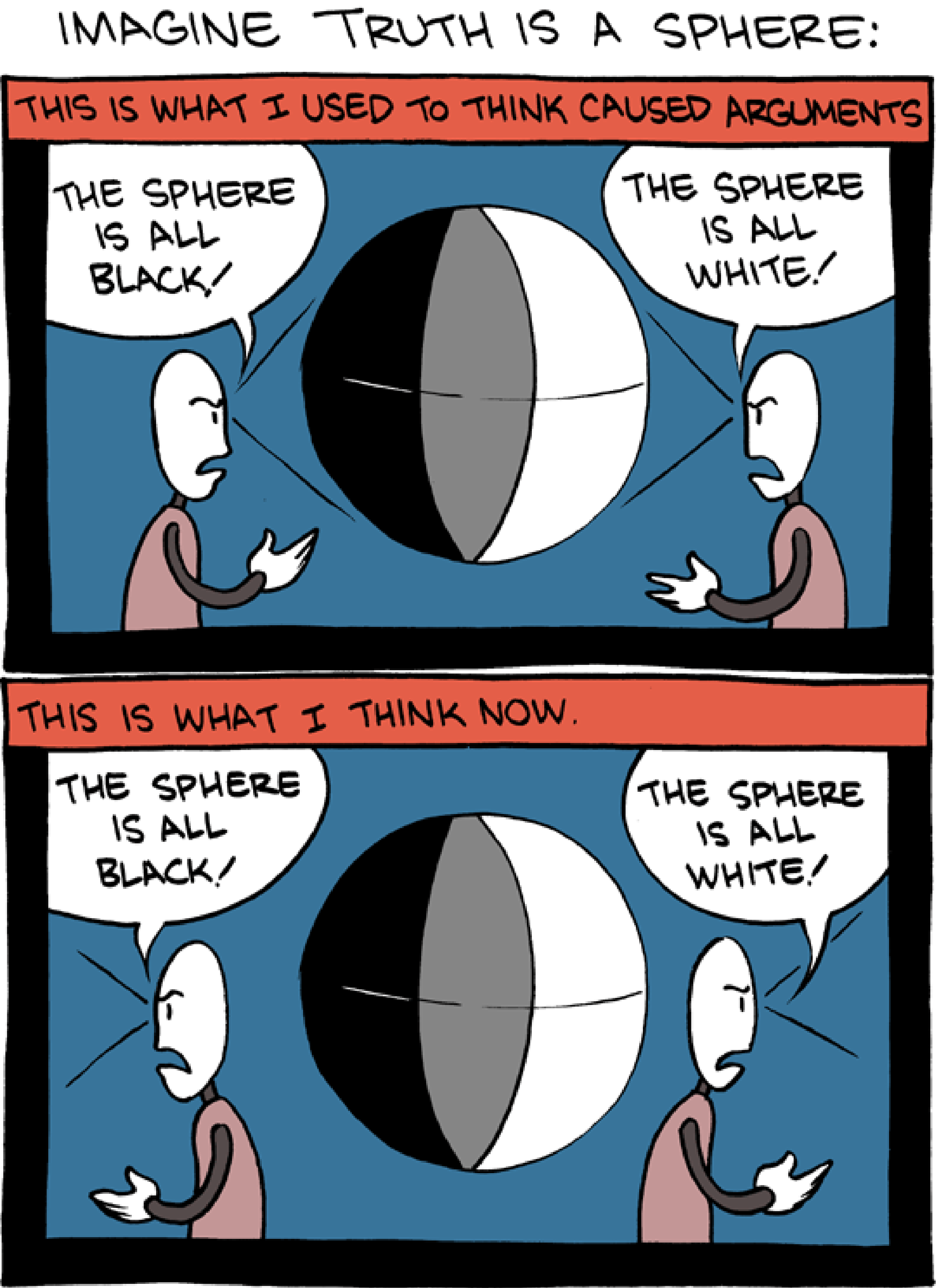

IMAGINE TRUTH IS A SPHERE:
THIS IS WHAT I USED TO THINK CAUSED ARGUMENTS
THE SPHERE IS ALL BLACK!
THE SPHERE IS ALL WHITE!
THIS IS WHAT I THINK NOW.
THE SPHERE IS ALL BLACK!
THE SPHERE IS ALL WHITE!

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?

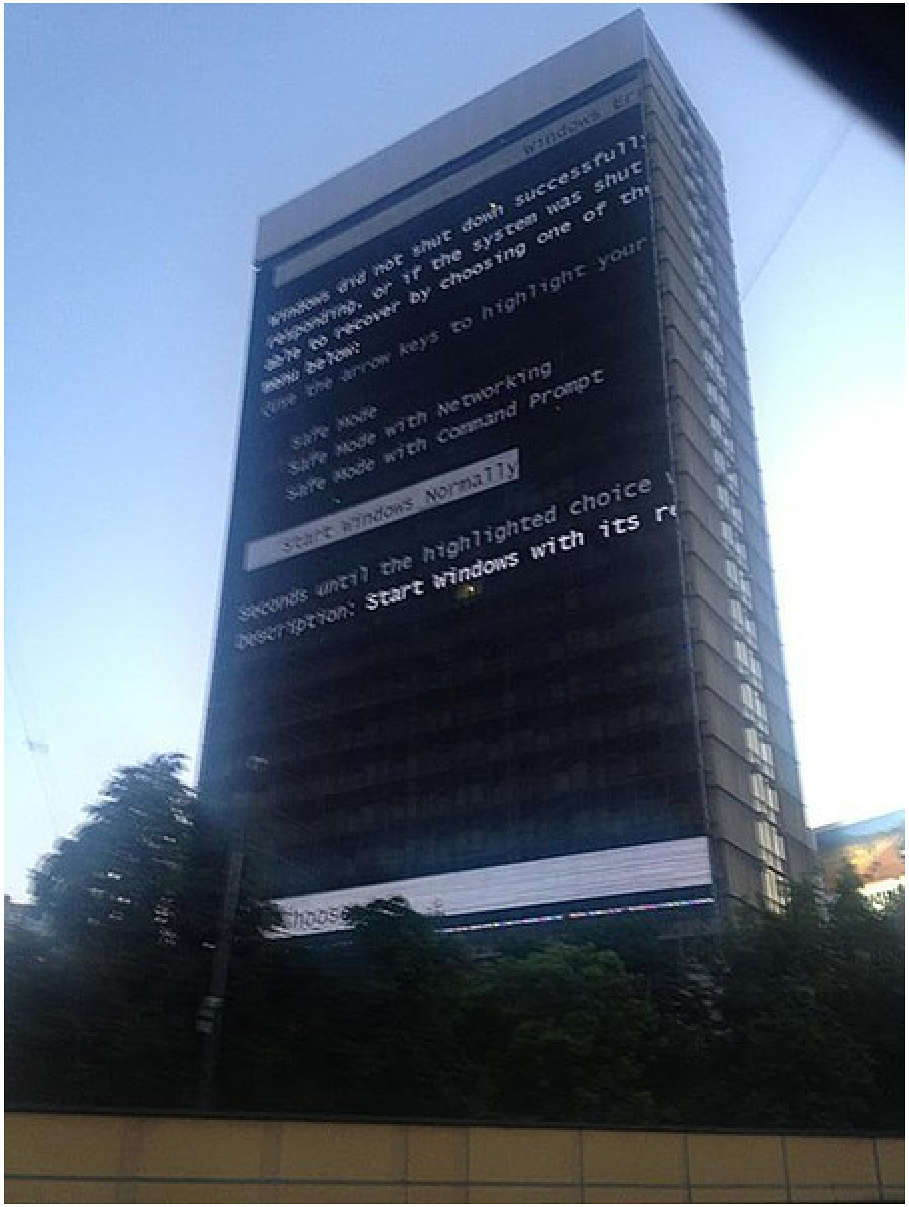
One-Slide Summary

- **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.

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;


Windows Er
Windows did not shut down successfull
responding, or if the system was shut
able to recover by choosing one of th
menu below:
(Use the arrow keys to highlight your
Safe Mode
Safe Mode with Networking
Safe Mode with Command Prompt
Start Windows Normally
Seconds until the highlighted choice
Description: Start Windows with its r
Choose

“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
  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;
}
```

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

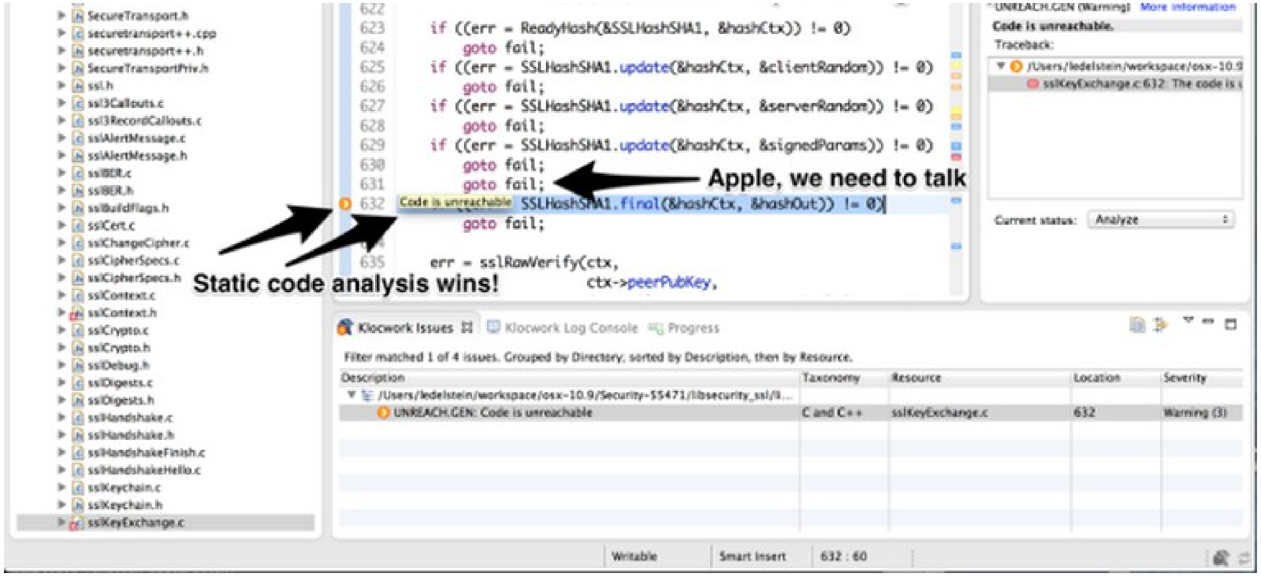
Klocwork: Our source code analyzer caught Apple's 'gotofail' bug

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

57 223 23 +1 5 More + Comments 25



```
SecureTransport.h
securetransport++.cpp
securetransport++.h
SecureTransportPriv.h
ssl.h
ssl3Callouts.c
ssl3RecordCallouts.c
sslAlertMessage.c
sslAlertMessage.h
sslBER.c
sslBER.h
sslBuildFlags.h
sslCert.c
sslChangeCipher.c
sslCipherSpecs.c
sslCipherSpecs.h
sslContext.c
sslContext.h
sslCrypto.c
sslCrypto.h
sslDebug.h
sslDigests.c
sslDigests.h
sslHandshake.c
sslHandshake.h
sslHandshakeFinish.c
sslHandshakeHello.c
sslKeychain.c
sslKeychain.h
sslKeyExchange.c
```

```
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
Code is unreachable SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
```

Apple, we need to talk

Static code analysis wins!

UNREACH.GEN (Warning) More information

Code is unreachable.

Traceback:

/Users/ledelstein/workspace/osx-10.9

sslKeyExchange.c:632: The code is

Current status: Analyze

Filter matched 1 of 4 issues. Grouped by Directory; sorted by Description, then by Resource.

Description	Taxonomy	Resource	Location	Severity
/Users/ledelstein/workspace/osx-10.9/Security-55471/libsecurity_ssl/li...				
UNREACH.GEN: Code is unreachable	C and C++	sslKeyExchange.c	632	Warning (3)

Writable | Smart Insert | 632 : 60

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.

Featured Posts

Google unveils Androi wearables
Internet & Media

Motorol powered
Internet

OK, Gla in my fa
Cutting E

Apple iF product
Apple

iPad wit comeba
Apple

Most Popular

Giant 3D house
6k Facel

Exclusiv Doesch
716 Twe

Google' four can
771 Goo

Connect With CNET

Facebook
Like Us

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”

Syntactic Analysis Example

- 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.”);
  }
}
```

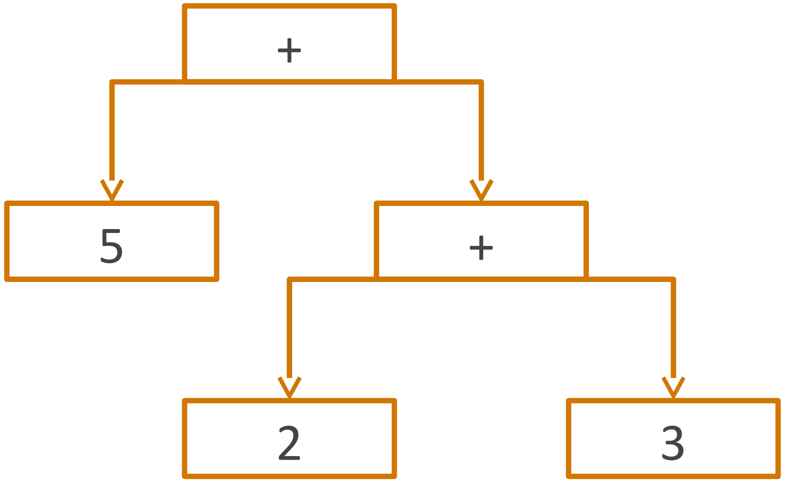
- What could go wrong? First attempt:

```
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.

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

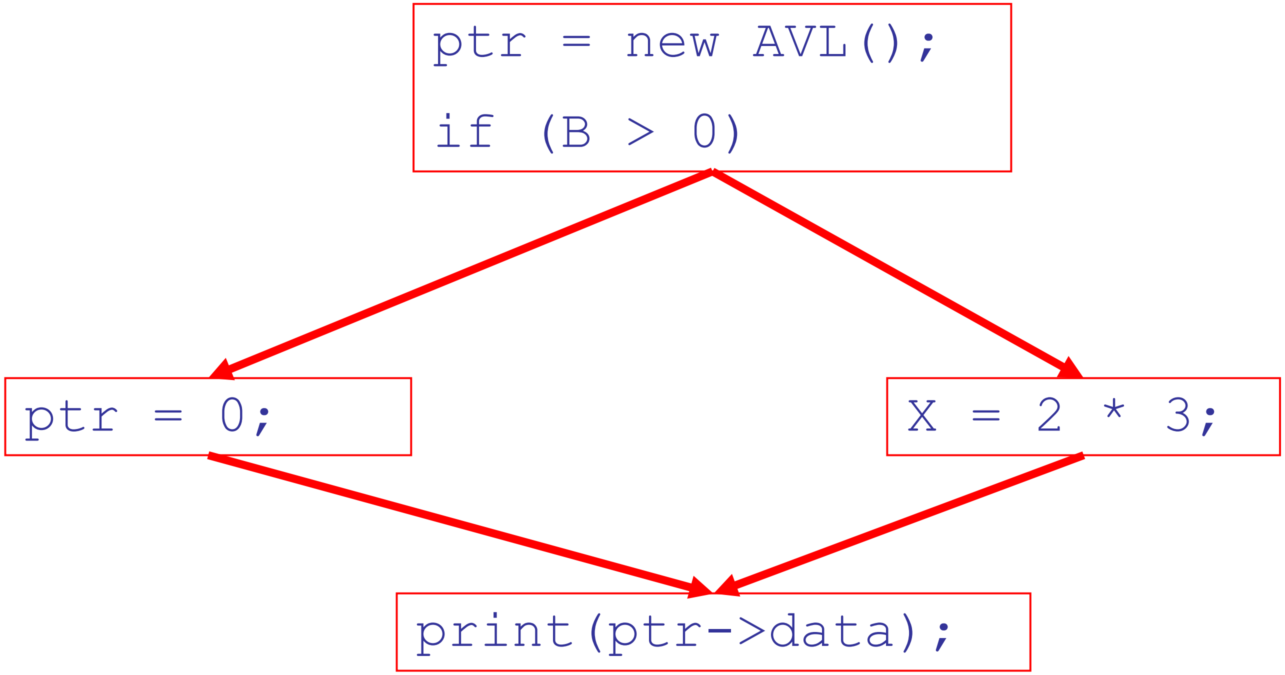
- *Definite* Null Dereference
  - "Whenever execution reaches *ptr at program location L, ptr will be NULL"
- *Potential* Secure Information Leak
  - "We read in a secret string at location L, but there is a possible future public use of it"

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?



