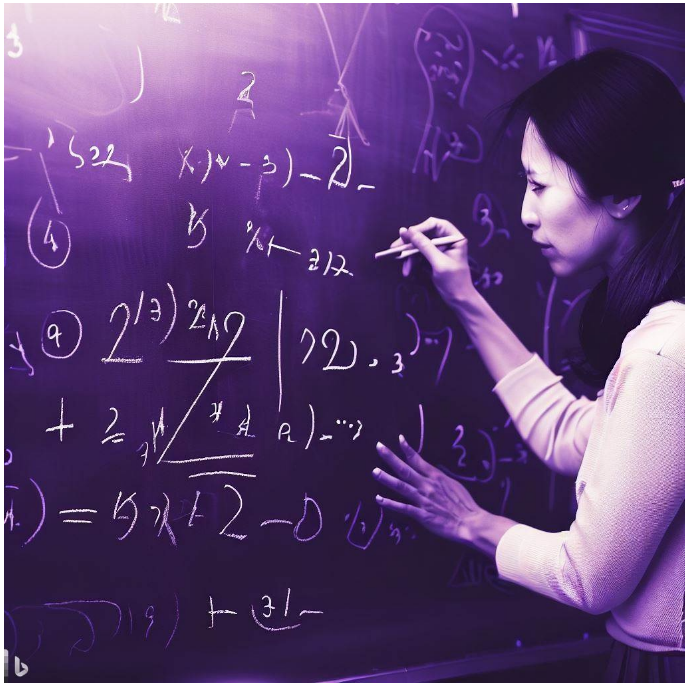


CSE 331

Reasoning About Straight-Line Code

Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox

Inductive Data Types

- **Previous saw records, tuples, and unions**
  - **very useful but limited**
    - can only create types that are “small” in some sense
  - **missing one more way of defining types**
    - arguably the most important

- One critical element is missing: **recursion**
    - Java classes can have fields of same type, but records cannot

- Inductive data types are defined recursively
  - combine union with recursion

Inductive Data Types

- Describe a set by ways of creating its elements
  - each is a "constructor"

$$\textbf{type}\ \mathrm{T} := \ \mathrm{C}(\mathrm{x} : \mathbb{Z}) \ \mid \ \mathrm{D}(\mathrm{x} : \mathbb{Z},\ \mathrm{y} : \mathrm{T})$$

  - second constructor is recursive
  - can have any number of arguments (even none)

will leave off the parentheses when there are none

- Examples of elements

$\mathrm{C}(1)$

$\mathrm{D}(2, \mathrm{C}(1))$

$\mathrm{D}(3, \mathrm{D}(2, \mathrm{C}(1)))$

in math, these are **not** function calls

Inductive Data Types

- **Each element is a description of how it was made**

$$\mathrm{C}(1)$$
$$\mathrm{D}(2, \mathrm{C}(1))$$
$$\mathrm{D}(3, \mathrm{D}(2, \mathrm{C}(1)))$$

- **Equal when they were made *exactly* the same way**
  - $\mathrm{C}(1) \neq \mathrm{C}(2)$
  - $\mathrm{D}(2, \mathrm{C}(1)) \neq \mathrm{D}(3, \mathrm{C}(1))$
  - $\mathrm{D}(2, \mathrm{C}(1)) \neq \mathrm{D}(2, \mathrm{C}(2))$

  - $\mathrm{D}(1, \mathrm{D}(2, \mathrm{C}(3))) = \mathrm{D}(1, \mathrm{D}(2, \mathrm{C}(3)))$

Natural Numbers

$$\textbf{type}\ \mathbb{N} := \text{zero} \mid \text{succ}(n : \mathbb{N})$$

- **Inductive definition of the natural numbers**

zero	0
succ(zero)	1
succ(succ(zero))	2
succ(succ(succ(zero)))	3

The most basic set we have is defined inductively!

Even Natural Numbers

$$\textbf{type}\ \mathbb{E} := \text{zero} \mid \text{two-more}(\text{n} : \mathbb{E})$$

- **Inductive definition of the even natural numbers**

zero	0
two-more(zero)	2
two-more(two-more(zero))	4
two-more(two-more(two-more(zero)))	6

much better notation

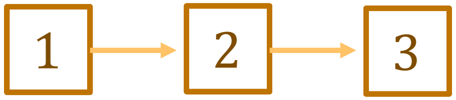
Lists

$$\textbf{type}\ \mathrm{List} := \mathrm{nil}\ \mid\ \mathrm{cons}(x : \mathbb{Z},\ L : \mathrm{List})$$

