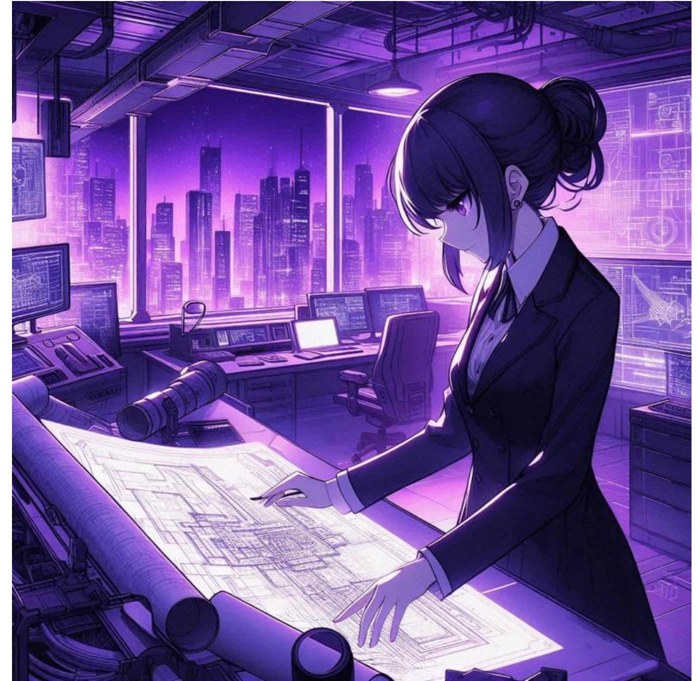


CSE 331

Specifications

James Wilcox and Kevin Zatloukal

Specifications

Specifications

- Correctness requires a <u>definition</u> of the correct answer

- Description must be <u>precise</u>
  - 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 <u>how</u> to calculate the answer
  - lays out the exact steps to perform to get the answer
- **Declarative** specification says <u>what</u> 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

- **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

- **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

- 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

- 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 <u>thinking</u> to make sure this code returns a number with the properties above

Example: Imperative Specification

- From HW3: Dijkstra's Algorithm

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

while active is not empty:
  minPath = active.removeMin()  // shortest active path

  if minPath.end is end:
    return minPath  // shortest path from start to end!

  if minPath.end is in finished:
    continue  // longer path to minPath.end than the one we found before

  add minPath.end to finished  // just found shortest path to here!

  // add all paths that have one step added to this shortest path
  for each edge e in adjacent.get(minPath.end):
    if e.end is not in finished:
      newPath = minPath + e
      add newPath to active

return undefined  // no path from start to end :(
```

steps are described fully
(just translate to TypeScript)

"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

Recall: 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

Examples from the Java APIs

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

```
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;
```

Imperative

Examples from the Java APIs

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

```
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.

Imperative

```
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.)

Declarative

Examples from the Java APIs

`java.util.Object`

```
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:

- If two objects are equal according to the `equals(Object)` 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(java.lang.Object)` 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.

Declarative

Next Up…

- 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 caches 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 ($\mathbb{B}$, $\mathbb{S}$, $\mathbb{S}^*$)

Basic Data Types in TypeScript

Condition	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**

Ways to Create New Types In Math

- **Union Types** $\mathbb{S}^* \cup \mathbb{N}$
  - contains every object in either (or both) of those sets
  - e.g., all strings and natural numbers

- If $x \in \mathbb{N} \cup \mathbb{S}^*$, then x could be a natural or string

- Two sets can contain common elements
  - in this case, the sets are disjoint

Ways to Create New Types in TypeScript

- **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

Type Narrowing With "If" Statements

- **Union Types** `string | bigint`
  - can be either one of these

- **How do we work with this code?**

```
const x: string | bigint = …;

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

Type Narrowing vs Casting

```
const x: string | bigint = …;

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

- Note that this does not require a **type cast**
  - TypeScript knows x is a `bigint` inside the “if” (narrowing)

- 331: there are **no type casts** (won’t even show syntax)
  - unlike Java, TypeScript casts are unchecked at runtime
  - seem designed to create extremely 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

Compound Types In Math

- Compound types combine multiple data types
  - multiple ways build them
- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{x\text{: }3, y\text{: }5\}$
- Note that $\{x\text{: }3, y\text{: }5\} = \{y\text{: }5, x\text{: }3\}$ 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);
```

Optional Fields in TypeScript

- Records can have optional fields

