

# Global Optimization

## Lecture 10

# Announcements

---

- PA 4
  - Due today at 11:59pm
  
- WA 4
  - Assigned today

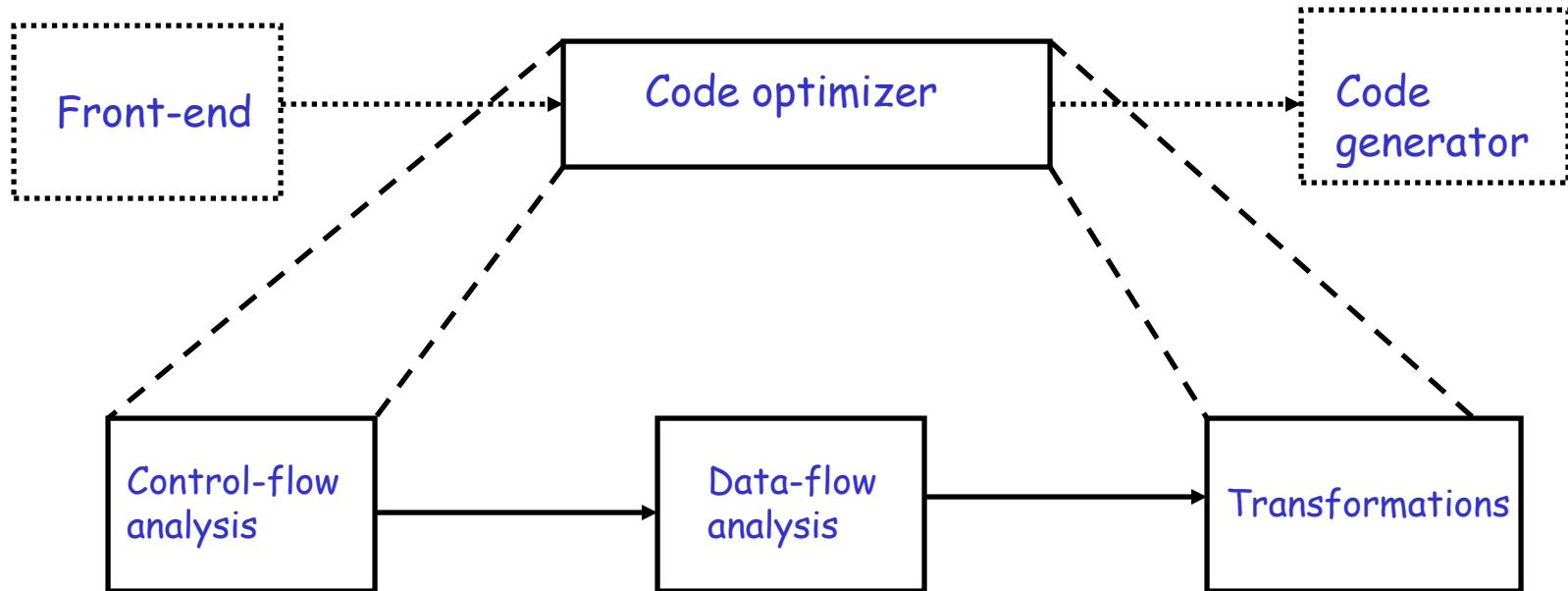
# Lecture Outline

---

- Global flow analysis
- Global constant propagation
- Liveness analysis

# Organization of a Code Optimizer (Revisited)

---

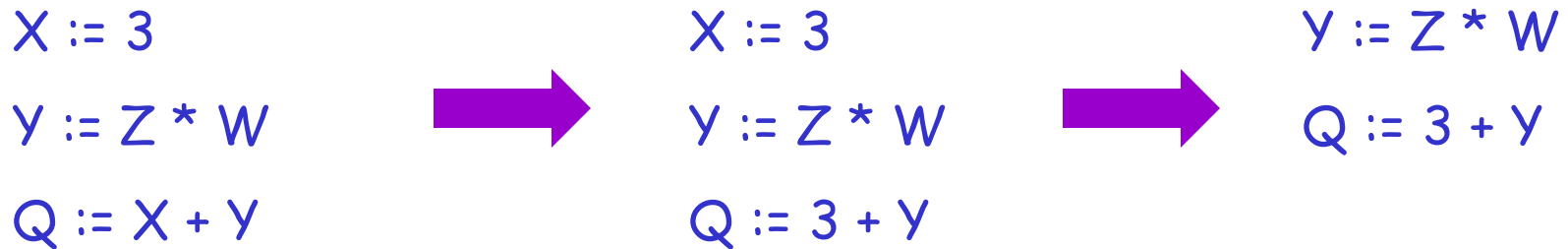


# Local Optimization

---

Recall the simple basic-block optimizations

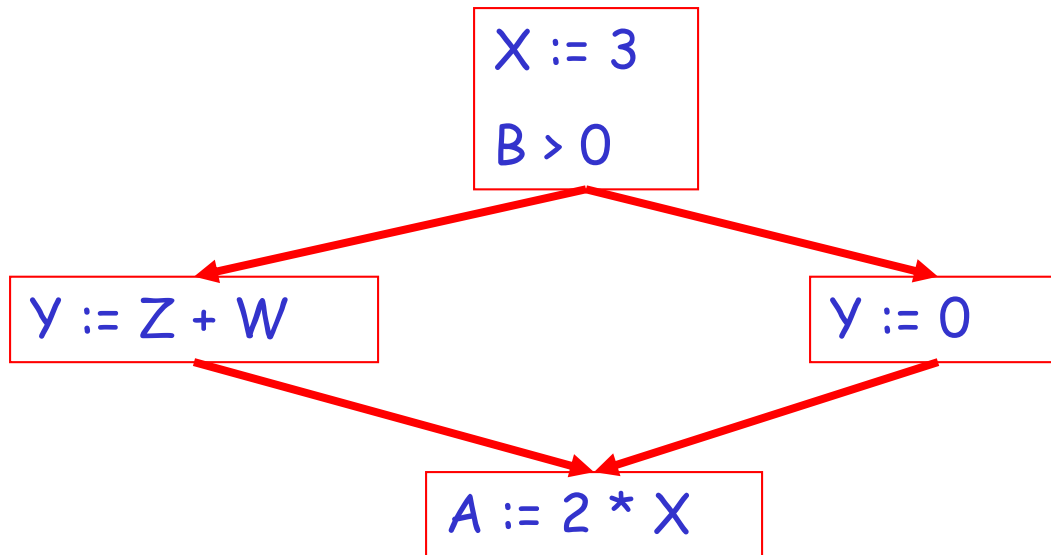
- Constant propagation
- Dead code elimination



# Global Optimization

---

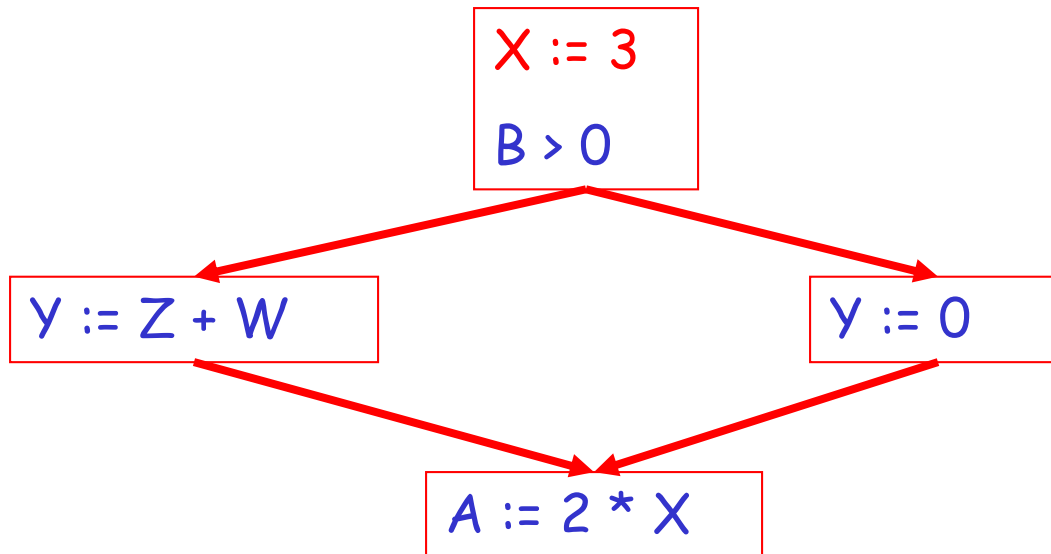
These optimizations can be extended to an entire control-flow graph



# Global Optimization

---

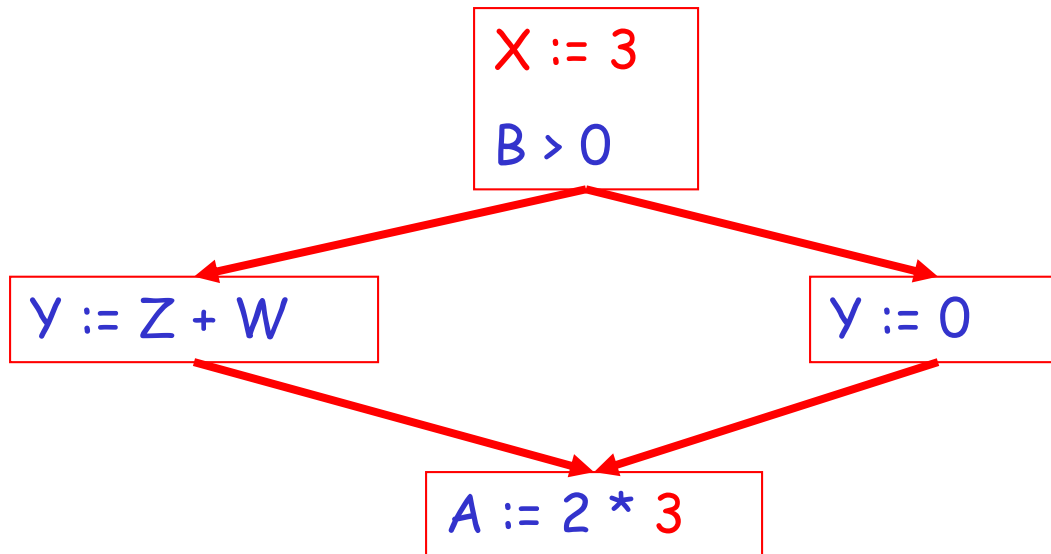
These optimizations can be extended to an entire control-flow graph



