MIT
6.102 Software Construction
Max Goldman and Prof. Rob Miller

solutions

6.102 Spring 2024 Quiz 2

You have **110** minutes to complete this quiz. There are **7** problems.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes handwritten directly on paper, and you may use blank scratch paper. You may not open or use anything else on your computer: no 6.102 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: enter *done* 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.

If you feel the need to write a note to the grader, you can click the gray pencil icon to the right of the answer.

Good luck!

This quiz is organized into three parts. Parts and problems are of varying length!

- Part I (Problem 1) does not refer to any code.
- Both Part II (Problems 2—4) and Part III (Problems 5—7) refer to code that you can find at the bottom of this page or **open in a separate tab**.

Part I

Problem ⤙1.

The following one-sentence explanations are **incorrect**. For each one, write a replacement for the underlined part to give a correct *one-sentence* explanation of that idea. Your replacement should clearly and succinctly state the most relevant and important facts. Do not use multiple sentences, do not use a run-on sentence; also no semicolons (and no parentheticals).

Example:

In TypeScript, integers and floating point numbers have different static types.

share the same static type “number”

(⤙a) Explaining TypeScript
Compiler option `noUncheckedIndexedAccess`, which we started using on Problem Set 2, makes it a static error to use an index that is out of bounds for an array.

makes the static type of array index access a union of the array element type with undefined

(⤙b) Explaining TypeScript
A variable declared with type `ReadonlyArray` will have an immutable array as its value.

will statically disallow calls to mutators even though an underlying built-in array value will be mutable

(⤙c) Explaining Testing
While we might aim to achieve 100% coverage of reachable lines of code in a function, our tests cannot cover lines inside `catch` blocks.

that only execute when a precondition is violated

(⤙d) Explaining Testing
With the visitor pattern, which we saw in the *Little Languages* reading, we expect the testing strategy for a new immutable visitor implementation will partition on what function that new visitor is computing.

which variant of the accepting type is being visited

(⤙e) Explaining Spec Diagrams (those are diagrams showing specs as regions in the space of all possible functions)
When specifications are incomparable, their regions in the diagram are disjoint.

their regions in the diagram are overlapping or disjoint, but they are not in a subset relation

(⤙f) Explaining Spec Diagrams (those are diagrams showing specs as regions in the space of all possible functions)
If we strengthen a spec’s postcondition, the functions that adhere to those stronger requirements on the outputs are added to its region in the diagram.

are the functions that remain inside the spec's now-smaller region in the diagram

Part II

A ***geographic point*** is a coordinate pair *(latitude, longitude)*, where latitude gives the angle north or south from the equator and longitude gives the angle east or west from the prime meridian.

Latitude ranges from −90° at the south pole to +90° at the north pole. The points at 0° latitude are on the equator. Longitude ranges from −180° to +180°. The points at 0° longitude are on the prime meridian.

Note that some distinct geographic points represent the same physical location, but we are considering only coordinate pairs, **not** physical locations. For example, (90°,0°) and (90°,42.36°) are distinct, un-equal points, even though both are the North Pole.

The shortest-distance segment between two geographic points is an arc along the curved surface of the Earth. For example, the shortest path from Boston to Beijing is an arc that passes close to the North Pole.

A ***route*** is a directed path from one geographic point to another point through zero or more intermediate points. A route may re-visit points along the way (forming loops), or may start and end at the same point if there are points in between, but adjacent points along the route are distinct.

(Code for both Part II and Part III is at the bottom of this page, or you can **open the code in a separate tab**.)

Problem ⤮2.

(⤮a) Create a well-written grammar that parses the `toString` representations produced by `LatLon` (in the provided code). The nonterminal *`point`* should match a single point, and nonterminals *`latitude`* and *`longitude`* should match the latitude and longitude, respectively, such that their text could be parsed into a number with `parseFloat`. *(Write just the grammar, without declaring it inside a TypeScript `string`.)*

```
point ::= '(' latitude ' lat, ' longitude ' lon)' ;
latitude ::= number ;
longitude ::= number ;
number ::= '-'? digit+ ('.' digit+)? ;
digit ::= [0-9] ;
```

(⤮b) Your grammar may very reasonably match some strings that are not valid geographic points, requiring code in *e.g.* a `makeAST` function to perform validation. Is that the case for your grammar?

