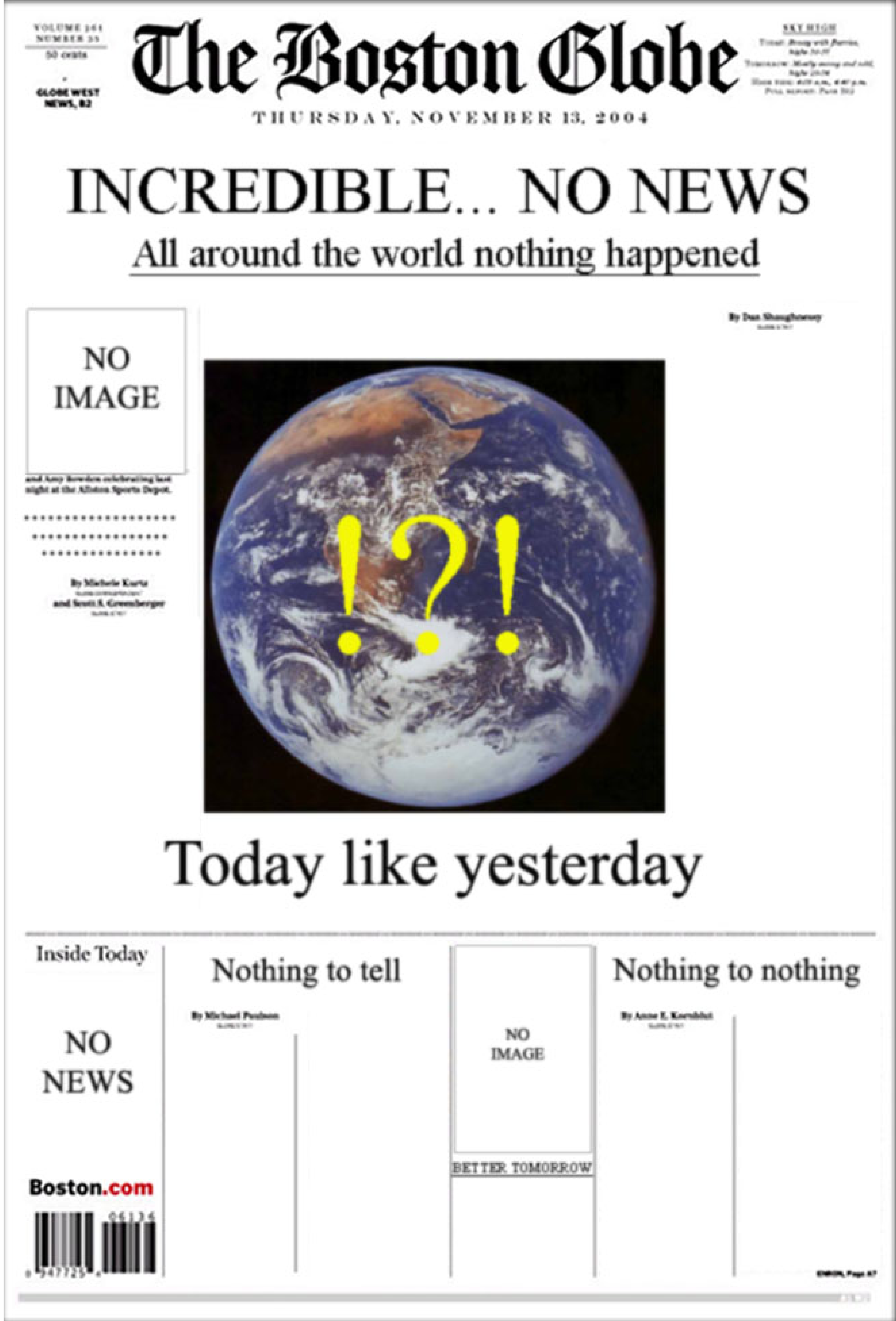
(How *Not* To Do)
Global Optimizations

One-Slide Summary

- A **global optimization** changes an entire method (consisting of multiple basic blocks).
- We must be **conservative** and only apply global optimizations when they preserve the original semantics.
- We use **global flow analyses** to determine if it is OK to apply an optimization.
- Flow analyses are built out of simple **transfer functions** and can work **forwards** or **backwards**.