# Global Optimization

---

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

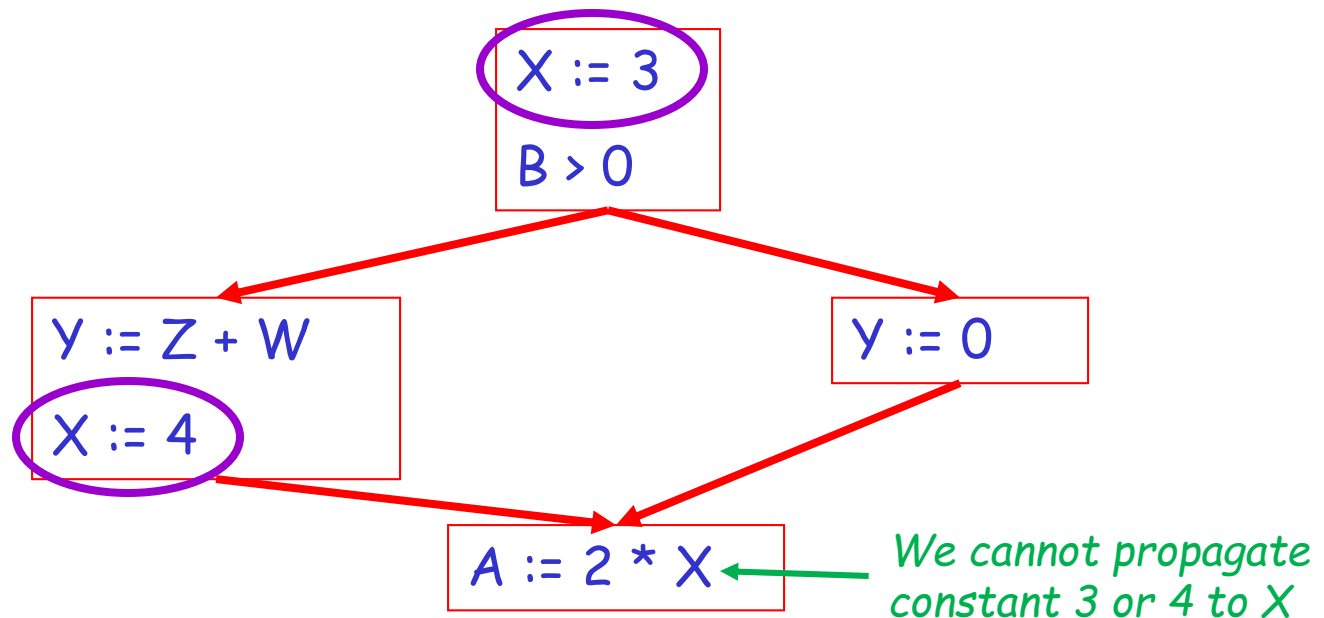




# Correctness

---

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



## Correctness (Cont.)

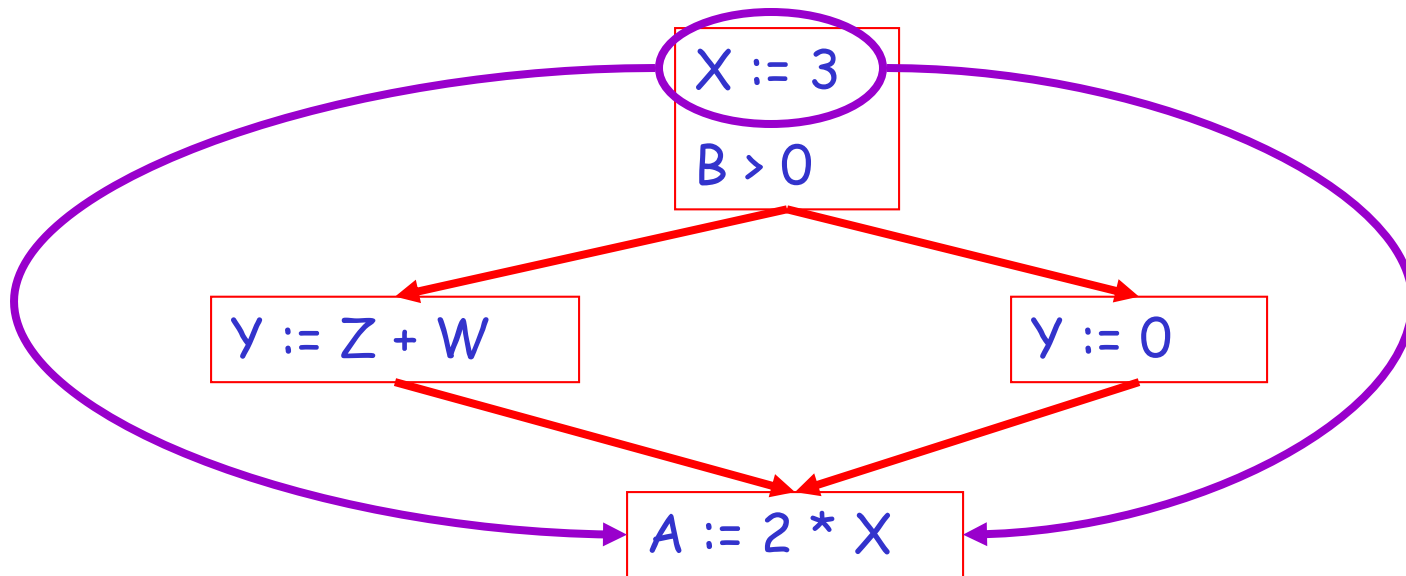
---

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  $x := k$  \*\**

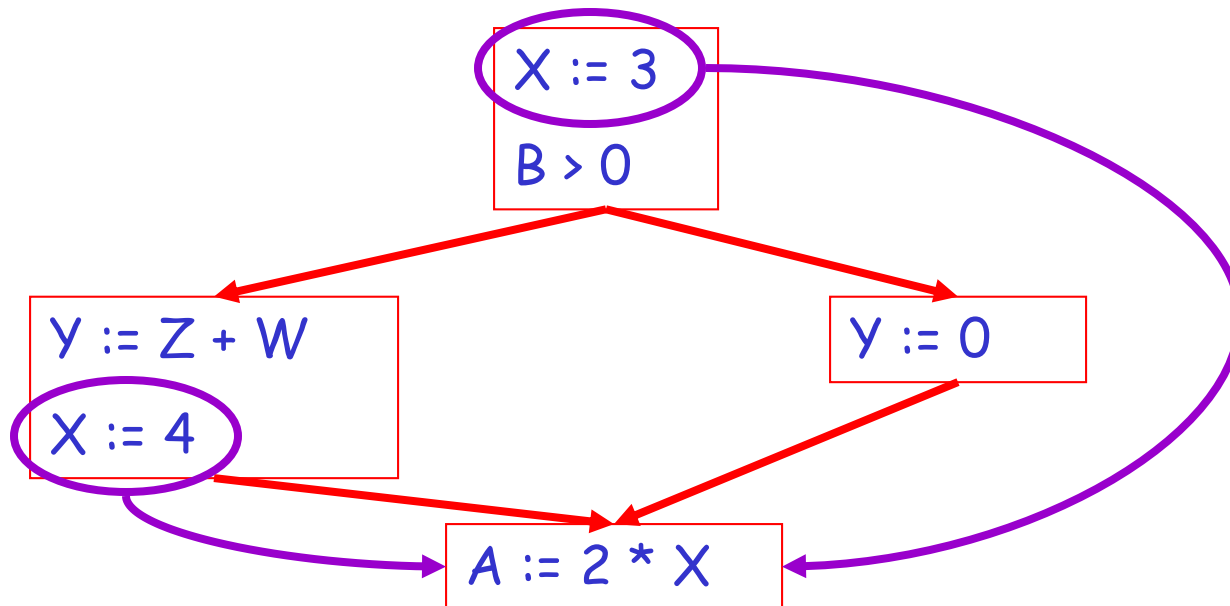
# Example 1 Revisited

---



# 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 dataflow 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 program
- 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 (we don't do the optimization)
- 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

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

- Consider the case of computing **\*\*** for a single variable  $x$  at all program points



# Global Constant Propagation (Cont.)

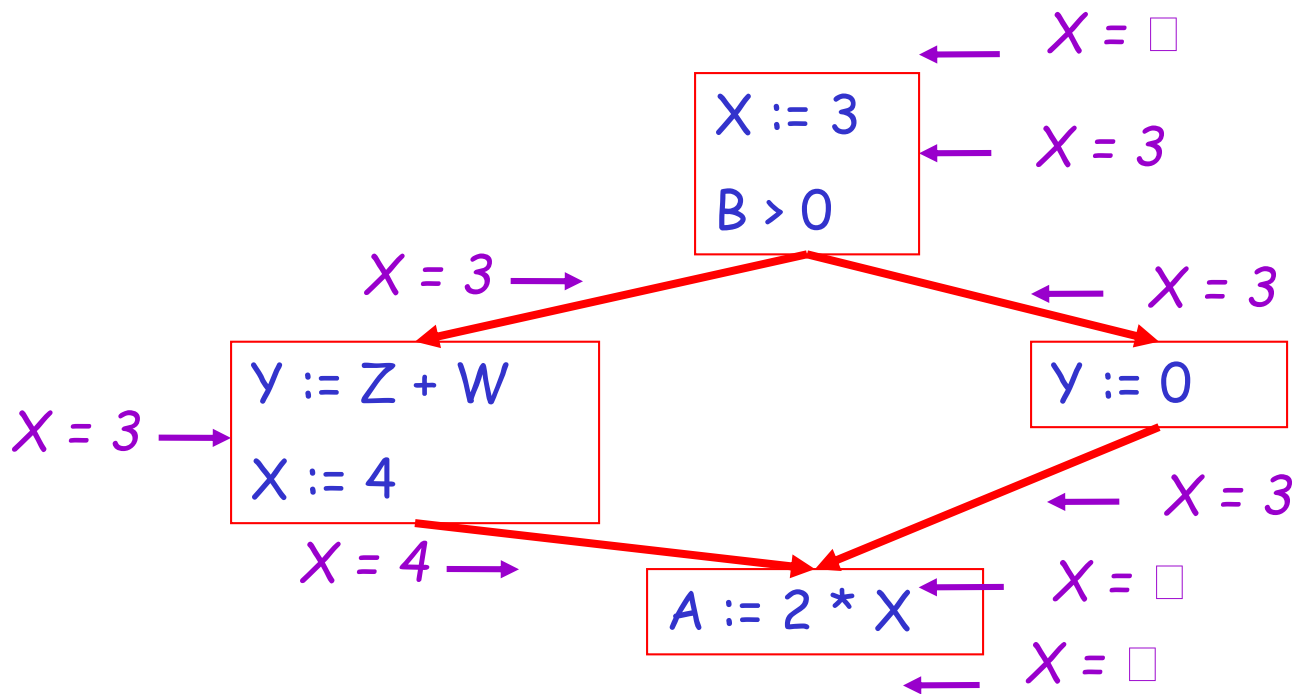
---

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

<i>value</i>	<i>interpretation</i>
⊗	This statement never executes
$c$	$X = \text{constant } c$
□	$X$ 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*

# 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  $s$

$C(x,s,in)$  = value of  $x$  before  $s$  is executed

$C(x,s,out)$  = value of  $x$  after  $s$  is executed

*Stands for  
'constant'  
information*

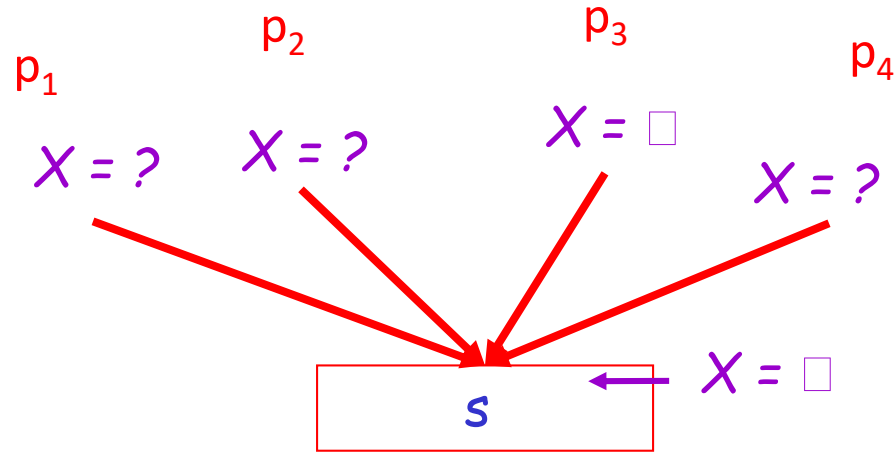
# Transfer Functions

---

- Define a *transfer* function that transfers information one statement to another
- In the following rules, let statement  $s$  have immediate predecessor statements  $p_1, \dots, p_n$

# Rule 1

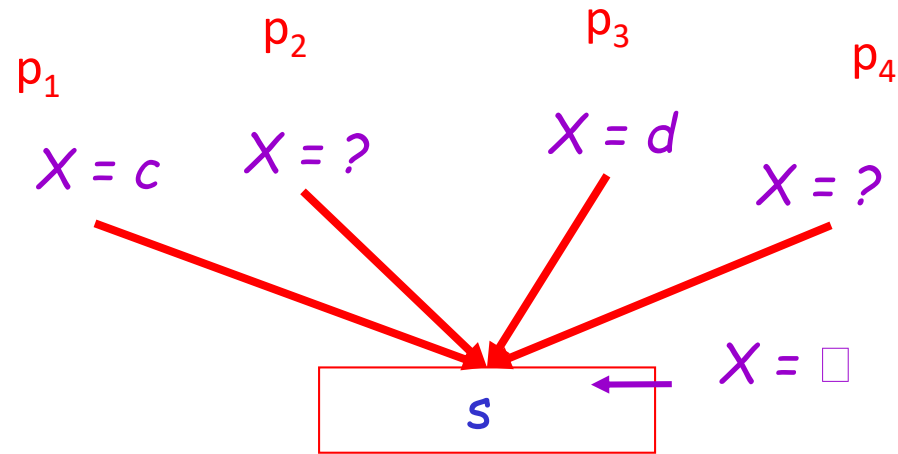
---



if  $\exists_i (C(p_i, x, \text{out}) = \square)$   
then  $C(s, x, \text{in}) = \square$

# Rule 2

---

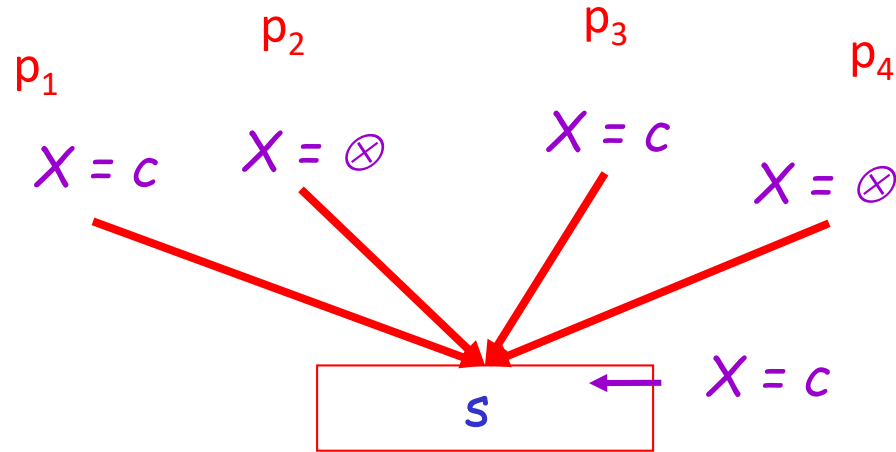


if  $\exists_{i,j} (C(p_i, x, \text{out}) = c \ \& \ C(p_j, x, \text{out}) = d \ \& \ d \leftrightarrow c)$   
then  $C(s, x, \text{in}) = \square$



# Rule 3

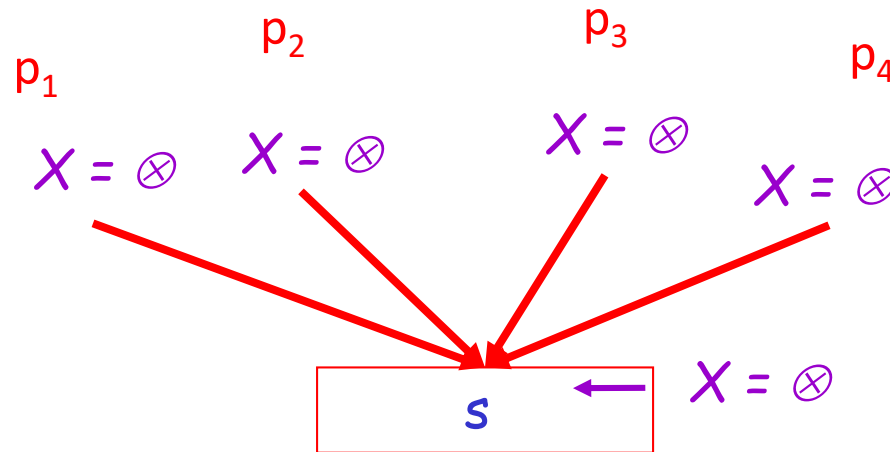
---



if  $\forall_i (C(p_i, x, \text{out}) = c \text{ or } \oplus)$   
then  $C(s, x, \text{in}) = c$

# Rule 4

---



if  $\forall_i (C(p_i, x, \text{out}) = \oplus)$   
then  $C(s, x, \text{in}) = \oplus$

