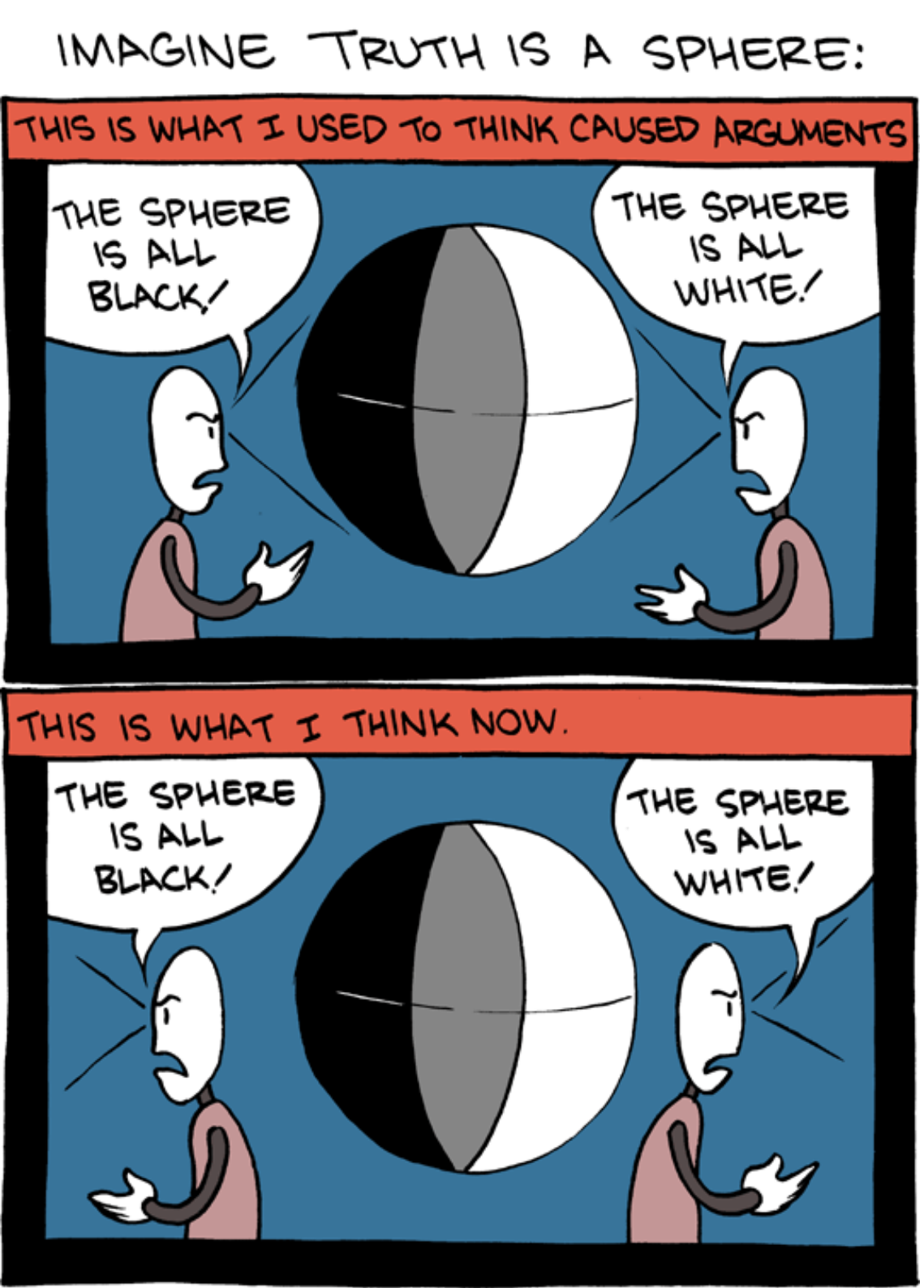


Static and Dataflow Analysis

(two part lecture)

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 the most common static approaches to QA.
- What other static analyses are commonly used and how do they work?

BUT FIRST, logistics ...

Logistics

- Exam Accommodations: Fill out our second confirmation form, not just the SSD letter, by midnight Wed 10/1:

481 SSD Exam Time Confirmation Form — after you have (1) obtained approval from SSD, then (2) complete this form to ensure you receive extra exam time

- https://forms.gle/dQNrGvde4jFaziUq8
- Exam #1 does cover Dataflow Analysis
  - Exam #1 covers both today's material and Thursday's material
  - It also allows you to use ChatGPT, course recordings, Stack Overflow, etc. See webpage!

Exam Course Staff Availability

- Your exam must be submitted by midnight eastern (so don't start a 2-hour exam at 11pm)
- Course staff will be available monitoring Piazza at these times:

Piazza Monitoring: Exam 1 10/3 (8AM-11PM)

			FILLED	FILLED	FILLED	FILLED	FILLED	FILLED	
Name	**IA/GSI**		**8-10:30AM**	**10:30-1PM**	**1-3:30PM**	**3:30-6PM**	**6-8:30PM**	**8:30-11PM**	**Selected 2 Slots**
Priscila	GSI		☑	☐	☐	☐	☐	☑	Done
Hanchi	GSI		☐	☐	☑	☑	☐	☐	Done
Rohit	GSI		☐	☐	☐	☐	☑	☑	Done
Derek	IA		☑	☑	☐	☐	☐	☐	Done
Sathvika	GSI		☐	☐	☐	☑	☑	☐	Done
Jesse	GSI		☐	☑	☑	☐	☐	☐	Done
2 slots per person (5 hours per person)			2	2	2	2	2	2	

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

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

What effect does **human** pair programming have on coding speed and defect density?

+41%
IN BUG RATE

Key Insight:

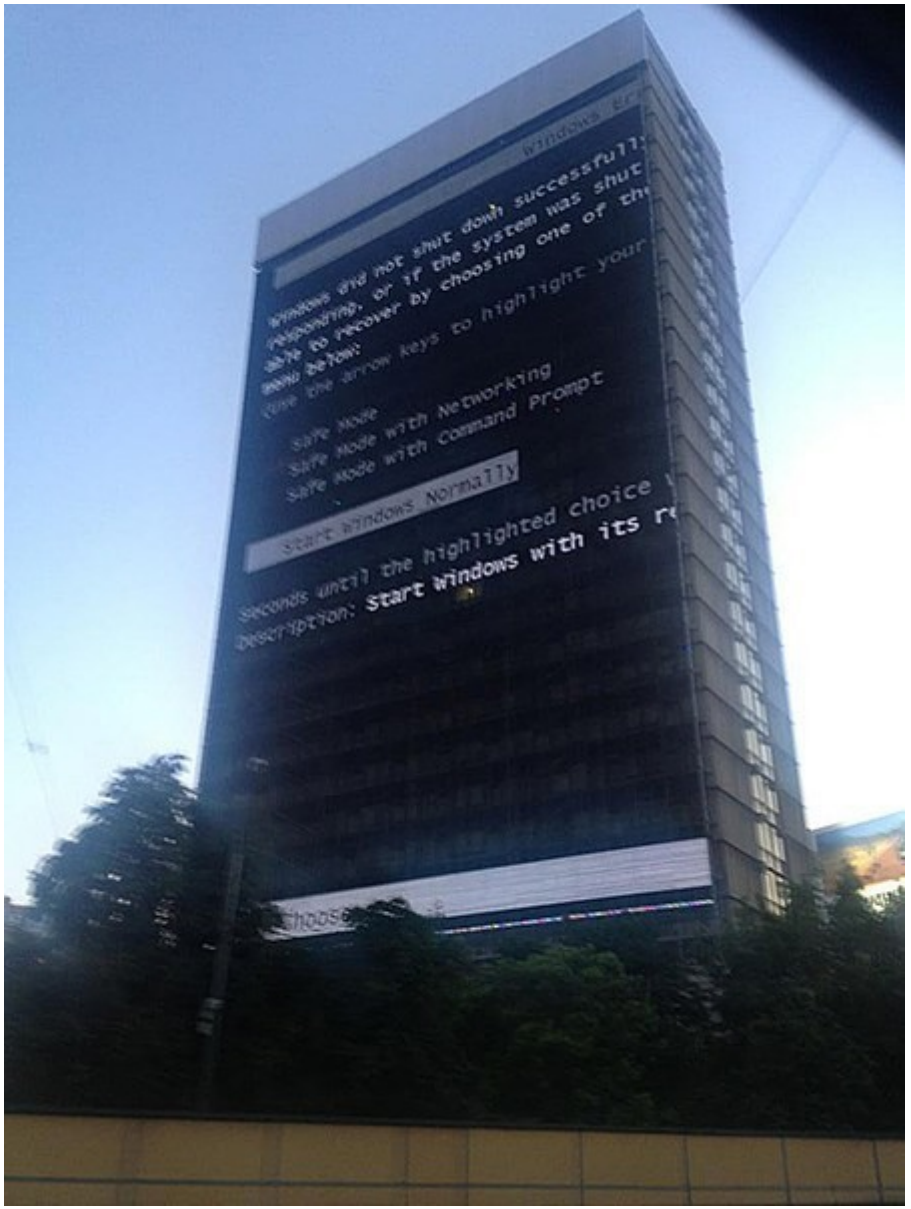
Developers with Copilot access saw a significantly higher bug rate while their issue throughput remained consistent.

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.

Fundamental Concepts

- Abstraction
  - Capture semantically-relevant details
  - Elide 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;


Safe Mode
Safe Mode with Networking
Safe Mode with Command Prompt
Start Windows Normally
Description: Start windows with its

“Unimportant” SSL 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;
}
```

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 ("get lock")
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // enables ints (unlock)
  return bh;
}
```

Could We Have Found Them?

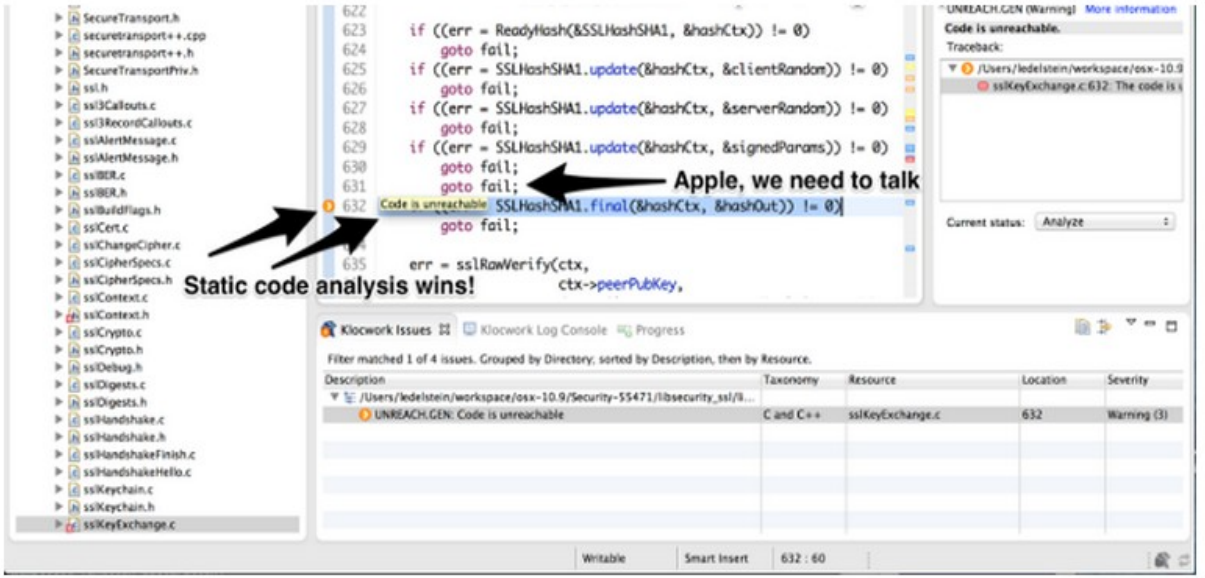
- How often would those bugs trigger?
- Linux example:
  - What happens if you return from a device driver with interrupts disabled?
  - Consider: that's just one function
    - … in a 2,000 LOC file
    - … in a 60,000 LOC module
    - … in the Linux kernel
- Some defects are very difficult to find 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

Follow



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

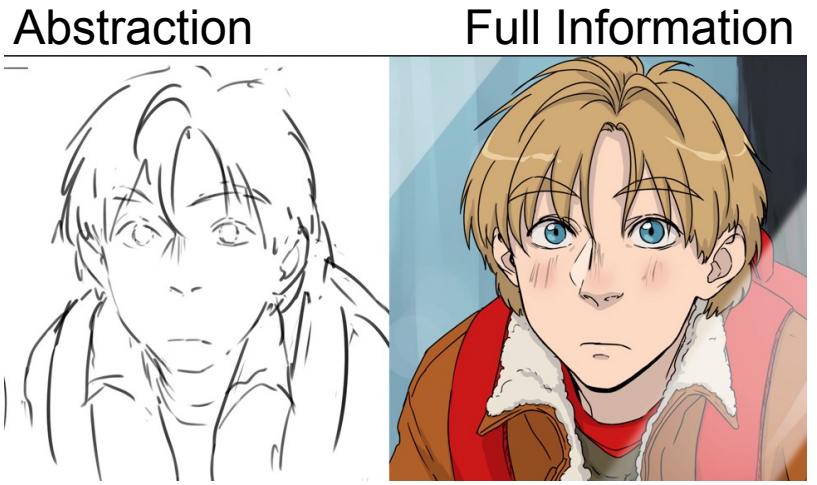
How And Where Should We Focus?

Static Analysis

- **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”

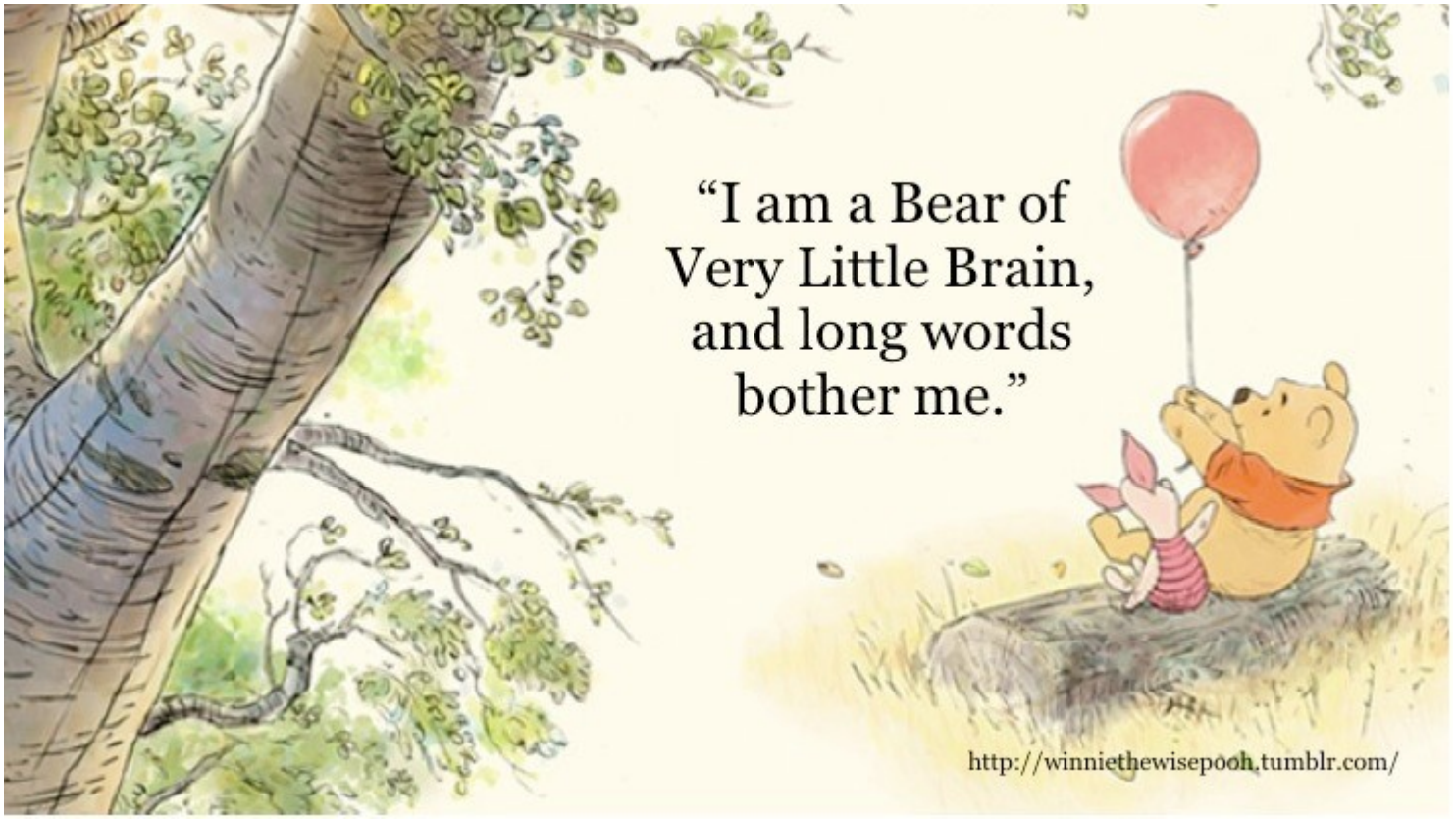
Let's Act It Out

- We're going to add some large numbers
- We'll keep track of the real answer
- But the Dataflow Analysis will only track the fact of whether the running total is **odd** or **even**
  - This is an abstraction!

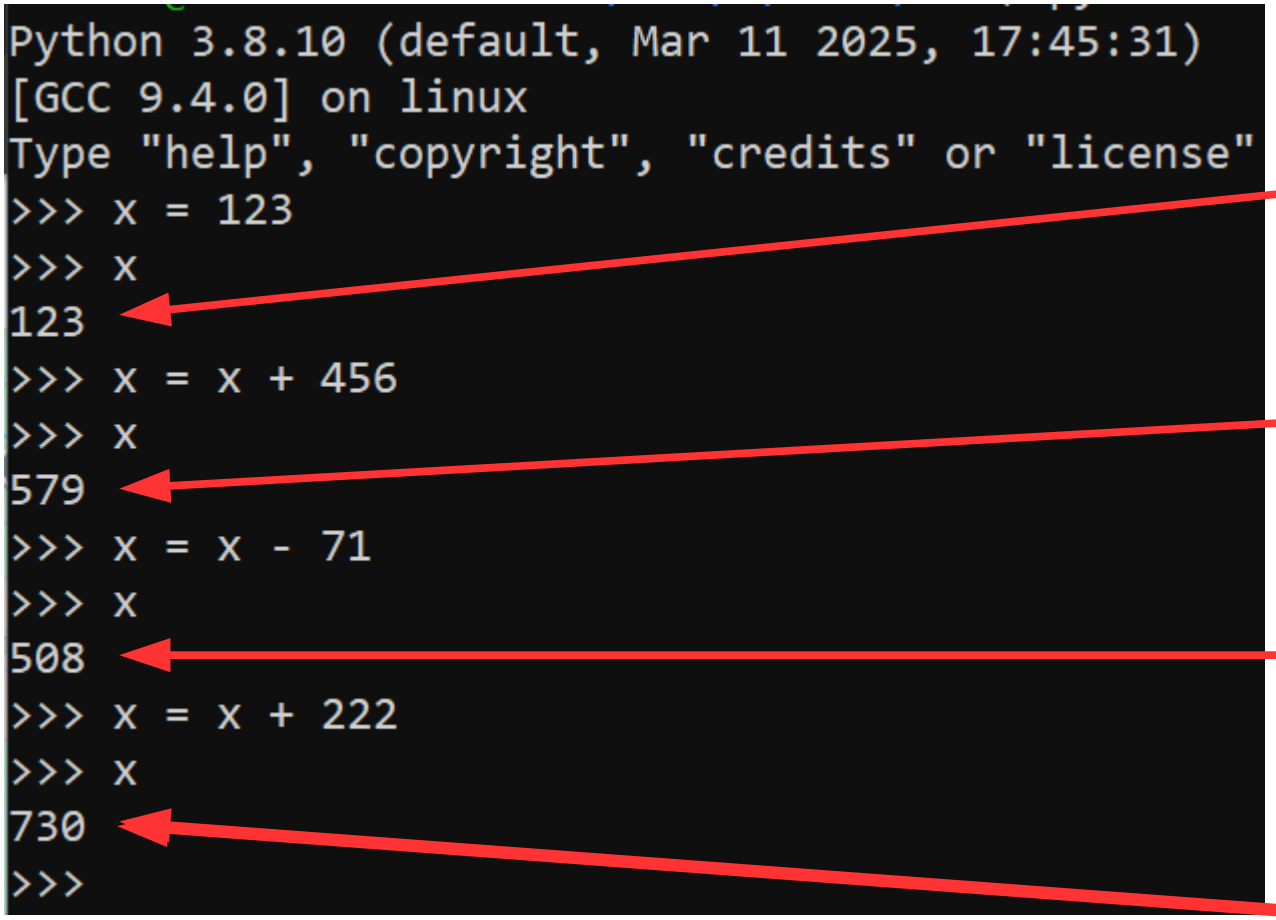


Analogy

- 4294967296 and 8589934591 are too long to track precisely.
- *even* and *odd* are short! Dataflow likes short.



Abstraction Example
Adding Big Numbers



But How Do We Track Facts In Program Source Code?

