CSE 331
Software Design & Implementation

Winter 2025

Section 4 - Specifications

Administrivia

- HW 4 released later today, due Wednesday Feb. 5th.

Review – Specifications

- **Imperative** specification says how to calculate the answer
  - lays out the exact steps to perform to get the answer
  - just have to translate math to typescript
  - ex: Absolute value: $|x| = x$ if $x \geq 0$ and $-x$ otherwise
  -
- **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
  - ex: Subtraction $(a - b)$: return x such that $b + x = a$

Review – Math Notation

Standard notations	$\mathbb{N}$	all non-negative integers (“natural” numbers)
	$\mathbb{Z}$	all integers
	$\mathbb{R}$	all real numbers
Made up for this class	$\mathbb{B}$	the boolean values (T and F)
	$\mathbb{S}$	any character
	$\mathbb{S}^*$	any sequence of characters (“strings”)

- **Union**: $A \cup B$ set including everything in A *and* B
- **Tuple**: $A \times B$ all pairs (a, b) where a $\in A$ and b $\in B$
- **Record**: $\{x\colon A, y\colon B\}$ all records with fields x, y of types A, B

Review – Math Notation

- **Side Conditions**: limiting / specifying input in right column
  - ex: $\text{abs} : \mathbb{R} \to \mathbb{R}$

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

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

  - conditions must be **<u>exclusive</u>** and **<u>exhaustive</u>**
- **Pattern Matching**: defining function based on input cases
  - Exactly **one** rule for every valid input

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

  func $f(0) := 0$

  $f(n+1) := n$

  - "n + 1" signifies that input must be > 0 since smallest $\mathbb{N}$ would be 0
  - Preferred over side conditions in most cases

- Course Website > Topics > Math Notation Notes

Question 1

Consider the following function, which calculates half when given an **even** number but also accepts other inputs (though it doesn't perform the same behavior in those cases):

$$\text{half} : (\text{undefined} \ \cup \ \mathbb{N}) \to \mathbb{Z}$$

$$\begin{aligned}
&\text{half}(\text{undefined}) && := 0 && \\
&\text{half}(n) && := n/2 && \text{if } n \text{ is even} \\
&\text{half}(n) && := -(n+1)/2 && \text{if } n \text{ is odd}
\end{aligned}$$

a) What would the declaration of this function look like in TypeScript based on the type?

b) What would the implementation of the body of this function look like in TypeScript?

Question 2

Consider the following TypeScript code:

```
const maybeDouble = (t: {b: boolean, v: [boolean, bigint]}): bigint => {
    const [bool, num] = t.v;
    if (t.b) {
        if (bool) {
            return 2n * num;
        } else {
            return num;
        }
    } else {
        return 0n;
    }
};
```

How would you translate this into our math notation using pattern matching?

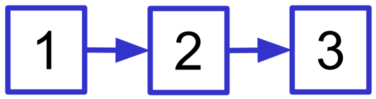
Review – Inductive Data Types

- Describe a set by ways of creating an element of the type
  - Each is a “constructor”
  - Second constructor is recursive
  - Can have any number of parameters

Ex: base case   recursive case

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

nil
3 :: nil
2 :: 3 :: nil
1 :: 2 :: 3 :: nil



Alternative notation:

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

Review – Structural Recursion

- **Inductive types:** builds new values from existing ones
- **Structural recursion:** recurse on smaller parts
  - Call on `n` recurses on `n.val`
  - Guarantees no infinite loops
  - Note: only kind of recursion used for this class

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

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

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

  - Any List is either nil or of the form “cons(x, L)” for some number x and List L (also written as “x:: L”)
  - Cases of function are exclusive and exhaustive based on ⤴️

Question 3

We are asked to write a function "twice" that takes a list as an argument and "returns a list of the same length but with every number in the list multiplied by 2".

a) This is an English definition of the problem, so our first step is to formalize it. Let's start by looking at examples. Fill in the blanks showing the result of applying twice to lists of different lengths.

nil	______________________
3 :: nil	______________________
2 :: 3 :: nil	______________________
1 :: 2 :: 3 :: nil	______________________

b) Now, let's write a formal definition that gives the correct output for **all** lists.

Write a formal definition of twice using recursion. The recursion can be identified by looking for a *pattern* in the answers to part (a).

Question 3

c) What would the implementation of the body of this function look like in TypeScript?

d) What is a set of test inputs that would meet all of our requirements?

Testing

```
describe(‘example’, function() {
    it(‘testBar’ function() {
        /* assert statements */
    })
})
```

- Use assertions to compare expected and actual output for each test case
  - `assert.deepStrictEqual(expected, actual);` should be used generally
- Keep your tests simple! Don’t want to have to write tests for your tests
- **Note:** Please do not submit commented out test cases to gradescope. The course staff will not count those as valid test cases. It is better to submit failing test cases than commented out test cases.

