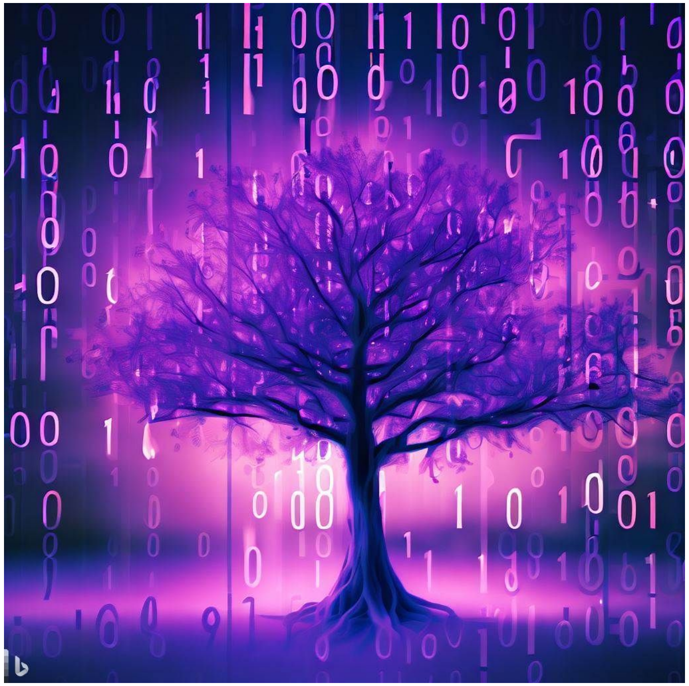


CSE 331

Trees, Structural Induction, Exceptions, Generics

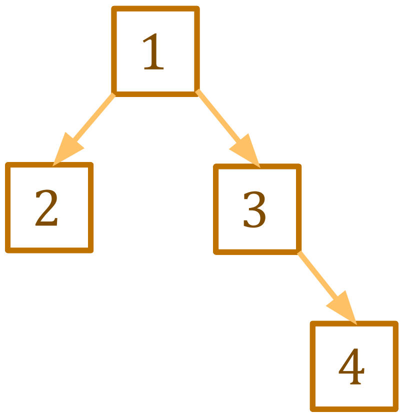
Binary Trees

Binary Trees

$$\textbf{type}\ \text{Tree} := \ \text{empty} \mid \ \text{node}(x : \mathbb{Z}, L : \text{Tree}, R : \text{Tree})$$

- Inductive definition of binary trees of integers

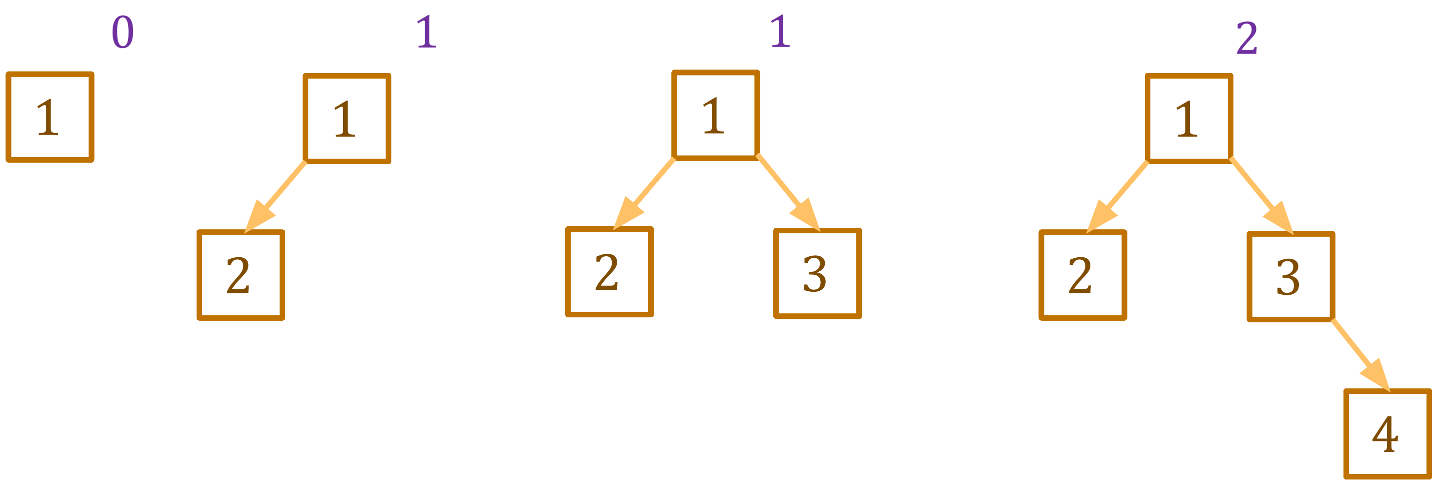
$$\text{node}(1, \text{node}(2, \text{empty}, \text{empty}), \ \text{node}(3, \text{empty}, \text{node}(4, \text{empty}, \text{empty})))$$



