program point if it is redefined before being used



# Example: live variables

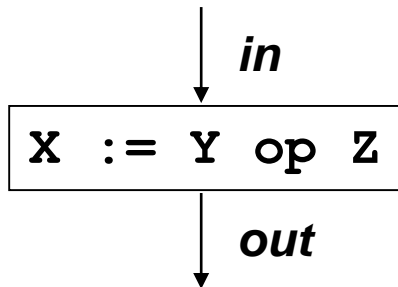
---

- Set  $D =$
- Lattice:  $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) =$

# Example: live variables

---

- Set  $D = 2^{\text{Vars}}$
- Lattice:  $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) = (2^{\text{Vars}}, \subseteq, \emptyset, \text{Vars}, \cup, \cap)$

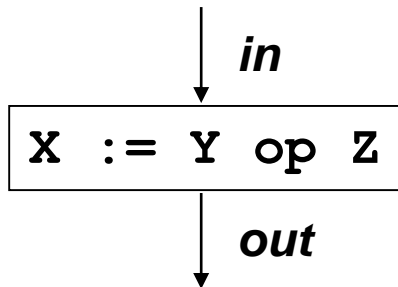


$$F_{X := Y \text{ op } Z}(\text{out}) =$$

# Example: live variables

---

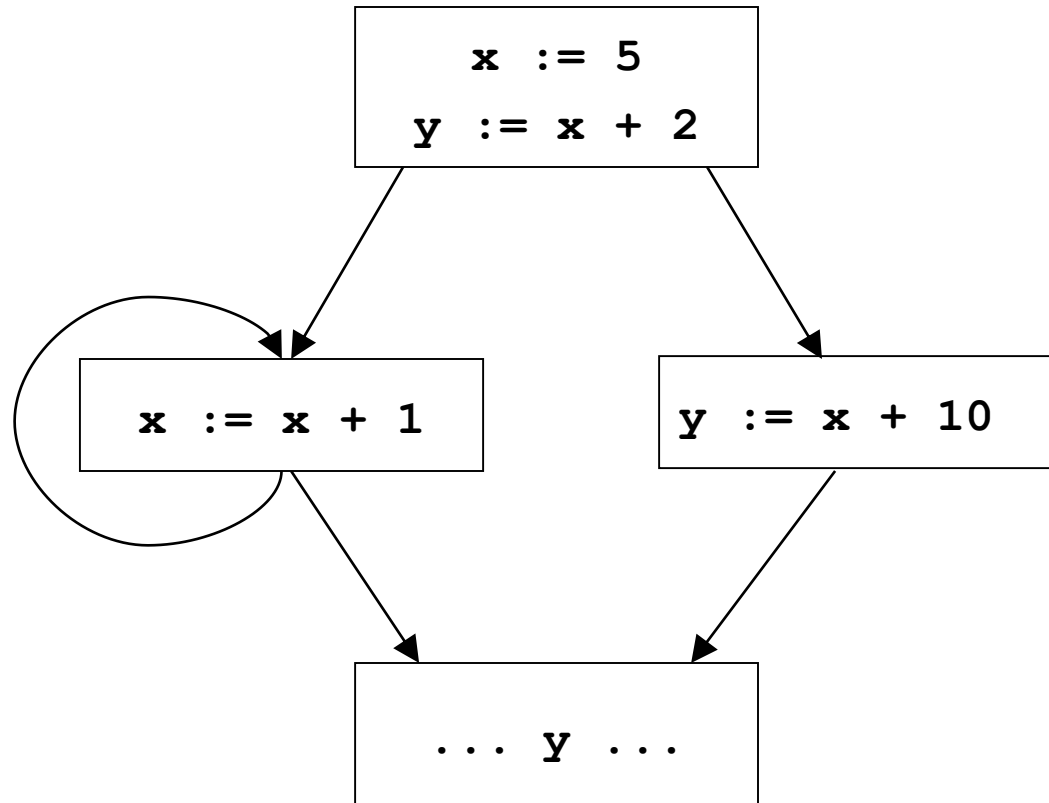
- Set  $D = 2^{\text{Vars}}$
- Lattice:  $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap) = (2^{\text{Vars}}, \subseteq, \emptyset, \text{Vars}, \cup, \cap)$



