Global Optimization

Lecture 15

Lecture Outline

- Global dataflow analysis
- Global constant propagation
- Liveness analysis

Local Optimization

Recall the simple basic-block optimizations

- Constant propagation
- Dead code elimination

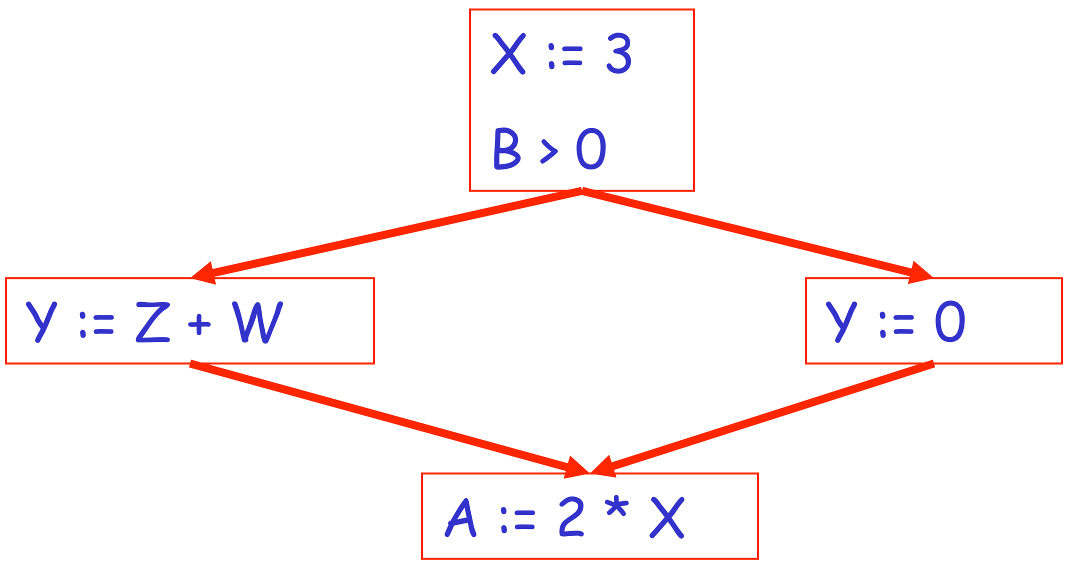
```
X := 3
Y := Z * W
Q := X + Y
```

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

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

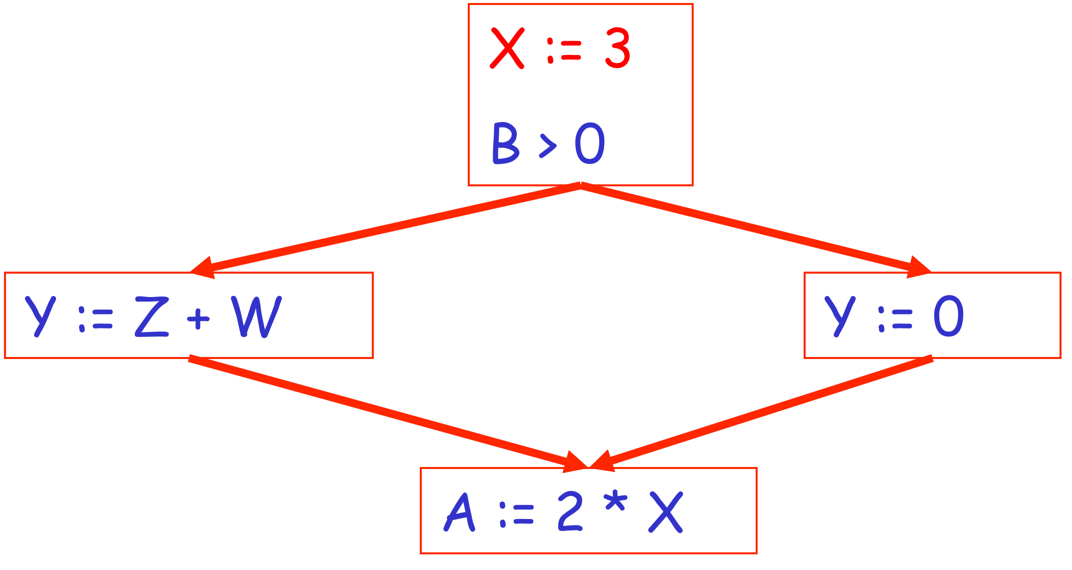
Global Optimization

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



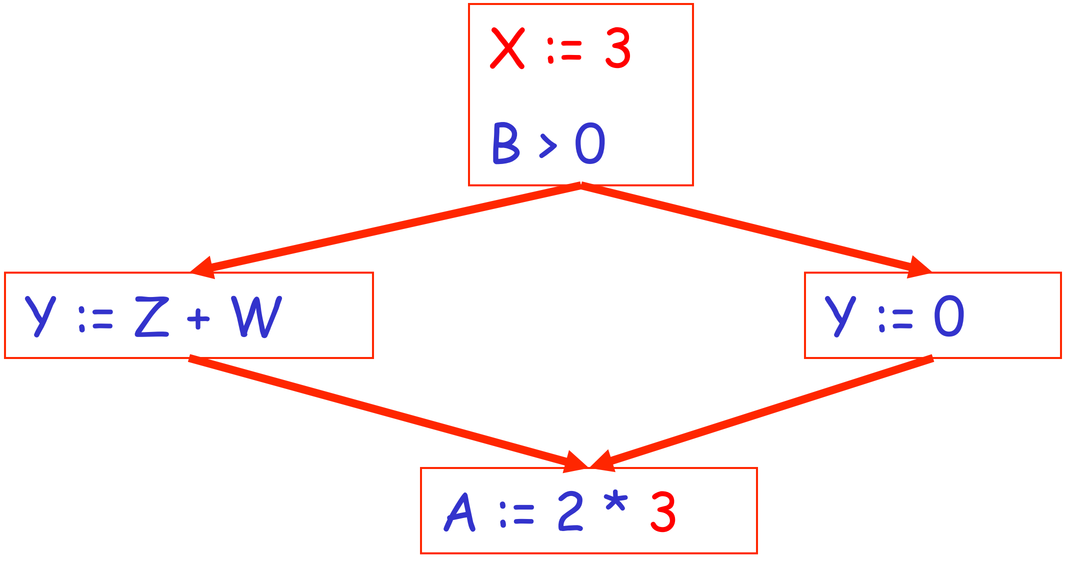
Global Optimization

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



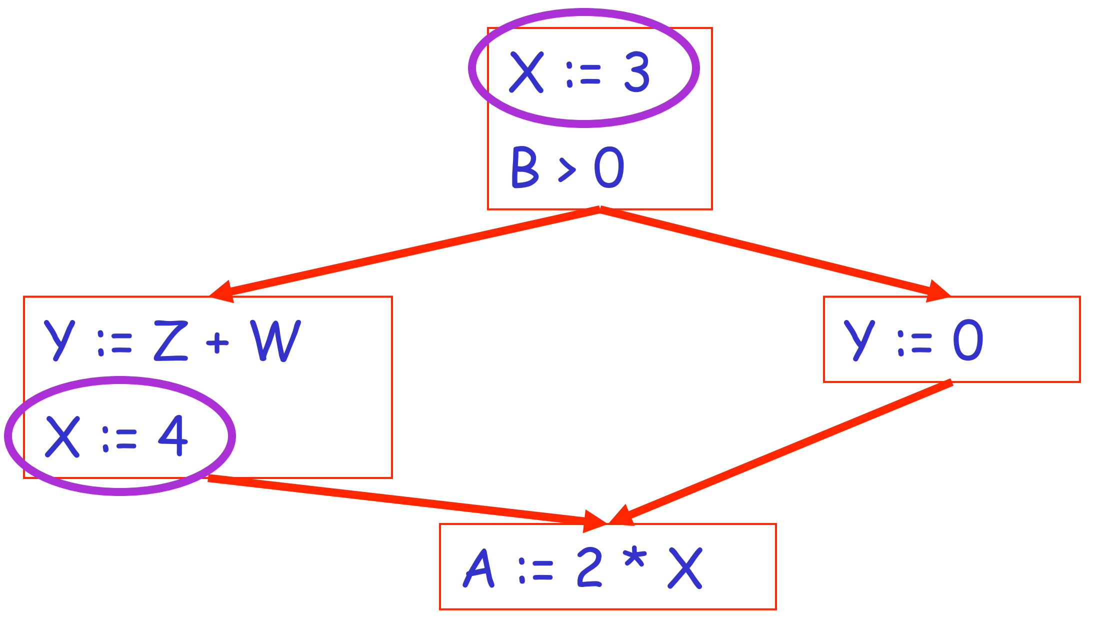
Global Optimization

These optimizations can sometimes 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:

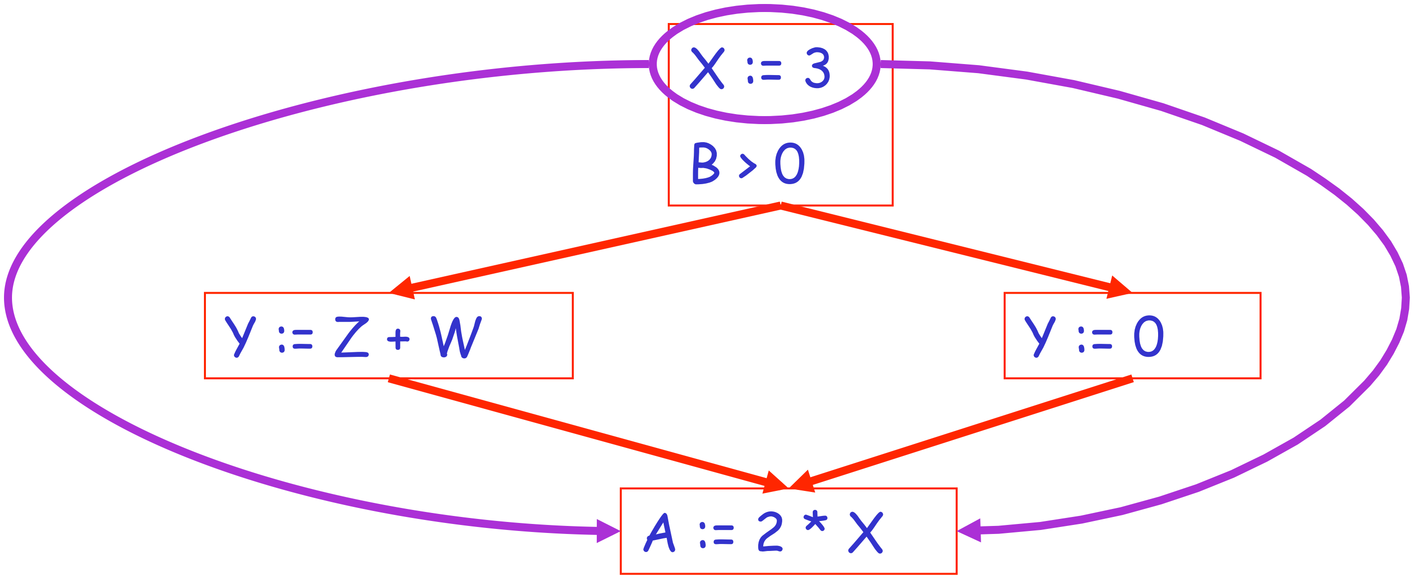


Correctness (Cont.)

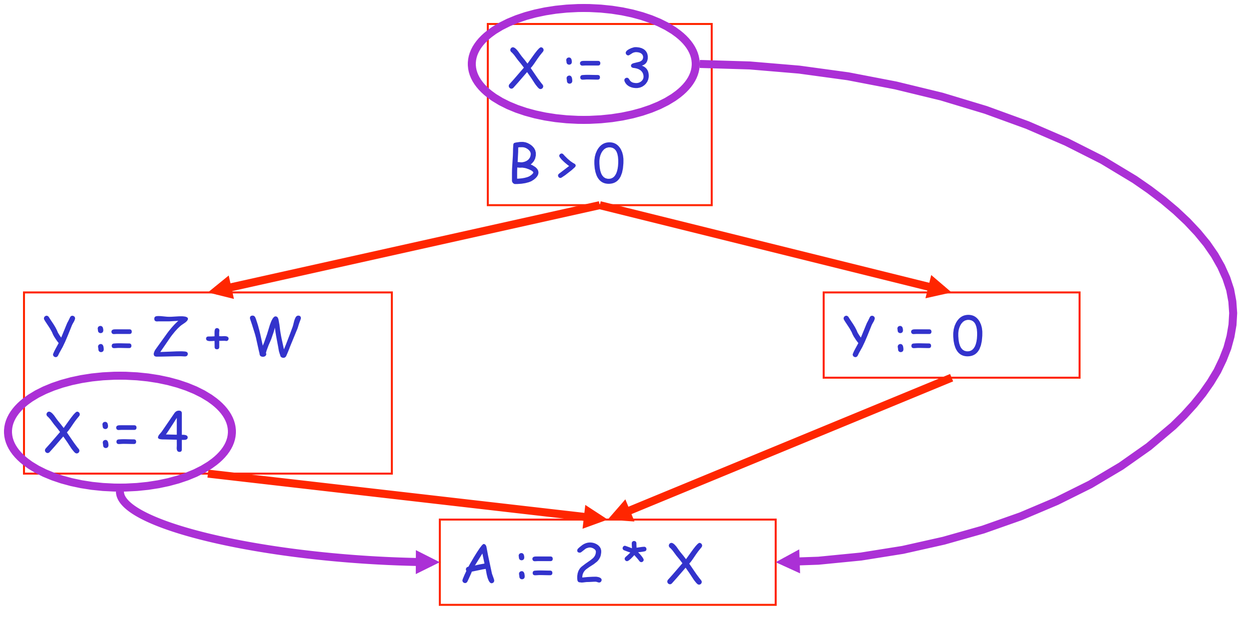
To replace a use of x by a constant k we must know that:

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
  - E.g., is X, at a particular point in the program, guaranteed to be a constant (this is a local property)
- Proving X at any point requires knowledge of the entire program (a global property)
  - E.g., all paths leading to X
- In general, this is a very difficult and expensive problem to solve. What saves us…

Global Analysis

Global optimization tasks share several traits:

- The optimization depends on knowing a property X at a particular point in program execution
  - E.g., is X, at a particular point in the program, guaranteed to be a constant (this is a local property)
- Proving X at any point requires knowledge of the entire program (a global property)
  - E.g., all paths leading to X
- …It is always OK to be conservative. If the optimization requires X to be true, then want to know X is definitely true
- But it's always safe to say "don't know"
  - Because in worst case, you just don't do the optimization

Global Analysis

So, having approximate techniques, that is techniques that don't always give the correct answers to the questions we want to ask, is OK, as long as we are always right when we say that the property holds, and otherwise we just say that we don't know whether the property holds or not.

