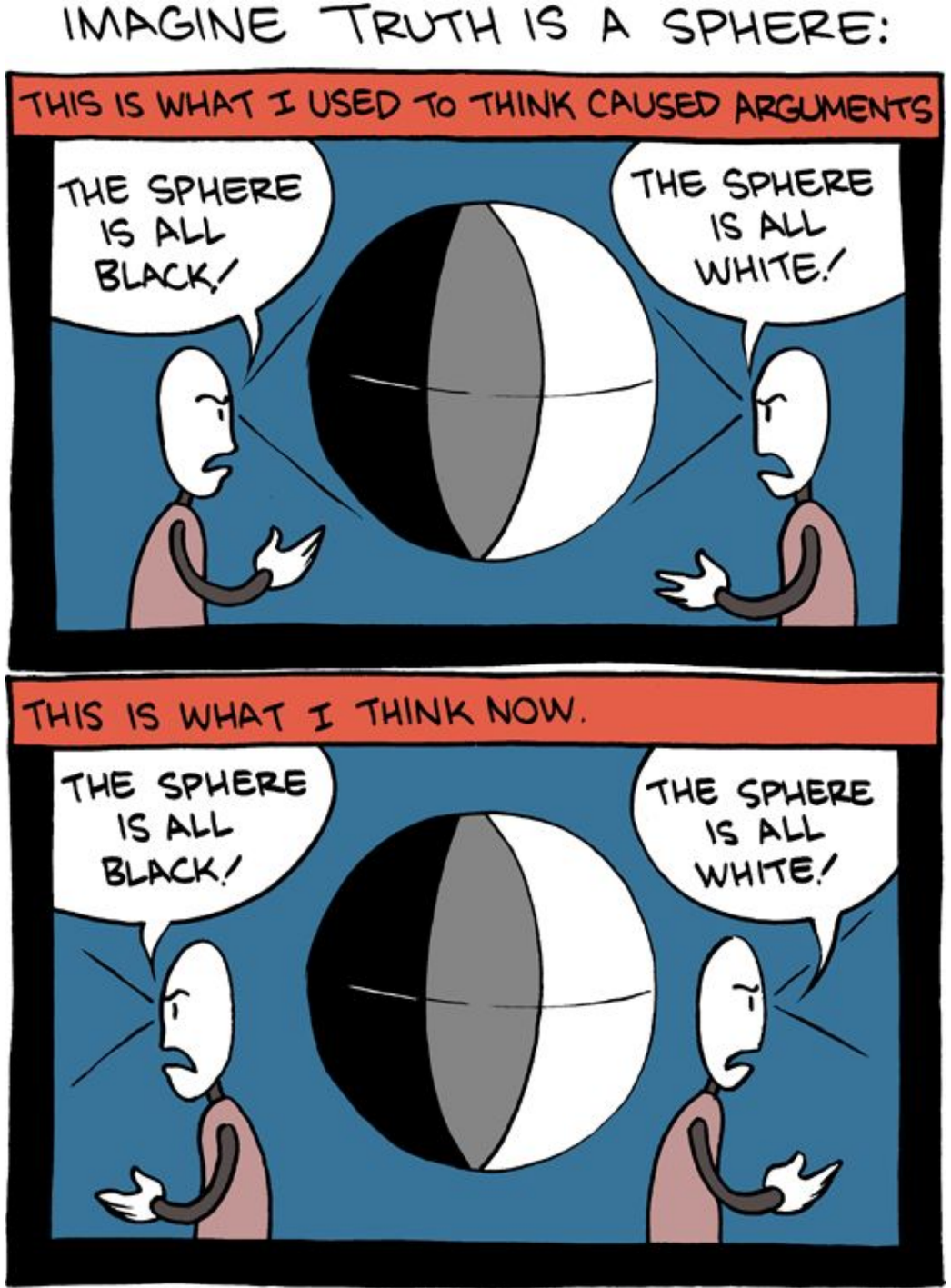


Static and Dataflow Analysis

(two-part lecture)

This week (in person + remote)

Next Mon (in person + remote)

Next Wed: Guest Lecture (Zoom only)

Feb 17 Mon	Static & Dataflow Analysis (1/2) [qa]	Note that HW3 is due before Exam 1.
Feb 19 Wed	Static & Dataflow Analysis (2/2) [qa]	
Feb 24 Mon	Fault Localization and Profiling [bugs]	HW 3 (Mutation Testing) Due
Feb 26 Wed	Guest Lecture by Natalia Sánchez Rocafort (Darktrace, UM alum!) (slides) [guest]	
Feb 28 Fri	—	Exam #1 Due (you pick a 2-hour window within this 24-hour day)

Work on HW3!

Exam 1

```
Foo(ptr, x) {
    if (x > 10) {
        deref ptr
    }
}
```

```
Foo(ptr, x, y, z, ...) {
    if (x > 10) {
        deref ptr
    }
    ...
}
```

The Story So Far …

- Quality assurance is critical to software engineering.
- Testing is the most common **dynamic** approach to QA.
  - But: race conditions, information flow, profiling …
- Code review and code inspection are common **static** approaches to QA.
- Today: **automated static analysis**

One-Slide Summary

- **Static analysis** is the systematic examination of an **abstraction** of program state space with respect to a property. Static analyses reason about all possible executions but they are **conservative**.
- **Dataflow analysis** is a popular approach to static analysis. It tracks a few broad values (“secret information” vs. “public information”) rather than exact information. It can be computed in terms of a local **transfer** of information.

Doesn't GenAI Save Us?

Research: Quantifying GitHub Copilot's impact in the enterprise with Accenture

by github copilot researchers

We conducted research with developers at Accenture to understand GitHub Copilot's real-world impact in enterprise organizations.

quality. We found that our AI pair programmer helps developers code up to 55% faster and that it made 85% of developers feel more confident in their code quality.

Can GenAI Actually Improve Developer Productivity?

Uplevel Data Labs analyzed the difference in key engineering metrics across a sample of 800 developers before and after GitHub Copilot access. The fi
from what devs report in surv

+41%
IN BUG RATE

Key Insight:

Developers with Copilot access saw a significantly higher bug rate while their issue throughput remained consistent.

This suggests that Copilot may negatively impact code quality. Engineering leaders may wish to dig deeper to find the PRs with bugs and put guardrails in place for the responsible use of generative AI.

Two Fundamental Concepts

- **Abstraction**
  - Capture semantically-relevant details
  - Elide (hide) other details
  - Handle “I don't know”: think about developers

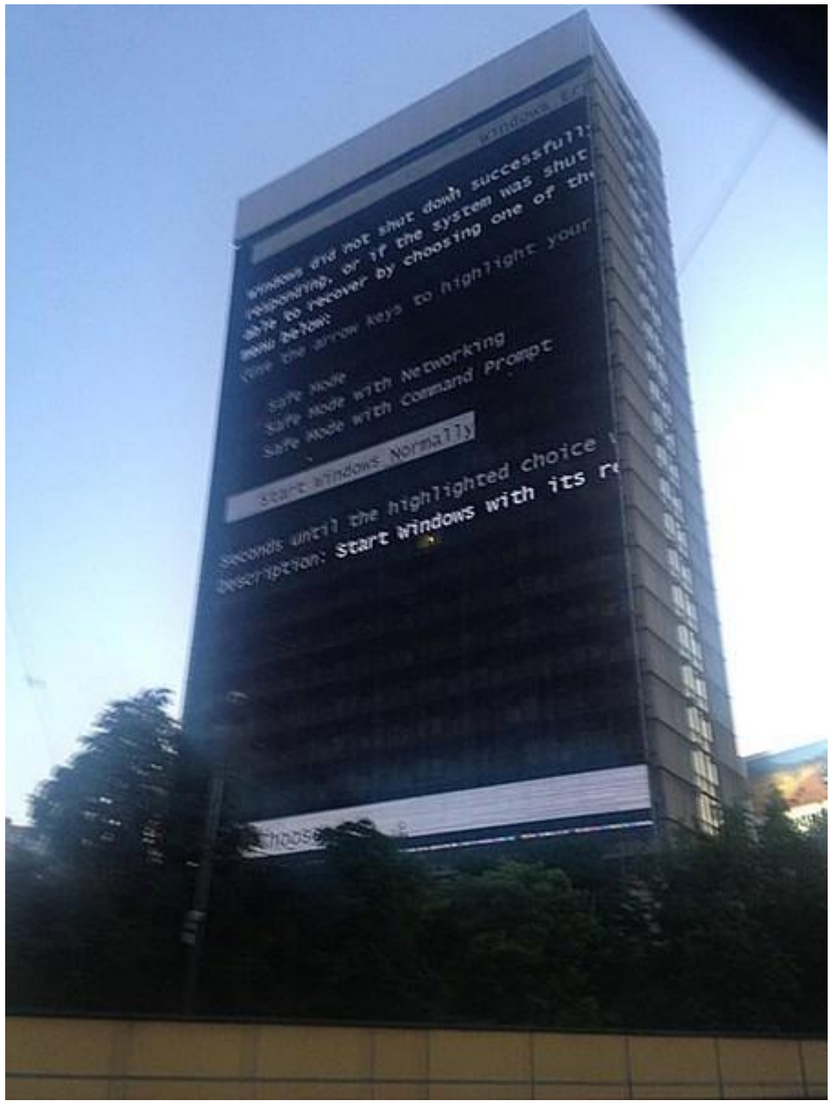
```
Foo(ptr, x, y, z, ...) {
    if (x > 10) {
        deref ptr
    }
    ...
}
```

Two Fundamental Concepts

- **Abstraction**
  - Capture semantically-relevant details
  - Elide (hide) other details
  - Handle “I don't know”: think about developers
- **Programs As Data**
  - Programs are just trees, graphs or strings
  - And we know how to analyze and manipulate those (e.g., visit every node in a graph)

goto fail;

Why care about **static** analysis?



“Unimportant” SSL (Secure Sockets Layer) Example

```
static OSStatus SSLVerifySignedServerKeyExchange(
                    SSLContext *ctx, bool isRsa,SSLBuffer signedParams,
                    uint8_t *signature,UInt16 signatureLen) {
   OSStatus err;
   ...
   if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
   if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;
   if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
   ...
fail:
   SSLFreeBuffer(&signedHashes);
   SSLFreeBuffer(&hashCtx);
   return err;}
```

How do you reason about this program?

Linux Driver Example

```
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *get_free_buffer(struct
  stripe_head * sh,int b_size) {
    struct buffer_head *bh;
    unsigned long flags;
    save_flags(flags);
    cli(); // disables interrupts
    if ((bh = sh->buffer_pool) == NULL)
      return NULL;
    sh->buffer_pool = bh -> b_next;
    bh->b_size = b_size;
    restore_flags(flags); // enables interrupts
    return bh;
}
```

How do you reason about this program?

Could We Have Found Them? (Testing? Manually?)

- How often would those bugs trigger?
- Linux example:
  - What happens if you return from a device driver with interrupts disabled?
  - Consider: that's just one function

  … in a 2,000 LOC file

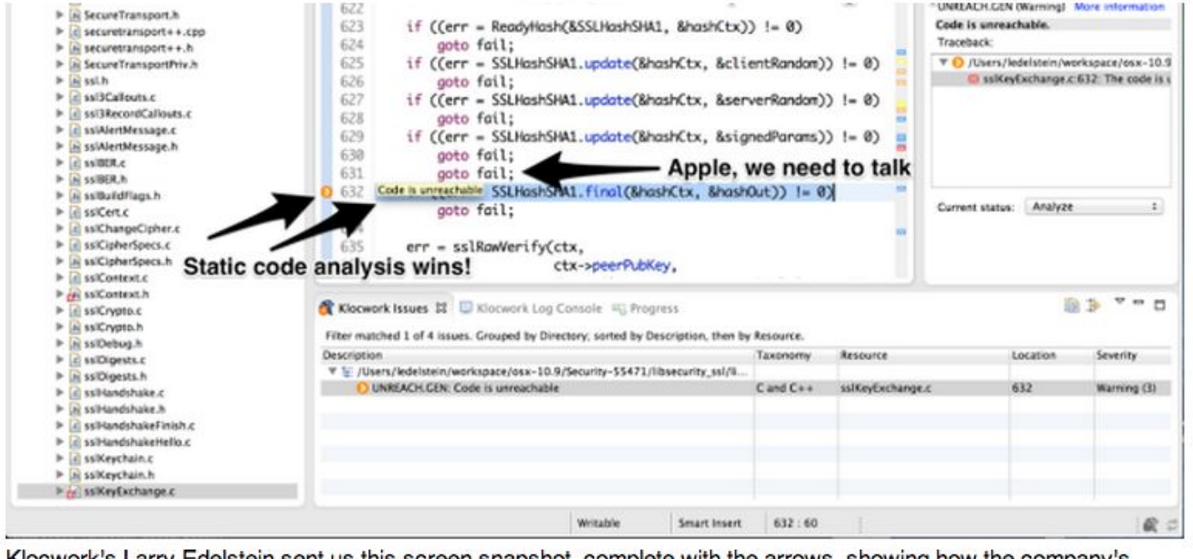
  … in a 60,000 LOC module

  … in the Linux kernel
- Some defects are very difficult to find via testing or manual inspection

Klocwork: Our source code analyzer caught Apple's 'gotofail' bug

If Apple had used a third-party source code analyzer on its encryption library, it could have avoided the "gotofail" bug.

by Declan McCullagh | February 28, 2014 1:13 PM PST



Klocwork's Larry Edelstein sent us this screen snapshot, complete with the arrows, showing how the company's product would have nabbed the "goto fail" bug.

(Credit: Klocwork)

It was a single repeated line of code -- "goto fail" -- that left millions of Apple users vulnerable to Internet attacks until the company finally fixed it Tuesday.

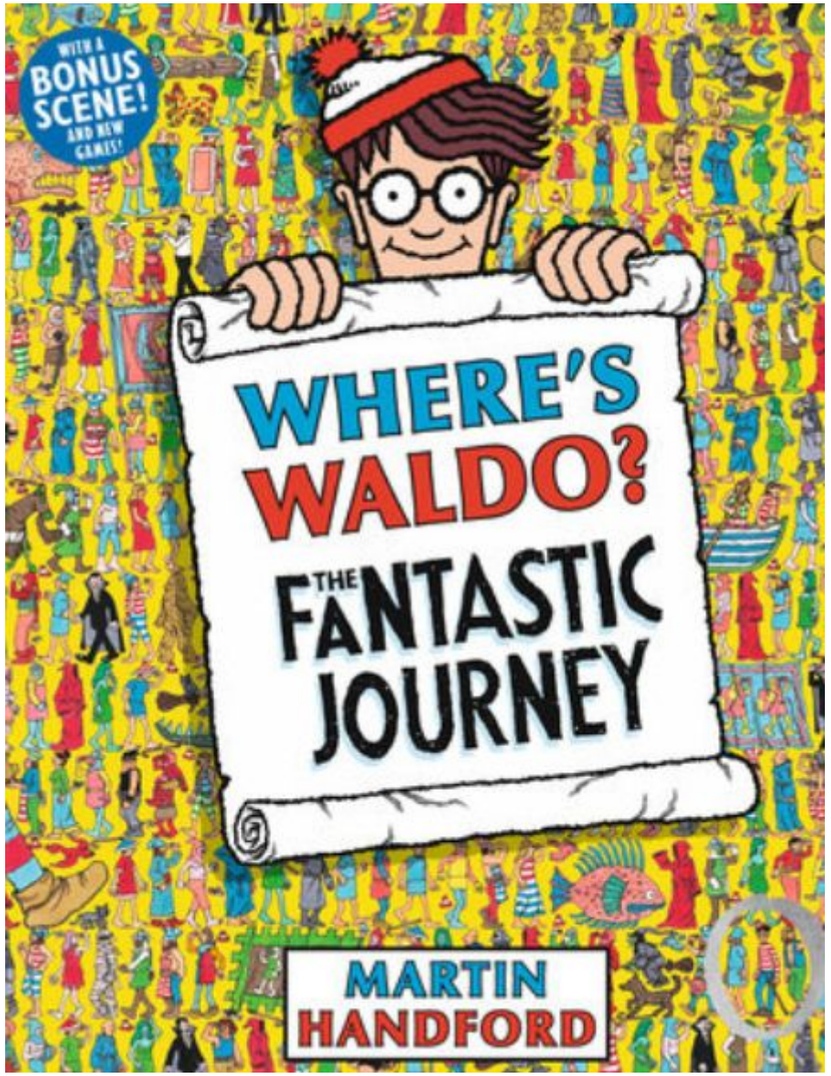
Many Interesting Defects

- … are on uncommon or difficult-to-exercise execution paths
  - Thus it is hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is not feasible
- We want to learn about "all possible runs" of the program for particular properties
  - Without actually running the program!
  - Bonus: we don't need test cases!

Static Analyses Often Focus On

- Defects that result from inconsistently following simple, mechanical design rules
  - Security: buffer overruns, input validation
  - Memory safety: null pointers, initialized data
  - Resource leaks: memory, OS resources
  - API Protocols: device drivers, GUI frameworks
  - Exceptions: arithmetic, library, user-defined
  - Encapsulation: internal data, private functions
  - Data races (again!): two threads, one variable

How And Where Should We Focus?



Static Analysis - Abstractions!

- **Static analysis** is the systematic examination of an abstraction of program state space
  - **Static analyses do not execute the program!**
- An **abstraction** is a selective representation of the program that is simpler to analyze
  - Abstractions have fewer states to explore
- Analyses check if a particular property holds
  - Liveness: “some good thing eventually happens”
  - Safety: “some bad thing never happens”

Syntactic Analysis Example

- Goal – Find every instance of this pattern:

```
public foo() {
  …
   logger.debug(“We have ” + conn + “connections.”);
}
```

```
public foo() {
  …
  if (logger.inDebug()) {
    logger.debug(“We have ” + conn + “connections.”);
  }
}
```

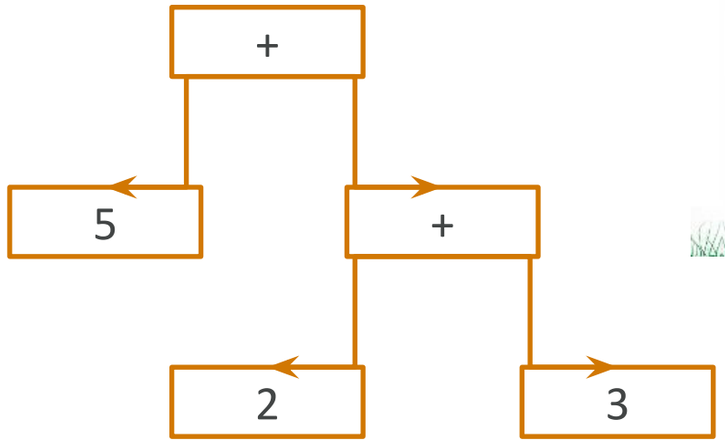
- What could go wrong? First attempt:

```
grep logger\.debug -r source_dir
```

Abstraction: Abstract Syntax Tree

- An **AST** is a tree representation of the syntactic structure of source code
  - Parsers convert concrete syntax into abstract syntax
- Records only semantically-relevant information
  - Abstracts away (, etc.
- AST captures program structure

Example: 5 + (2 + 3)



Programs As Data

- "grep" approach: treat program as string
- AST approach: treat program as tree
- The notion of treating a program as data is fundamental
  - Recall from 370: instructions are input to a CPU
    - Writing different instructions causes different execution
- It relates to the notion of a Universal Turing Machine.
  - Finite state controller and initial tape represented with a string
    - Can be placed as tape input to another TM

Dataflow Analysis

- **Dataflow analysis** is a technique for gathering information about the possible set of values calculated at various points in a program
  - We first abstract the program to an AST or CFG
  - We then abstract what we want to learn (e.g., to help developers) down to a small set of values
  - We finally give rules for computing those abstract values
    - Dataflow analyses take programs as input

Two Exemplar Analyses

- *Definite* Null Dereference
  - “Whenever execution reaches *ptr at program location L, ptr will be NULL”
- *Potential* Secure Information Leak
  - “We read in a secret string at location L, but there is a possible future public use of it”

Discussion

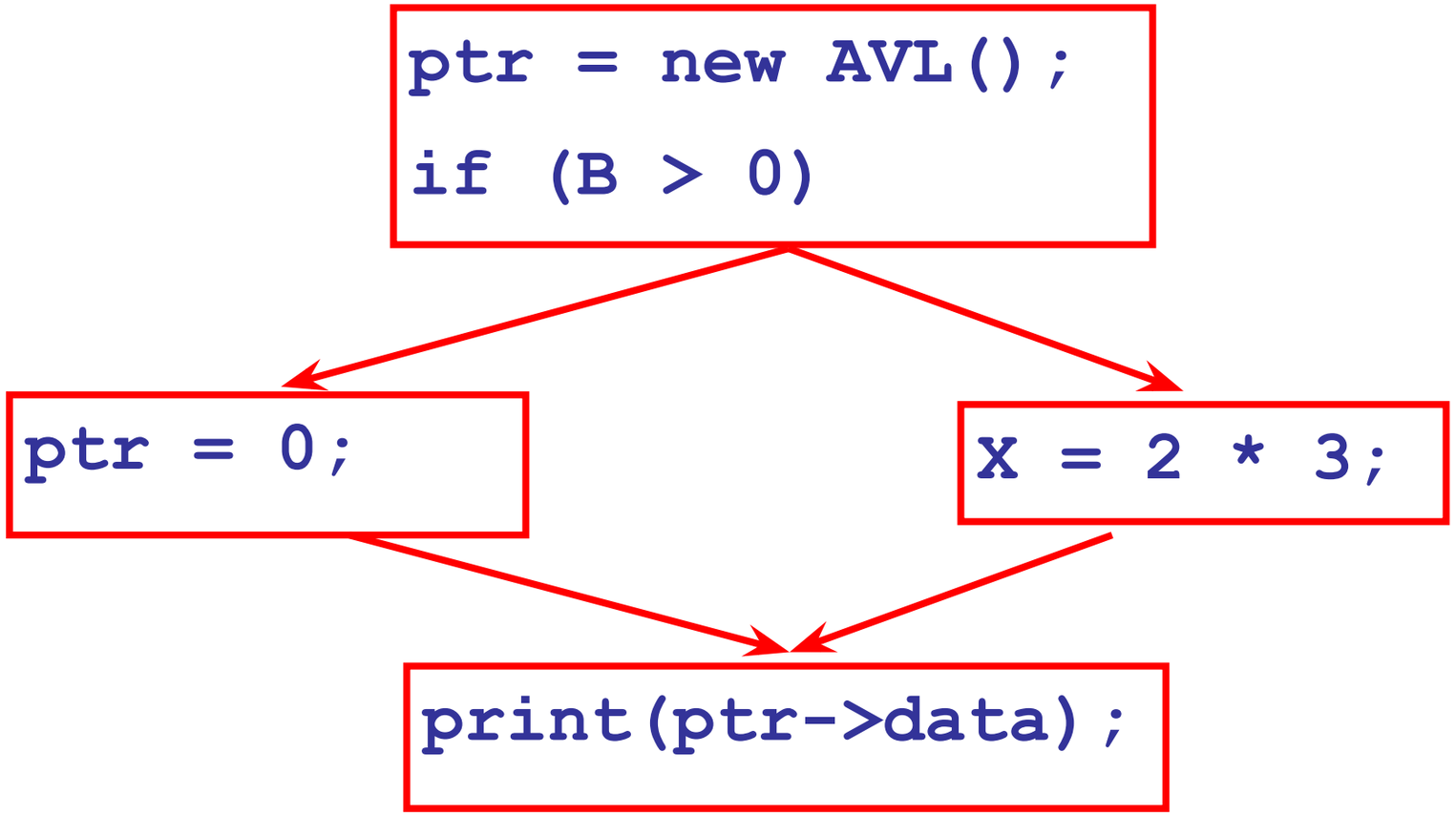
- These analyses are not trivial

- "Whenever execution reaches" → "**all paths**" → includes paths around loops and through branches of conditionals

