Static Analysis (1/2)

Martin Kellogg

Static Analysis (Part 1/2)

Today's agenda:

- **Finish slides on build systems**
- Motivations for static analysis
- Basics of dataflow analysis
- Reading Quiz

How to choose a build system

How to choose a build system

High level idea: same rules apply to choosing a language

How to choose a build system

High level idea: same rules apply to choosing a language

- **don’t change what’s already there** unless there is a good reason

How to choose a build system

High level idea: same rules apply to choosing a language

- **don't change what's already there** unless there is a good reason
- **follow convention** and prefer the tooling that's "idiomatic" to your language
  - e.g., use Gradle or Maven when working in Java

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)
    - build has become too complex for a declarative task language

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)
    - build has become too complex for a declarative task language
  - most projects keep the same build system **forever**

Best practices

- Automate everything

Best practices

- Automate everything
- Always use a build tool

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)

Your CI server is a good place to test that your build is hermetic. **Standard practice**: spin up a new CI server for **each build**.

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don't depend on anything that's not in the build file (hermetic)

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)
- Don’t break the build

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)
- Don’t break the build

> A **common mistake to avoid**: allowing the CI server to fail for a long time because “we know what the problem is.” Don’t do this: leads to complacency, missing real bugs.

Static Analysis (Part 1/2)

Today's agenda:

- **Motivations for static analysis**
- Basics of dataflow analysis
- Reading Quiz

Motivations for static analysis

Motivations for static analysis

- Quality assurance is critical to software engineering

Motivations for static analysis

- Quality assurance is critical to software engineering
- We’ve already covered three important QA techniques:

Motivations for static analysis

- Quality assurance is critical to software engineering
- We’ve already covered three important QA techniques:
  - **code review**, the most common **static** QA technique

Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique

Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique

Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique
- We've seen that both code review and testing have **significant limitations** in practice:

Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique
- We've seen that both code review and testing have **significant limitations** in practice:
  - code review is limited by human error

Motivations for static analysis

- Quality assurance is critical to software engineering
- We've already covered three important QA techniques:
  - **code review**, the most common **static** QA technique
  - **linting**, the second-most common static QA technique
  - **testing**, the most common **dynamic** QA technique
- We've seen that both code review and testing have **significant limitations** in practice:
  - code review is limited by human error
  - testing is limited by your choice of tests (Dijkstra again)

Motivations for static analysis

- Quality assurance is critical to
- We've already covered three i
  - **code review**, the most com
  - **linting**, the second-most c
  - **testing**, the most common
- We've seen that both code rev
  limitations in practice:
  - code review is limited by human error
  - testing is limited by your choice of tests (Dijkstra again)

Today's goal: discuss other **automated** static analysis techniques that complement testing and code review in a quality assurance process

Motivation: many defects are hard to test for

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
    - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties
    - Without actually running the program!

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- 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!

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyz… such defects is **not feasible** (cf.
- We want to learn about "**all pos**… particular properties
  - Without actually running t…
  - Bonus: we don't need test ca…

This is especially true for certain kinds of hard-to-test-for defects that might not be apparent even if you do exercise them, such as **resource leaks**

What does static analysis do well?

What does static analysis do well?

- Defects that result from inconsistently following **simple**, mechanical design **rules**

What does static analysis do well?

- 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: two threads, one variable

What does static analysis do well?

- Defects that result from inconsistent mechanical design **rules**
  - Security: buffer overruns, input
  - Memory safety: null pointers, in
  - Resource leaks: memory, OS resources
  - API Protocols: device drivers, GUI frameworks
  - Exceptions: arithmetic, library, user-defined
  - Encapsulation: internal data, private functions
  - Data races: two threads, one variable

There are **rules** for doing each of these things **correctly**, and a static analysis can automate those rules.

What is a static analysis?

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program
- an **abstraction**, in this context, is a **selective representation** of the program that is simpler to analyze

What is a static analysis?

Definition: *static analysis* is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program
- an **abstraction**, in this context, is a **selective representation** of the program that is simpler to analyze
  - **key idea**: the abstraction will have fewer states to explore
    - hopefully, many fewer!

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

- **Abstraction**

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

- **Abstraction**
    - Capture semantically-relevant details

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

- **Abstraction**
  - Capture semantically-relevant details
  - Elide other details

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

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

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

- **Abstraction**
  - Capture semantically-relevant details
  - Elide other details
  - Handle “I don't know”: think about developers
- Programs **As Data**

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

- **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

Key ideas in static analysis design

When thinking about static analyses, **two key ideas** to keep in mind:

- **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)

Treating programs as data: three ways

Treating programs as data: three ways

#1: treat the program **as a string**

Treating programs as data: three ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties

Treating programs as data: three ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties
  - for example, checking if a program contains the text “foo”

Treating programs as data: three ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties
    - for example, checking if a program contains the text “foo”
- key downside: cannot use the program’s **semantics**

Treating programs as data: three ways

#1: treat the program **as a string**

- allows us to easily decide syntactic properties
  - for example, checking if a program contains the text “foo”
- key downside: cannot use the program’s **semantics**
  - ***semantics*** is a fancy word for “meaning”

Treating programs as data: three ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties
  - for example, checking if a program contains the text “foo”
- key downside: cannot use the program’s **semantics**
  - ***semantics*** is a fancy word for “meaning”
  - semantics are relevant for properties related to **context** - that is, where the question to be decided depends on the rest of the program

Treating programs as data: three ways

#2: treat the program **as a tree**

Treating programs as data: three ways

#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

Treating programs as data: three ways

#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

- usually produced by a **parser**

Treating programs as data: three ways

#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

- usually produced by a **parser**
- nodes in the tree represent syntactic constructs

Treating programs as data: three ways

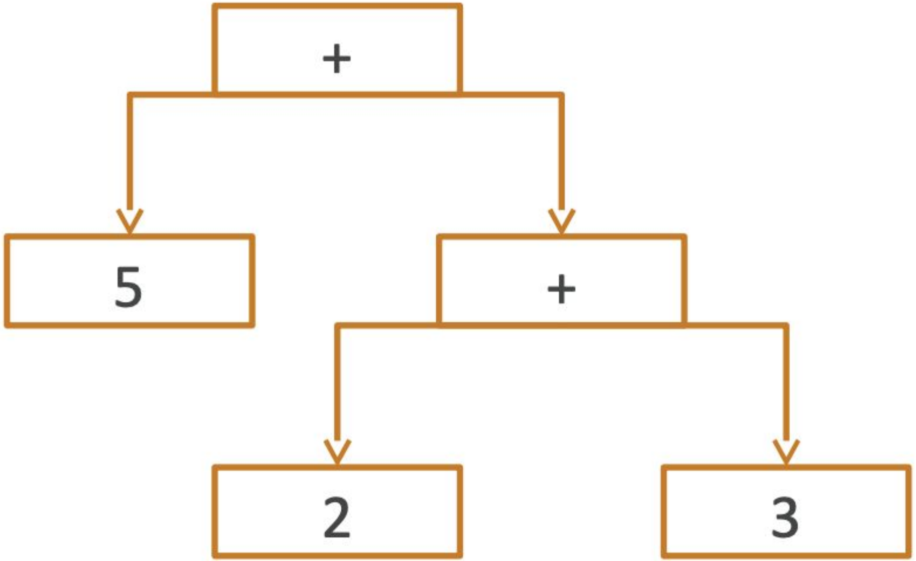
#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

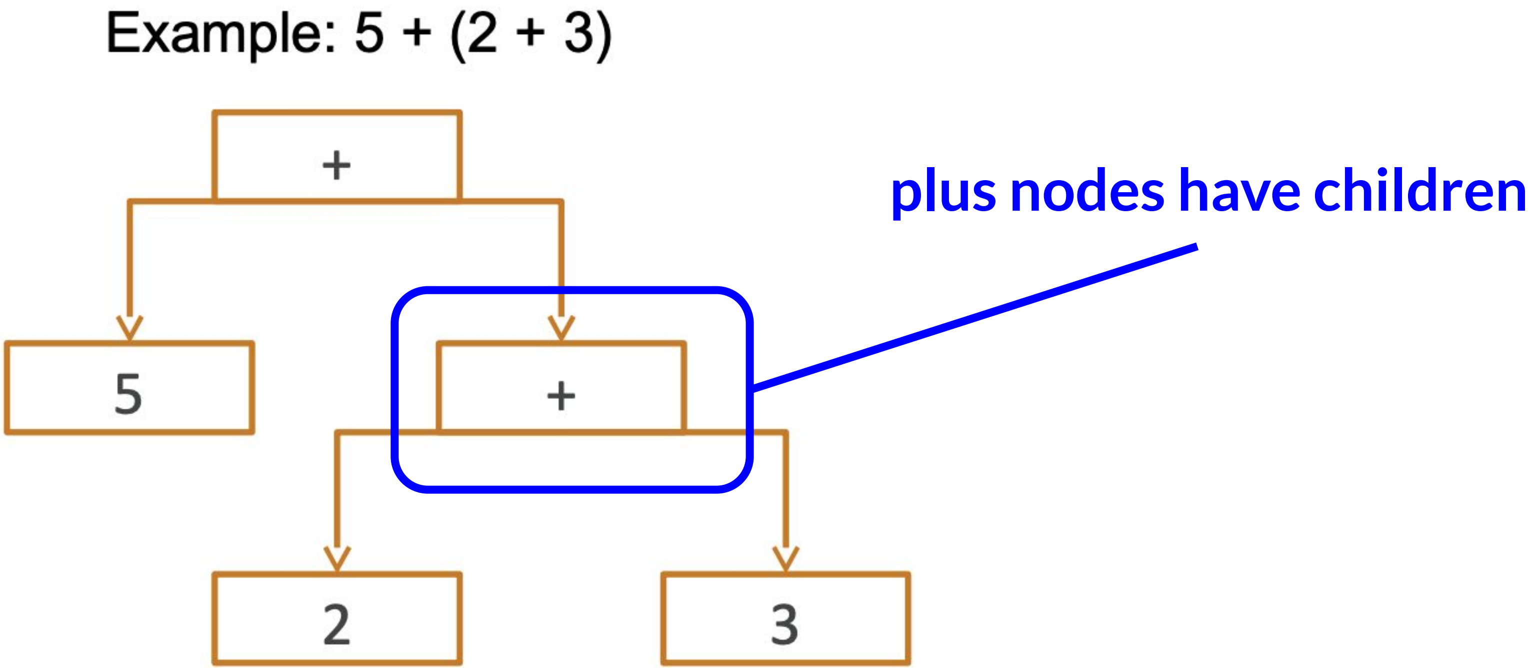
- usually produced by a **parser**
- nodes in the tree represent syntactic constructs
    - parent-child relationships in the AST represent **compound expressions** in the source code (e.g., a "plus node" might have two children: the left and right side expressions)

Treating programs as data: AST example

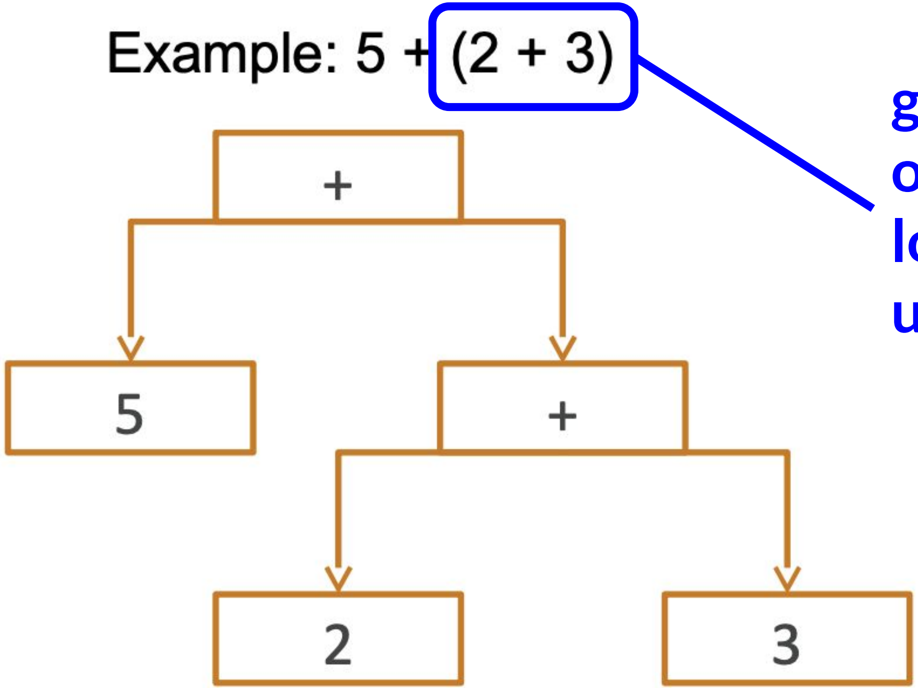
Example: 5 + (2 + 3)



Treating programs as data: AST example



Treating programs as data: AST example



grouping parentheses and other disambiguation is no longer necessary (AST is unambiguous, unlike text)

Treating programs as data: three ways

#3: treat the program **as a graph**

Definition: a ***control flow graph*** (or **CFG**) is a representation, using graph notation, of all paths that might be traversed through a program during its execution

Treating programs as data: three ways

#3: treat the program **as a graph**

Definition: a ***control flow graph*** (or ***CFG***) is a representation, using graph notation, of all paths that might be traversed through a program during its execution

- this is the internal representation used by most static analysis tools

Treating programs as data: three ways

CFG example on the whiteboard

Dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program

Dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
    - Dataflow analysis is the **core idea** behind most static analyses

Dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
    - Dataflow analysis is the **core idea** behind most static analyses
- We first abstract the program to an AST or CFG

Dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
  - Dataflow analysis is the **core idea** behind most static analyses
- 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 ***abstract values***

Dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
    - Dataflow analysis is the **core idea** behind most static analyses
- 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 ***abstract values***
- We finally give **rules** for computing those abstract values

Dataflow analysis

- ***Dataflow analysis*** is a technique for gathering information about the possible set of values calculated at various points in a program
    - Dataflow analysis is the **core idea** behind most static analyses
- 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 ***abstract values***
- We finally give **rules** for computing those abstract values
    - Dataflow analyses take programs as input

Example dataflow analyses

Throughout this lecture, we'll use two examples of dataflow analyses:

Example dataflow analyses

Throughout this lecture, we'll use two examples of dataflow analyses:

1. an analysis for finding **definite** null-pointer dereferences

   "Whenever execution reaches *ptr at program location L, ptr will be NULL"

Example dataflow analyses

Throughout this lecture, we'll use two examples of dataflow analyses:

1. an analysis for finding **definite** null-pointer dereferences

   "Whenever execution reaches *ptr at program location L, ptr will be NULL"

2. an analysis for finding **potential** secure information leaks

   "We read in a secret string at location L, but there is a possible future public use of it"

Definite vs potential

A “**definite**” null-pointer dereference exists if and only the pointer is NULL on **every** program execution

A “**potential**” secure information leak exists if and only if the secure information leaks on **any** program execution

Definite vs potential

A "**definite**" null-pointer dereference exists if and only the pointer is NULL on **every** program execution

A "**potential**" secure information leak exists if and only if the secure information leaks on **any** program execution

The use of "**every**" and "**any**" here guarantee that we must reason about **all paths** through the program!

Definite vs potential = false positives vs negatives

Did a tool **warn** us about X?	Can X **actually** happen? YES	Can X **actually** happen? NO
YES	**True positive**	**False positive**
NO	**False negative**	**True negative**

Definite vs potential = false positives vs negatives

Can X **actually** happen?

Did a tool **warn** us about X?	YES	NO
YES	**True positive**	**False positive**
NO	**False negative**	**True negative**

checking for "potential" properties usually comes with false positives

Definite vs potential = false positives vs negatives

Can X **actually** happen?

Did a tool **warn** us about X?	YES	NO
YES	True positive	False positive
NO	False negative	True negative

checking for "potential" properties usually comes with false positives

checking for "definite" properties usually comes with false negatives

Definite vs potential = false positives vs negatives

Can X **actually** happen?

Did a … t X?	YES	NO
YES	True positive	**False positive**
NO	**False negative**	True negative

Useful analyses in practice often have **both** false positives and false negatives.