Height of a Tree

type Tree := empty | node(x: $\mathbb{Z}$, L: Tree, R: Tree)

- Height of a tree: “maximum steps to get to a leaf”



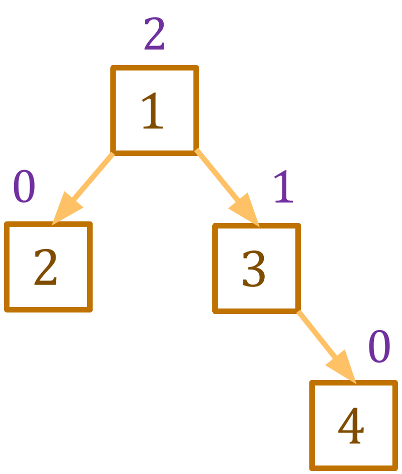
Height of a Tree

type Tree := empty | node(x: ℤ, L: Tree, R: Tree)

- Mathematical definition of height

func height(empty) :=
height(node(x, L, R)) :=

for any x ∈ ℤ and any L, R ∈ Tree



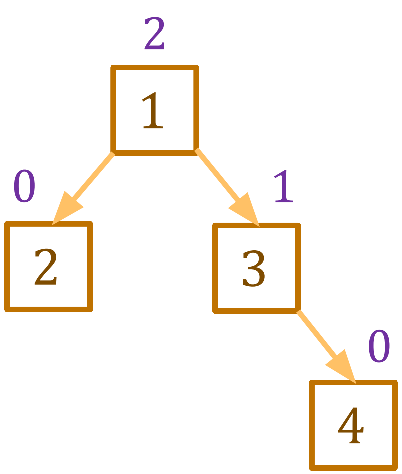
Height of a Tree

type Tree := empty | node(x: ℤ, L: Tree, R: Tree)

- Mathematical definition of height

func height(empty) := −1
height(node(x, L, R)) := 1 + max(height(L), height(R))

for any x ∈ ℤ and any L, R ∈ Tree



Using Definitions in Calculations

$$\begin{aligned} \textbf{func}\ \ & \text{height}(\text{empty}) && := -1 \\ & \text{height}(\text{node}(x, L, R)) && := 1 + \max(\text{height}(L), \text{height}(R)) \\ & && \quad \text{for any } x \in \mathbb{Z} \text{ and any } L, R \in \text{Tree} \end{aligned}$$

- Suppose "$T = \text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))$"
- Prove that $\text{height}(T) = 1$

$\text{height}(T)$

Using Definitions in Calculations

$$
\begin{aligned}
\textbf{func}\ & \text{height}(\text{empty}) && := -1 \\
& \text{height}(\text{node}(x, L, R)) && := 1 + \max(\text{height}(L), \text{height}(R)) \\
& && \quad \text{for any } x \in \mathbb{Z} \text{ and any } L, R \in \text{Tree}
\end{aligned}
$$

- Suppose "$T = \text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))$"
- Prove that $\text{height}(T) = 1$

$$
\begin{aligned}
\text{height}(T) &= \text{height}(\text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))) && \text{since } T = \ldots \\
&= 1 + \max(\text{height}(\text{empty}), \text{height}(\text{node}(2, \text{empty}, \text{empty}))) && \text{def of height} \\
&= 1 + \max(-1, \text{height}(\text{node}(2, \text{empty}, \text{empty}))) && \text{def of height} \\
&= 1 + \max(-1, 1 + \max(\text{height}(\text{empty}), \text{height}(\text{empty}))) && \text{def of height} \\
&= 1 + \max(-1, 1 + \max(-1, -1)) && \text{def of height } (x\ 2) \\
&= 1 + \max(-1, 1 + -1) && \text{def of max} \\
&= 1 + \max(-1, 0) && \\
&= 1 + 0 && \text{def of max} \\
&= 1 &&
\end{aligned}
$$

