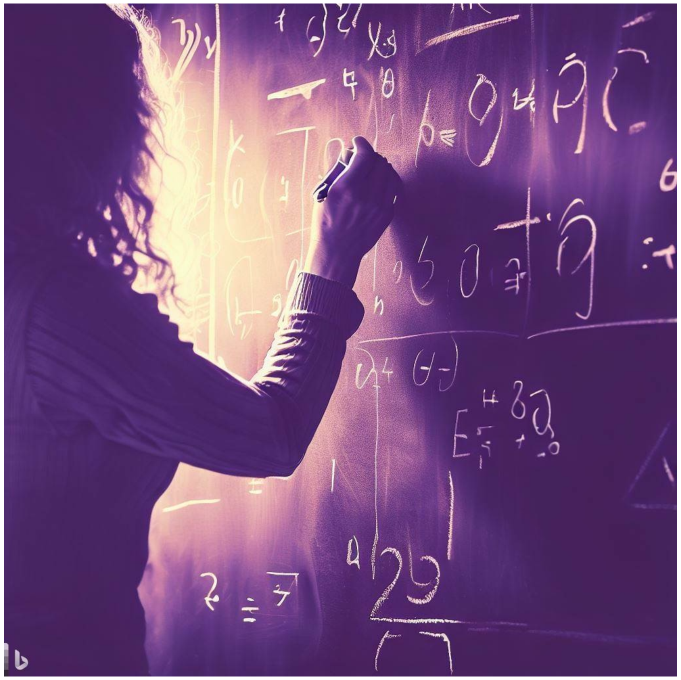


CSE 331

Structural Induction

James Wilcox & Kevin Zatloukal

Recall: Reasoning

- In an intro class, you might be asked:

  what does this code do on this input?

- In this class, we are often interested in:

  what does this code do on **all** *inputs?*

- This is a very different question!

Recall: Reasoning

- "Thinking through" what the code does on <u>all</u> inputs
  - neither testing nor type checking can do this
- Required in principle and in practice
  - you are expected to know what your code does
  - in practice, "reasoning is not optional:
    either reason up front or debug and then reason"
- Very different problem from analyzing specific inputs
  - will require <u>new</u> tools

Recall: Proving Correctness by Calculation

```
const f = (a: bigint, b: bigint): bigint => {
  const L: List = cons(a, cons(b, nil));
  const s: bigint = sum(L);  // = a + b
  …
};
```

- **Can prove the claim in the comments by calculation**

sum(L)	= sum(cons(a, cons(b, nil)))	**since** L = …
	= a + sum(cons(b, nil))	**def of** sum
	= a + b + sum(nil)	**def of** sum
	= a + b	**def of** sum

- **We checked correctness for infinitely many inputs!**

Proof by Calculation

- Our proofs so far have used fixed-length lists
  - e.g., $\mathrm{len}(\mathrm{twice}(\mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil})))) = \mathrm{len}(\mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil})))$
  - problems in HW3 restrict to this case

- Would like to prove correctness on <u>any length</u> list L

- Need more tools for this…
  - structural recursion *calculates* on inductive types
  - structural induction *reasons* about structural recursion

    or more generally, to prove facts containing variables of an inductive type
  - both tools are specific to inductive types

Code Without Mutation

- Code without mutation can be written with just…
  - straight-line code
  - conditionals
  - recursion
- Already saw how to reason about…
  - straight-line code
  - conditionals
- Recursion is all that is left

Structural Induction

Let $\mathrm{P(S)}$ be the claim "$\mathrm{len(twice(S)) = len(S)}$"

To prove $\mathrm{P(S)}$ holds for <u>any</u> list S, prove two implications

Base Case: prove $\mathrm{P(nil)}$

- use any known facts and definitions

Inductive Step: prove $\mathrm{P(cons(x, L))}$ for any $\mathrm{x} : \mathbb{Z}$, $\mathrm{L} : \mathrm{List}$

- x and L are variables
- use any known facts and definitions plus <u>one more fact...</u>
- make use of the fact that L is also a List

Structural Induction

To prove $\mathrm{P}(S)$ holds for any list S, prove two implications

Base Case: prove $\mathrm{P}(\mathrm{nil})$

- **use any known facts and definitions**

Inductive Hypothesis: assume P(L) is true

- **use this in the inductive step, but not anywhere else**

Inductive Step: prove $\mathrm{P}(\mathrm{cons}(x, L))$ for any $x : \mathbb{Z}$, $L : \mathrm{List}$

- **use known facts and definitions and <u>Inductive Hypothesis</u>**

Why This Works

With Structural Induction, we prove two facts

$\mathrm{P}(\mathrm{nil})$ $\quad$ $\mathrm{len}(\mathrm{twice}(\mathrm{nil})) = \mathrm{len}(\mathrm{nil})$

$\mathrm{P}(\mathrm{cons}(\mathrm{x}, \mathrm{L}))$ $\quad$ $\mathrm{len}(\mathrm{twice}(\mathrm{cons}(\mathrm{x}, \mathrm{L}))) = \mathrm{len}(\mathrm{cons}(\mathrm{x}, \mathrm{L}))$

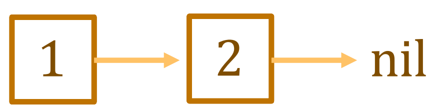
(**second assuming** $\mathrm{len}(\mathrm{twice}(\mathrm{L})) = \mathrm{len}(\mathrm{L})$)

Why is this enough to prove $\mathrm{P}(S)$ for any S : List?

Why This Works

Build up an object using constructors:

nil	**first constructor**
cons(2, nil)	**second constructor**
cons(1, cons(2, nil))	**second constructor**



nil **already exists when building** cons(2, nil)

cons(2, nil) **already exists when building** cons(1, cons(2, nil))

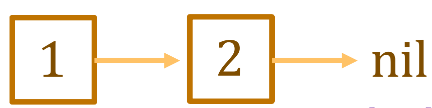
Why This Works

Build up a proof the same way we built up the object

$P(\text{nil})$ $\qquad$ $\text{len}(\text{twice}(\text{nil})) = \text{len}(\text{nil})$

$P(\text{cons}(x, L))$ $\qquad$ $\text{len}(\text{twice}(\text{cons}(x, L))) = \text{len}(\text{cons}(x, L))$

(**second assuming** $\text{len}(\text{twice}(L)) = \text{len}(L)$)

