MIT
6.031 Software Construction
Prof. Rob Miller and Max Goldman

6.031 Spring 2022 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, and you may use blank scratch 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 *check in* by having the course staff scan the QR code at the top of the page.

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

This page automatically saves your answers as you work. 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.

Good luck!

The questions on this quiz use the code at the bottom of this page. You can **open this introduction and the provided code in a separate tab**.

In the provided code at the bottom, the type `DNA` represents a strand of DNA, which is a string of bases, where a *base* is represented by one of the letters A, C, G, or T.

One kind of DNA strand is a *gene*, which encodes the description of a protein. Genes can also occur as substrings in other DNA strands.

The provided type `Crispr` represents a gene-editing process. The process starts with *precursors*, with are premade DNA strands bought from a supplier. A typical step of a gene-editing process is a gene *splice*, which replaces a gene in a DNA strand with another gene.

A `Crispr` process can be *simulated* entirely in software. The result of simulation is a `DNA` value in memory.

A `Crispr` process can also be *fabricated* in the real world using a `Lab`, which programmatically controls a gene-editing machine with `Tube` elements in which gene-editing reactions occur. The result of fabrication is a `Tube` containing real DNA.

Note that this is an oversimplification of biology, gene-editing, and CRISPR technology.

Problem ⤭1.

Write the data type definition for the provided `Crispr` type.

```
Crispr = Precursor(dna: DNA) + Splice(target: Crispr, oldGene: Crispr, newGene: Crispr)
```

The government requires that gene-editing processes should never intentionally or accidentally produce DNA containing a dangerous substring (one that might, for example, be a dangerous virus). This requirement must be enforced not just on all precursors and final product of the process, but for intermediate products as well (the result of each completed gene-splicing step).

Alyssa Hacker recommends providing an operation `everOccurs` that takes a DNA strand and returns true if and only if the strand ever occurs as a substring of any DNA strand used or produced during the process.

Specify and implement `everOccurs` as an instance method of `Crispr`.

```
interface Crispr {
    ...
```

```
/**
 * @param dna  strand to search for
 * @returns true if and only if dna ever occurs as a substring of any DNA strand
 * used or produced during this process
 */
everOccurs(dna: DNA): boolean;
```

```
}
```

```
class Precursor implements Crispr {
    ...
```

```
/** @inheritdoc */
public everOccurs(dna: DNA) {
  return this.dna.find(dna) !== undefined;
}
```

```
}
```

```
class Splice implements Crispr {
    ...
```

```
/** @inheritdoc */
public everOccurs(dna: DNA) {
  return this.target.everOccurs(dna)
      || this.oldGene.everOccurs(dna)
      || this.newGene.everOccurs(dna)
      || this.simulate().find(dna) !== undefined;
}
// other answers possible
```

```
}
```

Refactor your implementation of `Splice.everOccurs()` from the previous answer so that it does as much work as possible using the array `[this.target, this.oldGene, this.newGene]` with one or more operations like `map`, `filter`, `reduce`, `every`, `some`. (You may repeat the previous answer unchanged if it already met this requirement.)

```
class Splice implements Crispr {
    ...
```

```
/** @inheritdoc */
public everOccurs(dna: DNA) {
  return [this.target, this.oldGene, this.newGene]
             .some(crispr => crispr.everOccurs(dna))
      || this.simulate().find(dna) !== undefined;
}
// other answers possible
```

```
}
```

Problem ⤮2.

Implement `fabricate()` for `Precursor`.

```
class Precursor implements Crispr {
    ...
    public async fabricate(lab: Lab): Promise<Tube> {
```

```
return lab.get(this.dna);
```

```
    }
}
```

Implement `fabricate()` for `Splice`, using **as much concurrency as you can**.

```
class Splice implements Crispr {
    ...
    public async fabricate(lab: Lab): Promise<Tube> {
```

```
const target = this.target.fabricate(lab);
const oldGene = this.oldGene.fabricate(lab);
const newGene = this.newGene.fabricate(lab);
return lab.splice(await target, await oldGene, await newGene);
// other answers possible
```

```
    }
}
```

Problem ⤮3.