Trees

- Trees are inductive types with a constructor that has 2+ recursive arguments

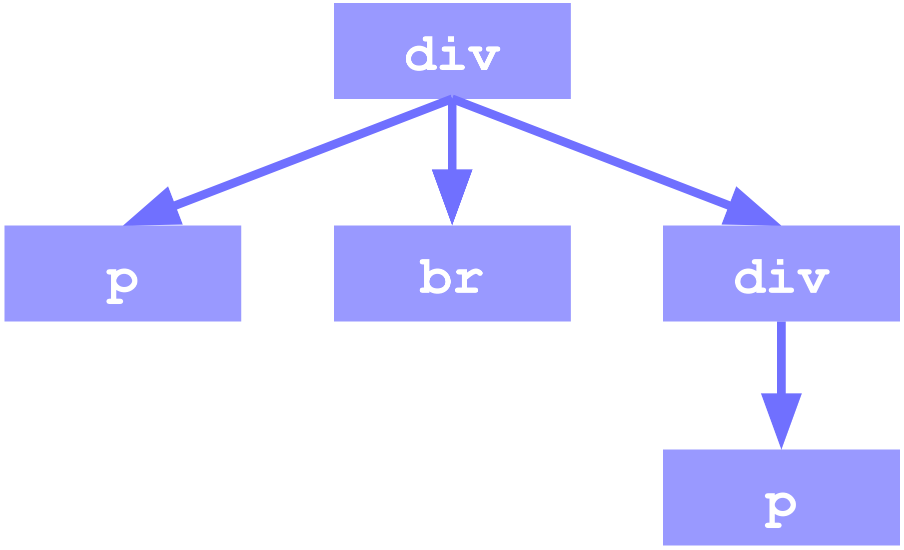
- These come up all the time…
  - no constructors with recursive arguments = “generalized enums”
  - constructor with 1 recursive arguments = “generalized lists”
  - constructor with 2+ recursive arguments = “generalized trees”

- Some prominent examples of trees:
  - HTML: used to describe UI
  - JSON: used to describe just about any data

Recall: HTML

- Nesting structure describes the tree

```
<div>
   <p id=”firstParagraph”> Some Text </p>
   <br>
   <div>
      <p>Hello</p>
   </div>
</div>
```



Custom Tags

- The React library lets you write “custom tags”
  - functions that return HTML

```
return (
  <div>
    <p>Hi, Alice!</p>
    <p>Hi, Bob!</p>
  </div>);
```

can become

```
return (
  <div>
    <SayHi name={“Alice”}/>
    <SayHi name={“Bob”}/>
  </div>);
```

Custom Tags

- The React library lets you write “custom tags”

```
return (
  <div>
    <SayHi name={"Alice"}/>
    <SayHi name={"Bob"}/>
  </div>);
```

makes two calls to this function

```
const SayHi = (props: {name: string}): JSX.Element => {
  return <p>Hi, {props.name}</p>;
};
```

  - attributes are passed as a record argument (“props”)

Custom Tags

```
return (
  <div>
    <SayHi name={“Alice”} lang={“es”}/>
    <SayHi name={“Bob”}/>
  </div>);
```

makes two calls to this function

```
type SayHiProps = {name: string, lang?: string};

const SayHi = (props: SayHiProps): JSX.Element => {
  if (props.lang === “es”) {
    return <p>Hola, {props.name}</p>;
  } else {
    return <p>Hi, {props.name}</p>;
  }
};
```

Custom Tags

- The React library lets you write "custom tags"
  - attributes are passed as a record argument ("props")

- In `render`, React will paste the parts together:

```
<div>
  <SayHi name={"Alice"} lang={"es"}/>
  <SayHi name={"Bob"}/>
</div>
```

becomes

```
<div>
  <p>Hola, Alice!</p>
  <p>Hi, Bob!</p>
</div>
```

Custom Tags

- HTML literal syntax allows any tags

```
return (
  <div>
    <SayHi name={"Alice"} lang={"es"}/>
    <SayHi name={"Bob"}/>
  </div>);
```

  - evaluates to a tree with two nodes with tag name "`SayHi`"
  - this matters when *testing* (comes up in HW3)

- React's `render` method is what calls `SayHi`
  - HTML returned is *substituted* where the "`SayHi`" tag was

React Render

- React's `render` pastes strings together

```
const name: String = "Fred";
return <p>Hi, {name}</p>;
```

returns a different tree than

```
return <p>Hi, Fred</p>;
```

  - in first tree, "p" tag has one child
  - in second tree, "p" tag has two children
  - render method concatenates text children into one string

- These differences matter for **testing**!

