

CSE 331

Summer 2025

Specifications

Jaela Field

<u>DIVISION NOTATION</u>	
$\left. \begin{array}{r} A \div B \\ B \overline{)A} \end{array} \right\}$	SCHOOLCHILD
A/B	SOFTWARE ENGINEER
$\frac{A}{B}$	NORMAL PERSON OR UNICODE ENTHUSIAST
$\frac{A}{B}$	SCIENTIST
AB^{-1}	FANCY SCIENTIST
$F(A,B)$ SUCH THAT $F(B) = A$	OH NO, RUN

xkcd #2687, ty Matt

Where We Are in the Course

Eight assignments split into these groups:

HW1

HW2

HW3

learn to write more complex apps
practice debugging

HW4

HW5

HW6

learn how to be 100% sure the code is correct
(most of the work done on *paper*)

HW7

HW8

Correctness and Specifications

- Correctness requires a definition of the correct answer
- Description must be precise
 - can't have disagreement about what is correct
- Informal descriptions (English) are usually imprecise
 - necessary to “formalize” the English
 - turn the English into a precise *mathematical* definition
 - professionals are very good at this
 - usually just give English definitions
 - important skill to practice
 - we will start out completely formal to make it easier

Kinds of Specifications

- **Imperative** specification says how to calculate the answer
 - lays out the exact steps to perform to get the answer
- **Declarative** specification says what the answer looks like
 - does not say how to calculate it
 - up to us to ensure that our code satisfies the spec
- Can implement a *different* imperative specification
 - again, up to us to ensure that our code satisfies the spec

Example: Imperative Specification (abs)

- Absolute value: $|x| = x$ if $x \geq 0$ and $-x$ otherwise
 - definition is an “if” statement

```
const abs = (x: bigint) : bigint => {  
  if (x >= 0n) {  
    return x;  
  } else {  
    return -x;  
  }  
}
```

just translating math to TypeScript

Example: Declarative Specification (sub)

- Subtraction ($a - b$): return x such that $b + x = a$
 - can see that $b + (a - b) = b + a - b = a$

```
const sub = (a : bigint, b: bigint) : bigint => {  
  
    ??  
  
}
```

we are left to figure out how to do this...
and convince ourselves it satisfies the spec

Example: Declarative Specification (sqrt)

- Square root of x is number y such that $y^2 = x$
 - not all positive integers have integer square roots, so... let's round up
 - $(y - 1)^2 \leq x \leq y^2$
smallest integer y such that $x \leq y^2$

```
const sqrt = (x: bigint) : bigint => {
```

```
  ??
```

```
}
```

we are left to figure out how to do this...
and convince ourselves it satisfies the spec

Example: Declarative Specification (abs)

- Absolute value $|x|$ is an integer y such that
 - $y \geq x$
 - $y \geq -x$
 - $y = x$ or $y = -x$

```
const abs = (x: bigint) : bigint => {  
  if (x >= 0) {  
    return x;  
  } else {  
    return -x;  
  }  
}
```

requires some thinking to make sure this code
returns a number with the properties above

"Straight From the Spec"

- If imperative, just translate math into code
 - TypeScript here, but could also be Java
 - we often call this "straight from the spec"
- if declarative (or implementing different imperative spec), then we will need new tools for checking its correctness

Examples from Java: Map .replace()

`java.util.Map` — set of (key, value) pairs

replace

```
default V replace(K key,  
                  V value)
```

Replaces the entry for the specified key only if it is currently mapped to some value.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
if (map.containsKey(key)) {  
    return map.put(key, value);  
} else  
    return null;
```

[sli.do #cse331](https://sli.do/#cse331)



Imperative

Examples from Java: Map .putAll()

`java.util.Map` — set of (key, value) pairs

putAll

```
void putAll(Map<? extends K,? extends V> m)
```

Copies all of the mappings from the specified map to this map (optional operation). The effect of this call is equivalent to that of calling `put(k, v)` on this map once for each mapping from key `k` to value `v` in the specified map. The behavior of this operation is undefined if the specified map is modified while the operation is in progress.

Parameters:

`m` - mappings to be stored in this map

Imperative

Examples from Java: Map .containsKey()

`java.util.Map` — set of (key, value) pairs

containsKey

`boolean containsKey(Object key)`

Returns true if this map contains a mapping for the specified key. More formally, returns true if and only if this map contains a mapping for a key `k` such that `Objects.equals(key, k)`. (There can be at most one such mapping.)

Parameters:

`key` - key whose presence in this map is to be tested

Returns:

true if this map contains a mapping for the specified key

Declarative (probably)

Examples from Java: Object .hashCode()

`java.util.Object`

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by [HashMap](#).

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

This Topic's Goal: Imperative Specifications

- Toolkit for writing **imperative** specifications
 - define math for data and code
 - write specifications that are language independent
 - (don't want a toolkit that only works for TypeScript)
 - describe how to translate imperative specs into TypeScript
 - try to make the translations as *straightforward* as possible (fewer mistakes)
 - mention new TypeScript features when related
 - critical to understand what bugs the type system catches and which it does not
- Will look at **declarative** specifications later

Math Notation

Basic Data Types in Math

- In math, the basic data types are “sets”
 - sets are collections of objects called **elements**
 - write $x \in S$ to say that “x” is an element of set “S”, and $x \notin S$ to say that it is not.

- Examples:

$x \in \mathbb{Z}$

x is an integer

$x \in \mathbb{N}$

x is a non-negative integer (natural)

$x \in \mathbb{R}$

x is a real number

$x \in \mathbb{B}$

x is T or F (boolean)

$x \in \mathbb{S}$

x is a character

$x \in \mathbb{S}^*$

x is a string

} non-standard names

Basic Data Types in TypeScript

English	Math	TypeScript	Up to Us
integer	$x \in \mathbb{Z}$	bigint	
natural	$x \in \mathbb{N}$	bigint	non-negative
real	$x \in \mathbb{R}$	number	
boolean	$x \in \mathbb{B}$	boolean	
character	$x \in \mathbb{S}$	string	length 1
string	$x \in \mathbb{S}^*$	string	

we will often write
 $x : \mathbb{Z}$ instead of $x \in \mathbb{Z}$

- only subtraction on non-negative can produce negative

CSE 331 Summer 2025

Specifications & Inductive Data Types

Jaela Field

Summary of Last Time: Specifications

- Specification is necessary to even discuss correctness
- **Goals:** develop a toolkit for
 - writing (imperative) specs that are fully precise
 - writing specs in a language-independent manner
- **Solution:** rely on standard mathematical notation

Creating New Types in Math (Unions)

- **Union Types** $S^* \cup \mathbb{N}$
 - contains every object in either (or both) of those sets
 - e.g., all strings and natural numbers
- If $x \in \mathbb{N} \cup S^*$, then x could be a natural or string
- **Two sets can contain common elements**
 - \mathbb{N} and S^* do not contain common elements, they are “disjoint”

Creating New Types in TypeScript (Unions)

- **Union Types** `string | bigint`

- can be either one of these

- How do we work with this code?

