# Global Optimization

## CS143

## Lecture 15

https://web.stanford.edu/class/cs143/

Instructor: Fredrik Kjolstad

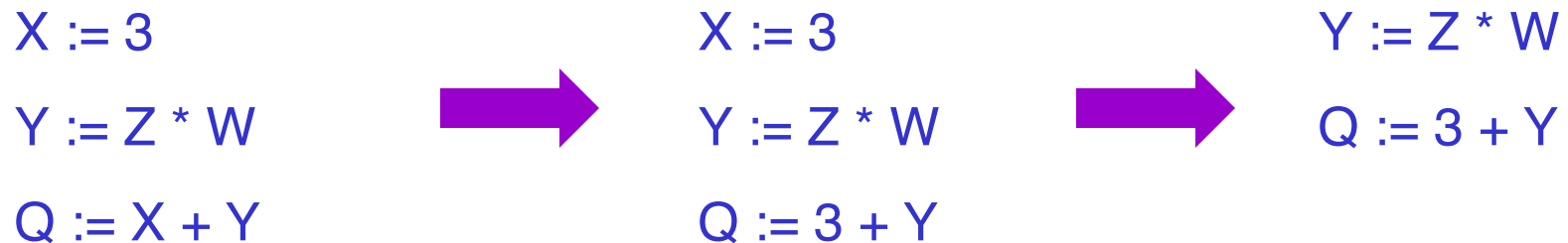Slide design by Prof. Alex Aiken, with modifications

# Lecture Outline

- Global flow analysis

- Global constant propagation
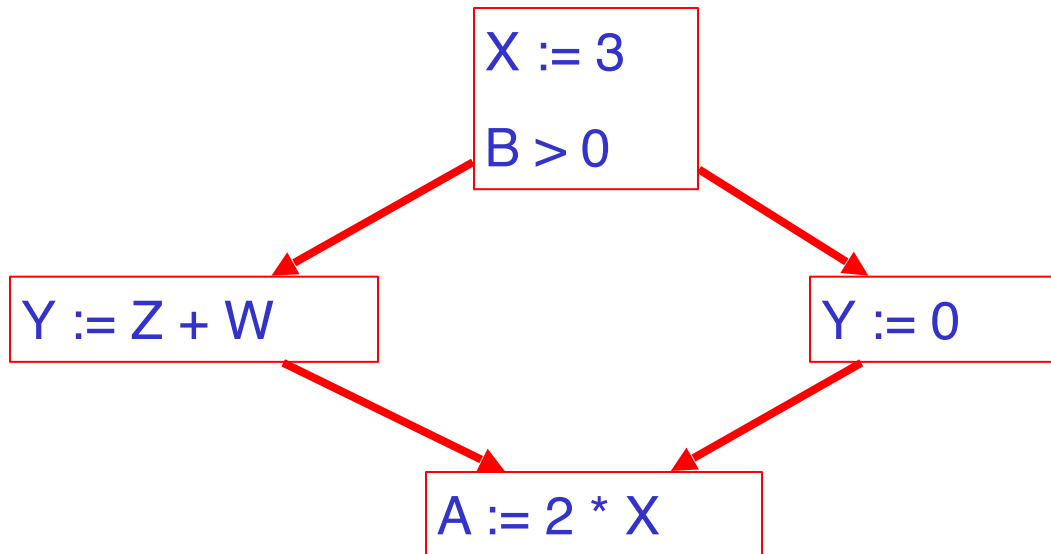
- Liveness analysis

# Local Optimization

Recall the simple basic-block optimizations
  – Constant propagation
  – Dead code elimination

X := 3

Y := Z * W          ➡          X := 3          ➡          Y := Z * W

Q := X + Y                     Y := Z * W                 Q := 3 + Y

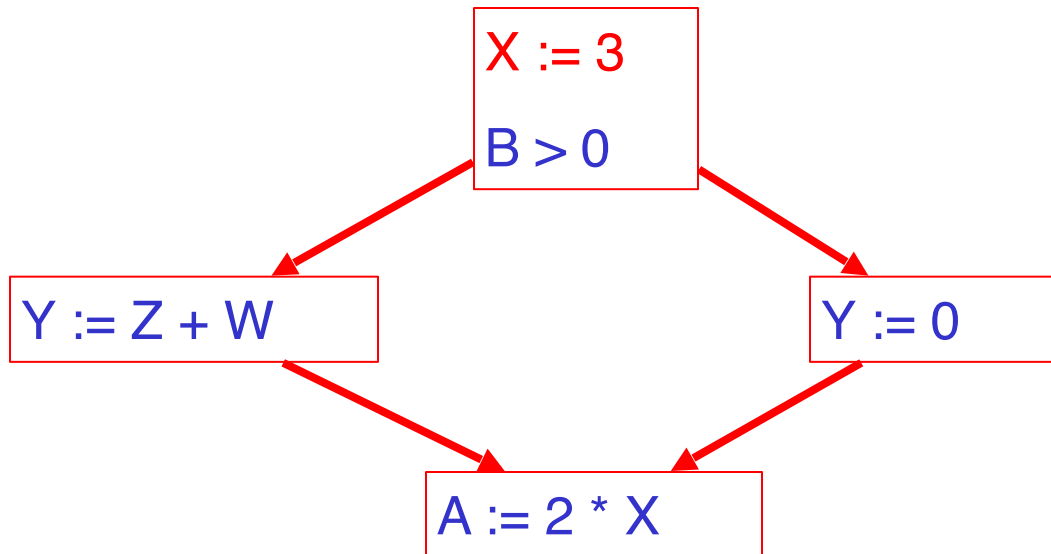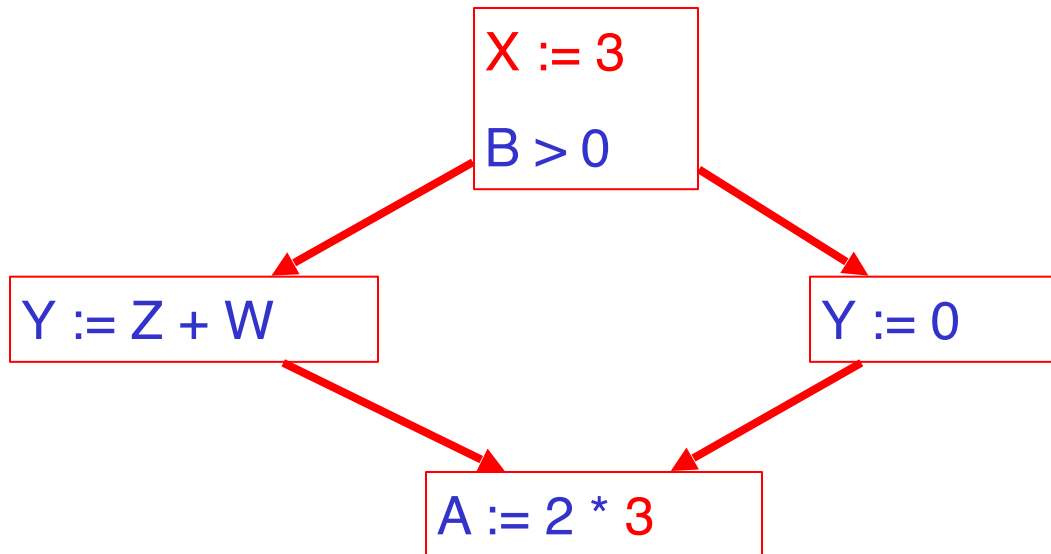                              Q := 3 + Y

