CSE 331

Correctness

Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox
Shipping Software

• Building shippable version is ~10x harder than demo
  – demo version needs to work when used properly
  – shipped version needs to work properly no matter what

• 1m users will try millions of cases that you didn’t
  – needs to work properly on all cases, even ones you didn’t try

• Users are completely unforgiving about bugs
  – no partial credit for effort

• How is this achieved in practice?
Standard Techniques for Correctness

Standard practice uses three techniques:

- **Testing**: try it on a well-chosen set of examples
- **Tools**: type checker, libraries, etc.
- **Reasoning**: *think through* your code on all inputs
  - have another person do the same ("code review")

Each removes \(~2/3^{rd}\) bugs but of different kinds
Combination removes >97% of bugs
Review: Software Development Process

Given: a problem description (in English)

Beta users are understanding about failures, but regular users are completely unforgiving!
Which Ones and How Much

• The first question to ask yourself:

   How much of each is needed for my program?

• Correctness is easier for some programs vs others

• Personally, I break this into several cases...
  – “levels” of difficulty
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Small Number of Inputs

• Set of possible inputs / configurations is small
  – say 20 or fewer...

• Just check them all!
  – this is the right answer

• This category does not require a programmer
  – anyone can check the answer
  – programming is hard, so skip it when you can
Small Number of Inputs

• Coding is the wrong tool for this job
  – can happen in part of a larger application

• iPhone development lets you draw the UI:
Small Number of Inputs

Greg Brockman (@gdb) of OpenAI just demoed GPT-4 creating a working website from an image of a sketch from his notebook.

It’s the coolest thing I’ve *ever* seen in tech.

If you extrapolate from that demo, the possibilities are endless.

A glimpse into the future of computing.
Small Number of Inputs

• Can happen as part of a larger application
  – would then require an understanding of the context
  – still a case where AI could be used (checked by a programmer)

• Happens more often than you might think
  – individual function can have a small number of inputs
    e.g., two boolean inputs (only 4 configurations)
  – quite common with UI
    e.g., when I click the button, it should say “hi”

• Be on the lookout for these cases
  – save yourself work by spotting them
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>no mutation</td>
<td>“</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>local variable mutation</td>
<td>“</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>array mutation</td>
<td>“</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>heap state mutation</td>
<td>“</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>no mutation</td>
<td>“</td>
<td>libraries</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>local variable mutation</td>
<td>“</td>
<td>“</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>array mutation</td>
<td>“</td>
<td>“</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>heap state mutation</td>
<td>“</td>
<td>“</td>
<td></td>
</tr>
</tbody>
</table>
# Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2</td>
<td>no mutation</td>
<td>“”</td>
<td>libraries</td>
<td>calculation induction</td>
</tr>
<tr>
<td>3</td>
<td>local variable mutation</td>
<td>“”</td>
<td>“”</td>
<td>Floyd logic</td>
</tr>
<tr>
<td>4</td>
<td>array mutation</td>
<td>“”</td>
<td>“”</td>
<td>for-any facts</td>
</tr>
<tr>
<td>5</td>
<td>heap state mutation</td>
<td>“”</td>
<td>“”</td>
<td>rep invariants</td>
</tr>
</tbody>
</table>
Reminders

• **We will set an extremely high bar for correctness**
  – “programmers overestimate the importance of **efficiency** and underestimate the difficulty of **correctness**.”

• **Now is the time to practice proper technique**
  – much harder to learn technique on harder problems

• **This topic: Testing and Tools**
  – you have important responsibilities here

• **Next topic: Reasoning (pt. 1)**
  – the most important part
    everything else can (and will) be automated
  – see the other two class slogans!
Specifications
Specifications

• Correctness requires a description of the correct answer
  – true at any level of correctness

• 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 extremely 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
  – future: prove our calculation meets the spec

• Can implement a *different* imperative specification
  – future: prove ours is equivalent to the original specification
Example: Imperative Specification

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

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

just translating math to TypeScript
straight from 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 \( x \leq y^2 \)

```javascript
const sqrt = (x: bigint): bigint => {
  ??
}
```

we are left to figure out how to do this... not straight from the spec
Example: Declarative Specification

