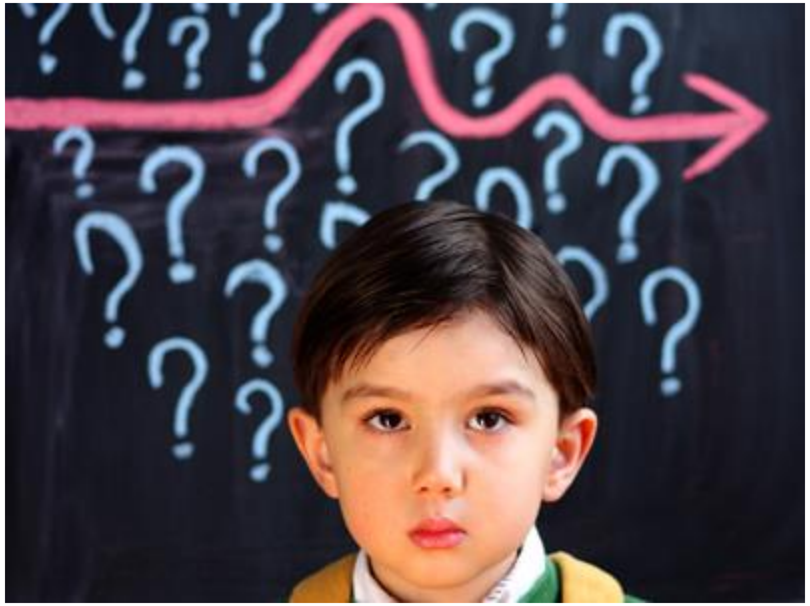
Reasoning About Code



Reasoning About Code

- Determines **before** execution what facts hold during program execution
- Reason about *conditions*:

    `0 <= index < names.length`

    `x > 0`

    array **`names`** is sorted

    `x > y`

`These are all conditions which could be true or false`

Why Reason About Code

- Our goal is to produce correct code!
- Two ways to ensure correctness
  - Testing
    - Can find bugs but doesn't guarantee code is bug free
  - Reasoning about code
    - Verification
- Reasoning about code
  - Verifies that code works correctly
  - Finds errors in code
    - Aids debugging
  - Helps understand errors

Specifications

- What does it mean for code to be correct?
  - (Informally) Code is correct if it conforms to its specification
- A *specification* consists of a precondition and a postcondition
  - **Precondition**: conditions that must hold before code executes
  - **Postcondition**: conditions that must hold after code finishes execution (if precondition held!)
- Precondition and Postcondition
  - Logical constraint on values

Specifications

Notation:
&& denotes logical AND
|| denotes logical OR

Precondition: arr != null && `arr.length = len && len > 0`

Postcondition: `result = arr[0]+…+arr[arr.length-1]`

```
// sum contents of arr
int sum(int[] arr, int len) {
    int result = 0;
    int i = 0;
    while (i < len) {
        result = result + arr[i];
        i = i+1;
    }
    return result;
}
```

To prove that `sum` is correct, we must prove that the implementation meets the specification. In other words, we must prove that if the precondition held, then after code finishes execution, the postcondition holds.

Specifications

- The specification is a contract between the function and its caller. Both caller and function have obligations:
  - Caller must pass arguments that obey the precondition.
  - If not, all bets are off --- function can break or return wrong result!
  - Function “promises” the postcondition, if precondition holds

  - In **`sum`**, how can the caller violate spec?
  - How can **`sum`** violate spec?

Type Signature is a Form of Specification

- Type signature is a contract too!
- `int sum(int[] arr, int len) {…}`
  - Precondition: arguments are an array of **`int`**s and an **`int`**
  - Postcondition: result is an **`int`**
- Java enforces the type constraint at compile time

- We need more than type signatures!
  - We need reasoning about behavior and effects (deeper properties)

Type Signature is a Specification

- Type checker (among other things) verifies that the parties meet the type contract
- If language is type safe we can “trust” the type checker
- But if language is type unsafe it would be possible for a caller to pass an argument of the wrong type!
- Python allows you to pass an object that might not have the needed methods or worse have a method of the same name that does something different than expected.
- Java catches argument type violations at compile time
- Python catches argument type violations at runtime

What is Wrong With this Code?

```
class NameList {
    int index;
    String[] names;
    …
    // Precondition: 0 ≤ index < names.length
    void addName(String name) {
        index++;
        if (index < names.length){
                names[index] = name;
        }
    }
    // Postcondition: 0 ≤ index < names.length
}
```

Is there a situation where the precondition holds, but postcondition is violated?