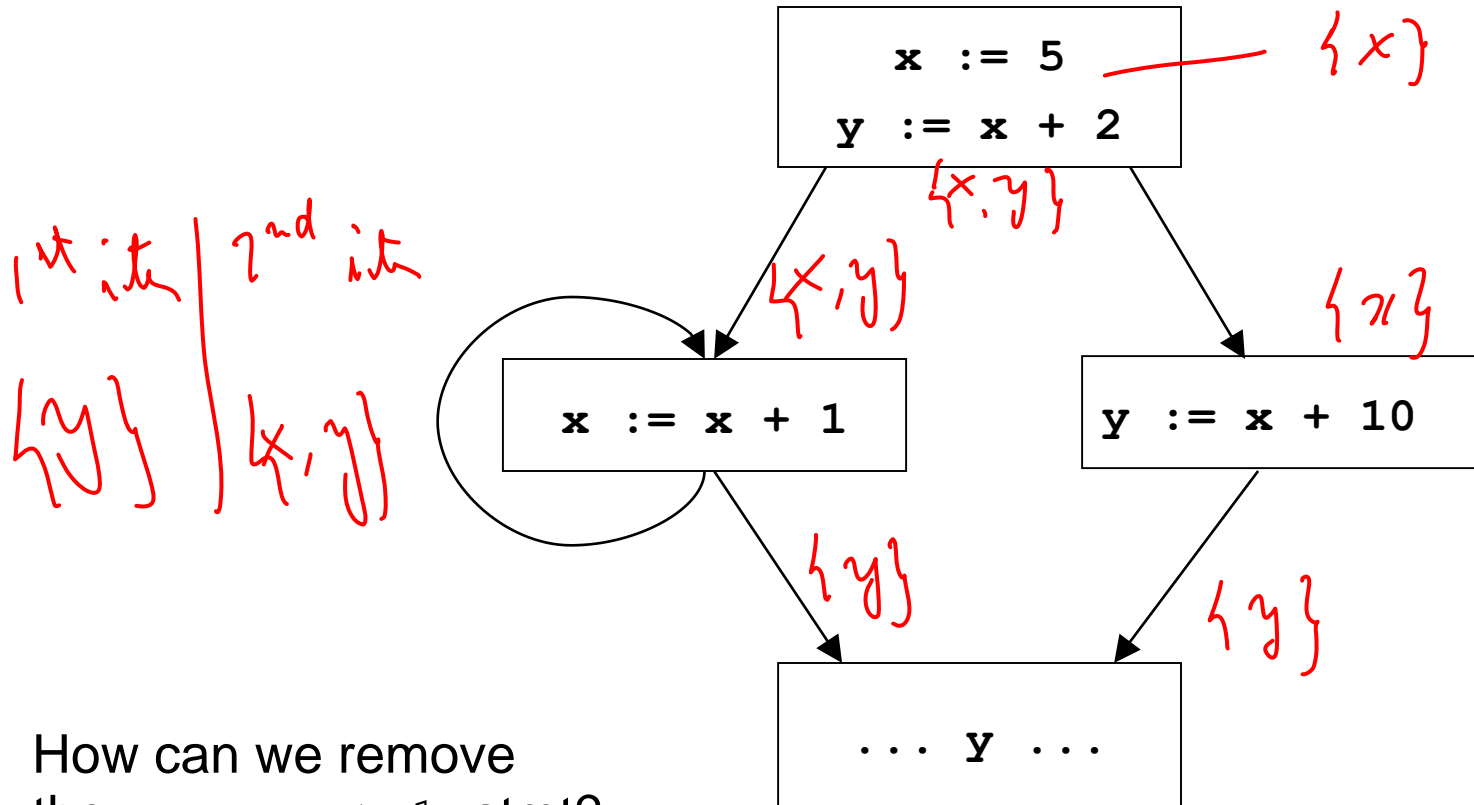
$$F_{x := y \text{ op } z}(\text{out}) = \text{out} - \{x\} \cup \{y, z\}$$

# Example: live variables

---



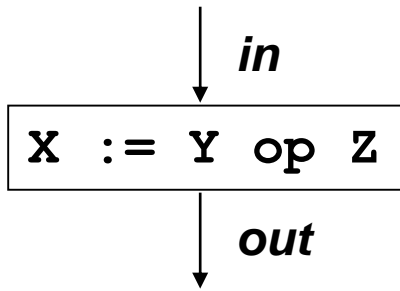
# Example: live variables



How can we remove  
the  $x := x + 1$  stmt?

# Revisiting assignment

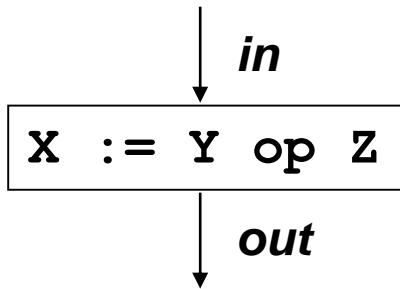
---



$$F_{x := Y \text{ op } z}(\text{out}) = \text{out} - \{ X \} \cup \{ Y, Z \}$$

# Revisiting assignment

---



$$F_{x := Y \text{ op } z}(\text{out}) = \text{out} - \{ X \} \cup \{ Y, Z \}$$

$$\text{out} - \{ x \} \cup$$

$$x \notin \text{out} ? \emptyset : \{ Y, Z \}$$

# Theory of backward analyses

---

- Can formalize backward analyses in two ways
- Option 1: reverse flow graph, and then run forward problem
- Option 2: re-develop the theory, but in the backward direction



# Precision

---

- Going back to constant prop, in what cases would we lose precision?