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

```javascript
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

correctness is inherently harder with a declarative spec
# Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Straight From the Spec

• Instructions say exactly how to calculate answer
  – given an imperative specification
  – we are just translating math into code

• Still easy to make mistakes!
  – too many inputs to test them all
  – need to additional ways of checking for bugs

• Still important to get it right!
Non-programming Example

- Important to calculate grades correctly!

\[ f(x) = 0.6 \times G_4 + 0.15 \times I_4 + 0.25 \times J_4 \]

<table>
<thead>
<tr>
<th></th>
<th>Homework</th>
<th>Extra Credit</th>
<th>Midterm</th>
<th>Final</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Student</td>
<td>87.5%</td>
<td>1</td>
<td>64.0%</td>
<td>91.6%</td>
<td>0.25*J_2</td>
</tr>
<tr>
<td>2nd Student</td>
<td>91.4%</td>
<td>1</td>
<td>87.9%</td>
<td>70.8%</td>
<td>85.8%</td>
</tr>
<tr>
<td>3rd Student</td>
<td>86.2%</td>
<td>5</td>
<td>93.0%</td>
<td>62.0%</td>
<td>81.8%</td>
</tr>
<tr>
<td>4th Student</td>
<td>96.5%</td>
<td>1</td>
<td>60.9%</td>
<td>69.0%</td>
<td>84.4%</td>
</tr>
<tr>
<td>5th Student</td>
<td>98.2%</td>
<td>0</td>
<td>88.6%</td>
<td>91.3%</td>
<td>95.0%</td>
</tr>
<tr>
<td>6th Student</td>
<td>86.3%</td>
<td>0</td>
<td>91.5%</td>
<td>63.0%</td>
<td>81.3%</td>
</tr>
</tbody>
</table>

- The syllabus says the formula
  - ask someone else to double-check ("code review")
  - spot check some of them
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Correctness when Straight From the Spec

Correctness requires these elements:

• Code review
  – second set of eyes

• Type checker
  – third set of eyes (so to speak)
    (tends to find different mistakes than human reviewers)

• Good set of tests
  – can’t test every case... need to pick the the right ones
    (more on this next lecture...)
Type Checkers

• The main part of “Tools” is the type checker
  – libraries are the other important part

• Type Checkers are very useful for finding bugs
  – another set of “eyes” helping us find them
  – you have probably learned this already
Type Checkers

• TypeScript and Java have different type systems
  – they can catch different bugs for us
    TypeScript ensures references are not null (Java does not)
    more examples coming soon...

• Critical to understand what the tools will miss
  – can ignore issues the tools would catch
  – must carefully think about issues the tools would miss
Important Java Examples

• In Java, your responsibility to catch e.g.

  1. **Using “==” between strings**
     Fine in principle, but not what you want
     Especially bad bug because it is likely to work in tests!

  2. **Wrong order of arguments**
     Famous example: `memset(int[] A, int n, int v)`
     You call `memset(A, 0, 100)` instead of `memset(A, 100, 0)`

  3. **Defining “equal(Object o)” instead of “equals”**
     Perfectly fine, but doesn’t override equals method
     (@Override annotations added to Java to catch this!)

not all of these apply in TypeScript! experienced programmers are always on the look out for these
John Carmack @ID_AA_Carmack 2h
I spent *hours* today debugging something that turned out to be a single wrong letter in the code: a .ge() should have been .gt().
How-To For Straight From the Spec

• We will start with completely **formal** specs

• **Translate math into our programming language**
  – TypeScript here, but could also be Java

• **Rest of this lecture:**
  – define math for data and code
  – describe how to translate those into TypeScript
    try to make the translations as *straightforward* as possible (fewer mistakes)
  – mention new TypeScript features when related
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:
  
  \[
  \begin{align*}
  x \in \mathbb{Z} & \quad x \text{ is an integer} \\
  x \in \mathbb{N} & \quad x \text{ is a non-negative integer (natural)} \\
  x \in \mathbb{R} & \quad x \text{ is a real number} \\
  x \in \mathbb{B} & \quad x \text{ is T or F (boolean)} \\
  x \in \mathbb{S} & \quad x \text{ is a character} \\
  x \in S^* & \quad x \text{ is a string}
  \end{align*}
  \]  

  [non-standard names]
### Basic Data Types in TypeScript

<table>
<thead>
<tr>
<th>Condition</th>
<th>Math</th>
<th>TypeScript</th>
<th>Up to Us</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>( x \in \mathbb{Z} )</td>
<td>bigint</td>
<td></td>
</tr>
<tr>
<td>natural</td>
<td>( x \in \mathbb{N} )</td>
<td>bigint</td>
<td>non-negative</td>
</tr>
<tr>
<td>real</td>
<td>( x \in \mathbb{R} )</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>( x \in \mathbb{B} )</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>character</td>
<td>( x \in \mathbb{S} )</td>
<td>string</td>
<td>length 1</td>
</tr>
<tr>
<td>string</td>
<td>( x \in \mathbb{S}^* )</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

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

  – can be either one of these

• **How do we work with this code?**

