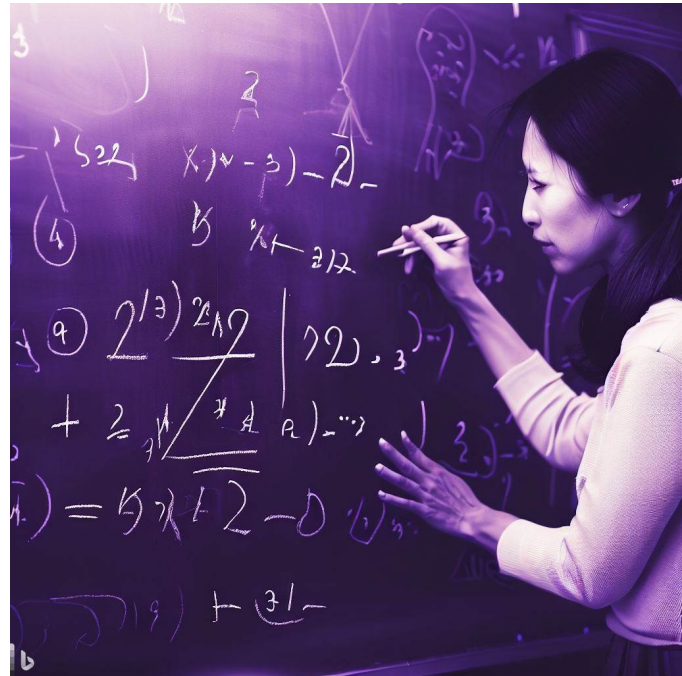


# CSE 331

## Reasoning

James Wilcox

with thanks to Kevin Zatloukal for many slides



# Java Code Without Mutation

---

- **Code without mutation consists of...**
  - straight-line code      variable declarations and `return`
  - conditionals              `if` statements
  - recursion
- **Code that only uses these properties is "functional"**
  - we will limit ourselves to functional code *initially*
- **Can translate any of our math functions into this**
- **Data is not so easy**
  - compound types need to be faked with classes

# Lists

---

```
type List := nil | cons(hd : int, tl : List)
```

- Inductive definition of lists of integers

nil

cons(3, nil)

cons(2, cons(3, nil))

cons(1, cons(2, cons(3, nil)))

**A very important data type!**



# Example: List Length

---

```
type List := nil | cons(hd : int, tl : List)
```

- **Mathematical definition of list length:**

$$\text{len} : (\text{List}) \rightarrow \mathbb{N}$$
$$\text{len}(\text{nil}) \quad := 0$$
$$\text{len}(x :: L) \quad := 1 + \text{len}(L)$$

- any list is either nil or  $x :: L$  for some  $x$  and  $L$
- cases are exclusive and exhaustive

# Inductive Data Types in Java

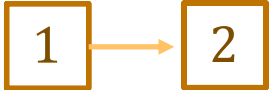
---

- **Java does not natively support these data types**  
other languages support these as well as tuples and records
- **Can fake them with classes, e.g.:**

```
public class List {  
    final int hd;  
    final List tl;  
  
    public static final List nil = null;  
  
    public static List cons(int hd, List tl) {  
        return new List(hd, tl);  
    }  
  
    private List(int hd, List tl) {  
        this.hd = hd;  
        this.tl = tl;  
    }  
}
```

**type List := nil | cons(hd :  $\mathbb{Z}$ , tl : List)**

**cons(1, cons(2, nil));**



The diagram shows a linked list structure. It consists of two square boxes, one containing the number '1' and the other containing the number '2'. An orange arrow points from the right side of the first box to the left side of the second box, representing the 'next' pointer in the list.

# Recall: Software Development Process

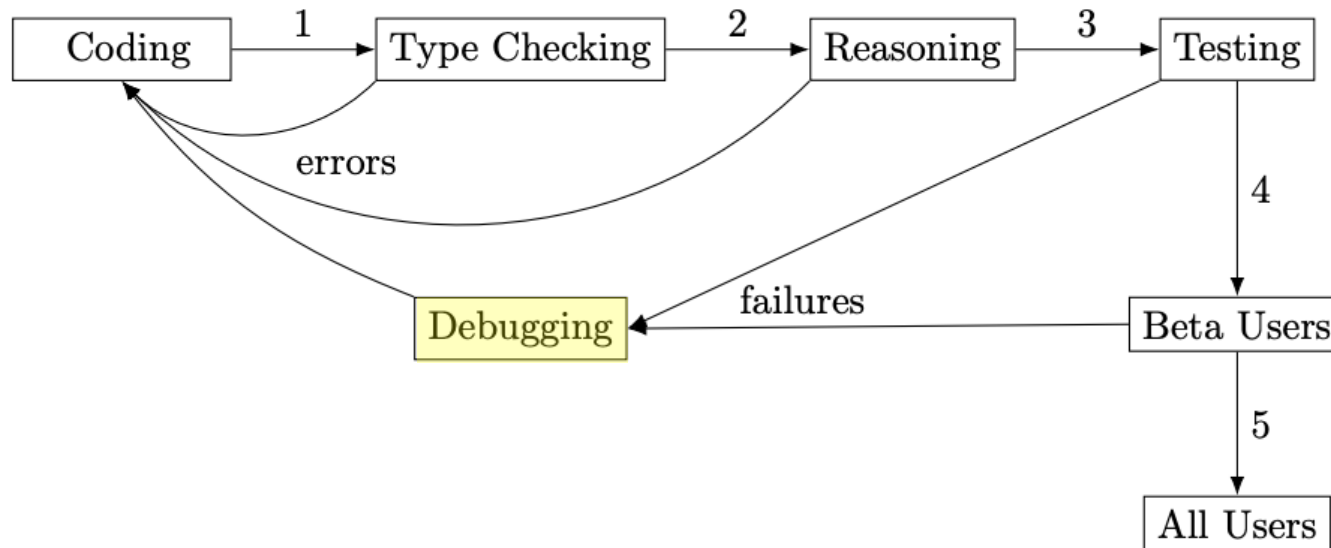
---



- **an imperative spec comes with a plan**  
can translate this directly into Java code
- **a declarative spec does not**

# Software Implementation

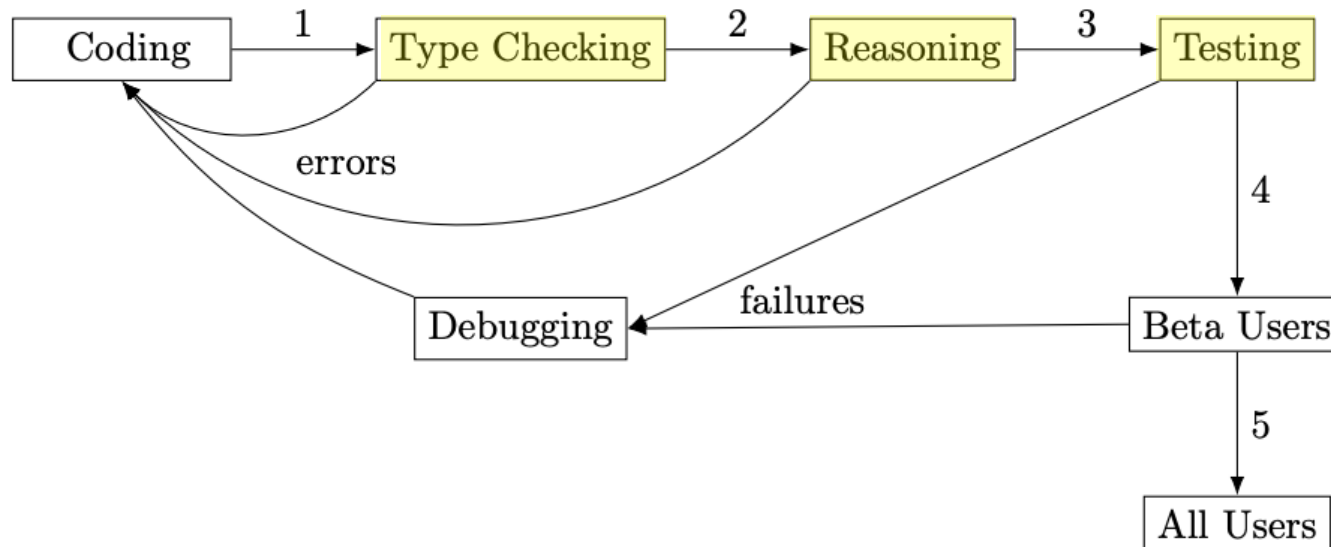
---



- **Debugging** is the search from failure to bug
  - harder the more code that must be searched
  - easiest possible case is a unit test failure
  - time required is *random* with a *long tail*

# Software Implementation

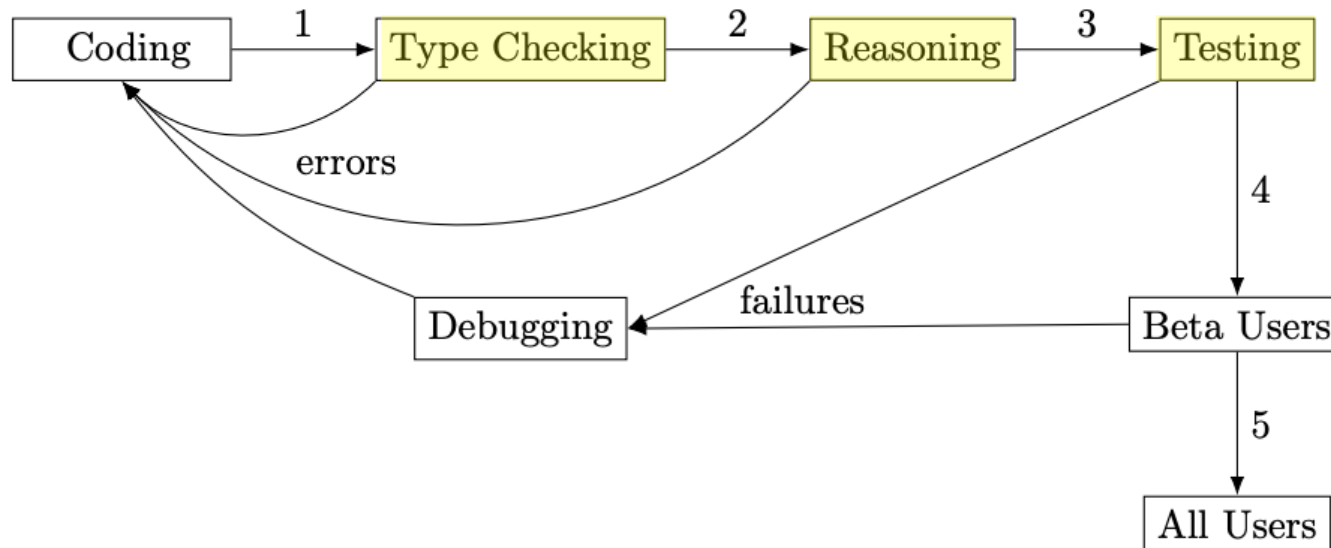
---



- **Three techniques used to check correctness**
  - each removes  $\sim 2/3^{\text{rd}}$  of the bugs present
  - but each tends to find *different* bugs
  - need all three techniques to get 99+% assurance