- Example: track every call to logger.debug()

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

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

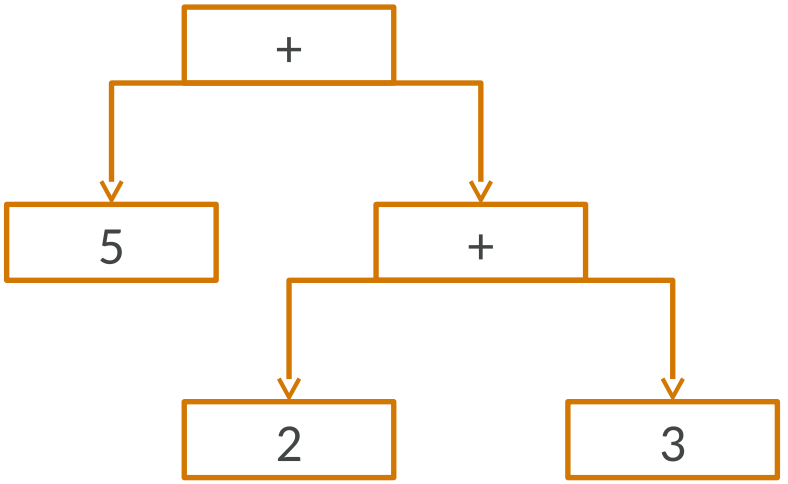
- What could go wrong? First attempt:

```
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.

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
- It relates to the notion of a Universal Turing Machine. A Turing Machine description (finite state controller, initial tape) can itself be represented as a string (and thus placed on a tape as 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

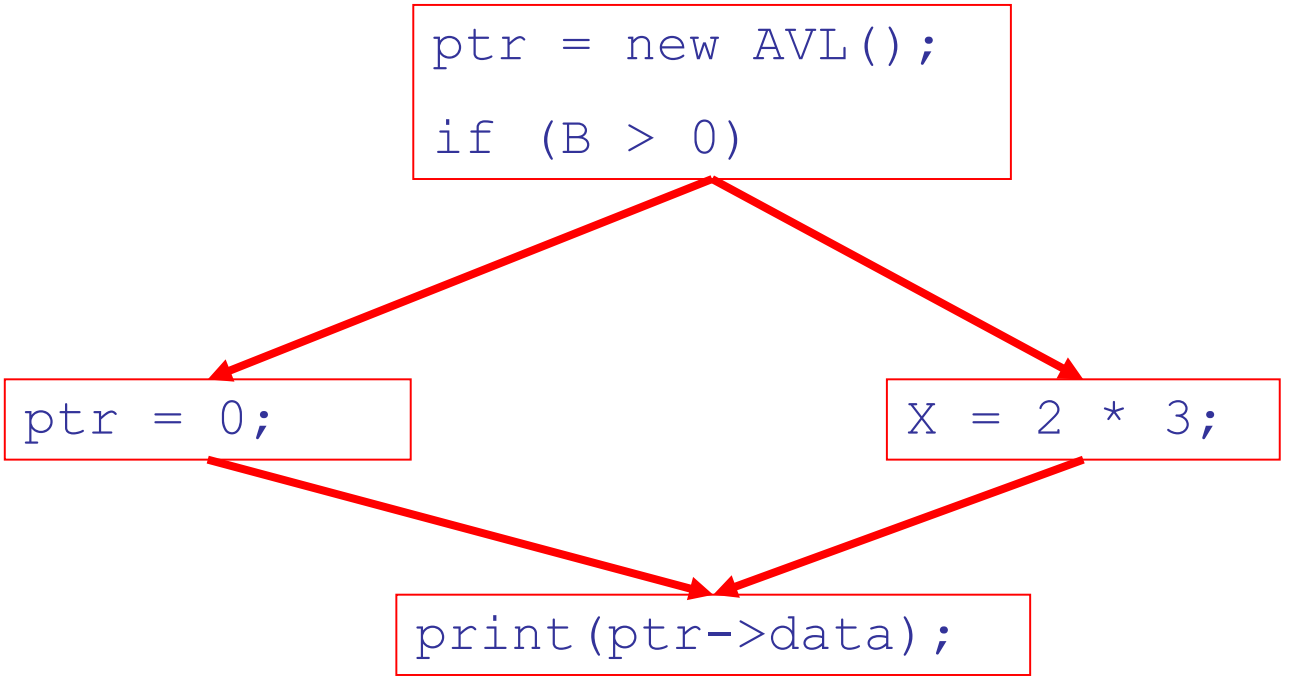
- *Definite* Null Dereference
  - "Whenever execution reaches *ptr at program location L, ptr will be NULL"
- *Potential* Secure Information Leak
  - "We read in a secret string at location L, but there is a possible future public use of it"

Discussion

- These analyses are not trivial to check
- "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

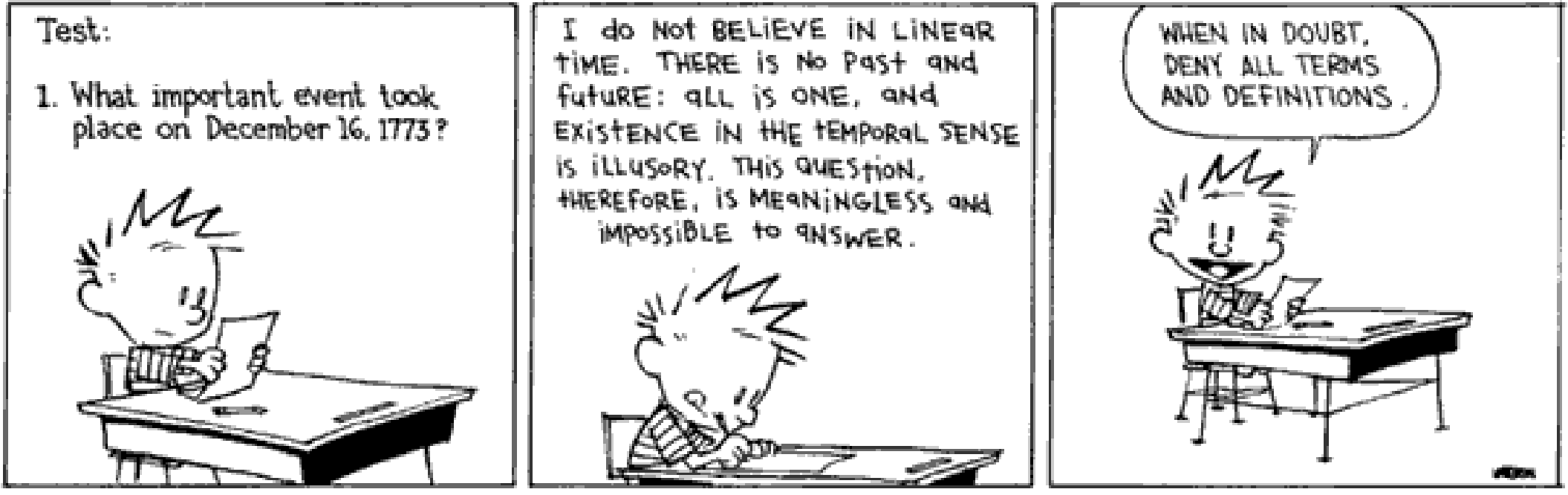
Is **`ptr`** *always* null when it is dereferenced?



Correctness (Cont.)

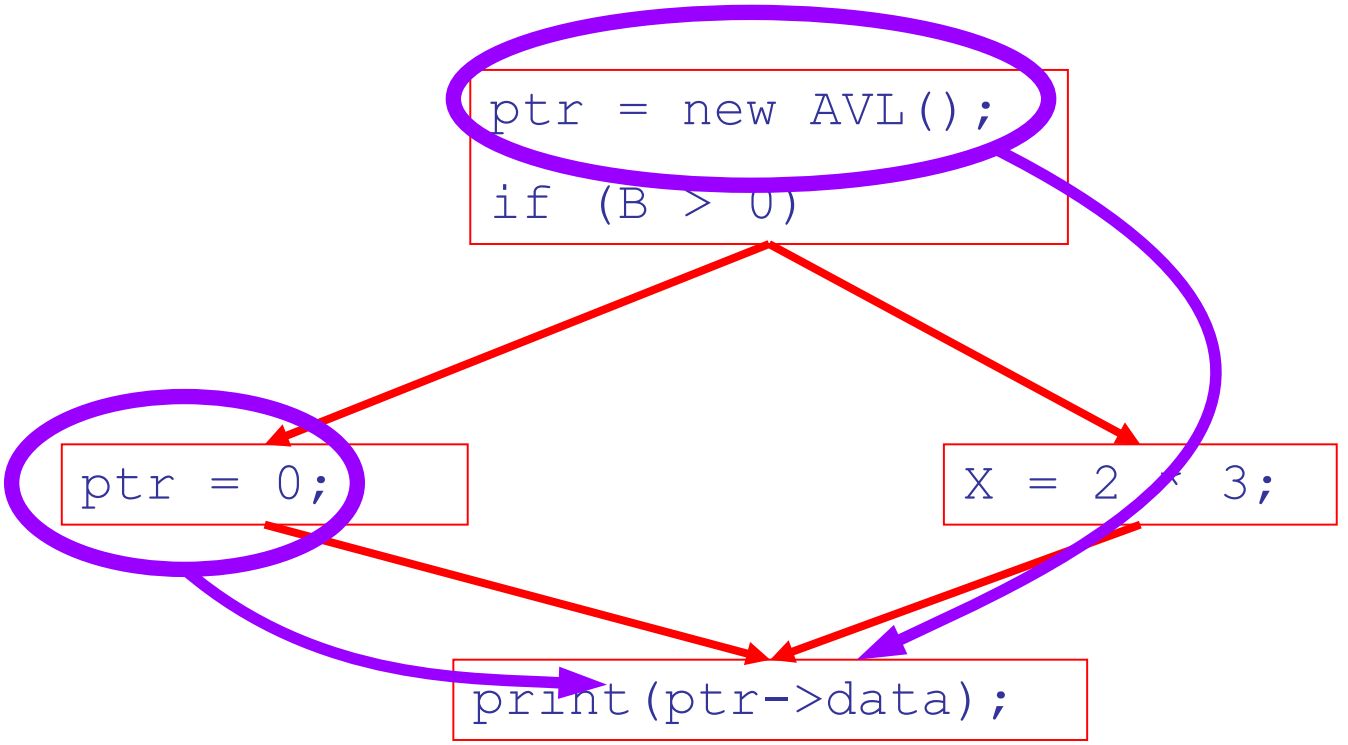
To determine that a use of x is always null, we must know this **correctness condition**:

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



Analysis Example Revisited

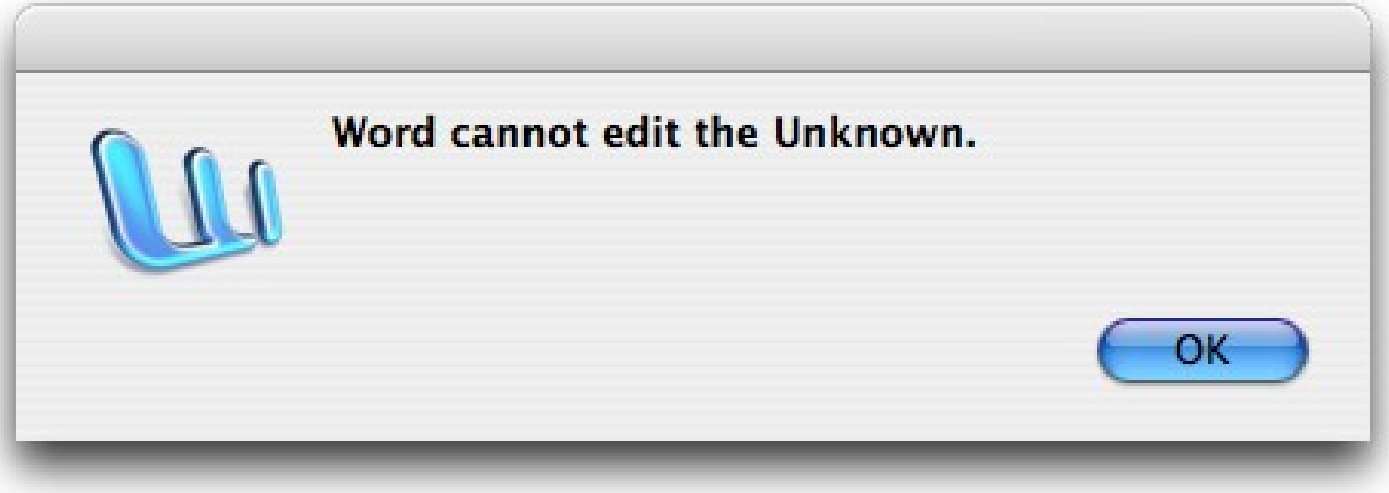
Is **`ptr`** *always* null when it is dereferenced?



Static Datalfow Analysis

Static dataflow analyses share several traits:

- The analysis 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*!**



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!*
- Syntactic properties are decidable!
  - e.g., How many occurrences of "x" are there?
- Programs without looping are also decidable!

Looping

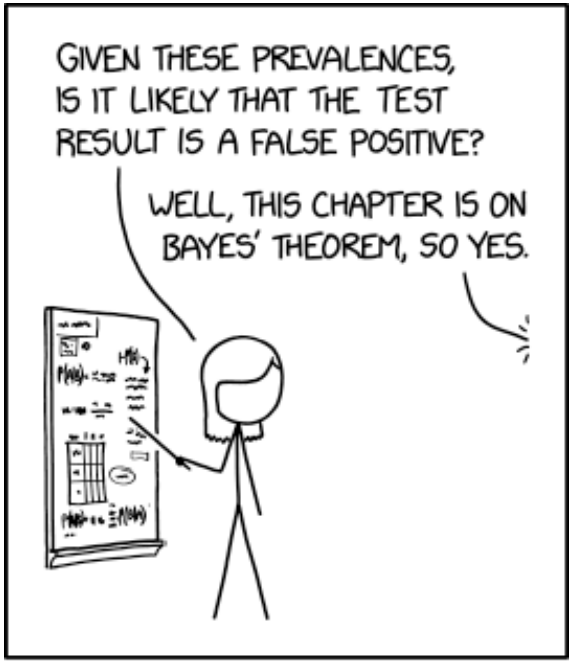
- Almost every important program has a loop
  - Often based on user input
- An algorithm always terminates
- So a dataflow analysis algorithm must terminate even if the input program loops
- This is one source of imprecision
  - Suppose you dereference the null pointer on the 500th iteration but we only analyze 499 iterations

Conservative Program Analyses

- We cannot tell for sure that `ptr` is always null (Rice)
  - So how can we carry out any sort of analysis?
- It is OK to be **conservative**.

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 analysis depends on whether or not P is true, then want to know either
  - P is definitely true
  - Don't know if P is true
- This is called a **conservative approximation**

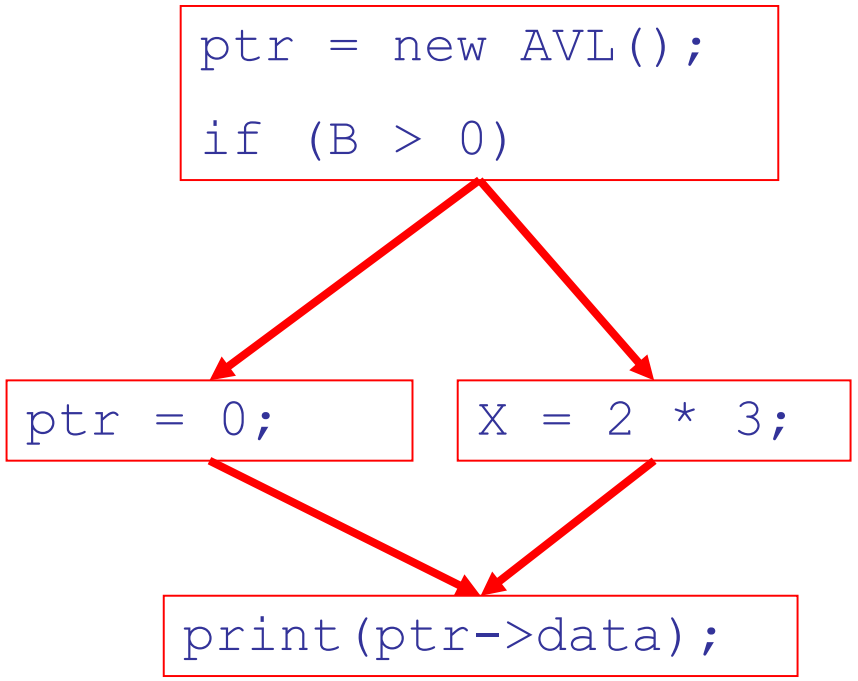


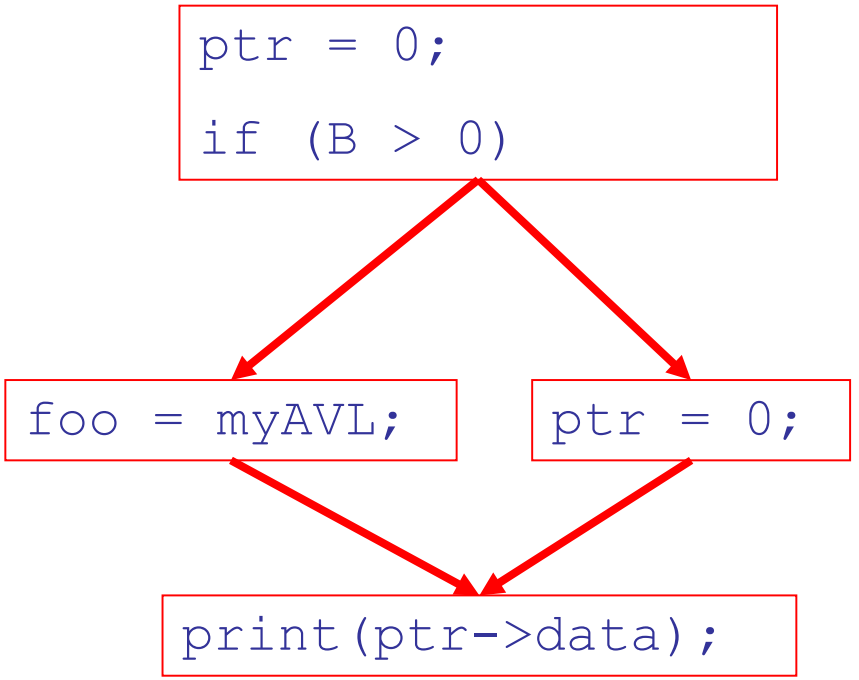
Conservative Program Analyses

- It is always correct to say “I don’t know”
  - We try to say don’t know as rarely as possible
- All program analyses are conservative
  - Never Claim safe unless sure
- **But whether we actually say ‘don’t know’ depends on context.**
  - 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

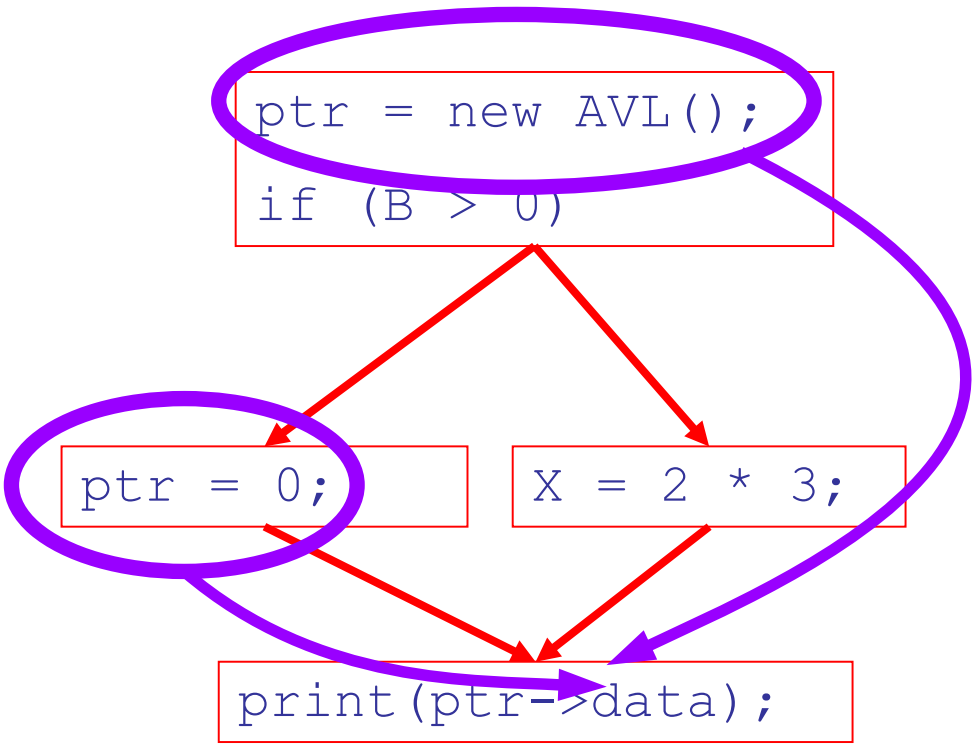
Is **`ptr`** *always* null when it is dereferenced?

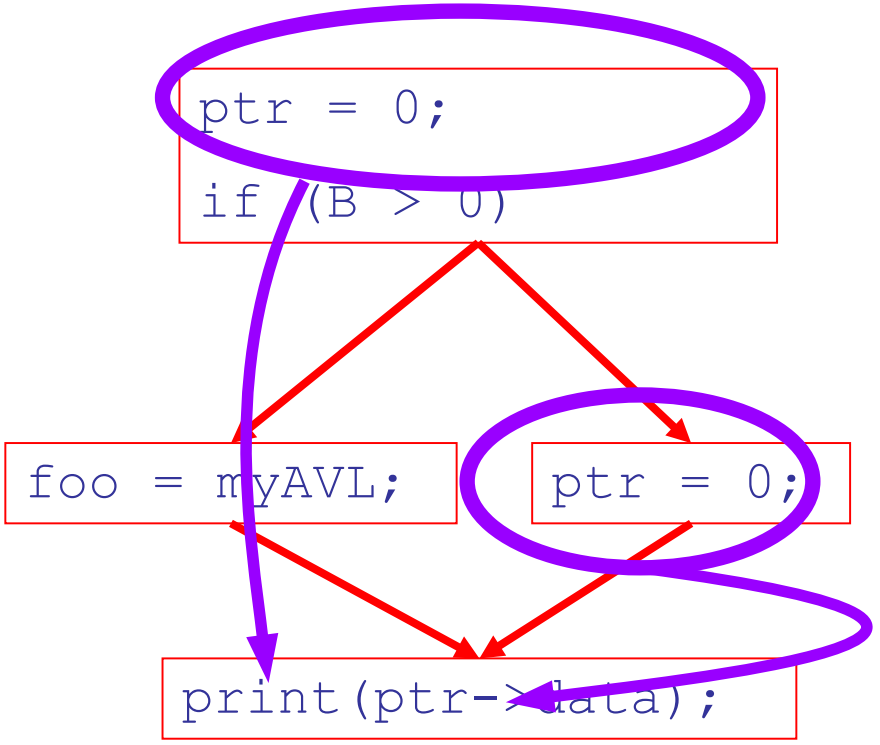




Definitely Null Analysis

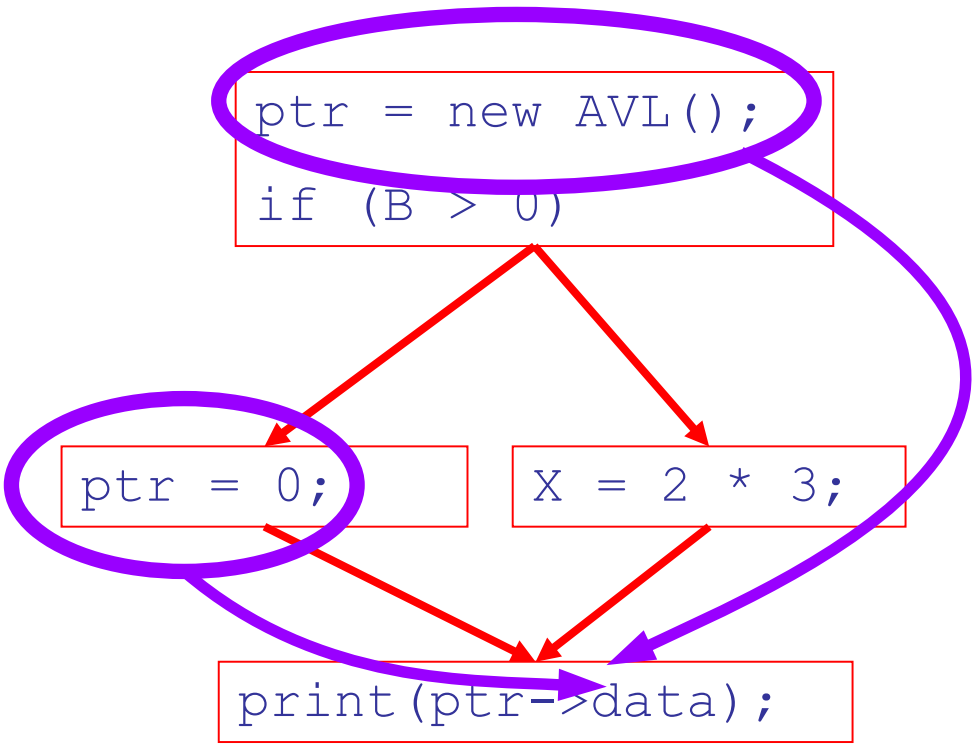
Is **`ptr`** *always* null when it is dereferenced?



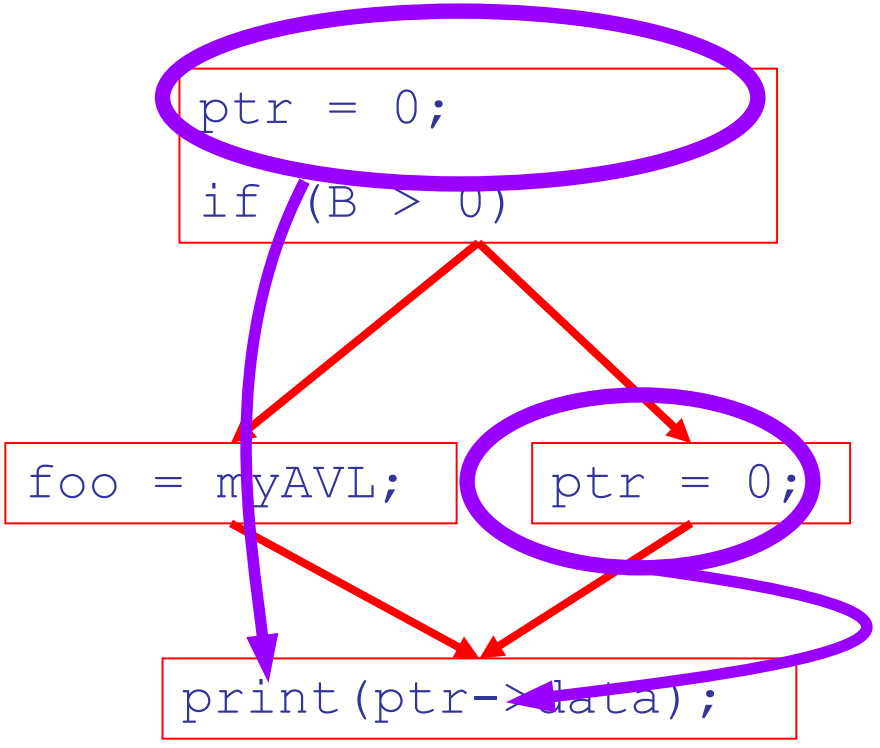


Definitely Null Analysis

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



No, not always.

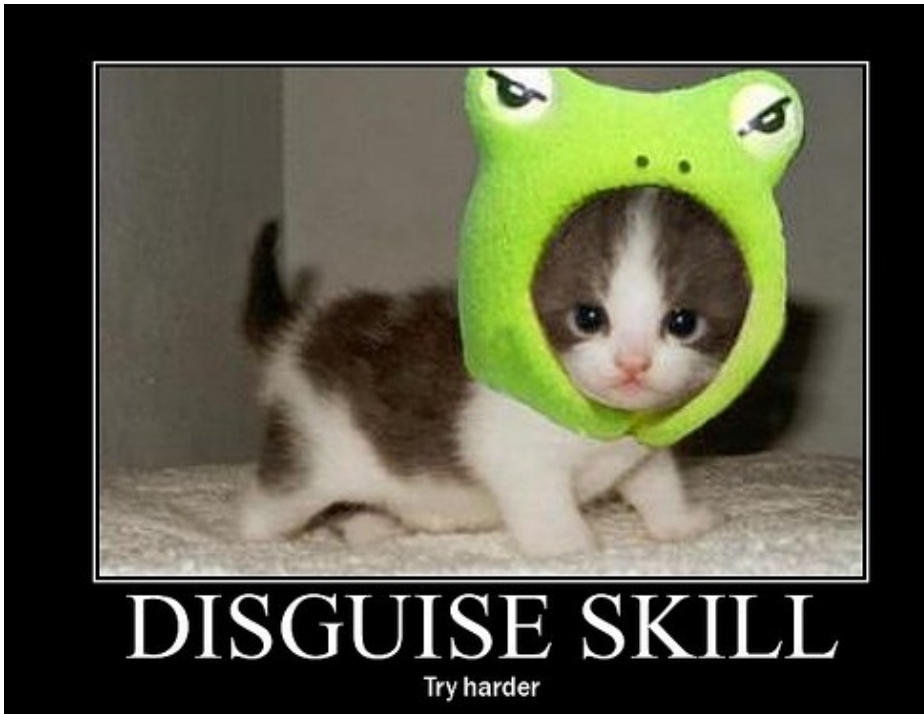


Yes, always.

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
- Consider the case of computing ** for a single variable ptr at all program points
- Valid points cannot hide!
- We will find you!
  - *(sometimes)*



Abstracting “Don't Know”

(Priscila's Revenge on Wes)