```
ptr = new AVL();
if (B > 0)
```

```
ptr = 0;
```

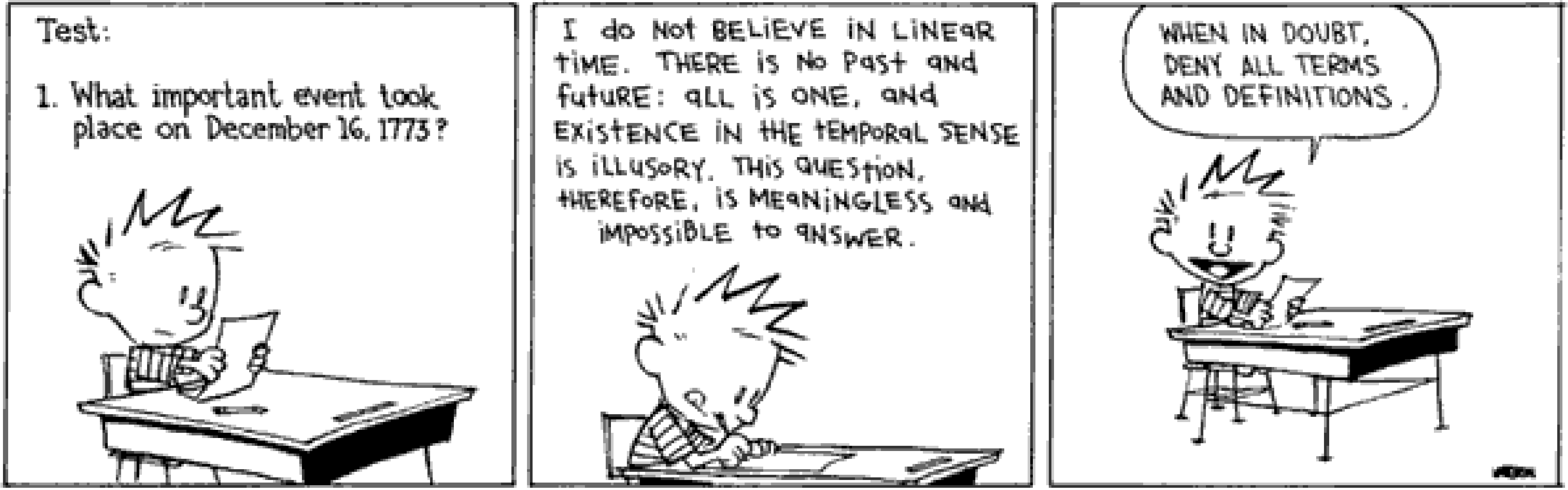
```
X = 2 * 3;
```

```
print(ptr->data);
```

Correctness (Cont.)

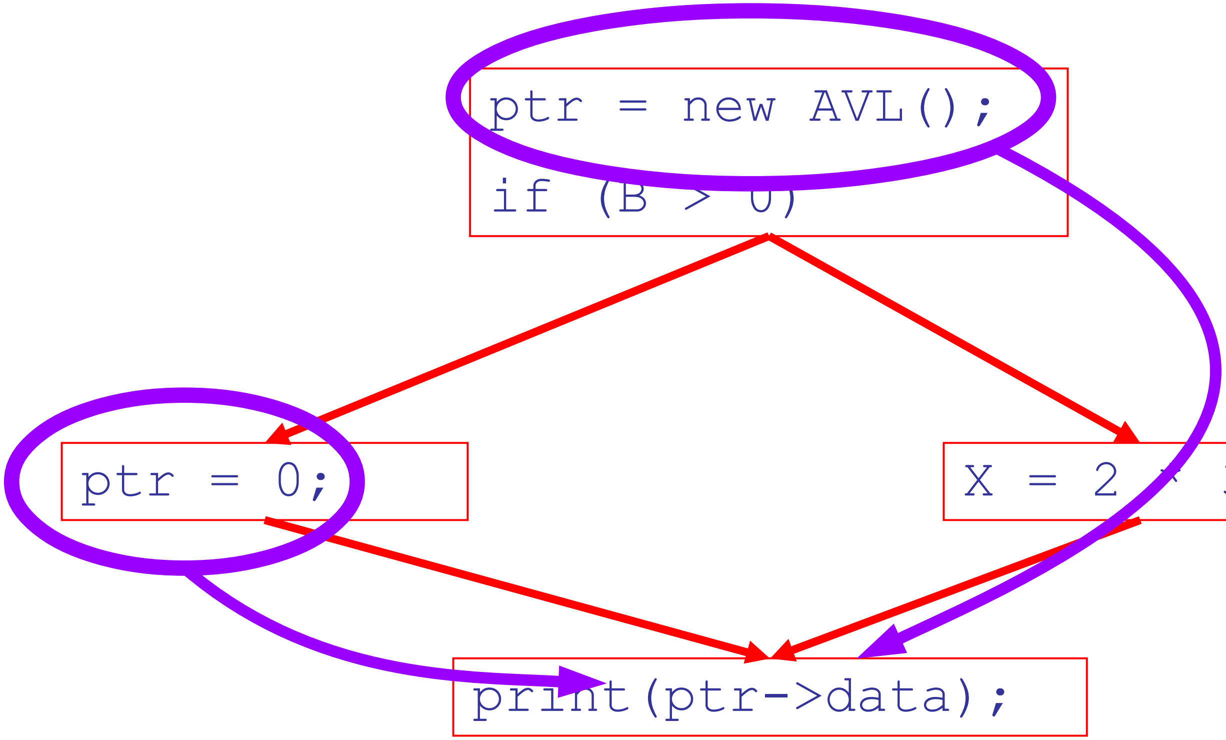
To determine that a use of x is always null, we must know this correctness condition:

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


Test:
1. What important event took place on December 16, 1773?
I do Not BELIEVE IN LINEAR TIME. THERE IS NO PAST AND FUTURE: ALL IS ONE, AND EXISTENCE IN THE TEMPORAL SENSE IS ILLUSORY. THIS QUESTION, THEREFORE, IS MEANINGLESS AND IMPOSSIBLE TO ANSWER.
WHEN IN DOUBT, DENY ALL TERMS AND DEFINITIONS.

Analysis Example Revisited

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



```
ptr = new AVL();
if (B > 0)
```

```
ptr = 0;
```

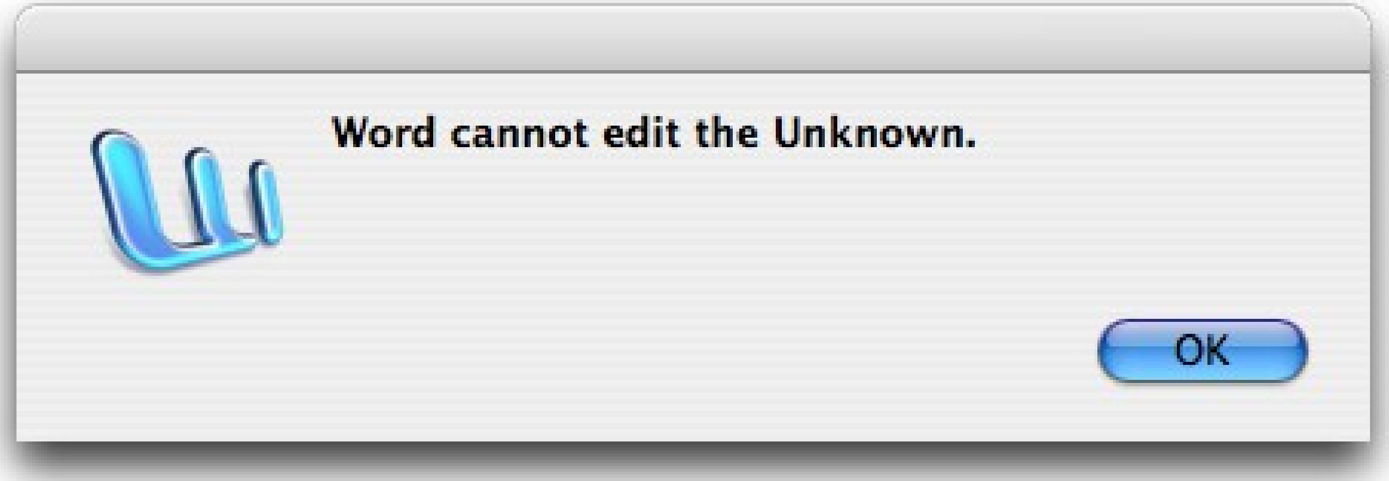
```
X = 2 * 3;
```

```
print(ptr->data);
```

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*!**



Word cannot edit the Unknown.

Undecidability of Program Properties

- **Rice's Theorem**: Most interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - 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
  - 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
- Let's call this *truthiness*

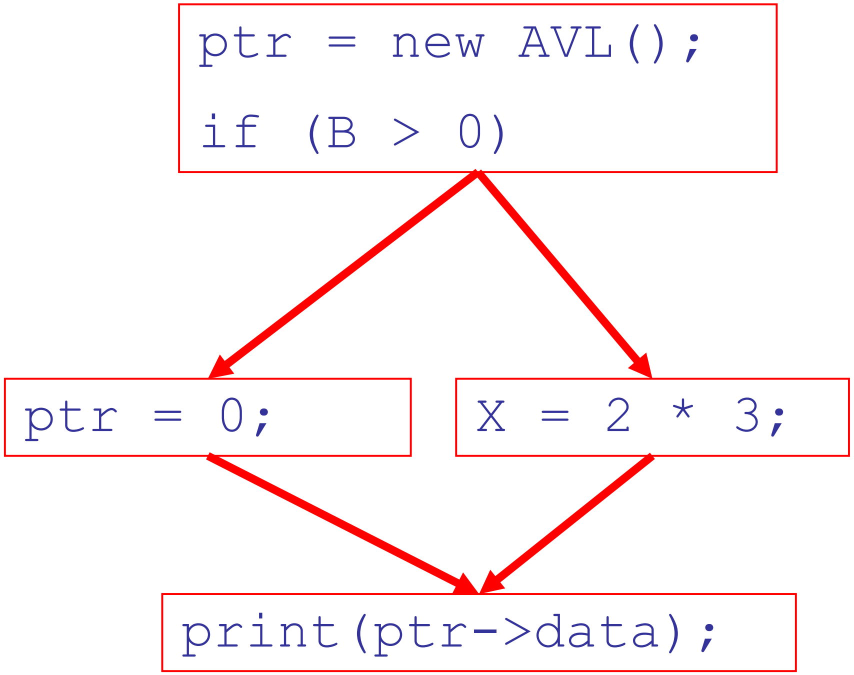
Conservative Program Analyses

- 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

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

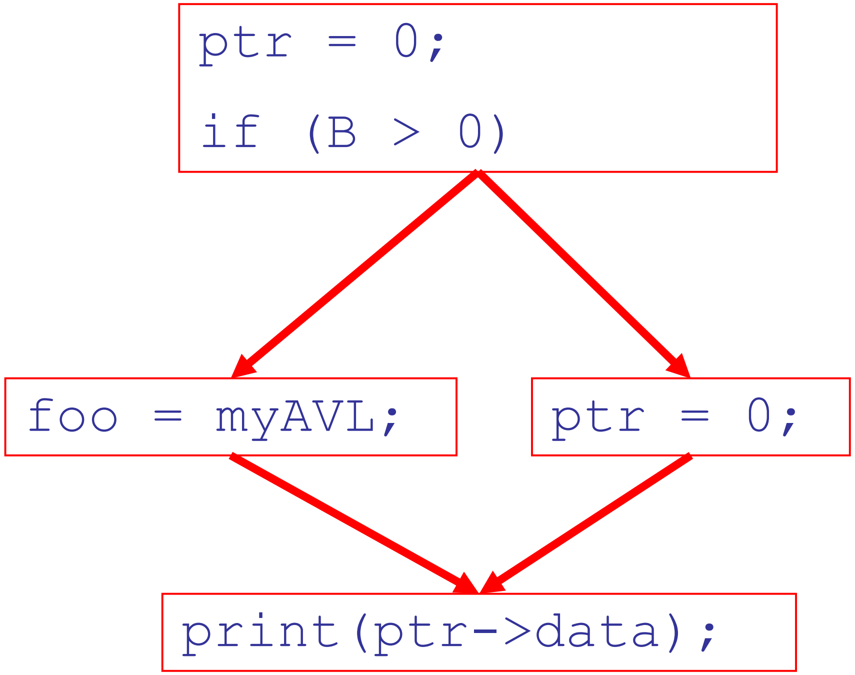


```
ptr = new AVL();
if (B > 0)
```

```
ptr = 0;
```

```
X = 2 * 3;
```

```
print(ptr->data);
```



```
ptr = 0;
if (B > 0)
```

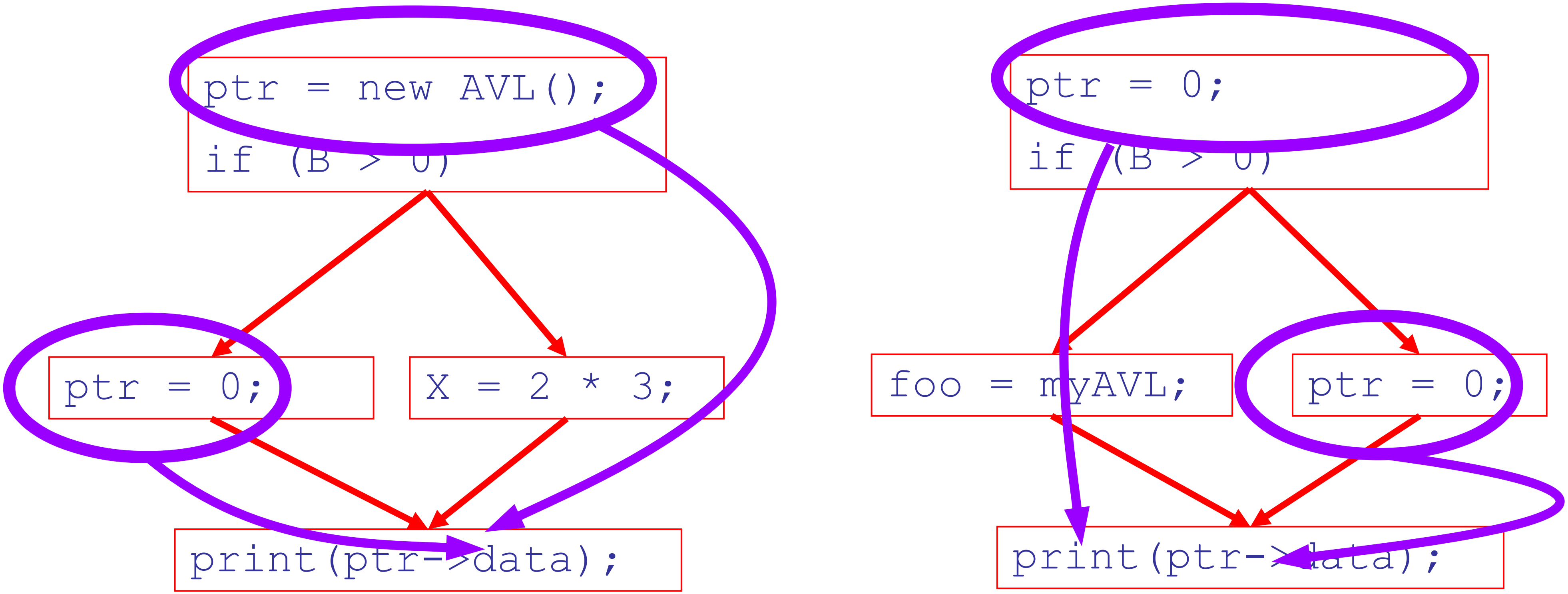
```
foo = myAVL;
```

```
ptr = 0;
```

```
print(ptr->data);
```

Definitely Null Analysis

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



```
ptr = new AVL();
if (B > 0)
```

```
ptr = 0;
```

```
X = 2 * 3;
```

```
print(ptr->data);
```

```
ptr = 0;
if (B > 0)
```

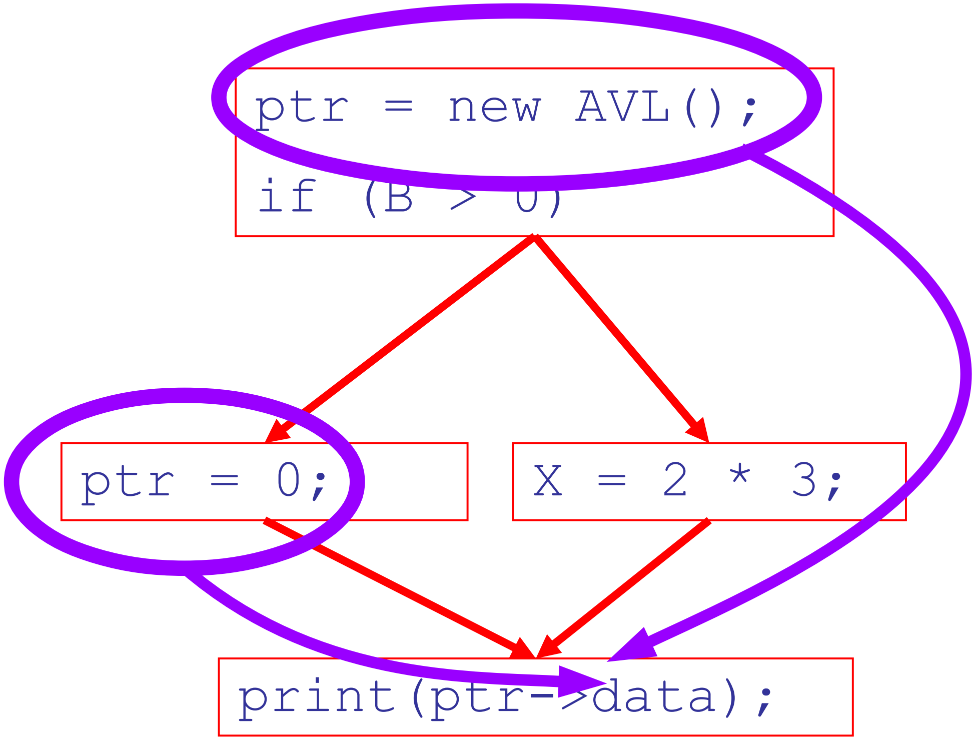
```
foo = myAVL;
```