# Software Implementation

---



- Three techniques used to check correctness
  - type checking is familiar (and more coming later)
  - already discussed testing
  - focus now on reasoning

# Reasoning

---

- “Thinking through” what the code does on all inputs
  - ensuring what it does is correct in *all cases*
- Type checking does **not** do this
  - only checks that return values have the right type
  - e.g., ensures that an `int` but not that it is `1`
- Testing does **not** do this
  - only verifies a correct output on *some* inputs

# Reasoning

---

- “Thinking through” what the code does on all inputs
  - neither testing nor type checking can do this
- Can be done formally or informally
  - most professionals reason *informally*
  - we will start with formal reasoning and move to informal
    - formal reasoning is a steppingstone to informal reasoning (same core ideas)
    - formal reasoning still needed for the **hardest** problems
- Definition of correctness comes from the specification...

# Correctness Requires a Specification

---

Specification contains two sets of facts

## Precondition:

facts we are *promised* about the inputs

## Postcondition:

facts we are required to *ensure* for the output

## Local Correctness (method satisfying its spec):

for every input satisfying the precondition,  
the output will satisfy the postcondition

# Facts

---

- **Starting point for reasoning is “facts” about the code**
  - **things we know to be true about the variables**  
these hold for all inputs (no matter what value the variable has)
  - **typically, “=” or “≤”**

```
// @param n a natural number
int f(int n) {
    final int m = 2 * n;
    return (m + 1) * (m - 1);
};
```

find facts by reading along path  
from top to return statement

- **At the return statement, we know these facts:**
  - $n \geq 0$
  - $m = 2n$

note: these hold for all valid inputs

# Facts

---

- **Starting point for reasoning is “facts” about the code**
  - **things we know to be true about the variables**  
these hold for all inputs (no matter what value the variable has)
  - **typically, “=” or “ $\leq$ ”**

```
// @param n a natural number
int f(int n) {
    final int m = 2 * n;
    return (m + 1) * (m - 1);
};
```

- **No need to include the fact that n is an integer**
  - that is true, but the type checker takes care of that
  - no need to repeat reasoning done by the type checker

# Finding Facts at a Return Statement

---

- Consider this code:

```
// Inputs a and b can be any integers.  
// Returns a non-negative integer.  
int f(int a, int b) {  
    final List L = cons(a, cons(b, nil));  
    if (a >= 0 && b >= 0)  
        return sum(L);  
    ...  
}
```

find facts by reading along path  
from top to return statement

facts are assertions about the code

- Known facts include “ $a \geq 0$ ”, “ $b \geq 0$ ”, and “ $L = \text{cons}(\dots)$ ”
- Remains to prove that “ $\text{sum}(L) \geq 0$ ”

# Implications

---

- **We can use the facts we know to prove more facts**
  - if we can prove R using facts P and Q,  
we say that R “follows from” or “is implied by” P and Q
  - proving this fact is proving an “**implication**”
- **We will see how to do this shortly...**
  - will be familiar to those who have taken 311

# Checking Correctness of Functional Code

---

- **Steps for checking correctness of functional code:**
  1. Collect facts at each return statement
  2. Ensure those facts **imply** each fact of the **postcondition**
- **Checking correctness requires proving **implications****
  - need to prove facts about the **return values**
  - return values must satisfy the facts of the **postcondition**
- **If the known facts do not imply the postcondition, then the code is **wrong****
  - **some valid input does not satisfy the postcondition**
    - the code will be correct in 99% of our examples,
    - but this is the reason why we do reasoning: to find mistakes

# Collecting Facts

---

- Saw how to collect facts in code consisting of
  - "final" variable declarations
  - "if" statements
  - collect facts by reading along path from top to return
- Those elements cover all code without mutation
  - covers everything describable by our math notation
  - we can calculate interesting values with *recursion*
- Will need more tools to handle code with mutation...

# Checking correctness

---

```
// Inputs x and y are positive integers
// Returns a positive integer.
int f(int x, int y) => {
    return x + y;
};
```

- Known facts “ $x \geq 1$ ” and “ $y \geq 1$ ”
- Correct if the return value is a positive integer

$x + y$

# Checking correctness

---

```
// Inputs x and y with x >= 9 and y >= -8
// Returns a positive integer.
int f(int x, int y) => {
    return x + y;
};
```

- Known facts “ $x \geq 9$ ” and “ $y \geq -8$ ”
- Correct if the return value is a positive integer

$x + y$

# Checking correctness

---

```
// Inputs x and y with x > 8 and y > -9
// Returns a positive integer.
int f(int x, int y) => {
    return x + y;
};
```

- Known facts “ $x > 8$ ” and “ $y > -9$ ”
- Correct if the return value is a positive integer

# Checking correctness

---

```
// Inputs x and y with x > 3 and y > 4
// Returns an integer that is 10 or larger.
int f(int x, int y) => {
    return x + y;
};
```

- Known facts “ $x \geq 4$ ” and “ $y \geq 5$ ”
- Correct if the return value is 10 or larger

$x + y$

# Recall: Finding Facts at a Return Statement

---

- Consider this code

```
// Inputs a and b must be integers.  
// Returns a non-negative integer.  
int f(int a, int b) {  
    final List L = cons(a, cons(b, nil));  
    if (a >= 0 && b >= 0)  
        return sum(L);  
    ...  
}
```

find facts by reading along path  
from top to return statement

- Known facts include “ $a \geq 0$ ”, “ $b \geq 0$ ”, and “ $L = \text{cons}(\dots)$ ”
- Must prove that  $\text{sum}(L) \geq 0$ 
  - True since  $\text{sum}(L) = a + b$

# Proving Correctness with Conditionals

---

```
// Inputs x and y are integers.  
// Returns a number less than x.  
int f(int x, int y) {  
    if (y < 0) {  
        return x + y;  
    } else {  
        return x - 1;  
    }  
};
```

- Known fact in then (top) branch: “ $y \leq -1$ ”

$x + y$	
$\leq x + -1$	since $y \leq -1$
$< x + 0$	since $-1 < 0$
$= x$	

# Proving Correctness with Conditionals

---

```
// Inputs x and y are integers.  
// Returns a number less than x.  
int f(int x, int y) {  
    if (y < 0) {  
        return x + y;  
    } else {  
        return x - 1;  
    }  
};
```

- Known fact in else (bottom) branch: “ $y \geq 0$ ”

$$\begin{aligned} &x - 1 \\ &< x + 0 && \text{since } -1 < 0 \\ &= x \end{aligned}$$

# Proving Correctness with Multiple Claims

---

- Need to check the claim from the spec at each `return`
- If spec claims multiple facts, then we must prove that each of them holds

```
// Inputs x and y are integers with x < y - 1
// Returns a number less than y and greater than x.
int f(int x, int y) { .. };
```

- multiple claims to prove:  $x < r$  and  $r < y$   
where “r” is the return value
- requires *two* calculation blocks

# Example Correctness with Conditionals

---

```
// Returns r with (r=a or r=b) and r >= a and r >= b
int max(int a, int b) {
    if (a >= b) {
        return a;
    } else {
        return b;
    }
};
```

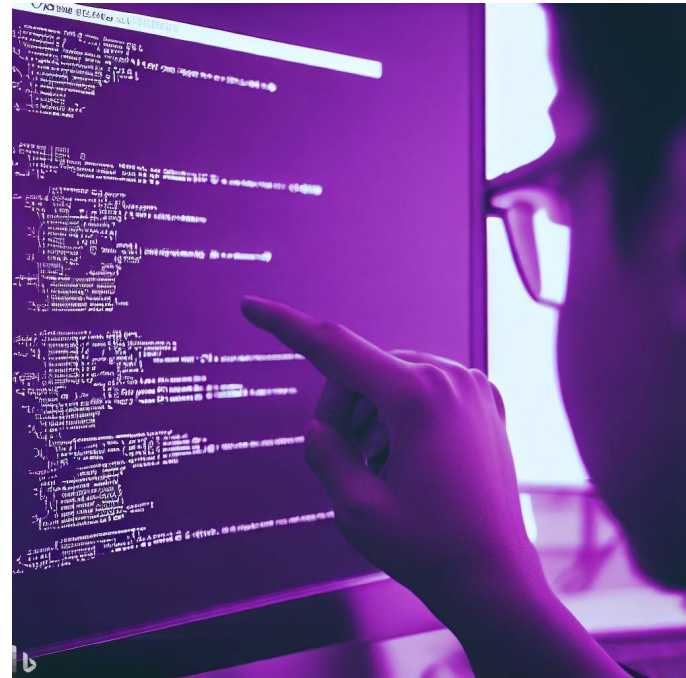
declarative spec of max

- Three different facts to prove at each **return**
- Two known facts in each branch (return value is “r”):
  - then branch:  $a \geq b$  and  $r = a$
  - else branch:  $a < b$  and  $r = b$
  - prove an "or" holds by proving one of the two options holds

# CSE 331

## Floyd Logic

James Wilcox and Kevin Zatloukal



# Reasoning So Far

---

- **Code so far made up of three elements**
  - straight-line code (just variable declarations and returns)
  - conditionals
  - recursion
- **All code without mutation looks like this**
- **Proving correctness is proving **implications****
  - check that known facts imply the required facts

# Recall: Finding Facts at a Return Statement

---

- Consider this code

```
// Inputs a and b must be integers.  
// Returns a non-negative integer.  
public int f(int a, int b) {  
    if (a >= 0 && b >= 0) {  
        final List L = cons(a, cons(b, nil));  
        return sum(L);  
    }  
    ...  
}
```

find facts by reading along path  
from top to return statement

- Known facts include “ $a \geq 0$ ”, “ $b \geq 0$ ”, and “ $L = \text{cons}(\dots)$ ”
- Prove that postcondition holds: “ $\text{sum}(L) \geq 0$ ”