```
Python 3.8.10 (default, Mar 11 2025, 17:45:31)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license"
>>> import random
>>> x = 123
>>> x
123
>>> x = x + random.randint(1,100)
>>> x
133
>>> x = x - 7
>>> x
126
>>> x = 456
>>> x
456
>>>
```

1st Fact: x is odd

2nd Fact: x is ???

3rd Fact: x is ???

4th Fact: x is even

Abstracting “Not There Yet”

(Continuing Revenge)

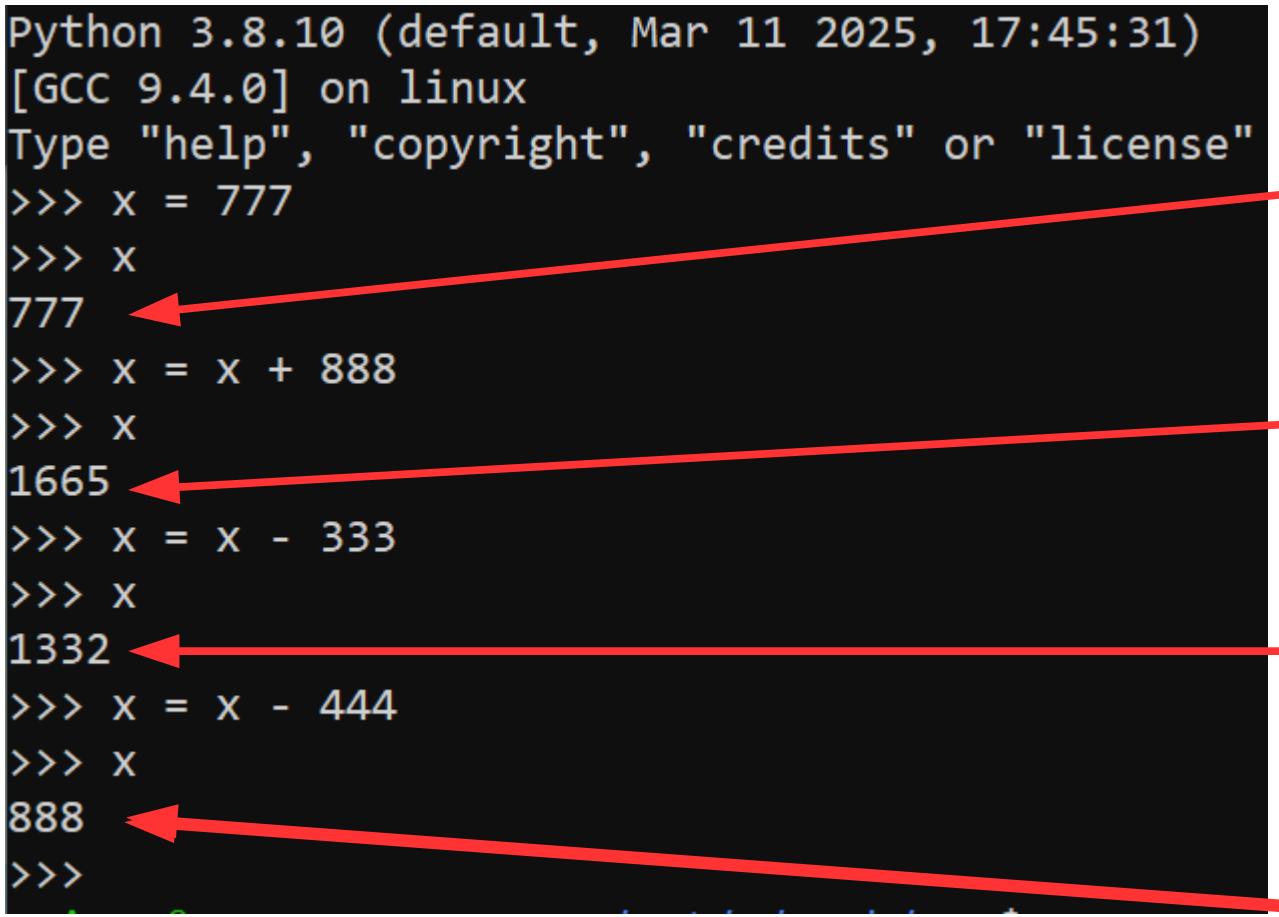
```
Python 3.8.10 (default, Mar 11 2025, 17:45:31)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license"
>>> x = 777
>>> x
777
>>> x = x + 888
>>> x
1665
>>> x = x - 333
>>> x
1332
>>> x = x - 444
>>> x
888
>>>
```

3rd Fact: x is “not there yet”

4th Fact: x is “not there yet”

Abstracting “Not There Yet”

(Updating Prior Answers)



1st Fact: x is odd

2nd Fact: x is odd

3rd Fact: x is ~~“not there yet”~~ even

4th Fact: x is ~~“not there yet”~~ even

Note how prior answers can get updated!

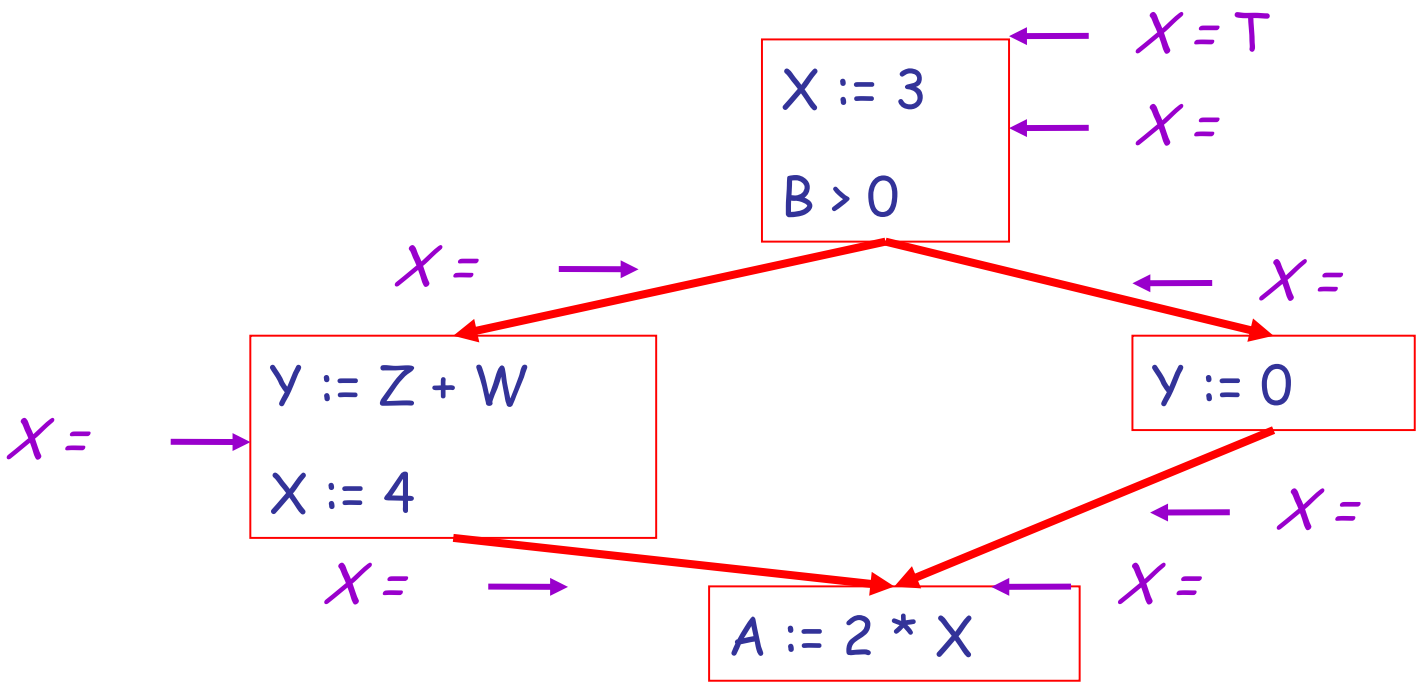
Definitely Null Analysis (Cont.)

- To make the problem precise, 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 (not there yet)
c	X = constant c
T (called “top”)	Don’t know if X is a constant (unknown)

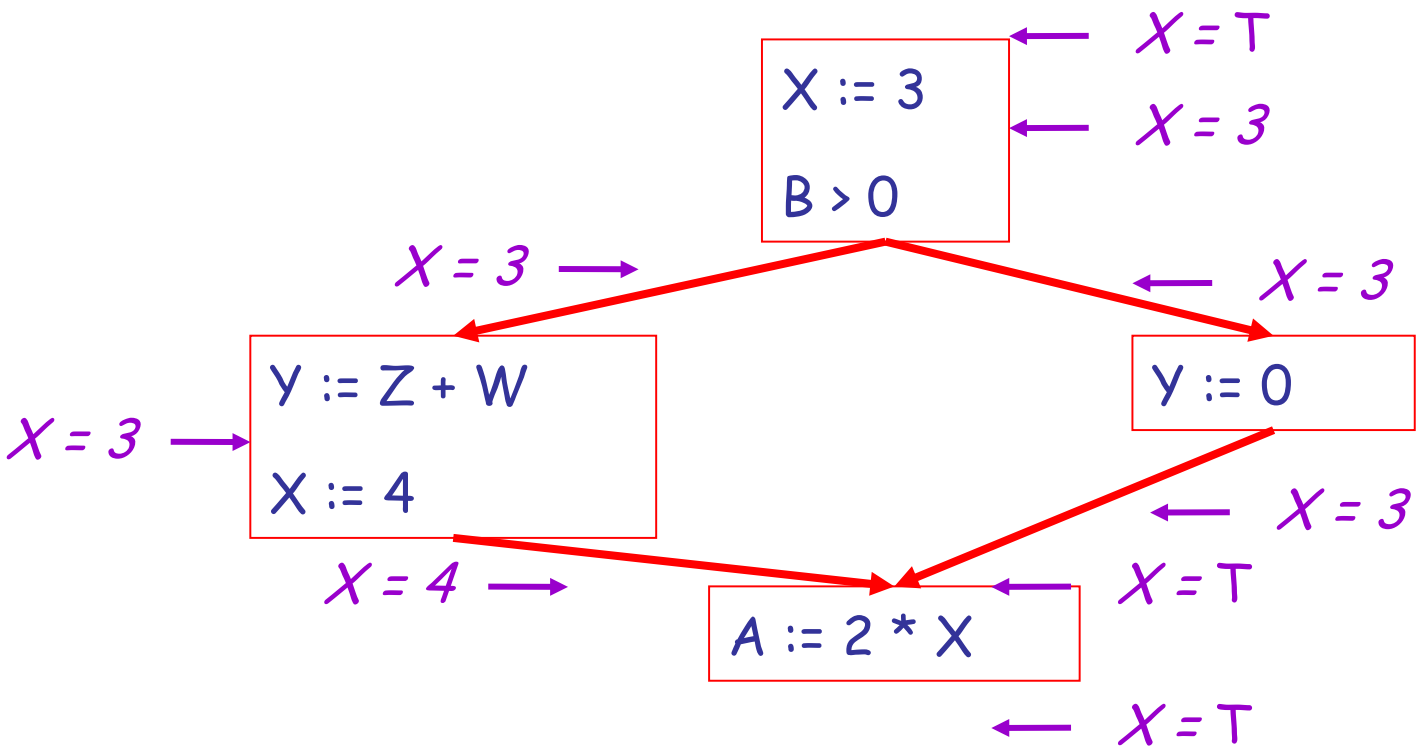
Example

Get out a piece of paper. Let's fill in these blanks now.



Recall: ⊥ = not reachable / not there yet, c = constant (write the **number**, don't write “c”), T = don't know.

Example Answers



Anime / Movie

- This anime, adapted from a manga by Koyoharu Gotouge, follows siblings whose family is slaughtered by demons, leaving the sister transformed into one herself. Its *Mugen Train* movie grossed over $506 million worldwide, making it the highest-grossing anime film at the time, though it has since been surpassed by the franchise's *Infinity Castle* film, which crossed $600 million globally.

Real-World Languages