```
const x: string | bigint = ...;
```

```
// can I call isPrime(x)?
```

- We can check the type of `x` using “`typeof`”
 - TypeScript understands these expressions
 - will “`narrow`” the type of `x` to reflect that information

Recall: Type Narrowing

```
const x: string | bigint = ...;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  ...                        // x is a string
}
```

- TypeScript knows `x` is a `bigint` inside the “if” (narrowing)
 - This does not require a **type cast**
 - **No type casts** in 331!
 - Unlike Java, TypeScript takes your word for it (no runtime checks)
 - = painful debugging

Type Narrowing Gotcha

```
const f = (x: bigint): string | bigint => ...;

if (typeof f(x) === "bigint") {
  console.log(isPrime(f(x)))    // why not allowed?
}
```

- TypeScript will (properly) reject this
 - no guarantee that `f(x)` returns the same value both times!

Type Narrowing of Function Calls

```
const f = (x: bigint): string | bigint => ...;

const y = f(x);
if (typeof y === "bigint") {
  console.log(isPrime(y))           // this works now
}
```

- TypeScript can see that the two values are the same
- Functions that return different values for the same inputs are confusing!
 - maybe better to avoid that

Record Types in Math

- **Record Types** $\{x : \mathbb{N}, y : \mathbb{N}\}$
 - record with fields “x” and “y” each containing a number
e.g., $\{x: 3, y: 5\}$
 - To get field values, use “.” notation
e.g., $r := \{x: 3, y: 5\}$, $r.x$ would get 3
- **Note that $\{x: 3, y: 5\} = \{y: 5, x: 3\}$ in math**
 - field names matter, not order
 - note that these are not “===” in JavaScript
 - in math, “=” means same values
 - in JavaScript, “===” is reference equality

Record Types in TypeScript

- **Record Types** `{x: bigint, y: bigint}`
 - anything with *at least* fields “x” and “y”
- Retrieve a part by name:

```
const t: {x: bigint, y: bigint} = ... ;  
console.log(t.x);
```

Tuple Types in Math

- **Record Types** $\{x : \mathbb{N}, y : \mathbb{N}\}$
 - record with fields “x” and “y” each containing a number
e.g., $\{x: 3, y: 5\}$
- **Tuple Types** $\mathbb{N} \times \mathbb{N}$
 - pair of two natural numbers, e.g., $(5, 7)$
 - can do tuples of 3, 4, or more elements also
- **Mostly equivalent alternatives**
 - both let us put parts together into a larger object
 - record distinguishes parts by name
 - tuple distinguishes parts by order

Retrieving Part of a Tuple

- To refer to tuple parts, we must give them names

- Tuple Types $\mathbb{N} \times \mathbb{N}$

Let $(a, b) := t$.

Suppose we know that $t = (5, 7)$

“:=” means a definition

Then, we have $a = 5$ and $b = 7$

- Tuple Types `[bigint, bigint]`

```
const t: [bigint, bigint] = ...;  
const [a, b] = t;  
console.log(a); // first part of t
```

Simple Functions in Math

- Simplest function definitions are single expressions
- Will write them in math like this:

$\text{double} : \mathbb{N} \rightarrow \mathbb{N}$

$\text{double}(n) := 2n$

- first line declares the type of double function
takes a natural number input to a natural number output
- second line shows the calculation
know that "n" is a natural number from the *first* line
- will often put the type in the text before the definition, e.g.,

The function $\text{double} : \mathbb{N} \rightarrow \mathbb{N}$ is defined by...

$\text{double}(n) := 2n$

Functions in TypeScript

- There are functions in TypeScript too!

```
// Implements double : N -> N
const double = (x: bigint): bigint => {
    return 2n * x;
}
```

- Recall, the type of double in TS is

