CSE 331
Software Design & Implementation

Spring 2025

Section 4 - Specifications

Worksheet up front or on the website!!

Administrivia

- HW 4 released later today, due Wednesday April 30th.

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

Made up for this class

$\mathbb{N}$	all non-negative integers ("natural" numbers)
$\mathbb{Z}$	all integers
$\mathbb{R}$	all real numbers
$\mathbb{B}$	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: abs : $\mathbb{R} \rightarrow \mathbb{R}$

    abs(x) := x if $x \geq 0$

    abs(x) := –x if $x < 0$
  - conditions must be **exclusive** and **exhaustive**
- **Pattern Matching**: defining function based on input cases
  - Exactly **one** rule for every valid input

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

  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 ….

Is this a Valid Example of using side conditions? Talk with the person next to you

- $\text{Func} : \mathbb{R} \rightarrow \mathbb{R}$

  $\text{Func}(x) := x \text{ if } x < -10$

  $\text{Func}(x) := -x \text{ if } x > -10$

  $\text{Func}(x) := x*x \text{ if } x > 0$

- yes (why?)
- no (why?)

Question ….

Is this a valid example of using side conditions? Talk with the person next to you

- $\text{Func} : \mathbb{R} \rightarrow \mathbb{R}$

  $\text{Func}(x) := x \quad \text{if } x < -10$

  $\text{Func}(x) := -x \quad \text{if } x > -10$

  $\text{Func}(x) := x*x \quad \text{if } x > 0$

- yes (why?)
- no (why?)

  - The conditions are not exhaustive or exclusive. What happens if the input is equal to -10? What about when it is greater than 0?

Question ….

What type is each of these specs?:

- Steps to calculate the square of a number n: multiply n by itself, n * n.
  - Declarative
  - Imperative

- Return a number, such that this number is the same value as the square of n
  - Declarative
  - Imperative

Question ….

What type is each of these specs?:

- Steps to calculate the square of a number n: multiply n by itself, n * n.
  - Declarative
  - **Imperative**

- Return a number, such that this number is the same value as the square of n
  - Declarative
  - Imperative

Question ….

What type is each of these specs?:

- Steps to calculate the square of a number n: multiply n by itself, n * n.
  - Declarative
  - **Imperative**

- Return a number, such that this number is the same value as the square of n
  - **Declarative**
  - Imperative

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):

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

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

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

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):

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

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

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

```
const half = ( n : undefined | bigint): bigint => { .. };
```

Question 1

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

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

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

```
const half = (n: undefined | bigint): bigint => {
  if (__________________) {

    ____________________
  } else if (_____________) {

    _______________
  } else {

  }
};
```

Question 1

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

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

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

```
const half = (n: undefined | bigint): bigint => {
  if (n === undefined) {

    ____________________
  } else if (_______________) {

    ________________
  } else {

  }
};
```

Question 1

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

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

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

```
const half = (n: undefined | bigint): bigint => {
  if (n === undefined) {
    return 0n;
  } else if (_____________) {

    _____________
  } else {

  }
};
```

Question 1

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

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

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

```
const half = (n: undefined | bigint): bigint => {
  if (n === undefined) {
    return 0n;
  } else if (n % 2n === 0n) {
    _____________
  } else {

  }
};
```

Question 1

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

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

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

```
const half = (n: undefined | bigint): bigint => {
  if (n === undefined) {
    return 0n;
  } else if (n % 2n === 0n) {
    return n / 2n;
  } else {

  }
};
```

Question 1

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

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

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

```
const half = (n: undefined | bigint): bigint => {
  if (n === undefined) {
    return 0n;
  } else if (n % 2n === 0n) {
    return n / 2n;
  } else {
    return -(n + 1n)/2n;
  }
};
```

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?

How many cases will we have?

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?

How many cases will we have?

We will have 3!

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?

How can we declare the function in math notation?

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?

How can we declare the function in math notation?

$$\mathbf{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$$

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?

$$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$$

What are the conditions for the 3 cases?

{b: T, v: (T, n)} {b: F, v: (d, n)} {b: F, v: (F, n)}

{b: F, v: (T, n)} {b: T, v: (F, n)} {b: d, v: (F, n)}

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?

$$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$$

What are the conditions for the 3 cases?