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

- In what follows, we'll be looking at global constant propagation, and another dataflow analysis, in more detail

Recall: Global Constant Propagation

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 (which we'll call property **)

Global Constant Propagation

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

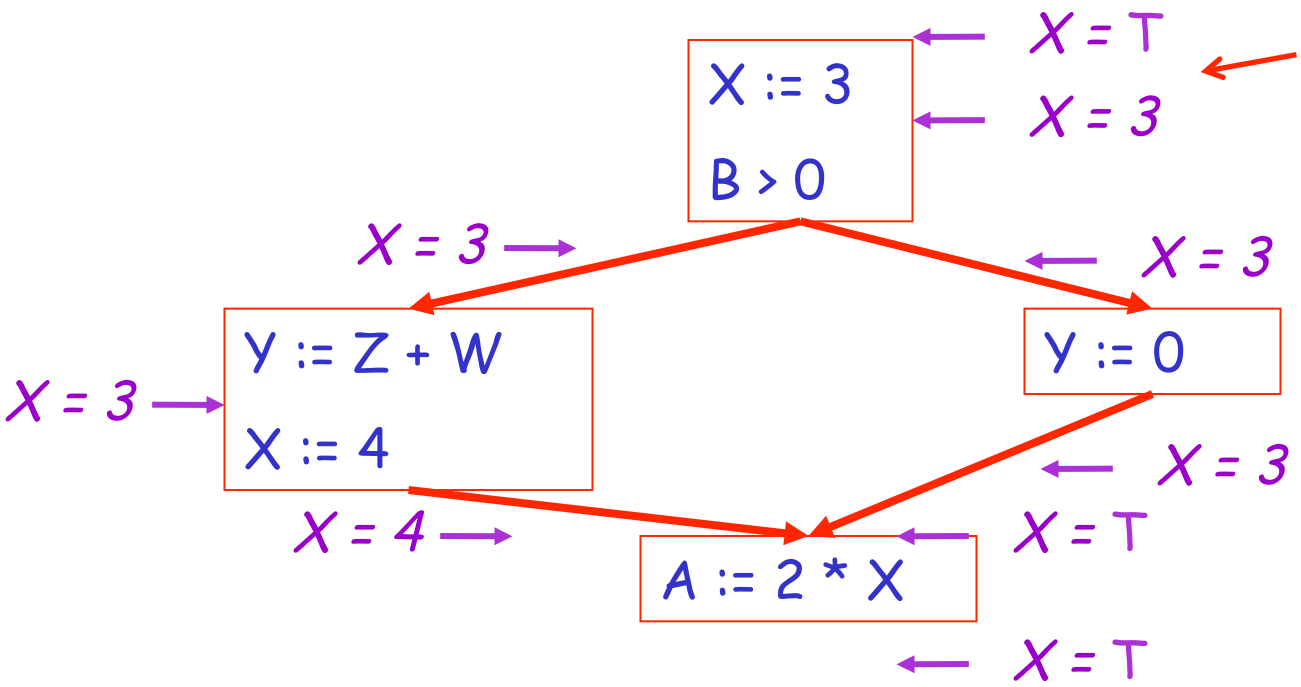
- Consider the case of computing property ** for a single variable X at all program points
  - Note it's easy to extend this to the case of all program variables
    - One simple, but inefficient way is to simply use any single variable algorithm repeatedly, once for each variable in the method body

Global Constant Propagation (Cont.)

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

value	*interpretation*
Bot ("bottom")	This statement never executes
c	X = constant c
T (pronounced "top")	X is not a constant

Example



note that program points are in between statements

E.g., X = 3 after assignment but before predicate

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 $x = ?$
  - That is, how, in a systematic fashion and on an arbitrary control flow graph, do we compute these properties for every program point

The Idea (Data flow analysis basic principle)

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

(put another way, we build up global information only by looking at local information)

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

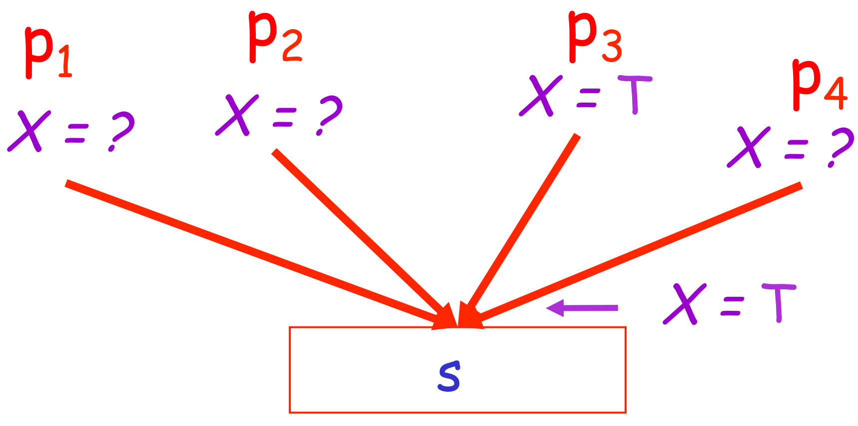
  $C(x,s,out)$ = value of x after s

Transfer Functions

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

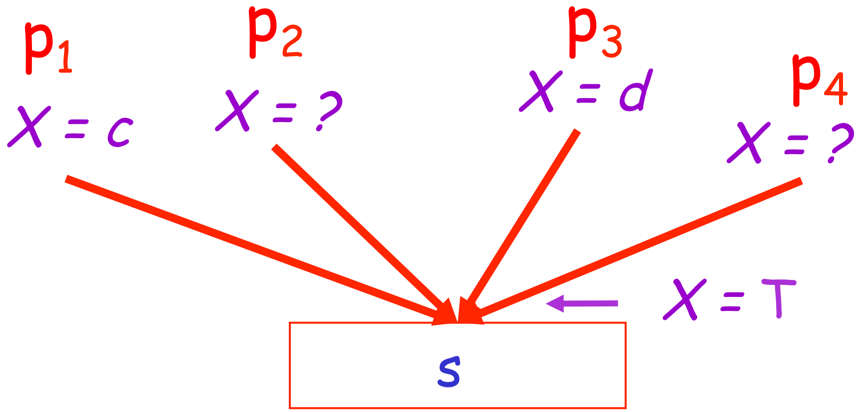
- In the following rules, let statement s have immediate predecessor statements $p_1,\ldots,p_n$
  - Note that though there are possibly multiple predecessors here, each leads to statement s in one step

Rule 1



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

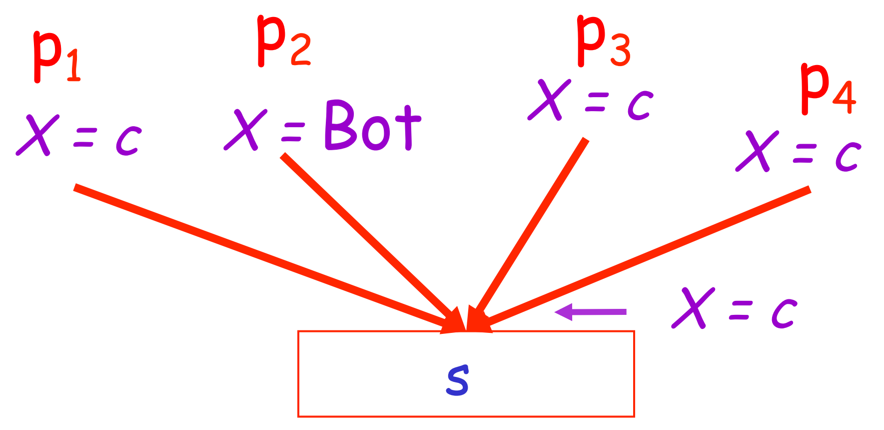
Rule 2



$C(p_i, x, out) = c$ & $C(p_j, x, out) = d$ & $d \neq c$ then

$C(s, x, in) = T$

Rule 3

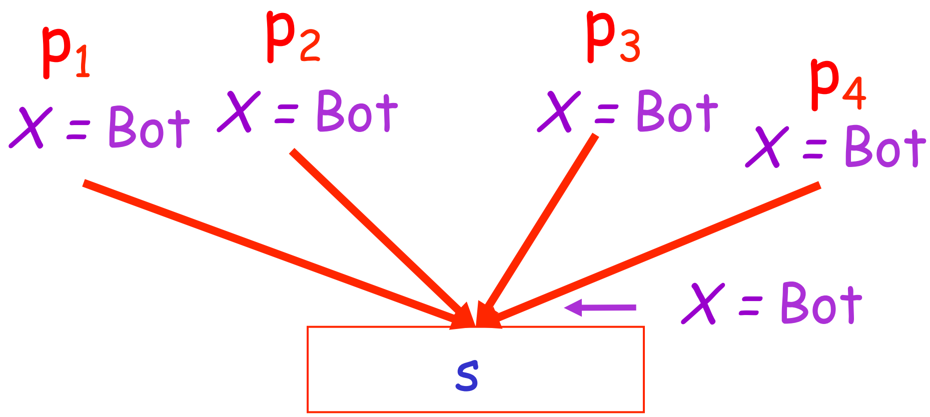


if $C(p_i, x, out) = c$ or Bot for all i,

then $C(s, x, in) = c$

This of course makes sense because Bot means that the statement is never reached

Rule 4



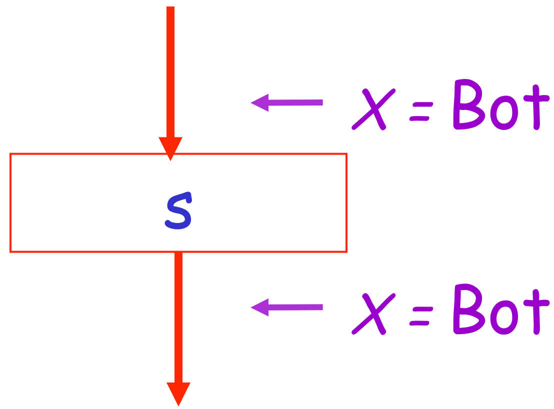
if $C(p_i, x, out)$ = Bot for all i,

then $C(s, x, in)$ = Bot

The Other Half

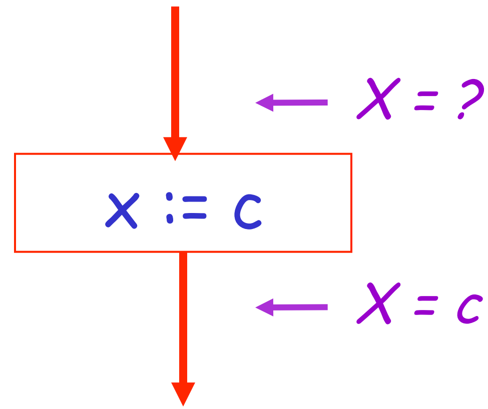
- Rules 1-4 relate the *out* of one statement to the *in* of the next statement
- Now we need rules relating the *in* of a statement to the *out* of the same statement

Rule 5



$C(s, x, out) = Bot$ if $C(s, x, in) = Bot$

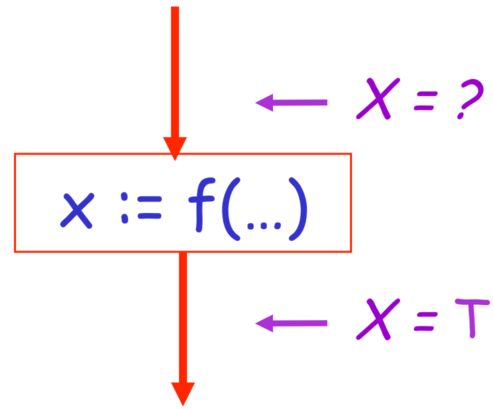
Rule 6



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

(note that Rule 5 has precedence over Rule 6: if X is Bot before the statement, then it's Bot after the statement, even though there is an assignment. In general, if we can assign Bot to something, we do.)

Rule 7

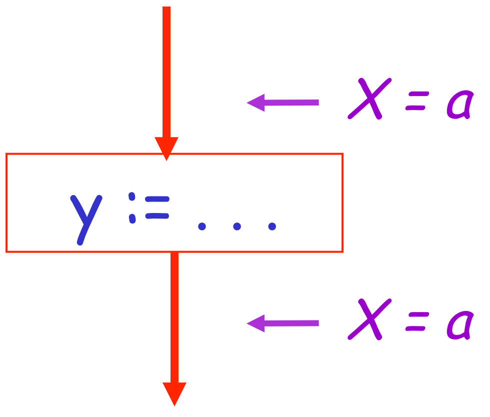


This rule handles every assignment other than a constant assignment

$$C(x := f(\ldots), x, out) = \top$$

(once again, Rule 5 takes precedence if it applies)

Rule 8



$C(y := \ldots, x, out) = C(y := \ldots, x, in)$ if $x \neq y$

An Algorithm

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

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

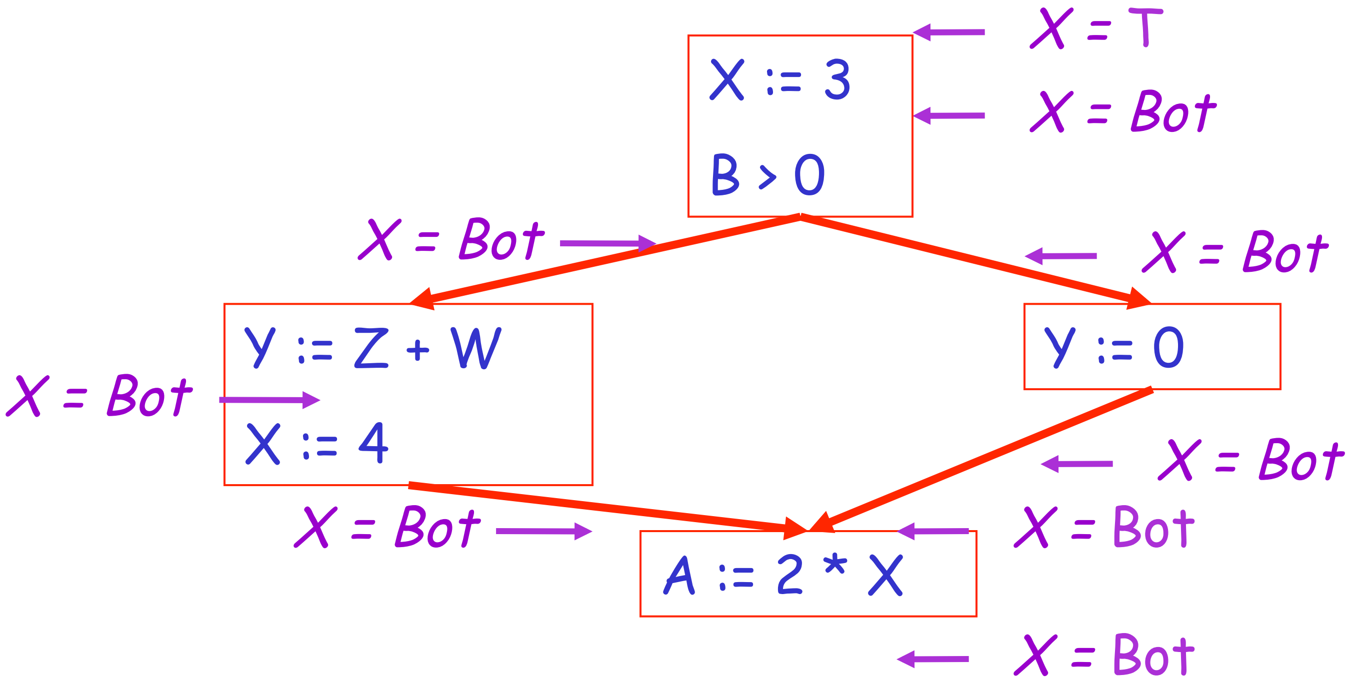
   E.g., we assume (at first) that none of the other statements are ever executed