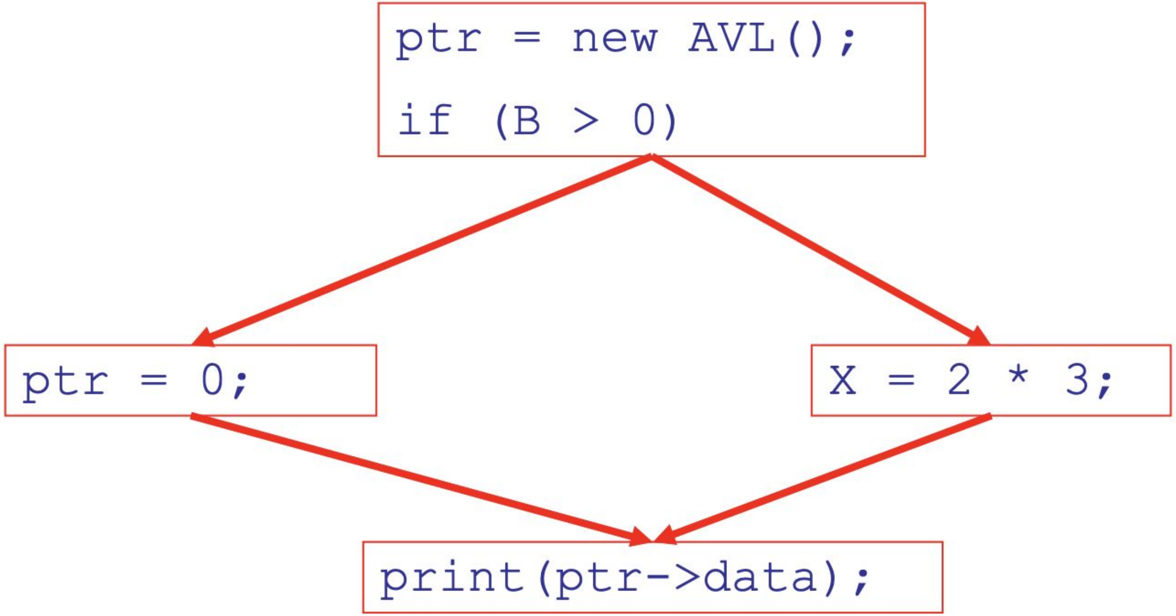
checking for "potential" properties usually comes with false positives

checking for "definite" properties usually comes with false negatives

Null-pointer analysis example

Null-pointer analysis example

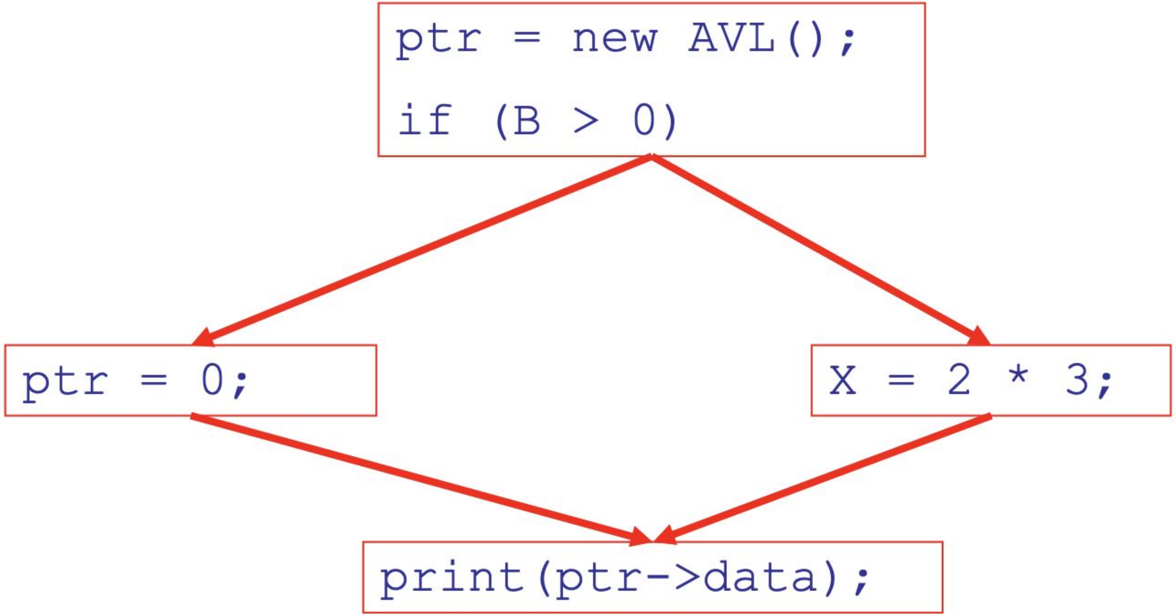
Question: is `ptr` ***always*** null when it is dereferenced?



Null-pointer analys

Q: what does “`ptr` always null” actually require about assignments to `ptr`?

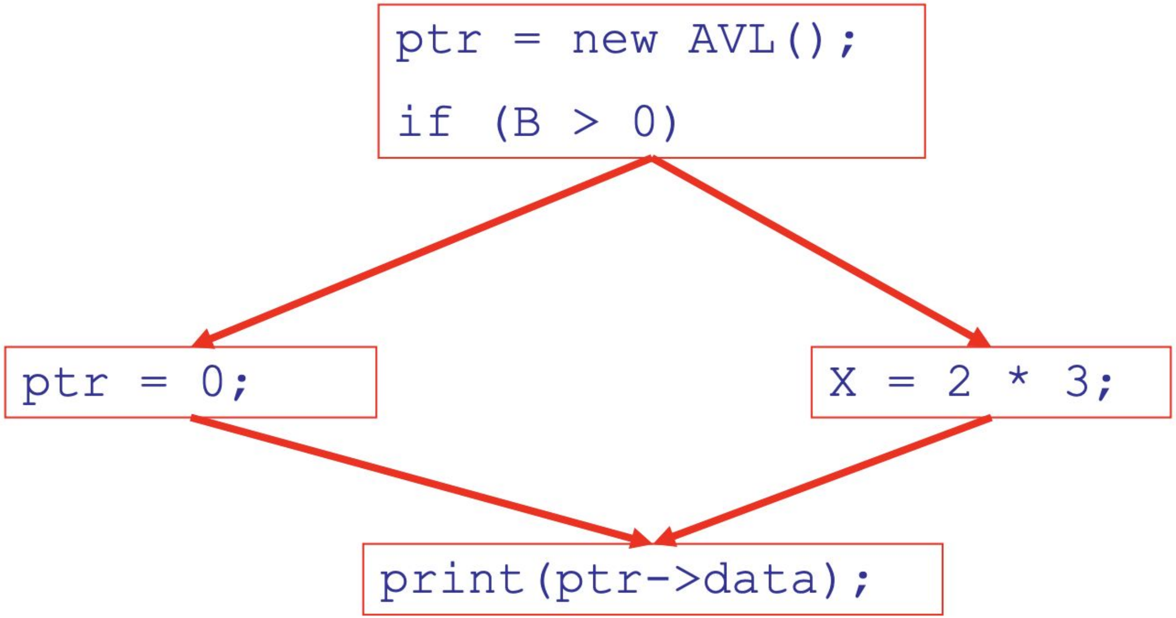
Question: is `ptr` ***always*** null when it is dereferenced?



Null-pointer analys

Q: what does “`ptr` always null” actually require about assignments to `ptr`?
A: on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

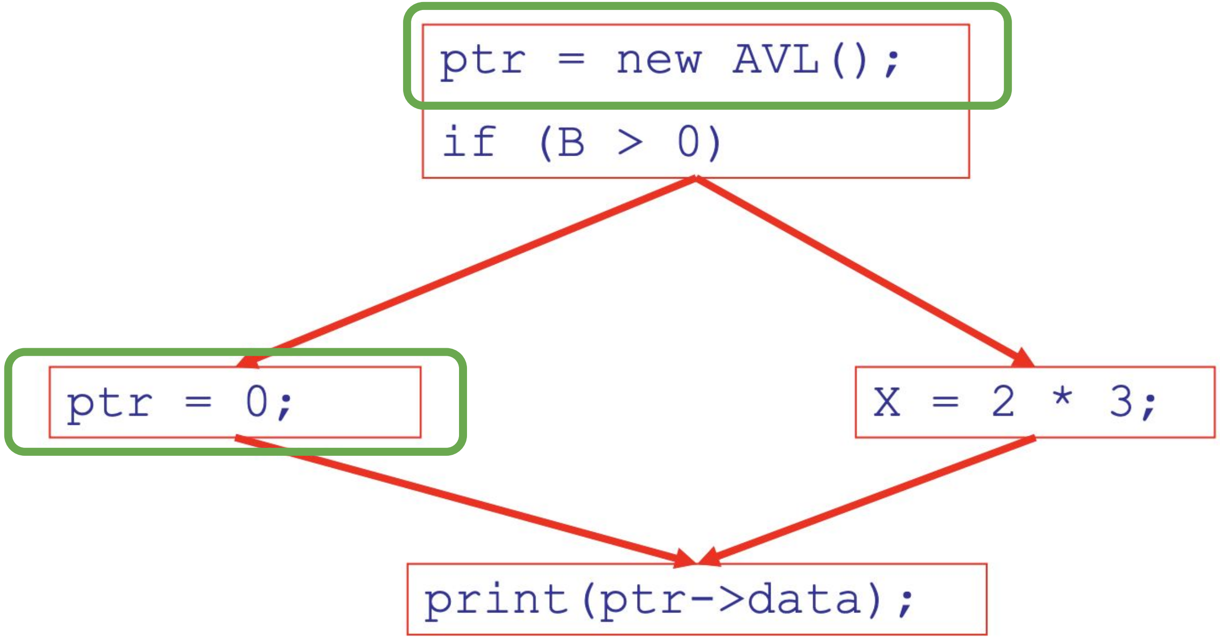
Question: is `ptr` ***always*** null when it is dereferenced?



Null-pointer analysis

Q: what does "`ptr` always null" actually require about assignments to `ptr`?
A: on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

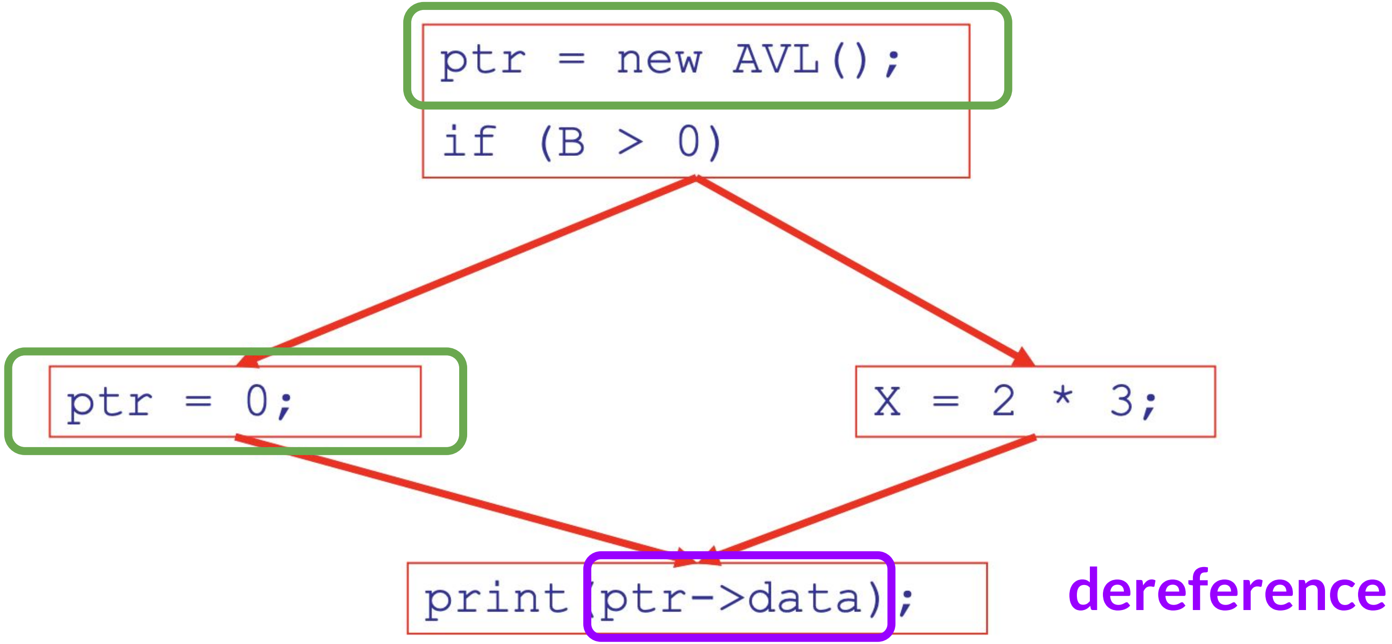
Question: is `ptr` ***always*** null when it is dereferenced?



Null-pointer analys

Q: what does “`ptr` always null” actually require about assignments to `ptr`?
A: on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

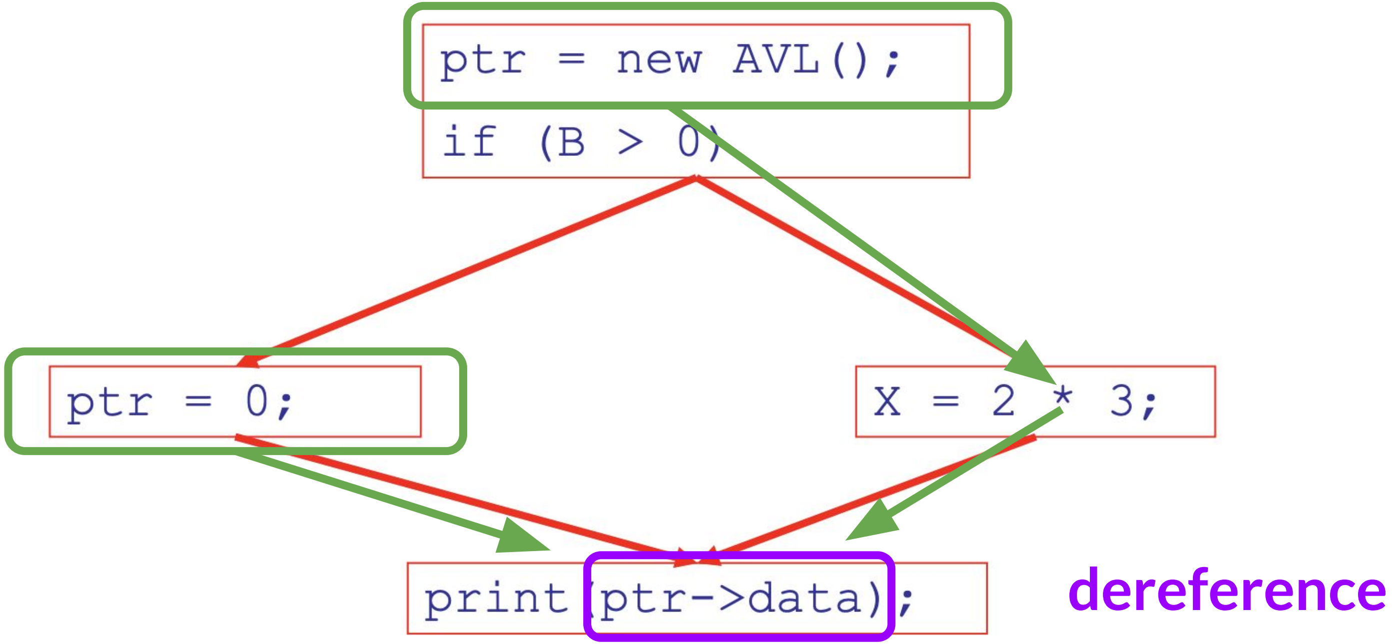
Question: is `ptr` ***always*** null



Null-pointer analys

Q: what does "`ptr` always null" actually require about assignments to `ptr`?
A: on all paths, the **last assignment** to `ptr` must have been null (= 0 in C)

Question: is `ptr` ***always*** null when it is dereferenced?



Common traits of dataflow analysis

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for “definite” analyses: **for all** executions, is P true at this point?
  - for “potential” analyses: does **there exist** an execution for which P is true at this point?

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution

  $\forall$

  - for "definite" analyses: **for all** executions, is P true at this point?
  - for "potential" analyses: does **there exist** an execution for which P is true at this point?

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for "definite" analyses: **for all** executions, is P true at this point?
  - for "potential" analyses: does **there exist** an execution for which P is true at this point?

$\exists$

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for "definite" analyses: **for all** executions, is P true at this point?
  - for "potential" analyses: does **there exist** an execution for which P is true at this point?
- Knowing P at any specific program point usually requires knowledge of the entire method body

Common traits of dataflow analysis

- The analysis depends on knowing a property P at a particular point in program execution
  - for "definite" analyses: **for all** executions, is P true at this point?
  - for "potential" analyses: does **there exist** an execution for which P is true at this point?
- Knowing P at any specific program point usually requires knowledge of the entire method body
- Property P is typically **undecidable**

Undecidability of program properties

Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:

Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:

"**interesting**" in this context means "not trivial", i.e., not uniformly true or false for all programs

Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
    - Does the program halt on all (some) inputs?
        - This is called the **halting problem**

Undecidability of program properties

- ***Rice's Theorem***: All 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?

Undecidability of program properties

- *Rice's Theorem*: All 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

Undecidability of program properties

- ***Rice's Theorem***: All 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.