Lecture Outline

- **Global** flow analysis
- **Global** constant propagation
- Liveness analysis



The Boston Globe

THURSDAY, NOVEMBER 13, 2004

INCREDIBLE... NO NEWS

All around the world nothing happened

NO IMAGE

Today like yesterday

Inside Today

NO NEWS

Nothing to tell

NO IMAGE

BETTER TOMORROW

Nothing to nothing

Boston.com

Local Optimization

Recall the simple basic-block optimizations

- Constant propagation
- Dead code elimination

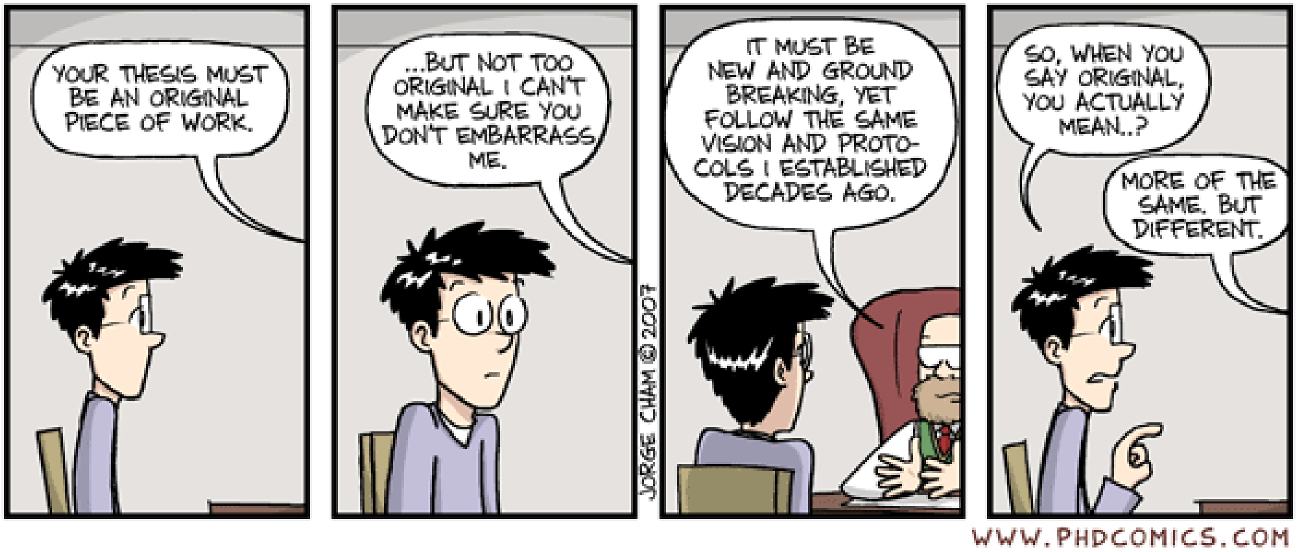
```
X := 3
Y := Z * W
Q := X + Y
```

→

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

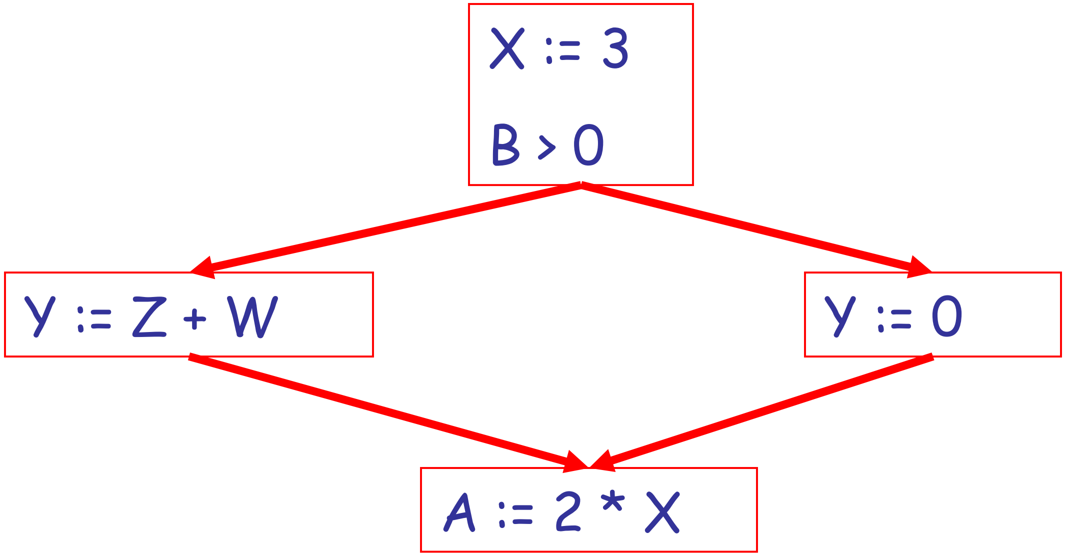
→

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



Global Optimization

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



X := 3

B > 0

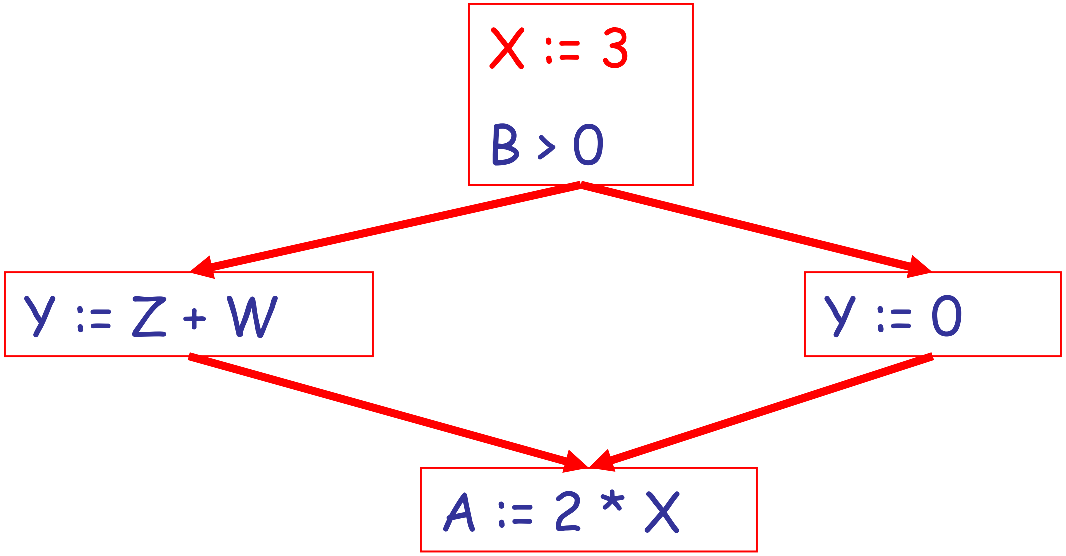
Y := Z + W

Y := 0

A := 2 * X

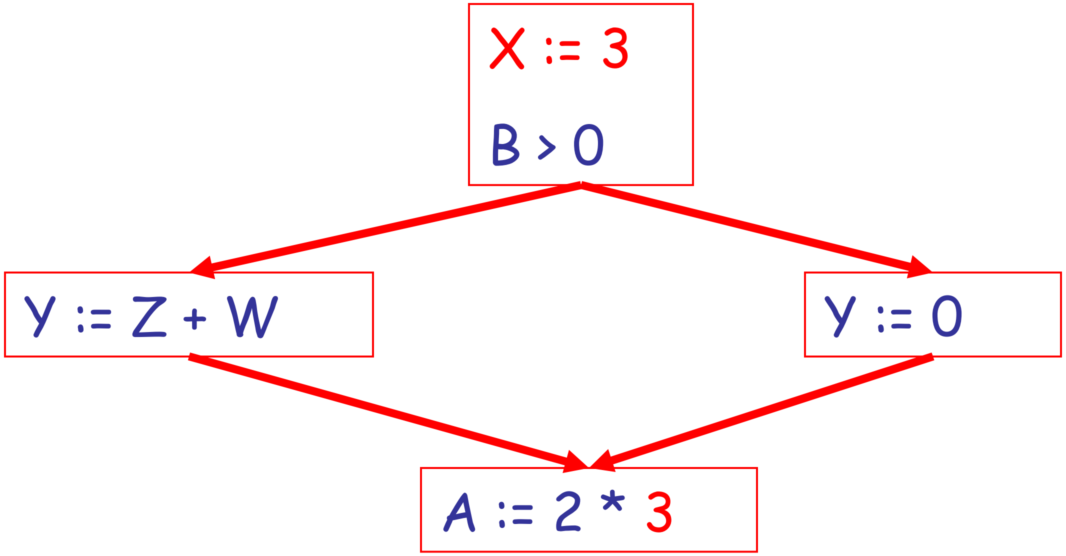
Global Optimization

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



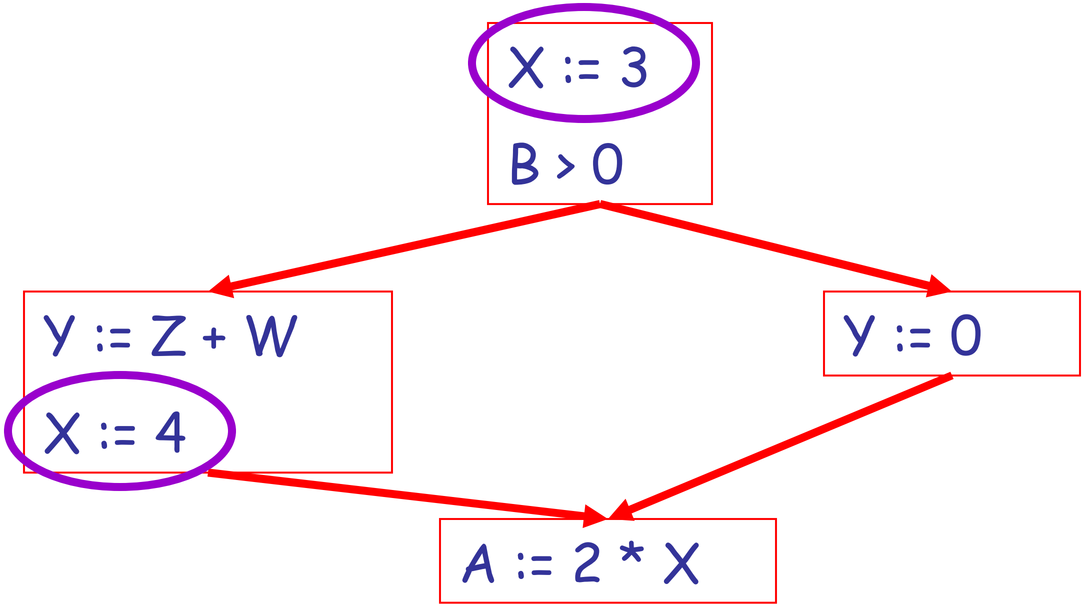
Global Optimization

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



Correctness

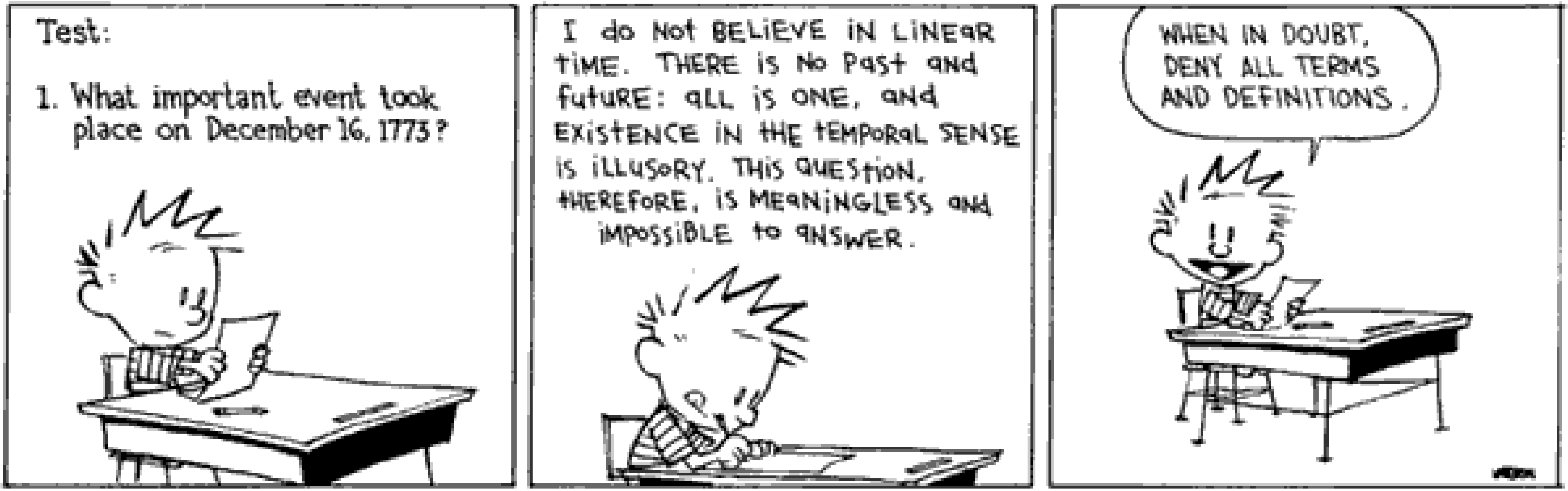
- How do we know it is OK to globally propagate constants?
- There are situations where it is incorrect:



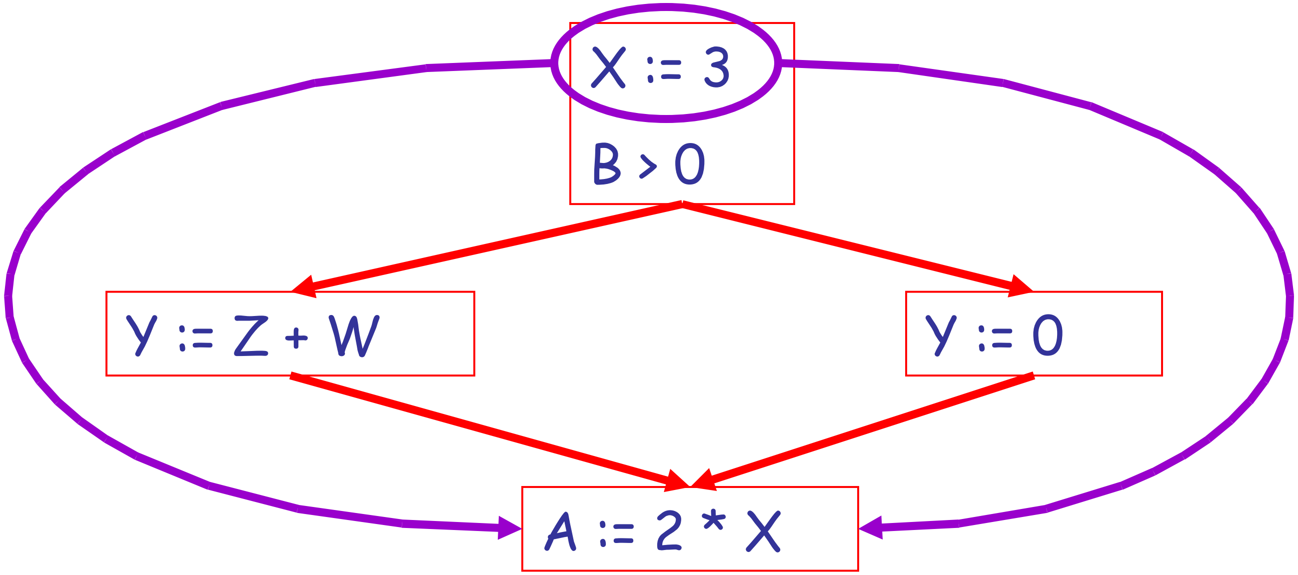
Correctness (Cont.)

To replace a use of x by a constant k we must know this correctness condition:

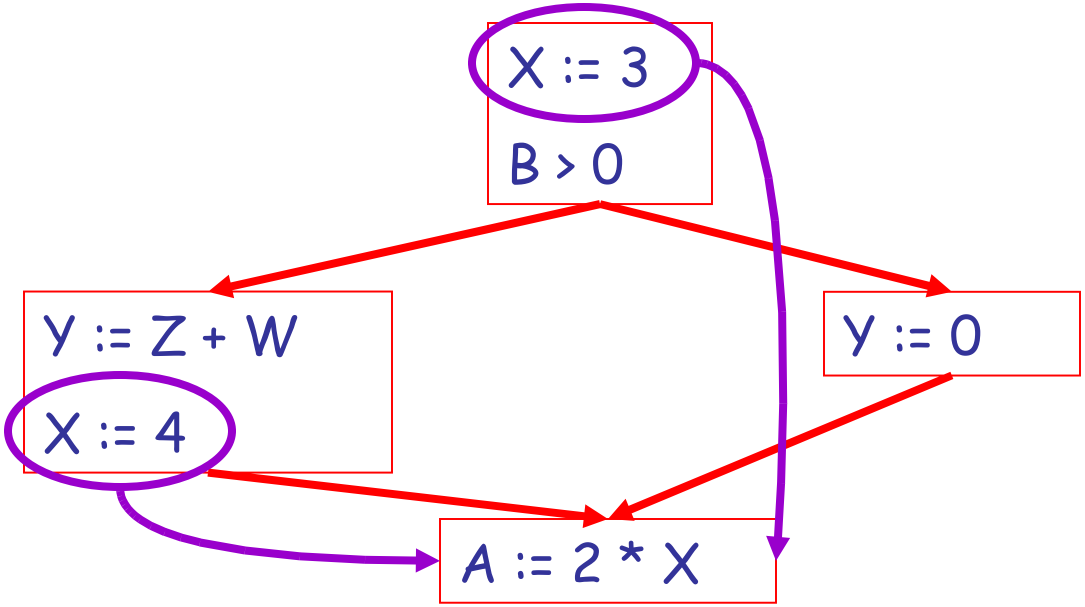
On every path to the use of x, the last assignment to x is x := k ******



Example 1 Revisited



Example 2 Revisited



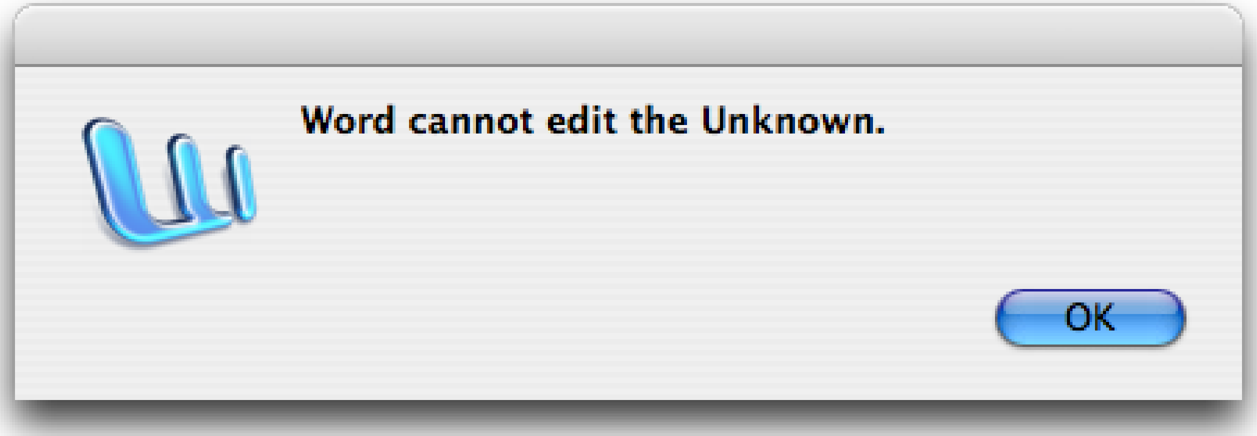
Discussion

- 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**
  - Global = an analysis of the entire control-flow graph for *one* method body

Global Analysis

Global optimization tasks share several traits:

- The optimization depends on knowing a property P at a particular point in program execution
- Proving P at any point requires knowledge of the entire method body
- **Property P is typically *undecidable*!**



Word cannot edit the Unknown.