- The official language of Sri Lanka and Singapore is spoken by over 66 million and boasts a rich literature stretching back over 2000 years. Unlike most Indian languages, it does not distinguish between aspirated and unaspirated consonants. It uses suffices to mark number, case and verb tense and uses a flexible S-O-V ordering. It uses *post*positions rather than prepositions.
  - Example: வணக்கம்

Fictional Magicians

- In Greek Mythology, this sorceress transforms her enemies into animals. In Homer's *Odyssey* she tangles with Odysseus (who defeats her magic); she ultimately suggests that he travel between Scylla and Charybdis to reach Ithaca.

Psychology: Predictions

- You are asked to read about a conflict and are given two alternative ways of resolving it.
- You are then asked to do:
  - Say which option you would pick
  - Guess which option other people will pick
  - Describe the attributes of a person who would choose each of the two options
- (Actually, let's be more specific …)

Psychology: Prediction

- Would you be willing to walk around campus for 30 minutes holding a sign that says “Eat at Joe's”?
  - (No information about Joe's restaurant is provided, you are free to refuse, but we claim you will learn “something useful” from the study.)

- Would you do it?

Psychology: False Consensus Effect

- Of those who agreed to carry the sign, 62% thought others would also agree
- Of those who refused, 67% thought others would also refuse
- *We think others will do the same as us, regardless of what we actually do*
  - We make extreme predictions about the personalities of those who chose differently
    - But choosing “like me” does not imply anything: it's common!
  - “Must be something wrong with you!”

Psychology: False Consensus Effect

- Replications with 200 college students, etc.
  - [Kathleen Bauman, Glenn Geher. WE think you agree: the detrimental impact of the false consensus effect on behavior. J. Current Psychology, 2002, 21(4).]
- Implications for SE: Myriad, whenever you design something someone else will use. Example: Do you think this static analysis should report possible defects or certain defects? By the way, what do you think the majority of our customers want?