  ```typescript
  // 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?

```javascript
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

```typescript
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

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

if (typeof f(x) === "bigint") {
  console.log(isPrime(f(x))) // is this okay?
}
```

- No! TypeScript will (properly) reject this
  - no guarantee that f(x) returns the same value both times!
Type Narrowing of Function Calls

```typescript
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., \(\{x: 3, \ y: 5\}\)

- **Note that** \(\{x: 3, \ y: 5\} = \{y: 5, \ x: 3\}\)
  - field names matter, not order
  - (also, “=” in math means same values)
Record Types in TypeScript

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

- Retrieve a part by name:

  ```typescript
  const t: {x: bigint, y: bigint} = ... ;
  console.log(t.x);
  ```
Optional Fields in TypeScript

• Records can have optional fields

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

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

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

• Functions can have optional arguments

```typescript
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
TUPLE TYPES IN TYPESCRIPT

• Tuple Types [bigint, bigint]

• Must assign names to the parts to refer to them

• How would we do this in math?
Retrieving Part of a Tuple

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

- Tuple Types \( \mathbb{N} \times \mathbb{N} \)

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

  \textbf{“:=” means a definition}

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

  Then, we have \(a = 5\) and \(b = 7\)

- Tuple Types \([\text{bigint}, \text{bigint}]\)

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

• TypeScript can ensure values aren’t modified
  – extremely useful! (mutation makes everything harder)

• Tuple types should always be readonly

```typescript
type IntPair = readonly [bigint, bigint];
```

• Individual fields of records should be marked readonly

```typescript
type IntPair = {readonly x: bigint,
  readonly y: bigint};
```
Simple Functions in Math

• Simplest function definitions are single expressions

• Will write them in math like this:

  \[
  \text{func double}(n : \mathbb{N}) := 2n
  \]

  \[
  \text{func dist}(p : \{x : \mathbb{R}, y : \mathbb{R}\}) := (p.x^2 + p.y^2)^{1/2}
  \]

  – any normal math allowed in the expression
Simple Functions in Math

• Can define short-hand for types in math also

  ```
  type Point := {x: ℝ, y: ℝ}
  
  func dist(p : Point) := (p.x^2 + p.y^2)^(1/2)
  ```

• Can put the argument type on the right instead

  ```
  func dist(p) := (p.x^2 + p.y^2)^(1/2)
  
  for any p : Point
  ```

  – needs to be described somewhere (we’re not too picky)
  – will need this in some cases coming shortly...
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

  \[
  \text{func } \quad \text{abs}(x : \mathbb{R}) := \begin{cases} 
  x & \text{if } x \geq 0 \\
  -x & \text{if } x < 0
  \end{cases}
  \]

  – conditions must be \textit{exclusive} and \textit{exhaustive}
    we do not want to require on \textit{order} to determine which applies
  – there is a \textit{better} way to do this in many cases...
Pattern Matching

• Can also define functions by “pattern matching”