```
type T = {x: bigint, y?: bigint};

const t: T = {x: 1n};
```

  – type of “ `t.y` ” is “ `bigint | undefined` ”

- Functions can have optional arguments

```
const f = (a: bigint, b?: bigint): bigint => {
  console.log(b);
};
```

  – type of “ `b` ” is “ `bigint | undefined` ”

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

":=" means a definition

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

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:**

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

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

  - **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 double : $\mathbb{N} \to \mathbb{N}$ is defined by...

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

Simple Functions in Math

- **Another example:**

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

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

  - **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**

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

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

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**

$$
\begin{aligned}
&\mathrm{abs} : \mathbb{R} \to \mathbb{R} && \\
&\mathrm{abs}(x) := x && \text{if } x \geq 0 \\
&\mathrm{abs}(x) := -x && \text{if } x < 0
\end{aligned}
$$

  - **conditions must be <u>exclusive</u> and <u>exhaustive</u>**
    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"**

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

  - **first case matches only** 0
  - **second case matches numbers** 1 **more than some** $n : \mathbb{N}$ **…**

    double(6) = 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** $2, 3, 4, \ldots$

- **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

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

  - **e.g., need to explain why** $\text{double}(n\text{-}1)$ **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 $\mathrm{not} : \mathbb{B} \to \mathbb{B}$ is defined as follows:

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

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

Pattern Matching on Records

- **Can pattern match on individual fields of a record**

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

$$\begin{aligned}&\text{change} : \text{Steps} \to \mathbb{Z}\\&\text{change}(\{n\text{: } m, \text{fwd: } T\}) := m\\&\text{change}(\{n\text{: } m, \text{fwd: } F\}) := -m\end{aligned}$$

  - **clear that the rules are exclusive and exhaustive**

- **Can match on multiple parameters**
  - **e.g.**, $\text{change}(\{n\text{: } m+5, \text{fwd: } T\}) := 2m$
  - **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

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

- **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 double(n)
but code says 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 function with all three types

```
// n 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?

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

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

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

Summary of Last Time

- 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
  - familiar data types: numbers, records, tuples
  - familiar coding: expressions, if statements, recursion
  - unfamiliar(?): pattern matching

all but pattern matching translate easily to TypeScript

- Most important data type is missing…

Inductive Data Types

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

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

Only possible to make **non-negative** integers

- **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

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

- **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

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

- **Inductive definition of lists of integers**

nil

$\text{cons}(3, \text{nil})$

$\text{cons}(2, \text{cons}(3, \text{nil}))$

$\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$

Our most important data type!

Shorthand Notation for Lists

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

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

- **Examples:**

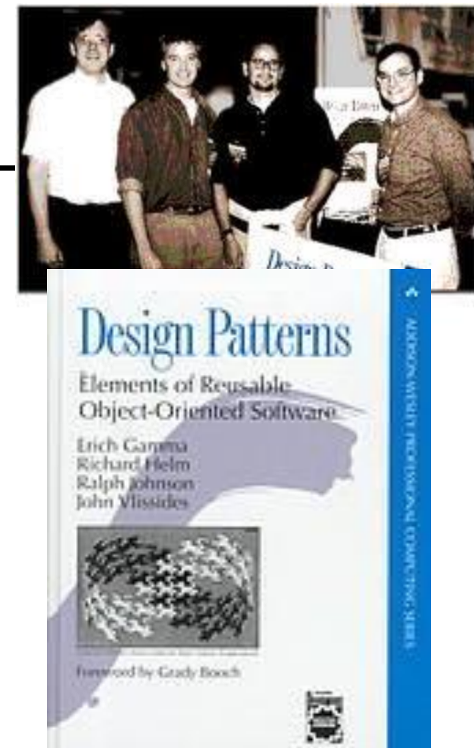
nil	nil	$[]$
$\text{cons}(3, \text{nil})$	$3 :: \text{nil}$	$[3]$
$\text{cons}(2, \text{cons}(3, \text{nil}))$	$2 :: 3 :: \text{nil}$	$[2, 3]$
$\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$	$1 :: 2 :: 3 :: \text{nil}$	$[1, 2, 3]$

