Program Analysis, Testing, and Repair

Yu Huang
Vanderbilt University
yu.huang@vanderbilt.edu

Program Analysis

Operate on the programs

- The systematic examination of a program to determine its properties
  - Is my program correct?
  - Where is the bug?
  - What does a program do (without running it)?
  - How to prove theorems about the behavior of a program?
  - ...
- Why should I care?
  - Automatic testing and bug finding
  - Language design and implementations (compilers, VMs)
  - Program transformation (optimization, repair)
  - Program synthesis

Program Analysis

Operate on the programs

- What issues can you find using program analysis?
  - Defects that result from inconsistently following simple design rules
    - Security: Buffer overruns, improperly validated input
    - Memory safety: Null Pointer Dereference, uninitialized data
    - Resource leaks: Memory, OS resources
    - API protocols: Device drivers, GUI frameworks
    - Exceptions: Arithmetic/library/user-defined
    - Encapsulation: Accessing internal data, calling private functions
    - Data races: Two threads access the same data without synchronization

Check compliance to simple, mechanical design rules

Program Analysis

- The systematic examination of a program to determine its properties
- Principle Techniques
  - Static:
    - Inspection: Human evaluation of code, design documents (specifications and models), etc.
    - Analysis: Tools reasoning about the program without executing it.
  - Dynamic:
    - Testing: direct execution of code on test data in a controlled environment.
    - Analysis: Tools extracting data from test runs.

The Bad News: Rice’s Theorem

“Any nontrivial property about the language recognized by a Turing machine is undecidable.”

Henry Gordon Rice, 1953

Soundness and Completeness

- An analysis is "sound" if every claim it makes is true
- An analysis is "complete" if it makes every true claim
- Soundness/Completeness correspond to under/over-approximation depending on context
  - E.g. compilers and verification tools treat "soundness" as over-approximation since they make claims over *all possible inputs*
  - E.g. code quality tools often treat "sound" analyses as under-approximation because they make claims about existence of bugs

Soundness and Completeness Tradeoffs

- Sound + Complete is impossible in general (which theorem again?)
- Most practical tools attempt to be either sound or complete for some specific application, using approximation
- Program analysis is a rich field because of the constant and never-ending battle to balance the trade-offs for accuracy and performance with ever-increasing software complexity

Fundamental Concepts

- Abstraction
  - Elide details of a specific implementation
  - Capture semantically-relevant details; ignore the rest
  - Handle “I don't know”
- Programs As Data
  - Programs are just trees, graphs or strings -> precise program representations!
  - And we know how to analyze and manipulate those (e.g., visit every node in a graph)

Program Analysis

- The systematic examination of a program to determine its properties
- Principle Techniques
  - **Static**:
    - Inspection: Human evaluation of code, design documents (specifications and models), etc.
    - **Analysis: Tools reasoning about the program without executing it.**
  - Dynamic:
    - Testing: direct execution of code on test data in a controlled environment.
    - Analysis: Tools extracting data from test runs.

“Unimportant” SSL Example

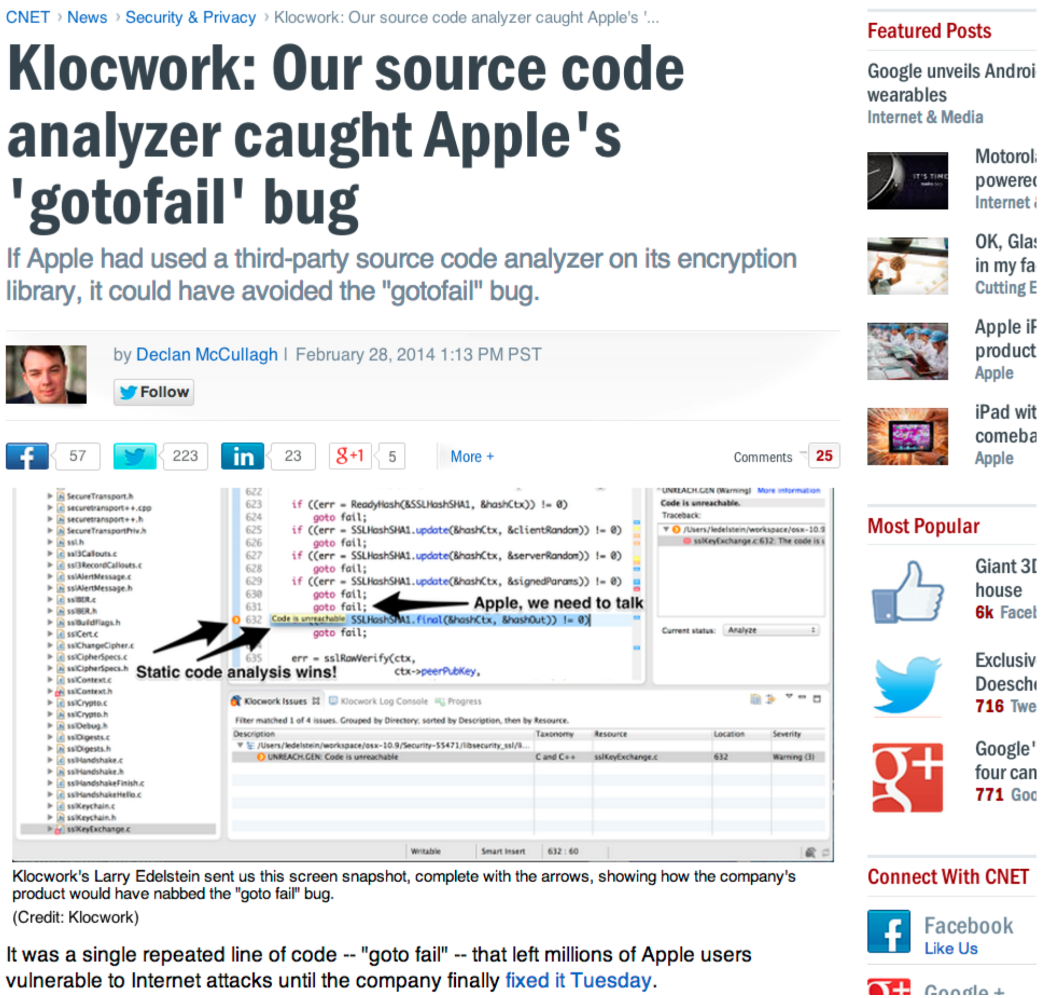
```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature,
                                 UInt16 signatureLen) {
        OSStatus err;
         …
        if ((err = SSLHashSHA1.update(&hashCtx, &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
;
}
```

The Apple goto fail vulnerability: lessons learned

David A. Wheeler
2021-01-16 (original 2014-11-23)

ns that we *should* learn from the Apple "goto fail" vulnerability. It first starts with some background, discusses the mis
ntifying what could have countered this, briefly discusses the Heartbleed countermeasures from my separate paper or

https://dwheeler.com/essays/apple-goto-fail.html



CNET › News › Security & Privacy › Klocwork: Our source code analyzer caught Apple's '...

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

```
623    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
624        goto fail;
625    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
626        goto fail;
627    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
628        goto fail;
629    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
630        goto fail;
631        goto fail;
632    Code is unreachable SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
           goto fail;

635    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

Klocwork Issues | Klocwork Log Console | Progress

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)

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, Glas in my fa
Cutting E

Apple iF product
Apple

iPad wit comeba
Apple

Most Popular

Giant 3D house
6k Facel

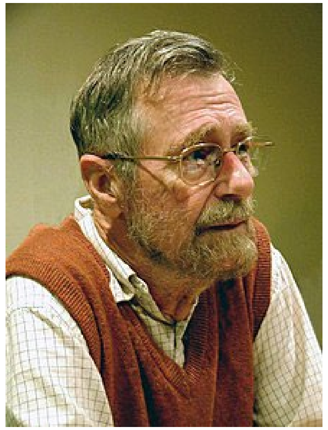
Exclusiv Doesch
716 Twe

Google' four can
771 Goo

Connect With CNET

Facebook
Like Us

Google +



"GOTO Statement Considered Harmful"
-- Edsger Dijkstra

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: 15+ millions LOC
- Some defects are very difficult to find via testing or manual inspection

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!

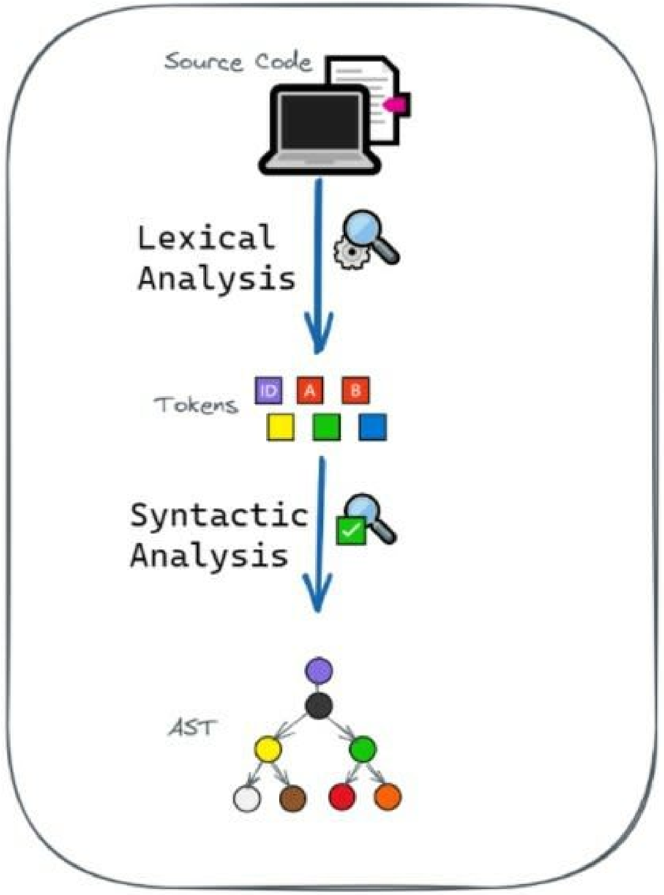
Static Analyses Often Focus On

- 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

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”

Abstraction: Abstract Syntax Tree

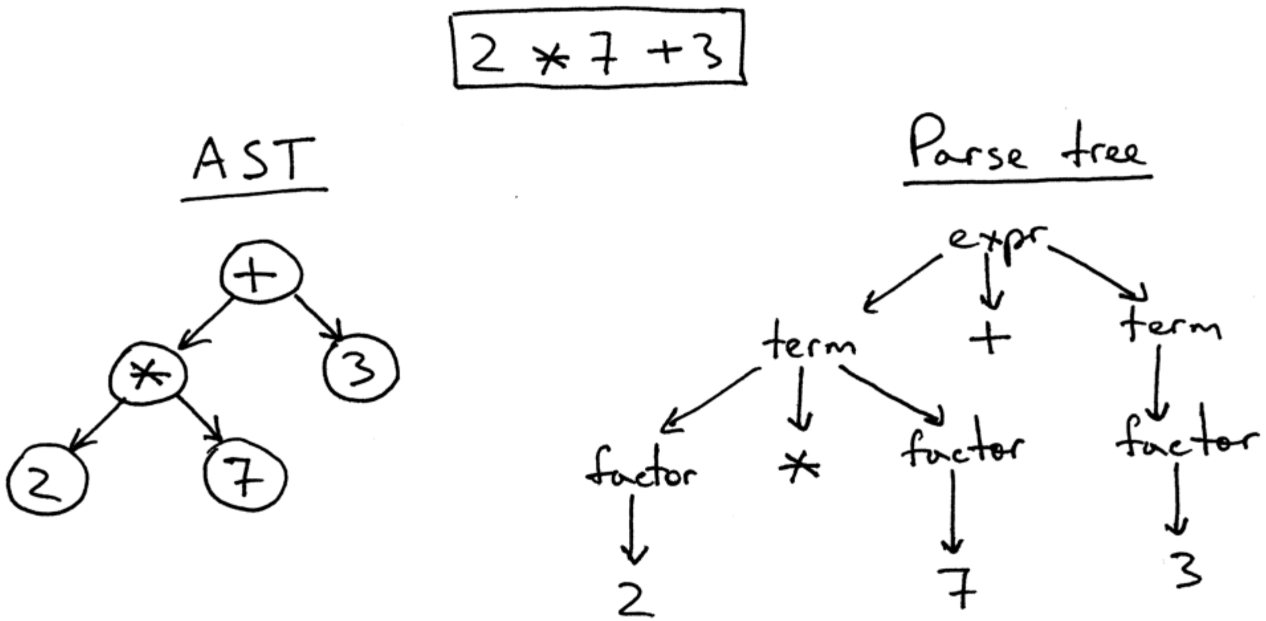


Steps in the processing of source code (Image by author)

https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38

Example of AST

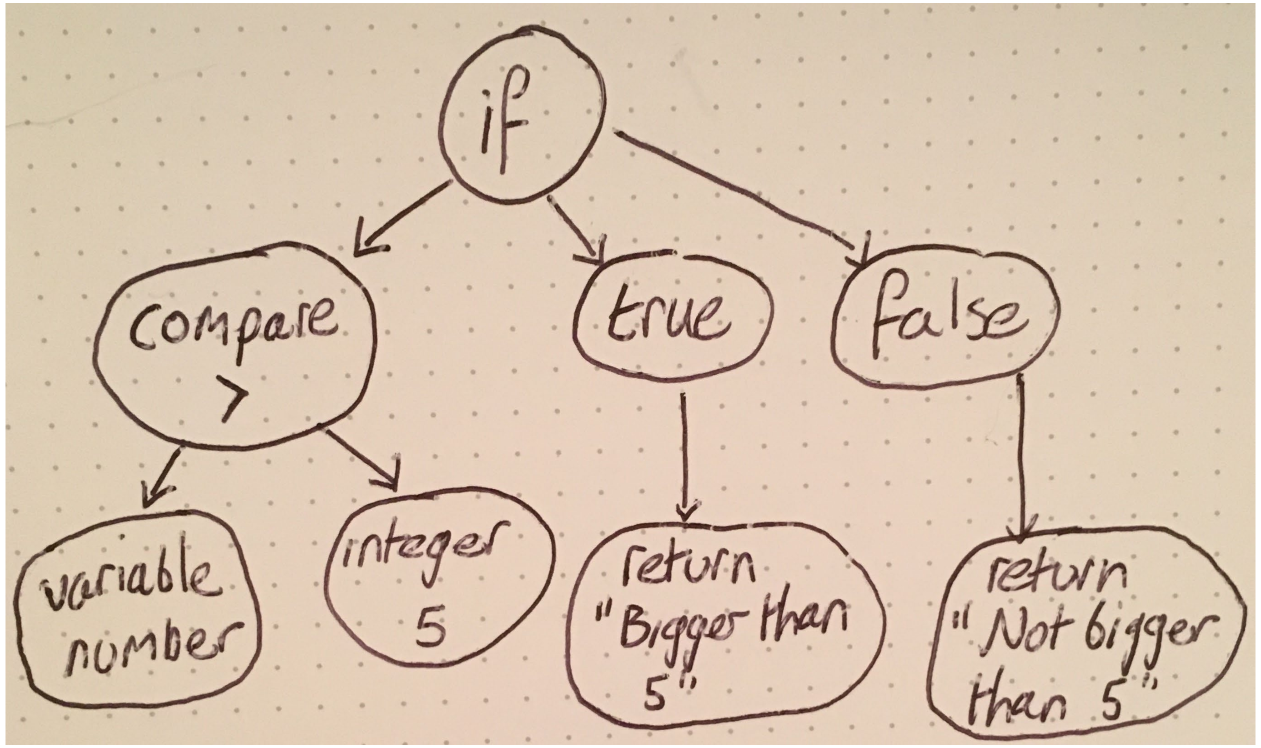
- https://astexplorer.net/
- For this course, the intuition is fine: "It is a tree representing a program" → You can walk through it!
- (Take Compilers if you want to learn how to parse for real.)



Example of AST

- https://astexplorer.net/
- For this course, the intuition is fine! “It is a tree representing a program” → You can walk through it!!!
- (Take Compilers if you want to learn how to parse for real.)

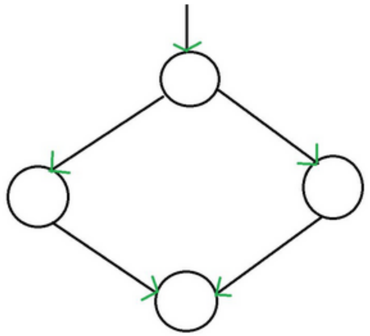
```
 if number > 5
   return 'Bigger than 5'
else
   return 'Not bigger than 5'
```



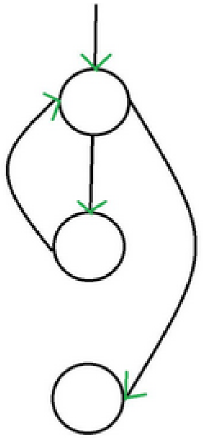
Abstraction: Control Flow Graph

- An CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution
  - Each node in the graph represents a basic block (i.e., a straight-line piece of code without any jumps)
  - Directed edges represents jumps

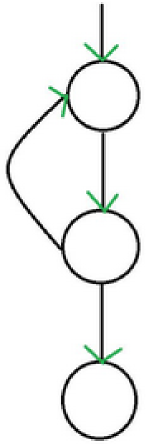
Example of CFG



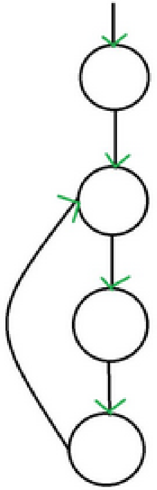
If-then-else



while



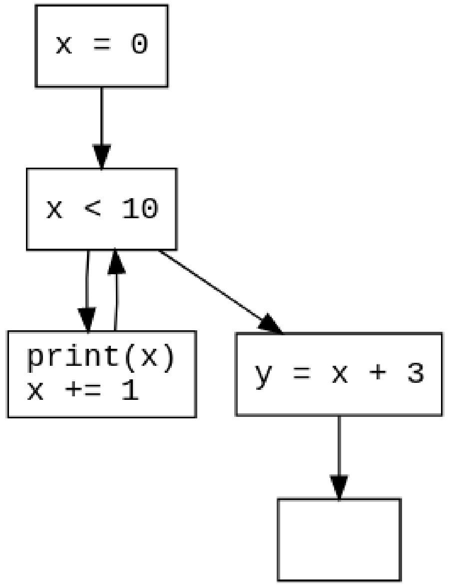
do-while



for