Testing – Documenting

- Document which subdomain you are testing. A justification: heuristic used, part of code it tests.

Name of class being tested

Ex:

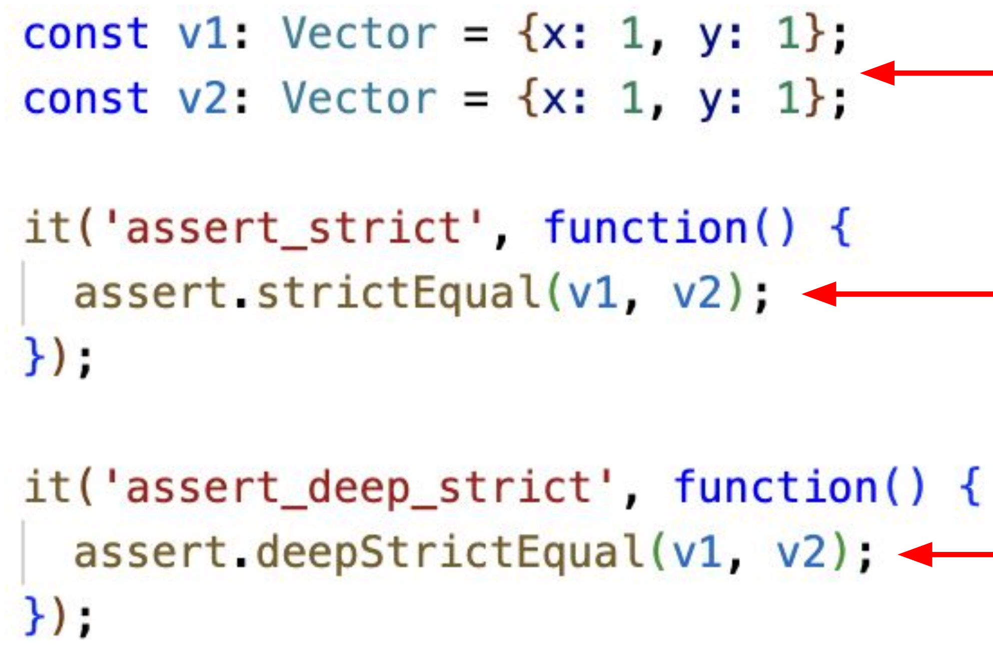
```
describe(‘example’, function() {
```

Name of test (can be function being tested)

```
     it(‘testBar’ function() {
    /* comment describing subdomain being tested */
    assert...
     })
})
```

Testing – Strict vs Deep

Assertion	**Failure Condition**
`assert.strictEqual(expected, actual)`	expected !== actual
`assert.deepStrictEqual(expected, actual)`	values/types of child objects are not equal



```
const v1: Vector = {x: 1, y: 1};
const v2: Vector = {x: 1, y: 1};

it('assert_strict', function() {
  assert.strictEqual(v1, v2);
});

it('assert_deep_strict', function() {
  assert.deepStrictEqual(v1, v2);
});
```

two different objects, but same record values

this will fail

this will pass

Question 4

a) For the following function, what is a set of test inputs that would meet all of our requirements?

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

Question 4

b) The following function allows only non-negative inputs. What is a set of test inputs that would meet all of our requirements?

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

Question 4

c) The following function claims to calculate $|x|$:

```
const abs_value = (x: bigint): bigint => {
  if (x > 1n) {
    return x;
  } else {
    return -x;
  }
};
```

Testing it on the inputs 2 and -2 would meet our requirements, but it would not identify the bug.

Which input do we need to test to see the bug? Which if our non-required (but recommended) heuristics would have found this?

Question 5

We are asked to write a function that takes a list as an argument and "returns a list of the same length but with *every other* number in the list, *starting with the first number*, multiplied by 2".

The first number in the list is at index 0, which is even; the second number in the list is at index 1, which is odd; the third number in the list is at index 2, which is even; and so on. Hence, we will call this function twice-evens because it multiples the numbers at even indexes by two and leaves those at odd indexes unchanged.

a) The definition of the problem was in English, so our first step is to formalize it. Let's start by writing this out in more detail. Fill in the blanks showing the result of applying twice-even to lists of different lengths.

nil	______________________
4 :: nil	______________________
3 :: 4 :: nil	______________________
2 :: 3 :: 4 :: nil	______________________
1 :: 2 :: 3 :: 4 :: nil	______________________

Question 5

b) Now, let's write a formal definition that gives the correct output for **all** lists.

Write a formal definition of twice-evens using recursion. The recursion can be identified by looking for a *pattern* in the answers to part (a). If the answer for one input does not appear related to and the one immediately before it, it could be related to an even *earlier* answer.

c) What would the implementation of the body of this function look like in TypeScript?

d) What is a set of test inputs that would meet all of our requirements?