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



- 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

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

- **Implemented in TypeScript as**

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

- **How do I check if my list is empty?**

```
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: Readonly<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" (there is a **design pattern** for ensuring this)

Inductive Data Types in TypeScript

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

Inductive Data Types in TypeScript

- **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…

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

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

- **Mathematical definition of list length:**

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

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

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

Length of a List

- **Mathematical definition of length**

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

- **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

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

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

$$\text{swap} : \text{List} \to \text{List}$$

$$\begin{aligned} \text{swap}(\text{nil}) &:= \text{nil} \\ \text{swap}(x :: \text{nil}) &:= x :: \text{nil} \\ \text{swap}(x :: y :: L) &:= y :: x :: \text{swap}(L) \end{aligned}$$

  - any list is either nil or $x :: \text{nil}$ or $x :: y :: L$ for some x, y and L
  - cases are exclusive and exhaustive

Swapping Elements in a List

$$
\begin{aligned}
\text{swap}(\text{nil}) &:= \text{nil} \\
\text{swap}(x :: \text{nil}) &:= x :: \text{nil} \\
\text{swap}(x :: y :: L) &:= y :: x :: \text{swap}(L)
\end{aligned}
$$

- **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)));
  }
};
```

= S

TypeScript will see that these are valid since `S.kind != "nil"` and `S.tl.kind != "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
  - that usually indicates recursion

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
3 :: nil	3
2 :: 3 :: nil	2+3
1 :: 2 :: 3 :: nil	1+2+3
…	…

Definition of Sum of Values in a 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
…	…

- **Mathematical definition of sum:**

sum(nil) :=

sum(x :: L) :=

Definition of Sum of Values in a List

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

- **Mathematical definition of sum:**

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

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

sum(1 :: 2 :: 3 :: nil)	
$= 1 + \text{sum}(2 :: 3 :: \text{nil})$	def of sum (2nd line)
$= 1 + 2 + \text{sum}(3 :: \text{nil})$	def of sum (2nd line)
$= 1 + 2 + 3 + \text{sum}(\text{nil})$	def of sum (2nd line)
$= 1 + 2 + 3$	def of sum (1st line)

Sum of Values in a List

- **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);
  }
};
```

Definition of 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	

Definition of List Equality

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** $\text{equal} : (\text{List}, \text{List}) \to \mathbb{B}$

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

Definition of Sum of Values in a List

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

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

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

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

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

Inductive Data Types in TypeScript

- **Translation to TypeScript**

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

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

math definition may be easier to read

Definition of 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
…		

Definition of List Concatenation

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** concat : (List, List) → List

concat(nil, R) :=

concat(x :: L, R) :=

Definition of List Concatenation

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

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

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

- **Check that this matches examples…**

$$\begin{aligned} &\text{concat}(1 :: 2 :: \text{nil}, 3 :: 4 :: \text{nil}) && \\ &\quad = 1 :: \text{concat}(2 :: \text{nil}, 3 :: 4 :: \text{nil}) && \text{def of concat (2}^{\text{nd}} \text{ line)} \\ &\quad = 1 :: 2 :: \text{concat}(\text{nil}, 3 :: 4 :: \text{nil}) && \text{def of concat (2}^{\text{nd}} \text{ line)} \\ &\quad = 1 :: 2 :: 3 :: 4 :: \text{nil} && \text{def of concat (1}^{\text{st}} \text{ line)} \end{aligned}$$

Definition of List Concatenation

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

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

- **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));
  }
};
```

Notes on Lists Posted on the Website

- Shorter version of everything we've discussed

- In addition:
  1. Defines a few more useful list functions
  2. Mentions 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 defined as lists

- Lists are our most important data type!

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
- 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 vs Integration Tests

- 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 Dijkstra in HW3?
  - we are doing end-to-end testing
  - this makes debugging harder! (more to search)

Unit vs Integration Tests

- 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

- 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

- Even individual functions might be too big…
  - split out pieces into their own functions

- Our coding conventions help enforce this

```
const heap = new Heap<MyObj>((a: MyObj, b: MyObj) => {
    … multi-line calculation …
  });
```

  - 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)
- No downside… unless 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 HW3?