```
ptr = 0;
```

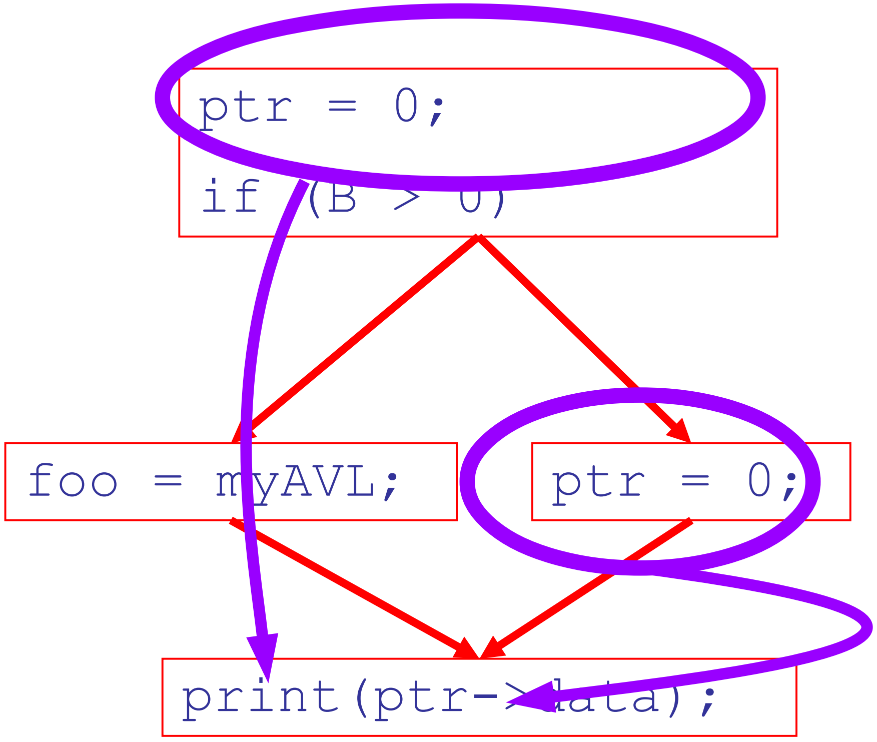
```
print(ptr->data);
```

Definitely Null Analysis

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



No, not always.

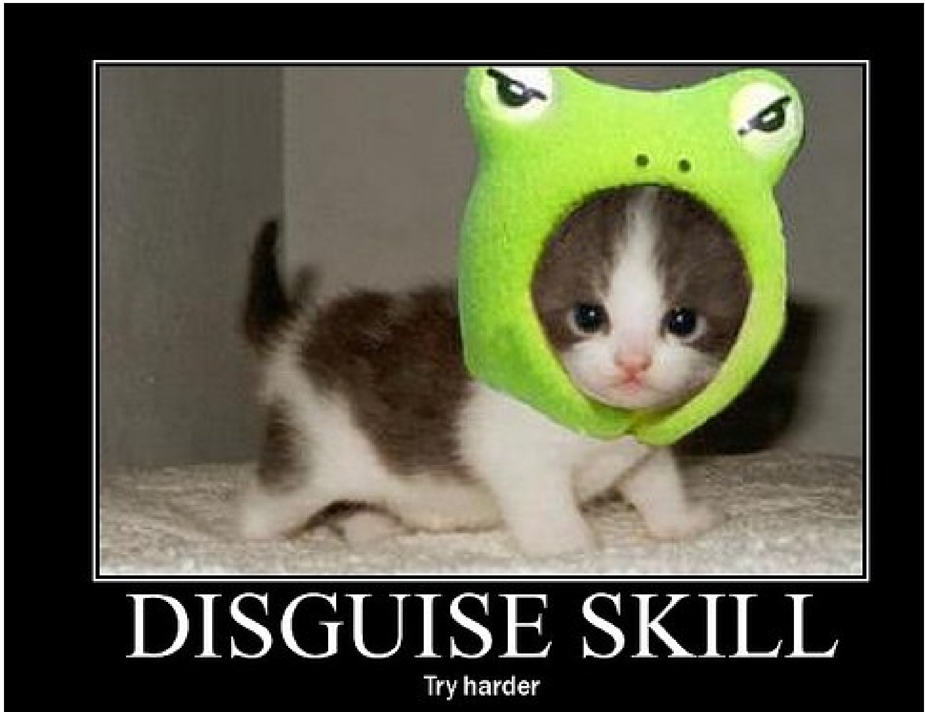


Yes, always.

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)*



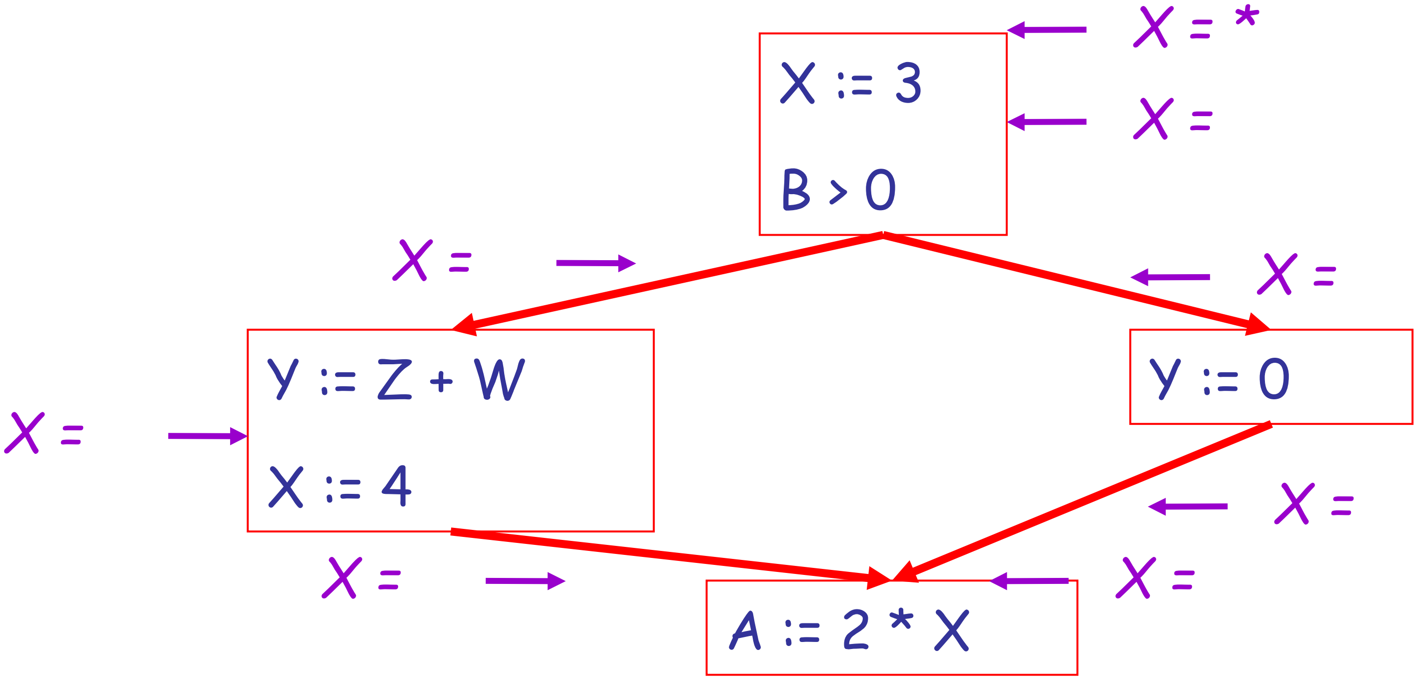
Definitely Null Analysis (Cont.)

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

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

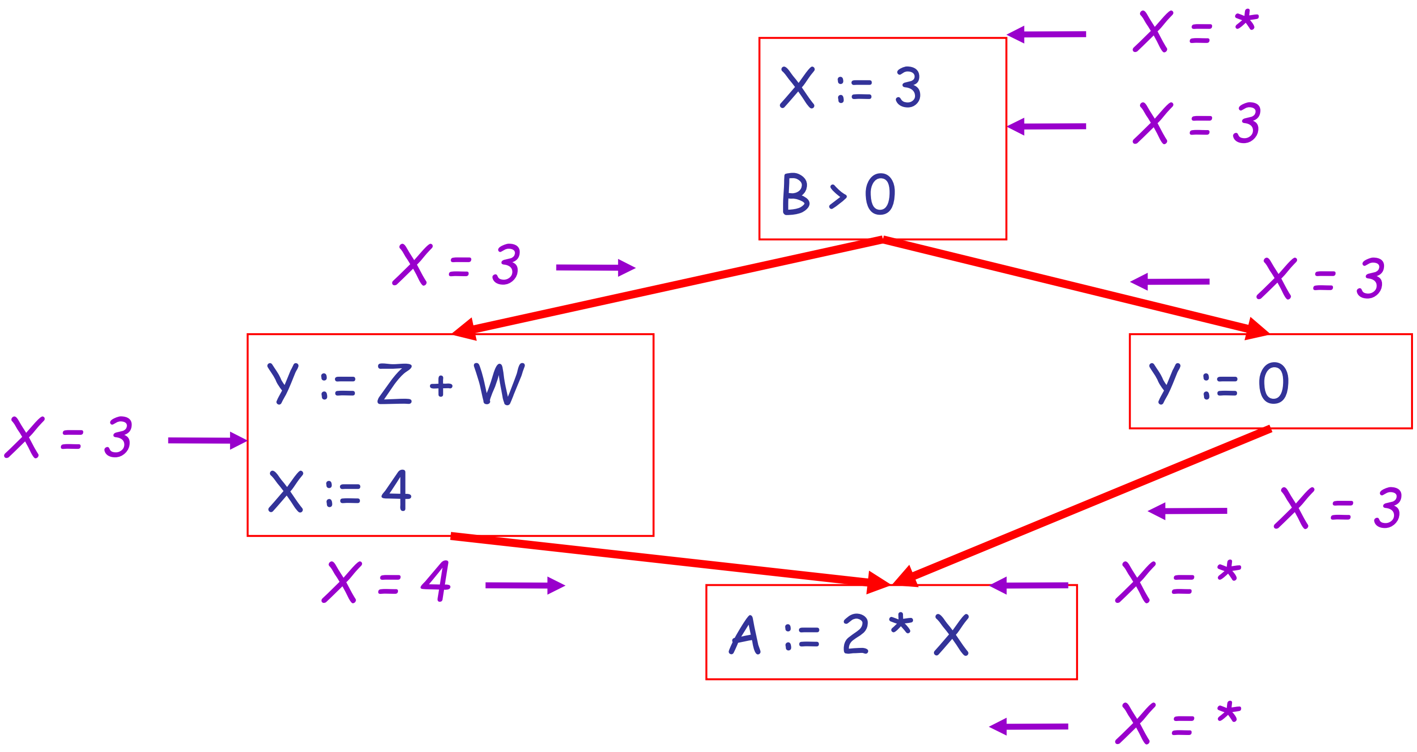
Example

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



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

Example Answers



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.

Curse of the Swine

Sorcery

Exile X target creatures. For each creature exiled this way, its controller puts a 2/2 green Boar creature token onto the battlefield.

Another imminent battle subsided in busy snuffling and carefree rooting.

James Ryman

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?

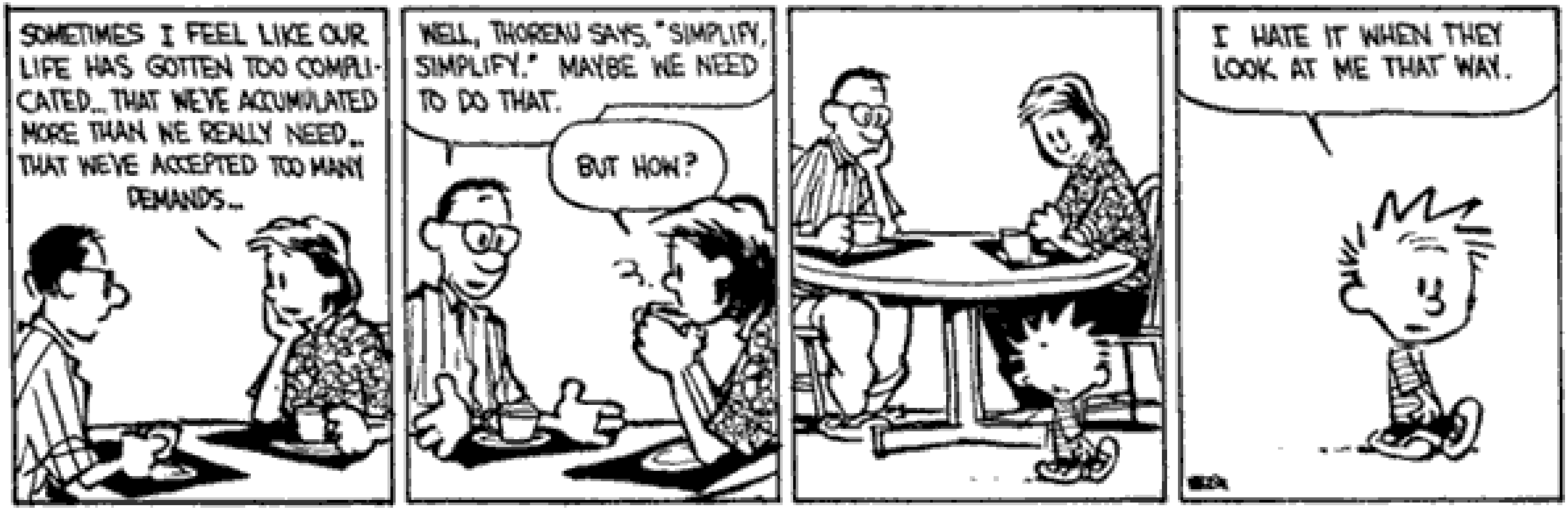
Using Abstract Information

- 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


SOMETIMES I FEEL LIKE OUR LIFE HAS GOTTEN TOO COMPLICATED.. THAT WE'VE ACCUMULATED MORE THAN WE REALLY NEED.. THAT WE'VE ACCEPTED TOO MANY DEMANDS..
WELL, THOREAU SAYS, "SIMPLIFY, SIMPLIFY." MAYBE WE NEED TO DO THAT.
BUT HOW?
I HATE IT WHEN THEY LOOK AT ME THAT WAY.

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

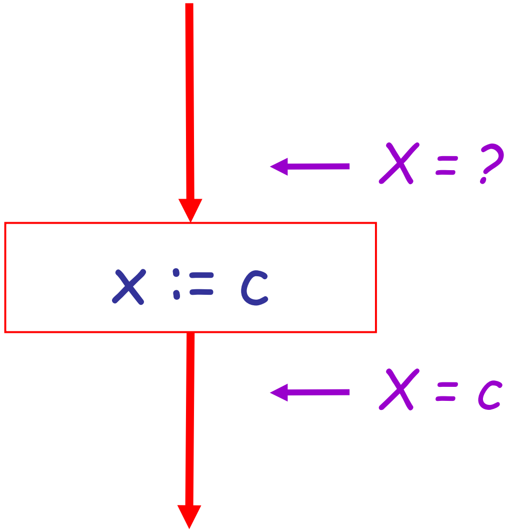
$C_{in}(x,s)$ = value of x before s

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

Transfer Functions

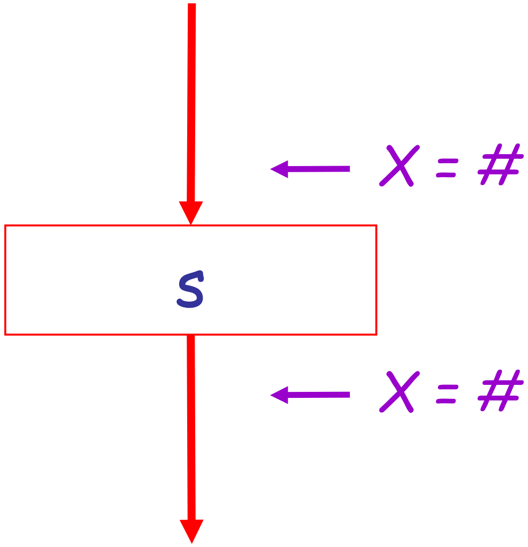
- Define a **transfer function** that transfers information from one statement to another

Rule 1



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

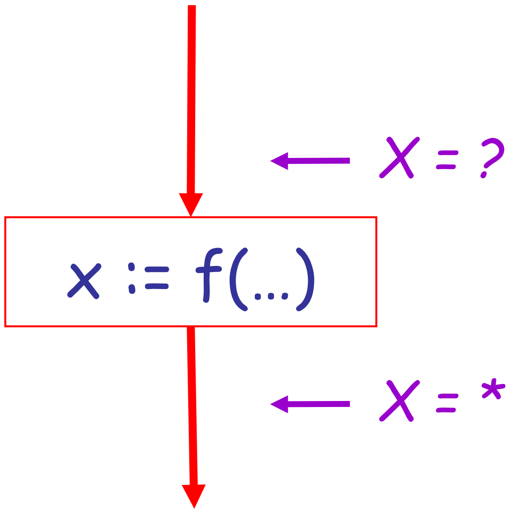
Rule 2



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

Recall: # = “unreachable code”

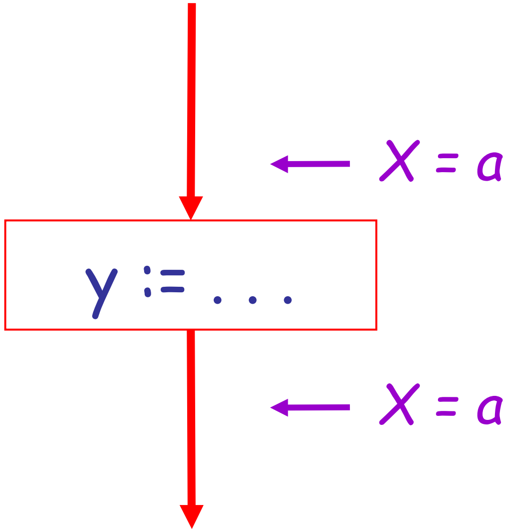
Rule 3



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

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



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

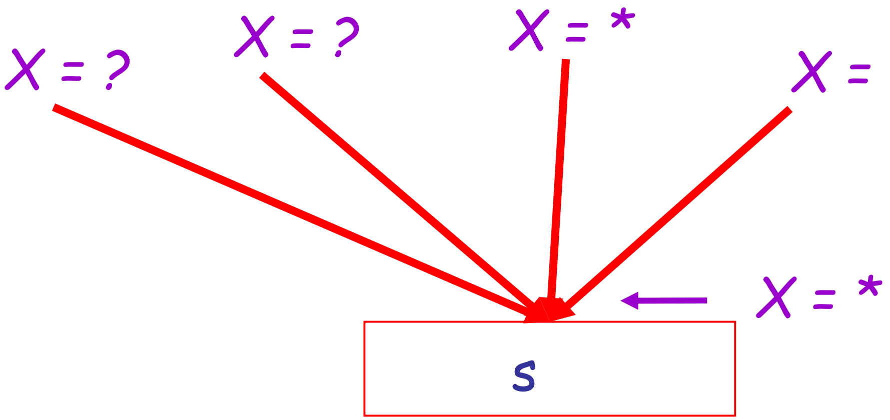
The Other Half

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