{b: T, v: (T, n)} **{b: F, v: (d, n)}** {b: F, v: (F, n)}

{b: F, v: (T, n)} **{b: T, v: (F, n)}** {b: d, v: (F, n)}

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?

$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$

What is the result of each condition?

maybeDouble({b: T, v: (T, n)}) :=
maybeDouble({b: T, v: (F, n)}) :=
maybeDouble({b: F, v: (d, n)}) :=

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?

$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$

What is the result of each condition?

$\text{maybeDouble}(\{b: T, v: (T, n)\}) := 2n$
$\text{maybeDouble}(\{b: T, v: (F, n)\}) :=$
$\text{maybeDouble}(\{b: F, v: (d, n)\}) :=$

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?

$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$

What is the result of each condition?

$\text{maybeDouble}(\{b: T, v: (T, n)\}) := 2n$
$\text{maybeDouble}(\{b: T, v: (F, n)\}) := n$
$\text{maybeDouble}(\{b: F, v: (d, n)\}) :=$

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?

$\text{maybeDouble} : \{b : \mathbb{B}, v : \mathbb{B} \times \mathbb{Z}\} \to \mathbb{Z}$

What is the result of each condition?

$\text{maybeDouble}(\{b: T, v: (T, n)\}) := 2n$
$\text{maybeDouble}(\{b: T, v: (F, n)\}) := n$
$\text{maybeDouble}(\{b: F, v: (d, n)\}) := 0$

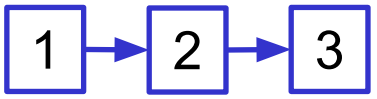
Review – Inductive Data Types

- 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

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

Which of the following is an example of an Inductive Data Type?

- type Bool := true | false
- type Tree := nil | bigint | node(left: Tree, val: ℤ, right: Tree)
- type Point := { x: ℝ, y: ℝ, z: ℝ}
- type ABC := 'a' | 'b' | 'c'

Question ….

Which of the following is an example of an Inductive Data Type?

- type Bool := true | false
- **type Tree := nil | bigint | node(left: Tree, val: ℤ, right: Tree)**
- type Point := { x: ℝ, y: ℝ, z: ℝ}
- type ABC := 'a' | 'b' | 'c'

Question ….

Which of the following is an example of structural recursion?

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

1)

sum : List → ℕ

sum(nil) := 0
sum(x::L) := x + sum(L)

2)

total: (List, x: ℤ) → ℕ

total(nil, 10) := 10
total(x::L, 5) := x * 5

Question ….

Which of the following is an example of structural recursion?

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

1)

sum : List → ℕ

sum(nil) := 0
sum(x::L) := x + sum(L)

2)

total: (List, x: ℤ) → ℕ

total(nil, 10) := 10
total(x::L, 5) := x * 5

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	_____

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	nil
3 :: nil	6 :: nil
2 :: 3 :: nil	4 :: 6 :: nil
1 :: 2 :: 3 :: nil	2 :: 4 :: 6 :: nil

Question 3

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).

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

Question 3

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).

$$\text{twice} : \text{List} \rightarrow \text{List}$$

$$\begin{aligned} \text{twice}(\text{nil}) &:= \text{nil} \\ \text{twice}(a :: L) &:= 2a :: \text{twice}(L) \end{aligned}$$

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

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Testing

If a function has more than 10 allowed inputs, the tests must meet these requirements:

- At least two tests
- Statement Coverage: every statement reachable by some allowed input is reached by some test
- Branch Coverage: every conditional with two reachable branches has both branches tested
- Loop/Recursion Coverage: every loop/Recursion must be tested to execute:
  - 0 times
  - 1 time
  - Many times/calls
- Boundary test when possible(optional)

Notes on Testing Requirements

Question 3

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

How can we ensure statement coverage?

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure statement coverage?

- [] executes first return statement.

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure statement coverage?

- [] executes first return statement.
- [3] executes second return statement.

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure branch coverage?

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure branch coverage?

- The code has an if-else statement. we need all of its branches to be executed.

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure branch coverage?

- The code has an if-else statement. we need all of its branches to be executed.
- The 2 inputs [], [3] already execute both branches, so we already have branch coverage

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure loop/recursive coverage?

