MIT
6.102 Software Construction
Max Goldman, Mitchell Gordon, and Rob Miller

# 6.102 Spring 2026 Exam 2

- Please arrive to the exam with this page open and **all other apps, windows, and tabs already closed**.
- Your exam location is: **26-100**.
- You have **120 minutes** to complete this exam. There are **6 problems** that vary in length.
- The exam is **closed-book** and closed-notes, but you are allowed to bring a single 8.5×11″ double-sided page of notes, handwritten directly on the paper (not computer-printed or photocopied), readable without a magnifying glass, created by you.
- You may also use blank scratch paper. Staff cannot provide scratch paper.
- You may use **nothing else on your computer** or other devices: no 6.102 website; no TypeScript interpreter 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.
- 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 exam.
- If you feel the need to **write a note to the grader**, or if you want to write a note to yourself, you can click the gray pencil icon to the right of the answer.
- If you have a question, or need to use the restroom, please raise your hand.
- If you find yourself bogged down on one part of the exam, remember to keep going and work on other problems, and then come back.
- To **leave early**: enter *done* at the very bottom of the page, and show your screen with the check-out code to a staff member.
- You **may not discuss** details of the exam with anyone other than course staff until exam grades have been assigned and released.

Good luck!

# Problem ⤮1

Ben is designing a little language for integer difference expressions. Every expression subtracts one expression from another, and the leaves are nonnegative integer constants.

Examples of valid expressions:

```
3
(3-5)
((9-2)-3)
(9-(2-3))
(((8-4)-(3-1))-7)
```

In this language:

- the only operation is binary minus;
- leaf constants are nonnegative integers, written without unnecessary leading zeroes;
- every compound expression used must be parenthesized; and
- the whole expression must not have unnecessary parentheses.
- do not allow whitespace.

**(⤮a)** Write a grammar for this language. The root nonterminal should be `diff`.

**(⤭b)** Define an immutable recursive data type `Diff` suitable for representing the abstract syntax tree for this language.

Write **TypeScript code** in the box below, with **no comments**.

- start with only one operation, `toString`, which returns a string in the language accepted by your grammar;
- don't include any parsing code.

Again, **do not include comments or documentation**, only TypeScript code.

**(⤭c)** Write a recursive data type definition for `Diff`.

**(⤭d)** Implement a new operation, `invert()`, for your data type. For an expression `e`, `e.invert()` must return an expression whose integer value is the negative of `e`'s integer value.

For example, if `e.toString()` is `(9-(2-3))`, then `e.invert().toString()` may be `((2-3)-9)`.

The returned expression should alias as many of `e`'s subexpressions as possible.

- You don't have to repeat unchanged code from the previous problem; just show what's new, making clear where it would go in the original code.
- Again, **don't** include any comments.

**(⤭e)** If `e.toString()` is `((1-2)-(3-4))`, how many new objects are created by `e.invert()`?

---

# Problem ⤭2

This problem is about `EventLog`, a circular event log that stores the most recent `capacity` events. Older events are overwritten when the log is full.

The code for `EventLog` is shown below.

```
/**
 * A mutable circular event log that stores the most recent `capacity` events.
 * Older events are overwritten when the log is full.
 */
export class EventLog {

    private readonly slots: Array<string>;
    private pos: number;
    private count: number;
    private readonly listeners: Array<(event: string) => void> = [];

    private checkRep(): void {
        // assert the rep invariant
    }

    /**
     * Make a new empty EventLog.
     * @param capacity maximum number of recent events to store, must be > 0
     */
    public constructor(public readonly capacity: number) {
        this.slots = new Array(capacity).fill('');
        this.pos = 0;
        this.count = 0;
        this.checkRep();
    }

    /**
     * Append an event to the log.
     * If the log is full, the oldest event is overwritten.
     * Notifies all registered listeners with the event.
     * @param event the event string to append
     */
    public append(event: string): void {
        this.slots[this.pos] = event;
        this.pos = (this.pos + 1) % this.capacity;
        this.count++;
        for (const listener of [...this.listeners]) {
            listener(event);
        }
        this.checkRep();
    }

    /**
     * @returns an array containing the min(total(), capacity) most recent events in this log, from ol
     */
    public recent(): Array<string> {
        if (this.count <= this.capacity) {
            return this.slots.slice(0, this.count);
        }
        return [...this.slots.slice(this.pos),
                ...this.slots.slice(0, this.pos)];
    }
```

```
    /**
     * @returns the total number of events ever appended
     */
    public total(): number {
        return this.count;
    }

    /**
     * Modifies this log by adding a listener.
     * Listeners are called synchronously, in the order they were registered.
     * @param listener called by this log each time an event is appended
     */
    public onAppend(listener: (event: string) => void): void {
        this.listeners.push(listener);
    }

}
```

**(⤭a)** Write a precise rep invariant and abstraction function for this class. Write them clearly as functions *RI(...) = ...* and *AF(...) = ...*:

Rep invariant:

Abstraction function:

**(⤭b)**

Suppose the file `log.txt` initially stores an `EventLog` with capacity 3. Its `recent()` returns `['c', 'd', 'e']`, from oldest to newest, and its `total()` returns 5, meaning that 5 events have ever been appended. Assume these helper functions are already implemented correctly:

```
/** @returns the EventLog currently stored in filename. */
function readLog(filename: string, capacity: number): EventLog { ... }

/** Replaces filename with the current state of log. */
function writeLog(filename: string, log: EventLog): void { ... }
```

Now suppose we run two Workers that read and write the same file.