# Finding Facts at a Return Statement

---

- Consider this code

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) => {
    if (a >= 0 && b >= 0) {
        a = a - 1;
        final List L = cons(a, cons(b, nil));
        return sum(L);
    }
    ...
}
```

$a \geq 0$

$a \geq 0?$  No!

- Facts no longer hold throughout straight-line code
- When we state a fact, we have to say where it holds

# Correctness Levels

---

Description	Testing	Tools	Reasoning
no mutation	coverage	type checking	calculation induction
local variable mutation	“	“	Floyd logic
heap state mutation	“	“	rep invariants
array mutation	“	“	for-any facts

# Finding Facts at a Return Statement

---

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
    if (a >= 0 && b >= 0) {
        {{a ≥ 0}}
        a = a - 1;
        {{a ≥ -1}}
        final List L = cons(a, cons(b, nil));
        return sum(L);
    }
}
```

- When we state a fact, we have to say where it holds
- `{{ .. }}` notation indicates facts true at that point
  - cannot assume those are true anywhere else

# Finding Facts at a Return Statement

---

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
    if (a >= 0 && b >= 0) {
        {{a ≥ 0}}
        a = a - 1;
        {{a ≥ -1}}
        final List L = cons(a, cons(b, nil));
        return sum(L);
    }
}
```

- There are mechanical tools for moving facts around
  - “**forward reasoning**” says how they change as we move down
  - “**backward reasoning**” says how they change as we move up

# Finding Facts at a Return Statement

---

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
    if (a >= 0 && b >= 0) {
        {{a ≥ 0}}
        a = a - 1;
        {{a ≥ -1}}
        final List L = cons(a, cons(b, nil));
        return sum(L);
    }
}
```

- Professionals are *insanely* good at forward reasoning
  - “programmers are the Olympic athletes of forward reasoning”
  - you’ll have an edge by learning backward reasoning too

# Floyd Logic

# Floyd Logic

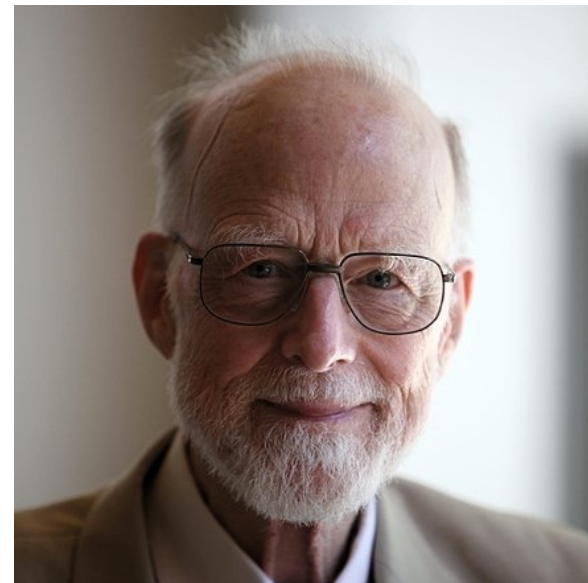
---

- **Invented by Robert Floyd and Sir Anthony Hoare**
  - Floyd won the Turing award in 1978
  - Hoare won the Turing award in 1980



Robert Floyd

picture from [Wikipedia](#)



Tony Hoare

# Floyd Logic Terminology

---

- The **program state** is the values of the variables
- An **assertion** (in  $\{\{ .. \}\}$ ) is a T/F claim about the state
  - an assertion “holds” if the claim is true
  - assertions are *math* not code  
(we do our reasoning in math)
- Most important assertions:
  - **precondition**: claim about the state when the function starts
  - **postcondition**: claim about the state when the function ends

# Hoare Triples

---

- A **Hoare triple** has two assertions and some code

$\{ \{ P \} \}$

$S$

$\{ \{ Q \} \}$

- $P$  is the precondition,  $Q$  is the postcondition
  - $S$  is the code
- 
- Triple is “**valid**” if the code is correct:
    - $S$  takes *any* state satisfying  $P$  into a state satisfying  $Q$   
does not matter what the code does if  $P$  does not hold initially
    - otherwise, the triple is invalid

# Correctness Example

---

```
/**
 * @param n an integer with  $n \geq 1$ 
 * @returns an integer m with  $m \geq 10$ 
 */
public int f(int n) {
    n = n + 3;
    return n * n;
};
```

# Correctness Example

---

```
/**
 * @param n an integer with  $n \geq 1$ 
 * @returns an integer m with  $m \geq 10$ 
 */
public int f(int n) {
    {{  $n \geq 1$  }}
    n = n + 3;
    {{  $n^2 \geq 10$  }}
    return n * n;
};
```

- Precondition and postcondition come from spec
- Remains to check that the triple is valid

# Hoare Triples with No Code

---

- Code could be empty:

$\{ \{ P \} \}$

$\{ \{ Q \} \}$

- When is such a triple valid?
  - valid iff  $P$  implies  $Q$
  - we already know how to check validity in this case:  
prove each fact in  $Q$  by calculation, using facts from  $P$

# Hoare Triples with No Code

---

- Code could be empty:

$$\{\{ a \geq 0, b \geq 0, L = \text{cons}(a, \text{cons}(b, \text{nil})) \}\}$$
$$\{\{ \text{sum}(L) \geq 0 \}\}$$

- Check that P implies Q by calculation

$$\begin{aligned} \text{sum}(L) &= \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) \\ &= a + \text{sum}(\text{cons}(b, \text{nil})) \\ &= a + b + \text{sum}(\text{nil}) \\ &= a + b \\ &\geq 0 + b \\ &\geq 0 + 0 \\ &= 0 \end{aligned}$$

since  $L = \dots$   
def of sum  
def of sum  
def of sum  
since  $a \geq 0$   
since  $b \geq 0$

# Hoare Triples with Code

---

- Code with code:

$\{ \{ P \} \}$

$S$

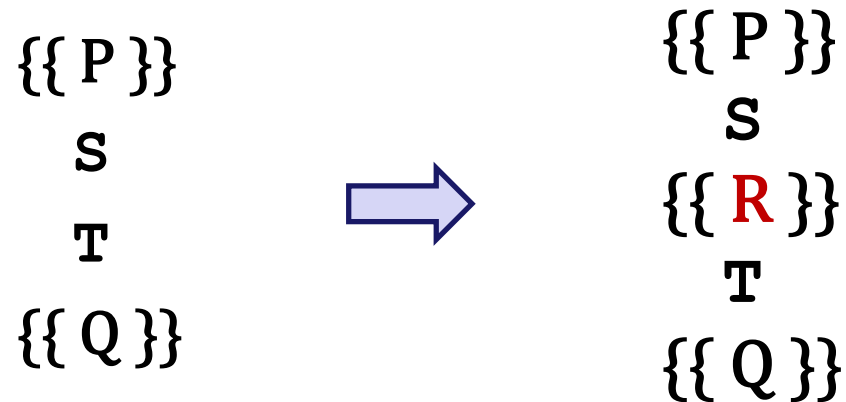
$\{ \{ Q \} \}$

- Easy if  $S$  is empty, but what if not?
- We can use **forward & backward** reasoning
  - move the assertions toward each other until they meet
  - then we have a triple with no code

# Hoare Triples with Multiple Lines of Code

---

- Code with multiple lines:

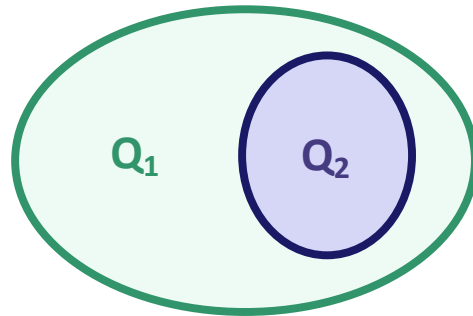


- Valid iff there exists an **R** making both triples valid
  - i.e.,  $\{\{ P \}\} S \{\{ R \}\}$  is valid and  $\{\{ R \}\} T \{\{ Q \}\}$  is valid
- Will see next how to put these to good use...

# Recall: Stronger Assertions

---

- **Assertion** is **stronger** iff it holds in a subset of states

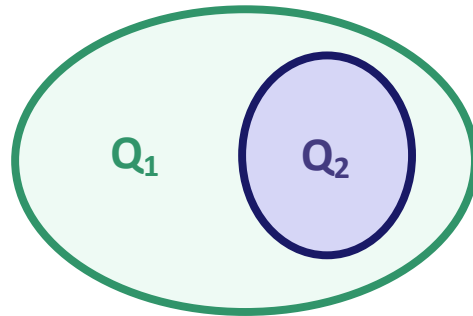


- **Stronger** assertion implies the **weaker** one
  - stronger is a synonym for “implies”
  - weaker is a synonym for “is implied by”

# Recall: Stronger Assertions

---

- **Assertion** is **stronger** iff it holds in a subset of states



- **Weakest** possible assertion is “true” (all states)
  - an empty assertion (“”) also means “true”
- **Strongest** possible assertion is “false” (no states!)