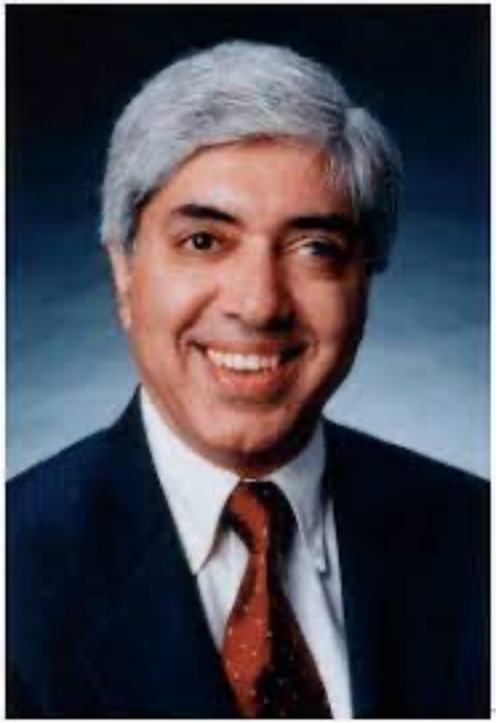
- **Inductive definition of lists of integers**

$$\begin{aligned}
&\mathrm{nil} && \approx [\,] \\
&\mathrm{cons}(3, \mathrm{nil}) && \approx [3] \\
&\mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil})) && \approx [2, 3] \\
&\mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil}))) && \approx [1, 2, 3]
\end{aligned}$$

array notation



“Lists are the original data structure for functional programming,
just as arrays are the original data structure of imperative programming”



Ravi Sethi

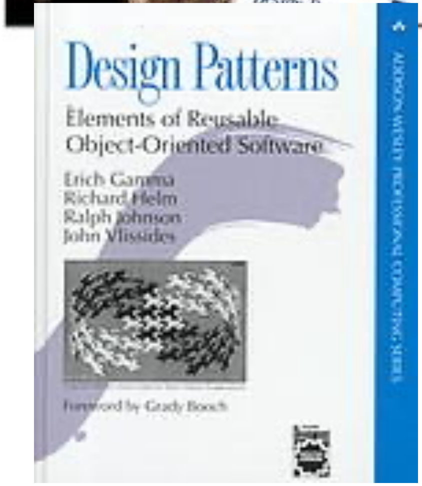
we will work with lists in HW Cipher+ and arrays HW Chatbot+

Inductive Data Types in TypeScript

- TypeScript does not natively support inductive types
  - some “functional” languages do (e.g., OCaml and ML)

- We must think of a way to cobble them together…
  - our answer is a design pattern

Design Patterns



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Introduced in the book of that name
  - written by the "Gang of Four"

    Gamma, Helm, Johnson, Vlissides
  - worked in C++ and SmallTalk
- Found that they independently developed many of the same solutions to recurring problems
  - wrote a book about them
- Many are problems with OO languages
  - authors worked in C++ and SmallTalk
  - some things are <u>not easy</u> to do in those languages

Type Narrowing with Records

- Use a literal field to distinguish records types
  - require the field to have one specific value
  - called a “tag” field

    cleanest way to make unions of records

```
type T1 = {kind: “T1”, a: bigint, b: number};
type T2 = {kind: “T2” a: bigint, b: string};

const x: T1 | T2 = …;
if (x.kind === “T1”) {    // legal for either type
  console.log(x.b);   // must be T1… x.b is a number
} else {
  console.log(x.b);   // must be T2… x.b is a string
}
```

Inductive Data Type Design Pattern

$$\textbf{type}\ \mathrm{T} := \mathrm{C}(\mathrm{x}:\mathbb{Z}) \mid \mathrm{D}(\mathrm{x}:\mathbb{S}^*, \mathrm{t}:\mathrm{T})$$

- Implement in TypeScript as

```
type T = {kind: “C”, x: number}
       | {kind: “D”, x: string, t: T};
```

Inductive Data Type Design Pattern

$$\textbf{type}\ \mathrm{T} := \mathrm{A} \mid \mathrm{B} \mid \mathrm{C}(x : \mathbb{Z}) \mid \mathrm{D}(x : \mathbb{S}^*, t : \mathrm{T})$$

- Implement in TypeScript as

```
type T = {kind: “A”}
       | {kind: “B”}
       | {kind: “C”, x: bigint}
       | {kind: “D”, x: string, t: T};
```

Inductive Data Types in TypeScript

type List := nil | cons(x : $\mathbb{Z}$, L : List)

- Implemented in TypeScript as

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

  - fields should also be “readonly”

How to check if a value `mylist` is nil?

```
if (mylist.kind === “nil”) {
  …
}
```

Inductive Data Types in TypeScript

- **Make this look more like math notation…**

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

const nil: List = {kind: “nil”};

const cons = (hd: bigint, tl: List): List => {
  return {kind: ”cons”, hd: hd, tl: tl};
}
```

  - **use <u>only</u> these two functions to create `List`s**

    do not create the records directly

  - **note that we only have one instance of `nil`**

    this is called a “singleton” (a **design pattern**)

Inductive Data Types in TypeScript

- Make this look more like math notation…

```
const nil: List = {kind: “nil”};

const cons = (hd: bigint, tl: List): List => { .. };
```

- Can now write code like this:

```
const L: List = cons(1, cons(2, nil));

if (L === nil) {
  return L;
} else {
  return cons(L.hd, R);  // head of L followed by R
}
```

if someone made their own nil, then this would fail ☹

and it doesn’t typecheck

Inductive Data Types in TypeScript

- **Make this look more like math notation…**