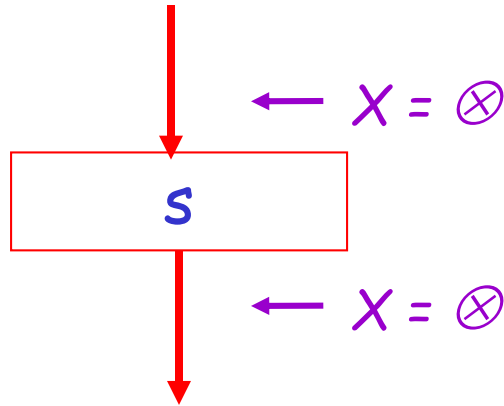
# The Other Half

---

- 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

---



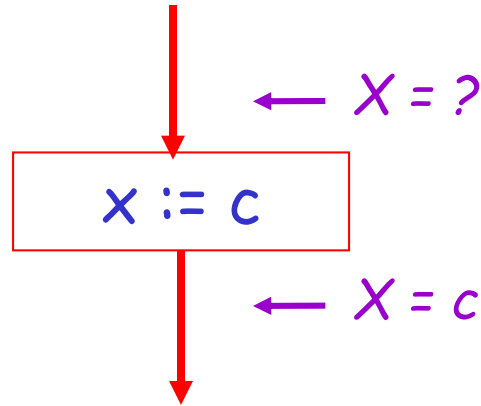
if  $C(s, x, \text{in}) = \otimes$   
then  $C(s, x, \text{out}) = \otimes$

# Rule 6

---

*Rule 6 has a lower priority than Rule 5*

*R6 is checked only when R5 cannot be applied*



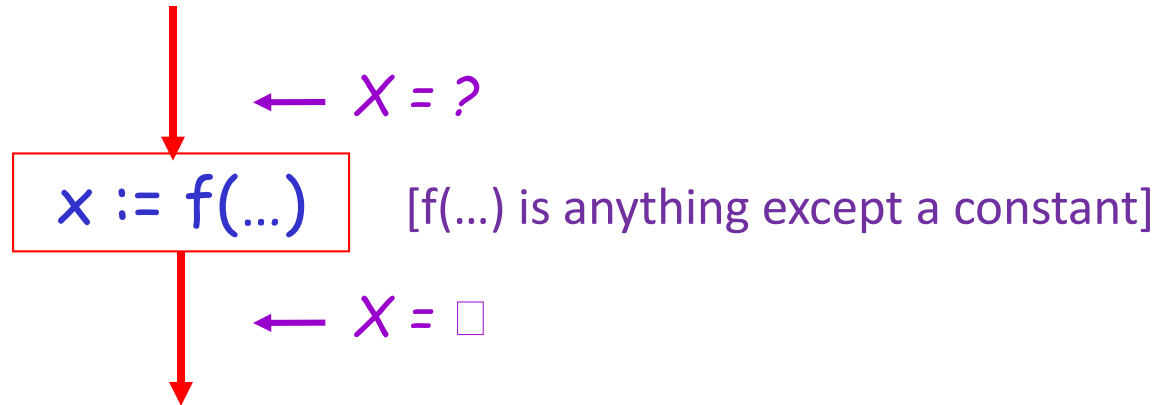
if  $c$  is a constant  
then  $C(x := c, x, out) = c$

# Rule 7

---

*Rule 7 has a lower priority than Rule 5*

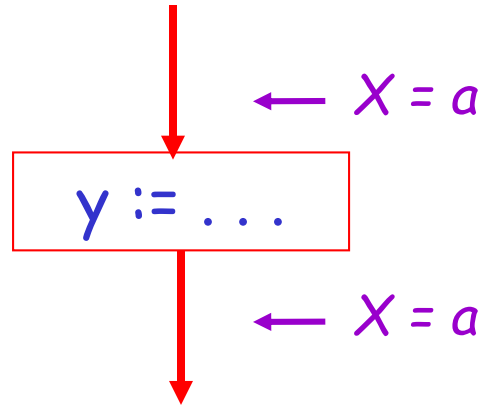
*R7 is checked only when R5 cannot be applied*



$$C(x := f(\dots), x, \text{out}) = \square$$

# Rule 8

---



if  $x \leftrightarrow y$   
then  $C(y := \dots, x, \text{out}) = C(y := \dots, x, \text{in})$

# An Algorithm

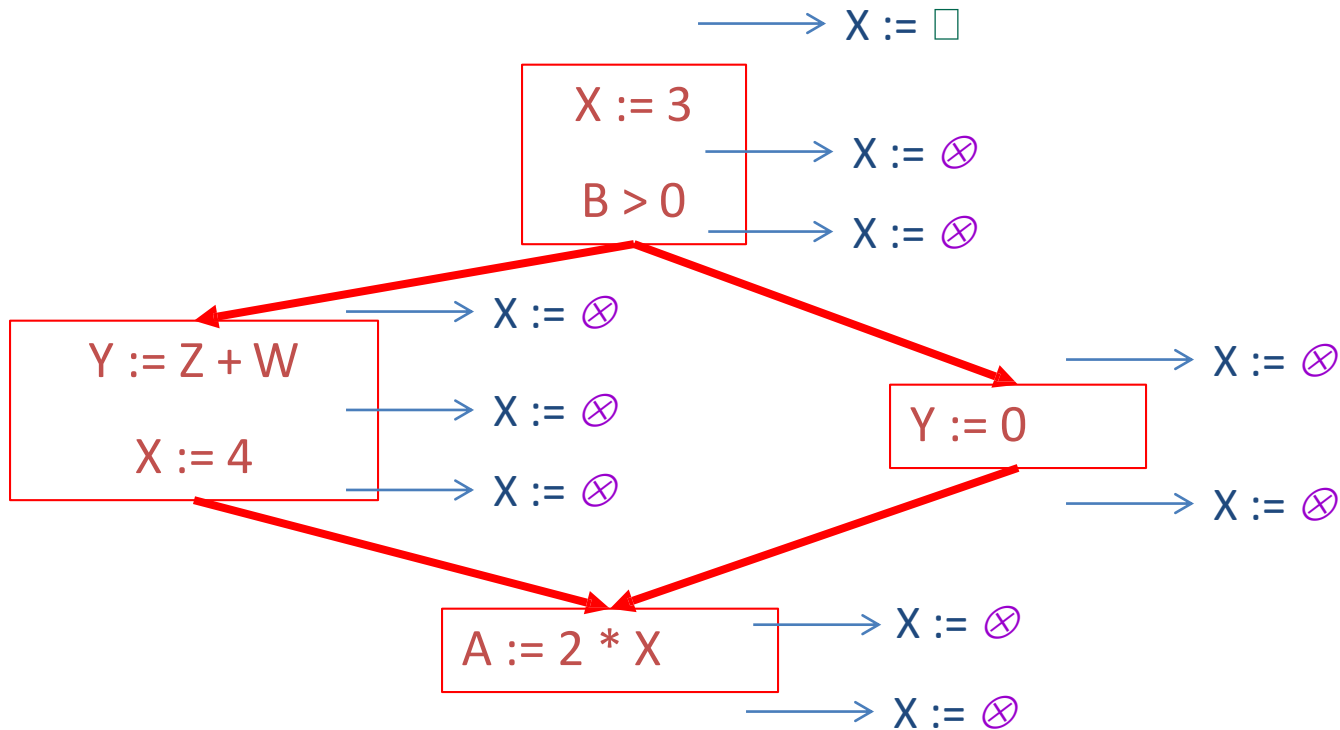
---

1. For every entry  $s$  to the program, set  $C(s, x, in) = \square$
2. Set  $C(s, x, in) = C(s, x, out) = \otimes$  everywhere else
3. Repeat until all points satisfy 1-8:  
Pick  $s$  not satisfying 1-8 and update using the appropriate rule