# Mechanical Reasoning Tools

---

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples

- **Forward** reasoning fills in postcondition

$$\{\{ P \} \} s \{\{ \_ \} \}$$

- gives *strongest* postcondition making the triple valid

- **Backward** reasoning fills in precondition

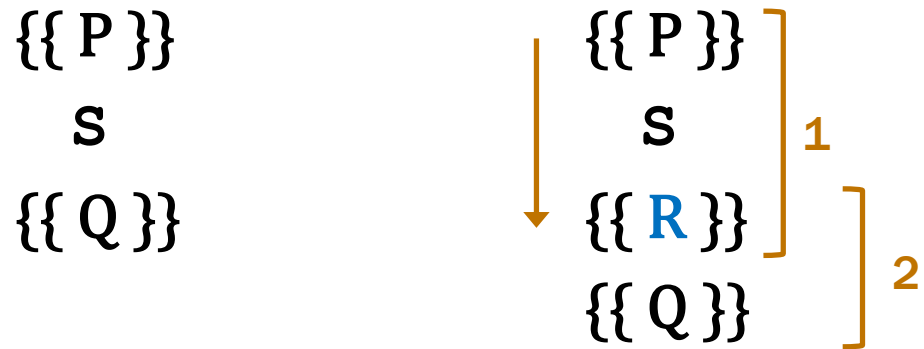
$$\{\{ \_ \} \} s \{\{ Q \} \}$$

- gives *weakest* precondition making the triple valid

# Correctness via Forward Reasoning

---

- Apply forward reasoning

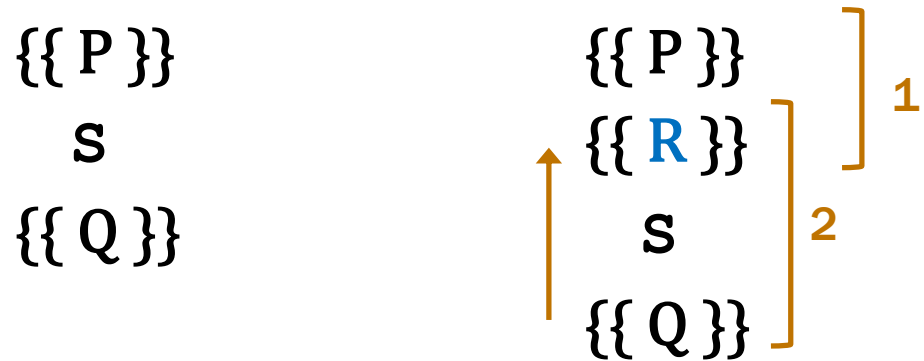


- first triple is always valid
- only need to check second triple
  - just requires proving an implication (since no code is present)
- If second triple is invalid, the code is **incorrect**
  - true because **R** is the strongest assertion possible here

# Correctness via Backward Reasoning

---

- Apply backward reasoning



- second triple is always valid
- only need to check first triple
  - just requires proving an implication (since no code is present)
- If first triple is invalid, the code is **incorrect**
  - true because **R** is the weakest assertion possible here

# Mechanical Reasoning Tools

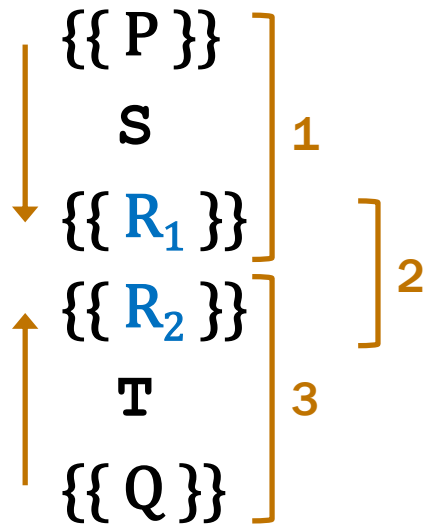
---

- **Forward / backward reasoning fill in assertions**
  - mechanically create valid triples
- **Reduce correctness to proving implications (again)**
  - this was already true for functional code
  - will soon have the same for imperative code
- **Implication will be false if the code is **incorrect****
  - reasoning can verify correct code
  - reasoning will never accept incorrect code

# Correctness via Forward & Backward

---

- Can use both types of reasoning on longer code



- first and third triples is always valid
- only need to check second triple  
verify that  $R_1$  implies  $R_2$

# **Forward & Backward Reasoning**

# Forward and Backward Reasoning

---

- Imperative code made up of
  - assignments (mutation)
  - conditionals
  - loops
- Anything can be rewritten with just these
- We will learn forward / backward rules to handle them
  - will also learn a rule for function calls
  - once we have those, we are done

# Example Forward Reasoning through Assignments

---

```
{{ w > 0 }}  
  x = 17;  
{{ _____ }}  
  y = 42;  
{{ _____ }}  
  z = w + x + y;  
{{ _____ }}
```

- **What do we know is true after  $x = 17$  ?**
  - want the strongest postcondition (most precise)

# Example Forward Reasoning through Assignments

---

↓  
{{ w > 0 }}  
x = 17;  
{{ w > 0 and x = 17 }}  
y = 42;  
{{ \_\_\_\_\_ }}  
z = w + x + y;  
{{ \_\_\_\_\_ }}

- **What do we know is true after  $x = 17$  ?**
  - w was not changed, so  $w > 0$  is still true
  - x is now 17
- **What do we know is true after  $y = 42$  ?**

# Example Forward Reasoning through Assignments

---

```
  {{ w > 0 }}
  x = 17;
  {{ w > 0 and x = 17 }}
  y = 42;
  ↓ {{ w > 0 and x = 17 and y = 42 }}
  z = w + x + y;
  {{ _____ }}
```

- **What do we know is true after  $y = 42$  ?**
  - $w$  and  $x$  were not changed, so previous facts still true
  - $y$  is now 42
- **What do we know is true after  $z = w + x + y$  ?**

# Example Forward Reasoning through Assignments

---

$\{ \{ w > 0 \} \}$

$x = 17;$

$\{ \{ w > 0 \text{ and } x = 17 \} \}$

$y = 42;$

$\{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \} \}$

$z = w + x + y;$

$\{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y \} \}$

- **What do we know is true after  $z = w + x + y$  ?**
  - $w$ ,  $x$ , and  $y$  were not changed, so previous facts still true
  - $z$  is now  $w + x + y$
- **Could also write  $z = w + 59$  (since  $x = 17$  and  $y = 42$ )**

# Example Forward Reasoning through Assignments

---

$\{ \{ w > 0 \} \}$

$x = 17;$

$\{ \{ w > 0 \text{ and } x = 17 \} \}$

$y = 42;$

$\{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \} \}$

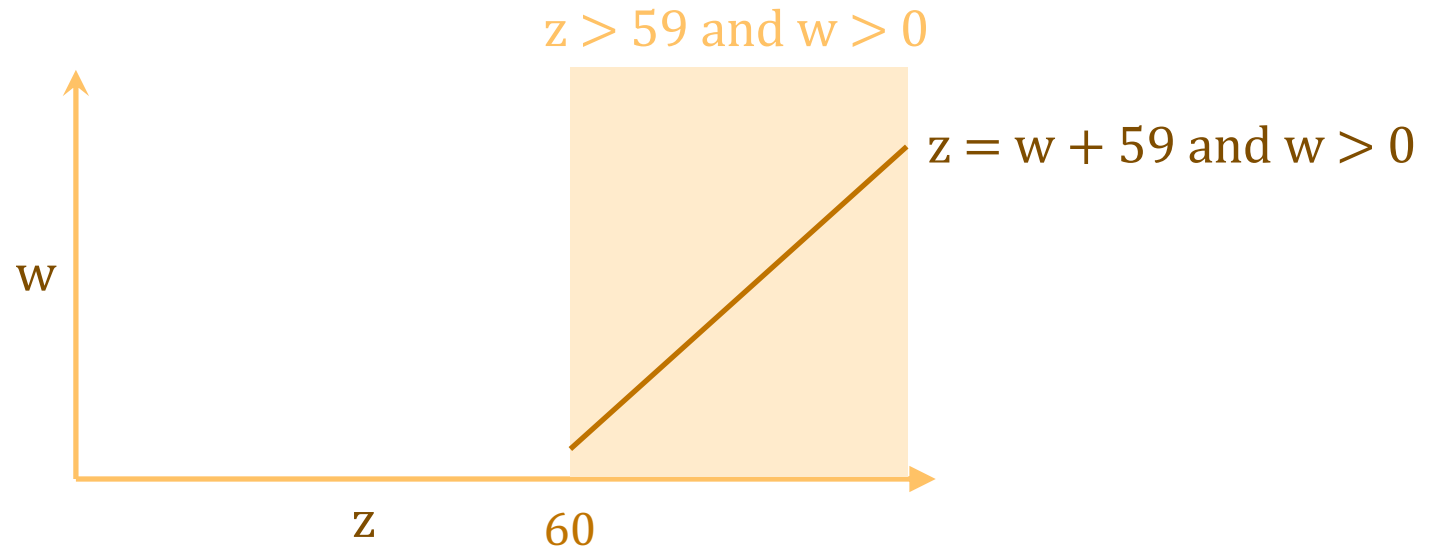
$z = w + x + y;$

$\{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y \} \}$

- **Could write  $z = w + 59$ , but do not write  $z > 59$  !**
  - that is true since  $w > 0$ , but...

# Example Forward Reasoning through Assignments

---



- **Could write  $z = w + 59$ , but do not write  $z > 59$  !**
  - that is true but it is not the strongest postcondition  
correctness check could now fail even if the code is right

# Code Example of Forward Reasoning

---

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
    int x = 17;
    int y = 42;
    int z = w + x + y;
    return z;
};
```

- Let's check correctness using Floyd logic...

# Code Example of Forward Reasoning

---


```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
    {{w > 0}}
    int x = 17;
    int y = 42;
    int z = w + x + y;
    {{z > 59}}
    return z;
};
```

- Reason forward...

# Code Example of Forward Reasoning

---

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
    {{ w > 0 }}
    int x = 17;
    int y = 42;
    int z = w + x + y;
    {{ w > 0 and x = 17 and y = 42 and z = w + x + y }}
    {{ z > 59 }}
    return z;
};
```



- Check implication:

$$\begin{aligned} z &= w + x + y \\ &= w + 17 + y \\ &= w + 59 \\ &> 59 \end{aligned}$$

since  $x = 17$   
since  $y = 42$   
since  $w > 0$

# Code Example of Forward Reasoning

---

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
    int x = 17;
    int y = 42;
    int z = w + x + y;
    return z;
};
```

find facts by reading along path  
from top to return statement

- How about if we use our old approach?
- **Known facts:**  $w > 0$ ,  $x = 17$ ,  $y = 42$ , and  $z = w + x + y$
- **Prove that postcondition holds:**  $z > 59$

# Code Example of Forward Reasoning

---