```
const nil: List = {kind: "nil"};

const cons = (hd: bigint, tl: List): List => { .. };
```

- **Still not perfect:**
  - **JS "===" (references to same object) does not match "="**

```
cons(1, cons(2, nil)) === cons(1, cons(2, nil))  // false!
```

  - **need to define an `equal` function for this**

Inductive Data Types in TypeScript

- **Objects are equal if they were built the same way**

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

const equal = (L: List, R: List): boolean => {
  if (L.kind === “nil”) {
    return R === nil;
  } else {
    if (R.kind === “nil”) {
      return false;
    } else {
      return L.hd === R.hd && equal(L.tl, R.tl);
    }
  }
};
```

Functions

Code Without Mutation

- Saw all types of code without mutation:
  - straight-line code
  - conditionals
  - recursion

- This is all that there is

- Saw TypeScript syntax for these already…

Code Without Mutation

Example function with all three types

```
// n must be a non-negative integer
const f = (n: bigint): bigint => {
  if (n === 0n) {
    return 1n;
  } else {
    return 2n * f(n - 1n);
  }
};
```

What does this compute? **2^n**

Recall: Natural Numbers

$$\textbf{type}\ \mathbb{N} := \ \text{zero} \ \mid \ \text{succ}(\text{prev}: \mathbb{N})$$

- **Inductive definition of the natural numbers**

zero	0
succ(zero)	1
succ(succ(zero))	2
succ(succ(succ(zero)))	3

Recall: Natural Numbers

$$\textbf{type}\ \mathbb{N} := \text{zero} \mid \text{succ}(\text{prev}: \mathbb{N})$$

- Potential definition in TypeScript

```
type Nat = {kind: “zero”}
         | {kind: “succ”, prev: Nat};

const zero: Nat = { kind: “zero” };

const succ = (prev: Nat): Nat => {
  return {kind: ”succ”, prev: prev};
};
```

Induction on Natural Numbers

Could use a type that only allows natural numbers:

```
const f = (n: Nat): bigint => {
  if (n.kind === “zero”) {
    return 1n;
  } else {
    return 2n * f(n.prev);
  }
};
```

n.prev represents “n – 1”

Cleaner definition of the function (though inefficient)

Structural Recursion

- **Inductive types: build new values from existing ones**
  - **only zero exists initially**
  - **build up 5 from 4 (which is built from 3 etc.)**

    4 is the argument to the constructor of 5 = succ(4)

- **Structural recursion: recurse on smaller parts**
  - **call on n recurses on n.prev**

    n.prev is the argument to the constructor (succ) used to create n
  - **guarantees no infinite loops!**

    limit to structural recursion whenever possible

- **We will try to restrict ourselves to structural recursion**
  - **for both math and TypeScript**

Defining Functions in Math

- Saw math notation for defining functions, e.g.:

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

- We need recursion to define interesting functions
  - we will primarily use structural recursion

- Inductive types fit esp. well with *pattern matching*
  - every object is created using some constructor
  - match based on which constructor was used (last)

Length of a List

type List := nil | cons(hd: $\mathbb{Z}$, tl: List)

- **Mathematical definition of length**

```
func len(nil)            := 0
     len(cons(x, S))     := 1 + len(S)        for any x ∈ ℤ
                                               and any S ∈ List
```

  - any list is either nil or cons(x, L) for some x and L
  - cases are exclusive and exhaustive

Length of a List

- **Mathematical definition of length**

func len(nil) := 0

len(cons(x, S)) := $1 + \text{len}(S)$ for any $x \in \mathbb{Z}$

and any $L \in$ List

- **Translation to TypeScript**

```
const len = (L: List): bigint => {
  if (L.kind === “nil”) {
    return 0n;
  } else {
    return 1n + len(L.tl);
  }
};
```

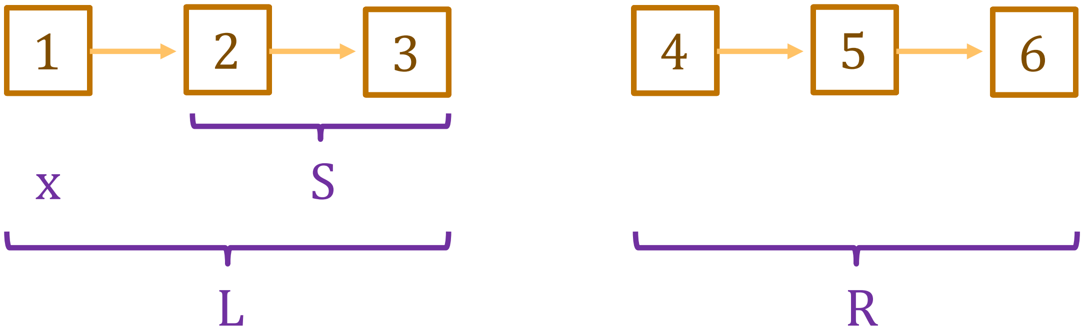
straight from the spec

Concatenating Two Lists

- **Mathematical definition of** concat(L, R)

```
func concat(nil, R)          :=  R                        for any R ∈ List
     concat(cons(x, S), R)   :=  cons(x, concat(S, R))    for any x ∈ ℤ and
                                                                any S, R ∈ List
```

  - concat(L, R) **defined by pattern matching on** L **(not** R**)**



Concatenating Two Lists

- **Mathematical definition of** concat(L, R)

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

- **Translation to TypeScript**

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

straight from the spec

Example

- See ex3 on the course website
  - Simple use of Nat in a webapp

Formalizing Specifications

Correctness Levels

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

“straight from spec” requires us to have a formal spec!