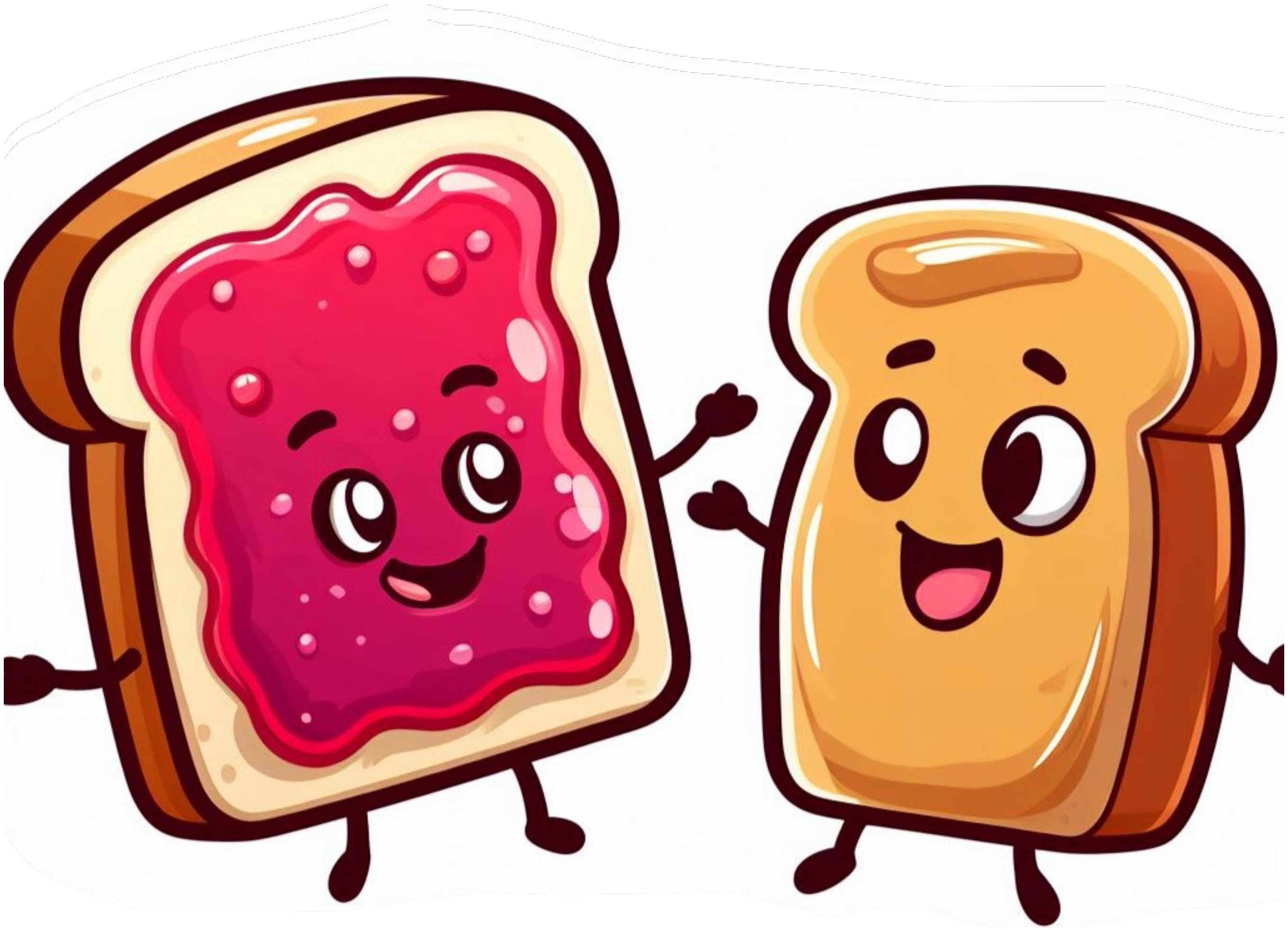


$P(\text{nil})$

$P(\text{nil})$ already proven when proving $P(\text{cons}(2, \text{nil}))$

$P(\text{cons}(2, \text{nil}))$ already proven when proving $P(\text{cons}(1, \text{cons}(2, \text{nil})))$

“We go together”



structural induction

inductive types

Structural Induction in General

- General case: assume P holds for constructor *arguments*

$$\textbf{type}\ \mathrm{T} := \mathrm{A} \mid \mathrm{B}(\mathrm{x} : \mathbb{Z}) \mid \mathrm{C}(\mathrm{y} : \mathbb{Z}, \mathrm{t} : \mathrm{T}) \mid \mathrm{D}(\mathrm{z} : \mathbb{Z}, \mathrm{u} : \mathrm{T}, \mathrm{v} : \mathrm{T})$$

- To prove $\mathrm{P}(\mathrm{t})$ for any t, we need to prove:
  - $\mathrm{P}(\mathrm{A})$
  - $\mathrm{P}(\mathrm{B}(\mathrm{x}))$ for any $\mathrm{x} : \mathbb{Z}$
  - $\mathrm{P}(\mathrm{C}(\mathrm{y}, \mathrm{t}))$ for any $\mathrm{y} : \mathbb{Z}$ and $\mathrm{t} : \mathrm{T}$ — assuming $\mathrm{P}(\mathrm{t})$ is true
  - $\mathrm{P}(\mathrm{D}(\mathrm{z}, \mathrm{u}, \mathrm{v}))$ for any $\mathrm{z} : \mathbb{Z}$ and $\mathrm{u}, \mathrm{v} : \mathrm{T}$ — assuming $\mathrm{P}(\mathrm{u})$ and $\mathrm{P}(\mathrm{v})$

- These four facts are enough to prove $\mathrm{P}(\mathrm{t})$ for any t
  - for each constructor, have proof that it produces an object satisfying P

Structural Induction in General

- General case: assume P holds for constructor *arguments*

$$\textbf{type } \mathrm{T} := \mathrm{A} \mid \mathrm{B}(\mathrm{x} : \mathbb{Z}) \mid \mathrm{C}(\mathrm{y} : \mathbb{Z}, \mathrm{t} : \mathrm{T}) \mid \mathrm{D}(\mathrm{z} : \mathbb{Z}, \mathrm{u} : \mathrm{T}, \mathrm{v} : \mathrm{T})$$

- To prove $\mathrm{P}(\mathrm{t})$ for any t, we need to prove:
  - $\mathrm{P}(\mathrm{A})$
  - $\mathrm{P}(\mathrm{B}(\mathrm{x}))$ for any $\mathrm{x} : \mathbb{Z}$
  - $\mathrm{P}(\mathrm{C}(\mathrm{y}, \mathrm{t}))$ for any $\mathrm{y} : \mathbb{Z}$ and $\mathrm{t} : \mathrm{T}$ assuming $\mathrm{P}(\mathrm{t})$ is true
  - $\mathrm{P}(\mathrm{D}(\mathrm{z}, \mathrm{u}, \mathrm{v}))$ for any $\mathrm{z} : \mathbb{Z}$ and $\mathrm{u}, \mathrm{v} : \mathrm{T}$ assuming $\mathrm{P}(\mathrm{u})$ and $\mathrm{P}(\mathrm{v})$

- Each inductive type has its own form of induction
  - special way to reason about that type

Example: Repeating List Elements

- **Consider the following function:**

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

- **Produces a list where every element is repeated twice**

echo(cons(1, cons(2, nil)))
= cons(1, cons(1, echo(cons(2, nil)))) **def of** echo
= cons(1, cons(1, cons(2, cons(2, echo(nil))))) **def of** echo
= cons(1, cons(1, cons(2, cons(2, nil)))) **def of** echo

Example: Repeating List Elements

$$\begin{aligned} \textbf{func}\ \mathrm{echo}(\mathrm{nil}) &:= \mathrm{nil} \\ \mathrm{echo}(\mathrm{cons}(x, L)) &:= \mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L))) \quad \text{for any } x : \mathbb{Z}, L : \mathrm{List} \end{aligned}$$

- **Suppose we have the following code:**

```
const m: bigint = len(S);      // S is some List
const R: List = echo(S);
…
return 2*m;  // = len(echo(S))
```

not straight from the spec

  - **spec says to return** $\mathrm{len}(\mathrm{echo}(S))$ **but code returns** $2\ \mathrm{len}(S)$

- **Need to prove that** $\mathrm{len}(\mathrm{echo}(S)) = 2\ \mathrm{len}(S)$

Example: Repeating List Elements

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

- **Prove that** $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$ **for any** S : List

Base Case (nil):

Need to prove that $\text{len}(\text{echo}(\text{nil})) = 2\ \text{len}(\text{nil})$

$\text{len}(\text{echo}(\text{nil})) \quad =$

func len(nil) := 0
len(cons(x, L)) := 1 + len(L) for any x ∈ $\mathbb{Z}$ and any L ∈ List

Example: Repeating List Elements

$$\begin{aligned} &\textbf{func}\ \mathrm{echo}(\mathrm{nil}) && := \mathrm{nil} \\ &\quad\ \mathrm{echo}(\mathrm{cons}(x, L)) && := \mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L))) \quad \text{for any } x : \mathbb{Z}, L : \mathrm{List} \end{aligned}$$

- **Prove that** $\mathrm{len}(\mathrm{echo}(S)) = 2\ \mathrm{len}(S)$ **for any** $S : \mathrm{List}$

Base Case (nil):

$$\begin{aligned} \mathrm{len}(\mathrm{echo}(\mathrm{nil})) &= \mathrm{len}(\mathrm{nil}) && \textbf{def of } \mathrm{echo} \\ &= 0 && \textbf{def of } \mathrm{len} \\ &= 2 \cdot 0 \\ &= 2\ \mathrm{len}(\mathrm{nil}) && \textbf{def of } \mathrm{len} \end{aligned}$$