# Global Optimization

These optimizations can be extended to an entire
control-flow graph

# Global Optimization

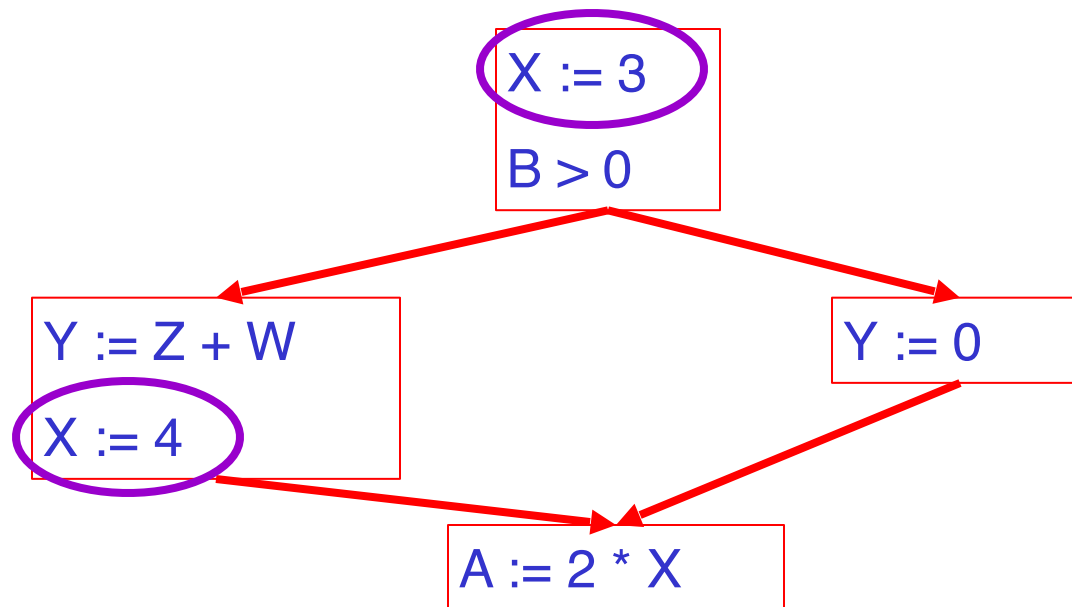These optimizations can be extended to an entire control-flow graph

# Global Optimization

These optimizations can be extended to an entire control-flow graph

# **Correctness**

- How do we know it is OK to globally propagate constants?
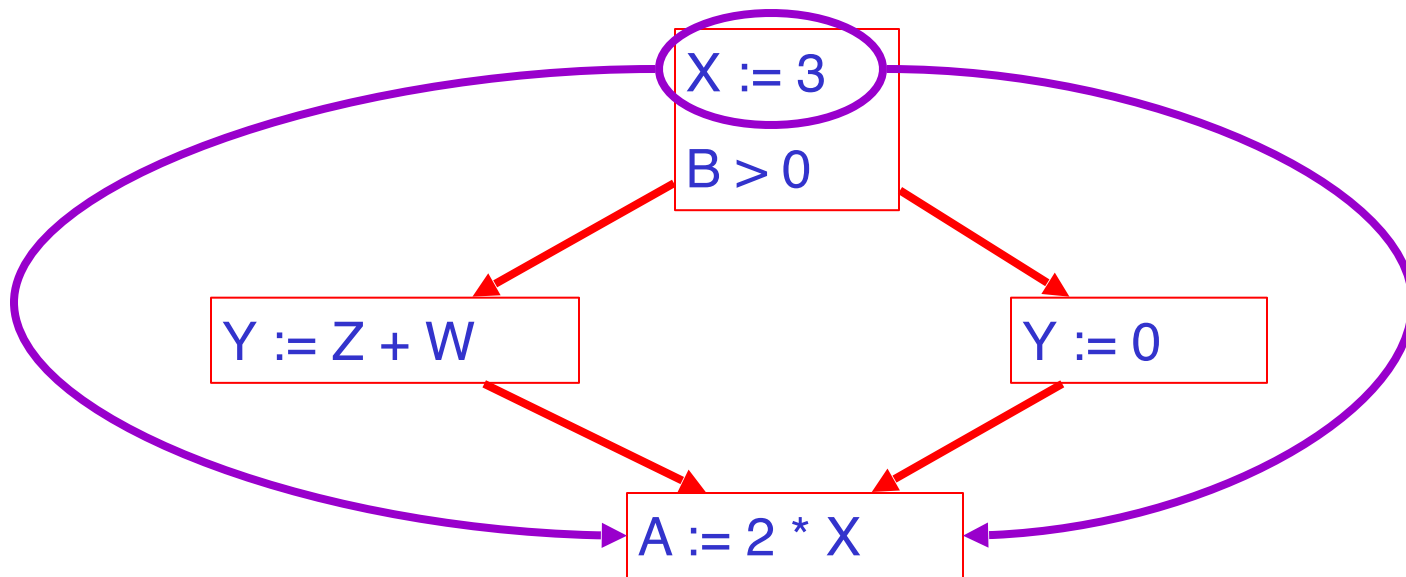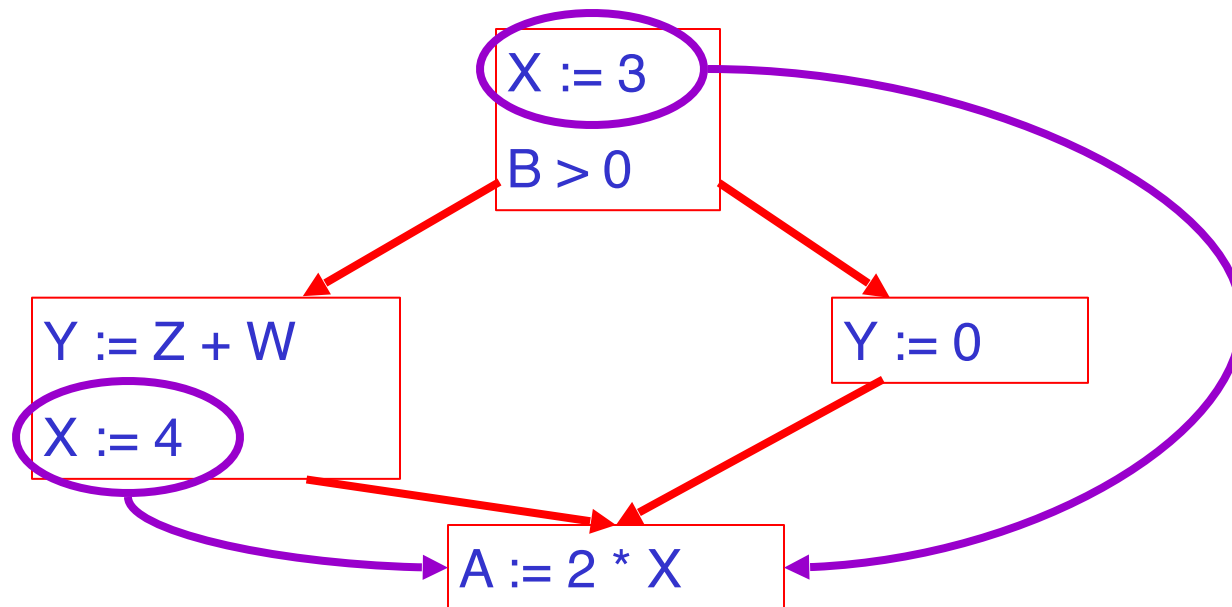- There are situations where it is incorrect:

# Correctness (Cont.)

To replace a use of x by a constant k we must know that:

On every path to the use of x, the last assignment to x is x := k   **

# Example 1 Revisited

# Example 2 Revisited

# Discussion

- The correctness condition is not trivial to check

- "All paths" includes paths around loops and through branches of conditionals

- Checking the condition requires global analysis
  - An analysis of the entire control-flow graph

# **Global Analysis**

Global optimization tasks share several traits:

- The optimization depends on knowing a property $X$ at a particular point in program execution
- Proving $X$ at any point requires knowledge of the entire function
- It is OK to be conservative.  If the optimization requires $X$ to be true, then want to know either
  - $X$ is definitely true
  - Don't know if $X$ is true

- It is always safe to say "don't know"

# Global Analysis (Cont.)

- Global dataflow analysis is a standard technique for solving problems with these characteristics

- Global constant propagation is one example of an optimization that requires global dataflow analysis
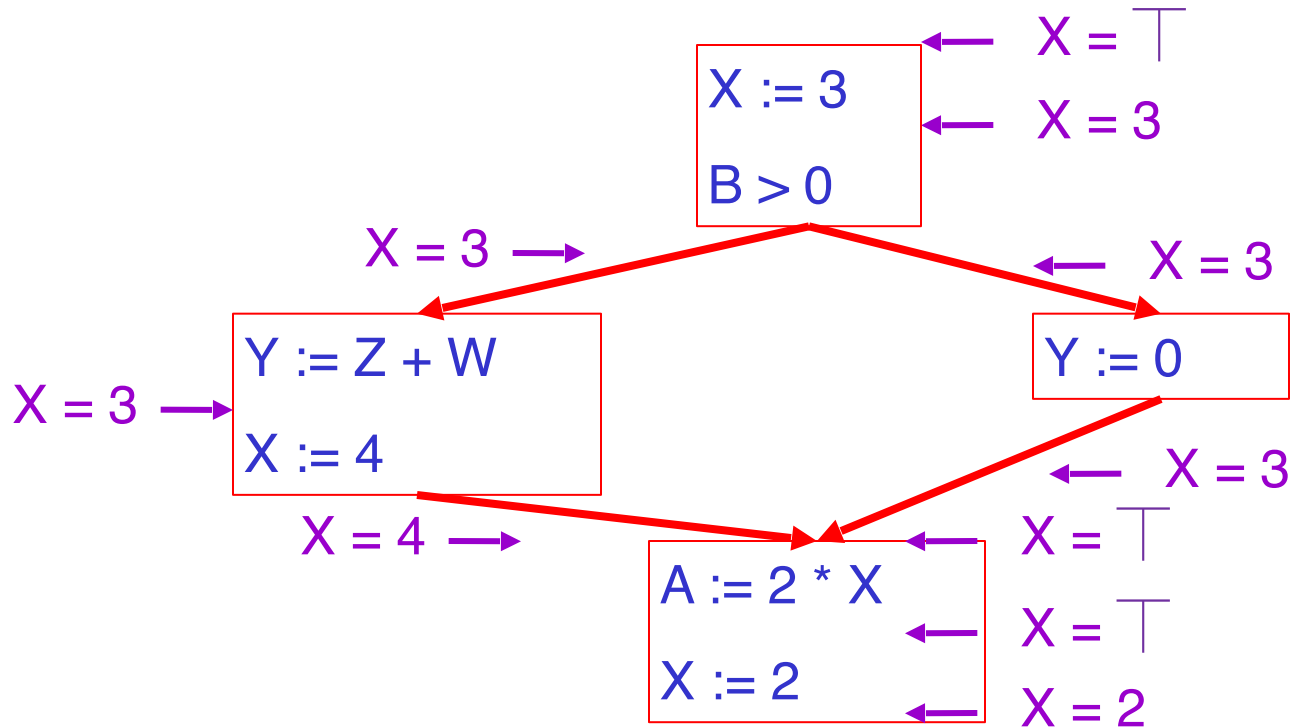
# Global Constant Propagation

- Global constant propagation can be performed at any point where ** holds

- Consider the case of computing ** for a single variable X at all program points

# Global Constant Propagation (Cont.)

- To make the problem precise, we associate one of the following values with X at every program point

| value | interpretation |
|-------|----------------|
| $\perp$ | This statement is unreachable |
| c | X = constant c |
| $\top$ | X is not a constant |

# Example

# Using the Information

- Given global constant information, it is easy to perform the optimization
  - Simply inspect the x = ? associated with a statement using x
  - If x is constant at that point replace that use of x by the constant


- But how do we compute the properties x = ?

# The Idea

The analysis of a complicated program can be expressed as a combination of simple rules relating the change in information between adjacent statements

# Between statements (forward flow):
1. If ANY predecessor has $X = \top \rightarrow X = \top$
2. If predecessors have DIFFERENT constants $\rightarrow X = \top$
3. If ALL predecessors have SAME constant c $\rightarrow X = c$
4. If ALL predecessors have $X = \bot \rightarrow X = \bot$

# Within statements:
5. $X := \bot \rightarrow X = \bot$
6. $X := c \rightarrow X = c$
7. $X :=$ non-constant $\rightarrow X = \top$
8. $Y := \dots$ doesn't affect X if $X \neq Y$

# Explanation

- The idea is to "push" or "transfer" information from one statement to the next

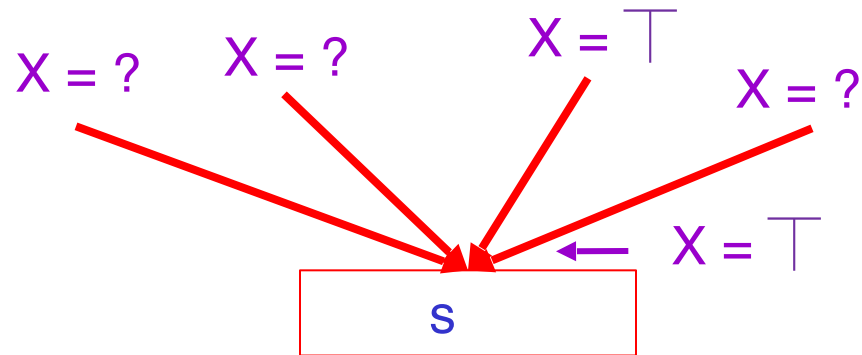- For each statement s, we compute information about the value of x immediately before and after s

$$C(s,x,in) = \text{value of } x \text{ before } s$$
$$C(s,x,out) = \text{value of } x \text{ after } s$$
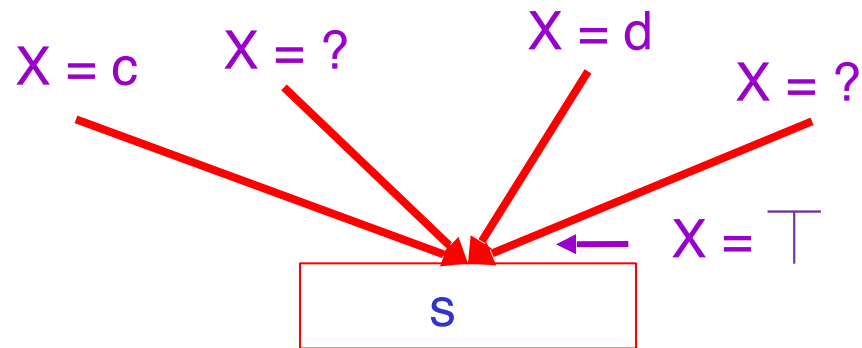
# Transfer Functions