3. Repeat until all points satisfy 1-8:

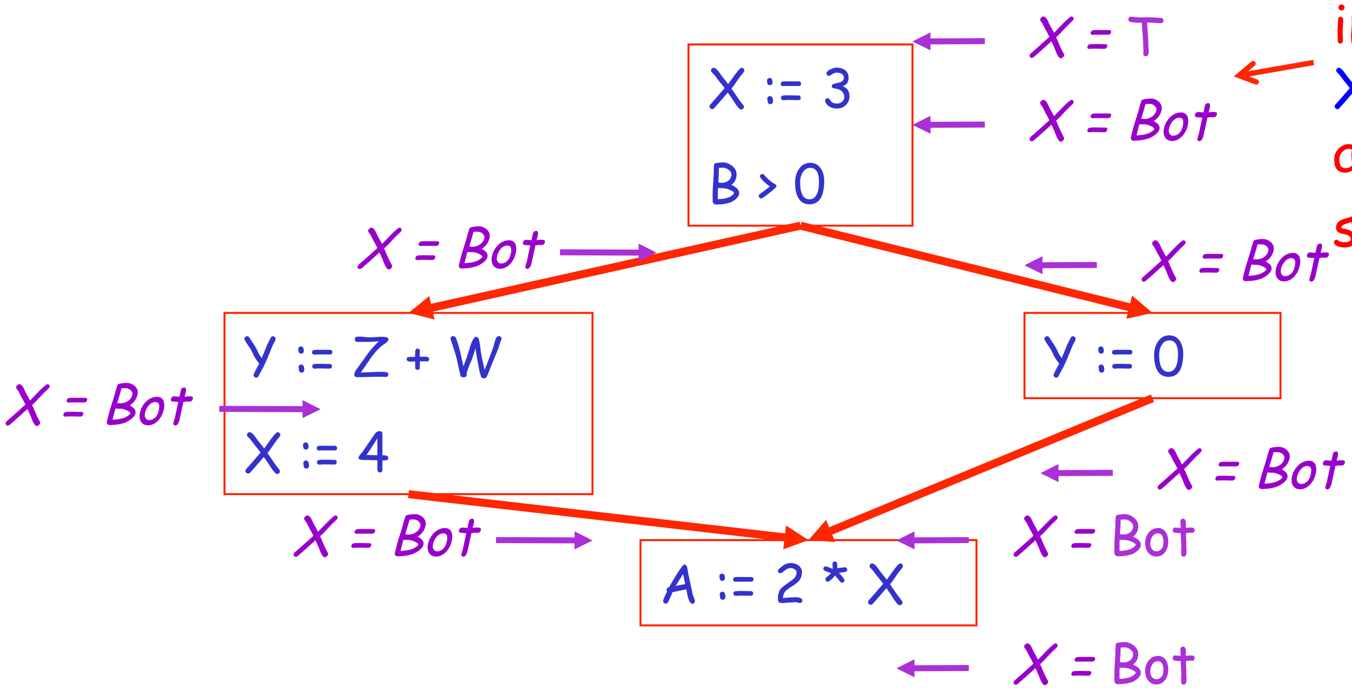
   Pick s not satisfying any of Rules 1-8 and update using the appropriate rule

   (look for places in the CFG where information is inconsistent according to the rules, and update)

Example

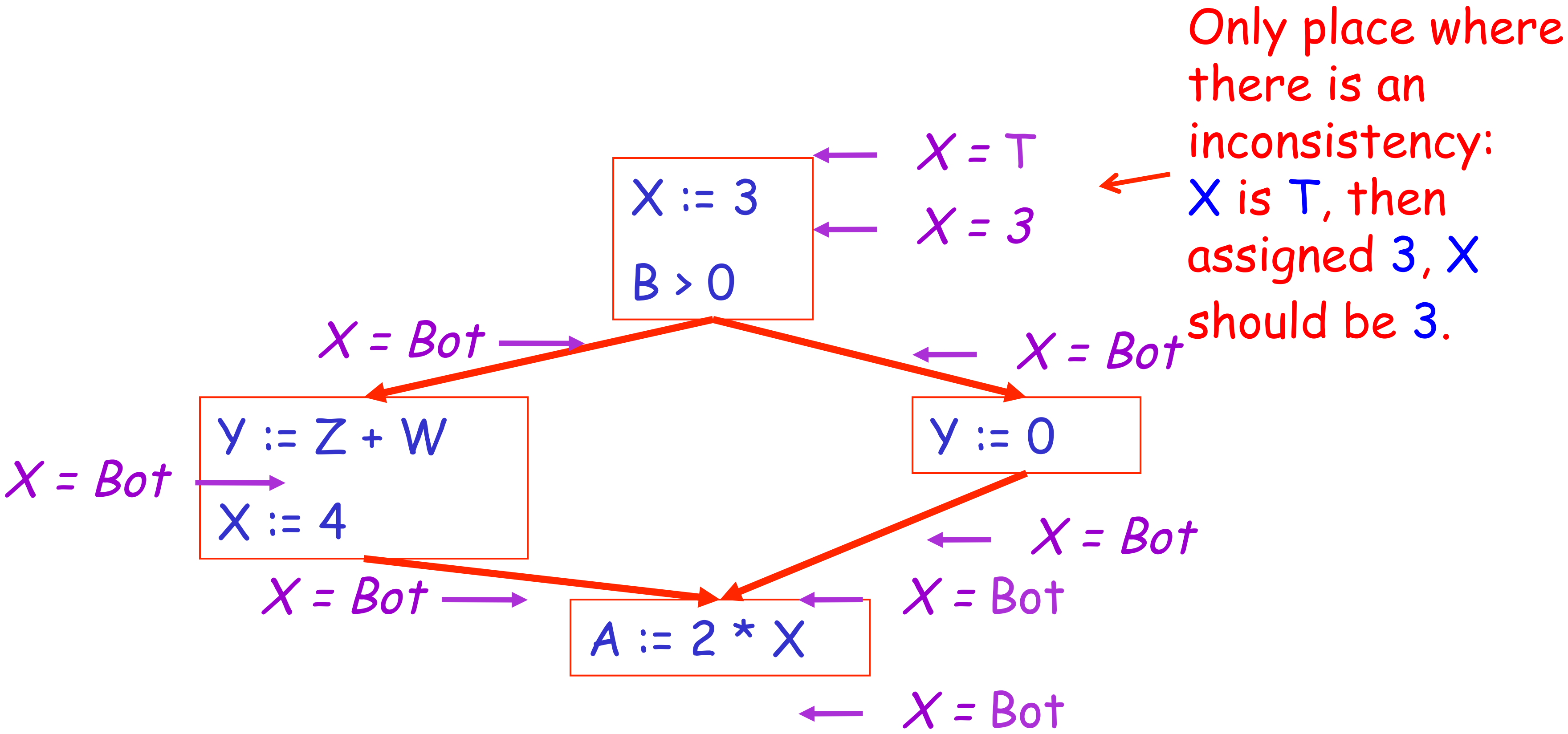


Example



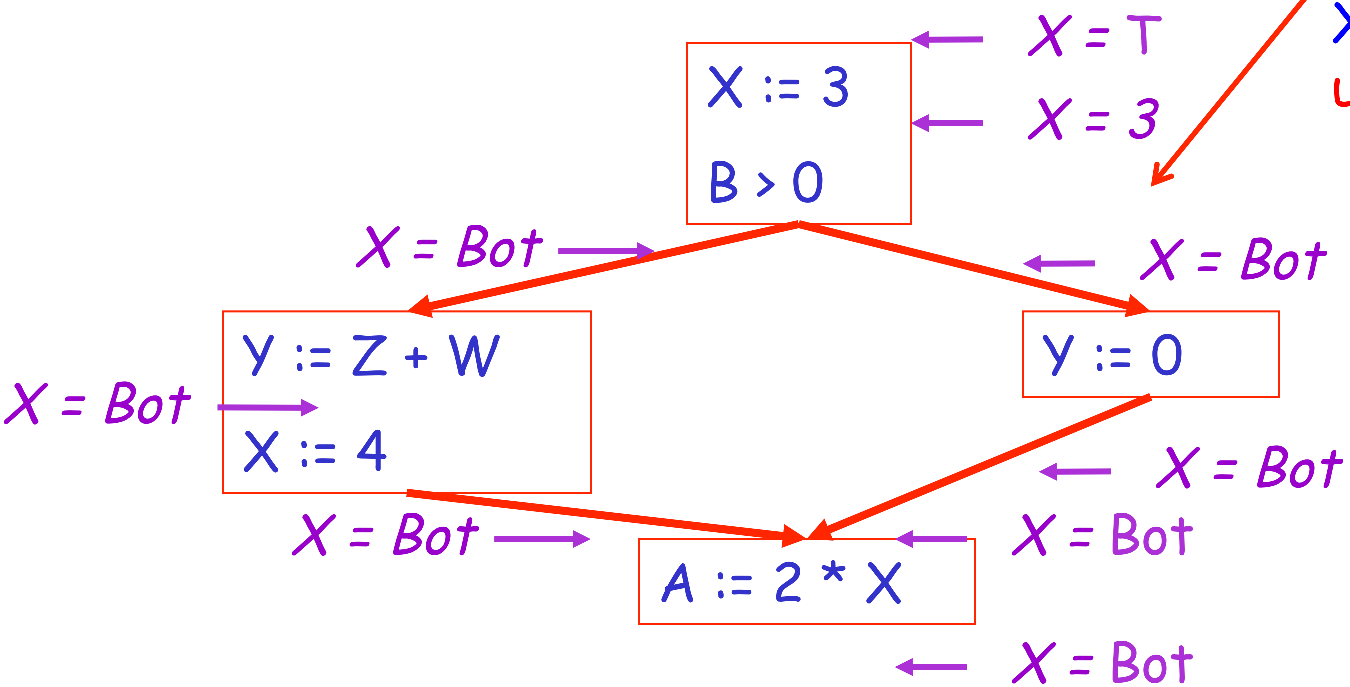
Only place where there is an inconsistency: X is T, then assigned 3, X should be 3.

Example



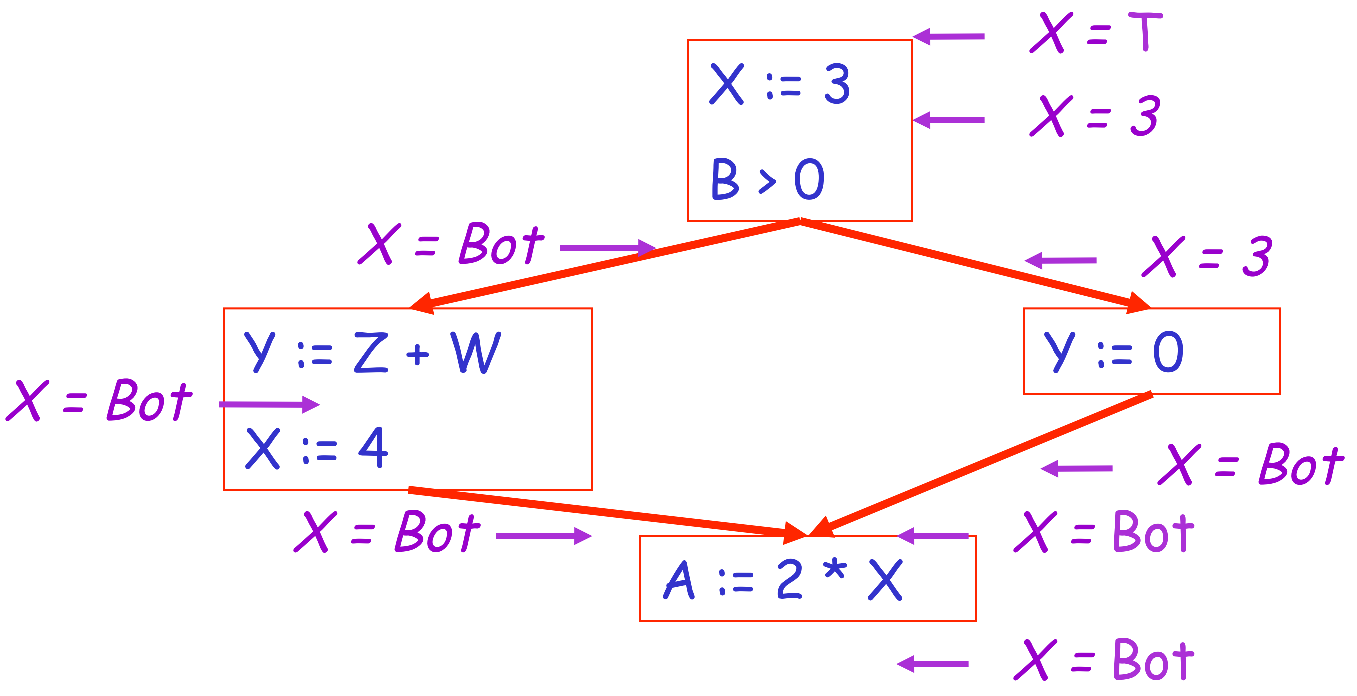
X = T
X := 3
X = 3
B > 0
X = Bot
X = Bot
Y := Z + W
X = Bot
X := 4
Y := 0
X = Bot
X = Bot
X = Bot
A := 2 * X
X = Bot
Only place where there is an inconsistency: X is T, then assigned 3, X should be 3.