- If yes, write YES and give an example input that is matched by the grammar but is not a valid geographic point.
- Otherwise, write NO.

YES, e.g. "(200 lat, 200 lon)"

And in one sentence, using principles we have studied, argue for why that is a reasonable design choice:

Validating the numerical ranges in code makes the grammar more ETU, and makes the checks more SFB, ETU, and RFC.

(⤭c) Now extend your grammar to parse the example inputs below, and inputs like them. Define a new root nonterminal *`route`*, and structure your grammar so that it is well-suited for parsing inputs into instances of `Route` using `Segment` and `Combined` (examine those classes in the provided code). Use *`point`*, *`latitude`*, and/or *`longitude`* from above, do not repeat them. Example inputs: *(shown one per line)*

```
[(0 lat, 0 lon)(1 lat, 1 lon)]
[[(0 lat, 0 lon)(1 lat, 1 lon)][(1 lat, 1 lon)(2 lat, 2 lon)]]
[[(0 lat, 0 lon)(1 lat, 1 lon)][(1 lat, 1 lon)(2 lat, 2 lon)][(2 lat, 2 lon)(3 lat, 3 lon)]]
[[(0 lat, 0 lon)(1 lat, 1 lon)][[(1 lat, 1 lon)(2 lat, 2 lon)][(2 lat, 2 lon)(3 lat, 3 lon)]]]
```

```
route ::= '[' (segment | combined) ']' ;
segment ::= point point ;
combined ::= route+ ;
```

Problem ⤮3.

`Segment` represents a shortest-distance directed segment from one geographic point to another, distinct, point.

Remember that the TypeScript type `[ number, number ]` is the type of 2-element `Arrays` where both elements are `numbers`.

(⤮a) Write an excellent rep invariant for `Segment` given the provided code and specs. Assume an appropriate spec and/or assertions in the constructor. Do not implement `checkRep`. Here is the rep as a reminder, but your answer must work with all the provided code and specs:

```
private readonly start: LatLon;
private readonly delta: [ number, number ];
```

RI(start, delta) =
  delta[0] or delta[1] is nonzero
  and -90 <= start.latitude + delta[0] <= 90
  and -180 <= start.longitude + delta[1] <= 180

(⤮b) And write an excellent abstraction function for `Segment`. Make sure it is a function, with the appropriate range and domain.

AF(start, delta) = the shortest-distance arc from start to (start.latitude+delta[0] lat, start.longitude+delta[1] lon)

`Combined` represents a route constructed from a sequence of sub-routes, such that each sub-route starts where the previous sub-route ended.

(⤮c) Write an excellent rep invariant for `Combined` given the provided code and specs. Assume an appropriate spec and/or assertions in the constructor. Do not implement `checkRep`. Here is the rep as a reminder, but your answer must work with all the provided code and specs:

```
private readonly subroutes: Array<Route>;
private readonly first: Route;
```

RI(subroutes, first) =
  subroutes[0] exists and is .equalValue(first)
  and for all 0 < i < subroutes.length, subroutes[i-1].getEnd().equalValue(subroutes[i].getStart())

Problem ⤭4.

We would like to introduce a new interface to represent `Routes` where the details are only retrieved (maybe from a server) or computed (maybe with an expensive algorithm) as needed:

```
interface LazyRoute {
    /** @returns (a promise of) the first point in this route */
    getStart(): Promise<LatLon>;

    // no other operations for now
}
```

Suppose we update `Segment` so that it implements `LazyRoute`, and we write...

(1) a new class:

```
class LazyCombined implements LazyRoute {
    private readonly subrouteIDs: number[];
    private readonly first: Promise<LazyRoute>;

    constructor(private readonly myID: number, subrouteIDs: number[]) {
        this.subrouteIDs = subrouteIDs.slice();
        this.first = fetchRoute(this.subrouteIDs[0] ?? assert.fail());
    }

    async getStart(): Promise<LatLon> {
        console.log(`getStart ${this.myID}`);
        return (await this.first).getStart();
    }
}
```

and (2) a placeholder implementation of `fetchRoute(..)`:

```
async function fetchRoute(id: number): Promise<LazyRoute> {
    console.log(`fetchRoute ${id}`);
    await timeout(100 + Math.random() * 100);
    if (id === 1) { return new LazyCombined(1, [ 2, 3 ]); }
    if (id === 2) { return new Segment(...); }
    if (id === 3) { return new Segment(...); }
    throw new Error(`invalid route ID ${id}`);
}
```