Undecidability of Program Properties

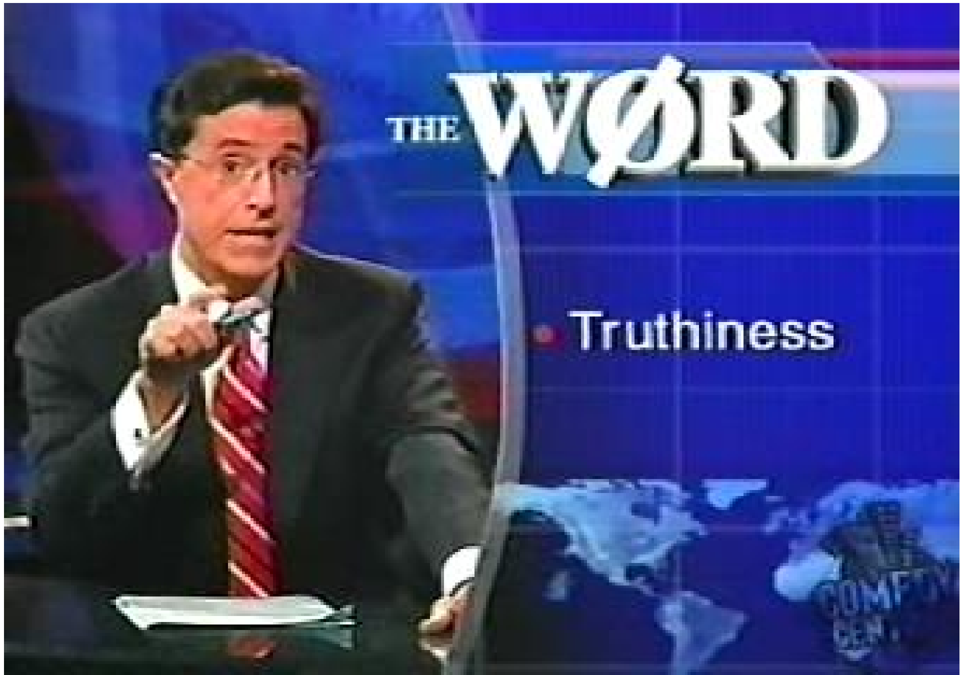
- **Rice's Theorem**: Most interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Take function H and find out if it halts by testing function `F(x) { H(x); return 1; }` whether it has positive result
    - *Contradition!*
- Syntactic properties are decidable!
  - e.g., How many occurrences of "x" are there?
- Programs without looping are also decidable!

Conservative Program Analyses

- So, we cannot tell for sure that “x” is always 3
  - Then, how can we apply constant propagation?
- It is OK to be **conservative**.

Conservative Program Analyses

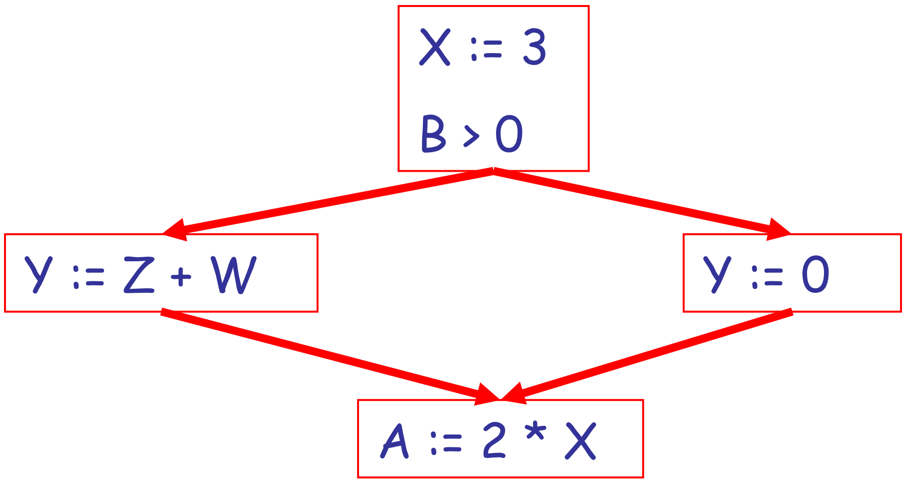
- So, we cannot tell for sure that “x” is always 3
  - Then, how can we apply constant propagation?
- It is OK to be **conservative**. If the optimization requires P to be true, then want to know either
  - P is definitely true
  - Don’t know if P is true
- Let's call this *truthiness*



Conservative Program Analyses

- So, we cannot tell for sure that “x” is always 3
  - Then, how can we apply constant propagation?
- It is OK to be **conservative**. If the optimization requires P to be true, then want to know either
  - P is definitely true
  - Don’t know if P is true
- It is always correct to say “don’t know”
  - We try to say don’t know as rarely as possible
- All program analyses are conservative

Global Optimization: Review



X := 3

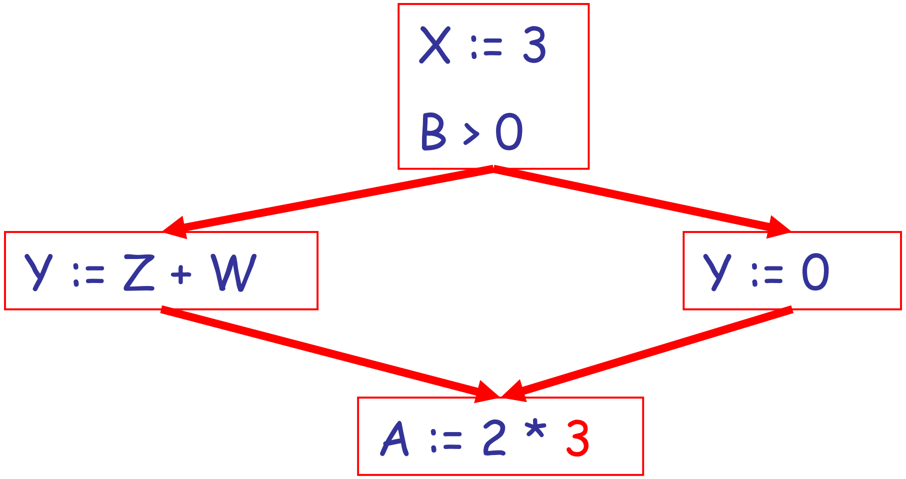
B > 0

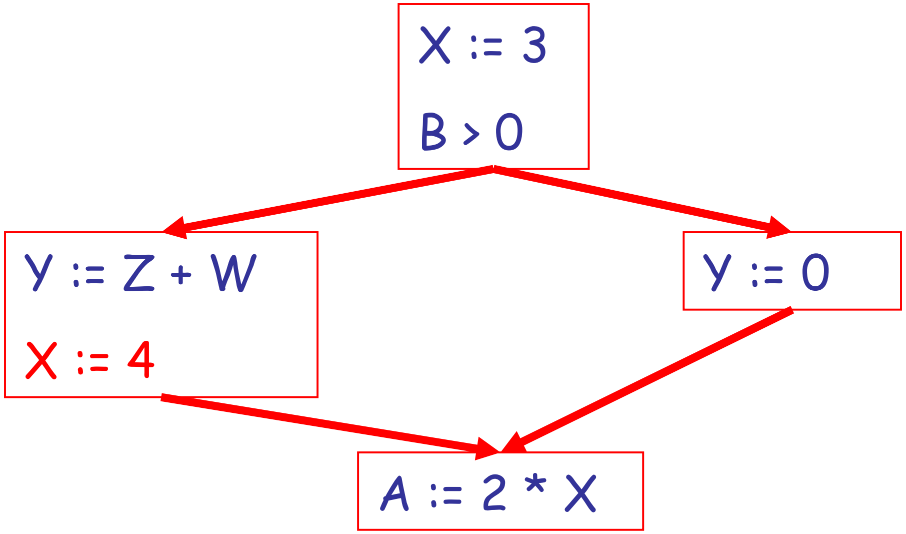
Y := Z + W

Y := 0

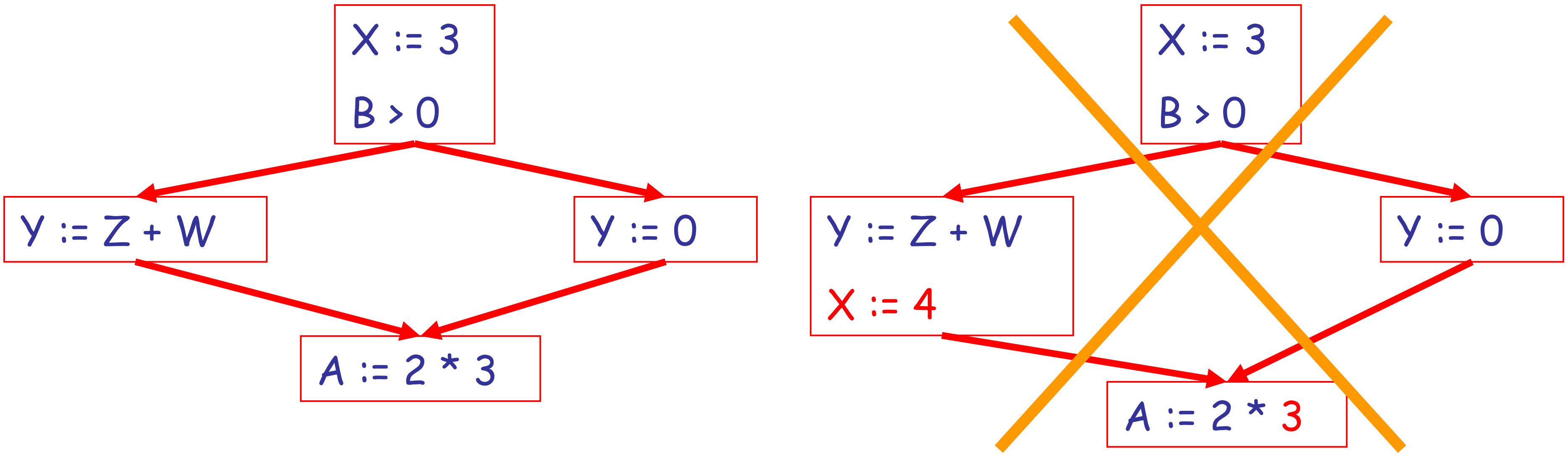
A := 2 * X

Global Optimization: Review





Global Optimization: Review

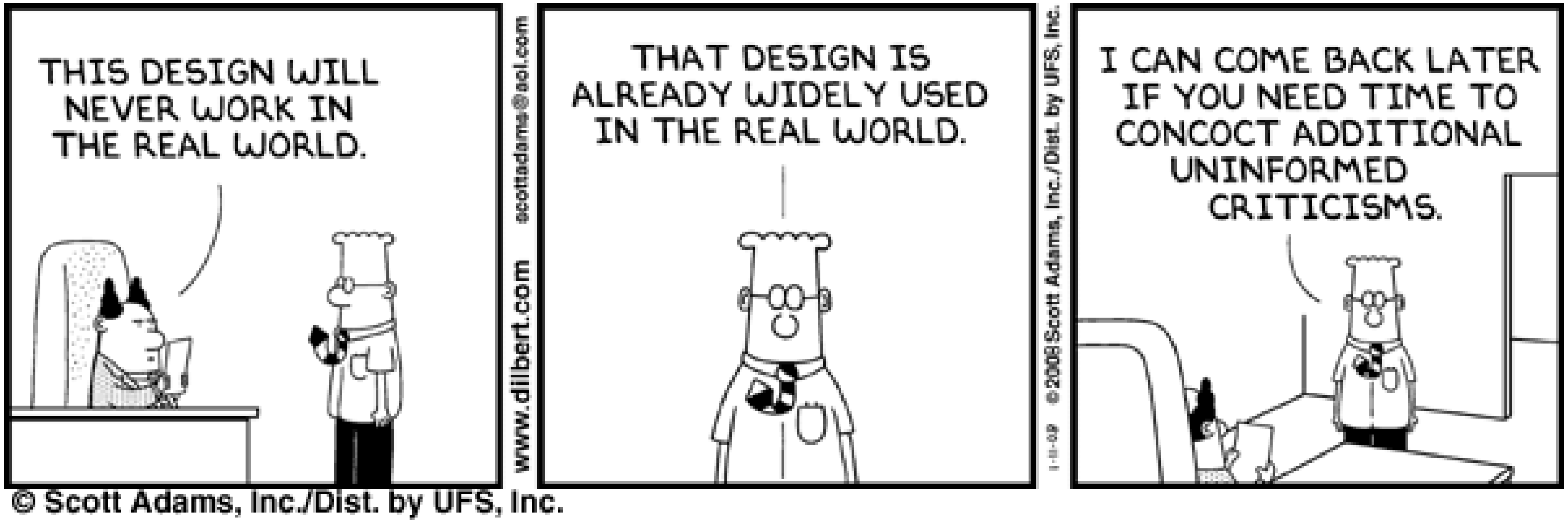


- To replace a use of x by a constant k we must know that:

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

Review

- The correctness condition is hard to check
- Checking it requires *global* analysis
  - An analysis of the entire control-flow graph for one method body
- We said that was impossible, right?