```
(x: bigint): bigint
```

Simple Functions in Math (and shorthands)

- Another example:

$$\text{dist} : \{x: \mathbb{R}, y: \mathbb{R}\} \rightarrow \mathbb{R}$$

$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

- first line tells us that "p" is a record and "p.x" is a real number

- Can define short-hand for types in math also

$$\text{type Point} := \{x: \mathbb{R}, y: \mathbb{R}\}$$

$$\text{dist} : \text{Point} \rightarrow \mathbb{R}$$

$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

Complex Functions in Math

- Most interesting functions are not simple expressions
 - need to use different expressions in different cases
- Can use side-conditions to split into cases

$\text{abs} : \mathbb{R} \rightarrow \mathbb{R}$

$\text{abs}(x) := x$ if $x \geq 0$

$\text{abs}(x) := -x$ if $x < 0$

- conditions must be exclusive and exhaustive
 - we do not want to require on *order* to determine which applies
- there is a **better** way to do this in many cases...

Pattern Matching

- Can also define functions by “pattern matching”

$\text{double} : \mathbb{N} \rightarrow \mathbb{N}$

$\text{double}(0) := 0$

$\text{double}(n+1) := \text{double}(n) + 2$

- first case matches only 0
 - second case matches numbers 1 more than some $n : \mathbb{N}$...
 - $\text{double}(6) = \text{double}(5+1)$ so it matches with $n = 5$
 - since $n \geq 0$, we have $n+1 \geq 1$, so it matches 1, 2, 3, ...
 - pattern “ $n+2$ ” would match 2, 3, 4, ...
- Simplifies the math in multiple ways...

Pattern Matching on Natural Numbers

- Pattern matching definition

$$\begin{aligned}\text{double}(0) &:= 0 \\ \text{double}(n+1) &:= \text{double}(n) + 2\end{aligned}$$

is simpler than using side conditions

$$\begin{aligned}\text{double}(n) &:= 0 && \text{if } n = 0 \\ \text{double}(n) &:= \text{double}(n-1) + 2 && \text{if } n > 0\end{aligned}$$

- e.g., need to explain why $\text{double}(n-1)$ is legal
easy in this case, but it gets harder

- We will prefer pattern matching **whenever possible**

Pattern Matching on Booleans

- Booleans have only two legal values: T and F
- Can pattern match just by listing the values:
 - the function $\text{not} : \mathbb{B} \rightarrow \mathbb{B}$ is defined as follows:

$\text{not}(T) := F$

$\text{not}(F) := T$

- negates a boolean value
- no simpler way to define this function!

Pattern Matching on Records

- Can pattern match on individual fields of a record

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

change : Steps $\rightarrow \mathbb{Z}$

change({n: m, fwd: T}) := m

change({n: m, fwd: F}) := -m

- clear that the rules are exclusive and exhaustive

- Can match on multiple parameters
 - e.g., change({n: m+5, fwd: T}) := 2m
 - just make sure the rules are exclusive and exhaustive

Pattern Matching in TypeScript

- TypeScript does not provide pattern matching
 - some other languages do! (see 341)
- We must translate into “`if`”s on our own

```
type Steps = {n: number, fwd: boolean};
```

```
const change = (s: Steps) => {  
  if (s.fwd) {  
    return s.n;  
  } else {  
    return -s.n;  
  }  
};
```

still straight from the spec
but easy to make mistakes

Pattern Matching in TypeScript: Gotcha

$\text{double}(0) \quad := 0$
 $\text{double}(n+1) := \text{double}(n) + 2$

- Also need to be careful with natural numbers

```
// m is non-negative
const double = (m: bigint) => {
  if (m === 0n) {
    return 0n;
  } else {
    return double(m - 1n) + 2n;
  }
};
```

spec says $\text{double}(n)$
but code says $\text{double}(m - 1)$

- pattern matching uses “ $n+1$ ” but the code uses “ m ” (or “ n ”)
sadly, TypeScript will not let “ $n+1$ ” be the argument value

Code Without Mutation

- **Saw all types of code without mutation:**
 - straight-line code
 - conditionals
 - recursion
- **This is all that there is!**
 - can write anything computable with just these
- **Saw TypeScript syntax for these already...**

Code Without Mutation Example

Example function with all three types

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

What does this compute?

$$f(m) = 2^m$$

$$f: \mathbb{N} \rightarrow \mathbb{N}$$

$$f(0) := 1$$

$$f(n+1) := 2 \cdot f(n)$$

Concept Check – Partner Practice

How do we declare the type of this function in math?

```
// m must contain non-negative integer
const sometimesSquare =
  (m: [bigint, boolean]) : bigint => {
```

How does the body of the function translate into math?

```
    const [n, b] = m;
    if (!b) {
      return n;
    } else {
      return n ** 2n;
    }
  }
```

Concept Check - Solutions

```
// m must contain non-negative integer
const sometimesSquare =
  (m: [bigint, boolean]) : bigint => {
    const [n, b] = m;
    if (!b) {
      return n;
    } else {
      return n ** 2n;
    }
  }
}
```

sometimesSquare: $\mathbb{N} \times \mathbb{B} \rightarrow \mathbb{N}$

sometimesSquare((n, F)) := n
sometimesSquare((n, T)) := n²

JSDocs with math definitions

```
/**
 * Finds square of int sometimes
 * @param m, tuple containing non-negative int and a bool
 *       to indicate if int should be squared
 * @returns sometimesSquare(m)
 */
const sometimesSquare =
  (m: [bigint, boolean]) : bigint => {
    const [n, b] = m;
    if (!b) {
      return n;
    } else {
      return n ** 2n;
    }
  }
}
```

Inductive Data Types

Inductive Data Types: Recursion in Data

- **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 (math) records cannot
- **Inductive data types are defined recursively**
 - **combine union with recursion**

Inductive Data Types: Constructors

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

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

- second constructor is recursive
- can have any number of arguments (even none)
 - will leave off the parentheses when there are none

- Examples of elements

$C(1)$

$D(2, C(1))$

$D(3, D(2, C(1)))$

in math, these are not function calls

Inductive Data Types: Equality

- Each element is a description of how it was made

$C(1)$

$D(2, C(1))$

$D(3, D(2, C(1)))$

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

Natural Numbers

`type N := zero | succ(n : N)`

- Inductive definition of the natural numbers

<code>zero</code>	<code>0</code>
<code>succ(zero)</code>	<code>1</code>
<code>succ(succ(zero))</code>	<code>2</code>
<code>succ(succ(succ(zero)))</code>	<code>3</code>

The most basic set we have is defined inductively!
(interested? see “Peano axioms”)

Even Natural Numbers

`type \mathbb{E} := zero | two-more(n : \mathbb{E})`

- Inductive definition of the even natural numbers

<code>zero</code>	<code>0</code>
<code>two-more(zero)</code>	<code>2</code>
<code>two-more(two-more(zero))</code>	<code>4</code>
<code>two-more(two-more(two-more(zero)))</code>	<code>6</code>

much better notation

Lists

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

- Inductive definition of lists of integers

nil

cons(3, nil)

cons(2, cons(3, nil))

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

Our most important data type!



Shorthand Notation for Lists

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

- **We will use:**
 - "x :: L" to mean "cons(x, L)"
 - "[1, 2, 3]" to mean "1 :: 2 :: 3 :: nil"

- **Examples:**

<code>nil</code>	<code>nil</code>	<code>[]</code>
<code>cons(3, nil)</code>	<code>3 :: nil</code>	<code>[3]</code>
<code>cons(2, cons(3, nil))</code>	<code>2 :: 3 :: nil</code>	<code>[2, 3]</code>
<code>cons(1, cons(2, cons(3, nil)))</code>	<code>1 :: 2 :: 3 :: nil</code>	<code>[1, 2, 3]</code>

Practice:

Create a list with a 9 at the beginning of the list, a 3 in the middle, and a 7 at the end.

`cons(9, cons(3, cons(7, nil)))`

`9::3::7::nil`

To create this , how many times do we utilize the list constructor?

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

4

Inductive Data Types in TypeScript

- TypeScript does not natively support inductive types
 - some “functional” languages do (e.g., OCaml and Haskell)
- We must think of a way to cobble them together...
 - our answer is a **design pattern**...

Design Patterns

- Popularized in 1994 book of that name
 - written by the “Gang of Four”
Gamma, Helm, Johnson, Vlissides
 - worked in C++ and SmallTalk
(SmallTalk hugely influenced OOP in Java, etc.)
- 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 not easy 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: number};
type T2 = {kind: "T2", a: string, b: bigint};

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

Inductive Data Type Design Pattern (1/2)

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

- Implement in TypeScript as