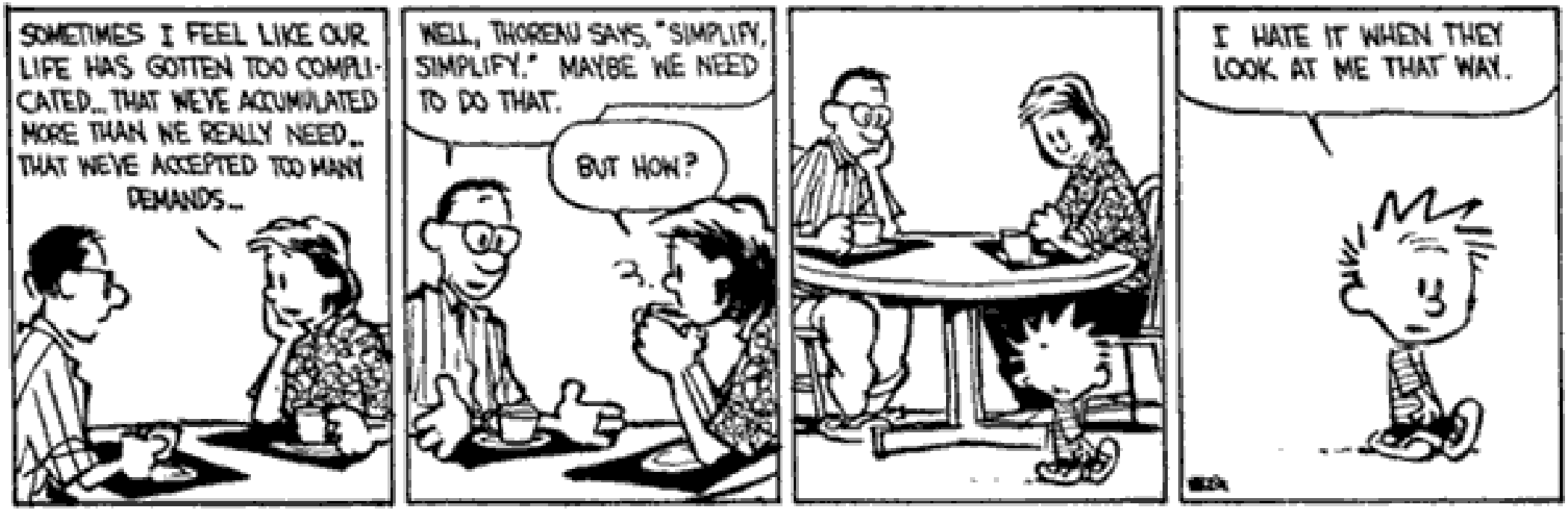
Using Abstract Information

- 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!

- But 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

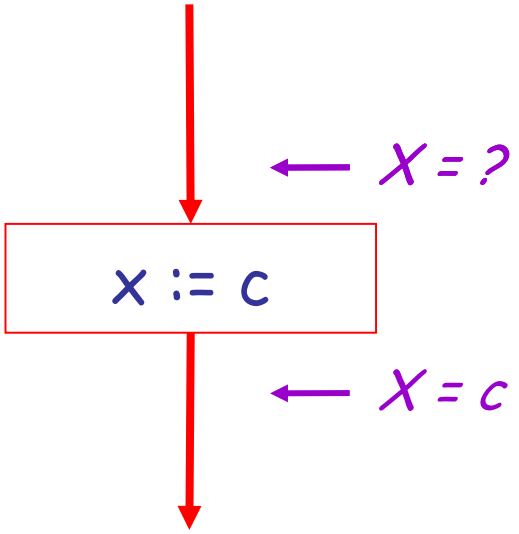
$$C_{in}(x,s) = \text{value of } x \text{ before } s$$

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

Transfer Functions

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

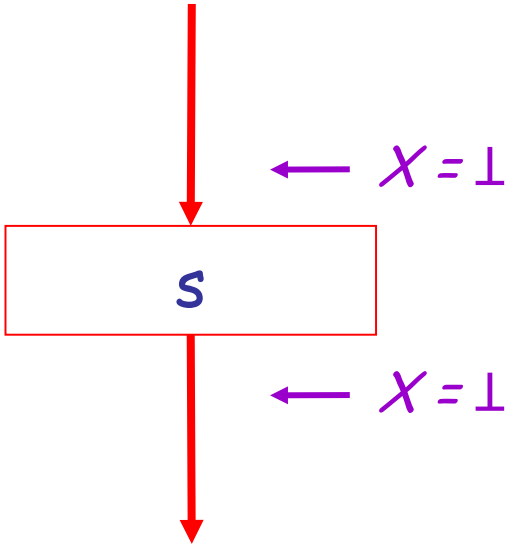
Rule 1



(does it matter what x is here?)

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

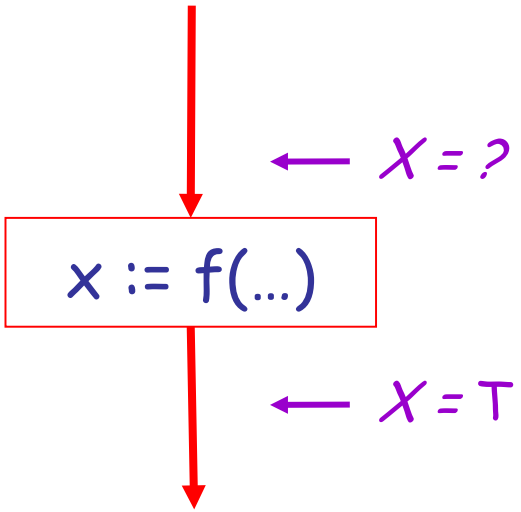
Rule 2



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

Recall: ⊥ = “unreachable code or 'haven't analyzed this yet'”

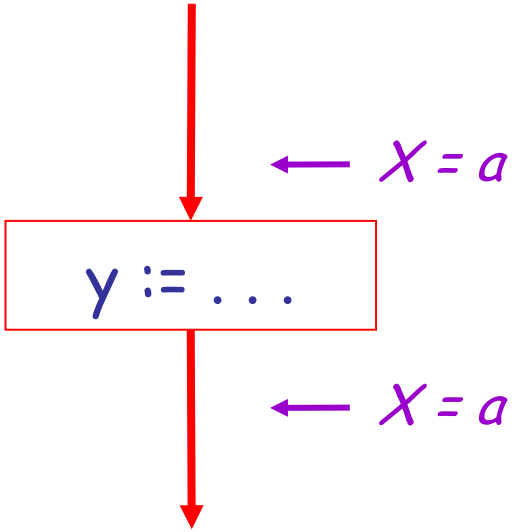
Rule 3



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

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

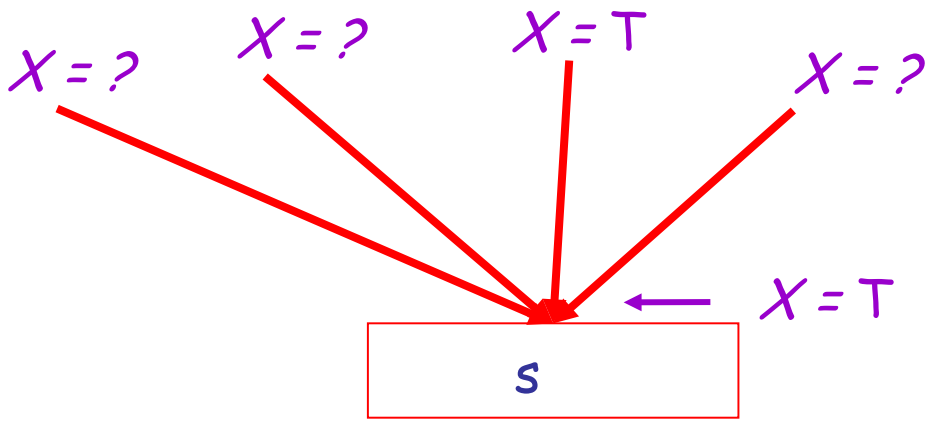


$C_{out}(x,\ y := \ldots) = C_{in}(x,\ y := \ldots)$ if $x \neq y$

The Other Half

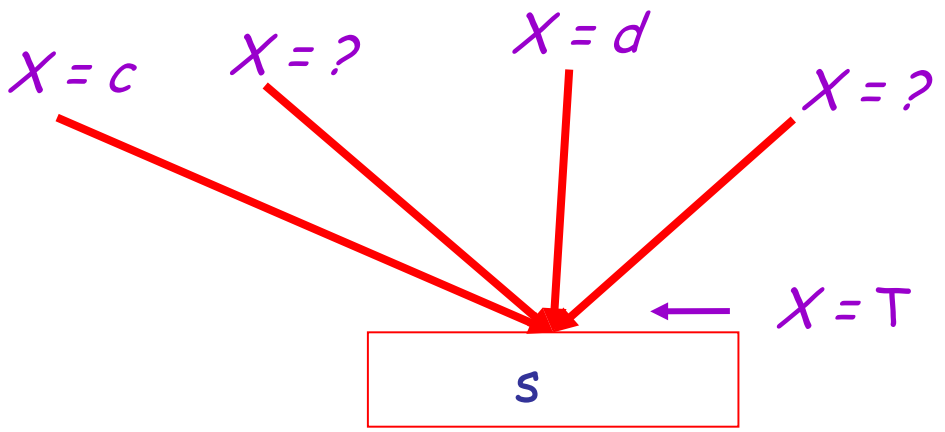
- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information across statements
- Now we need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information forward along paths
- In the following rules, let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

Rule 5



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

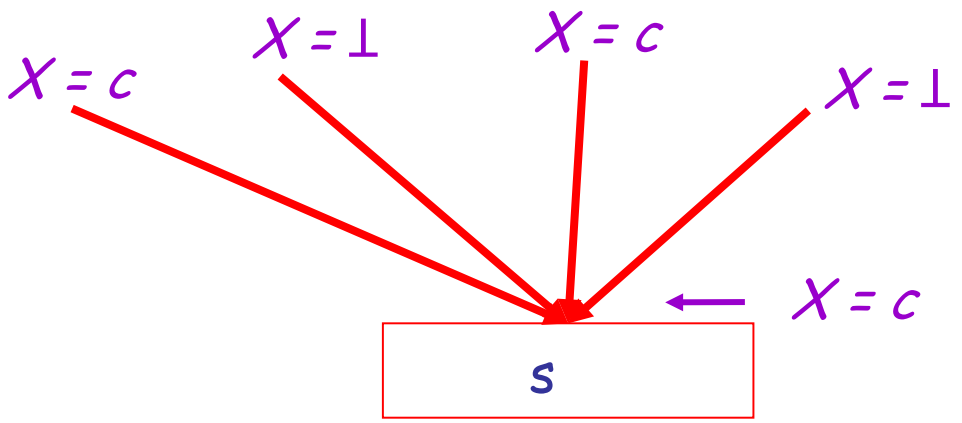
Rule 6



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

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

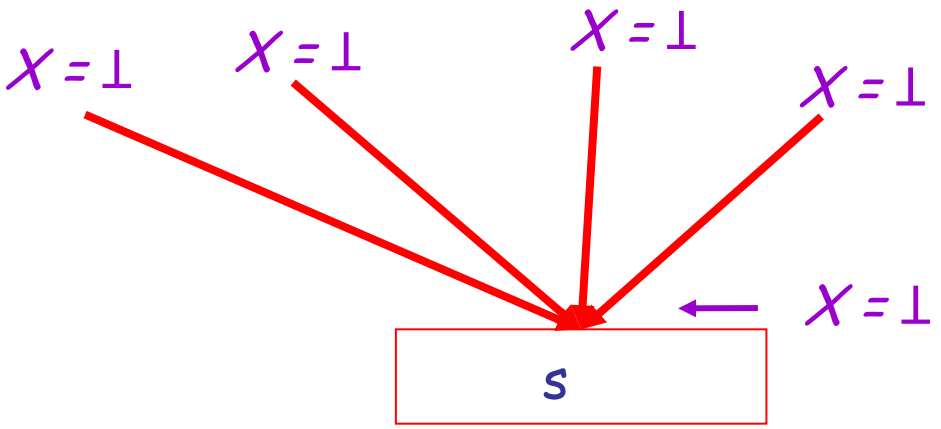
Rule 7



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

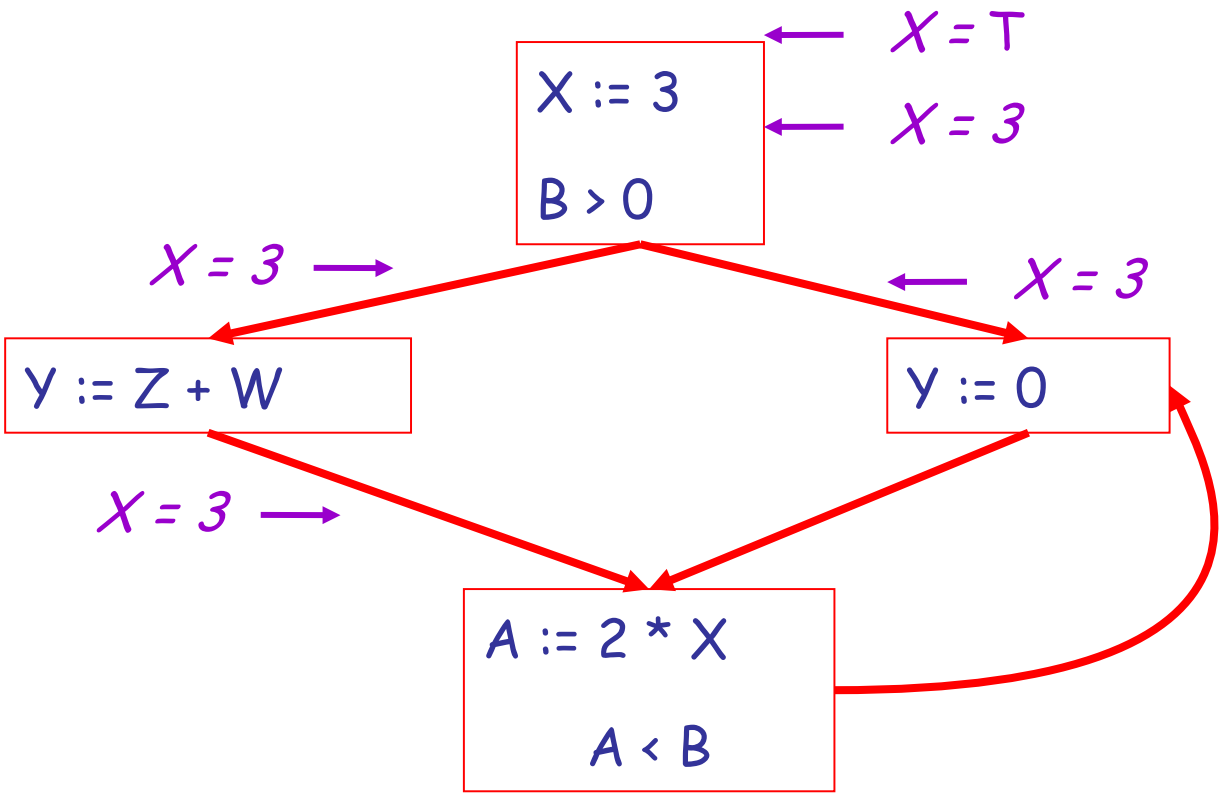
Static Analysis Algorithm

- For every entry s to the program, set $C_{in}(x, s) = T$
- Set $C_{in}(x, s) = C_{out}(x, s) = \perp$ everywhere else
- Repeat until all points satisfy 1-8:

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