- Now we need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information forward along paths

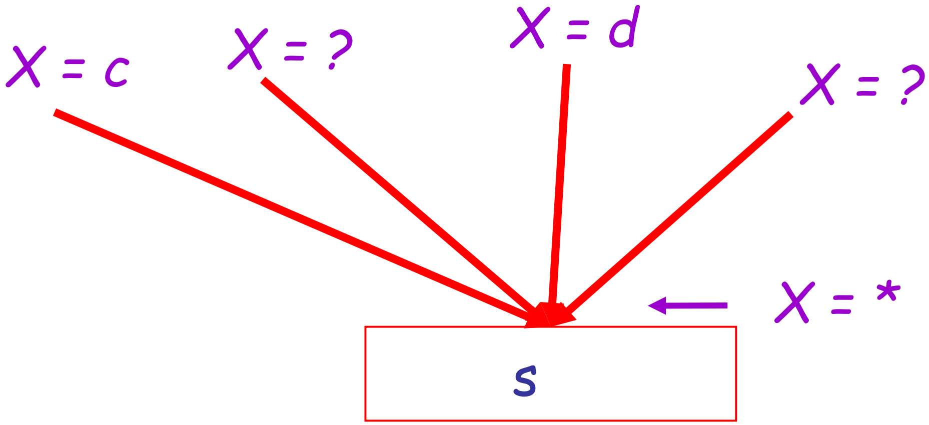
- In the following rules, let statement s have immediate predecessor statements $p_1,\ldots,p_n$

Rule 5



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

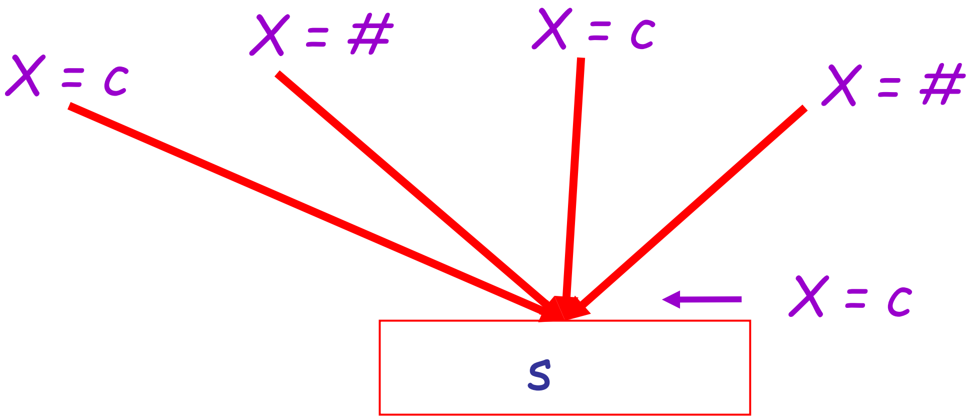
Rule 6



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

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

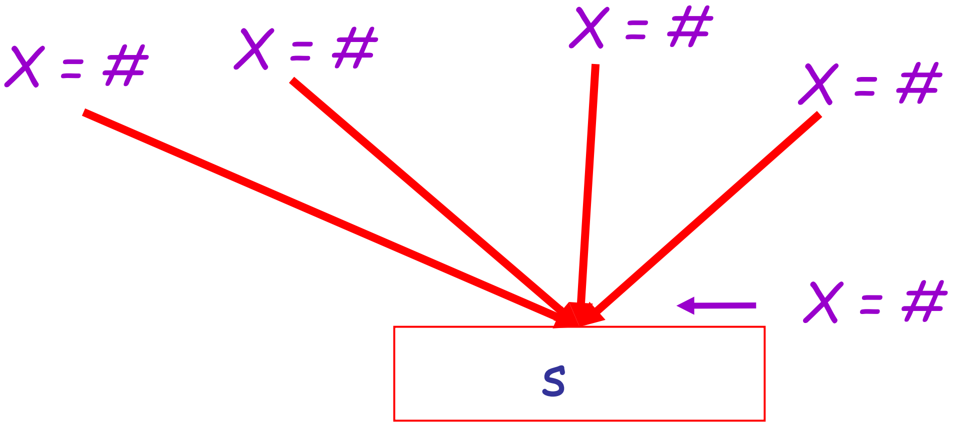
Rule 7



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

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

Rule 8



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

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

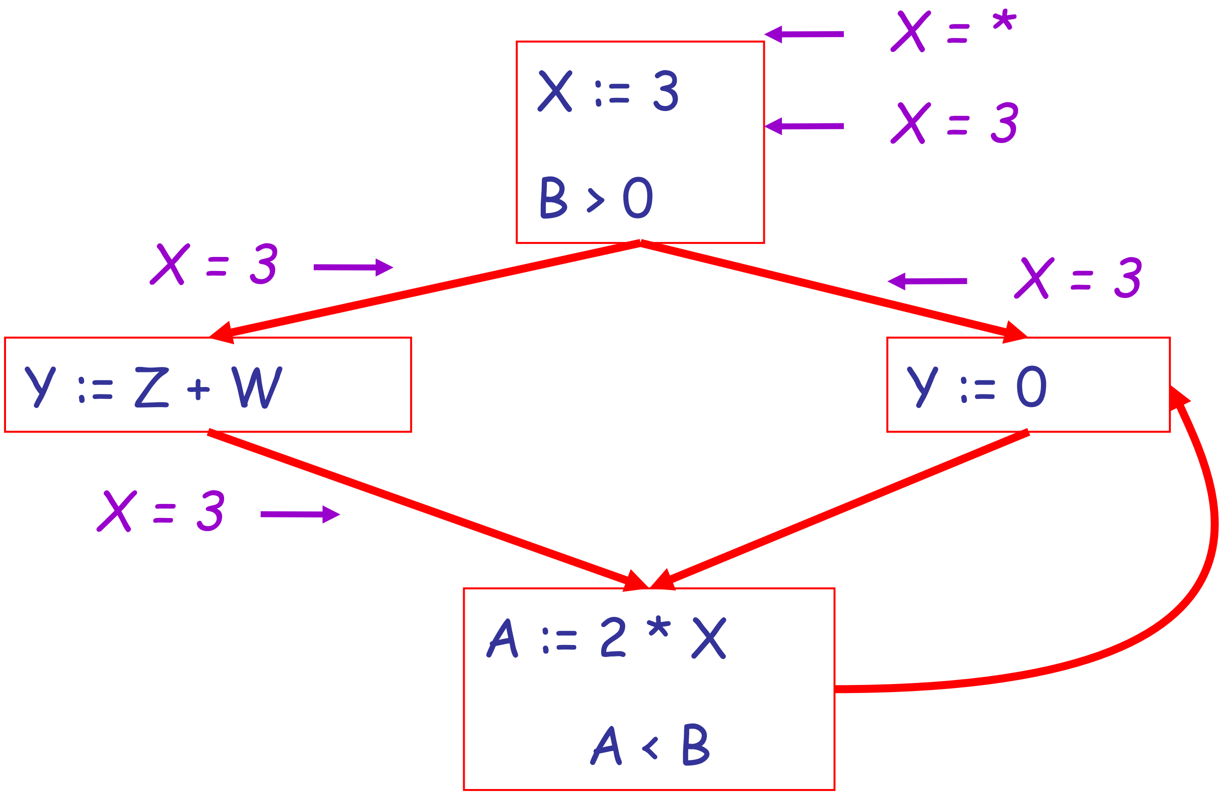
Static Analysis Algorithm

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

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

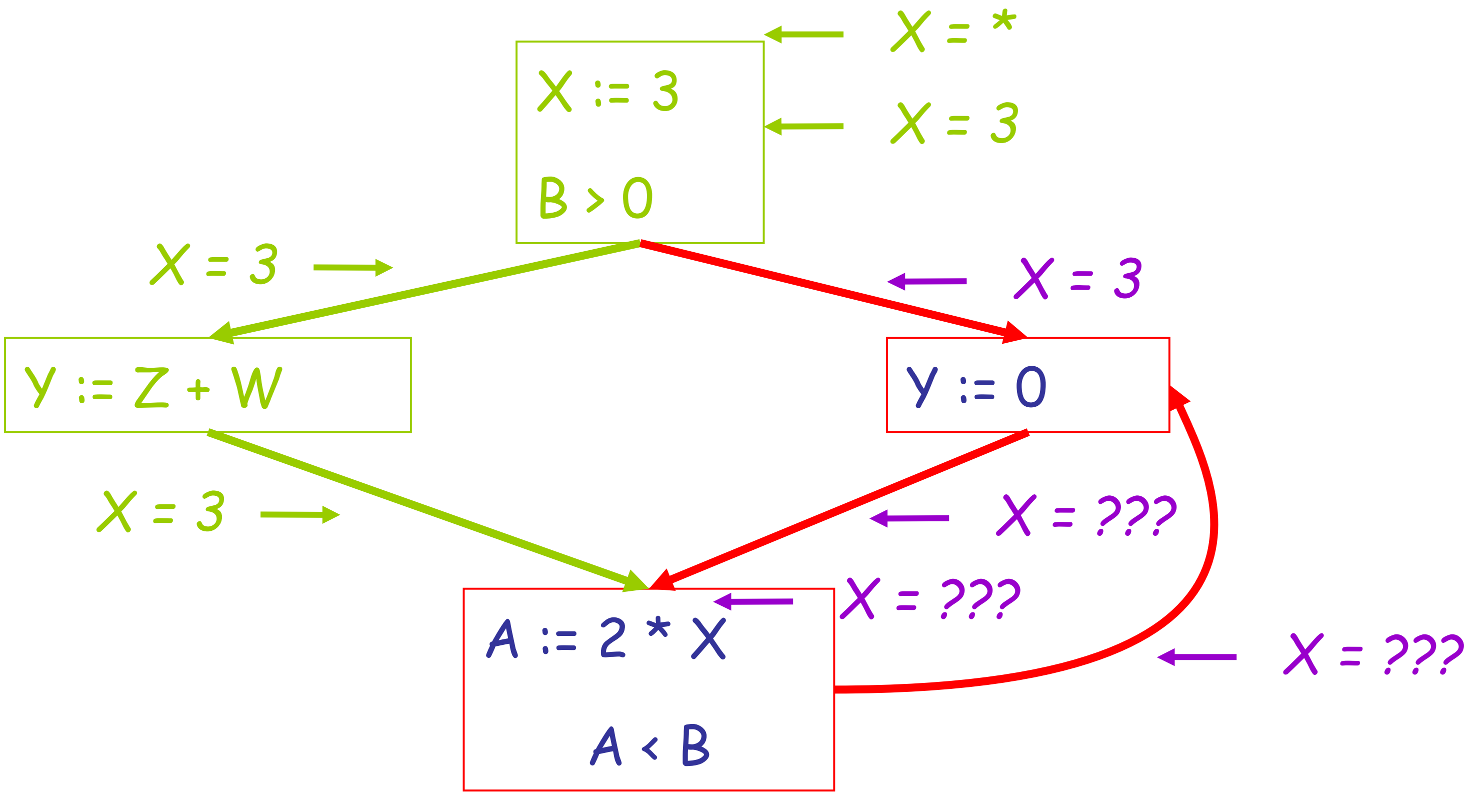
The Value

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



The Value

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



The Value # (Cont.)

- Because of cycles, all points must have values at all times during the analysis

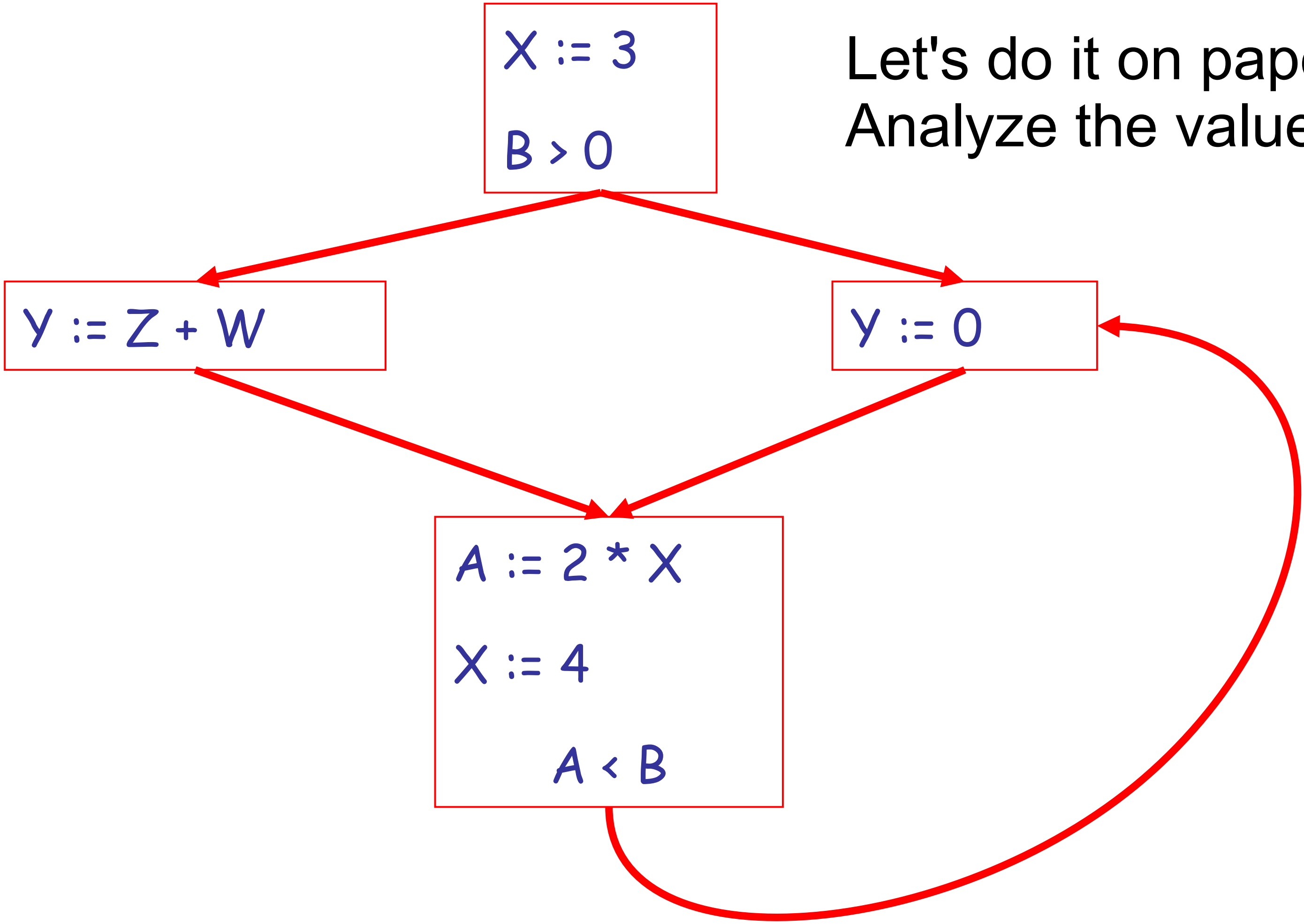
- Intuitively, assigning some initial value allows the analysis to break cycles

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

Sometimes all paths lead to the same place.

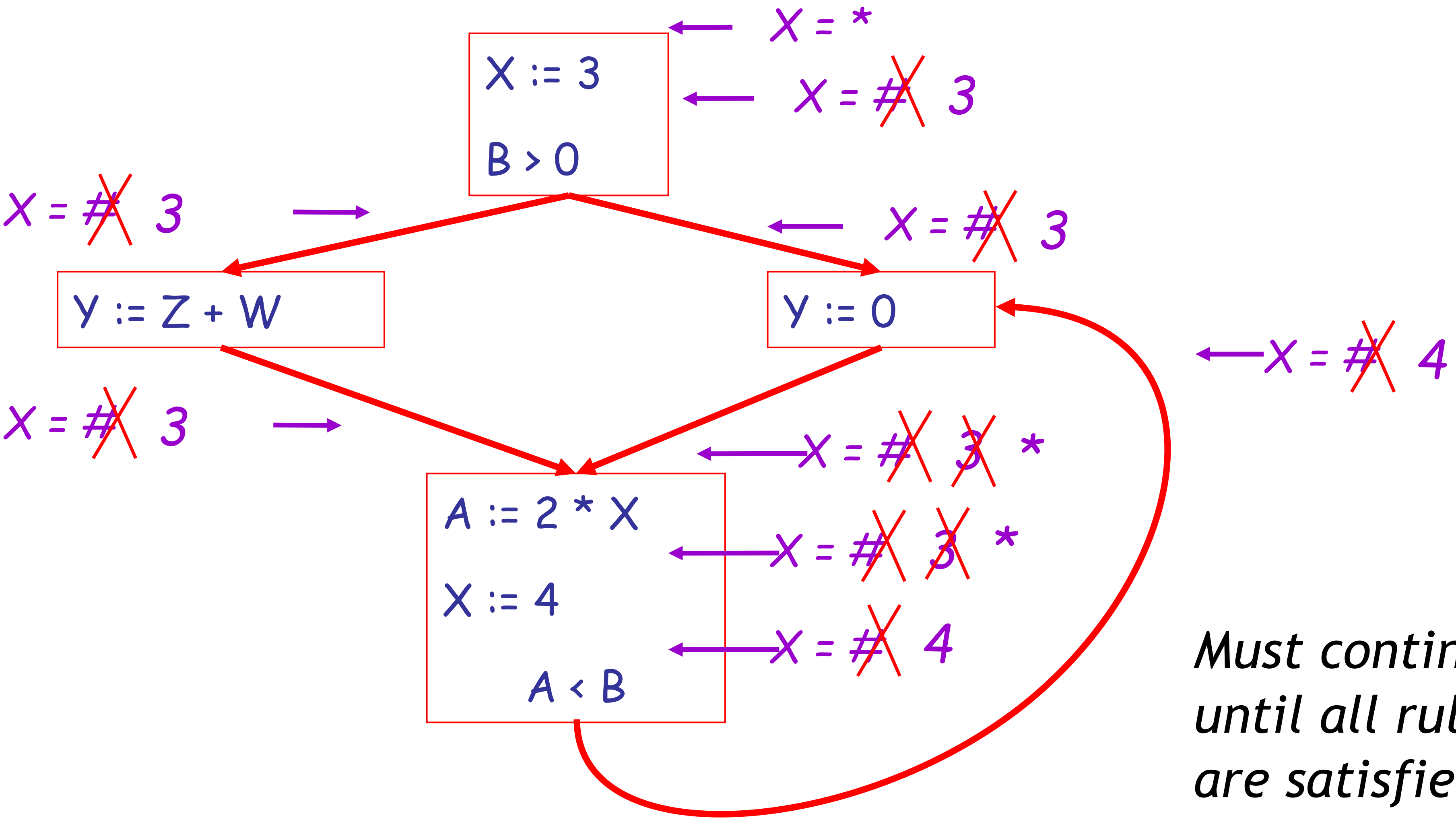
Thus you need #.

Another Example



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

Another Example: Answer



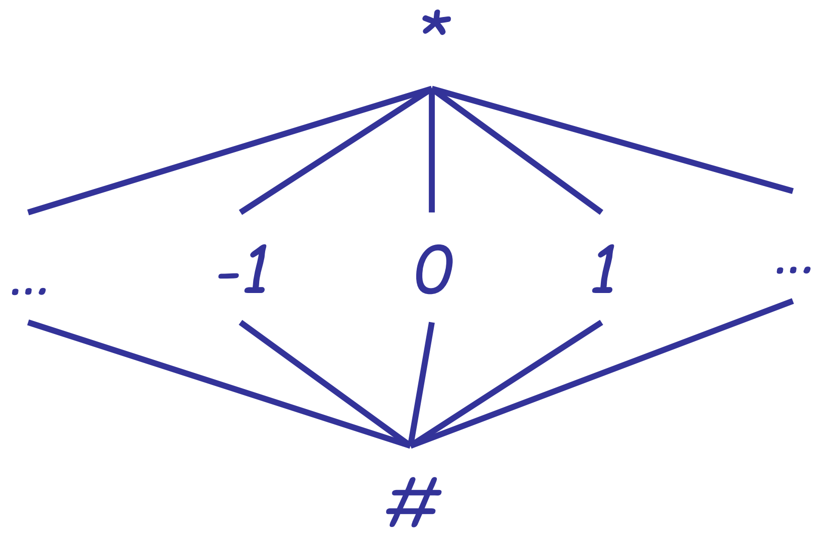
Must continue until all rules are satisfied !

Orderings

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

$$\# \quad < \quad c \quad < \quad *$$

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



I am called a **lattice!**

Orderings (Cont.)

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

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

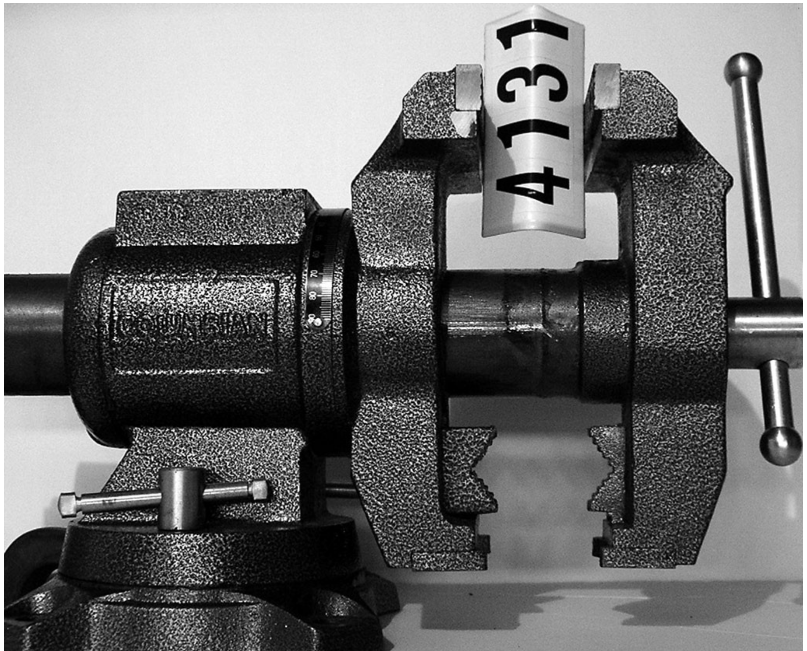
Number Crunching

The algorithm is polynomial in program size:

Number of steps =

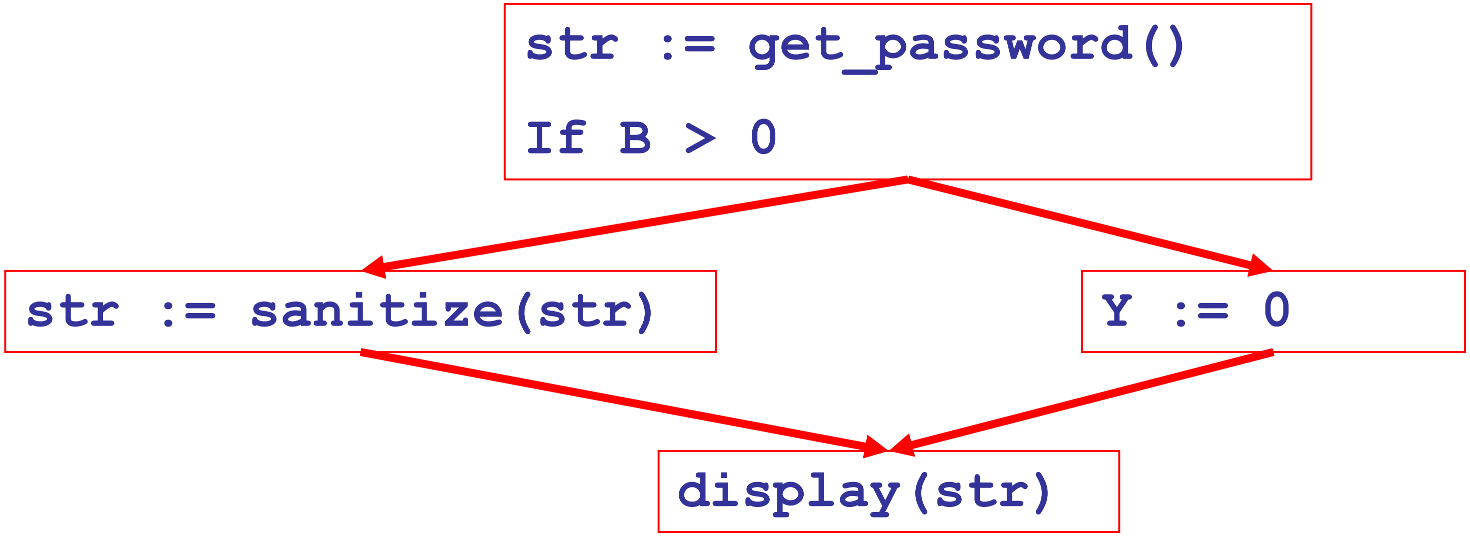
Number of C_(….) values changed * 2 =

(Number of program statements)^2 * 2



“Potential Secure Information Leak” Analysis

Could sensitive information possibly reach an insecure use?