Example: Repeating List Elements

$$\begin{aligned} &\textbf{func}\ \mathrm{echo}(\mathrm{nil}) && := \mathrm{nil} \\ &\qquad \mathrm{echo}(\mathrm{cons}(x, L)) && := \mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L))) \quad \text{for any } x : \mathbb{Z},\ L : \mathrm{List} \end{aligned}$$

- **Prove that** $\mathrm{len}(\mathrm{echo}(S)) = 2\ \mathrm{len}(S)$ **for any** $S : \mathrm{List}$

  Inductive Step $(\mathrm{cons}(x, L))$:

  Need to prove that $\mathrm{len}(\mathrm{echo}(\mathrm{cons}(x, L))) = 2\ \mathrm{len}(\mathrm{cons}(x, L))$

  Get to assume claim holds for L, **i.e., that** $\mathrm{len}(\mathrm{echo}(L)) = 2\ \mathrm{len}(L)$

Example: Repeating List Elements

func $\text{echo}(\text{nil}) \quad := \text{nil}$

$\text{echo}(\text{cons}(x, L)) \quad := \text{cons}(x, \text{cons}(x, \text{echo}(L)))$ for any $x : \mathbb{Z}$, L : List

- **Prove that** $\text{len}(\text{echo}(S)) = 2\,\text{len}(S)$ **for any** S : List

Inductive Hypothesis: assume that $\text{len}(\text{echo}(L)) = 2\,\text{len}(L)$

Inductive Step $(\text{cons}(x, L))$:

$\text{len}(\text{echo}(\text{cons}(x, L)))$

$= 2\,\text{len}(\text{cons}(x, L))$

func $\text{len}(\text{cons}(x, L)) \quad := 1 + \text{len}(L)$ for any $x \in \mathbb{Z}$ and any $L \in$ List

Example: Repeating List Elements

$$
\begin{aligned}
&\textbf{func } \text{echo}(\text{nil}) && := \text{nil} \\
&\quad \text{echo}(\text{cons}(x, L)) && := \text{cons}(x, \text{cons}(x, \text{echo}(L))) \quad \text{for any } x : \mathbb{Z}, L : \text{List}
\end{aligned}
$$

- **Prove that** $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: assume that $\text{len}(\text{echo}(L)) = 2\ \text{len}(L)$

Inductive Step $(\text{cons}(x, L))$:

$$
\begin{aligned}
\text{len}(\text{echo}(\text{cons}(x, L))) &= \text{len}(\text{cons}(x, \text{cons}(x, \text{echo}(L)))) && \textbf{def of } \text{echo} \\
&= 1 + \text{len}(\text{cons}(x, \text{echo}(L))) && \textbf{def of } \text{len} \\
&= 2 + \text{len}(\text{echo}(L)) && \textbf{def of } \text{len} \\
&= 2 + 2\ \text{len}(L) && \textbf{Ind. Hyp.} \\
&= 2(1 + \text{len}(L)) && \\
&= 2\ \text{len}(\text{cons}(x, L)) && \textbf{def of } \text{len}
\end{aligned}
$$

Example 2: Repeating List Elements

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

- **Suppose we have the following code:**

```
const y: bigint = sum(S);      // S is some List
const R: List = echo(S);
...
return 2*y;  // = sum(echo(S))
```

<u>not</u> straight from the spec

  - **spec says to return** $\mathrm{sum}(\mathrm{echo}(S))$ **but code returns** $2\ \mathrm{sum}(S)$

- **Need to prove that** $\mathrm{sum}(\mathrm{echo}(S)) = 2\ \mathrm{sum}(S)$

Example 2: Repeating List Elements

$$\begin{aligned} &\textbf{func}\ \text{echo}(\text{nil}) && := \ \text{nil} \\ &\qquad \text{echo}(\text{cons}(x, L)) && := \text{cons}(x, \text{cons}(x, \text{echo}(L))) \quad \text{for any } x : \mathbb{Z}, L : \text{List} \end{aligned}$$

- **Prove that** $\text{sum}(\text{echo}(S)) = 2\ \text{sum}(S)$ **for any** $S : \text{List}$

Base Case (nil):

$$\text{sum}(\text{echo}(\text{nil})) \quad =$$

$$= 2\ \text{sum}(\text{nil})$$

$$\begin{aligned} &\textbf{func}\ \text{sum}(\text{nil}) && := \ 0 \\ &\qquad \text{sum}(\text{cons}(x, L)) && := \ x + \text{sum}(L) \qquad \text{for any } x \in \mathbb{Z} \text{ and any } L \in \text{List} \end{aligned}$$

Example 2: Repeating List Elements

$$\begin{aligned}
&\textbf{func}\ \mathrm{echo}(\mathrm{nil}) && := \mathrm{nil} \\
&\qquad \mathrm{echo}(\mathrm{cons}(x, L)) && := \mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L))) \quad \text{for any } x : \mathbb{Z},\ L : \mathrm{List}
\end{aligned}$$

- **Prove that** $\mathrm{sum}(\mathrm{echo}(S)) = 2\ \mathrm{sum}(S)$ **for any** $S : \mathrm{List}$

Base Case (nil):

$$\begin{aligned}
\mathrm{sum}(\mathrm{echo}(\mathrm{nil})) &= \mathrm{sum}(\mathrm{nil}) && \textbf{def of}\ \mathrm{echo} \\
&= 0 && \textbf{def of}\ \mathrm{sum} \\
&= 2 \cdot 0 && \\
&= 2\ \mathrm{sum}(\mathrm{nil}) && \textbf{def of}\ \mathrm{sum}
\end{aligned}$$

Inductive Step (cons(x, L)):

Need to prove that $\mathrm{sum}(\mathrm{echo}(\mathrm{cons}(x, L))) = 2\ \mathrm{sum}(\mathrm{cons}(x, L))$

Get to assume claim holds for L**, i.e., that** $\mathrm{sum}(\mathrm{echo}(L)) = 2\ \mathrm{sum}(L)$

Example 2: Repeating List Elements

func $\text{echo}(\text{nil}) := \text{nil}$

$\text{echo}(\text{cons}(x, L)) := \text{cons}(x, \text{cons}(x, \text{echo}(L)))$ for any $x : \mathbb{Z}$, $L : \text{List}$

- **Prove that** $\text{sum}(\text{echo}(S)) = 2\ \text{sum}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: **assume that** $\text{sum}(\text{echo}(L)) = 2\ \text{sum}(L)$

Inductive Step $(\text{cons}(x, L))$:

$\text{sum}(\text{echo}(\text{cons}(x, L))) =$

$= 2\ \text{sum}(\text{cons}(x, L))$

func $\text{sum}(\text{nil}) := 0$

$\text{sum}(\text{cons}(x, L)) := x + \text{sum}(L)$ for any $x \in \mathbb{Z}$ and any $L \in \text{List}$

Example 2: Repeating List Elements

$$
\begin{aligned}
&\textbf{func}\ \mathrm{echo}(\mathrm{nil}) && := \mathrm{nil} \\
&\qquad \mathrm{echo}(\mathrm{cons}(x, L)) && := \mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L))) \quad \text{for any } x : \mathbb{Z},\ L : \mathrm{List}
\end{aligned}
$$

- **Prove that** $\mathrm{sum}(\mathrm{echo}(S)) = 2\ \mathrm{sum}(S)$ **for any** $S : \mathrm{List}$

Inductive Hypothesis: **assume that** $\mathrm{sum}(\mathrm{echo}(L)) = 2\ \mathrm{sum}(L)$

