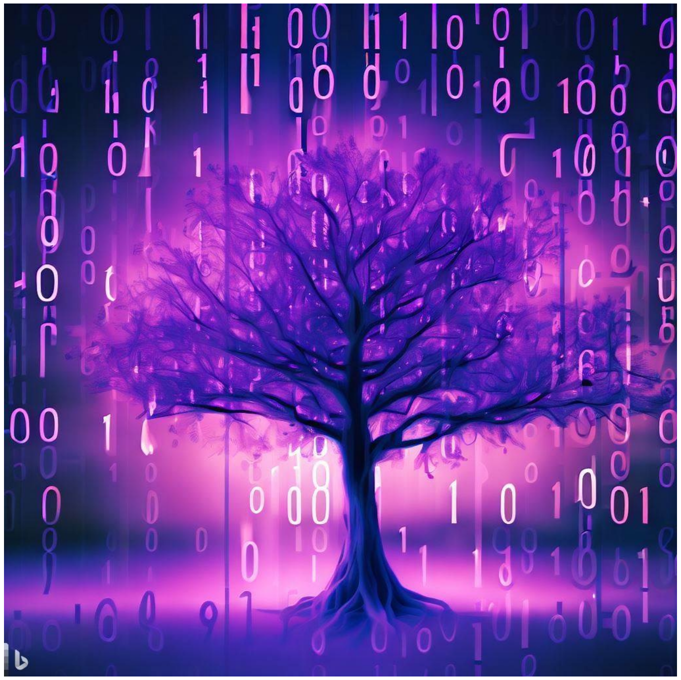


CSE 331

Binary Trees

Kevin Zatloukal

Administrivia

- HW2 released yesterday
  - due next Wednesday by 11pm

- HW2 is much longer than HW1

  HW1 was a ~half assignment

  - HW2 is more coding than paper

  - HW2 has lots of repetition

    lots of new ideas, needs practice

Proof by Calculation

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 each of them holds

```
// Inputs x and y are integers with x < y + 1
// Returns a number less than y and greater than x.
function 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

Example Correctness with Conditionals

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

Level 0

Example Correctness with Conditionals

```
// Returns x with (x=a or x=b) and x >= a and x >= b
function 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 "x"):
  - then branch: $a \geq b$ and $x = a$
  - else branch: $a < b$ and $x = b$

Example Correctness with Conditionals

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

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

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

Example Correctness with Conditionals

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

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

$$\begin{aligned} x &= b \\ &> a && \text{since } a \geq b \text{ is false} \end{aligned}$$

Sum of a List

```
// a and b must be integers
function 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 + sum(cons(b, nil))	**def of** sum
= a + b + sum(nil)	**def of** sum
= a + b	**def of** sum

Sum of a List

```
// a and b must be integers
function 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)

Proofs of "For All" Claims In Math

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

- They would write the proof like this

  Let $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ be any integers.

  [calculation block]

  Since a and b were arbitrary, we have proven the equality for any a and b.

  – in reasoning about code, we'll skip the first and last parts
  – variables in the code are always "any" value of that type

- We won't worry about this distinction
  – some facts use variables, and some don't

Proofs of “For All” Claims

We will learn three ways of proving “for all” claims:

1. Calculation (“Direct Proof”)
2. Proof by Cases
3. Structural Induction

- Saw that the first is just a calculation block.

- Second two gives us a few implications to prove
  - those implications are usually proven by calculation
  - calculation is the workhorse for reasoning w/out mutation