- We will use **(global) dataflow analysis** to learn about the program
  - Global = an analysis of the entire method body, not just one { block }

Analysis Example

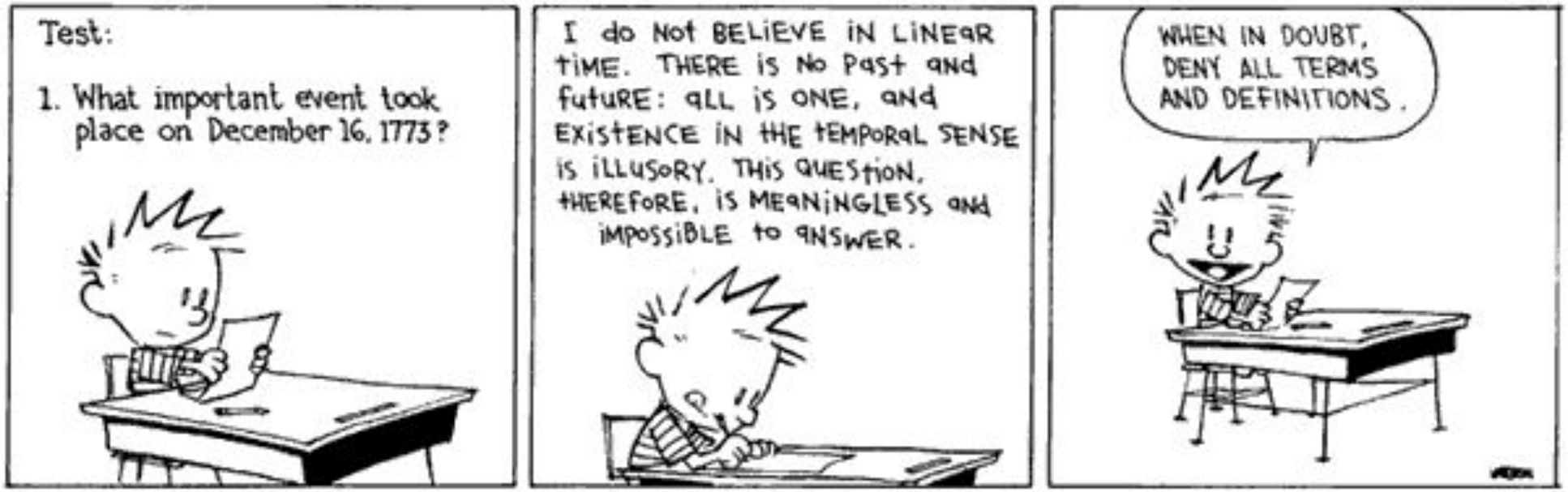
- Is **ptr** *always* null when it is dereferenced?



Correctness

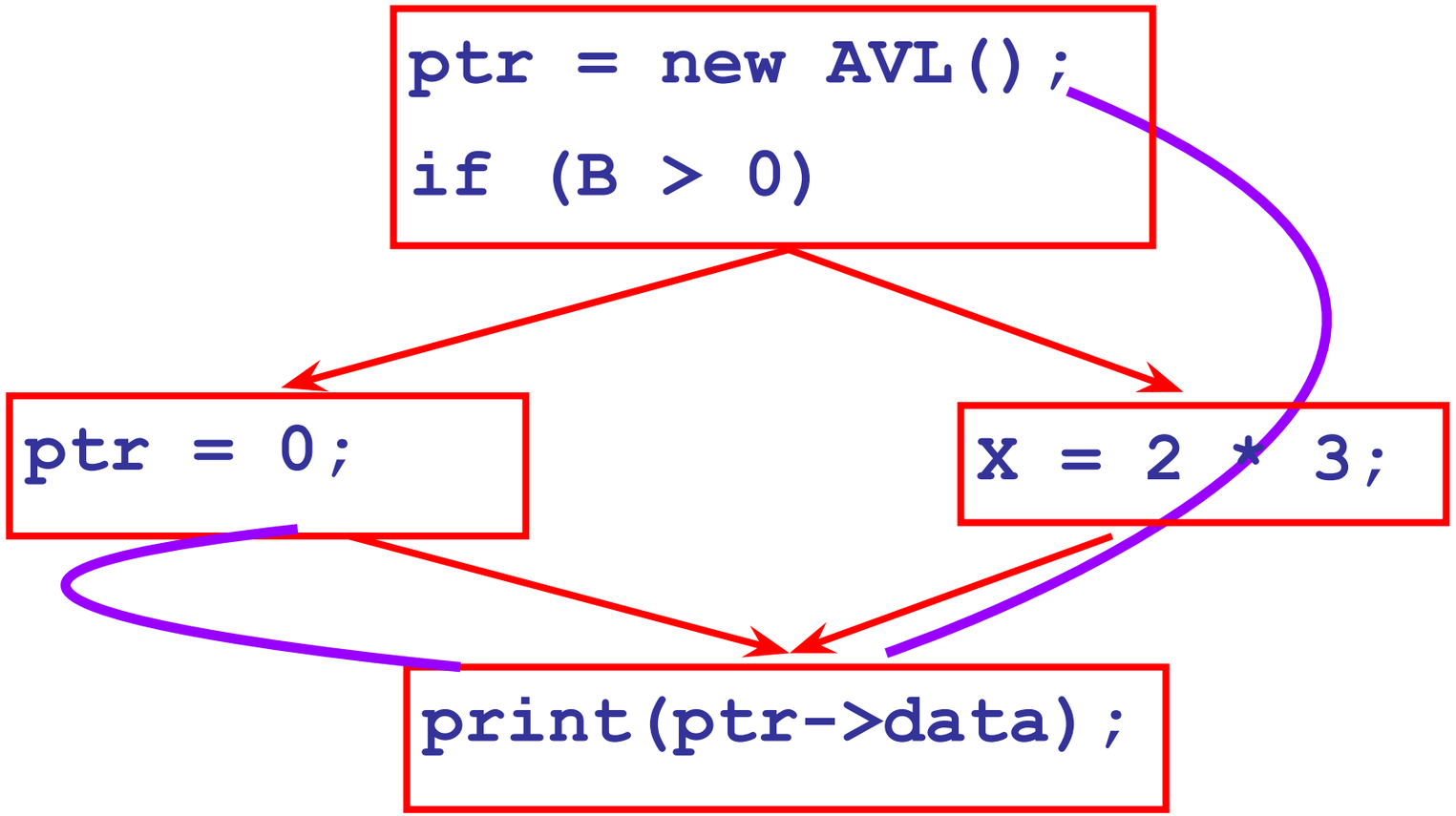
- To determine that a use of x is **always** null, we must know this **correctness condition**:

On every path to the use of x,
the last assignment to x is x := 0 **



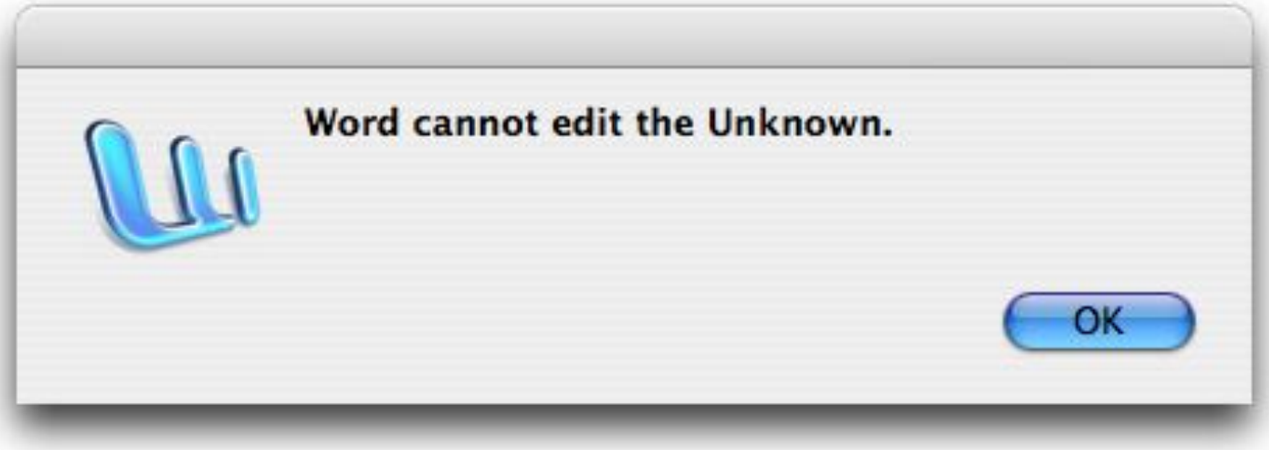
Analysis Example Revisited

- Is **ptr** *always* null when it is dereferenced?



Static Dataflow Analysis

- Static dataflow analyses share several traits:
  - Knowing a given property P (at particular program points)
  - Proving P at any point requires knowledge of the entire method body
  - **Property P is typically *undecidable*!**



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
    - Oops: We can now solve the halting problem.
    - Take function H and find out if it halts by testing function F(x) = { H(x); return 1; } to see if it has a positive result
    - *Contradiction!*

Undecidability of Program Properties

- So, if *interesting* properties are out, what can we do?

- Syntactic properties are decidable!
  - e.g., How many occurrences of "x" are there?

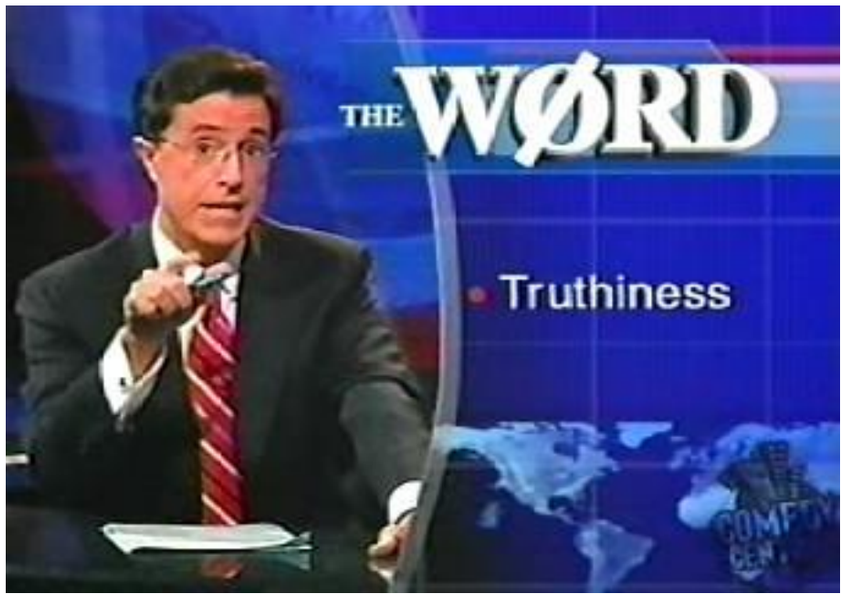
- Programs without looping are also decidable!

Looping

- Almost every important program has a loop
  - … loop bound is based on user input
- An algorithm always terminates
- So a dataflow analysis algorithm must terminate even if the input program loops (forever)
- But how to reason about all loop iterations?
  - Suppose you dereference the null pointer on the 500th iteration but we only analyze 499 iterations
  - One source of **imprecision**

Conservative Program Analyses

- We cannot tell for sure that **ptr** is always null
  - So how can we carry out any sort of analysis?
- It is OK to be **conservative**. If the goal is to check whether or not P is true, then (conservative) analysis reports either
  - P is definitely true
  - Don't know if P is true



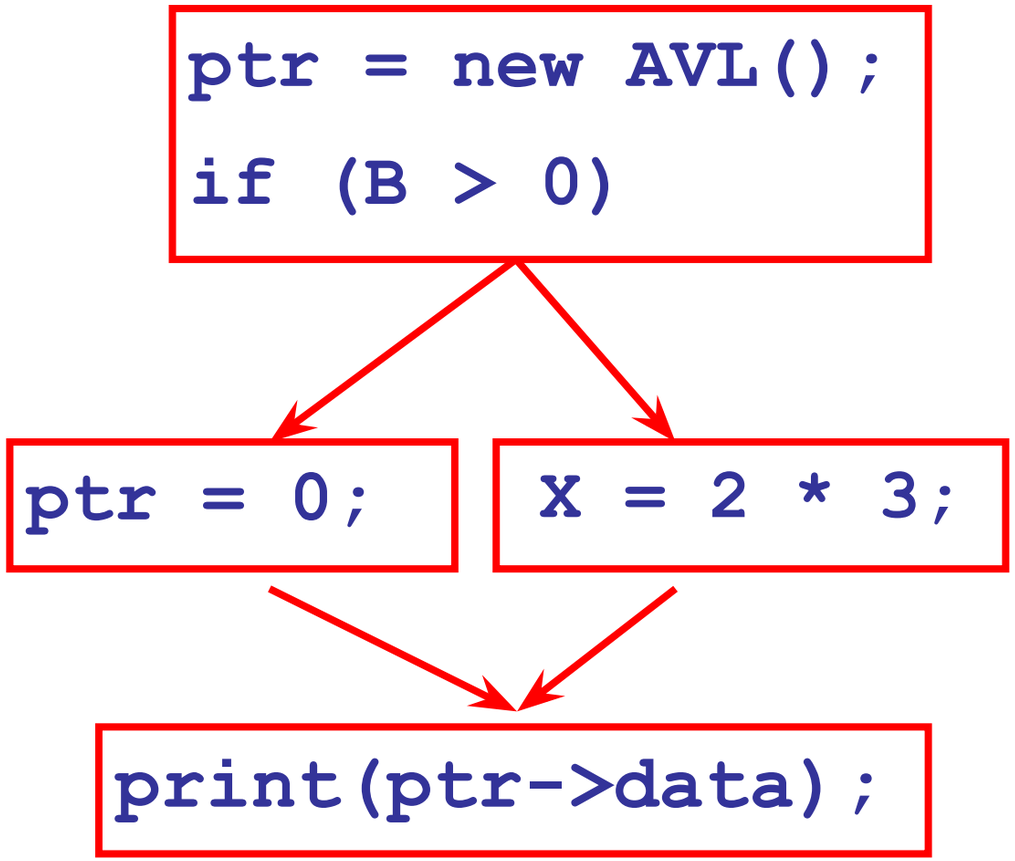
Conservative Program Analyses

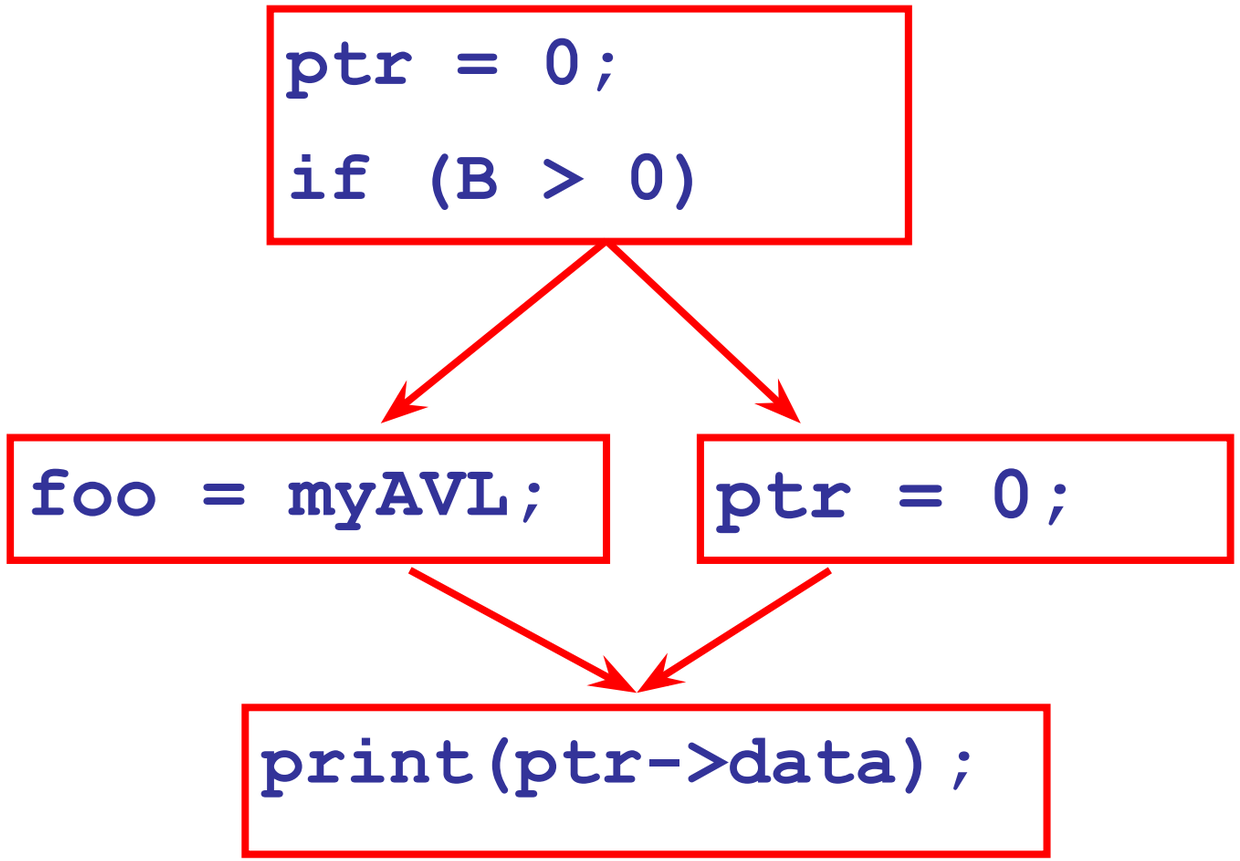
- It is always correct to say “don’t know”
  - We try to say don’t know as rarely as possible
- All program analyses are conservative

- Must think about your software engineering process
  - Bug finding analysis for developers?
    They hate “false positives”, so if we don't know, stay silent.
  - Bug finding analysis for airplane autopilot?
    Safety is critical, so if we don't know, give a warning.

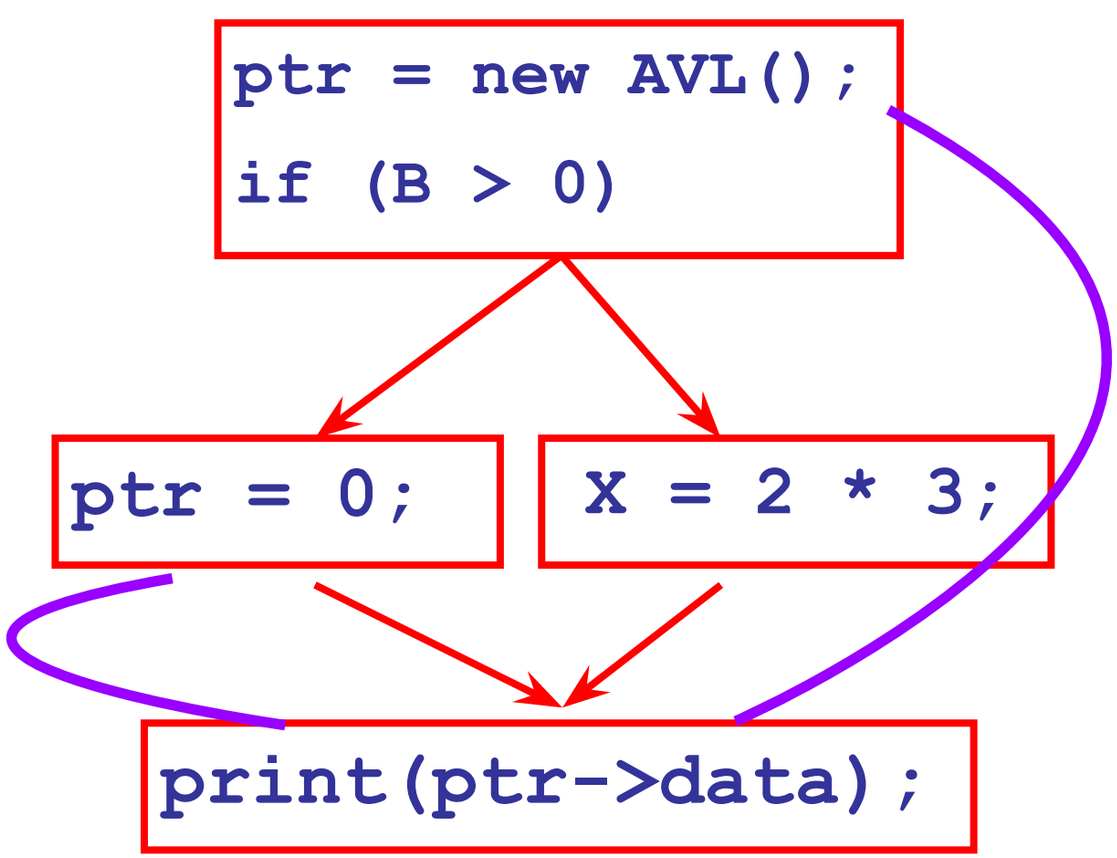
Definitely Null Analysis (Quiz)

- Is **ptr** *always* null when it is dereferenced?