```python
x = 0
while x < 10:
    print(x)
    x += 1

y = x + 3
```



Static Analysis: 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

One Exemplar Analysis

- *Definite* Null Dereference
  - “Whenever execution reaches *ptr at program location L, ptr will be NULL”

One Exemplar Analysis

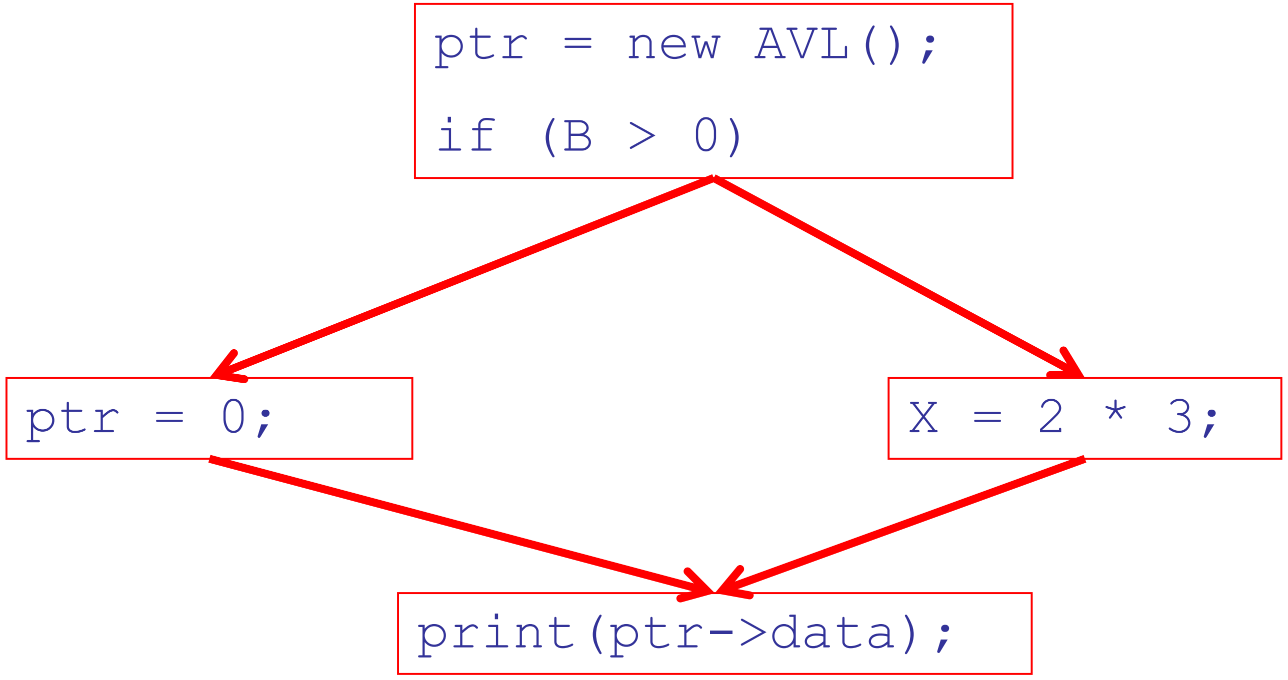
- *Definite* Null Dereference
  - "Whenever execution reaches *ptr at program location L, ptr will be NULL"
- *Potential* Secure Information Leak
  - "We read in a secret string at location L, but there is a possible future public use of it"

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 }

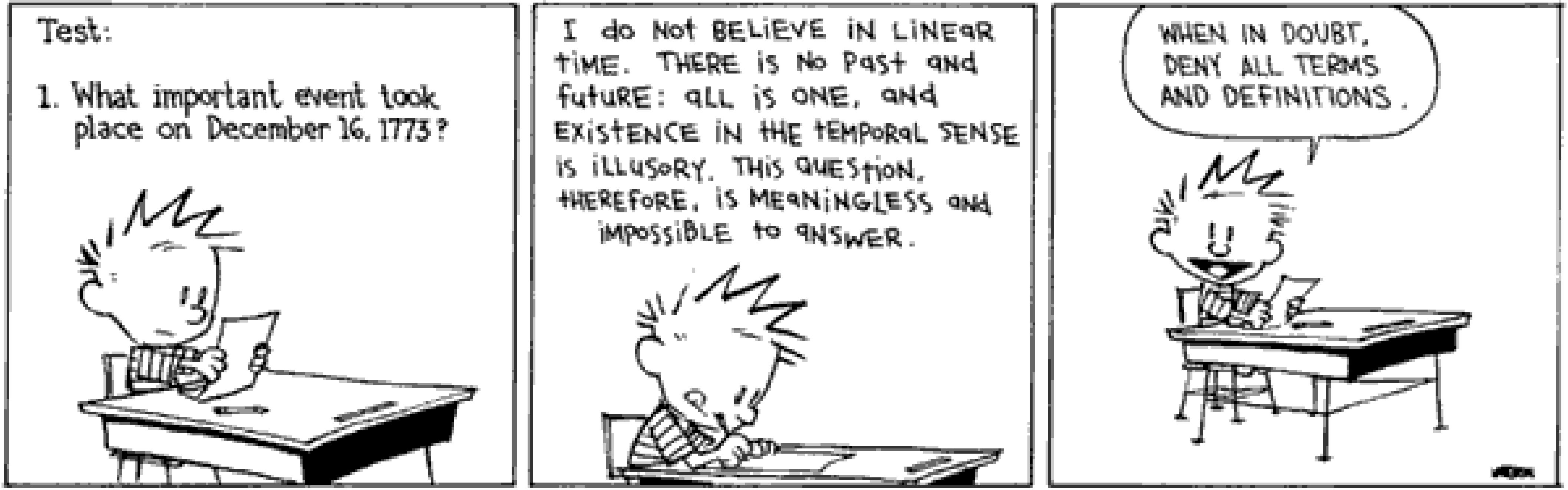
Data Flow Analysis Example: Null Ptr Dereference

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



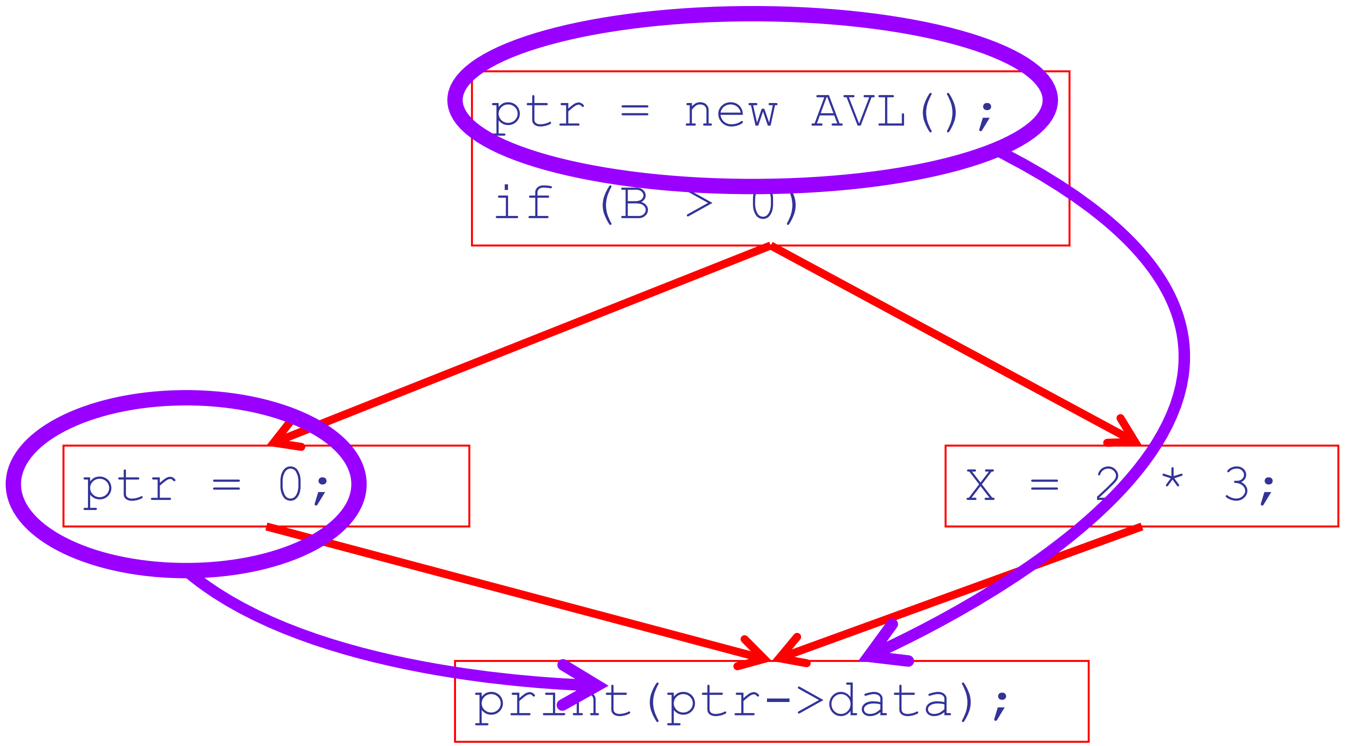
Correctness

- 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 **



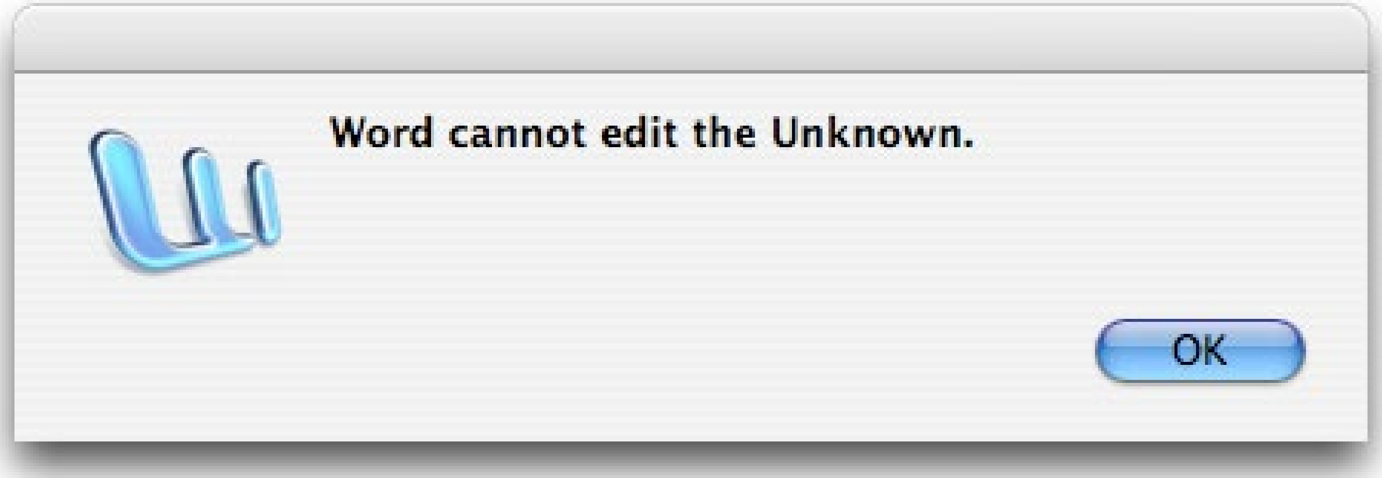
Analysis Example Revisited

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



Static Dataflow Analysis

- Static dataflow analyses share several traits:
  - 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

- 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

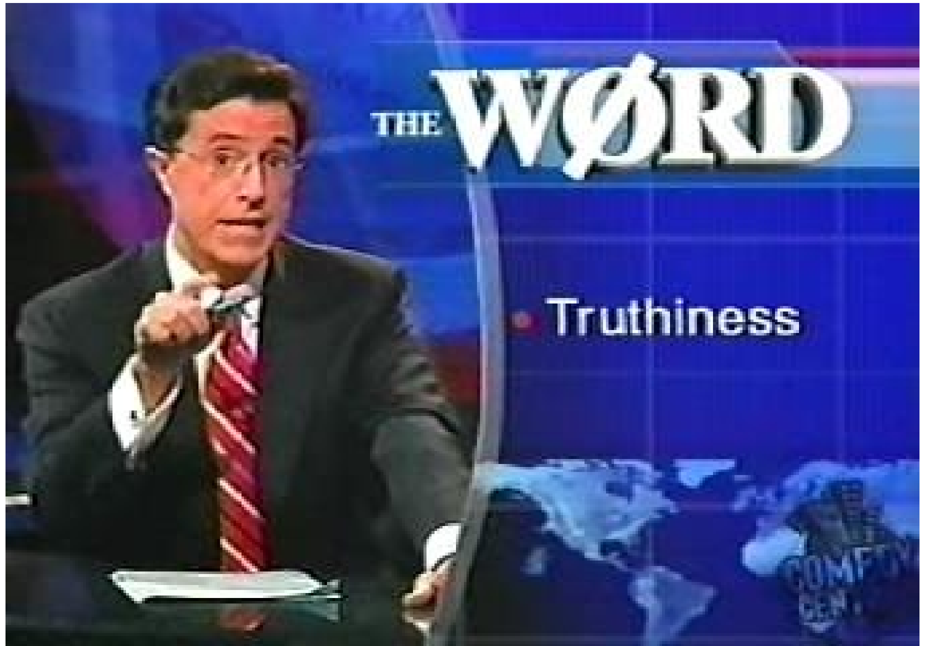
- Almost every important program has a loop
  - Often based on user input
- An algorithm always terminates
- So a dataflow analysis algorithm must terminate even if the input program loops
- This is one source of imprecision
  - Suppose you dereference the null pointer on the 500th iteration but we only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

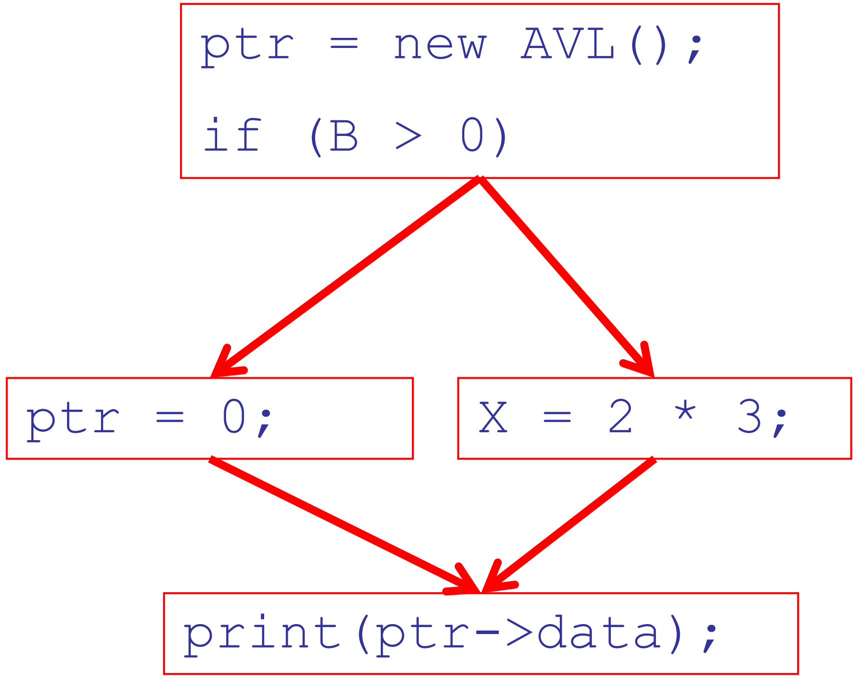


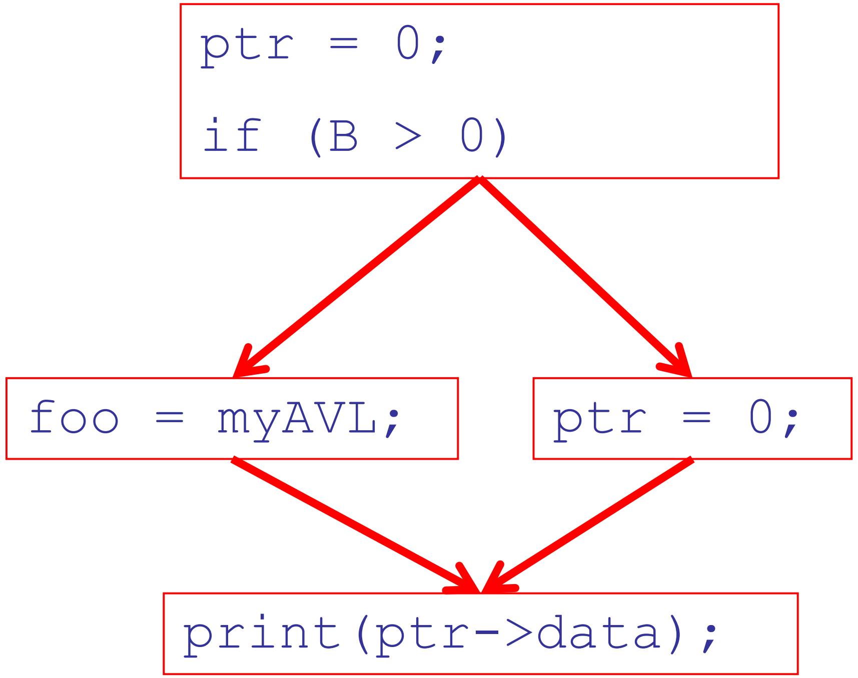
Conservative Program Analyses

- 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

Definitely Null Analysis

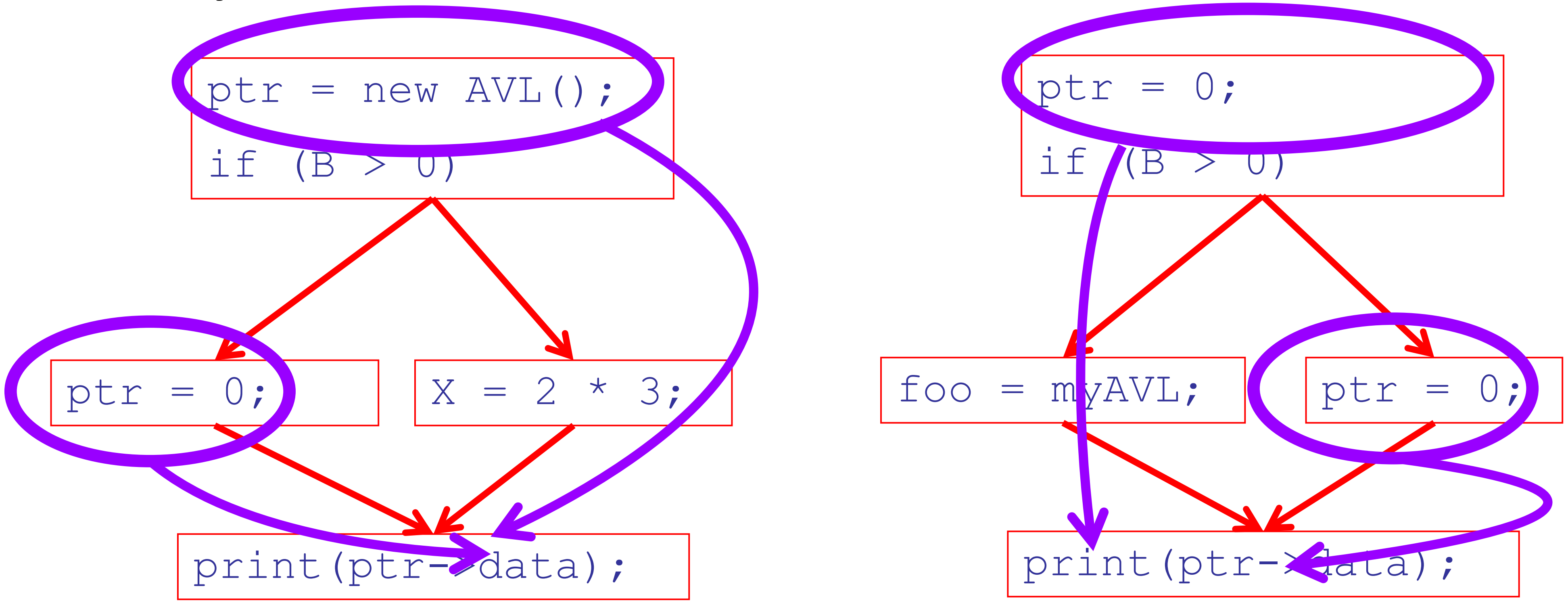
- Is **ptr** *always* null when it is dereferenced?





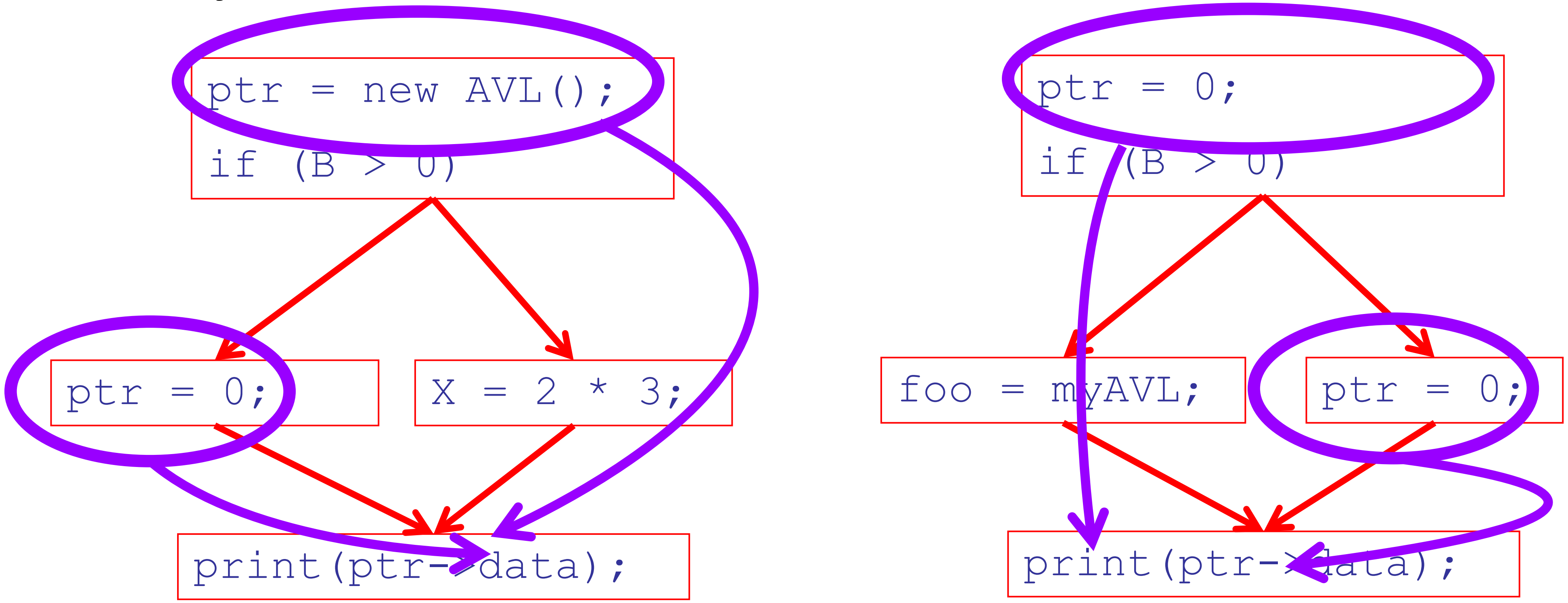
Definitely Null Analysis