```
define double(0) := 0
define double(n+1) := double(n) + 2  for all n \in \mathbb{N}
```

– first case matches only 0
– second case matches numbers 1 more than some $n \in \mathbb{N}$ ...

\[
\begin{align*}
double(6) &= double(5+1) \text{ so it matches with } n = 5 \\
&= double(4+1) \text{ so it matches with } n = 4 \\
&= \ldots \\
&= double(0) + 2 \text{ for } n = 0 \\
&= 0 + 2 = 2
\end{align*}
\]

- pattern “n+2” would match 2, 3, 4, ...

• Simplifies the math in multiple ways...
Pattern Matching on Natural Numbers

• Pattern matching definition

\[
\text{func double(0) := 0}\\
double(n+1) := double(n) + 2 \quad \text{for any } n : \mathbb{N}
\]

is simpler than using side conditions

\[
\text{func double(n) := 0} \quad \text{if } n = 0 \quad \text{for any } n : \mathbb{N}\\
double(n) := double(n-1) + 2 \quad \text{if } n > 0 \quad \text{for any } n : \mathbb{N}
\]

– e.g., need to explain why \(\text{double}(n-1)\) is legal
   easy in this case, but it gets harder
   – (also makes the reasoning easier, as we will see later...)

• We will prefer pattern matching \textit{whenever possible}
• Booleans have only two legal values: T and F

• Can pattern match just by listing the values:

```
func not(T) := F
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

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

  \[\text{func} \ \text{change}({n : n, f : T}) := n \quad \text{for any } n : \mathbb{N}\]
  \[\text{change}({n : n, f : F}) := -n \quad \text{for any } n : \mathbb{N}\]

  – clear that the rules are exclusive and exhaustive

• Can match on multiple parameters
  – e.g., \[\text{change}({n : m+5, f : T}) := 2m \quad \text{for any } m : \mathbb{N}\]
  – 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

```typescript
// 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

\[
\text{func} \quad \text{double}(0) \quad := 0 \\
\quad \text{double}(n+1) := \text{double}(n) + 2 \quad \text{for any } n : \mathbb{N}
\]

- Also need to be careful with natural numbers

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

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

$$\begin{align*}
\text{func} & \quad \text{double}(0) := 0 \\
& \quad \text{double}(n+1) := \text{double}(n) + 2 \quad \text{for any } n : \mathbb{N}
\end{align*}$$

- This implementation returns the same thing:

```typescript
// m is non-negative
const double = (m: bigint) => {
  return 2n * m;
};
```

- but that’s not what the spec says!
  - spec is imperative but this is a different implementation
- requires reasoning tools to check that this is correct
  - (will see in post-HW Levels)
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testing
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*
Unit vs Integration Tests

• A unit test checks one component
  – ideally, without testing anything else (not always possible)

• You will be expected to write unit tests in industry

• There are also integration tests and end-to-end tests
  – someone will write them, but maybe not you

• We will focus on unit testing in this course
“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 some code (UI especially), manual is still easier
  – if written tests are 3x as hard to create,
    then you’re better off unless you change it 3+ times
  – for UI, written tests aren’t perfect anyway
    need to see it in the browser to be sure that it looks right
Writing a Test

1. Choose an input / configuration
   – description of the inputs / configuration is the “test case”

2. **Think** through what the answer should be
   – if you run the code to get the answer, you are not testing

3. Write code that
   – calls the function that input
   – compares the actual answer to the expected one
   – useful libraries that do this
     we will use “mocha” in JS / TS
Key Idea

• **Key question is what cases to test**
  – small # inputs: we can test all of them
  – otherwise, we cannot

• **Set of inputs can be infinite, but the code is finite**
  – **key idea**: use knowledge of the code structure to test it
Key Idea

• Key question is what cases to test
  – small # inputs: we can test all of them
  – otherwise, we cannot

• Split the allowed inputs into subdomains
  – for inputs in one subdomain, code “does the same thing”

• Hope: code is entirely right or wrong for subdomain
  – one example in the subdomain will tell us if there is a bug
  – (note: this is not always true... see sec and HW Levels)

• Plan: Look at the code. See when it “does the same thing”
// 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?
// Returns true iff n is a prime number
const isPrime = (n: bigint): boolean => {
    if (n < 100n) {
        return PRIME_CACHE[n]; // precomputed answers
    } else {
        let k = 2n;
        while (k*k <= n) {
            if (n % k === 0n)
                return false;
            k = k + 1n;
        }
        return true;
    }
};
// Returns true iff n is a prime number
const isPrime = (n: bigint): boolean => {
    if (n < 100n) {
        return PRIME_CACHE[n];
    } else {
        ...
    }
}

• Cases 2 ... 100 are just table lookups!
Primary Heuristic: Clear-Box Testing

• We need to look at the code to know what to test
  – this 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 tests are also good
  – but other programmers may not use the same rules
Testing Straight-Line Code

Straight-line Code looks like