What Inputs Cause What Output?

```
String[] parseName(String name) {
  int comma = name.indexOf(",");
  String firstName = name.substring(0, comma);
  String lastName = name.substring(comma + 2);
  return new String[] { lastName, firstName };
}
```

What input produces array [“Doe”, ”Jane”]?

What input produces array [“oe”, ”Jane”]?

What input produces StringIndexOutOfBoundsException?

Types of Reasoning

- **Forward reasoning:** given a precondition, does the postcondition hold?
  - Verify that code works correctly
  - Does the code produce output that matches the postcondition?

- **Backward reasoning:** given a postcondition, what is the proper precondition?
  - Again, verify that code works correctly
  - What input caused an error

Forward Reasoning

- We know what is true before running the code. What is true after running the code?

```
// precondition: x is even && x >= 0
x = x + 3;
y = 2x;
x = 5;
// What is the postcondition here?
// I.e., what is true about the program state at this point?
```

Strongest Postcondition

- Many postconditions hold from this precondition and code!

// precondition: x is even && x >= 0

```
x = x + 3;
y = 2x;
x = 5;
```

// postcondition: x = 5 && y >= 6 && y % 4 = 2

// postcondition: x = 5 && y % 4 = 2

// postcondition: x = 5 && y is even

// postcondition: x > -42 && y is even

x=5 && y>=6 && y%4 = 2 is the strongest postcondition. It implies all other postconditions. More on stronger and weaker conditions later.

Forward Reasoning Example

// precondition: **`x > y`**

```
z = x;
x = y;
y = z;
```

// What is the postcondition ??

Forward Reasoning Example

- // precondition: x > y
  - { x0 > y0 } // x0, y0 means the initial values of x and y
- z = x
  - { z = x0 && x0 > y0 }
- x = y
  - { x = y0 && z = x0 && x0 > y0 } => { x = y0 && z = x0 && z > y0 } => { x = y0 && z = x0 && z > x }
- y = z

  Not the strongest!
  - { y = z && x = y0 && z = x0 && z > x } => { y = z && z > x } => { y > x }
- The interesting post condition is y > x, but there are other conditions which are true {y = z && x = y0 && z = x0 }
  - Are they relevant to what comes next?

Backward Reasoning

- We know what we want to be true after running the code. What must be true beforehand to ensure that?

// precondition: ??

```
x = x + 3;
y = 2x;
x = 5;
```

// postcondition: y > x

Backward Reasoning

- Precondition: $\{ 2(x+3) > 5 \} \Rightarrow \{ 2x > -1 \}$
- x = x + 3;
  - $\{ 2x > 5 \}$
- y = 2x;
  - $\{ y > 5 \}$
- x = 5;
  - Postcondition: $\{ y > x \}$

Forward vs. Backward Reasoning

- Forward reasoning may seem more intuitive, just simulates the code
  - Introduces facts that may be irrelevant to the goal
  - Takes longer to prove task or realize task is hopeless
- Backward reasoning is usually more helpful
  - Given a specific goal, shows what must hold beforehand in order to achieve this goal
  - Given an error, gives input that exposes error

Forward Reasoning: Putting Statements Together

Does the postcondition hold?

Precondition: x >= 0; Postcondition: z > 0

```
z = 0;
```

{ x >=0 && z = 0 }

```
if (x != 0) {
```

{ x >= 0 && x != 0 && z = 0 } =>{ x > 0 && z = 0 }

```
    z = x;
```

{ x > 0 && z = x} => {z > 0}

```
} else {
```

{ x >= 0 && x = 0 && z = 0 } => { x = 0 && z = 0 }

```
    z = z + 1
```

{ x = 0 && z = 1 }

```
}
```

{(z > 0) || (x=0 && z=1)}

either way z > 0;

Therefore, postcondition holds!

Reasoning About Loops

- A loop represents an unknown number of paths
  - Case analysis can be tricky
  - Recursion presents the same problem
- Might not be able to enumerate all paths
  - Testing and reasoning about loops can be tricky

Forward Reasoning With a Loop

Does the postcondition hold?

```
Precondition: x >= 0;

i = x;
```

{ x >=0 && i = x }

```
z = 0;
```

{ x >= 0 && i = x && z = 0 }

```
while (i != 0) {
```

???

```
  z = z+1;
```

???

```
  i = i-1;
```

???

```
}
Postcondition: x = z;
```

The key is to **choose** a loop invariant. Then prove by induction over the iterations of the loop.