© Scott Adams, Inc./Dist. by UFS, Inc.

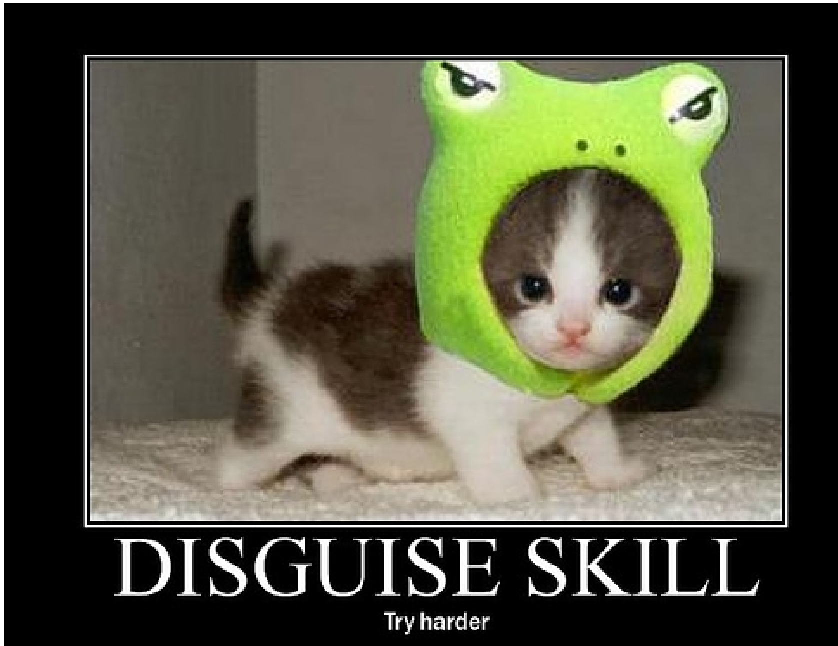
Global Analysis

- **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
- Valid points cannot hide!
- We will find you!
  - *(sometimes)*



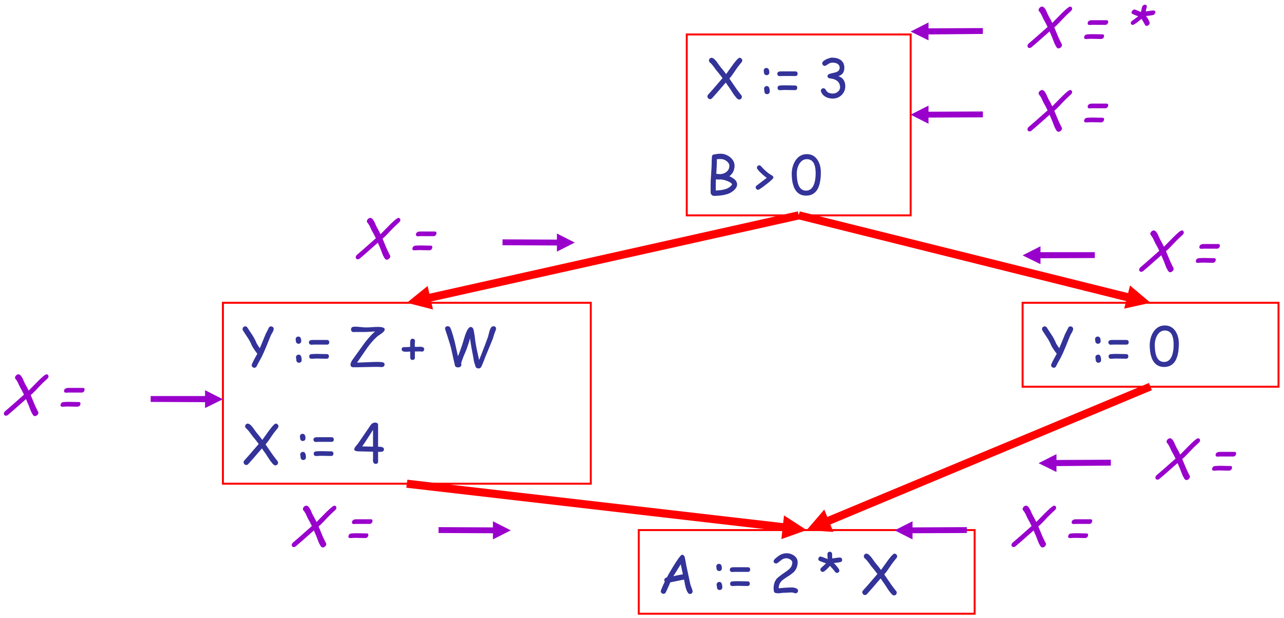
Global Constant Propagation (Cont.)

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

value	*interpretation*
#	This statement is not reachable
c	X = constant c
*	Don't know if X is a constant

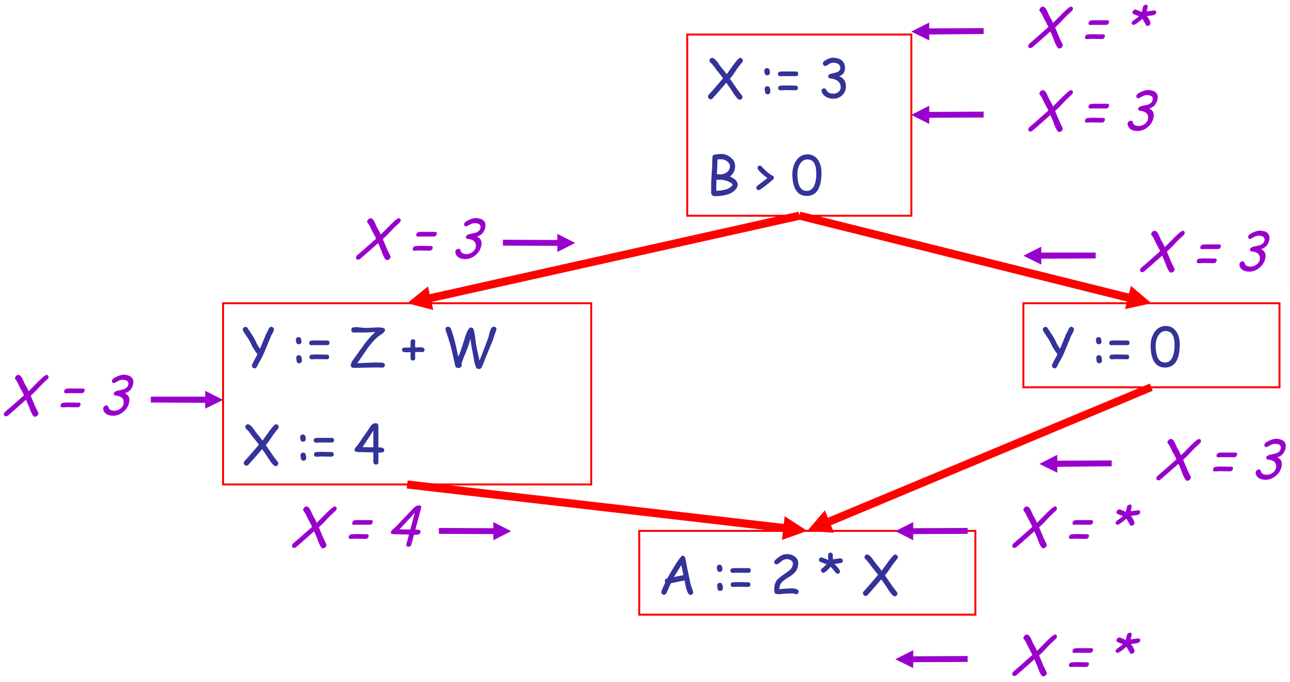
Example

Let's do it on the board!



Recall: # = not reachable, c = constant, * = don't know.

Example Answers



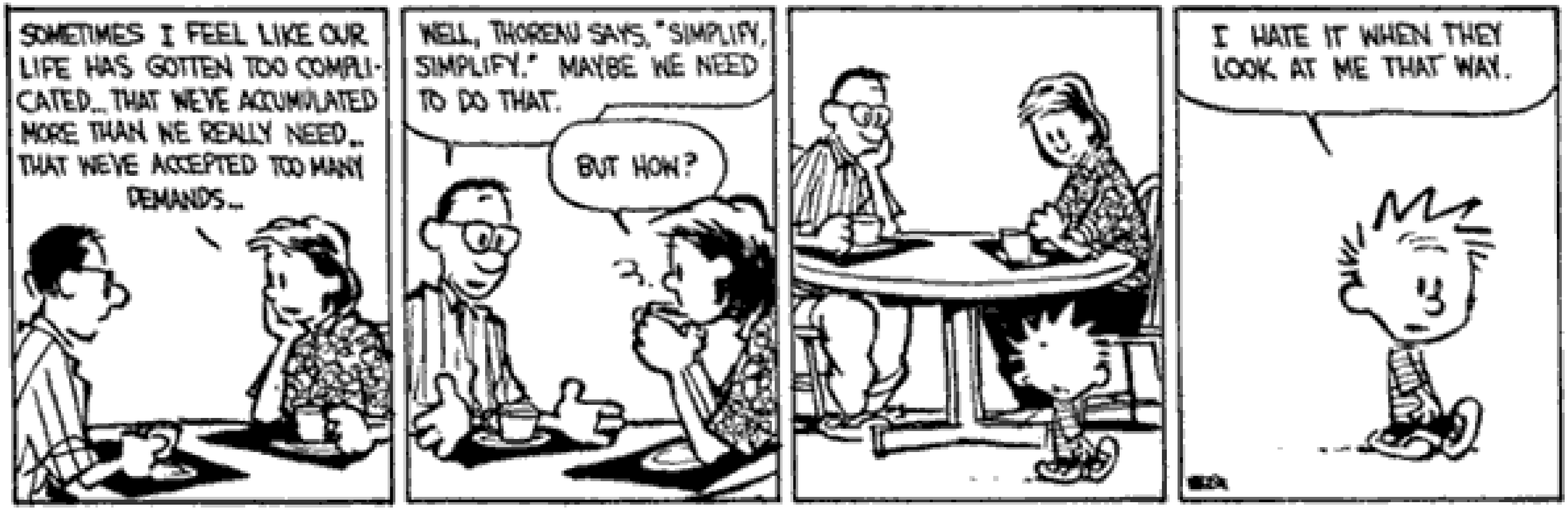
Using the Information

- 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

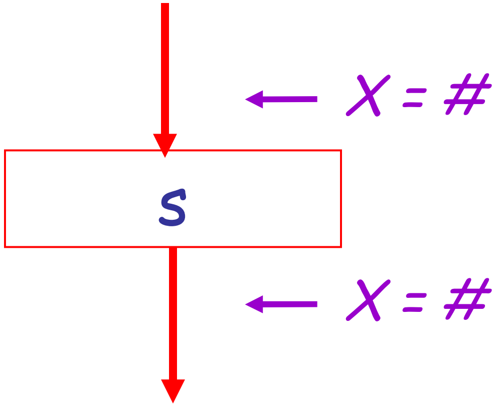
$$C_{in}(x,s) = \text{value of } x \text{ before } s$$

$$C_{out}(x,s) = \text{value of } x \text{ after } s$$

Transfer Functions

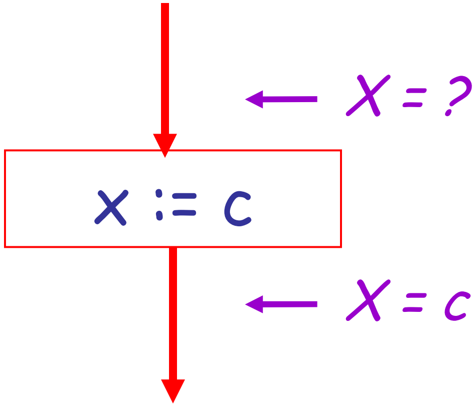
- Define a **transfer function** that transfers information from one statement to another