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



Definitely Null Analysis

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

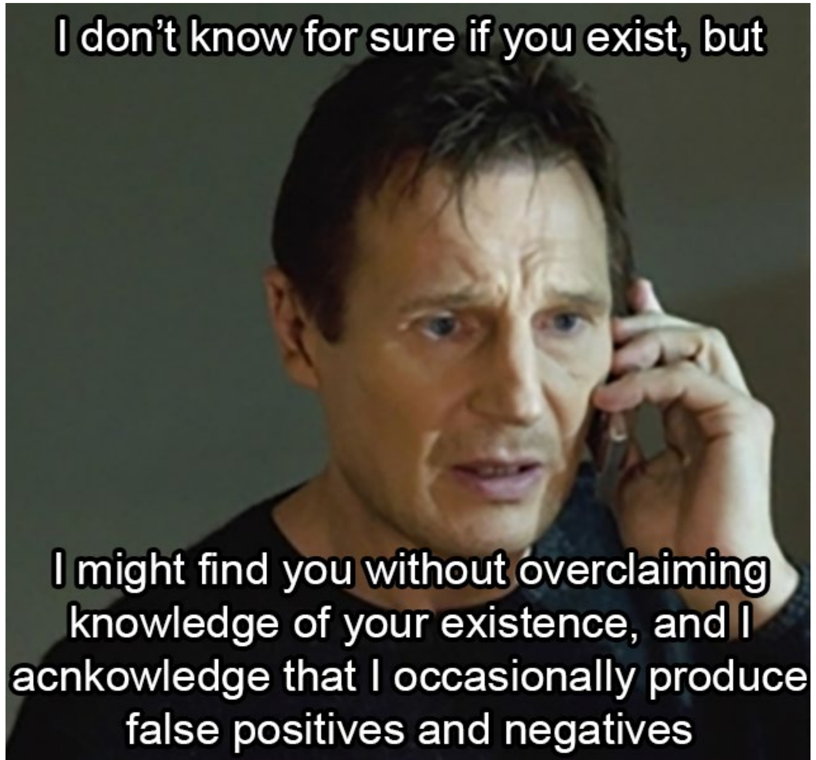


No, not always. **Yes, always.**

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)*



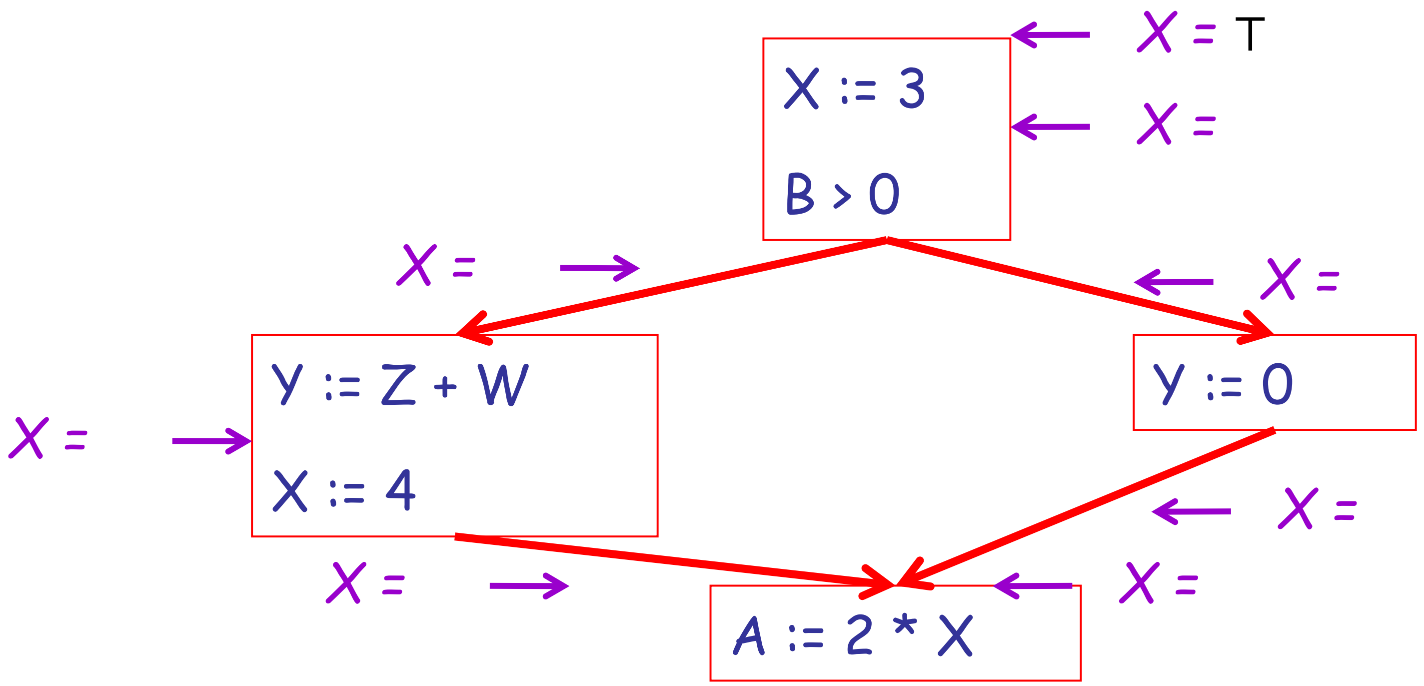
Definitely Null Analysis (Cont.)

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

value	*interpretation*
$\perp$ **(called *Bottom*)**	This statement is not reachable
c	X = constant c
T **(called *Top*)**	Don't know if X is a constant

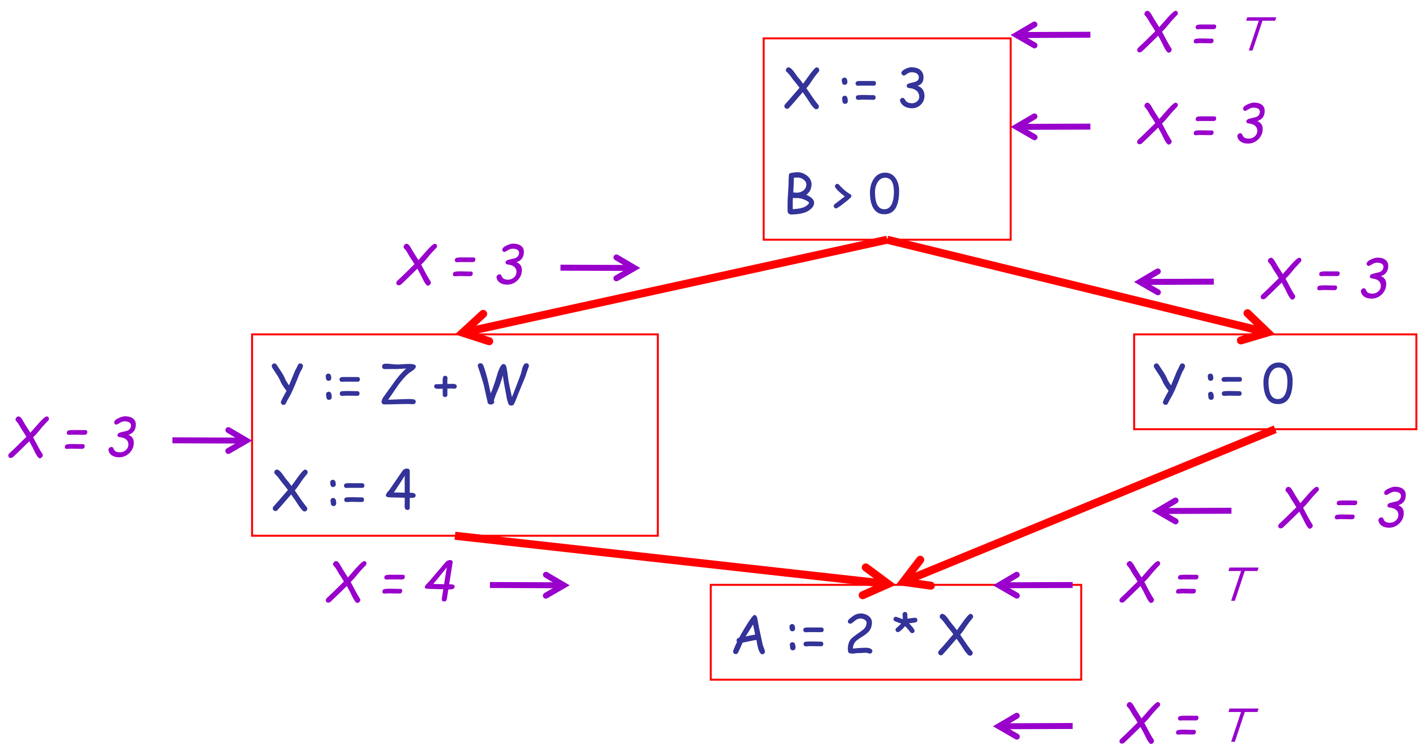
Example

Let's fill in these blanks now.



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

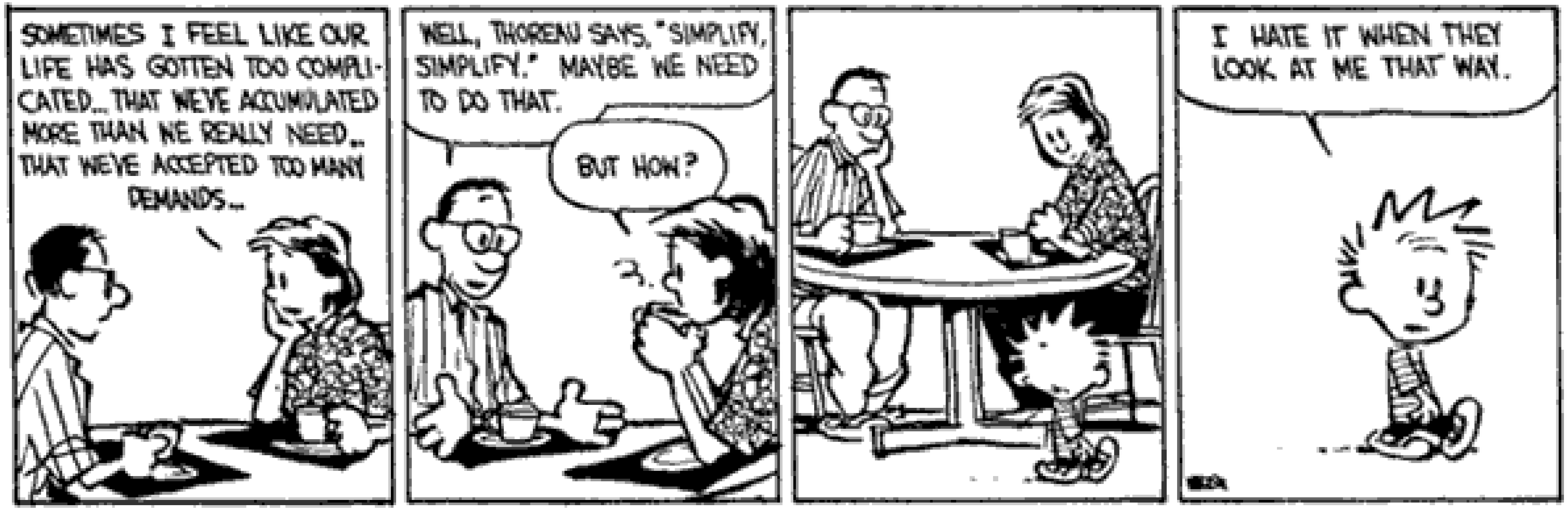
Example Answers



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

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

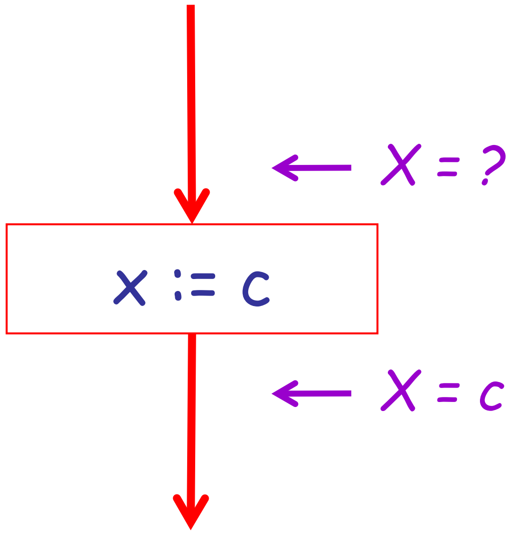
- The idea is to "push" or "**transfer**" information from one statement to the next

- 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:

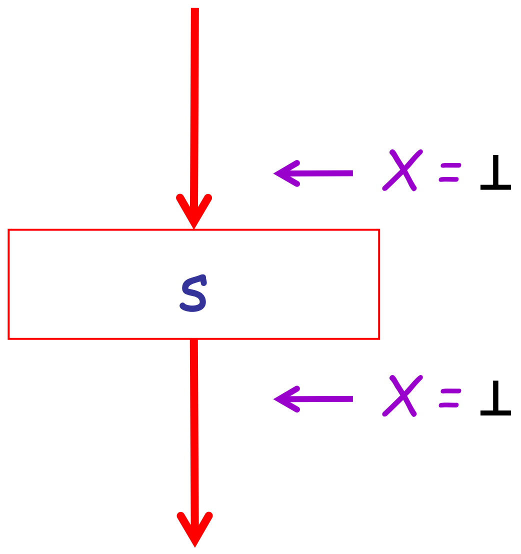
- Define a **transfer function** that transfers information from one statement to another

Rule 1



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

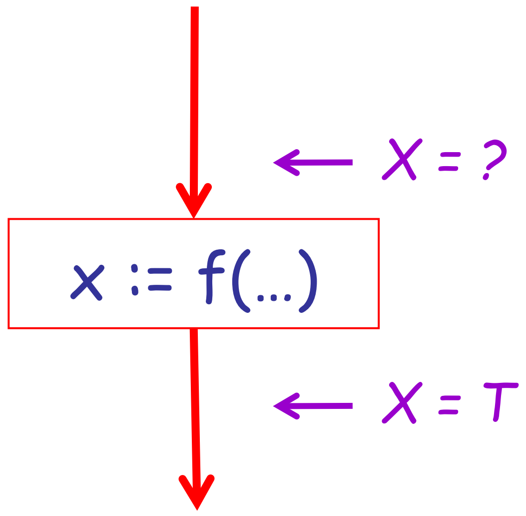
Rule 2



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

Recall: ⊥ = “unreachable code”

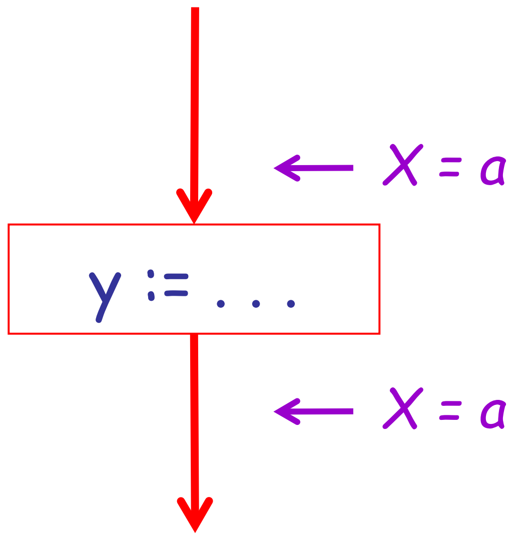
Rule 3



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

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

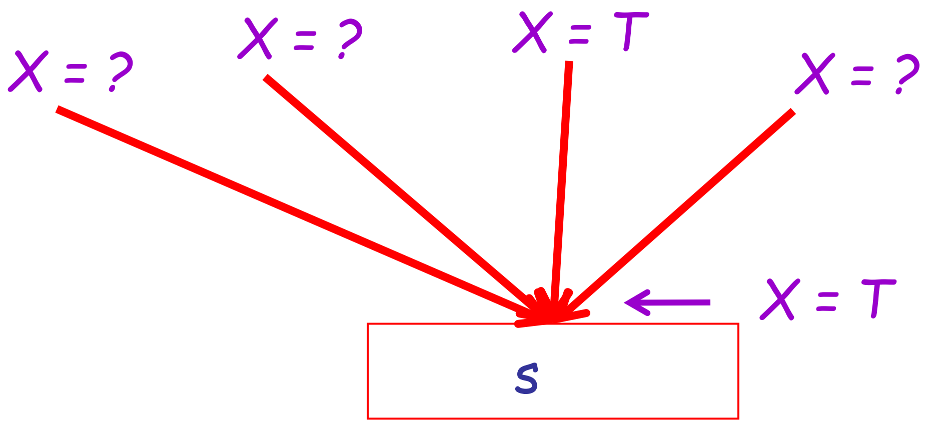


- $C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ if $x \neq y$

The Other Half

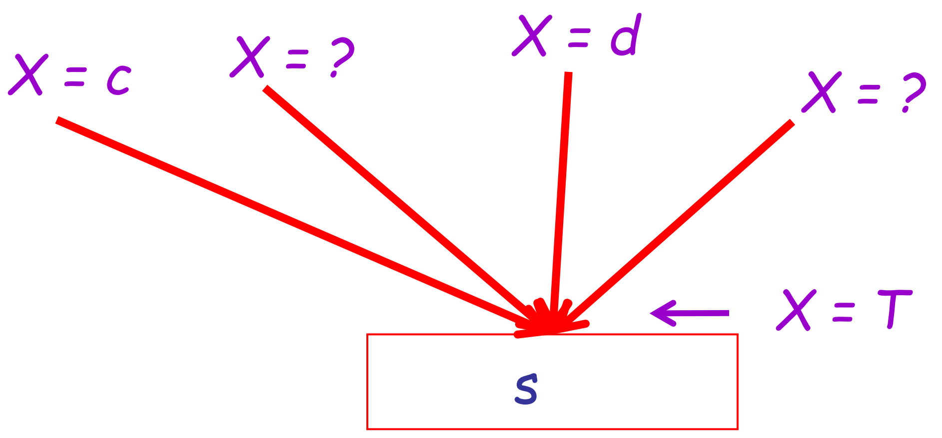
- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information across statements
- Now we need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information forward along paths
- In the following rules, let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

Rule 5