“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)

- No downside… unless the code changes
  - then, you need to do the tests again

- 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

Writing a Programmatic Test

1. Choose an input / configuration
   - description of the inputs / configuration is the "test case"

2. **Think** through what the answer should be
   - look at the specification for the correct answer
   - if you run the code to get the answer, you are **not testing**

“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
   - 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

```
// number.ts

/** Returns the greatest common divisor of a and b. */
export const gcd = (a: bigint, b: bigint): bigint => {
  …
};

/** Determines whether n is a prime number. */
export const isPrime = (n: bigint): boolean => {
  …
};
```

Writing a Programmatic Test with Mocha

```
// number_test.ts
import * as assert from "assert";
import { gcd, isPrime } from "./number";

describe("number", () => {
  it("isPrime", () => {
    assert.strictEqual(isPrime(2n), true);
    assert.strictEqual(isPrime(3n), true);
    assert.strictEqual(isPrime(4n), false);
  });
  it("gcd", () => {
    assert.strictEqual(gcd(3n, 2n), 1n);
    assert.strictEqual(gcd(9n, 3n), 1n);
    assert.strictEqual(gcd(12n, 9n), 3n);
  });
});
```

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

Writing a Programmatic Test with Mocha

```
// number_test.ts
import * as assert from "assert";
import { gcd, isPrime } from "./number";

describe("number", () => {
  it("isPrime", () => {
    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
    ✓ isPrime
    ✓ gcd

  2 passing (3ms)
```

Ground Rules for Testing

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

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!

How Many Tests are 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

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

```
// 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, I want a clear rule for how many tests
  - want homework and tests to have clear right/wrong answers

- Outside of class, these rules are also good
  - most programmers will be familiar with these concepts

Statement Coverage

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

- 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

Conditionals are "`if`" statements

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

Every conditional has two branches ("then" and "else")

– missing "`else`" still has an empty else branch

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

Branch Coverage

- 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

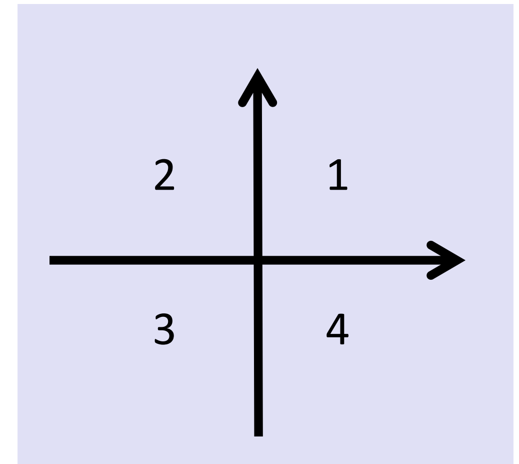
- Must test all branches reachable on allowed inputs
  - can ignore branches that are unreachable
    - i.e., the ones that `throw new Error` on bad inputs

- Are we done?

Branch Coverage

- 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 wrong… it never returns 3!

How Many Tests Are Required?

- 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?

How Many Tests Are Required?

- Consider the following function:

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

Testing on s="a"/"b"/"c" A=["b"] 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.

How Many Tests Are Required?

- 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

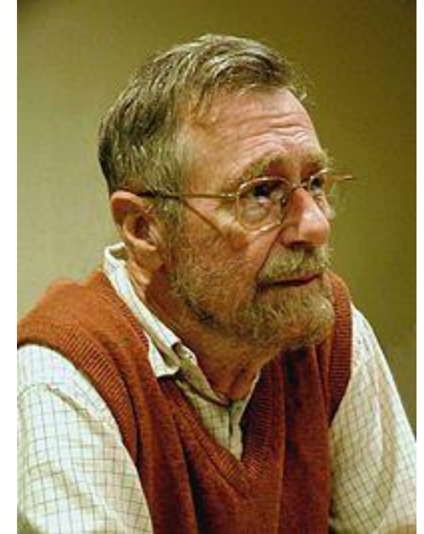
- Are we done?
  - no!

What Can We Learn From Testing?

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

Edsgar 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
- Summary notes posted on the website
  - includes other heuristics are also useful in practice

Example 1

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

How many tests? Which ones?

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

  statement coverage = branch coverage since no "fall through"

Example 2

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

How many tests? Which ones?

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

Example 3

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

How many tests? Which ones?

- $m=2$, $n=1$ gives full statement coverage
- adding $m=1$, $n=2$ gives branch coverage

Example 4

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

How many tests? Which ones?

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

Example 5

```
// n must be an integer between 1 and 10
const f = (n: bigint): bigint => {
  if (n === 1n) {
    return 0n;
  } else {
    return 1n + 2n * f(n - 1n);
  }
}
```

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 "$\text{if } x \geq 0$" but code is "`if (x > 0)`"