Formalizing a Specification

- Sometimes the instructions are written in English
    - English is often imprecise or ambiguous

- First step is to "formalize" the specification:
    - translate it into math with a precise meaning

- How do we tell if the specification is wrong?
    - specifications can contain bugs
    - we can only **test** our definition on some examples

        (formal) reasoning can only be used *after* we have a formal spec

- Usually best to start by looking at some examples

Definition of Sum of Values in a List

- **Sum of a List: “add up all the values in the list”**

- **Look at some examples…**

L	sum(L)
nil	0
cons(3, nil)	3
cons(2, cons(3, nil))	2+3
cons(1, cons(2, cons(3, nil)))	1+2+3
…	…

Definition of Sum of Values in a List

- **Look at some examples…**

L	sum(L)
nil	0
cons(3, nil)	3
cons(2, cons(3, nil))	2+3
cons(1, cons(2, cons(3, nil)))	1+2+3
…	…

- **Mathematical definition**

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

Sum of Values in a List

- **Mathematical definition of sum**

func sum(nil) := 0
sum(cons(x, S)) := x + sum(S)   for any $x \in \mathbb{Z}$
and any $S \in$ List

- **Translation to TypeScript**

```
const sum = (L: List): bigint => {
  if (L.kind === “nil”) {
    return 0n;
  } else {
    return L.hd + sum(L.tl);
  }
};
```

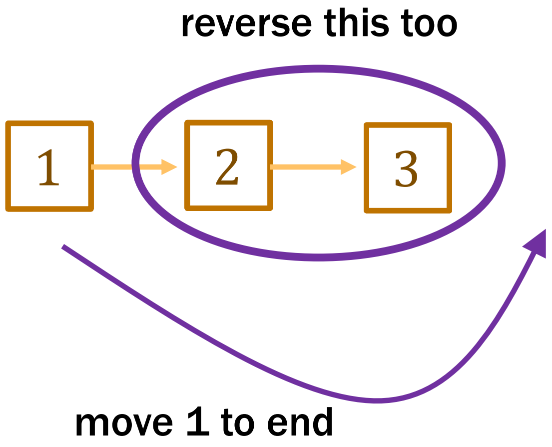
straight from the spec

Definition of Reversal of a List

- Look at some examples...

L	rev(L)
nil	nil
cons(3, nil)	cons(3, nil)
cons(2, cons(3, nil))	cons(3, cons(2, nil))
cons(1, cons(2, cons(3, nil)))	cons(3, cons(2, cons(1, nil)))

- Draw a picture?

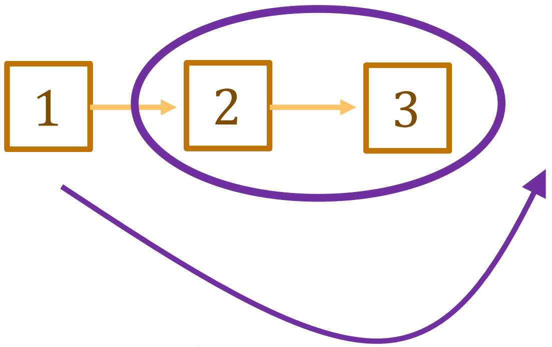


reverse this too

move 1 to end

Reversing A Lists

- Draw a picture?



reverse this too

move 1 to end

- Mathematical definition of rev

func rev(nil) :=

rev(cons(x, S)) := for any $x \in \mathbb{Z}$ and any $S \in \text{List}$

Reversing A Lists

- Mathematical definition of rev

```
func rev(nil)          := nil
     rev(cons(x, S))   := concat(rev(S), cons(x, nil))   for any x ∈ ℤ and
                                                            any S ∈ List
```