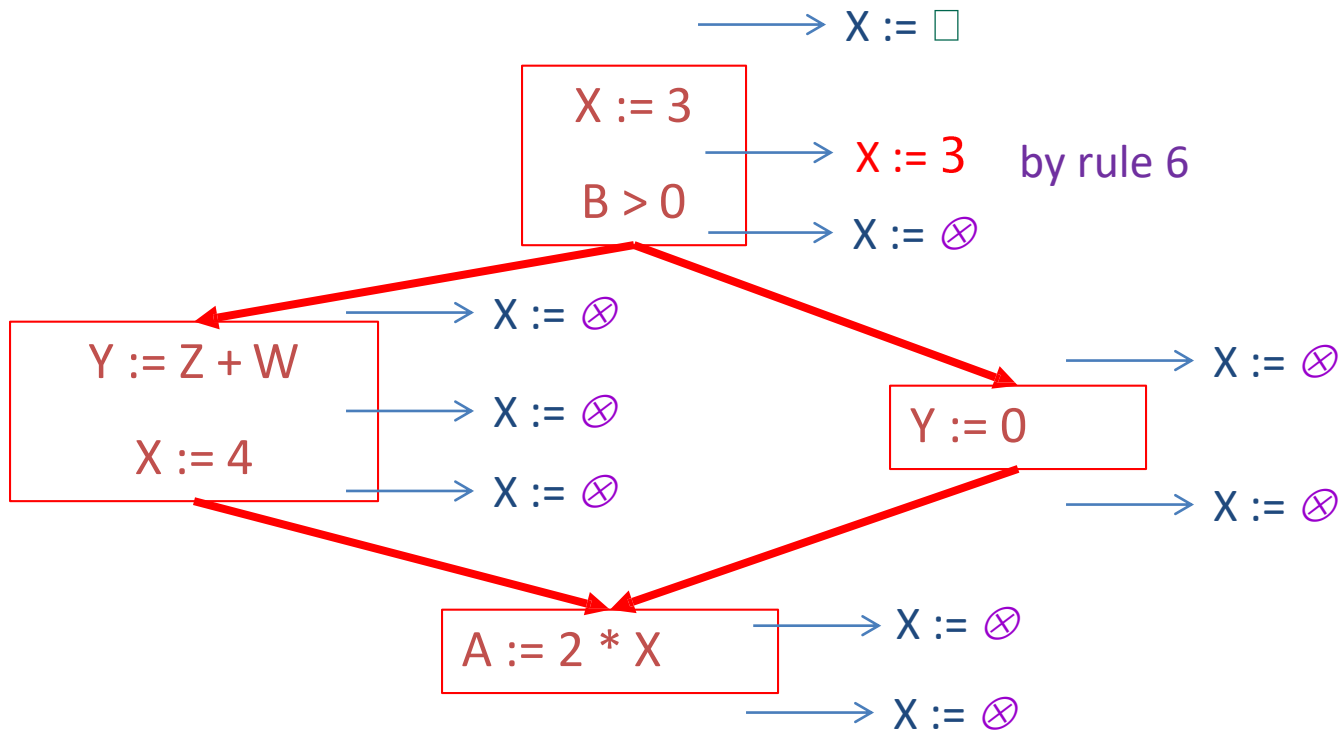


# Constant Propagation

---

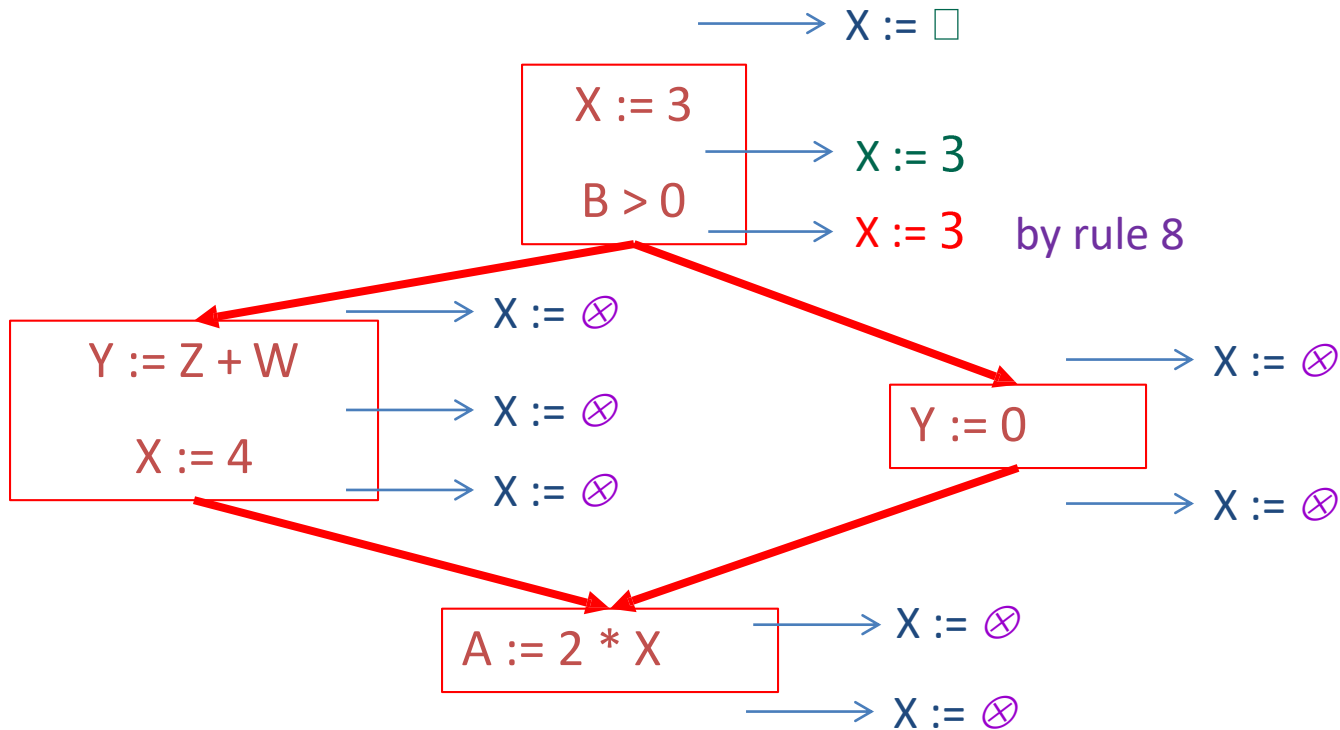


# Constant Propagation



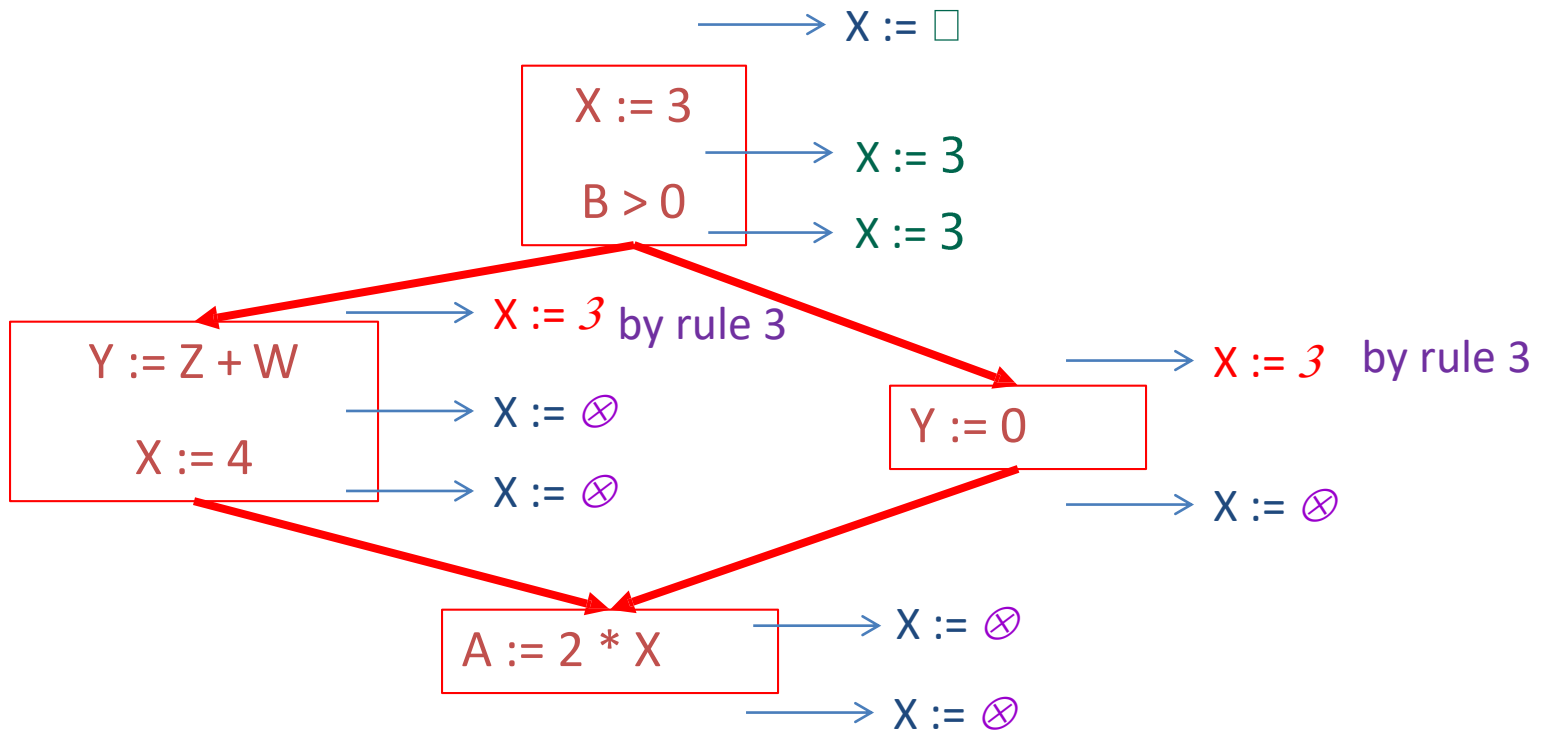
Rule 6: if  $c$  is a constant then  $C(x := c, x, out) = c$

# Constant Propagation



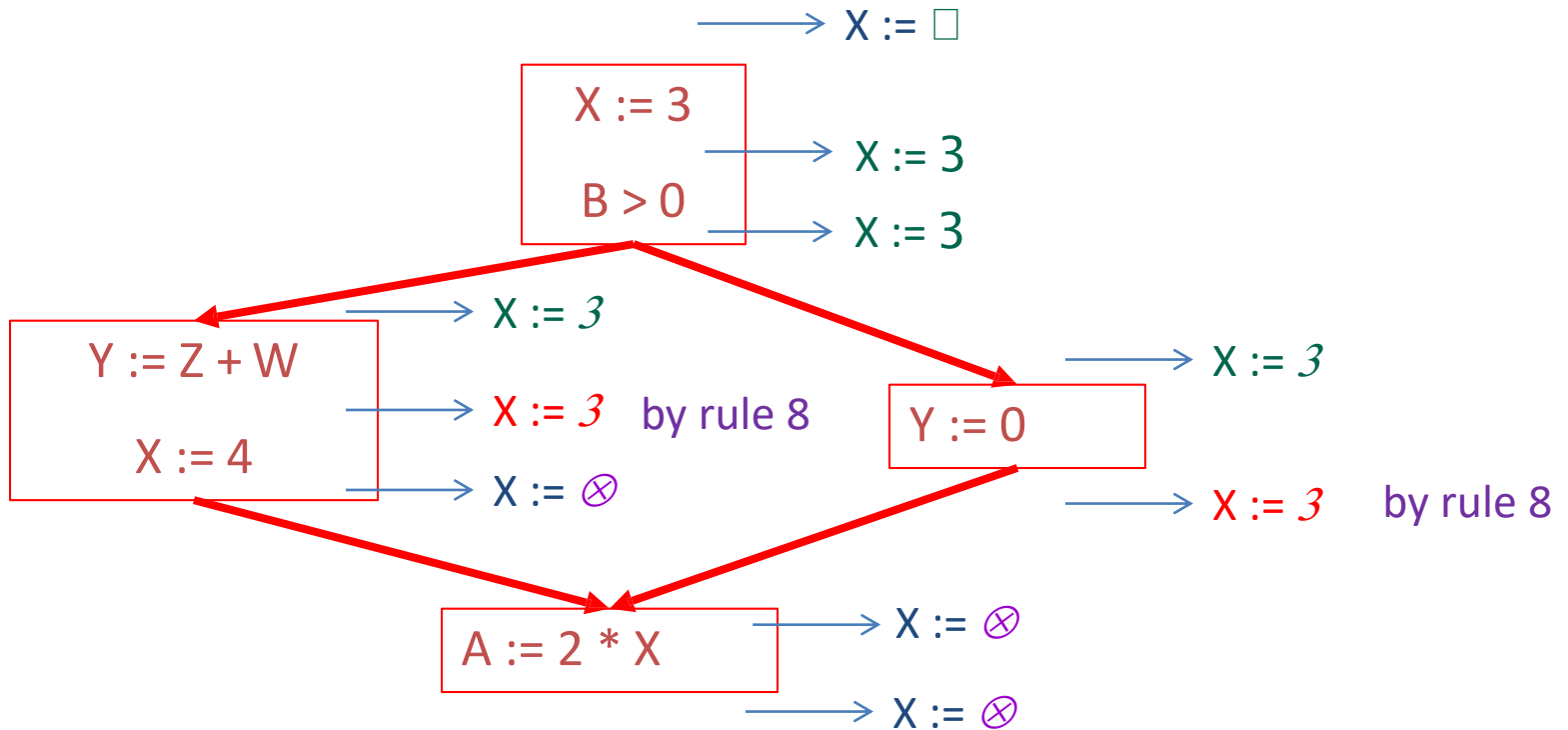
Rule 8: if  $x \leftrightarrow y$  then  $C(y := \dots, x, \text{out}) = C(y := \dots, x, \text{in})$

# Constant Propagation



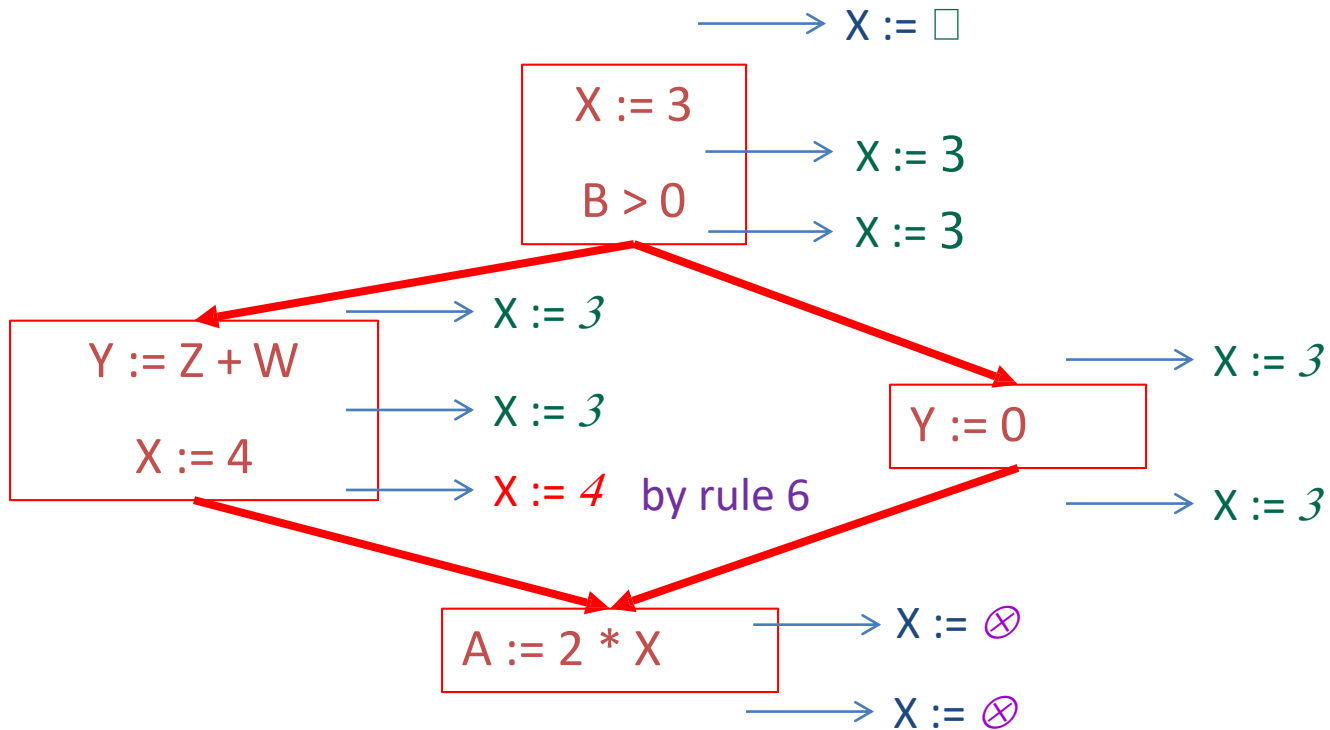
Rule 3: if  $\forall_i (C(p_i, x, out) = c \text{ or } \otimes)$  then  $C(s, x, in) = c$

# Constant Propagation



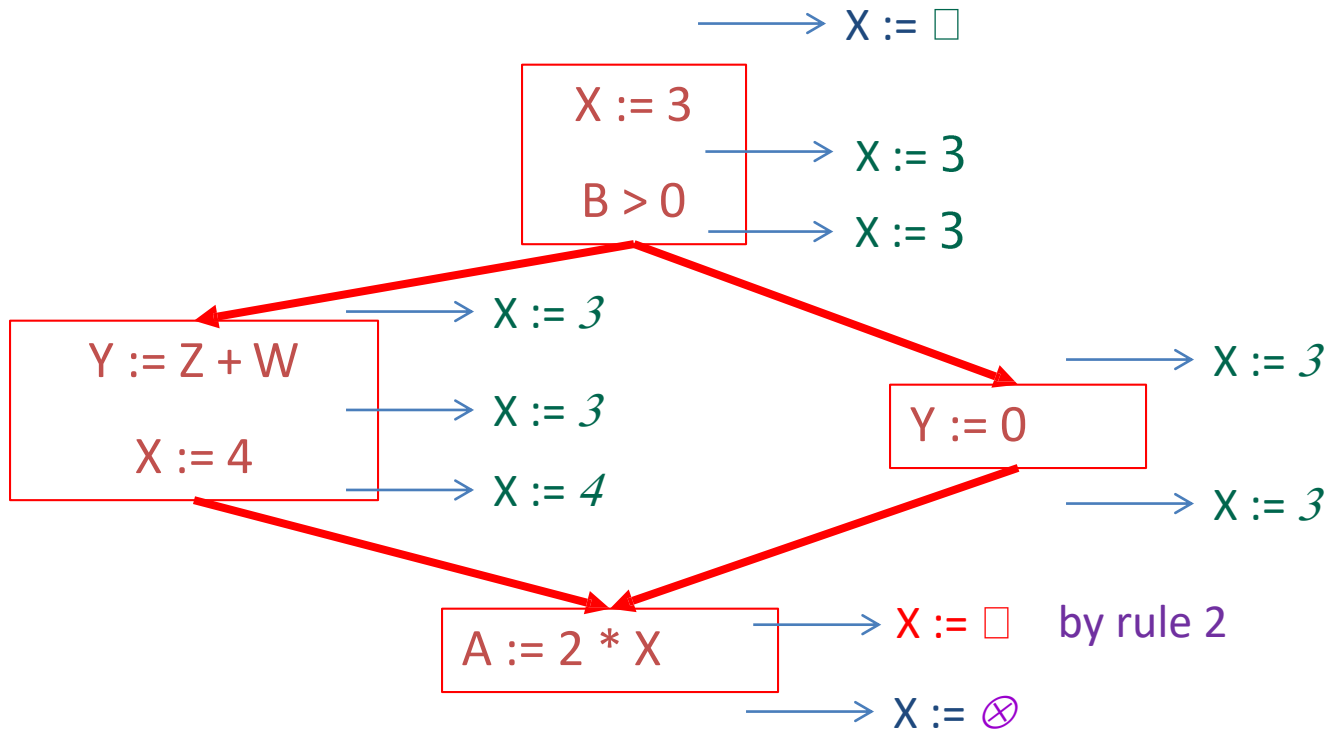
Rule 8: if  $x \leftrightarrow y$  then  $C(y := \dots, x, \text{out}) = C(y := \dots, x, \text{in})$