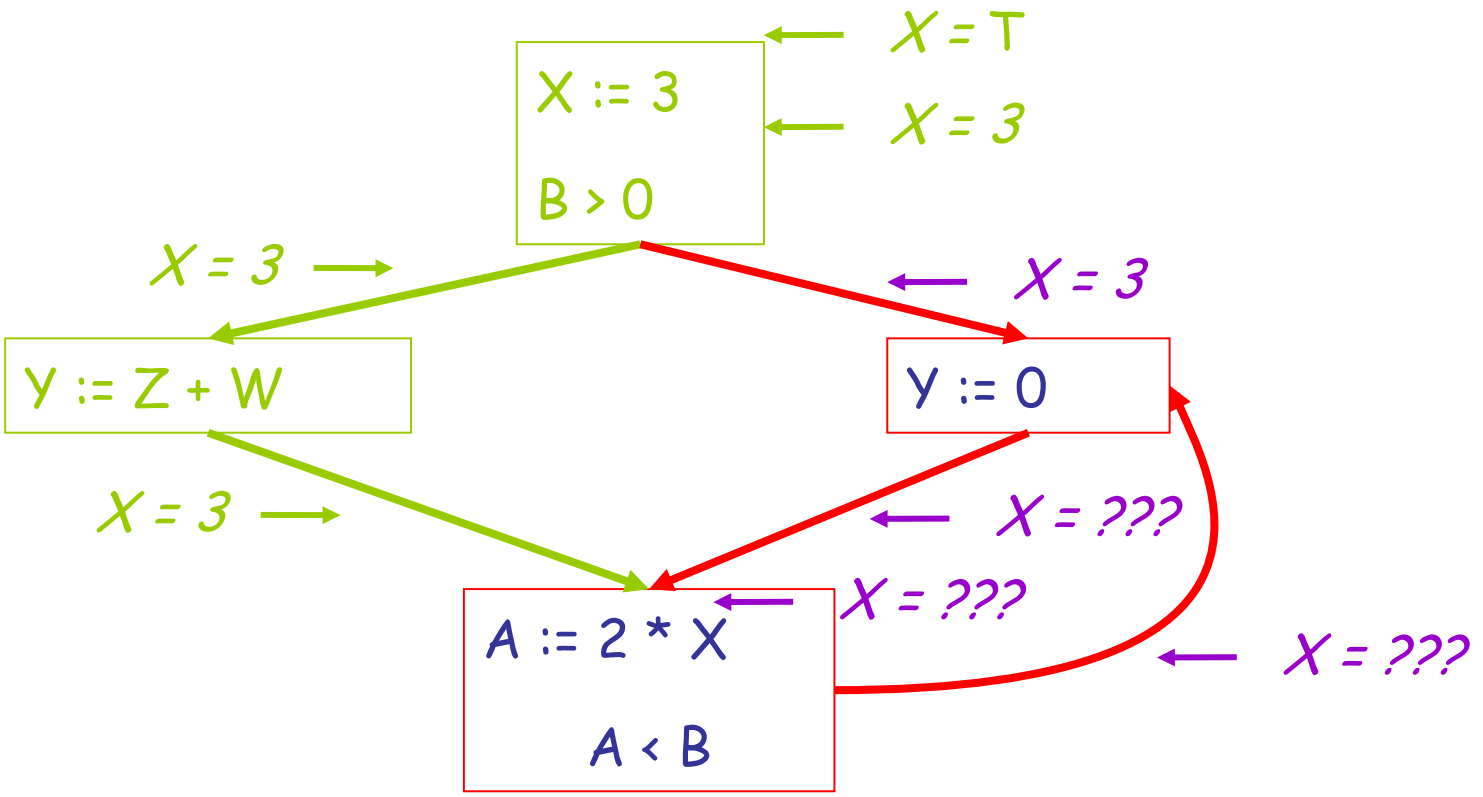
The Value ⊥

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



The Value ⊥

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



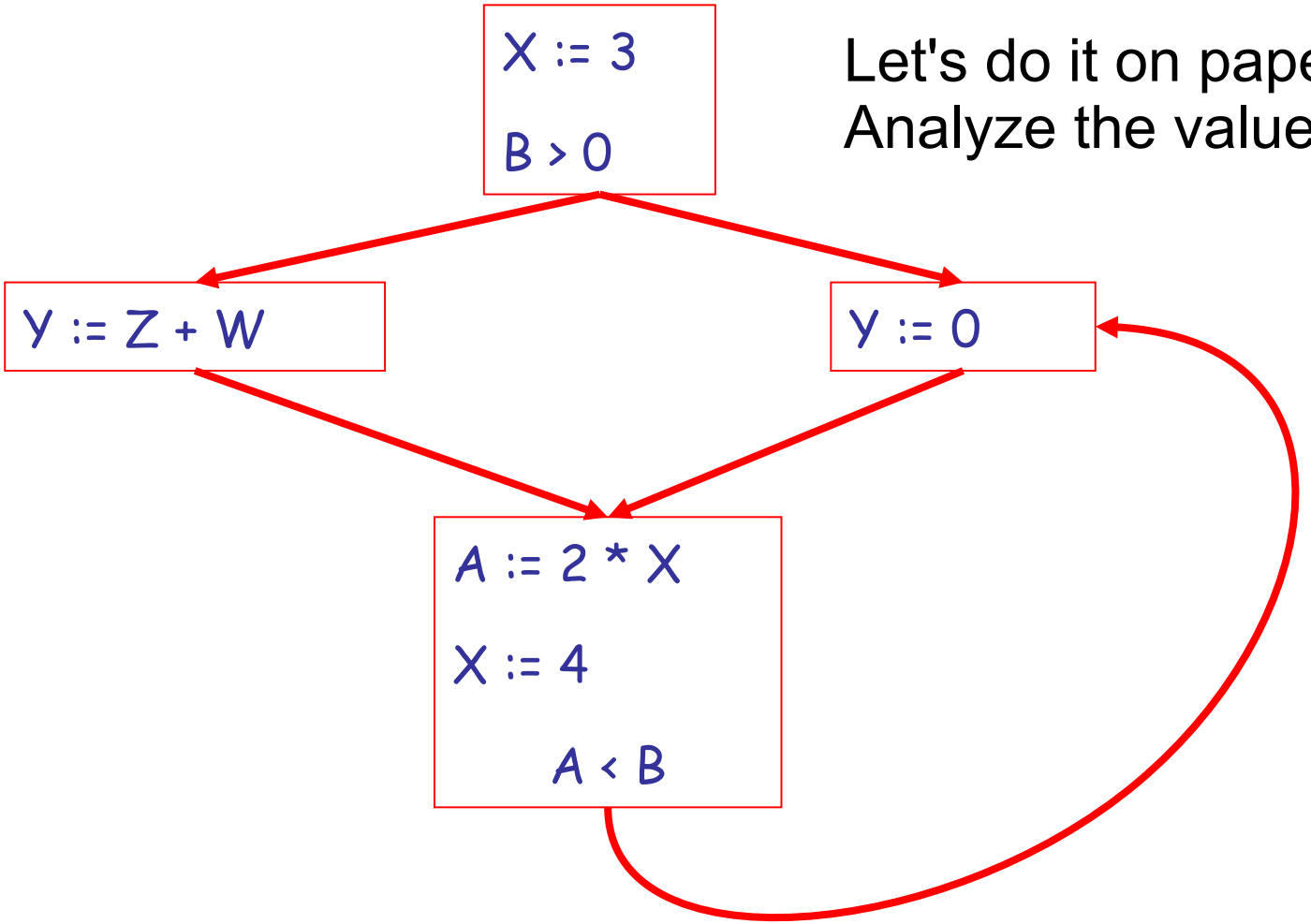
The Value ⊥ (Cont.)

- 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 "we have not yet analyzed control reaching this point"

Sometimes all paths lead to the same place.

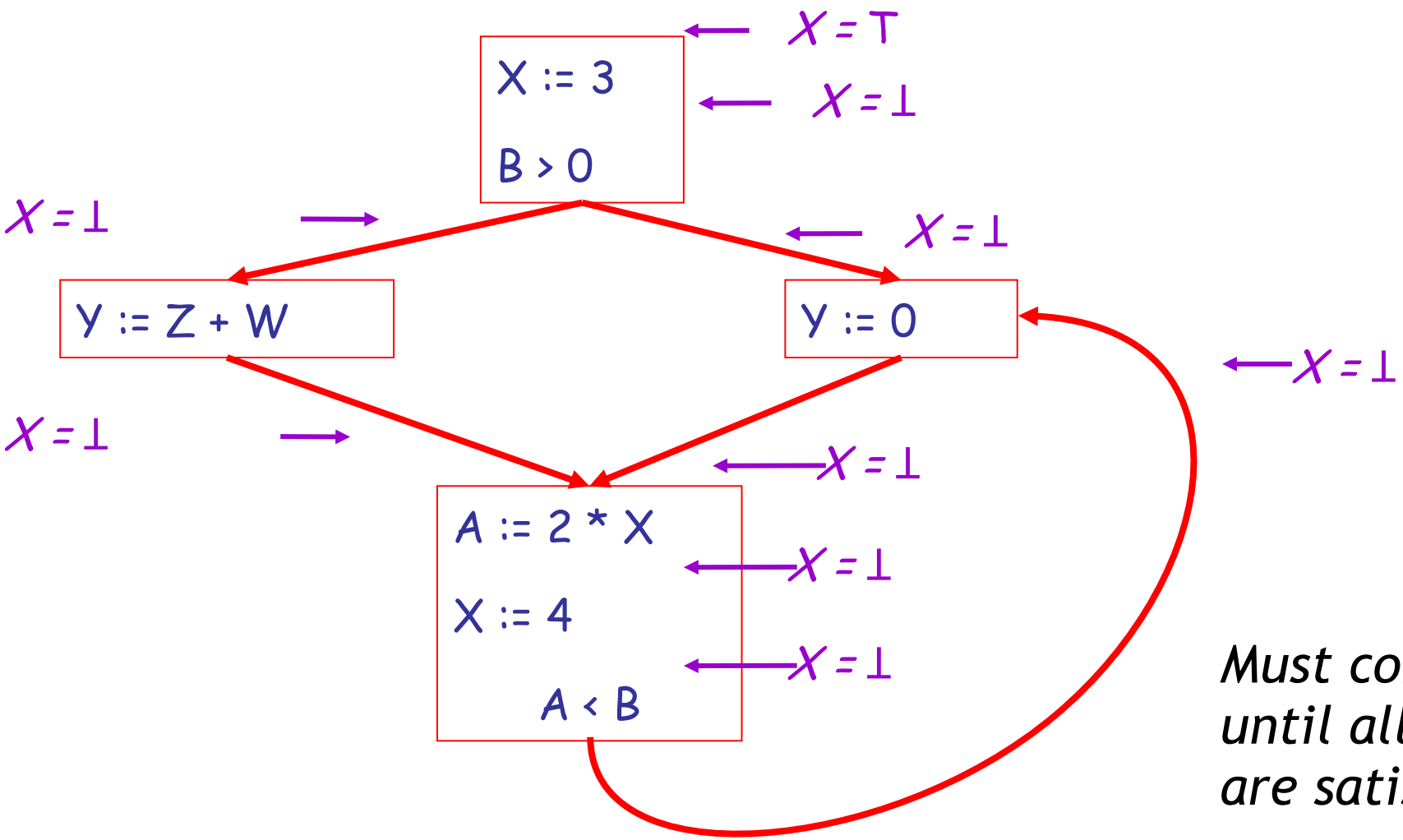
Thus you need ⊥.

Another Example



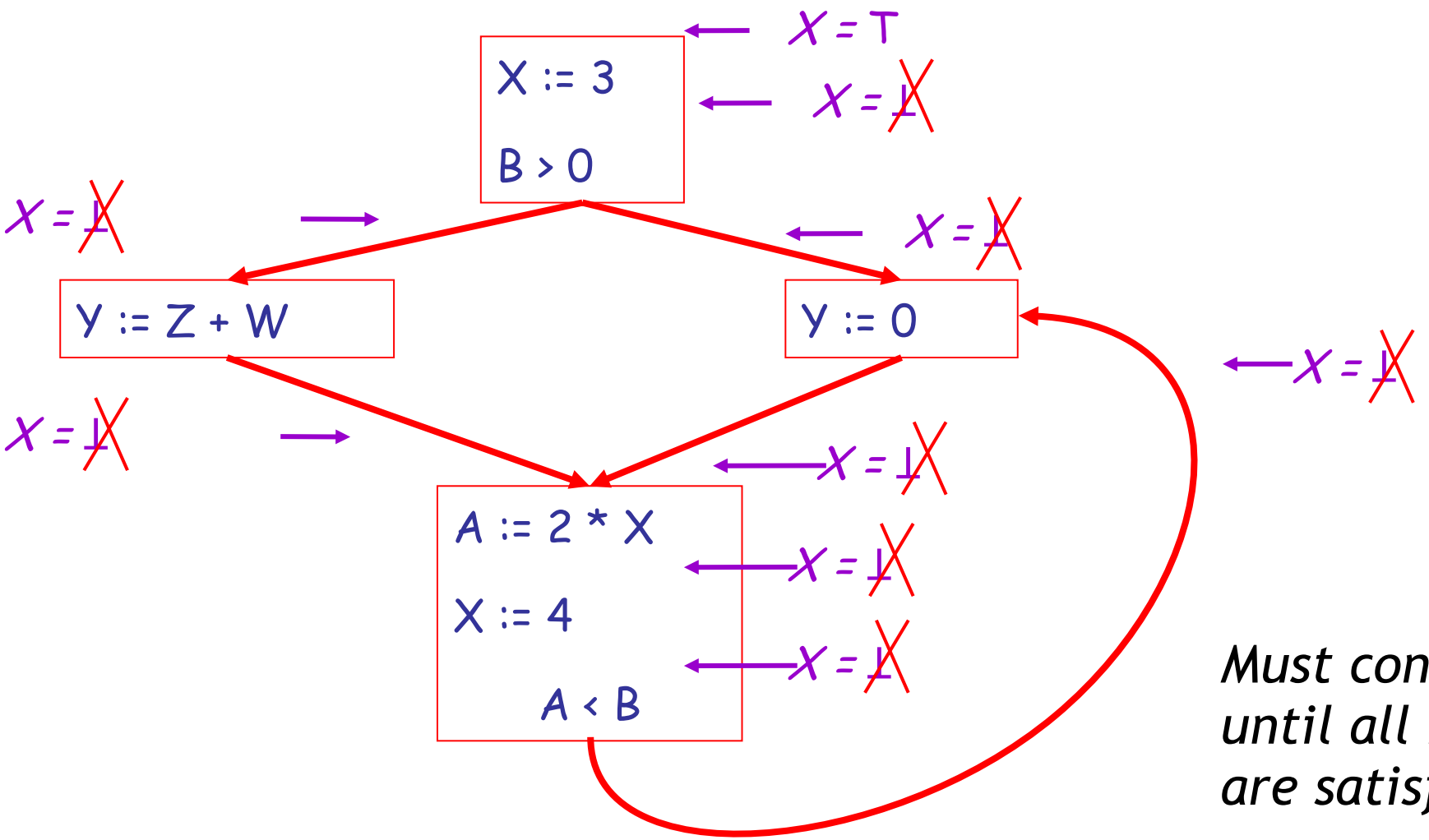
Let's do it on paper!
Analyze the value of X …

Another Example: Intermediate



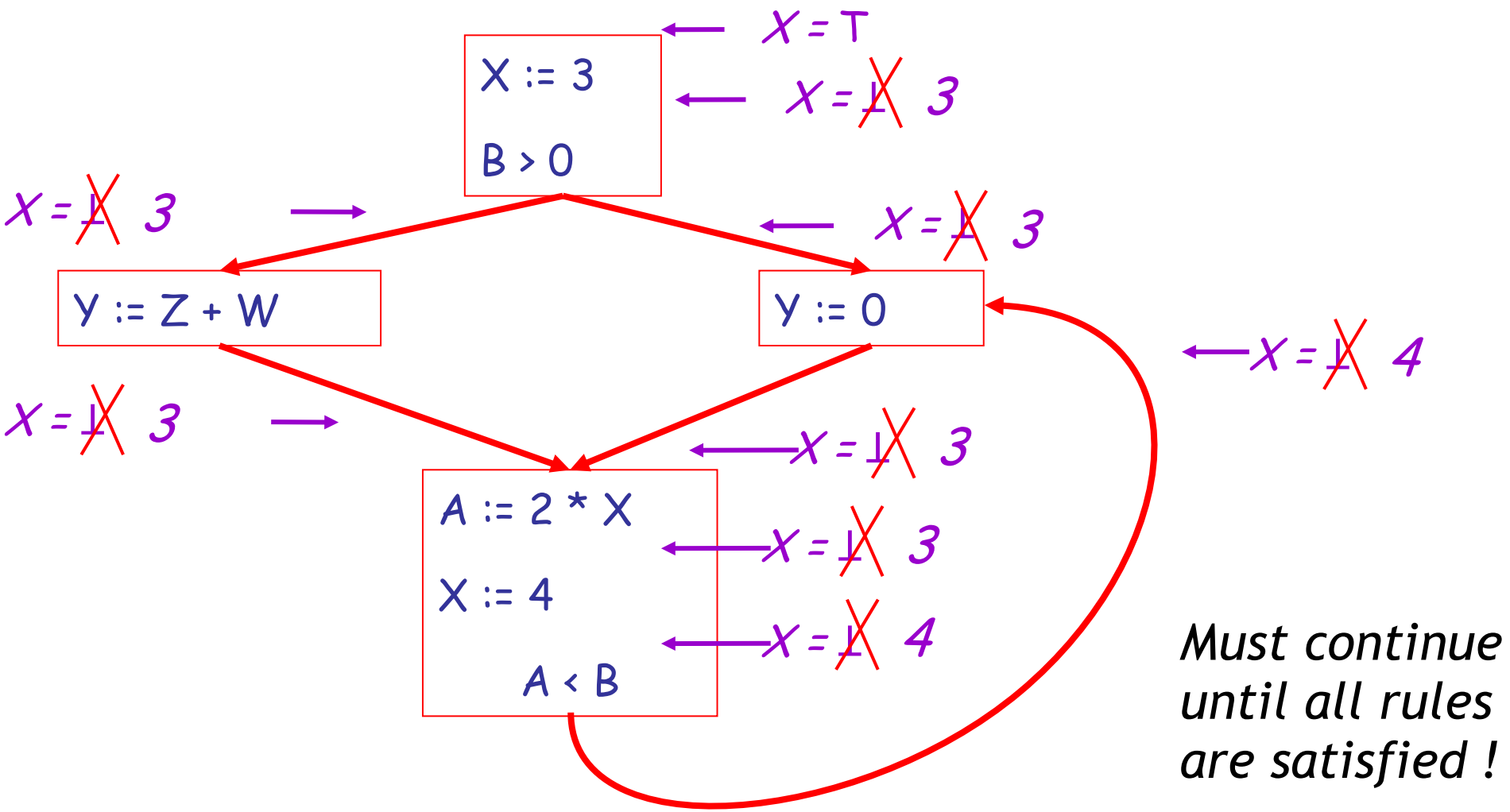
Must continue until all rules are satisfied !

Another Example: Intermediate

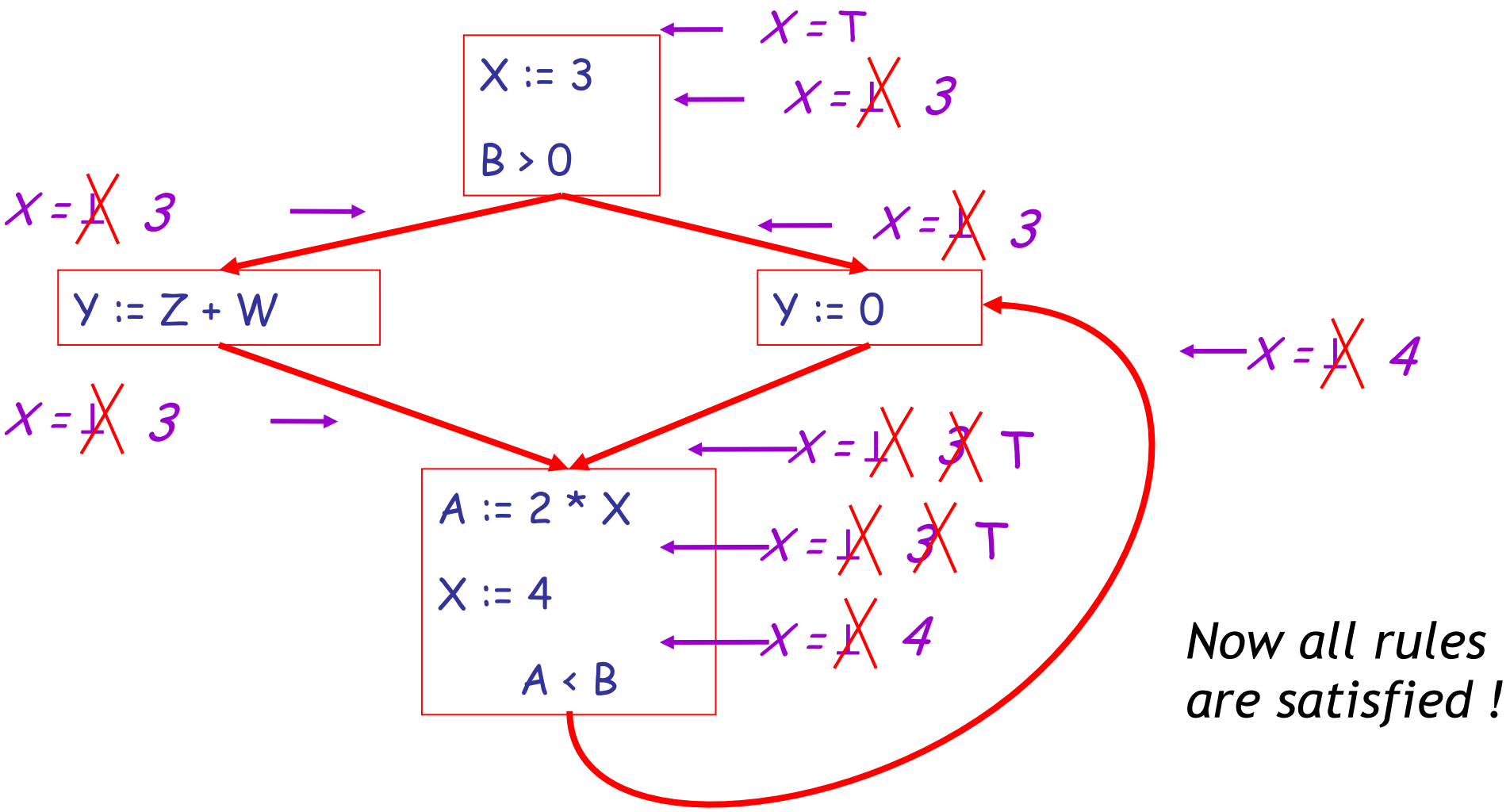


Must continue until all rules are satisfied !

Another Example: Intermediate



Another Example: Answer



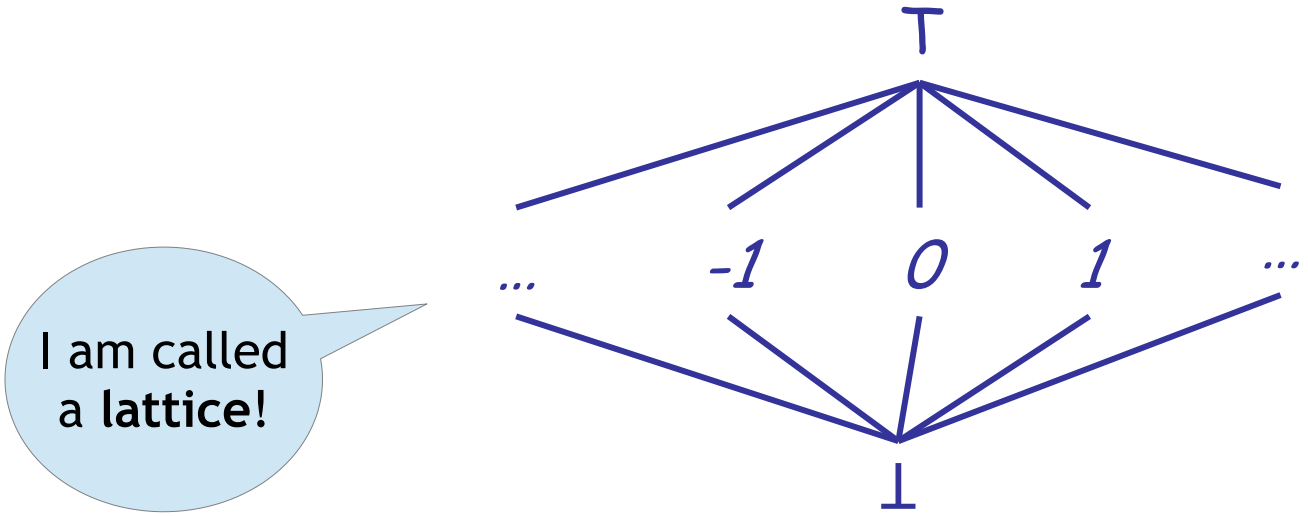
Now all rules are satisfied !

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



Orderings (Cont.)

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

- Let ***lub*** be the least-upper bound in this ordering
  - 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 ⊥ and only *increase*

    ⊥ can change to a constant, and a constant to ⊤
  - Thus, C_(x, s) can change at most twice

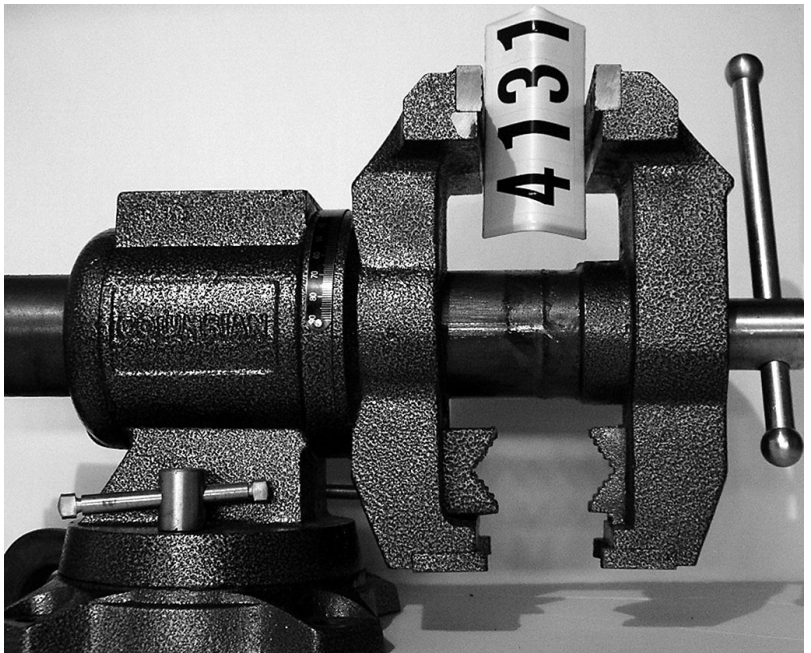
Number Crunching

The algorithm is polynomial in program size:

Number of steps =

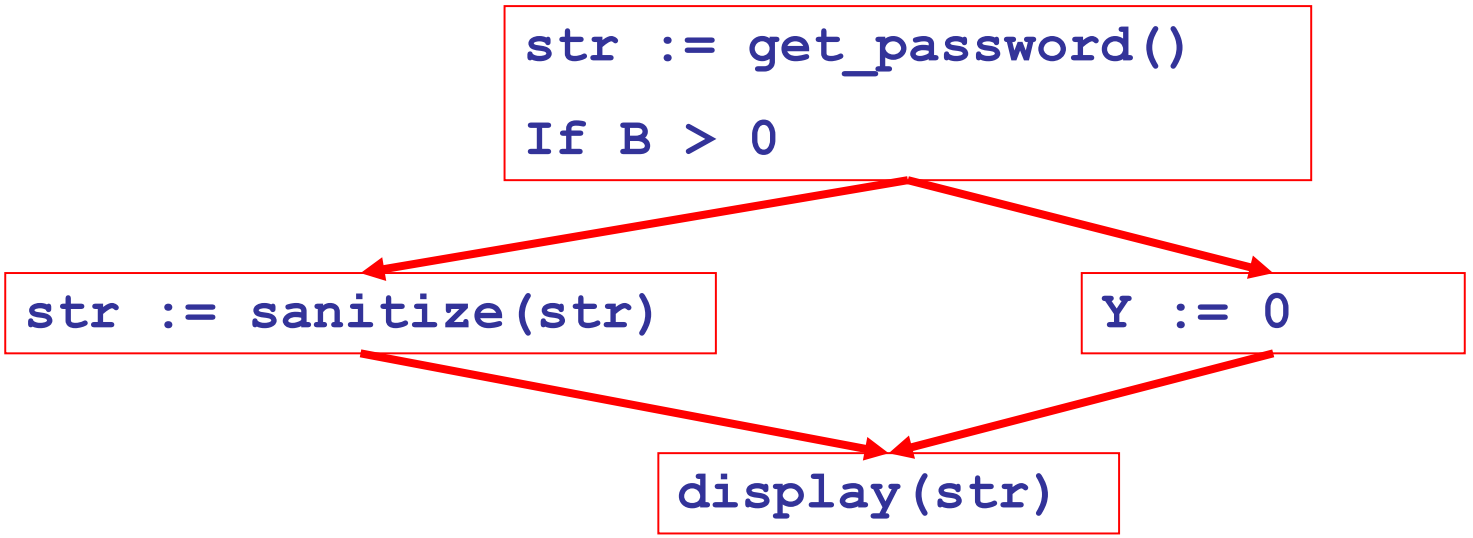
Number of C_(….) values changed * 2 =

(Number of program statements)^2 * 2



“Potential Secure Information Leak” Analysis

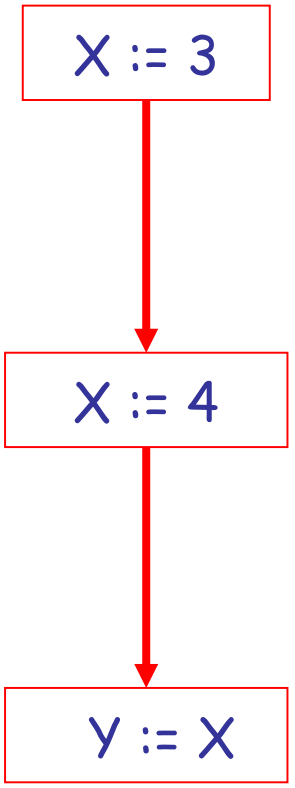
Could sensitive information possibly reach an insecure use?



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

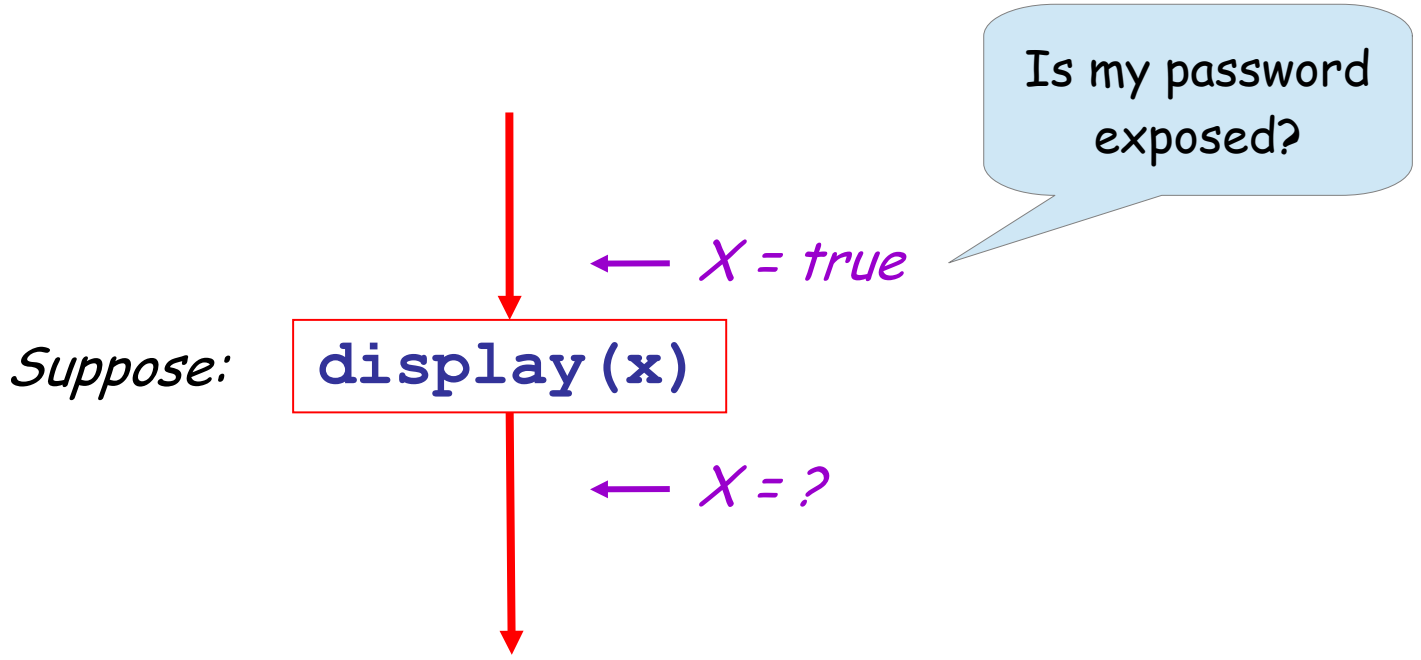
A variable x at stmt s is a possible sensitive (high-security) information leak if

- There exists a 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 the 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

