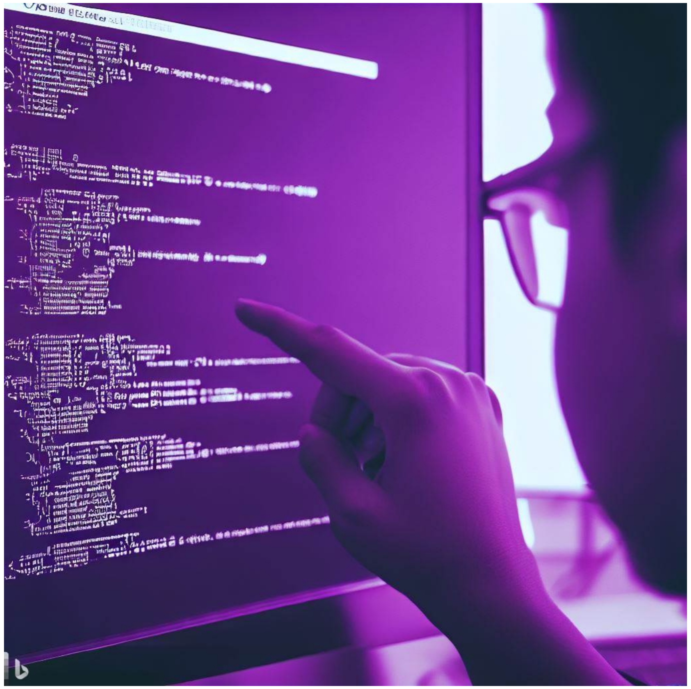


CSE 331

Floyd Logic

Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox

Reasoning So Far

- Code so far made up of three elements
  - straight-line code
  - conditionals
  - recursion

- Know how to reason (think) about these already
  - saw the first two already
  - we reasoned about recursion in math,
    but this can be done in code also

    our code is direct translation of math, so easy to switch between

Recall: Finding Facts at a Return Statement

- Consider this code

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    const L: List = 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 "$L = \text{cons}(\ldots)$"

- Prove that postcondition holds: "$\text{sum}(L) \geq 0$"

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else {
    return square(n - 1n) + n + n - 1n;
  }
};
```

- How do we check correctness?
- Option 1: translate this to math

func square(0) := 0
square(n+1) := square(n) + 2(n+1) – 1   for any n : $\mathbb{N}$

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: bigint): bigint => { … };
```

func square(0)	:= 0		
square(n+1)	:= square(n) + 2(n+1) – 1		for any n : ℕ

- **Prove that** $\text{square}(n) = n^2$ **for any** $n : \mathbb{N}$

- **Structural induction requires proving two implications**
  - **base case: prove** $\text{square}(0) = 0^2$
  - **inductive step: prove** $\text{square}(n+1) = (n+1)^2$

    can use the fact that $\text{square}(n) = n^2$

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else {
    return square(n - 1n) + n + n - 1n;
  }
};
```

- **Option 2: reason directly about the code**
- **Known fact at top return:** $n = 0$

$$\begin{aligned} \text{square}(0) &= 0 && \text{(code)} \\ &= 0^2 \end{aligned}$$

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else {
    return square(n - 1n) + n + n - 1n;
  }
};
```

why is it okay to assume square is correct when we're checking it?

Inductive Hypothesis

- **Known fact at bottom return:** $n > 0$

$$\begin{aligned} \text{square}(n) &= \text{square}(n - 1) + 2n - 1 && \text{(code)} \\ &= (n - 1)^2 + 2n - 1 && \text{spec of square} \\ &= n^2 - 2n + 1 + 2n + 1 \\ &= n^2 \end{aligned}$$

Reasoning So Far

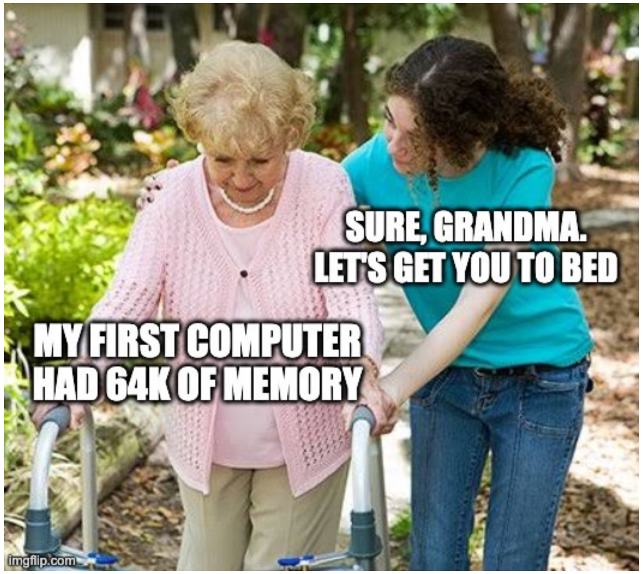
- Code so far made up of three elements
  - straight-line code
  - conditionals
  - structural recursion
- Any[1] program can be written with just these
  - we could stop the course right here!
- For performance reasons, we often use more
  - this week: mutation of local variables
  - later: mutation of arrays and heap data

[1] only exception is code with infinite loops

Brief History of Software

- **Computers used to be very slow**

  Kevin's first computer had 64k of memory



- **Software <u>had</u> to be extremely efficient**
  - loops, mutation all over the place
  - very hard to write correctly, so it did *very little*

Brief History of Software

- Computers used to be very slow
  - software had to be extremely efficient

- Today, programmers are the scarcest resource
  - we have enormous computing resources

- Anti-pattern: favoring efficiency over correctness
  - programmers overestimate importance of efficiency

    "programmers are notoriously bad" at guessing what is slow — B. Liskov

    "premature optimization is the root of all evil" — D. Knuth
  - programmers are overconfident about correctness

    routinely takes 3x as long as expected to get it right

“Programmers overestimate the importance of **efficiency** and underestimate the difficulty of **correctness**.”

— Class slogan #3

Correctness Levels

Level	Description	Testing	Tools	Reasoning
0	small # of inputs	exhaustive		
1	straight from spec	heuristics	type checking	code reviews
2	no mutation	“	libraries	calculation induction
3	local variable mutation	“	“	Floyd logic
4	array mutation	“	“	for-any facts
5	heap state mutation	“	“	rep invariants

Recall: Finding Facts at a Return Statement

- Consider this code

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

$a \geq 0$

$a \geq 0$? No!