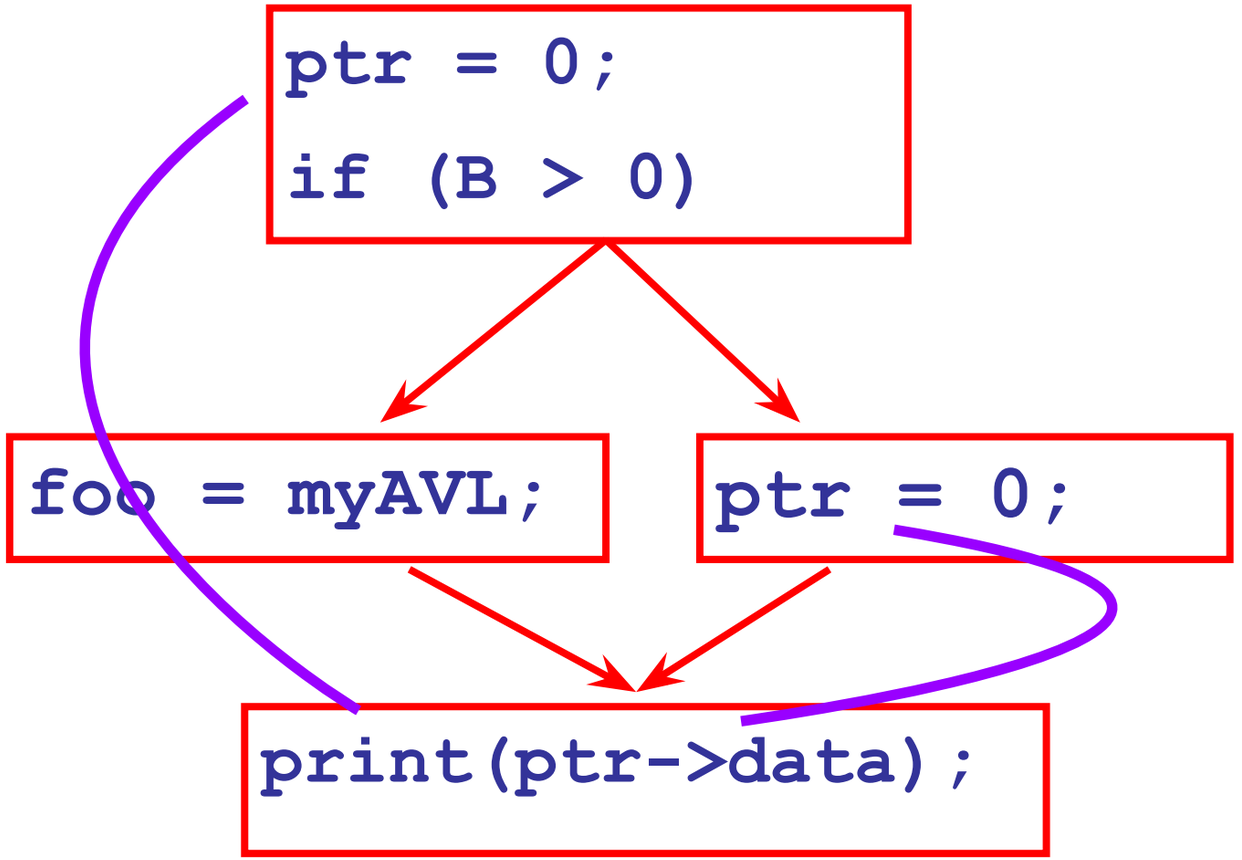




Definitely Null Analysis



No, not always.



Yes, always.

On every path to the use of ptr, the last assignment to ptr is ptr := 0 **

Definitely Null Information

- We can warn about definitely null pointers at any point where ** holds
  - … by computing ** for a single variable ptr at all program points

On every path to the use of ptr, the last assignment to ptr is ptr := 0 **

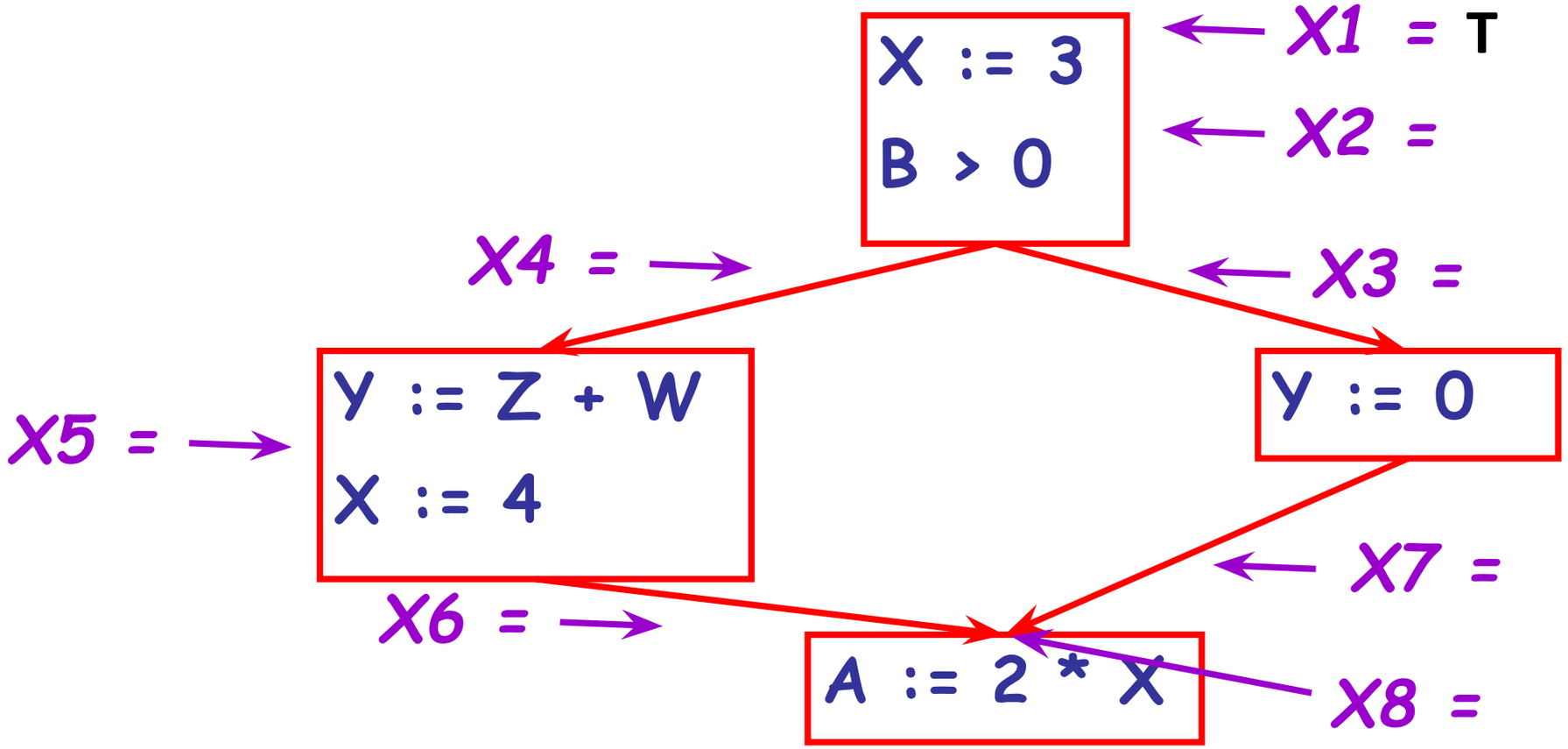
Definitely Null Analysis (Cont.)

- To define the problem, we associate one of the following values with ptr *at every program point*
  - Recall: abstraction and property

value	*interpretation*
⊥ **(called *Bottom*)**	This statement is not reachable
c	X = constant c
⊤ **(called *Top*)**	Don't know if X is a constant

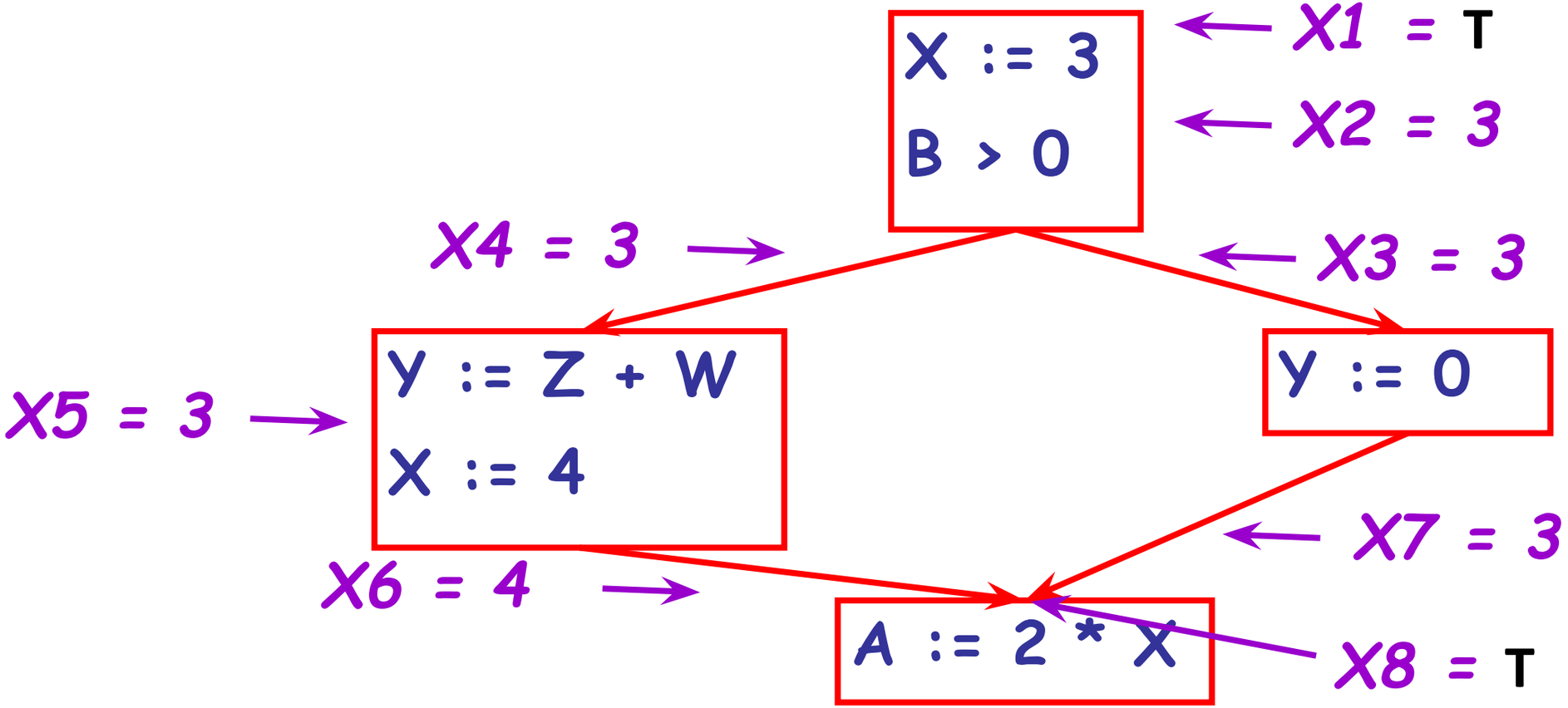
Example

Fill in these blanks



Recall: ⊥ = not reachable, c = constant, ⊤ = don't know.

Example



Recall: ⊥ = not reachable, c = constant, ⊤ = don't know.

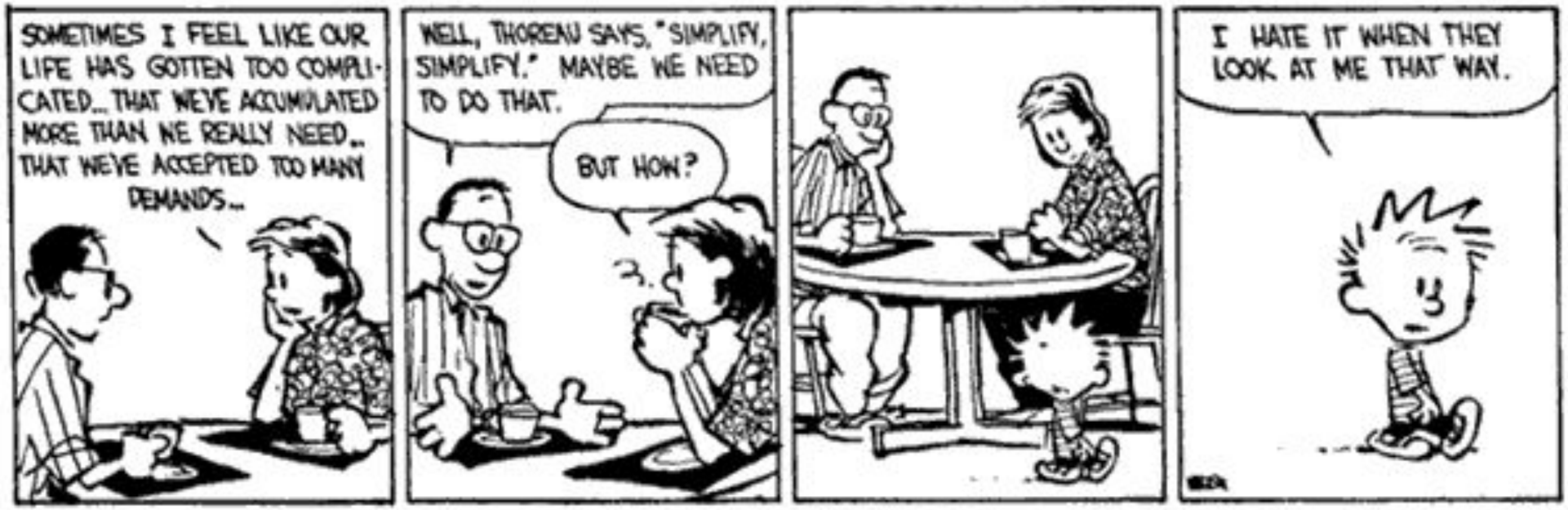
Using Abstract Information

- Given analysis information (and a policy about false positives/negatives), it is easy to decide whether or not to issue a warning
  - Simply inspect the x = ? associated with a statement using x
  - If x is the constant **0** at that point, issue a warning!

- **Big question: how can an algorithm compute 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

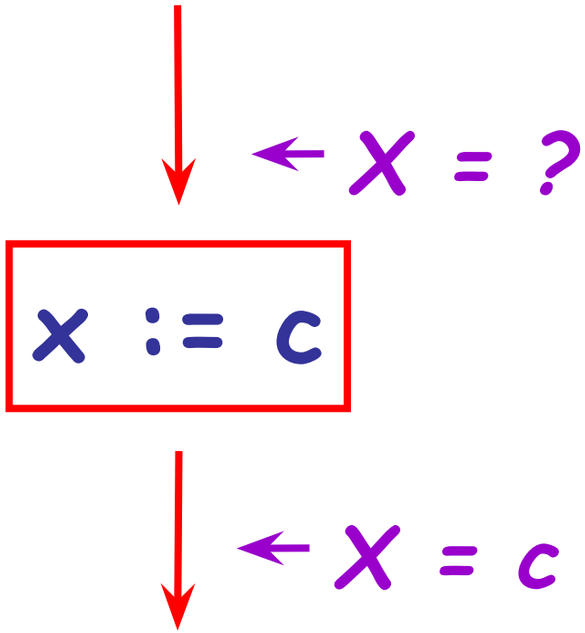
- The idea is to "push" or "**transfer**" information from one statement to the next

- For each statement s, we compute information about the value of x immediately before and after s
  - $C_{in}(x,s)$ = value of x before s
  - $C_{out}(x,s)$ = value of x after s

Transfer Functions

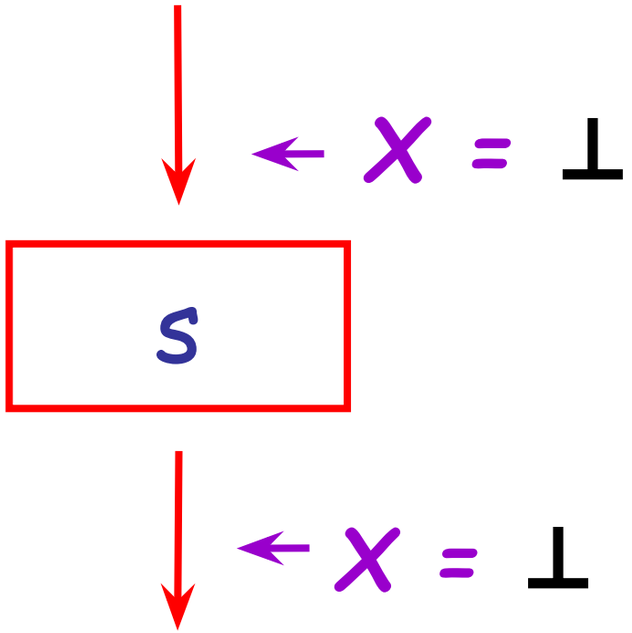
- Define a **transfer function** that transfers information from one statement to another

Rule 1



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

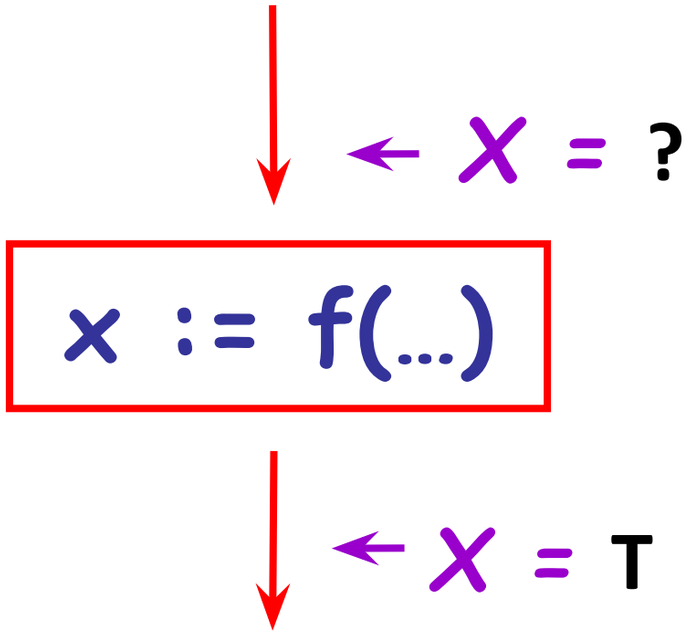
Rule 2



- $C_{out}(x, s) = \bot$ if $C_{in}(x, s) = \bot$

Recall: $\bot$ = “unreachable code”

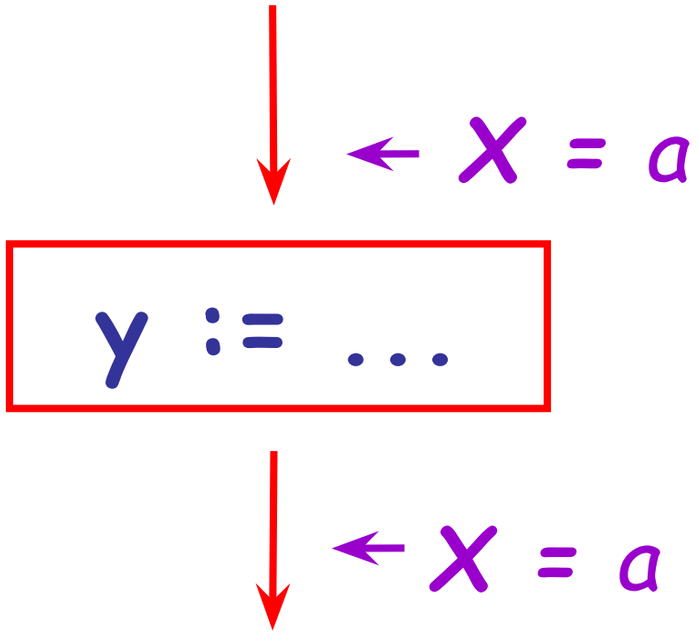
Rule 3



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

This is a conservative approximation! It might be possible to figure out that f(...) always returns 0, but we won't even try!

Rule 4

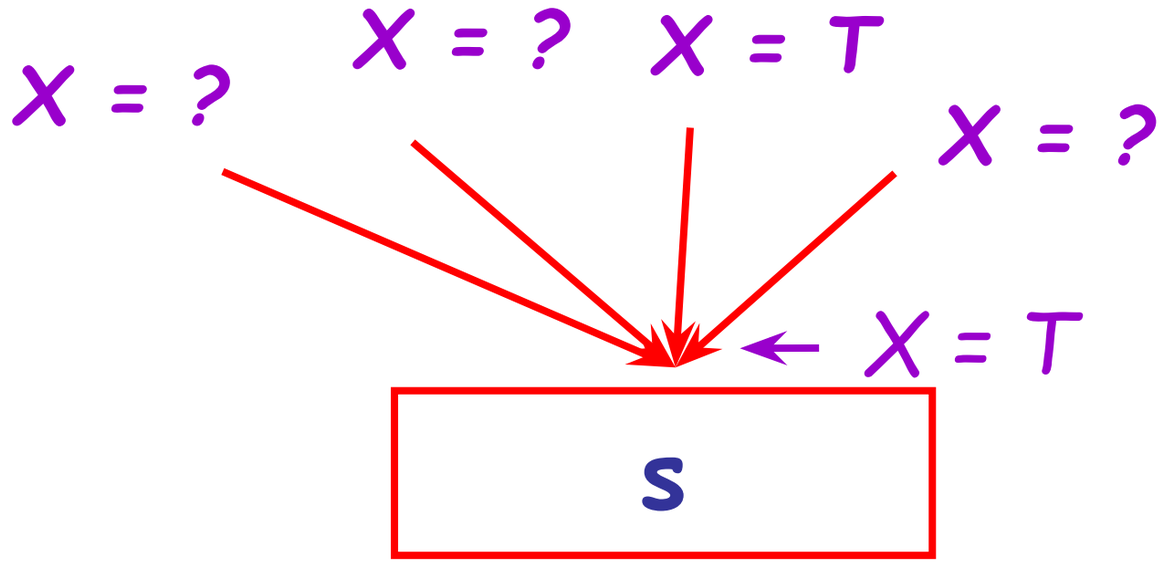


- $C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ if $x \neq y$

The Other Half

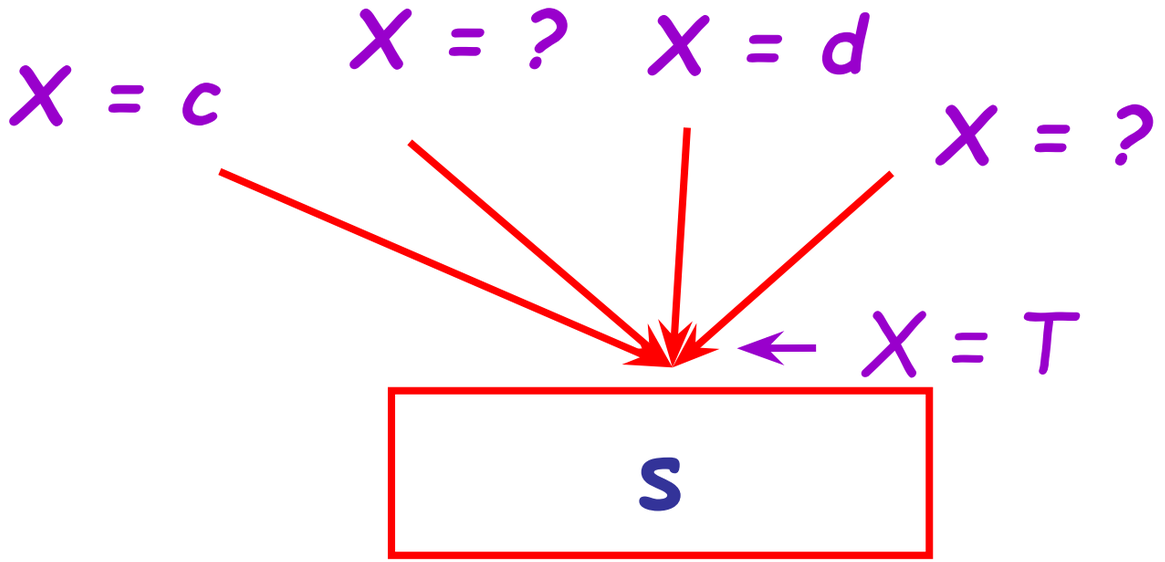
- Rules 1-4 relate the ***in*** of a statement **to** the ***out*** of the same statement
  - they propagate information through a statement
- Now we need rules relating the ***out*** of one statement **to** the ***in*** of the successor statement
  - to propagate information forward along paths
- In the following rules, let statement s have immediate predecessor statements $p_1,\ldots,p_n$