- Define a transfer function that transfers information from one statement to another

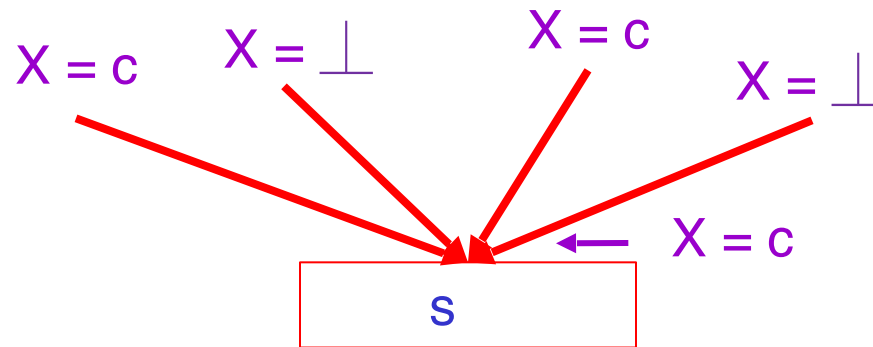- In the following rules, let statement s have immediate predecessor statements $p_1, \ldots, p_n$

# Rule 1



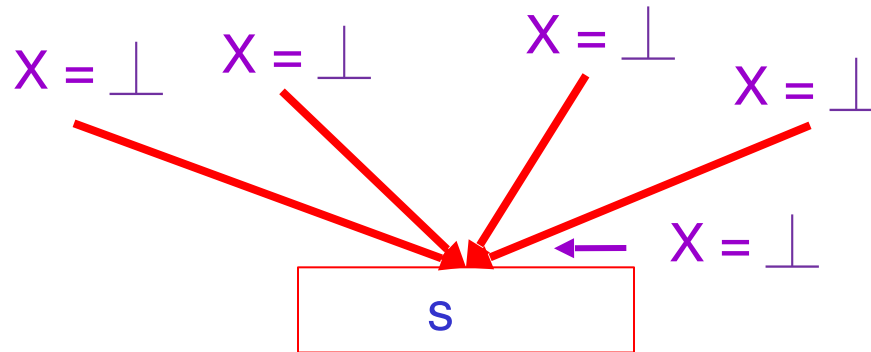if $C(p_i, x, out) = \top$ for any i, then $C(s, x, in) = \top$

# Rule 2



$$C(p_i, x, out) = c \ \& \ C(p_j, x, out) = d \ \& \ d <> c \ \text{then}$$
$$C(s, x, in) = \top$$

# Rule 3



if $C(p_i, x, \text{out}) = c$ or $\perp$ for all i,
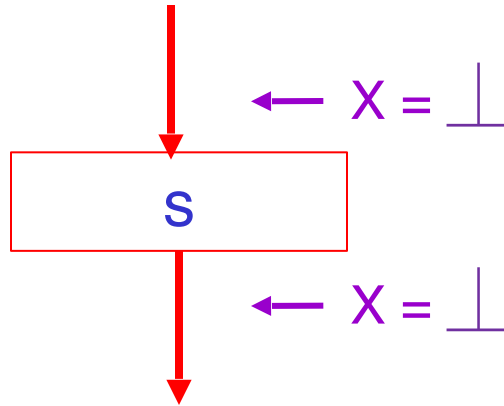
then $C(s, x, \text{in}) = c$

# Rule 4

$$X = \bot \quad X = \bot \quad X = \bot \quad X = \bot$$

$$X = \bot$$

s

if $C(p_i, x, out) = \bot$ for all i,
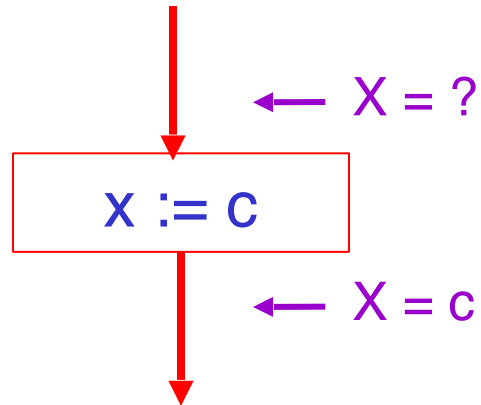
then $C(s, x, in) = \bot$

# The Other Half

- Rules 1-4 relate the out  of one statement to the in of the next statement

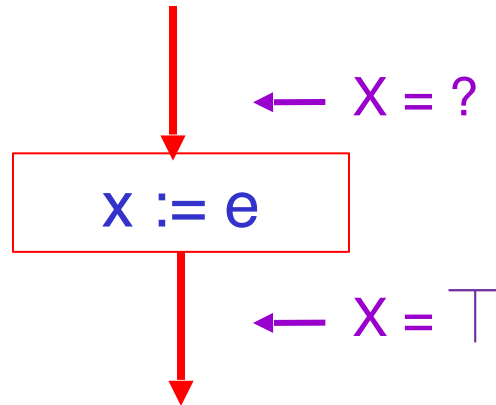- Now we need rules relating the in of a statement to the out of the same statement

# Rule 5



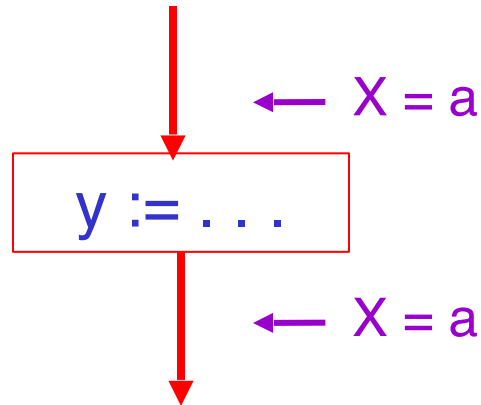$$C(s, x, \text{out}) = \bot \ \text{if} \ C(s, x, \text{in}) = \bot$$