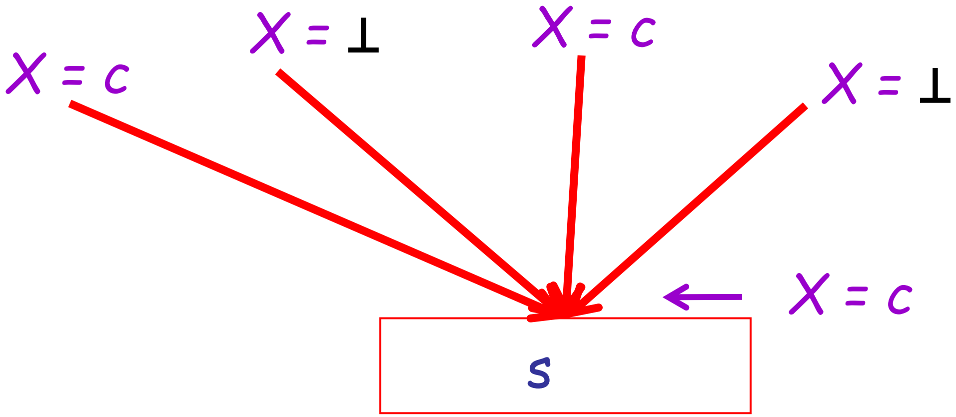
- if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

Rule 6



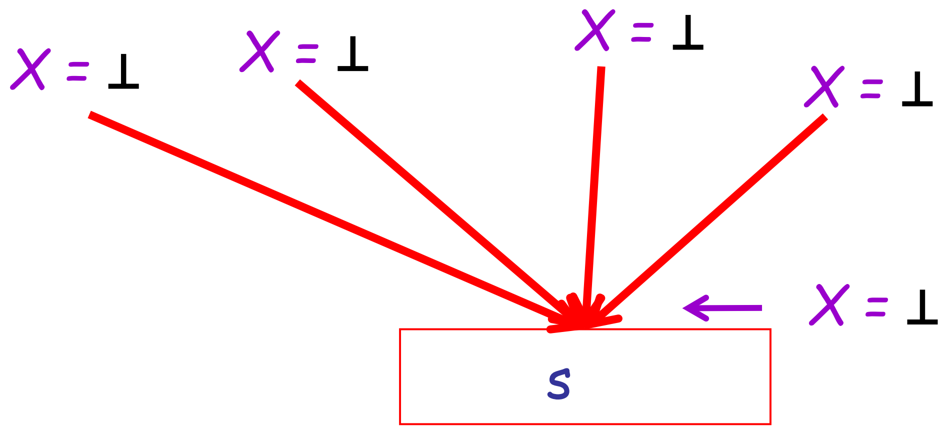
if $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$, then $C_{in}(x, s) = T$

Rule 7



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

Rule 8



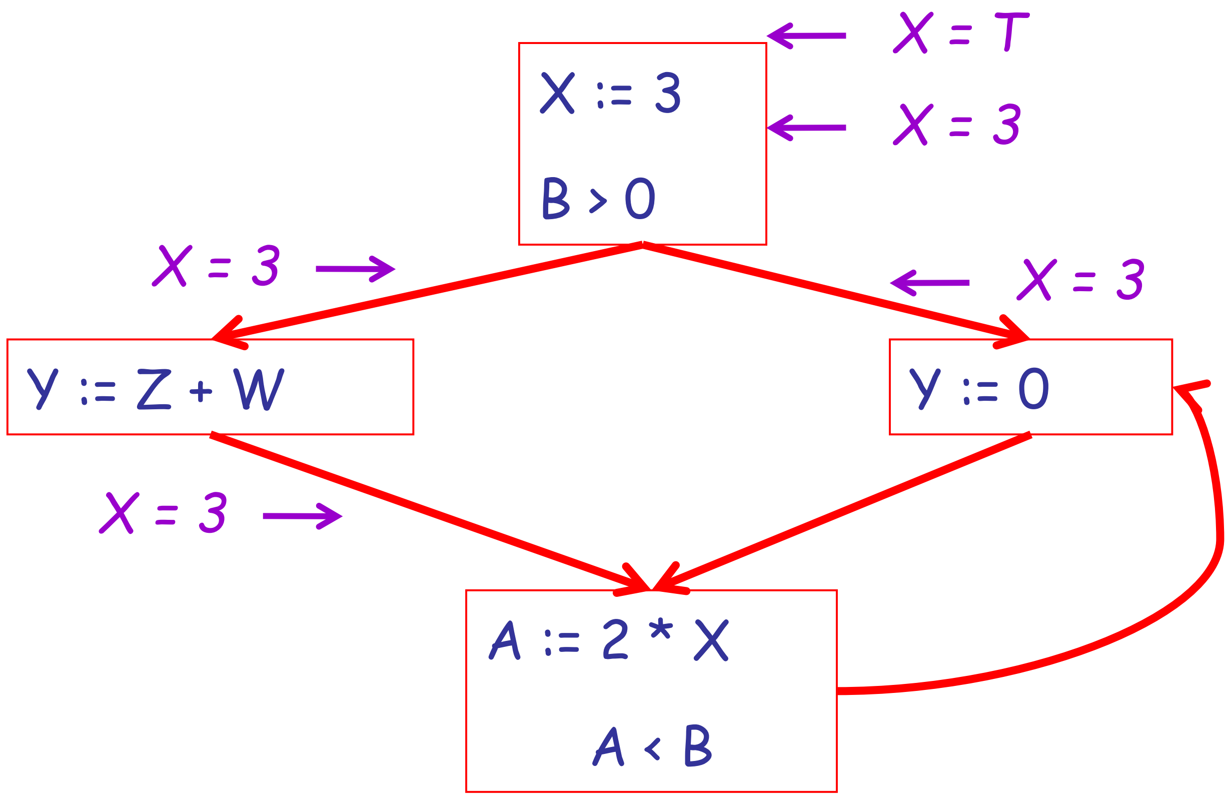
if $C_{out}(x, p_i) = \perp$ for all i, then $C_{in}(x, s) = \perp$

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

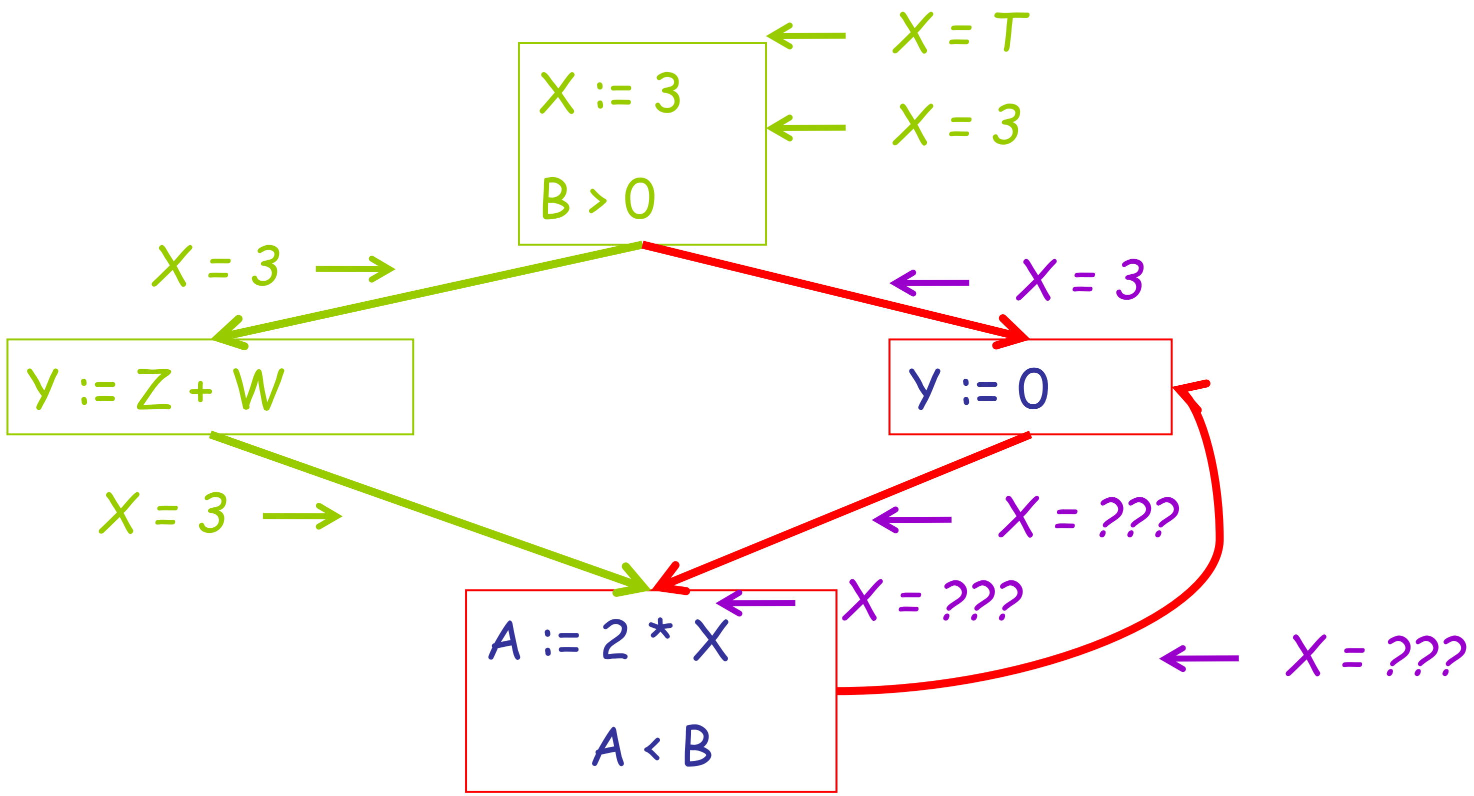
The Value ⊥

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



The Value ⊥

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



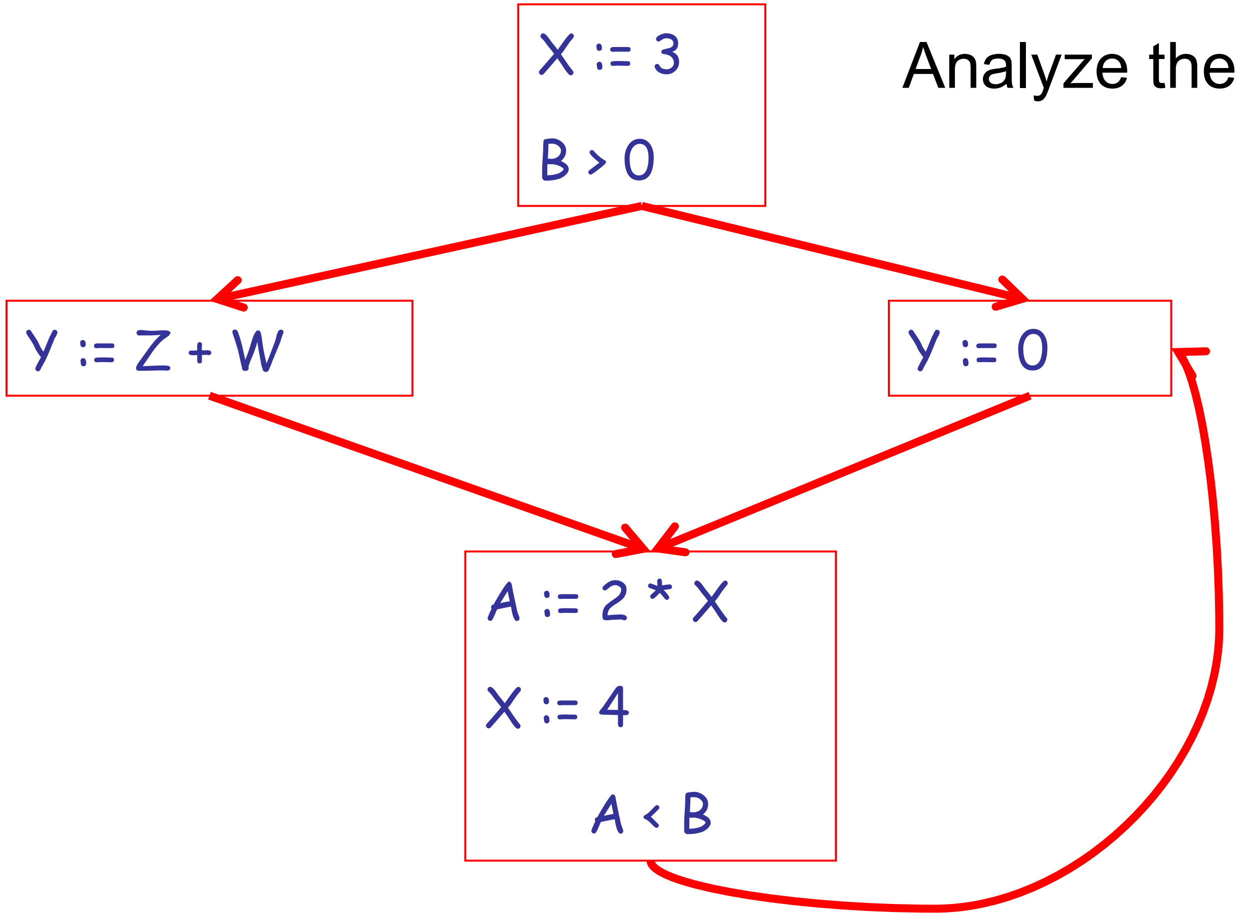
The Value ⊥ (Cont.)

- 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

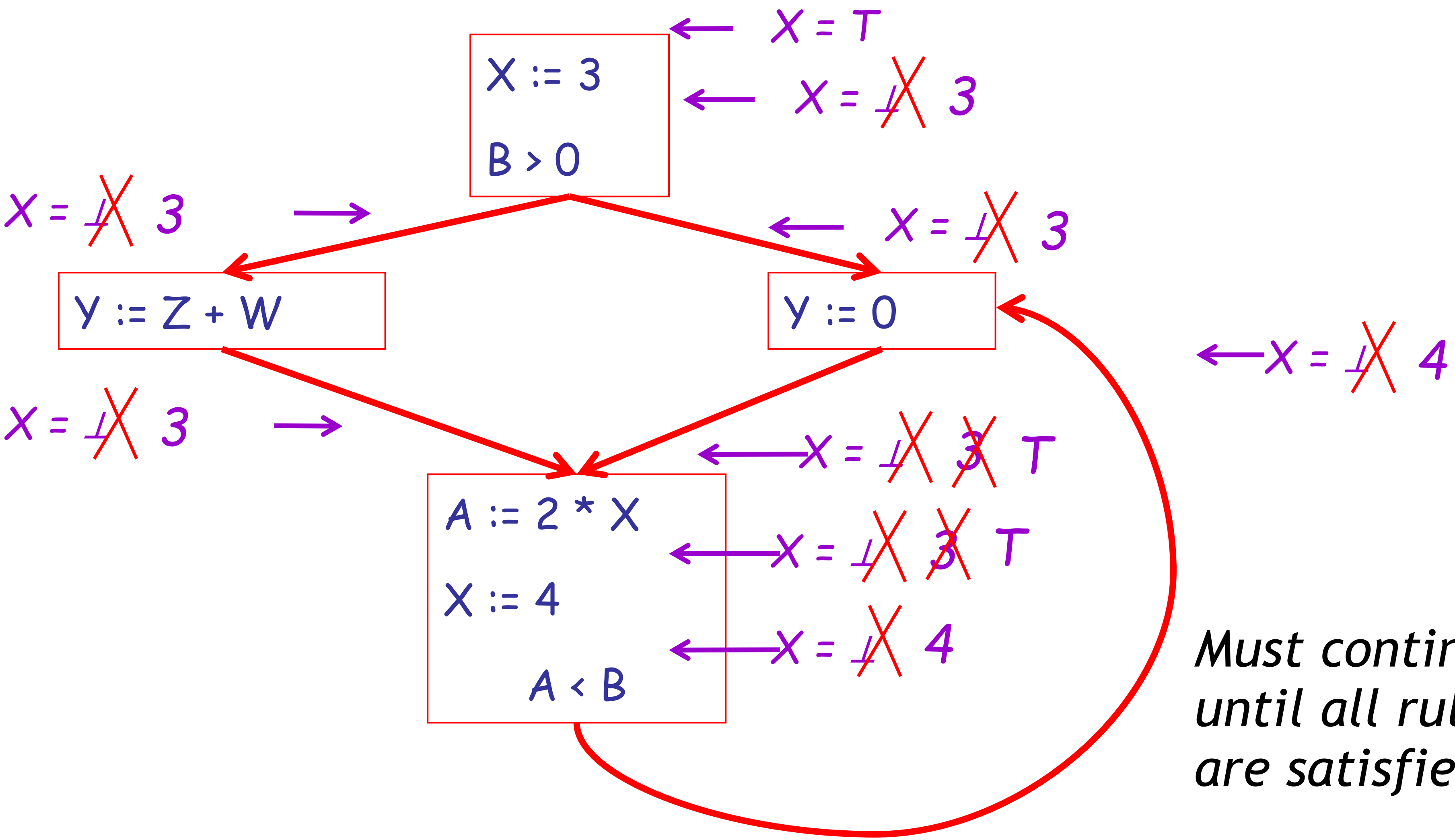
- The initial value ⊥ means "we have not yet analyzed control reaching this point"

Another Example



Analyze the value of X …

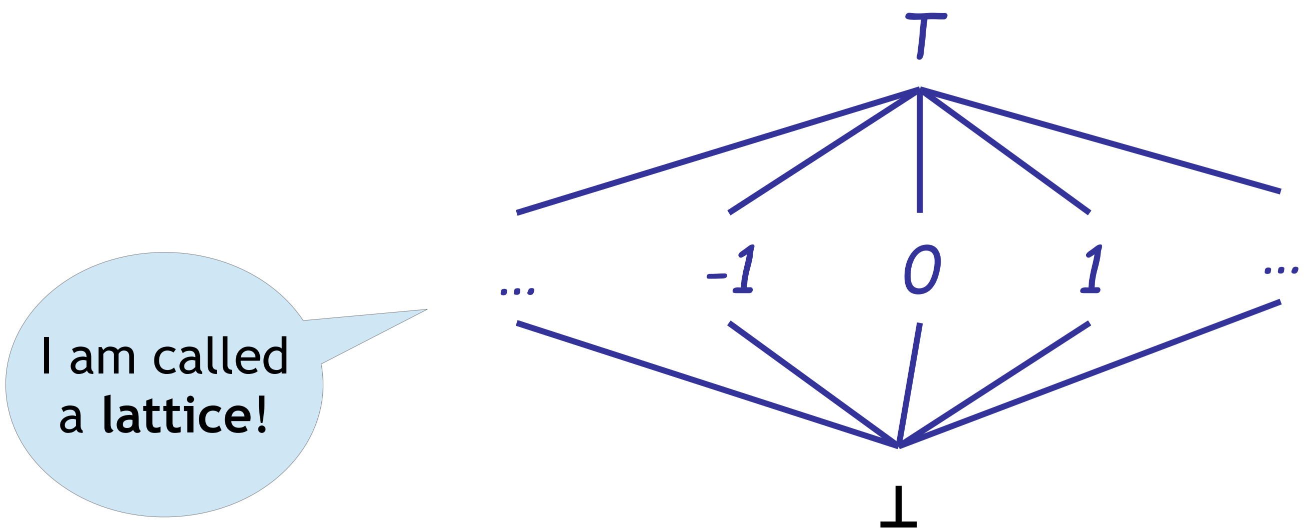
Another Example: Answer



Must continue until all rules are satisfied !

Orderings

- We can simplify the presentation of the analysis by ordering the values
  - ⊥ < c < T
- Making a picture with “lower” values drawn lower, we get



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 ⊥ and only *increase*

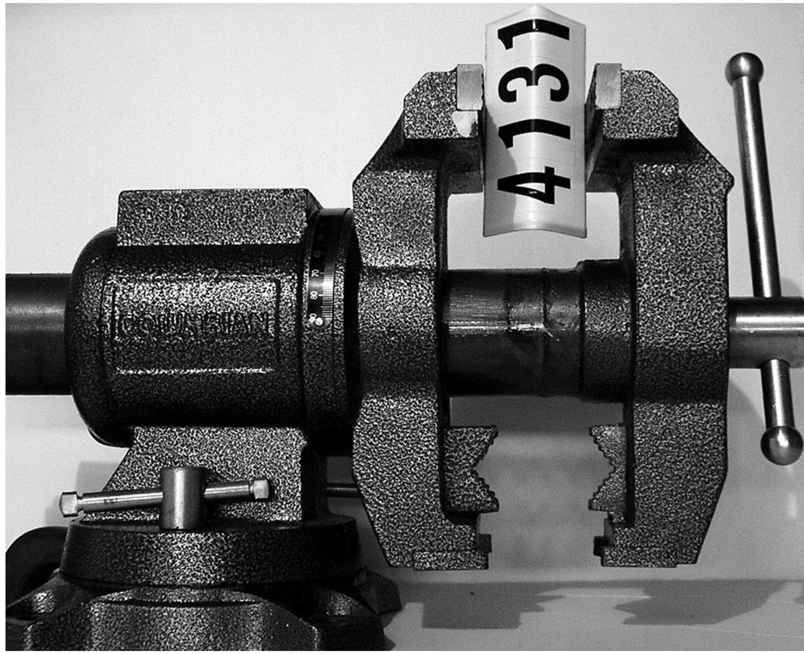
⊥ can change to a constant, and a constant to T
  - Thus, C_(x, s) can change at most twice

Number Crunching

- The algorithm is polynomial in program size:
- **Number of steps =**

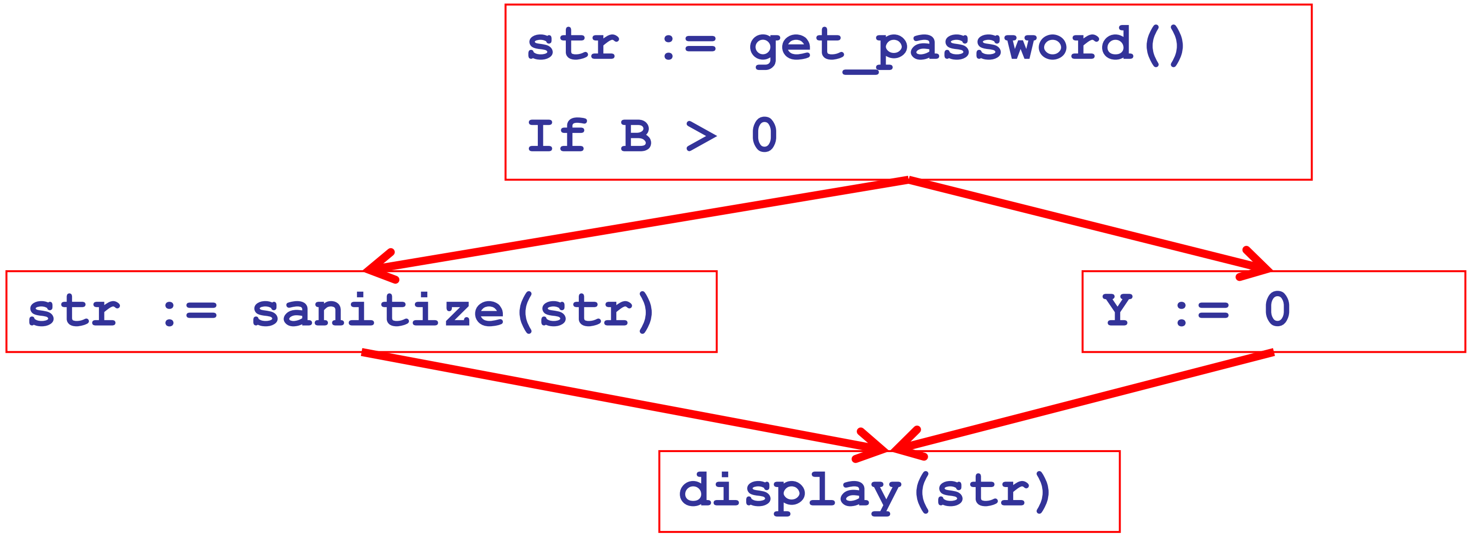
Number of C_(….) values changed * 2 =

(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)

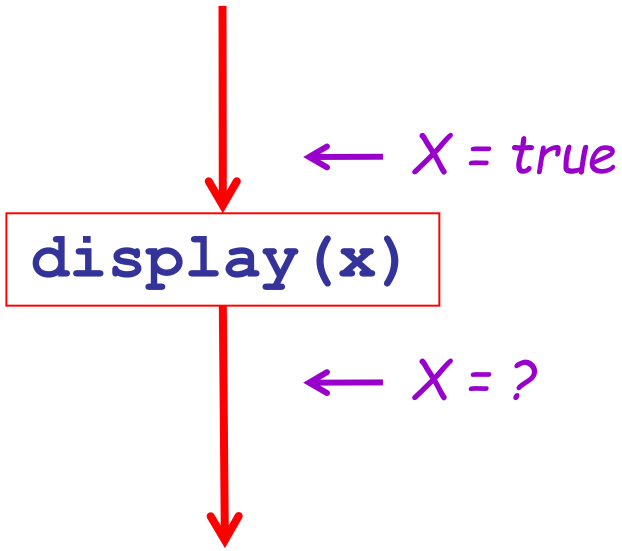
Sensitive Information

- 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

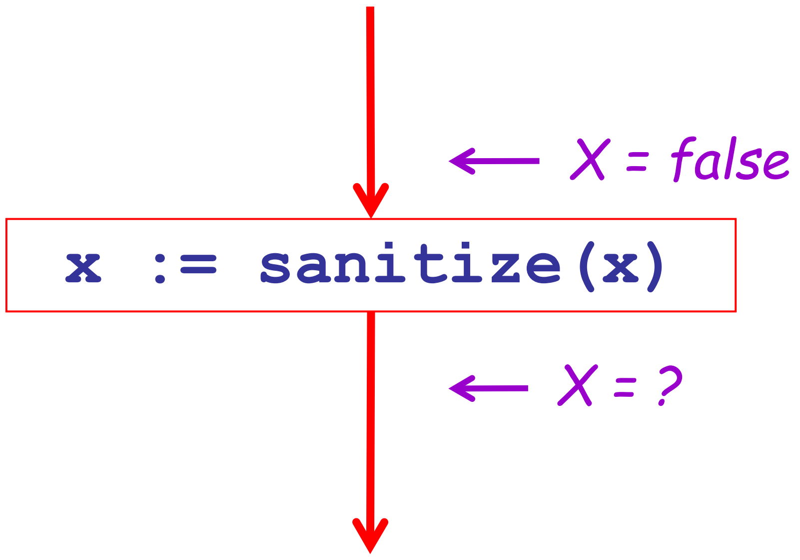


$H_{in}(x, s) = true$ if s displays x publicly

true means “if this ends up being a secret variable

then we have a bug!”

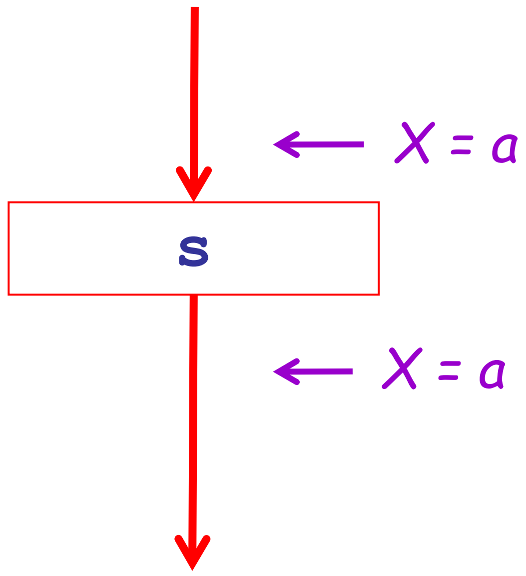