- Facts no longer hold throughout the function

- When we state a fact, we have to say <u>where</u> it holds

Recall: Finding Facts at a Return Statement

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

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

Recall: Finding Facts at a Return Statement

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

- There are <u>mechanical</u> 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

Recall: Finding Facts at a Return Statement

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

- Professionals are *insanely* good at forward reasoning
  - "programmers are the Olympic athletes of forward reasoning" - James
  - you'll have an edge by learning backward reasoning too

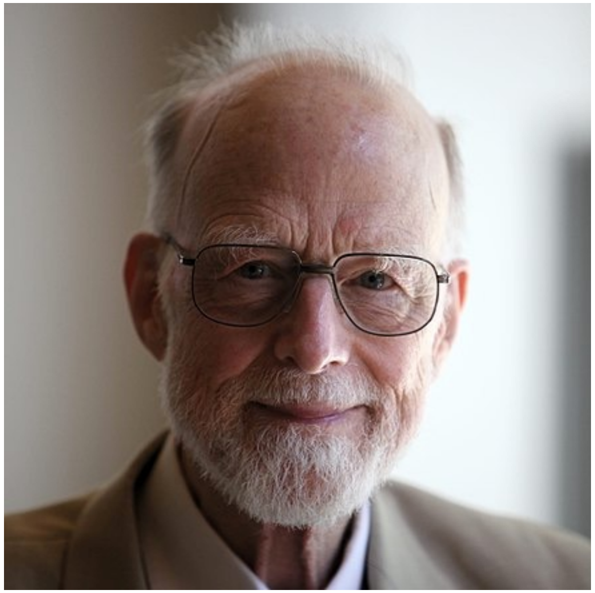
Floyd Logic

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 >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  n = n + 3n;
  return n * n;
};
```

- **Check that value returned,** $m = n^2$**, satisfies** $m \geq 10$

Correctness Example

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n ≥ 1 }}
  n = n + 3n;
  {{ n² ≥ 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:**

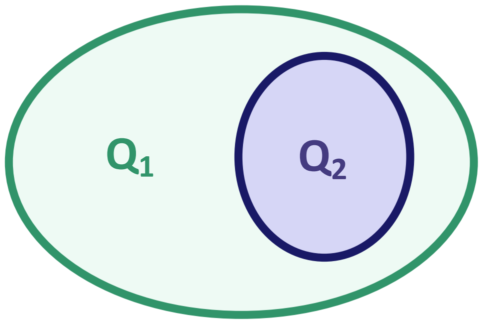
$$\{\{\ a \geq 0,\ \ b \geq 0,\ \ L = \text{cons}(a, \text{cons}(b, \text{nil}))\ \}\}$$
$$\{\{\ \text{sum}(L) \geq 0\ \}\}$$

- **Check that P implies Q by calculation**

$\text{sum}(L)$	$= \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil})))$	since $L = \ldots$
	$= a + \text{sum}(\text{cons}(b, \text{nil}))$	def of sum
	$= a + b + \text{sum}(\text{nil})$	def of sum
	$= a + b$	def of sum
	$\geq 0 + b$	since $a \geq 0$
	$\geq 0 + 0$	since $b \geq 0$
	$= 0$	

Stronger Assertions vs Specifications

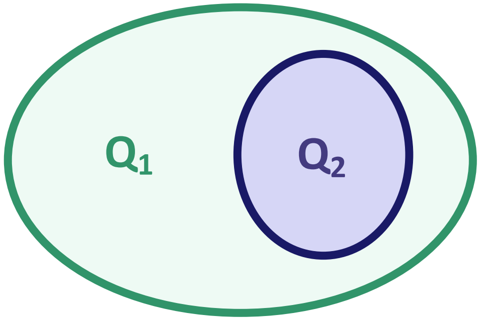
- **Assertion** is **stronger** iff it holds in a subset of states



- **Stronger** assertion <u>implies</u> the **weaker** one
  - stronger is a synonym for "implies"
  - weaker is a synonym for "is implied by"

Stronger Assertions vs Specifications

- 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!)

Hoare Triples with Multiple Lines of Code

- Code with multiple lines:

```
{{ P }}
   S
   T
{{ Q }}
```

```
{{ P }}
   S
{{ R }}
   T
{{ Q }}
```

- 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…

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

```
{{ P }}          {{ P }}  ┐
   S                S     │ 1
{{ Q }}          {{ R }}  ┘ ┐
                 {{ Q }}    ┘ 2
```

  - 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

{{ P }}
S
{{ Q }}

{{ P }}
{{ R }}
S
{{ Q }}

1
2

  – second triple is always valid
  – only need to check first triple

    just requires proving an implication (since no code is present)

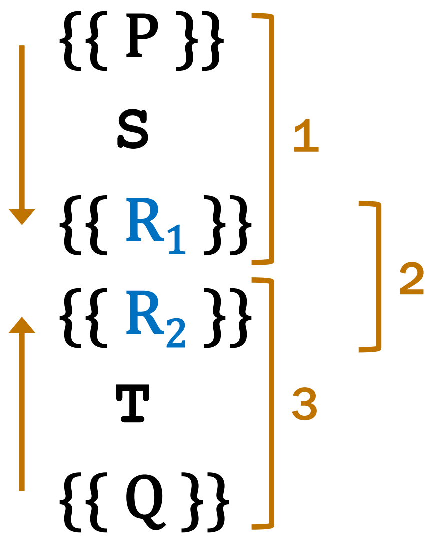
- If first triple is invalid, the code is **incorrect**
  – true because R is the weakest assertion possible here

Mechanical Reasoning Tools

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples
- Reduce correctness to proving implications
  - 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 = 17n;
{{ _______________ }}
  y = 42n;
{{ _______________ }}
  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 = 17n;
```

{{ w > 0 and x = 17 }}

```
y = 42n;
```

{{ ________________ }}

```
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 = 17n;
```

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

```
y = 42n;
```

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

```
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 = 17n;
```

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

```
y = 42n;
```

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

```
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 = 17n;
```

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

```
y = 42n;
```

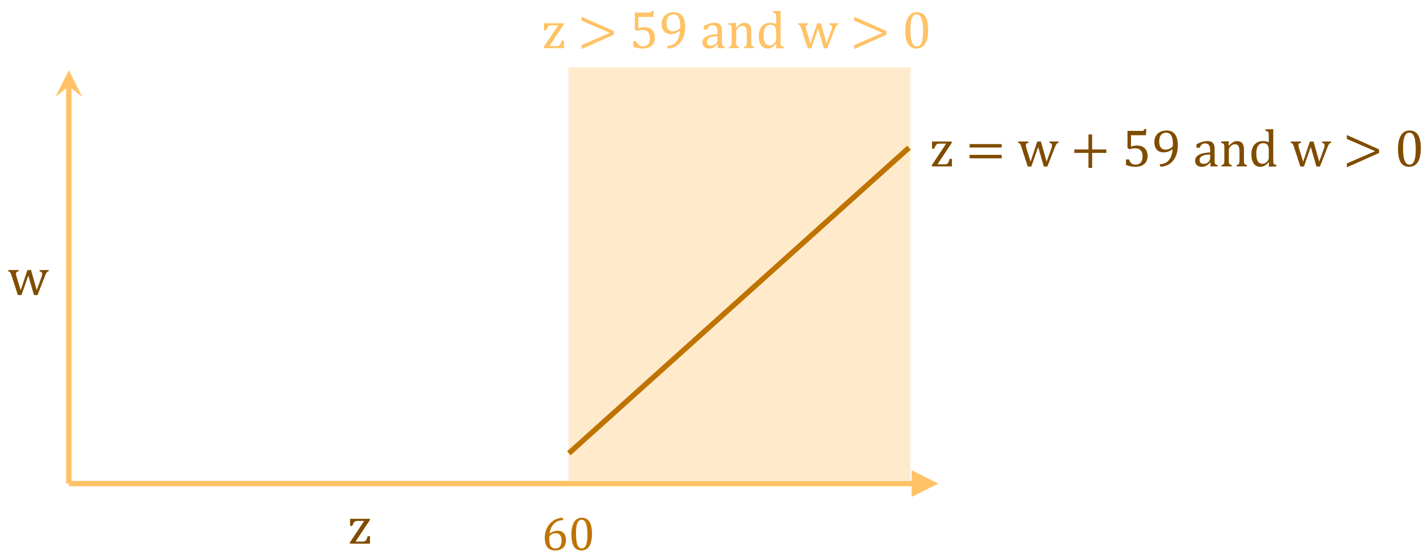
$\{\{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \}\}$

```
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 <u>do not</u> write $z > 59$!**
  - **that is true since $w > 0$, but…**

Example Forward Reasoning through Assignments



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

Example Forward Reasoning through Assignments

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