```
return 2 * (n - 1n) + 1n;
```

Or, more generally, like this

```
const m = n - 1n;
return 2n * m + 1n;
```

- Any number of constant values allowed
  - often makes the code easier to read, but no different

- Inputs where it executes the same straight-line code are “doing the same thing”
Testing Straight-Line Code

**Rule:** same straight-line code is one subdomain

Straight-line Code looks like

```c
return 2n * (n - 1n) + 1n;
```

Or, more generally, like this

```c
const m = n - 1n;
return 2n * m + 1n;
```
Testing Subdomains

**Rule:** at least two test cases per subdomain

(assuming subdomain contains at least two inputs)

- Real bugs will be missed if only one test case
  - common bug: copy-and-paste a stub and forget to change it!
  - will happen to you eventually if you don’t test two cases

- Still doesn’t guarantee the code is right! (see HW Levels)

- More is obviously also okay
  - not a contest to write the fewest tests
Testing Function Calls

In general, function calls are still straight-line code

```javascript
const m = n - 1n;
return Math.sin(2n * m + 1n);
```

• All inputs are still are “the same”
  – two cases is still enough
  – not your job to test the other function!
    test each function on its own assuming the others work
    exception: recursive calls (more later)
Testing Conditionals

Conditionals look like this

```java
if (n > 0n) {
    return 2n * (n - 1n) + 1n;
} else {
    return 0n;
}
```

Two branches ("then" and "else")

- in this case, both branches are straight-line code
- note: the ternary operator creates branches
Testing Conditionals

**Rule**: branches are in separate subdomains

- Would be **negligent** not to test both branches
- If both are straight-line code, then 4 tests
- With if/else if/else, we’d need 6 tests
  - 3 branches x 2 per straight-line block = 6 cases
Other Heuristics

Some other heuristics are also useful

- **Boundary Cases**: if 10 and 11 are separated, then make sure you test 10 and 11
  - easy to have “off by one” bugs
  - happens if you use “< n” instead of “<= n”
    behavior changes between n-1 and n instead
    (see John Carmack’s tweet!)

- Often doesn’t require any more tests
  - can be one of two cases for straight-line code
Testing Conditionals

Conditionals look like this (with \( n \) an integer)

\[
\text{if} \ (n > 0) \ {\text{\{}} \\
\quad \text{return} \ 2n * (n - 1) + 1; \\
\} \ \text{else} \ {\text{\{}} \\
\quad \text{return} \ 0; \\
\} \\
\]

- Boundary cases are 0 and 1
  - cases for “then” block could be 1 and 10 (say)
  - cases for “else” block could be 0 and -1 (say)
Testing Subdomains

Another rule for subdomains with *obvious* boundaries

**Rule**: test each boundary case and at least one non-boundary case

- If there are no boundaries, test two non-boundary cases.
- If there is one boundary, then test it and one non-boundary case.
- If there are two boundaries, then test both and one non-boundary case.
  - e.g., if branch is executed for x between 3 and 10
  - 3 tests are now necessary (e.g., 3, 6, and 10)
Testing Recursion

Recursive calls are more complicated