Worker A appends “x”:

```
const logA = readLog('log.txt', 3);            // A1
logA.append('x');                              // A2
writeLog('log.txt', logA);                     // A3
```

Worker B appends “y”:

```
const logB = readLog('log.txt', 3);            // B1
logB.append('y');                              // B2
writeLog('log.txt', logB);                     // B3
```

When A and B are running concurrently, steps A1–A3 and B1–B3 can interleave with each other. For this question, assume each labeled step runs to completion before the next labeled step starts. The steps within each worker must stay in order, but steps from different workers may be interleaved. Do not consider partial reads or writes of `log.txt`.

Show an interleaving of all 6 steps where event “x” is lost – that is, the final file does not contain “x”. Write the steps in the order they run for this bad interleaving:

After your interleaving, what would `recent()` return, from oldest to newest, for the `EventLog` stored in the resulting file? What would `total()` return?

`recent()` =

`total()` =

**(d)**

Assume `checkRep()` asserts the rep invariant. Notice that `append` calls `checkRep()` at the end.
During the interleaving from the previous question, does either call to `checkRep()` at the end of `append()` fail? Explain why or why not in one clear sentence:

○ Yes
○ No

**(e)**

To fix this race condition, we redesign the system so that only one process accesses the log file. Workers send append requests via message passing, and a single **coordinator** handles them.

Complete the coordinator's message handler in at most 4 lines of code. The handler receives messages of the form `{ event: string }` and should append the event to the log file:

```
import { parentPort } from 'node:worker_threads';

const filename = 'log.txt';
const capacity = 3;

parentPort.on('message', (msg: { event: string }) => {
    // your code here
});
```

In one clear sentence, explain why this approach prevents the race condition from the previous question:

# Problem ×3

This problem also uses `EventLog`.

The questions below ask about `onAppend(...)` listeners and callback order. Note that `EventLog.onAppend(listener)` adds a listener that is called **synchronously**, in the order added, each time `append` is called. Also, `setTimeout(callback, delay)` stores a callback to run later, after control returns to the event loop. When multiple callbacks are waiting in the event queue, they are handled first-in-first-out.

**(×a)**

Suppose we first try the following code:

```
const log = new EventLog(3);

log.onAppend((event) => {
    console.log('heard ' + event);
});

console.log('start');
log.append('a');
console.log('middle');

setTimeout(() => {
    log.append('b');
    console.log('done');
}, 0);

log.append('c');
console.log('end');
```

List each line that is printed to the console, in order:

**(×b)**

Now consider this code, which registers two listeners:

```
const log = new EventLog(2);

log.onAppend((event) => {
    console.log('L1:' + event);
});

log.append('a');
```

```
log.onAppend((event) => {
    console.log('L2:' + event);
});

log.append('b');
log.append('c');
```

List each line that is printed to the console, in order:

**(⤭c)**

Here is some code that uses `setTimeout` inside a listener:

```
const log = new EventLog(2);

log.onAppend((event) => {
    console.log('heard ' + event);
    setTimeout(() => {
        console.log('later ' + event);
    }, 0);
});

log.append('a');
log.append('b');
console.log('done');
```

List each line that is printed before any timer callback has run:

List all lines printed by the whole program, in order:

# Problem ⤭4

**(⤭a)**

Given this code, in which `f`, `g`, and `h` are pure functions with no side-effects on each other:

```
function compute(): number {
    return f(g() + h()) / f(0);
}
```

Suppose `f`, `g`, and `h` have been changed to asynchronous functions. Rewrite `compute`, including its function signature, so that it runs as concurrently as possible. Your answer should fit in the box without scrolling.

**(⤭b)**

Implement the function below. You may assume you have access to `Deferred` or `Promise.withResolvers`.

```
/**
 * Wrap a function with a "watcher" that transfers the function's return value
 * into a promise that another client may wait for and observe.
 * @param func  any function
 * @return an object record with two properties, `funcWithWatch` and `promise`,
 * where `funcWithWatch` is a function that calls `func`
 * and resolves `promise` to the return value of that call.
 */

function watch<T>(func: void=>T): {
    funcWithWatch: void=>T,
    promise: Promise<T>
} {
```

```
}
```

The spec leaves unspecified what should happen if `func` throws an exception. What does your implementation do? In one sentence each, state the effect of the exception on:

- the caller of `watch()`

- the caller of `funcWithWatch()`

- the promise

Louis Reasoner calls `watch` and gets back `funcWithWatch` and `promise`. He wants to use this `promise` to observe what *every* call of this `funcWithWatch` function returns. Explain why this won't work.