```
x = 17n;
```

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

```
y = 42n;
```

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

```
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…**
  - **that 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
const f = (w: bigint): bigint => {
  const x = 17n;
  const y = 42n;
  const 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
const f = (w: bigint): bigint => {
  {{ w > 0 }}
  const x = 17n;
  const y = 42n;
  const 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
const f = (w: bigint): bigint => {
  {{ w > 0 }}
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  {{ w > 0 and x = 17 and y = 42 and z = w + x + y }}
  {{ z > 59 }}
  return z;
};
```

- **Check implication:**

$$
\begin{aligned}
z &= w + x + y \\
&= w + 17 + y && \text{since } x = 17 \\
&= w + 59 && \text{since } y = 42 \\
&> 59 && \text{since } w > 0
\end{aligned}
$$

Code Example of Forward Reasoning

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

- We’ve been doing forward reasoning all quarter!
  - forward reasoning is (only) “and” with *no mutation*
- Line-by-line facts are for “`let`” (not “`const`”)

Forward Reasoning through Assignments

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

```
w = x + y;
```
$\{\{\ w = x + y\ \}\}$
```
x = 4n;
```
$\{\{\ w = x + y \text{ and } x = 4\ \}\}$
```
y = 3n;
```
$\{\{\ w = x + y \text{ and } x = 4 \text{ and } y = 3\ \}\}$

- **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 recognizing old versus new values**

Forward Reasoning through Assignments

- **Fix this by denoting original variable values**
  - **can use "x" and "y" to refer to current values**
  - **can use "x_0" and "y_0" to refer to original values**

    rewrite existing facts in terms of the original values

{{ $w = x + y$ }}

```
x = 4n;
```

← no relation to original value of x

{{ $w = x_0 + y$ and $x = 4$ }}

```
y = 3n;
```

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

WARNING:
Cannot use original values notation when values maintain a relationship to their previous value (i.e. incrementing, etc.)

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

Forward Reasoning through Assignments

- **For assignments, general forward reasoning rule is**

$\{\{ P \}\}$

```
x = y;
```

$\{\{ P[x \mapsto x_0] \text{ and } x = y \}\}$

  - **replace all "x"s in P with "x_0"s**

- **This process should be simplified in many cases**
  - **<u>do not</u> use x_0 if we can write it *in terms of* new value**

    we refer to this as the "OLD/NEW variable value technique"

  - **assertions will be easier to read without original 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

$\{\{ P \}\}$

```
x = y;
```

← no relation to orig x val

$\{\{ P[x \mapsto x_0] \text{ and } x = y \}\}$

x_0 is any **new** variable name

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

$\{\{ P \}\}$

```
x = … x …;
```

← in terms of orig x val

$\{\{ P[x \mapsto f(x)] \}\}$

no need for, e.g., "$\text{and } x = x_0 + 1$"

  - 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)

} OLD/NEW values technique

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n ≥ 1 }}          ← earlier or "old" n value
  n = n + 3n;          ← current or "new" n value
  {{ ________ }}
  {{ n² ≥ 10 }}
  return n * n;
};
```

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n_OLD ≥ 1 }}
  n = n + 3n; // SCRATCH WORK: n_NEW = n_OLD + 3 → n_OLD = n_NEW - 3
  {{ ________ }}
  {{ n^2 ≥ 10 }}
  return n * n;
};
```

want $n_{OLD} \geq 1$ terms of "new" n value

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n_OLD ≥ 1 }}
  n = n + 3n; // SCRATCH WORK: n_NEW = n_OLD + 3 → n_OLD = n_NEW - 3
  {{ n_NEW - 3 ≥ 1 }}    ← n_NEW is current n value
  {{ n^2 ≥ 10 }}
  return n * n;
};
```

Assertions: $\{\{ n_{OLD} \geq 1 \}\}$, $\{\{ n_{NEW} - 3 \geq 1 \}\}$, $\{\{ n^2 \geq 10 \}\}$

SCRATCH WORK: $n_{NEW} = n_{OLD} + 3 \rightarrow n_{OLD} = n_{NEW} - 3$

n_{NEW} is current n value

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n ≥ 1 }}
  n = n + 3n; // SCRATCH WORK: n_NEW = n_OLD + 3 → n_OLD = n_NEW - 3
  {{ n - 3 ≥ 1 }}
  {{ n² ≥ 10 }}
  return n * n;
};
```

This is the preferred approach.
Must avoid subscripts when possible.

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n ≥ 1 }}
  n = n + 3n;
  {{ n – 3 ≥ 1 }}
  {{ n² ≥ 10 }}
  return n * n;
};
```

check this implication

$n^2 \geq 4^2$ since $n - 3 \geq 1$ (i.e., $n \geq 4$)

$= 16$

> 10

Example Backward Reasoning with Assignments

$\{\{$ __________________ $\}\}$

```
x = 17n;
```

$\{\{$ __________________ $\}\}$

```
y = 42n;
```

$\{\{$ __________________ $\}\}$

```
z = w + x + y;
```

$\{\{ z < 0 \}\}$

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

Example Backward Reasoning with Assignments

{{ ________________ }}
```
x = 17n;
```
{{ ________________ }}
```
y = 42n;
```
{{ $w + x + y < 0$ }}
```
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

$\{\{\ \underline{\qquad\qquad\qquad\qquad\qquad}\ \}\}$

```
x = 17n;
```

$\{\{\ w + x + 42 < 0\ \}\}$

```
y = 42n;
```

$\{\{\ w + x + y < 0\ \}\}$

```
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 = 17n;
```

$\{\{\ w + x + 42 < 0\ \}\}$

```
y = 42n;
```

$\{\{\ w + x + y < 0\ \}\}$

```
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 <u>substitute</u> 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 EVER!!! ☺

Correctness Example by Backward Reasoning

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

check this implication

$(n+3)^2 \geq (1+3)^2$ since $n \geq 1$

$= 16$

> 10

Conditionals

Conditionals in Functional Programming

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

- Conditionals introduce extra facts in forward reasoning
  - simple "and" case since nothing is mutated

Conditionals in Floyd Logic

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

- Code like this was impossible without mutation
  - cannot write to a "`const`" after its declaration

- How do we handle it now?