Rule 5



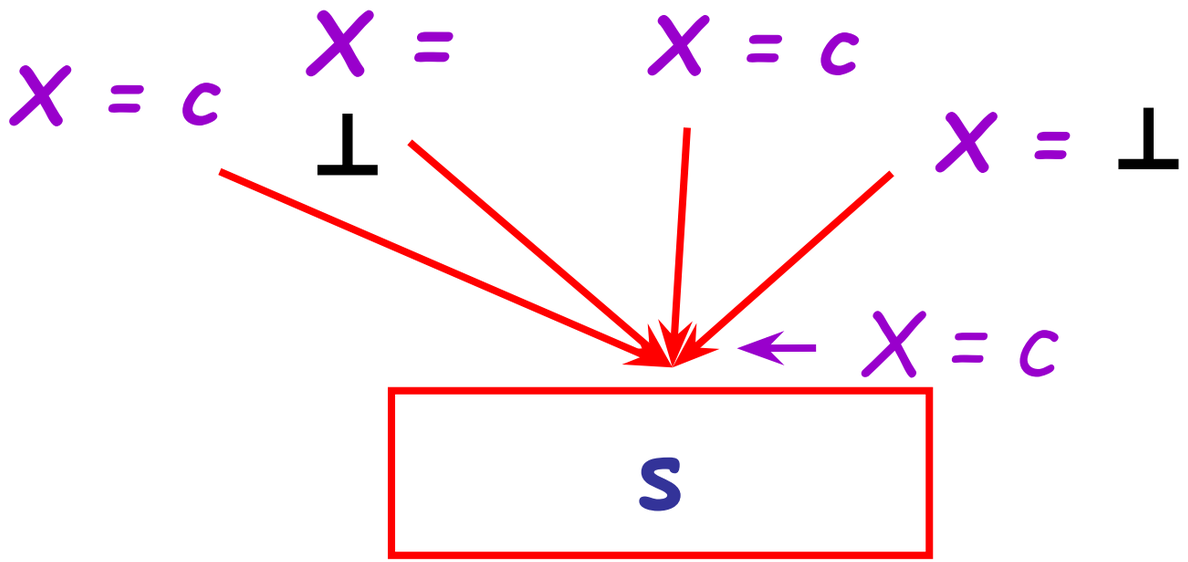
- if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

Rule 6



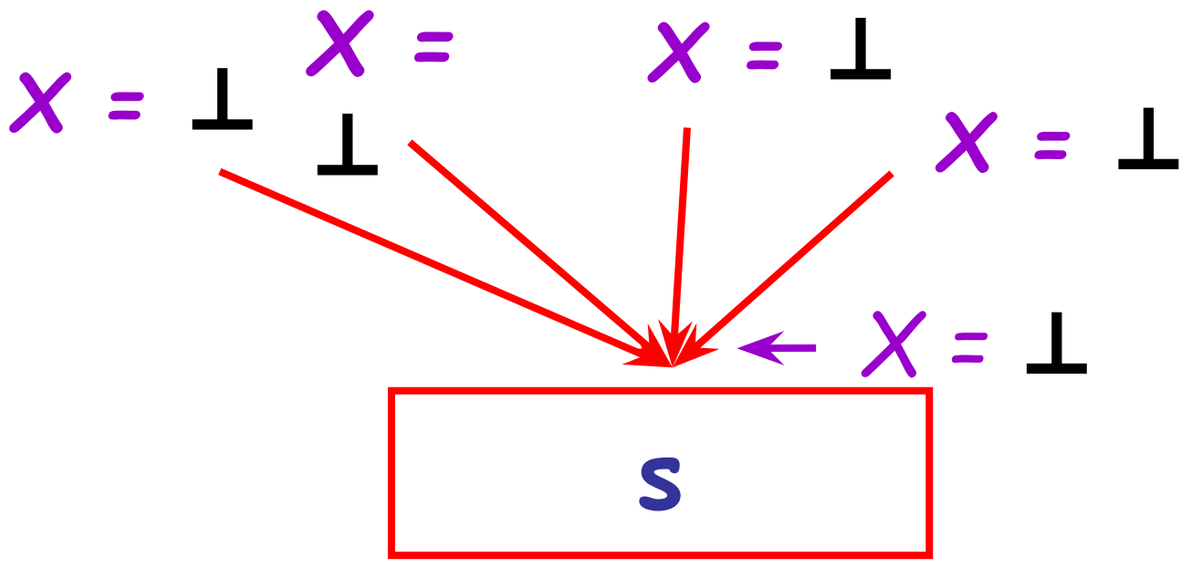
if $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$, then $C_{in}(x, s) = T$

Rule 7



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

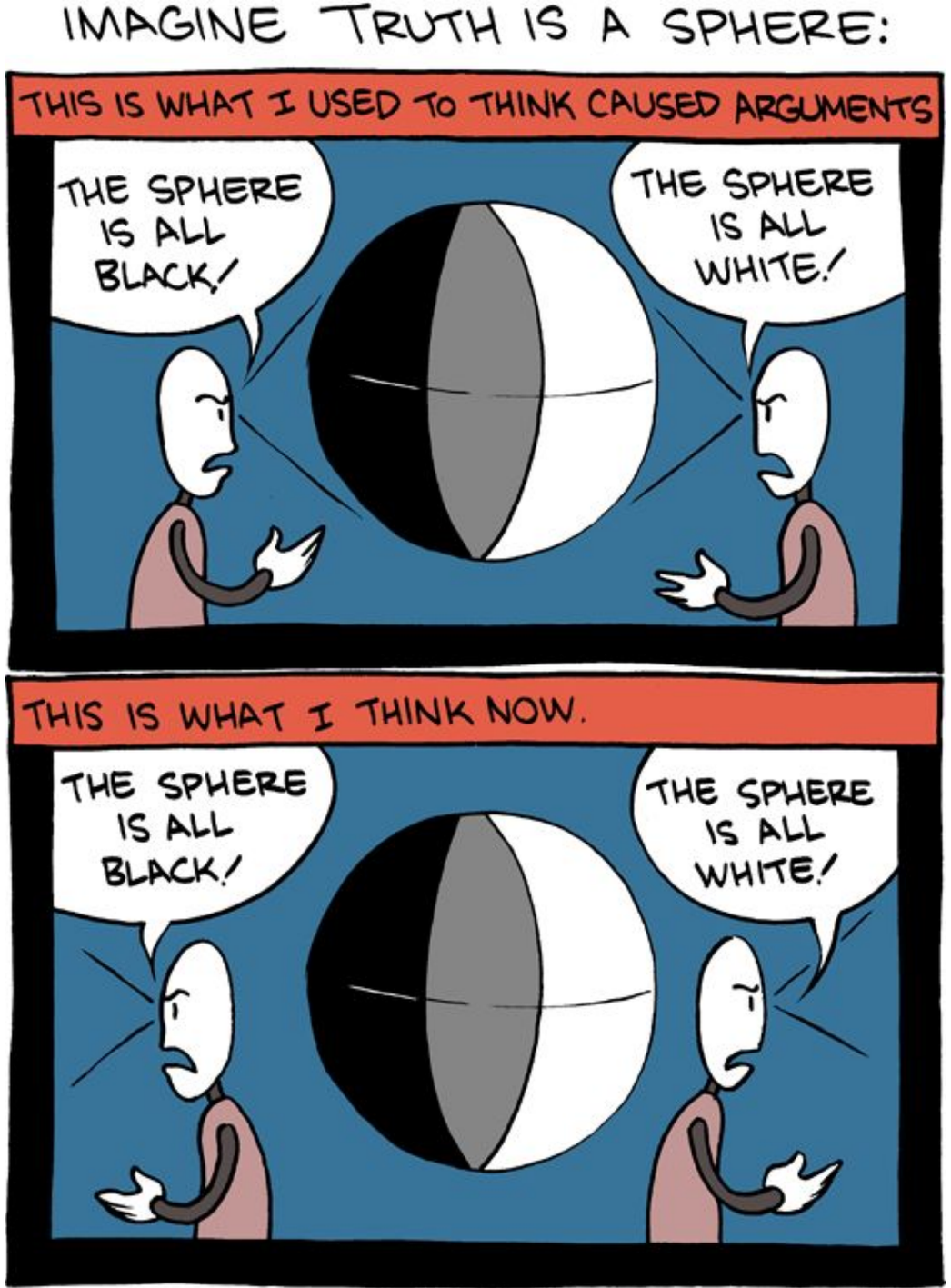
Rule 8



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

Static Analysis Algorithm

- For every entry s to the program, set $C_{in}(x, s) = T$
- Set $C_{in}(x, s) = C_{out}(x, s) = \bot$ everywhere else
- Repeat until all points satisfy rules 1-8:
  - Pick *s* not satisfying rules 1-8 and update using the appropriate rule



Static and Dataflow Analysis

(two-part lecture)

“Static” means?

Programs are viewed as ___?

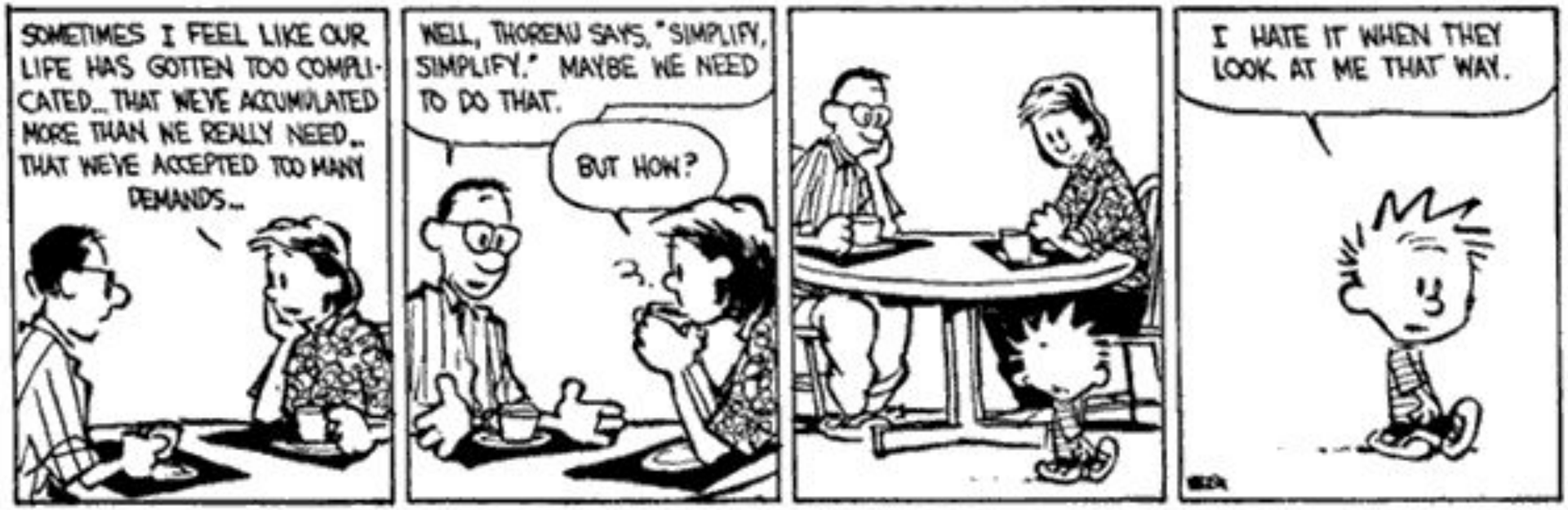
Abstraction: what are special abstract values?

One-Slide Summary

- **Static analysis** is the systematic examination of an **abstraction** of program state space with respect to a property. Static analyses reason about all possible executions but they are **conservative**.
- **Dataflow analysis** is a popular approach to static analysis. It tracks a few broad values (“secret information” vs. “public information”) rather than exact information. It can be computed in terms of a local **transfer** of information.

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

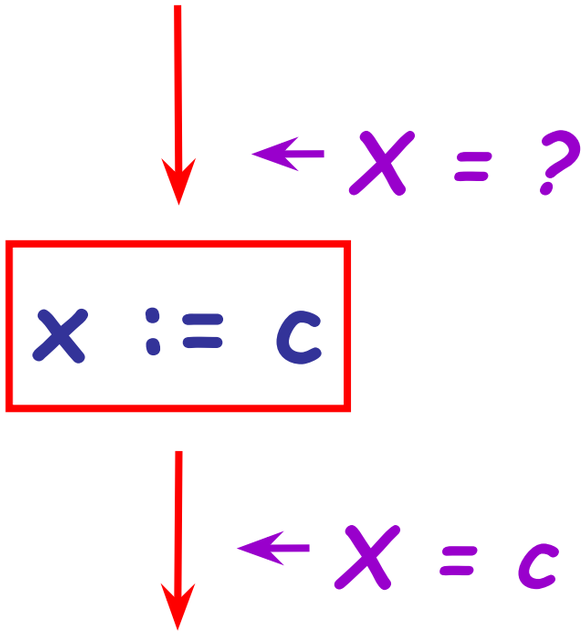
- The idea is to "push" or "**transfer**" information from one statement to the next

- For each statement s, we compute information about the value of x immediately before and after s
  - $C_{in}(x,s)$ = value of x before s
  - $C_{out}(x,s)$ = value of x after s

Transfer Functions

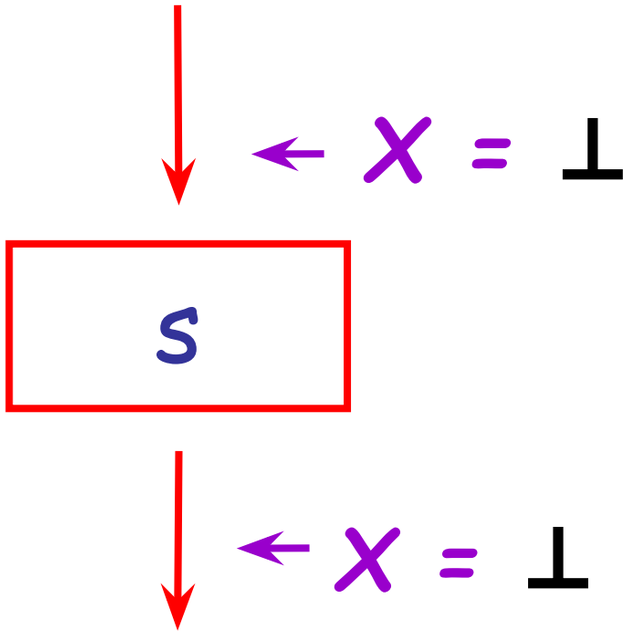
- Define a **transfer function** that transfers information from one statement to another

Rule 1



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

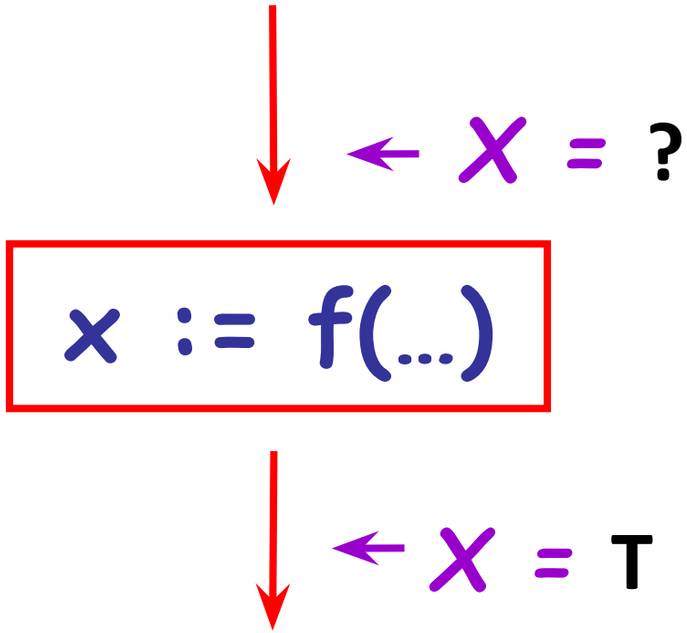
Rule 2



- $C_{out}(x, s) = \bot$ if $C_{in}(x, s) = \bot$

Recall: $\bot$ = “unreachable code”

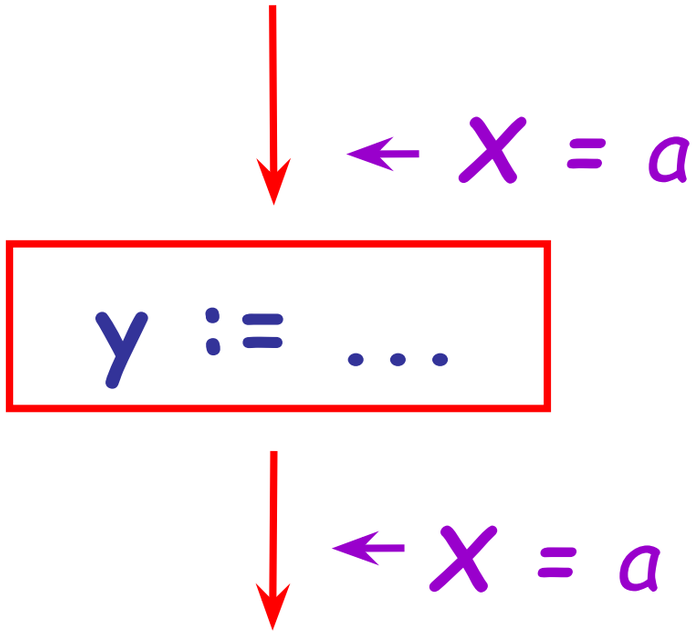
Rule 3



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

This is a conservative approximation! It might be possible to figure out that f(...) always returns 0, but we won't even try!

Rule 4

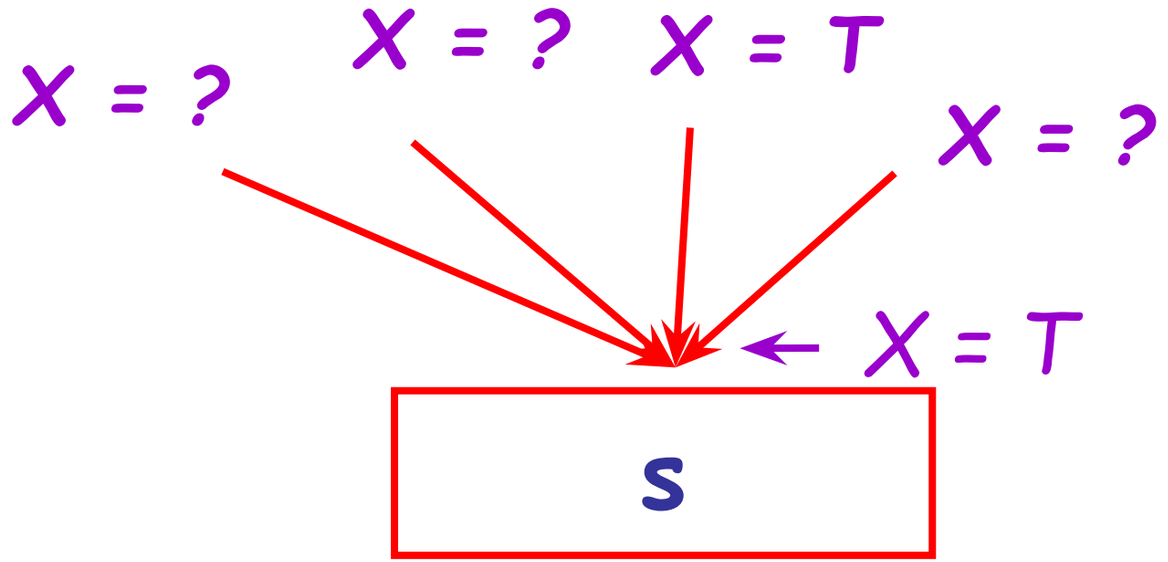


- $C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ if $x \neq y$

The Other Half

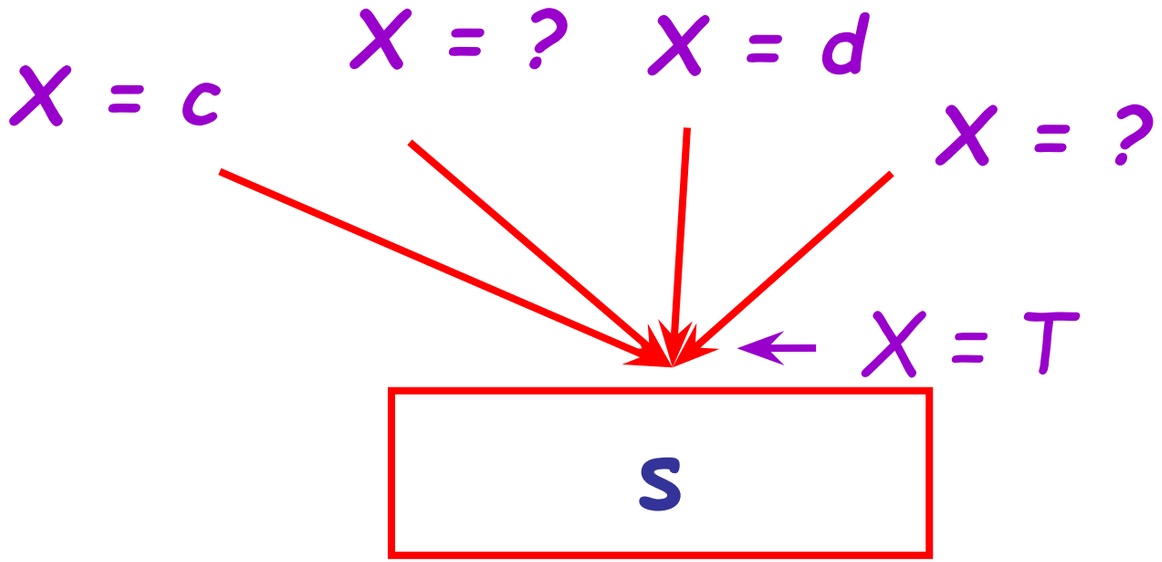
- Rules 1-4 relate the ***in*** of a statement **to** the ***out*** of the same statement
  - they propagate information through a statement
- Now we need rules relating the ***out*** of one statement **to** the ***in*** of the successor statement
  - to propagate information forward along paths
- In the following rules, let statement s have immediate predecessor statements $p_1,\ldots,p_n$