```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
    int x = 17;
    int y = 42;
    int z = w + x + y;
    return z;
};
```

- We've been doing forward reasoning already!
  - forward reasoning is (only) “and” with *no mutation*
- Line-by-line facts are for mutation (not “**final**”)

# Forward Reasoning through Assignments

---

- **Forward reasoning is trickier with mutation**
  - gets harder if we mutate a variable



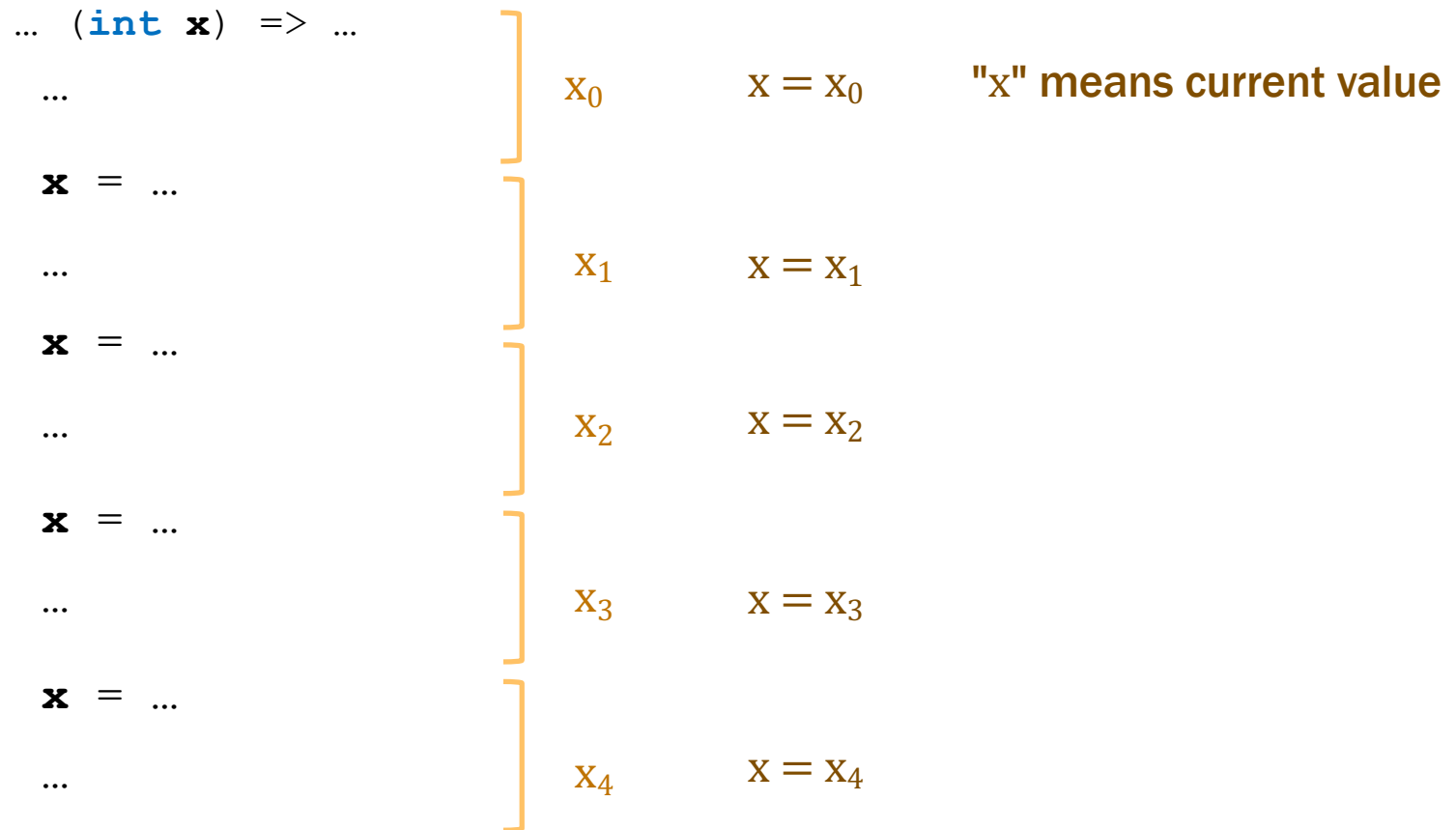
```
w = x + y;  
{{ w = x + y }}  
x = 4;  
{{ w = x + y and x = 4 }}  
y = 3;  
{{ w = x + y and x = 4 and y = 3 }}
```

- **Final assertion is not necessarily true**
  - $w = x + y$  is true with their old values, not the new ones
  - changing the value of “x” can invalidate facts about x
    - facts refer to the old value, not the new value
  - avoid this by using different names for old and new values

# Forward Reasoning through Assignments

---


- Can use subscripts to refer to values at different times



# Forward Reasoning through Assignments

---

- **Rewrite** existing facts to use names of earlier values
  - will use “x” and “y” to refer to current values
  - can use “x<sub>0</sub>” and “y<sub>0</sub>” (or other subscripts) for earlier values



$\{ \{ w = x + y \} \}$   
 $x = 4;$   
 $\{ \{ w = x_0 + y \text{ and } x = 4 \} \}$   
 $y = 3;$   
 $\{ \{ w = x_0 + y_0 \text{ and } x = 4 \text{ and } y = 3 \} \}$

- **Final assertion is now accurate**
  - w is equal to the sum of the initial values of x and y

# Forward Reasoning through Assignments

---

- For assignments, general forward reasoning rule is

$$\begin{array}{l} \{\{ P \}\} \\ \downarrow \\ x = y; \\ \{\{ P[x \mapsto x_k] \text{ and } x = y[x \mapsto x_k] \}\} \end{array}$$

- replace all “x”s in P and y with “x<sub>k</sub>”s
- This process can be simplified in many cases
  - no need for x<sub>0</sub> if we can write it in terms of new value
  - e.g., if “x = x<sub>0</sub> + 1”, then “x<sub>0</sub> = x - 1”
  - assertions will be easier to read without old values  
(Technically, this is weakening, but it’s usually fine  
Postconditions usually do not refer to old values of variables.)

# Forward Reasoning through Assignments

---

- For assignments, general forward reasoning rule is

$$\begin{array}{l} \{\{ P \}\} \\ \downarrow \\ x = y; \\ \{\{ P[x \mapsto x_k] \text{ and } x = y[x \mapsto x_k] \}\} \end{array} \quad x_k \text{ is name of previous value}$$

- If  $x_0 = f(x)$ , then we can simplify this to

$$\begin{array}{l} \{\{ P \}\} \\ \downarrow \\ x = \dots x \dots; \\ \{\{ P[x \mapsto f(x)] \}\} \end{array} \quad \text{no need for, e.g., "and } x = x_0 + 1\text{"}$$

- if assignment is “ $x = x_0 + 1$ ”, then “ $x_0 = x - 1$ ”
- if assignment is “ $x = 2x_0$ ”, then “ $x_0 = x/2$ ”
- does not work for integer division (an un-invertible operation)

# Correctness Example by Forward Reasoning

---

```
/**
 * @param n an integer with  $n \geq 1$ 
 * @returns an integer m with  $m \geq 10$ 
 */
public int f = (int n) {
    {{  $n \geq 1$  }}
    n = n + 3;
    {{  $n - 3 \geq 1$  }}
    {{  $n^2 \geq 10$  }}
    return n * n;
};
```

$n = n_0 + 3$  means  $n - 3 = n_0$

check this implication

$n^2 \geq 4^2$   
 $= 16$   
 $> 10$

since  $n - 3 \geq 1$  (i.e.,  $n \geq 4$ )

This is the preferred approach.  
Avoid subscripts when possible.

# Mutation in Straight-Line Code

---

- Alternative ways of writing this code:

```
n = n + 3;  
return n * n;
```

```
final int n1 = n + 3;  
return n1 * n1;
```

- Mutation in *straight-line* code is unnecessary
  - can always use different names for each value
- Why would we prefer the former?
  - seems like it might save memory...
  - but it doesn't!
    - most compilers will turn the left into the right on their own (SSA form)
    - it's better at saving memory than you are, so it does it itself

# Example Backward Reasoning with Assignments

---

{{ \_\_\_\_\_ }}

x = 17;

{{ \_\_\_\_\_ }}

y = 42;

{{ \_\_\_\_\_ }}

z = w + x + y;

{{ z < 0 }}

- **What must be true before  $z = w + x + y$  so  $z < 0$  ?**
  - want the weakest precondition (most allowed states)

# Example Backward Reasoning with Assignments

---

```


  {{ _____ }}
  x = 17;
  {{ _____ }}
  y = 42;
  {{ w + x + y < 0 }}
  ↑
  z = w + x + y;
  {{ z < 0 }}

```

- **What must be true before  $z = w + x + y$  so  $z < 0$  ?**
  - must have  $w + x + y < 0$  beforehand
- **What must be true before  $y = 42$  for  $w + x + y < 0$  ?**

# Example Backward Reasoning with Assignments

---

$\{\{ \text{_____} \}\}$   
 $x = 17;$   
  $\{\{ w + x + 42 < 0 \}\}$   
 $y = 42;$   
 $\{\{ w + x + y < 0 \}\}$   
 $z = w + x + y;$   
 $\{\{ z < 0 \}\}$

- **What must be true before  $y = 42$  for  $w + x + y < 0$  ?**
  - must have  $w + x + 42 < 0$  beforehand
- **What must be true before  $x = 17$  for  $w + x + 42 < 0$  ?**

# Example Backward Reasoning with Assignments

---

↑  
{{  $w + 17 + 42 < 0$  }}  
   $x = 17;$   
{{  $w + x + 42 < 0$  }}  
   $y = 42;$   
{{  $w + x + y < 0$  }}  
   $z = w + x + y;$   
{{  $z < 0$  }}

- **What must be true before  $x = 17$  for  $w + x + 42 < 0$  ?**
  - must have  $w + 59 < 0$  beforehand
- **All we did was substitute right side for the left side**
  - e.g., substitute “ $w + x + y$ ” for “ $z$ ” in “ $z < 0$ ”
  - e.g., substitute “42” for “ $y$ ” in “ $w + x + y < 0$ ”
  - e.g., substitute “17” for “ $x$ ” in “ $w + x + 42 < 0$ ”

# Backward Reasoning through Assignments

---

- For assignments, backward reasoning is substitution

↑  
{{  $Q[x \mapsto y]$  }}  
   $x = y;$   
{{  $Q$  }}

- just replace all the “x”s with “y”s
  - we will denote this substitution by  $Q[x \mapsto y]$
- Mechanically simpler than forward reasoning
    - no need for subscripts

# Correctness Example by Backward Reasoning

---

```
/**
 * @param n an integer with  $n \geq 1$ 
 * @returns an integer m with  $m \geq 10$ 
 */
public int f(int n) {
    {{  $n \geq 1$  }}
    n = n + 3;
    {{  $n^2 \geq 10$  }}
    return n * n;
};
```

- Code is correct if this triple is valid...

# Correctness Example by Backward Reasoning

---

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
    {{ n ≥ 1 }}
    {{ (n + 3)2 ≥ 10 }}
    ↑
    n = n + 3;
    {{ n2 ≥ 10 }}
    return n * n;
};
```

check this implication

$$\begin{aligned}(n+3)^2 &\geq (1+3)^2 && \text{since } n \geq 1 \\ &= 16 \\ &> 10\end{aligned}$$