- Other definitions are possible, but this is **simplest**

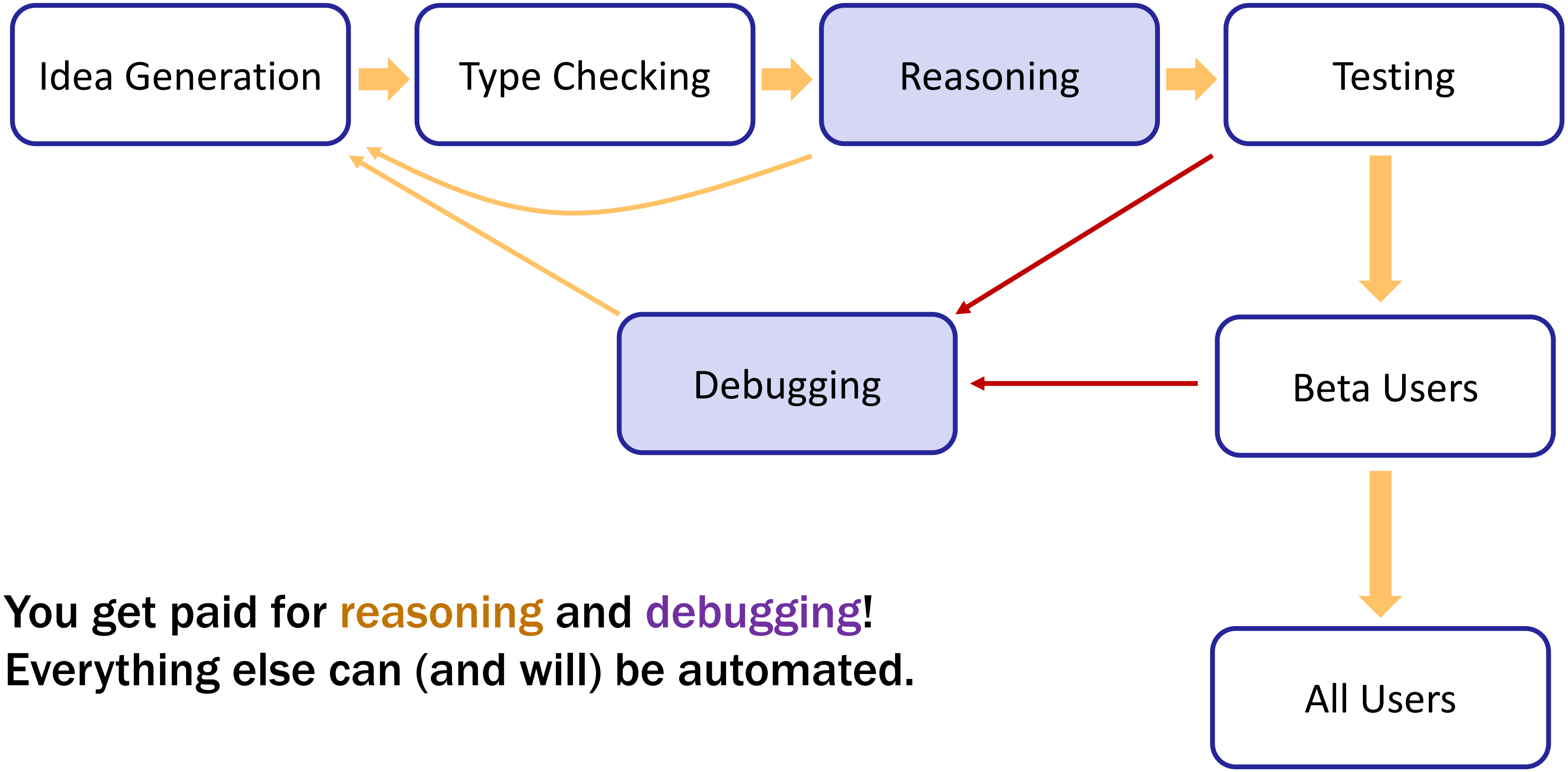
- No help from reasoning tools until later
  - only have testing and thinking about what the English means

- Always make definitions as **simple as possible**

Reasoning

Review: Software Development Process

Given: a problem description (in English)



Idea Generation

Type Checking

Reasoning

Testing

Debugging

Beta Users

All Users

You get paid for reasoning and debugging!
Everything else can (and will) be automated.

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
  - a professional responsibility to know what your code does
  - in practice, "reasoning is not optional:
    either reason up front or debug and then reason"

- Can be done formally or informally
  - most professionals reason informally

    requires years of practice
  - we will teach formal reasoning

    steppingstone to informal reasoning and needed for the **hardest** problems

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 <u>**all**</u> *inputs?*

- **This is a very different question!**

Correctness Levels

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

Facts

- Basic inputs to reasoning are “facts”
  - things we know to be true about the variables
  - typically, “=” or “≤”

```
// n must be a natural number
const f = (n: bigint): bigint => {
  const m = 2n * n;
  return (m + 1n) * (m - 1n);
};
```

find facts by reading along path from top to return statement

- At the return statement, we know these facts:
  - $n \in \mathbb{N}$ (or $n \in \mathbb{Z}$ and $n \geq 0$)
  - $m = 2n$

Facts

- Basic inputs to reasoning are “facts”
  - things we know to be true about the variables
  - typically, “=” or “≤”

```
// n must be a natural number
const f = (n: bigint): bigint => {
  const m = 2n * n;
  return (m + 1n) * (m - 1n);
};
```

- No need to include the fact that n is an integer ($n \in \mathbb{Z}$)
  - that is true, but the type checker takes care of that
  - no need to repeat reasoning done by the type checker

Implications

- We can use the facts we know to prove more facts
  - if we can prove R using facts P and Q,
    we say that R "follows from" or "is implied by" P and Q
  - proving this fact is proving an "implication"

- Proving implications is necessary for checking correctness…

Checking Correctness

- Specifications include two kinds of facts
  - promised facts about the inputs (P and Q)
  - required facts about the outputs (R)

- Checking correctness is just proving implications
  - proving facts about the **return** values