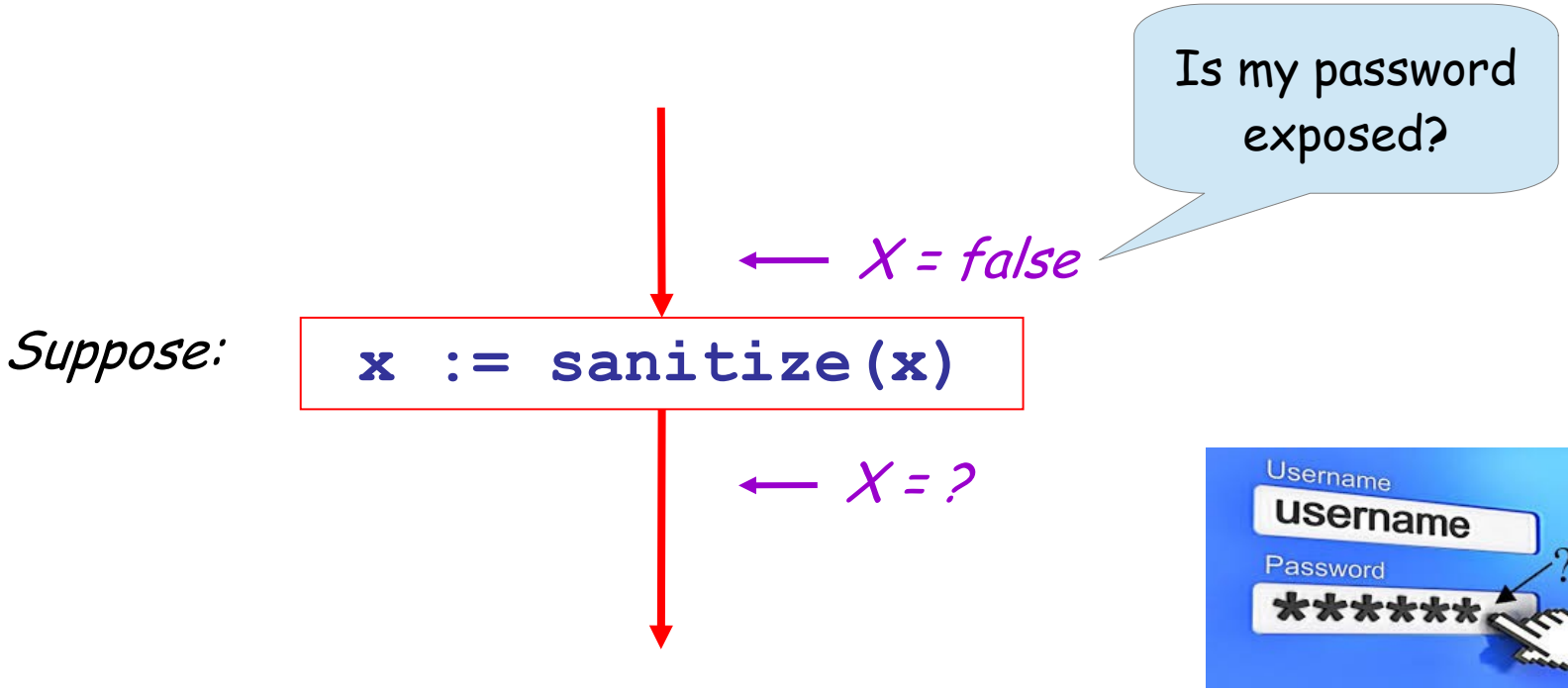
Secure Information Flow Rule 1



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

true means "if this ends up being a secret variable then we have a bug!"

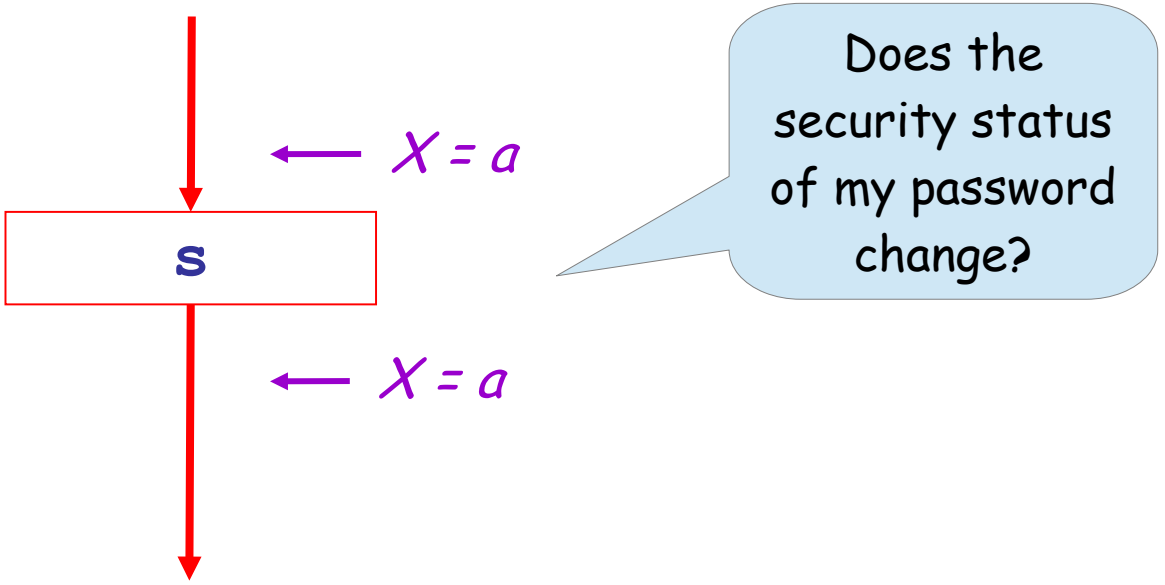
Secure Information Flow Rule 2



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

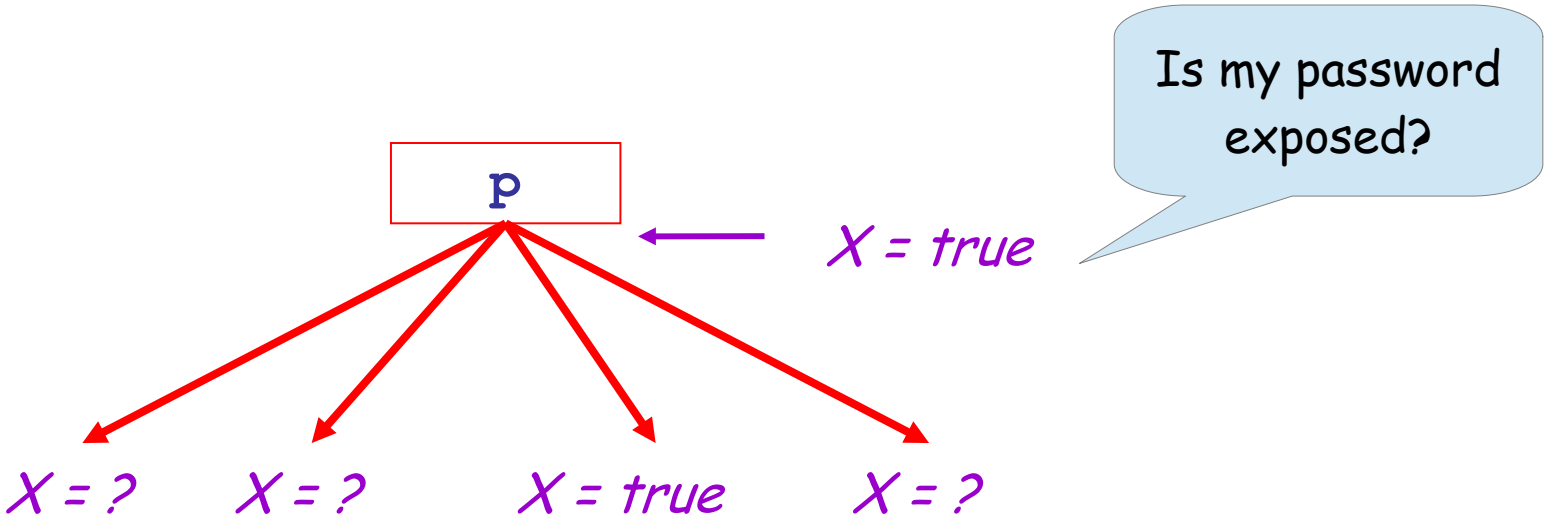
(any subsequent use is safe)

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

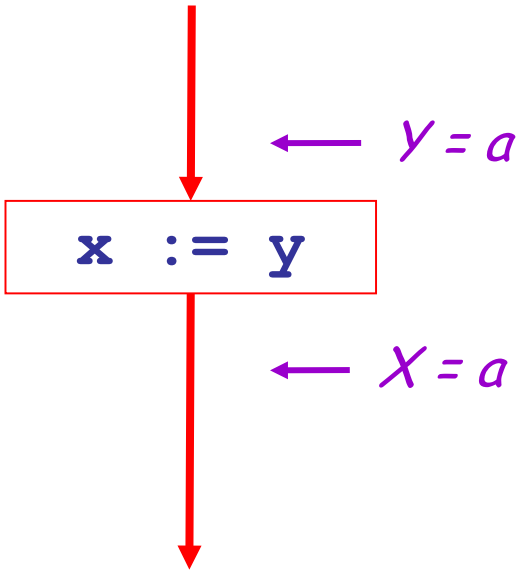


$H_{out}(x, p) = \vee \{ H_{in}(x, s) \mid s$ a successor of $p \}$

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

Secure Information Flow Rule 5

(Bonus – What if we have to reason about 2 variables?)



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

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

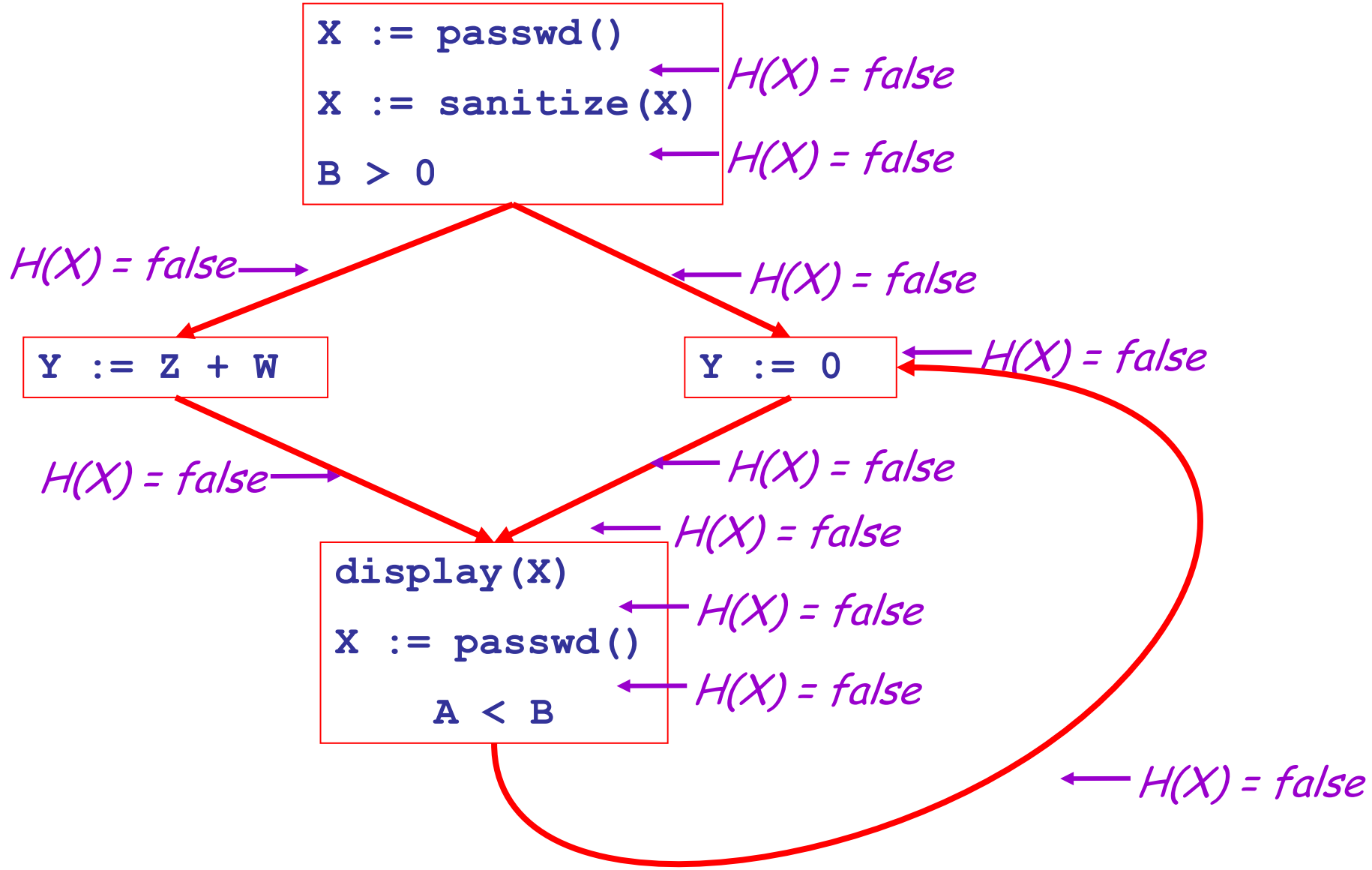
Algorithm: How so we start?

- We are going to let all H_(…) = false initially

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

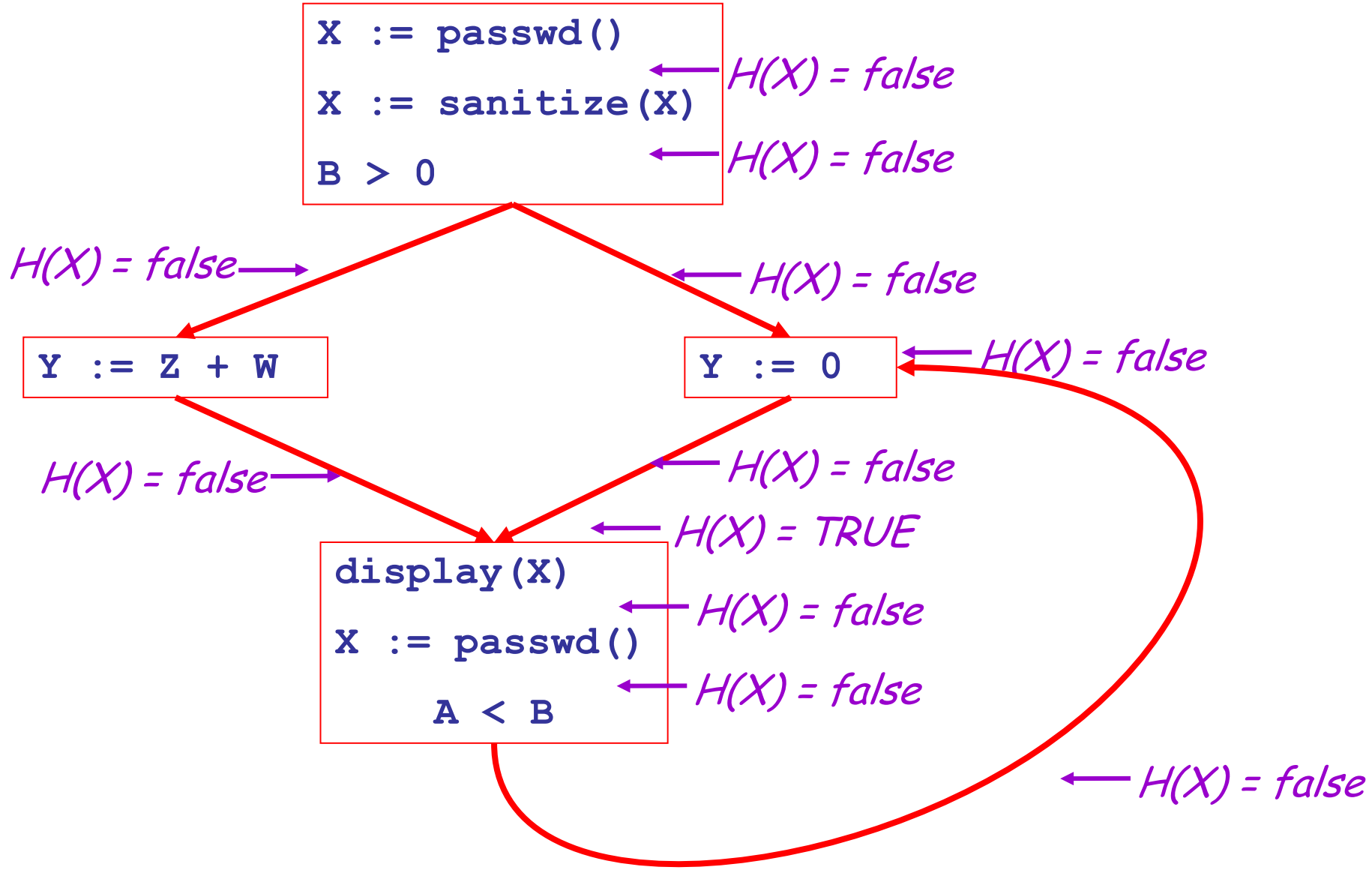
  Pick s where one of 1-4 does not hold and update using the appropriate rule

Secure Information Flow Example

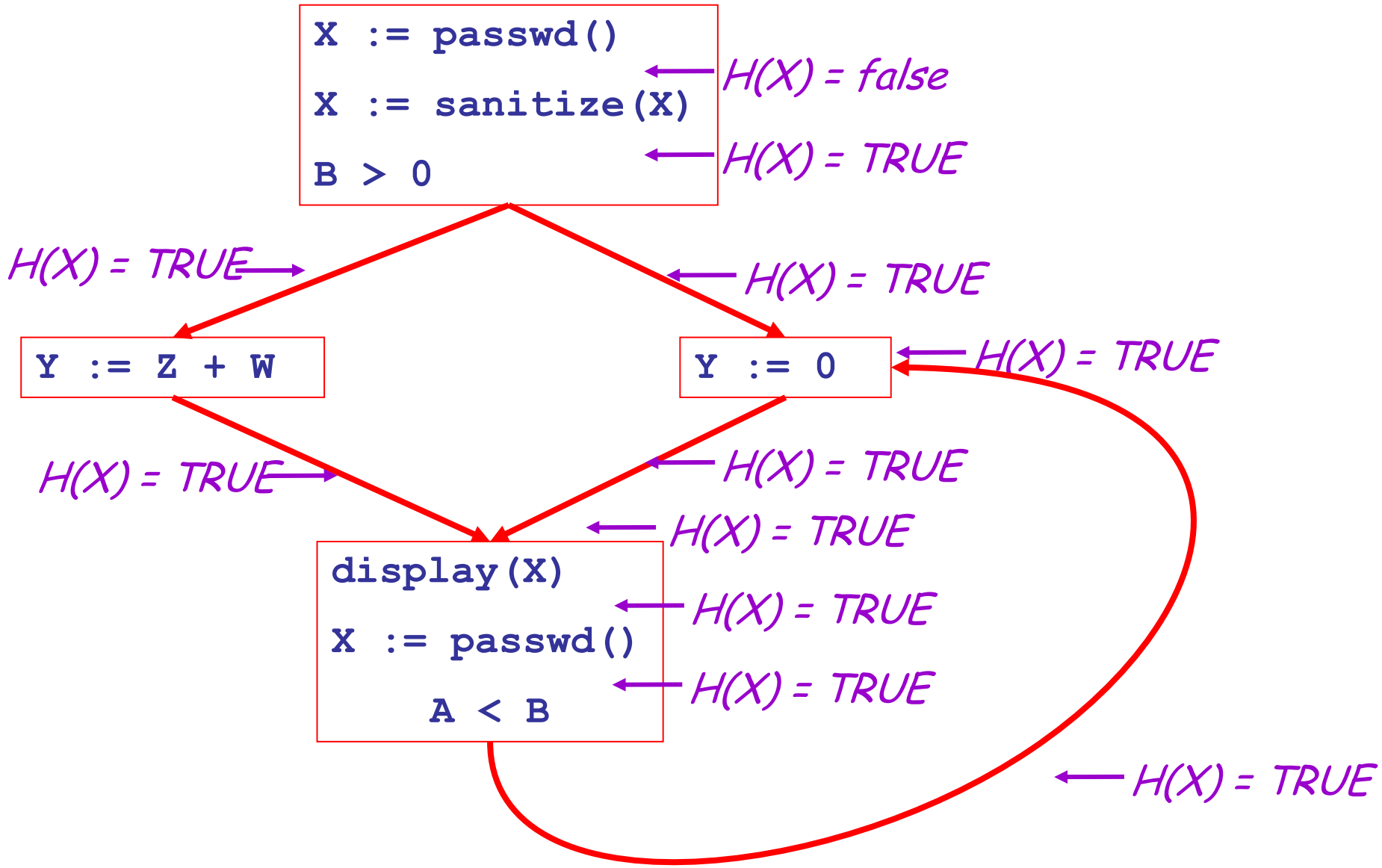


(Time permitting: discuss with your partner. Where does the analysis **start**?)

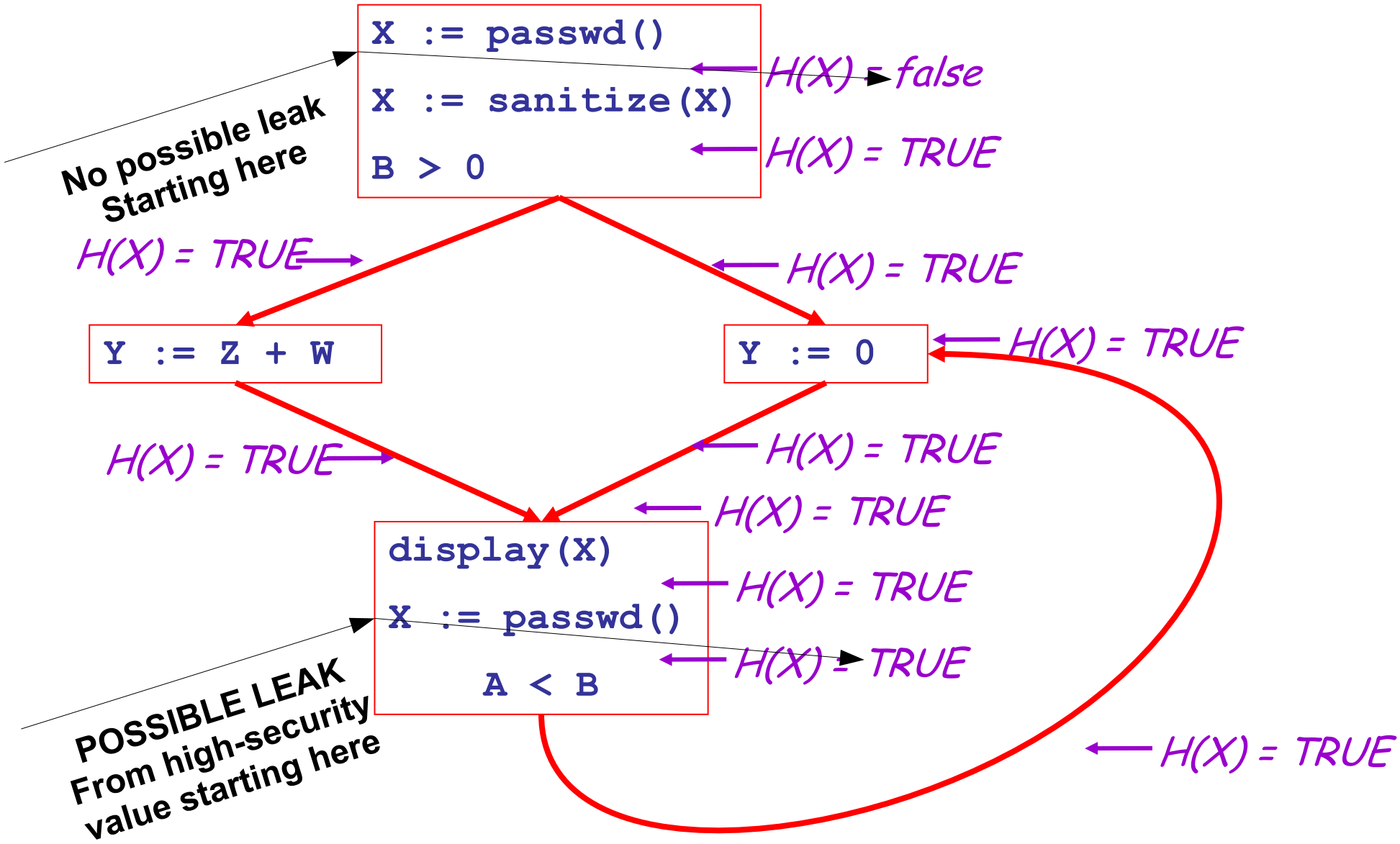
Secure Information Flow Example



Secure Information Flow Example



Secure Information Flow Example

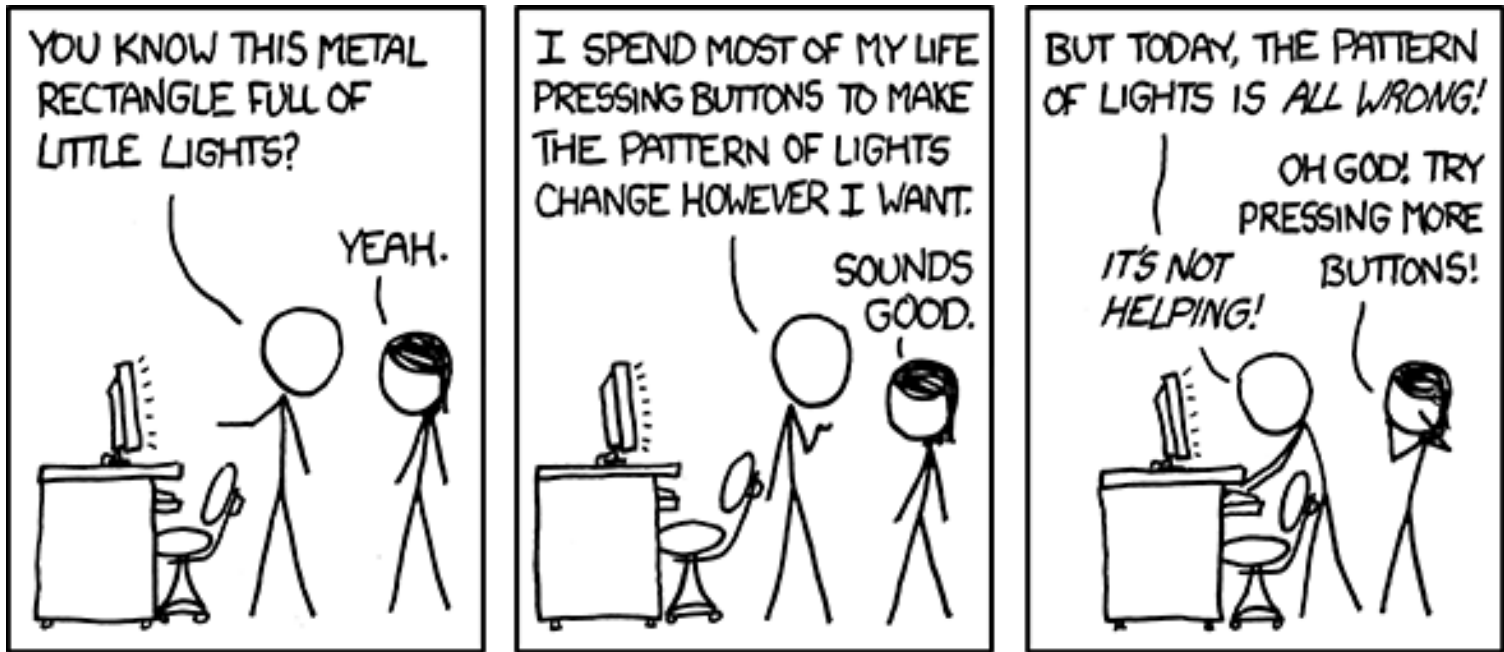


Termination

- 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 entry point for sensitive information

Static Analysis Limitations

- Where might a static analysis go wrong?
- If I asked you to construct the shortest program you can that causes one of our static analyses to get the “wrong” answer, what would you do?



Static Analysis

- Potential exam practice and thought question
- You are asked to design a static analysis to detect bugs related to **file handles**