Undecidability of program properties

- *Rice's Theorem*: All 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

Undecidability of program properties

- *Rice's Theorem*: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.
    - Take function H and find out if it halts by testing function `F(x) = { H(x); return 1; }` to see if it has a positive result
    - Contradiction!

Undecidability of program properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program hal
    - This is called the **h**
  - Is the result of a funct
    - *Assume* we can ans
    - Oops: We can now
    - Take function H an
      `F(x) = { H(x); return 1; }` to see if it has a positive result
    - Contradiction!

Rice's theorem caveats:

- only applies to **semantic** properties (syntactic properties are decidable)
- "programs" only includes programs **with loops**

Loops

- Almost every important program has a loop
    - Often based on user input

Loops

- Almost every important program has a loop
  - Often based on user input
- An algorithm always terminates (remember your theory class!)
  - So a dataflow analysis algorithm must terminate even if the input program loops

Loops

- Almost every important program has a loop
  - Often based on user input
- An algorithm always terminates (remember your theory class!)
  - So a dataflow analysis algorithm must terminate even if the input program loops
- This is one source of imprecision
  - "imprecision" = "not always getting the right answer"
  - Suppose you dereference the null pointer on the 500th iteration but we only analyze 499 iterations

Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable

Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(

Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(
- **Solution**: when in doubt, allow the analysis to answer "I don't know"

Conservative program analysis

- Because our analysis must run on a computer, we need the analysis itself to be decidable
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(
- **Solution**: when in doubt, allow the analysis to answer "I don't know"
    - this is called ***conservative*** analysis

Conservative program analysis

- It's **always correct** to say "I don't know"

Conservative program analysis

- It’s **always correct** to say “I don’t know”
  - **key challenge** in program analysis: say “I don’t know” as rarely as possible

Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

Definition: a ***sound*** program analysis has no false negatives

Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

Definition: a ***sound*** program analysis has no false negatives

- always answers "I don't know" if there is a **potential** bug

Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

Definition: a ***sound*** program analysis has no false negatives

- always answers "I don't know" if there is a **potential** bug

Definition: a ***complete*** program analysis has no false positives

Conservative program analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

Definition: a ***sound*** program analysis has no false negatives

- always answers "I don't know" if there is a **potential** bug

Definition: a ***complete*** program analysis has no false positives

- always answers "I don't know" if there isn't a **definite** bug

Soundness vs completeness

- Building a sound or complete analysis is **easy**

Soundness vs completeness

- Building a sound or complete analysis is **easy**
    - trivially sound analysis: report a bug **on every line**

Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug

Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug
- Building a sound and ***precise*** (= "few false positives") analysis or a complete analysis with ***high recall*** (= "few false negatives") is **very hard**

Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug
- Building a sound and ***precise*** (= "few false positives") analysis or a complete analysis with ***high recall*** (= "few false negatives") is **very hard**
  - "sound and precise" analyses are my research area :)

Soundness vs completeness

- Building a sound or complete analysis is **easy**
  - trivially sound analysis: report a bug **on every line**
  - trivially complete analysis: **never** report a bug
- Building a sound and ***precise*** (= “few false positives”) analysis or a complete analysis with ***high recall*** (= “few false negatives”) is **very hard**
  - “sound and precise” analyses are my research area :)
  - also relevant in practice: “fast”, “easy to use”, etc.

Soundness vs completeness

- Which is more important: **soundness** or **completeness**?

Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!

Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.

Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.
    - "I don't know" = don't issue a warning

Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.
    - “I don’t know” = don’t issue a warning
  - Are you writing a bug-finding analysis for aircraft autopilots? False negatives cause crashes, so choose soundness.

Soundness vs completeness

- Which is more important: **soundness** or **completeness**?
- Answer: **it depends**!
  - Are you writing a bug-finding analysis for websites that show pictures of cats? False positives waste time, so choose completeness.
    - “I don’t know” = don’t issue a warning
  - Are you writing a bug-finding analysis for aircraft autopilots? False negatives cause crashes, so choose soundness.
    - “I don’t know” = do issue a warning

Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete

Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives

Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound

Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound
    - remember: type systems are just another static analysis

Soundness vs completeness

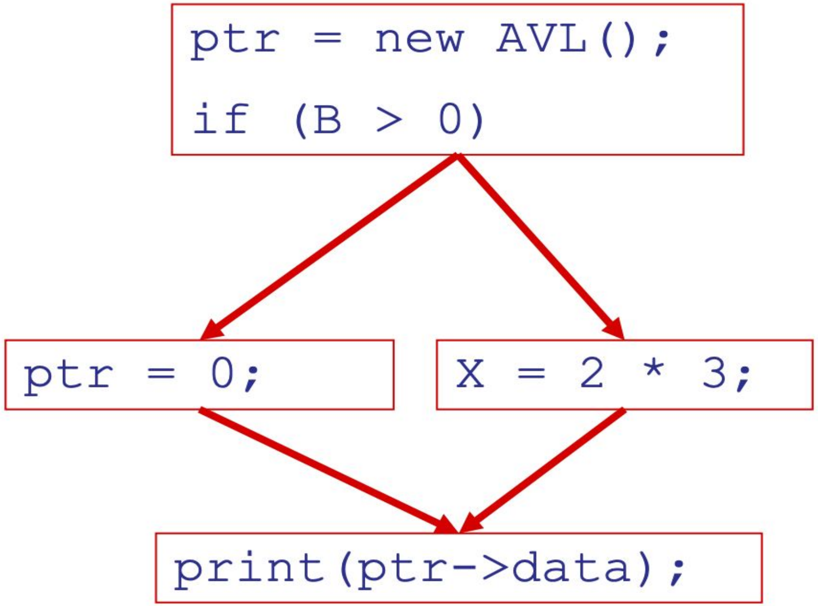
- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound
    - remember: type systems are just another static analysis
  - few complete analyses exist in practice

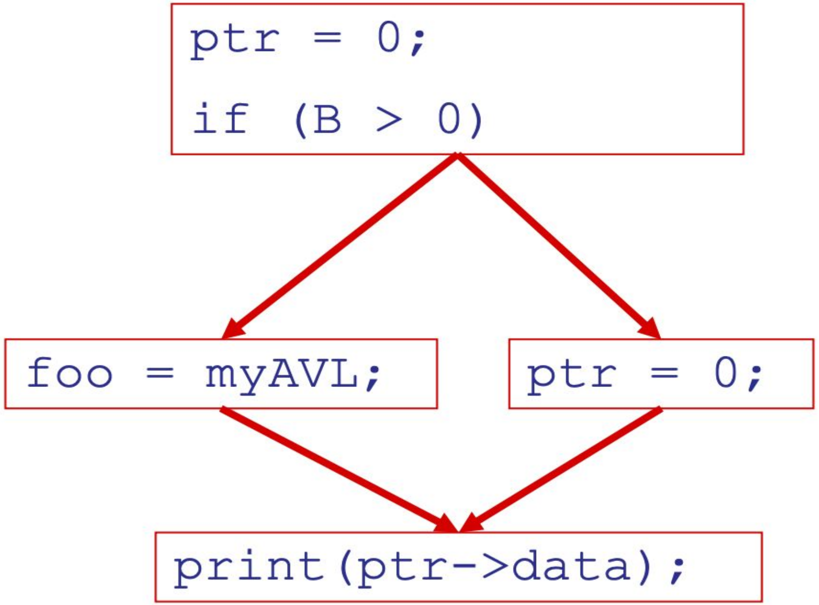
Soundness vs completeness

- In practice, most static analyses are **neither** sound nor complete
  - e.g., FindBugs from today's reading has both false positives and false negatives
  - most common exception: most **type systems** are sound
    - remember: type systems are just another static analysis
  - few complete analyses exist in practice
    - theory is underdeveloped, but another area of active research!

Null-pointer analysis example

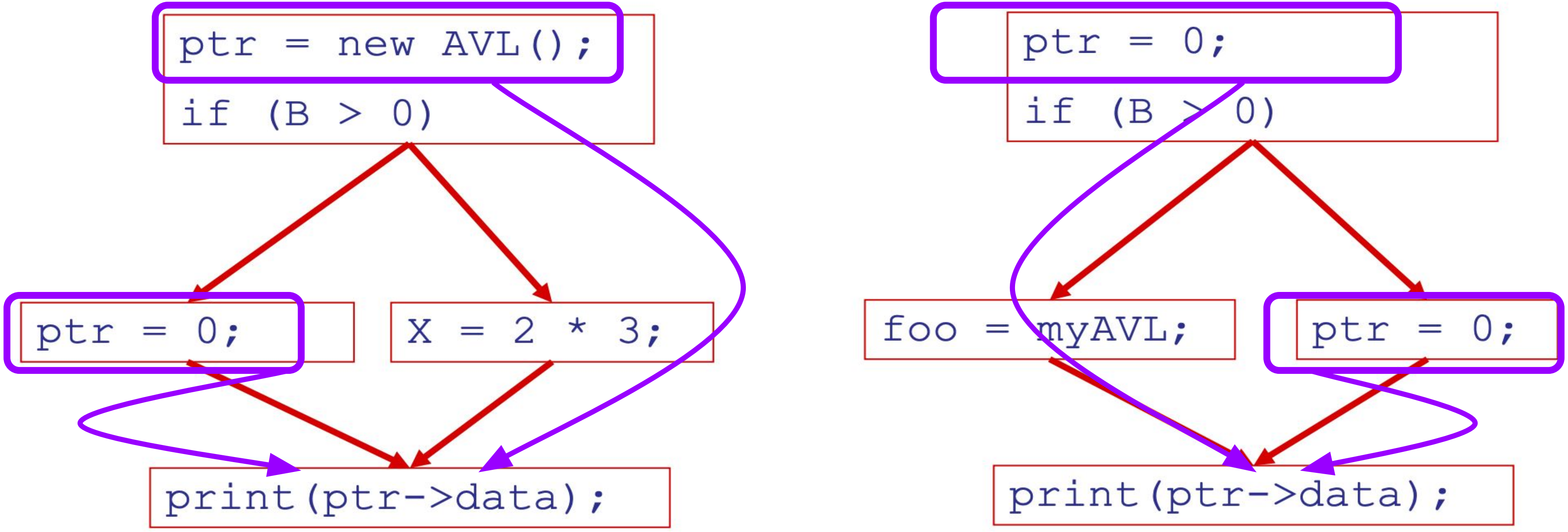
Question: is `ptr` ***always*** null when it is dereferenced?





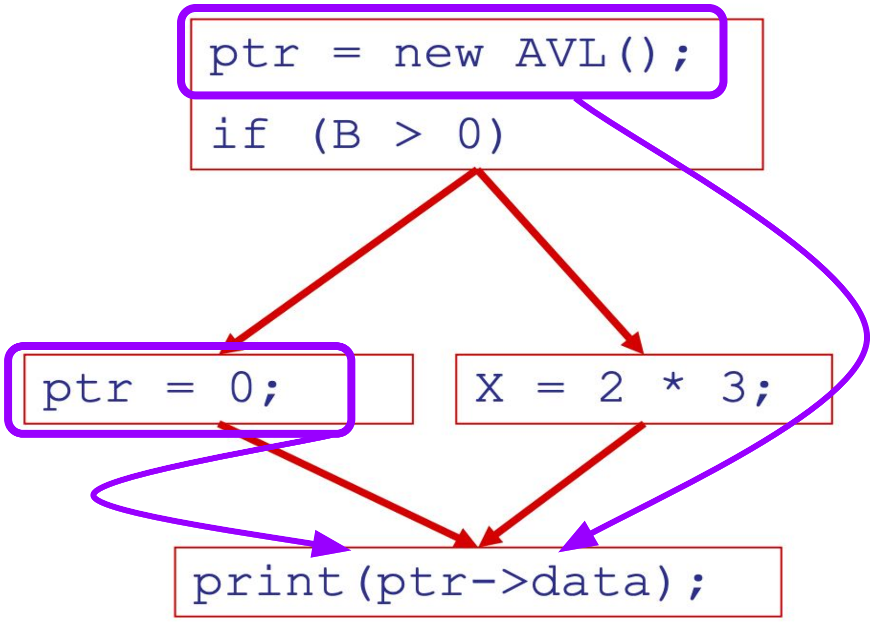
Null-pointer analysis example

Question: is `ptr` ***always*** null when it is dereferenced?

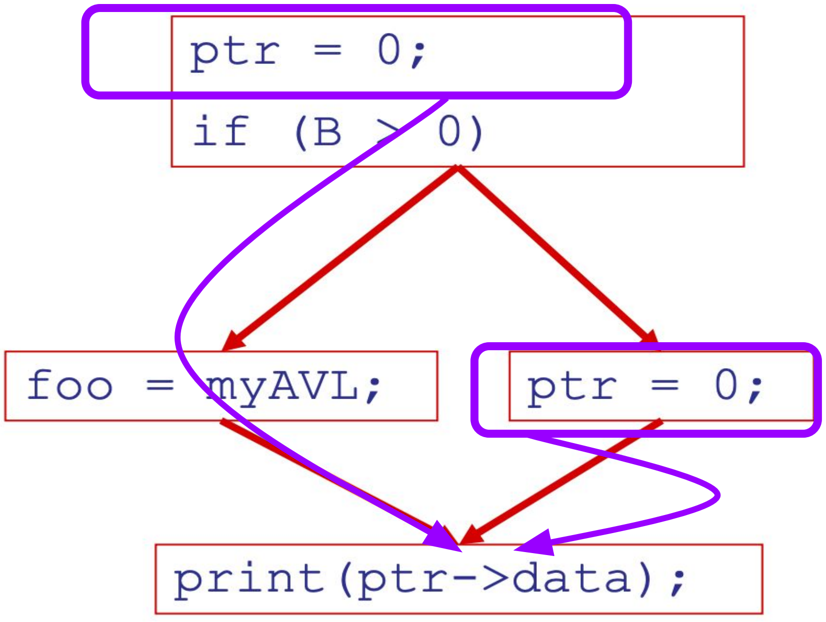


Null-pointer analysis example

Question: is `ptr` ***always*** null when it is dereferenced?



NO: only sometimes null



YES: always null

Null-pointer analysis example: abstraction

Formalizing our reasoning:

Null-pointer analysis example: abstraction

Formalizing our reasoning:

- We associate one of the following **abstract values** with `ptr` at every program point:

Null-pointer analysis example: abstraction

Formalizing our reasoning:

- We associate one of the following **abstract values** with `ptr` at every program point:
  - T ("top") = "don't know if X is a constant"

Null-pointer analysis example: abstraction

Formalizing our reasoning:

- We associate one of the following **abstract values** with `ptr` at every program point:
  - T ("top") = "don't know if X is a constant"
  - constant *c* = "the last assignment to X was X = c"

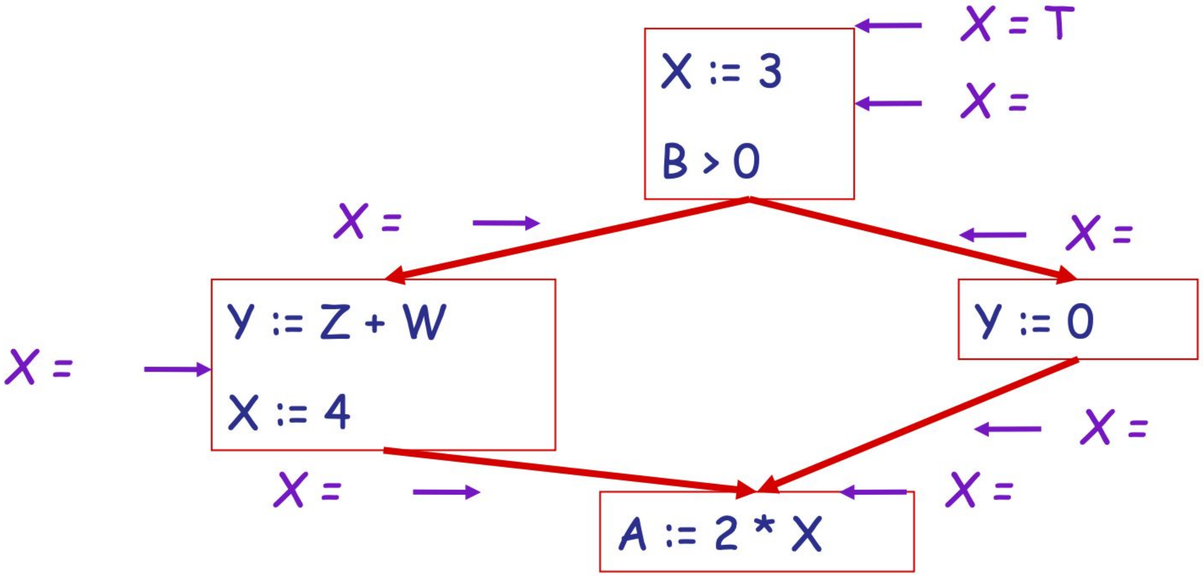
Null-pointer analysis example: abstraction