```
type T = {kind: "C", x: bigint}  
        | {kind: "D", x: string, t: T};
```


Inductive Data Type Design Pattern (2/2)

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

- Implement in TypeScript as

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

Inductive Lists 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};
```

Inductive Lists in TypeScript: empty?

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

- How do I check if my list is empty?

```
if (mylist.kind === "nil") {  
  ...  
}
```

- Importantly, *not*:

```
if (mylist === null) {  
  ...  
}
```

- (very different from Java!)

Defining Inductive Lists

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



```
const list: List = {  
  kind: "cons",  
  hd: 1n,  
  tl: {  
    kind: "cons",  
    hd: 2n,  
    tl: {  
      kind: "cons",  
      hd: 3n,  
      tl: { kind: "nil" }  
    }  
  }  
}
```

Correct but clunky.
Like math: can we define easier notation?

Inductive List Helpers (1/2)

- Make this look more like math notation...

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

```
const nil: Readonly<List> = {kind: "nil"};
```

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

- use **only** these two functions to create `Lists`
do not create the records directly
- note that we only have one instance of `nil`
this is called a “singleton” (there is a **design pattern** for ensuring this)

Inductive List Helpers (2/2)

- Make this look more like math notation...

```
const nil: Readonly<List> = {kind: "nil"};
```

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

- Can now write code like this:

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

Inductive List Helpers: Equality Gotcha

- Make this look more like math notation...

```
const nil: Readonly<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
will see this later...

CSE 331

Summer 2025

Inductive Data Types & Testing

Jaela Field

Unit Tests: Keep 'em Simple



©2018 Blake O'Hare <http://linkedlistcomic.com/29>

Recall: Inductive Lists

- Defined in math as

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

- Implemented in TypeScript as

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

- For convenience and to look more like our math, we define `nil` “singleton” and `cons` function

```
const list: List = cons(1, cons(2, cons(3, nil)))
```

Functions Defined on Inductive Data Types

- We need recursion to define interesting functions
- Inductive types fit esp. well with *pattern matching*
 - every object is created using some **constructor**
 - **match** based on which constructor was used

Length of a List (in Math)

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

- **Mathematical definition of list length:**

$\text{len} : \text{List} \rightarrow \mathbb{N}$

$\text{len}(\text{nil}) \quad := 0$

$\text{len}(x :: L) \quad := 1 + \text{len}(L)$

- any list is either nil or $x :: L$ for some x and L
- thus, cases are exclusive and exhaustive

Length of a List (translated to TypeScript)

- Mathematical definition of length

$$\begin{aligned}\text{len}(\text{nil}) &:= 0 \\ \text{len}(x :: L) &:= 1 + \text{len}(L)\end{aligned}$$

- Translation to TypeScript

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

TypeScript will see that this is valid
since `S.kind !== "nil"`

Swapping Elements in a List (in Math)

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

- **Function that swaps adjacent elements in a list:**

swap : List \rightarrow List

swap(nil) := nil

swap(x :: nil) := x :: nil

swap(x :: y :: L) := y :: x :: swap(L)

- any list is either nil or x :: nil or x :: y :: L for some x, y, and L
- thus, cases are exclusive and exhaustive

Swapping Elements in a List (in TypeScript)