Rule 5



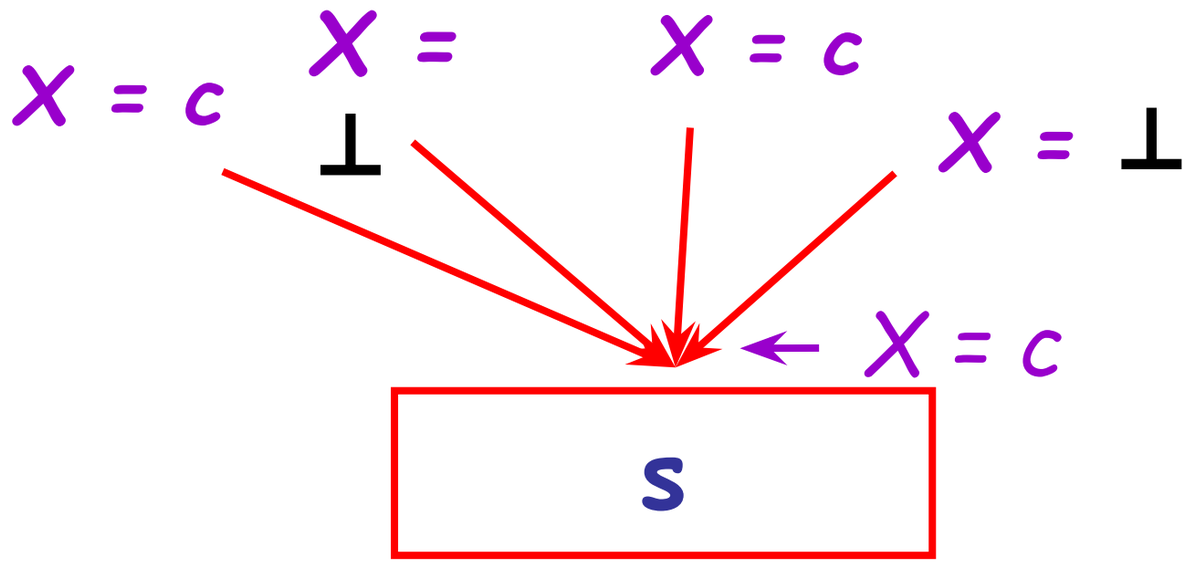
- if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

Rule 6



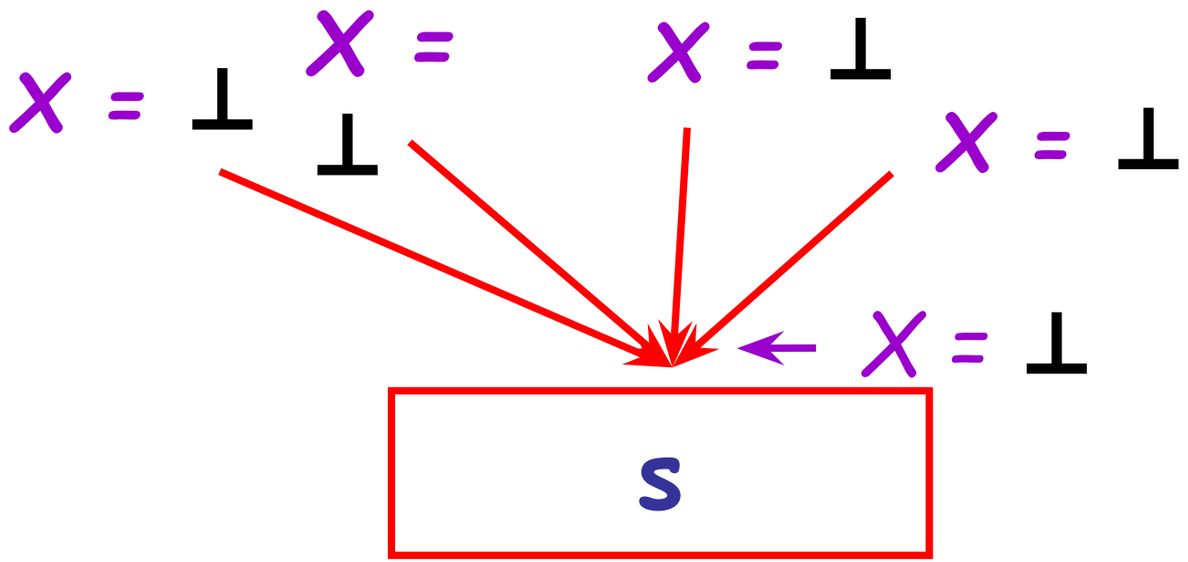
if $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$, then $C_{in}(x, s) = T$

Rule 7



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

Rule 8



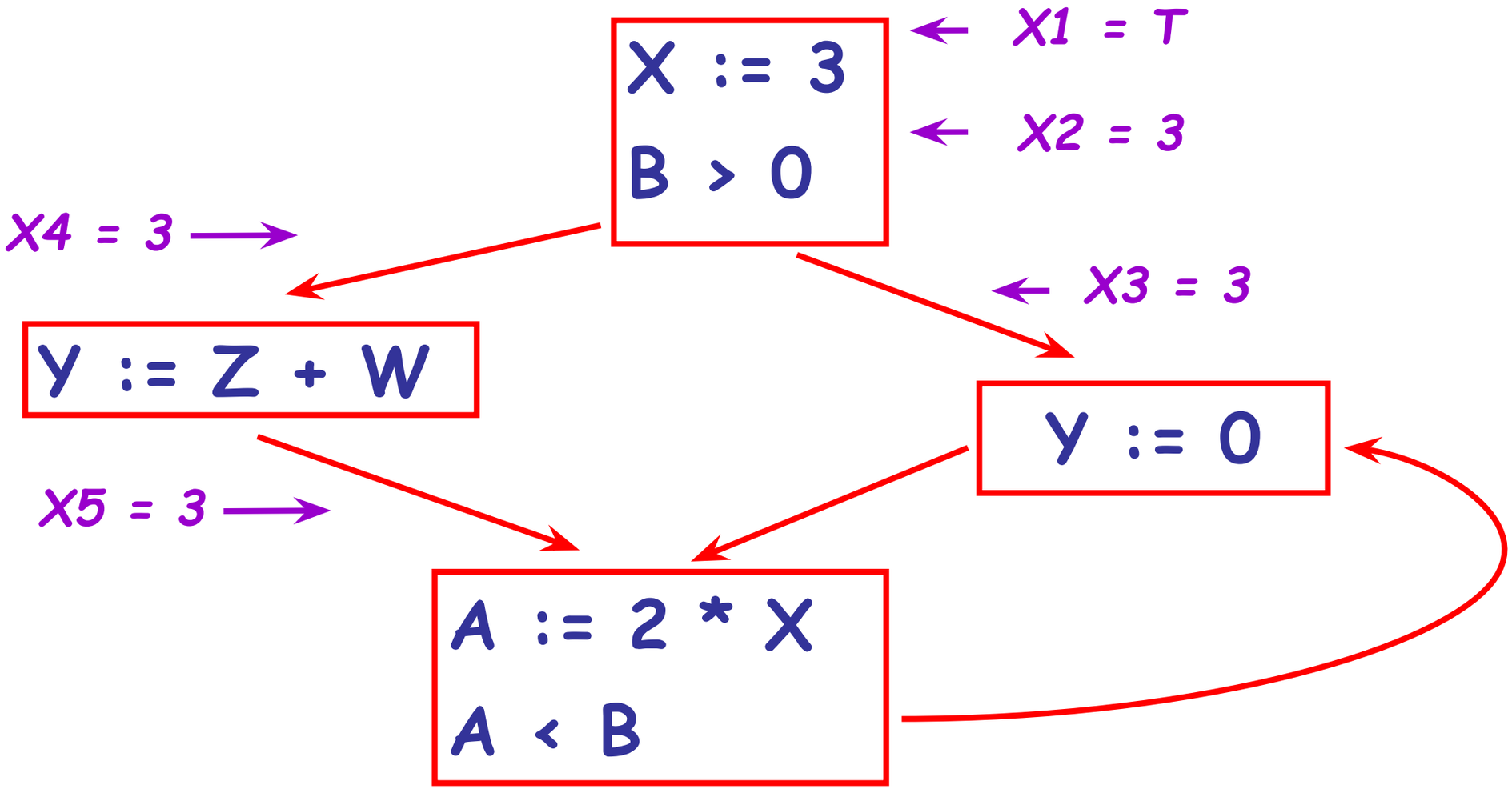
if $C_{out}(x, p_i) = \bot$ for all i, then $C_{in}(x, s) = \bot$

Static Dataflow Analysis Algorithm

- For **every entry** s to the program, set $C_{in}(x, s) = T$
- Set $C_{in}(x, s) = C_{out}(x, s) = \perp$ **everywhere else**
- **Repeat** until all points satisfy rules 1-8:
  - Pick *s* not satisfying rules 1-8 and update using the appropriate rule

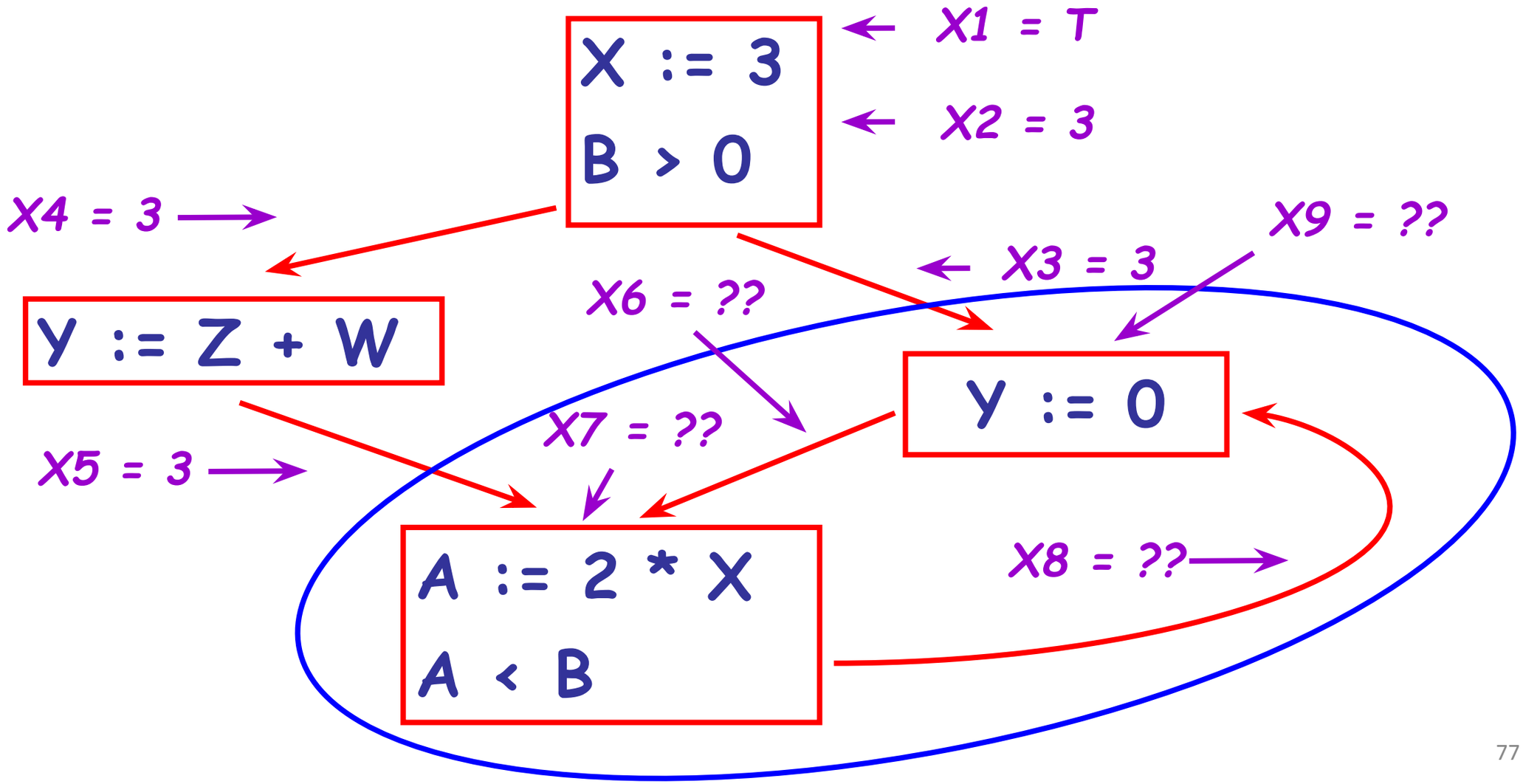
The Value ⊥

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



The Value ⊥

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



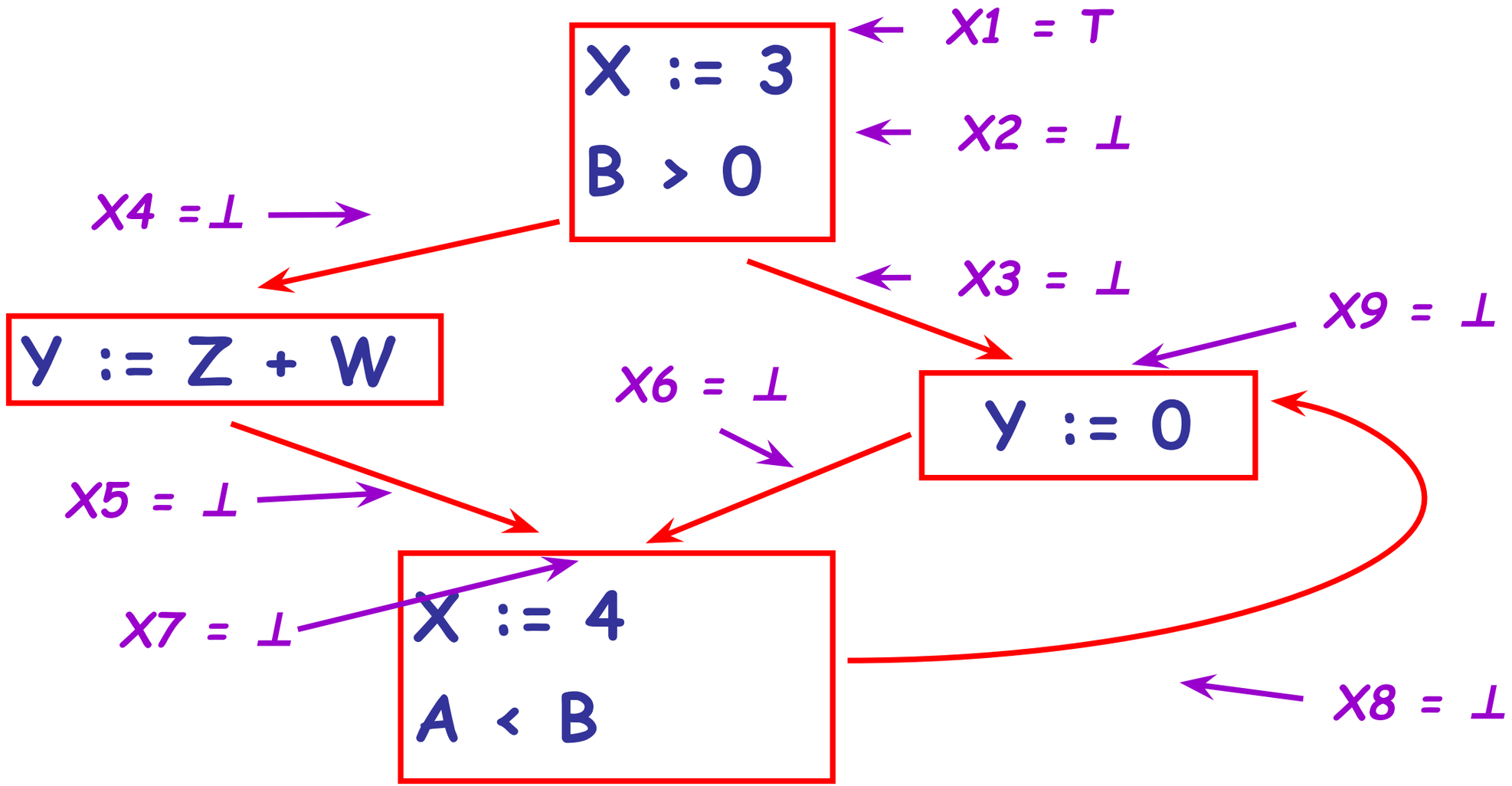
The Value ⊥ (Cont.)

- We want all points to have values at all times during the analysis; but with cycles, we cannot…

- Solution: assigning **some initial value** allows the analysis to break cycles

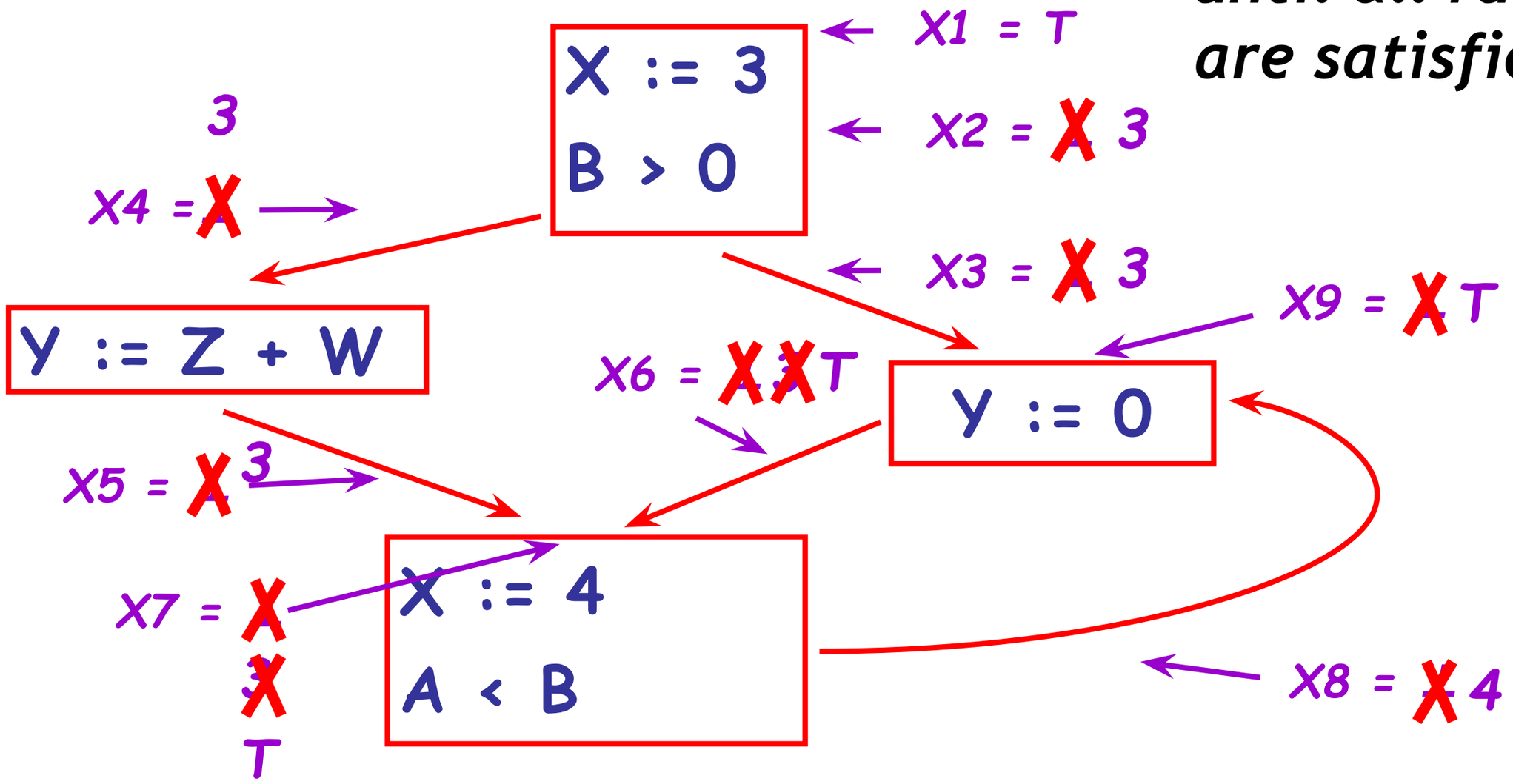
- The initial value ⊥ means **“we have not yet analyzed control reaching this point”**

Another Example: Analyze the value of X …



Another Example: Analyze the value of X …

Must continue until all rules are satisfied !

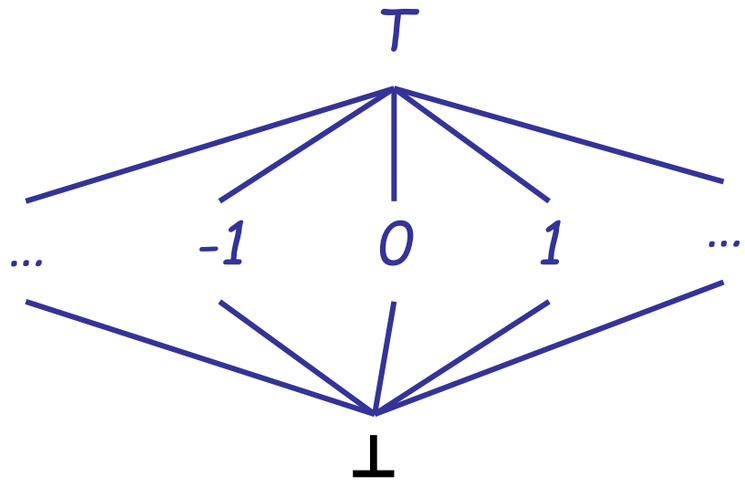


Orderings

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

$$\bot < c < \top$$

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



This is called a "lattice"

Orderings (Cont.)