# Rule 6



$$C(x := c, x, \text{out}) = c \text{ if } c \text{ is a constant}$$

# Rule 7



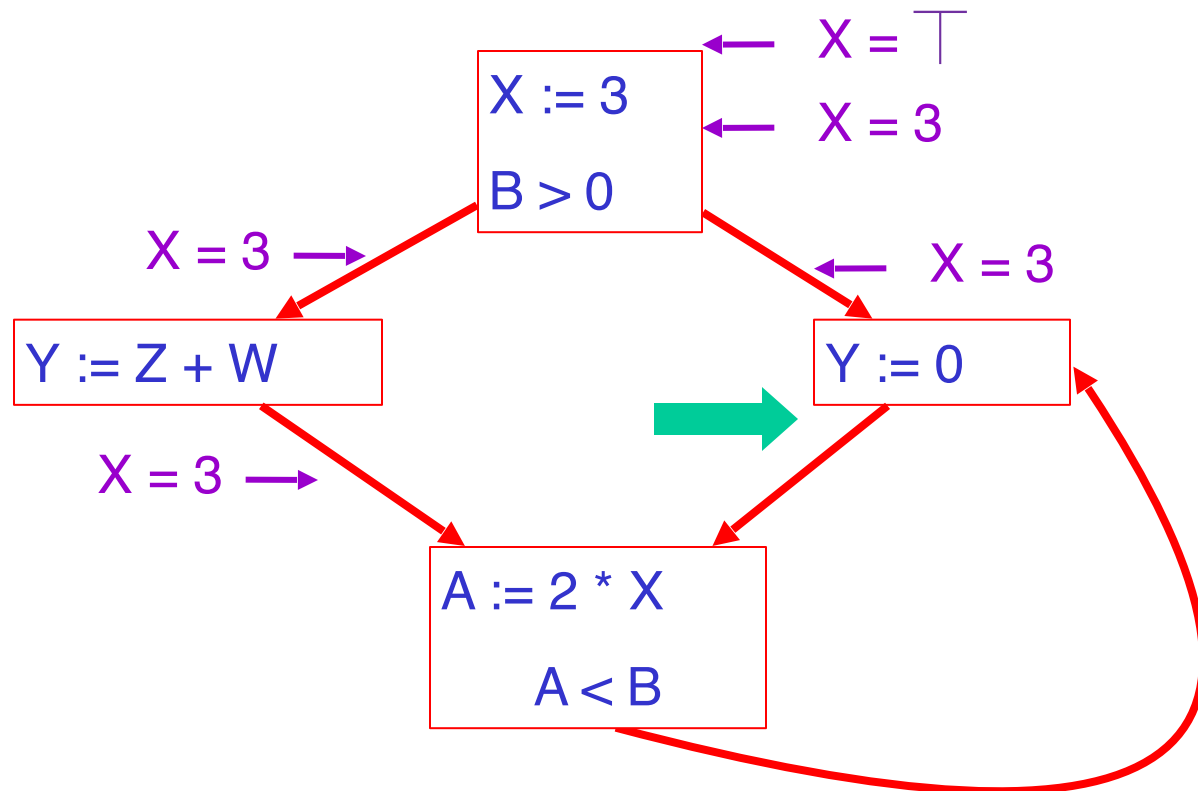$C(x := e, x, out) = \top$, where e is an expression that is not a constant

# Rule 8



$$C(y := …, x, out) = C(y := …, x, in) \text{ if } x <> y$$

# An Algorithm

1. For every entry s to the program, set C(s, x, in) = $\top$

2. Set C(s, x, in) = C(s, x, out) = $\bot$ everywhere else

3. Repeat until all points satisfy 1-8:
   Pick s not satisfying 1-8 and update using the appropriate rule

# The Value ⊥

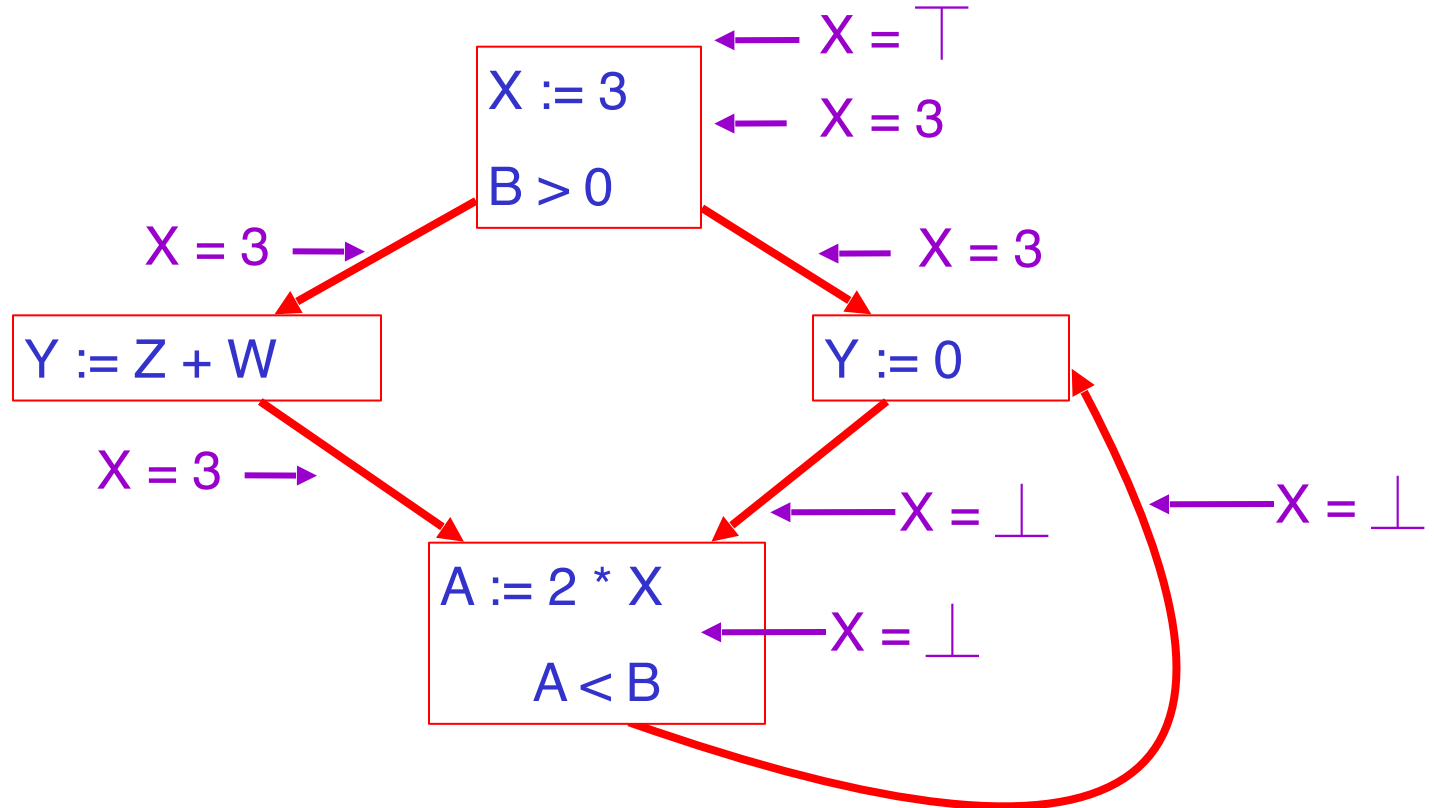- To understand why we need ⊥, look at a loop

# **Discussion**

- Consider the statement Y := 0

- To compute whether X is constant at this point, we need to know whether X is constant at the two predecessors
  - X := 3
  - A := 2 * X