- Two ways reasoning could be required:
  - declarative spec has facts that must hold for the return value
  - *different* imperative spec: must check expressions are "="

Implications

- We can use the facts we know to prove more facts
  - if we can prove R using facts P and Q,
    we say that R "follows from" or "is implied by" P and Q
- Proving implications is the **core step** of reasoning
  - other techniques output implications for us to prove
- The techniques we will learn are
  - proof by calculation
  - proof by cases
  - structural induction } gives us two implications,
    each usually proven by calculation

Proof by Calculation

- Proves an implication
  - fact to be shown is an equation or inequality

- Uses known facts and definitions
  - latter includes, e.g., the fact that $\mathrm{len}(\mathrm{nil}) = 0$

Example Proof by Calculation

- Given $x = y$ and $z \le 10$, prove that $x + z \le y + 10$
  - show the third fact follows from the first two

- Start from the left side of the inequality to be proved

$$x + z \;=\; y + z \;\le\; y + 10$$

since $x = y$

since $z \le 10$

All together, this tells us that $x + z \le y + 10$

Example Proof by Calculation

- Given $x = y$ and $z \leq 10$, prove that $x + z \leq y + 10$
  - show the third fact follows from the first two

- Start from the left side of the inequality to be proved

$$\begin{aligned} x + z &= y + z && \text{since } x = y \\ &\leq y + 10 && \text{since } z \leq 10 \end{aligned}$$

  - easier to read when split across lines
  - “calculation block”, includes explanations in right column

    proof by calculation means using a calculation block
  - “$=$” or “$\leq$” relates that line to the previous line

Calculation Blocks

- Chain of "=" shows first = last

$$\begin{aligned} a &= b && \textbf{since } a = b \\ &= c && \textbf{since } b = c \\ &= d && \textbf{since } c = d \end{aligned}$$

  - proves that $a = d$
  - all 4 of these are the same number

Calculation Blocks

- Chain of “=” and “≤” shows first ≤ last

$$\begin{aligned} x + z &= y + z && \textbf{since } x = y \\ &\leq y + 10 && \textbf{since } z \leq 10 \\ &= y + 3 + 7 && \\ &\leq w + 7 && \textbf{since } y + 3 \leq w \end{aligned}$$

  - each number is equal or strictly larger that previous
  - analogous for “≥”

Using Calculation to Prove Correctness

```
// Inputs x and y are positive integers
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts “$x \geq 1$” and “$y \geq 1$”

- Correct if the return value is a positive integer

$$\begin{aligned} x + y &\geq x + 1 && \text{since } y \geq 1 \\ &\geq 1 + 1 && \text{since } x \geq 1 \\ &= 2 \\ &\geq 1 \end{aligned}$$

  - calculation shows that $x + y \geq 1$

Using Calculation to Prove Correctness

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x \geq 9$" and "$y \geq -8$"

- Correct if the return value is a positive integer

$$x + y$$

Using Calculation to Prove Correctness

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts “$x \geq 9$” and “$y \geq -8$”
- Correct if the return value is a positive integer

$$\begin{aligned} x + y &\geq x + -8 && \text{since } y \geq -8 \\ &\geq 9 - 8 && \text{since } x \geq 9 \\ &= 1 \end{aligned}$$

Using Calculation to Prove Correctness

```
// Inputs x and y are integers with x > 3 and y > 4
// Returns an integer that is 10 or larger.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts “$x \geq 4$” and “$y \geq 5$”

- Correct if the return value is 10 or larger

$$x + y$$

Using Calculation to Prove Correctness

```
// Inputs x and y are integers with x > 3 and y > 4
// Returns an integer that is 10 or larger.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- **Known facts “$x \geq 4$” and “$y \geq 5$”**

- **Correct if the return value is 10 or larger**

$$\begin{aligned} x + y &\geq x + 5 && \text{since } y \geq 5 \\ &\geq 4 + 5 && \text{since } x \geq 4 \\ &= 9 \end{aligned}$$

proof doesn’t work because the **code is wrong**!

Using Calculation to Prove Correctness

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts “$x > 8$” and “$y > -9$”

- Correct if the return value is a positive integer

$$\begin{aligned} x + y &> x + -9 && \text{since } y > -9 \\ &> 8 - 9 && \text{since } x > 8 \\ &= -1 \end{aligned}$$

proof doesn’t work because the **proof is wrong**

warning: avoid using “>” (or “<“) *multiple* times in a calculation block

Using Definitions in Calculations

- Most useful with function calls
  - cite the definition of the function to get the return value

- For example:

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

- Can cite facts such as
  - $\mathrm{sum}(\mathrm{nil}) = 0$
  - $\mathrm{sum}(\mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil}))) = a + \mathrm{sum}(\mathrm{cons}(b, \mathrm{nil}))$

second case of definition with $x = a$ and $L = \mathrm{cons}(b, \mathrm{nil})$

Using Definitions in Calculations

