MIT
6.031 Software Construction
Prof. Rob Miller and Max Goldman

rcm

6.031 Fall 2021 Quiz 2

You have **50** minutes to complete this quiz. There are **5** problems, each worth approximately an equal share of points.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes on paper. You may not open or use anything else on your computer: no 6.031 website or readings; no VS Code, TypeScript compiler, or programming tools; no web search or discussion with other people.

Before you begin: you must enter a *check-in code*, provided when you arrive, in the green box above. By checking in, you indicate that you will take this quiz under closed-book conditions, and you will not discuss this quiz with anyone other than 6.031 staff members until the solutions are officially released.

To leave the quiz early: click the *done* button at the very bottom of the page and show your screen with the provided *check-out* code to a staff member.

This page automatically saves your answers as you work. Saved answers are marked with a green cloud-with-up-arrow icon. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz. There is no *save* or *submit* button.

Only the answers you write in boxes or select from choices will be graded. Purple margin notes are for your use only during the quiz, and will not be seen by the graders. To ask a question, please raise your hand.

Good luck!

The questions in this quiz use **`NatSet`** in the provided code to represent sets of natural numbers (i.e., positive integers). You can read the code at the bottom of this page, or **open the provided code in a separate tab**.

Problem 1.

(a) Fill in the TODO parts of the `Element` variant:

```
class Element implements NatSet {
    /**
     * @param x  TODO
     */
    public constructor(private readonly x: number) { }
    /** @inheritdoc (i.e., @returns true if and only if x is an element of this set. ) */
    public has(x: number): boolean {
        /* TODO */
    }
}
```

Precondition of the `Element` constructor:

x is a natural number (or x is a positive integer, or x is an integer > 0)

Reminder: margin notes are for your use only, graders will not see them.

Body of the `Element.has()` operation:

return x === this.x;

(b) Fill out the TODO parts of the `Union` variant:

```
/** Represents the union of two sets of natural numbers. */
class Union implements NatSet {
    private readonly children: /* TODO */;
    // Rep invariant: TODO
    public constructor(left: NatSet, right: NatSet) {
        /* TODO */
    }
}
```

Static type of `children`:

Array<NatSet> (other answers possible)

Rep invariant:

children.length === 2 (other answers possible)

Body of the `Union` constructor:

this.children = [left, right]; /* other answers possible */

(c) Using only the variants provided in the code at the end of the quiz, write one line of TypeScript code that creates a value of `NatSet` that represents the empty set.

new Intersection(new Element(1), new Element(2))

(d) What is the datatype definition for `NatSet`?

`NatSet =`

Intersection(left:NatSet, right:NatSet) + Union(children:Array<NatSet>) + Element(x:number)

Problem ⋊2. Now extend the `NatSet` type with a new variant:

```
/** Represents a range of natural numbers. */
class Range implements NatSet {
    /** constructor spec omitted */
    public constructor(private readonly n: number, private readonly m: number) {
    }
    /** @inheritdoc (i.e., @returns true if and only if x is an element of this set. ) */
    public has(x: number): boolean {
        return x >= this.n && x < this.m;
    }
}
```

(⋊a) Write an abstraction function and rep invariant for `Range`. (You may assume the omitted constructor spec is whatever you need it to be.)

Abstraction function

AF(n, m) = { x | x is a natural number in the interval [n,m) }

Rep invariant

true (other answers possible)

(⋊b) Ben Bitdiddle complains that `Range` doesn't follow the spec of `NatSet`, and offers this example:

```
const set: NatSet = new Range(1, 10);
set.has(3.3)
```

In one sentence, explain why `set.has(3.3)` violates the spec.

set.has(3.3) returns true, which it should only do if 3.3 is a member of the set {1, 2, 3, 4, 5, 6, 7, 8, 9}, but 3.3 is not a natural number so it is cannot be a member of the set.

In one sentence, suggest a specific way to fix the problem that *would not* require changing any existing clients of `NatSet`.

In the body of Range.has(), test whether x is a natural number, and immediately return false if it isn't.

In one sentence, suggest a specific way to fix the problem that *would* require reviewing and possibly changing existing clients of `NatSet`.

Change the precondition of NatSet.has() to require x to be a natural number.

Problem ⋊3. This problem extends the original `NatSet` type with a variant that *filters* a set of natural numbers using a client-provided filter function `f`:

```
/** Represents the subset of a set of natural numbers that satisfies a filter function. */
class Filter implements NatSet {
    /** @param f TODO */
    public constructor(
        private readonly f: TODO,
        private readonly child: NatSet
    ) { }
    /** @inheritdoc (i.e., @returns true if and only if x is an element of this set. ) */
    public has(x: number): boolean {
        return this.child.has(x) && this.f(x);
    }
    // other parts of variant omitted
}
```

(⋊a) Write the static type of `f`.

```
        private readonly f:
```

(x: number) => boolean

(⋊b) Suppose `ALL` is a constant of type `NatSet` representing all positive integers. Write one line of TypeScript code that represents all even positive integers.

```
const EVENS: NatSet =
```

new Filter((x: number) => x % 2 === 0, ALL)

(⋊c) Choosing from among the labeled statements below, give the *weakest possible* specification for `f` that works for `Filter`.