# Correctness Example by Forward Reasoning

---

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
    {{ n ≥ 1 }}
    n = n + 3;
    {{ n - 3 ≥ 1 }}
    {{ n² ≥ 10 }}
    return n * n;
};
```

check this implication

$$\begin{aligned}n^2 &\geq 4^2 \\ &= 16 \\ &> 10\end{aligned}$$

since  $n - 3 \geq 1$  (i.e.,  $n \geq 4$ )

Forward reasoning produces known facts.  
Backward reasoning produces fact to prove.

# Conditionals

# Conditionals in Functional Programming

---


```
// Inputs a and b must be integers.  
// Returns a non-negative integer.  
public int f(int a, int b) {  
    if (a >= 0 && b >= 0) {  
        final List L = cons(a, cons(b, nil));  
        return sum(L);  
    }  
    ...  
}
```

- **Prior reasoning also included *conditionals***
  - what does that look like in Floyd logic?

# Conditionals in Floyd Logic

---

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  {}
  if (a >= 0 && b >= 0) {
    {a ≥ 0 and b ≥ 0}
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
  ...
}
```



- **Conditionals introduce extra facts in forward reasoning**
  - simple “and” since nothing is mutated

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
    int m;
    if (n >= 0) {
        m = 2 * n + 1;
    } else {
        m = 0;
    }
    return m;
}
```

- Code like this was impossible without mutation
  - cannot write to a “**final**” after its declaration
- How do we handle it now?

# Conditionals in Floyd Logic

---


```
// Returns an integer m with m > n
public int g(int n) {
    int m;
    if (n >= 0) {
        m = 2 * n + 1;
    } else {
        m = 0;
    }
    return m;
}
```

- Reason *separately* about each **path** to a **return**
  - handle each path the same as before
  - but now there can be multiple paths to one **return**

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
    {}
    int m;
    if (n >= 0) {
        m = 2 * n + 1;
    } else {
        m = 0;
    }
    {{ m > n }}
    return m;
}
```


A diagram consisting of a vertical orange line on the left side of the code block. It starts at the level of the opening curly brace of the function, goes down to the level of the opening curly brace of the if statement, then turns right to the level of the opening curly brace of the 'then' branch, then turns down to the level of the closing curly brace of the 'then' branch, and finally turns right to point at the closing curly brace of the if statement.

- Check correctness path through “then” branch

# Conditionals in Floyd Logic

---


```
// Returns an integer m with m > n
public int g(int n) {
    {}
    int m;
    if (n >= 0) {
        {{n ≥ 0}}
        m = 2 * n + 1;
    } else {
        m = 0;
    }
    {{m > n}}
    return m;
}
```



# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
    {}
    int m;
    if (n >= 0) {
        {{ n ≥ 0 }}
        m = 2 * n + 1;
        {{ n ≥ 0 and m = 2n + 1 }}
    } else {
        m = 0;
    }
    {{ m > n }}
    return m;
}
```

A diagram consisting of a vertical line on the left side of the code block, starting from the level of the 'if' statement and extending down to the level of the assignment 'm = 2 \* n + 1;'. From the bottom of this vertical line, a horizontal line extends to the right, ending in an arrowhead that points to the assignment statement.

# Conditionals in Floyd Logic

---


```
// Returns an integer m with m > n
public int g(int n) {
  {}
  int m;
  if (n >= 0) {
    {{ n ≥ 0 }}
    m = 2 * n + 1;
    {{ n ≥ 0 and m = 2n + 1 }}
  } else {
    m = 0;
  }
  {{ n ≥ 0 and m = 2n + 1 }}
  {{ m > n }}
  return m;
}
```

$m = 2n + 1$   
 $> 2n$       since  $1 > 0$   
 $\geq n$         since  $n \geq 0$

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
    {}
    int m;
    if (n >= 0) {
        m = 2 * n + 1;
    } else {
        m = 0;
    }
    {{ n ≥ 0 and m = 2n + 1 }}
    {{ m > n }}
    return m;
}
```

A diagram consisting of a vertical line on the left side of the code block. It starts at the level of the opening curly brace of the function, goes down, then turns right to point at the opening curly brace of the if statement. It then goes down again, then turns right to point at the closing curly brace of the if statement. Finally, it goes down and turns right to point at the return statement.

- Note: **no mutation**, so we can do this in our head
  - read along the **path**, and collect all the facts

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
    {}
    int m;
    if (n >= 0) {
        m = 2 * n + 1;
    } else {
        m = 0;
    }
    {{ n < 0 and m = 0 }}
    {{ m > n }}
    return m;
}
```

$m = 0$   
 $> n$       since  $0 > n$

- Check correctness path through “else” branch
  - note: **no mutation**, so we can do this in our head

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
  {}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
    {{ n ≥ 0 and m = 2n + 1 }}
  } else {
    m = 0;
    {{ n < 0 and m = 0 }}
  }
  {{ _____ }}
  {{ m > n }}
  return m;
}
```

What do we know is true  
even if we don't know  
which branch was taken?

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
  {}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  {(n ≥ 0 and m = 2n + 1) or (n < 0 and m = 0)}
  {m > n}
  return m;
}
```

- The “or” means we must reason by cases anyway!

# Conditionals in Floyd Logic

---

```

{{ P }}
if (cond) {
    {{ P and cond }}
    S1
    {{ A }}
} else {
    {{ P and not cond }}
    S2
    {{ B }}
}
{{ A or B }}
{{ Q }}
```

- **Postcondition is of the form** `{{ A or B }}`
  - `A` being what we know if we had taken the `if` branch
  - `B` being what we know if we had taken the `else`

# Conditionals *Backward* in Floyd Logic

---

```

{{ P }}
{{ (A and cond) or (B and not cond) }}
if (cond) {
    {{ A }}
    S1
    {{ Q }}
} else {
    {{ B }}
    S2
    {{ Q }}
}
{{ Q }}

```

- **Precondition** is of the form  $\{\{ (A \text{ and } \text{cond}) \text{ or } (B \text{ and not cond}) \}\}$ 
  - **A** being what must hold if we take the **if** branch
  - **B** being what must hold if we take the **else**

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
  {}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    return 0;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and ??) }}
  {{ m > n }}
  return m;
}
```

- What is the state after a “**return**”?

# Conditionals in Floyd Logic

---

```
// Returns an integer m with m > n
public int g(int n) {
  {}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    return 0;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and false) }}
  {{ m > n }}
  return m;
}
```

simplifies to just  $n \geq 0$  and  $m = 2n + 1$

- State after a “**return**” is false (no states)

# Conditionals With Returns

---

- Latter rule for "if .. return" is useful:

```
  {{P}}  
  if (cond)  
    return something;  
  {{ P and not cond }}  
  ...  
  return something else;
```

- Only reach the line after the "if" if `cond` was false
- Only one path to each "return" statement
  - forward reason to the "return" inside the "if"
  - forward reason to the "return" after the "if"

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        m = m * -1;
    } else if (x == 0) {
        return 1;
    }
    { _____ }
    m = m + 1;
    {{ m > 0 }}
    return m;
}
```

How many paths can  
the code take?

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
  {}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } else {
    // do nothing
  }
  {{_____ or _____ or _____}}
  m = m + 1;
  {{m > 0}}
  return m;
}
```

3 paths! **else** branch is not written out, but it's there implicitly

After the conditional, there are 3 sets of facts that could be true

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        {}
        m = m * -1;
        {}
    } else if (x == 0) {
        return 1;
    } // else: do nothing
    {} or {} or {}
    m = m + 1;
    {} m > 0 {}
    return m;
}
```

The diagram consists of a vertical line on the left side. From the top of this line, a horizontal line extends to the right, then a vertical line goes down to the level of the first curly brace of the 'if' block. From there, a horizontal line extends to the right, then a vertical line goes down to the level of the 'return 1;' statement. Finally, a horizontal line extends to the right, then a vertical line goes down to the level of the 'return m;' statement, ending in an arrowhead pointing downwards.

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        {{ m = x and x < 0 }}
        m = m * -1;
        {{ _____ }}
    } else if (x == 0) {
        return 1;
    } // else: do nothing
    {{ _____ or _____ or _____ }}
    m = m + 1;
    {{ m > 0 }}
    return m;
}
```

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
  {}
  int m = x;
  if (x < 0) {
    {m = x and x < 0}
    m = m * -1;
    {m = -x and x < 0}
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {(m = -x and x < 0) or _____ or _____}
  m = m + 1;
  {m > 0}
  return m;
}
```

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
  {}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    {}
    return 1;
  } // else: do nothing
  {} (m = - x and x < 0) or _____ or _____ {}
  m = m + 1;
  {} m > 0 {}
  return m;
}
```

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        m = m * -1;
    } else if (x == 0) {
        {} x = 0 and m = x {}
        return 1;
    } // else: do nothing
    {} (m = - x and x < 0) or _____ or _____ {}
    m = m + 1;
    {} m > 0 {}
    return m;
}
```

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        m = m * -1;
    } else if (x == 0) {
        {{ x = 0 and m = x }}
        return 1;
    } else {
        // else: do nothing
    }
    {{ (m = - x and x < 0) or (x = 0 and m = x and false) or _____ }}
    m = m + 1;
    {{ m > 0 }}
    return m;
}
```

Must prove that post condition holds here

false: no states can reach beyond return

# Conditionals in Floyd Logic

---


```
// Returns an integer m, with m > 0
public int h(int x) {
  {}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ (m = - x and x < 0) or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

What do we know in  
implicit else case?  
When *neither* of the then  
cases were entered

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        m = m * -1;
    } else if (x == 0) {
        return 1;
    } // else: do nothing
    {{ (m = -x and x < 0) or (x > 0 and m = x) }}
    m = m + 1;
    {{ m > 0 }}
    return m;
}
```

A diagram consisting of a vertical line on the left side of the code block. It starts at the level of the 'else if' block, moves down, then right, then down again, ending with an arrowhead pointing to the first line of the Floyd logic block: `{{ (m = -x and x < 0) or (x > 0 and m = x) }}`.