Loop Invariant

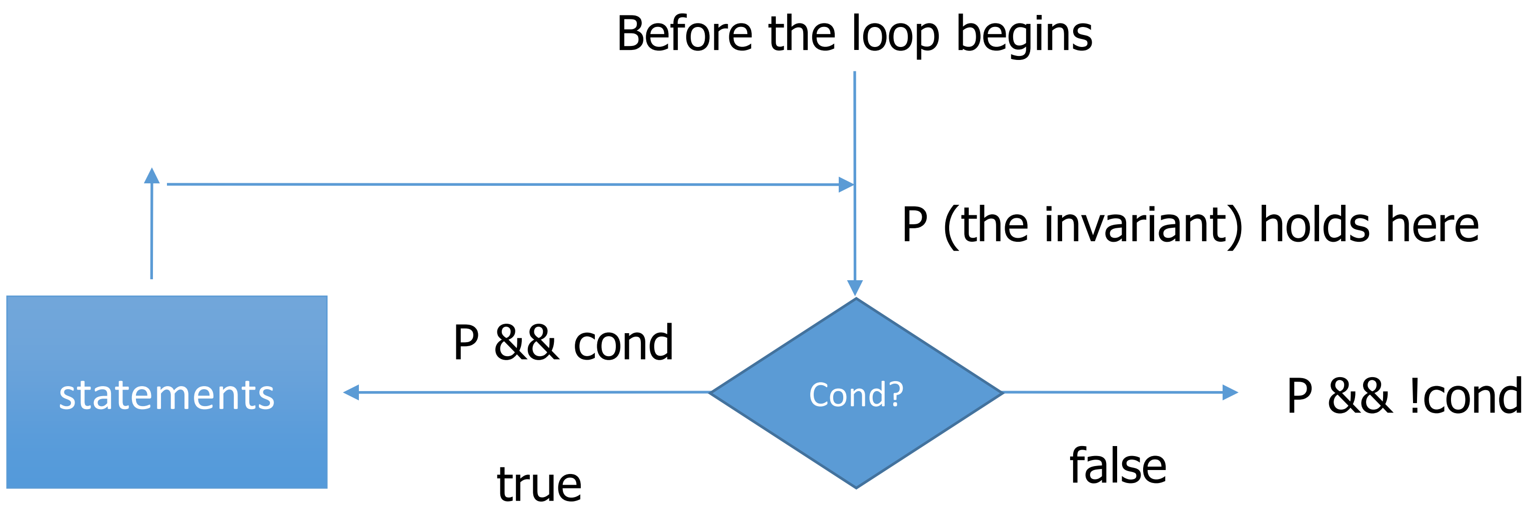
- A loop invariant is a property that is preserved by execution of the loop body
  - That doesn't mean that just **any** property is a useful loop invariant
  - Loop invariants must be effective
    - i.e., involve the loop variables and postcondition in a useful way
- A loop invariant is a condition that is true immediately *before* and immediately *after* each iteration of a loop
  - Doesn't say anything about truth part way through
- We reason about loop invariants using induction

Forward Reasoning With a Loop

- A loop invariant must be true before, after the loop exits, and after each iteration of the loop
  - Is it true before loop starts?
    - Base case
  - Assume the invariant is true for iteration n-1
  - Prove it is true for iteration n
  - Is the invariant true after the loop completes?
- A loop invariant must be useful/relevant

```
while ( cond ) {     <=== define loop invariant P
   statements
}
```

Forward Reasoning With a Loop



Forward Reasoning With a Loop

```
Precondition: x >= 0;
i = x;
z = 0;
while (i != 0) {
  z = z + 1;
  i = i - 1;
}
Postcondition: x = z;
```

Invariant: $i + z = x$

Before:

$x = 0 + i$

Induction - assume invariant holds for iteration n-1: $\mathrm{i}_{n-1} + z_{n-1} = x$

$\mathrm{z}_n = z_{n-1} + 1$

$i_n = i_{n-1} - 1$

invariant: $i_n + z_n = i_{n-1} - 1 + z_{n-1} + 1 = \mathrm{i}_{n-1} + z_{n-1} = x$

After:

$i = 0 \,\&\&\, i + z = x \rightarrow x = z$

Reasoning About Loops

- Where did i + z = x come from?
- We guessed...
  - But not just some random guess
- A good loop invariant should involve the loop variable and the post condition.
- ! Condition && invariant must imply the postcondition at exit.
  - { !(i != 0) && x = i + z) } => { x = z } at exit

Hoare Logic

- Formal framework for reasoning about code
  - **mechanize** the process of reasoning about code

- Sir Anthony Hoare (Sir Tony Hoare or Sir C.A.R. Hoare)
  - Hoare logic
  - Quicksort algorithm
  - Other contributions to programming languages
  - Turing Award in 1980

Hoare Triples

- A Hoare Triple: { P } **`code`** { Q }
  - P and Q are logical statements about program values, and **`code`** is program code (in our case, Java code)

- "{ P } **`code`** { Q }" means "If program **code** is started in a state satisfying condition P, if it terminates, it will terminate in a state satisfying condition Q."
- In other words "if P is true and we satisfactorily execute **`code`**, then Q is true afterwards"
  - "{ P } **`code`** { Q }" is a logical formula, just like "$0 \leq$ index"

Examples of Hoare Triples

{ x>0 } **`x++`** { x>1 } is true

{ x>0 } **`x++`** { x>-1 } is true

{ x≥0 } **`x++`** { x>1 } is false. Why?