# Constant Propagation



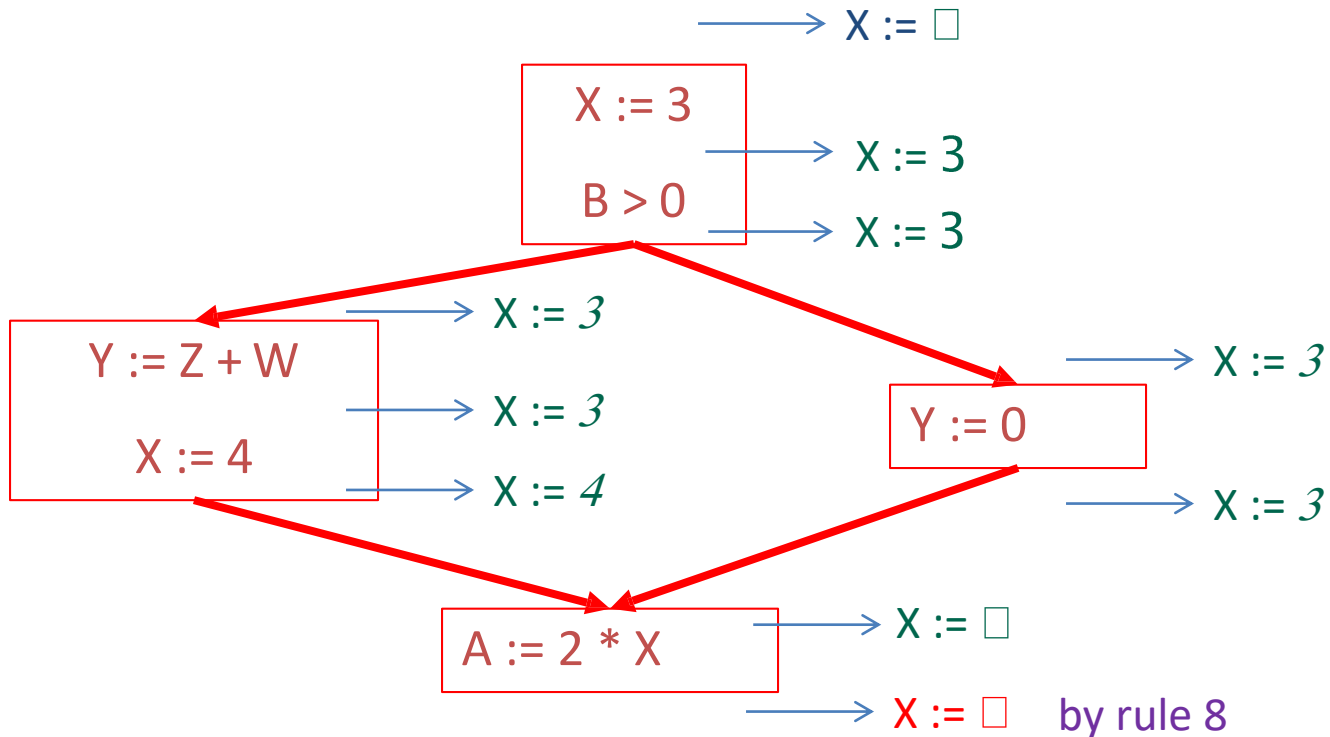
Rule 6: if  $c$  is a constant then  $C(x := c, x, \text{out}) = c$

# Constant Propagation



Rule 2: if  $\exists_{i,j} (C(p_i, x, out) = c \ \& \ C(p_j, x, out) = d \ \& \ d \leftrightarrow c)$   
 then  $C(s, x, in) = \square$

# Constant Propagation

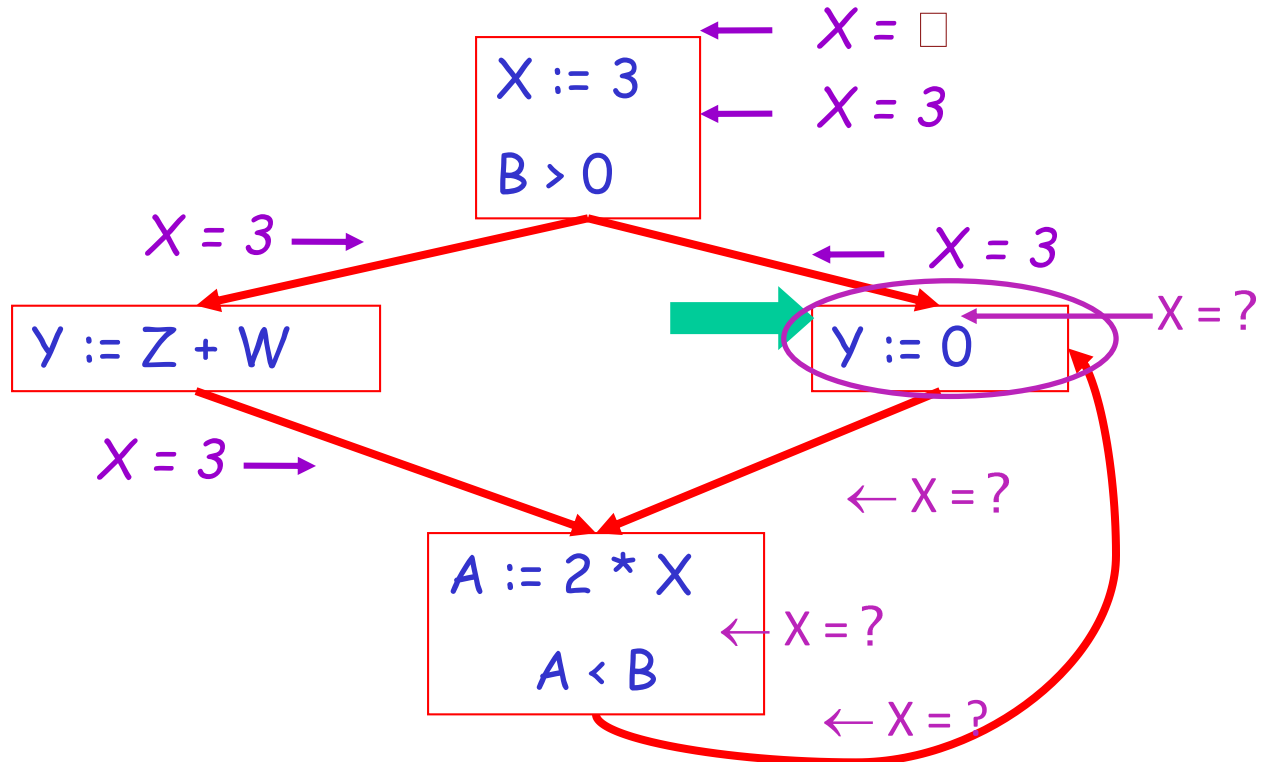


Rule 8: if  $x \leftrightarrow y$  then  $C(y := \dots, x, \text{out}) = C(y := \dots, x, \text{in})$



# The Value $\otimes$

- To understand why we need  $\otimes$ , 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  $Y := 0$ !

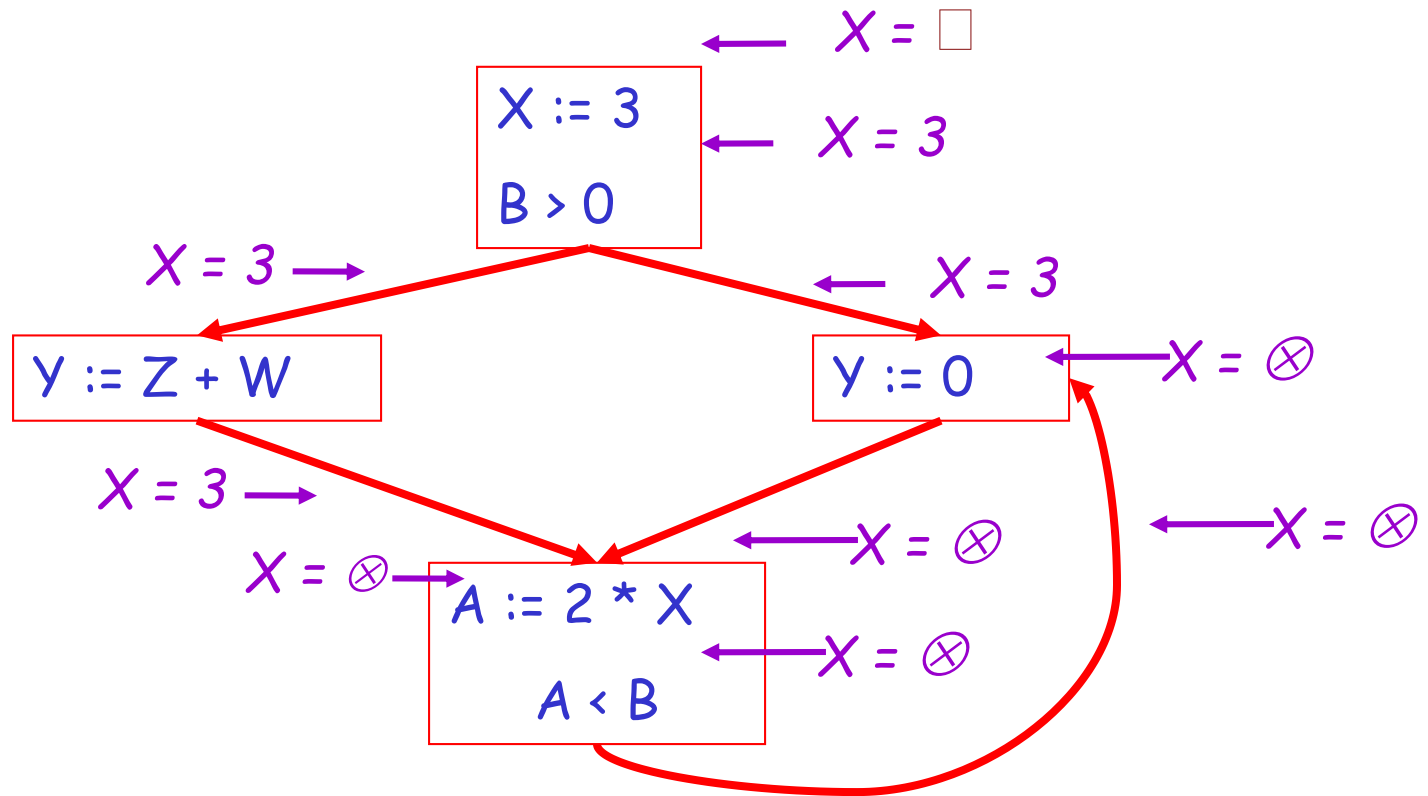
## The Value $\otimes$ (Cont.)

---

- 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  $\otimes$  means “So far as we know, control never reaches this point”