A. input is a natural number
B. input is a number
C. output is a boolean
D. output is false if input is not a natural number
E. output is true if input is a natural number

Use the letters A-E in your answers.

precondition

A /* the weakest spec is the one with the strongest precondition */

postcondition

C /* the weakest spec is the one with the weakest postcondition */

Problem ⤖4. In order to accommodate sets that may take some time to compute – for example, a set of prime numbers – this problem changes the original `NatSet` type so that its `has` operation is asynchronous:

```
/** Immutable type representing a set of natural numbers (positive integers). */
interface NatSet {
    /** @returns true if and only if x is an element of this set. */
    async has(x: number): TODO;
}
```

(⤖a) Write the return type for `has()`.

```
Promise<boolean>
```

(⤖b) Implement the body of `Intersection.has()` so that it is *as concurrent as possible*. (Assume TODO is filled in with the appropriate return type from the previous question.)

```
class Intersection implements NatSet {
    public constructor(
        private readonly left: NatSet,
        private readonly right: NatSet
    ) { }
    public async has(x: number): TODO {
```

```
const leftPromise: Promise<boolean> = left.has(x);
const rightPromise: Promise<boolean> = right.has(x);
return (await leftPromise) && (await rightPromise);
```

```
    }
}
```

(⤖c) Suppose prime numbers are defined by a class that sends requests to a remote web server to determine which numbers are prime:

```
/** Represents the set of prime numbers.
    Uses a remote web server to test numbers for membership in the set. */
class Primes implements NatSet {
    public constructor() { }
    /** @inheritdoc (i.e., @returns true if and only if x is an element of this set. ) */
    public has(x: number): TODO {
        console.log(`asking server about ${x}`);
        // rest of implementation omitted
    }
}
```

For example, `new Primes().has(19)` prints `"asking server about 19"`, makes a web request to ask the remote server whether 19 is prime, and eventually produces `true` as its result.

Alyssa Hacker devises a general way to speed up this code, by defining a new `Cached` variant:

```
/** Represents a set of natural numbers, using a cache to speed up
    repeated calls to has() that use the same number x. */
class Cached implements NatSet {
    private cache: Map<number, boolean>;
    public constructor(private readonly child: NatSet) {
        this.cache = new Map();
    }
```

```
    public async has(x: number): TODO {
        let hasX = this.cache.get(x);
        if (hasX !== undefined) {
1           return hasX;
        } else {
2           hasX = await this.child.has(x);
            this.cache.set(x, hasX);
3           return hasX;
        }
    }
}
```

Using this code, Alyssa runs the example sequence below:

```
const PRIMES: NatSet = new Cached(new Primes());
await PRIMES.has(19); // A
await PRIMES.has(19); // B
PRIMES.has(25); // C
PRIMES.has(25); // D
```

... and discovers from the print statements that the code is making only *one* web request for 19, but *two* web requests for 25:

```
asking server about 19
asking server about 25
asking server about 25
```

Show an interleaving of the code above that explains this behavior, using A, B, C, D to refer to the four labeled calls to `PRIMES.has()`, and the line numbers 1-3 shown for `Cached.has()`.

Every box you fill in below must match the regular expression `[ABCD] reaches [1-3]`. Your answer should explain all four calls (A, B, C, D). There may be more boxes than you need.

A reaches 2

A reaches 3

B reaches 1

C reaches 2

D reaches 2

C reaches 3

D reaches 3

(⤫d) Louis Reasoner writes in a code review: "Wait, `Cached` is doing mutation! I thought NatSets are immutable." In one sentence, say specifically in what sense Louis is right about the mutation.

Cached mutates the Map in its rep by calling this.cache.set().

In one sentence, state in what sense `Cached` is immutable.

The mutation is beneficent -- the Cached object still represents the same abstract set value.

Problem ⤙5. Write a grammar that allows expressing `NatSet` values using conventional set notation. Examples of legal strings that your grammar should accept:

```
{3,19,1}
{5,5}
{}
```

The notation does *not* include other set operations (such as intersection and union).

Your grammar should match all finite sets that can be represented as a `NatSet` value, but it should not match any sets that cannot be represented by `NatSet`.

The root nonterminal of your grammar should be called `set`. Use no more than 3 nonterminals.

```
set ::= '{' values? '}' ;
values ::= number (',' number)* ;
number ::= [1-9][0-9]* ;
```

You can **open the code below in a separate tab**.

```
/** Immutable type representing a set of natural numbers (positive integers). */
interface NatSet {
    /** @returns true if and only if x is an element of this set. */
    has(x: number): boolean;
}

/** Represents a set consisting of a single natural number. */
class Element implements NatSet {
    /**
     * @param x  TODO
     */
    public constructor(private readonly x: number) { }
    /** @inheritdoc (i.e., @returns true if and only if x is an element of this set. ) */
    public has(x: number) {
        /* TODO */
    }
    // AF, RI, SRE omitted
}

/** Represents the intersection of two sets of natural numbers. */
class Intersection implements NatSet {
    public constructor(
        private readonly left: NatSet,
        private readonly right: NatSet
    ) { }
    /** @inheritdoc (i.e., @returns true if and only if x is an element of this set. ) */
    public has(x: number): boolean {
        return this.left.has(x) && this.right.has(x);
    }
    // AF, RI, SRE omitted
}

/** Represents the union of two sets of natural numbers. */
class Union implements NatSet {
    private readonly children: /* TODO */;
    public constructor(left: NatSet, right: NatSet) {
        /* TODO */
    }
    // has(), AF, RI, SRE omitted
}
```