Formalizing our reasoning:

- We associate one of the following **abstract values** with `ptr` at every program point:
  - T ("top") = "don't know if X is a constant"
  - constant *c* = "the last assignment to X was X = c"
  - ⊥ ("bottom") = "X has no value here"

Null-pointer analysis example: formalized

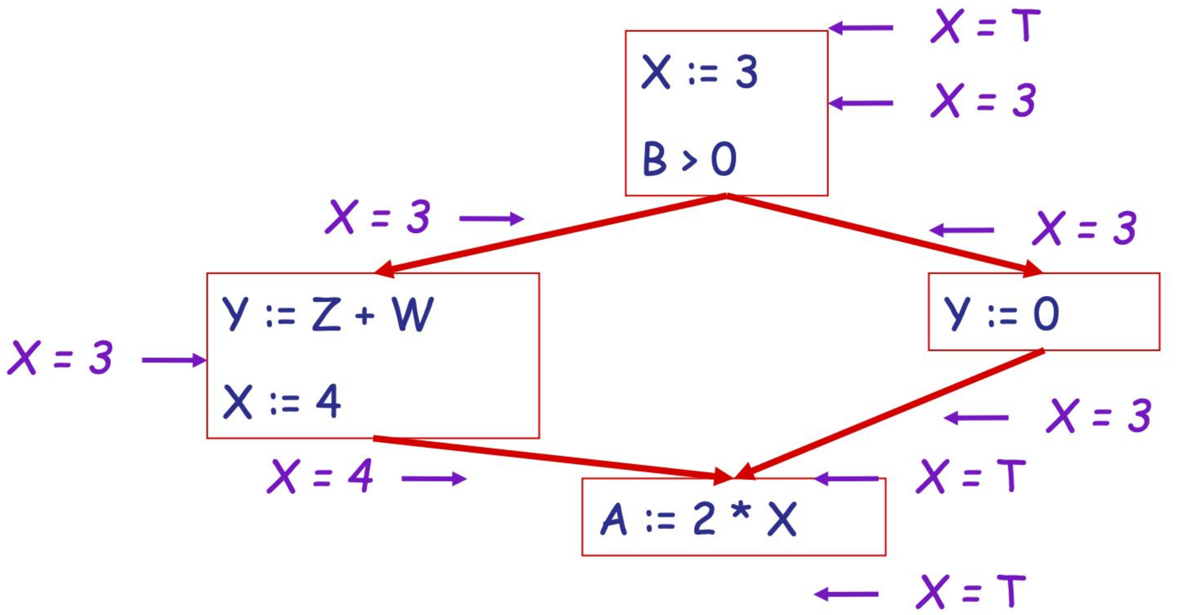
Get out a piece of paper. Fill in these blanks:



Recall:
T = "don't know"
c = constant
⊥ = unreachable

Null-pointer analysis example: formalized

Get out a piece of paper. Fill in these blanks:



Recall:
T = “don’t know”
c = constant
⊥ = unreachable

Issuing warnings

Issuing warnings

- Given analysis information (and a policy about false positives/negatives), it is easy to decide whether or not to issue a warning

Issuing warnings

- 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*

Issuing warnings

- 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!

Issuing warnings

- 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 $x = ?$

Static analysis (2/2?)

- **nullness analysis: how it works**
- secure information flow analysis
- limitations of static analysis
- static analysis in practice
- reading quiz

Key idea behind dataflow analysis

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

Key idea behind dataflow analysis

Explanation:

Key idea behind dataflow analysis

Explanation:

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

Key idea behind dataflow analysis

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:

Key idea behind dataflow analysis

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

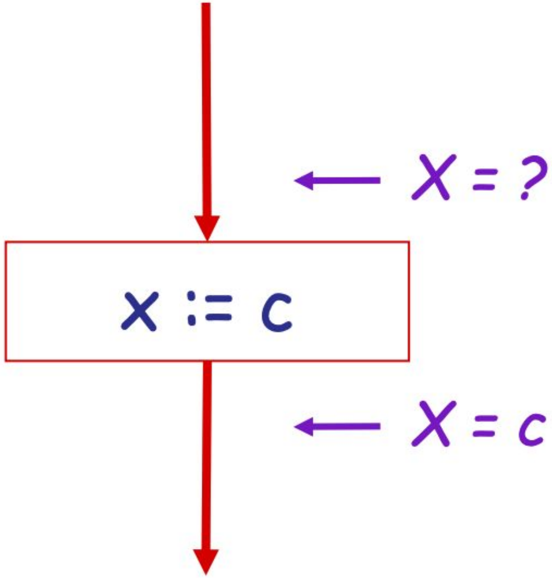
Key idea behind dataflow analysis

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

Definition: a ***transfer function*** expresses the relationship between $C_{in}(x, s)$ and $C_{out}(x, s)$

Transfer functions: rule 1

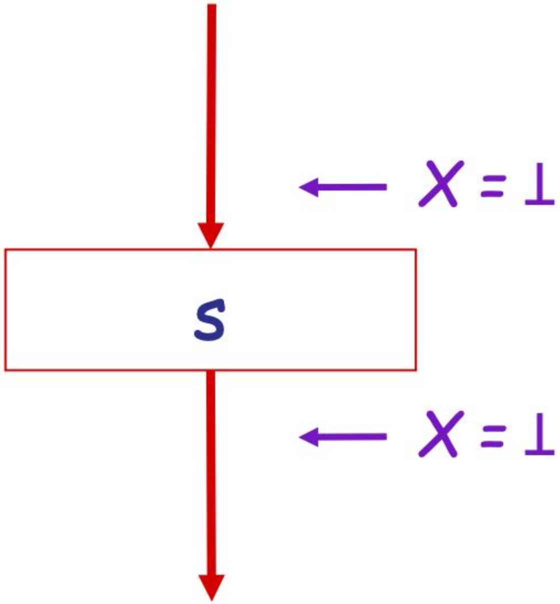


$C_{out}(x, x := c) = c$ if c is a constant

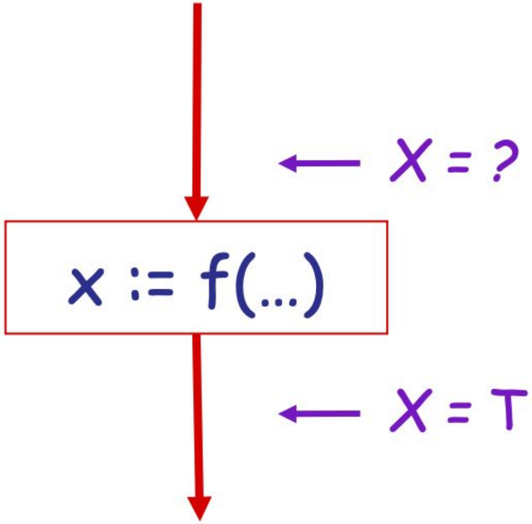
Transfer functions: rule 2

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

Recall □ =
"unreachable code"

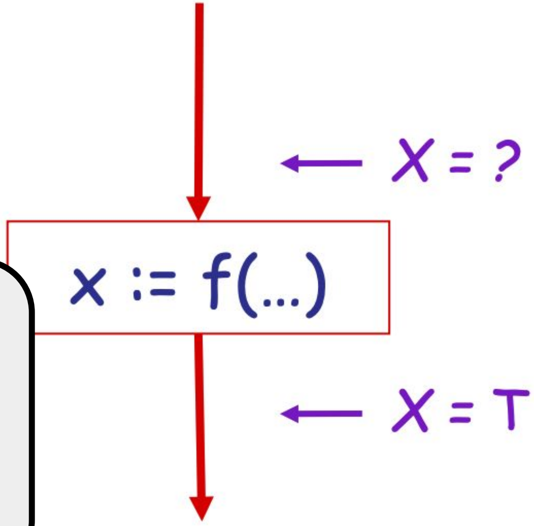


Transfer functions: rule 3



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

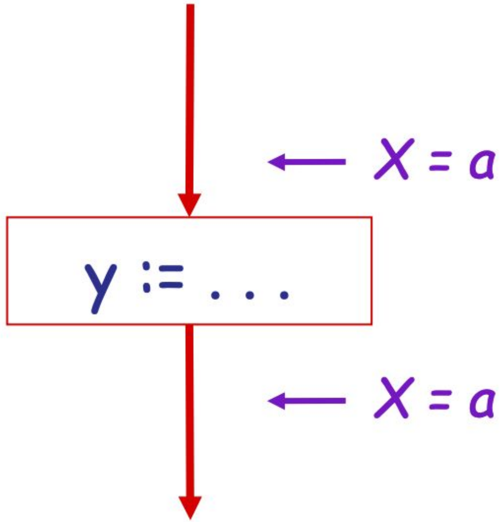
Transfer functions: rule 3



This is a **conservative approximation**! f(...) might always return 0, but we don't even try!

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

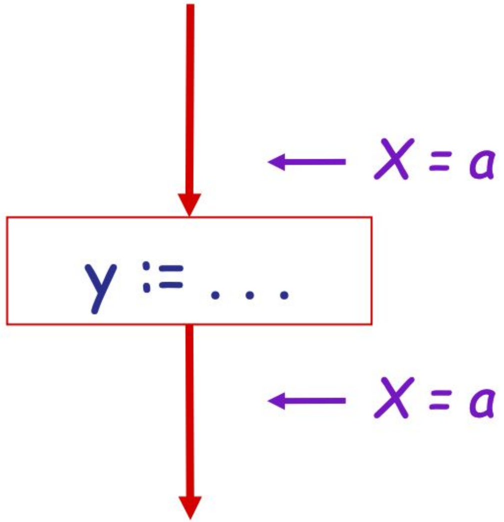
Transfer functions: rule 4



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

Transfer functions: rule 4

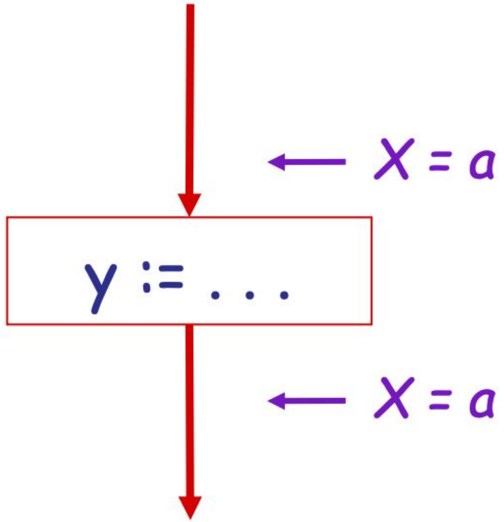
How hard is it to check if x ≠ y on all executions?



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

Transfer functions: rule 4

How hard is it to check if x ≠ y on all executions? (oh no)



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

Propagation between statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement

Propagation between statements

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

Propagation between statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements
- We also need rules relating the *out* of one statement to the *in* of the successor statement

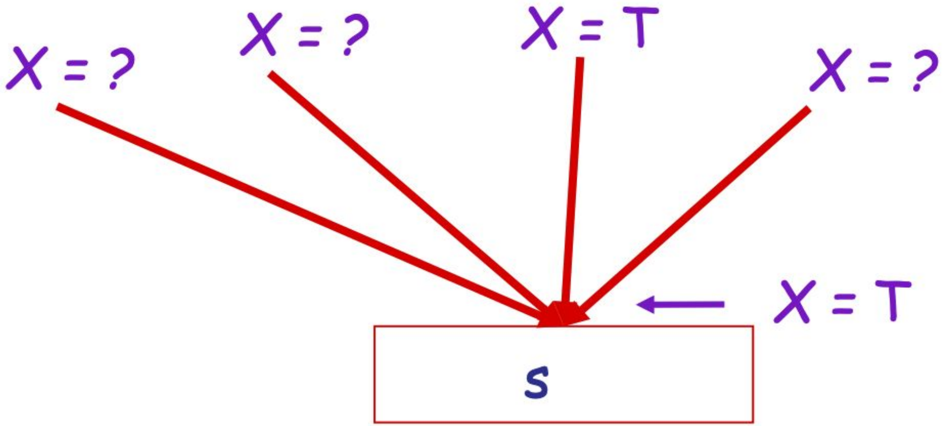
Propagation between statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements
- We also need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information **forward** along paths

Propagation between statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements
- We also 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 $p_1, \ldots, p_n$

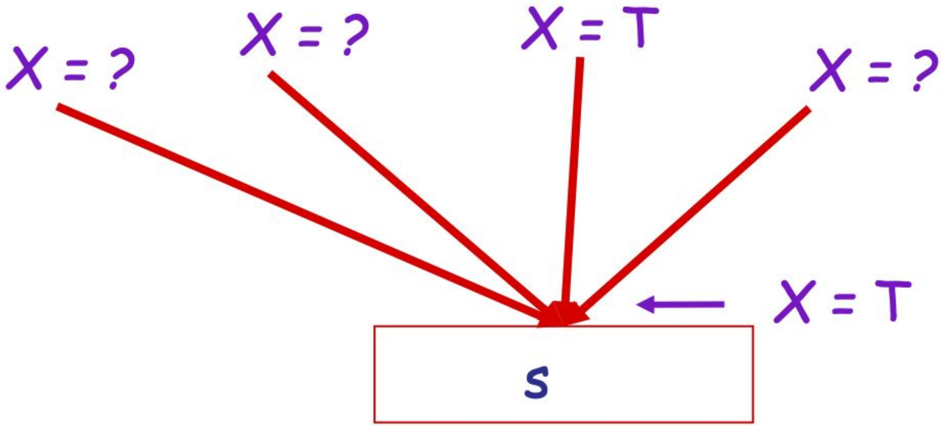
Transfer functions: rule 5



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

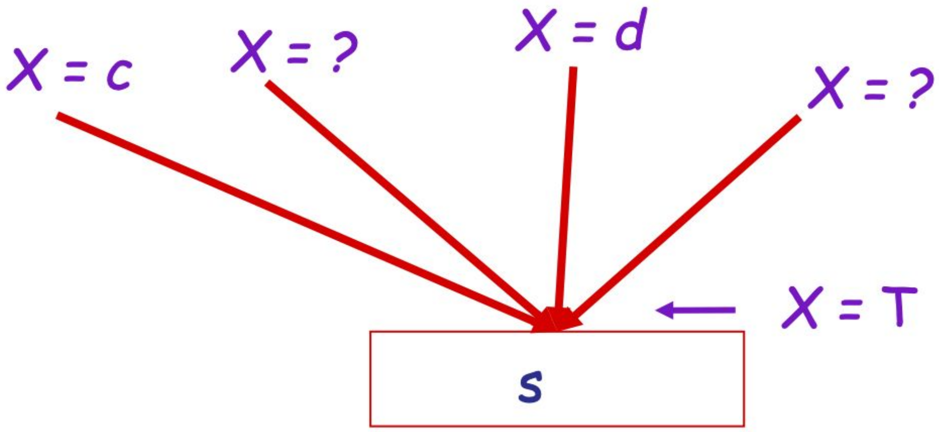
Transfer functions: rule 5

If there's **any** path on which we don't know, then we don't know at all



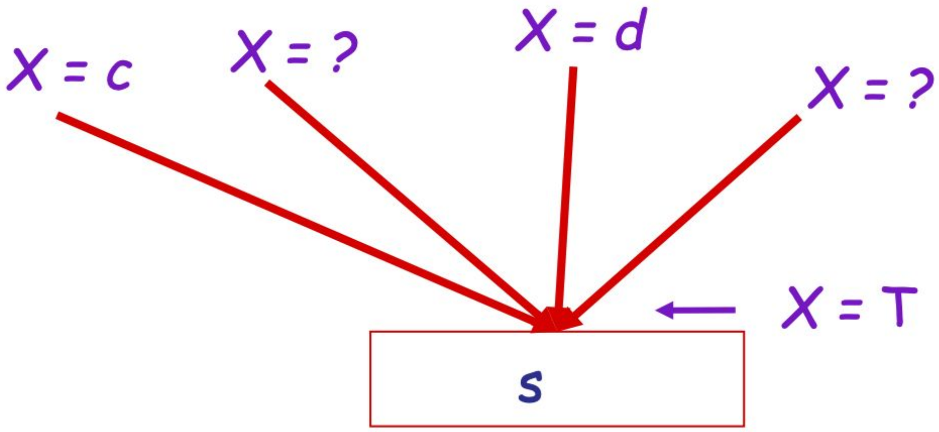
if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