React Render

- React’s `render` pastes arrays into child list

```
const L = [<span>Hi</span>, <span>Fred</span>];
return <p>{L}</p>;
```

returns a different tree than

```
return <p><span>Hi</span><span>Fred</span></p>;
```

  - in first tree, “p” tag has one child
  - in second tree, “p” tag has two children
  - render method turns the first into the second

- These differences matter for **testing**!

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</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

Structural Induction

Example: Repeating List Elements

- Consider the following function:

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

- Produces a list where every element is repeated twice

$$
\begin{aligned}
&\mathrm{echo}(\mathrm{cons}(1, \mathrm{cons}(2, \mathrm{nil}))) && \\
&\quad = \mathrm{cons}(1, \mathrm{cons}(1, \mathrm{echo}(\mathrm{cons}(2, \mathrm{nil})))) && \text{def of echo} \\
&\quad = \mathrm{cons}(1, \mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(2, \mathrm{echo}(\mathrm{nil}))))) && \text{def of echo} \\
&\quad = \mathrm{cons}(1, \mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(2, \mathrm{nil})))) && \text{def of echo}
\end{aligned}
$$

Structural Induction

Let $\mathrm{P}(S)$ be the claim "$\mathrm{len}(\mathrm{echo}(S)) = 2 * \mathrm{len}(S)$"

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

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

- use any known facts and definitions

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

- x and L are variables

  if this all you need, then we have "direct proof" (by cases)
- 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 $\mathrm{P}(\mathrm{S})$ holds for any list S, prove two implications

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

- use any known facts and definitions

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

- direct proof
- use known facts and definitions and Inductive Hypothesis

Inductive Hypothesis: assume P(L) is true

use this in the inductive step, but not anywhere else

Why This Works

With Structural Induction, we prove two facts

$\mathrm{P}(\mathrm{nil})$ $\quad$ $\mathrm{len}(\mathrm{echo}(\mathrm{nil})) = 2 * \mathrm{len}(\mathrm{nil})$

$\mathrm{P}(\mathrm{cons}(x, L))$ $\quad$ $\mathrm{len}(\mathrm{echo}(\mathrm{cons}(x, L))) = 2 * \mathrm{len}(\mathrm{cons}(x, L))$

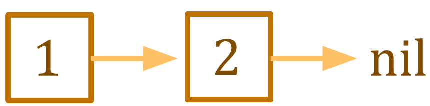
(second assuming $\mathrm{len}(\mathrm{echo}(L)) = \mathrm{len}(L)$)