Rule 1



$$C_{out}(x, s) = \# \text{ if } C_{in}(x, s) = \#$$

Rule 2



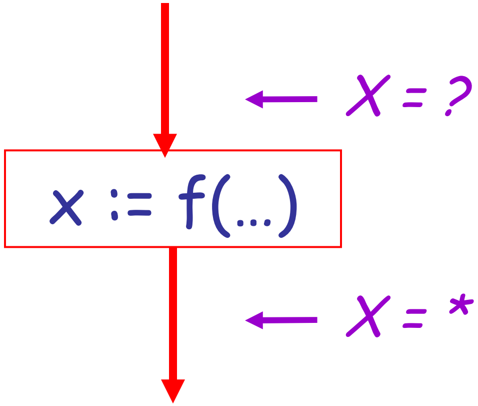
X = ?

x := c

X = c

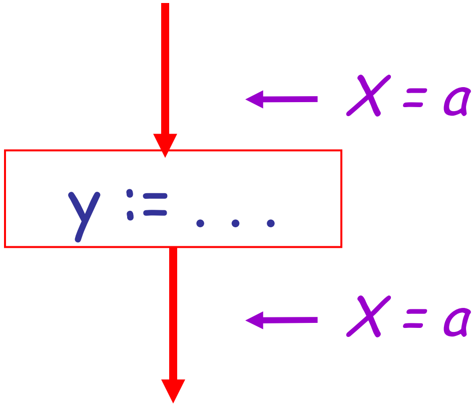
$C_{out}(x, x := c) = c$ if c is a constant

Rule 3



$$C_{out}(x, x := f(\ldots)) = *$$

Rule 4



$$C_{out}(x,\ y := \ldots) = C_{in}(x,\ y := \ldots) \quad \text{if } x \neq y$$

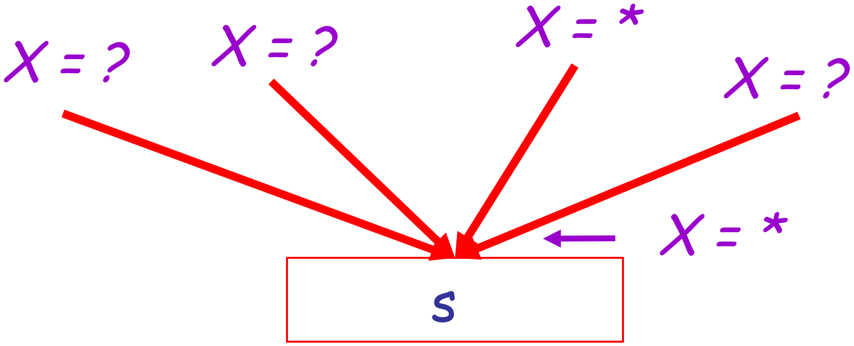
The Other Half

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information across statements

- Now we need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information forward across CFG edges

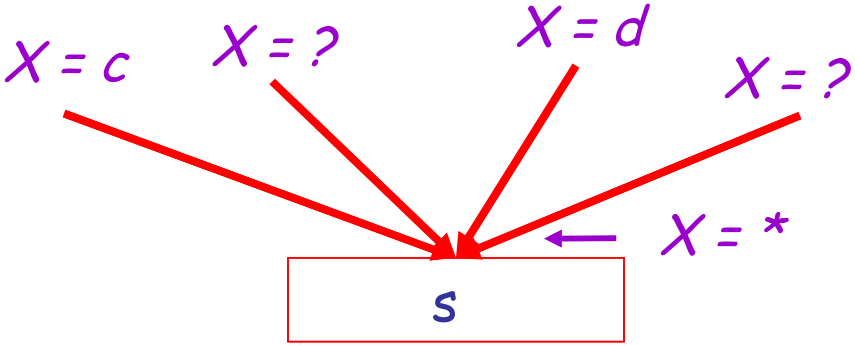
- In the following rules, let statement s have immediate predecessor statements $p_1,\ldots,p_n$

Rule 5



if $C_{out}(x, p_i) = *$ for some i, then $C_{in}(x, s) = *$

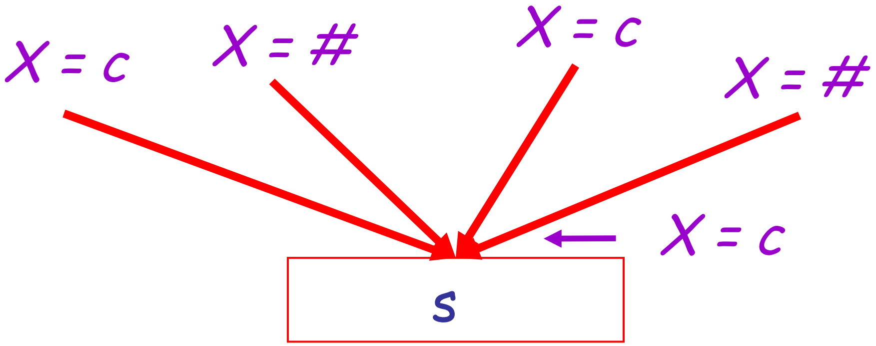
Rule 6



if $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$

then $C_{in}(x, s) = *$

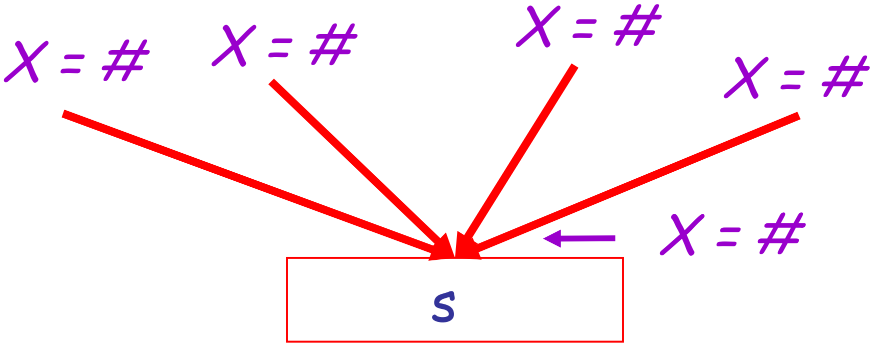
Rule 7



if $C_{out}(x, p_i) = c$ or # for all i,

then $C_{in}(x, s) = c$

Rule 8



if $C_{out}(x, p_i) = \#$ for all i,

then $C_{in}(x, s) = \#$

An Algorithm

- For every entry s to the program, set $C_{in}(x, s) = *$

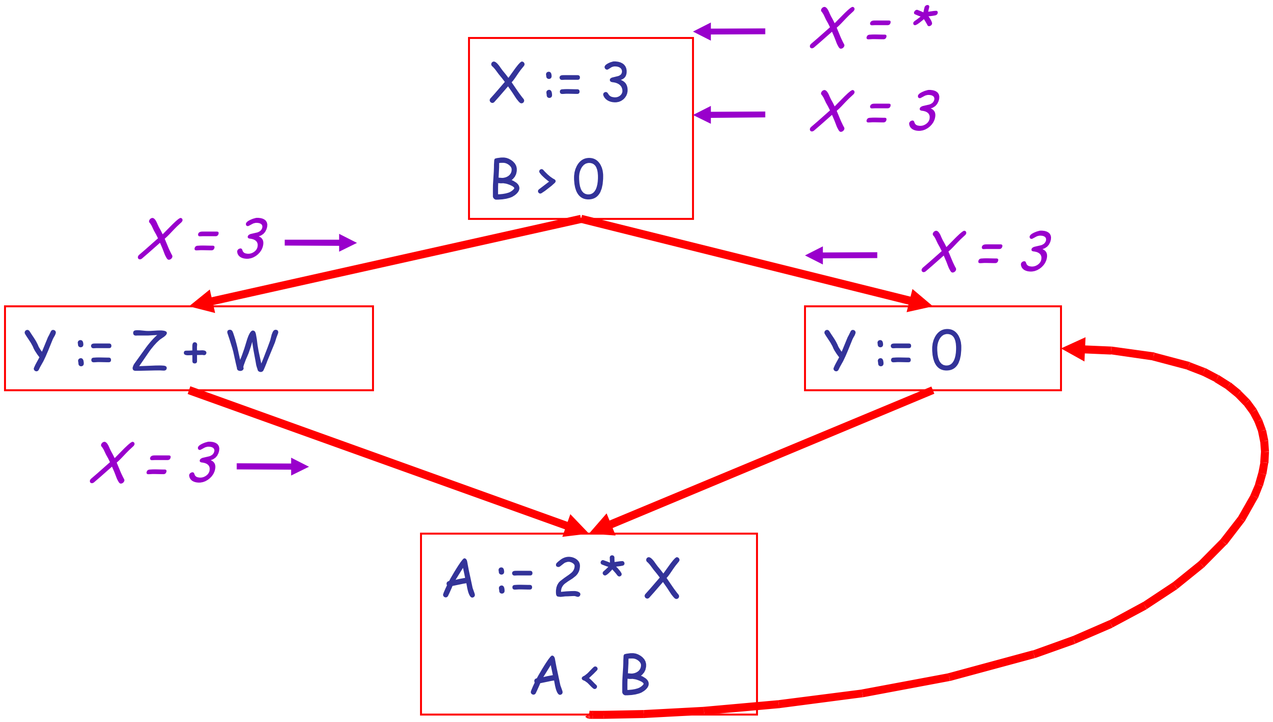
- Set $C_{in}(x, s) = C_{out}(x, s) = \#$ everywhere else

- Repeat until all points satisfy 1-8:

  Pick s not satisfying 1-8 and update using the appropriate rule

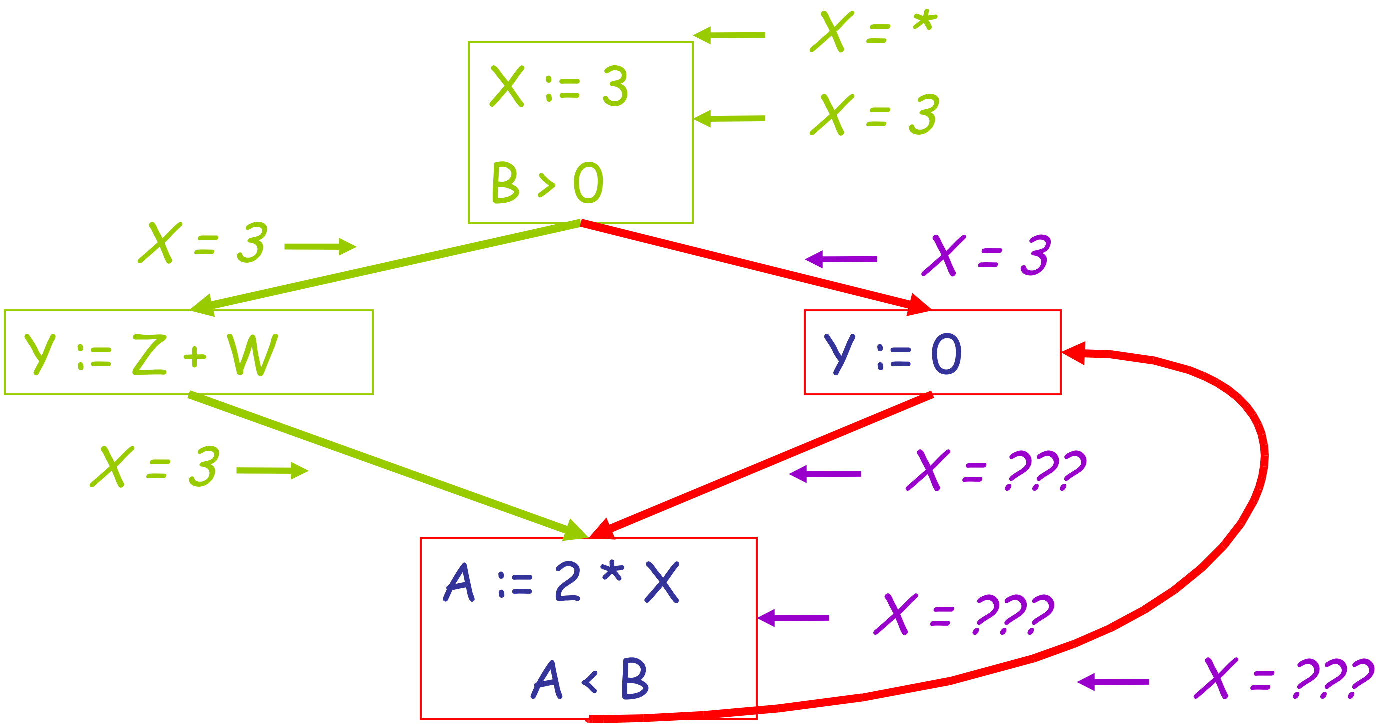
The Value

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



The Value

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



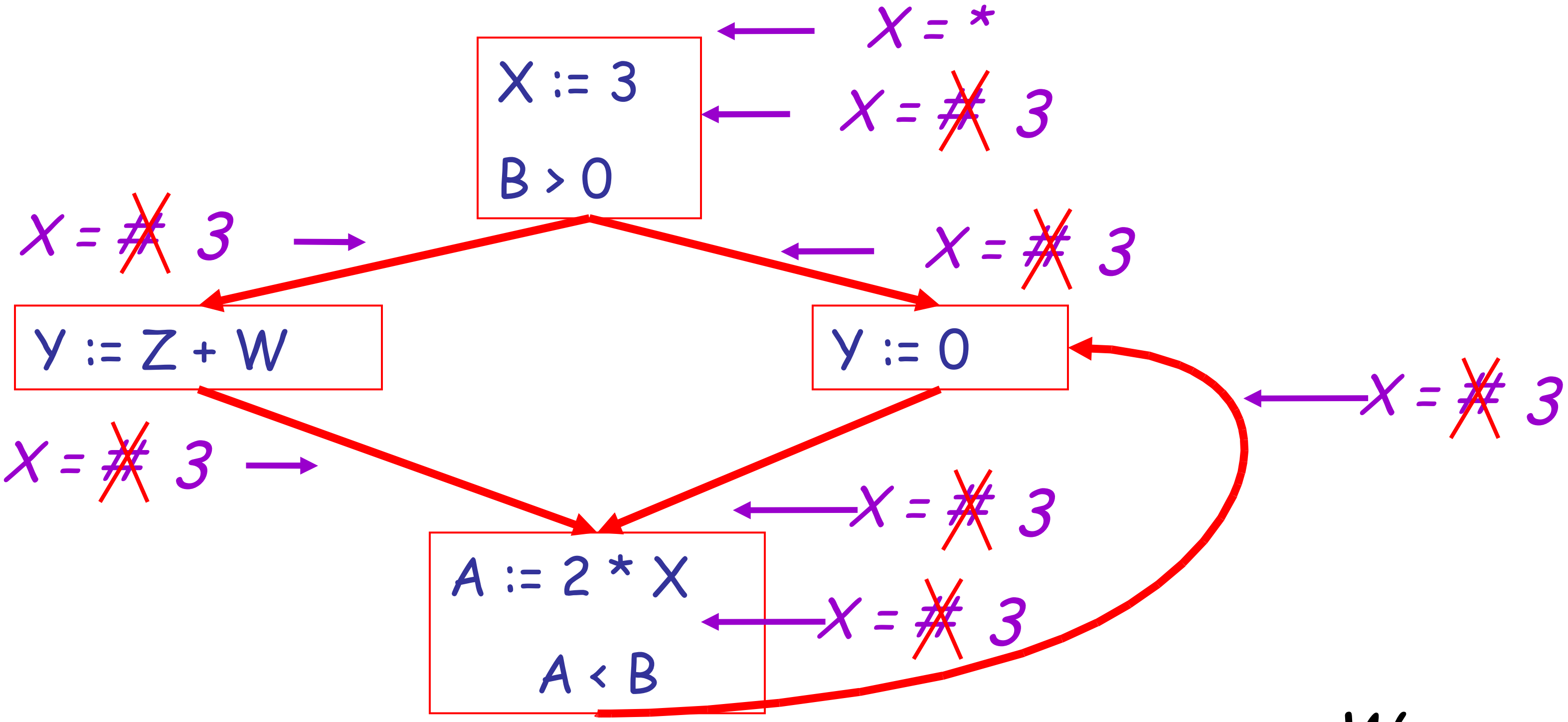
The Value # (Cont.)

- Because of cycles, all points must have values at all times during the analysis
- Intuitively, assigning some initial value allows the analysis to break cycles
- The initial value # means “so far as we know, control never reaches this point”

Sometimes all paths lead to the same place.

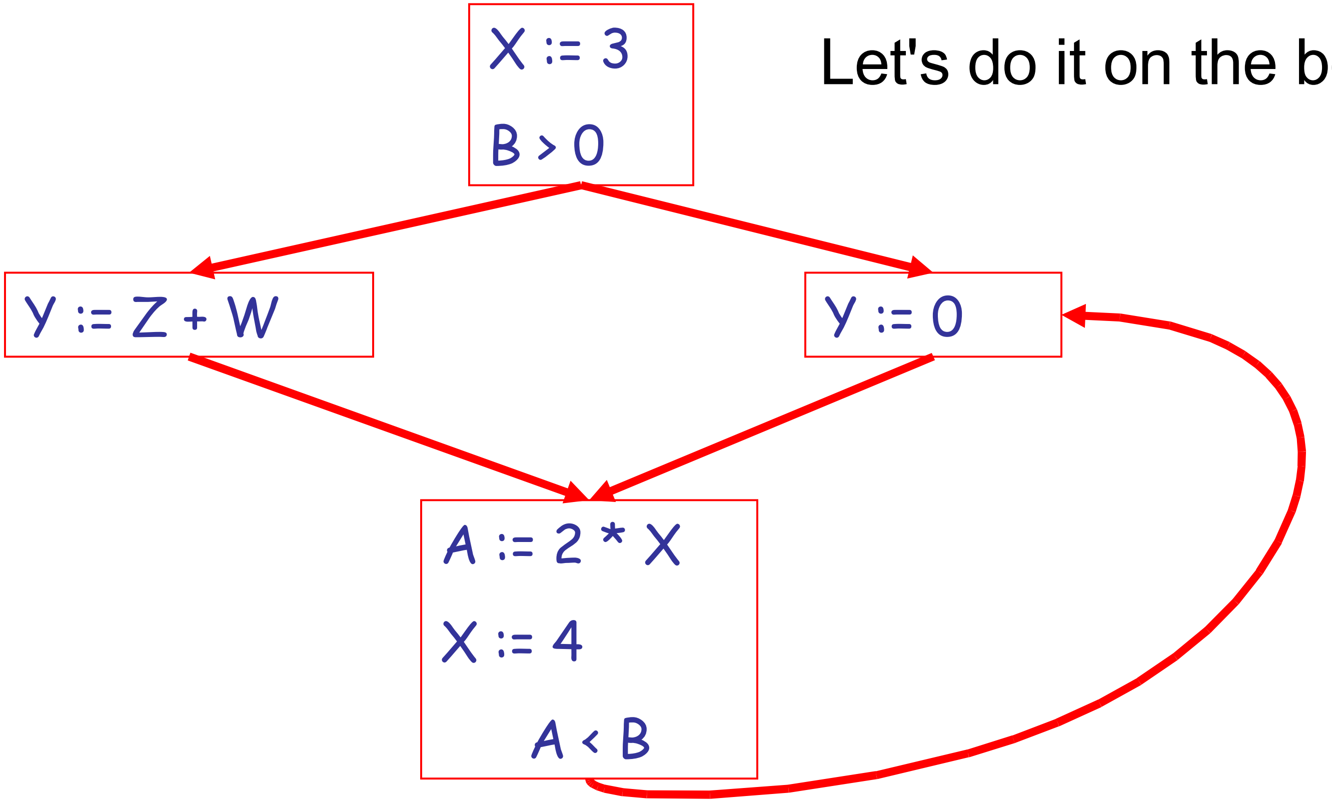
Thus you need #.

Example



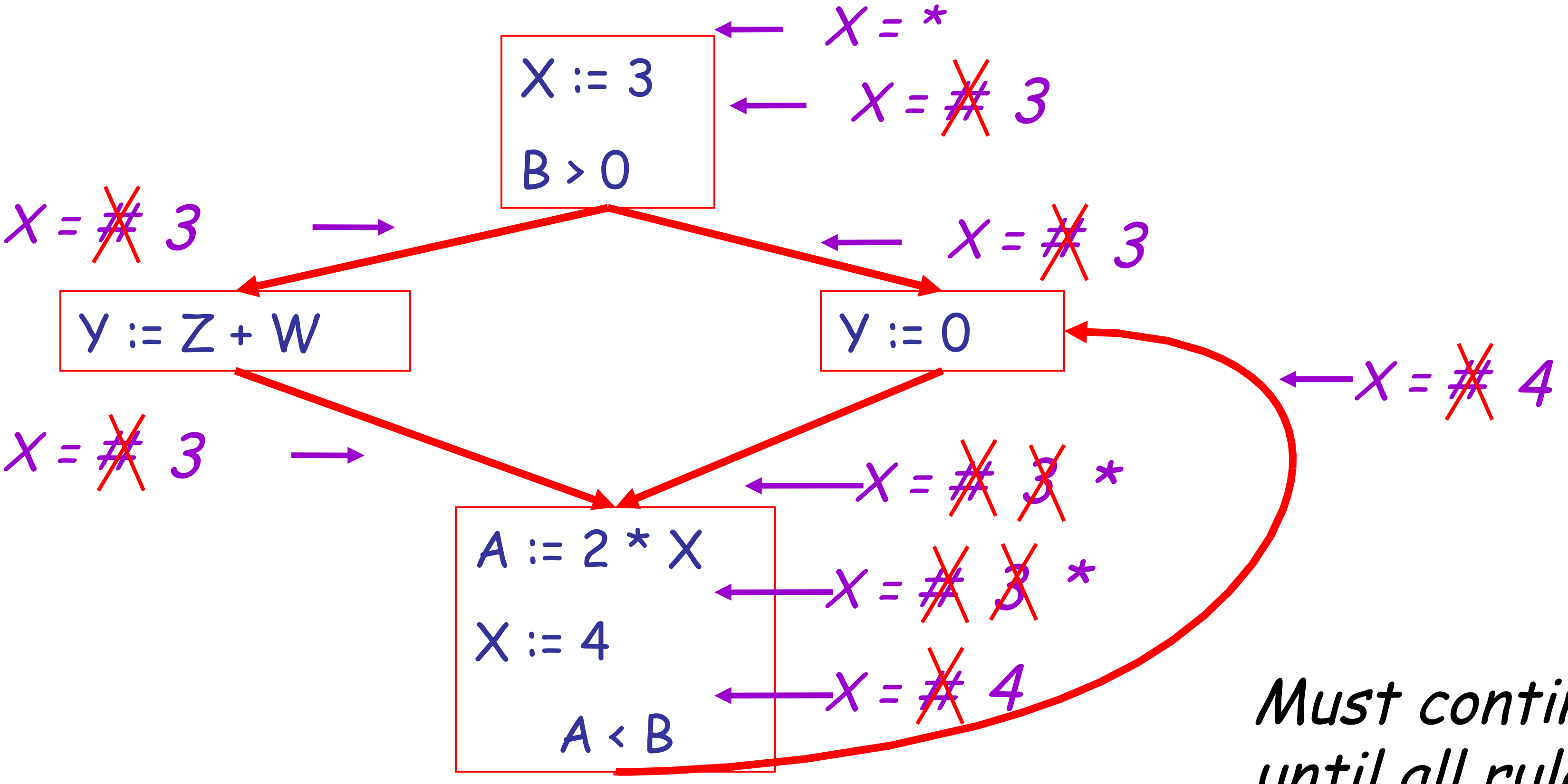
We are done when all rules are satisfied !