Transfer functions: rule 6



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

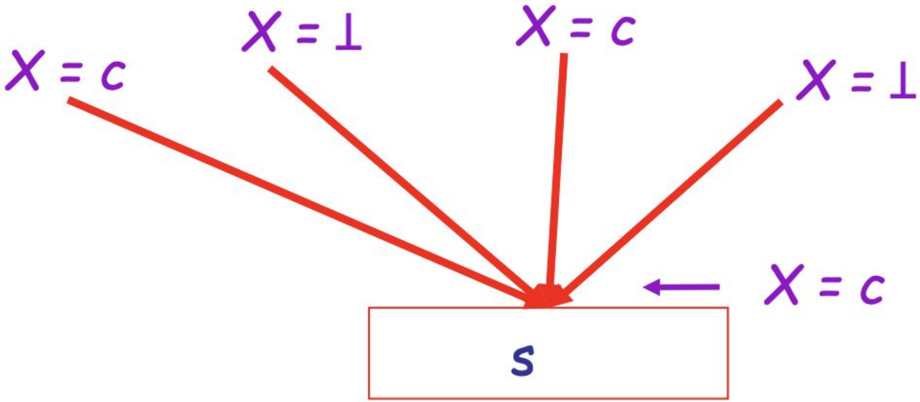
Transfer functions: rule 6



We **don't know** which of the paths a given execution will take (so assume T)

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

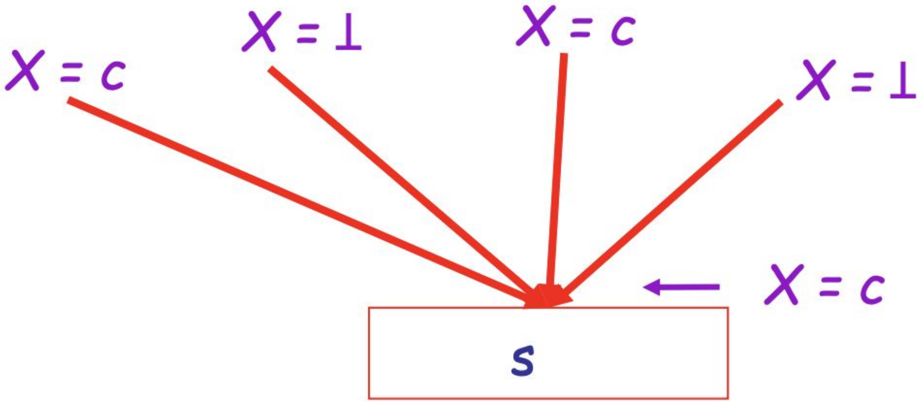
Transfer functions: rule 7



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

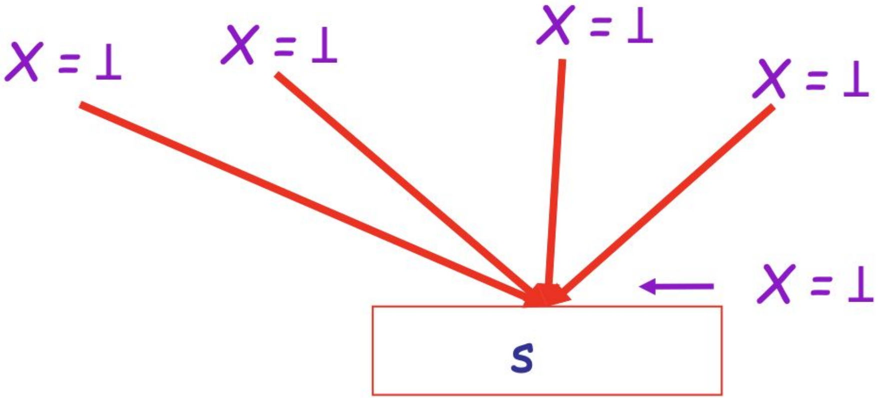
Transfer functions: rule 7

If x has the **same** value (or □) on all input edges, it has that value in s



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

Transfer functions: rule 8



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

A static analysis algorithm

A static analysis algorithm

- For every **entry point** *e* to the program, set $C_{in}(x, e) = T$

A static analysis alg

Definition: an ***entry point*** of a program is any program location *L* for which there exists an execution trace beginning with *L*

- For every **entry point** *e* to the program, set $C_{in}(x, e) = T$

A static analysis algorithm

- For every **entry point** *e* to the program, set $C_{in}(x, e) = T$
    - why top? Top models “we don’t know”, and we don’t know the inputs to the program.

A static analysis algorithm

- For every **entry point** *e* to the program, set $C_{in}(x, e) = T$
  - why top? Top models "we don't know", and we don't know the inputs to the program.
- Set $C_{in}(x, s) = C_{out}(x, s) =$ □ everywhere else

A static analysis algorithm

- For every **entry point** *e* to the program, set $C_{in}(x, e) = T$
  - why top? Top models "we don't know", and we don't know the inputs to the program.
- Set $C_{in}(x, s) = C_{out}(x, s) =$ □ everywhere else
- **Repeat** until all points satisfy rules 1-8:
  - Pick s not satisfying rules 1-8 and update using the appropriate rule

A static analysis alg

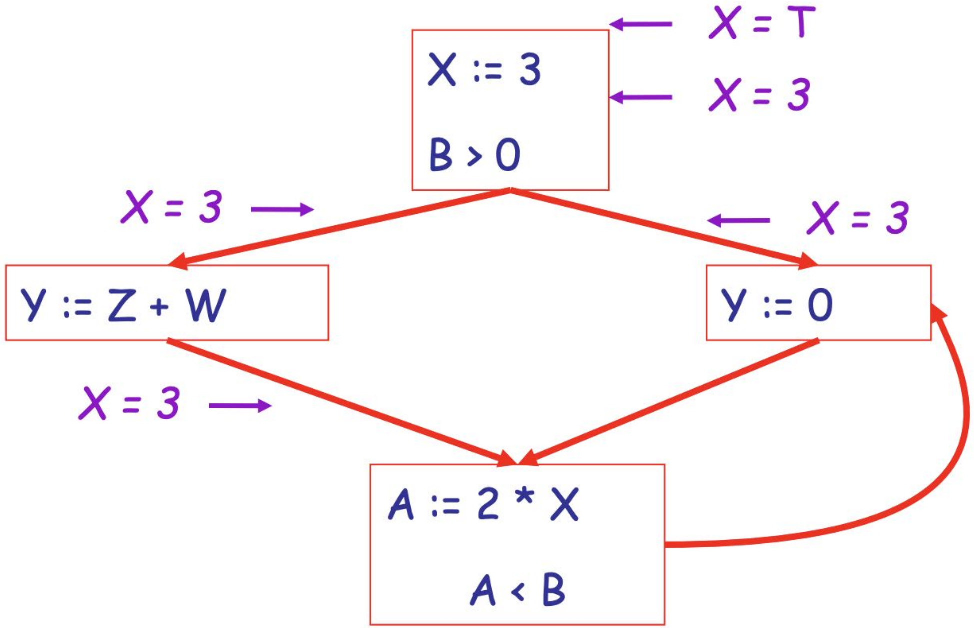
- For every **entry point** *e* to
  - why top? Top models
    inputs to the program
- Set $C_{in}(x, s) = C_{out}(x, s) =$ □
- **Repeat** until all points satisfy rules 1-8:
  - Pick s not satisfying rules 1-8 and update using the appropriate rule

This is a ***fixpoint*** (or ***fixed point***) ***iteration*** algorithm. Such algorithms are characterized by a finite set of rules, which are applied until they "reach fixpoint", which means that applying any rule produces no change.

Why do we need □?

Why do we need □?

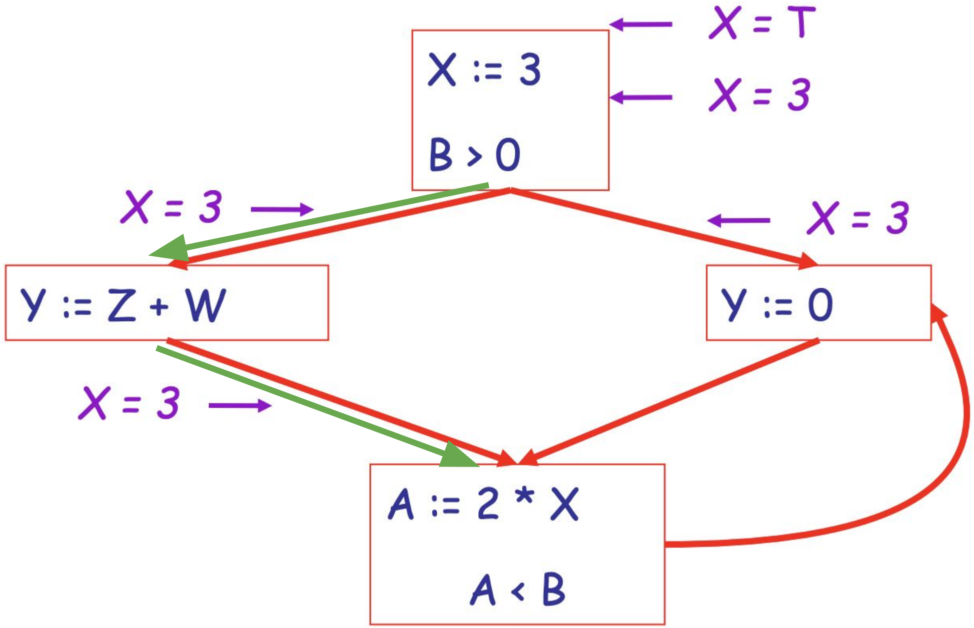
- To understand why we need to set non-entry points to □ initially, consider a program with a loop:



Why do we need □?

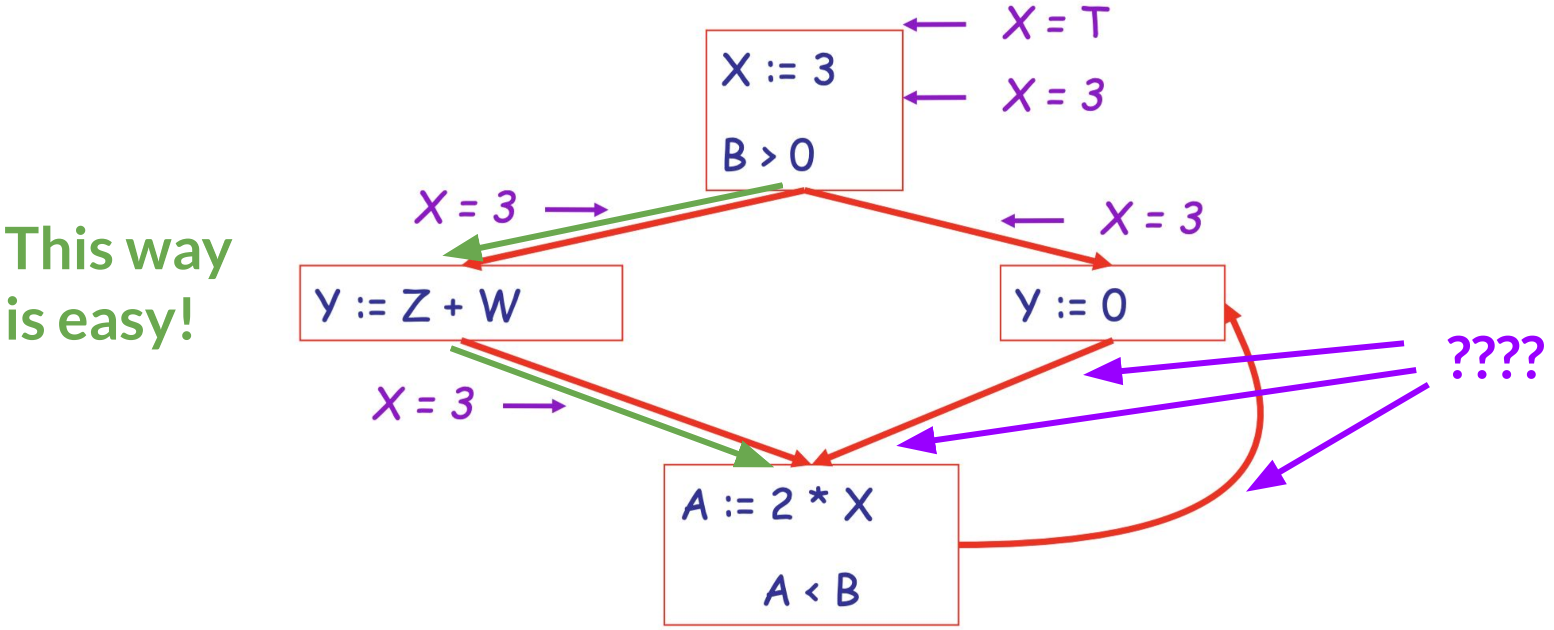
- To understand why we need to set non-entry points to □ initially, consider a program with a loop:

This way is easy!



Why do we need □?

- To understand why we need to set non-entry points to □ initially, consider a program with a loop:



Why do we need □?

- To understand why we need to set non-entry points to □ initially, consider a program with a loop.
- Because of **cycles**, all points must have values at all times during the analysis

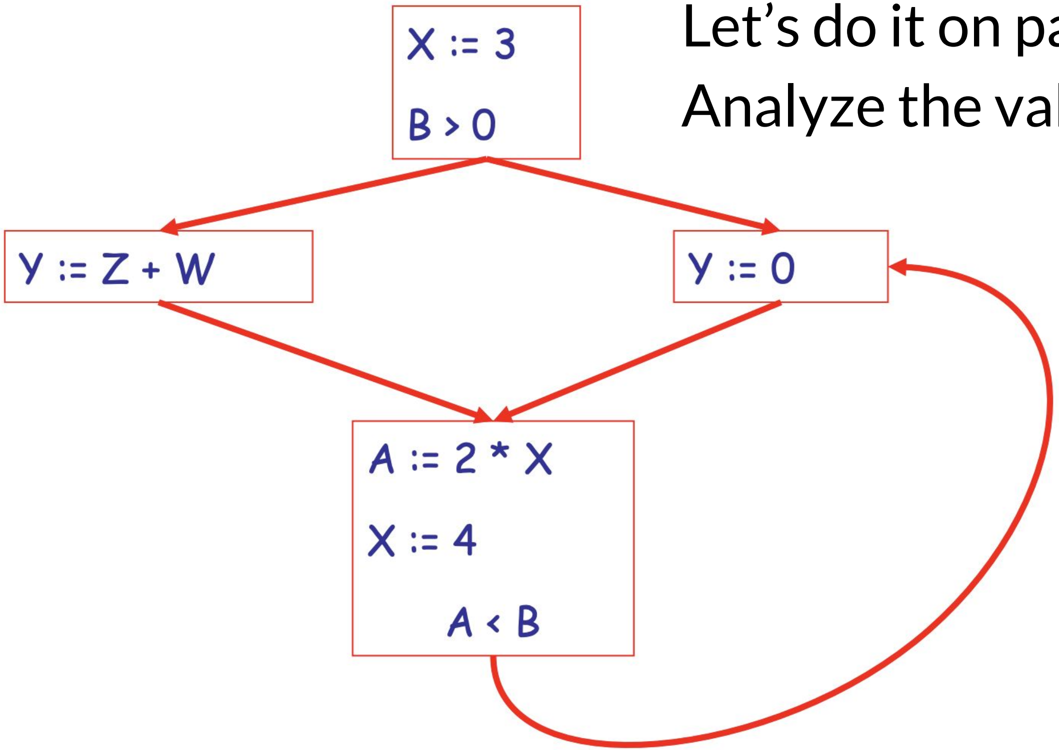
Why do we need □?

- To understand why we need to set non-entry points to □ initially, consider a program with a loop.
- 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**

Why do we need □?