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

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

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

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

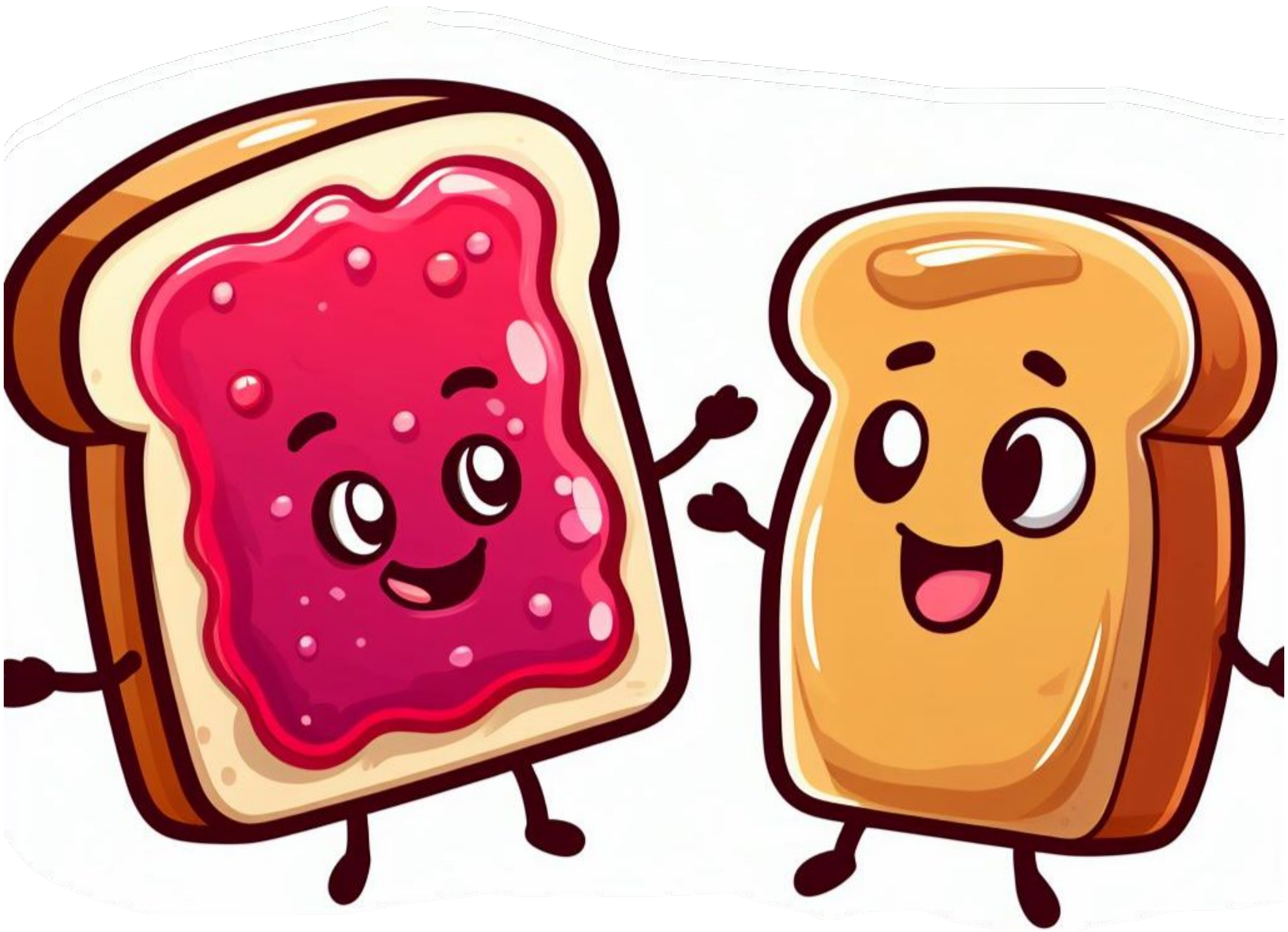
(second assuming $\text{len}(\text{twice}(L)) = \text{len}(L)$)

$P(\text{nil})$

$P(\text{nil})$ 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:

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:

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

- Produces a list where every element is repeated twice

$$
\begin{aligned}
&\text{echo}(\text{cons}(1, \text{cons}(2, \text{nil}))) \\
&\quad = \text{cons}(1, \text{cons}(1, \text{echo}(\text{cons}(2, \text{nil})))) && \text{def of echo} \\
&\quad = \text{cons}(1, \text{cons}(1, \text{cons}(2, \text{cons}(2, \text{echo}(\text{nil}))))) && \text{def of echo} \\
&\quad = \text{cons}(1, \text{cons}(1, \text{cons}(2, \text{cons}(2, \text{nil})))) && \text{def of echo}
\end{aligned}
$$

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

Level 1

  - 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 : ℤ, L : List

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

Example: Repeating List Elements

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

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

Base Case (nil):

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

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

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 $S : \mathrm{List}$

Base Case (nil):

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

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

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

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

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

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

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

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

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

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

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

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 2: Repeating List Elements

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

- Suppose we have the following code:

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

Level 1

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

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

Example 2: Repeating List Elements

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

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

Base Case (nil):

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

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

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

Example 2: Repeating List Elements

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

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

Base Case (nil):

$$\begin{array}{lll} \mathrm{sum}(\mathrm{echo}(\mathrm{nil})) & = \mathrm{sum}(\mathrm{nil}) & \text{def of } \mathrm{echo} \\ & = 0 & \text{def of } \mathrm{sum} \\ & = 2 \cdot 0 & \\ & = 2\ \mathrm{sum}(\mathrm{nil}) & \text{def of } \mathrm{sum} \end{array}$$

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

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

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

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

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

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

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

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

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

Example 2: 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 $\mathrm{sum}(\mathrm{echo}(S)) = 2\ \mathrm{sum}(S)$ for any $S : \mathrm{List}$