Notice the `console.log` calls in `getStart()` and `fetchRoute(..)`. Enumerate all the possible outputs we might see logged if we run:

```
  const route = await fetchRoute(1);
  const start = await route.getStart();
```

Write one possible output per box. If there are fewer possibilities than boxes, leave extra boxes blank.

fetchRoute 1 fetchRoute 2 getStart 1			

*Example (**not** a correct possible output):*

```
getStart 1
fetchRoute 1
fetchRoute 2
fetchRoute 3
```

Part III

In our ***client/server room reservation system***, the server maintains a database of meeting rooms that users may reserve, perhaps through a web interface.

In the provided code, the immutable type `Room` represents a reservable room with a unique `string` identifier and other, unspecified, properties (which might include things like seating capacity, A/V capabilities, *etc.*).

Time slots for which rooms can be reserved are identified by unique `bigints`. (Just as we aren't considering all the details of rooms, we aren't considering actual dates/times/intervals, just uniquely-identified time slots.)

The type `RoomCalendar` provides operations for finding rooms to reserve and then making those reservations.

(Code for both Part II and Part III is at the bottom of this page, or you can **open the code in a separate tab**.)

Problem ⤭5.

The provided code for `reserve(..)` in `RoomCalendar` does not work. When you call it like this...

```
async function main() {
    const roomcal: RoomCalendar = // ...
    const result = await roomcal.reserve(new Set([ 4242n ]), 'Big Conference Room');
    console.log(result);
}
```

... `result` is always the empty `Set`, even if the reservation is made successfully on the server.

You try to fix the problem by awaiting everything. `await` all the things!

```
    async reserve(timeslotIDs: Set<bigint>, roomID: string): Promise<Set<bigint>> {
        const success = ~~await~~ new Set<bigint>();
        const fn = ~~await~~ (async (timeslot: bigint): Promise<void> => {
            if (await requestReservation(~~await~~ timeslot, ~~await~~ roomID)) {
                ~~await~~ success.add(~~await~~ timeslot);                    // add(…) returns the Set
            }
        });
        ~~await~~ setToArray(~~await~~ timeslotIDs).forEach(~~await~~ fn);     // forEach(…) returns void
        return ~~await~~ success;
    }
```

... but the problem remains, and every single one of those newly-added `await`s has a warning in your editor:
`'await' has no effect on the type of this expression`
So you revert back to the provided code.

(⤭a) Why is `result` always empty? Explain clearly and succinctly, saying exactly what happens with `success`, `result`, and any relevant `Promise`s. Your answer must fit in the box.

When we call reserve, its call to forEach calls fn but does not await fn's execution. While fn is awaiting the result of requestReservation, the call to reserve returns a Promise that immediately resolves to the Set success, which is still empty.

(⤙b) What will you observe if you add `await timeout(2000);` in the `main()` function above, before the `console.log`? Assume the server responds very quickly and the reservation is successful.

What is `result`: Set containing 4242n

(You do not need to say exactly how it is printed, only what its value is.)
Explain why, clearly and succinctly. Say exactly what happens with `success`, `result`, and any relevant `Promises`. Your answer must fit in the box.

Waiting 2 seconds gives the server time to respond, and for execution inside of fn to continue, mutating success. Since result is an alias to success, we observe the mutated Set.

(⤙c) What will you observe if you both:

- add `await timeout(2000);` in the `main()` function above, before the `console.log`
- and change the last line of `reserve` to be: `return new Set(success);`

Again assume the server responds very quickly and the reservation is successful.

What is `result`: empty Set

Explain why, clearly and succinctly. Say exactly what happens with `success`, `result`, and any relevant `Promises`. Your answer must fit in the box.

There is time for fn to mutate the Set success as before, but result is no longer an alias to that Set. Instead, reserve has made a copy of success when it returned, without awaiting fn and before we waited 2 seconds, when success was still empty.

Problem ⤙6.