- To understand why we need to set non-entry points to □ initially, consider a program with a loop.
- 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**"

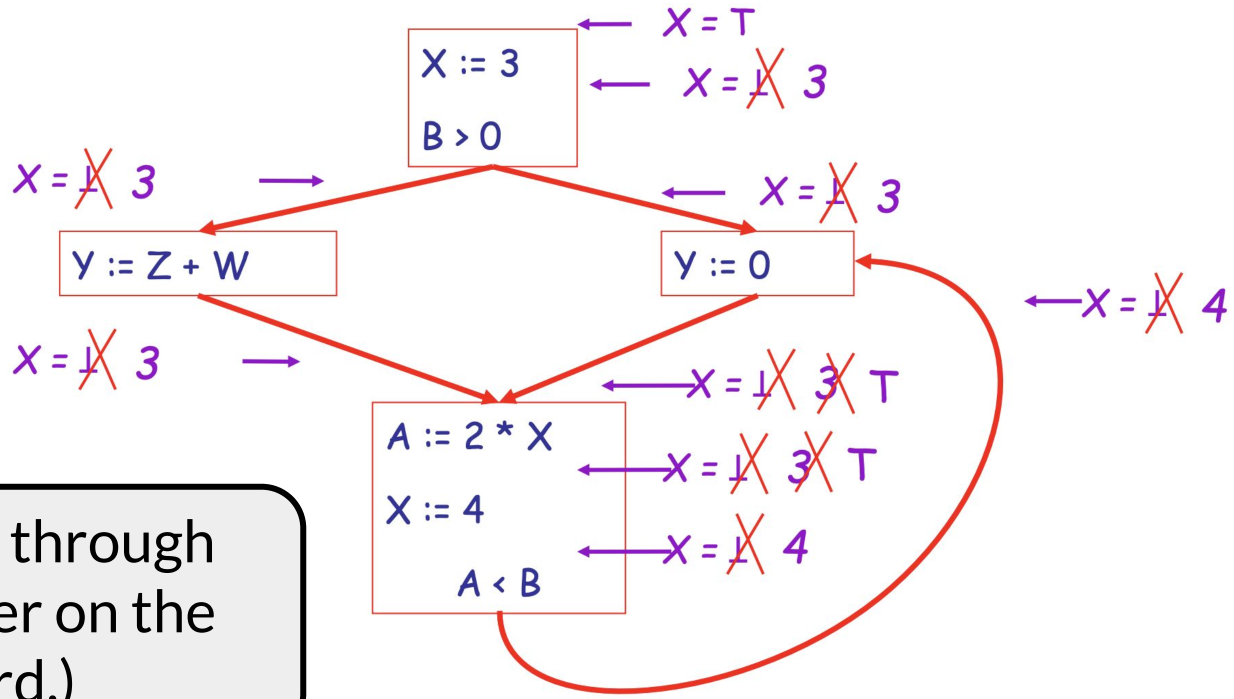
Another example: dealing with loops



Let's do it on paper!

Analyze the value of X.

Another example: dealing with loops



(We went through this answer on the whiteboard.)

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our nullness analysis

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our nullness analysis
  - (Most) locations start as □

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our nullness analysis
  - (Most) locations start as □
  - Locations whose current value is □ might become c or T

Lattices & Orderings

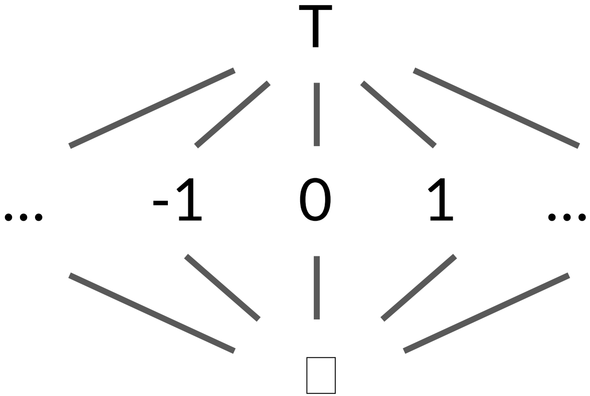
- You may have observed that there is a natural ***order*** to the different abstract values in our nullness analysis
  - (Most) locations start as □
  - Locations whose current value is □ might become c or T
  - Locations whose current value is c might become T
    - but never go back to □!

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our nullness analysis
  - (Most) locations start as □
  - Locations whose current value is □ might become c or T
  - Locations whose current value is c might become T
    - but never go back to □!
  - Locations whose current value is T never change

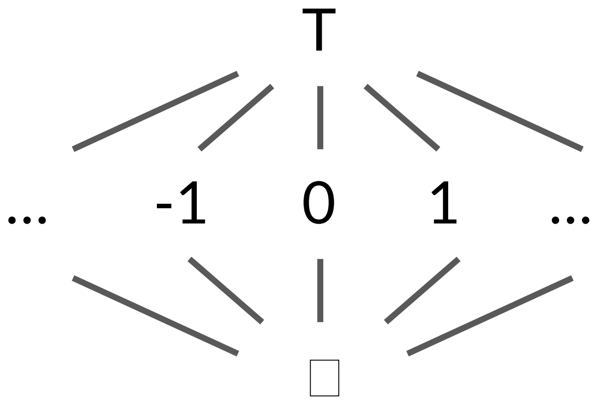
Lattices & Orderings

This structure between values is called a ***lattice***:



Lattices & Orderings

This structure between values is called a ***lattice***:



How to read a lattice:

- abstract values **higher** in the lattice are **more general** (e.g., T is true of more things than 0)
- easy to compute ***least upper bound***: it's the lowest common ancestor of two abstract values

Lattices (continued)

- least upper bound ("**lub**") has useful properties:

Lattices (continued)

- least upper bound ("**lub**") has useful properties:
    - ***monotonicity***: implicitly captures that values only flow in one direction as the analysis progresses

Lattices (continued)

- least upper bound ("**lub**") has useful properties:
  - ***monotonicity***: implicitly captures that values only flow in one direction as the analysis progresses
  - we can rewrite rules 5-8 in our nullness analysis using lub:

$$C_{in}(x, s) = \text{lub}\ \{\ C_{out}(x, p) \mid p \text{ is a predecessor of } s\ \}$$

Lattices (continued)

- least upper bound ("**lub**") has useful properties:
  - *monotonicity*: implicitly captures that values only flow in one direction as the analysis progresses
  - we can rewrite rules 5-8 in c

$C_{in}(x, s) = \text{lub} \{ C_{out}(x, p) \mid p$ i

lub is the reason dataflow analysis is an **algorithm**: because lub is monotonic, we only need to analyze each loop **as many times as the lattice is tall**

Termination

- let’s formalize the argument that our nullness analysis terminates
  - saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes, after all

Termination

- let’s formalize the argument that our nullness analysis terminates
  - saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes, after all
- the use of **lub** explains why the algorithm terminates:

Termination

- let's formalize the argument that our nullness analysis terminates
  - saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes, after all
- the use of **lub** explains why the algorithm terminates:
  - values start as □ and only increase

Termination

- let's formalize the argument that our nullness analysis terminates
  - saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes, after all
- 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

Termination

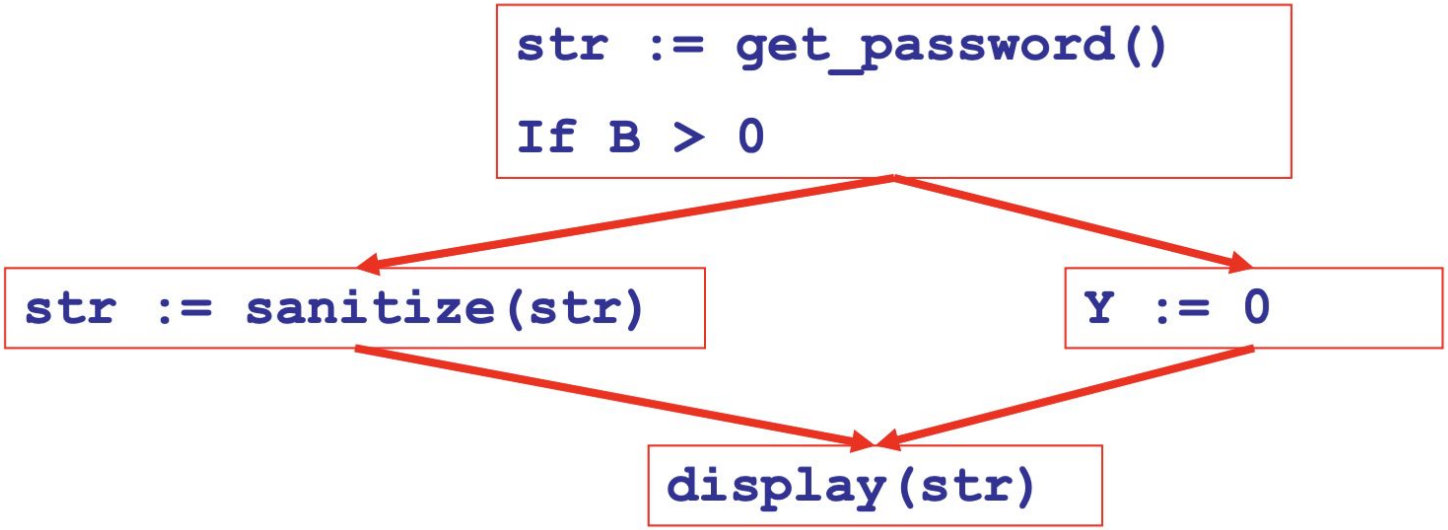
- let's formalize the argument that our nullness analysis terminates
  - saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes, after all
- the use of **lub** explains why the algorithm terminates:
  - values start as □ and only increase
  - □ can change to a constant, and a constant to T
  - thus, C_(x, s) can change at most **twice** (= lattice height minus one)

Another example: secure information flow

Analysis goal: report a warning if any ***source*** of secure information (e.g., a password) potentially connects to a public ***sink***, like a display function

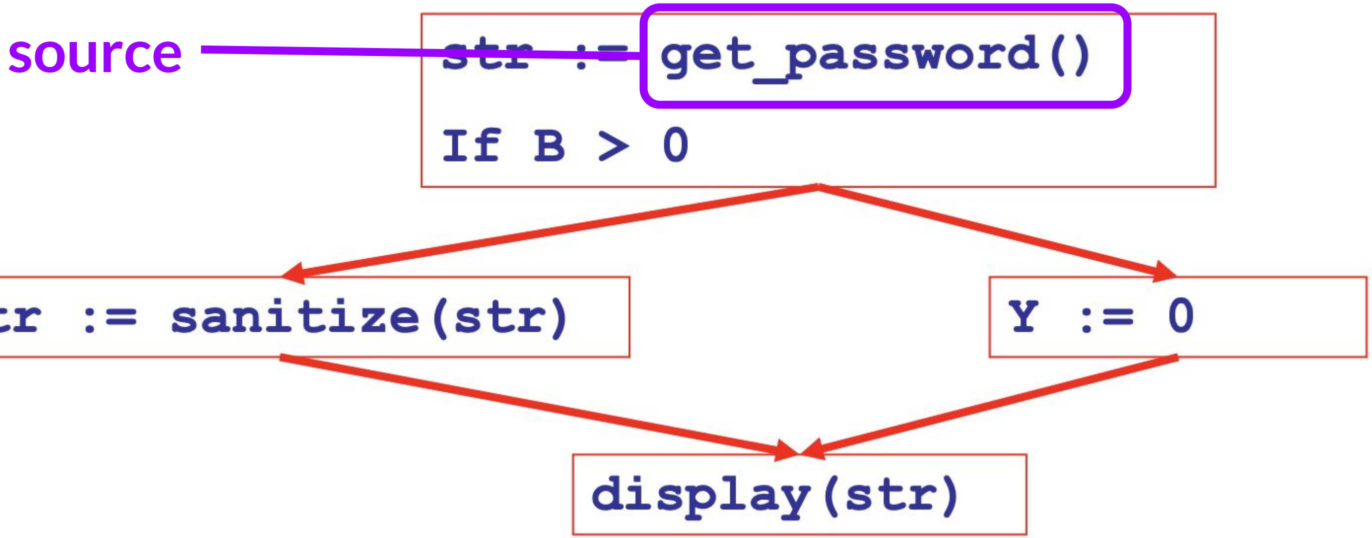
Another example: secure information flow

Analysis goal: report a warning if any ***source*** of secure information (e.g., a password) potentially connects to a public ***sink***, like a display function



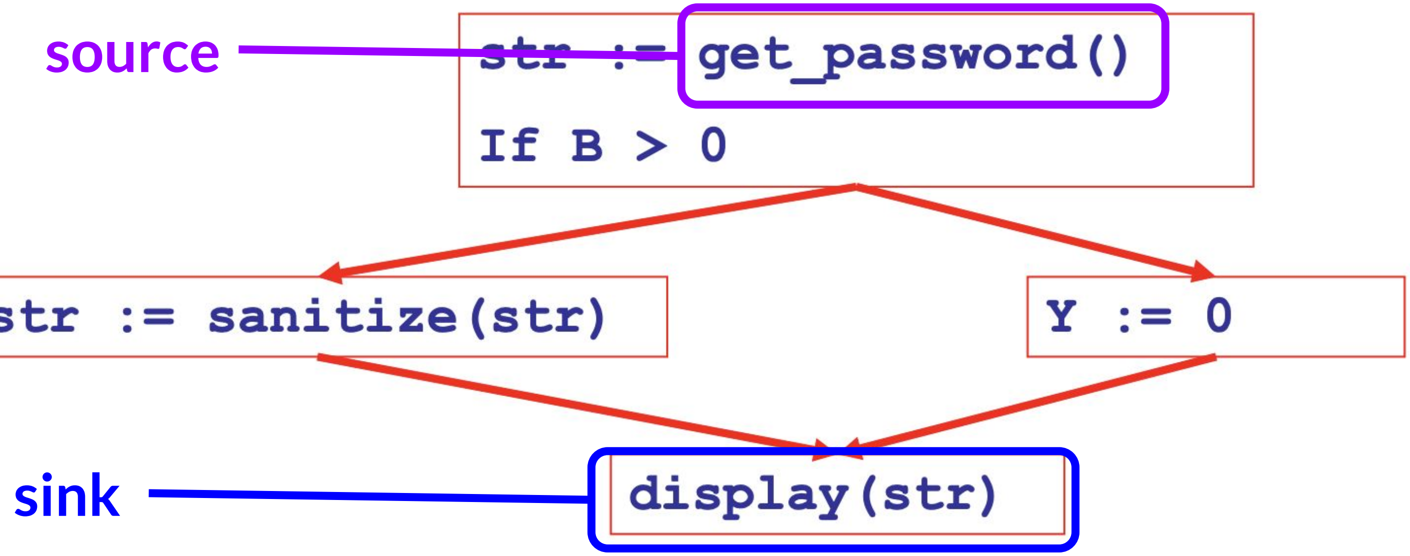
Another example: secure information flow

Analysis goal: report a warning if any ***source*** of secure information (e.g., a password) potentially connects to a public ***sink***, like a display function



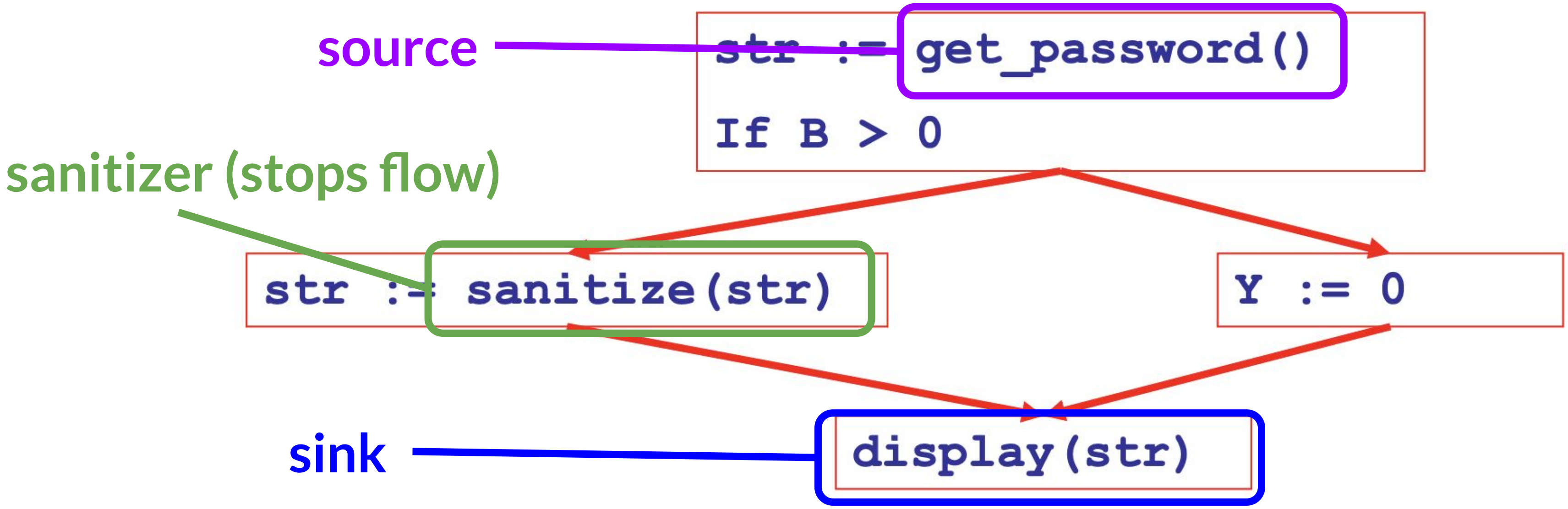
Another example: secure information flow

Analysis goal: report a warning if any ***source*** of secure information (e.g., a password) potentially connects to a public ***sink***, like a display function