Inductive Step $(\mathrm{cons}(x, L))$**:**

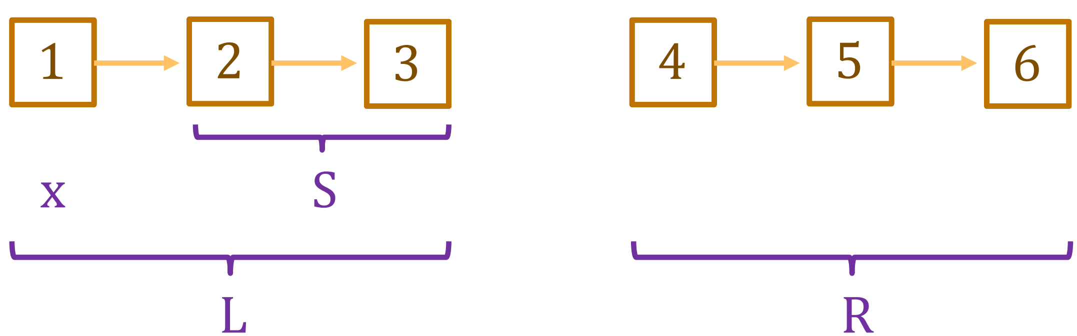
$$
\begin{aligned}
\mathrm{sum}(\mathrm{echo}(\mathrm{cons}(x, L))) &= \mathrm{sum}(\mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L)))) && \textbf{def of } \mathrm{echo} \\
&= x + \mathrm{sum}(\mathrm{cons}(x, \mathrm{echo}(L))) && \textbf{def of } \mathrm{sum} \\
&= 2x + \mathrm{sum}(\mathrm{echo}(L)) && \textbf{def of } \mathrm{sum} \\
&= 2x + 2\ \mathrm{sum}(L) && \textbf{Ind. Hyp.} \\
&= 2(x + \mathrm{sum}(L)) && \\
&= 2\ \mathrm{sum}(\mathrm{cons}(x, L)) && \textbf{def of } \mathrm{sum}
\end{aligned}
$$

Recall: Concatenating Two Lists

- **Mathematical definition of** $\text{concat}(S, R)$

func concat(nil, R)	:=	R	for any $R \in \text{List}$
$\text{concat}(\text{cons}(x, L), R)$	:=	$\text{cons}(x, \text{concat}(L, R))$	for any $x \in \mathbb{Z}$ and any $L, R \in \text{List}$

  – $\text{concat}(S, R)$ **defined by pattern matching on** S **(not** R**)**



Example 3: Length of Concatenated Lists

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

- **Suppose we have the following code:**

```
const m: bigint = len(S);       // S is some List
const n: bigint = len(R);       // R is some List
…
return m + n;  // = len(concat(S, R))
```

<u>not</u> straight from the spec

  – **spec returns** len(concat(S, R)) **but code returns** len(S) + len(R)

- **Need to prove that** len(concat(S, R)) = len(S) + len(R)

Example 3: Length of Concatenated Lists

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

- **Prove that** $\mathrm{len}(\mathrm{concat}(S, R)) = \mathrm{len}(S) + \mathrm{len}(R)$
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

$\mathrm{len}(\mathrm{concat}(\mathrm{nil}, R)) =$

$= \mathrm{len}(\mathrm{nil}) + \mathrm{len}(R)$

Example 3: Length of Concatenated Lists

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

- **Prove that** $\text{len}(\text{concat}(\text{S}, \text{R})) = \text{len}(\text{S}) + \text{len}(\text{R})$
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

$$
\begin{aligned}
\text{len}(\text{concat}(\text{nil}, \text{R})) &= \text{len}(\text{R}) && \textbf{def of } \text{concat} \\
&= 0 + \text{len}(\text{R}) \\
&= \text{len}(\text{nil}) + \text{len}(\text{R}) && \textbf{def of } \text{len}
\end{aligned}
$$

Example 3: Length of Concatenated Lists

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

- **Prove that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

Inductive Step (cons(x, L)):

Need to prove that

$$\text{len}(\text{concat}(\text{cons}(x, L), R)) = \text{len}(\text{cons}(x, L)) + \text{len}(R)$$

Get to assume claim holds for L**, i.e., that**

$$\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)$$

Example 3: Length of Concatenated Lists

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

- **Prove that** len(concat(S, R)) = len(S) + len(R)

Inductive Hypothesis: assume that len(concat(L, R)) = len(L) + len(R)

Inductive Step (cons(x, L))**:**

len(concat(cons(x, L), R)) =

= len(cons(x, L)) + len(R)

func len(nil) := 0
len(cons(x, L)) := 1 + len(L) for any x ∈ $\mathbb{Z}$ and any L ∈ List

Example 3: Length of Concatenated Lists

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

- **Prove that** len(concat(S, R)) = len(S) + len(R)

Inductive Hypothesis: **assume that** len(concat(L, R)) = len(L) + len(R)

Inductive Step (cons(x, L)):

len(concat(cons(x, L), R))	= len(cons(x, concat(L, R)))	**def of** concat
	= 1 + len(concat(L, R))	**def of** len
	= 1 + len(L) + len(R)	**Ind. Hyp.**
	= len(cons(x, L)) + len(R)	**def of** len

Comparing Reasoning vs Testing

```
const concat = (S: List, R: List): List => {
  if (S.kind === "nil") {
    return R;
  } else {
    return cons(S.hd, concat(S.tl, R));
  }
};
```

- Testing: 5 cases
  - 3 subdomains require 1, 2, and 2 cases
- Reasoning: 2 calculations

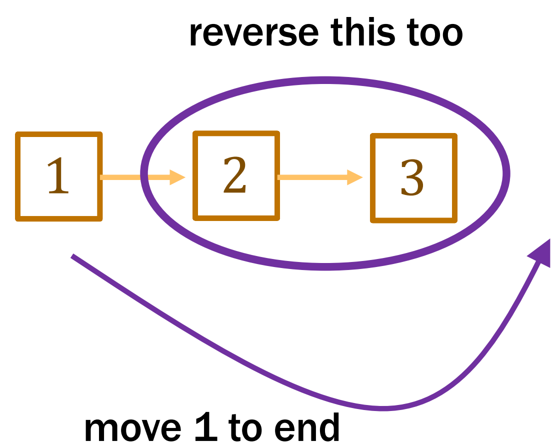
Recall: Reversing a List

- **Mathematical definition of $\mathrm{rev}(S)$**

func rev(nil) := nil

rev(cons(x, L)) := concat(rev(L), cons(x, nil)) for any $x \in \mathbb{Z}$ and any $L \in$ List

– **note that** rev **uses** concat **as a helper function**



Example 4: Length of Reversed List

$$
\begin{aligned}
\textbf{func } & \mathrm{rev}(\mathrm{nil}) && := \mathrm{nil} \\
& \mathrm{rev}(\mathrm{cons}(x, L)) && := \mathrm{concat}(\mathrm{rev}(L), \mathrm{cons}(x, \mathrm{nil})) \quad \text{for any } x : \mathbb{Z} \text{ and} \\
& && \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{any } L : \mathrm{List}
\end{aligned}
$$

- **Suppose we have the following code:**

```
const m: bigint = len(S);      // S is some List
const R: List = rev(S);
…
return m;  // = len(rev(S))
```

not straight from the spec

  - **spec returns** $\mathrm{len}(\mathrm{rev}(S))$ **but code returns** $\mathrm{len}(S)$

- **Need to prove that** $\mathrm{len}(\mathrm{rev}(S)) = \mathrm{len}(S)$ **for any** $S : \mathrm{List}$

Example 4: Length of Reversed List

func $\text{rev}(\text{nil}) := \text{nil}$

$\text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))$ for any $x : \mathbb{Z}$ and any $L : \text{List}$

- **Prove that** $\text{len}(\text{rev}(S)) = \text{len}(S)$ **for any** $S : \text{List}$

Base Case (nil):

$\text{len}(\text{rev}(\text{nil})) = \text{len}(\text{nil})$ **def of** rev

Inductive Step ($\text{cons}(x, L)$):

Need to prove that $\text{len}(\text{rev}(\text{cons}(x, L))) = \text{len}(\text{cons}(x, L))$

Get to assume that $\text{len}(\text{rev}(L)) = \text{len}(L)$

Example 4: Length of Reversed List

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

- **Prove that** $\text{len}(\text{rev}(S)) = \text{len}(S)$ **for any** S : List

Inductive Hypothesis: assume that $\text{len}(\text{rev}(L)) = \text{len}(L)$

Inductive Step (cons(x, L))**:**

$$\begin{aligned} &\text{len}(\text{rev}(\text{cons}(x, L))) \\ &\quad = \\ &\quad = \text{len}(\text{cons}(x, L)) \end{aligned}$$