Example

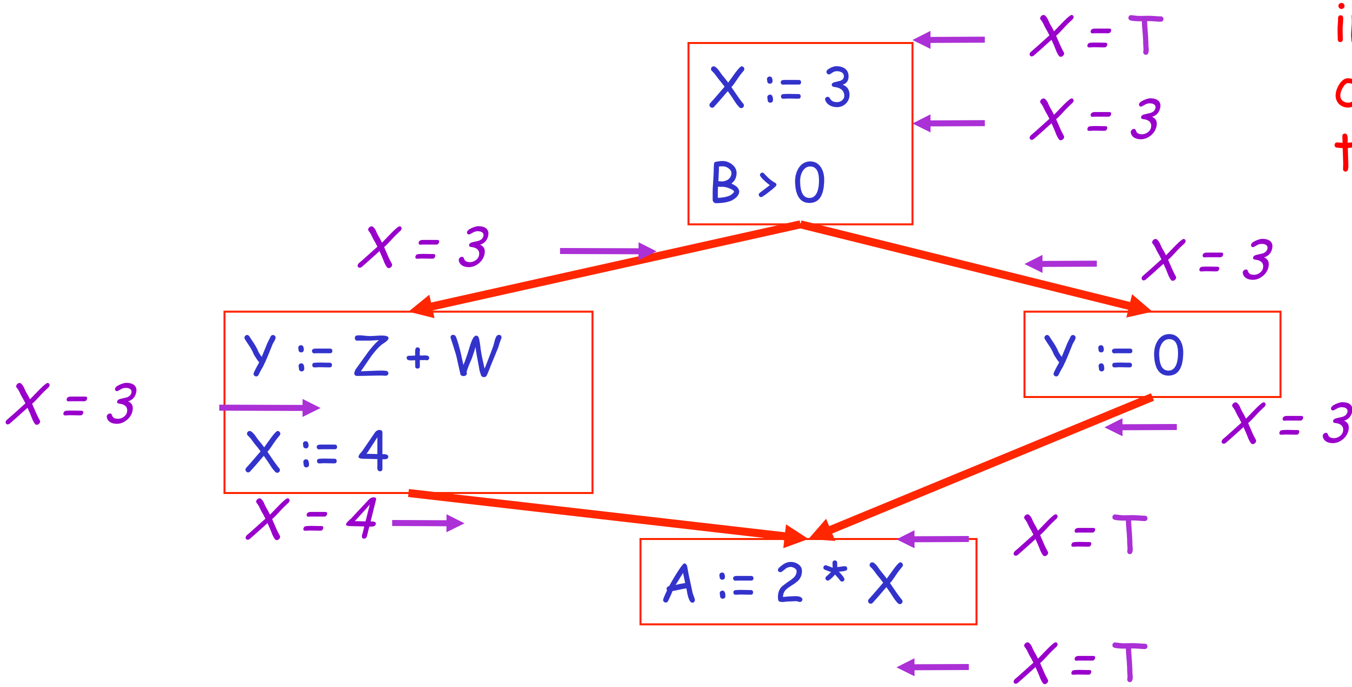


But now this is inconsistent: X is 3, but then unreachable

Example



Example



At this point, all the information is consistent with the rules

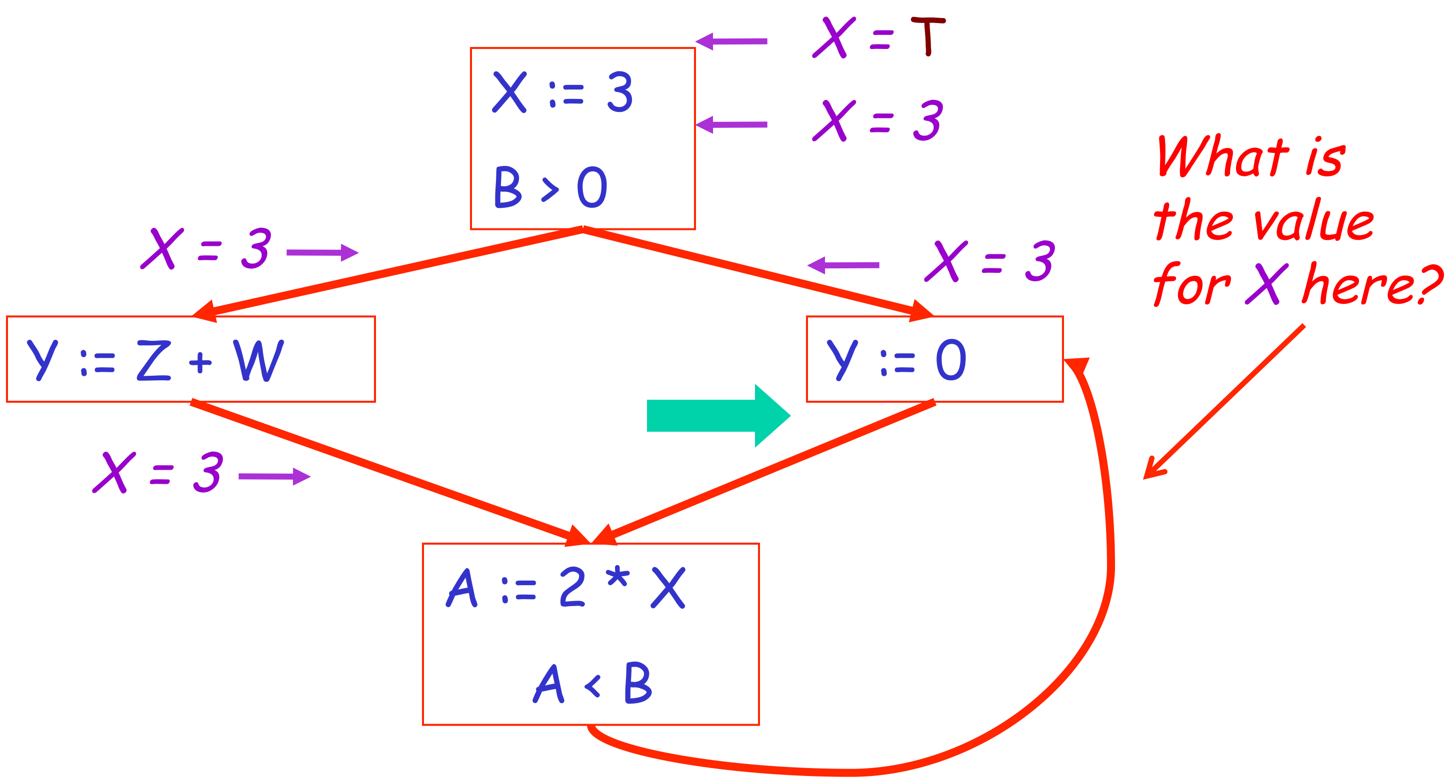
More Control Flow Analysis: Analysis of Loops

- This is probably the most interesting part of control flow analysis!

- But first, lets take another look at that Bot thing
  - Because the need for it is intimately tied to the analysis of loops

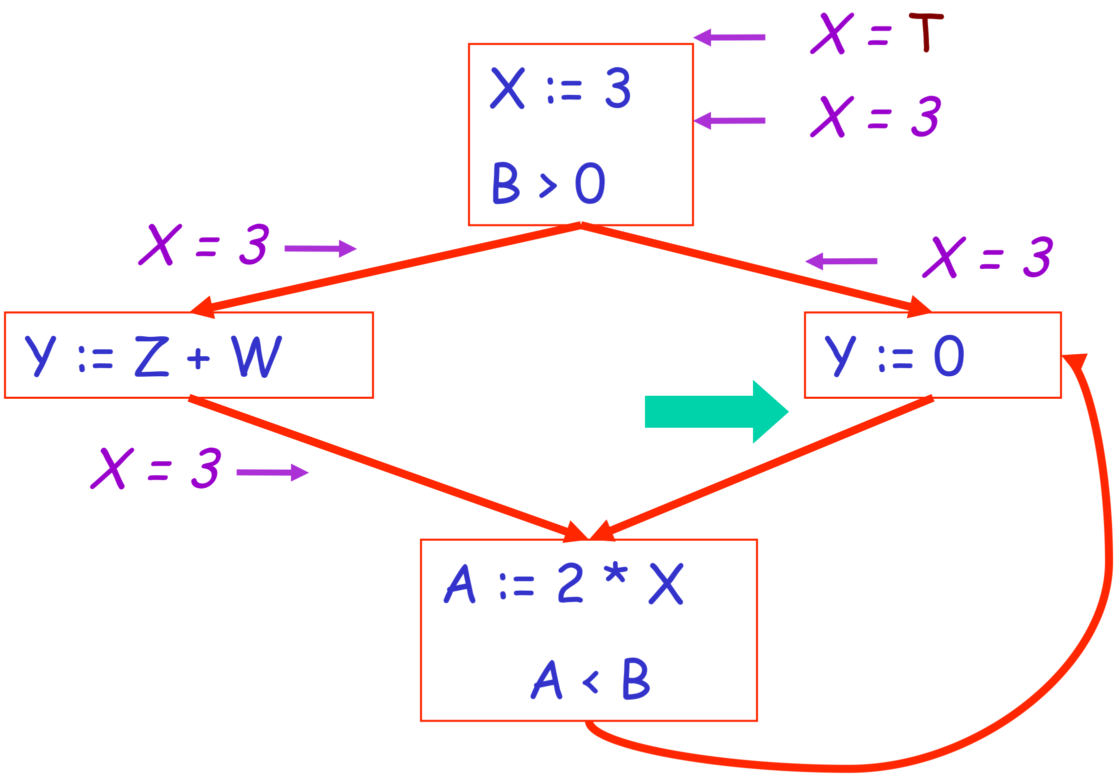
The Value Bot

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



The Value Bot

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



Consider the predecessors. What are they?

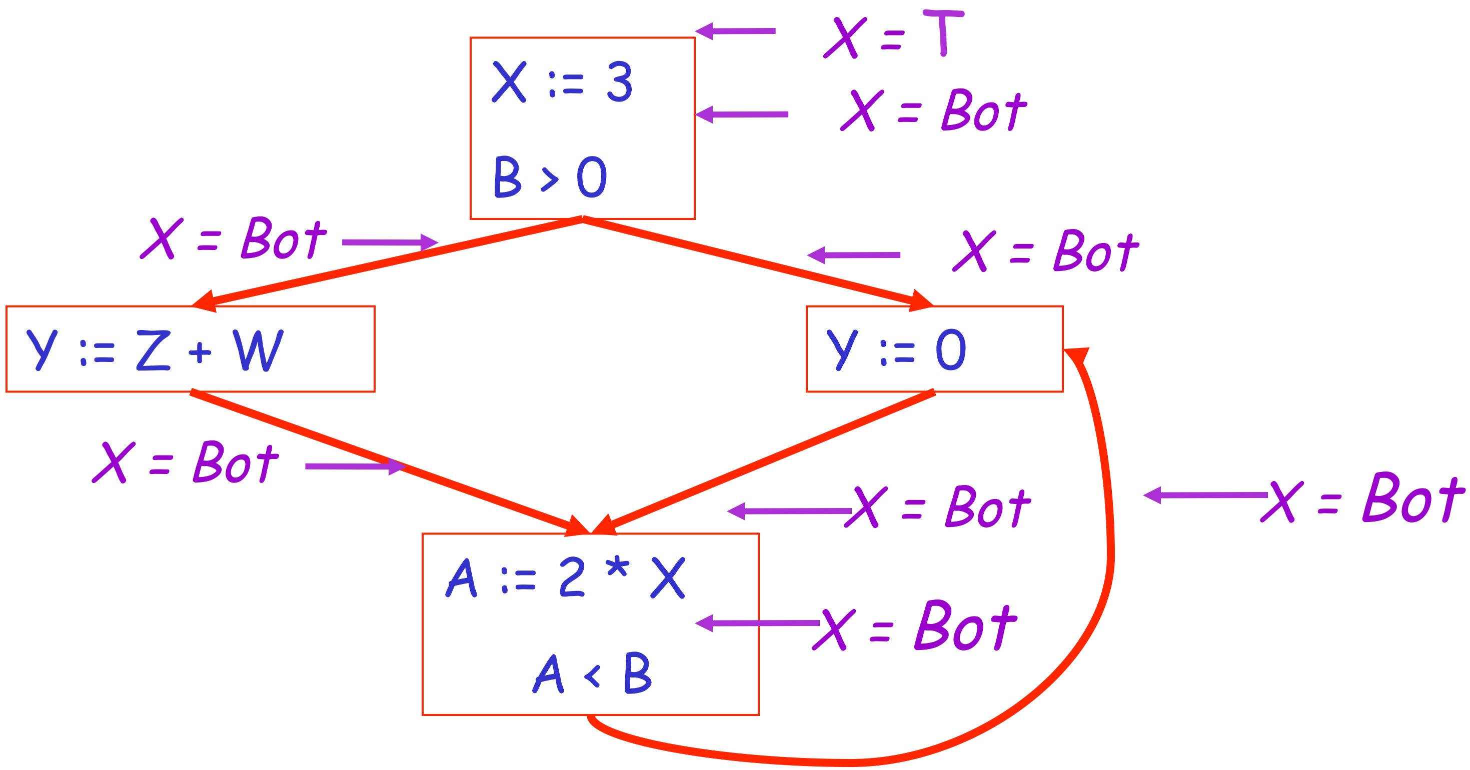
Discussion

- 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
    - Note other statements don't involve X
- But info for A := 2 * X depends on its predecessors, including Y := 0!
  - So how do we get information about the predecessors of Y:= 0 when they depend on themselves?

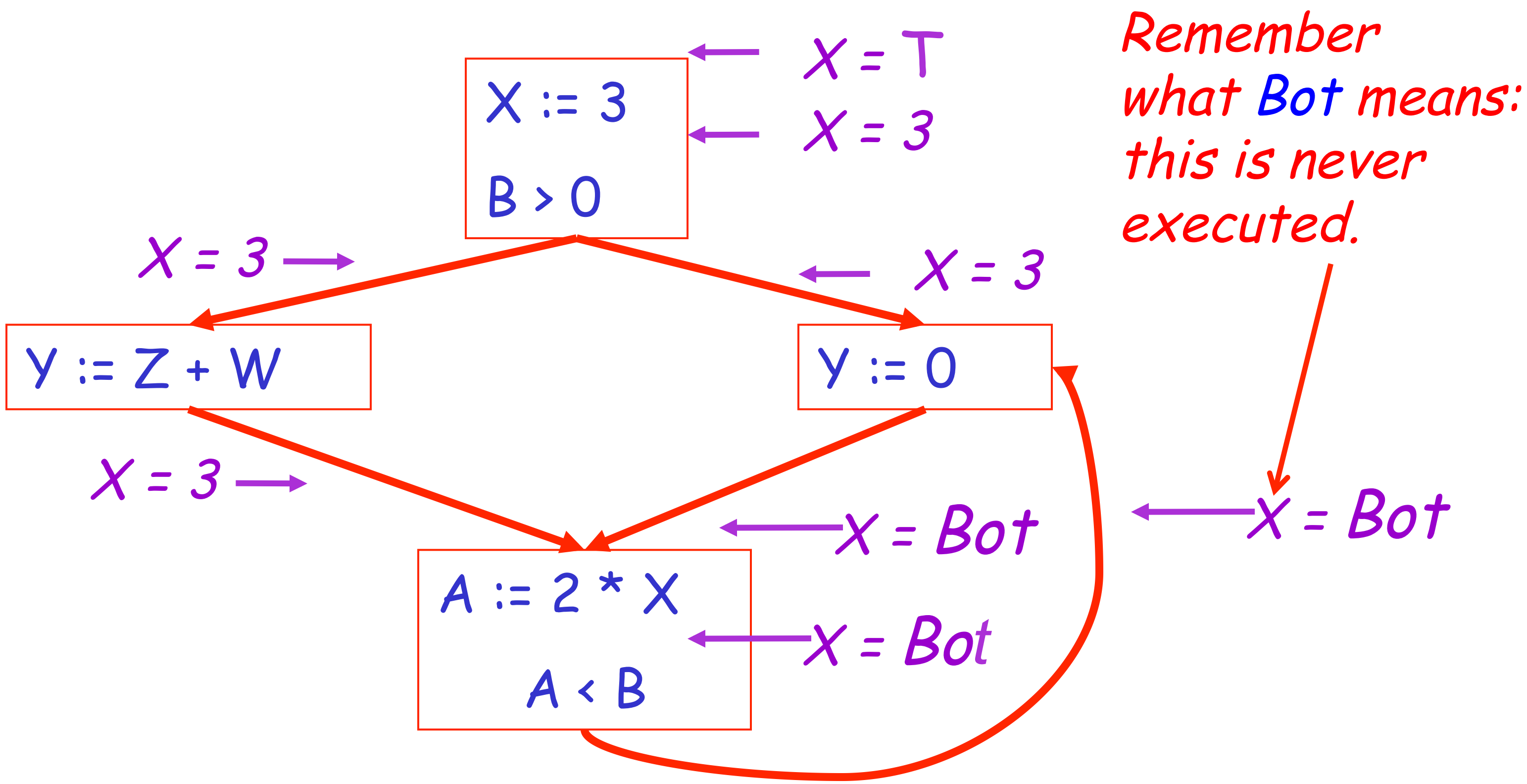
A Standard Solution…

- Used in many areas of math when you have recursive or recurrence relations
- Because of cycles, all points must have values at all times
- Intuitively, assigning some initial value allows the analysis to break cycles
  - I.e., Break the cycle be starting with some initial guess (which here, turns out to be Bot)
- The initial value Bot means "So far as we know, control never reaches this point"