Another Example



Let's do it on the board!

Another Example: Answer



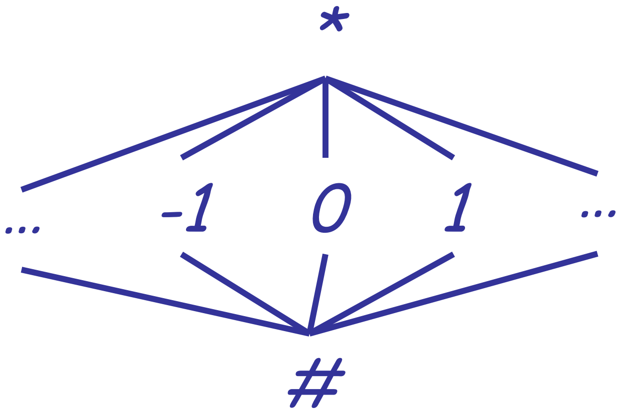
Must continue until all rules are satisfied !

Orderings

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

$$\# < c < *$$

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



Orderings (Cont.)

- * is the greatest value, # is the least
  - All constants are in between and incomparable

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

- Rules 5-8 can be written using lub:

$C_{in}(x, s) = \text{lub } \{ \, C_{out}(x, p) \mid p \text{ is a predecessor of } s \, \}$

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 # and only *increase*
  - # can change to a constant, and a constant to *
  - Thus, C_(x, s) can change at most twice

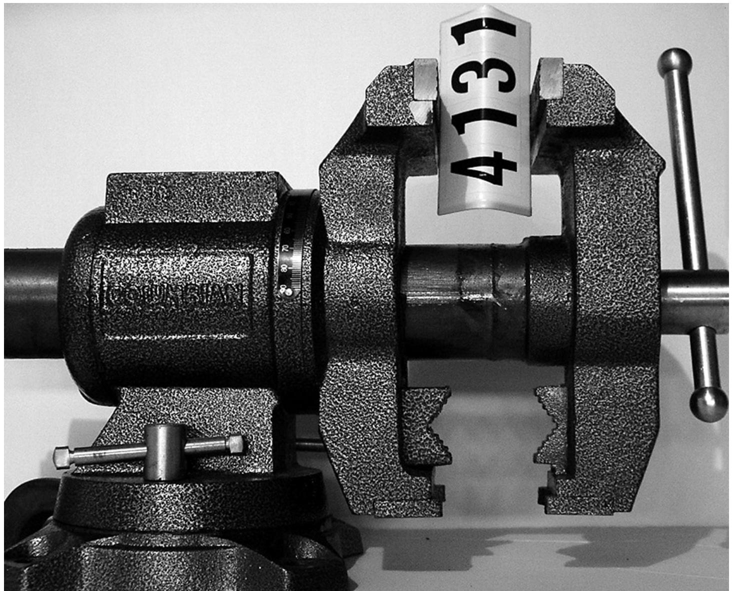
Number Crunching

Thus the algorithm is linear in program size:

Number of steps =

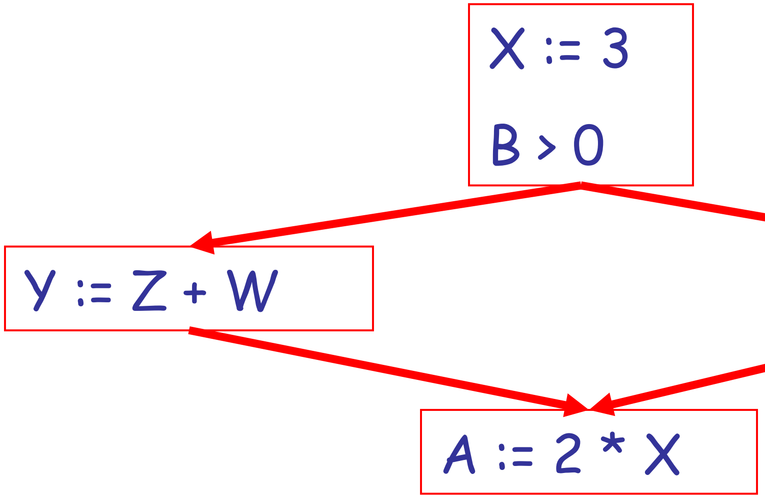
Number of C_(….) values computed * 2 =

Number of program statements * 4



Liveness Analysis

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

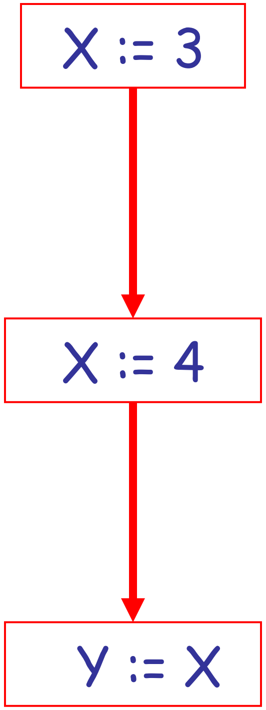


After constant propagation, X := 3 is dead ?

(assuming this is the entire CFG)

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



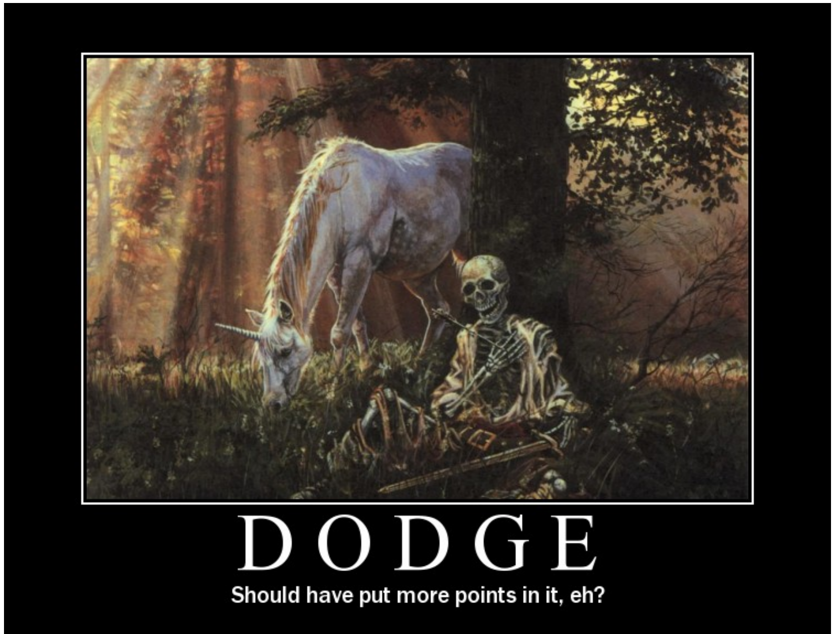
Liveness

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 code can be deleted from the program
- But we need liveness information first . . .

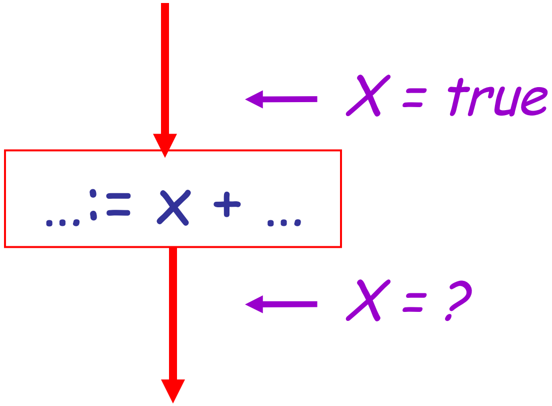


Computing Liveness

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

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

Liveness Rule 1



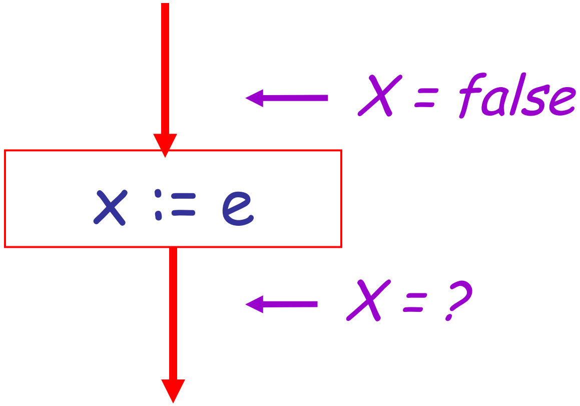
X = true

…:= x + …

X = ?

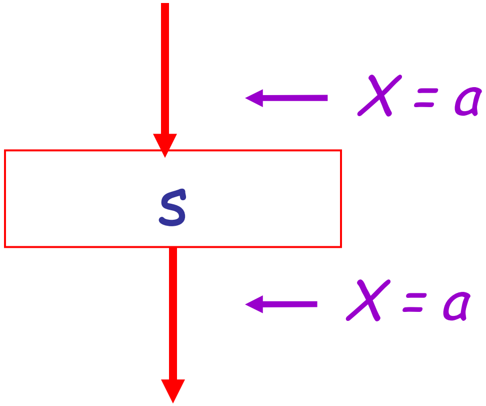
$L_{in}(x, s)$ = true if s refers to x on the rhs

Liveness Rule 2



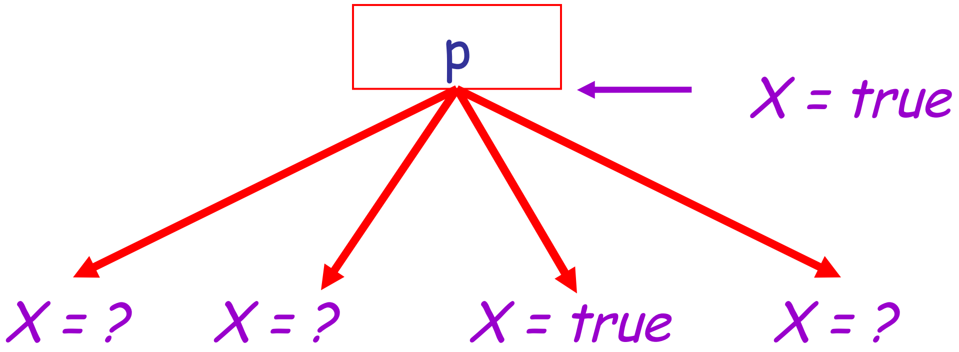
L_{in}(x, x := e) = false if e does not refer to x

