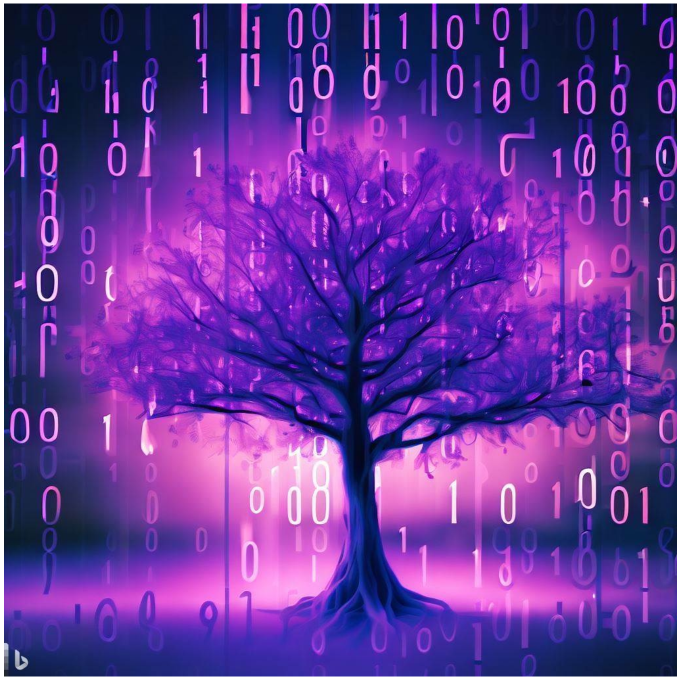


CSE 331

Trees

Kevin Zatloukal

What We Get from Reasoning

- If the proof works, the code is correct
  - why reasoning is useful for finding bugs

- If the code is incorrect, the proof will not work

- If the proof does not work, the code is probably wrong

    could potentially be an issue with the proof (e.g., two “<”s)

    but that is a rare occurrence

Proof by Calculation

Finding Facts at a Return Statement

- Consider this code

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

find facts by reading along path
from top to return statement

- Known facts include "$a \geq 0$", "$b \geq 0$", and "$L = cons(\ldots)$"

Proving Correctness with Conditionals

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

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

$x + y$

Proving Correctness with Conditionals

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

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

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

Proving Correctness with Conditionals

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

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

$x - 1$

Proving Correctness with Conditionals

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

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

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

Proving Correctness with Conditionals

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

- Conditionals give us extra known facts
  - get known facts from
    1. specification
    2. conditionals
    3. constant declarations

find facts by reading along path
from top to the return statement

Proving Correctness with Multiple Claims

- Need to check the claim from the spec at each `return`

- If spec claims multiple facts, then
  we must prove that <u>each</u> of them holds

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

  – multiple known facts: $x : \mathbb{Z}$, $y : \mathbb{Z}$, and $x < y - 1$
  – multiple claims to prove: $x < r$ and $r < y$
    where "r" is the return value
  – requires *two* calculation blocks

Recall: Max With an Imperative Specification

```
// Returns a if a >= b and b if a < b
const max = (a: number, b, number): number => {
  if (a >= b) {
    return a;
  } else {
    return b;
  }
};
```

Level 0

Example Correctness with Conditionals

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

Level 1

- Three different facts to prove at each `return`
- Two known facts in each branch (return value is "r"):
  - then branch: $a \geq b$ and $r = a$
  - else branch: $a < b$ and $r = b$

Example Correctness with Conditionals

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

Know $a \geq b$ **and** $r = a$

- **Correctness of return in "then" branch:**
  - $r = a$ **holds so "**$r = a$ or $r = b$**" holds,**
  - $r = a$ **holds so "**$r \geq a$**" holds, and**

$$
\begin{aligned}
r &= a \\
&\geq b && \text{since } a \geq b
\end{aligned}
$$

Example Correctness with Conditionals

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

Know $a < b$ and $r = b$

- Correctness of return in "else" branch:
  - $r = b$ holds so "$r = a$ or $r = b$" holds,
  - $r = b$ holds so "$r \geq b$" holds, and
  - $r \geq a$ holds since we have $r > a$:

$$\begin{aligned} r &= b \\ &> a && \text{since } a < b \end{aligned}$$

Sum of a List

```
// a and b must be integers
const f = (a: number, b: number): number => {
  const L: List = cons(a, cons(b, nil));
  const s: number = sum(L);   // = a + b
  ...
};
```