```
str := get_password()
If B > 0
```

```
str := sanitize(str)
```

```
Y := 0
```

```
display(str)
```

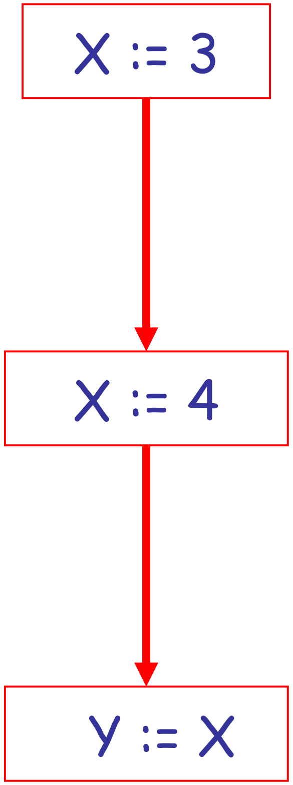
In this example, the password contents can potentially flow into a public display (depending on the value of `B`*)*

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"



X := 3

X := 4

y := X

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

Chronicle.com - Today's News

⊞ Textbook Sales Drop, and University Presses Search for Reasons Why

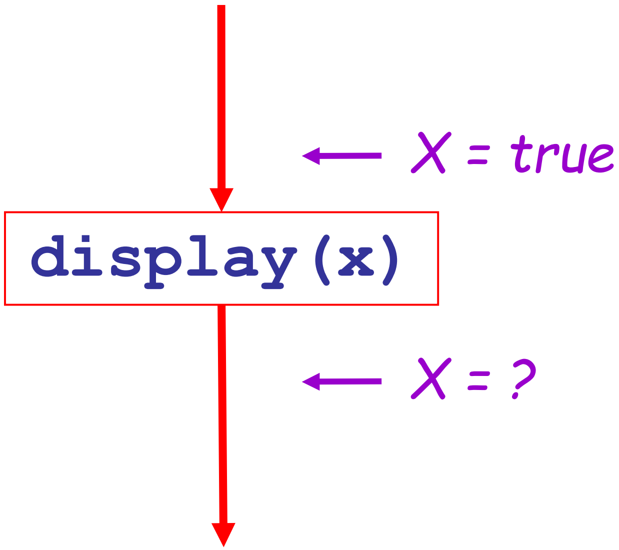
⊞ Students Flock to Web Sites Offering Pirated Textbooks

⊞ Convention Notebook: Students Cover Proce

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



X = true

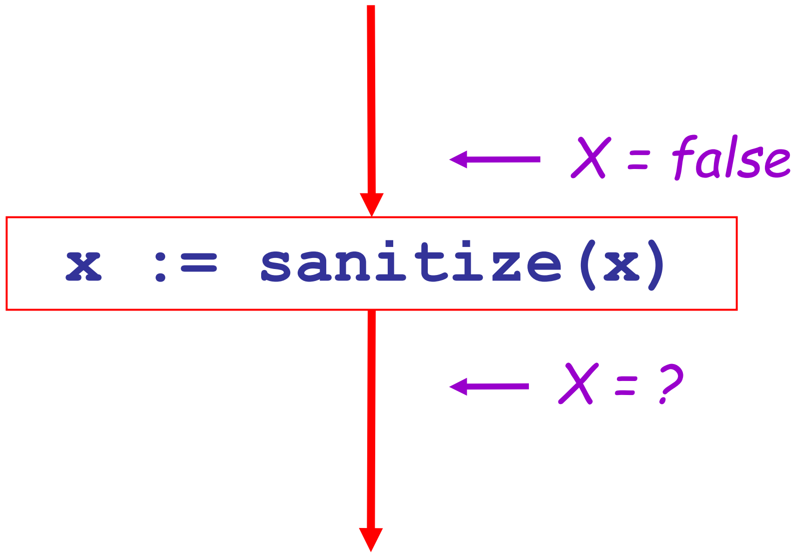
`display(x)`

X = ?

$H_{in}(x, s)$ = 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



X = false

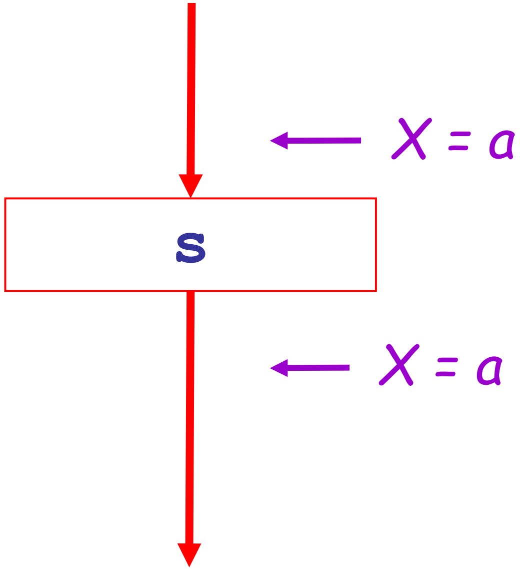
`x := sanitize(x)`

X = ?

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

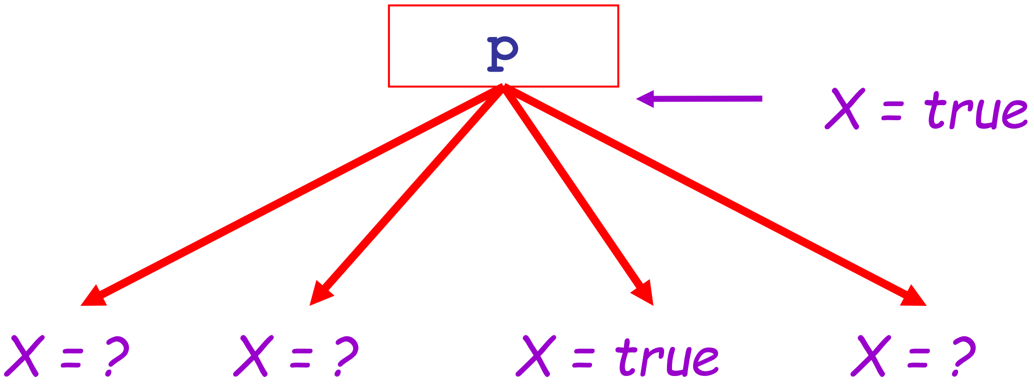
(any subsequent use is safe)

Secure Information Flow Rule 3



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

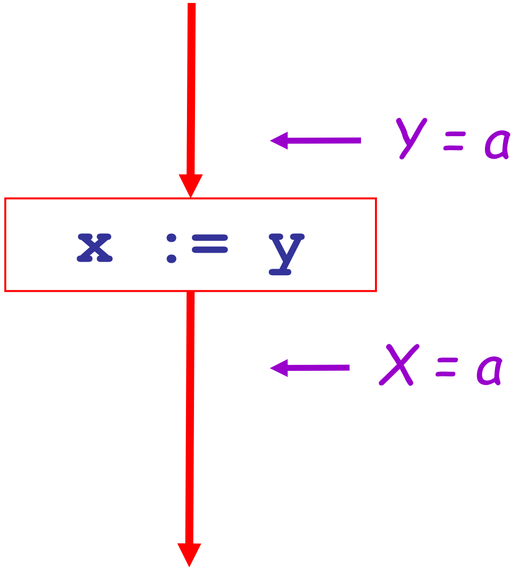
Secure Information Flow Rule 4



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



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

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

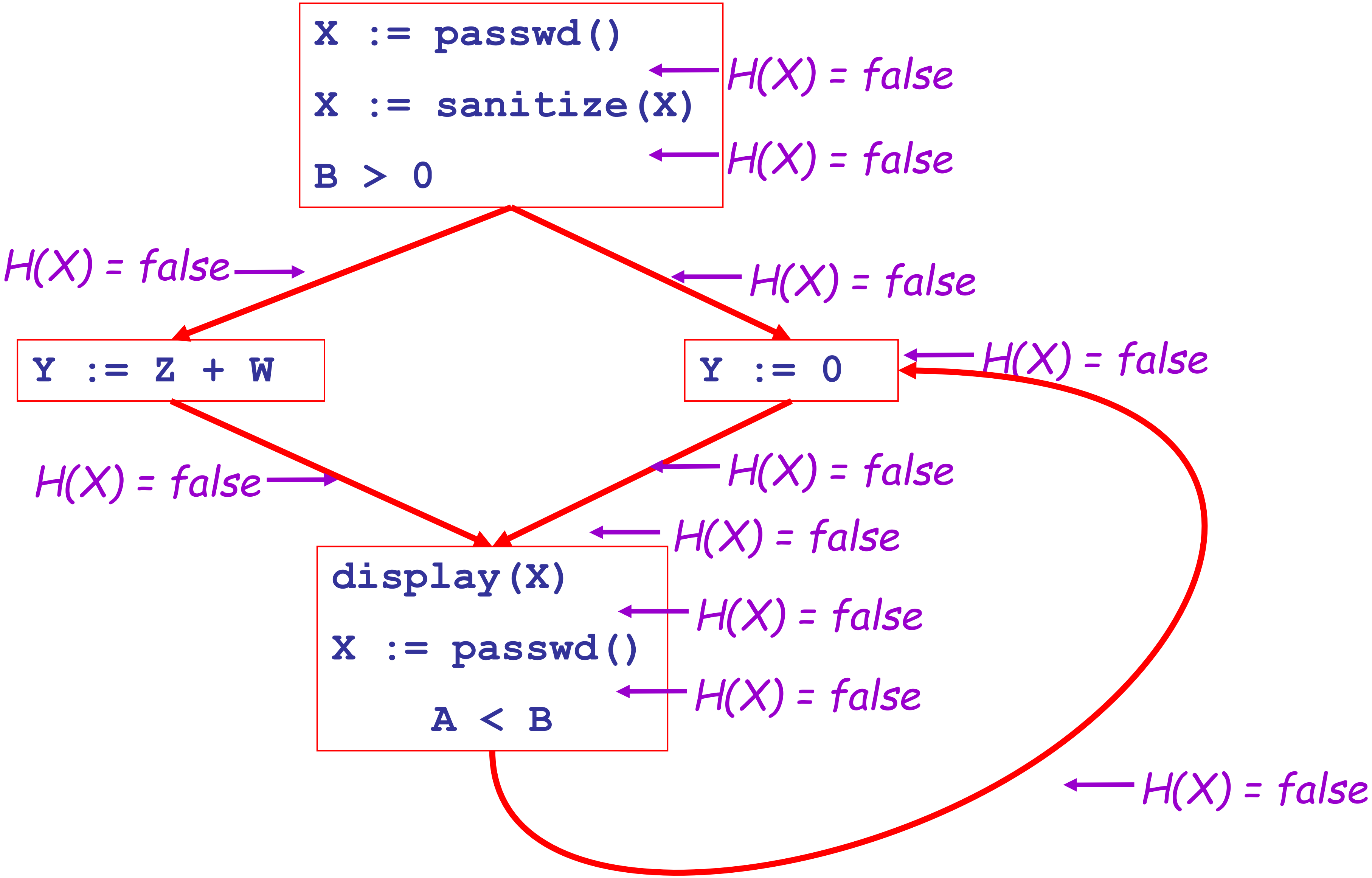
Algorithm

- 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



```
X := passwd()
X := sanitize(X)
B > 0
```

H(X) = false

H(X) = false

H(X) = false

H(X) = false