```
swap(nil)      := nil
swap(x :: nil) := x :: nil
swap(x :: y :: L) := y :: x :: swap(L)
```

- Translation to TypeScript

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

TypeScript will see that these are valid since
 $S.kind \neq \text{"nil"}$ and $S.tl.kind \neq \text{"nil"}$

Structural Recursion

- Examples only recurse on *parts* of the input

$\text{len}(x :: L) := 1 + \text{len}(L)$

- call on $x :: L$ recurses on L

$\text{swap}(x :: y :: L) := y :: x :: \text{swap}(L)$

- call on $x :: y :: L$ recurses on L
 - such cases are called "structural recursion"
- Guarantees no infinite recursion!
 - one argument gets *strictly smaller* on each call
 - restrict ourselves to structural recursion in math and TS

Formalizing Specifications

Formalizing a Specification

- Sometimes the instructions are written in English
 - English is often imprecise or ambiguous
- First step then is to “formalize” the specification:
 - translate it into math with a precise meaning
- Best to start by looking at some examples
 - try to spot a pattern
 - recursive data usually involves recursive functions
 - each base case *usually* corresponds to a pattern-match
 - can have one or more base cases

Formalizing Sum of a List

- Sum of a List: “add up all the values in the list”
- Look at some examples...

L	sum(L)
nil	0
3 :: nil	3
2 :: 3 :: nil	2+3
1 :: 2 :: 3 :: nil	1+2+3
...	...

Formalizing Sum of a List: the Math

- Look at some examples...

L	sum(L)
nil	0
3 :: nil	3
2 :: 3 :: nil	2+3
1 :: 2 :: 3 :: nil	1+2+3
...	...

- Mathematical definition of sum:

$\text{sum}: \text{List} \rightarrow \mathbb{N}$

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

$\text{sum}(x :: L) \quad :=$

Formalizing Sum of a List: Trying Examples

L

1 :: 2 :: 3 :: nil

sum(L)

1+2+3

- **Mathematical definition of sum: List $\rightarrow \mathbb{N}$:**

sum(nil) := 0

sum(x :: L) := x + sum(L)

- **Check that this works on the examples...**

sum(1 :: 2 :: 3 :: nil)

= 1 + sum(2 :: 3 :: nil)

= 1 + 2 + sum(3 :: nil)

= 1 + 2 + 3 + sum(nil)

= 1 + 2 + 3

def of sum (2nd line)

def of sum (2nd line)

def of sum (2nd line)

def of sum (1st line)

Converting Sum of a List to TypeScript

- Mathematical definition of sum

$$\begin{aligned}\text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L)\end{aligned}$$

- Translation to TypeScript

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

Formalizing List Equality

- Equal lists: “built with same steps”
- Look at some examples...

L	R	equal(L, R)
nil	nil	
nil	1 :: nil	
1 :: nil	nil	
1 :: nil	1 :: nil	
2 :: nil	3 :: nil	
1 :: 2 :: nil	1 :: 3 :: nil	

Formalizing List Equality: the Math

L	R	equal(L, R)
nil	nil	T
nil	1 :: nil	F
1 :: nil	nil	F
1 :: nil	1 :: nil	T
2 :: nil	3 :: nil	F
1 :: 2 :: nil	1 :: 3 :: nil	F

- **Mathematical definition of $\text{equal} : (\text{List}, \text{List}) \rightarrow \mathbb{B}$**

$\text{equal}(\text{nil}, \text{nil}) \quad := \text{T}$
 $\text{equal}(\text{nil}, y :: R) \quad := \text{F}$
 $\text{equal}(x :: L, \text{nil}) \quad := \text{F}$
 $\text{equal}(x :: L, y :: R) \quad := (x = y) \text{ and } \text{equal}(L, R)$

Formalizing List Equality: Trying Examples

L	R	equal(L, R)
$1 :: 2 :: \text{nil}$	$1 :: 3 :: \text{nil}$	F

- **Mathematical definition of $\text{equal} : (\text{List}, \text{List}) \rightarrow \mathbb{B}$**

$\text{equal}(\text{nil}, \text{nil})$	$:= \text{T}$
$\text{equal}(\text{nil}, y :: R)$	$:= \text{F}$
$\text{equal}(x :: L, \text{nil})$	$:= \text{F}$
$\text{equal}(x :: L, y :: R)$	$:= (x = y) \text{ and } \text{equal}(L, R)$

- **Check that this works on the examples...**

$\text{equal}(1 :: 2 :: \text{nil}, 1 :: 3 :: \text{nil})$	
$= (1 = 2) \text{ and } \text{equal}(2 :: \text{nil}, 3 :: \text{nil})$	def of equal (4 th line)
$= (1 = 2) \text{ and } (2 = 3) \text{ and } \text{equal}(\text{nil}, \text{nil})$	def of equal (4 th line)
$= \text{T and F and T}$	def of equal (1 st line)

Converting List Equality to TypeScript

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

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

note: can combine some cases!
but this is easier to read

Formalizing List Concatenation

- Concatenate L and R: “a single list containing the elements of L followed by the elements of R”
- Look at some examples...

L	R	concat(L, R)
nil	nil	nil
nil	3 :: 4 :: nil	3 :: 4 :: nil
2 :: nil	3 :: 4 :: nil	2 :: 3 :: 4 :: nil
1 :: 2 :: nil	3 :: 4 :: nil	1 :: 2 :: 3 :: 4 :: nil
...		

Formalizing List Concatenation: the Math

L	R	concat(L, R)
nil	nil	nil
nil	3 :: 4 :: nil	3 :: 4 :: nil
2 :: nil	3 :: 4 :: nil	2 :: 3 :: 4 :: nil
1 :: 2 :: nil	3 :: 4 :: nil	1 :: 2 :: 3 :: 4 :: nil

- **Mathematical definition of $\text{concat} : (\text{List}, \text{List}) \rightarrow \text{List}$**

$\text{concat}(\text{nil}, R) \quad :=$

$\text{concat}(x :: L, R) \quad :=$

Formalizing List Concatenation: Examples

$1 :: 2 :: \text{nil}$

$3 :: 4 :: \text{nil}$

$1 :: 2 :: 3 :: 4 :: \text{nil}$

- **Mathematical definition of $\text{concat} : (\text{List}, \text{List}) \rightarrow \text{List}$**

$\text{concat}(\text{nil}, R) \quad := R$

$\text{concat}(x :: L, R) \quad := x :: \text{concat}(L, R)$

- **Check that this matches examples...**

$\text{concat}(1 :: 2 :: \text{nil}, 3 :: 4 :: \text{nil})$

$= 1 :: \text{concat}(2 :: \text{nil}, 3 :: 4 :: \text{nil})$

$= 1 :: 2 :: \text{concat}(\text{nil}, 3 :: 4 :: \text{nil})$

$= 1 :: 2 :: 3 :: 4 :: \text{nil}$

def of concat (2nd line)

def of concat (2nd line)

def of concat (1st line)

Converting List Concatenation to TypeScript

- Mathematical definition of $\text{concat} : (\text{List}, \text{List}) \rightarrow \text{List}$

$$\begin{aligned}\text{concat}(\text{nil}, R) &:= R \\ \text{concat}(x :: L, R) &:= x :: \text{concat}(L, R)\end{aligned}$$

- Translation to TypeScript

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

Generic List

- “generic” list type takes a “type parameter”

```
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 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

Formalizing a Specification

- We “formalize” English instructions into precise specifications to define intended behavior
- How do we tell if the specification is wrong?
 - specifications can contain bugs!
Is it obvious that equal & concat are correct? Maybe not.
- We **tested** our definition on *a few* examples
 - what can we do to increase the odds we spot bugs?