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure loop/recursive coverage?

- This function is recursive, so we must consider this heuristic

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure loop/recursive coverage?

- This function is recursive, so we must consider this heuristic
- we need to to cover 0 calls, 1 call, and many calls.

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure loop/recursive coverage?

- This function is recursive, so we must consider this heuristic
- we need to to cover 0 calls, 1 call, and many calls.
- [] makes 0 calls. [3] makes 1 call.

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure loop/recursive coverage?

- This function is recursive, so we must consider this heuristic
- we need to to cover 0 calls, 1 call, and many calls.
- [] makes 0 calls. [3] makes 1 call.

Question:

- What's an example input that can satisfy “many” recursive calls?

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
};
```

Question 3

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

How can we ensure loop/recursive coverage?

- This function is recursive, so we must consider this heuristic
- we need to to cover 0 calls, 1 call, and many calls.
- [] makes 0 calls. [3] makes 1 call.

Question:

- What's an example input that can satisfy “many” recursive calls?

Example answer: [1,2,3]

- this triggers 3 calls. (FYI, there can be more than one answer)

Question 3

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

Hence, one acceptable set of tests inputs is: [], [3], and [1, 2, 3].

Testing Notation

```
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.

Ex:

Name of class being tested

```
describe('example', function() {

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

Name of test (can be function being tested)

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

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;
  }
}
```

Statement coverage requires at least three cases. The input $(0, 0)$ executes the first return statement, the input $(0, -1)$ executes the second return statement, and the input $(-1, 0)$ executes the third return statement. Thus, the three inputs $(0, 0)$, $(0, -1)$ and $(-1, 0)$ give us statement coverage.

Since there are two `if` statements, we also need to ensure branch coverage. The first and third tests, together, execute both branches of the first `if` statement, and the first and second tests, together, exercise both branches of the second (inner) `if` statement. Thus, these three tests already have branch overage.

Finally, this code has no loops or recursion, so loop coverage is (vacuously) satisfied.

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

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;
  }
};
```

The input 0 exercises the first return statement, the input 1 exercises the second return statement, the input 3 exercises the third return statement, and the input 2 exercises the fourth return statement. Thus, the test inputs $0, 1, 2, 3$ give us statement coverage.

The code above has an `if` statement with 4 branches[1]. The four inputs above execute all four branches, so they give us branch coverage as well.

Finally, this function is recursive, so we also need to consider loop coverage. The inputs 0 and 1 make 0 recursive calls and the inputs 2 and 3 make 1 recursive call. If we add the input 4, which makes 2 recursive calls, then we would achieve loop coverage. Thus, the five tests $0, 1, 2, 3, 4$ meet all of our requirements.

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 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?

The bug appears only on input 1. The function returns -1 instead of $|1| = 1$.

The boundary testing heuristic would have told us to test 1.

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

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	nil
4 :: nil	8 :: nil
3 :: 4 :: nil	6 :: 4 :: nil
2 :: 3 :: 4 :: nil	4 :: 3 :: 8 :: nil
1 :: 2 :: 3 :: 4 :: nil	2 :: 2 :: 6 :: 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?

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.

$$\text{twice-evens} : \text{List} \rightarrow \text{List}$$

$$\begin{aligned}
\text{twice-evens}(\text{nil}) &:= \text{nil} \\
\text{twice-evens}(x :: \text{nil}) &:= 2x :: \text{nil} \\
\text{twice-evens}(x :: y :: L) &:= 2x :: y :: \text{twice-evens}(L)
\end{aligned}$$

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

```
const twice_evens = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else if (L.tl.kind === "nil") {
    return cons(2 * L.hd, nil);
  } else {
    return cons(2 * L.hd, cons(L.tl.hd, twice_evens(L.tl.tl)));
  }
};
```

Question 5

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

Question 5

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

The three inputs [], $[1]$, $[1, 2]$ would execute all statements and branches. However, since there is recursion, we must also consider loop coverage. The first two inputs make 0 recursive calls and the last input makes 1 recursive call. Adding the input $[1, 2, 3, 4]$, which makes 2 recursive calls, gives us a total of 4 test inputs and meets our requirements.