Ben Bitdiddle is working on the `Lab` implementation, as shown in the provided code.

Its `get` operation can be costly in both time and money, requiring a human to load the tube with premade DNA bought from another company.

But a tube of DNA can be reused any number of times, since gene-editing requires only a tiny amount of DNA, and the process regenerates the amount that was used.

So Ben has implemented `Lab.get()` as shown in the provided code, using the `tubeMap` field to keep track of and reuse tubes if possible.

(Ben has also modified `Lab.splice()` so that its target tube is removed from `tubeMap`, because its contents will be mutated by the gene-splicing operation. This code is not shown.)

Write a rep invariant for `tubeMap` which would achieve Ben's goal of reusing tubes as much as possible.

```
- no two DNA values in tubeMap are equalValue()
- for every tube t in tubeMap, t is loaded with DNA corresponding to tubeMap.get(t)
```

Suppose that:

- two different gene-editing processes A and B are running asynchronously using the **same** `Lab`
- A and B both call `lab.get(dnaX)` for the **same** precursor `dnaX`
- no other asynchronous processes are using `lab`

For each of the following interleavings, referring to the line numbers 1-7 in `get()` in the provided code, decide whether the interleaving is impossible, leads to a race condition or deadlock, or runs safely; then explain your answer in one sentence.

A runs lines 1, 4, 5, 6, 7, then B runs lines 1, 4, 5, 6, 7.

- (●) impossible
- ( ) race condition
- ( ) deadlock
- ( ) safe

Why?

After A runs line 5, there will be at least one tube in tubeMap, so B must proceed from line 1 to line 2.

A runs lines 1, 4, 5, 6a; then B runs lines 1, 4, 5, 6a; then A finishes lines 6b and 7, then B finishes lines 6b and 7.

- (●) impossible
- ( ) race condition
- ( ) deadlock
- ( ) safe

Why?

After A runs line 5, there will be at least one tube in tubeMap, so B must proceed from line 1 to line 2.

A runs lines 1 and 4, then B runs lines 1 and 4, then A runs lines 5, 6, 7, then B runs lines 5, 6, 7.

- (●) impossible
- ( ) race condition
- ( ) deadlock
- ( ) safe

Why?

A cannot lose control to B after line 4, it can only lose control at the await in line 6.

A runs lines 1, 4, 5, 6a, then B runs lines 1, 2, 3, then A finishes lines 6b and 7.

- ( ) impossible
- (●) race condition
- ( ) deadlock
- ( ) safe

Why?

B returns the tube that A is loading before the tube has finished loading, violating the postcondition of get().

Problem ⤭4.

Louis is working on the simulation part of the gene-editing process. He pushes a git commit that adds a new method to the `DNA` type (a type that already exists and has many clients throughout the system). The lines marked + are Louis's commit:

```
class DNA {
    ... // existing code of DNA omitted

+    public replace(oldGene: string, newGene: string): void {
+        this.bases = this.bases.replaceAll(oldGene, newGene);
+        this.checkRep();
+    }
}
```

What kind of ADT operation is `replace()`?

mutator

As one of Louis's teammates, you are assigned to code-review Louis's git commit.

First state two different things that are *missing* from Louis's commit – that Louis should have committed but didn't. Put one in each box below.

specification for replace()

tests for replace()

Now critique Louis's method signature in two different ways, using the two boxes below, one sentence in each box.

DNA is an immutable type, so it must not have a mutator.

oldGene and newGene should have type DNA, rather than string, so that static-checking can catch attempts to call replace() with strings that aren't legal DNA strands.

Problem ⤭5.