Testing

Unit, Integration, and End-to-End Tests (1/2)

- A unit test checks one function
 - ideally, without testing anything else (not always possible)
- An integration test makes sure units work together
 - many (most?) bugs in practice are here
- An end-to-end test exercises almost all the code
- How are we testing in HW3?
 - we are doing end-to-end testing
 - this makes debugging harder! (more to search)

Unit, Integration, and End-to-End Tests (2/2)

- A unit test checks one function
- An integration test makes sure units work together
- An end-to-end test exercises almost all the code
- You will be expected to write unit tests in industry
- There will also be integration and end-to-end tests
 - someone will write them, but maybe not you
 - (requires **understanding** the whole system)
- We will focus on unit testing

Unit Testing: Dijkstra's

- Even individual functions might be too big...

Dijkstra's Algorithm

The pseudocode below assumes we have the following data structures: **a lot of code here...**

adjacent A *map* from an (x, y) location to the list of all edges that start at that location. These give us all the locations you can get to from that location in one step.

finished A *set* of (x, y) locations for which we have already found the shortest path. The algorithm will avoid considering new paths to these locations. **more in here...**

active A (*priority*) *queue* containing all paths to locations that are one step from a finished node. The key idea of the algorithm is that the shortest path in the queue to a non-finished node must be the shortest path to that node.²

With those data structures in hand, Dijkstra's algorithm proceeds as follows:

add a 0-step (empty) path from start to itself to active

while active is not empty:

Unit Testing via Small Functions

- Even individual functions might be too big...
 - split out pieces into their own functions
- Our coding conventions help enforce this

```
<input type="text" onChange={ (evt) => {  
    ... multi-line calculation ...  
}}></input>
```

- make this its own function
 - makes the code more **understandable** and **testable**
- Purposefully design the code to be testable
 - important part of programming in practice

“Manual” vs Programmatic Tests

- Usually possible to run the code by hand (“manually”)
 - open it in node and execute it
 - open it in the browser and look at it (UI)
- What if the code changes?
 - then, you need to do the tests again
- Programmatic tests are code that tests other code
 - easy to run them again whenever the code changes
 - these are generally preferred
- What did we do in HW1 – HW3?

“Manual” vs Programmatic Tests in UI

- **For UI, manual testing is still common**
 - **written tests are hard to write and imperfect**
need to see it in the browser to be sure that it looks right
 - **we will test UI manually**
 - **non-UI functions and all server code tested programmatically**

“Engineers are paid to think and understand.”

— Class slogan #1

Writing a Programmatic Test

1. Choose an input / configuration
 - description of the inputs / configuration is the “test case”
2. **Think** through what the expected answer is
 - look at the specification for the correct answer
 - if you run the code to get the answer, you are **not testing**
3. Write code that
 - a) calls the function that input
 - b) compares the actual answer to the expected one
 - c) throws an error if they do not match
 - useful libraries for doing this...

Writing a Programmatic Test: Example Code

```
// number.ts
```

```
/** Determines whether m is a prime number. */
```

```
export const isPrime = (m: bigint): boolean => {  
    ...  
};
```

```
/** Returns the mth Fibonacci number. */
```

```
export const fib = (m: bigint): bigint => {  
    ...  
};
```

Writing a Programmatic Test with Mocha

```
// number_test.ts
```

```
import * as assert from "assert";  
import { isPrime, fib } from "../number";
```

```
describe("number", function() {  
  it("isPrime", function() {  
    assert.strictEqual(isPrime(2n), true);  
    assert.strictEqual(isPrime(3n), true);  
    assert.strictEqual(isPrime(4n), false);  
  });  
  
  it("fib", function() {  
    assert.strictEqual(fib(1n), 1n);  
    assert.strictEqual(fib(3n), 2n);  
    assert.strictEqual(fib(9n), 34n);  
  });  
});
```

Don't worry too much
about the details here....

Mocha Asserts & Equality

```
// number_test.ts
```

```
import * as assert from "assert";
```

```
import { isPrime, fib } from "../number";
```

```
describe("number", function() {
```

```
  it("isPrime", function() {
```

```
    assert.strictEqual(isPrime(2n), true);
```

```
    assert.strictEqual(isPrime(3n), true);
```

```
    assert.strictEqual(isPrime(4n), false);
```

```
  });
```

- **use** `assert.strictEqual` **to compare primitive values**
- **use** `assert.deepStrictEqual` **to compare records & arrays**

Running Programmatic Tests with Mocha

```
$ npm run test
```

```
number
```

```
  1) isPrime
```

```
    ✓ fib
```

```
1 passing (5ms)
```

```
1 failing
```

```
1) number
```

```
    isPrime:
```

```
      AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
```

```
      false !== true
```

```
      + expected - actual
```

```
      -false
```

```
      +true
```

```
      at Context.<anonymous> (src/number_test.ts:9:12)
```

```
      at processImmediate (node:internal/timers:511:21)
```

Running Programmatic Tests with Mocha

```
$ npm run test
```

```
number
```

```
✓ isPrime
```

```
✓ fib
```

```
2 passing (2ms)
```

Ground Rules for Testing (1/4)

1. Only need to test inputs **allowed** by the spec
 - there is no correct answer for other inputs

```
/** Determines whether a positive integer is prime. */  
export const isPrime = (n: bigint): boolean => {  
  if (n <= 0n)  
    throw new Error(`not a positive integer: ${n}`);  
  
  ...  
};
```

Ground Rules for Testing (2/4)

1. Only need to test inputs allowed by the spec
 - there is no correct answer for other inputs
2. Choose tests for each function **individually**
 - pick tests to do a good job of testing that one function

```
/** Determines whether a positive integer is prime. */  
export const isPrime = (n: bigint): boolean => {  
  if (n <= 0n)  
    throw new Error(`not a positive integer: ${n}`);  
  
  const m = intSqrt(n); // integer square root of n  
  ...  
};
```

intSqrt has its own tests!