Secure Information Flow Rule 2



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

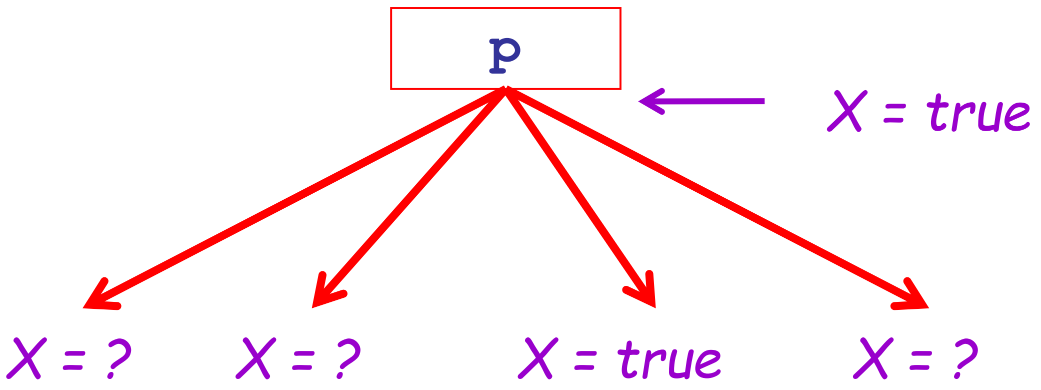
(any subsequent use is safe)

Secure Information Flow Rule 3



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

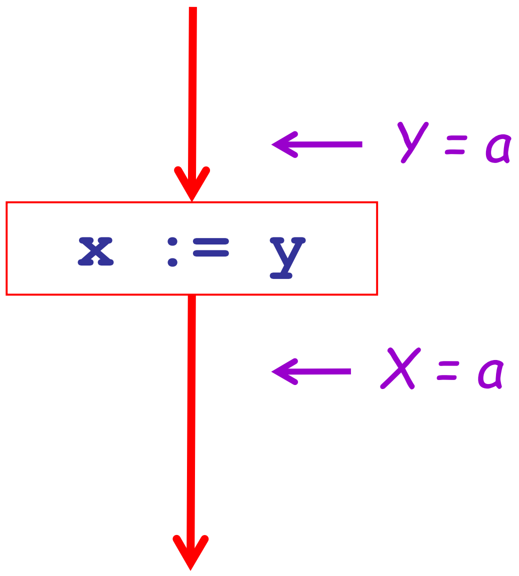
Secure Information Flow Rule 4



- $H_{out}(x, p) = \lor \{ H_{in}(x, s) \mid s$ 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

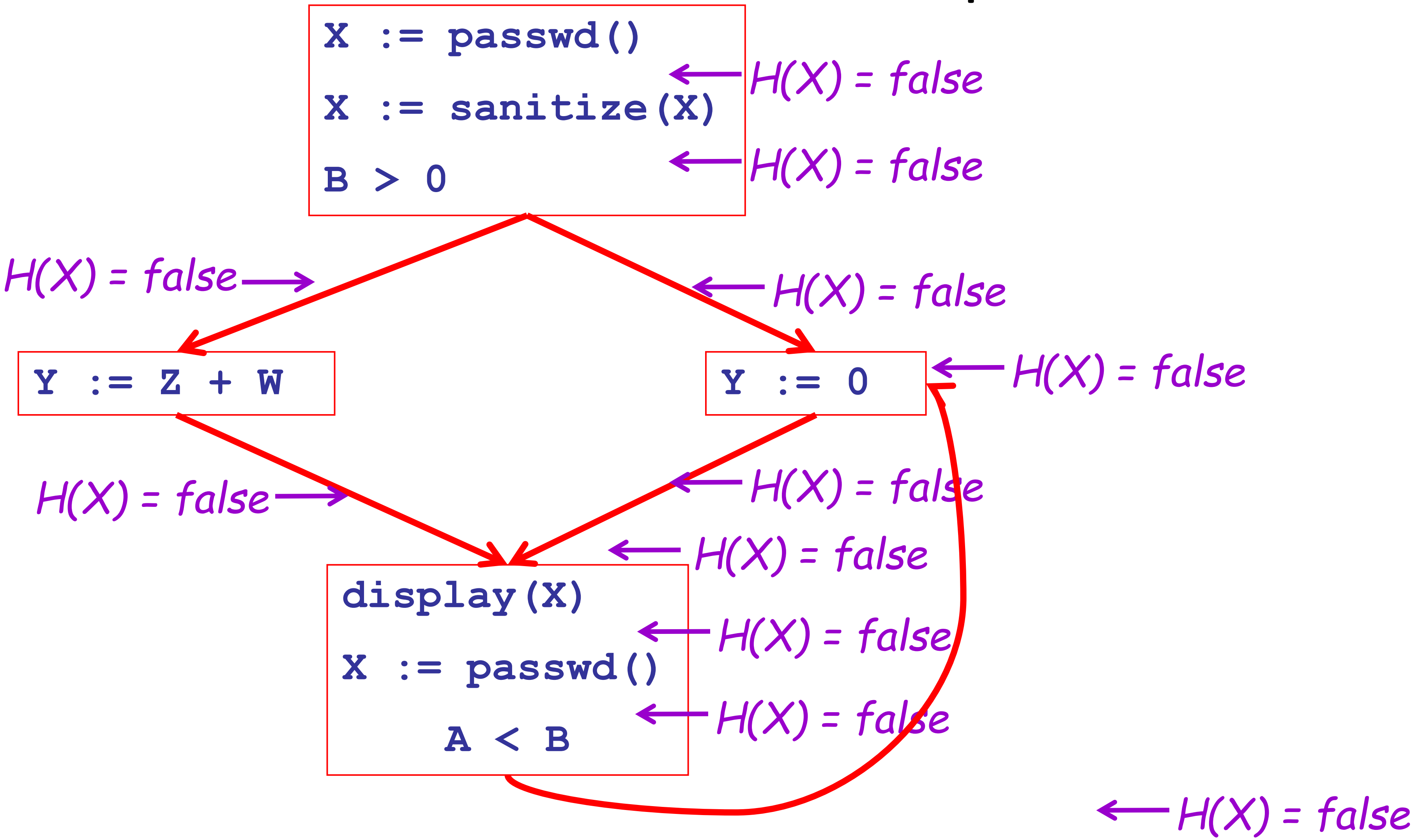
display(x). Do we care about y above?)

Algorithm

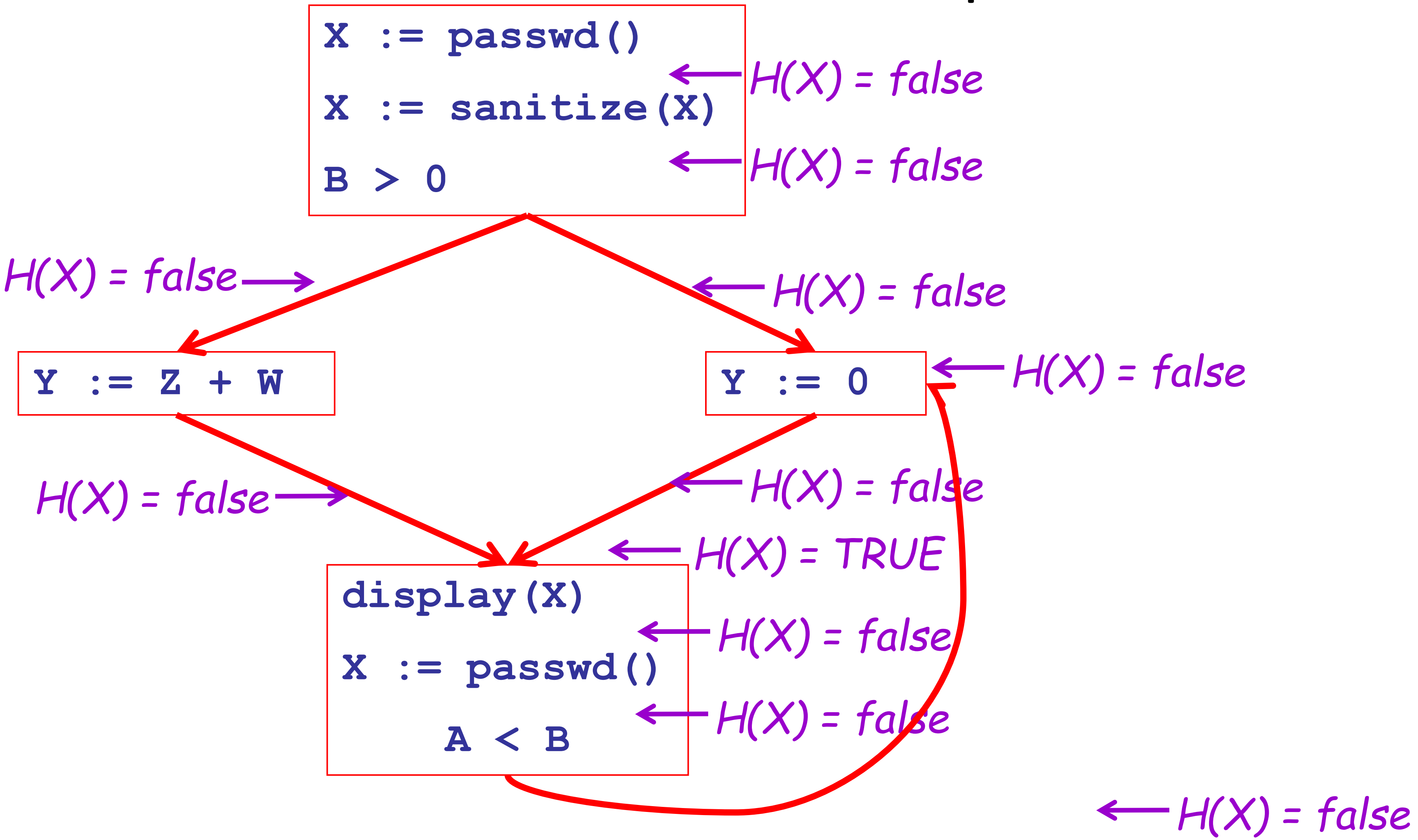
- Let all H_(…) = false initially

- 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

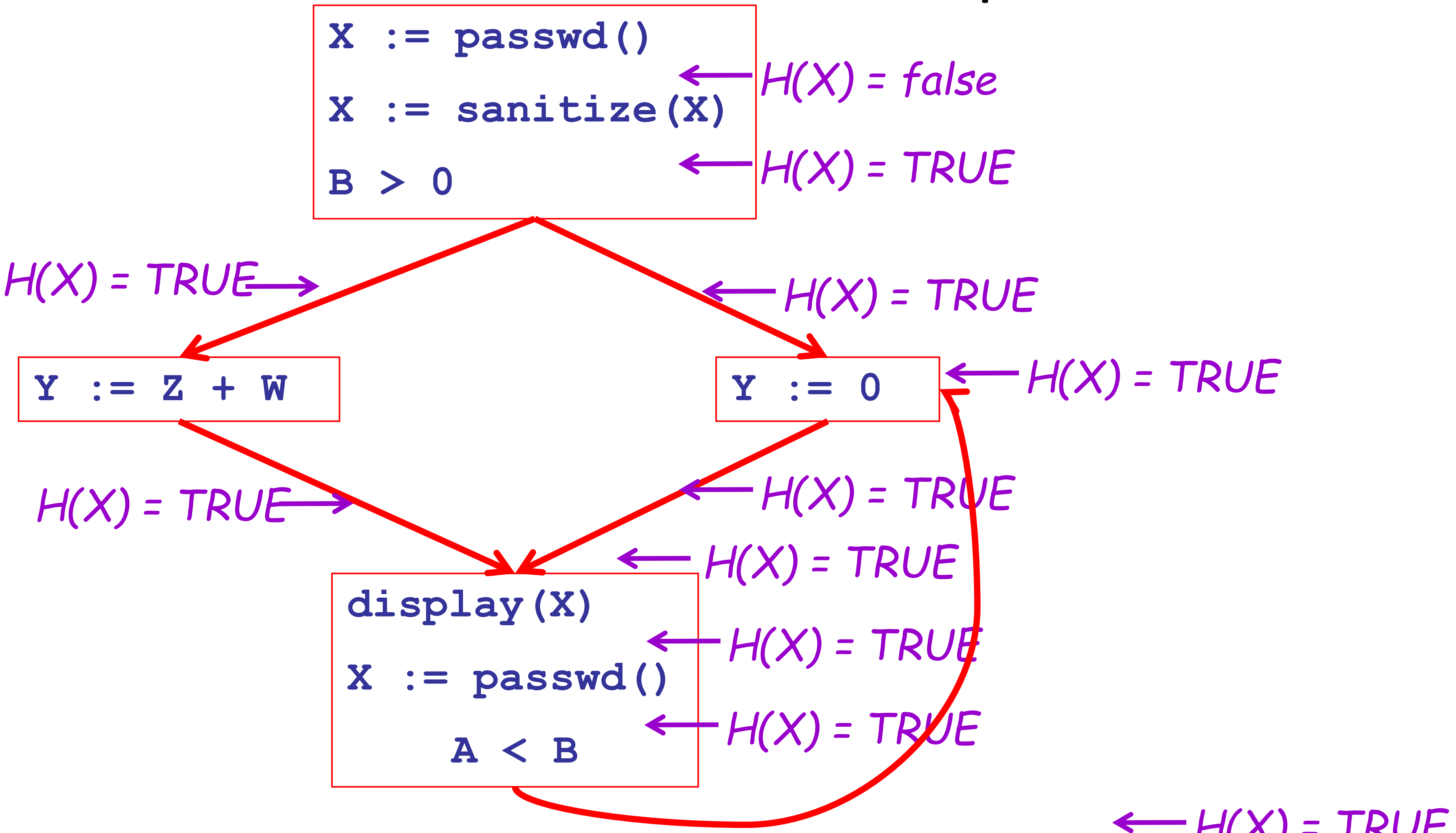
Secure Information Flow Example



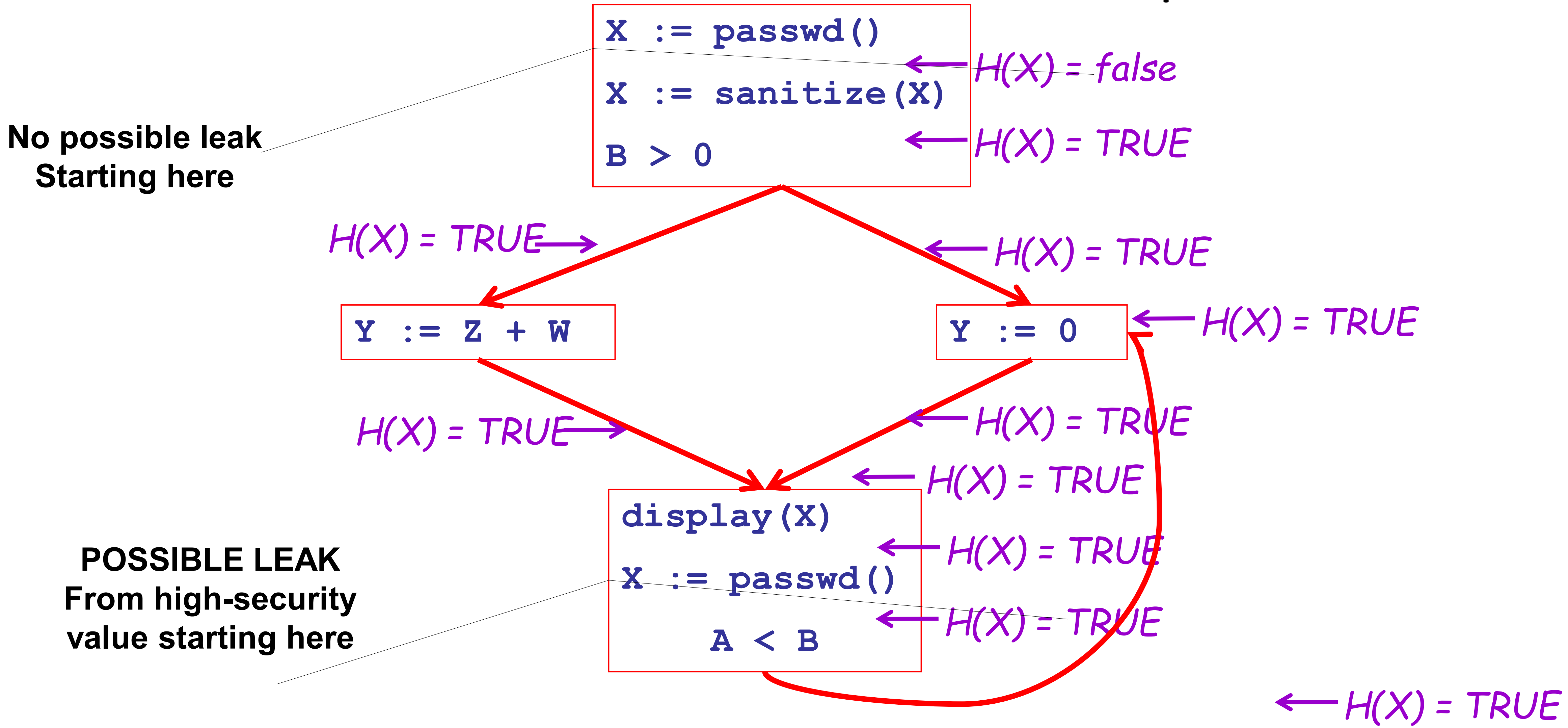
Secure Information Flow Example



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

- 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?

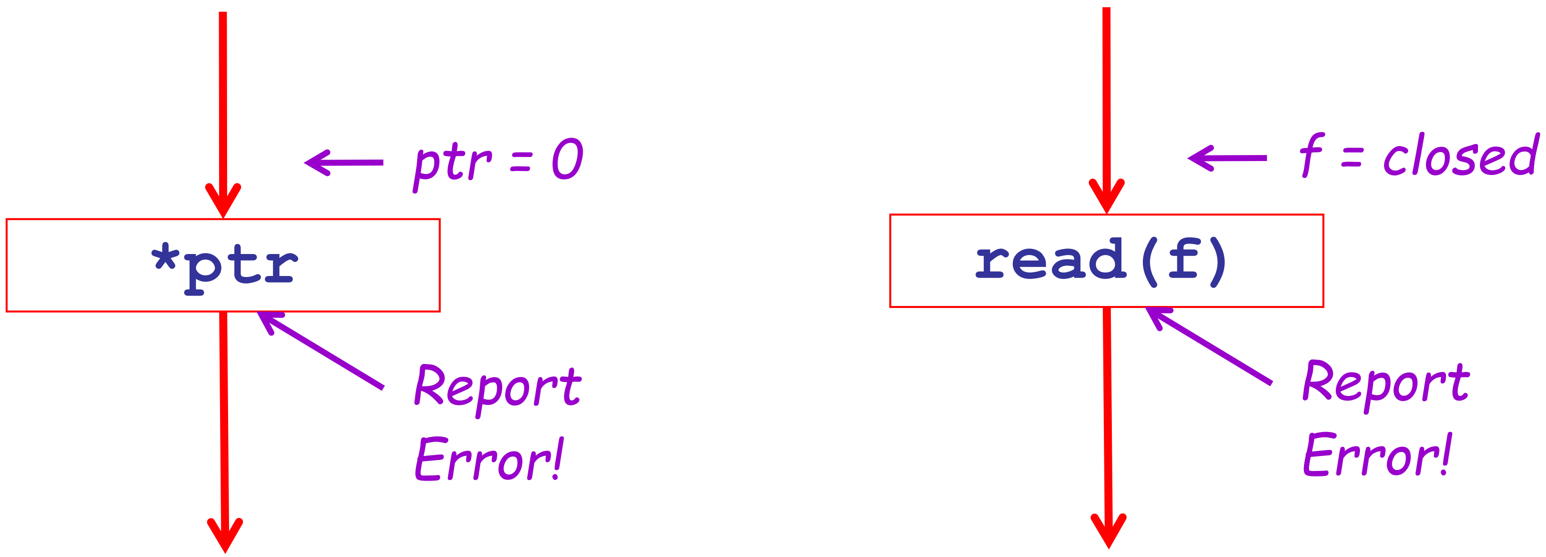
Abstract Information

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

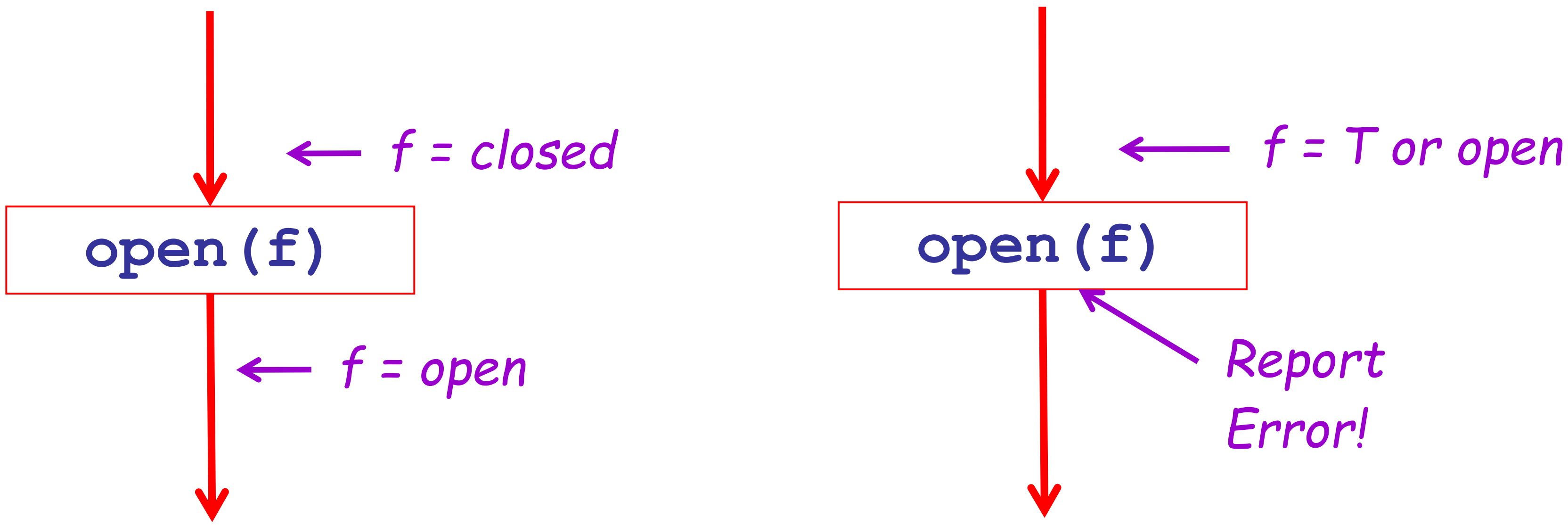
T	file handle state is unknown
⊥	haven't reached here yet
closed	file handle is closed
open	file handle is open

Rules

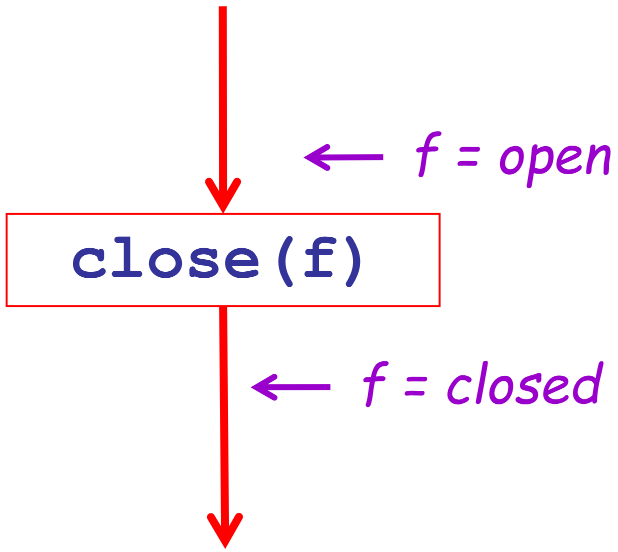
- Previously: “null ptr”
- Now: “file handles”

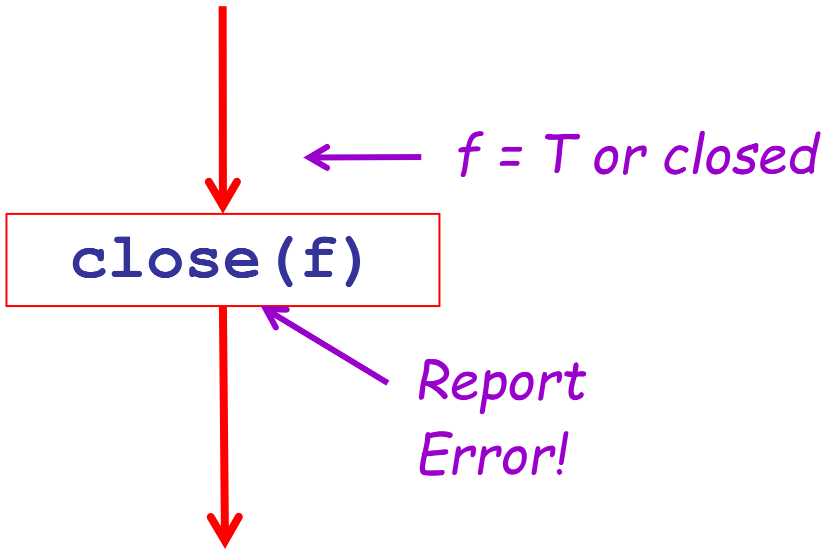


Rules: open



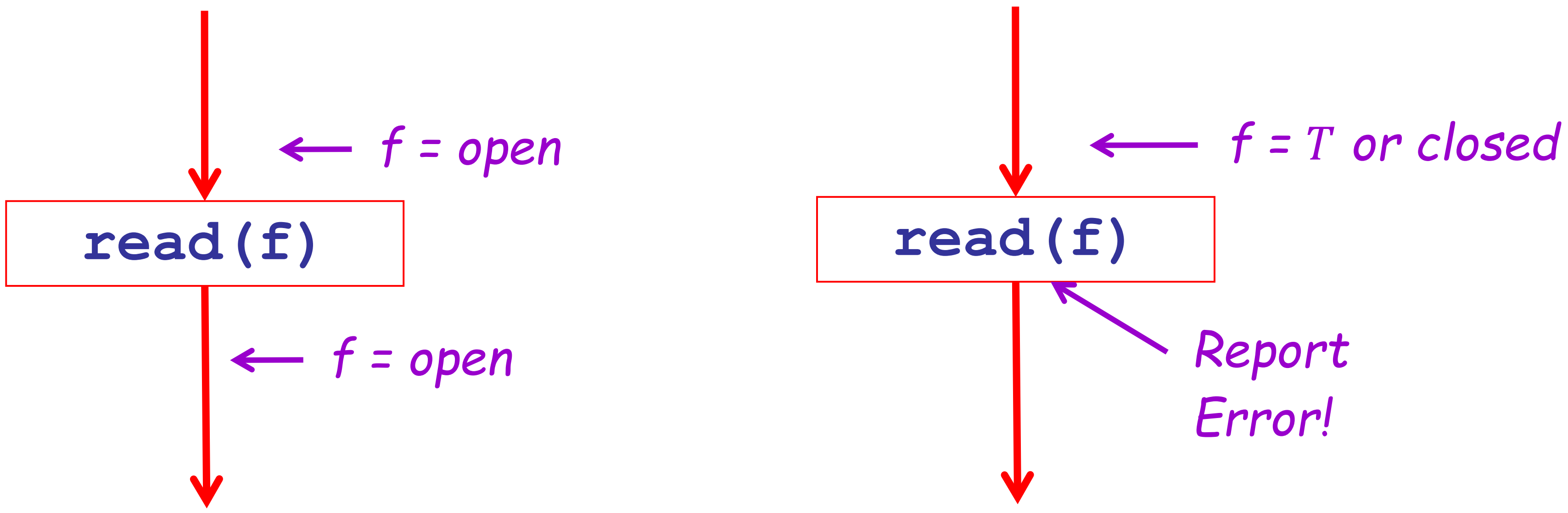
Rules: close



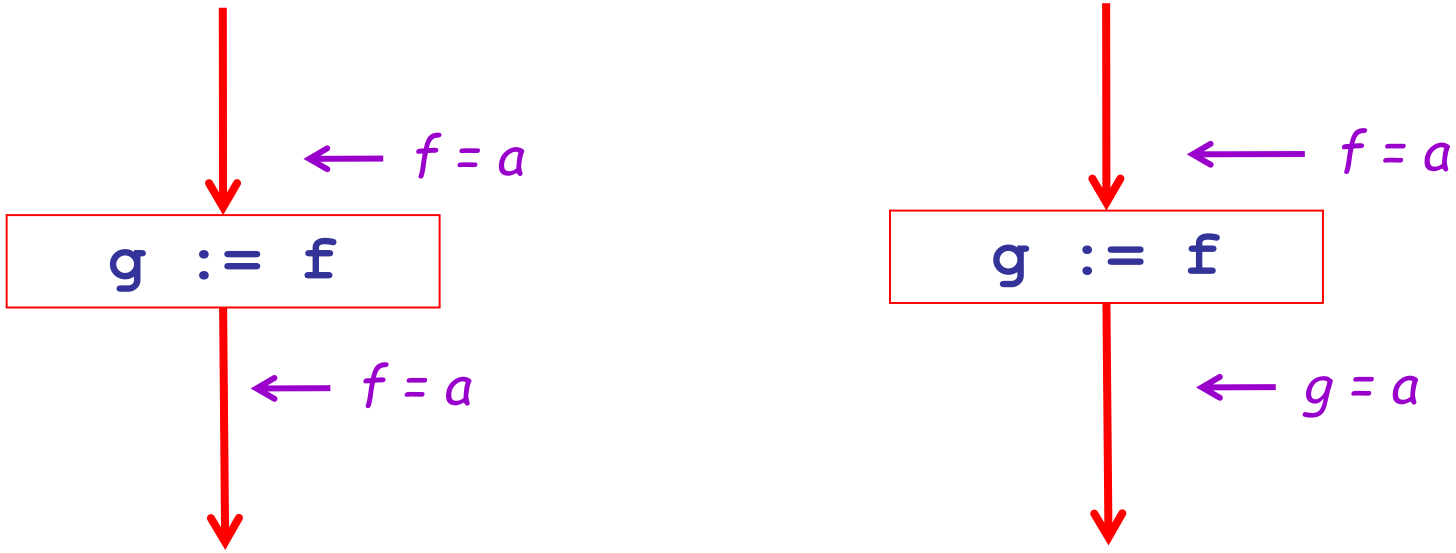


Rules: read/write

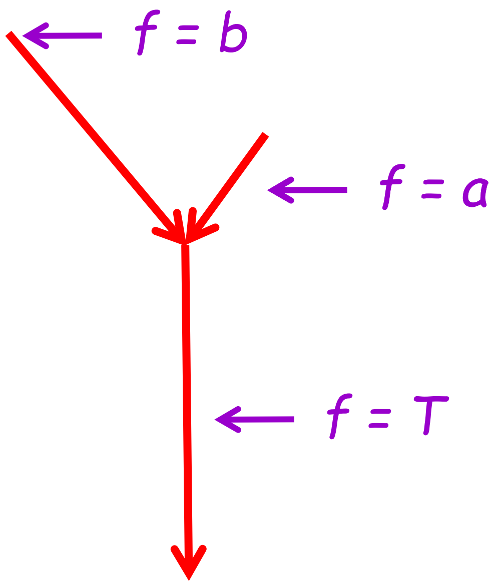
- (write is identical)

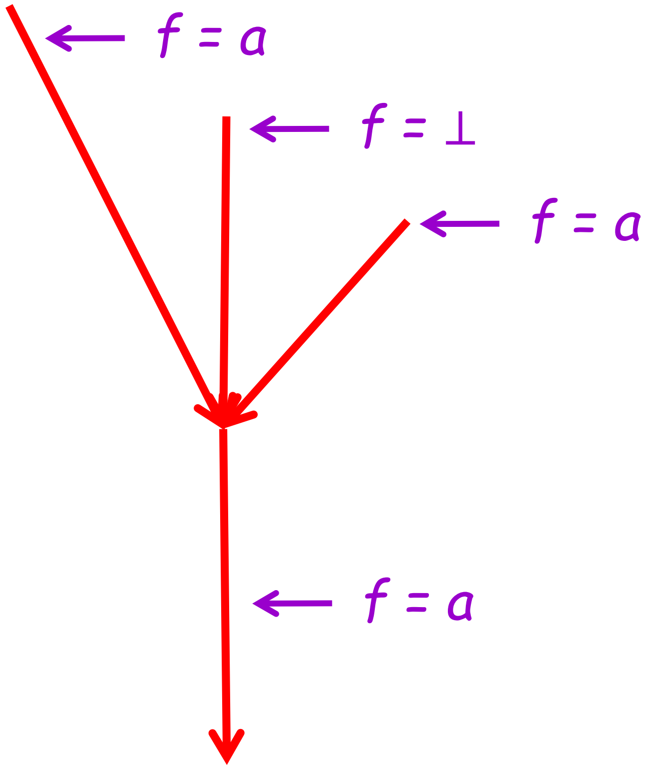


Rules: Assignment



Rules: Multiple Possibilities

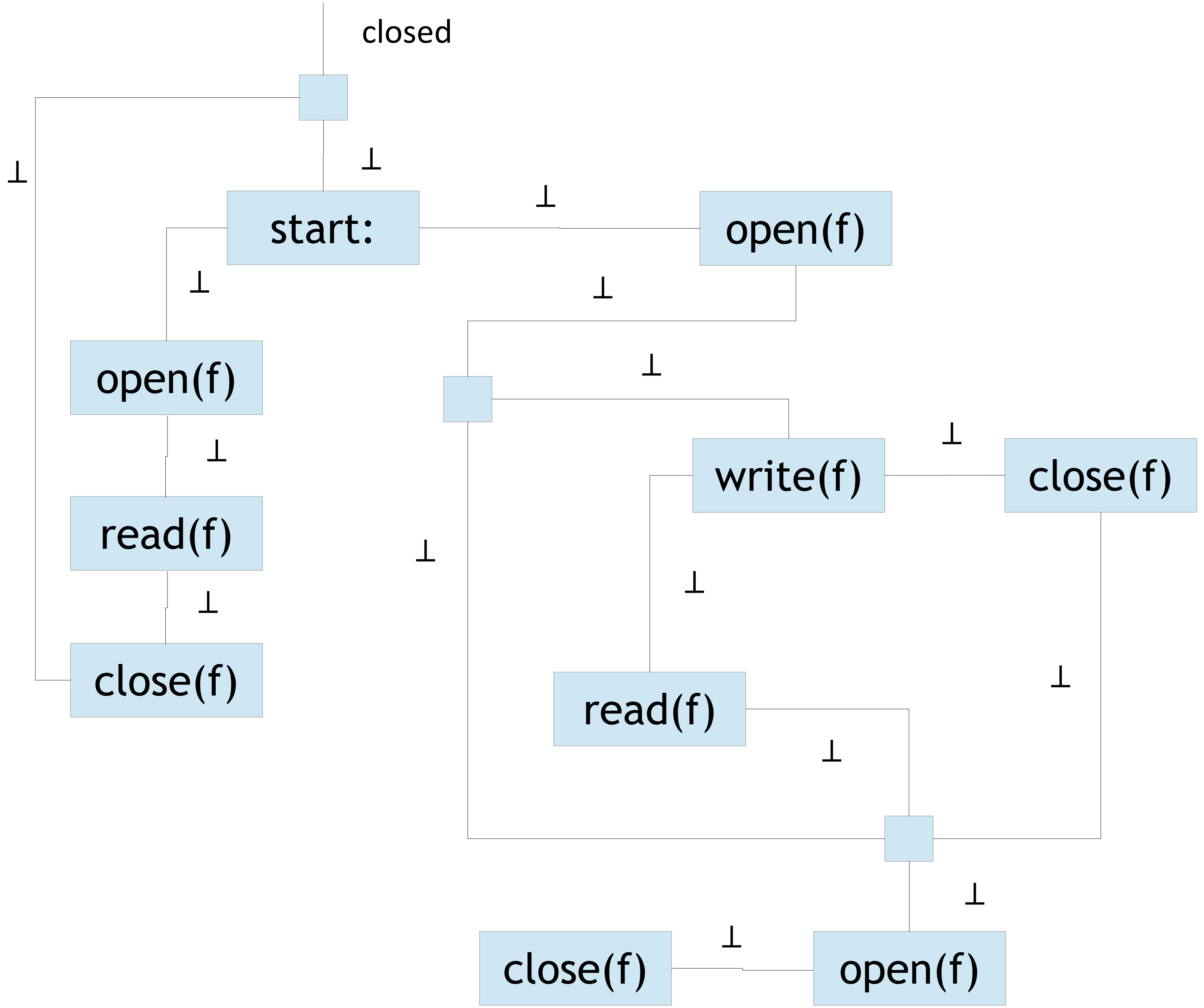


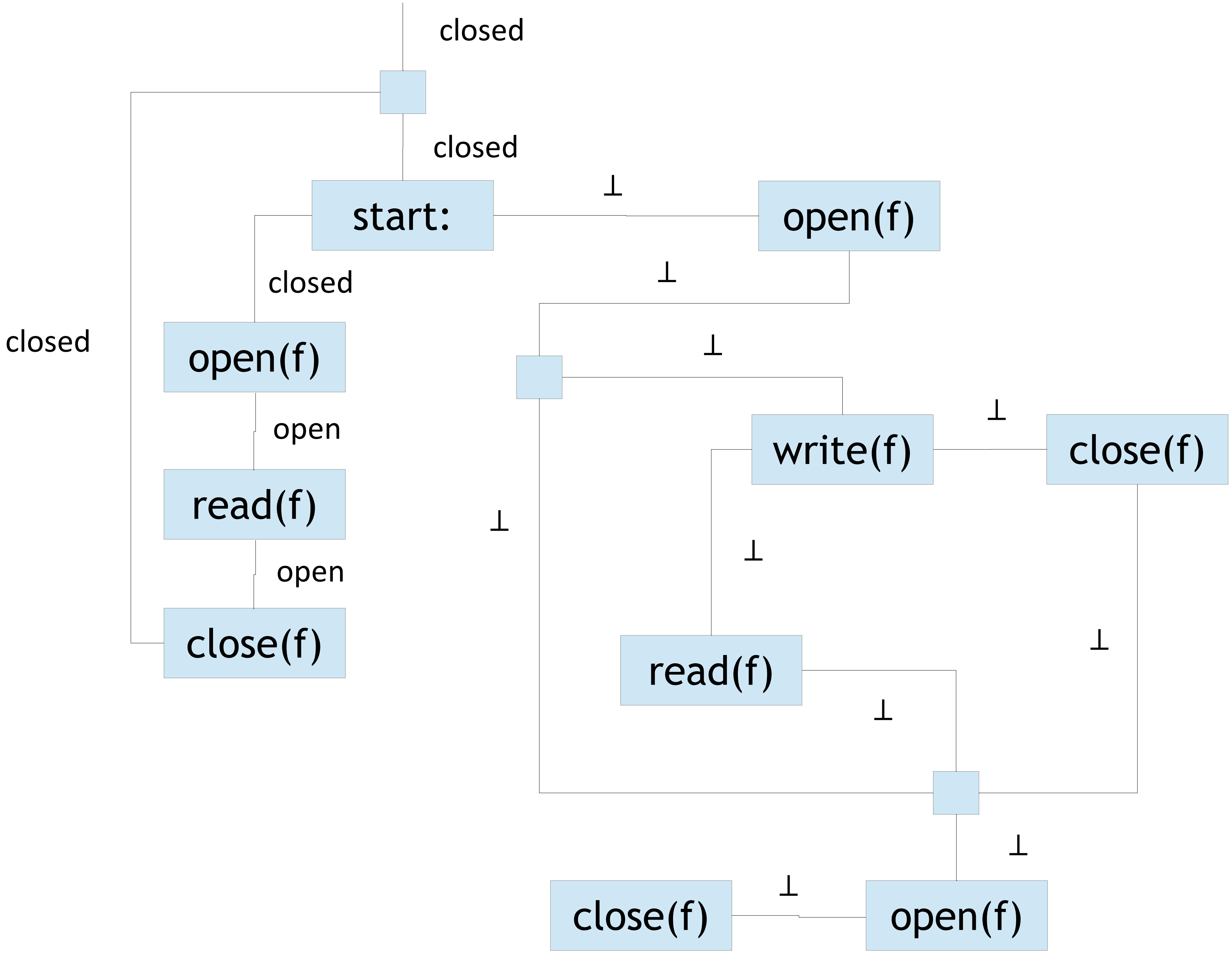


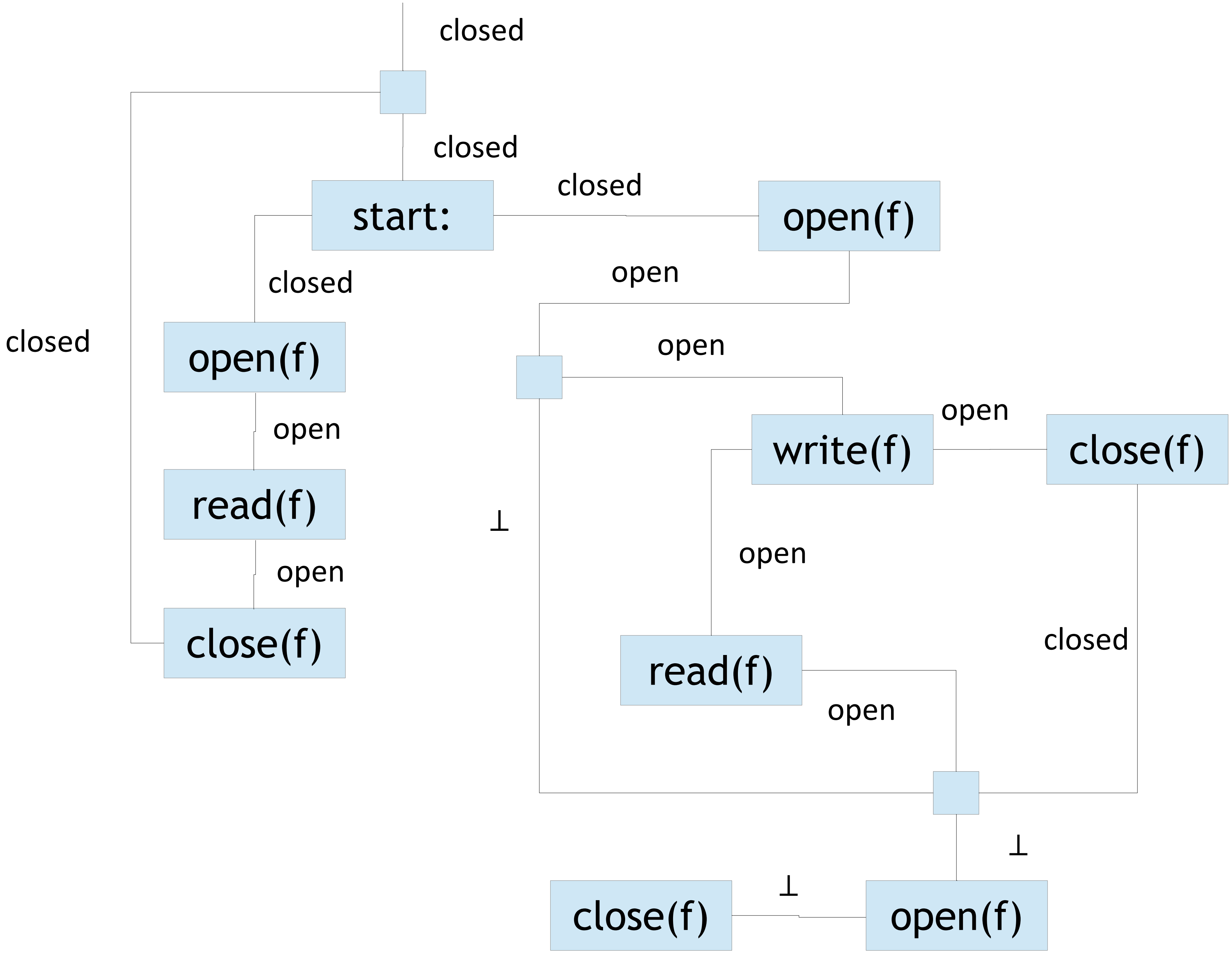
A Tricky Program

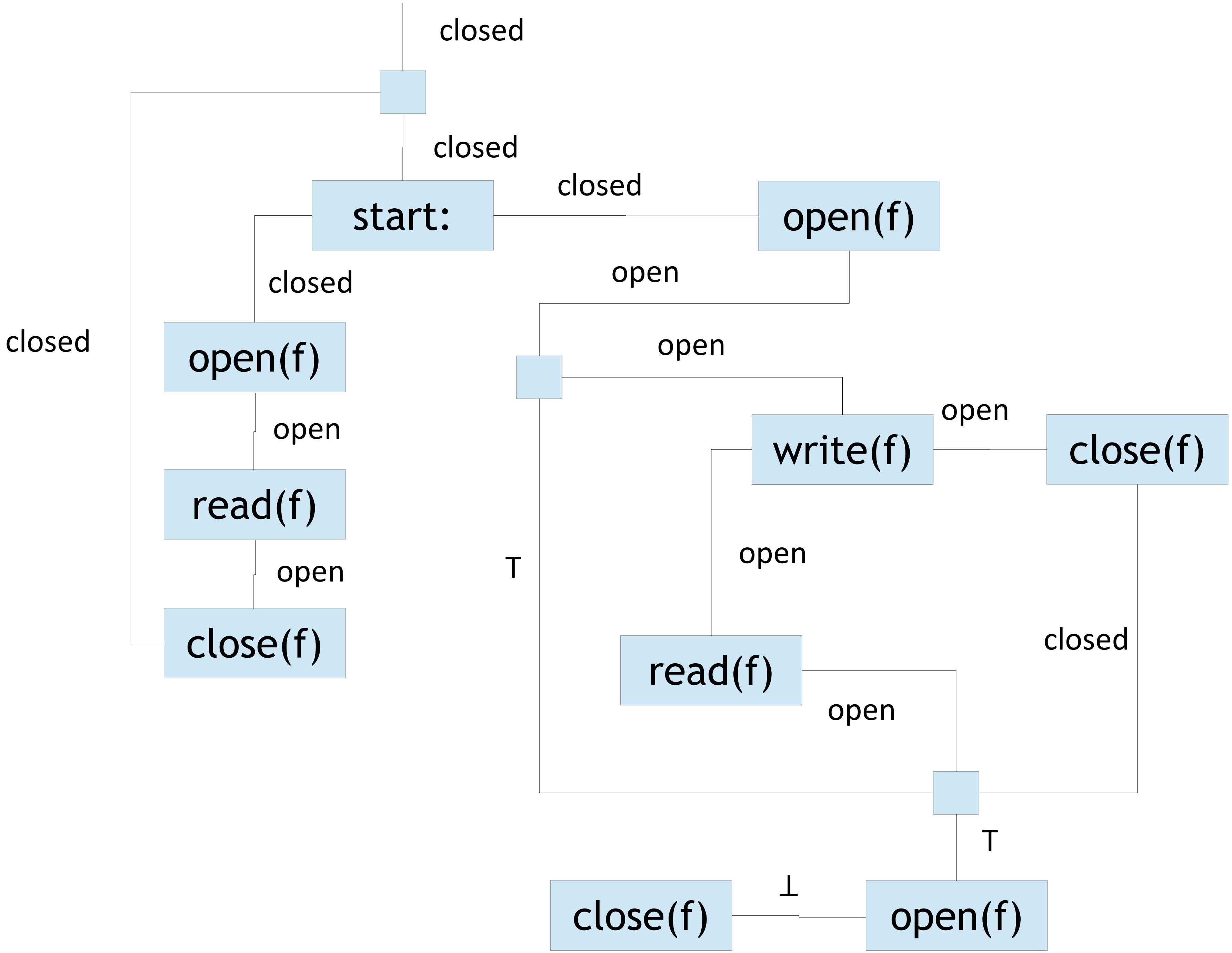
```
start:
switch (a)
  case 1: open(f); read(f); close(f); goto start
  default: open(f);
do {
  write(f) ;
  if (b):      read(f);
  else: close(f);
} while (b)
open(f);
close(f);
```