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

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

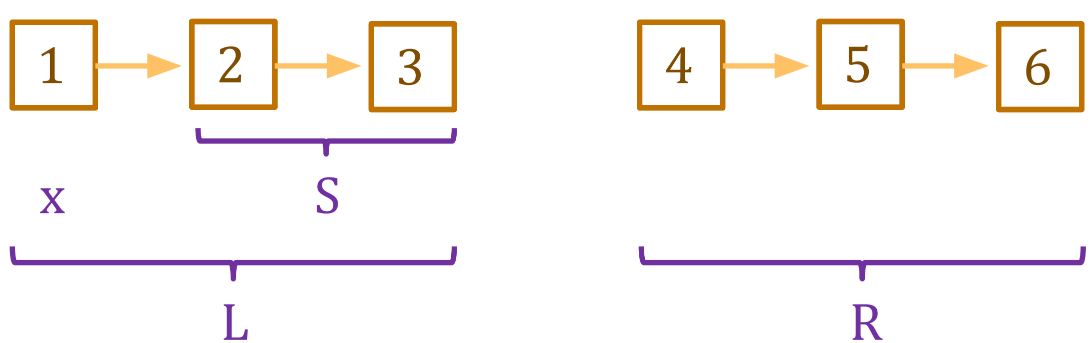
$$\begin{aligned}
\mathrm{sum}(\mathrm{echo}(\mathrm{cons}(x, L))) &= \mathrm{sum}(\mathrm{cons}(x, \mathrm{cons}(x, \mathrm{echo}(L)))) && \text{def of echo} \\
&= x + \mathrm{sum}(\mathrm{cons}(x, \mathrm{echo}(L))) && \text{def of sum} \\
&= 2x + \mathrm{sum}(\mathrm{echo}(L)) && \text{def of sum}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

Recall: Concatenating Two Lists

- Mathematical definition of concat(S, R)