# Problem ⤭5

Recall the `map` operation that applies a function to an array, for example:

```
[1,2,3].map(x => x+1);  // returns [2,3,4]
```

**(⤭a)** The function `f` passed to `map` has a spec, with a precondition and postcondition. The spec of `f` interacts with other parts of the spec of `map`. In particular, how does the spec of `f` constrain the input array, and is this effect a precondition or postcondition of `map`? Your answer should fit in the box without scrolling.

Louis Reasoner suggests changing the spec of `map` so that the returned array may be smaller than the input array, only containing the return values of legal calls to `f` and omitting the ones that violate `f`'s precondition.

Is Louis's spec STRONGER, WEAKER, or INCOMPARABLE to the original spec of `map`? Why? Your answer should fit in the box without scrolling.

Louis implements his new spec for `map` as follows:

```
const result = [];
for (const val of this) {
    try {
        result.push(f(val));
    } catch (e) {
    }
}
return result;
```

Alyssa Hacker code-reviews Louis's implementation, saying "this doesn't satisfy your new spec for `map`." Explain what she means. Your answer should fit in the box without scrolling.

Ben Bitdiddle also code-reviews, and writes "even if this satisfied your new spec for `map`, it doesn't fail fast." What does Ben mean? Your answer should fit in the box without scrolling.

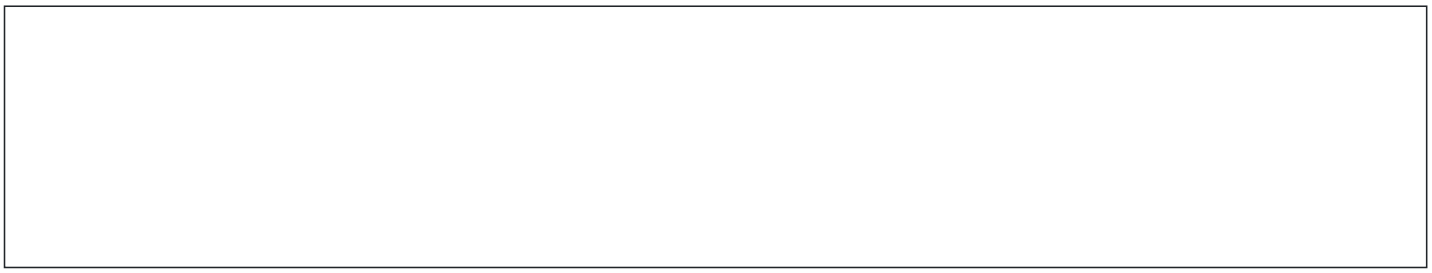
**(⤭b)**

Let's consider another approach: a function `mapper` that converts a function from `number` to `number` into a function from `Array<number>` to `Array<number>`:

```
const f = (x => x+1);
const g = mapper(f);
g([1,2,3]); // returns [2,3,4]
```

Implement `mapper` below. You may use `map`.

```
function mapper(f: number=>number): (arr:Array<number>)=>Array<number> {
```



```
}
```

Since `mapper` will be used only in specific ways in our program, let's add the requirement that the input array can only contain integers.

How does this change affect the specification of each of the functions involved in `mapper`? For each box below, answer whether the new spec is STRONGER, WEAKER, INCOMPARABLE, or UNCHANGED, and briefly explain why in terms of preconditions and/or postconditions.

The spec of the function passed to `mapper` (labeled `f` in the example above)

The spec of the function returned by `mapper` (labeled `g` above)

The spec of `mapper` itself

# Problem ×6

Recall the Star Battle project that you just completed. For each of the design changes below, briefly explain which part(s) of the system would need to change and how.

- do not specify filenames or exact class names
- but do clearly state **where** each change is in the running system
- even if your final project already had this design change, you still need to say which parts of the system were affected

Your answers should fit in the boxes without scrolling.

Display a list of available puzzles

A way for the user to mark cells that they think *must* be empty

---

Additional information for problem 4(b):

`Deferred<T>` is an abstract data type that bundles up a `Promise<T>` together with its mutators:

- `new Deferred<T>()` makes a new `Deferred<T>` object
- `deferred.promise` is the `Promise<T>` associated with the `Deferred` object
- `deferred.resolve(t:T)` resolves the associated promise with the value `t`
- `deferred.reject(err:Error)` rejects the associated promise with the given error

`Promise<T>.withResolvers()` returns a record type (an object) with three properties:

- `promise:Promise<T>` is the new promise
- `resolve:(t:T)->void` is a mutator that resolves the promise with the value `t`
- `reject:(err:Error)->void` is a mutator that rejects the promise with the given error