```
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);
```

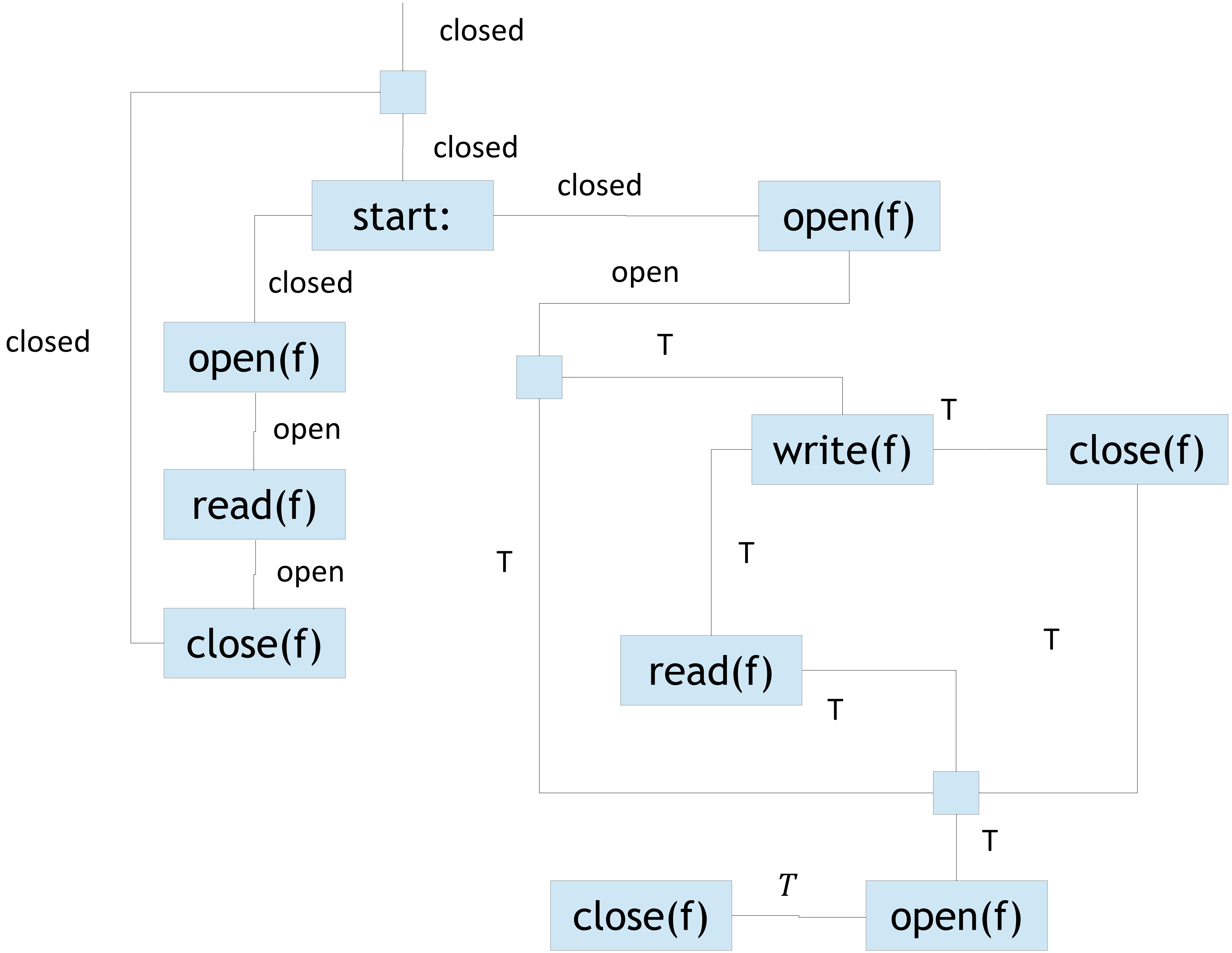


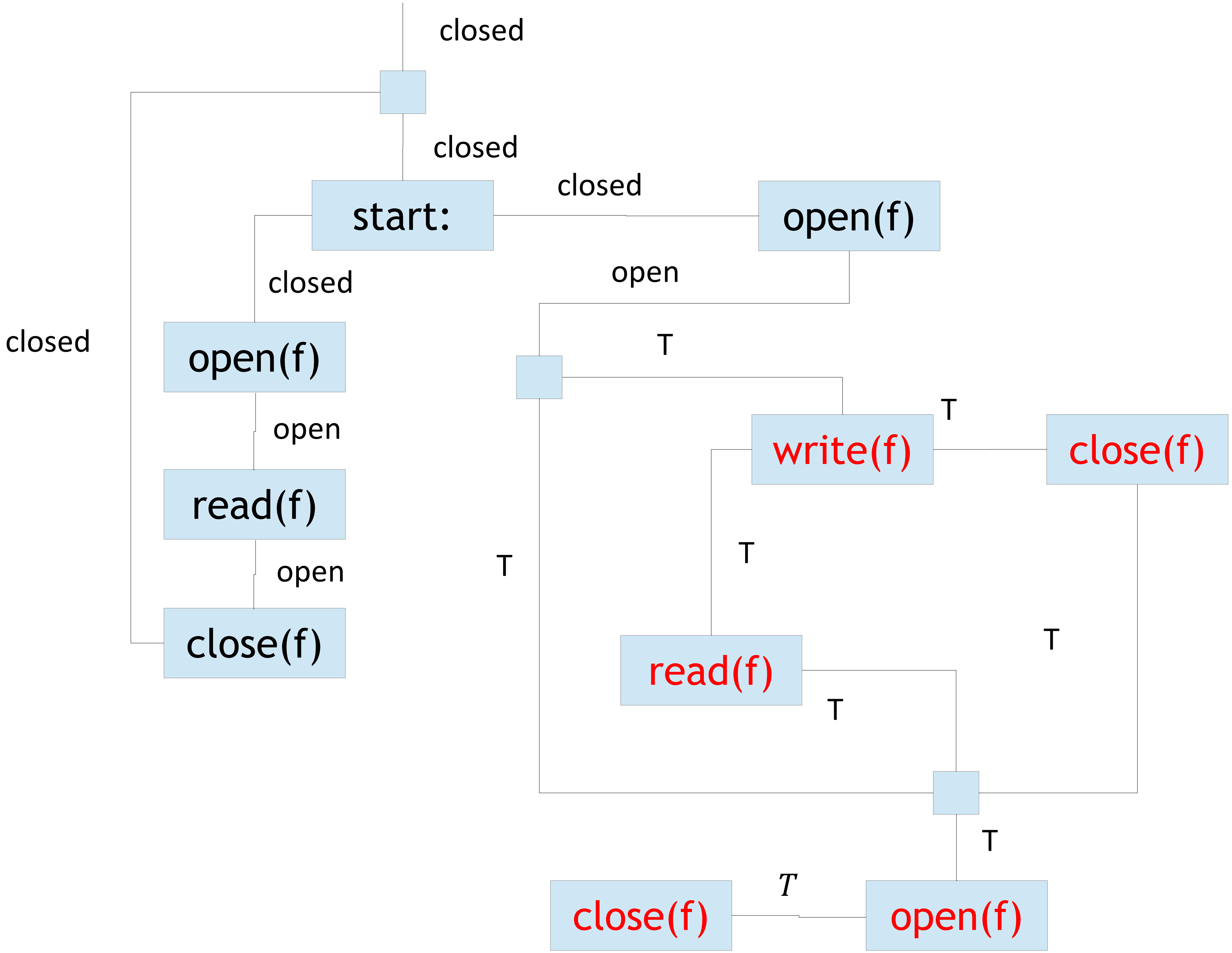

closed
closed
start:
⊥
open(f)
closed
closed
open(f)
⊥
⊥
open
read(f)
write(f)
⊥
close(f)
open
close(f)
⊥
⊥
read(f)
⊥
⊥
⊥
⊥
close(f)
open(f)


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



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


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


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

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);
```