Example

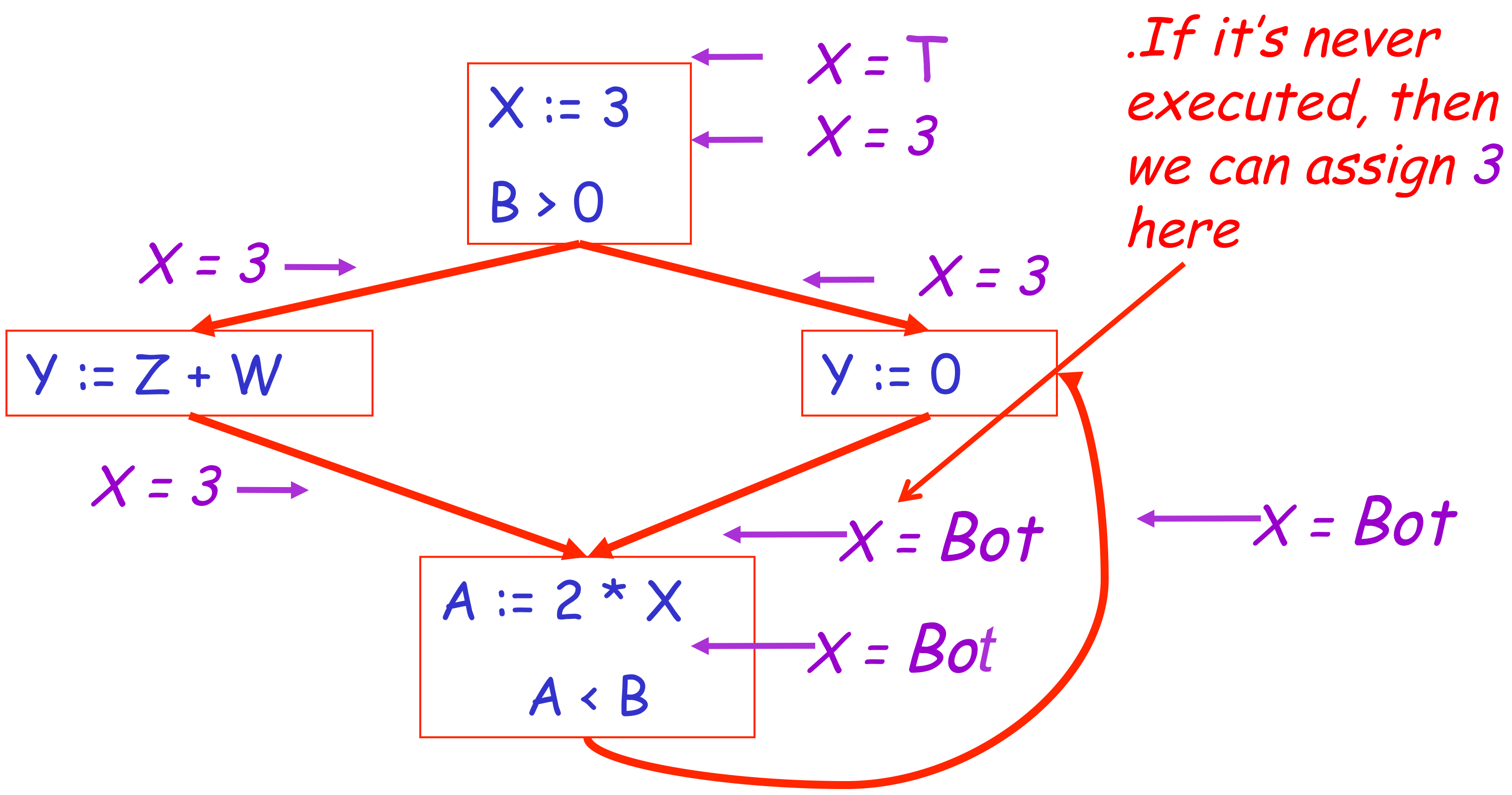


Example

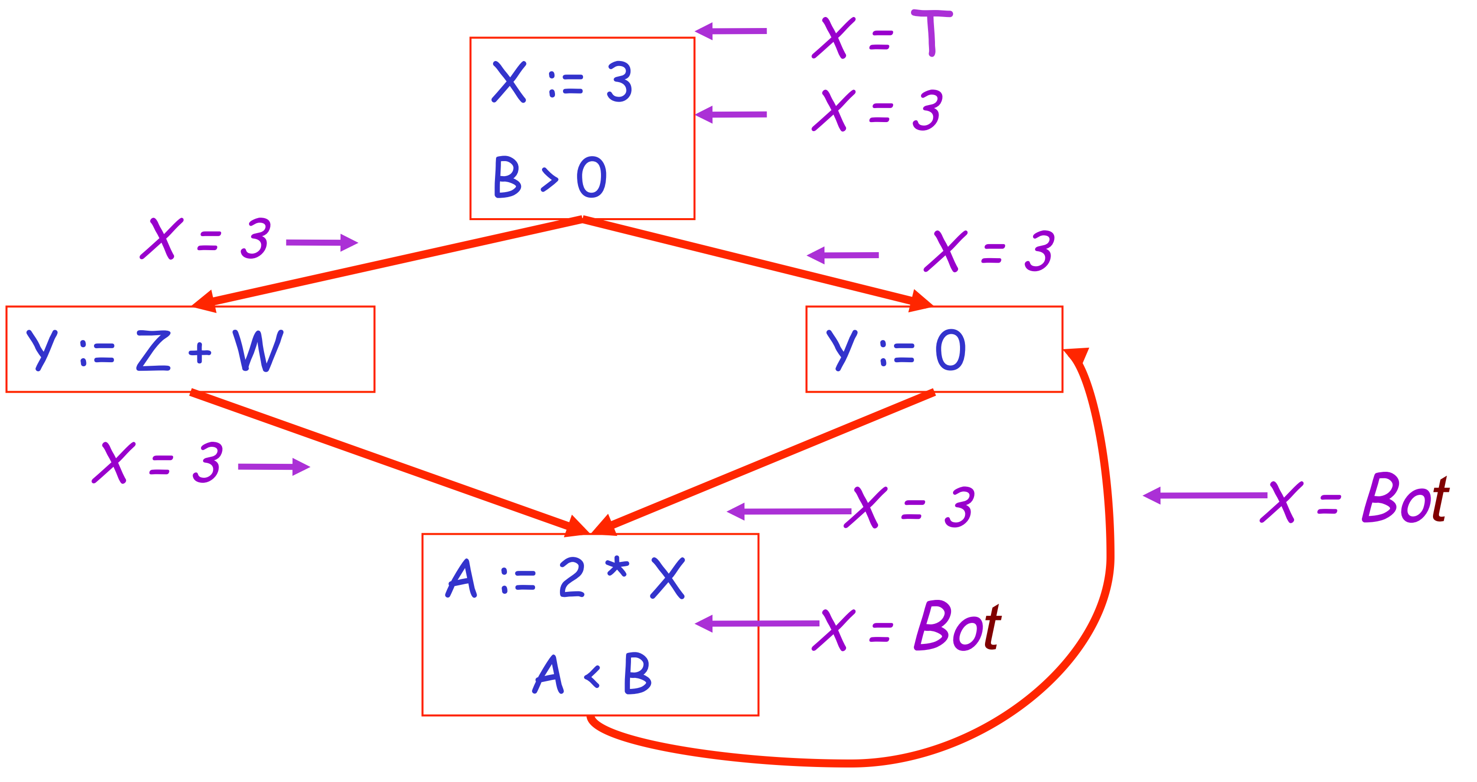


X := 3
B > 0
X = T
X = 3
X = 3
X = 3
Y := Z + W
Y := 0
X = 3
X = Bot
X = Bot
X = Bot
A := 2 * X
A < B
Remember what Bot means: this is never executed.

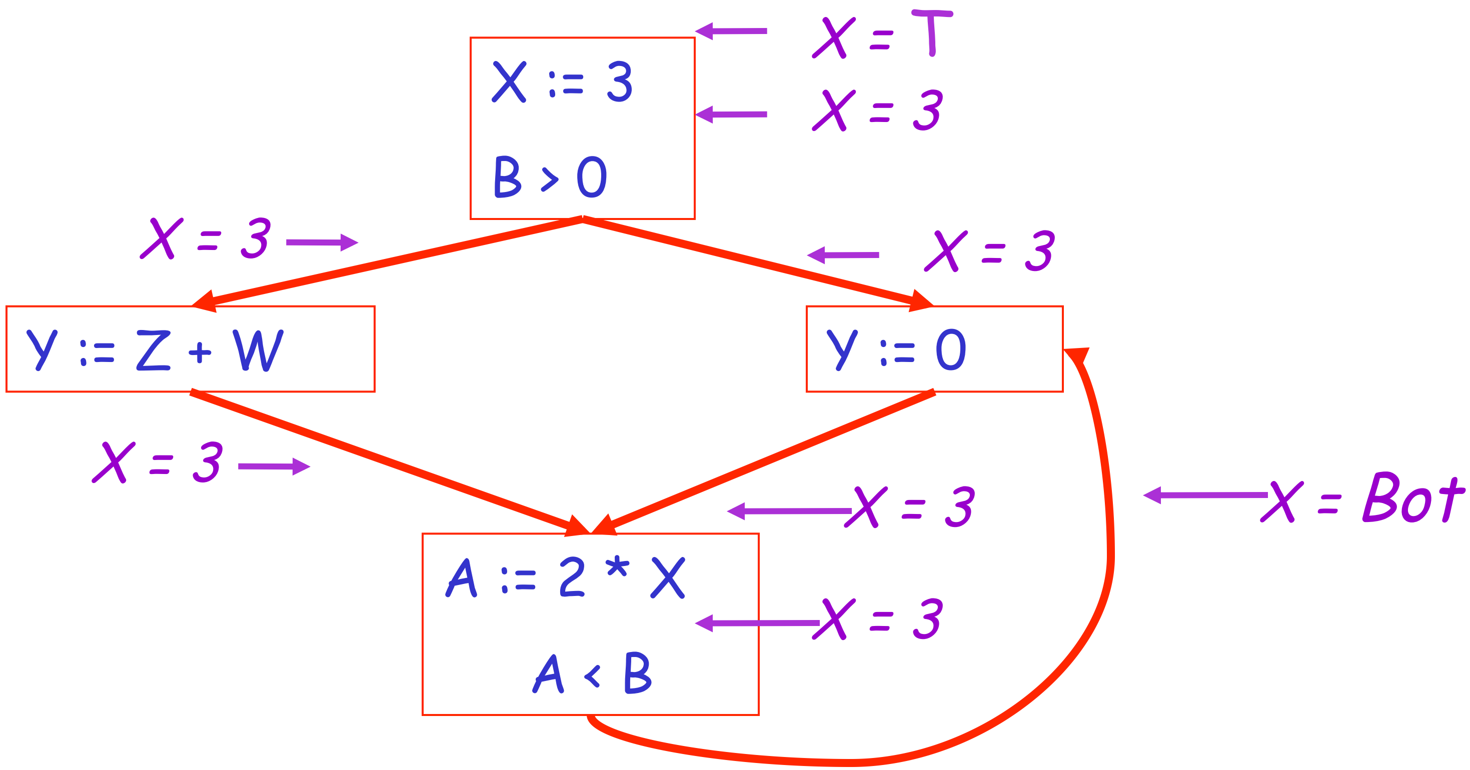
Example


X := 3
B > 0
X = T
X = 3
X = 3
X = 3
Y := Z + W
Y := 0
X = 3
X = Bot
X = Bot
A := 2 * X
A < B
X = Bot
.If it's never executed, then we can assign 3 here

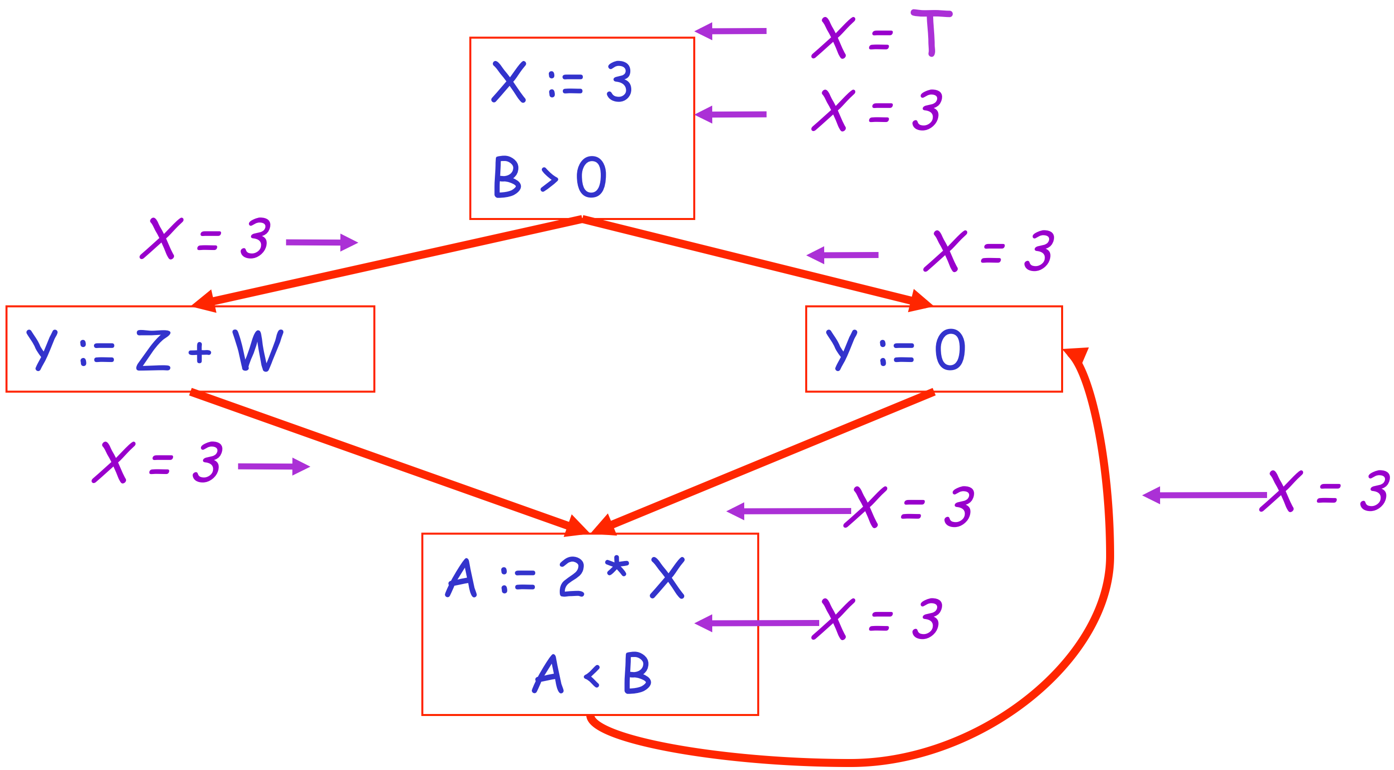
Example



Example



Example



Note at this point we need to check that everything is OK (no inconsistencies). And it turns out it is OK.

Orderings

- In the last few slides we talked about a kind of abstract computation, using elements like Bot, the constants, and T
  - And in fact, things like Bot, the constants, and T are called *abstract values,* to distinguish them from the *concrete values* (the actual run-time values that a program computes with)
  - And in fact these things are in general more abstract, since, for example, they can stand for sets of possible concrete values
    - T in particular can stand for any possible run-time value!

Orderings

- In the last few slides we talked about a kind of abstract computation, using elements like Bot, the constants, and T

- We now start to generalize those ideas a bit

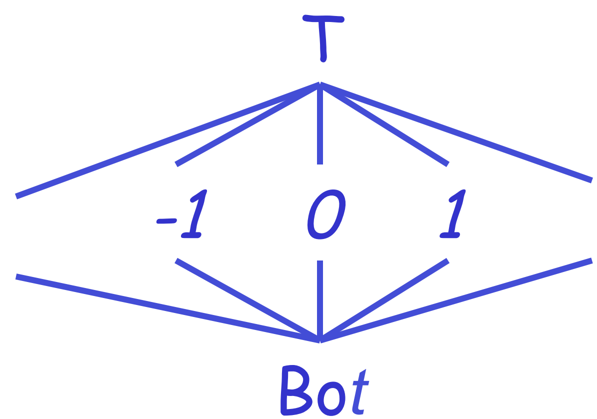
- The first step towards that generalization is to talk about orderings of those values

Orderings

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

Bot < constants < T

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



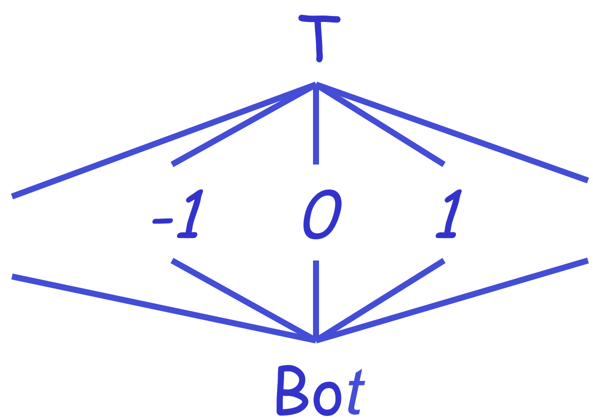
Edges between values where there is a relationship

Orderings

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

Bot < constants < T

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



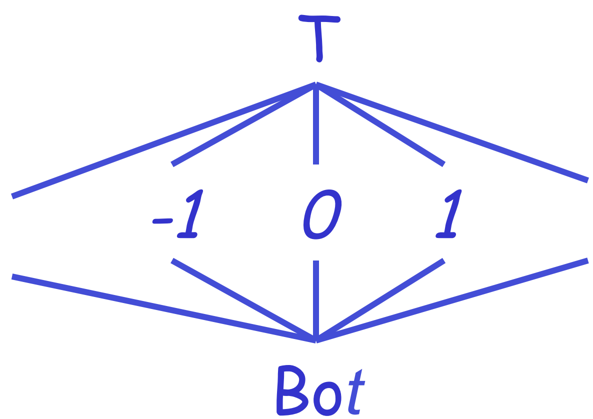
Bot less than all constants, Top greater than all constants

Orderings

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

  Bot < constants < T

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



Note in this ordering, constants are NOT comparable to each other

Orderings (Cont.)