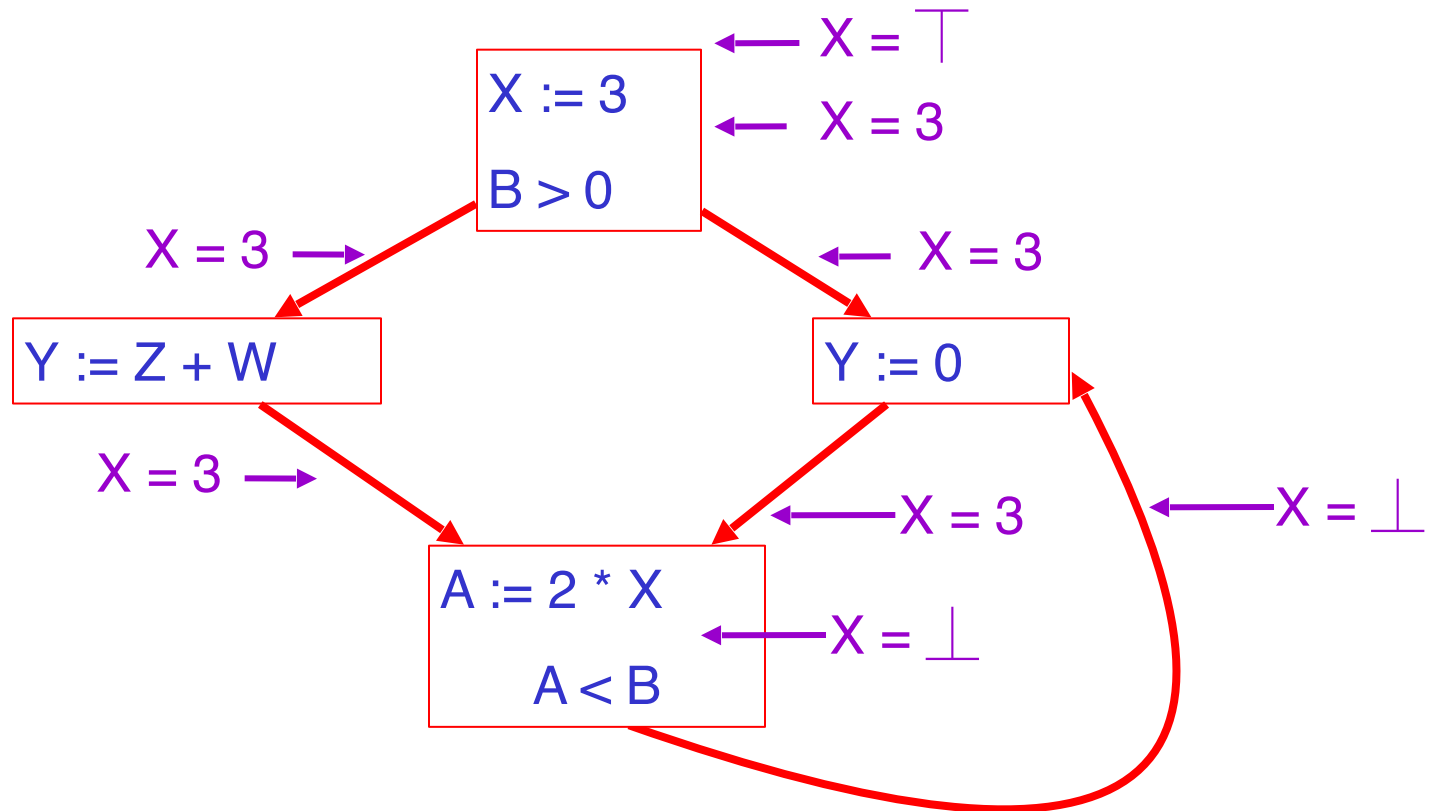- But info for A := 2 * X depends on its predecessors, including Y := 0!

# The Value **Z** (Cont.)

- Because of cycles, all points must have values at all times

- Intuitively, assigning some initial value allows the analysis to break cycles

- The initial value $\perp$ means "So far as we know so far, control never reaches this point"

# Example



A flow-graph with the following nodes and edge annotations:

- Node: X := 3 / B > 0
  - X = ⊤ (entering from top)
  - X = 3 (entering from right)
- Left branch: X = 3 → Y := Z + W
- Right branch: X = 3 → Y := 0
- Y := Z + W: X = 3 → A := 2 * X / A < B
- Y := 0: X = ⊥ → A := 2 * X
- A := 2 * X / A < B: X = ⊥
- Loop back to Y := 0: X = ⊥