```
func sum(nil)          :=  0
     sum(cons(x, L))   :=  x + sum(L)        for any x ∈ ℤ
                                             and any L ∈ List
```

- **Know** "$a \geq 0$", "$b \geq 0$", **and** "$L = \text{cons}(a, \text{cons}(b, \text{nil}))$"
- **Prove the** "$\text{sum}(L)$" **is non-negative**

$\text{sum}(L)$

Using Definitions in Calculations

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

- **Know** “$a \geq 0$”, “$b \geq 0$”, **and** “$L = \mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil}))$”
- **Prove the** “$\mathrm{sum}(L)$” **is non-negative**

$$
\begin{aligned}
\mathrm{sum}(L) &= \mathrm{sum}(\mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil}))) && \textbf{since } L = \mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil})) \\
&= a + \mathrm{sum}(\mathrm{cons}(b, \mathrm{nil})) && \textbf{def of } \mathrm{sum} \\
&= a + b + \mathrm{sum}(\mathrm{nil}) && \textbf{def of } \mathrm{sum} \\
&= a + b && \textbf{def of } \mathrm{sum} \\
&\geq 0 + b && \textbf{since } a \geq 0 \\
&\geq 0 && \textbf{since } b \geq 0
\end{aligned}
$$

Proof by Calculation

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

Finding Facts at a Return Statement

- Consider this code

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

find facts by reading along path
from top to return statement

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

Proving Correctness with Conditionals

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

- 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: bigint, y, bigint): bigint => {
  if (y < 0n) {
    return x + y;
  } else {
    return x - 1n;
  }
};
```

- 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: bigint, y, bigint): bigint => {
  if (y < 0n) {
    return x + y;
  } else {
    return x - 1n;
  }
};
```

- 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: bigint, y, bigint): bigint => {
  if (y < 0n) {
    return x + y;
  } else {
    return x - 1n;
  }
};
```

- 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: bigint, y, bigint): bigint => {
  if (y < 0n) {
    return x + y;
  } else {
    return x - 1n;
  }
};
```

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

find facts by reading along <u>path</u>
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 each 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: bigint, y, bigint): bigint => { .. };
```

  - 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: bigint, b, bigint): bigint => {
  if (a >= b) {
    return a;
  } else {
    return b;
  }
};
```

straight from the spec
(imperative spec)

Example Correctness with Conditionals

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

<u>**not**</u> straight from the spec
(declarative spec)

- 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: bigint, b, bigint): bigint => {
  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: bigint, b, bigint): bigint => {
  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

```
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)

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

```
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**

$$
\begin{aligned}
\mathrm{sum}(L) &= \mathrm{sum}(\mathrm{cons}(a, \mathrm{cons}(b, \mathrm{nil}))) && \textbf{since } L = \ldots \\
&= a + \mathrm{sum}(\mathrm{cons}(b, \mathrm{nil})) && \textbf{def of } \mathrm{sum} \\
&= a + b + \mathrm{sum}(\mathrm{nil}) && \textbf{def of } \mathrm{sum} \\
&= a + b && \textbf{def of } \mathrm{sum}
\end{aligned}
$$

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

Sum of a List

```
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**

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

- **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
  - $\text{sum}(\text{cons}(3, \text{cons}(5, \text{nil}))) = 8$
  - $\text{sum}(\text{cons}(\text{-}5, \text{cons}(2, \text{nil}))) = \text{-}3$
  - …
  - 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

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

- 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

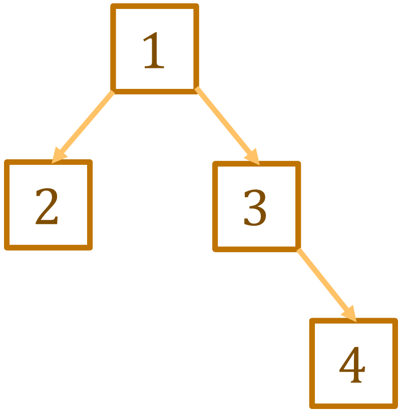
Binary Trees

Binary Trees

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

- **Inductive definition of binary trees of integers**

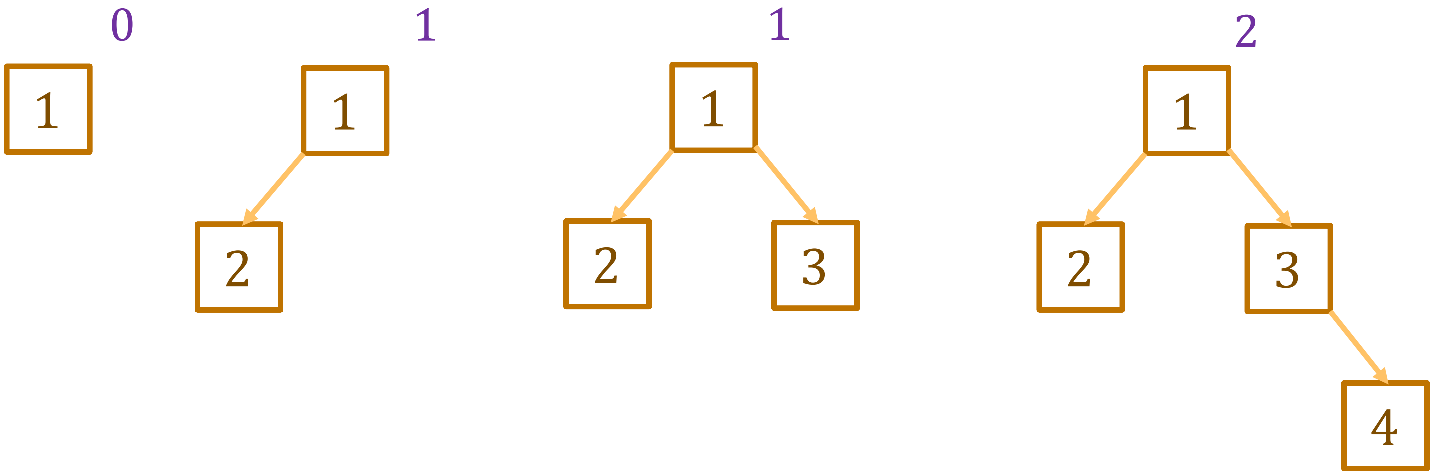
node(1, node(2, empty, empty), node(3, empty, node(4, empty, empty)))