Example 4: Length of Reversed List

$$
\begin{aligned}
&\textbf{func } \mathrm{rev}(\mathrm{nil}) && := \mathrm{nil} \\
&\quad \mathrm{rev}(\mathrm{cons}(x, L)) && := \mathrm{concat}(\mathrm{rev}(L), \mathrm{cons}(x, \mathrm{nil})) \quad \text{for any } x : \mathbb{Z} \text{ and} \\
& && \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{any } L : \mathrm{List}
\end{aligned}
$$

- **Prove that** $\mathrm{len}(\mathrm{rev}(S)) = \mathrm{len}(S)$ **for any** $S : \mathrm{List}$

Inductive Hypothesis: assume that $\mathrm{len}(\mathrm{rev}(L)) = \mathrm{len}(L)$

Inductive Step $(\mathrm{cons}(x, L))$**:**

$$
\begin{aligned}
&\mathrm{len}(\mathrm{rev}(\mathrm{cons}(x, L))) \\
&\quad = \mathrm{len}(\mathrm{concat}(\mathrm{rev}(L), \mathrm{cons}(x, \mathrm{nil}))) && \textbf{def of } \mathrm{rev} \\
&\quad = \mathrm{len}(\mathrm{rev}(L)) + \mathrm{len}(\mathrm{cons}(x, \mathrm{nil})) && \textbf{by Example 3} \\
&\quad = \mathrm{len}(L) + \mathrm{len}(\mathrm{cons}(x, \mathrm{nil})) && \textbf{Ind. Hyp.} \\
&\quad = \mathrm{len}(L) + 1 + \mathrm{len}(\mathrm{nil}) && \textbf{def of } \mathrm{len} \\
&\quad = \mathrm{len}(L) + 1 && \textbf{def of } \mathrm{len} \\
&\quad = \mathrm{len}(\mathrm{cons}(x, L)) && \textbf{def of } \mathrm{len}
\end{aligned}
$$

Finer Points of Structural Induction

- Structural Induction is how we reason about recursion
- Reasoning also follows structure of code
  - code uses structural recursion, so reasoning uses structural induction
- Note that rev is defined in terms of concat
  - reasoning about $\mathrm{len}(\mathrm{rev}(\ldots))$ used fact about $\mathrm{len}(\mathrm{concat}(\ldots))$
  - this is common

Example 5: Reversing a List

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

- This correctly reverses a list but is slow
  - concat takes $\Theta(n)$ time, where n is length of L
  - n calls to concat takes $\Theta(n^2)$ time
- Can we do this faster?
  - yes, but we need a helper function

Example 5: Reversing a List

- **Helper function** rev-acc(S, R) **for any** S, R : List

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

$$\text{rev-acc}\left(\boxed{3} \to \boxed{4} \to \text{nil},\ \boxed{2} \to \boxed{1} \to \text{nil}\right)$$

$$= \text{rev-acc}\left(\boxed{4} \to \text{nil},\ \boxed{3} \to \boxed{2} \to \boxed{1} \to \text{nil}\right)$$

$$= \text{rev-acc}\left(\text{nil},\ \boxed{4} \to \boxed{3} \to \boxed{2} \to \boxed{1} \to \text{nil}\right)$$

Recall: Reversing a List

$$
\begin{array}{lll}
\textbf{func } \text{rev-acc}(\text{nil}, R) & := R & \text{for any } R : \text{List} \\
\quad \text{rev-acc}(\text{cons}(x, L), R) & := \text{rev-acc}(L, \text{cons}(x, R)) & \text{for any } x : \mathbb{Z} \text{ and} \\
 & & \quad \text{any } L, R : \text{List}
\end{array}
$$

- **Can prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$ **(Lemma 1)**
- **Can prove that** $\text{concat}(L, \text{nil}) = L$ **(Lemma 2)**
  - **structural induction like prior examples**
- **Prove that** $\text{rev}(S) = \text{rev-acc}(S, \text{nil})$

$$
\begin{array}{lll}
\text{rev-acc}(S, \text{nil}) & = \text{concat}(\text{rev}(S), \text{nil}) & \textbf{Lemma 1} \\
 & = \text{rev}(S) & \textbf{Lemma 2}
\end{array}
$$

doing our analysis in math so that
it works for <u>any</u> programming language

Example 5: Helper Lemma 2

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

- **Prove that** $\text{concat}(S, \text{nil}) = S$

Base Case (nil):

concat(nil, nil) = nil **def of** concat

Inductive Hypothesis: assume that concat(L, nil) = nil

Inductive Step (cons(x, L)): **prove that** concat(cons(x, L), nil) = cons(x, L)

Example 5: Helper Lemma 2

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

- **Prove that** $\text{concat}(S, \text{nil}) = S$

Inductive Hypothesis: **assume that** $\text{concat}(L, \text{nil}) = L$

Inductive Step $(\text{cons}(x, L))$:

$$\text{concat}(\text{cons}(x, L), \text{nil}) =$$

$$= \text{cons}(x, L) \qquad \textbf{Ind. Hyp.}$$

Example 5: Helper Lemma 2

func $\text{concat}(\text{nil}, R) := R$ for any R : List

$\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))$ for any $x : \mathbb{Z}$ and any L, R : List

- **Prove that** $\text{concat}(S, \text{nil}) = S$

Inductive Hypothesis: **assume that** $\text{concat}(L, \text{nil}) = L$

Inductive Step $(\text{cons}(x, L))$:

$$\begin{aligned} \text{concat}(\text{cons}(x, L), \text{nil}) &= \text{cons}(x, \text{concat}(L, \text{nil})) && \textbf{def of } \text{concat} \\ &= \text{cons}(x, L) && \textbf{Ind. Hyp.} \end{aligned}$$

Example 5: Helper Lemma 1

$$
\begin{array}{llll}
\textbf{func}\ \text{rev-acc}(\text{nil}, R) & := & R & \text{for any } R : \text{List} \\
\text{rev-acc}(\text{cons}(x, L), R) & := & \text{rev-acc}(L, \text{cons}(x, R)) & \text{for any } x : \mathbb{Z} \text{ and} \\
 & & & \text{any } L, R : \text{List}
\end{array}
$$

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
  - **prove by structural induction**

- **Need the following property of** concat

$$\text{concat}(A, \text{concat}(B, C)) = \text{concat}(\text{concat}(A, B), C) \quad \text{for any } A, B, C : \text{List}$$

  - **with strings, we know that "A + (B + C) = (A + B) + C"**
  - **this says the same thing for lists**

Example 5: Helper Lemma 1

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

- **Prove that** rev-acc(S, R) = concat(rev(S), R)
  - **prove by induction on** S **(so** R **is a variable)**

Base Case (nil):

rev-acc(nil, R) =

= concat(rev(nil), R)

func concat(nil, R) := R concat(cons(x, L), R) := cons(x, concat(L, R))	**func** rev(nil) := nil rev(cons(x, L)) := concat(rev(L), cons(x, nil))

Example 5: Helper Lemma 1

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

- **Prove that** rev-acc(S, R) = concat(rev(S), R)
  - **prove by induction on** S **(so** R **is a variable)**

Base Case (nil):

rev-acc(nil, R)	= R	**def of** rev-acc
	= concat(nil, R)	**def of** concat
	= concat(rev(nil), R)	**def of** rev

func concat(nil, R) := R	**func** rev(nil) := nil
concat(cons(x, L), R) := cons(x, concat(L, R))	rev(cons(x, L)) := concat(rev(L), cons(x, nil))

Example 5: Helper Lemma 1

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

- **Prove that** rev-acc(S, R) = concat(rev(S), R)

  Inductive Hypothesis: assume that rev-acc(L, R) = concat(rev(L), R) **for any** R

  Inductive Step (cons(x, L))**:**

  rev-acc(cons(x, L), R) =

  = concat(rev(cons(x, L)), R)

func concat(nil, R)	:=	R	**func** rev(nil)	:=	nil
concat(cons(x, L), R)	:=	cons(x, concat(L, R))	rev(cons(x, L))	:=	concat(rev(L), cons(x, nil))

Example 5: Helper Lemma 1

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

- **Prove that** rev-acc(S, R) = concat(rev(S), R)

Inductive Hypothesis: assume that rev-acc(L, R) = concat(rev(L), R) **for any** R