{ x > 0 } **`x++`** { x > 0 } is ??

{ x < 0 } **`x = x + 1`** { x < 0 } is ??

{ x = a } **`if (x < 0) x=-x`** { x = |a| } is ??

{ x = y } **`x = x + 3`** { x = y } is ??

Examples of Hoare Triples

- { $x \geq 0$ } **`x++`** { $x>1$ } is a logical formula
- The meaning of "{ $x \geq 0$ } **`x++`** { $x>1$ }"
  - "If x>=0 and we execute x++, then x>1 will hold".
  - Counterexample
    - this statement is false because when x=0, x++ will be 1
    - x > 1 won't hold
- One way to show that a Hoare triple is false is to find a counterexample

Hoare Triples

- Why do we care?
  - We have some conclusion that we want to guarantee
    - Do preconditions guarantee the postcondition?
  - We have some preconditions
    - Do they guarantee the postcondition?
  - Given the code and the postcondition, what are the preconditions that guarantee the postcondition holds?
    - Typically requires backward reasoning
    - Can we reason about the code to find some precondition that will guarantee our postcondition?
    - Can we find a precondition that makes the Hoare triple true?

Hoare Triples and the Weakest Precondition

- The following Hoare triples are true (valid)
  - Assume x, y are ints
  - { $y > -1$ } $x = y + 1$ { $x > 0$ }
  - { $y > 0$ } $x = y + 1$ { $x > 0$ }
  - { $y > 10$ } $x = y + 1$ { $x > 0$ }
    - $y > 10$ implies $y > -1$
- The first is the most useful.
  - It is the weakest precondition
- A Hoare triple is still true if we replace the precondition with a stronger condition
  - You can't replace the precondition with a condition that is weaker than the weakest precondition and still have the triple be true.

Rules for Backward Reasoning: Assignment

// precondition: ??

`x = expression`

// postcondition: Q

Rule: precondition is: Q with all occurrences of **`x`** in Q replaced by **`expression`**

// precondition: { y + 1 > 0 } => { y > -1 }

`x = y + 1;`

// postcondition: { x > 0 }

Read from bottom

Weakest Precondition

Rule derives the weakest precondition

```
// precondition: { y + 1 > 0 } (equivalently {y > -1}) x = y + 1
// postcondition: { x > 0 }
```

{ (y + 1) > 0 } is the weakest precondition for code **`x = y + 1`** and postcondition { x > 0 }

Notation: wp stands for weakest precondition

wp("**`x=expression;`**", {Q}) = {Q'}

Q' is Q with all occurrences of **`x`** replaced by **`expression`**

Why do we want the weakest precondition?

There are many preconditions that can make a Hoare triple with code `x = y + 1` and postcondition x > 0 true.

E.g., { y > -1 } x = y + 1 { x > 0 }
but also { y > 0 } x = y + 1 { x > 0 }.
This is because y > 0 implies y > -1

The weakest precondition is the *minimal* input conditions that guarantee the postcondition

The weakest precondition places the least restriction on the client

Backward Reasoning

"wp" is a function that takes code **c** and a postcondition **Q** and returns a precondition.

Read **wp(c, Q)** as "the weakest precondition of code c w.r.t. Q"

wp(c, Q) is a precondition for c that ensures Q as a postcondition.
Satisfies the Hoare triple {wp(c, Q)} c {Q}.

If wp(c, Q) is the weakest precondition
for any P such that {P} c {Q} is true then P => wp(c, Q)
i.e., P is stronger than wp(c, Q)

If we want to prove {P} c {Q}, we may prove P => wp(c, Q) instead.