Write a regular expression describing a nonempty strand of DNA.

```
[ACTG]+
```

Write a regular expression describing a strand of DNA containing the triple CAG somewhere, and also containing the triple TCG somewhere.

```
[ACTG]*(CAG[ACTG]*TCG|TCG[ACTG]*CAG)[ACTG]*
```

A *chromosome* is a nonempty sequence of genes, where each *gene* is a nonempty sequence of codons followed by a *terminator* TTT, where each *codon* is a triple of bases that does not begin with T. Write a grammar expressing these relationships, with a nonterminal named for, and correctly defining, each italicized word.

```
chromosome ::= gene+
gene ::= codon+ terminator
codon ::= [ACG][ACTG][ACTG]
terminator ::= 'TTT'
```

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

```
/**
 * Immutable type representing a strand of DNA.
 */
class DNA {

    /** omitted */
    public constructor(bases: string) {
        // omitted
    }

    /**
     * @returns zero-based index of first occurence of `dna` as a substring of this strand,
     *     or undefined if `dna` never occurs.
     */
    public find(dna: DNA): number|undefined {
        // omitted
    }

    /**
     * @returns true iff this and that are observationally equivalent
     */
    public equalValue(that: DNA): boolean {
        // omitted
    }

    // other code omitted
}

/**
 * Immutable type representing a gene-editing process.
 */
interface Crispr {

    /**
     * Simulates this gene-editing process entirely in software, without using chemicals or a lab.
     * @returns DNA strand that would result from this process
     */
    simulate(): DNA;

    /**
     * Run this gene-editing process using the given `lab`.
     * @returns the tube of `lab` in which the final DNA from this process
     * can be found.
     */
    async fabricate(lab: Lab): Promise<Tube>;

    // other code omitted
}

/**
 * Represents an already-existing DNA strand (a "precursor") in a gene-editing
 * process.  Precursors are bought premade from a supplier.
 */
class Precursor implements Crispr {
```

```
    /**
     * Make a gene-splicing step that results in the given `dna` strand.
     */
    public constructor(private readonly dna: DNA) {
    }

    // other code omitted
}

/**
 * Represents a gene-splicing step in a gene-editing process,
 * which replaces all instances of one gene with another.
 */
class Splice implements Crispr {
    /**
     * Make a gene-splicing step that finds all occurrences of
     * oldGene in target and substitutes newGene in place of each one.
     */
    public constructor(
        private readonly target: Crispr,
        private readonly oldGene: Crispr,
        private readonly newGene: Crispr
    ) {
    }

    // other code omitted
}

/**
 * Mutable type controlling an automated gene-editing machine.
 */
class Lab {

    /**
     * Modifies the DNA in targetTube to replace all occurrences of the DNA from oldGeneTube with the
     * DNA from newGeneTube.
     * @returns a promise that fulfills with the same tube as targetTube, after the process is complet
     */
    public async splice(targetTube: Tube, oldGeneTube: Tube, newGeneTube: Tube): Promise<Tube> {
        // omitted
    }

    private tubeMap: Map<Tube, DNA> = new Map();

    /**
     * @returns a tube containing DNA strands corresponding to `dna`
     */
    public async get(dna: DNA): Promise<Tube> {
        for (const tube of this.tubeMap.keys()) {
            if (this.tubeMap.get(tube).equalValue(dna)) {
                return tube;
            }
        }
        const tube = new Tube();
```

```
5         this.tubeMap.set(tube, dna);
6a/b      await this.load(tube, dna); // "line 6a" is the load() call, "line 6b" is the await
7         return tube;
    }

    /**
     * Ask a human to order premade DNA from a supplier
     * and load it into the tube.
     * @returns a promise that fulfills once this tube contains `dna`.
     */
    private async load(tube: Tube, dna: DNA): Promise<void> {
        // omitted
    }

    // other code omitted
}

/**
 * Mutable type representing a test tube containing DNA.
 */
class Tube {

    /** Make a new Tube. */
    public constructor() {
    }

    // other code omitted
}
```