Inductive Step (cons(x, L)):

rev-acc(cons(x, L), R)	= rev-acc(L, cons(x, R))	**def of** concat
	= concat(rev(L), cons(x, R))	**Ind. Hyp.**
	= concat(concat(rev(L), cons(x, nil)), R)	??
	= concat(rev(cons(x, L)), R)	**def of** rev

func concat(nil, R) := R concat(cons(x, L), R) := cons(x, concat(L, R))	**func** rev(nil) := nil rev(cons(x, L)) := concat(rev(L), cons(x, nil))

Example 5: Helper Lemma 1

$$
\begin{array}{lll}
\textbf{func}\ \text{rev-acc}(\text{nil}, R) & := R & \text{for any } R : \text{List} \\
\quad \text{rev-acc}(\text{cons}(x, L), R) & := \text{rev-acc}(L, \text{cons}(x, R)) & \text{for any } x : \mathbb{Z} \text{ and} \\
 & & \quad \text{any } L, R : \text{List}
\end{array}
$$

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$

Inductive Hypothesis: **assume that** $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

Inductive Step $(\text{cons}(x, L))$:

$$
\begin{array}{lll}
\text{rev-acc}(\text{cons}(x, L), R) & = \text{rev-acc}(L, \text{cons}(x, R)) & \textbf{def of } \text{concat} \\
 & = \text{concat}(\text{rev}(L), \text{cons}(x, R)) & \textbf{Ind. Hyp.} \\
 & & \\
 & = \text{concat}(\text{rev}(L), \text{concat}(\text{cons}(x, \text{nil}), R)) & \\
 & = \text{concat}(\text{concat}(\text{rev}(L), \text{cons}(x, \text{nil})), R) & \textbf{Prop of } \text{concat} \\
 & = \text{concat}(\text{rev}(\text{cons}(x, L)), R) & \textbf{def of } \text{rev}
\end{array}
$$

$\textbf{func}\ \text{concat}(\text{nil}, R) := R$ $\quad \text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))$	$\textbf{func}\ \text{rev}(\text{nil}) := \text{nil}$ $\quad \text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))$

Example 5: Helper Lemma 1

$$
\begin{array}{lll}
\textbf{func } \text{rev-acc}(\text{nil}, R) & := R & \text{for any } R : \text{List} \\
\quad \text{rev-acc}(\text{cons}(x, L), R) & := \text{rev-acc}(L, \text{cons}(x, R)) & \text{for any } x : \mathbb{Z} \text{ and} \\
 & & \quad \text{any } L, R : \text{List}
\end{array}
$$

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$

  Inductive Hypothesis: assume that $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

  Inductive Step $(\text{cons}(x, L))$**:**

$$
\begin{array}{lll}
\text{rev-acc}(\text{cons}(x, L), R) & = \text{rev-acc}(L, \text{cons}(x, R)) & \textbf{def of } \text{concat} \\
 & = \text{concat}(\text{rev}(L), \text{cons}(x, R)) & \textbf{Ind. Hyp.} \\
 & = \text{concat}(\text{rev}(L), \text{cons}(x, \text{concat}(\text{nil}, R))) & \textbf{def of } \text{concat} \\
 & = \text{concat}(\text{rev}(L), \text{concat}(\text{cons}(x, \text{nil}), R)) & \textbf{def of } \text{concat} \\
 & = \text{concat}(\text{concat}(\text{rev}(L), \text{cons}(x, \text{nil})), R) & \textbf{Prop of } \text{concat} \\
 & = \text{concat}(\text{rev}(\text{cons}(x, L)), R) & \textbf{def of } \text{rev}
\end{array}
$$

$\textbf{func } \text{concat}(\text{nil}, R) := R$ $\quad \text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))$	$\textbf{func } \text{rev}(\text{nil}) := \text{nil}$ $\quad \text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))$

Proof By Cases

Defining Functions by Cases

- Usually combine pattern matching with recursion

- Can use pattern matching on its own

$$
\begin{aligned}
&\textbf{func}\ \ \mathrm{empty}(\mathrm{nil}) && := \mathrm{T} && \\
&\phantom{\textbf{func}}\ \ \mathrm{empty}(\mathrm{cons}(\mathrm{x}, \mathrm{L})) && := \mathrm{F} && \quad \textbf{for any } \mathrm{x} : \mathbb{Z}, \mathrm{L} : \mathrm{List}
\end{aligned}
$$

  – every list is either nil or $\mathrm{cons}(\mathrm{x}, \mathrm{L})$ for some x and L
  – rule can be applied to any list

- Pattern matching is one way to define <u>by cases</u>
  – we've seen another way to do this...

Defining Functions by Cases

- Pattern matching is one way to define <u>by cases</u>

- Side conditions also define by cases
  - e.g., define $f(m)$ where $m : \mathbb{Z}$

$$
\begin{array}{lll}
\textbf{func} & f(m) := 2m + 1 & \quad \textbf{if } m \geq 0 \\
 & f(m) := 0 & \quad \textbf{if } m < 0
\end{array}
$$

  - to use the definition on $f(x)$, need to know if $x < 0$ or not

- Need ways to reason about these functions as well

Proof By Cases

- New code structure means new proof structure
- Can split a proof into cases
  - e.g., $x \geq 0$ and $x < 0$
  - need to be sure the cases are exhaustive

    (don't need to be exclusive in this case)
- If we can prove both cases, it is true in general

Proof By Cases

func $f(m) := 2m + 1$ **if** $m \geq 0$
$f(m) := 0$ **if** $m < 0$

- **Prove that** $f(m) > m$ **for any** $m : \mathbb{Z}$

Case $m \geq 0$:

$f(m) =$

$> m$

Proof By Cases

func $f(m) := 2m + 1$ **if** $m \geq 0$

$f(m) := 0$ **if** $m < 0$

- **Prove that** $f(m) > m$ **for any** $m : \mathbb{Z}$

Case $m \geq 0$:

$f(m) = 2m + 1$ **def of** f (**since** $m \geq 0$)

$\geq m + 1$ **since** $m \geq 0$

$> m$ **since** $1 > 0$

Proof By Cases

func $f(m) := 2m + 1$	**if** $m \geq 0$
$f(m) := 0$	**if** $m < 0$

- **Prove that** $f(m) > m$ **for any** $m : \mathbb{Z}$

Case $m \geq 0$:

$$f(m) = \ldots > m$$

Case $m < 0$:

$f(m) = 0$	**def of** f **(since** $m < 0$**)**
$> m$	**since** $m < 0$

Since these two cases are exhaustive, $f(m) > m$ **holds in general.**

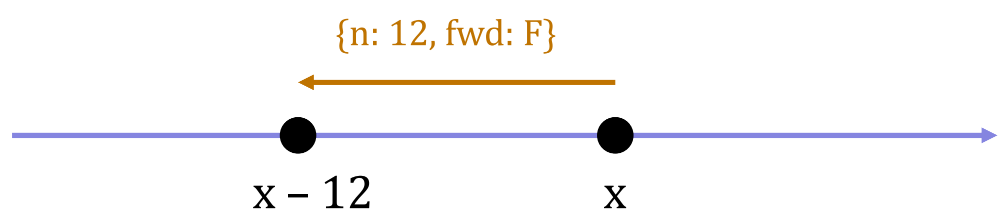
Recall: Pattern Matching

- Define a function by an exhaustive set of patterns

type Steps := {n : ℕ, fwd : 𝔹}

func change({n: n, fwd: T}) := n for any n : ℕ
change({n: n, fwd: F}) := −n for any n : ℕ

  - Steps describes movement on the number line
  - change(s : Steps) says how the position changes



  - one of these two rules always applies

More Proof By Cases

func change({n: n, fwd: T}) := n for any n : $\mathbb{N}$
change({n: n, fwd: F}) := -n for any n : $\mathbb{N}$

- **Prove that** $|\text{change}(s)| = \text{n}$ **for any** $s = \{\text{n: n, fwd: f}\}$
  - **we need to know if** $\text{f} = \text{T}$ **or** $\text{f} = \text{F}$ **to apply the definition!**

Case $\text{f} = \text{T}$:

$|\text{change}(\{\text{n: n, fwd: f}\})|$
$= |\text{change}(\{\text{n: n, fwd: T}\})|$ **since** $\text{f} = \text{T}$
$= |\text{n}|$ **def of** change
$= \text{n}$ **since** $\text{n} \geq 0$

More Proof By Cases

func change({n: n, fwd: T}) := n for any n : $\mathbb{N}$
change({n: n, fwd: F}) := -n for any n : $\mathbb{N}$

- **Prove that** $|\text{change}(s)| = \text{n}$ **for any** $s = \{\text{n: n, fwd: f}\}$

Case f = T: |change({n: n, fwd: f})| = … = n

Case f = F:

|change({n: n, fwd: f})|
= |change({n: n, fwd: F})| **since** f = F
= |-n| **def of** change
= n **since** n $\geq$ 0

Since these two cases are exhaustive, the claim holds in general.