Weaker and Stronger Conditions

- P is stronger than Q if P implies Q
  - P => Q
- If P is stronger than Q then P is more likely to be false than Q
- Example from politics:
  - “I will keep unemployment below 3%” is stronger than “I will keep unemployment below 15%”
- The strongest possible statement is always *False*
  - I will keep unemployment below 0%
  - More properly, null set is strongest possible statement – subset of everything
- The weakest possible statement is always *True*
  - I will keep unemployment below 101%
  - Universe set is weakest

Weaker and Stronger Conditions

- “P is stronger than Q” means “P implies Q”
- “P is stronger than Q” means
    - “P’s set of true values is a subset of Q’s”
        - $x > 0$ is stronger than $x > -1$
        - "P is more restrictive"

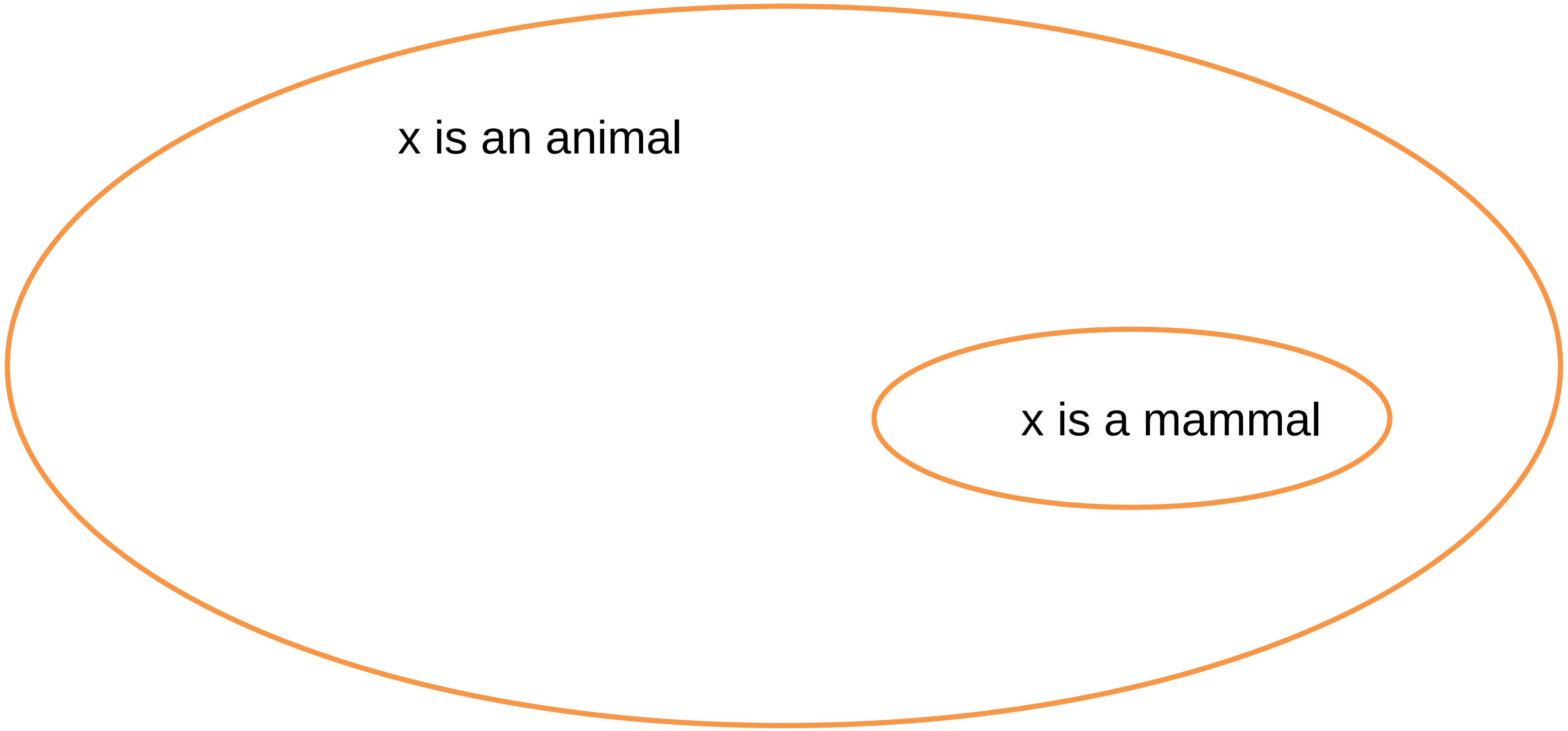
Which one is stronger?