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

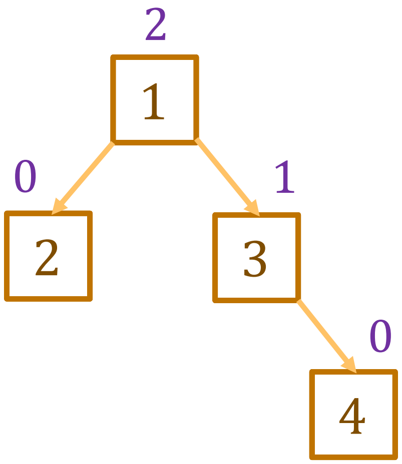
type Tree := empty | node(x: $\mathbb{Z}$, L: Tree, R: Tree)

- **Mathematical definition of height**

func height(empty) :=

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

for any x ∈ $\mathbb{Z}$ and any L, R ∈ Tree

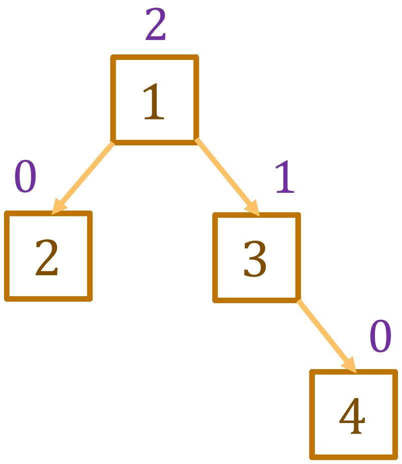


Height of a Tree

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 $L, R \in \text{Tree}$
```

- **Suppose** “$\mathrm{T} = \mathrm{node}(1, \mathrm{empty}, \mathrm{node}(2, \mathrm{empty}, \mathrm{empty}))$”

- **Prove that** $\mathrm{height}(\mathrm{T}) = 1$

height(T)

Using Definitions in Calculations

func height(empty) := −1
height(node(x, L, R)) := 1 + max(height(L), height(R))
for any x ∈ 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, height(empty)))	**def of** height
	= 1 + max(-1, 1+ max(-1, -1))	**def of** height
	= 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

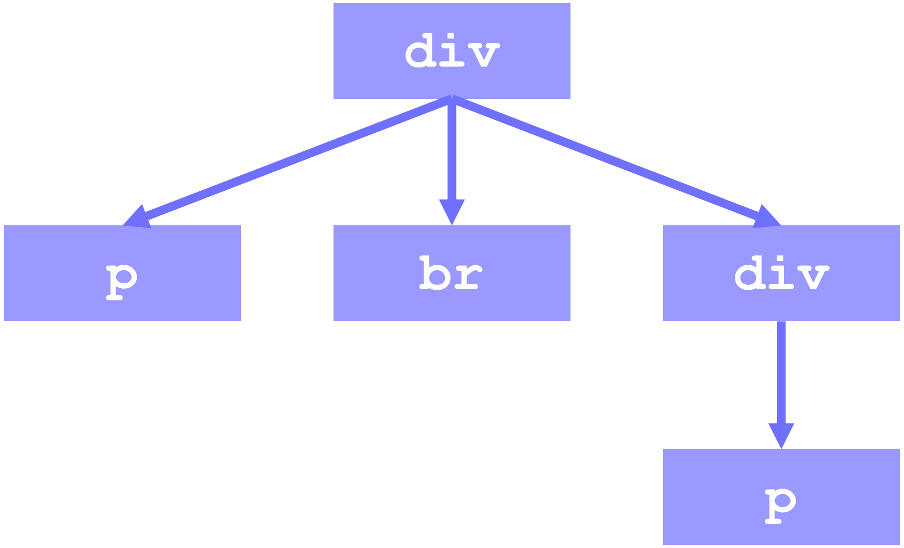
- These come up all the time…
  - no constructors with recursive arguments = “generalized enums”
  - constructor with 1 recursive arguments = “generalized lists”
  - constructor with 2+ recursive arguments = “generalized trees”

- 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 for Modularity

- 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 for Modularity

- 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 for Modularity

```
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 for Modularity

- 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 for Modularity

- **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**!