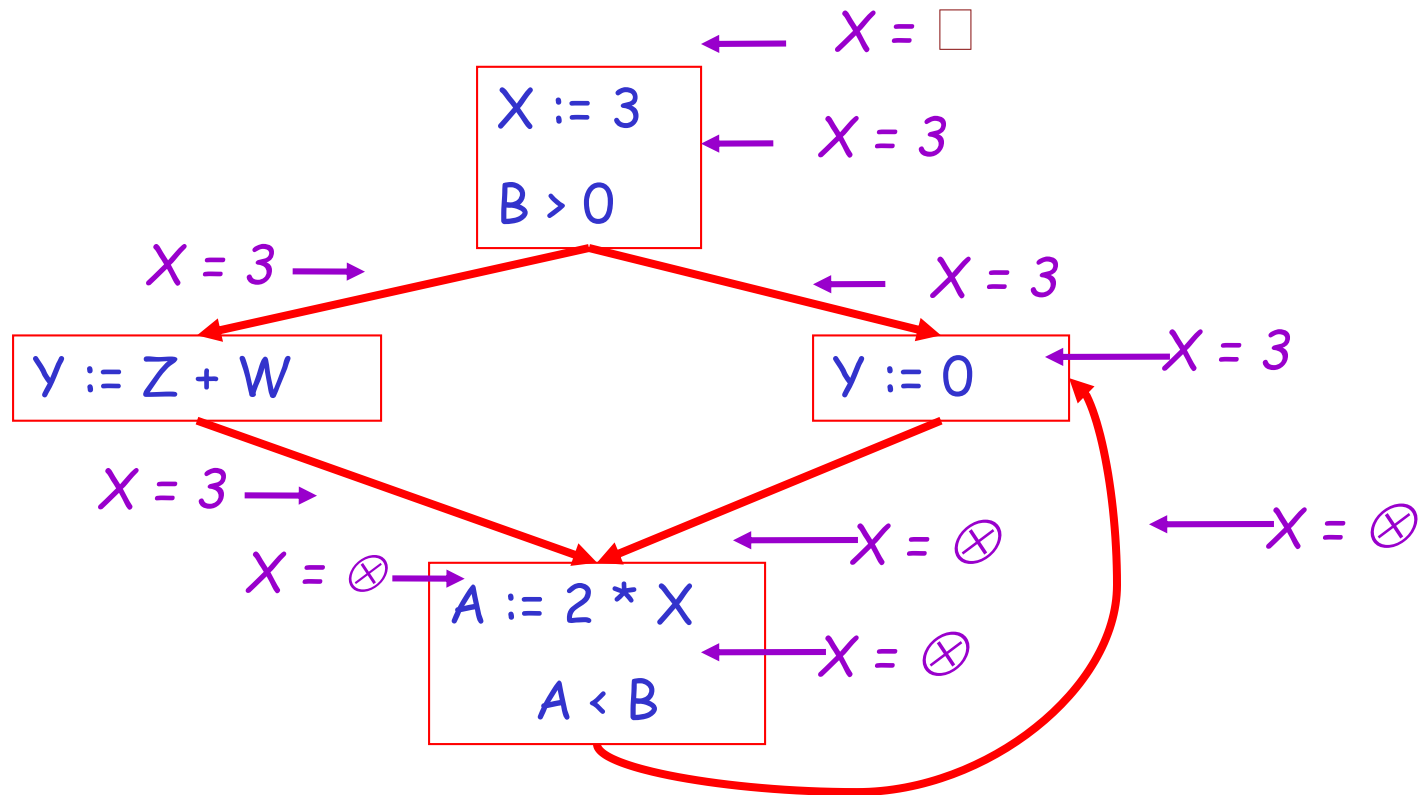
# Example

---



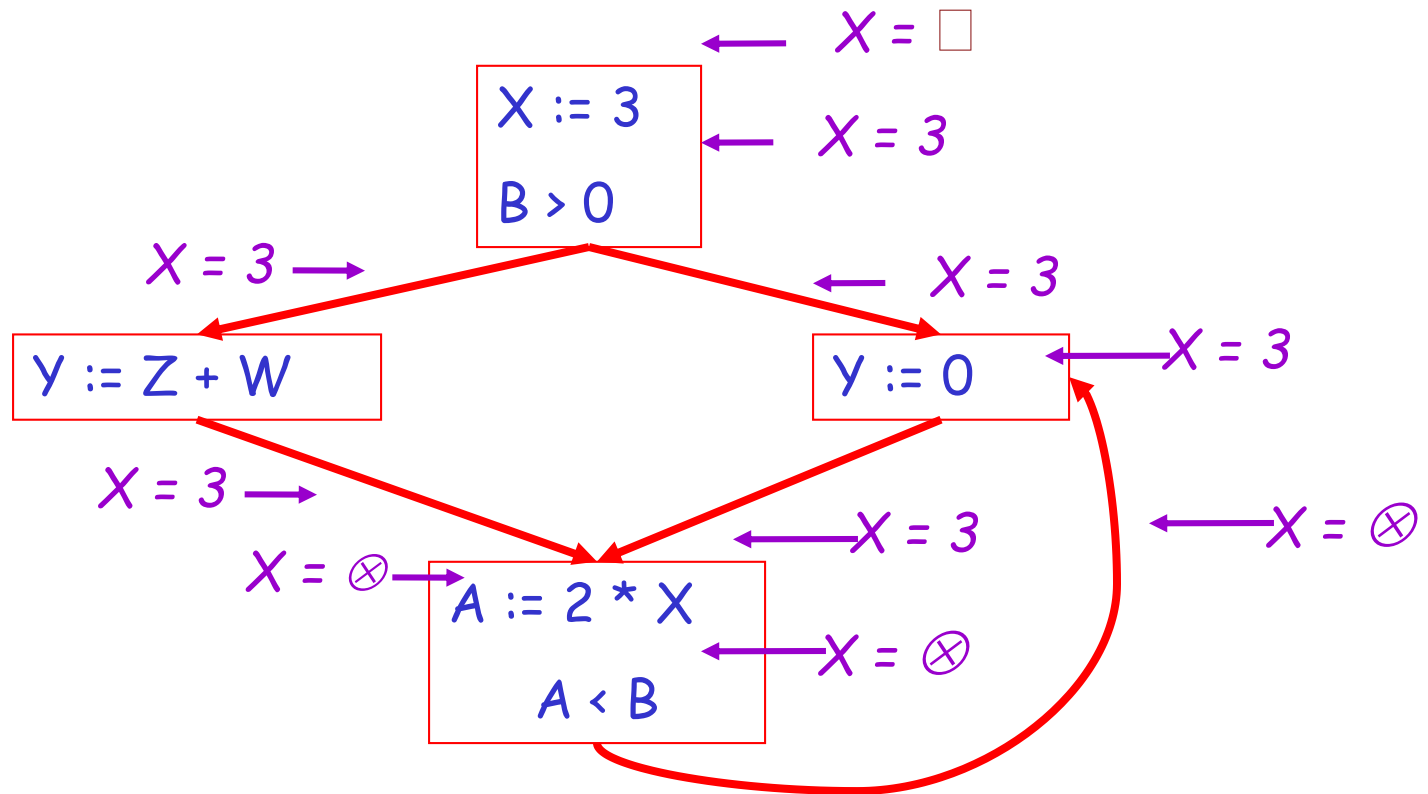
# Example

---



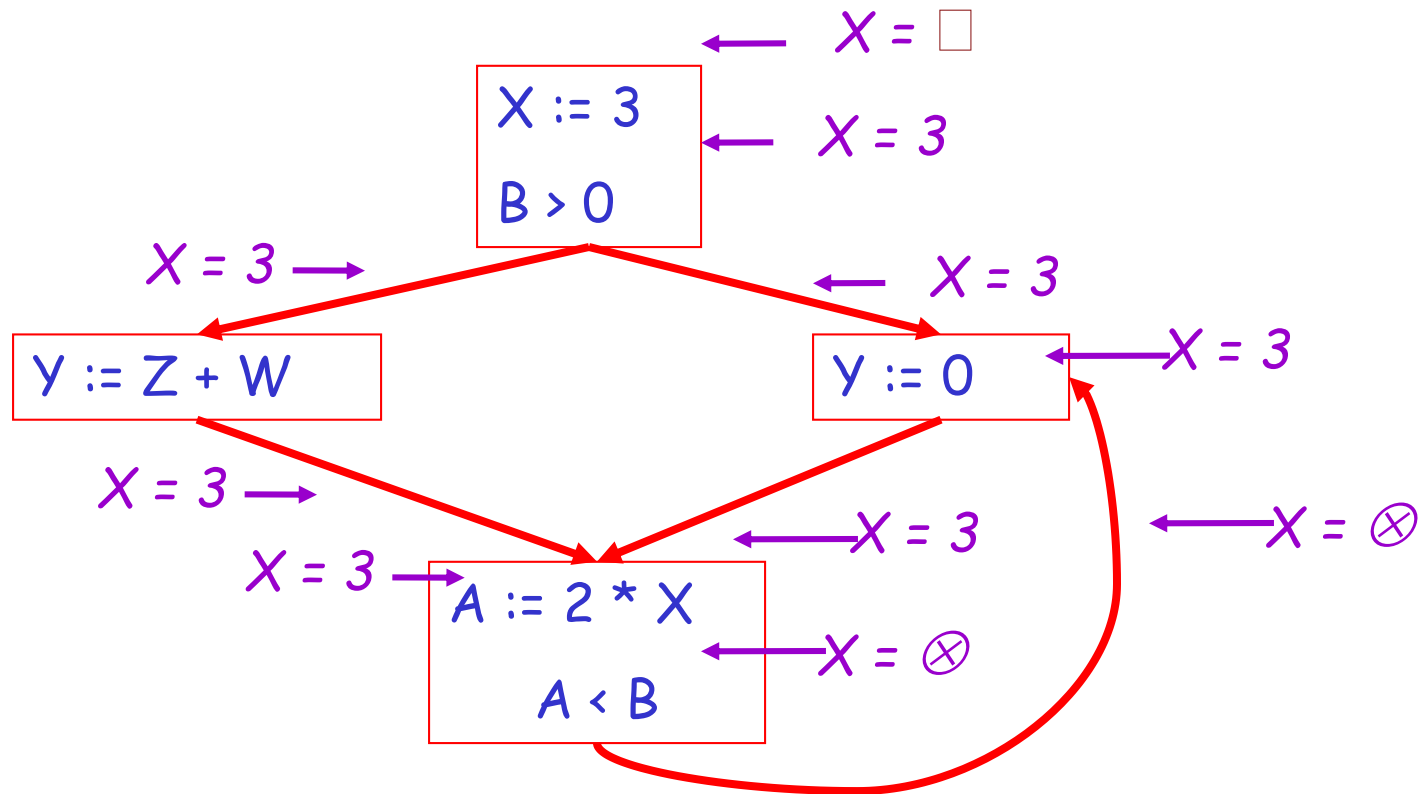
# Example

---



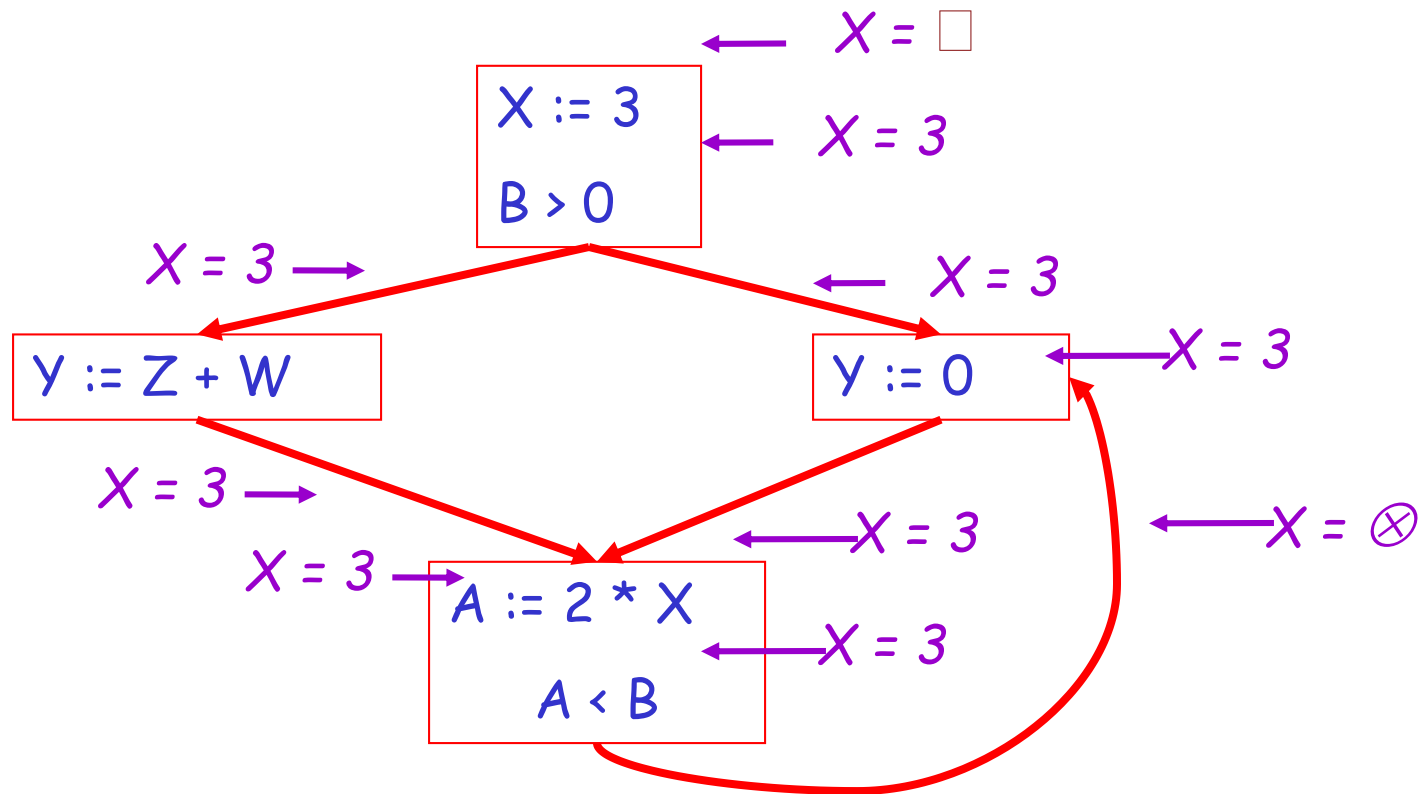
# Example

---



# Example

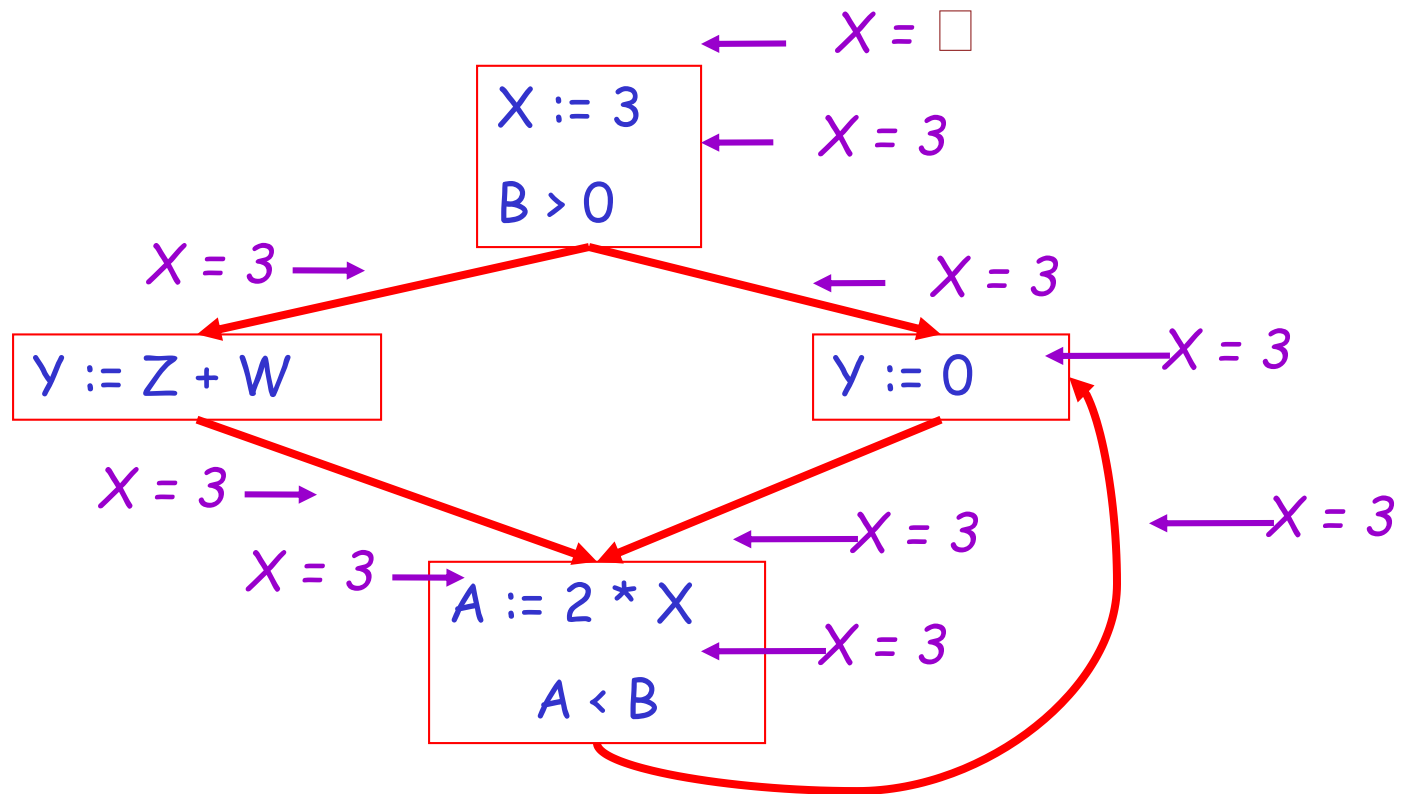
---





# Example

---



# Orderings

---

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

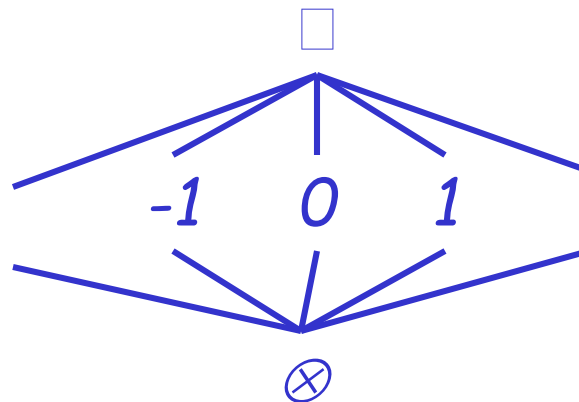
these are  
abstract values

$$\otimes < c < \square$$

e.g.,  $\square$  stands for any  
possible run time value

- Drawing a picture with “lower” values drawn lower, we get

constants are not  
comparable



e.g., 0 is not less  
than 1

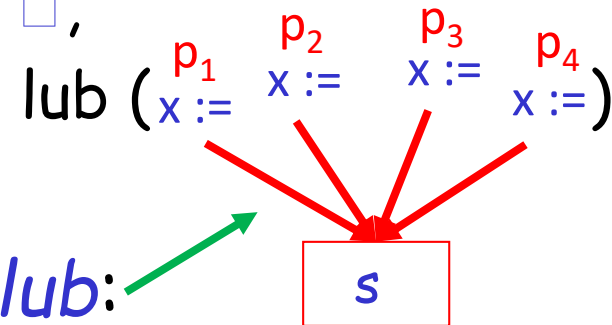
# Orderings (Cont.)