Fix the problem by replacing this existing line in `reserve`:

```
setToArray(timeslotIDs).forEach(fn);
```

... with exactly one new line of code that uses some of these tools:

- the provided `setToArray` helper function
- `Array` methods `forEach`, `map`, `filter`, or `reduce`
- functions `Promise.any`, `all`, or `race`, or method `then`

The replacement code: *(For partial credit, replace that single line with multiple new lines, at most three.)*

```
await Promise.all(setToArray(timeslotIDs).map(fn));
```

Problem ⤭7.

On the server side of the room reservation system, suppose we have:

```
/** TODO */
class Server {

    // ... possibly other fields or methods ...

    /** TODO */
    async markReserved(timeslotID: bigint, roomID: string): Promise<boolean> {
        // reserved timeslot IDs for the room are stored in a text file, one ID per line, no duplicate
        const filename = `rooms/${roomID}.txt`;
        const roomFile = await fs.promises.readFile(filename, { encoding: 'utf-8' });
        const timeslotString = timeslotID.toString();
        for (const line of roomFile.split('\n')) {
            if (line === timeslotString) {
                return false;
            }
        }
        await fs.promises.appendFile(filename, timeslotString + '\n');
        return true; // caller has reserved roomID at timeslotID
    }
}
```

(⤭a) The specs for `Server` and `markReserved` are not provided, but there is definitely a race condition here. In one sentence, what reasonable and expected part of their spec (*e.g.* a postcondition or invariant) can be violated when `markReserved` runs concurrently?

Only one caller can reserve a given room for a given timeslot.

(⤭b) Give a pair of calls to `markReserved` that can exhibit the race condition:

1. `markReserved(42n, 'Earth');`

2. `markReserved(42n, 'Earth');`

(⤭c) And explain clearly and succinctly a sequence of events for those calls that demonstrate the race condition. Your answer must fit in the box.

Suppose the reservation file is empty.
Call 1 enters the function, calls readFile, and awaits I/O. Then call 2 enters the function and does the same.
The same empty string is returned by both calls to readFile.
Now both calls to markReserved will fail to find the timeslot in the file contents, both will call append, and both will return true.

(⤭d) We have seen race condition fixes that create *mutual exclusion* by not giving up control with `await` during actions that must be atomic. We cannot simply do that here: the filesystem functions we need to use are asynchronous.

Instead, fix the race condition by completing the revised version below. Use the data structure `marking` to ensure that only one caller at a time is working with the file data for a room. The code has four new lines marked `// ✨new!✨`. Your additional code should fit in the box. You can find the spec of `Deferred` at the bottom of all the provided code.

(Worth noting: a correct solution still depends on code that runs atomically, not giving up control with `await` — but that section of code is relatively short.)

If you need an assumption about how `marking` is initialized when `Server` is constructed, state that assumption here:

(No, we may await undefined in the code below, when keys are not in the map, with no ill effect. Alternatively, we might initialize marking with resolved Promises.)

```
class Server {

    // ... possibly other fields or methods ...

    private readonly marking = new Map<string, Promise<void>>(); // ✨new!✨

    async markReserved(timeslotID: bigint, roomID: string): Promise<boolean> {
        const defer = new Deferred<void>(); // ✨new!✨

        const previous = this.marking.get(roomID);
        this.marking.set(roomID, defer.promise);
        await previous;


        const filename = `rooms/${roomID}.txt`;
        const roomFile = await fs.promises.readFile(filename, { encoding: 'utf-8' });
        const timeslotString = timeslotID.toString();
        for (const line of roomFile.split('\n')) {
            if (line === timeslotString) {
                defer.resolve(); // ✨new!✨
                return false;
            }
        }
        await fs.promises.appendFile(filename, timeslotString + '\n');
        defer.resolve(); // ✨new!✨
        return true; // caller has reserved roomID at timeslotID
    }
}
```