Static analysis:
it is ok to be *conservative*

Forward vs. Backwards Analysis

- We've seen two kinds of analysis:

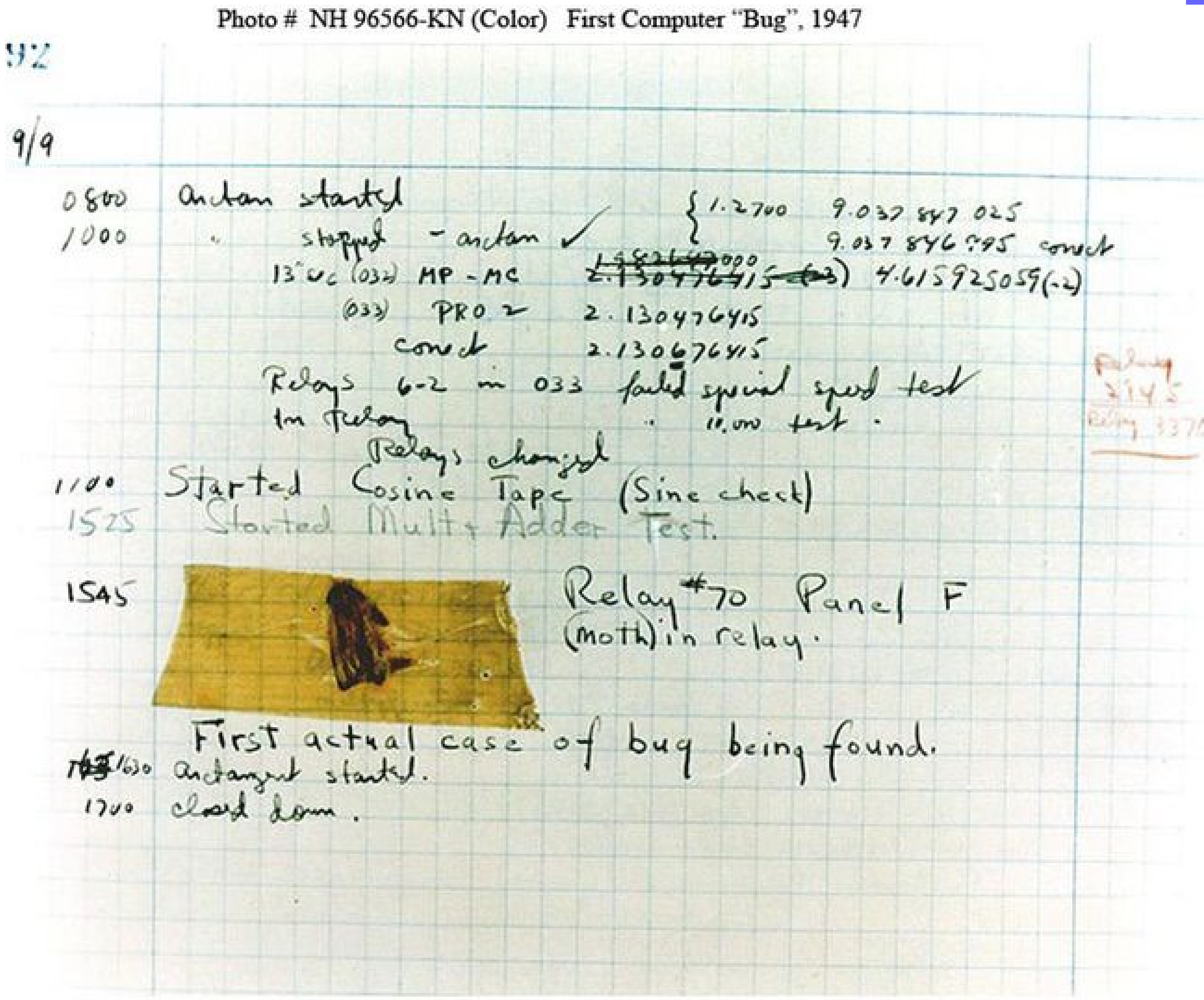
- Definitely null (cf. constant propagation) is a **forward** analysis: information is pushed from inputs to outputs

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

Trivia: Software “bug”

This computer scientist was one of the first programmers of the Harvard Mark I computer, a pioneer of computer programming who invented one of the first linkers and was the first to devise the theory of machine-independent PL (later extended to create COBOL).

In 1947, “First actual case of bug being found” in the Mark II computer at Harvard: a moth in the hardware. This computer scientist was not the one who found and reported the bug, but was the person who likely made the incident famous.



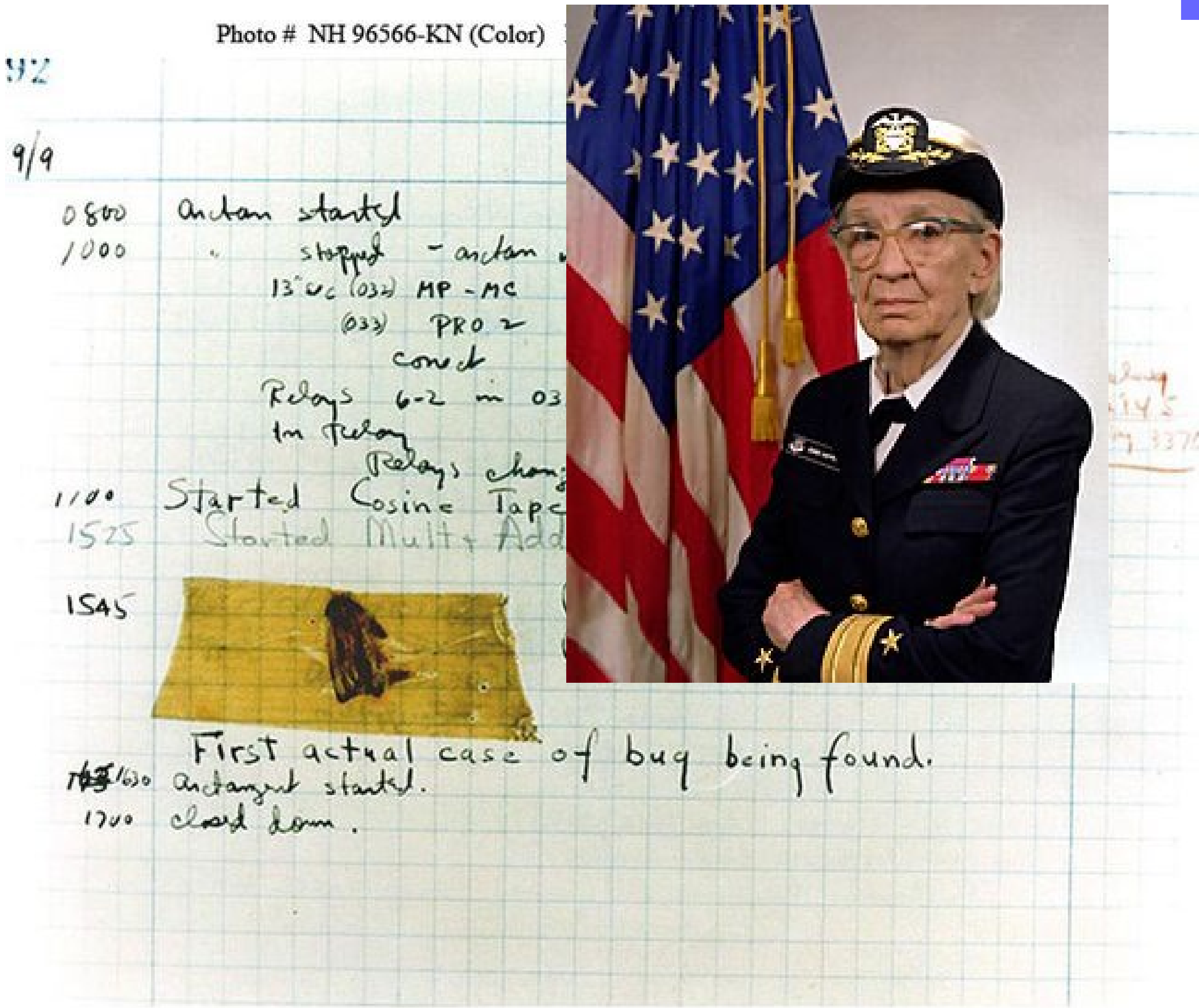
Trivia: Software "bug"

Grace Hopper

The Grace Hopper Celebration of Women in Computing (GHC)

This computer scientist was one of the first programmers of the Harvard Mark I computer, a pioneer of computer programming who invented one of the first linkers and was the first to devise the theory of machine-independent PL (later extended to create COBOL).

In 1947, "First actual case of bug being found" in the Mark II computer at Harvard: a moth in the hardware. This computer scientist was not the one who found and reported the bug, but was the person who likely made the incident famous.



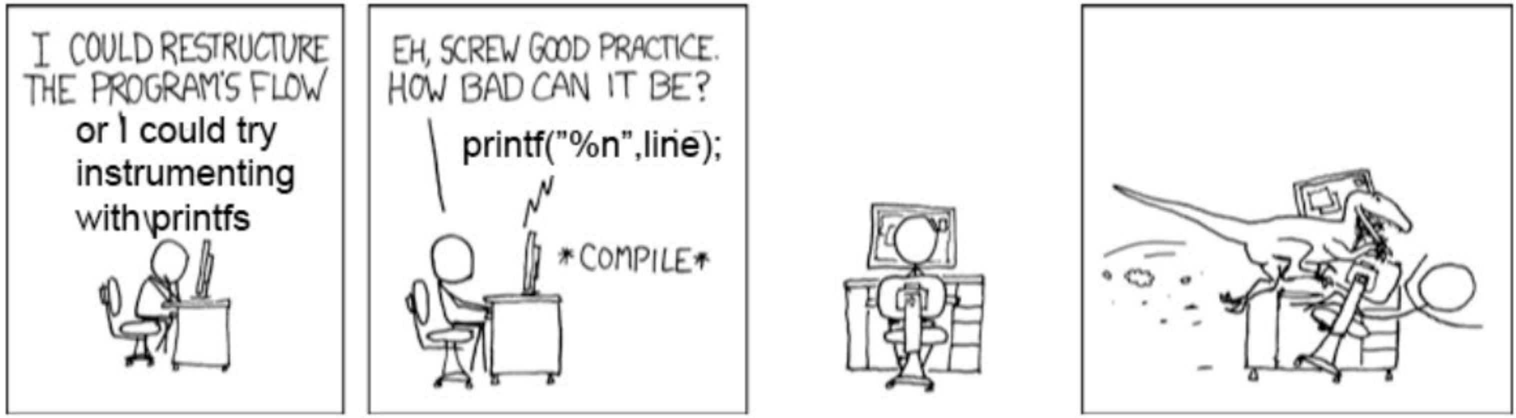
Dynamic Analysis

- The “easier” way?
  - **Testing**
    - Edge/Path Coverage
  - Information flow tracking
  - Execution time profiling
- A **dynamic analysis** runs an instrumented program in a controlled manner to collect information which can be analyzed to learn about a property of interest.