- T 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
  - E.g., lub(Bot, 1) = 1
  - E.g., lub(1,2) = T

- 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, however, explains why the algorithm terminates
  - Values start as Bot and rules stipulate they can only *increase*
  - Bot can change to a constant, and a constant to T
  - Thus, C(s, x, _) can change at most twice

Termination (Cont.)

Thus the constant propagation algorithm is linear in program size

Number of steps =

Number of C(….) value computed * 2 =

Number of program statements * 4

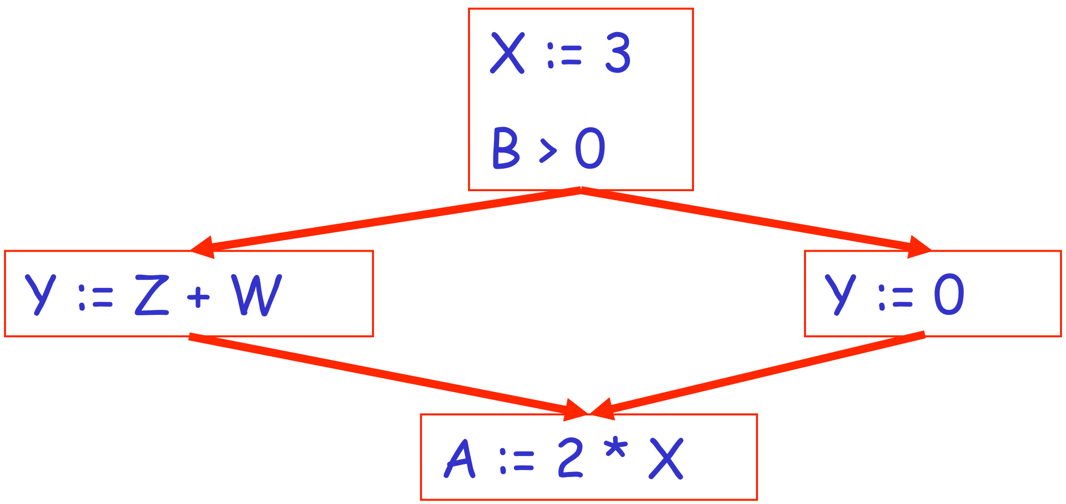
(because there is an in and out for each program statement)

Liveness Analysis

We consider now another form of global analysis: liveness analysis

Liveness Analysis

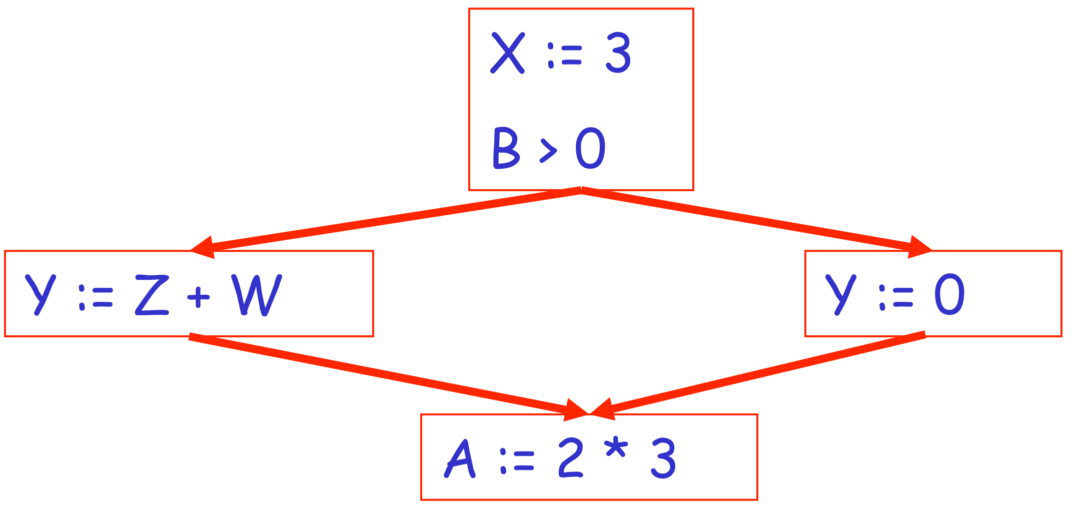
Once constants have been globally propagated, we would like to eliminate dead code



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

Liveness Analysis

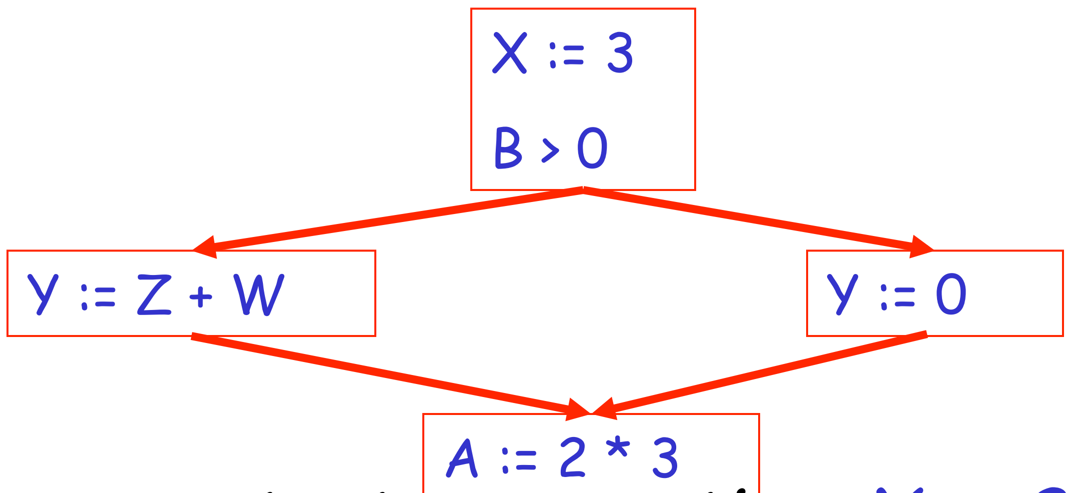
Once constants have been globally propagated, we would like to eliminate dead code



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

Liveness Analysis

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

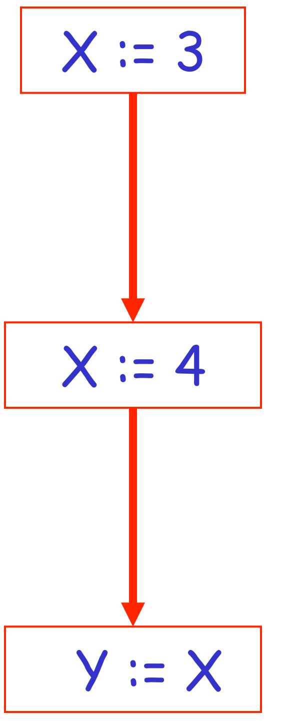


After constant propagation, X := 3 is dead (assuming X not used elsewhere)

Let's be a little more careful in what we mean by saying X is not used…

Live and Dead

- The first value of x is *dead* (never used)
  - Value 3 is overwritten before it is ever used for anything
- The second value of x is *live* (i.e., value may be used by some subsequent instruction)
  - In this particular example, it's actually guaranteed to be used
- Liveness is an important concept



Liveness (Summary)

A variable x is live at statement s if

- There exists a statement s' that uses x
  - E.g., an assignment to x
- There is a path from s to s'
- That path has no intervening assignment to x

Global Dead Code Elimination

- A assignment 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 . . .

So once again…

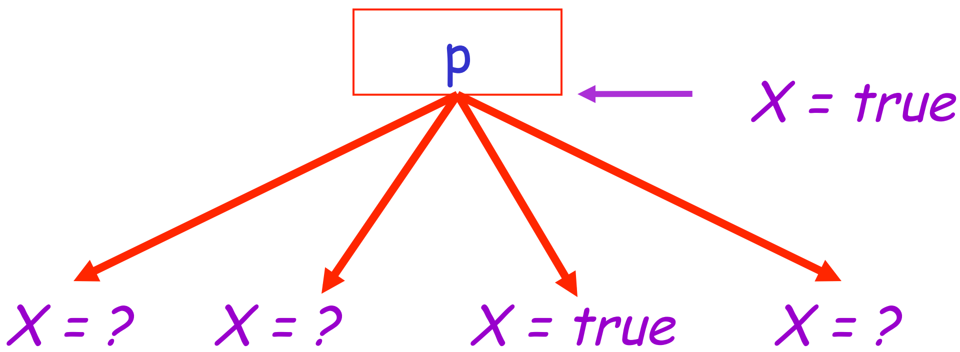
- We need global information about the control flow graph, in this case the property describing whether X will be used in the future, and we want to make that information local to a specific point in the program so that we can make a local optimization decision.
- And just like with constant propagation, we are going to define an algorithm for performing liveness analysis.
- We'll use a framework similar to what we did with constant propagation…

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

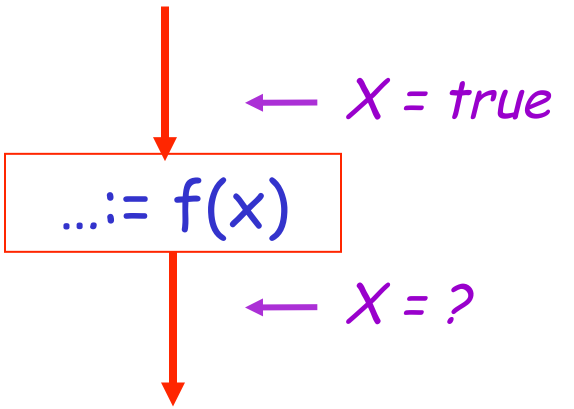


So the question: immediately after p, is X live?

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

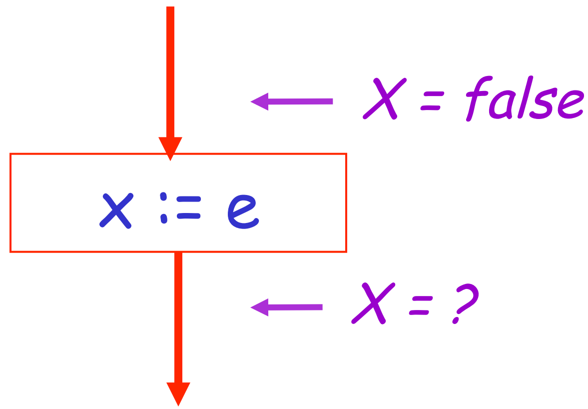
(put another way, the value of x is live right after p if the value of x is used on some path originating at p

Liveness Rule 2



L(s, x, in) = true if s refers to x (i.e., reads x) on the rhs

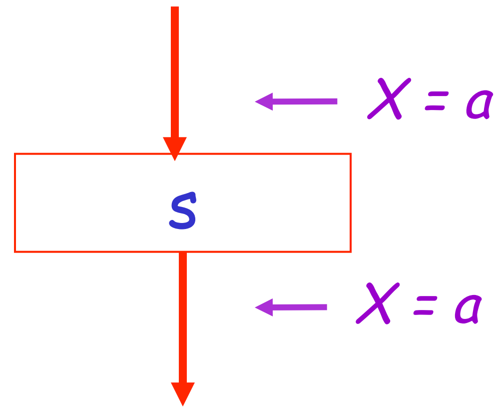
Liveness Rule 3



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

(why? well we are overwriting the value of x in the statement, so whatever value x had prior to the statement is dead)