$x > 0$ && $y = 0$ or $x > 0$ && $y \geq 0$

$0 \leq x \leq 10$ or $0 \leq x \leq 1$

$x = 5$ && $y \% 4 = 2$ or $x = 5$ && y is even (% is mod operator)

Weaker and Stronger Conditions



Weakest Precondition

- Starting with a postcondition, what is the weakest precondition that makes the postcondition true?
  - What must be true beforehand to make the postcondition true after
  - Weakest preconditions yield the strongest specifications for computation

- If A => B but not (B => A), then B is "weaker" than A, and A is "stronger" than B
- The weakest possible precondition is *true*
  - Since A => true is always true
  - Anything is allowed
- The strongest possible precondition is *false*
  - Nothing is allowed

Weakest Precondition

- For each Q there are many P such that {P} code {Q}
- For each P there are many Q such that {P} code {Q}
- For each Q there is exactly one assertion wp(code, Q)
  - S.t. {wp(code, Q)} code {Q} is true
- wp(code Q) is unique
  - Logical simplifications are the same Q
    - { x > -1 } = { x >= 0 } for ints

Weaker and Stronger Conditions

Let the following be true:

P => Q   Q => R   S => T   T => U

{ Q } **code** { T }

"T => U" means "T implies U" or "T is stronger than U"

Then which of the following are true?

{ P } **code** { T }

{ R } **code** { T }

{ Q } **code** { S }

{ Q } **code** { U }

Weaker and Stronger Conditions

Let the following be true:

P => Q   Q => R   S => T   T => U

{ Q } **code** { T }

“T => U” means “T implies U” or “T is stronger than U”

Then which of the following are true?

{ P } **code** { T }	*true*
{ R } **code** { T }	*not necessarily*
{ Q } **code** { S }	*not necessarily*
{ Q } **code** { U }	*true*

Weaker and Stronger Conditions

- We can substitute a stronger precondition and the triple can still be true.
  - We usually want the weakest precondition.
  - Requires less of the client code
- We can substitute a weaker postcondition and the triple can still be true.
  - We usually want the strongest postcondition.
  - Guarantees more to the client code

Weaker and Stronger Conditions

- In backward reasoning, we determine the precondition, given **`code`** and a postcondition Q
  - We want the weakest precondition, wp(**`code`**,Q)
  - Find the minimal restriction the code places on the caller
  - We want the code to work in as many places as possible

- In forward reasoning, we determine the postcondition, given **`code`** and a precondition P
  - Normally we want the strongest postcondition
  - We want to guarantee as much as we can

Weakest Precondition

- Consider x = x+1 and postcondition x > 0
- x > 0 is a valid precondition
  - {x > 0} x = x + 1 {x > 0} is true
- x > -1 is also a valid precondition
  - {x > -1} x = x + 1 {x > 0} is true
- x > -1 is weaker than x > 0
  - {x > 0} => {x > -1}
- x > -1 is the weakest precondition
  - wp(x = x + 1, x > 0) = {x > -1}

Another Example

- Consider
  - a = a + 1;
  - b = b - 1;
  - Postcondition { a*b = 0 }
- A very strong precondition
  - { (a = -1) && (b = 1) }
- A weaker precondition
  - { a = -1 }
- Another weak precondition
  - { b = 1 }
- The weakest precondition
  - { (a = -1) || (b = 1) }
- wp("a = a + 1; b = b - 1;", a * b = 0) = { (a = -1) || (b = 1) }

Backward Reasoning: Rule for Assignment

{ wp("x=<expression>", Q) }
x = <expression>;
{ Q }

Rule: the weakest precondition wp("x=expression", Q)
is Q with all occurrences of x in Q replaced
by <expression>

Assignment Operations

- wp(x = y + 5, (x > 5)) = { y + 5 > 5 }   (substitute y + 5 for x)
  - = { y > 0 } (simplify)
- wp(x = x + 1, (x > 3)) = { x + 1 > 3 }   (substitute x + 1 for x)
  - = { x > 2 } (simplify)

Rules for Backward Reasoning: Sequence

// precondition: ??
`S1;` // statement
`S2;` // another statement
// postcondition: Q

Work backwards:
precondition is wp(“`S1;S2;`”, Q) = wp(“`S1;`”,wp(“`S2;`”,Q))

Example:

```
// precondition: ??
x = 0;
y = x + 1;
// postcondition: y > 0
```

```
// precondition: ??
x = 0;
// postcondition for x = 0; same as
// precondition for y = x + 1;
y = x + 1;
// postcondition y>0
```

Example

$$precondition: true$$

$$wp(x = 0; x > -1) = \{0 > -1\} = \{true\}$$

$$x = 0$$

$$wp(y = x + 1; y > 0) = \{x + 1 > 0\} = \{x > -1\}$$

$$y = x + 1$$

$$postcondition: y > 0$$

Work from the bottom up

Example

- Precondition: {b = 1 || a = -1}
  - wp(a = a + 1, b = 1 || a = 0) = { b = 1 || a + 1 = 0 } = { b = 1 || a = -1 }
- a = a + 1;
  - wp(b = b - 1, a * b = 0) = { a * (b - 1) = 0 } = { b = 1 || a = 0 }
- b = b - 1;
- Postcondition a * b = 0