- T is the greatest value, ⊥ is the least
  - All constants are in between and incomparable
    - (with respect to this analysis)
- Let ***lub*** be the least-upper bound in this ordering
  - cf. “least common ancestor” in Java/C++
- 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 $\bot$ and only *increase*

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

Number Crunching

- The algorithm is polynomial in program size:

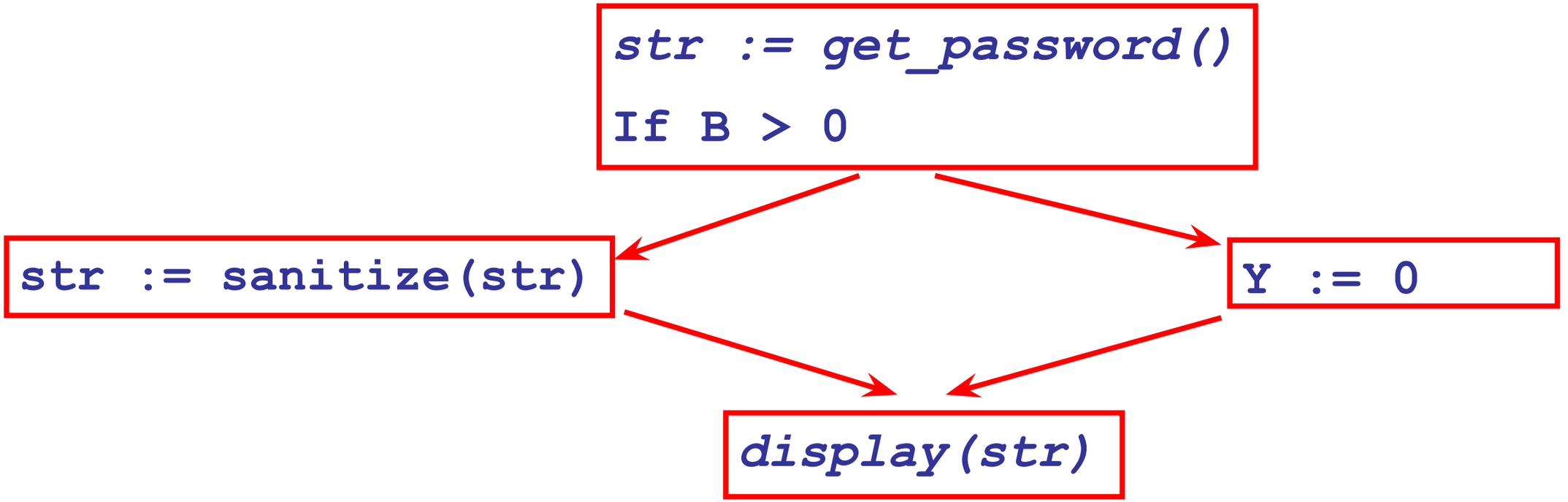
  Number of steps

  = Number of C_(….) values * 2

  = (Number of program statements)^2 * 2

“Potential Secure Information Leak” Analysis

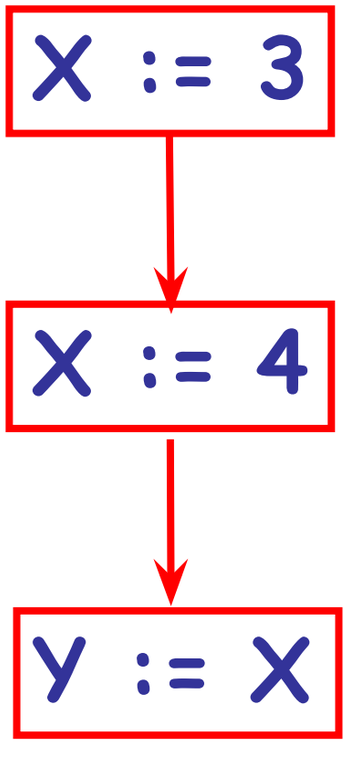
- Could **sensitive** information **possibly** reach an **insecure** use?



In this example, the password contents can potentially flow into a public display (depending on the value of B)

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
  - We can generalize it to reason about "potential secure information leaks"



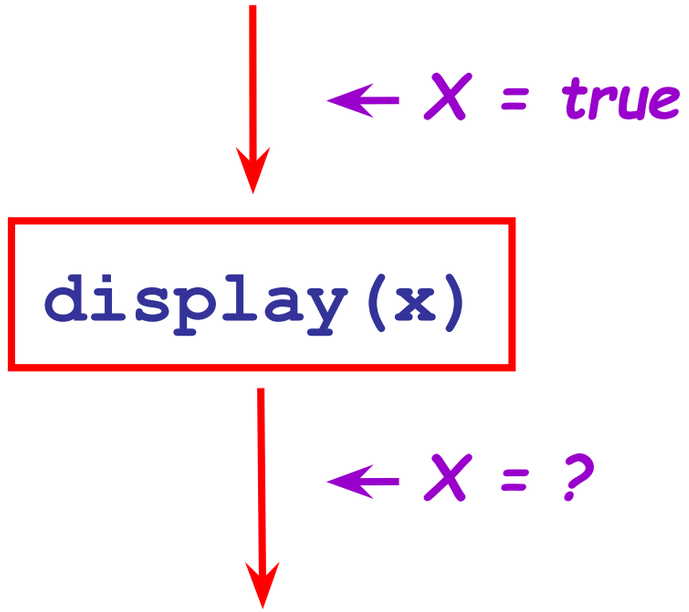
Sensitive Information

- A variable x at statement s is a **possible** sensitive (high-security) information leak if
  - There exists a ("display") statement s' that uses x
  - There is a path from s to s'
  - That path has **no intervening low-security assignment** to x

Computing Potential Leaks

- We can express **high- or low-security status** of a variable in terms of information transferred between adjacent statements, just as in our “definitely null” analysis
- In this formulation of security status we only care about “high” (secret) or “low” (public), not the actual value
  - We have *abstracted away* the value
- This time we will start at the public display of information and work backwards

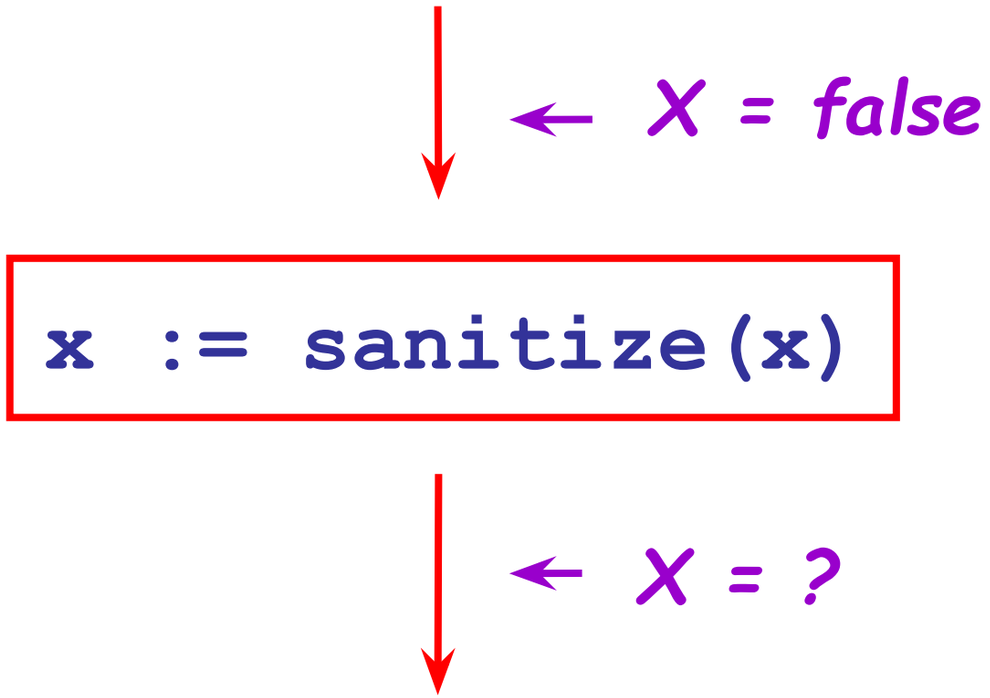
Secure Information Flow Rule 1



$H_{in}(x, s)$ = true if s displays x publicly

true means “the value in x at this point can potentially be leaked”

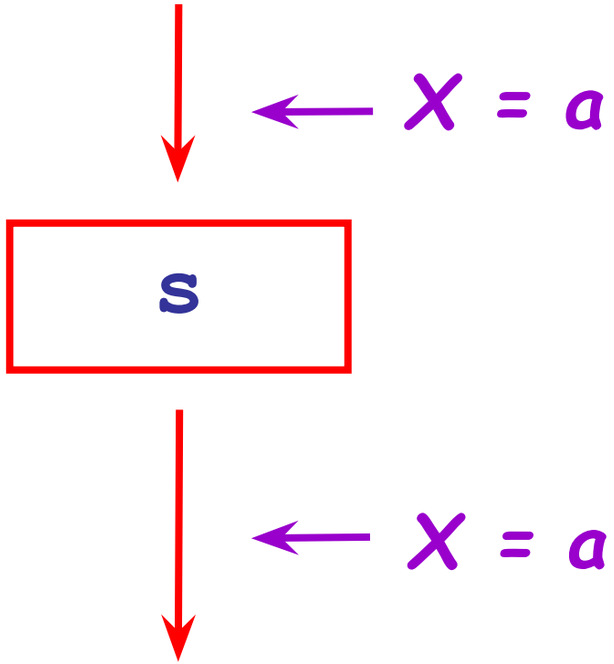
Secure Information Flow Rule 2



$H_{in}(x, x := e) = \text{false}$

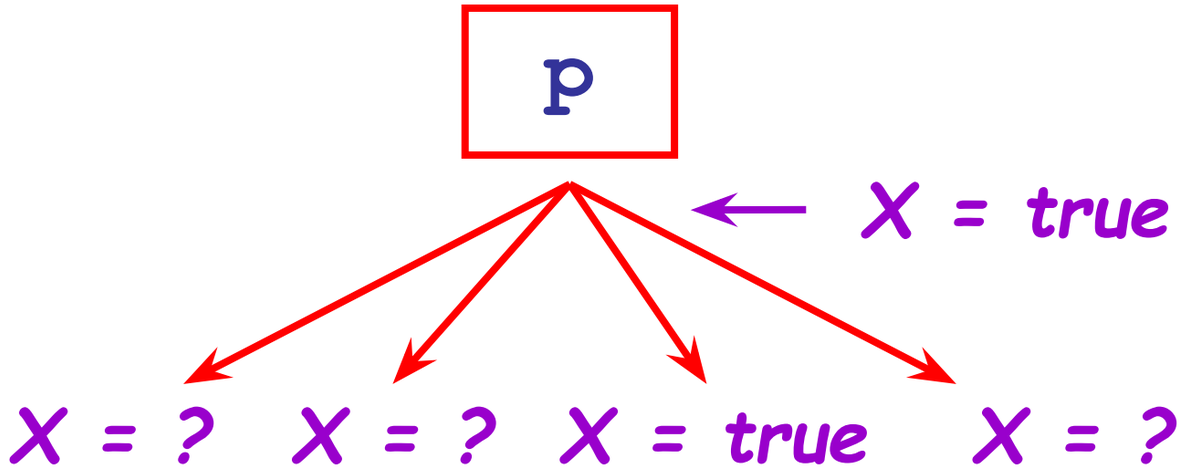
(any subsequent use is safe)

Secure Information Flow Rule 3



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

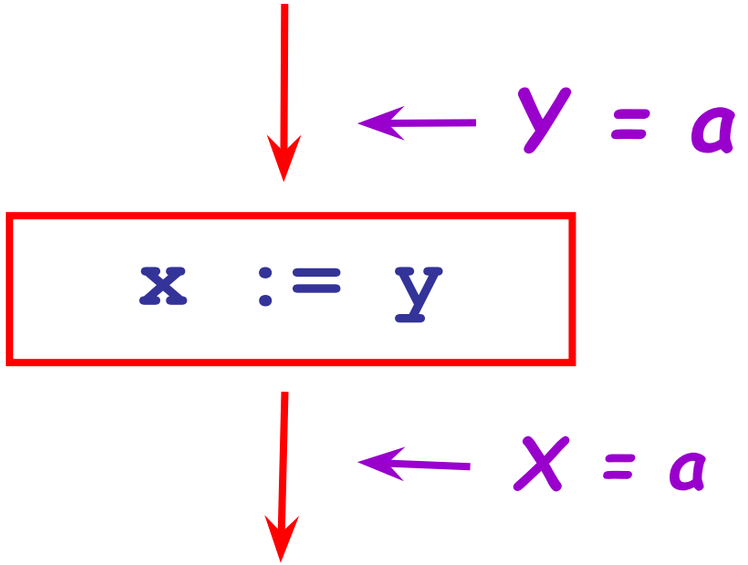
Secure Information Flow Rule 4



- $H_{out}(x, p) = \bigvee \{ H_{in}(x, s) \mid s \text{ a successor of } p \}$

(if there is even one way to potentially have a leak, we potentially have a leak!)

Secure Information Flow Rule 5 (Bonus!)



- $H_{in}(y, x := y) = H_{out}(x, x := y)$

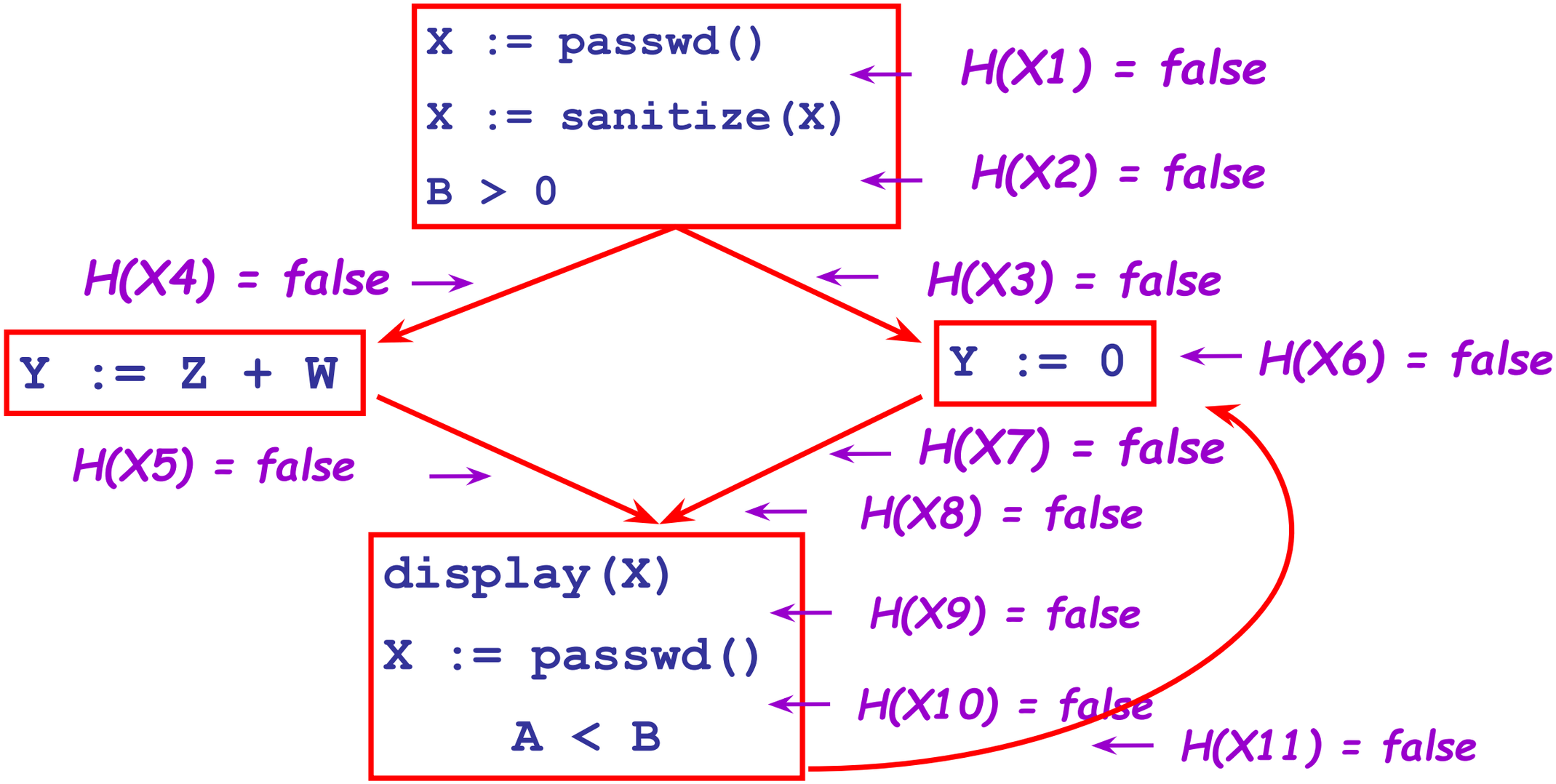
(To see why, imagine the next statement is display(x). Do we care about y above?)