Conditionals in Floyd Logic

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

- Check correctness path through “then” branch

Conditionals in Floyd Logic

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

Conditionals in Floyd Logic

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

Conditionals in Floyd Logic

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

- 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
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  {{ 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
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  {{ ________________________________________ }}
  {{ m > n }}
  return m;
}
```

- What is true after the either branches?

Conditionals in Floyd Logic

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

- **What is true after the either branches?**
  - the “or” means we have to reason by cases anyway!

Conditionals in Floyd Logic

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

Function Calls

Reasoning about Function Calls

```
// @requires P₂               -- preconditions a, b
// @returns x such that R -- conditions on a, b, x
const f = (a: bigint, b: bigint): bigint => {..}
```

- **Forward reasoning rule is**

{{ P }}

```
x = f(a, b);
```

{{ $P[x \mapsto x_0]$ and R }}

Must also check that P implies P_2

- **Backward reasoning rule is**

{{ Q_1 and P_2 }}

```
x = f(a, b);
```

{{ Q_1 and Q_2 }}

Must also check that R implies Q_2

Q_2 is the part of postcondition using "x"

Loops

Correctness of Loops

- Assignment and condition reasoning is mechanical

- Loop reasoning <u>cannot</u> 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 <u>every time</u> 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 assertion only claims to be true the first time at the loop

Loop Invariants

- Loop invariant is true <u>**every time**</u> 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 and cond }} ***S*** {{ I }} — **S preserves** I
3. {{ I and not cond }} {{ Q }} — Q **holds when loop exits**

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n) for any n : ℕ

- This loop claims to calculate it as well

```
{{ }}
let i: bigint = 0n;
let s: bigint = 0n;
{{ Inv: s = sum-to(i) }}
while (i != n) {
  i = i + 1n;
  s = s + i;
}
{{ s = sum-to(n) }}
```

Easy to get this wrong!
- might be initializing "i" wrong ($i = 1$?)
- might be exiting at the wrong time ($i \neq n-1$?)
- might have the assignments in wrong order
- ...

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

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

$$\textbf{func}\ \text{sum-to}(0) := 0$$
$$\text{sum-to}(n+1) := (n+1) + \text{sum-to}(n) \qquad \text{for any } n : \mathbb{N}$$

- This loop claims to calculate it as well

```
{{ }}
let i: number = 0n;
let s: number = 0n;
{{ i = 0 and s = 0 }}
{{ Inv: s = sum-to(i) }}
while (i != n) {
  ...
```

sum-to(i)
= sum-to(0) **since** i = 0
= 0 **def of** sum-to
= s

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : $\mathbb{N}$

- This loop claims to calculate it as well

{{ **Inv**: s = sum-to(i) }}
```
while (i != n) {
```
{{ s = sum-to(i) and i ≠ n }}
```
  i = i + 1n;
  s = s + i;
```
{{ s = sum-to(i) }}
```
}
```

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- This loop claims to calculate it as well

```
{{ Inv: s = sum-to(i) }}
while (i != n) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1n;
  {{ s = sum-to(i–1) and i–1 ≠ n }}
  s = s + i;
  {{ s = sum-to(i) }}
}
```

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- This loop claims to calculate it as well

```
{{ Inv: s = sum-to(i) }}
while (i != n) {
   {{ s = sum-to(i) and i ≠ n }}
   i = i + 1n;
   {{ s = sum-to(i–1) and i–1 ≠ n }}
   s = s + i;
   {{ s – i = sum-to(i–1) and i–1 ≠ n }}
   {{ s = sum-to(i) }}
}
```

s = i + sum-to(i-1)   **since** s – i = sum-to(i-1)
  = sum-to(i)   **def of** sum-to

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n) for any n : ℕ

- This loop claims to calculate it as well

{{ **Inv**: s = sum-to(i) }}

```
while (i != n) {
  i = i + 1n;
  s = s + i;
}
```

{{ s = sum-to(i) and i = n }}
{{ s = sum-to(n) }}

sum-to(n)
= sum-to(i) **since** i = n
= s **since** s = sum-to(i)

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(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

Loops & Recursion

Loops and Recursion

- To check a loop, we need a loop invariant

- Where does this come from?
  - part of the algorithm idea / design

    see 421 for more discussion
  - Inv and the progress step **formalize** the algorithm idea

    most programmers can easily formalize an English description

    (very tricky loops are the exception to this)

- Today, we'll focus on converting *recursion* into a loop
  - HW Weave will fit these patterns
  - (more loops later)

Example Loop Correctness

- Recursive function to calculate n^2 without multiplying

$$\textbf{func}\ \text{square}(0) := 0$$
$$\text{square}(n+1) := \text{square}(n) + 2n + 1 \qquad \text{for any } n : \mathbb{N}$$

- We already proved that this calculates n^2
  - we can implement it directly with recursion

- Let’s try writing it with a loop instead…

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : $\mathbb{N}$

- **Loop idea for calculating** square(n)**:**
  - **calculate** i = 0, 1, 2, …, n
  - **keep track of** square(i) **in** "s" **as we go along**

i =	0	1	2	…	n
s =	0	1	4	…	n^2

- **Formalize that idea in the loop invariant**

along with the fact that we make **progress** by advancing i to i+1 each step

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

```
let i: bigint = 0n;
let s: bigint = 0n;
{{ Inv: s = square(i) }}
while (i != n) {
  s = s + i + i + 1n;
  i = i + 1n;
}
return s;
```

Loop invariant says how i and s relate
s holds square(i), for whatever i

i starts at 0 and increases to n

Now we can check correctness...

Example Loop Correctness

func square(0) := 0

square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

{{ }}

```
let i: bigint = 0n;
let s: bigint = 0n;
```

{{ i = 0 and s = 0 }}

{{ **Inv**: s = square(i) }}

```
while (i != n) {
  s = s + i + i + 1n;
  i = i + 1n;
}
return s;
```

square(i)
= square(0) **since** i = 0
= 0 **def of** square
= s **since** s = 0

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1   for any n : ℕ

- **Loop implementation**

```
let i: bigint = 0n;
let s: bigint = 0n;
{{ Inv: s = square(i) }}
while (i != n) {
  s = s + i + i + 1n;
  i = i + 1n;
}
{{ s = square(i) and i = n }}
{{ s = square(n) }}
return s;
```

square(n)
= square(i)   **since** i = n
= s   **since** s = square(i)

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

```
{{ Inv: s = square(i) }}
while (i != n) {
  {{ s = square(i) and i ≠ n }}
  s = s + i + i + 1n;
  i = i + 1n;
  {{ s = square(i) }}
}
return s;
```

Example Loop Correctness

func square(0) := 0

square(n+1) := square(n) + 2n + 1   for any n : ℕ

- **Loop implementation**

{{ **Inv**: s = square(i) }}

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

{{ s = square(i) and i ≠ n }}

```
s = s + i + i + 1n;
```

{{ s = square(i+1) }}

```
i = i + 1n;
```

{{ s = square(i) }}

```
}
return s;
```

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

{{ **Inv**: s = square(i) }}
```
while (i != n) {
```
{{ s = square(i) and i ≠ n }}
{{ s + 2i + 1 = square(i+1) }}
```
s = s + i + i + 1n;
```
{{ s = square(i+1) }}
```
i = i + 1n;
```
{{ s = square(i) }}
```
}
return s;
```

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

{{ **Inv**: s = square(i) }}
```
while (i != n) {
```
{{ s = square(i) and i ≠ n }}
{{ s + 2i + 1 = square(i+1) }}
```
  s = s + i + i + 1n;
```
{{ s = square(i+1) }}
```
  i = i + 1n;
```
{{ s = square(i) }}
```
}
return s;
```

s + 2i + 1 = square(i) + 2i + 1 — **since** s = square(i)
= square(i+1) — **def of** square

“Bottom Up” Loops on Natural Numbers

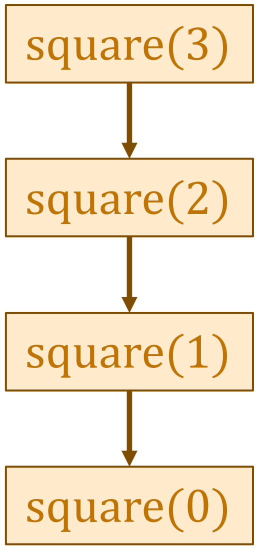
- Previous examples store function value in a variable

  {{ **Inv**: s = sum-to(i) }}

  {{ **Inv**: s = square(i) }}

- Start with $i = 0$ and work up to $i = n$

- Call this a “bottom up” implementation
  - evaluates in the same order as **recursion**
  - from the base case up to the full input



“Bottom Up” Loops on the Natural Numbers

func f(0) := ...

f(n+1) := ... f(n) ... for any n : ℕ

- Can be implemented with a loop like this