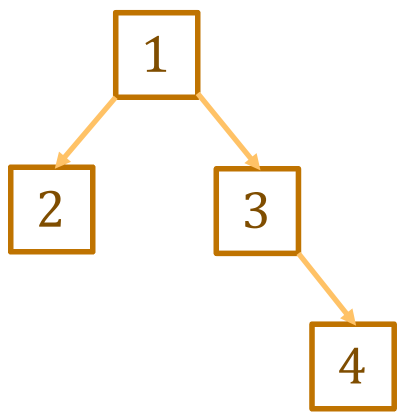
Binary Trees

Binary Trees

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

- **Inductive definition of trees of integers**

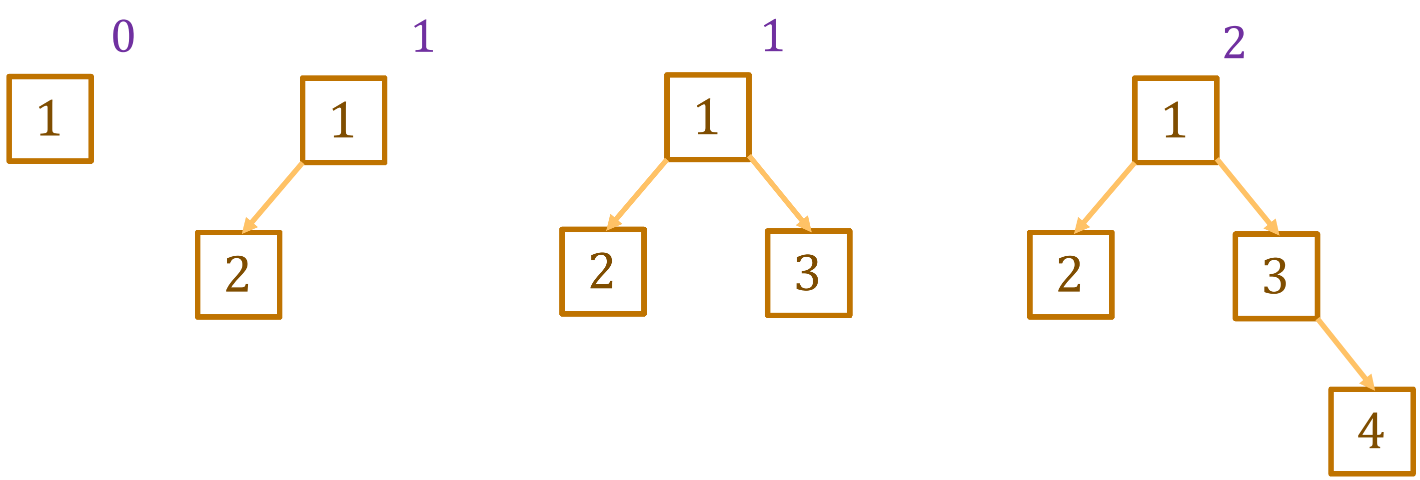
node(1, node(2, empty, empty), node(3, empty, node(4, empty, empty))))



Height of a Tree

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

- **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

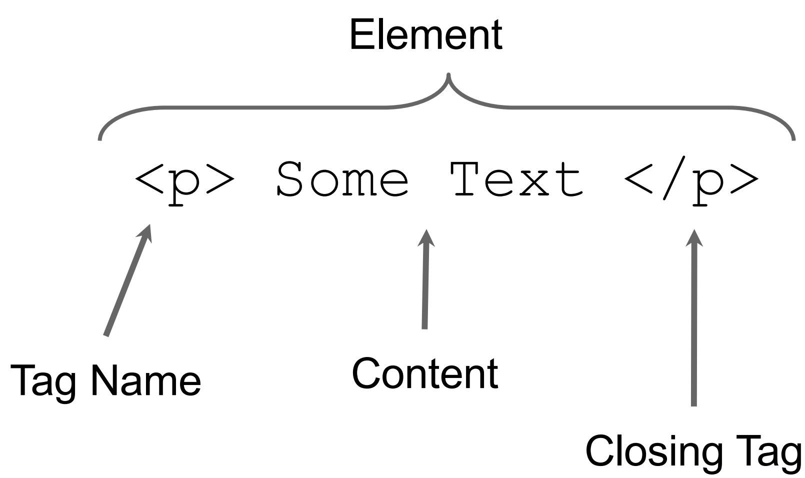
- Trees are inductive types with a constructor that has 2+ recursive arguments