Algorithm

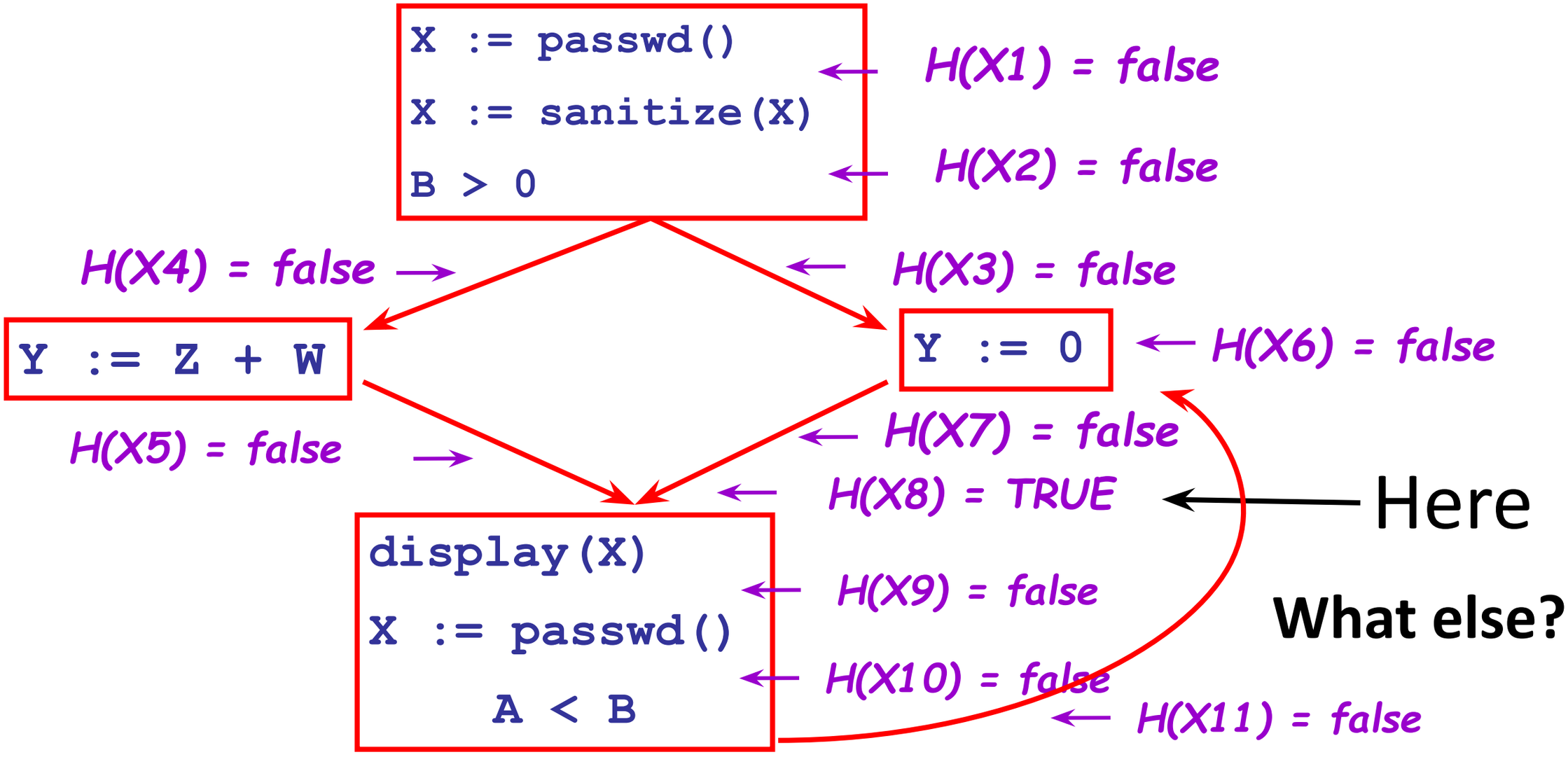
- Let all H_(…) = false initially

- 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

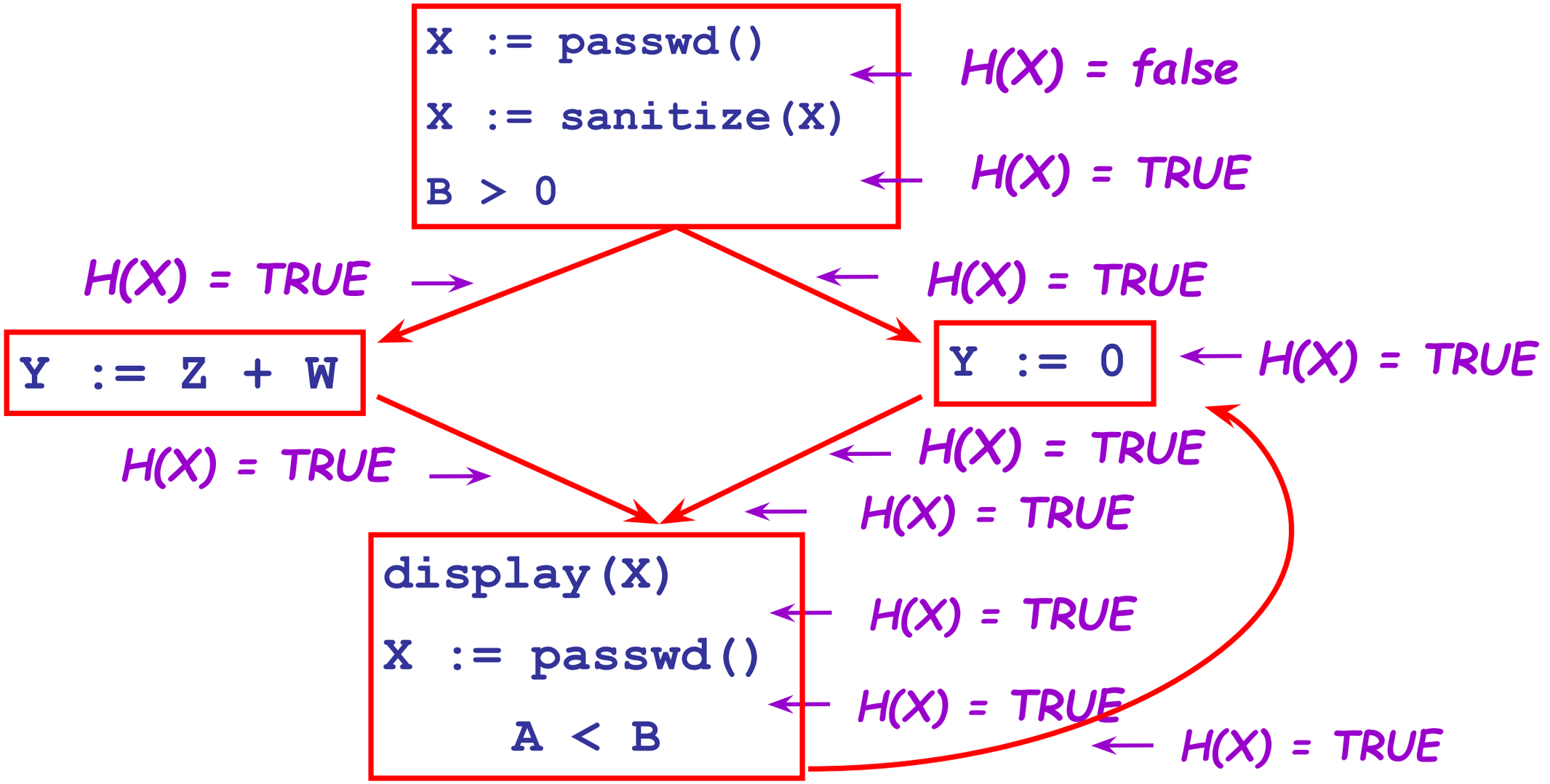
Secure Information Flow Example



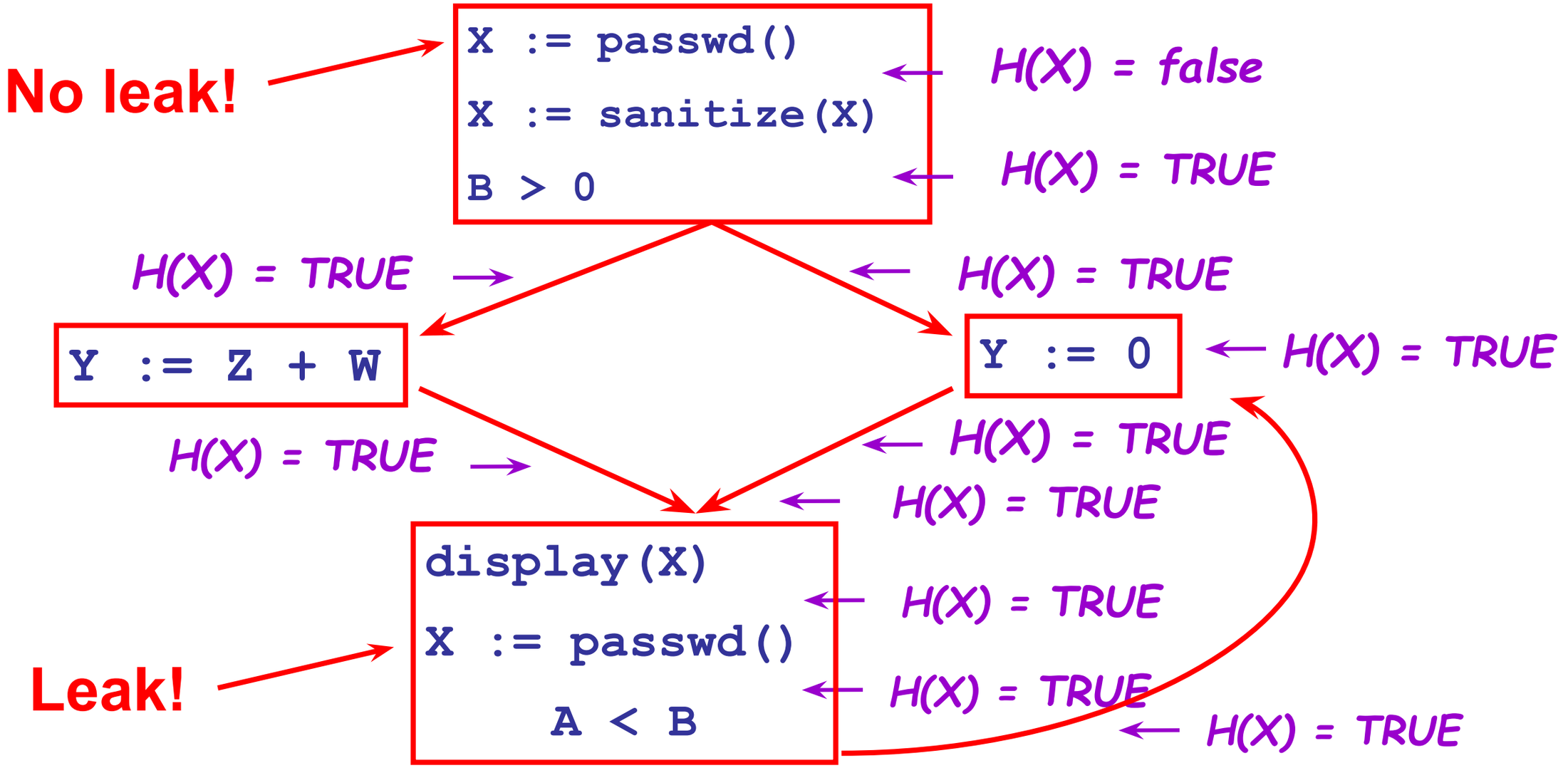
Secure Information Flow Example



Secure Information Flow Example



Secure Information Flow Example



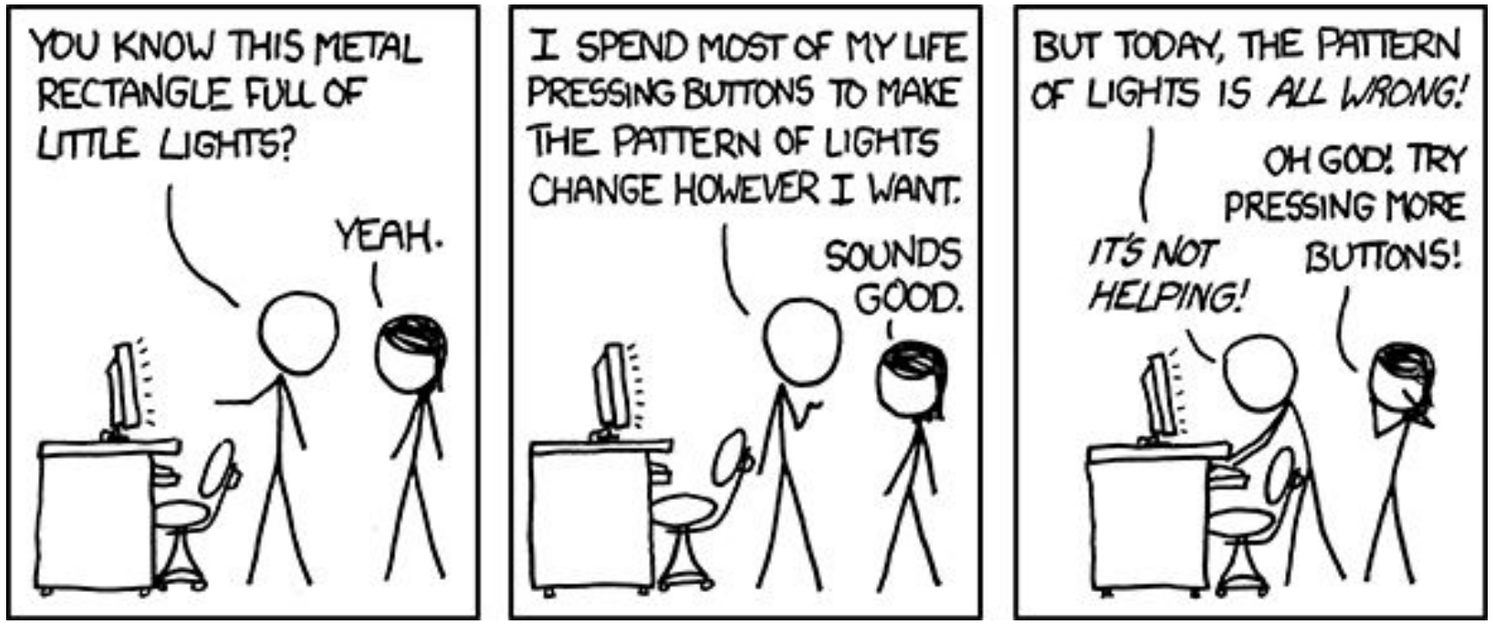
Termination

- 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 issue a warning at a particular sensitive information point (if right after it, the analysis says true)

Static Analysis Limitations

- Where might a static analysis go “wrong”?
- Construct the shortest program that causes a static analysis to get the “wrong” answer?



```
x = new AST()
y = identity(x)
deref y
```

Report Error!

(False Positive)

Static Analysis

- You are asked to design a static analysis to detect bugs related to **file handle**s
  - A file starts out *closed*. A call to open() makes it *open*; open() may only be called on *closed* files. read() and write() may only be called on *open* files. A call to close() makes a file *closed*; close may only be called on *open* files.
  - Report if a file handle is potentially used incorrectly
- What abstract information do you track?
- What do your transfer functions look like?

Abstract Information

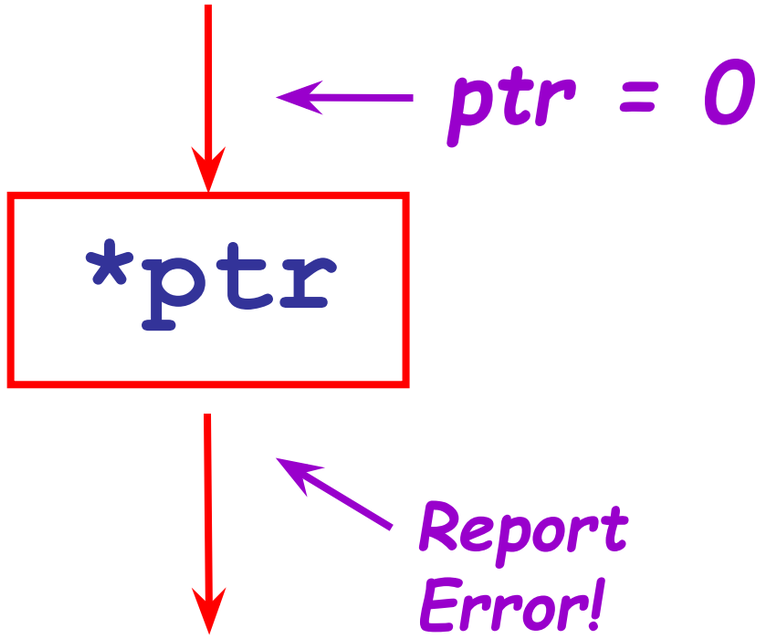
- We will keep track of an abstract value for a given file handle variable

- Values and Interpretations

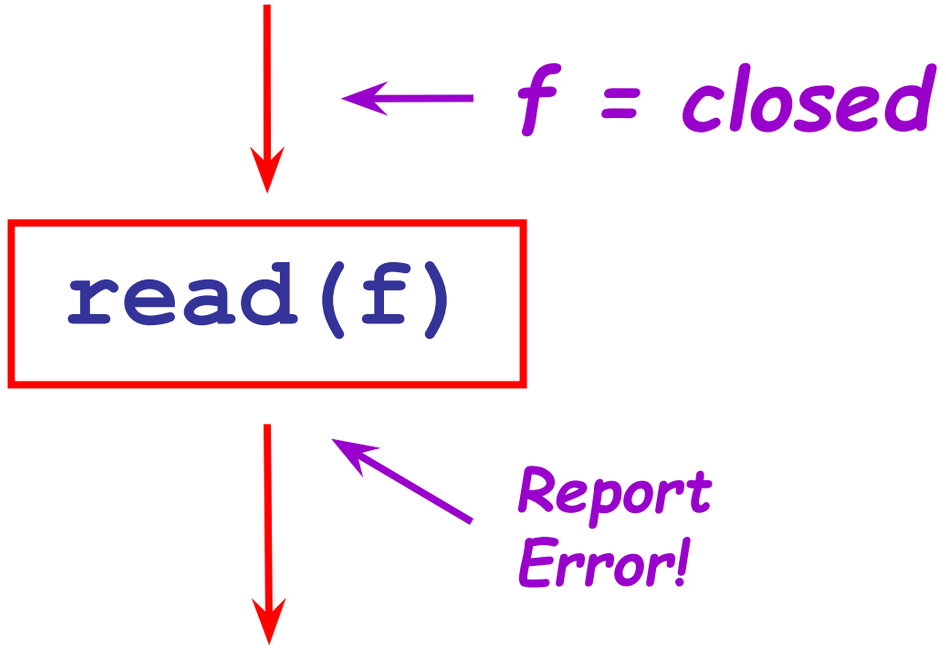
T	file handle state is unknown
⊥	haven't reached here yet
closed	file handle is closed
open	file handle is open

“Null Ptr” vs. “File Handles”

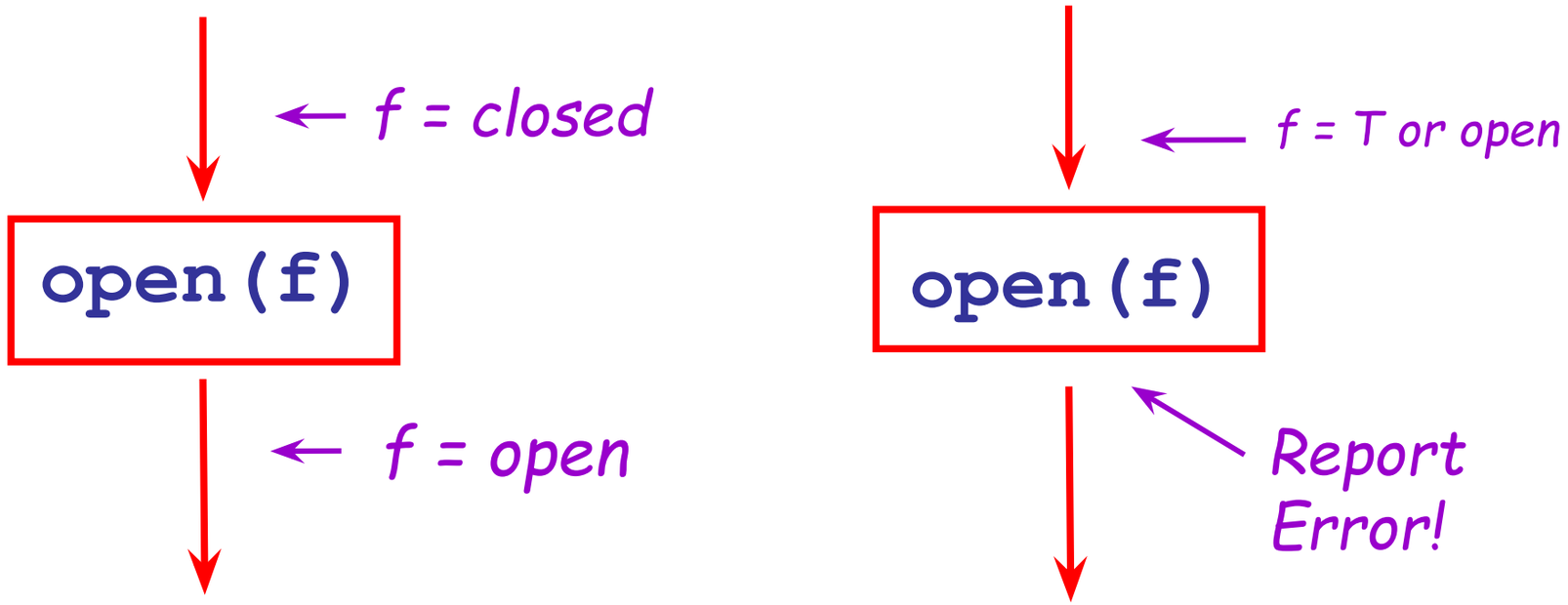
- Previously: “null ptr”



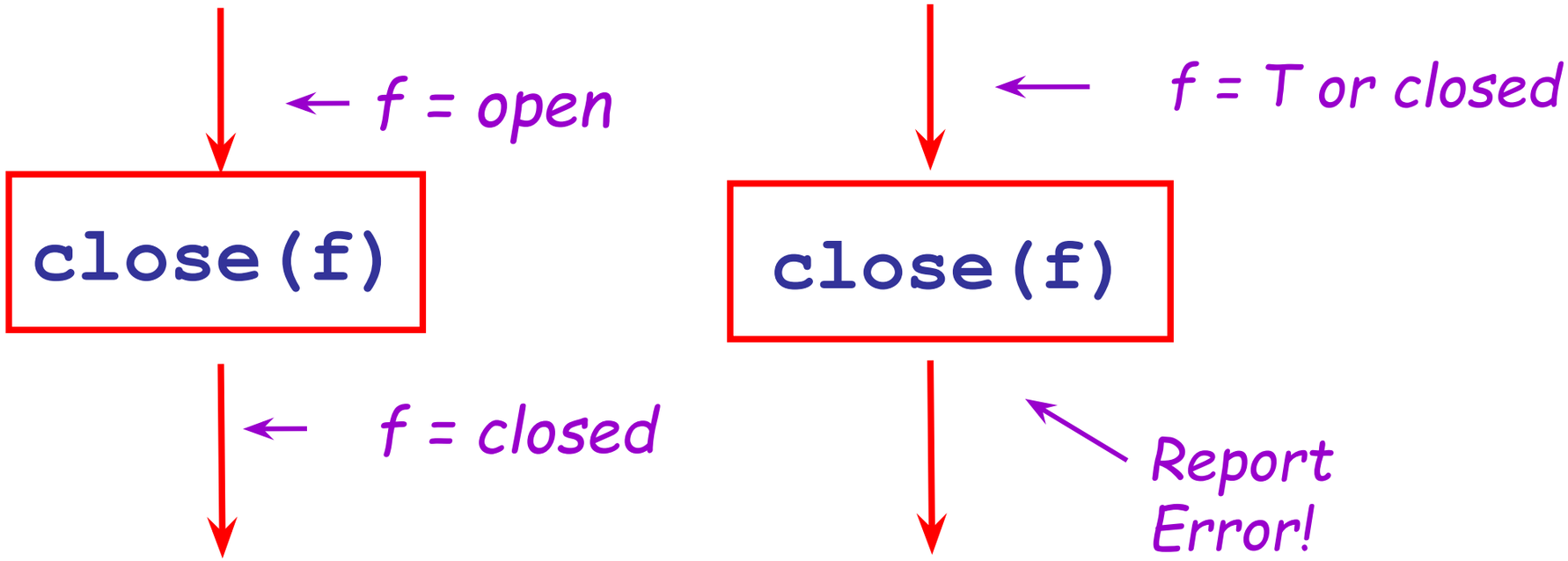
- Now: “file handles”



Rules: open

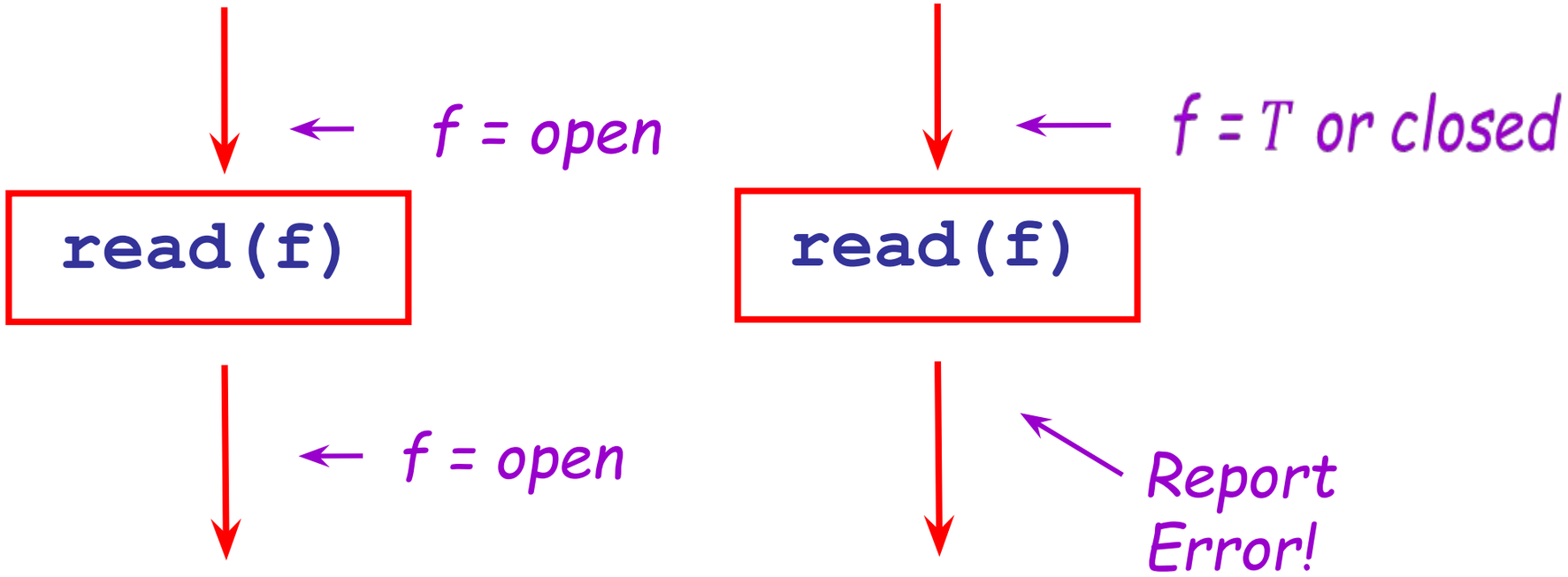


Rules: close

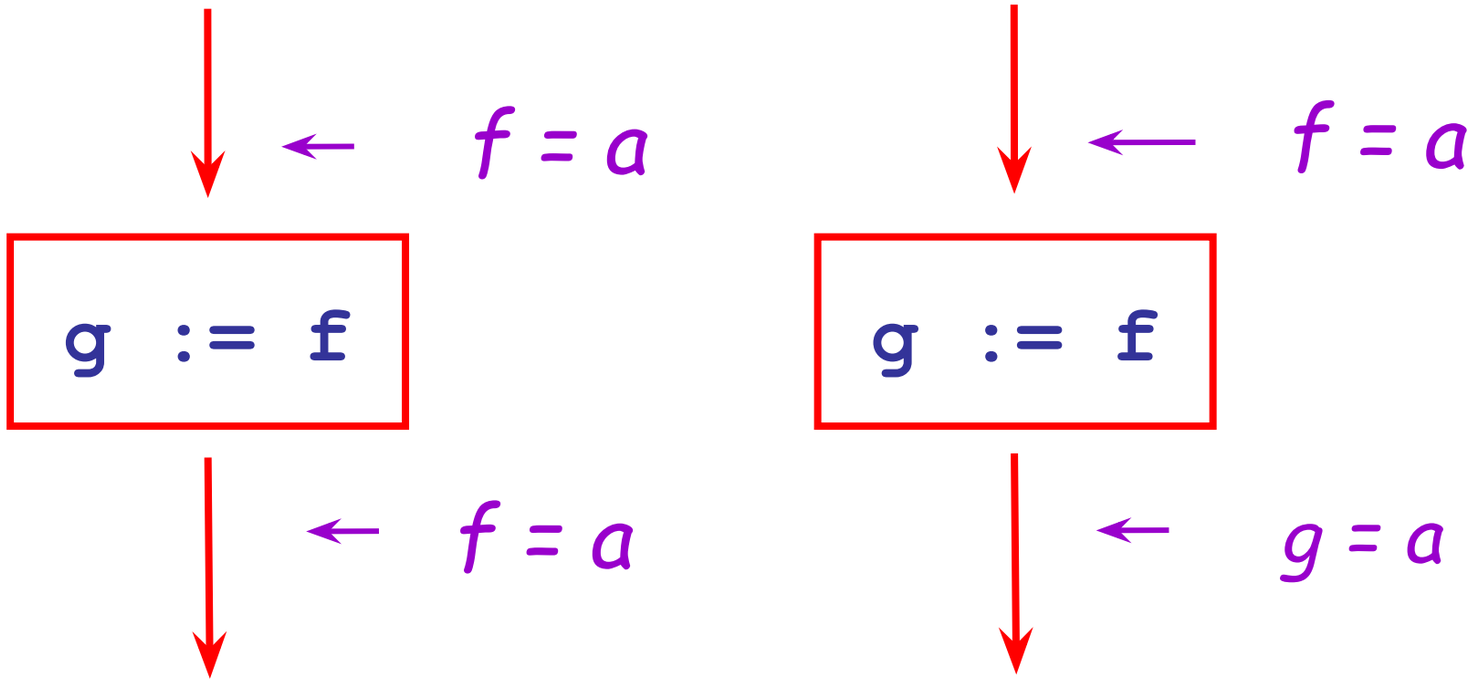


Rules: read/write

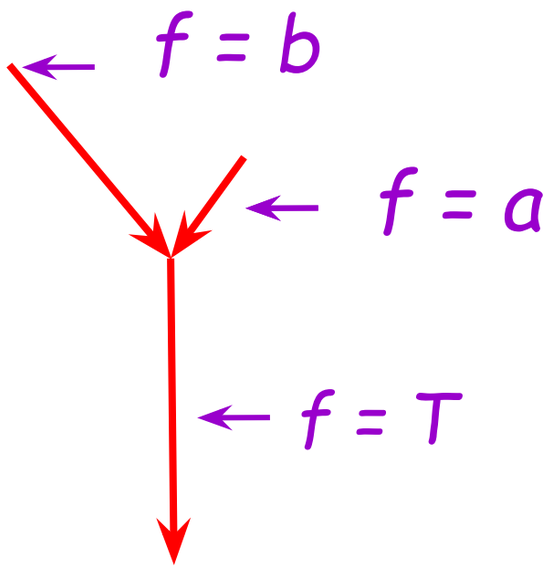
- only show read(f); write(f) is the same