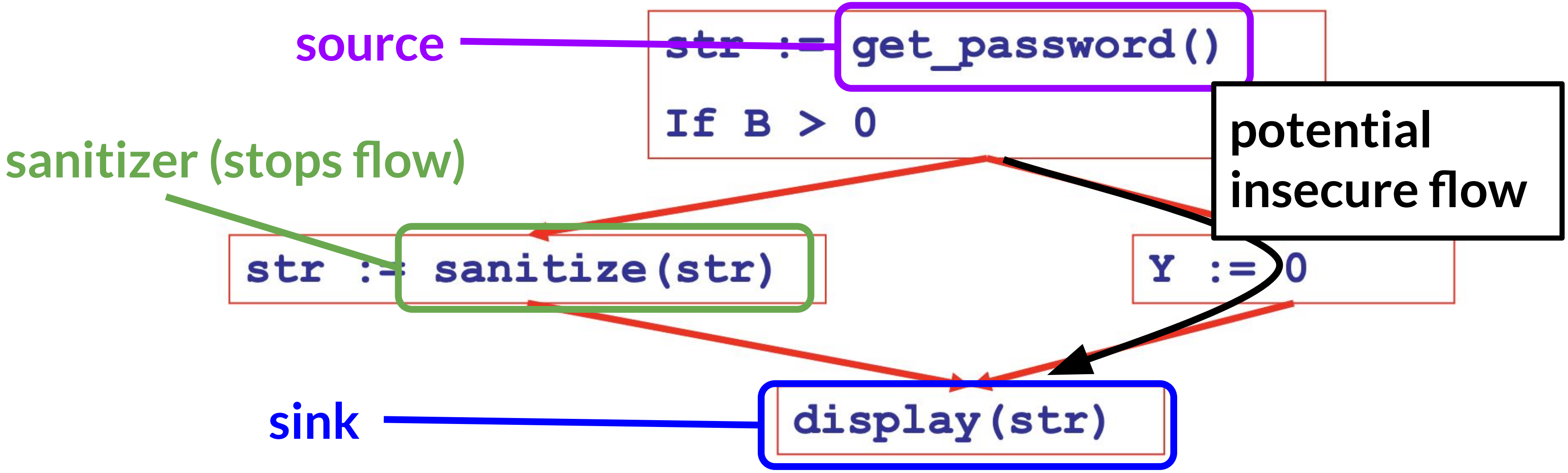
Another example: secure information flow

Analysis goal: report a warning if any ***source*** of secure information (e.g., a password) potentially connects to a public ***sink***, like a display function



Another example: secure information flow

Analysis goal: report a warning if any ***source*** of secure information (e.g., a password) potentially connects to a public ***sink***, like a display function



Taint analysis

Definition: A ***taint analysis*** (or ***reachability analysis***) tracks whether (any/all) value(s) from a set of sources reach a set of sinks

- applications in security: e.g., secure information flow

Taint analysis

Definition: A ***taint analysis*** (or ***reachability analysis***) tracks whether (any/all) value(s) from a set of sources reach a set of sinks

- applications in security: e.g., secure information flow
- stand-in here for a broad class of dataflow analyses

Taint analysis

Definition: A ***taint analysis*** (or ***reachability analysis***) tracks whether (any/all) value(s) from a set of sources reach a set of sinks

- applications in security: e.g., secure information flow
- stand-in here for a broad class of dataflow analyses
- how would we build it?
  - we'll write a set of rules, just as we did for our nullness analysis

Secure information flow analysis

- first step: decide what **abstract values** to track

Secure information flow analysis

- first step: decide what **abstract values** to track
  - only need a **single boolean**: can it be sensitive

Secure information flow analysis

- first step: decide what **abstract values** to track
  - only need a **single boolean**: can it be sensitive
  - **define** $H_{in/out}(x, s)$ = true if variable x can be sensitive before/after statement s, = false otherwise

Secure information flow analysis

- first step: decide what **abstract values** to track
  - only need a **single boolean**: can it be sensitive
  - **define** $H_{in/out}(x, s)$ = true if variable x can be sensitive before/after statement s, = false otherwise
    - note that we are abstracting away almost everything!

Secure information flow analysis

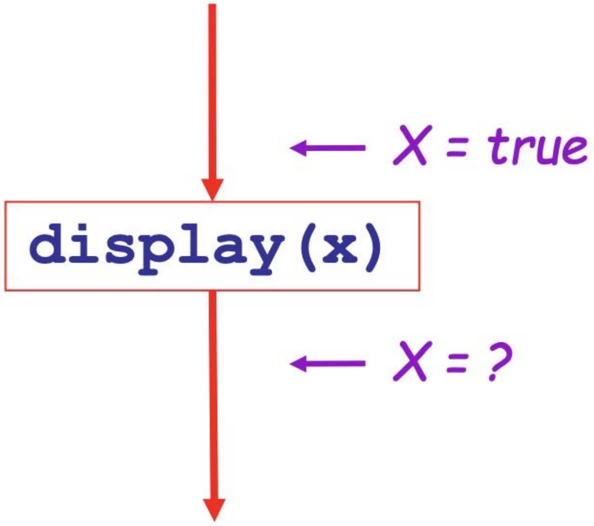
- first step: decide what **abstract values** to track
  - only need a **single boolean**: can it be sensitive
  - **define** $H_{in/out}(x, s)$ = true if variable x can be sensitive before/after statement s, = false otherwise
    - note that we are abstracting away almost everything!
- second step: **statement-by-statement rules** to express how this works

Secure information flow analysis

- first step: decide what **abstract values** to track
  - only need a **single boolean**: can it be sensitive
  - **define** $H_{in/out}(x, s)$ = true if variable x can be sensitive before/after statement s, = false otherwise
    - note that we are abstracting away almost everything!
- second step: **statement-by-statement rules** to express how this works

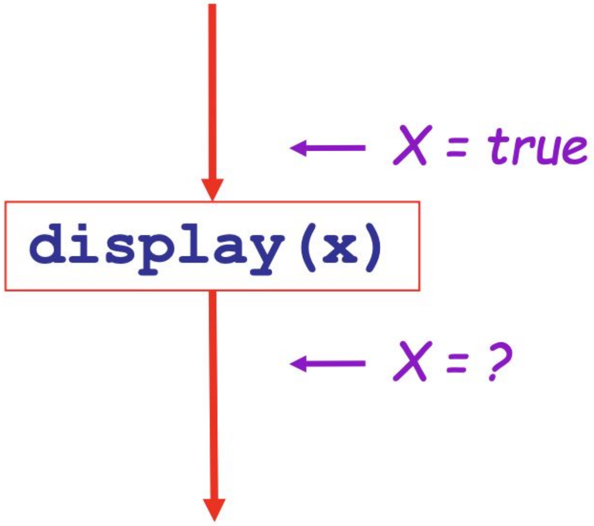
Note that the rules for this analysis are intended to be applied "backwards"

Secure information flow analysis: rule 1



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

Secure information flow analysis: rule 1

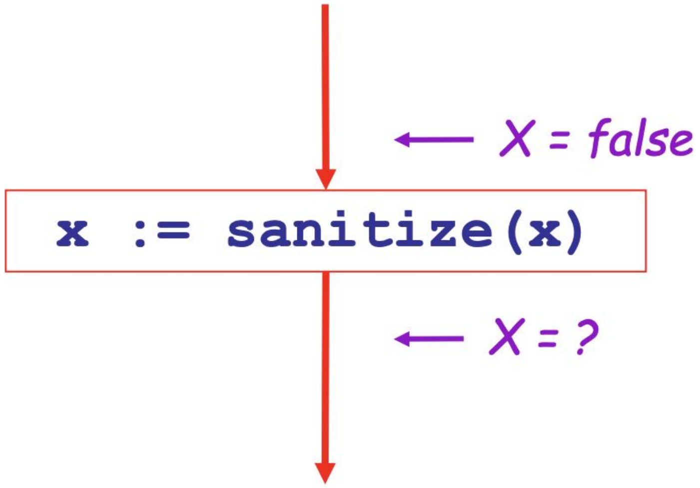


$H_{in}(x, s)$ = true if s displays x publicly

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

Secure information flow analysis: rule 2

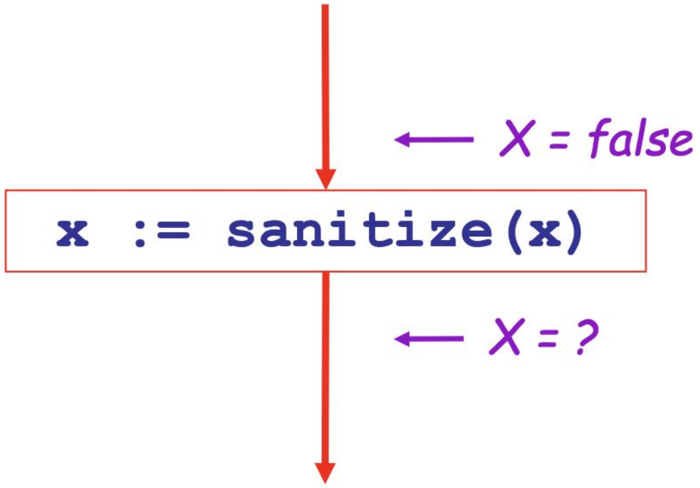
$H_{in}(x, x := e) = false$



Secure information flow analysis: rule 2

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

This means any value that is sanitized is not sensitive

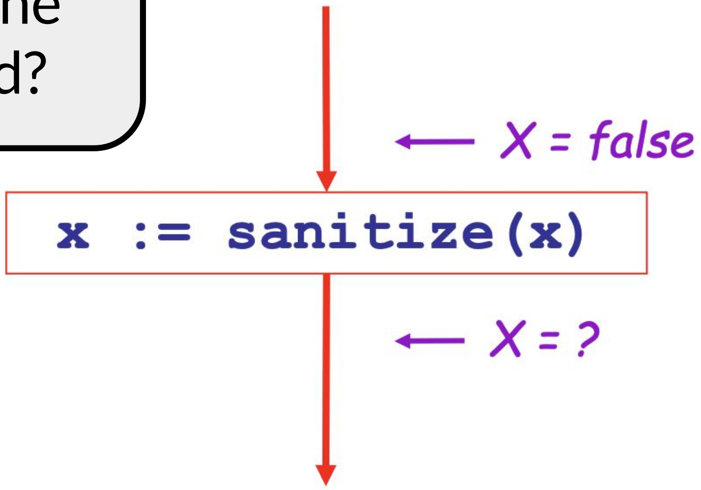


Secure information flow analysis: rule 2

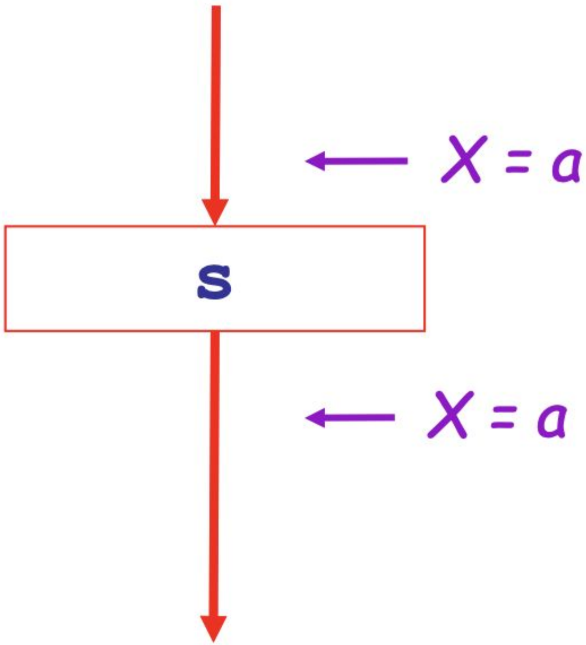
Does this rule say anything about the sanitize() method?

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

This means any value that is sanitized is not sensitive

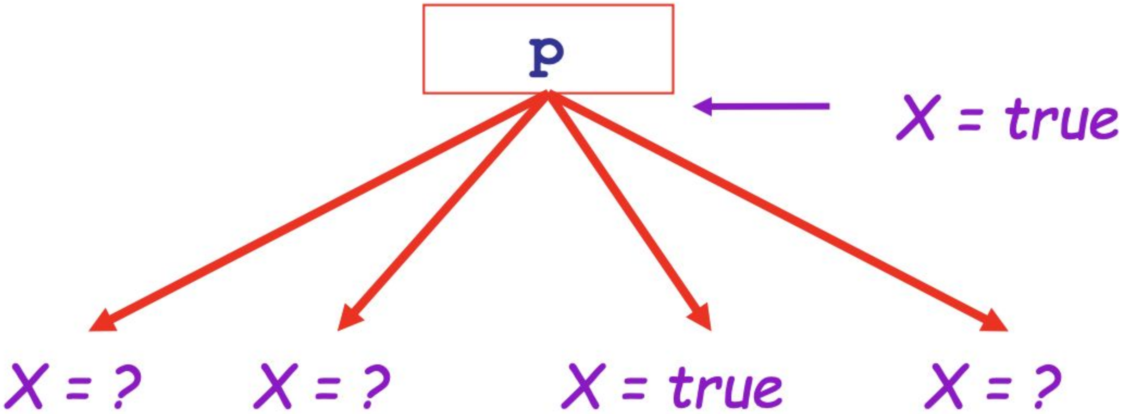


Secure information flow analysis: rule 3



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

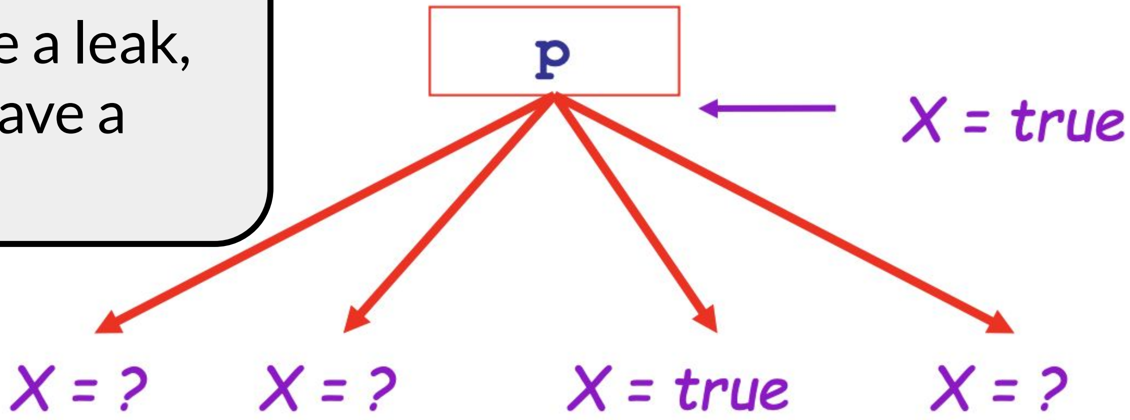
Secure information flow analysis: rule 4



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

Secure information flow analysis: rule 4

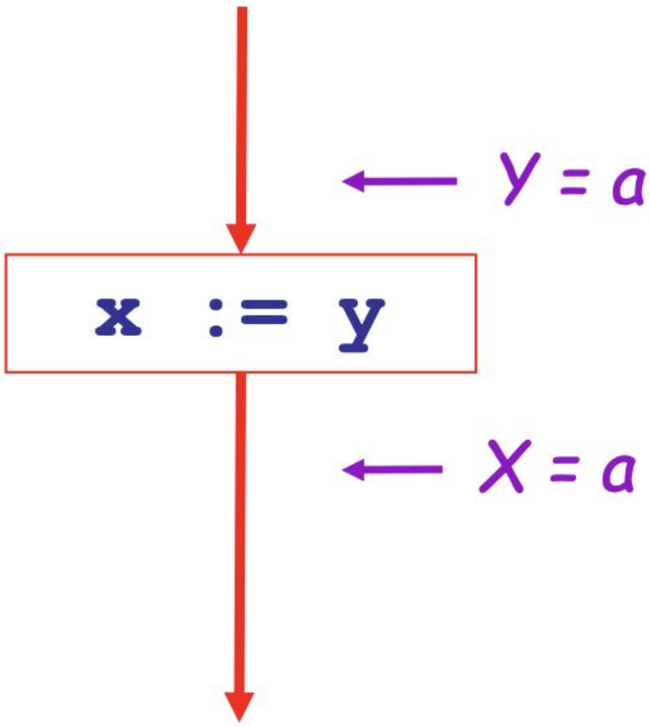
if there is **even one** way to have a leak, we might have a leak!