Exercise

// precondition: ??

x = x + 1;

y = x + y;

// postcondition y > 1

Exercise

$$precondition: x + y > 0$$

$$wp(x = x+1; x + y > 1) = \{x+1+y > 1\} = \{x + y > 0\}$$

$$x = x+1$$

$$wp(y = x + y; y > 1) = \{x + y > 1\} \; // substitute\ for\ y$$

$$y = x + y$$

$$postcondition: y > 1$$

Check by forward reasoning

$$precondition: x_0 + y_0 > 0$$

$$x = x_0 + 1$$

$$\{x = x_0 + 1 \,\&\&\, x_0 + y_0 > 0\} = \{x - 1 + y_0 > 0\} = \{x + y_0 > 1\}$$

$$y = x + y_0$$

$$\{y = x + y_0 \,\&\&\, x + y_0 > 1\} = \{y > 1\}$$

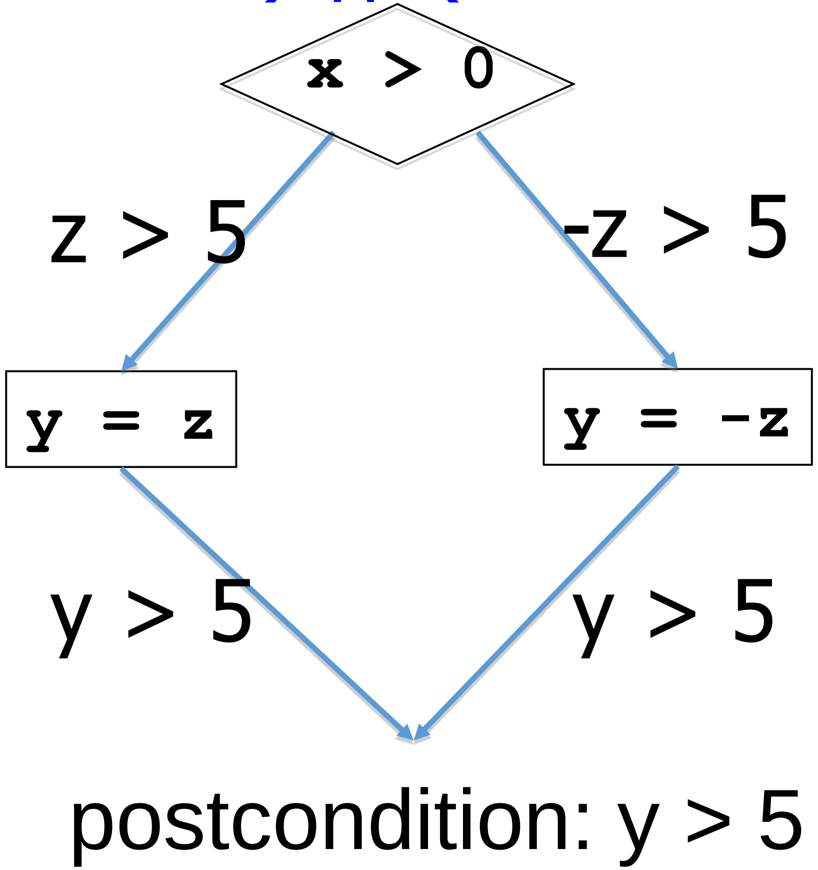
$$postcondition: y > 1$$

If-then-else Statement Example

// precondition: ?? $(z > 5 \;\&\&\; x > 0) \;||\; (z < -5 \;\&\&\; x \leq 0)$

```
if (x > 0) {
  y = z;
}
else {
  y = -z;
}
```

// postcondition: $y > 5$



Rules for Backward Reasoning: If-then-else

// precondition: ??

`if (b) S1 else S2`

// postcondition: Q

Case analysis, just as we did in the example:

wp("**`if (b) S1 else S2`**", Q)

= { (b && wp("**`S1`**",Q)) || (not(b) && wp("**`S2`**",Q)) }

If-else Statement Example

$$wp(if\ (x > 0)\ y = z; else\ y = -z;, y > 5)$$

$$= \{(x > 0 \ \&\& z > 5) \,||\, (x \le 0 \ \&\& z < -5)\}$$

$$if\ (x > 0)\{$$

$$wp(y = z, y > 5) = \{z > 5\}$$

$$y = z;$$

$$\}else\{$$

$$wp(y = -z, y > 5) = \{-z > 5\} = \{z < -5\}$$

$$y = -z;$$

$$\}$$

$$postcondition : y > 5$$

Exercise

Precondition: ??

```
z = 0;
if (x != 0) {
   z = x;
} else {
   z = z + 1;
  }
```