```javascript
const f = (n: bigint): bigint => {
  if (n >= 2n) {
    return 2n * f(n / 2n) + 1n;
  } else {
    return 0n;
  }
}
```

- Heuristics thus far would allow 0, 1, 2, 3
  - only tests 0 or 1 recursive calls
  - not enough! (see Section Levels)
Testing Recursion

Clear-box Testing for recursive calls:

**Rule:** inputs that cause 0, 1, and 2+ recursive calls are in separate subdomains

- Call this the “0–1–many” heuristic

- Split into 3 subdomains, then apply other rules
  - if subdomains run the same straight-line code, then 6 tests
  - if “0 recursive calls” has two branches, then 8 tests
  - if a subdomain has only one input, then just one test
    e.g., “0” is in its own subdomain, that’s just one test
Testing Recursion

Clear-box Testing for recursive calls:

**Rule:** inputs that cause 0, 1, and 2+ recursive calls are in separate subdomains

- Call this the “0–1–many” heuristic
- Note the savings of using a **library** (part of “tools”)
  - still one subdomain if you call someone else’s function
  - but usually, 5-6 test cases at minimum if you call your own
Summary of Heuristics

• Split into subdomains where code is different
  – branches of conditionals
  – 0, 1, many recursive calls

• At least two tests per subdomain
  (unless subdomain is only 1 input)
  – include all boundaries and a non-boundary

• Not a contest to write the fewest tests!
Summary of Heuristics

• Continue splitting until no more splits needed
  – e.g., two inputs that both make 0 recursive calls BUT are in separate branches... are in separate subdomains

• For “2+ recursive calls”, look at first two calls
  – different paths are split into separate subdomains
  – e.g., same branch on first call but different on second

• Complete summary in the notes on website
What Else?

• We only have rules for:
  – straight-line code
  – conditionals (“if” statements)
  – recursion

• What about everything else?

• Without mutation, this is all we need
  – loops require mutation
Typical Numbers of Tests

• Typical function should have ~10-30 tests

• Should be **embarrassed** if you have <10 tests
  – better have a good explanation in terms of subdomains

• More than 30 starts to feel too time consuming
  – not bad and may be necessary
  – but we will adjust our rules to try to avoid this
    see the later slide...
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, 5 (bottom branch)
Example 2

```javascript
// 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?

- 0, 1, 2 (top) and 3, 6, 9 (middle) and 10, 12 (bottom)
- note that 0 is also a boundary case
Example 3

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

How many tests? Which ones?

- 1 (0 recursive calls)
- 2, 3 (1 recursive call)
- 4, 10 (many recursive calls)
Example 4

```javascript
// 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?

- small # of inputs, so... all of them
Example 5 (Section Levels Q4c)

```javascript
const f = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  }
  else if (n === 1n) {
    return 1n;
  }
  else if (n % 2 === 1n) {  // n is > 1 and odd
    return f(n - 2n) + 1n;
  }
  else {                    // n is > 1 and even
    return f(n - 2n) + 3n;
  }
}

How many tests? Which ones?
```
Example 5 (Section Levels Q4c)

```javascript
const f = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else if (n === 1n) {
    return 1n;
  } else if (n % 2 === 1n) {  // n is > 1 and odd
    return f(n - 2n) + 1n;
  } else {                    // n is > 1 and even
    return f(n - 2n) + 3n;
  }
}
```

0 recursive calls:
Look for base case

How many tests? Which ones?
– 0, 1 (0 recursive calls)
Example 5 (Section Levels Q4c)

```javascript
const f = (n: bigint): bigint => {
    if (n === 0n) {
        return 0n;
    } else if (n === 1n) {
        return 1n;
    } else if (n % 2 === 1n) {  // n is > 1 and odd
        return f(n - 2n) + 1n;
    } else { // n is > 1 and even
        return f(n - 2n) + 3n;
    }
}
```

How many tests? Which ones?
- 0, 1 (0 recursive calls)
- 2, 3 (1 recursive call)

1 recursive call:
For each recursive branch, determine which base case(s) it can enter (this is a subdomain)
Example 5 (Section Levels Q4c)

```javascript
const f = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else if (n === 1n) {
    return 1n;
  } else if (n % 2 === 1n) {  // n is > 1 and odd
    return f(n - 2n) + 1n;
  } else {                     // n is > 1 and even
    return f(n - 2n) + 3n;
  }
}
```

How many tests? Which ones?
- 0, 1 (0 recursive calls)
- 2, 3 (1 recursive call)
- 4, 5, 10, 11 (many recursive calls)

many recursive calls:
For each recursive branch, determine which recursive branch(es) it can enter (this is a subdomain)
Escape Hatches

• If the previous rules give >20 tests...

  1. Look only at the outermost recursive call (rather than the outmost *two* calls)

• If there are still >20 tests...

  2. Pick only one example per subdomain (shudders)

• If there are still >20 tests...

  3. Maybe rewrite your function to be simpler?
Not required 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 \( x > 0 \)”