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

sum(cons(a, cons(b, nil)))	
$= a + \text{sum}(\text{cons}(b, \text{nil}))$	**def of** sum
$= a + b + \text{sum}(\text{nil})$	**def of** sum
$= a + b$	**def of** sum

func sum(nil) := 0

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

Sum of a List

```
// a and b must be integers
const f = (a: number, b: number): number => {
  const L: List = cons(a, cons(b, nil));
  const s: number = sum(L);   // = a + b
  ...
}
```

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

$$\text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) = \ldots = a + b$$

- **For which values of a and b does this hold?**

holds for **<u>any</u>** $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$

What We Have Proven

- We proved by calculation that

  $$\text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) = a + b$$

- This holds for <u>any</u> $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$

- We have proven *infinitely* many facts
  - $\text{sum}(\text{cons}(3, \text{cons}(5, \text{nil}))) = 8$
  - $\text{sum}(\text{cons}(-5, \text{cons}(2, \text{nil}))) = -3$
  - …
  - replacing all the ‘a’s and ‘b’s with those numbers gives a calculation proving the “=” for those numbers

What We Have Proven

- We proved by calculation that

$$\text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) = a + b \qquad \textbf{for any } a, b \in \mathbb{Z}$$

- We can use this fact for any a and b we choose
  - our proof is a "recipe" that can be used for any a and b
  - just as a function can be used with any argument values, our proof can be used with any values for the "any" variables
    (any values satisfying the specification)
  - use "for any …" to make clear which things are variables

- This is called a "direct proof" of the "for any" claim

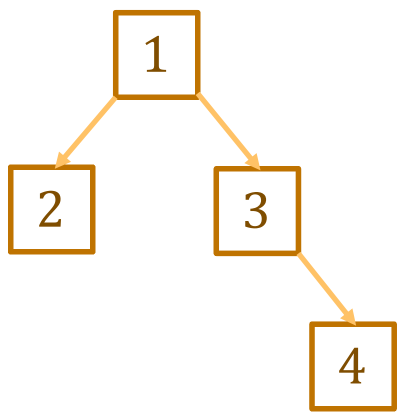
Binary Trees

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**

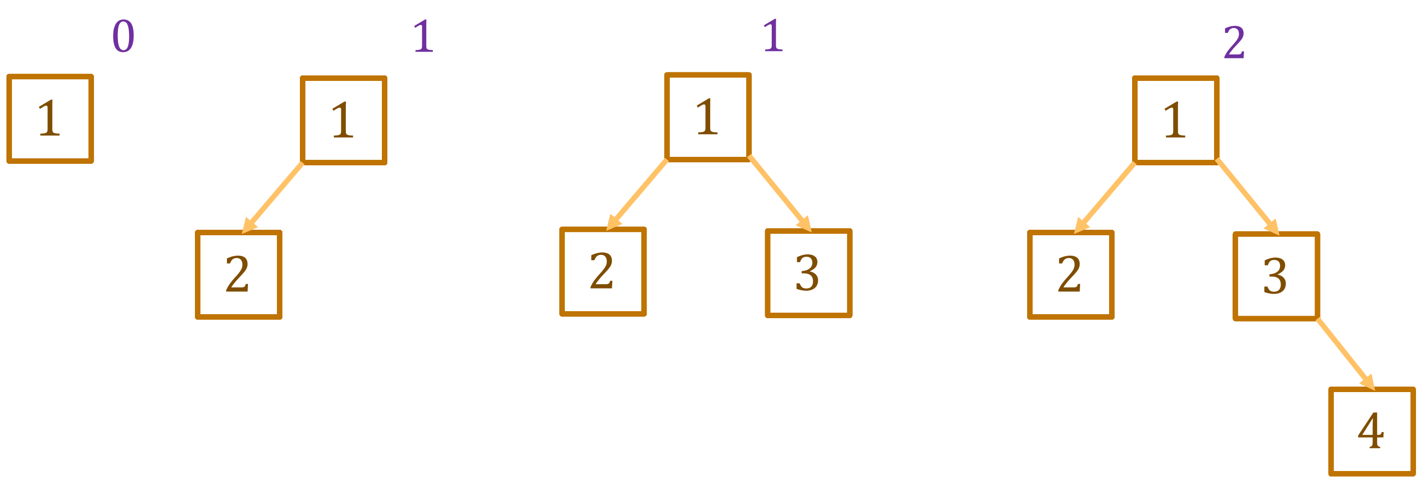
node(1, node(2, empty, empty), node(3, empty, node(4, empty, empty))))