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

  – 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))
```

Level 1

  - spec returns $\text{len}(\text{concat}(S, R))$ but code returns $\text{len}(S) + \text{len}(R)$

- Need to prove that $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

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 $\mathrm{len}(\mathrm{concat}(S, R)) = \mathrm{len}(S) + \mathrm{len}(R)$
  - 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

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

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

Base Case (nil):

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

Example 3: Length of Concatenated Lists

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

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

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

Need to prove that

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

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

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

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

Inductive Hypothesis: assume that $\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)$

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

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

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

Example 3: Length of Concatenated Lists

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

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

Inductive Hypothesis: assume that $\mathrm{len}(\mathrm{concat}(\mathrm{L}, \mathrm{R})) = \mathrm{len}(\mathrm{L}) + \mathrm{len}(\mathrm{R})$

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

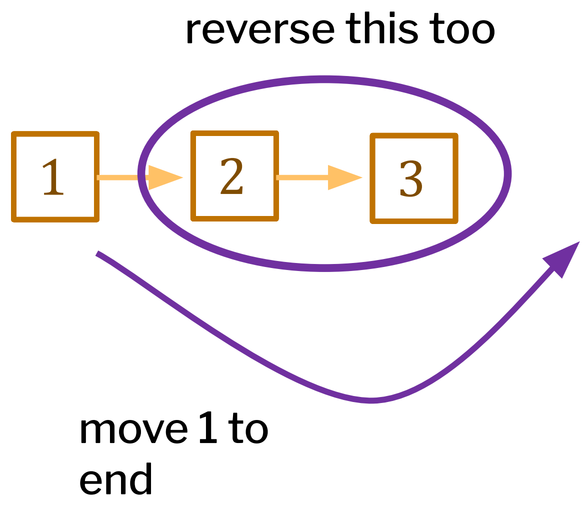
$$
\begin{array}{lll}
\mathrm{len}(\mathrm{concat}(\mathrm{cons}(\mathrm{x}, \mathrm{L}), \mathrm{R})) & = \mathrm{len}(\mathrm{cons}(\mathrm{x}, \mathrm{concat}(\mathrm{L}, \mathrm{R}))) & \text{def of concat} \\
 & = 1 + \mathrm{len}(\mathrm{concat}(\mathrm{L}, \mathrm{R})) & \text{def of len} \\
 & = 1 + \mathrm{len}(\mathrm{L}) + \mathrm{len}(\mathrm{R}) & \text{Ind. Hyp.} \\
 & = \mathrm{len}(\mathrm{cons}(\mathrm{x}, \mathrm{L})) + \mathrm{len}(\mathrm{R}) & \text{def of len}
\end{array}
$$

Recall: Reversing a List

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

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

  - note that rev uses concat as a helper function



Example 4: Length of Reversed List

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

- Suppose we have the following code:

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

Level 1

  – 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

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

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

$\mathrm{len}(\mathrm{rev}(\mathrm{cons}(x, L)))$

$=$

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

Example 4: Length of Reversed List

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

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

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

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

Proof By Cases

Defining Functions by Cases

- Usually combine pattern matching with recursion

- Can use pattern matching on its own

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

  – 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 by cases

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

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

  - 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

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

- 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.

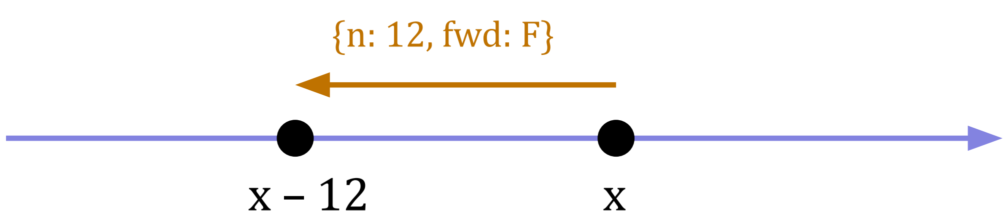
Recall: Pattern Matching

- Define a function by an exhaustive set of patterns

type Steps := $\{n : \mathbb{N}, fwd : \mathbb{B}\}$

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

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

Case $f = T$:

$$
\begin{aligned}
&|\text{change}(\{n\text{: } n\text{, fwd: } f\})| && \\
&\quad = |\text{change}(\{n\text{: } n\text{, fwd: } T\})| && \text{since } f = T \\
&\quad = |n| && \text{def of change} \\
&\quad = n && \text{since } n \geq 0
\end{aligned}
$$

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

Case $f = T$: $|\text{change}(\{n: n, \text{fwd}: f\})| = \ldots = n$

Case $f = F$:

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

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

Reminders

- "Engineers are paid to think and **understand**"
  - you should be able to understand **all** the code in HW3

- Professional programmers are required to
  - **understand** 100% of the code they write
  - **understand** what code does on **100%** of the allowed inputs

- For Level 1+, this requires **reasoning**
  - must use reasoning to think about all allowed inputs

    not okay to give the wrong answer on even one allowed input

Exceptions

More List Functions

Functions to return the first or last element of a list

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{first}(\mathrm{nil}) & := \ ? & \\
\quad \mathrm{first}(\mathrm{cons}(x, L)) & := \ x & \text{for any } L : \mathrm{List}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{last}(\mathrm{nil}) & := \ ? & \\
\quad \mathrm{last}(\mathrm{cons}(x, \mathrm{nil})) & := \ x & \text{for any } x : \mathbb{Z} \\
\quad \mathrm{last}(\mathrm{cons}(x, \mathrm{cons}(y, L))) & := \mathrm{last}(\mathrm{cons}(y, L)) & \text{for any } x, y : \mathbb{Z} \text{ and} \\
 & & \quad \text{any } L : \mathrm{List}
\end{array}
$$

- 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 we have it return "undefined" in this case
  - return type is $\mathbb{Z} \cup \{\text{undefined}\}$

More List Functions

Functions to return the first or last element of a list

func first(cons(x, L))	:= x	for any L : List
func 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

- You may see partial functions defined by non-exhaustive pattern matches.

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 crash is closer to 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};
};
```

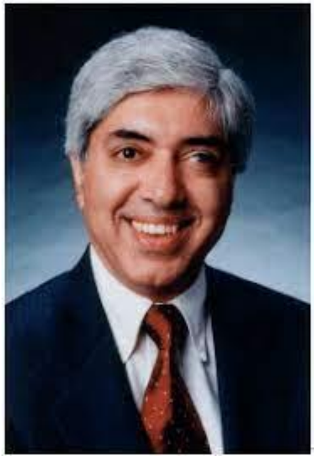
- Parameters to generic types must be provided
- Parameters to generic functions are usually ***inferred***

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

Generic Types & Functions

- We won't ask you to write 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)

More Recursion

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

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

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

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

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

Example 5: Reversing a 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

- Can prove that rev-acc(S, R) = concat(rev(S), R)
  - more on this next time…