```
Y := Z + W
```

```
Y := 0
```

H(X) = false

H(X) = false

H(X) = false

H(X) = false

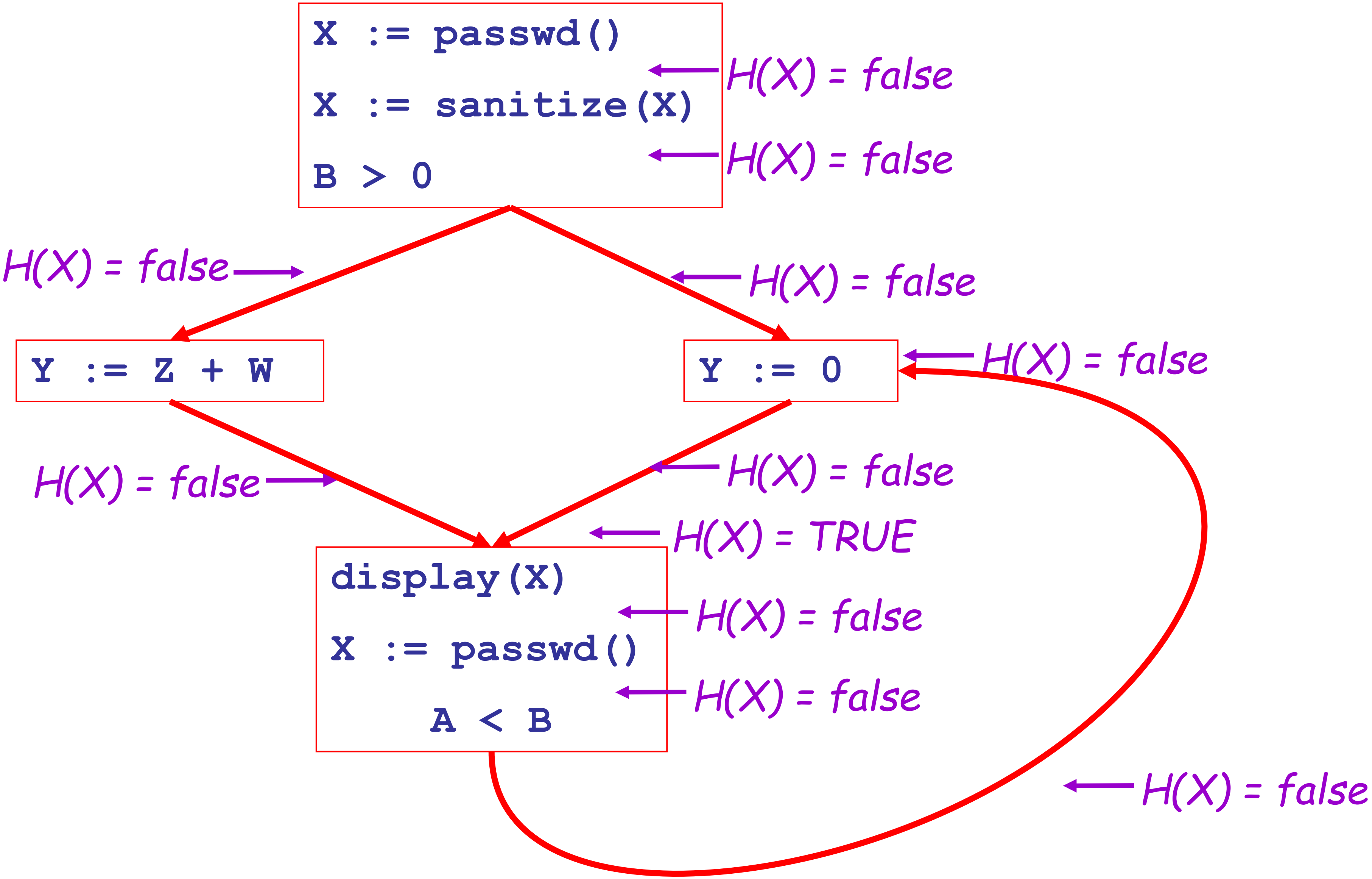
```
display(X)
X := passwd()
    A < B
```

H(X) = false

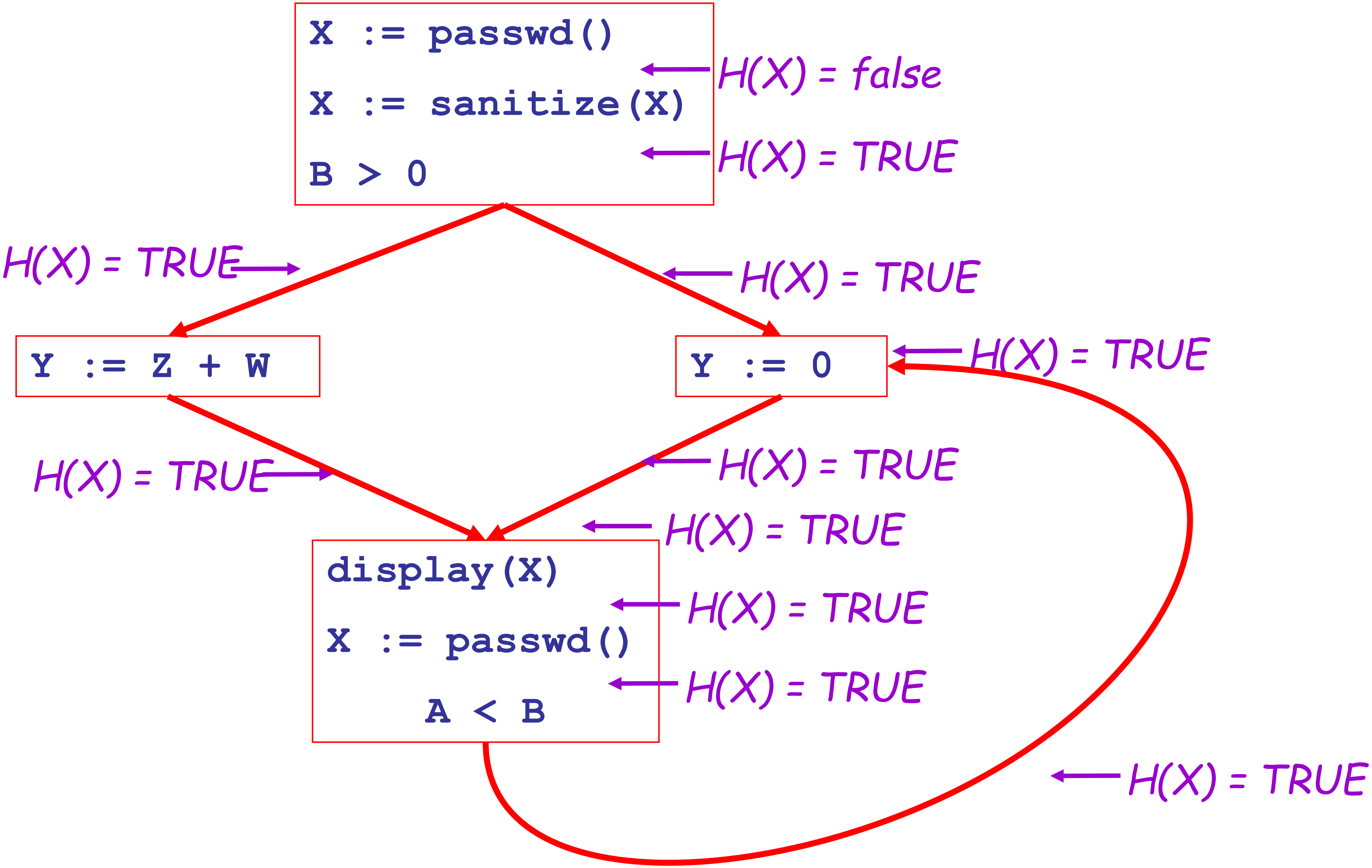
H(X) = false

H(X) = false

Secure Information Flow Example



Secure Information Flow Example



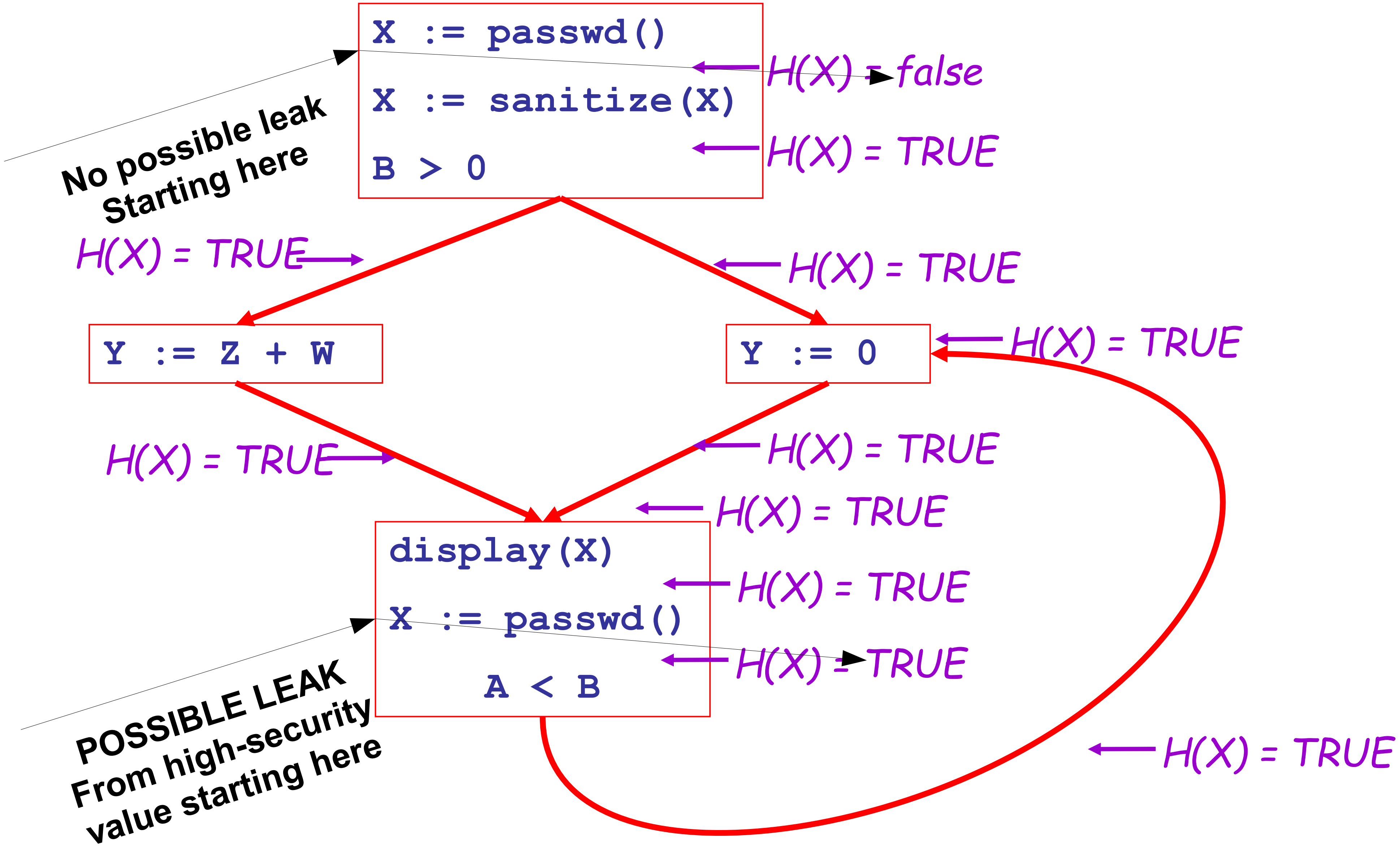
```
X := passwd()
X := sanitize(X)
B > 0
```

```
Y := Z + W
```

```
Y := 0
```

```
display(X)
X := passwd()
A < B
```

Secure Information Flow Example



```
X := passwd()
X := sanitize(X)
B > 0
```

H(X) = false

H(X) = TRUE

No possible leak
Starting here

H(X) = TRUE

H(X) = TRUE