```
const f = (n: bigint): bigint => {
  let i: bigint = 0n;
  let s: bigint = "...";    // = f(0)
  {{ Inv: s = f(i) }}
  while (i != n) {
    s = "... f(i) ..."[f(i) ↦ s]    // = f(i+1)
    i = i + 1n;
  }
  return s;
};
```

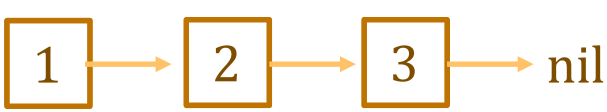
“Bottom Up” Loops on Lists

- **Works nicely on** $\mathbb{N}$
  - **numbers are built up from** 0 **using** succ $(+1)$
  - **e.g., build** $n = 3$ up **from** 0

$$n = \; 3 \xleftarrow{+1} 2 \xleftarrow{+1} 1 \xleftarrow{+1} 0$$

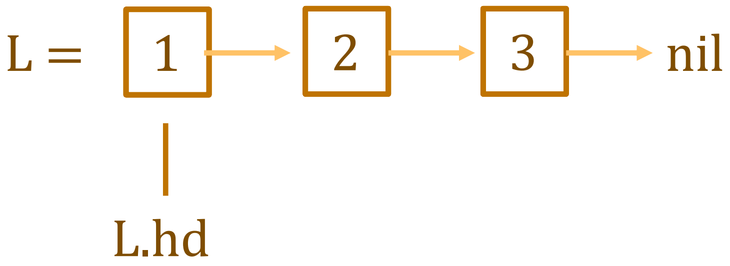
- **What about** List**?**
  - **lists are built up from** nil **using** cons
  - **e.g., build** $L = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$ **from** nil**:**

$L =$



“Bottom Up” Loops on Lists?

- **What about** List**?**
  - **lists are built up from** nil **using** cons
  - **e.g., build** $L = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$ **from** nil**:**



- **First step to build** L **is to build** $\text{cons}(3, \text{nil})$ **from** nil
  - **how do we know what number to put in front of** nil**?**

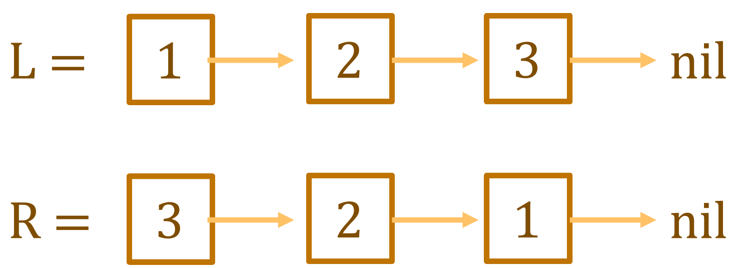
    3 is all the way at the end of the list!
  - **how can we fix this?**
  - **reverse the list!**

Example "Bottom Up" List Loop

func twice(nil) := nil

twice(cons(x, L)) := cons(2x, twice(L))   for any x : $\mathbb{Z}$ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R"

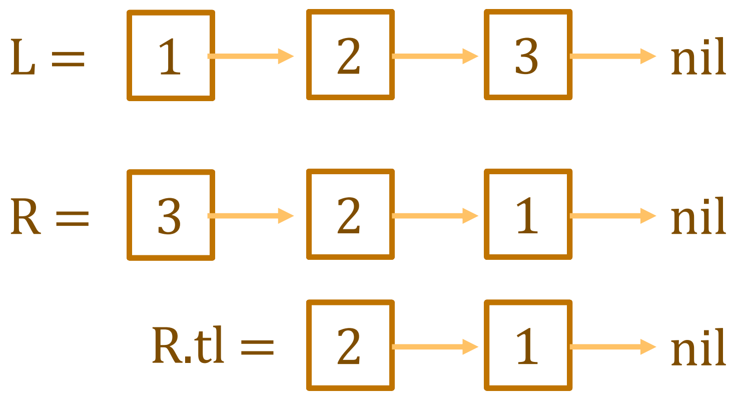


  - **watch what happens as we move** R **forward…**

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R"
  - **moving forward in** R **is moving backward in** L**...**



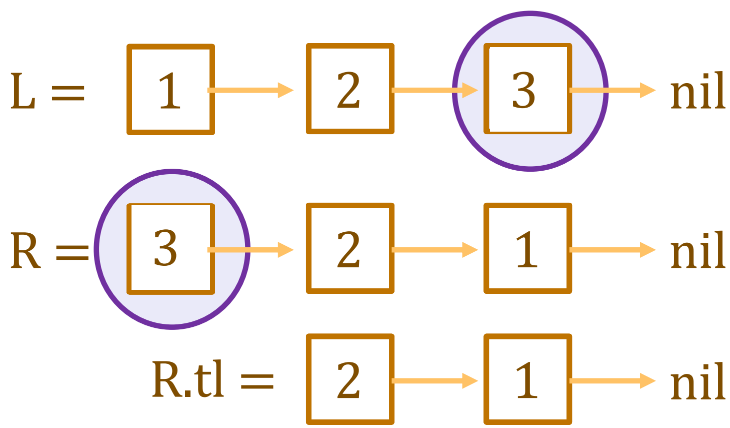
  - **as** R **moves forward,** rev(R) **remains a** **<u>prefix</u>** **of** L

Example "Bottom Up" List Loop

func twice(nil) := nil

twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **Loop idea for calculating** twice(L)**:**
  - **store** rev(L) **in** "R"
  - **moving forward in** R **is moving backward in** L**...**



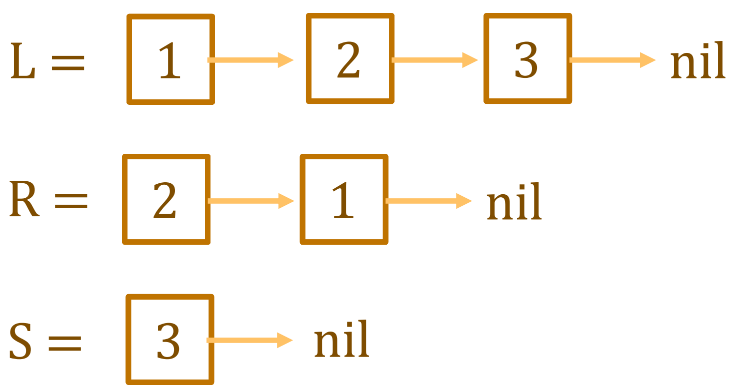
  - **value dropped from** R **was** last(L) = 3

    can use it to build cons(3, nil)

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R" **initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of** "S"

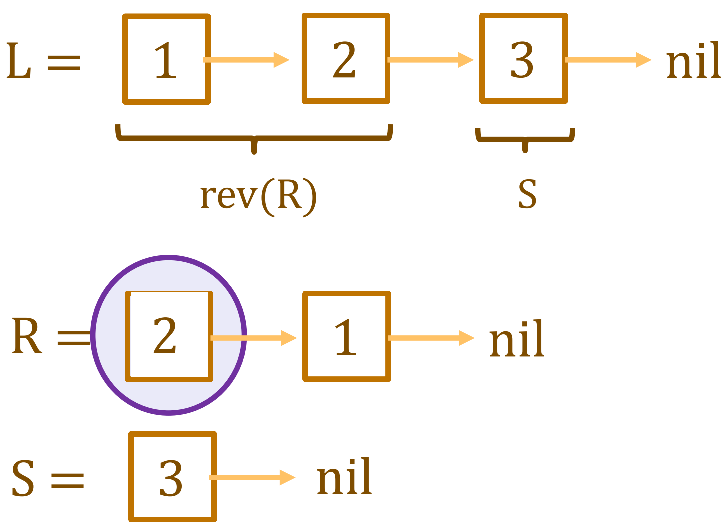


  - **as** R **moves forward,** S **stores a** **<u>suffix</u>** **of** L

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List

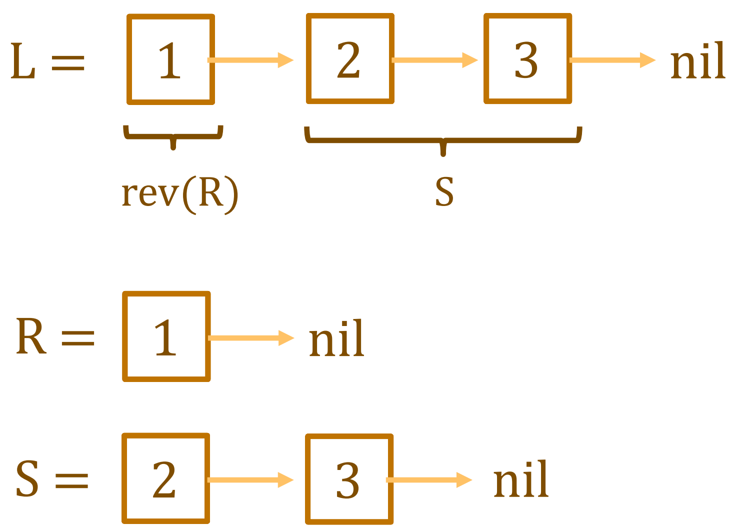
- **Loop idea for calculating** twice(L)**:**
  - **store** rev(L) **in "**R**" initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of "**S**"**



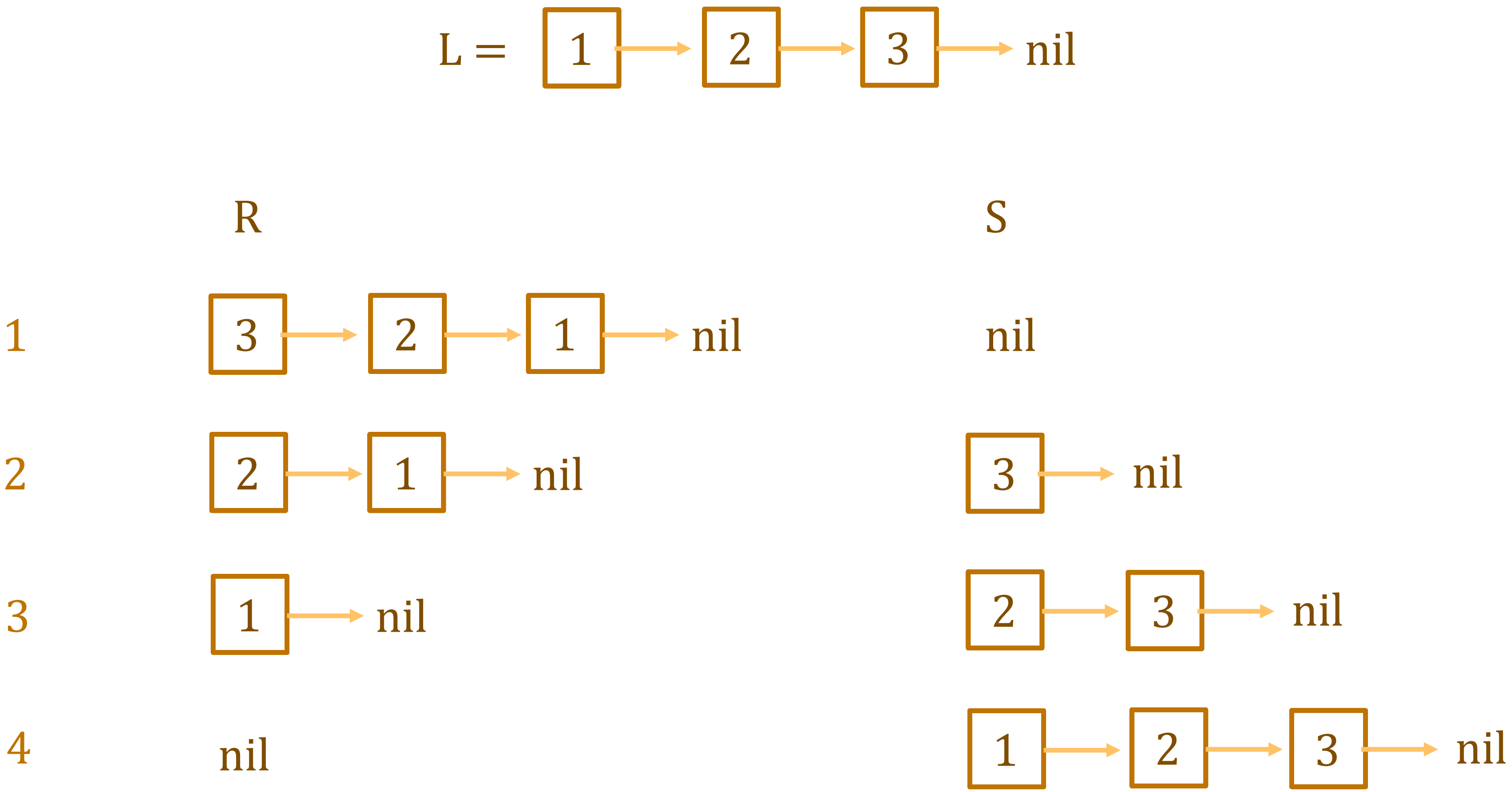
Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : $\mathbb{Z}$ and L : List

- **Loop idea for calculating** twice(L)**:**
  - **store** rev(L) **in "**R**" initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of "**S**"**

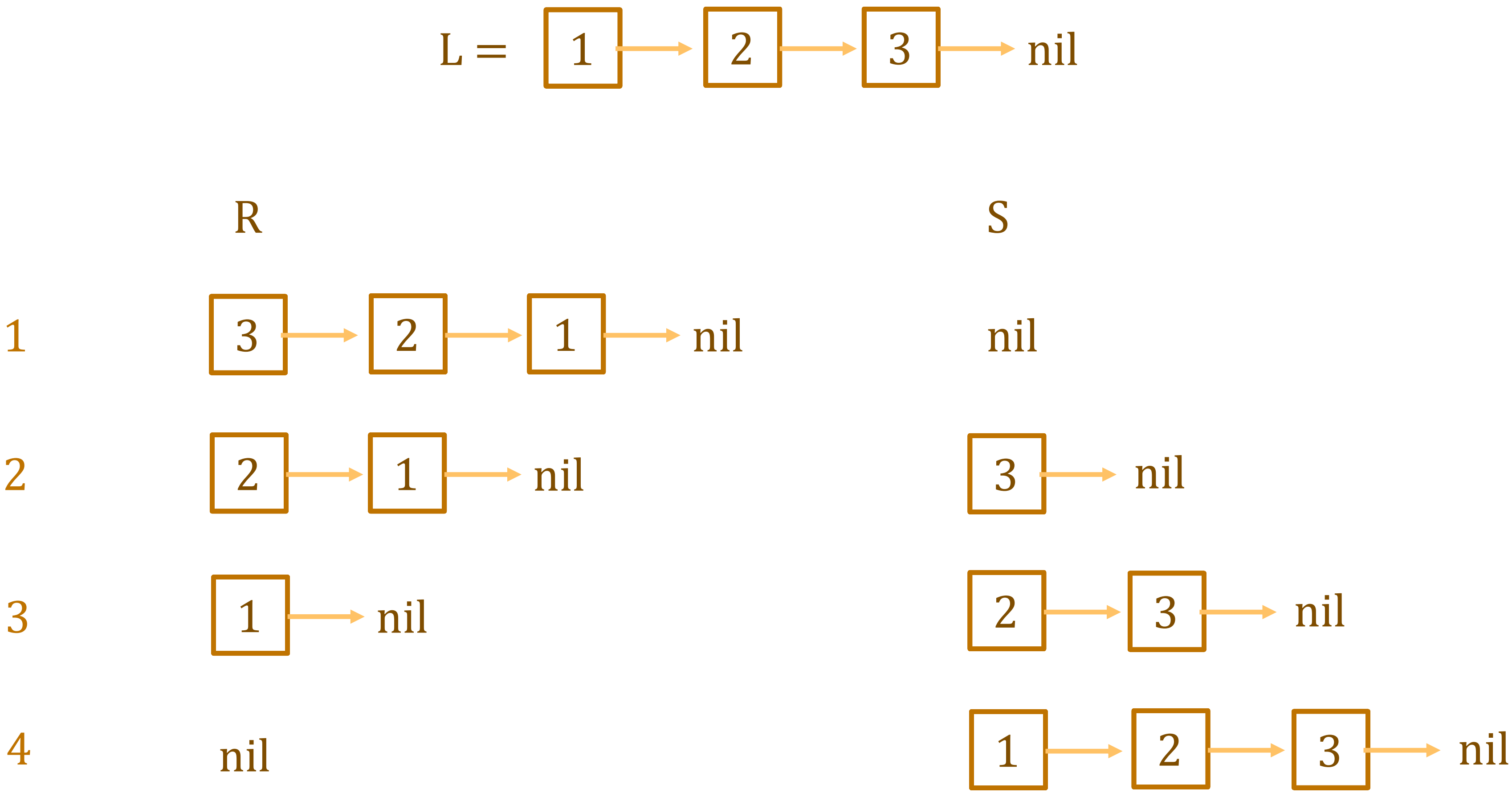


Example "Bottom Up" List Loop



L = concat(rev(R), S)

Example “Bottom Up” List Loop



S **rebuilds the list** L **“bottom up”**

calculate twice(L) **“bottom up” as we go**

Example "Bottom Up" List Loop

func twice(nil) := nil

twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **Loop idea for calculating** twice(L)**:**
  - **store** rev(L) **in "**R**" initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of "**S**"**

    S rebuilds the list L "bottom up"
  - **calculate** twice(S)**, as we go, in "**T**"**

- **Formalize that idea in the loop invariant**

$$L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S)$$

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **This loop claims to calculate** twice(L)**...**