  - 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?

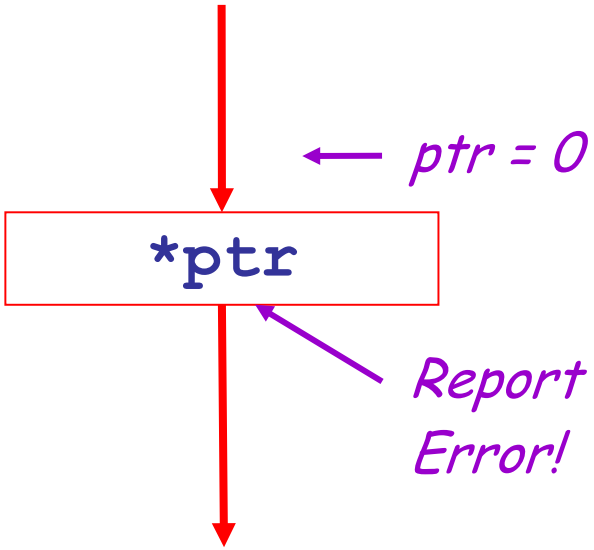
Abstract Information

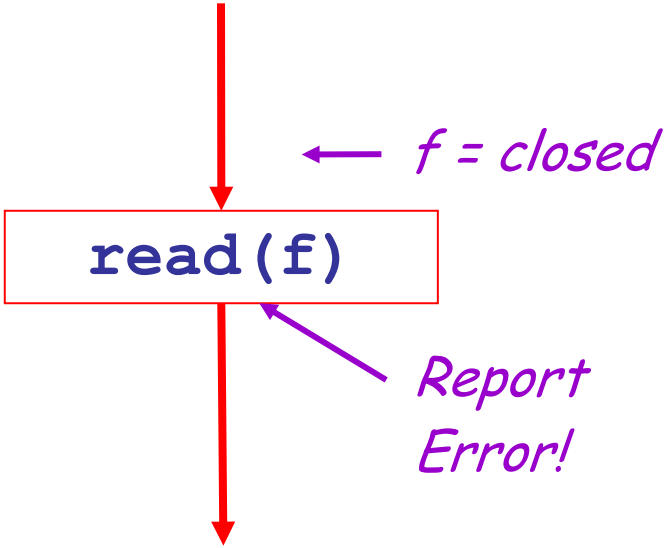
- We will keep track of an abstract value for a given file handle variable
- Values and Interpretations

$\top$	file handle state is unknown
$\bot$	haven't reached here yet
closed	file handle is closed
open	file handle is open

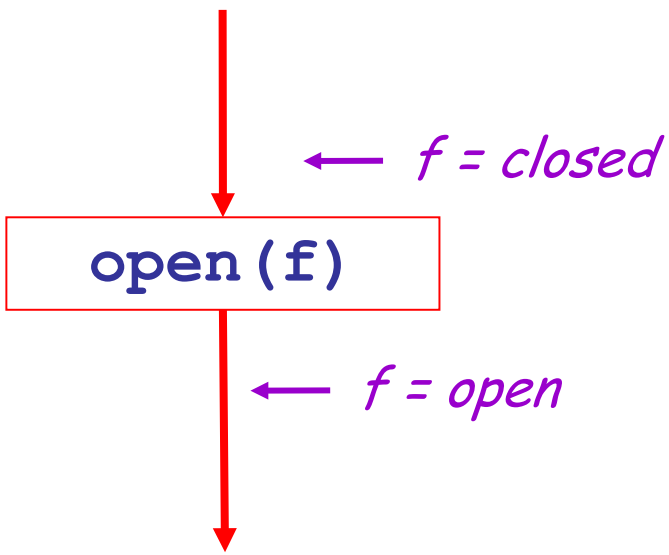
Rules

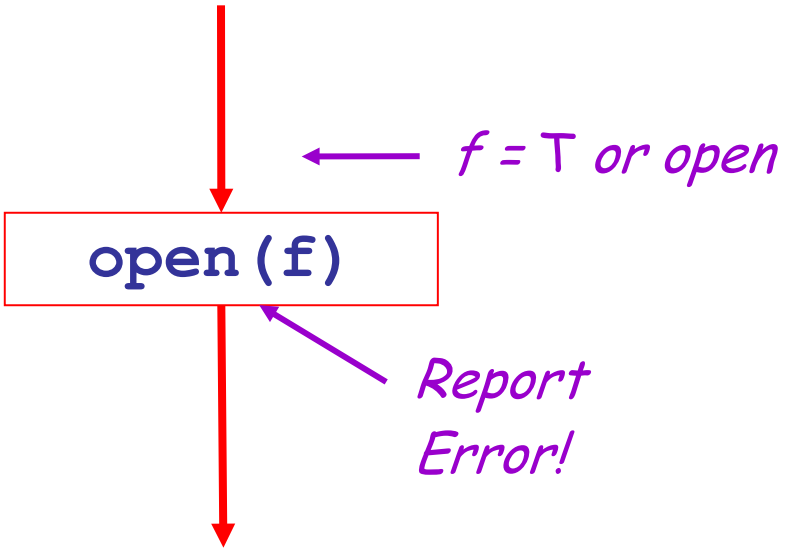
- Previously: “null ptr”
- Now: “file handles”



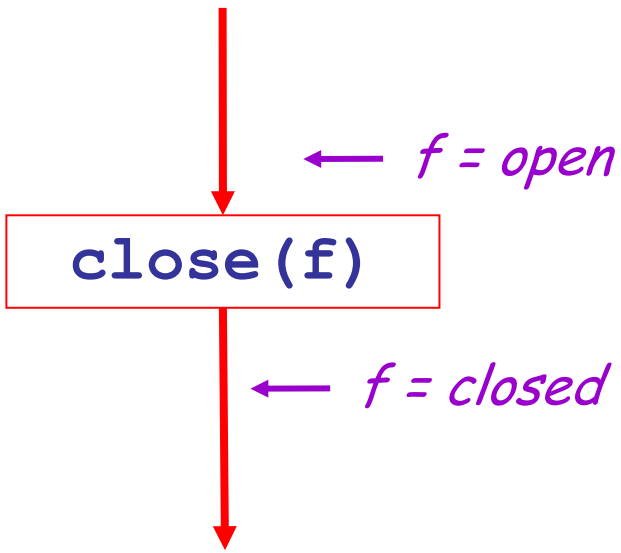


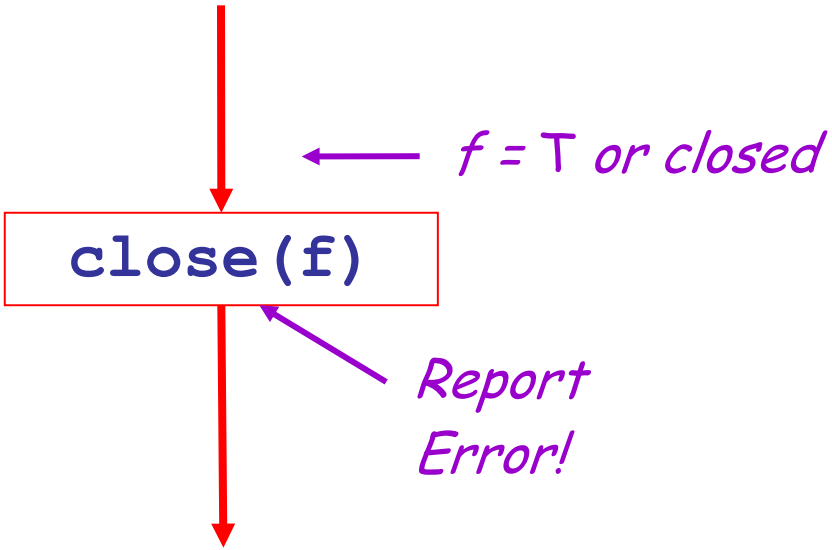
Rules: open





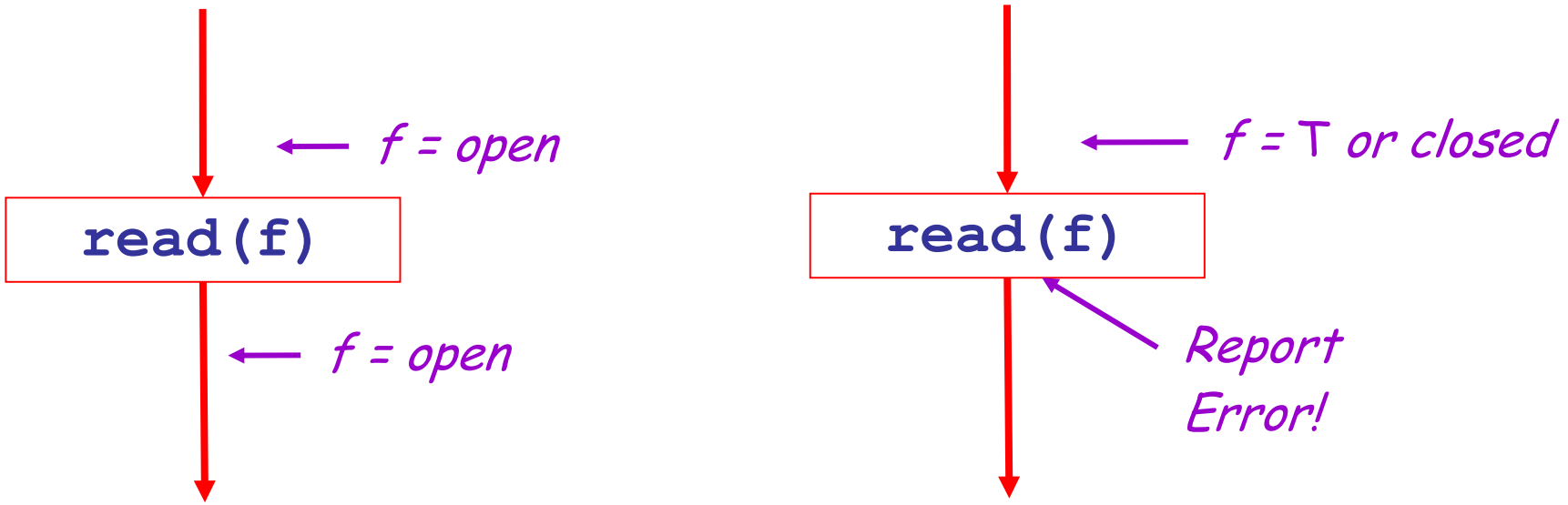
Rules: close



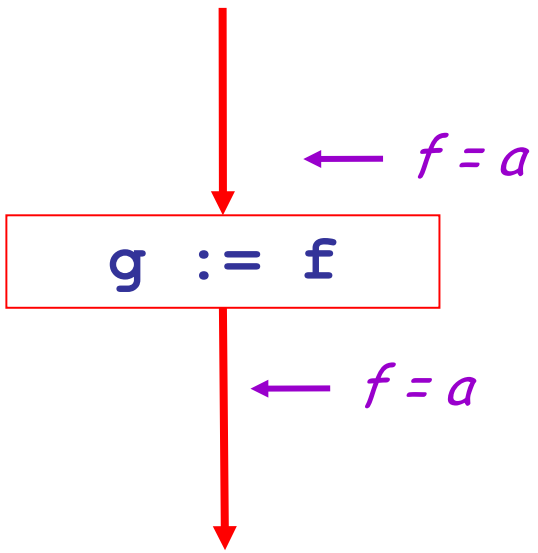


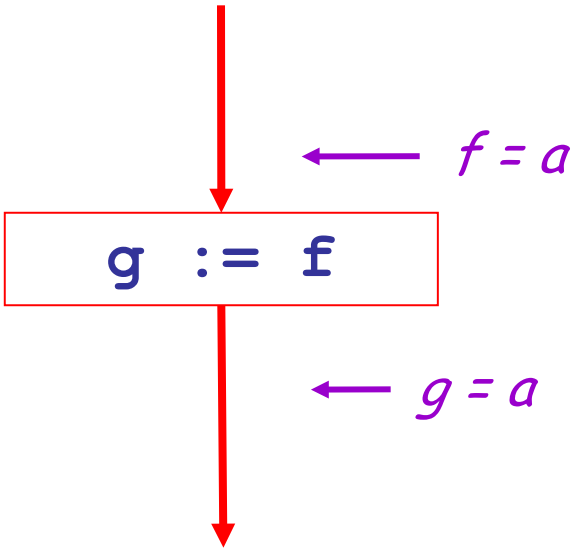
Rules: read/write

(write is identical)

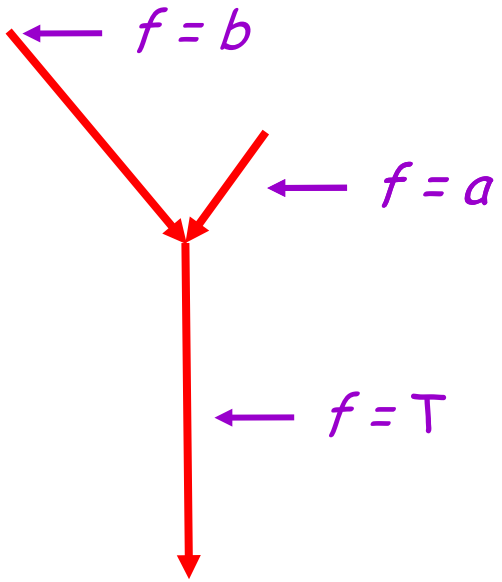


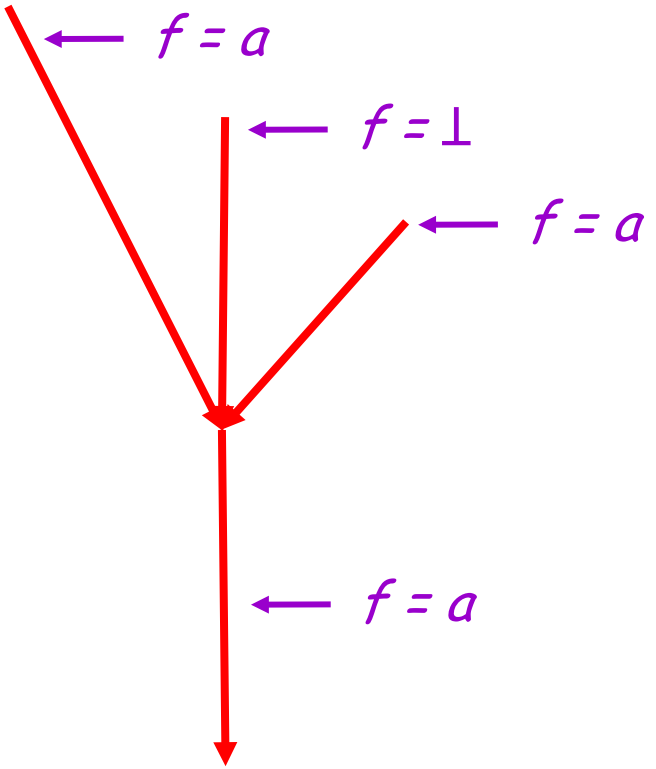
Rules: Assignment





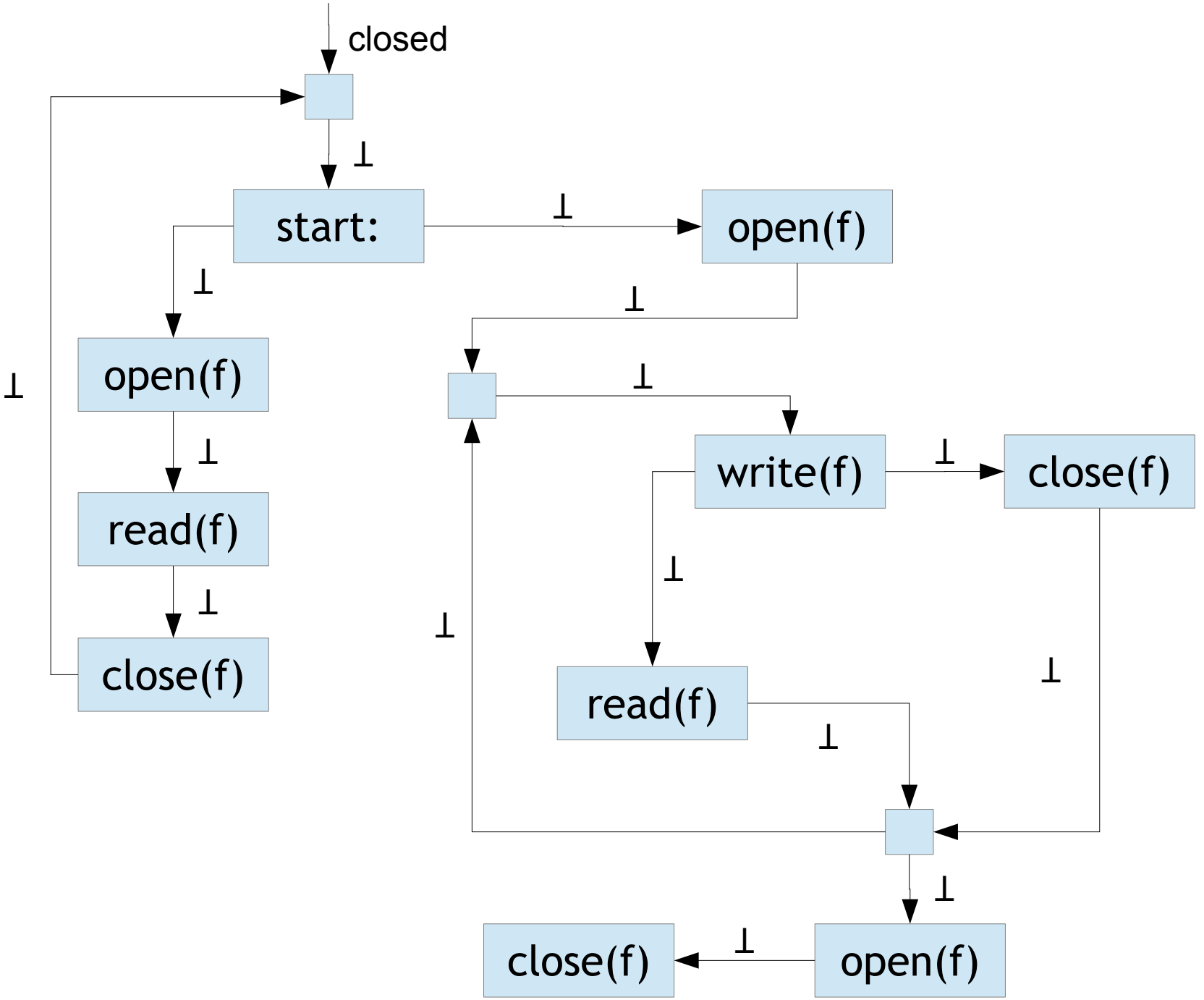
Rules: Multiple Possibilities

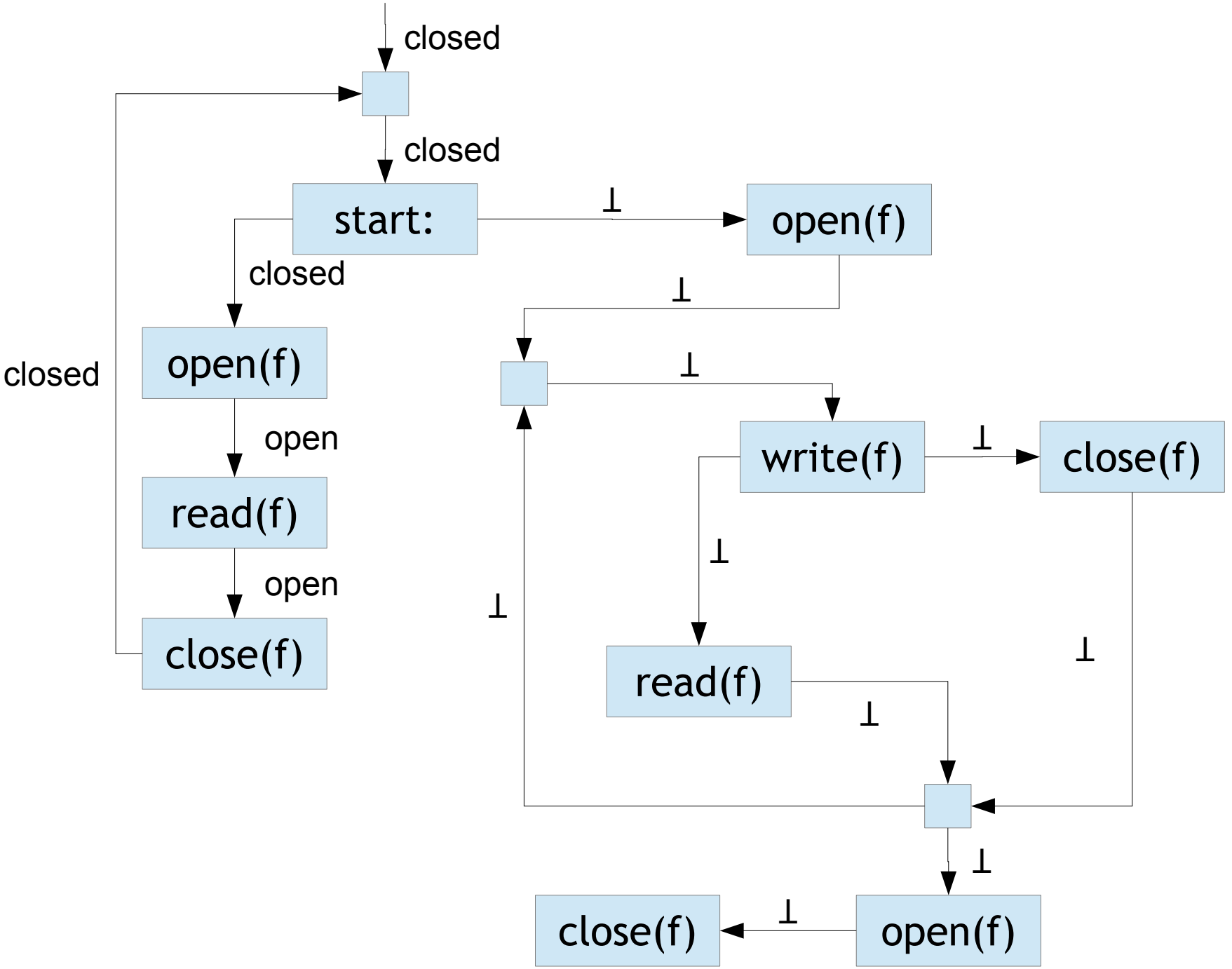


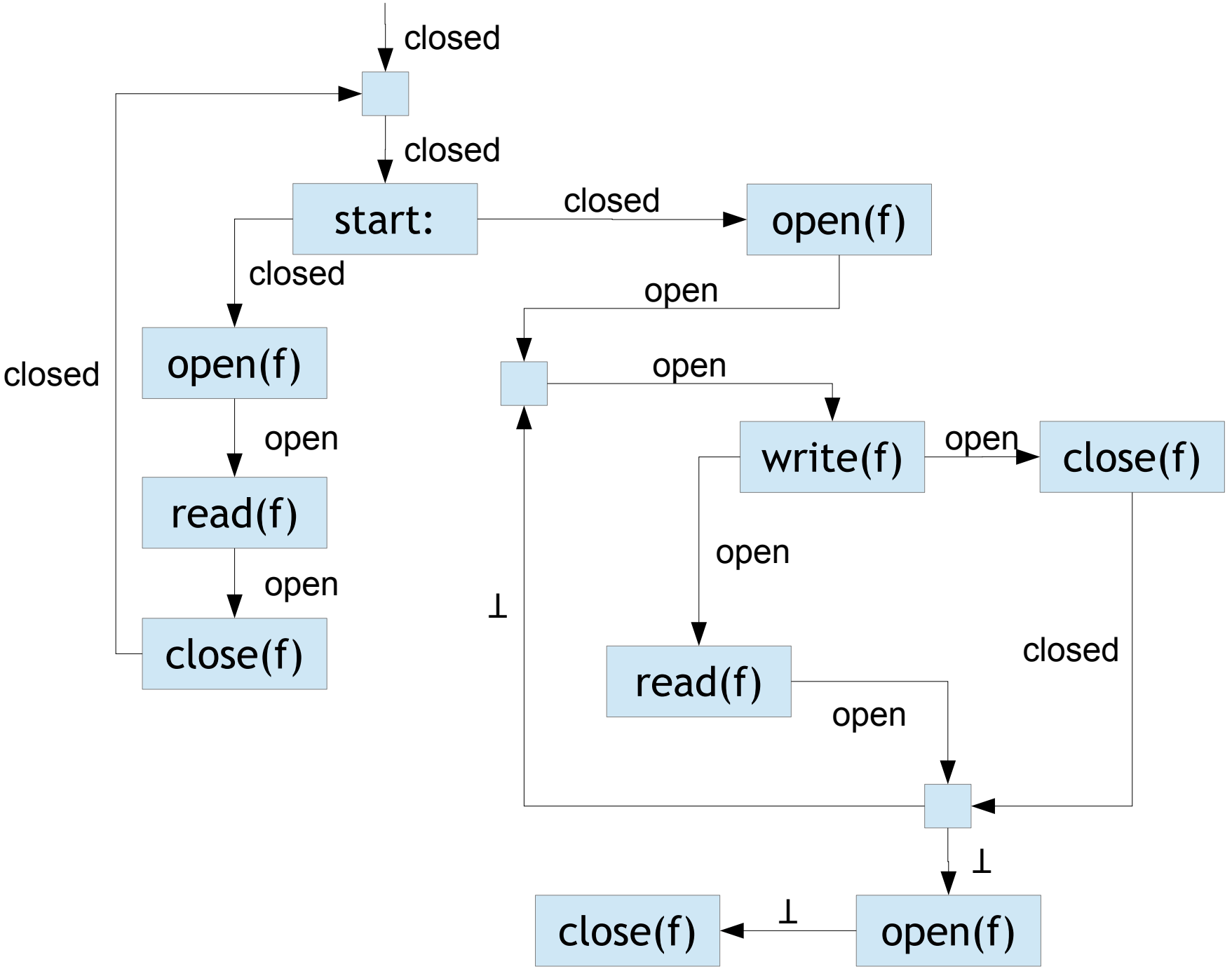


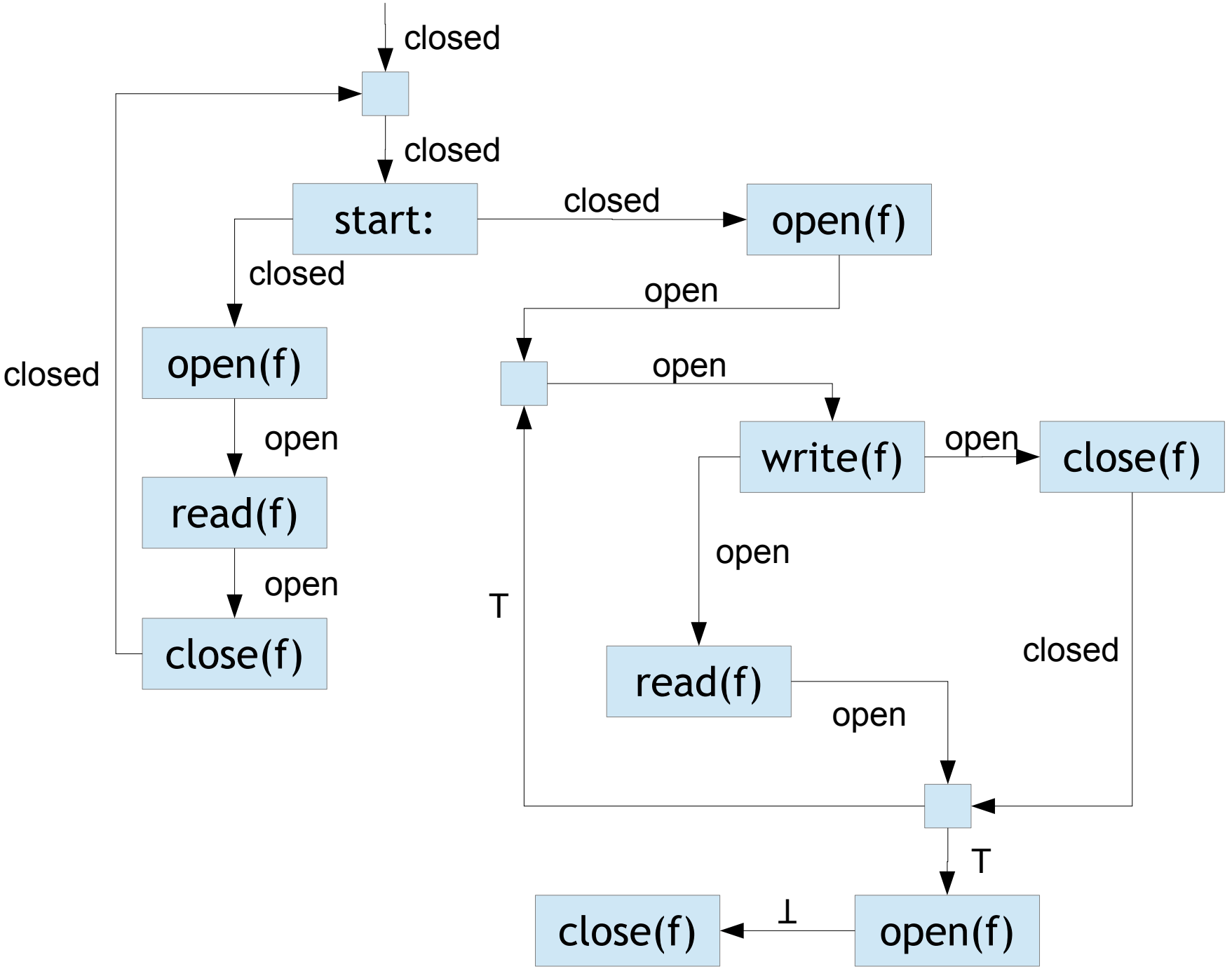
A Tricky Program

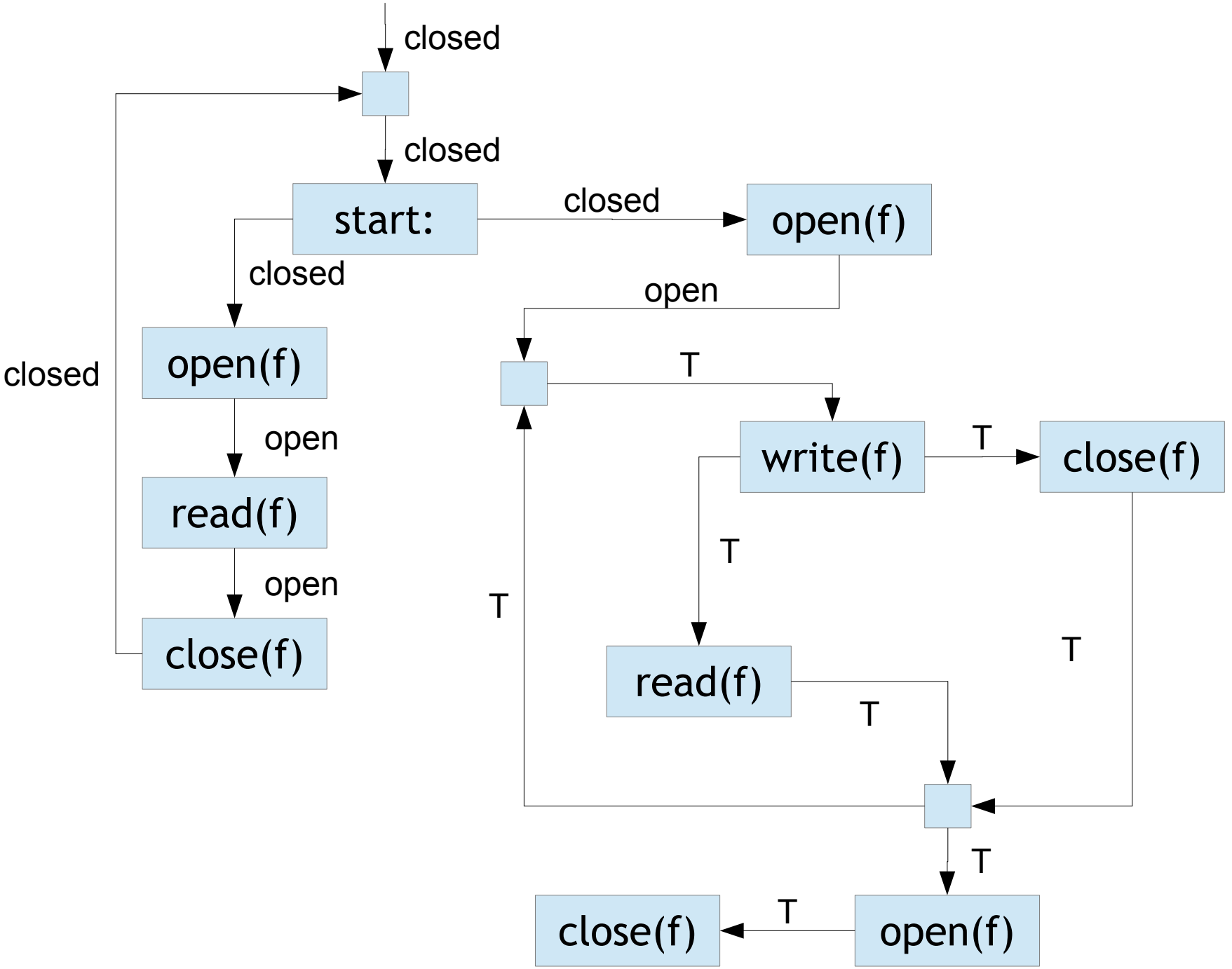
```
start:
switch (a)
  case 1: open(f); read(f); close(f); goto start
  default: open(f);
do {
  write(f) ;
  if (b):   read(f);
  else:     close(f);
} while (b)
open(f);
close(f);
```

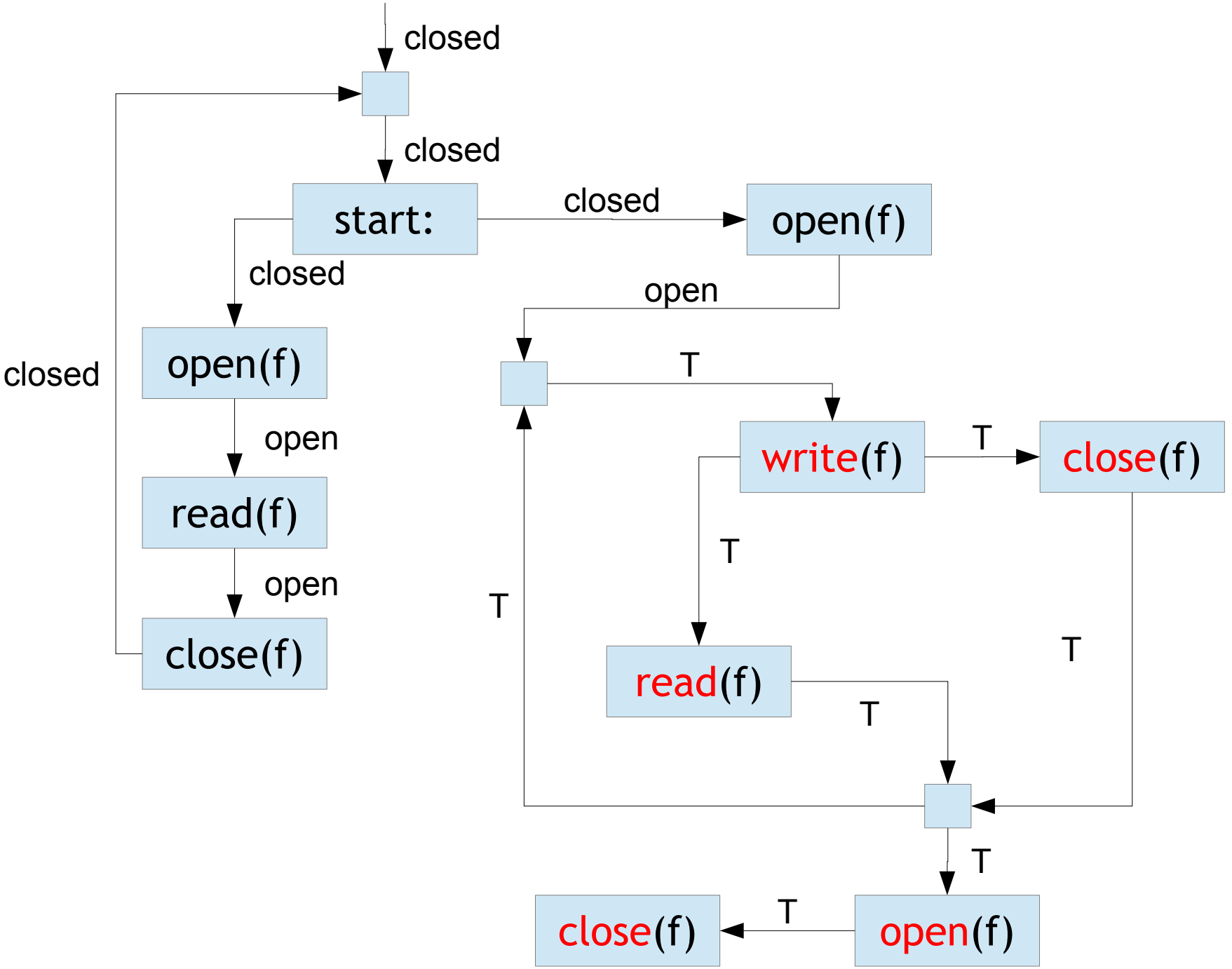

closed
start:
open(f)
open(f)
read(f)
close(f)
write(f)
close(f)
read(f)
close(f)
open(f)
⊥


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


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


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


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


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

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

Questions?

- Exam 1 shortly!
- Priscila's Exam Review Thursday (now?) in LCSIB1355 until 4pm

- How's the homework going?
  - Don't neglect the homework while studying for the exam.