```
Y := Z + W
```

```
Y := 0
```

H(X) = TRUE

H(X) = TRUE

H(X) = TRUE

H(X) = TRUE

```
display(X)
X := passwd()
A < B
```

H(X) = TRUE

H(X) = TRUE

POSSIBLE LEAK
From high-security
value starting here

H(X) = TRUE

Termination

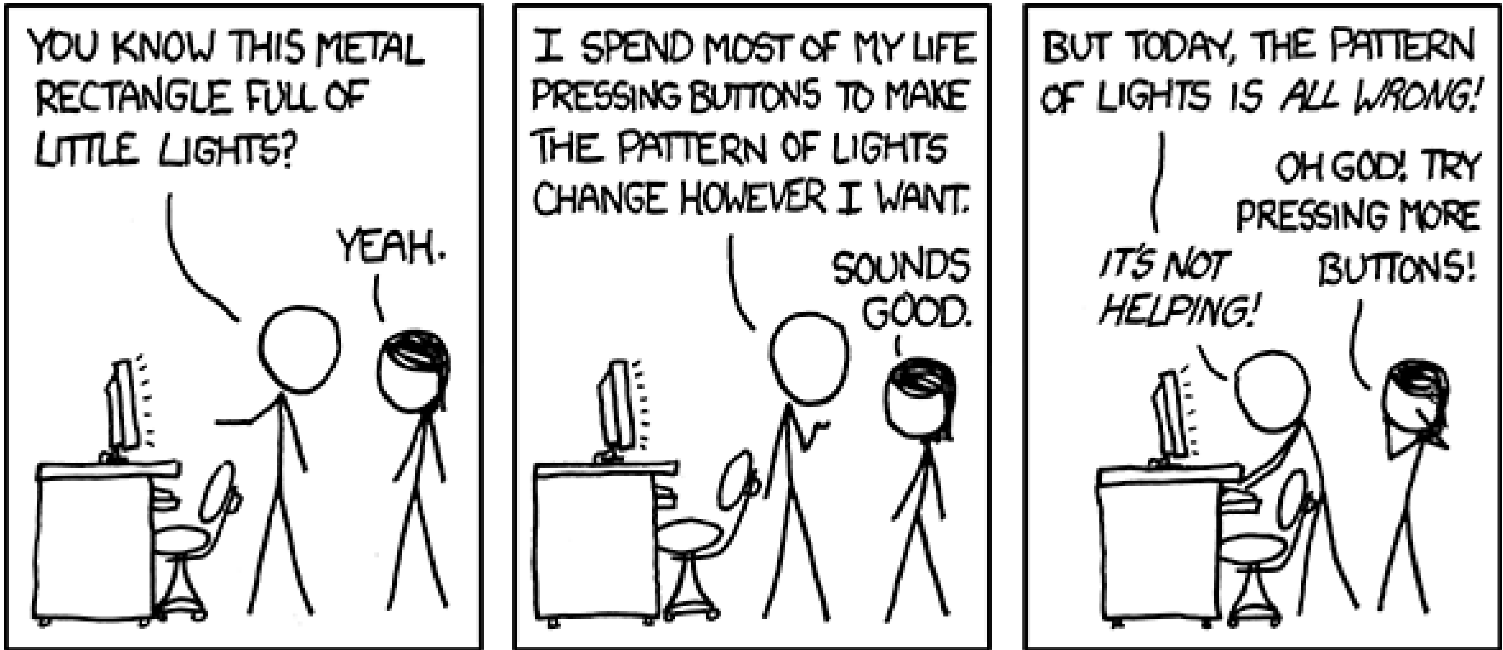
- A value can change from false to true, but not the other way around

- 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?


YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS ALL WRONG!
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

Static Analysis

- Discuss with your neighbor; I will call on you
- 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?

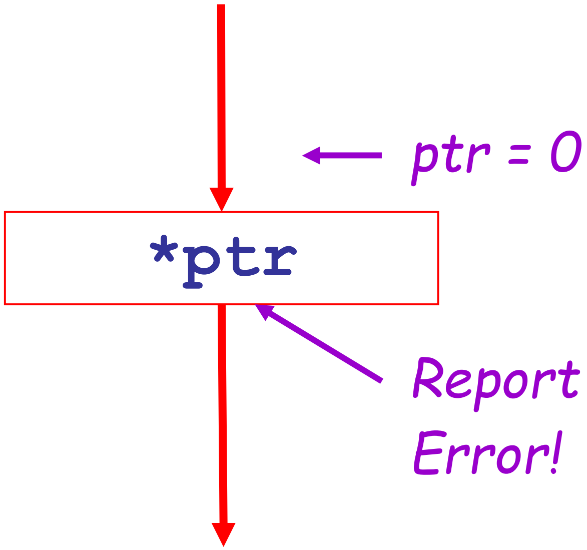
Abstract Information

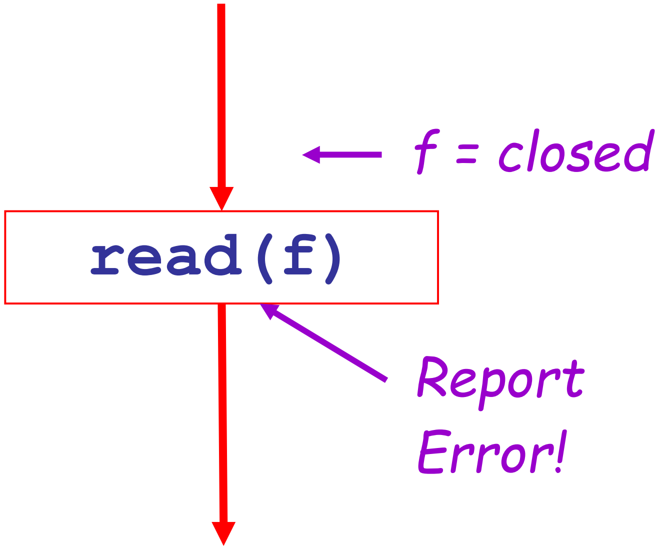
- We will keep track of an abstract value for a given file handle variable
- Values and Interpretations

*	file handle state is unknown
#	haven't reached here yet
closed	file handle is closed
open	file handle is open

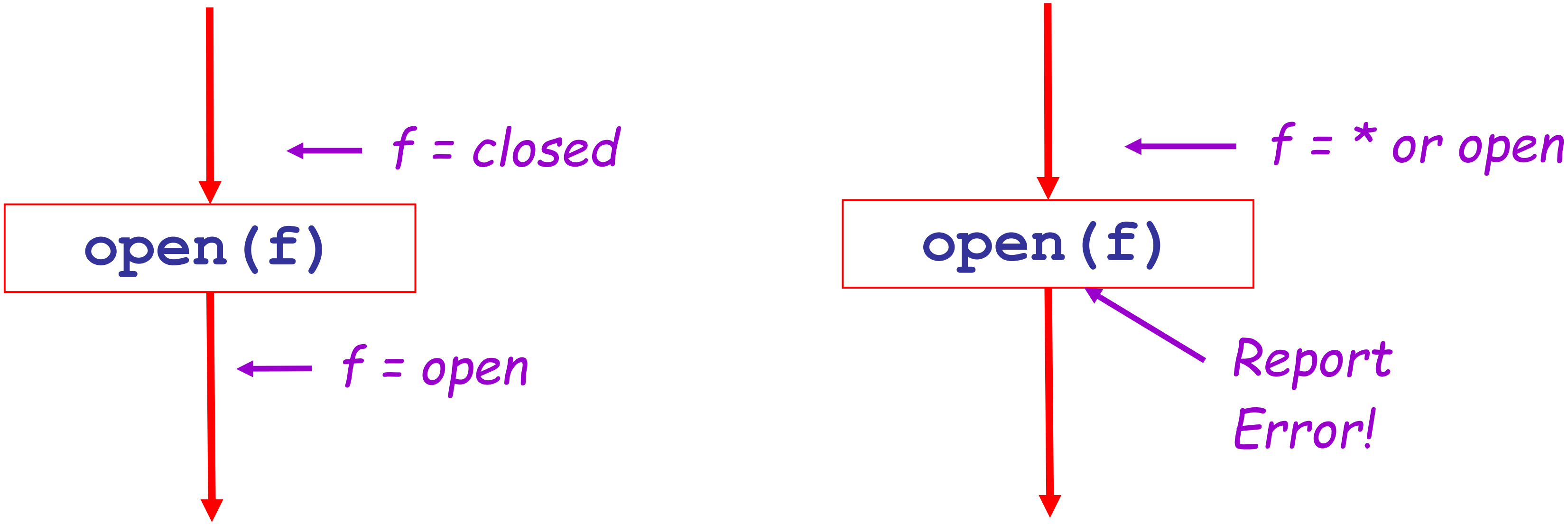
Rules

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

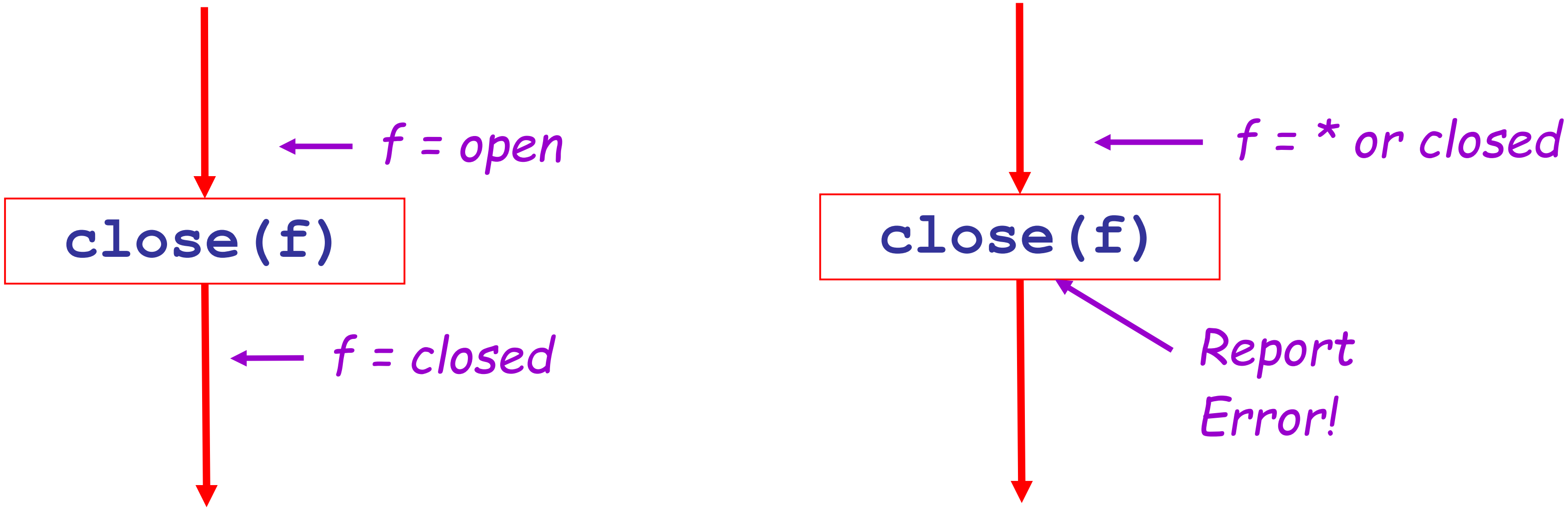




Rules: open



Rules: close



f = open

`close(f)`

f = closed

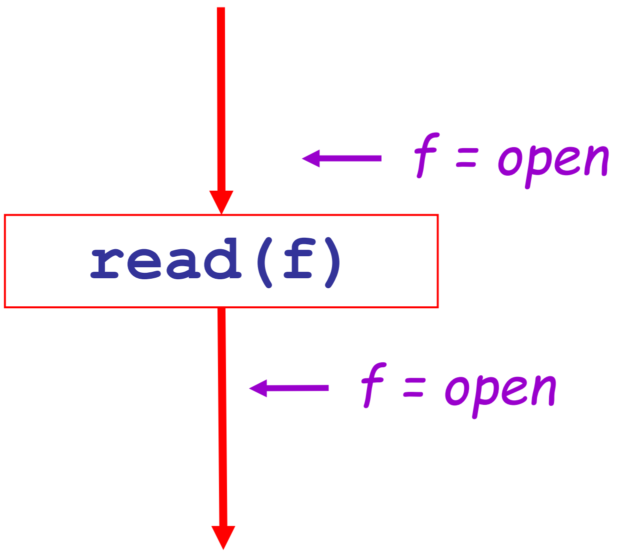
*f = * or closed*

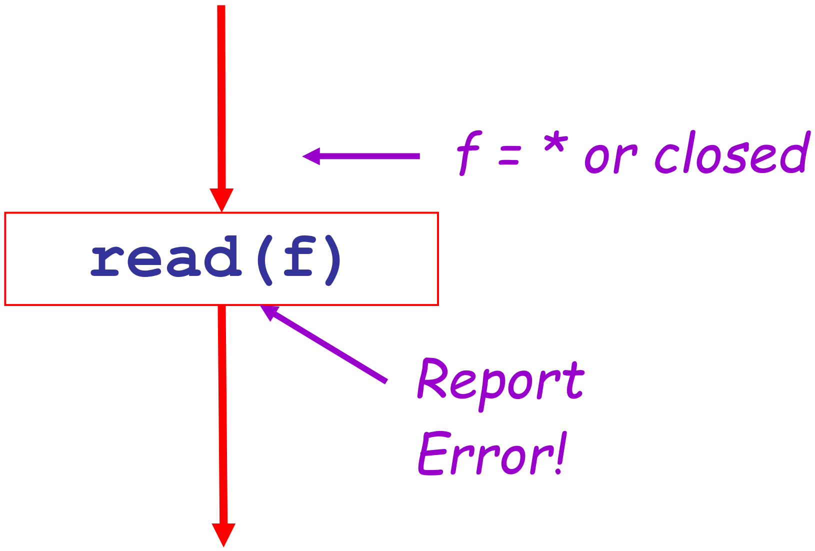
`close(f)`

Report Error!

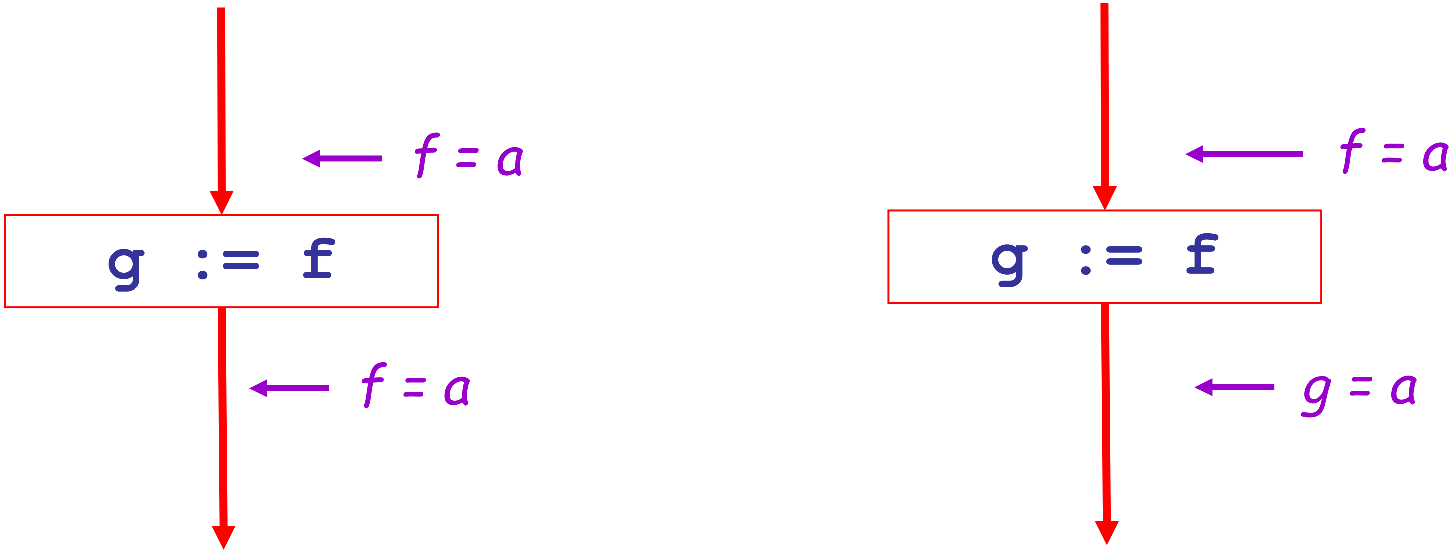
Rules: read/write