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
  {}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ (m = -x and x < 0) or (x > 0 and m = x) }}
  {{ _____ }}
  ↑
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

Can reason backward and forward  
and meet in the middle

# Conditionals in Floyd Logic

---

```
// Returns an integer m, with m > 0
public int h(int x) {
    {}
    int m = x;
    if (x < 0) {
        m = m * -1;
    } else if (x == 0) {
        return 1;
    } // else: do nothing
    {{ (m = -x and x < 0) or (x > 0 and m = x) }}
    {{ m + 1 > 0 }}
    m = m + 1;
    {{ m > 0 }}
    return m;
}
```

check this implication

Does the set of facts we know at this point in the program satisfy what must be true to reach our post condition

# Conditionals in Floyd Logic

---

- Prove by cases

$$\{ \{ (m = -x \text{ and } x < 0) \text{ or } (x > 0 \text{ and } m = x) \} \}$$
$$\{ \{ m + 1 > 0 \} \}$$

**Case 1:**  $m = -x$  and  $x < 0$

$$\begin{aligned} m + 1 &= -x + 1 && \text{since } m = -x \\ &> 1 && \text{since } x < 0 \\ &> 0 \end{aligned}$$

**Case 2:**  $x > 0$  and  $m = x$

$$\begin{aligned} m + 1 &= x + 1 && \text{since } m = x \\ &> 1 && \text{since } x > 0 \\ &> 0 \end{aligned}$$

- Already proved for the branch with the return, so proved the postcondition holds, in general

# Loops

# Correctness of Loops

---

- **Assignment and condition reasoning is mechanical**
- **Loop reasoning cannot be made mechanical**
  - no way around this  
(311 alert: this follows from Rice's Theorem)
- **Thankfully, one *extra* bit of information fixes this**
  - need to provide a “loop invariant”
  - with the invariant, reasoning is again mechanical

# Loop Invariants

---

- Loop invariant is true every time at the top of the loop

```
  {{ Inv: I }}  
  while (cond) {  
    S  
  }
```

- must be true when we get to the top the first time
  - must remain true each time execute S and loop back up
- Use “Inv:” to indicate a loop invariant  
 otherwise, this only claims to be true the first time at the loop

# Loop Invariants

---

- Loop invariant is true every time at the top of the loop

```
  {{ Inv: I }}  
  while (cond) {  
    S  
  }
```

- must be true 0 times through the loop (at top the first time)
  - if true n times through, must be true n+1 times through
- Why do these imply it is always true?
    - follows by structural induction (on  $\mathbb{N}$ )

# Checking Correctness with Loop Invariants

---

```
  {{ P }}  
  {{ Inv: I }}  
  while (cond) {  
    S  
  }  
  {{ Q }}
```

- How do we check validity with a loop invariant?
  - intermediate assertion splits into *three* triples to check

# Checking Correctness with Loop Invariants

---

```
  {{ P }}  
  {{ Inv: I }}  
  while (cond) {  
    S  
  }  
  {{ Q }}
```

1. I holds initially

**Splits correctness into three parts**

1. I holds initially
2. S preserves I
3. Q holds when loop exits

# Checking Correctness with Loop Invariants

---

```
  {{ P }}  
  {{ Inv: I }}  
  while (cond) {  
    {{ I and cond }}  
    S  
    {{ I }}  
  }  
  {{ Q }}
```

1. I holds initially

2. S preserves I

Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits

# Checking Correctness with Loop Invariants

---

```

{{ P }}
{{ Inv: I }}
while (cond) {
  {{ I and cond }}
  S
  {{ I }}
}
{{ I and not cond }}
{{ Q }}

```

1. I holds initially

2. S preserves I

3. Q holds when loop exits

## Splits correctness into three parts

1. I holds initially      implication
2. S preserves I      forward/back then implication
3. Q holds when loop exits      implication

# Checking Correctness with Loop Invariants

---

```
  {{ P }}  
  {{ Inv: I }}  
  while (cond) {  
    S  
  }  
  {{ Q }}
```

**Formally, invariant split this into three Hoare triples:**

1.  $\{\{ P \}\} \{\{ I \}\}$       **I holds initially**
2.  $\{\{ I \text{ and } \text{cond} \}\} S \{\{ I \}\}$       **S preserves I**
3.  $\{\{ I \text{ and not cond} \}\} \{\{ Q \}\}$       **Q holds when loop exits**

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```
{{ }}  
int j = 0;  
int s = 0;  
{{ Inv:  $s = j^2$  }}  
while (j != n) {  
    j = j + 1;  
    s = s + j + j - 1;  
}  
{{  $s = n^2$  }}
```

**Easy to get this wrong!**

- might be initializing “j” wrong ( $j = 1$ ?)
- might be exiting at the wrong time ( $j \neq n-1$ ?)
- might have the assignments in wrong order
- ...

**Fact that we need to check 3 implications is a strong indication that more bugs are possible.**

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```
{}  
int j = 0;  
int s = 0;  
{ { Inv:  $s = j^2$  } }  
while (j != n) {  
    j = j + 1;  
    s = s + j + j - 1;  
}  
{ {  $s = n^2$  } }
```

## Loop Idea

- move  $j$  from 0 to  $n$
- keep track of  $j^2$  in  $s$

$j$	$s$
0	0
1	1
2	4
3	9
4	16
...	...

Loop Invariant formalizes the Loop Idea

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```
  {{ }}
  int j = 0;
  int s = 0;
  {{j = 0 and s = 0}}
  {{ Inv: s = j2 }}
  while (j != n) {
    j = j + 1;
    s = s + j + j - 1;
  }
  {{ s = n2 }}
```

] s = 0  
= 0<sup>2</sup>  
= j<sup>2</sup>

since j = 0

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```

{{ Inv:  $s = j^2$  }}
while (j != n) {
    j = j + 1;
    s = s + j + j - 1;
}

```

{{  $s = j^2$  and  $j = n$  }}
]
 $s = j^2$   
 $= n^2$ 
since  $j = n$

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```
  {{ Inv:  $s = j^2$  }}  
  while (j != n) {  
    {{  $s = j^2$  and  $j \neq n$  }}  
    j = j + 1;  
    s = s + j + j - 1;  
    {{  $s = j^2$  }}  
  }  
  {{  $s = n^2$  }}
```

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```
  {{ Inv:  $s = j^2$  }}
  while (j != n) {
    {{  $s = j^2$  and  $j \neq n$  }}
    j = j + 1;
    {{  $s = (j - 1)^2$  and  $j - 1 \neq n$  }}
    s = s + j + j - 1;
    {{  $s = j^2$  }}
  }
  {{  $s = n^2$  }}
```

$j = j_0 + 1$  means  $j_0 = j - 1$

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

$\{\{ \text{Inv: } s = j^2 \}\}$

**while** ( $j \neq n$ ) {

$\{\{ s = j^2 \text{ and } j \neq n \}\}$

$j = j + 1;$

$\{\{ s = (j - 1)^2 \text{ and } j - 1 \neq n \}\}$

$s = s + j + j - 1;$

$s = s_0 + 2j - 1$  means  $s_0 = s - 2j + 1$

$\{\{ s - 2j + 1 = (j - 1)^2 \text{ and } j - 1 \neq n \}\}$

$\{\{ s = j^2 \}\}$

}

$\{\{ s = n^2 \}\}$

# Loop Correctness Example 1

---

- This loop claims to calculate  $n^2$

```

{{ Inv:  $s = j^2$  }}
while (j != n) {
  {{  $s = j^2$  and  $j \neq n$  }}
  j = j + 1;
  {{  $s = (j - 1)^2$  and  $j - 1 \neq n$  }}
  s = s + j + j - 1;
  {{  $s - 2j + 1 = (j - 1)^2$  and  $j - 1 \neq n$  }}
  {{  $s = j^2$  }}
}
{{  $s = n^2$  }}

```

$s = 2j - 1 + (j - 1)^2$   
 $= 2j - 1 + j^2 - 2j + 1$   
 $= j^2$

since  $s - 2j + 1 = (j - 1)^2$

# Loop Correctness Example 2

---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- This loop claims to calculate it as well:

```
{{ L = L0 }}
int s = 0;
{{ Inv: sum(L0) = s + sum(L) }}
while (L != null) {
    s = s + L.hd;
    L = L.tl;
}
{{ s = sum(L0) }}
```

## Loop Idea

- move through L front-to-back
- keep sum of *prior* part in s

# Loop Correctness Example 2

---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the invariant holds initially

```

  {{ L = L0 }}
  int s = 0;
  {{ L = L0 and s = 0 }}
  {{ Inv: sum(L0) = s + sum(L) }}
  while (L != null) {
    ...

```

↓

$\text{sum}(L_0)$	
$= \text{sum}(L)$	since $L = L_0$
$= 0 + \text{sum}(L)$	
$= s + \text{sum}(L)$	since $s = 0$

# Loop Correctness Example 2

---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the postcondition holds at loop exit

```
{ { Inv: sum(L0) = s + sum(L) } }
while (L != null) {
  s = s + L.hd;
  L = L.tl;
}
{ { sum(L0) = s + sum(L) and L = nil } }
{ { s = sum(L0) } }
```

$\sum(L_0)$   
= s + sum(L)  
= s + sum(nil)    since L = nil  
= s                def of sum

# Loop Correctness Example 2

---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the loop body preserves the invariant

```
{{ Inv: sum(L0) = s + sum(L) }}
while (L != null) {
  {{ sum(L0) = s + sum(L) and L ≠ nil }}
  s = s + L.hd;
  L = L.tl;
  {{ sum(L0) = s + sum(L) }}
}
```

**L ≠ nil means L = L.hd :: L.tl**

# Loop Correctness Example 2

---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the loop body preserves the invariant

```
{{ Inv: sum(L0) = s + sum(L) }}
while (L != null) {
  {{ sum(L0) = s + sum(L) and L = L.hd :: L.tl }}
  s = s + L.hd;
  L = L.tl;
  {{ sum(L0) = s + sum(L) }}
}
```

# Loop Correctness Example 2


---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the loop body preserves the invariant

```
  {{ Inv: sum(L0) = s + sum(L) }}
  while (L != null) {
    {{ sum(L0) = s + sum(L) and L = L.hd :: L.tl }}
    s = s + L.hd;
    {{ sum(L0) = s + sum(L.tl) }}
    L = L.tl;
    {{ sum(L0) = s + sum(L) }}
  }
```