# Example

# Example



X = ⊤

X := 3
B > 0

X = 3

X = 3
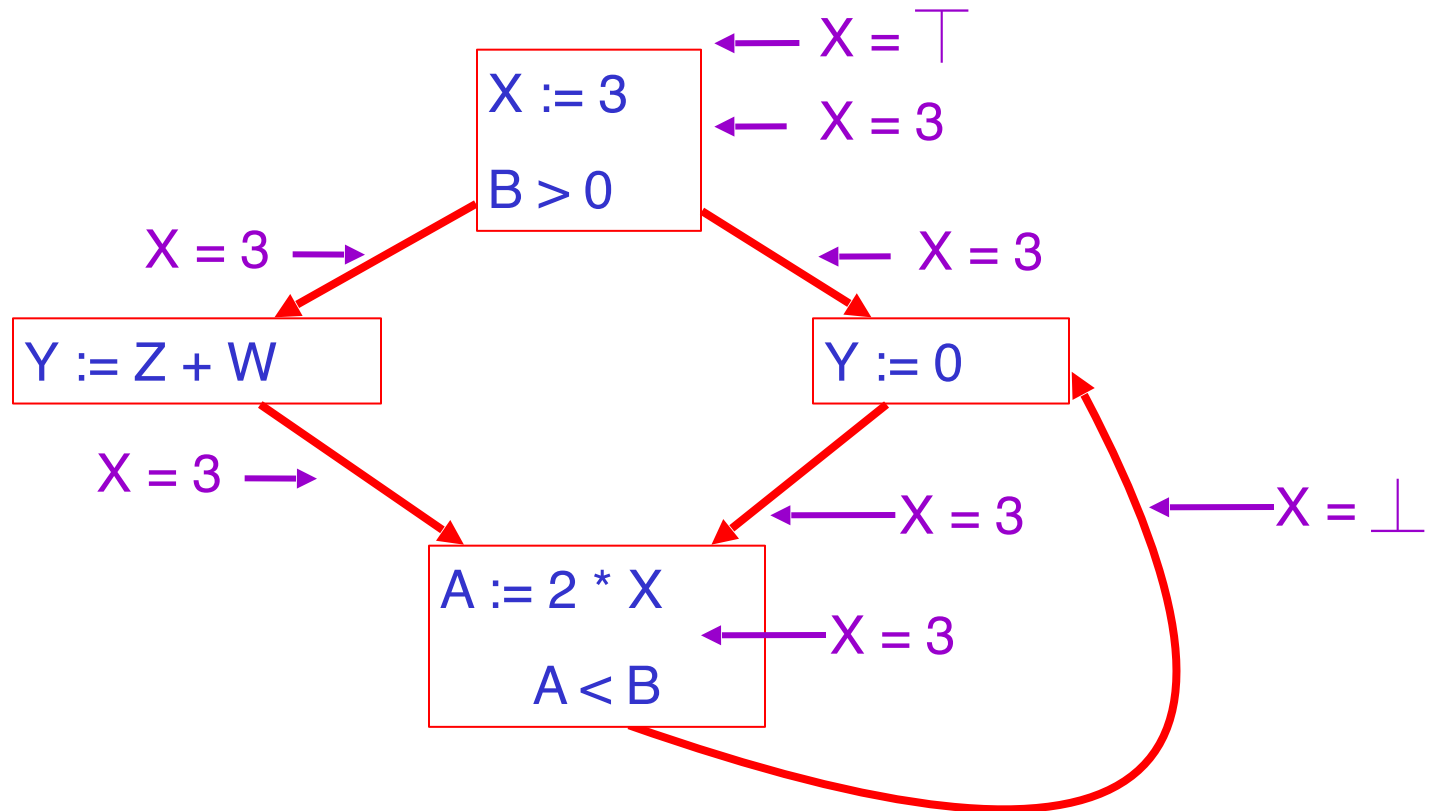
X = 3
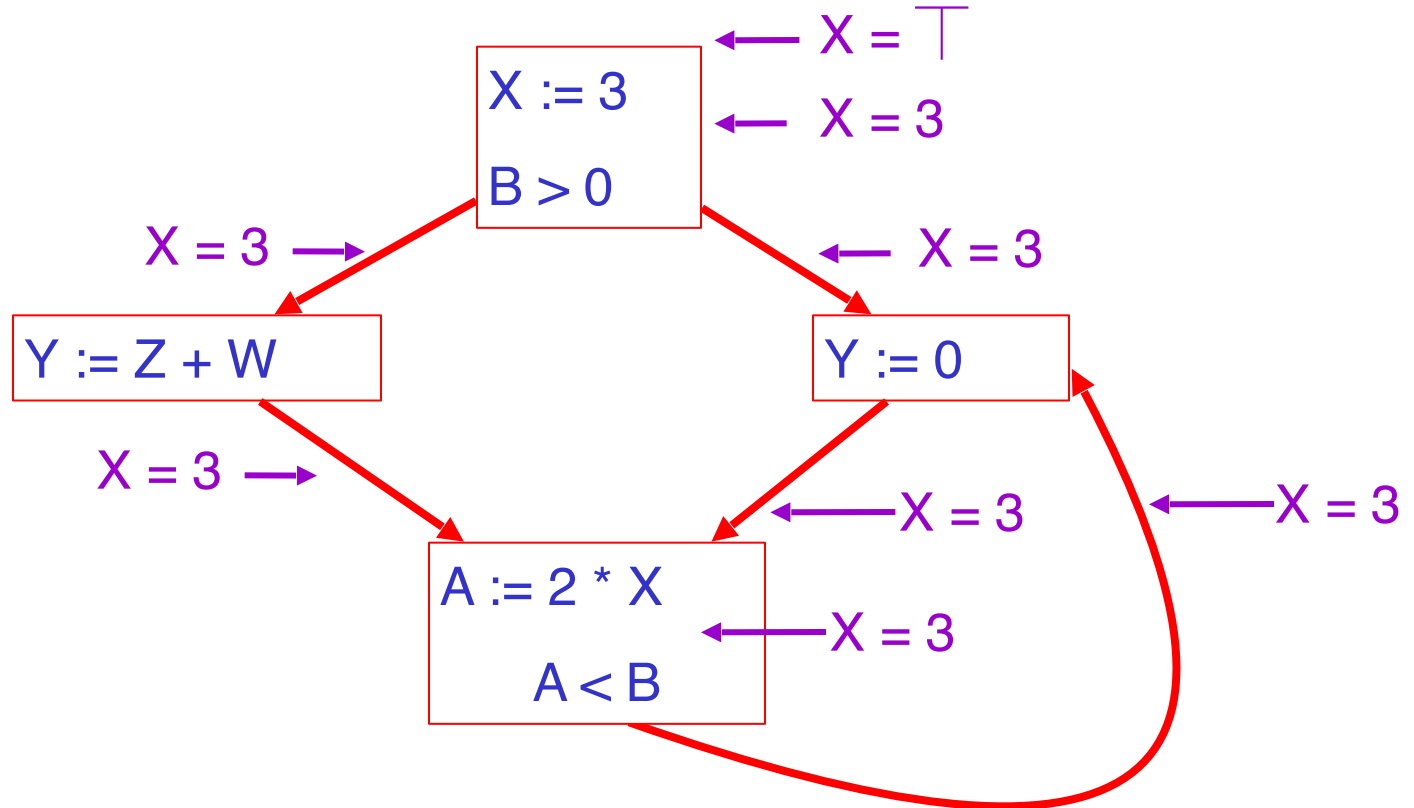
Y := Z + W

Y := 0

X = 3
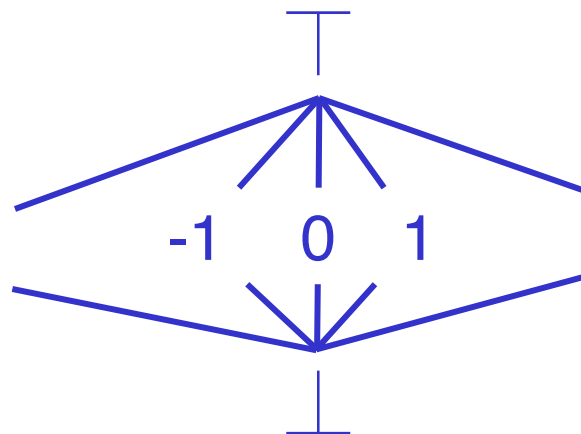
X = 3

X = ⊥

A := 2 * X

A < B

X = 3

# Example

# Orderings

- We can simplify the presentation of the analysis by ordering the values

$$\bot < c < \top$$

- Drawing a picture with "lower" values drawn lower, we get

# Orderings (Cont.)

- $\top$ is the greatest value, $\bot$ is the least
  - All constants are in between and incomparable

- Let lub be the least-upper bound in this ordering

- Rules 1-4 can be written using lub:

  C(s, x, in) = lub { C(p, x, out) | p is a predecessor of s }

# Termination

- Simply saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes

- The use of lub explains why the algorithm terminates
  - Values start as $\bot$ and only increase

    $\bot$ can change to a constant, and a constant to $\top$
  - Thus, C(s, x, _) can change at most twice