Postcondition: z > 0;

Exercise

$$wp(z = 0, (x > 0) \,||\, (x == 0 \,\&\&\, z > -1))$$
$$= \{(x > 0) \,||\, (x == 0 \,\&\&\, 0 > -1)\}$$
$$= \{(x > 0) \,||\, (x == 0 \,\&\&\, true)\}$$
$$= \{(x > 0) \,||\, (x == 0)\}$$
$$= \{(x >= 0)\}$$

$z = 0;$

$$wp(if\ (x != 0)\ z = x; else\ z = z + 1;, z > 0)$$
$$= \{(x != 0 \,\&\&\, x > 0) \,||\, (x == 0 \,\&\&\, z > -1)\}$$
$$= \{(x > 0) \,||\, (x == 0 \,\&\&\, z > -1)\}$$

$if\ (x != 0)\{$

$$wp(z = x, z > 0) = \{x > 0\}$$

$z = x;$

$\}$

$else\{$

$$wp(z = z + 1, z > 0) = \{z + 1 > 0\} = \{z > -1\}$$

$z = z + 1;$

$\}$

$postcondition: \{z > 0\}$

Exercise

```
// precondition: ??
if (x < 5) {
  x = x * x;
}
else {
  x = x + 1;
}
// postcondition: x ≥ 9
```

Assume x is an int

Exercise

$$wp(if\,(...)\{...\},\, x \ge 9)$$
$$= \{(x < 5 \,\&\&\, |x| >= 3) \,||\, (x \ge 5 \,\&\&\, x \ge 8)\}$$
$$= \{x \le -3 \,||\, x == 3 \,||\, x = 4 \,||\, x \ge 8\}$$

$if\,(x < 5)\{$

$$wp(x = x * x, x \ge 9) = \{x * x \ge 9\} = \{|x| >= 3\} = \{x \ge 3 \,||\, x \le -3\}$$

$x = x * x;$

$\}else\{$

$$wp(x = x + 1, x \ge 9) = \{x + 1 \ge 9\} = \{x \ge 8\}$$

$x = x + 1;$

$\}$

$postcondition : \{x \ge 9\}$

If-then-else Statement Review

Forward reasoning

{ P }
`if b`
 { P && **`b`** }
 `S1`
 { Q1 }
`else`
 { P && not(b) }
 `S2`
 { Q2 }
{ Q1 || Q2 }

Backward reasoning

{ (**`b`** && wp("**`S1`**",Q)) || (not(**`b`**) && wp("**`S2`**",Q)) }
`if b`
 { wp("**`S1`**",Q) }
 `S1`
 { Q }
`else`
 { wp("**`S2`**",Q) }
 `S2`
 { Q }
{ Q }

If-then Statement

```
// precondition: ??
if (x > y) {
   z = x;
   x = y;
   y = z;
}
// postcondition: x < y
```

If Statement

$$wp(if\,(\ldots), x < y)$$

$$= \{(x > y \,\&\&\, y < x) \,\|\, (x <= y \,\&\&\, x < y)\}$$

$$= \{x > y \,\|\, x < y\} = \{x \neq y\}$$

$$if\,(x > y)\{$$

$$wp(z = x, y < z) = \{y < x\}$$

$$z = x;$$

$$wp(x = y, x < z) = \{y < z\}$$

$$x = y;$$

$$wp(y = z, x < y) = \{x < z\}$$

$$y = z;$$

$$\}$$

$$postcondition : \{x < y\}$$

Backward Reasoning: Rule for Assignment

{ wp("x=<expression>", Q) }
x = <expression>;
{ Q }

Rule: the weakest precondition wp("x=expression", Q)
is Q with all occurrences of x in Q replaced
by <expression>

Backward Reasoning: Rule for Sequence

```
// find weakest precondition for sequence S1;S2 and Q

{ wp( S1, wp( S2, Q ) ) }
S1; // statement Postcondition for S1 is wp(S2, Q)
{ wp( S2, Q ) }
S2; // another statement
{ Q }
```

Backward Reasoning: Rule for If-then-else

```
{ ( b && wp( S1, Q ) )  ||  ( not b && wp( S2, Q ) ) }
if ( b ) {
  S1;  // S1 and S2 could be multiple statements
}
else {
  S2;
}
{ Q }
```

... without the else:

```
{ ( b && wp( S1, Q ) )  ||  ( not b && Q ) }
if ( b ) {
  S1;
}
{ Q }
```