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

Code for Part II: geographic routes

```
/** A geographic point. */
class LatLon {

    /**
     * Make a new geographic point.
     * @param latitude the latitude, between -90 and 90, inclusive
     * @param longitude the longitude, between -180 and 180, inclusive
     */
    constructor(public readonly latitude: number, public readonly longitude: number) {
        // TODO assertions
    }

    /** @returns coordinate-pair format of this point */
    toString(): string {
        return `(${this.latitude} lat, ${this.longitude} lon)`;
    }

    /** TODO */
    equalValue(that: LatLon): boolean {
        return this.latitude === that.latitude && this.longitude === that.longitude;
    }
}

/**
 * A route from one geographic point to another point through zero or more intermediate points.
 * The route may re-visit points (forming loops), or may start and end at the same point if
 * there are points in between, but adjacent points in the route are distinct.
 */
interface Route {
    /** @returns the first point in this route */
    getStart(): LatLon;

    /** @returns the last point in this route */
    getEnd(): LatLon;

    /** @returns the number of points along this route */
    countPoints(): number;

    /** TODO */
    equalValue(that: Route): boolean;

    // ... other operations ...
}
```

```
/** Shortest-distance directed segment from one geographic point to another, distinct, point. */
class Segment implements Route {

    private readonly start: LatLon;
    private readonly delta: [ number, number ];

    /** TODO */
    constructor(start: LatLon, latDelta: number, lonDelta: number) {
        // TODO assertions
        this.start = start;
        this.delta = [ latDelta, lonDelta ];
    }

    getStart(): LatLon {
        return this.start;
    }
    getEnd(): LatLon {
        return new LatLon(this.start.latitude+this.delta[0], this.start.longitude+this.delta[1]);
    }
    countPoints(): number {
        return 2;
    }
    equalValue(that: Route): boolean {
        // ... TODO ...
    }
    // ... other operations ...
}

/**
 * Route constructed from a sequence of sub-routes.
 * Each sub-route starts where the previous sub-route ended.
 */
class Combined implements Route {

    private readonly subroutes: Array<Route>;
    private readonly first: Route;

    /** TODO */
    constructor(subroutes: Array<Route>) {
        // TODO assertions
        this.subroutes = subroutes.slice();
        this.first = this.subroutes[0] ?? assert.fail();
    }

    getStart(): LatLon {
        return this.first.getStart();
    }
    getEnd(): LatLon {
        return (this.subroutes.at(-1) ?? assert.fail()).getEnd();
    }
    countPoints(): number {
        return this.subroutes.reduce((count, s) => count+s.countPoints()-1, 1);
    }
```

```
    equalValue(that: Route): boolean {
        // ... TODO ...
    }
    // ... other operations ...
}
```

Code for Part III: room reservation calendar

```
/** A room in the room reservation system, with unique identifier roomID. */
class Room {

    /** TODO */
    constructor(public readonly roomID: string,
                /* ... other public readonly immutable properties ... */) { }

    equalValue(that: Room): boolean {
        // ... TODO ...
    }
}

// helper function, requests a reservation from the server
// returns (a promise of) true if the reservation was made successfully, false otherwise
async function requestReservation(timeslotID: bigint, roomID: string): Promise<boolean> {
    // ... TODO ...
}

// helper function, converts a Set to an Array so we can use Array operations
function setToArray<T>(set: Set<T>): Array<T> {
    return [ ...set.values() ];
}

class RoomCalendar {

    /**
     * Attempt to reserve the given room for the given set of times.
     * @param timeslotIDs timeslot IDs
     * @param roomID room ID
     * @returns the subset of the given timeslots for which the room has been successfully reserved
     */
    async reserve(timeslotIDs: Set<bigint>, roomID: string): Promise<Set<bigint>> {
        const success = new Set<bigint>();
        const fn = async (timeslot: bigint): Promise<void> => {
            if (await requestReservation(timeslot, roomID)) {
                success.add(timeslot);
            }
        };
        setToArray(timeslotIDs).forEach(fn);
        return success;
    }

    // ... other operations ...
}
```

And as seen in class and readings:

```
type Resolver<T> = (value: T | PromiseLike<T>) => void;
type Rejector = (reason: Error) => void;

/** Deferred represents a promise plus operations to resolve or reject it. */
export class Deferred<T> {

    /** The promise. */
    public readonly promise: Promise<T>;

    /** Mutator: fulfill the promise with a value of type T. */
    public readonly resolve: Resolver<T>;

    /** Mutator: reject the promise with an Error value. */
    public readonly reject: Rejector;

    /** Make a new Deferred. */
    public constructor() {
        // ... implementation omitted ...
    }
}
```