Liveness Rule 4



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

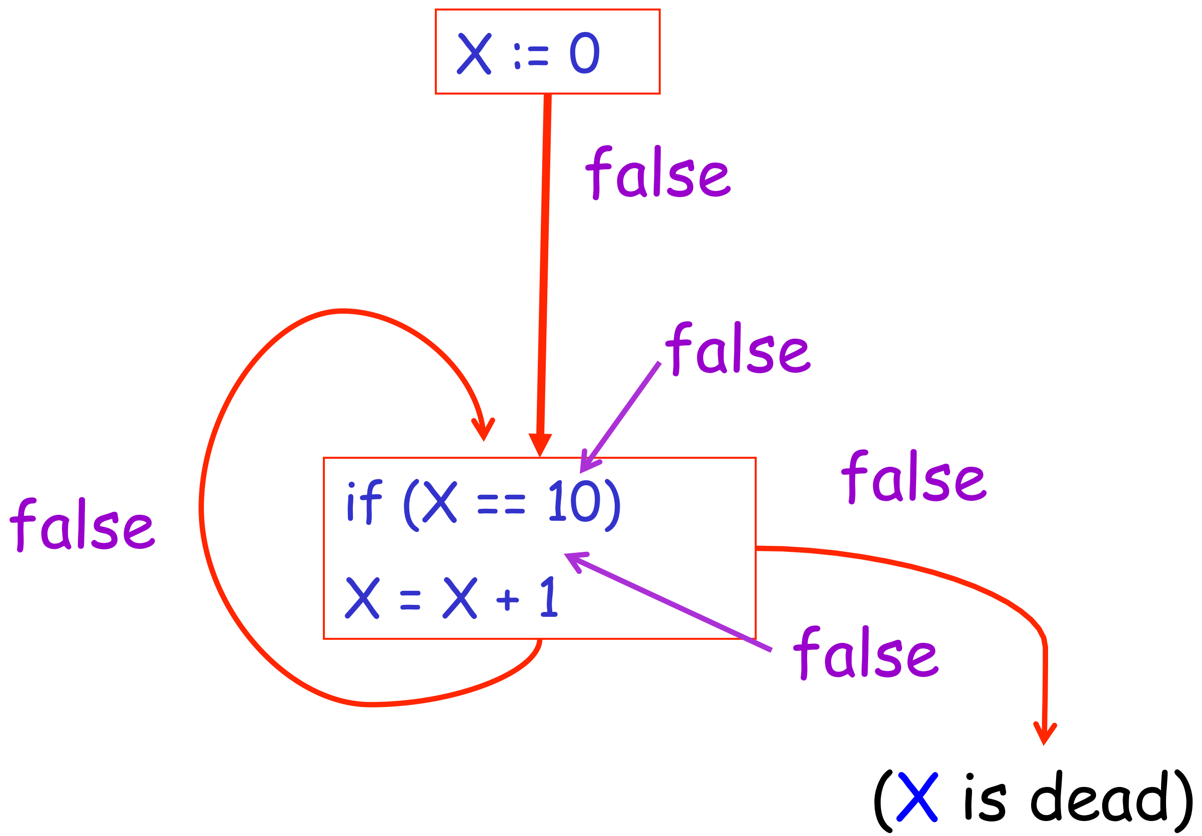
Algorithm

1. Let L(...) = false initially at all program points

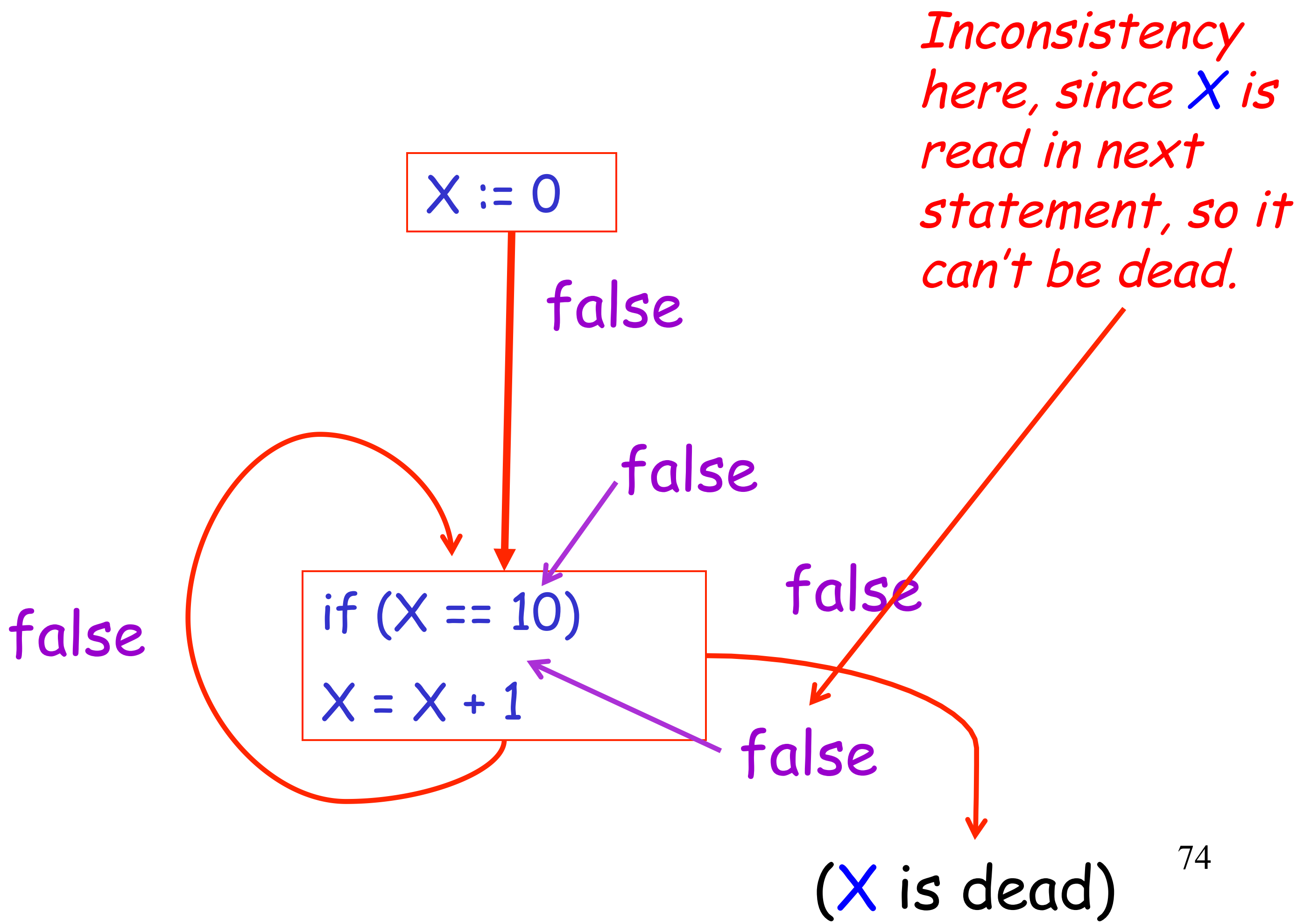
2. Repeat until all statements s satisfy rules 1-4

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

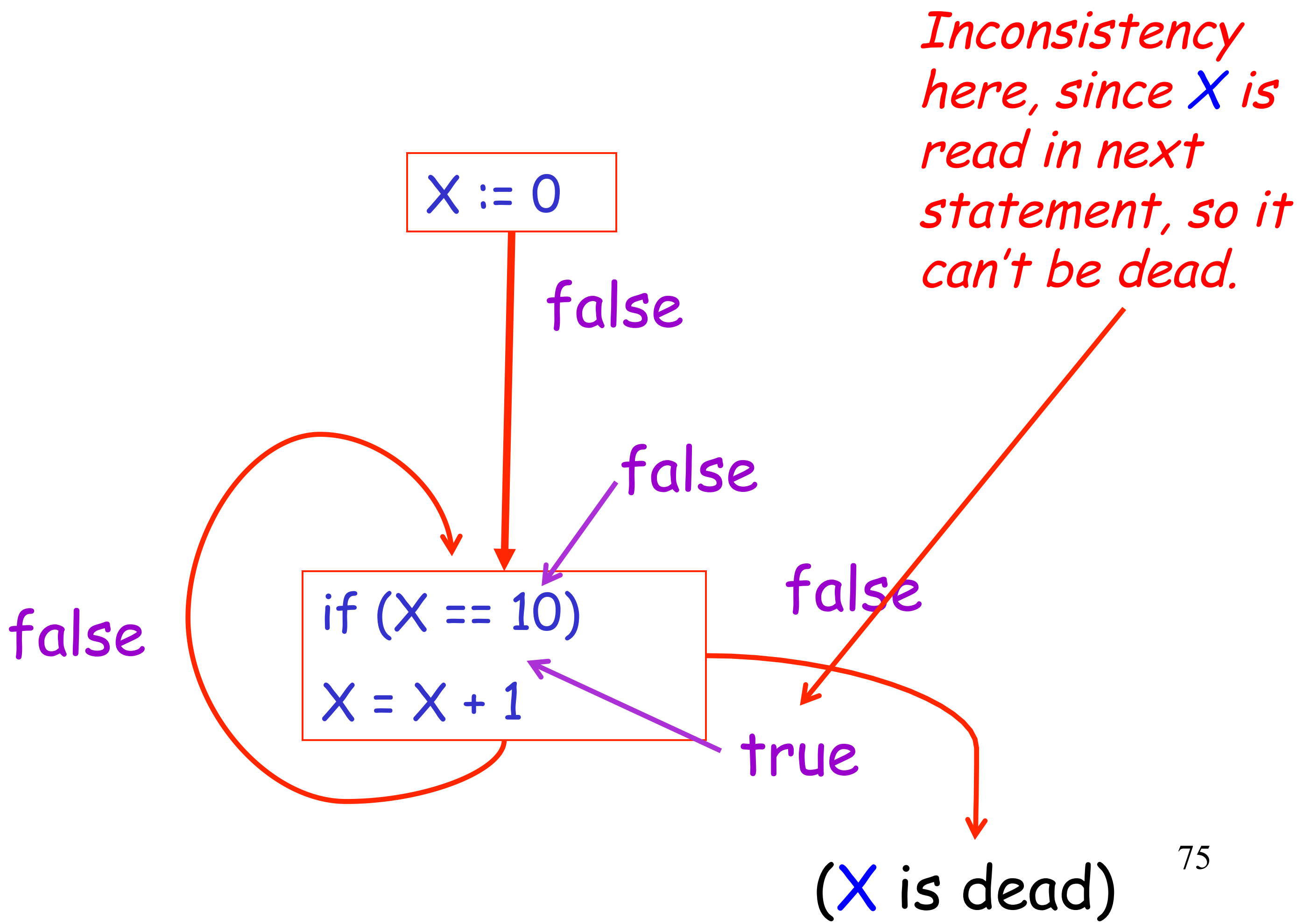
Liveness Analysis Example



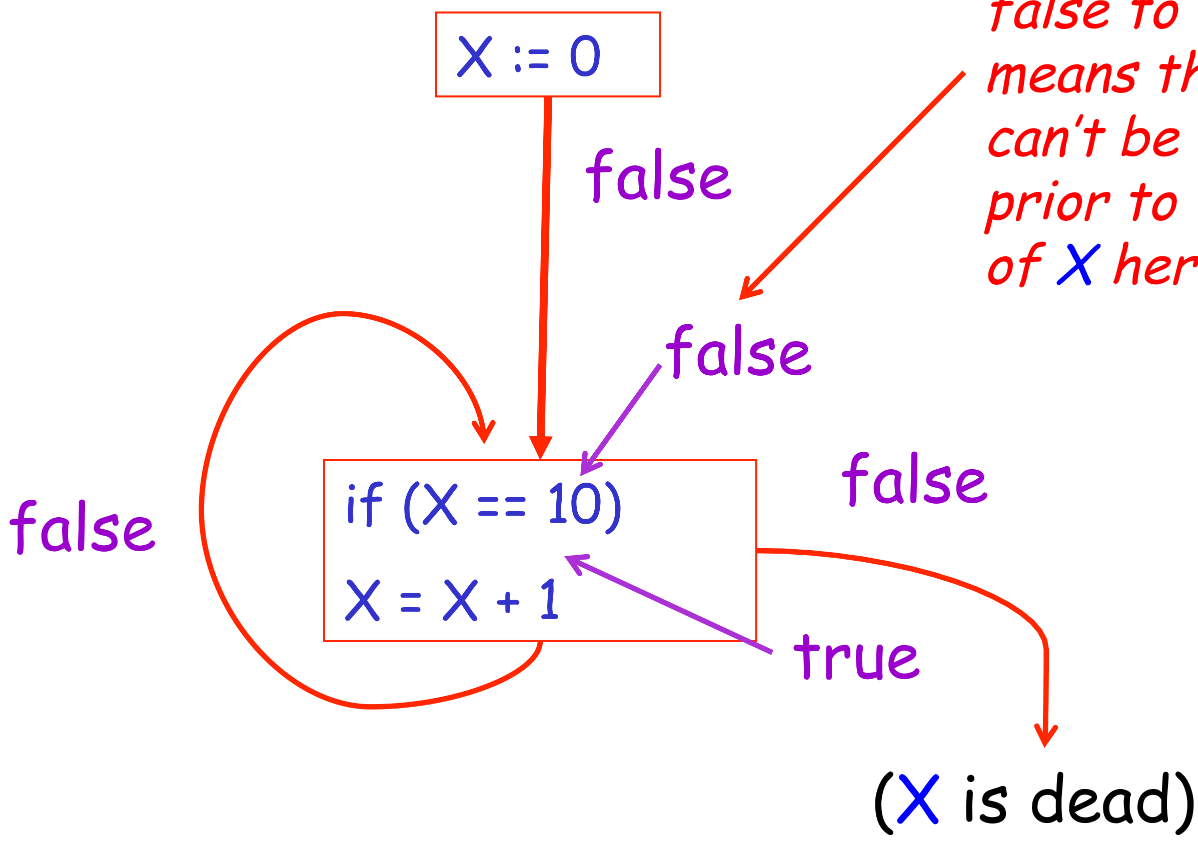
Liveness Analysis Example



Liveness Analysis Example

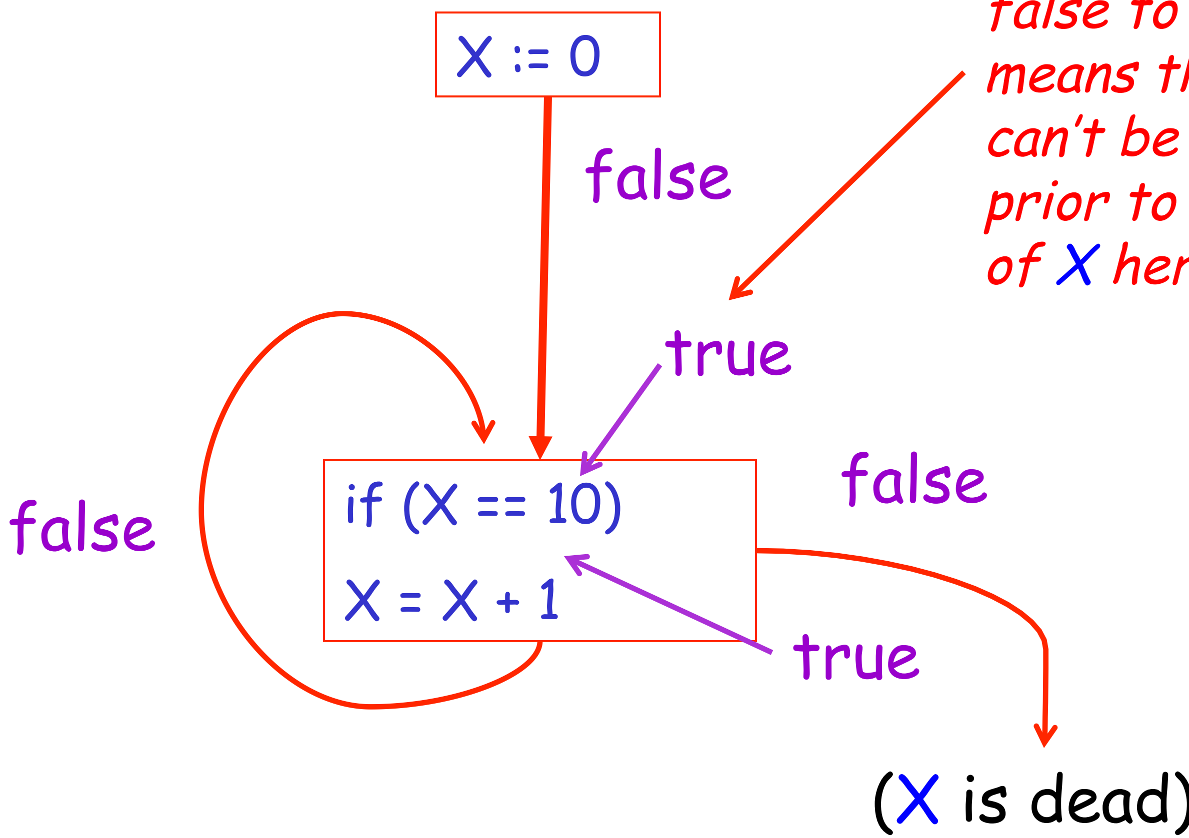


Liveness Analysis Example



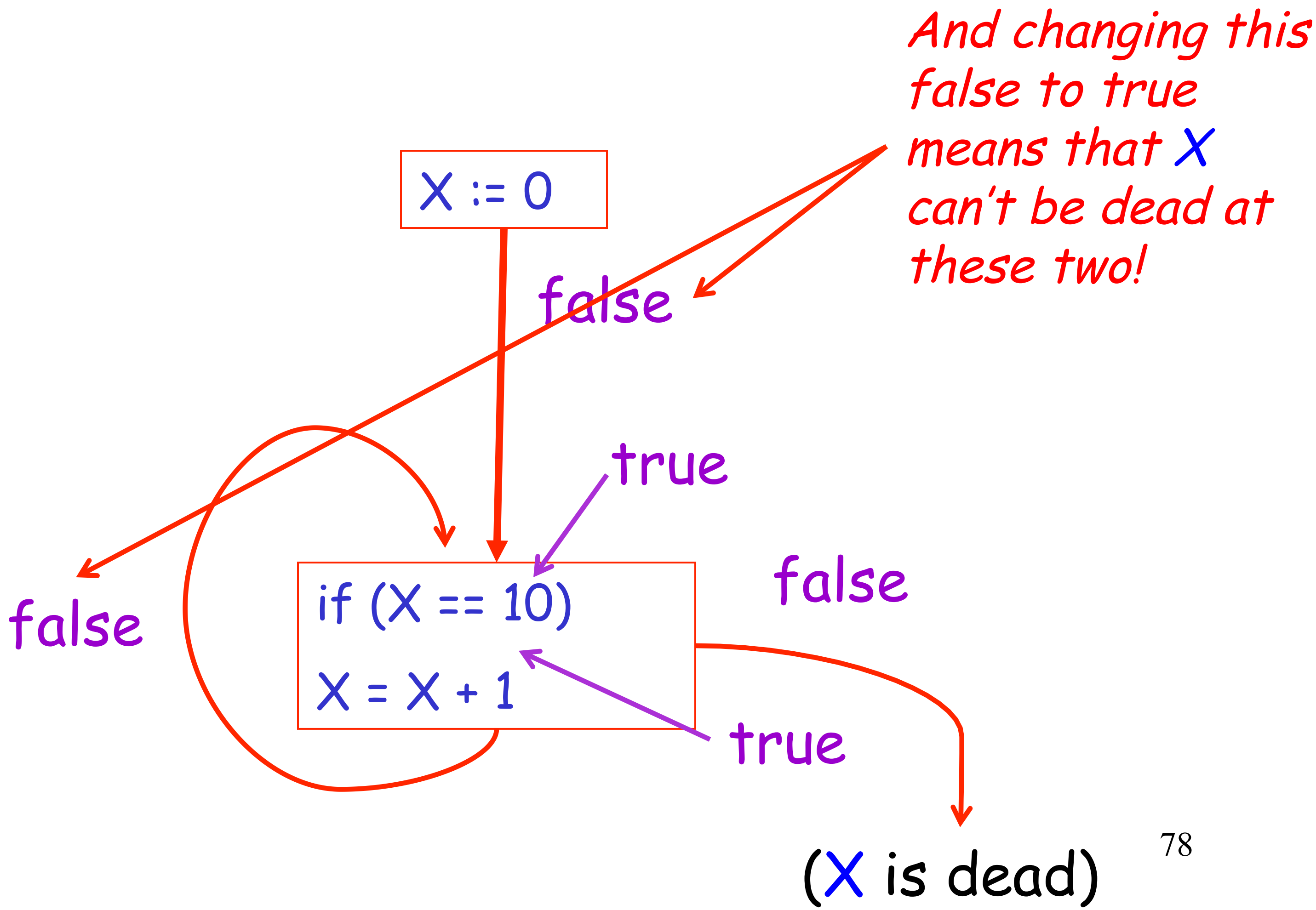
But notice that changing that false to true means that X can't be dead prior to the read of X here.