Exceptions

More List Functions

Functions to return the first or last element of a list

func first(nil)	:= ?	
first(cons(x, L))	:= x	for any L : List
func last(nil)	:= ?	
last(cons(x, nil))	:= x	for any x : $\mathbb{Z}$
last(cons(x, cons(y, L))	:= last(cons(y, L))	for any x, y : $\mathbb{Z}$ and any L : List

- Only makes sense for **non-empty** lists
  - there is no first or last element of an empty list
- What do we do when the input is nil?

Partial Functions in Math

Some functions do not have answers for some inputs

func first(nil)	:= undefined	
first(cons(x, L))	:= x	for any L : List
func last(nil)	:= undefined	
last(cons(x, nil))	:= x	for any x : $\mathbb{Z}$
last(cons(x, cons(y, L))	:= last(cons(y, L))	for any x, y : $\mathbb{Z}$ and any L : List

- **In math, we want functions to always be defined, so I had it return "undefined" in this case**
  - **return type is** $\mathbb{Z} \cup \{\text{undefined}\}$

Partial Functions in Code

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```
// L must be a non-empty list
const last = (L: List): bigint => {
  if (L.kind === "nil") {
    throw new Error("empty list! Boooo");
  } else if (L.tl.kind === "nil") {
    return L.hd;
  } else {
    return last(L.tl);
  }
};
```

Partial Functions in Code

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```
// L must be a non-empty list
const last = (L: List): bigint => {
  if (L.kind === "nil") {
    throw new Error("empty list! Boooo");
  …
};
```

- Specification says L will not be nil
  – we assume it is not nil when reasoning
  – do not assume it is not nil at run time

  an example of **defensive programming**

Partial Functions in Code

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```
// L must be a non-empty list
const last = (L: List): bigint => {
  if (L.kind === "nil") {
    throw new Error("empty list! Boooo");
  …
};
```

- In this case, we don't want to return undefined
  - better to "fail fast"…
  - debugging is easier if the crash is closer to the bug

Defensive Programming Rules

- Fine to disallow any inputs you don't want to handle
  - spec can say which inputs are allowed

    (the type system cannot always express this)

- Should also **check** that the inputs are valid
  - throw an exception if not
  - skip this only if the check is too expensive:

    if checking would make the function asymptotically slower, then skip it
  - after you spend 4 hours debugging a problem like this, you'll wish you had written the check

Generics

Lots of Lists of Things

We have now seen lists of

- integers
- squares (`Row` in HW3)
- rows (`Quilt` in HW3)
- HTML elements (`JsxList` in HW3)

These are all "the same" in some sense

- have nil and cons
- cons puts a new value at the front

Generic Types

We can describe this pattern with a "generic" list type

```
type List<A> = {kind: "nil"}
             | {kind: "cons", hd: A, tl: List<A>};
```

- We can pick any type for **A**
  – TypeScript replaces all the "A"s by the type we give
  – e.g., List<bigint> is this type:

```
type List<bigint> =
  | {kind: "nil"}
  | {kind: "cons", hd: bigint, tl: List<bigint>};
```

Generic Types

We can describe this pattern with a “generic” list type

```
type List<A> = {kind: “nil”}
             | {kind: ”cons”, hd: A, tl: List<A>};
```

Can now have

- `List<bigint>` = `List`
- `List<Square>` = `Row`
- `List<List<Square>>` = `Quilt`
- `List<JSX.Element>` = `JsxList`

Generic Types

We can describe this pattern with a "generic" list type

```
type List<A> = {kind: "nil"}
             | {kind: "cons", hd: A, tl: List<A>};
```

- "A" is called a type parameter
- `List` is a function that takes a type as an argument and returns a new type
  - argument is the type of elements, result is list type

    (this is an *analogy* in Java, but it's literally true in TypeScript)
- Illegal to write "`List`" without its argument

Generic Functions

We also need to update the `cons` helper function

```
type List<A> = {kind: “nil”}
             | {kind: ”cons”, hd: A, tl: List<A>};

const cons = <A,>(x: A, L: List<A>): List<A> => {
  return {kind: “cons”, hd: x, tl: L};
};
```

- **This is now a “generic function”**
  - **it has its own type parameter `<A,>`**
  - **extra comma is weird but required**

    compiler thinks `<A>` is an HTML tag

Generic Functions

We also need to update the `cons` helper function

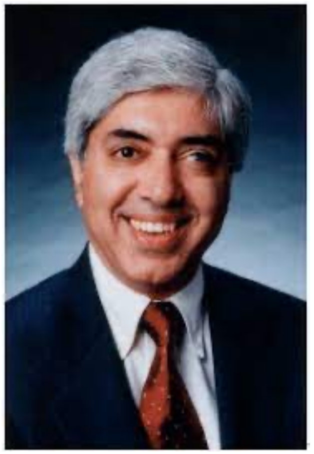
```
type List<A> = {kind: "nil"}
             | {kind: "cons", hd: A, tl: List<A>};

const cons = <A,>(x: A, L: List<A>): List<A> => {
  return {kind: "cons", hd: x, tl: L};
};
```

- Parameters to generic types must be provided
- Parameters to generic functions are usually *inferred*

```
cons(1n, cons(2n, nil))     // has type List<bigint>
```

Generic Types & Functions

- We won't ask you to define generic types this quarter
- But you will need to use them
  - we will use `List<A>` in every assignment from now on
  - lists are the basic data structure of functional programming



Type Erasure

Type Checkers

- Type checkers eliminate large classes of bugs
  - e.g., cannot pass a string where an int is expected
  - critical part of ensuring **correctness**

- Sometimes give you ways to <u>opt out</u> of type checking
  - type casts says “just trust me”
  - “any” type

Run-Time Type Checking

- Java will double-check at **run-time** that you were right
  - type cast will fail with `ClassCastException`
  - however, there are cases where it **cannot** double-check

```
Integer n = (Integer) obj;                     // okay
List<Integer> L = (List<Integer>) obj;   // okay?
```

- Java can do some checks at run-time
  - can check if `obj` is an `Integer`
  - can check if `obj` is a `List<?>` (list of something)
  - **cannot** check if `obj` is a `List<Integer>`!

Run-Time Type Checking

- **Java will double-check at run-time that you were right**
  - **type cast will fail with** `ClassCastException`
  - **however, there are cases where it cannot double-check**

```
Integer n = (Integer) obj;                  // okay
List<Integer> L = (List<Integer>) obj;      // not okay
```

- **Cannot check if** `obj` **is a** `List<Integer>`
  - **all type parameters are "erased"**
  - **all** `List`**s are** `List<Object>` **at run-time**

    if it is correct, it is a `List<Object>` that happens to hold `Integer`s

Type Erasure in Java

```
if (obj instanceof List<Integer>) {      // not okay
```

- Java will give you an **error** on this line
  - it can tell if `L` is a `List`
  - it cannot tell if `L` is a `List<Integer>` (vs `List<String>`)