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”**

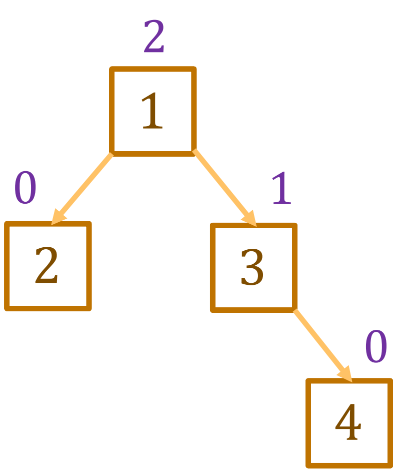


Height of a Tree

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

- **Mathematical definition of height**

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



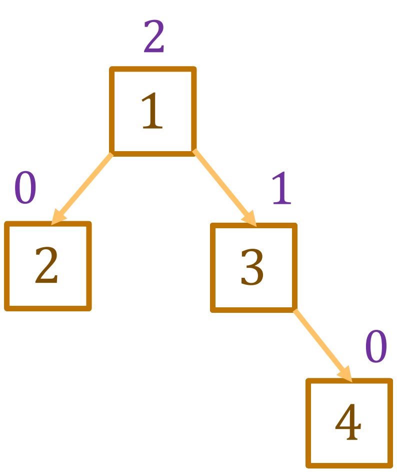
for any x ∈ ℤ and any L, R ∈ Tree

Height of a Tree

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

- **Mathematical definition of height**

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



Using Definitions in Calculations

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

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

height(T) =

Using Definitions in Calculations

func height(empty) := −1
height(node(x, L, R)) := 1 + max(height(L), height(R))
for any $x \in \mathbb{Z}$ and any L, R ∈ Tree

- **Suppose** "T = node(1, empty, node(2, empty, empty))"
- **Prove that** height(T) = 1

height(T)	= height(node(1, empty, node(2, empty, empty))		**since** T = …
	= 1 + max(height(empty), height(node(2, empty, empty)))		**def of** height
	= 1 + max(-1, height(node(2, empty, empty)))		**def of** height
	= 1 + max(-1, 1+ max(height(empty), height(empty)))		**def of** height
	= 1 + max(-1, 1+ max(-1, -1))		**def of** height (x 2)
	= 1 + max(-1, 1+ -1)		**def of** max
	= 1 + max(-1, 0)		
	= 1 + 0		**def of** max
	= 1		

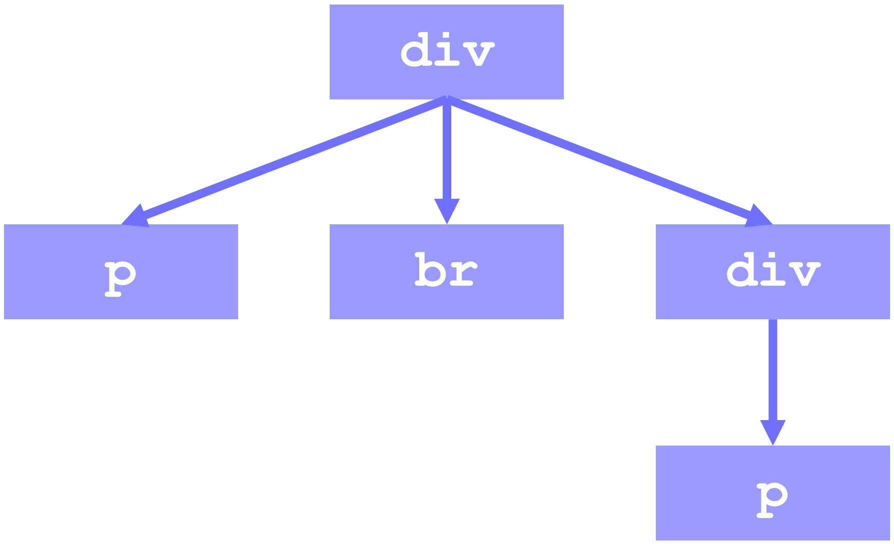
Trees

- 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!**