# Precision

---

- Going back to constant prop, in what cases would we lose precision?

```
x := 5
if (<expr>) {
  x := 6
}
... x ...
```

where <expr> is  
equiv to false

```
if (p) {
  x := 5;
} else
  x := 4;
}
...
```

```
if (p) {
  y := x + 1
} else {
  y := x + 2
}
... y ...
```

```
if (...) {
  x := -1;
} else
  x := 1;
}
y := x * x;
... y ...
```

# Precision

---

- The first problem: Unreachable code
  - solution: run unreachable code removal before
  - the unreachable code removal analysis will do its best, but may not remove all unreachable code
- The other two problems are path-sensitivity issues
  - Branch correlations: some paths are infeasible
  - Path merging: can lead to loss of precision

# MOP: meet over all paths

---

- Information computed at a given point is the meet of the information computed by each path to the program point

```
if (...) {  
    x := -1;  
} else  
    x := 1;  
}  
y := x * x;  
... y ...
```

# MOP

---

- For a path  $p$ , which is a sequence of statements  $[s_1, \dots, s_n]$ , define:  $F_p(\text{in}) = F_{s_n}(\dots F_{s_1}(\text{in}) \dots)$
- In other words:  $F_p = F_{s_1} \circ \dots \circ F_{s_n}$
- Given an edge  $e$ , let  $\text{paths-to}(e)$  be the (possibly infinite) set of paths that lead to  $e$
- Given an edge  $e$ ,  $\text{MOP}(e) = \bigsqcup_{p \in \text{paths-to}(e)} F_p(\perp)$
- For us, should be called JOP (ie: join, not meet)

# MOP vs. dataflow

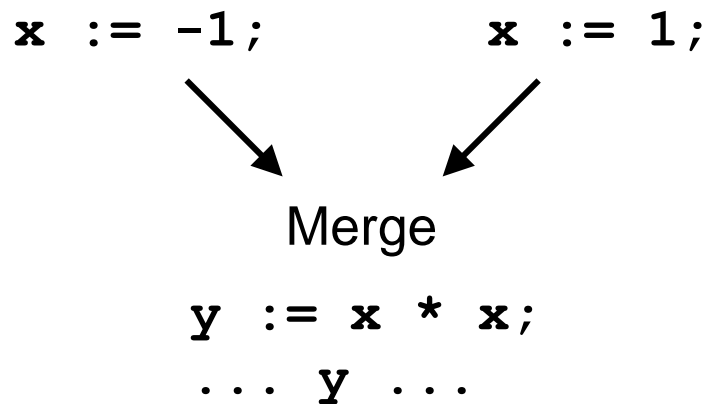
---

- MOP is the “best” possible answer, given a fixed set of flow functions
  - This means that  $\text{MOP} \sqsubseteq \text{dataflow}$  at edge in the CFG
- In general, MOP is not computable (because there can be infinitely many paths)
  - vs dataflow which is generally computable (if flow fns are monotonic and height of lattice is finite)
- And we saw in our example, in general,  $\text{MOP} \neq \text{dataflow}$

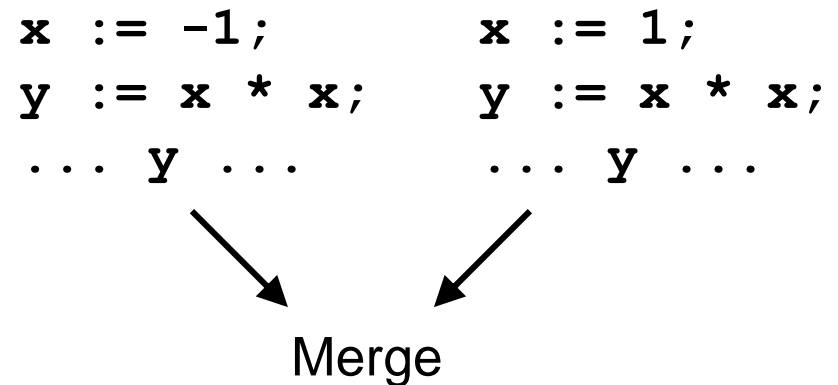
# MOP vs. dataflow

- However, it would be great if by imposing some restrictions on the flow functions, we could guarantee that dataflow is the same as MOP. What would this restriction be?

## Dataflow



## MOP



# MOP vs. dataflow

---

- However, it would be great if by imposing some restrictions on the flow functions, we could guarantee that dataflow is the same as MOP. What would this restriction be?
- Distributive problems. A problem is distributive if:  
$$\forall a, b . F(a \sqcup b) = F(a) \sqcup F(b)$$
- If flow function is distributive, then MOP = dataflow



# Summary of precision

---

- Dataflow is the basic algorithm
- To basic dataflow, we can add path-separation
  - Get MOP, which is same as dataflow for distributive problems
  - Variety of research efforts to get closer to MOP for non-distributive problems
- To basic dataflow, we can add path-pruning
  - Get branch correlation
- To basic dataflow, can add both:
  - meet over all feasible paths