```
Integer n = (Integer) obj;               // okay
List<Integer> L = (List<Integer>) obj;   // not okay
```

- Java only gives a **warning** about the second cast
  - should really be an **error**
  - programs with these warnings are unsafe

Type Erasure in TypeScript

- In TypeScript, all declared type information is erased!
  - no way to tell what type anything had in the source code

- Type casts are not double-checked at run-time
  - the only run-time type checks are ones <u>you</u> write

- If you use casts or "`any`" types, expect **pain**
  - variables will have values of types you didn't expect
  - code will fail in bizarre ways

Handling Type Erasure

Options for avoiding painful debugging

1. Do not use (unchecked) type casts or "`any`" types
   - almost certainly the best option

2. Check the types yourself at run-time
   - lots of extra work
   - easy to make mistakes
   - (sometimes the only option)

Debugging

A Bug's Life

- **Defect** ("the bug"): mistake made by a human
- **Error**: computation performed incorrectly
- **Failure**: mistake visible to the user

Debugging is the <u>search</u>
from failure back to defect

Debugging

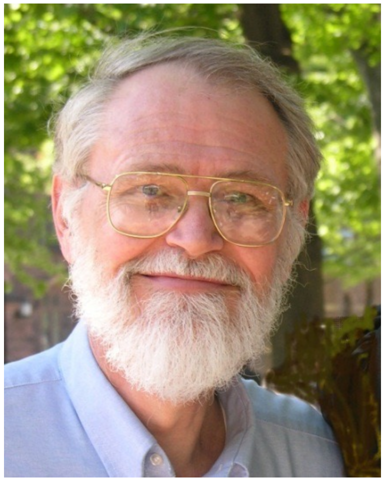
- **Debugging is <u>different</u> from coding**
  - **only happens when states are not as expected**
    - variable has an unexpected type
    - state does not satisfy the expected assertions
- **Never know how long it will take**
  - **happens when you made a mistake in reasoning**
    - initially, you don’t understand what is going on (by definition!)
  - **requires a full understanding of all code involved**
    - could be a lot of code…
  - **important to start early!**

Debugging

- **Debugging is <u>different</u> from coding**
  - **only happens when states are not as expected**
    - variable has an unexpected type
    - state does not satisfy the expected assertions

- **Arguably harder than coding…**

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



Brian Kernighan

  - **write code as simply as possible**
    - if not straight from the spec, then no mutation
    - if not no mutation, then only local variable mutation

Debugging Sucks

- Rule #1 for debugging: **avoid** it

- Tips for *avoiding* debugging:
  1. Write the code as **simply** as possible
     save complication for complicated problems
  2. Apply rigorous testing and **reasoning**
     get the code right the first time
  3. Practice **defensive** programming
     catch errors as quickly as possible (reduce the search space)

- Fight the temptation to skip these steps…

Debugging Sucks

"Easy is the road that leads to destruction."

— Ancient Wisdom (on Programming?)

Debugging Sucks

- Rule #1 for debugging: **avoid** it

- Tips for *avoiding* debugging:
  1. Write the code as **simply** as possible

     save complication for complicated problems
  2. Apply rigorous testing and **reasoning**

     get the code right the first time
  3. Practice **defensive** programming

     catch errors as quickly as possible (reduce the search space)

- Tips for *doing* (surviving) debugging…
  – concise notes published on website

Debugging Tip #1

- Check the easy stuff first
  - make sure all the files are saved
  - restart the server
  - restart your computer
  - make sure someone didn't already fix it

- If it is one of the first 3, you will not find it debugging
  - every minute you spend until you hit save / restart is wasted

Debugging Tip #2

- Create a minimal example that demonstrates the bug
  - easier to look through everything in the debugger

- Shrink the input that fails:

Find “very happy” in “Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all.”

Find “very happy” in “I am very very happy to see you all.”

not the accent characters

Find “very happy” in “very very happy”

something to do with partial match

Find “ab” in “aab”

How to Fix a Bug

- Start with a test that **fails**
  - make sure you see it fail!
  - can mistakenly write a test that worked already
- Understand why it fails
  - understand where your reasoning was wrong
- Fix the bug
- Make sure the all the tests now pass
  - new test and all previous tests

Debugging Tip #3

- Look for common silly mistakes
  - comparing records with `===`
  - misspelling the name of a method you were implementing
    in Java, implementing "`equal`" instead of "`equals`"
  - passing arguments in the wrong order
- Easy for these to slip past reasoning
  - better chance of finding them with tools or testing
    tools will miss wrong order if both arguments have the same type
  - but some will slip through

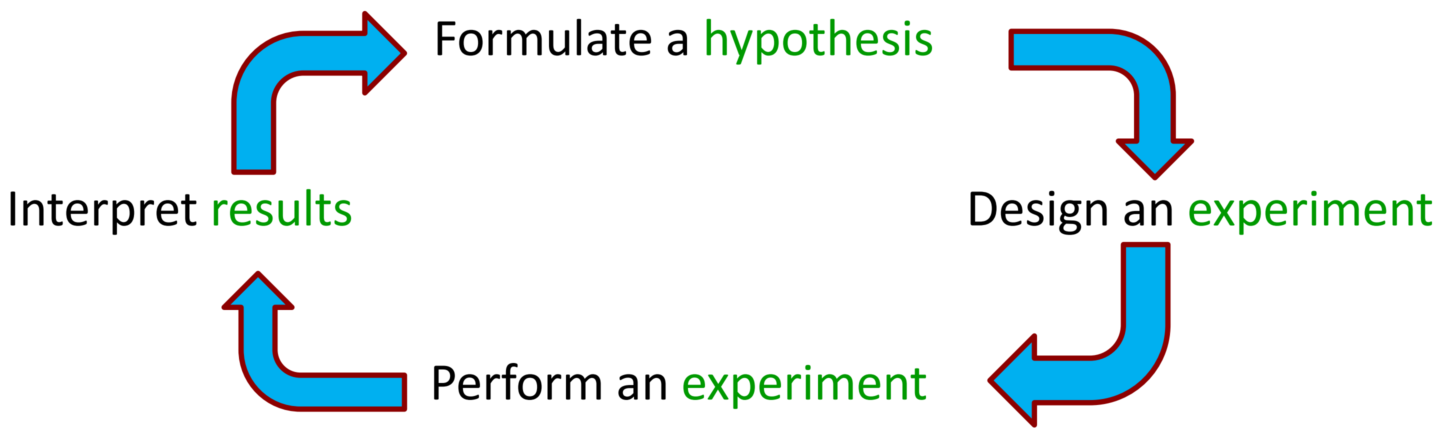
Debugging Tip #4

- Make sure it is a bug!
  - check the spec carefully
  - tricky specs can trick you

- These are the absolute worst
  - spend hours and then discover the code was right all along

Debugging Tip #5

- After 20+ min debugging, be systematic
  - don’t just try things you think might fix it
- Write down what you have tried
  - don’t try the same thing again and again
- Use the Scientific Method:



Debugging Tip #5

- **Use Binary Search to find the error**

  state is good when the object is created

  …

  state is bad when user clicks “submit”

- **Find an event that happens somewhere in the middle**

  state is good when the object is created

  …

  Is the state good when the user clicks on the dropdown?

  …

  state is bad when user clicks “submit”

  - **save an alias to the object when created**

Debugging Tip #6

- Try explaining the problem to someone / something
  - can even be a rubber duck

    Pragmatic Programmer calls this “rubber ducking”

- Talking through the problem often helps you spot it
  - this happens all the time

Debugging Tip #7

- Get some sleep!
  - the later it gets, the dumber I get
  - often don't realize it until 4–5am
- Common to wake up and instantly see the problem
- Important to start early!
  - can't do this the night it is due

Debugging Tip #8

- Get some help!
  - easy for bugs to hide in your blind spots
- After some number of hours, continuing is not helpful
  - need new ideas about where to look
- Important to start early!
  - no office hours late at night

Defensive Programming Tip #4

- If you spent 30+ min debugging, make it a **test case**
  - solid evidence that it's a tricky case
- Bugs that happen once often come back
  - code is changed in the future
  - good chance the same error will happen in the new version
- These are called "regression tests"
  - avoid the bug coming back ("regressing")

Reasoning Is Not Optional

- Debugging happens after a reasoning mistake
  - you missed that case when reasoning
- Fixing one case can break another case
  - increasingly likely as the problems get harder

    fixing individual cases is unlikely to ever make it work on all inputs
  - eventually, you get the reasoning right…