Difficult Questions

- Does this program have a race condition?
- Does this program run quickly enough?
- How much memory does this program use?
- Is this predicate an invariant of this program?
- Does this test suite cover all of this program?
- Can an adversary's input control this variable?
- How resilient is this distributed application to failures?

Common Dynamic Analyses



- Run the program
- In a systematic manner
  - On controlled inputs
  - On randomly-generated inputs
  - In a specialized VM or environment
- Monitor internal state at runtime
  - Instrument the program: capture data to learn more than “pass/fail”
- Analyze the results

Testing

- “**Software testing** is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.”
- A typical test involves input data and a comparison of the output. (More next lecture!)
- Note: unless your input domain is finite, testing does *not* prove the absence of all bugs.
- Testing gives you confidence that your implementation adheres to your specification.

Fuzz Testing (Fuzzing)

- How can we generate many different inputs fast?
- Input massive amounts of random data ("fuzz"), to the test program in an attempt to make it crash/expose bad behavior

Fuzz Testing (Fuzzing)

- Barton Miller, University of Wisconsin, 1989
  - A night in 1988 with thunderstorm and heavy rain
  - Connected to his office Unix system via a dial up connection
  - The heavy rain introduced noise on the line
  - Crashed many UNIX utilities he had been using everyday
  - He realized that there was something deeper
  - Asked three groups in his grad-seminar course to implement this idea of fuzz testing:
    - Two groups failed to achieve any crash results!
    - The third group succeeded! Crashed 25-33% of the utility programs on the seven Unix variants that they tested

Fuzz Testing (Fuzzing)

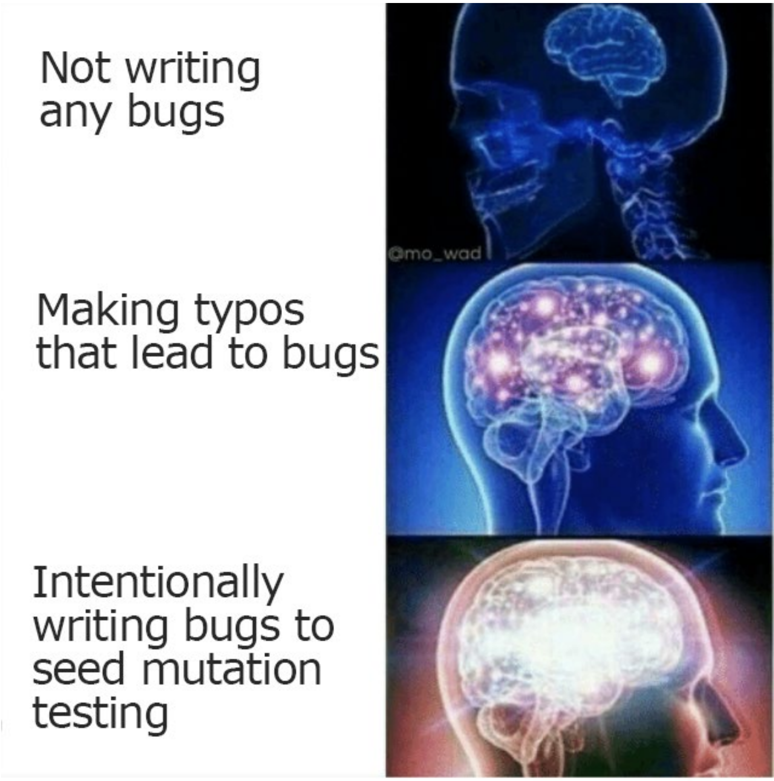
- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate random inputs with crashes
  - Errors found: Not checking returns, Array indices out of bounds, not checking null pointers, …
- American Fuzzy Lop (AFL)
  - Fuzzing by applying various modifications to the input file

Mutation Testing

- **Mutation testing** (or **mutation analysis**) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds.
- Informally: "You claim your test suite is really great at finding security bugs? Well, I'll just intentionally add a bug to my source code and see if your test suite finds it!"

Defect Seeding

- **Defect seeding** is the process of *intentionally introducing* a defect into a program. The defect introduced is similar to defects introduced by *real developers*. The seeding is typically done by changing the **source code**.
- For mutation testing, defect seeding is typically done automatically (given a model of what human bugs look like)
  - You will do this in Homework 3



Mutation Operators

- A **mutation operator** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects. Examples:
- `if (a < b)` → `if (a <= b)`
- `if (a == b)` → `if (a != b)`
- `a = b + c` → `a = b – c`
- `f(); g();` → `g(); f();`
- `x = y;` → `x = z;`

Mutant

- A **mutant** (or **variant**) is a version of the original program produced by applying one or more mutation operators to one or more program locations. The **order** of a mutant is the number of mutation applied.

```
// original
if (a < b):
  x = a + b
  print(x)
```

→

```
// 2nd-order mutant
if (a <= b):
  x = a − b
  print(x)
```

Competent Programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
- Programmers write programs that are largely correct. Thus the mutants simulate the likely effect of real faults. Therefore, if the test suite is good at catching the artificial mutants, it will also be good at catching the unknown but real faults in the program.

Do Humans Really Make Simple Mistakes?

Competent?

- Is the competent programmer hypothesis true?

```
// return true if x is greater
// than or equal to y
bool value_to_return;
if(x > y) {
  value_to_return = true;
}
if(x < y) {
  value_to_return = false;
}
if(x == y) {
  value_to_return = true;
}
return value_to_return;
```

Competent?

- Is the competent programmer hypothesis true?

- Yes and no.
- It is certainly true that humans often make simple typos (e.g., + to -).
- But it is also true that some bugs are more complex than that.

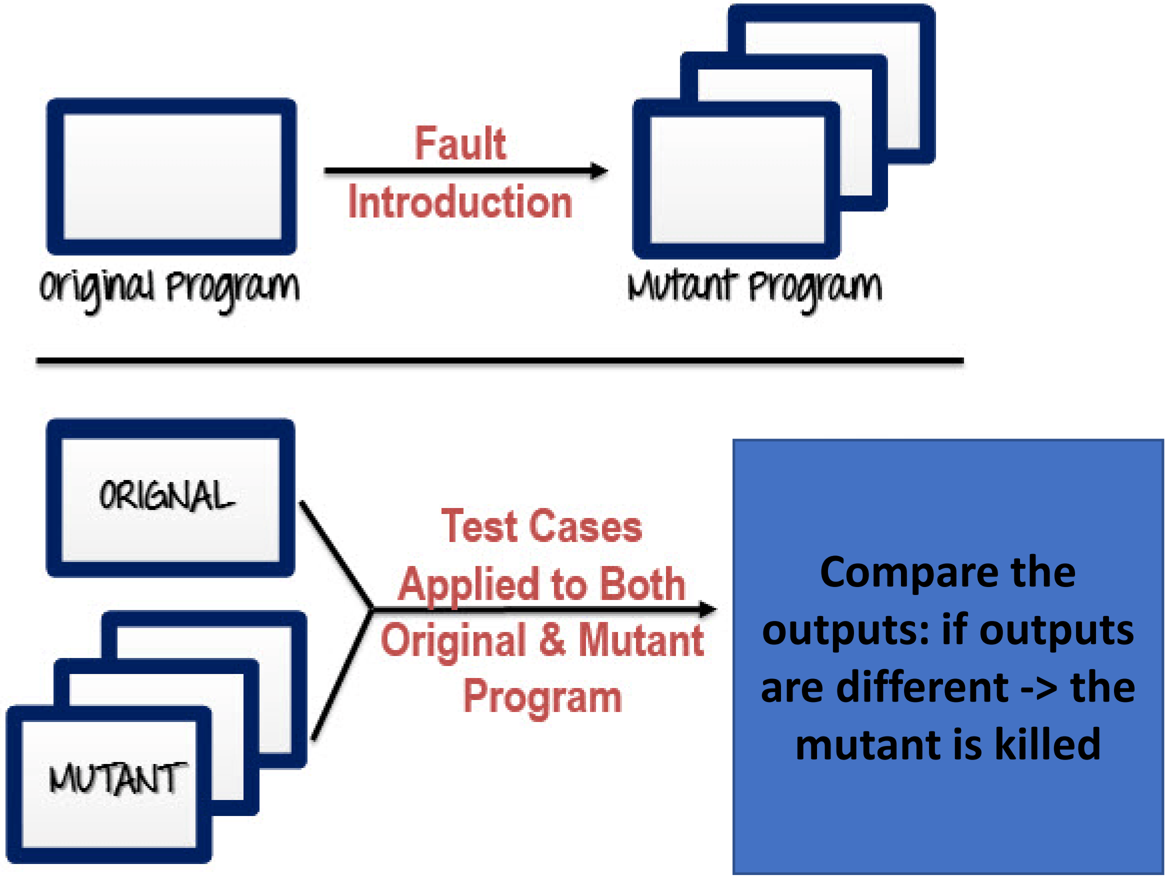
Coupling Effect

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is it true?
  - Tests that detect simple mutants were also able to detect over 99% of second- and third-order mutants historically
    [A. J. Offutt. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, Jan. 1992.]

Mutation Testing

- A test suite is said to **kill** (or **detect**, or **reveal**) a mutant if the mutant fails a test that the original passes.
- **Mutation testing** (or **mutation analysis**) of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the **mutation adequacy score** (or **mutation score**).
  - A test suite with a higher score is better.

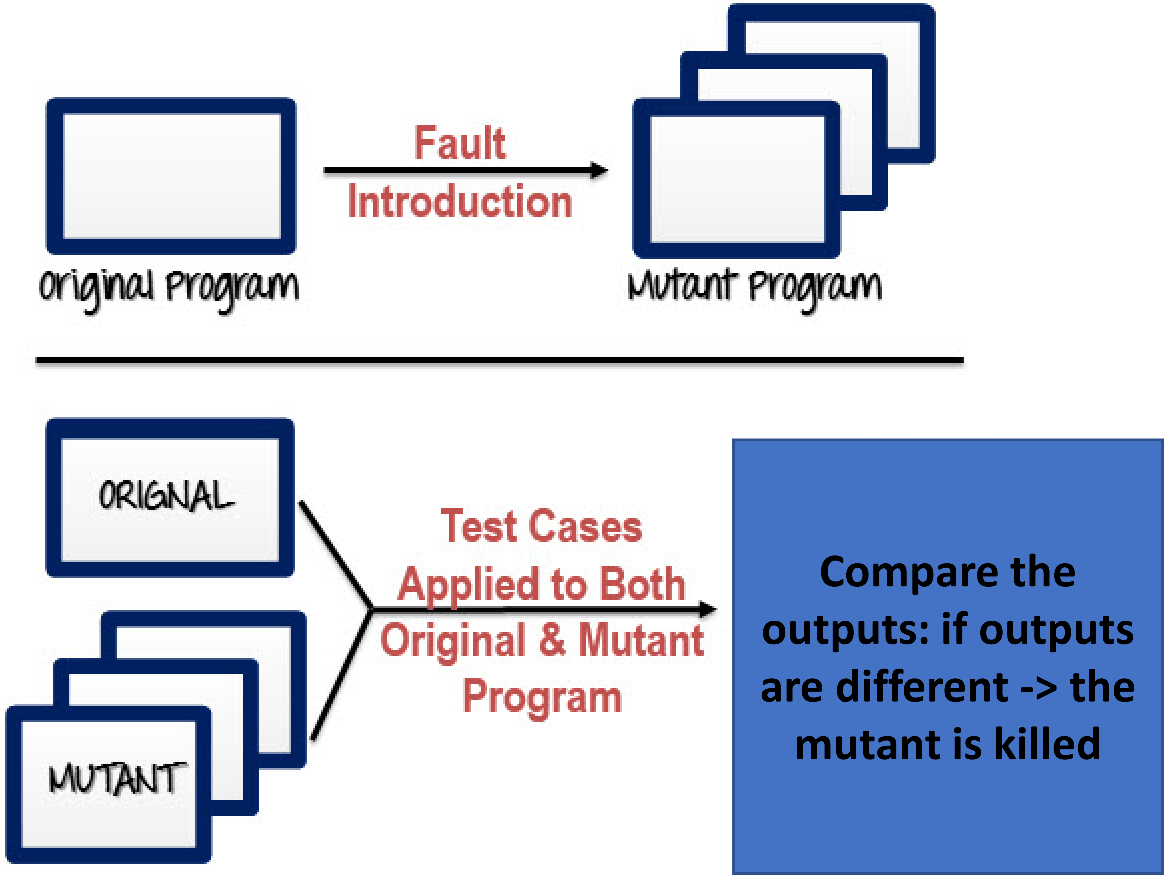
Mutation Testing



Mutation Testing

- Stillborn mutants
  - Syntactically incorrect, killed by compiler: e.g., x=a++b
- Trivial mutants
  - Killed by almost any test case
- Equivalent mutants → HARD
  - Always acts in the same behavior as the original program: e.g., x=a+b and x=a-(-b)

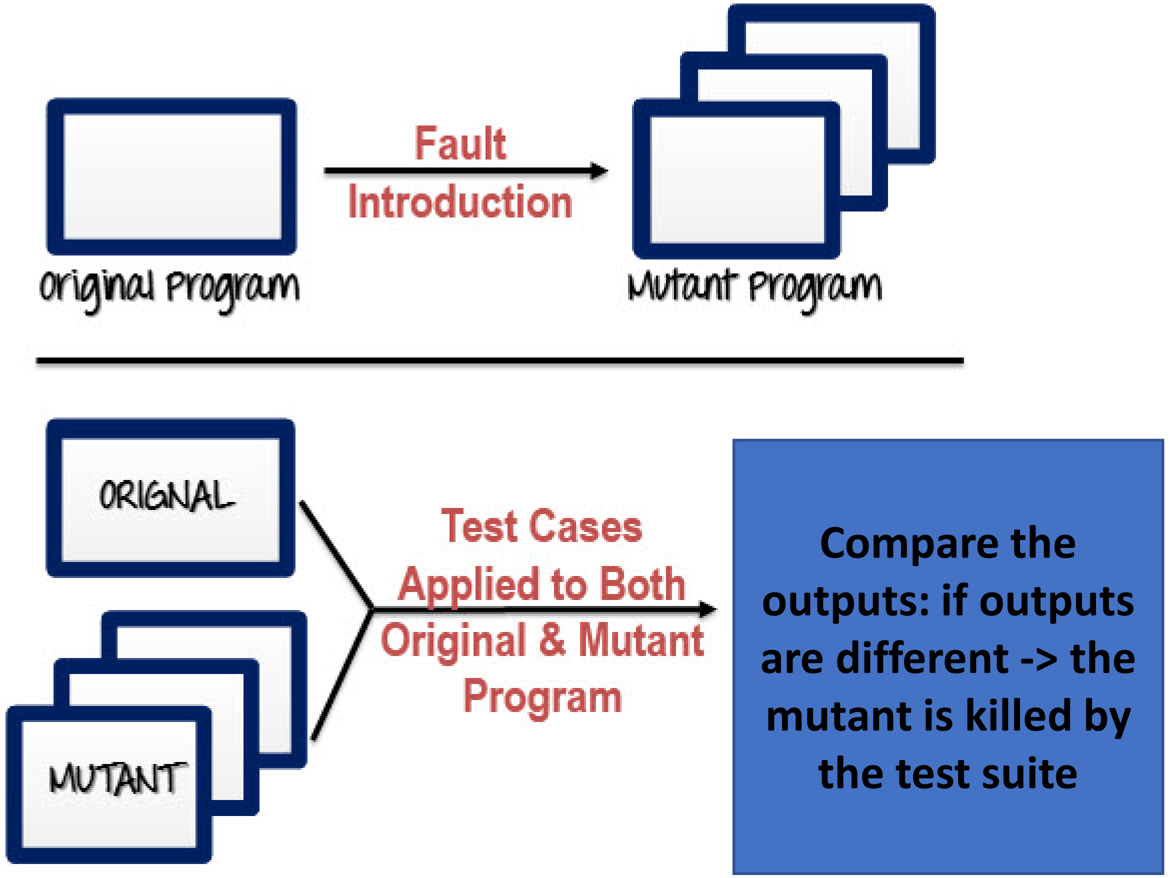
•None of the above is interesting.
•We care about mutants that behave differently but we don't have test cases to identify them



Mutation Testing

- Mutation score =

number of mutants killed / total number of mutants * 100



Equivalent Mutant Problem

- Suppose you have “x = a + b; y = c + d;” and you swap those two statements.
- The resulting program is a mutant, but it is semantically **equivalent** to the original.
  - So it will pass and fail all of the tests that the original passes and fails.
- So it will dilute the mutation score
- Detecting equivalent mutants is a big deal. How hard is it?

Equivalent Mutant Problem

- Detecting equivalent mutants is a big deal. How hard is it?
- It is **undecidable**!
  - By direct reduction to the halting problem, or by Rice's Theorem

```
foo:                            # foo halts if and only if
  if p1() == p2():             # p1 is equivalent to p2
    return 0
  foo()
```

Fault Localization

- With testing, you know there is a bug. But, where is it?!

Fault Localization

- **Fault localization** is the task of identifying source code regions implicated in a bug
  - “This regression test is failing. Which lines should we change to fix things?”
- Answer is not unique: there are often many places to fix a big
  - Example: check for null at caller or callee?
- Debugging includes fault localization
- Answer may take the form of a list (e.g., of lines) ranked by suspiciousness

Spectrum-Based Fault Localization

- **Spectrum-based fault localization** uses a dynamic analysis to rank suspicious statements implicated in a fault by comparing the statements covered on failing tests to the statements covered on passing tests
- Basic idea:
  - Instrument the program for coverage (put print statements everywhere)
  - Run separately on normal inputs and bug-inducing inputs
  - Compute the set difference on coverage!

Fault Localization Example

- Consider this simple buggy program:

```
int mid(int x, int y, int z) {
  int m;
  m = z;
  if (y < z) {
    if (x < y) m = y;
    else if (x < z) m = y; /* BUG: m=x; */
  } else {
    if (x > y) m = y;
    else if ( x > z) m = x;
  }
  return m;
}
```

Coverage-Based Fault Localization

Statement	3,3,5	1,2,3	3,2,1	3,2,1	5,5,5	2,1,3
int m;						
m = z;						
if (y < z)						
if (x < y)						
m = y;						
else if (x<z)						
m = y; // bug						
else						
if (x > y)						
m = y;						
else if (x>z)						
m = x;						
return m;						
	Pass	Pass	Pass	Pass	Pass	Fail

Insight: Print-Statement Debugging

- If you do not execute X but you do observe the bug, X cannot be related to that bug
- If Y is primarily executed when you observe the bug, it is more likely to be implicated than Z which is primarily executed when you do not observe the bug
- **Suspiciousness Ranking**

$$Suspicious(s) = \frac{fail(s) / totalfail}{fail(s) / totalfail + pass(s) / totalpass}$$

[Jones and Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. ASE 2005.]

Fault Localization Ranking

Statement	3,3,5	1,2,3	3,2,1	3,2,1	5,5,5	2,1,3	susp(s)
int m;							**0.5**
m = z;							**0.5**
if (y < z)							**0.5**
if (x < y)							**0.63**
m = y;							**0**
else if (x<z)							**0.71**
m = y; // bug							**0.83**
else							**0**
if (x > y)							**0**
m = y;							**0**
else if (x>z)							**0**
m = x;							**0**
return m;							**0.5**
	Pass	**Pass**	**Pass**	**Pass**	**Pass**	**Fail**	

Then what? Automated Program Repair (APR)

- Testing: I know there is a bug!
- Fault localization: I know where the bug is!! (approximately…)
- Automated program repair: It can fix the bug for me!!!
  - “Fix the bug” = = Apply a patch so that the program can pass all the previously failing test cases (also pass all the previously passing test cases)

APR: How could that work? – The approach

- How do novices fix a buggy program?
  - Randomly change the program…until it works

APR: How could that work? – The Simplest approach

- How do novices fix a buggy program?
  - Randomly change the program…until it works

Mutation

APR: How could that work? – The Simplest approach

- How do novices fix a buggy program?
  - Randomly change the program…until it works

Fault Localization Mutation Testing Again

APR: How could that work?

- Many faults can be localized to a small area
  - Even if your program is a million lines of code, **fault localization** can narrow it to 10-100 lines
- Many defects can be fixed with small changes
  - **Mutation** (test metrics) can generate candidate patches from simple edits
  - A **search-based software engineering** problem
- Can use regression **testing** (inputs and oracles, continuous integration) to assess patch quality
- [Weimer et al. *Automatically Finding Patches Using Genetic Programming*. Best Paper Award. IFIP TC2 Manfred Paul Award. SIGEVO “Humies” Gold Award. Ten-Year Impact Award.]

APR: A More Sophisticated Approach

- If we had a cheap way to approximately decide if two programs are equivalent
  - We wouldn't need to test any candidate patch that is equivalent to a previously-tested patch
  - (Cluster or quotient the search space into equivalence classes with respect to this relation)
- We use **static analysis** (like a dataflow analysis for dead code or constant propagation) to decide this: 10x reduction in search space
- [Weimer et al. *Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results.*]

APR: A More Sophisticated Approach

- In mutation testing, the mutation operators are based on common human mistakes
- Instead, use human edits or **design patterns**
  - "Add a null check" or "Use a singleton pattern"
- Mine 60,000 human-written patches to learn the 10 most common fix templates
  - Resulting approach fixes 70% more bugs
  - Human study of non-student developers (n=68): such patches are 20% more acceptable
- [Kim et al. *Automatic Patch Generation Learned from Human-Written Patches.* Best paper award.]

Relationship with Mutation Testing

- This program repair approach is a dual of **mutation testing**
  - This suggests avenues for cross-fertilization and helps explain some of the successes and failures of program repair.

- Very informally:
  - PR Exists M in Mut. Forall T in Tests. M(T)
  - MT Forall M in Mut. Exists T in Tests. Not M(T)