Ground Rules for Testing (3/4)

1. Only need to test inputs allowed by the spec
 - there is no correct answer for other inputs
2. Test each function individually
 - assume anything it calls is correct (its own tests will check)
3. Test code should be **simple**
 - any loops in tests need their own tests!

How Many Tests are Strictly Necessary?

- Consider the following function:

```
// Allows inputs 0 <= a, b, c <= 10,000 ...  
const f = (a: bigint, b: bigint, c: bigint) => {  
  ...  
};
```

- How many tests needed guarantee correctness?
 - 1 trillion!
 - "just write a loop and ..."
 - the code in that loop could also be wrong
 - cannot **think** through even 1000 tests
 - most code we write cannot be *exhaustively* tested

Ground Rules for Testing (4/4)

1. Only need to test inputs allowed by the spec
 - there is no correct answer for other inputs
2. Test each function individually
 - assume anything it calls is correct (its own tests will check)
3. Test code should be **simple**
 - any loops in tests need their own tests!
4. If there are fewer than 10 allowed inputs, then do test them all!
 - take advantage of the easy case

Choosing Test Cases

// Returns true iff n is a prime number

```
const isPrime = (n: bigint): boolean => { ... }
```

- How about if we test 2, 3, 4, 7, 12, 97, 99?
 - seems okay?

Choosing Test Cases (Gone Wrong!)

```
// Returns true iff n is a prime number
const isPrime = (n: bigint): boolean => {
  if (n < 100n) {
    return PRIME_CACHE[n]; // precomputed answers
  } else {
    for (let k = 2n; k*k <= n; k++) {
      if (n % k === 0n)
        return false;
    }
    return true;
  }
};
```

Cases 2 .. 100 are table lookups!

We didn't test the loop at all!

Impossible to know this without
looking at the actual code.

Clear-Box Testing

- We need to look at the code to know what to test
 - this is called "clear-box testing"
 - it will be our **primary** heuristic
- In this class, we want a clear rule for how many tests
 - you want homework and tests to have clear right answers
- Outside of class, these rules are also good
 - most programmers will be familiar with these concepts
 - but, not comprehensive (usually a “floor”)

Statement Coverage by Example (isPrime)

- Simplest metric is "statement coverage"
 - what percentage of the statements in the code are executed by *at least one* test
 - this should be nearly 100%

```
export const isPrime = (n: bigint): boolean => {  
  if (n <= 0n)  
    throw new Error(`not a positive integer: ${n}`);  
  
  ... // code for positive integer inputs  
};
```

- The "throw" is not executed by any *allowed* input
 - we only test the allowed inputs

Statement Coverage

- Simplest metric is "statement coverage"
 - what percentage of the statements in the code are executed by *at least one* test
- Must test **100%** of code reachable on allowed inputs
 - cannot send code to users that you didn't even try!
 - we will refer to this as having "full statement coverage"
- Are we done?

Statement Coverage and Conditionals

- Consider the following function:

```
/** Returns the smaller of a and b. */  
const min = (a: bigint, b: bigint): bigint => {  
  let m = a;  
  if (a <= b)  
    m = a;  
  return m;  
};
```

- testing on a=1 b=2 gives full statement coverage
- what is the bug?
 - gives the wrong answer whenever a > b
- we never tested the case where the "if" doesn't execute

Conditionals

Conditionals are "if" statements

```
if (n > 0) {  
    x = 2 * (n - 1);  
} else {  
    x = 0;  
}
```

Every conditional has two branches (“then” and “else”)

Conditionals and Implicit Else

Conditionals are "if" statements

```
if (n > 0) {  
    x = 2 * (n - 1);  
}  
= if (n > 0) {  
    x = 2 * (n - 1);  
}  
= } else {  
    }  
= }
```

Every conditional has two branches (“**then**” and “**else**”)

- missing "else" still has an empty else branch

Branch Coverage: Back to min

- Consider the following function:

```
/** Returns the smaller of a and b. */  
const min = (a: bigint, b: bigint): bigint => {  
  let m = a;  
  if (a <= b)  
    m = a;  
  return a;  
};
```

- problem only arises when "if" falls through to code after
- if every branch ends with `return` / `throw`,
then statement coverage = branch coverage
always true for code without mutation of local variables

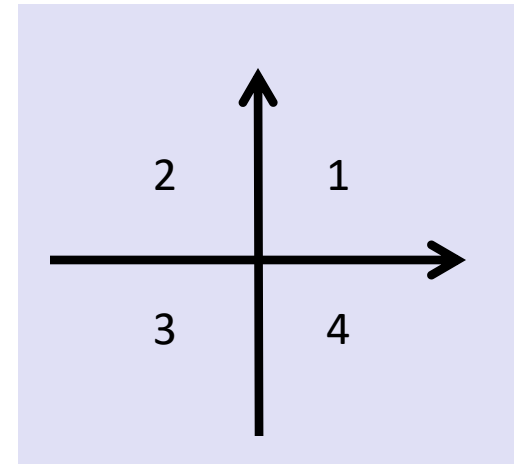
Branch Coverage

- Next metric is "branch coverage"
 - for what percentage of the conditionals, are both branches executed by some test
- Must test all branches reachable on allowed inputs
 - can ignore branches that are unreachable
i.e., the ones that `throw new Error` on bad inputs
- Are we done?

Is Branch Coverage Enough?

- Consider the following function:

```
/** Returns quadrant containing (x, y). */  
const quad = (x: number, y: number): 1|2|3|4 => {  
  let answer;  
  if (x >= 0) {  
    answer = 1;  
  } else {  
    answer = 2;  
  }  
  if (y < 0)  
    answer = 4;  
  return answer;  
};
```




- testing on (2, -2) and (-2, 2) gives full branch coverage
- this code is *really* wrong... it never returns 3!

Path Coverage?

- More advanced metrics could fix this
 - "path coverage" would require 4 tests
 - #paths can grow exponentially in #branches
- For straight-line code and conditionals, we will only require branch coverage
- What about loops / recursion?