```
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== "nil") {
  T = cons(2n * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
return T;  // = twice(L)
```

Still need to check this.

Hopefully obvious that it could be wrong.
(Testing length 0, 1, 2, 3 is not enough!)

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

```
...
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== "nil") {
  T = cons(2n * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
{{ L = concat(rev(R), S) and T = twice(S) and R = nil }}
{{ T = twice(L) }}
return T;  // = twice(L)
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **Check that** Inv **is implies the postcondition:**

{{ L = concat(rev(R), S) and T = twice(S) **and R = nil** }}
{{ T = twice(L) }}

L = concat(rev(R), S)
= concat(rev(nil), S) **since** R = nil
= concat(nil, S) **def of** rev
= S **def of** concat

T = twice(S)
= twice(L) since L = S

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

```
{{ }}
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
{{ R = rev(L) and S = nil and T = nil }}
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== "nil") {
  T = cons(2n * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
    twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **Check that** Inv **is true initially:**

{{ R = rev(L) and S = nil and T = nil }}
{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

concat(rev(R), S)
 = concat(rev(rev(L)), S)   **since** R = rev(L)
 = concat(L, S)   **Lemma 3**
 = concat(L, nil)   **since** S = nil
 = L   **Lemma 2**

twice(S)
 = twice(nil)   **since** S = nil
 = nil   **def of** twice
 = T   **since** T = nil

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R.kind !== "nil") {
  {{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
  T = cons(2n * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
  {{ L = concat(rev(R), S) and T = twice(S) }}
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R.kind !== "nil") {
```

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}

```
  T = cons(2n * R.hd, T);
  S = cons(R.hd, S);
```

{{ L = concat(rev(R.tl), S) and T = twice(S) }}

```
  R = R.tl;
```

{{ L = concat(rev(R), S) and T = twice(S) }}

```
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R.kind !== "nil") {
  {{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
  T = cons(2n * R.hd, T);
  {{ L = concat(rev(R.tl), cons(R.hd, S)) and T = twice(cons(R.hd, S)) }}
  S = cons(R.hd, S);
  {{ L = concat(rev(R.tl), S) and T = twice(S) }}
  R = R.tl;
  {{ L = concat(rev(R), S) and T = twice(S) }}
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil

twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R.kind !== "nil") {
```

{{ L = concat(rev(R), S) and T = twice(S) and R $\neq$ nil }}

{{ L = concat(rev(R.tl), cons(R.hd, S)) and cons(2·R.hd, T) = twice(cons(R.hd, S)) }}

```
T = cons(2n * R.hd, T);
```

{{ L = concat(rev(R.tl), cons(R.hd, S)) and T = twice(cons(R.hd, S)) }}

```
S = cons(R.hd, S);
```

{{ L = concat(rev(R.tl), S) and T = twice(S) }}

```
R = R.tl;
```

{{ L = concat(rev(R), S) and T = twice(S) }}

```
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **Check that Inv is preserved by the loop body:**

{{ $L = \text{concat}(\text{rev}(R), S)$ and $T = \text{twice}(S)$ and $R \neq \text{nil}$ }}
{{ $L = \text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, S))$ and $\text{cons}(2 \cdot R.hd, T) = \text{twice}(\text{cons}(R.hd, S))$ }}

twice(cons(R.hd, S))
= cons(2 R.hd, twice(S)) **def of** twice
= cons(2 R.hd, T) **since** $T = \text{twice}(S)$

Note that $R \neq \text{nil}$ **means** $R = \text{cons}(R.hd, R.tl)$

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **Check that Inv is preserved by the loop body:**

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
{{ L = concat(rev(R.tl), cons(R.hd, S)) and cons(2·R.hd, T) = twice(cons(R.hd, S)) }}

L	= concat(rev(R), S)	
	= concat(rev(cons(R.hd, R.tl)), S)	**since** R ≠ nil
	= concat(concat(rev(R.tl), cons(R.hd, nil)), S)	**def of** rev
	= concat(rev(R.tl), concat(cons(R.hd, nil), S))	**Lemma 4**
	= concat(rev(R.tl), cons(R.hd, concat(nil, S)))	**def of** concat
	= concat(rev(R.tl), cons(R.hd, S))	**def of** concat

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List

- **This loop claims to calculate** twice(L)