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

Secure information flow analysis: rule 5

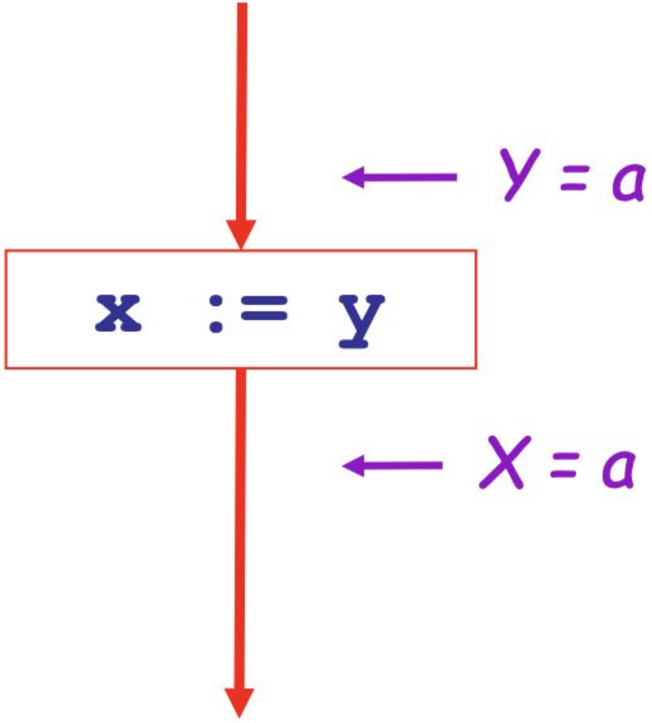
$H_{in}(y, x := y) = H_{out}(x, x := y)$



Secure information flow analysis: rule 5

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

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



Secure information flow analysis: algorithm

Secure information flow analysis: algorithm

1. let all H_(…) = **false initially**

Secure information flow analysis: algorithm

1. let all H_(...) = **false initially**

false is like □ in our nullness analysis!

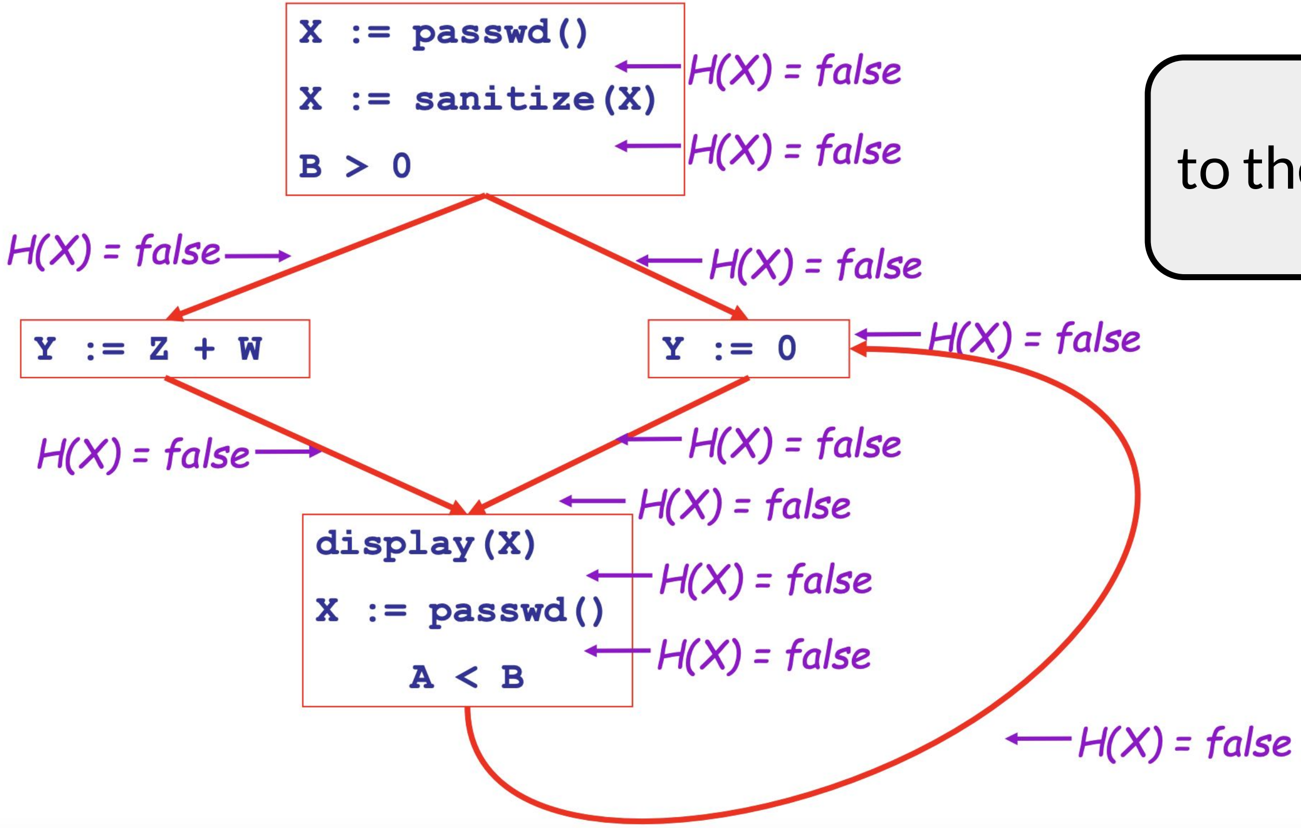
Secure information flow analysis: algorithm

1. let all H_(…) = **false initially**
2. **repeat** until all statements s satisfy rules 1-5:
    - pick a statement where one of the rules does not hold and update using the appropriate rule

Secure information flow analysis: algorithm

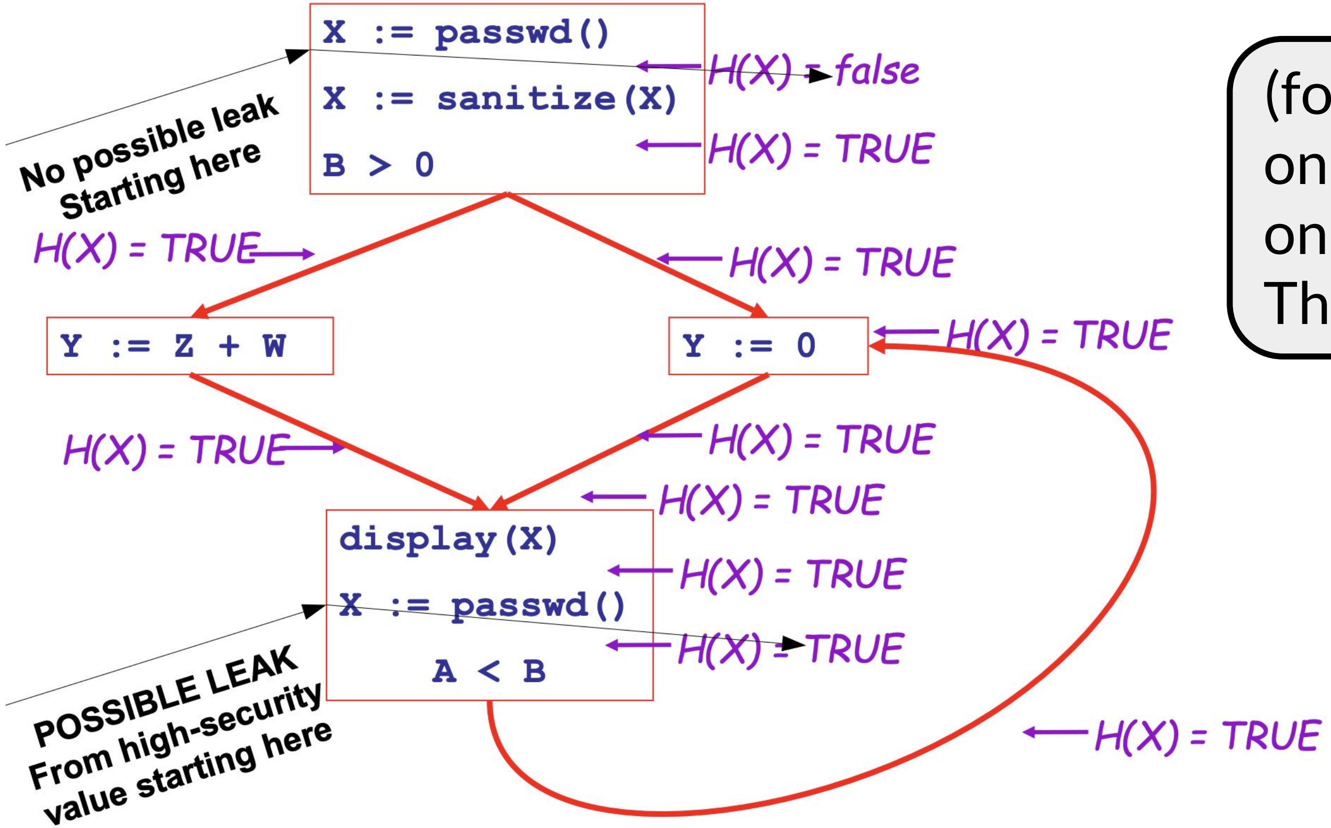
1. let all H_(...) = **false initially**
2. **repeat** until all statements s satisfy rules 1-5:
   - pick a statement where one of the rules does not hold and update using the appropriate rule
3. once the analysis reaches a fixed point, **issue a warning** at any source (x, s) where H_{out}(x, s) is true (= leaks sensitive information)

Secure information flow analysis: example



to the whiteboard!

Secure information flow analysis: example



(for those reading online later, solved on the whiteboard. This is the solution.)

Limitations of static analysis

Limitations of static analysis

- static analysis **abstracts away** information to remain decidable

Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?

Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
    - can we come up with a program for which one of our example static analyses “gets the wrong answer”?

Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
    - can we come up with a program for which one of our example static analyses “gets the wrong answer”?
  - can we ever have a “**perfect**” abstraction?

Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
    - can we come up with a program for which one of our example static analyses “gets the wrong answer”?
  - can we ever have a “**perfect**” abstraction?
    - of course not (Rice’s theorem again)

Limitations of static analysis

- static analysis **abstracts away** information to remain decidable
  - **potential problem**: what if the information that was abstracted away is important?
    - can we come up with a program for which one of our example static analyses “gets the wrong answer”?
  - can we ever have a “**perfect**” abstraction?
    - of course not (Rice’s theorem again)
    - but, in practice, we can get very close

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:
    - x86/64 calling convention

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:
    - x86/64 calling convention
    - complex API protocols ("call A then B then C then …")

Limitations of static analysis

- static analysis is **best** when the rules it enforces are:
  - simple to express to the computer
  - hard for a human to apply
- **implication**: if you find yourself struggling to follow a well-defined (but complicated for a human) rule set while writing code, it might be time to reach for a static analysis
  - this sort of situation comes up often:
    - x86/64 calling convention
    - complex API protocols (“call A then B then C then …”)
    - security rules, etc.

Static analysis in practice

You’re likely to encounter:

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses

heuristic is a fancy word for "best effort"

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
  - built into modern IDEs

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
  - built into modern IDEs
  - aim for low false positive rates

Static analysis in practice

You're likely to encounter:

- static **type systems** (sound)
- **linters** or other style checkers (**syntactic** = not dataflow)
- "***heuristic***" bug-finding tools backed by dataflow analyses
    - built into modern IDEs
    - aim for low false positive rates
    - widely used in industry:
        - ErrorProne at Google, Infer at Meta, SpotBugs at many places (including Amazon), Coverity, Fortify, etc.

Static analysis in practice

Less common, but useful to know about:

Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java

Static analysis in practice

Less common, but useful to know about:

- ***pluggable*** type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)

What is a pluggable type?

```
@Positive int x
```

What is a pluggable type?

```
@Positive int x
```

Basetype

What is a pluggable type?

```
@Positive int x
```

Type qualifier — Basetype

What is a pluggable type?

`@Negative int x`

Type qualifier — Basetype

What is a pluggable type?

```
@NonConstant int x
```

Type qualifier   Basetype

What is a pluggable type?

`@Positive int x`

Type qualifier — Basetype

What is a pluggable type?

```
@Positive int x
```

Type qualifier — Basetype

Qualified type

Pluggable type systems: key ideas

Pluggable type systems: key ideas

- developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other sound static analyses)

Pluggable type systems: key ideas

- developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other sound static analyses)
- type qualifiers **encode** property of interest
  - effectively a "second" type system

Pluggable type systems: key ideas

- developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other sound static analyses)
- type qualifiers **encode** property of interest
  - effectively a "second" type system
- qualified types are a **Cartesian product** of a type from the pluggable type system and a type from the base type system

Pluggable type systems: key ideas

- developers already use static type systems, so they're familiar with the general idea of types => **relatively easy to use** (compared to other sound static analyses)
- type qualifiers **encode** property of interest
  - effectively a "second" type system
- qualified types are a **Cartesian product** of a type from the pluggable type system and a type from the base type system
- typechecking is naturally **modular** = fast
  - but this comes at a cost: programmers need to write types

Pluggable type systems: key ideas

- developers already use static type … ith the general idea of types => **relativ** … other sound static analyses)
- type qualifiers **encode** property of …
  - effectively a "second" type syst…
- qualified types are a **Cartesian product** of a type from the pluggable type system and a type from the base type system
- typechecking is naturally **modular** = fast
  - but this comes at a cost: programmers need to write types

designing better (more expressive, more usable, etc.) pluggable type systems is an area of active research (mine!)

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification* (subject of 10/25 reading)

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification* (subject of 10/25 reading)
  - you write a specification

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification* (subject of 10/25 reading)
  - you write a specification
  - tool verifies that code matches that specification

Static analysis in practice

Less common, but useful to know about:

- *pluggable* type systems
  - these are extensions to a type system that lets it prove more properties, e.g., adding nullness-checking to Java
  - most common sound analysis (used by Google, Uber, others)
- *formal verification* (subject of 10/25 reading)
  - you write a specification
  - tool verifies that code matches that specification
  - very high effort, but enables sound reasoning about complex properties (= worth it for very high value systems)

Static analysis in practice: soundiness

- all “**sound**” static analyses have a ***trusted computing base*** (***TCB***)

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (***TCB***)
    - the TCB is the code whose correctness must be assumed for the analysis to actually be sound

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)

Static analysis in practice: soundness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
  - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
  - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)
  - TCB for some formal verifiers is **very small** (< 1000 LoC)
    - but these tools (e.g., Coq) are **much harder to use**

Static analysis in practice: soundiness

- all "**sound**" static analyses have a ***trusted computing base*** (*TCB*)
    - the TCB is the code whose correctness must be assumed for the analysis to actually be sound
- **TCB size** is an important differentiator between "sound" analyses
    - e.g., TCB for many of my pluggable type systems includes the entire Java compiler (limits soundness a lot!)
    - TCB for some formal verifiers is **very small** (< 1000 LoC)
        - but these tools (e.g., Coq) are **much harder to use**
- soundness theorems also usually make some **assumptions** about the code being analyzed (e.g., no calls to native code, no reflection)

Reading quiz: static analysis (1)

Reading quiz: static analysis (1)

Q1: **TRUE** or **FALSE**: very few users of FindBugs (at the time the article was written) use an automatic build system where new issues are automatically identified and flagged

Q2: How many "infinite recursive loop" bugs did FindBugs find in Google's codebase?

A. 0
B. 1
C. more than 70

Reading quiz: static analysis (1)

Q1: **TRUE** or **FALSE**: very few users of FindBugs (at the time the article was written) use an automatic build system where new issues are automatically identified and flagged

Q2: How many "infinite recursive loop" bugs did FindBugs find in Google's codebase?

A. 0
B. 1
C. more than 70

Reading quiz: static analysis (1)

Q1: **TRUE** or **FALSE**: very few users of FindBugs (at the time the article was written) use an automatic build system where new issues are automatically identified and flagged

Q2: How many "infinite recursive loop" bugs did FindBugs find in Google's codebase?

A. 0
B. 1
C. more than 70

Reading Quiz: static analysis (2)

Reading Quiz: static analysis (2)

Q1: the author advocates which of the following progamming language paradigms for writing executable specifications?

A. functional
B. imperative
C. declarative
D. object-oriented

Q2: **TRUE** or **FALSE**: the author claims that a disadvantage of formal verification is that it only identifies bugs, but doesn’t indicate how to fix them

Reading Quiz: static analysis (2)

Q1: the author advocates which of the following progamming language paradigms for writing executable specifications?

A. functional
B. imperative
C. declarative
D. object-oriented

Q2: **TRUE** or **FALSE**: the author claims that a disadvantage of formal verification is that it only identifies bugs, but doesn't indicate how to fix them

Reading Quiz: static analysis (2)

Q1: the author advocates which of the following progamming language paradigms for writing executable specifications?

A. functional
B. imperative
C. declarative
D. object-oriented

Q2: **TRUE** or **FALSE**: the author claims that a disadvantage of formal verification is that it only identifies bugs, but doesn't indicate how to fix them

Static analysis: summary

- static analysis is very good at enforcing **simple rules**
  - **much** better than humans at this
- all interesting semantic properties of programs are **undecidable**, so all static analyses must **approximate**
  - goal in analysis design is to **abstract away unimportant details**, but keep important details
  - **dataflow analysis** is one technique for static analysis
  - trade-offs between false positives, false negatives, analysis time
- soundness & completeness are **possible, but rare**
  - all soundness guarantees come with caveats about the TCB