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

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

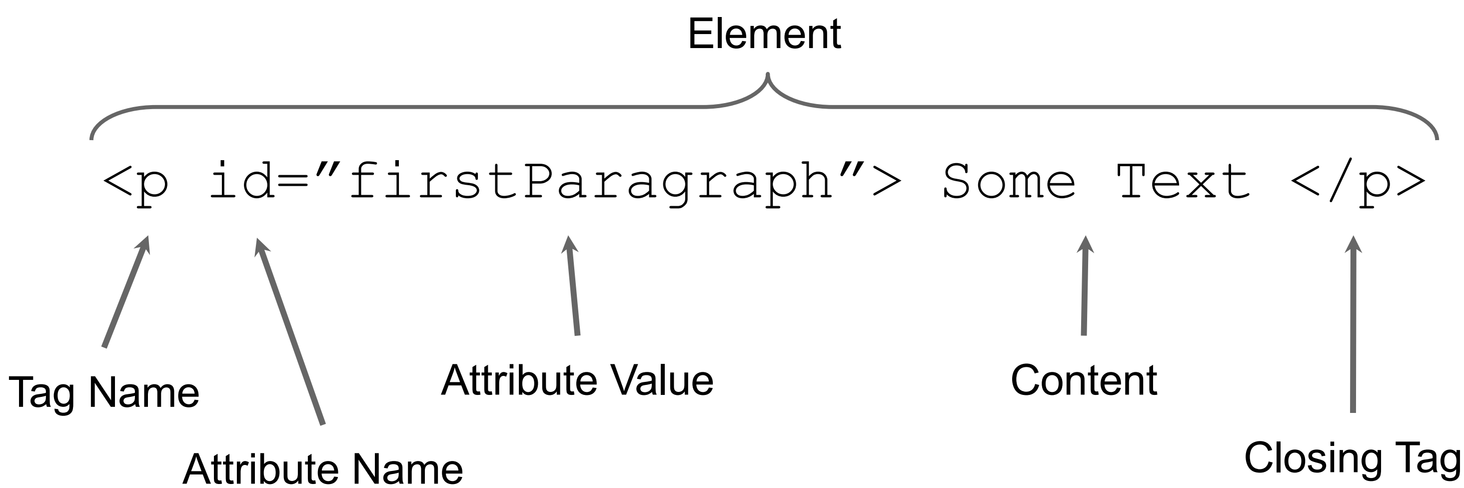
HTML

- Hyper Text Markup Language
  - used to describe UI
  - each document is a tree containing tags and text



HTML

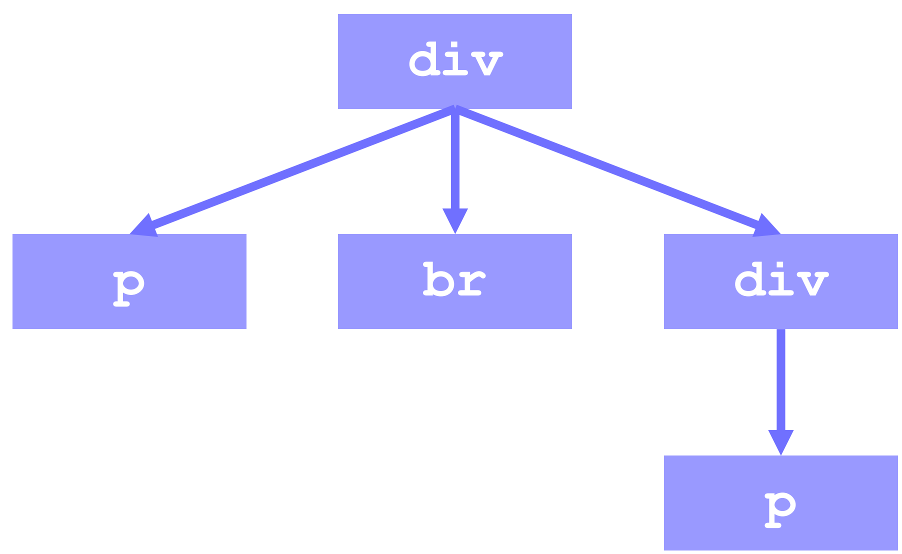
- Hyper Text Markup Language
  - used to describe UI
  - each document is a tree containing tags and text



HTML

- **Nesting structure describes the tree**

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



JSX

- HTML literals are allowed in JS / TS
  - change the file name to `.jsx` or `.tsx`

```
const x = <p>Hi, Fred.</p>;
```

  - if written on multiple lines, you must use (..)

```
const x = (
  <p>
    Hi, Fred.
  </p>);
```

JSX

- HTML literals are allowed in JS / TS
  - can substitute values of expression using {..}

```
const name = "Fred";
const x = <p>Hi, {name}.</p>
```

- Body of P tag becomes "Hi, Fred".
  - arbitrary expressions allowed in `{..}`
- Type checker ensures that the HTML is valid
  - e.g., attribute names exist and are set to valid values

JSX Gotchas

- Put (..) around HTML if more than one line

- Some attribute names are keywords
  - e.g., “class” and “for”
  - instead use “className” and “htmlFor”

- HTML expressions must have one root
  - illegal: `return <p>one</p><p>two<p>;`
  - usually fixed by adding a new parent (e.g., 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

```
function 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};

function 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”)

- At run-time, 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"**
  - **keep this in mind when *testing* (comes up in HW2)**

- **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!