```
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== "nil") {
  T = cons(2n * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
return T;  // = twice(L)
```

"S" is unused! We could remove it.

"S" is useful for proving correctness but it is not needed at run-time. (Example of a "*ghost*" variable.)

"Bottom Up" Loops on Lists

func f(nil) := ...

f(cons(x, L)) := ... f(L) ... for any x : ℤ and L : List

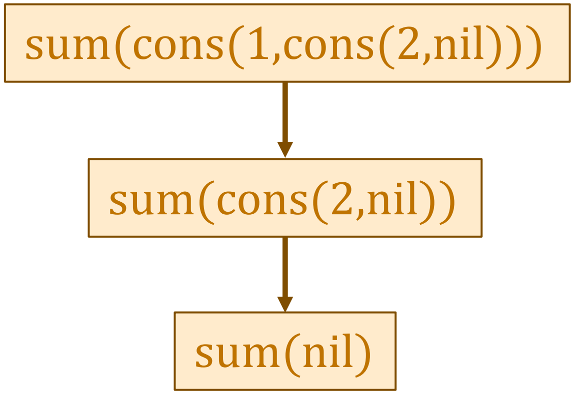
- **Can be implemented with a loop like this**

```
const f = (L: List): List => {
  let R: List = rev(L);
  let S: List = nil;
  let T: List = ...;    // = f(nil)
  {{ Inv: L = concat(rev(R), S) and T = f(S) }}
  while (R.kind !== "nil") {
    T = "... f(L) ..." [f(L) ↦ T]
    S = cons(R.hd, S);
    R = R.tl;
  }
  return T;  // = f(L)
};
```

Recursion versus Loops

$$\textbf{func}\ \mathrm{sum}(\mathrm{nil}) := 0$$
$$\mathrm{sum}(\mathrm{cons}(x, L)) := x + \mathrm{sum}(L) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \mathrm{List}$$

- **This is bottom-up: to calculate** $\mathrm{sum}(\mathrm{cons}(x, L))$
  - **computation order is back-to-front**
  - **recursively calculate** $n = \mathrm{sum}(L)$
  - **when that returns, compute** $x + n$



Recursion versus Loops

$$\textbf{func } \mathrm{sum}(\mathrm{nil}) := 0$$
$$\mathrm{sum}(\mathrm{cons}(x, L)) := x + \mathrm{sum}(L) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \mathrm{List}$$

- **This is bottom-up: to calculate** $\mathrm{sum}(\mathrm{cons}(x, L))$
  - **computation order is back-to-front**
  - **recursively calculate** $n = \mathrm{sum}(L)$
  - **when that returns, compute** $x + n$
- **The natural loop is front-to-back.**
- **This is a fundamental tension!**

Recursion versus Loops

- There is a fundamental tension between:
  - Natural recursive order (bottom-up, aka back-to-front)
  - Natural loop order (front-to-back)

- Three ways to bridge this gap:
  - Make the loop serve the recursion

    We just saw this with the bottom-up list loop template calling $\mathrm{rev}(\mathrm{L})$
  - Make the recursion serve the loop

    Tail recursion, up next
  - Change the data structure

    Arrays

Tail Recursion

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **To calculate** twice(cons(x, L)):
  - **recursively calculate** $S = \text{twice}(L)$
  - **when that returns, construct and return** cons(2x, S)

- **Not all functions require work *after* recursion:**

func rev-acc(nil, R) := R for any R : List
rev-acc(cons(x, L), R) := rev-acc(L, cons(x, R)) for any x : $\mathbb{Z}$ and any L, R : List

  - **such functions are called "tail recursive"**

“Top Down” List Loop

- **We can write a top-down sum function:**

func sum-acc(nil, acc) := acc
sum-acc(cons(x, L), acc) := sum-acc(L, x + acc)

- **Translate to code without reversing the list:**

It’s immediate that the invariant is initially true

```
let s: bigint = 0n;
{{ Inv: sum-acc(L0, 0) = sum-acc(L, s) }}
while (L.kind !== "nil") {
  s = L.hd + s;
  L = L.tl;
}
return s;  // sum-acc(L0, 0)
```

$$\begin{aligned}\text{sum-acc}(L_0, 0) &= \text{sum-acc}(L, s)\\ &= \text{sum-acc}(\text{nil}, s)\\ &= s \qquad \textbf{def}\ \text{sum-acc}\end{aligned}$$

“Top Down” List Loop

- **Check the body preserves invariant**

```
let s: bigint = 0n;
{{ Inv: sum-acc(L0, 0) = sum-acc(L, s) }}
while (L.kind !== “nil”) {
  {{ sum-acc(L0, 0) = sum-acc(L, s) and L ≠ nil }}
  s = L.hd + s;
  L = L.tl;
  {{ sum-acc(L0, 0) = sum-acc(L, s) }}
}
return s;
```

“Top Down” List Loop

- **Check the body preserves invariant**

```
let s: bigint = 0n;
{{ Inv: sum-acc(L_0, 0) = sum-acc(L, s) }}
while (L.kind !== "nil") {
  {{ sum-acc(L_0, 0) = sum-acc(L, s) and L ≠ nil }}
  s = L.hd + s;
  {{ sum-acc(L_0, 0) = sum-acc(L.tl, s) }}
  L = L.tl;
  {{ sum-acc(L_0, 0) = sum-acc(L, s) }}
}
return s;
```

"Top Down" List Loop

- **Check the body preserves invariant**

```
let s: bigint = 0n;
{{ Inv: sum-acc(L₀, 0) = sum-acc(L, s) }}
while (L.kind !== "nil") {
  {{ sum-acc(L₀, 0) = sum-acc(L, s) and L ≠ nil }}
  {{ sum-acc(L₀, 0) = sum-acc(L.tl, L.hd + s) }}
  s = L.hd + s;
  {{ sum-acc(L₀, 0) = sum-acc(L.tl, s) }}
  L = L.tl;
  {{ sum-acc(L₀, 0) = sum-acc(L, s) }}
}
return s;
```

$$\begin{aligned}\text{sum-acc}(L_0, 0) &= \text{sum-acc}(L, s) \\ &= \text{sum-acc}(\text{cons}(L.hd, L.tl), s) && \text{since } L \neq \textbf{nil} \\ &= \text{sum-acc}(L.tl, L.hd + s) && \textbf{def } \text{sum-acc}\end{aligned}$$

"Top down" Loops on Lists

func f(nil, acc) := acc

f(cons(x, L), acc) := **f(L, … x … acc …)**

- Can be implemented with a loop like this

```
const f = (L: List, acc: bigint): List => {
  {{ Inv: f(L₀, acc₀)  = f(L, acc) }}
  while (L.kind !== "nil") {
    acc = "… x … acc …"
    L = L.tl;
  }
  return acc;  // = f(L₀, acc₀)
};
```

Tail Recursion Elimination

- Most functional languages eliminate tail recursion
  - acts like a loop at run-time

    Fast and no extra space usage
  - true of JavaScript as well

- Alternatives implementing recursion:
  1. Find a loop that implements it

     check correctness with Floyd logic
  2. Find an equivalent tail-recursive function

     check equivalence with structural induction

Recursion versus Loops

- There is a fundamental tension between:
  - Natural recursive order (bottom-up, aka back-to-front)
  - Natural loop order (front-to-back)

- Three ways to bridge this gap:
  - Make the loop serve the recursion
    - Bottom-up list loop template calling $\mathrm{rev}(\mathrm{L})$
  - Make the recursion serve the loop
    - Tail recursion
  - Change the data structure
    - Arrays, up next