---

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

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

- Examples:  $\text{lub}(1, 2) = \square$ ,  $\text{lub}(\square, \otimes) = \square$ ,  
 $\text{lub}(1, \otimes) = 1, \dots$



- Rules 1-4 are in fact computing *lub*:  
 $C(s, x, in) = \text{lub} \{ C(p, x, out) \mid p \text{ is a predecessor of } s \}$

# 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  $\otimes$  and only *increase*
  - $\otimes$  can change to a constant, and a constant to  $\square$
  - Thus,  $C(s, x, \_)$  can change at most twice

# Termination (Cont.)

---

Thus the algorithm is linear in program size

Number of steps =

Number of  $C(\dots)$  value computed \* 2 =

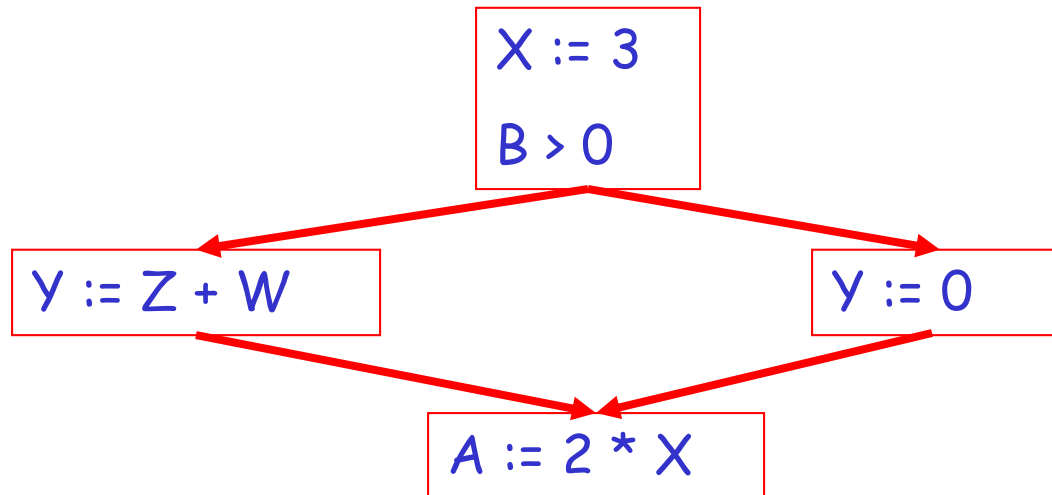
Number of program statements \* 4

for each statement  $s$ , we have  
one  $c(s, x, in)$  and one  $c(s, x, out)$ ,  
each can change at most twice

# Liveness Analysis

---

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

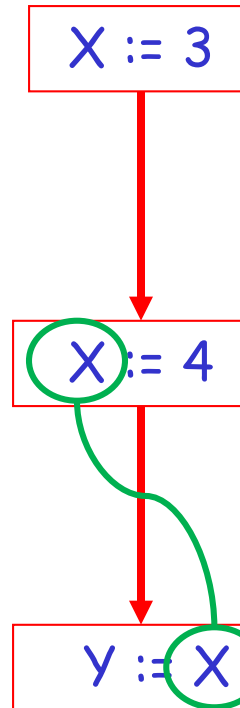


*After constant propagation,  $X := 3$  is dead (assuming  $X$  not used elsewhere)*

# Live and Dead

---

- The first value of  $x$  is *dead* (never used)
- The second value of  $x$  is *live* (may be used in the future)
- Liveness is an important concept



this value of  $x$   
is never used

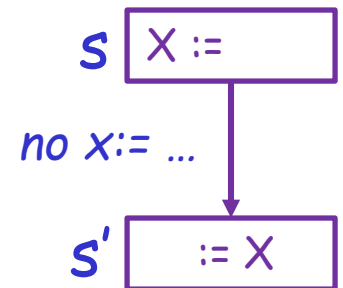
in this eg.,  $x$   
is guaranteed  
to be used

# 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  $x := \dots$  is dead code if  $x$  is dead after the assignment
- Dead statements can be deleted from the program
- But we need liveness information first . . .

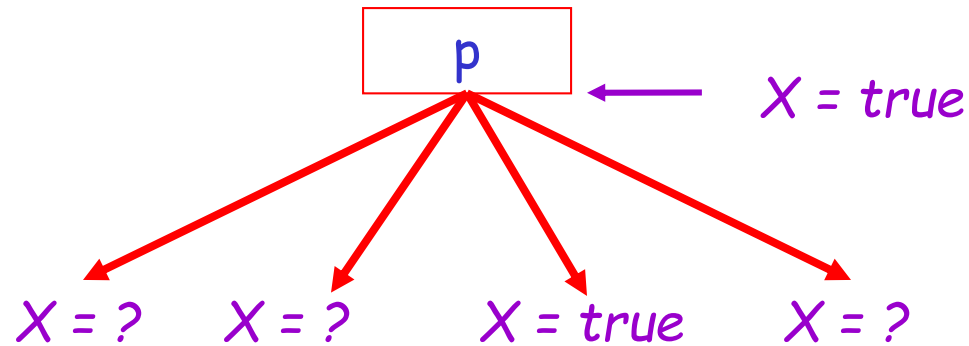
# Computing Liveness

---

- 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

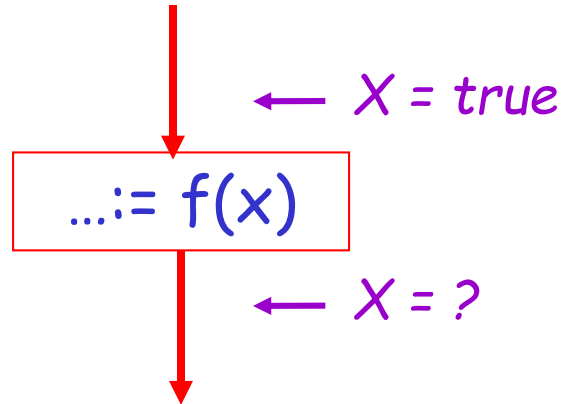
---



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

# Liveness Rule 2

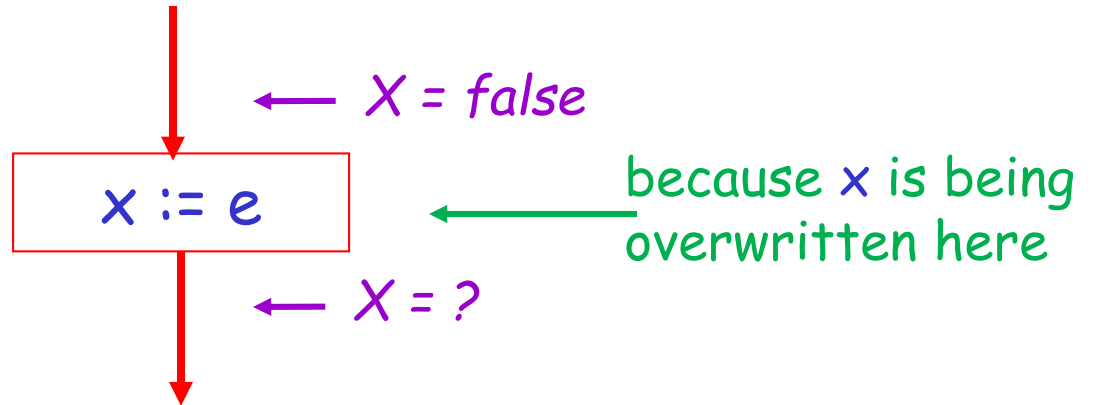
---



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

# Liveness Rule 3

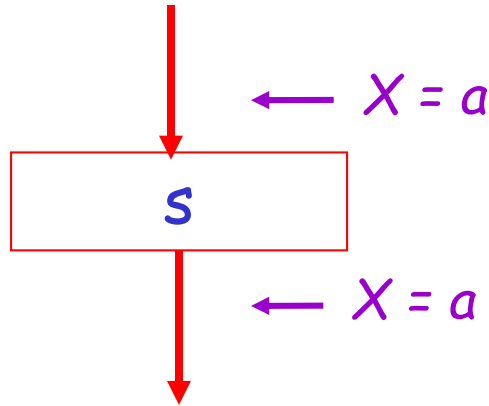
---



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

# Liveness Rule 4

---



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

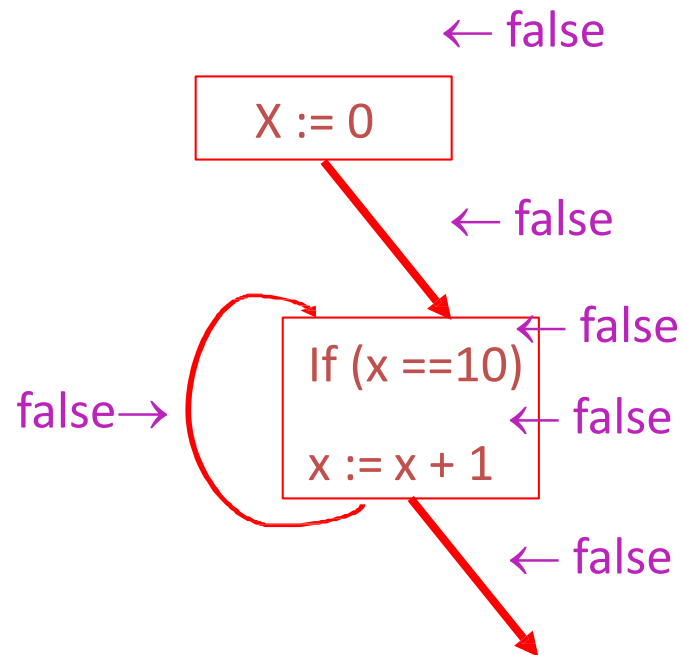
# Algorithm

---

1. Let all  $L(\dots) = \text{false}$  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

# Example

---

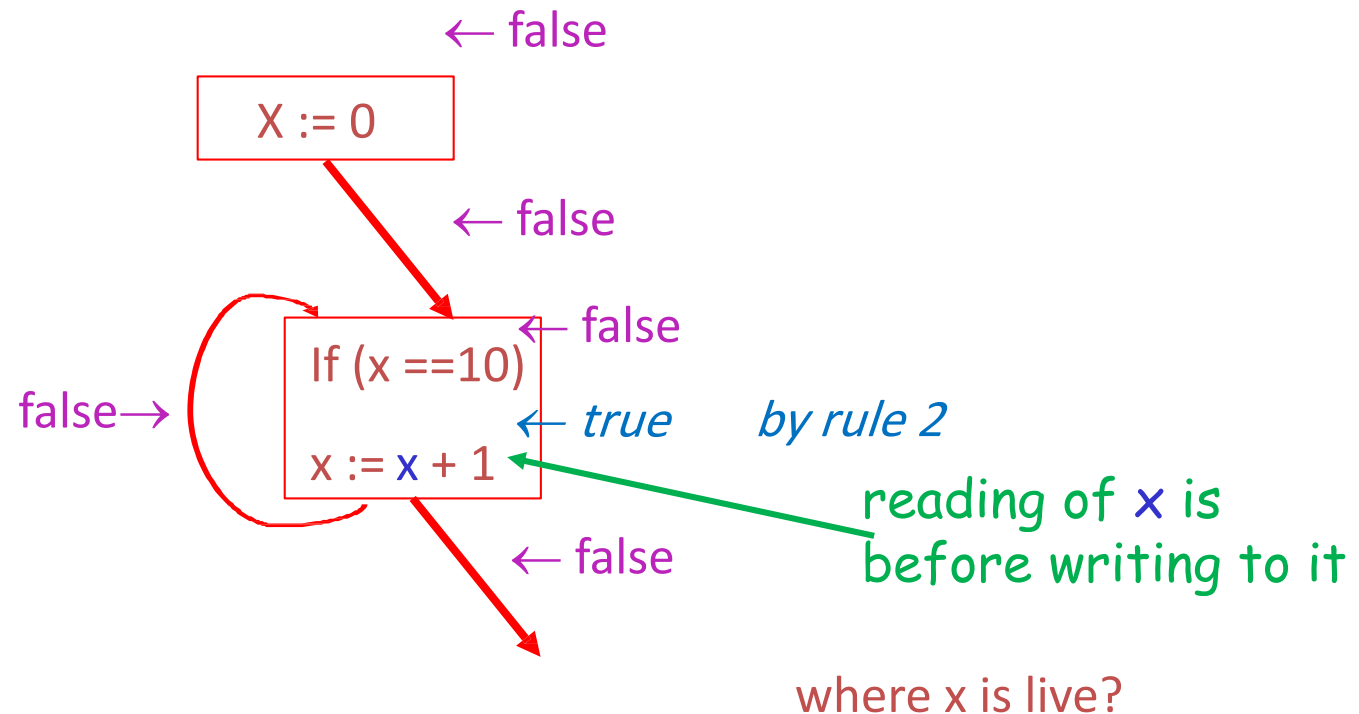


where x is live?