(write is identical)





Rules: Assignment



$f = a$

`g := f`

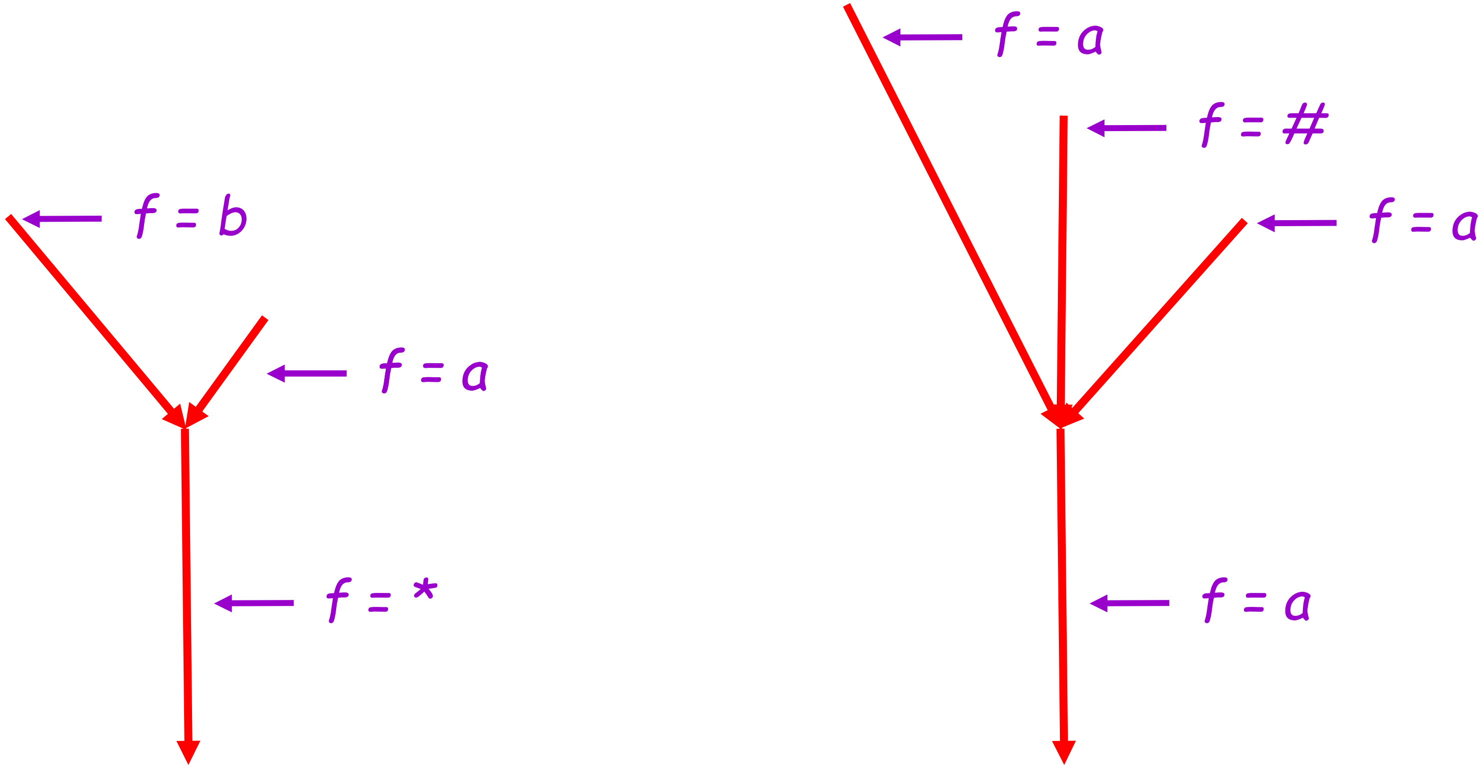
$f = a$

$f = a$

`g := f`

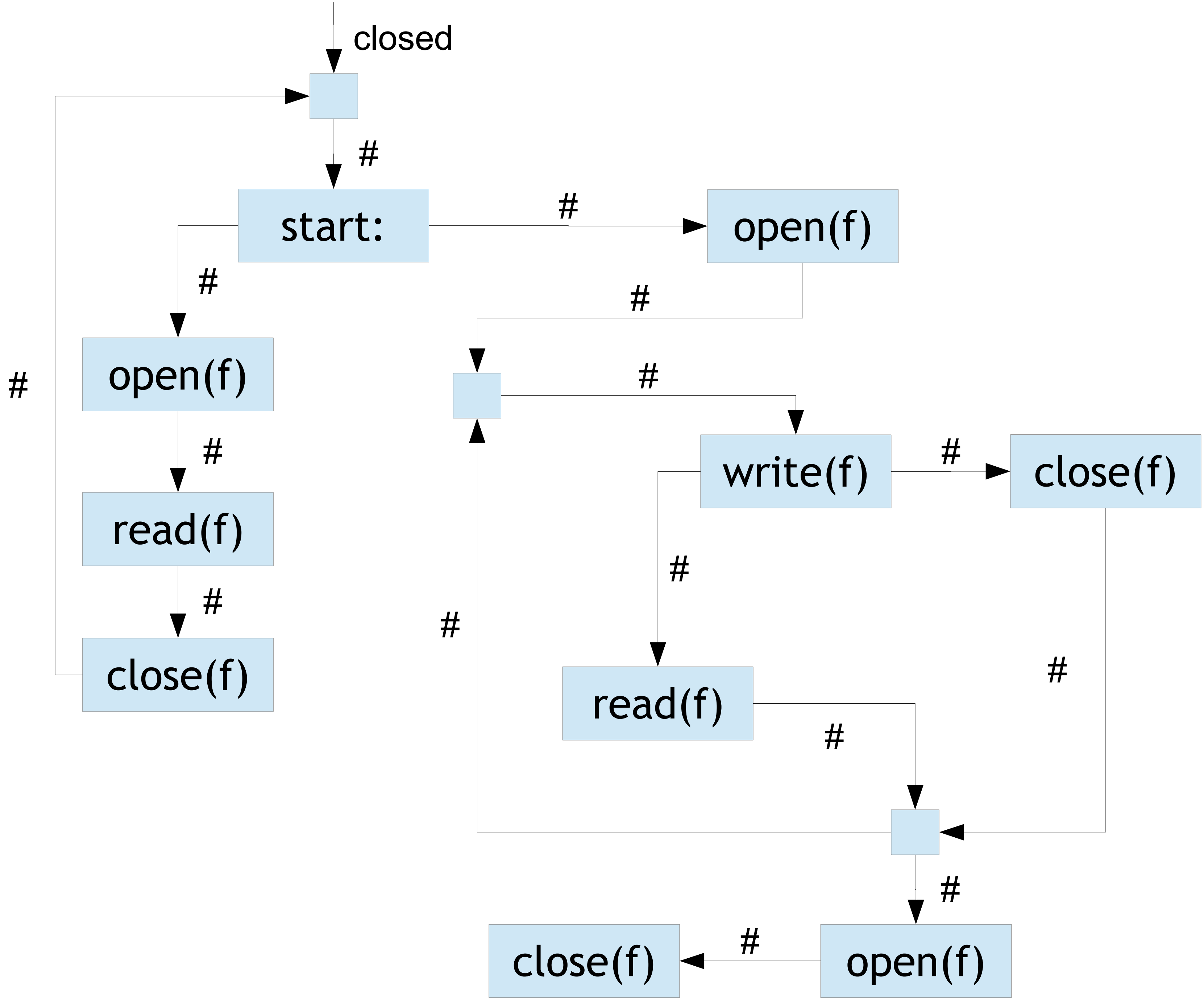
$g = a$

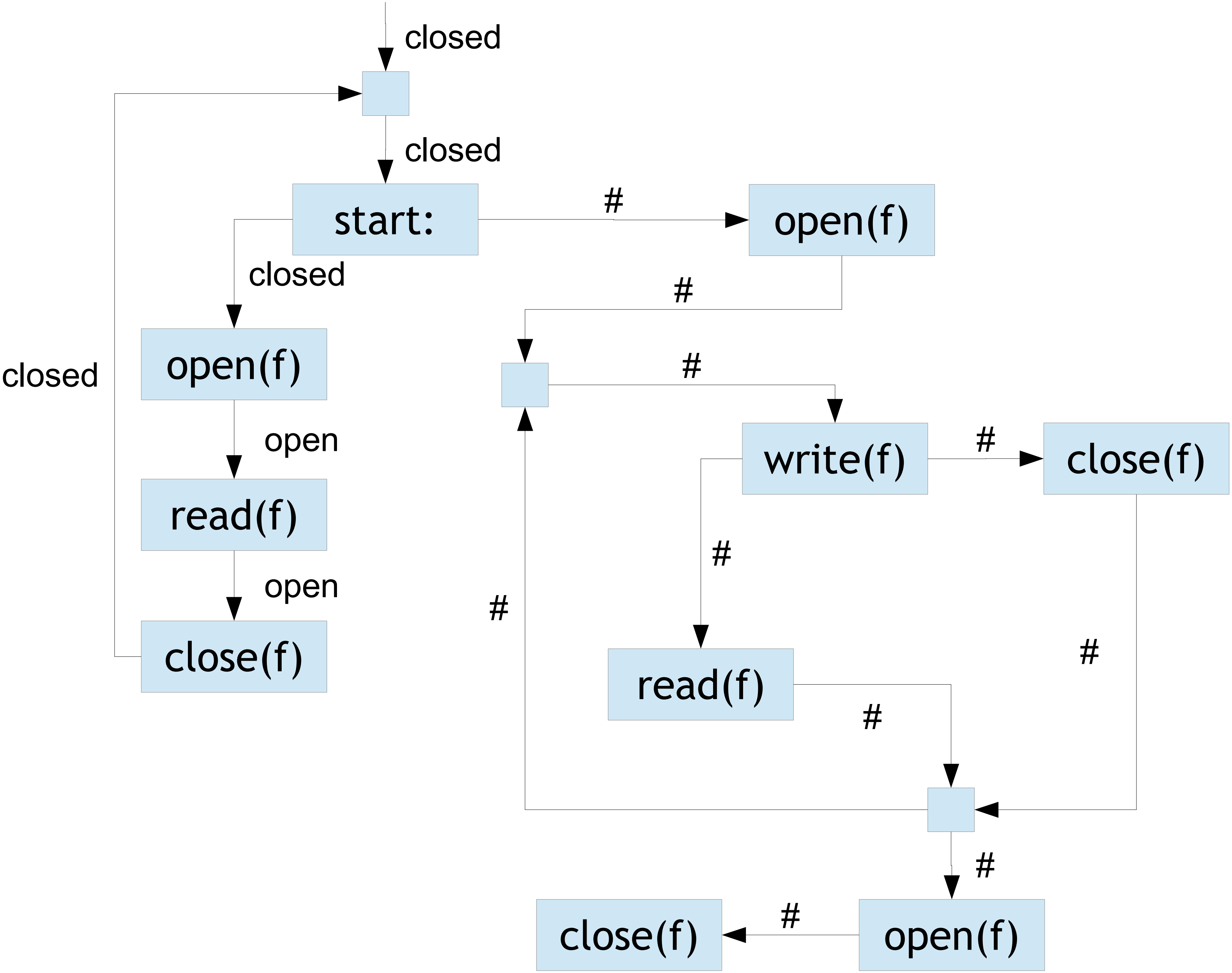
Rules: Multiple Possibilities

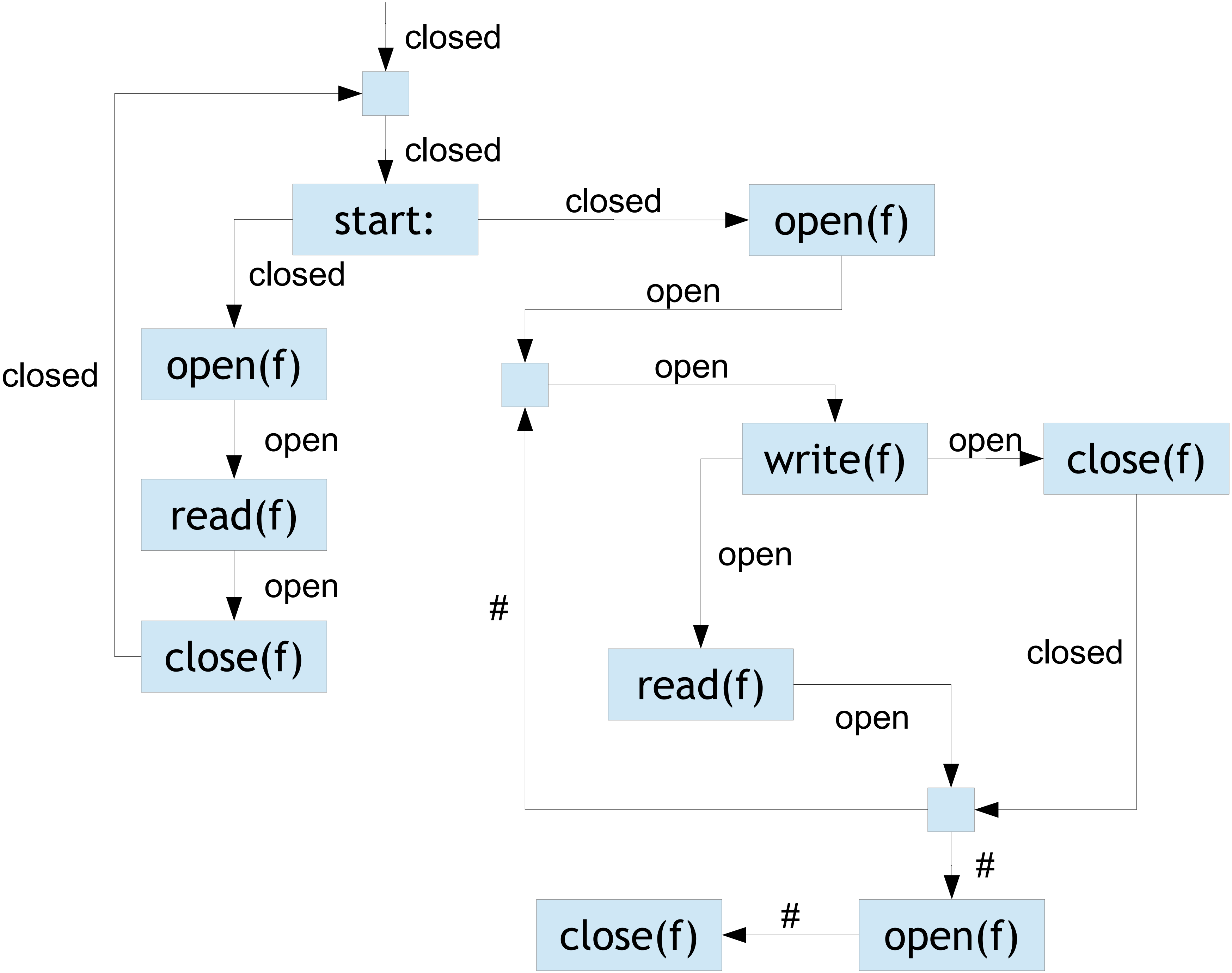


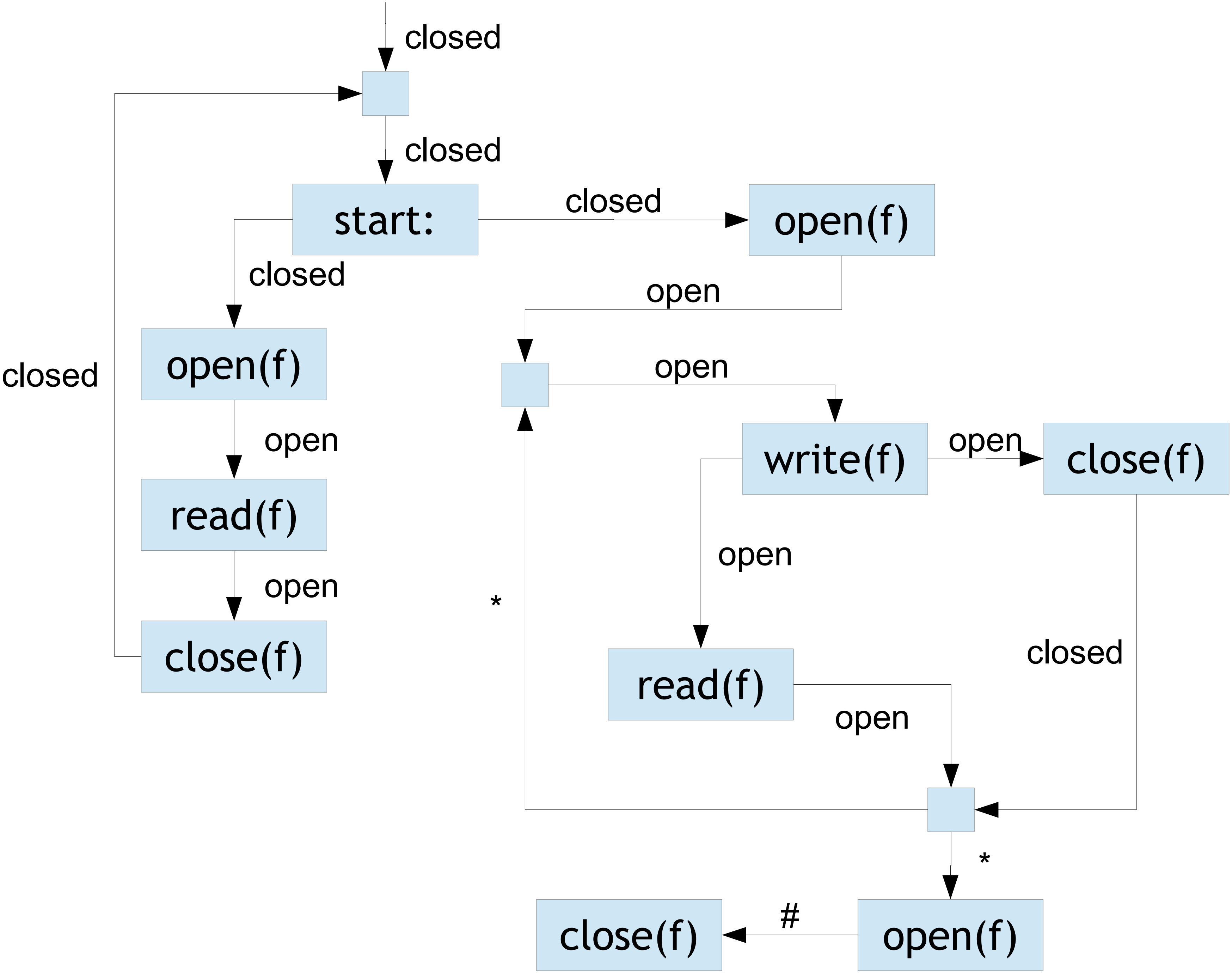
A Tricky Program

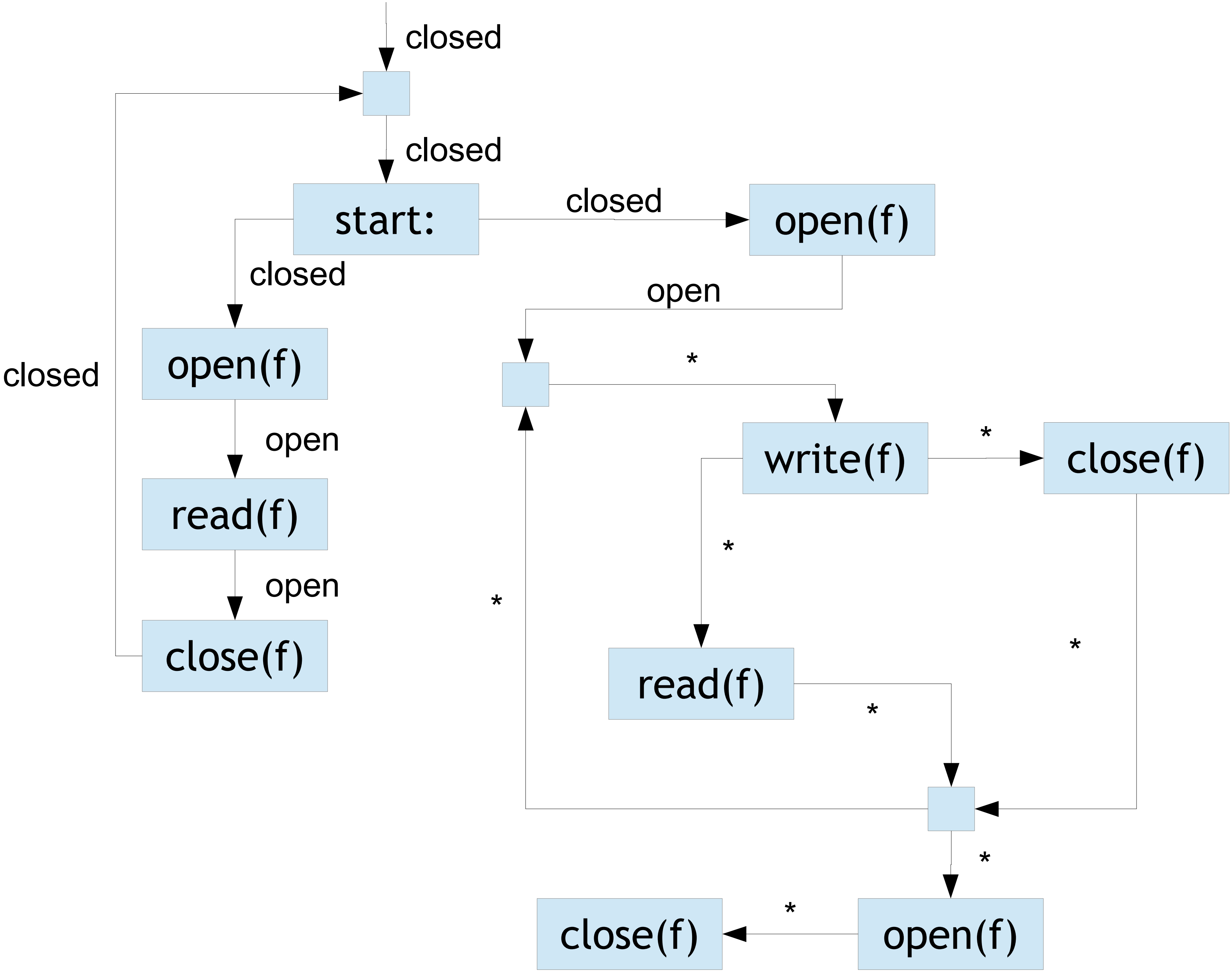
```
start:
switch (a)
  case 1: open(f); read(f); close(f); goto start
  default: open(f);
do {
  write(f) ;
  if (b):   read(f);
  else:     close(f);
} while (b)
open(f);
close(f);
```


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


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


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


closed
closed
start:
closed
open(f)
closed
closed
open(f)
open
open
open
read(f)
write(f)
open
close(f)
open
open
*
close(f)
read(f)
closed
open
*
close(f)
#
open(f)



closed

closed

start:

closed

open(f)

closed

closed

open(f)

open

open

*

read(f)

write(f)

*

close(f)

open

*

*

close(f)

read(f)

*

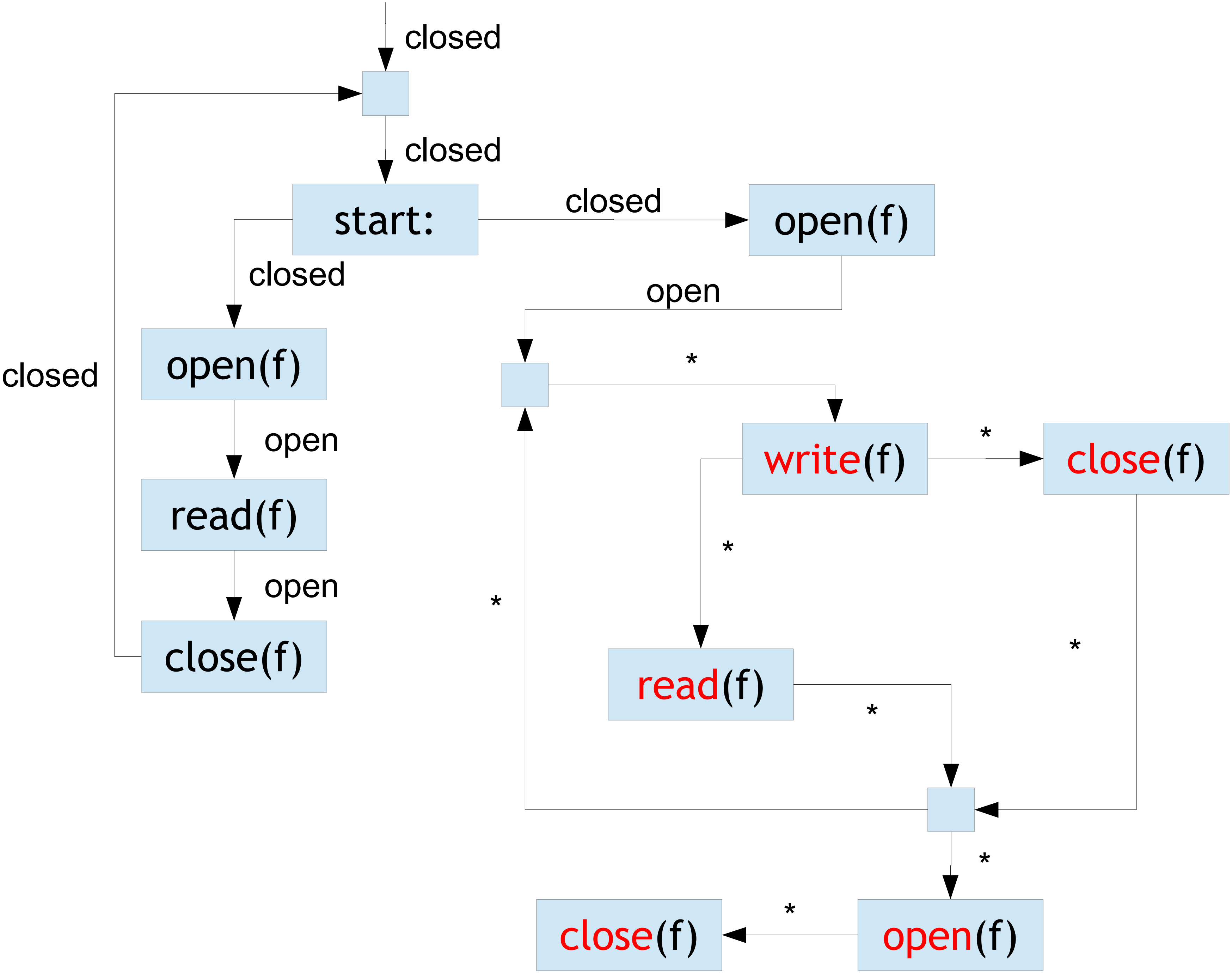
*

*

*

close(f)

open(f)


closed
closed
start:
closed
open(f)
closed
closed
open(f)
open
read(f)
open
close(f)
open
*
write(f)
*
close(f)
*
*
read(f)
*
*
*
open(f)
*
close(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?

- How's the homework going?