Test Coverage and Loops

- Consider the following function:

```
const bsearch = (x: bigint, arr: Array<bigint>): number => {  
  let lo = 0;  
  let hi = arr.length;  
  while (lo < hi) { // x could be in arr[lo .. hi-1]  
    const m = Math.floor((lo + hi) / 2);  
    if (x < arr[m]) {  
      hi = m;  
    } else if (x > arr[m]) {  
      lo = m + 2;   
    } else {  
      return m;  
    }  
  }  
  return hi;  
};
```

Testing on $x=1, 2, 3$ $arr=[2]$
gives full statement coverage

But the code is wrong.

In general, values written inside the loop
are not read until the next time around,
so you need 2+ iterations to test them.

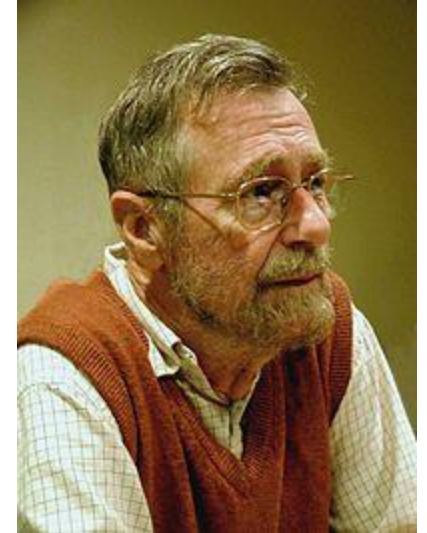
Loop Coverage (and Recursion Coverage)

- Our last metric is "loop coverage" (non-standard terminology)
 - what percent of loops are executed 0, 1, and many (2+) times by some test case
- Same idea applies to recursion
 - some arguments passed to recursive calls may not be read until the second recursive call
 - full loop coverage means every recursive call is executed 0, 1, and many times by some test
- Are we done?
 - no!

What Can We Learn From Testing?

**“Program testing can be used to show the presence of bugs,
but never to show their absence!”**

Edsger Dijkstra
Notes on Structured Programming, 1970



**“Beware of bugs in the above code;
I have only proved it correct, not tried it.”**

Donald Knuth, 1977

Summary of testing requirements

- At least two tests for any function (non-UI)
- Must have full coverage of *reachable*
 - statements: must be executed
 - branches: must execute both branches
 - loops / recursion: must run 0, 1, & many times

Agenda

- ✓ Administrivia
- ✓ Finish Testing (finish topic 4)
 - Practice exercises
- Reasoning (start topic 5)

How Many Tests: Example 0

```
// x must be a non-negative integer
const f = (x: bigint): number => {
  if (x === 0n) {
    return 0;
  } else {
    return Math.sin(Math.PI * (Number(x) + 0.5));
  }
}
```

How many tests? Which ones?

- 0 (top branch) and 1 (bottom branch)

statement coverage = branch coverage since no "fall through"

How Many Tests: Example 1

```
// x must be a non-negative integer
const f = (x: bigint): bigint => {
  if (x < 3n) {
    return 0n;
  } else if (x < 10n) {
    return (x - 3n) / 10n;
  } else {
    return 1n;
  }
}
```

Think-pair-share!



sli.do #cse331

How many tests? Which ones?

- 2 (top), 6 (middle), and 10 (bottom)

How Many Tests: Example 2

```
// a and b must be a non-negative
const f = (a: number, b: number): number => {
  if (a > b)
    a = b;
  return Math.abs(a);
}
```

Think-pair-share!



sli.do #cse331

How many tests? Which ones?

- a=2, b=1 gives full statement coverage
- adding a=1, b=2 gives branch coverage

How Many Tests: Example 3

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

How many tests? Which ones?

- 1 (0 recursive calls)
- 2 (1 recursive call)
- 5 (2 recursive calls)

Think-pair-share!



sli.do #cse331

How Many Tests: Example 4

```
// x must be an integer between 1 and 10
```

```
const f = (x: bigint): bigint => {
  if (x === 1n) {
    return 0n;
  } else if (x % 2 === 1n) {
    return -1n * f(x - 1n);
  } else {
    return 4n + f(x - 2n);
  }
}
```

Think-pair-share!



slj.do #cse331

How many tests? Which ones?

- only 10 inputs, so... all of them

Other Heuristics

Not mandatory for 331 but useful in practice:

- Make sure every argument value is changed
- Look at special values
 - null, undefined, NaN, empty array, etc. often have bugs
- Look at the specification for branches
 - maybe the code doesn't split inputs where it should!
 - e.g., spec splits into “if $x \geq 0$ ” but code is “**if** ($x > 0$)”

How to Fix a Bug

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

Notes Posted on the Website

- **Shorter version of everything we've discussed**
- List notes additionally include
 1. Defines a few more useful list functions
 2. Describes important properties of concat:
 - operator notation " $\#$ "
 - associativity and identity
 3. Mentions important applications of lists
 - maps *are* lists of (key, value) pairs, sets can be defined as lists
 - Lists are our most important data type!

Administrivia

- Office hours
 - Polled y'all, no long-term changes for now
 - Jaela's Friday OH cancelled (this week only)

Monday	Tuesday	Wednesday	Thursday	Friday
10:50–11:50 Lecture DEM 102		10:50–11:50 Lecture DEM 102	10:50–11:50 Quiz Sections AfA - ARC G070, AB - ECE 037	10:50–11:50 Lecture DEM 102
12:00–13:00 OH (Katherine) CSE 2/F Breakout		12:00–13:00 OH (Raio) CSE 4/F Breakout	12:00–13:00 OH (Katherine) CSE 2/F Breakout	
14:00–15:00 OH (Jaela) CSE 4/F Breakout & Zoom	14:00–15:00 OH (Jon) Zoom	14:30–15:30 OH (Saan) CSE 4/F Breakout	13:30–14:30 OH (Jon) CSE 2/F Breakout	13:00–14:00 OH (Jaela) CSE 2/F Breakout & Zoom
15:00–16:00 OH (Raio) CSE 4/F Breakout				
		18:00–19:00 OH (Ali) Zoom	18:00–19:00 OH (Ali) Zoom	

- Tomorrow: section + HW3 due (on-time)!