Liveness Analysis Example

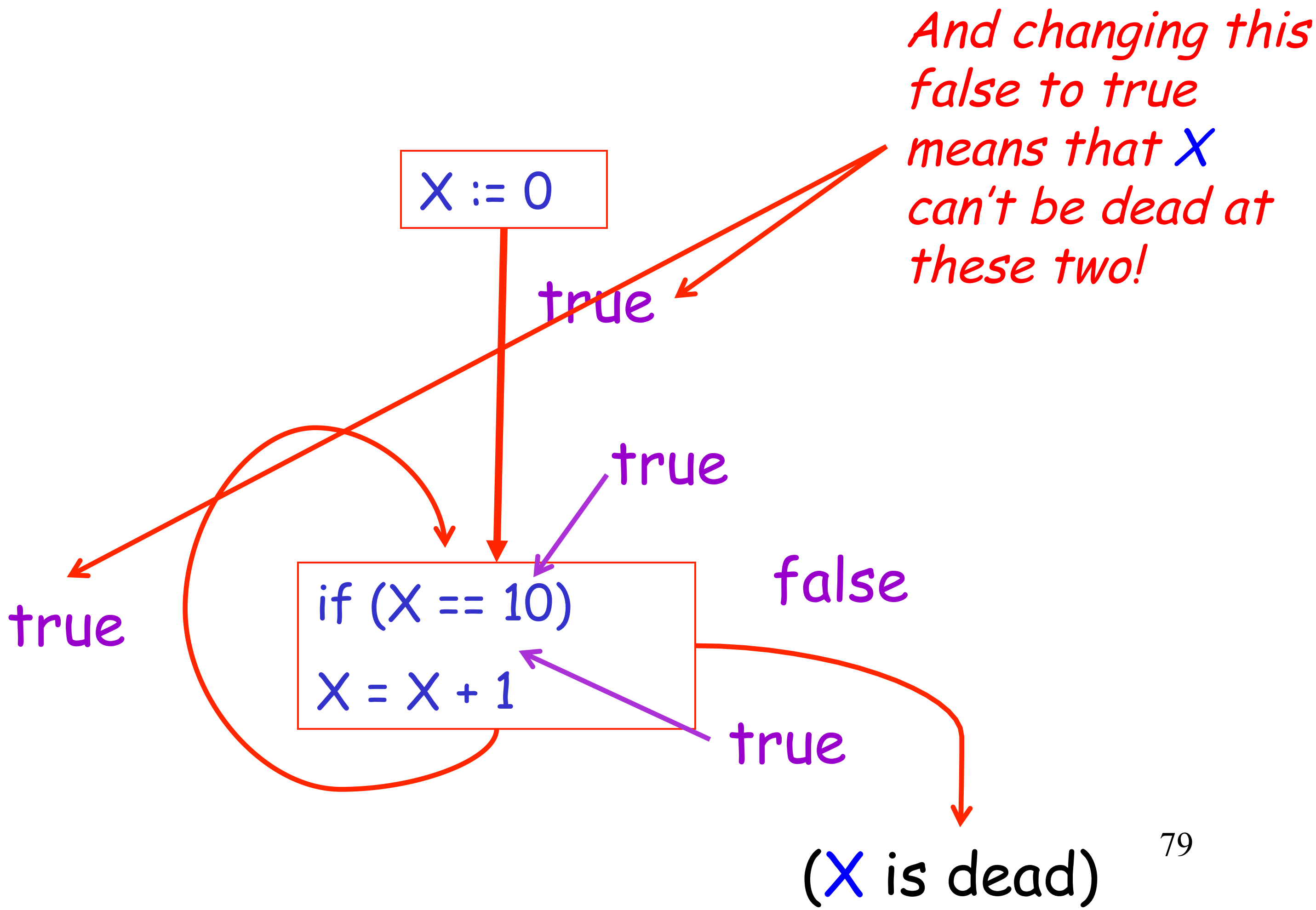


But notice that changing that false to true means that X can't be dead prior to the read of X here.

Liveness Analysis Example

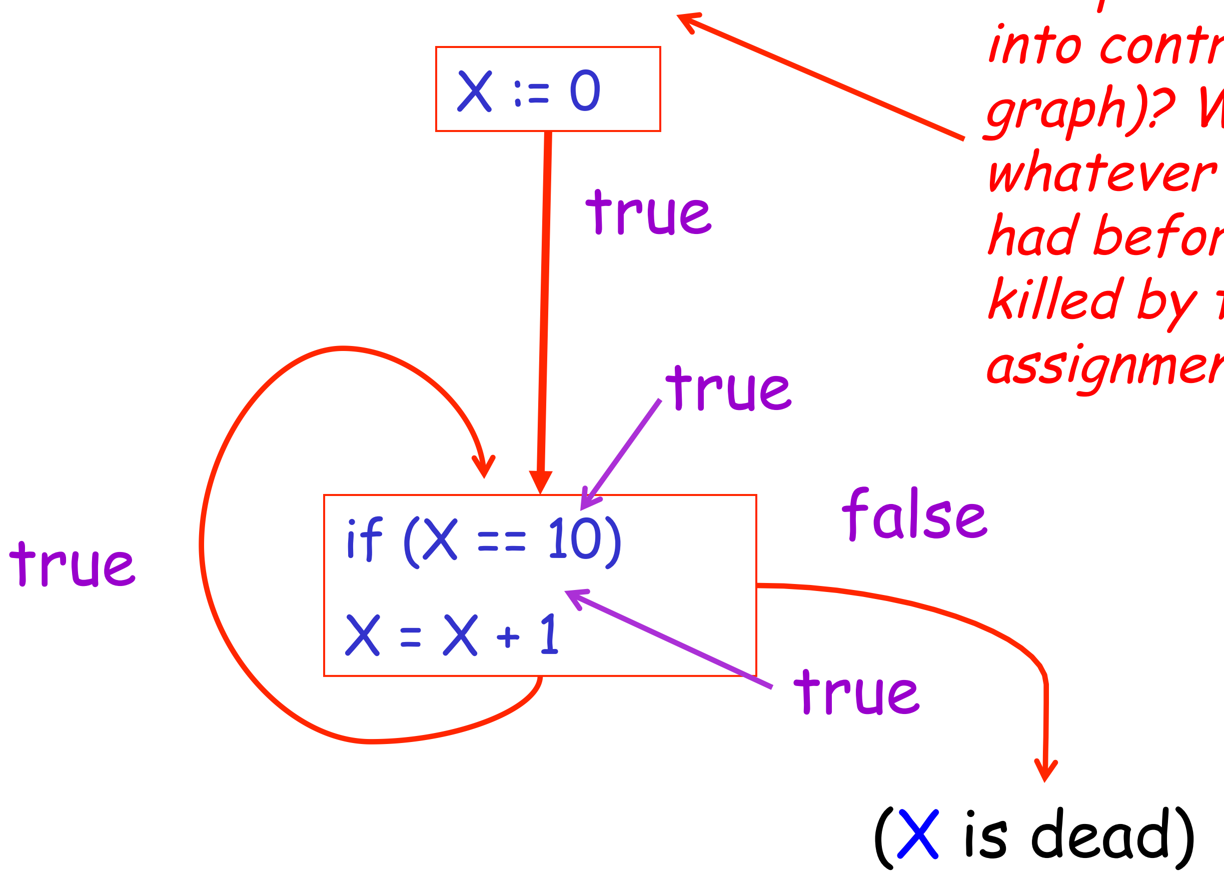


Liveness Analysis Example

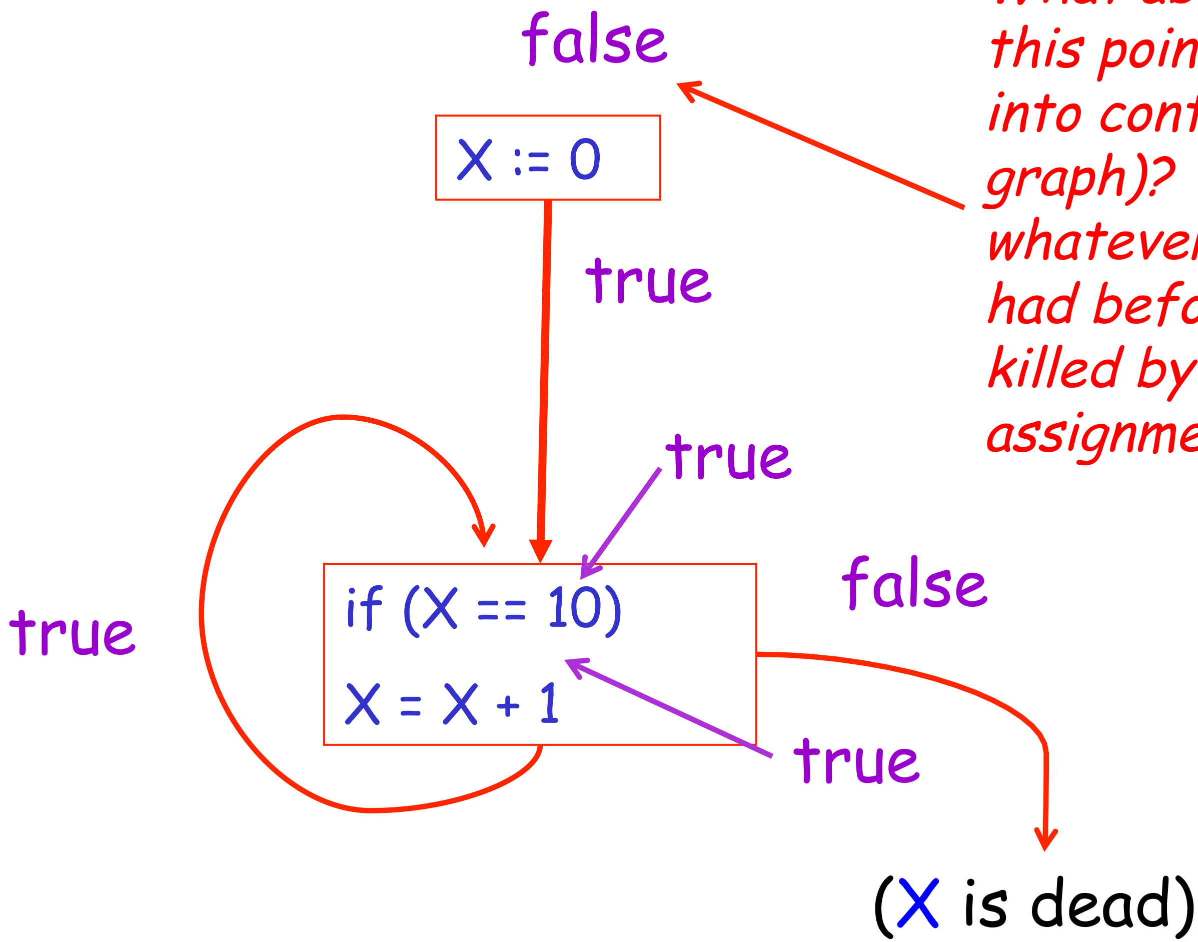


Liveness Analysis Example

What about at this point (entry into control flow graph)? Well, whatever value X had before is killed by the assignment, so…

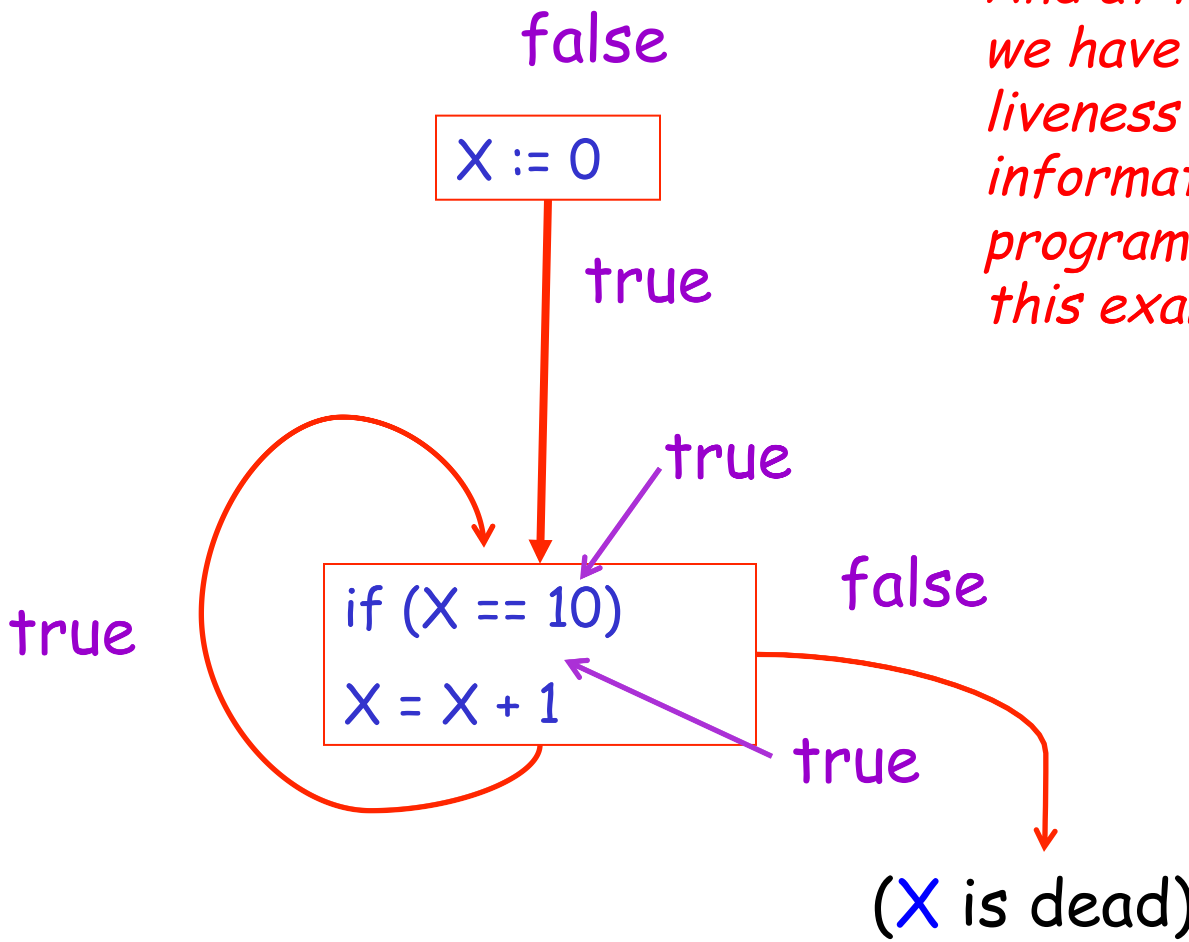


Liveness Analysis Example



What about at this point (entry into control flow graph)? Well, whatever value X had before is killed by the assignment, so…

Liveness Analysis Example



And at this point we have correct liveness information at all program points in this example

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

Termination

- A value can change from false to true, but not the other way around
  - Regarding orderings, note that we only have two values, and the only ordering is false < true
  - So everything starts at the lowest possible element of the ordering and can only go up
- 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