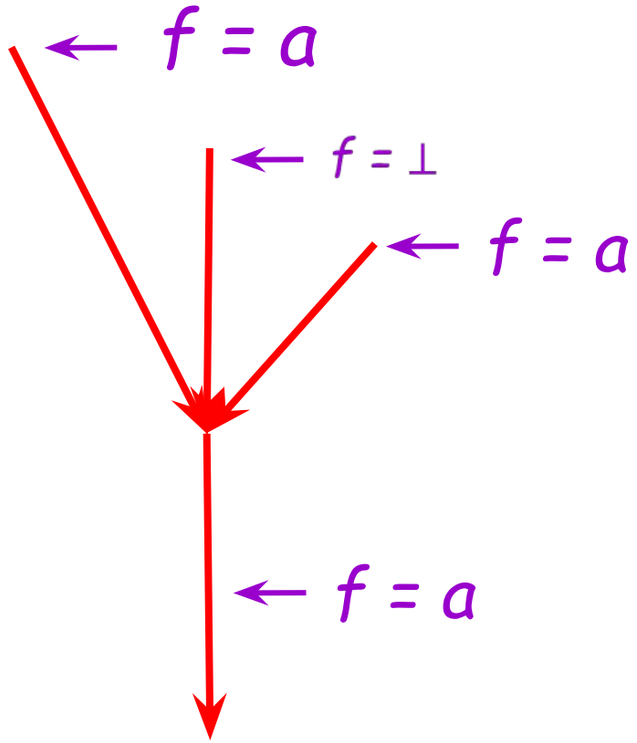


Rules: Assignment



Rules: Multiple Possibilities

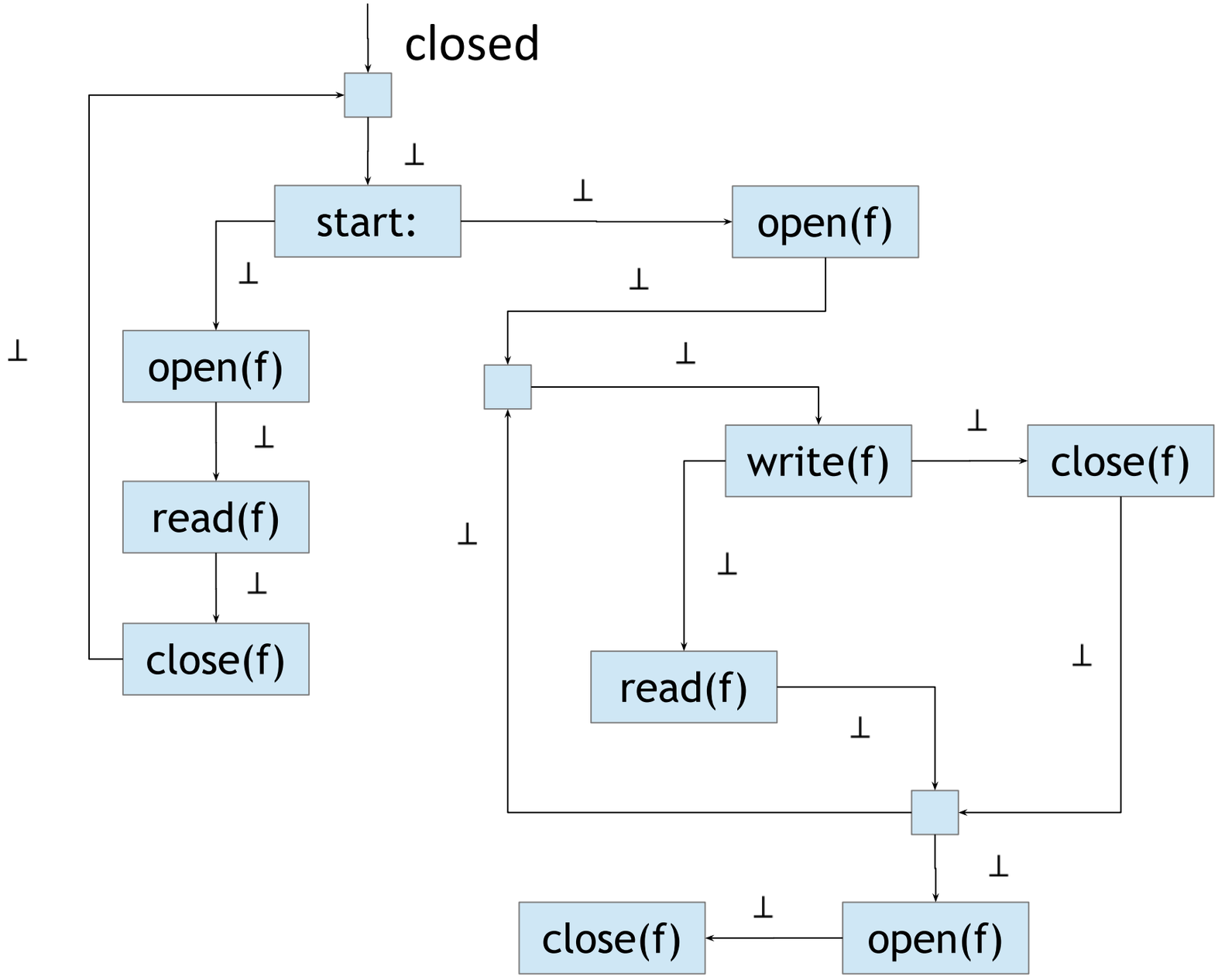


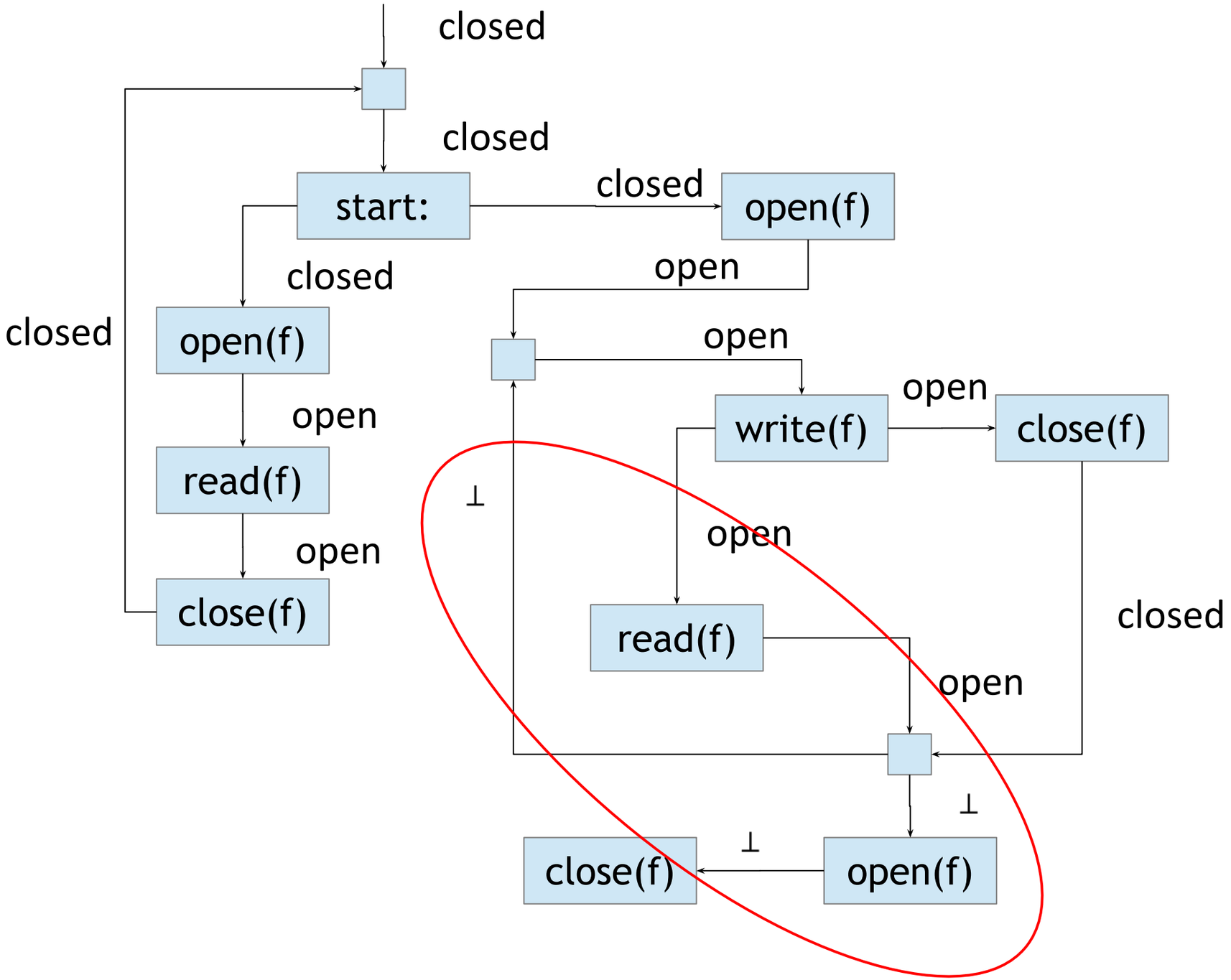


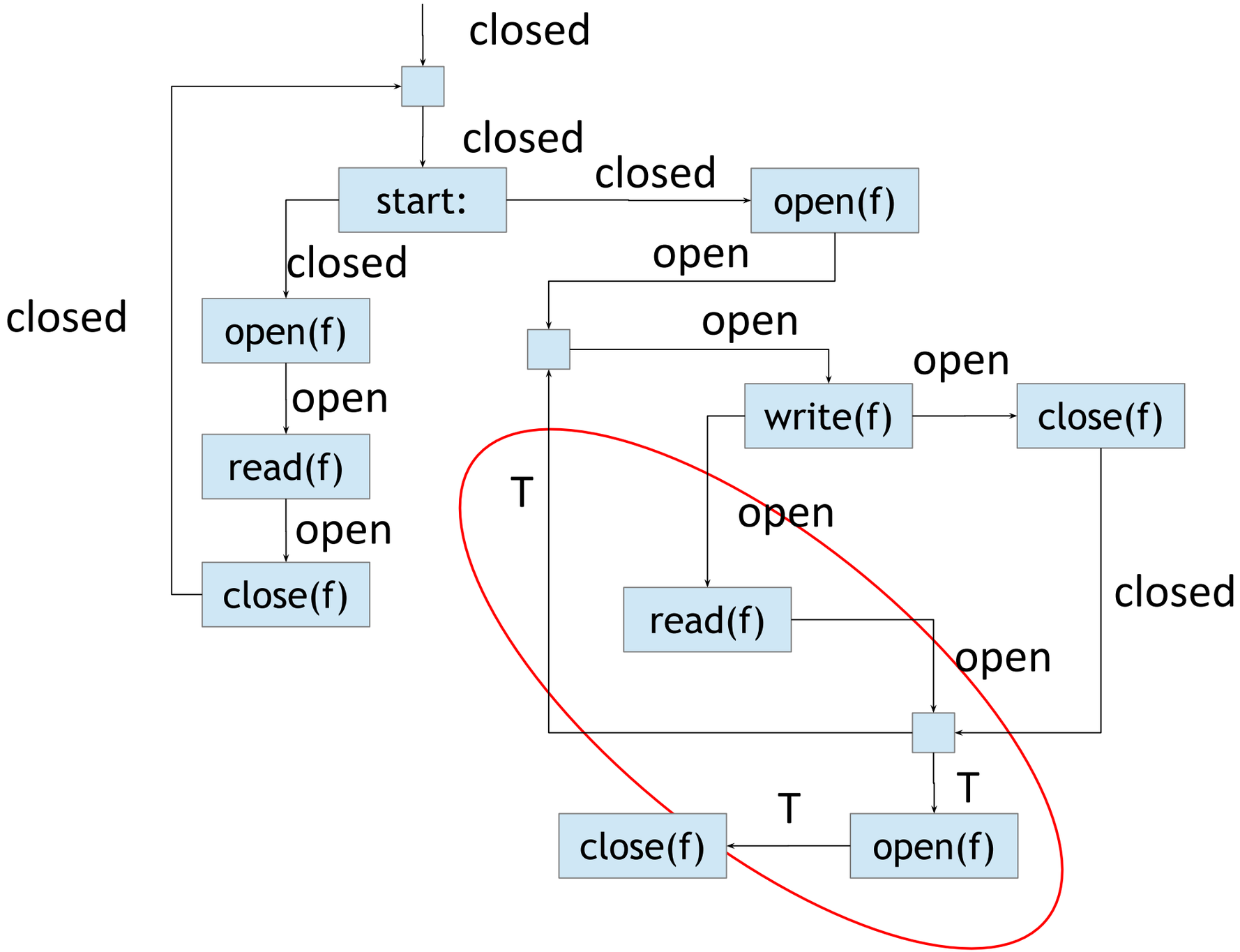
A Tricky Program

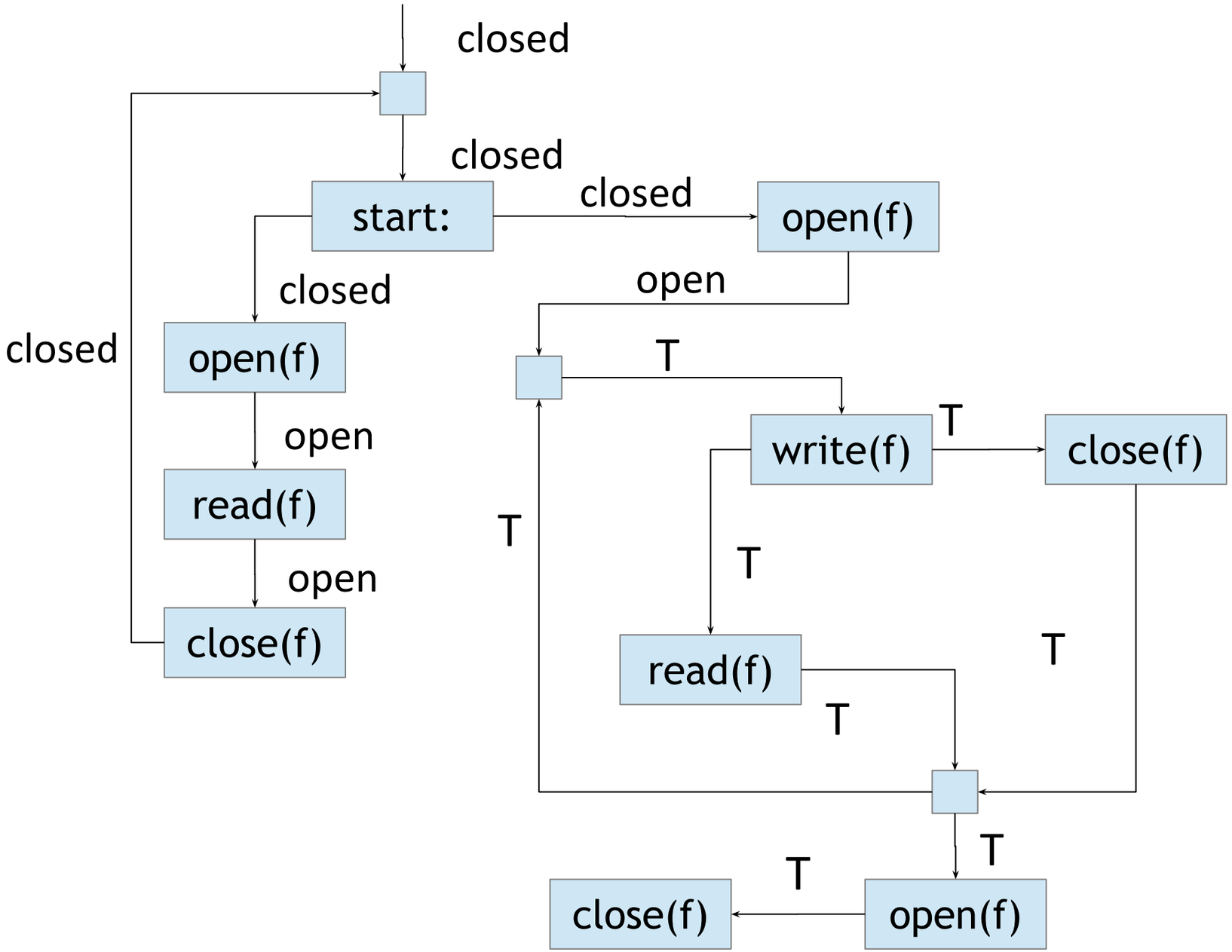
```
start:
switch (a)
  case 1: open(f); read(f); close(f); goto start
  default: open(f);
do {
  write(f) ;
  if (b):  read(f);
  else:  close(f);
} while (b)
open(f);
close(f);
```

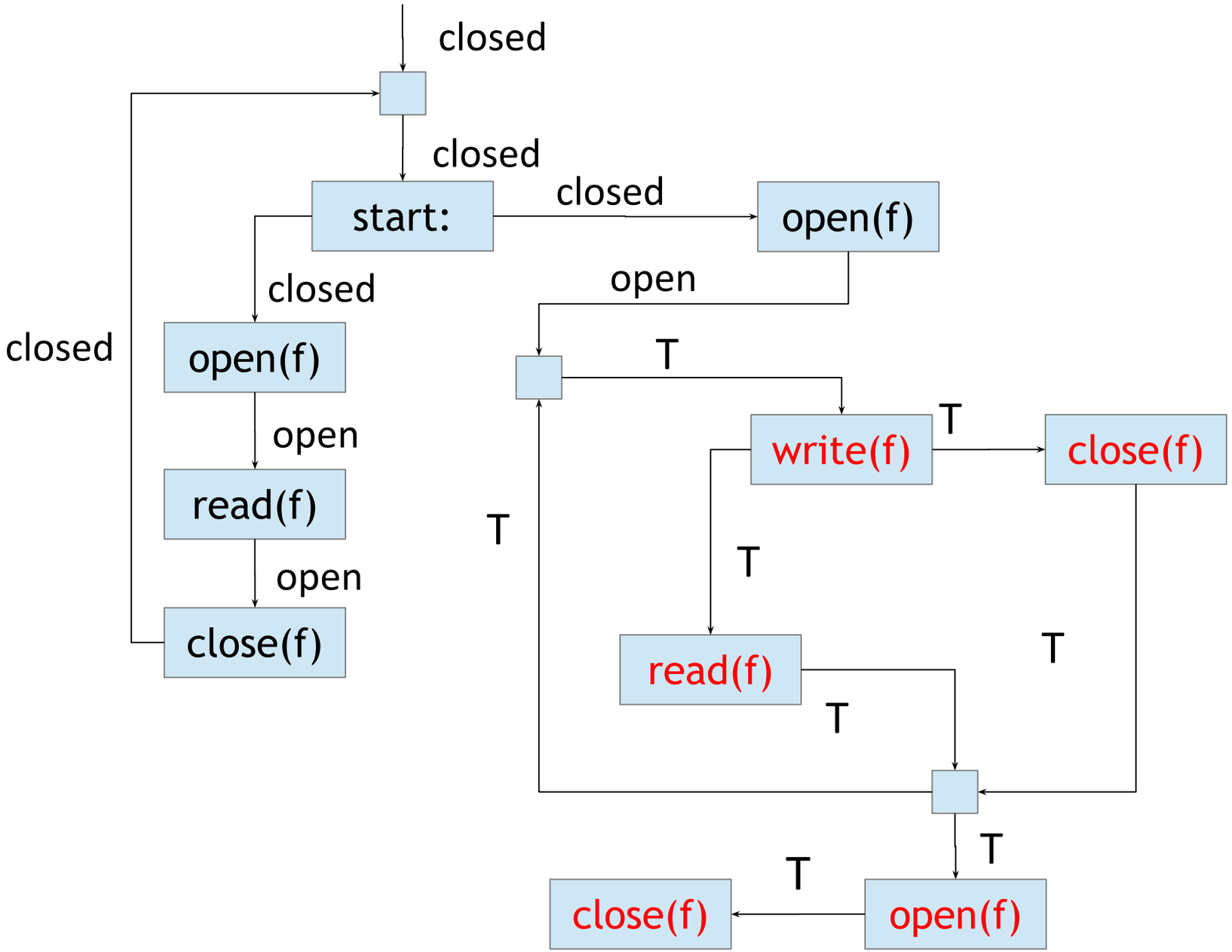
```
start:
switch (a)
  case 1: open(f);
          read(f);
          close(f);
          goto start
  default: open(f);
do {
  write(f) ;
  if (b):  read(f);
  else:    close(f);
} while (b)
open(f);
close(f);
```




closed
closed
start:
closed
open(f)
closed
open
closed
open(f)
open
open
write(f)
open
close(f)
open
read(f)
⊥
open
read(f)
closed
open
⊥
⊥
close(f)
open(f)


closed
closed
start:
closed
open(f)
closed
closed
open
open(f)
open
open
read(f)
open
close(f)
write(f)
close(f)
open
T
read(f)
closed
open
T
T
close(f)
open(f)


closed
closed
start:
closed
open(f)
closed
open
closed
open(f)
T
open
read(f)
T
write(f)
close(f)
open
T
T
read(f)
T
T
T
close(f)
open(f)


closed
closed
start:
closed
open(f)
closed
closed
open(f)
open
read(f)
open
close(f)
open
T
T
write(f)
close(f)
T
T
read(f)
T
T
T
close(f)
T
open(f)
T

Is There Really A Bug?

```
start:
switch (a)
  case 1: open(f); read(f); close(f); goto start
  default: open(f);
do {
  write(f) ;
  if (b): read(f);
  else: close(f);
} while (b)
open(f);
close(f);
```

Forward vs. Backward Analysis

- We've seen two kinds of analysis:

- Definitely null (cf. constant propagation) is a **forwards** analysis: information is pushed from inputs to outputs

- Secure information flow (cf. liveness) is a **backwards** analysis: information is pushed from outputs back towards inputs