# Termination (Cont.)
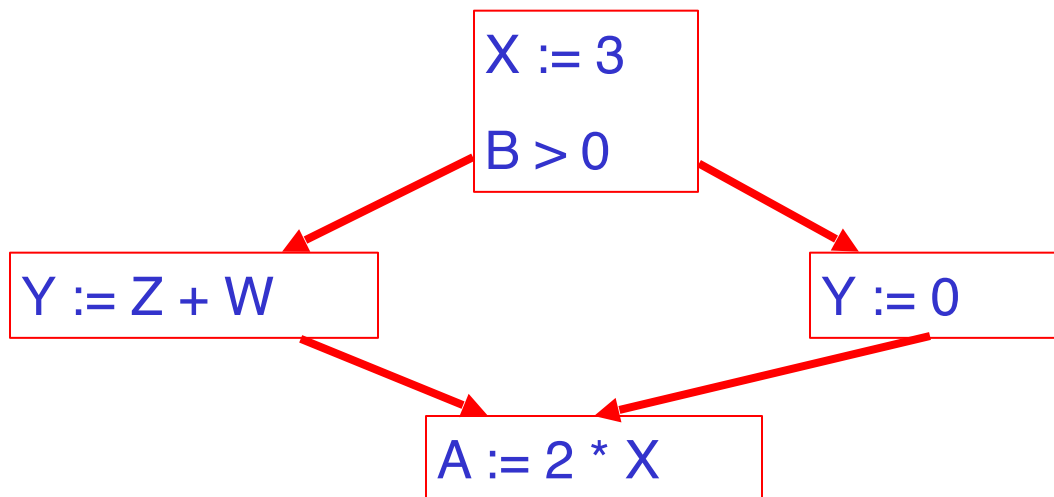
Thus the algorithm is linear in program size

Number of steps =

Number of C(….) value computed * 2 =
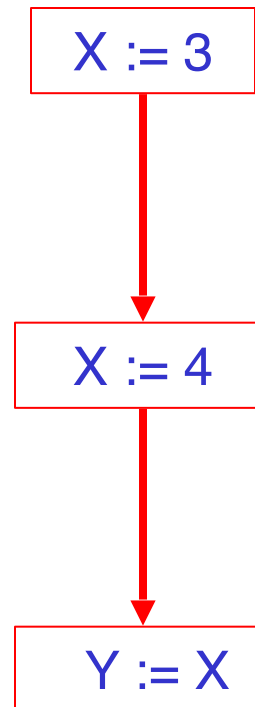
Number of program statements * 4

# Liveness Analysis

Once constants have been globally propagated, we would like to eliminate dead code



After constant propagation, X := 3 is dead (assuming X not used elsewhere)

# Live and Dead

- The first value of x is dead (never used)

- The second value of x is live (may be used)

- Liveness is an important concept

X := 3

X := 4

Y := X

# Liveness

A variable x is live at statement s if
- – There exists a statement s' that uses x

- – There is a path from s to s'
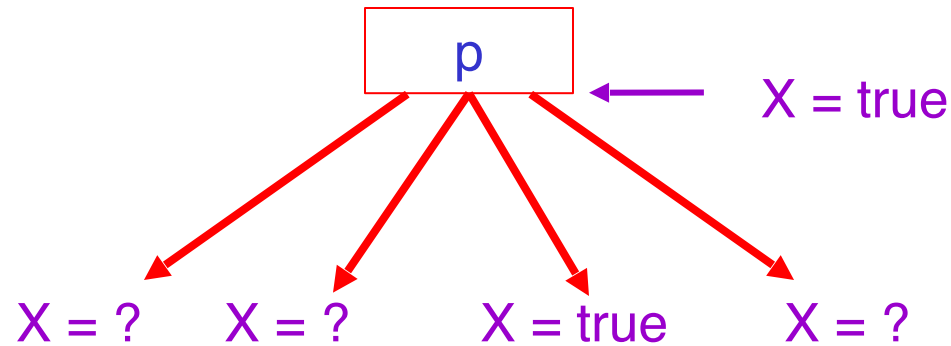
- – That path has no intervening assignment to x

# Global Dead Code Elimination

- A statement x := … is dead code if x is dead after the assignment

- Dead statements can be deleted from the program

- But we need liveness information first . . .
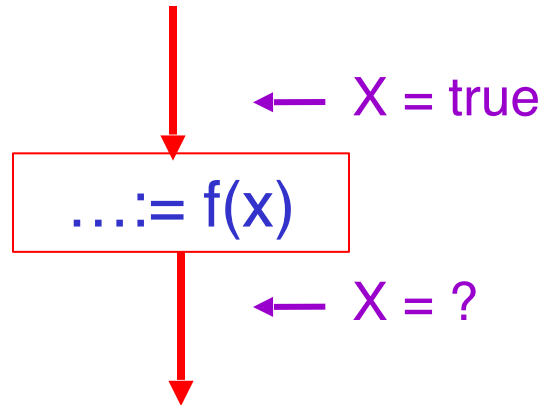
# Computing Liveness

- We can express liveness in terms of information transferred between adjacent statements, just as in copy propagation

- Liveness is simpler than constant propagation, since it is a boolean property (true or false)
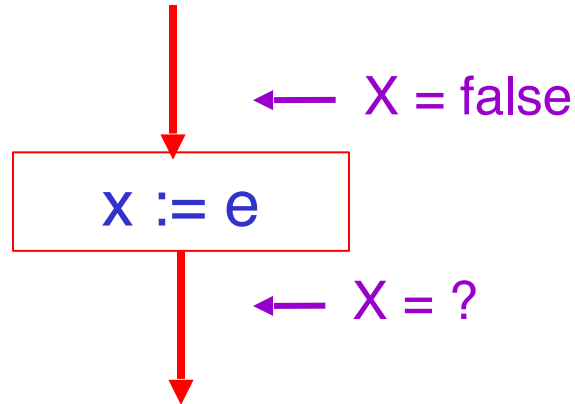
# Liveness Rule 1



$$L(p, x, out) = \vee \{ L(s, x, in) \mid s \text{ a successor of } p \}$$

# Liveness Rule 2



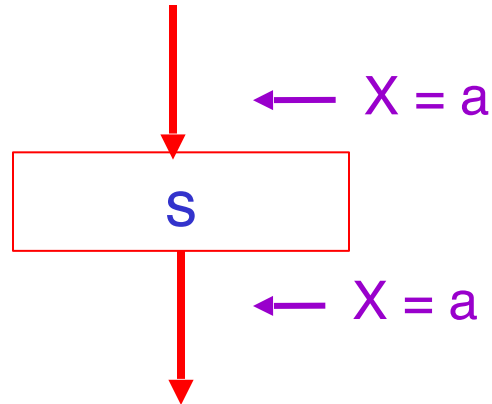$$L(s, x, in) = true \quad \text{if } s \text{ refers to } x \text{ on the rhs}$$

# Liveness Rule 3

x := e

X = false

X = ?

$L(x := e, x, in) = false$  if e does not refer to x

# Liveness Rule 4



L(s, x, in) = L(s, x, out) if s does not refer to x

# Algorithm

1. Let all L(…) = false initially

2. Repeat until all statements s satisfy rules 1-4
   Pick s where one of 1-4 does not hold and update using the appropriate rule

# Termination

- A value can change from <span style="color:blue">false</span> to <span style="color:blue">true</span>, but not the other way around

- Each value can change only once, so termination is guaranteed

- Once the analysis is computed, it is simple to eliminate dead code

# Forward vs. Backward Analysis

We've seen two kinds of analysis:

Constant propagation is a forwards analysis: information is pushed from inputs to outputs

Liveness is a backwards analysis: information is pushed from outputs back towards inputs

# **Analysis**

- There are many other global flow analyses

- Most can be classified as either forward or backward

- Most also follow the methodology of local rules relating information between adjacent program points