Liveness Rule 3



$L_{in}(x, s) = L_{out}(x, s)$ if s does not refer to x

Liveness Rule 4



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

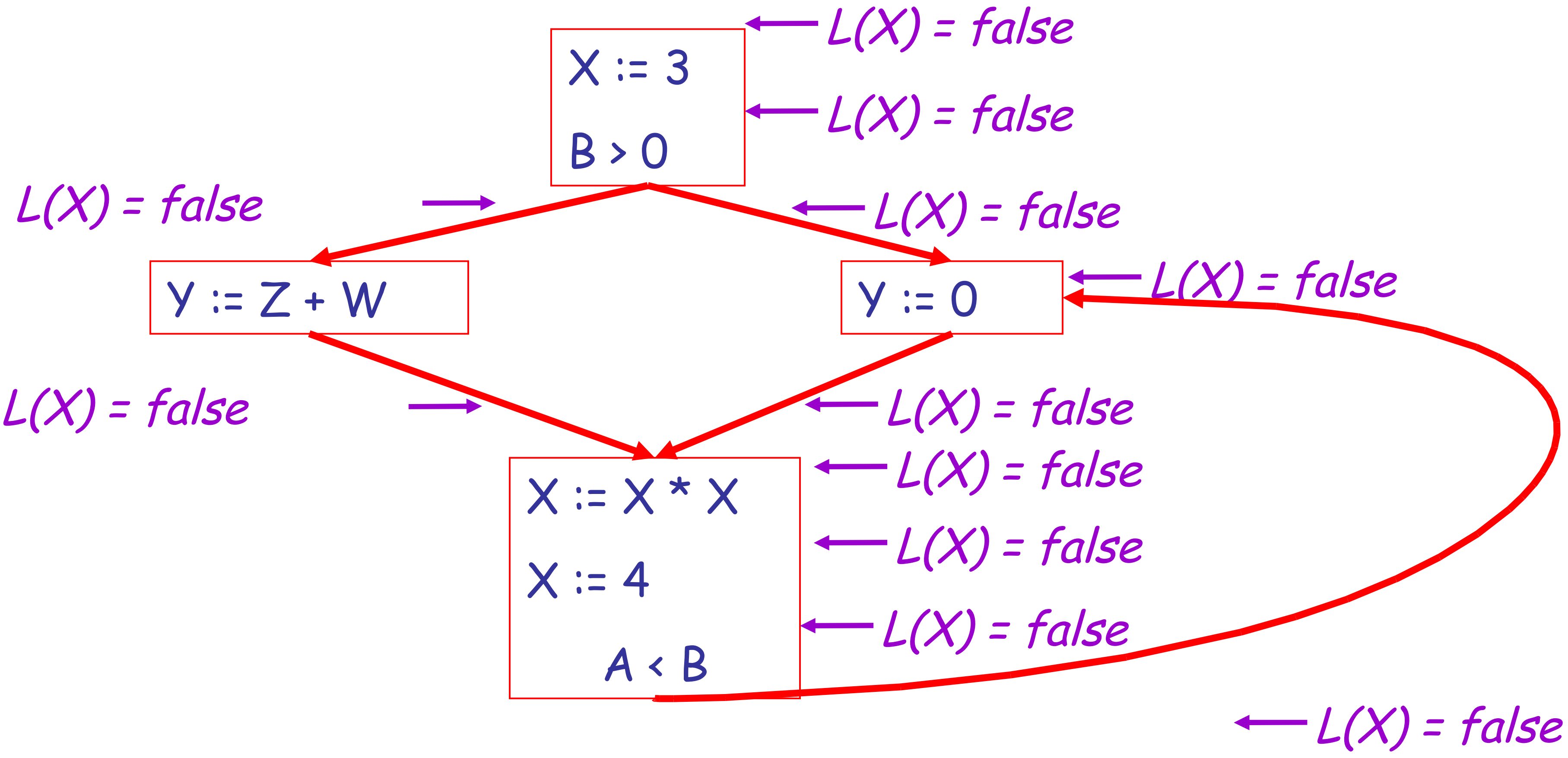
Algorithm

- Let all L_(...) = false initially

- Repeat process until all statements s satisfy rules 1-4 :

  Pick s where one of 1-4 does not hold and update using the appropriate rule

Liveness Example



L(X) = false

X := 3

L(X) = false

B > 0

L(X) = false

L(X) = false

Y := Z + W

Y := 0

L(X) = false

L(X) = false

L(X) = false

L(X) = false

X := X * X

L(X) = false

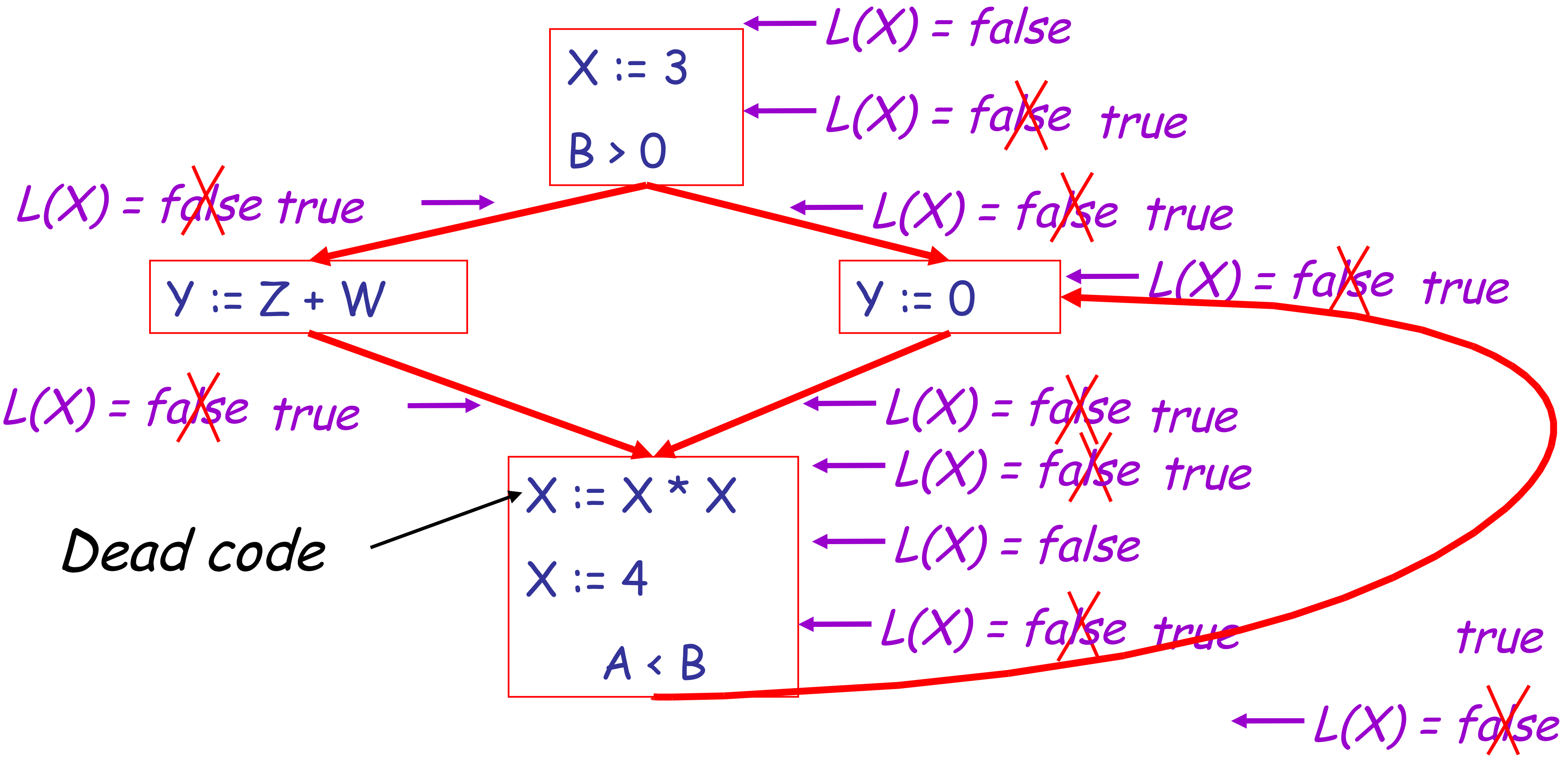
X := 4

L(X) = false

A < B

L(X) = false

Liveness Example Answers



L(X) = false

X := 3

B > 0

L(X) = ~~false~~ true

L(X) = ~~false~~ true

L(X) = ~~false~~ true

Y := Z + W

Y := 0

L(X) = ~~false~~ true

L(X) = ~~false~~ true

L(X) = ~~false~~ true

X := X * X

L(X) = ~~false~~ true

Dead code

X := 4

L(X) = false

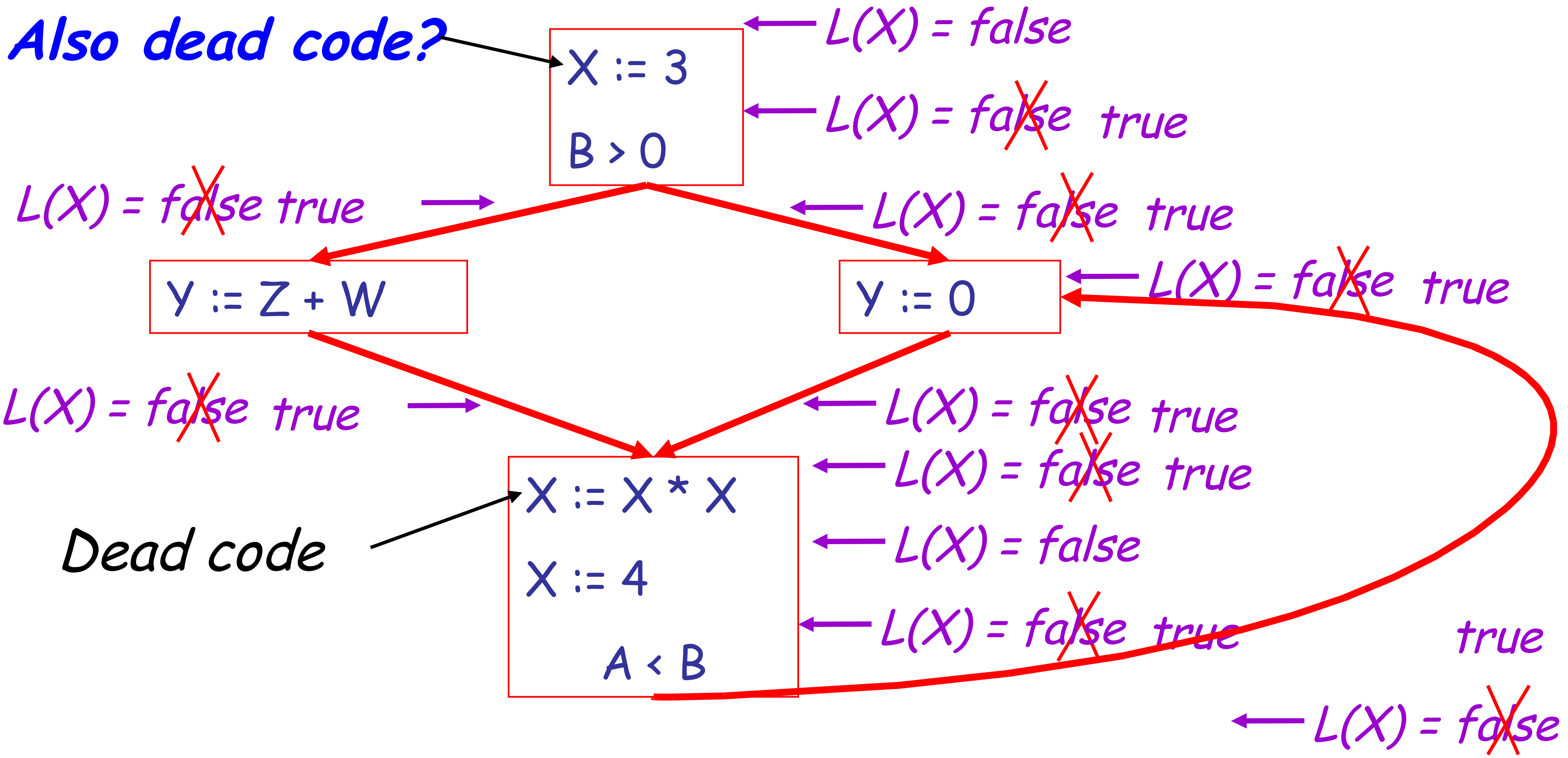
A < B

L(X) = ~~false~~ true

true

L(X) = ~~false~~

Liveness Example Answers



Also dead code?

X := 3

B > 0

L(X) = false

L(X) = ~~false~~ true

L(X) = ~~false~~ true

L(X) = ~~false~~ true

Y := Z + W

Y := 0

L(X) = ~~false~~ true

L(X) = ~~false~~ true

L(X) = ~~false~~ true

L(X) = ~~false~~ true

Dead code

X := X * X

X := 4

A < B

L(X) = false

L(X) = ~~false~~ true

true

L(X) = ~~false~~

Termination

- A value can change from false to true, 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 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

Homework

- WA6 Due Tuesday
- Read chapter 7.7
  - Optional David Bacon article
- Midterm 2 – Tue Apr 15