# Loop Correctness Example 2


---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the loop body preserves the invariant

```
  {{ Inv: sum(L0) = s + sum(L) }}
  while (L != null) {
    {{ sum(L0) = s + sum(L) and L = L.hd :: L.tl }}
    {{ sum(L0) = s + L.hd + sum(L.tl) }}
    s = s + L.hd;
    {{ sum(L0) = s + sum(L.tl) }}
    L = L.tl;
    {{ sum(L0) = s + sum(L) }}
  }
```



# Loop Correctness Example 2

---

- Recursive function to calculate sum of list

```
sum(nil)    := 0
sum(x :: L) := x + sum(L)
```

- Check that the loop body preserves the invariant

```
{ { Inv: sum(L0) = s + sum(L) } }
while (L != null) {
  { { sum(L0) = s + sum(L) and L = L.hd :: L.tl } }
  { { sum(L0) = s + L.hd + sum(L.tl) } }
  s = s + L.hd;
  { { sum(L0) = s + sum(L.tl) } }
  L = L.tl;
  { { sum(L0) = s + sum(L) } }
}
```

$\text{sum}(L_0)$   
=  $s + \text{sum}(L)$   
=  $s + \text{sum}(L.\text{hd} :: L.\text{tl})$  since  $L = L.\text{hd} :: L.\text{tl}$   
=  $s + L.\text{hd} + \text{sum}(L.\text{tl})$  def of sum

# Loop Correctness Example 3

---

- Recursive function to check if  $y$  appears in list  $L$

```
contains(y, nil)    := false
contains(y, x :: L) := true           if  $x = y$ 
contains(y, x :: L) := contains(y, L) if  $x \neq y$ 
```

- This loop claims to calculate it as well:

```
{ { Inv: contains(y, L0) = contains(y, L) } }
while (L != null) {
  if (L.hd == y)
    return true;
  L = L.tl;
}
return false;
```

## Loop Idea

- move through  $L$  front-to-back
- answer remains the same as on the original list  $L_0$
- can only do that if  $y$  is *not* found

# Loop Correctness Example 3

---

- Check that the invariant holds initially

```
  {{ L0 = L }}  
  {{ Inv: contains(y, L0) = contains(y, L) }} ]  
  while (L != null) {  
    if (L.hd == y)  
      return true;  
    L = L.tl;  
  }  
  return false;
```

contains(y, L<sub>0</sub>)  
= contains(y, L)      since L<sub>0</sub> = L

```
contains(y, nil)    := false  
contains(y, x :: L) := true      if x = y  
contains(y, x :: L) := contains(y, L)    if x ≠ y
```

# Loop Correctness Example 3

---

- Check that the invariant implies the postcondition

```
  {{ Inv: contains(y, L0) = contains(y, L) }}
```

```
  while (L != null) {
```

```
    if (L.hd == y)
```

```
      return true;
```

```
    L = L.tl;
```

```
  }
```

```
  {{ contains(y, L0) = contains(y, L) and L = nil }}
```

```
  {{ contains(y, L0) = false }}
```

```
  return false;
```

contains(y, L<sub>0</sub>)

= contains(y, L)

= contains(y, nil)

= false

since L = nil

def of contains

contains(y, nil) := false

contains(y, x :: L) := true

contains(y, x :: L) := contains(y, L)

if x = y

if x ≠ y

# Loop Correctness Example 3

---

- Check that the body preserves the invariant

```

{{ Inv: contains(y, L0) = contains(y, L) }}
while (L != null) {
  {{ contains(y, L0) = contains(y, L) and L ≠ nil }}
  if (L.hd == y)
    return true;
  L = L.tl;
  {{ contains(y, L0) = contains(y, L) }}
}
return false;

```

$L \neq \text{nil}$  means  $L = L.\text{hd} :: L.\text{tl}$

<code>contains(y, nil)</code>	<code>:= false</code>	
<code>contains(y, x :: L)</code>	<code>:= true</code>	<code>if x = y</code>
<code>contains(y, x :: L)</code>	<code>:= contains(y, L)</code>	<code>if x ≠ y</code>

# Loop Correctness Example 3

---

- Check that the body preserves the invariant

```

{{ Inv: contains(y, L0) = contains(y, L) }}
while (L != null) {
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl }}
  if (L.hd == y)
    {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl and L.hd = y }}
    {{ contains(y, L0) = true }}
    return true;
  L = L.tl;
  {{ contains(y, L0) = contains(y, L) }}
}
return false;
```

contains(y, nil) := false

contains(y, x :: L) := true

contains(y, x :: L) := contains(y, L)

contains(y, L<sub>0</sub>)

= contains(y, L)

= contains(y, L.hd :: L.tl)

= true

if x = y

if x ≠ y

since L = L.hd :: L.tl

def of contains

(since y = L.hd)

# Loop Correctness Example 3

---

- Check that the body preserves the invariant

```

{{ Inv: contains(y, L0) = contains(y, L) }}
while (L != null) {
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl }}
  if (L.hd == y)
    {{ contains(y, L0) = true }}
    return true;
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl and L.hd ≠ y }}
  L = L.tl;
  {{ contains(y, L0) = contains(y, L) }}
}
return false;

```

```

contains(y, nil)    := false
contains(y, x :: L) := true           if x = y
contains(y, x :: L) := contains(y, L) if x ≠ y

```

# Loop Correctness Example 3

---

- Check that the body preserves the invariant

```

{{ Inv: contains(y, L0) = contains(y, L) }}
while (L != null) {
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl }}
  if (L.hd == y)
    {{ contains(y, L0) = true }}
    return true;
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl and L.hd ≠ y }}
  {{ contains(y, L0) = contains(y, L.tl) }}
  L = L.tl;
  {{ contains(y, L0) = contains(y, L) }}
}
return false;

```

contains(y, nil) := false

contains(y, x :: L) := true

contains(y, x :: L) := contains(y, L)

if x = y

if x ≠ y

contains(y, L<sub>0</sub>)

= contains(y, L)

= contains(y, L.hd :: L.tl)

= contains(y, L.tl)

since L = L.hd :: L.tl

def of contains

(since y ≠ L.hd)

# Loop Correctness Example 4

---

- **Declarative spec of  $\text{sqrt}(x)$**

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- precondition that  $x$  is positive:  $0 < x$
- precondition that  $x$  is not too large:  $x < 10^{12} = (10^6)^2$

# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- This loop claims to calculate it:

```
int a = 0;
int b = 1000000;
{{ Inv:  $a^2 < x \leq b^2$  }}
while (a != b - 1) {
    int m = (a + b) / 2;
    if (m*m < x) {
        a = m;
    } else {
        b = m;
    }
}
return b;
```

## Loop Idea

- maintain a range  $a \dots b$   
with  $x$  in the range  $a^2 \dots b^2$

# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- Check that the invariant holds initially:

```
  {{ Pre:  $0 < x \leq 10^{12}$  }}  
  int a = 0;  
  int b = 1000000;  
  {{ Inv:  $a^2 < x \leq b^2$  }}  
  while (a != b - 1) {  
    ...  
  }  
  return b;
```



# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- Check that the postcondition hold after exit

```
  {{ Inv:  $a^2 < x \leq b^2$  }}  
  while (a != b - 1) {  
    ...  
  }  
  {{  $a^2 < x \leq b^2$  and  $a = b - 1$  }}  
  {{  $(b - 1)^2 < x \leq b^2$  }}  
  return b;
```

Does  $(y - 1)^2 < x < y^2$  hold with  $y = b$ ?

$$\begin{aligned} &(b - 1)^2 \\ &= a^2 \quad \text{since } a = b - 1 \\ &< x \end{aligned}$$

# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- Check that the body preserves the invariant:

```

{{ Inv:  $a^2 < x \leq b^2$  }}
while (a != b - 1) {
  {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  }}
  int m = (a + b) / 2;
  if (m*m < x) {
    a = m;
  } else {
    b = m;
  }
  {{  $a^2 < x \leq b^2$  }}
}

```

# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- Check that the body preserves the invariant:

```

{{ Inv:  $a^2 < x \leq b^2$  }}
while (a != b - 1) {
  {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  }}
  int m = (a + b) / 2;
  if (m*m < x) {
    {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  and  $m^2 < x$  }}
    a = m;
  } else {
    {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  and  $x \leq m^2$  }}
    b = m;
  }
  {{  $a^2 < x \leq b^2$  }}
}

```

# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- Check that the body preserves the invariant:

```

{{ Inv:  $a^2 < x \leq b^2$  }}
while (a != b - 1) {
  int m = (a + b) / 2;
  if (m*m < x) {
    {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  and  $m^2 < x$  }}
    {{  $m^2 < x \leq b^2$  }}
    a = m;
  } else {
    {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  and  $x \leq m^2$  }}
    b = m;
  }
  {{  $a^2 < x \leq b^2$  }}
}

```

Immediate!

# Loop Correctness Example 4

---

return  $y \in \mathbb{Z}$  such that  $(y - 1)^2 < x \leq y^2$

- Check that the body preserves the invariant:

```

{{ Inv:  $a^2 < x \leq b^2$  }}
while (a != b - 1) {
  int m = (a + b) / 2;
  if (m*m < x) {
    a = m;
  } else {
    {{  $a^2 < x \leq b^2$  and  $a \neq b - 1$  and  $x \leq m^2$  }}
    {{  $a^2 < x \leq m^2$  }}
    b = m;
  }
  {{  $a^2 < x \leq b^2$  }}
}

```

Immediate!

Correctness of binary search is pretty easy  
once you have the invariant clear!

# Termination

---

- **This analysis does not check that the code terminates**
  - it shows that the postcondition holds if the loop exits
  - but we never showed that the loop does exit
- **Termination follows from the running time analysis**
  - e.g., if the code runs in  $O(n^2)$  time, then it terminates
  - an infinite loop would be  $O(\text{infinity})$
  - any finite bound on the running time proves it terminates
- **Normal to also analyze the running time of our code, and we get termination already from that analysis**

# Correctness of Loops

---

- With straight-line code and conditionals, if the triple is not valid...
  - the code is **wrong**
  - there is *some* test case that will prove it  
(doesn't mean we found that case in our tests, but it exists)
- With loops, if the triples are not valid...
  - the code is **wrong** *with that invariant*
  - there may **not** be any test case that proves it  
the code may behave correctly on all inputs
  - the code could be right but with a *different* invariant
- Loops are inherently more complicated