# Example

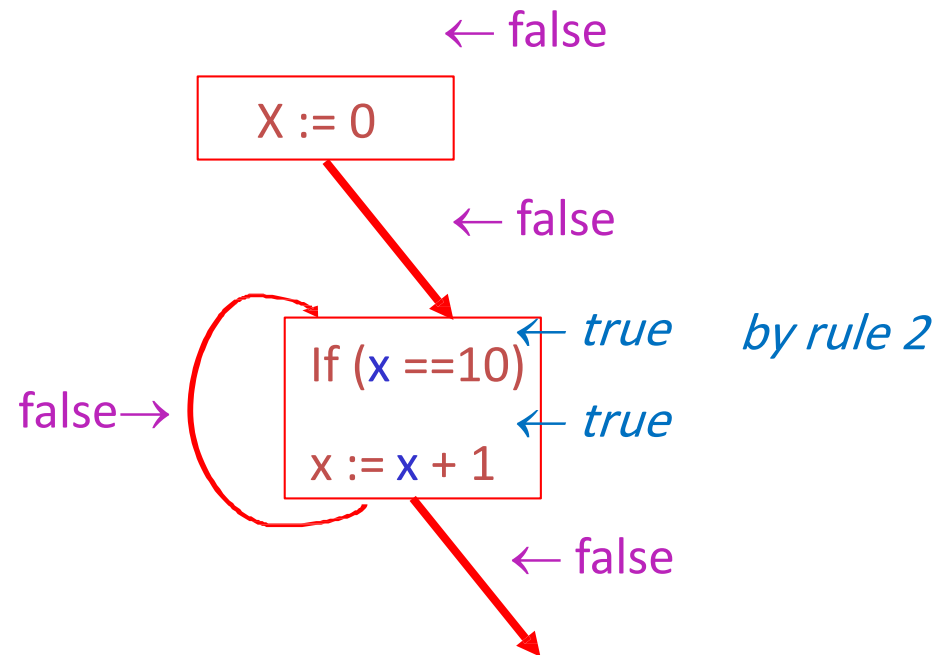
---



Rule 2:  $L(s, x, in) = true$  if  $s$  refers to  $x$  on the rhs

# Example

---

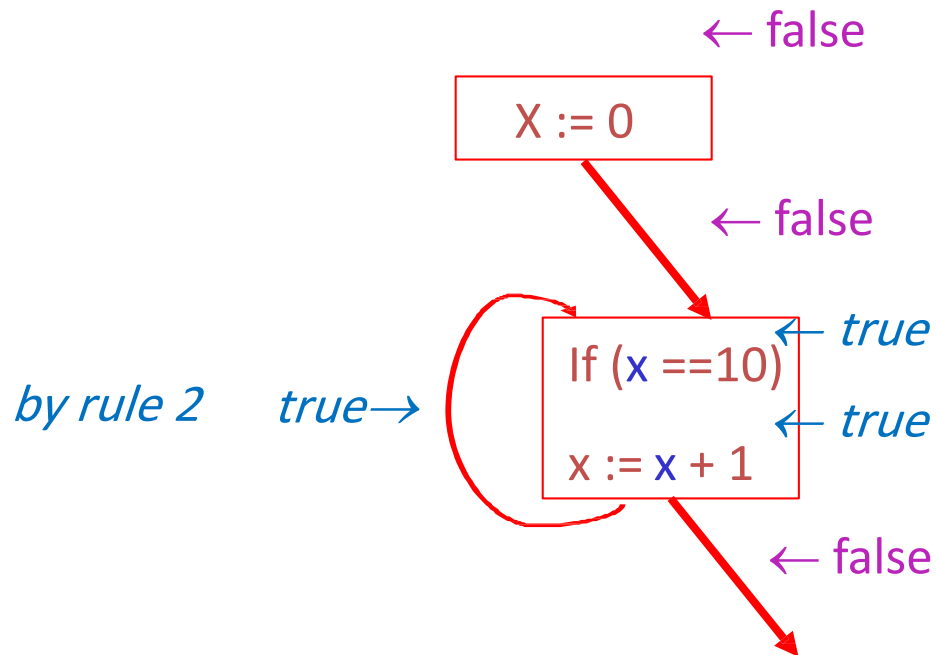


where x is live?

Rule 2:  $L(s, x, in) = true$  if  $s$  refers to  $x$  on the rhs

# Example

---

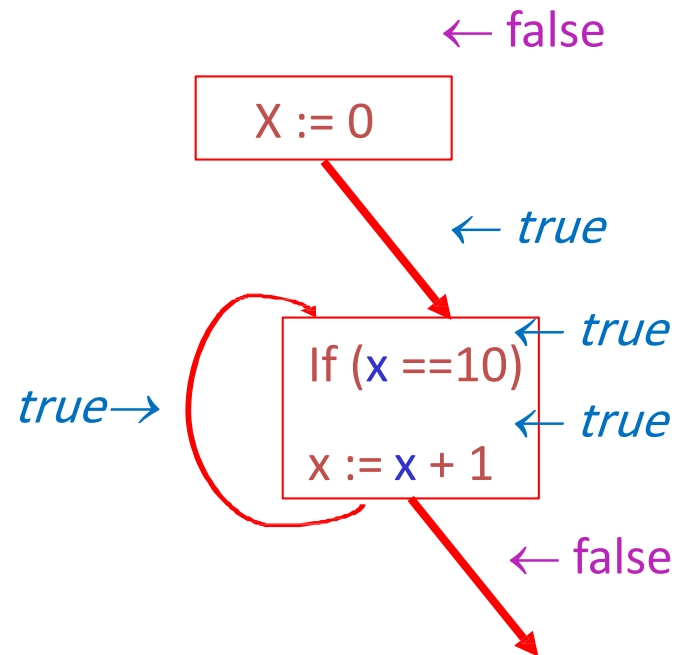


where x is live?

Rule 2:  $L(s, x, in) = true$  if  $s$  refers to  $x$  on the rhs

# Example

---

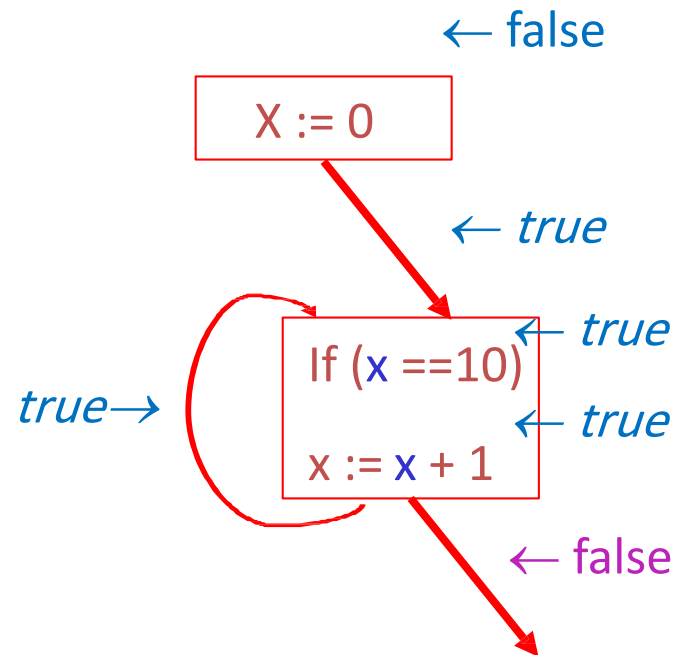


where `x` is live?

Rule 1:  $L(p, x, out) = \bigvee \{ L(s, x, in) \mid s \text{ a successor of } p \}$

# Example

---



where x is live?

Rule 3:  $L(x := e, x, in) = \text{false}$  if  $e$  does not refer to  $x$

# Termination

---

- A value can change from *false* to *true*, but not the other way around  
*false < true*
- 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

---

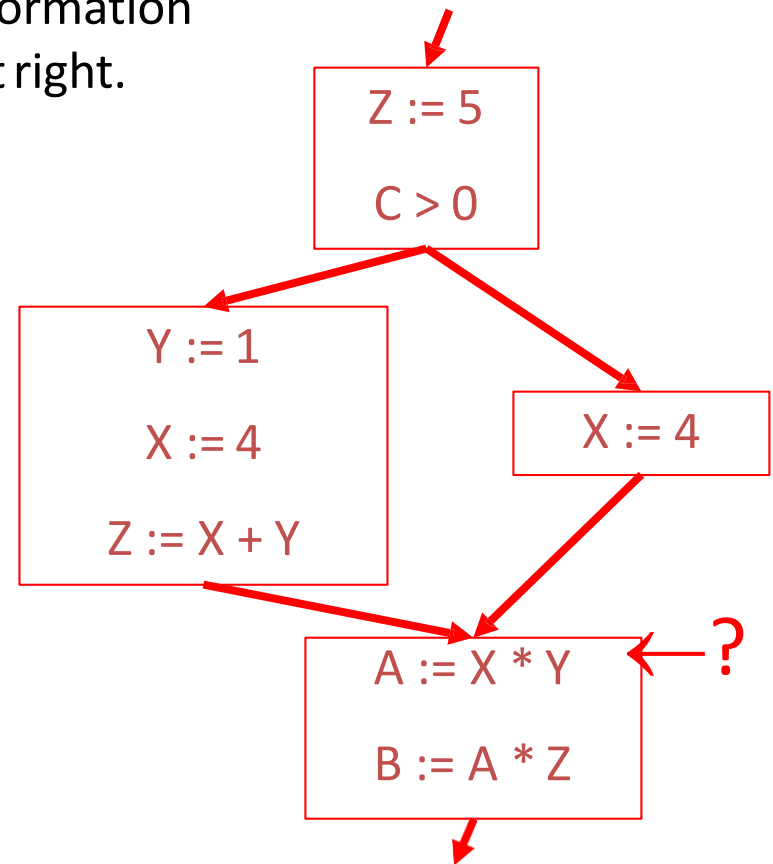
- There are many other global flow analyses
- Most can be classified as either forward or backward
- Most also follow the methodology of local rules relating information between adjacent program points



# Question?

After running the constant propagation algorithm to completion, choose the correct dataflow information for  $X$ ,  $Y$ , and  $Z$  at the program point labeled at right.

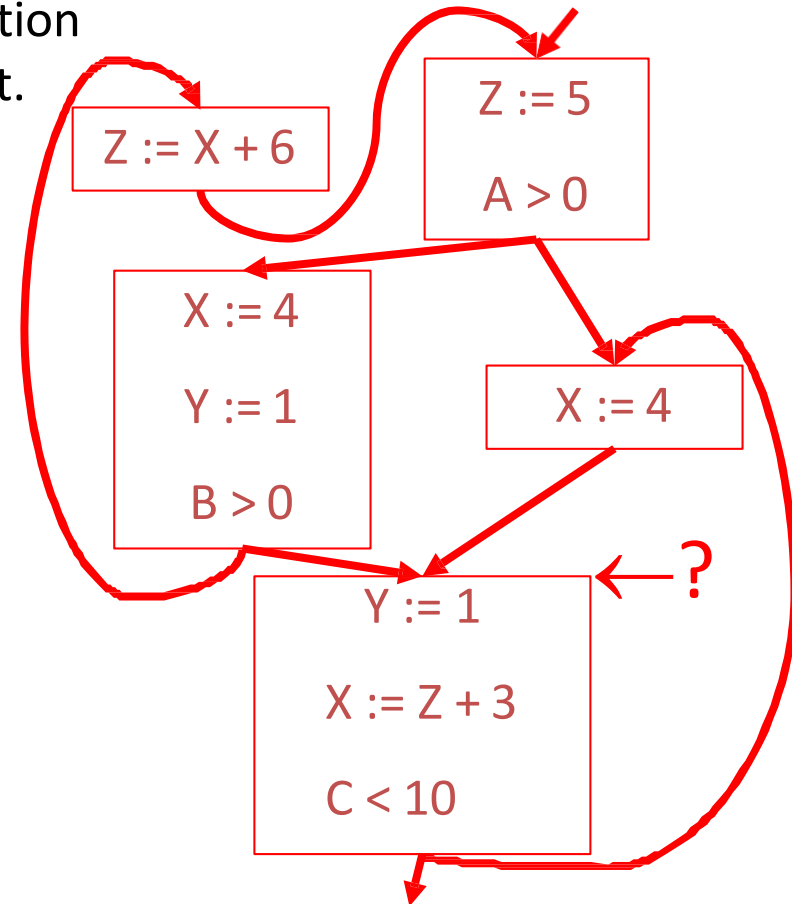
	$X$	$Y$	$Z$
<input type="radio"/>	4	<input type="checkbox"/>	<input type="checkbox"/>
<input type="radio"/>	4	<input type="checkbox"/>	5
<input type="radio"/>	4	1	5
<input type="radio"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>



# Question?

After running the constant propagation algorithm to completion, choose the correct dataflow information for  $X$ ,  $Y$ , and  $Z$  at the program point labeled at right.

	$X$	$Y$	$Z$
<input type="radio"/>	<input type="checkbox"/>	1	<input type="checkbox"/>
<input type="radio"/>	4	<input type="checkbox"/>	5
<input type="radio"/>	4	1	5
<input type="radio"/>	4	<input type="checkbox"/>	<input type="checkbox"/>



# Question?

After running the liveness analysis algorithm to completion, which of  $W$ ,  $X$ ,  $Y$ , and  $Z$  are live at the program point labeled at right? Assume all variables are dead on exit.

- $W$
- $X$
- $Y$
- $Z$

