6.102 Spring 2025 Quiz 2

- 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!

You can open this introduction and subsequent examples and code in a separate tab.

On this quiz we will consider a *database system* where data are organized into tables. Each **table** consists of:

- a fixed ordered list of one or more **columns**, where each column has a unique *name*; and
- an unordered collection of zero or more **rows**, where every row has a single *value* for each column of the table.

In a more powerful database, every column might also have a *type* shared by all its values. For simplicity, in our database every value has type `string`.

For example, here is a table of baseball teams:

place	teamname	division
Seattle	Mariners	AL West
Boston	Red Sox	AL East
San Francisco	Giants	NL West
St. Louis	Cardinals	NL Central

And a table of baseball team mascots:

teamname	mascotname
Mariners	Mariner Moose
Red Sox	Wally the Green Monster
Giants	Lou Seal
Cardinals	Fredbird

Problem ⋈1

Let's create an ADT to represent tables as defined above. The most fundamental operation on tables is iterating over their rows, which we will do with a generator.

In these specifications, be precise and complete but absolutely DRY.

(⋈a) Complete the missing specifications of `Table`:

```
/** An immutable database table. */
export interface Table {
  /**

    e.g.
    @returns the ordered list of unique column names for this table

  */
  columns(): Array<string>;

  /**

    @returns a generator that iterates over the rows of this table in arbitrary order

  */
  rows(): Generator<Row>;
}
```

(✂b) And complete the missing specifications of Row:

```typescript
/** An immutable database table row. */
export interface Row {

  /**

     @returns the ordered list of unique column names for this row's table

   */
  colNames(): Array<string>;

  /**

     @param column integer column index
     @returns the value of the column-th column in this row
     @throws Error if column < 0 or column >= colNames().length

   */
  value(column: number): string;
}
```

Problem ✂2

Let's start with an implementation of Table where data for each table is stored in a file on the filesystem: (You can [open this code in a separate tab](#).)

Assume we have a function `parseData : string → Array<Row>` that does the parsing.

```typescript
/** An immutable database table stored on the filesystem. */
export class DataFile implements Table {

    private readonly data: Array<Row>;

    /** ... TODO ... */
    public constructor(filename: string) {
        this.data = parseData(fs.readFileSync(filename, { encoding: 'utf-8' }));
    }

    /** @inheritdoc */
    public columns(): Array<string> {
        return this.data[0].colNames();
    }

    /** @inheritdoc */
    public * rows(): Generator<Row> {
        yield * this.data; // "yield *" is like Python's "yield from"
    }
}
```

Working with this code as given, even though there may be problems we will investigate below...

(⚔**a**) Write an excellent, straightforward RI:

RI(data) = data.length > 0 and row.colNames() are identical for all row in data

(⚔**b**) Write an excellent, straightforward AF:

AF(data) = the table with columns data[0].colNames() and the rows in data

(⚔**c**) Write an excellent, straightforward SRE:

data is private, unreassignable, and never returned; yielded Rows are immutable; returned Array<string> is from immutable Row

Considering the code as given and our ADT as defined...

(⚔**d**) There are many distinct abstract values that this rep cannot represent — an unbounded number of them! Why? Explain clearly and specifically in one sentence:

The implementation of columns() prevents us from representing any empty table.

And is this a bug? Say yes or no and explain clearly and specifically in one sentence:

Yes, empty tables are valid abstract values we should be able to represent.

(⚔**e**) There are many distinct rep values that represent the same abstract value — an unbounded number! Why? Explain clearly and specifically in one sentence:

data is an ordered array, and all orderings of the same rows represent the same abstract table with unordered rows.

And is this a bug? Say yes or no and explain clearly and specifically in one sentence:

No, we can have multiple reps for the same abstract value, so long as the behavior of the operations is correct.

(⚔**f**) There are many rep values that statically type-check but which do not represent an abstract value — an unbounded number! Why? Explain clearly and specifically in one sentence:

If different rows in data have different colNames(), there is no type error, but this is not an abstract table.

And is this a bug? Say yes or no and explain clearly and specifically in one sentence:

No, we can exclude these invalid reps in our RI and defend against bugs by checking the RI.

Problem ⚔3

(You can open this description in a separate tab.)

So far, we have only discussed two operations on tables: creating them from a data file, and iterating over their rows. Let's add two more operations for performing *queries* on our data:

1. **FILTER** to filter the rows (with keyword WHERE) and columns (with keyword SELECT) of a table, keeping only rows whose values match a predicate and keeping only certain columns.
2. **JOIN** to join together two tables by matching rows that have the same value for same-name columns.

As we've done with arithmetic expressions, image memes, and music, let's have a textual language for describing these operations: a *database query language*. Every valid expression in our language will represent a **table**.

Assume we have two data files on the filesystem, teams and mascots, that correspond to our baseball teams and mascots above. Here are examples of expressions in our database query language. We include the resulting tables they represent, but you are **not** implementing these queries, just parsing them.

Textual query	Resulting table
`teams`	**place** **teamname** **division** Seattle — Mariners — AL West Boston — Red Sox — AL East San Francisco — Giants — NL West St. Louis — Cardinals — NL Central
`FILTER teams WHERE division="AL East" SELECT teamname, place`	**teamname** **place** Red Sox — Boston
`FILTER teams SELECT teamname, place`	**teamname** **place** Mariners — Seattle Red Sox — Boston Giants — San Francisco Cardinals — St. Louis
`FILTER teams WHERE division="AL East" SELECT *`	**place** **teamname** **division** Boston — Red Sox — AL East
`JOIN teams WITH mascots`	**teamname** **place** **division** **mascotname** Mariners — Seattle — AL West — Mariner Moose Red Sox — Boston — AL East — Wally the Green Monster Giants — San Francisco — NL West — Lou Seal Cardinals — St. Louis — NL Central — Fredbird

`JOIN`
`FILTER teams SELECT *`
`WITH`
`FILTER mascots SELECT *`
	same as previous example

`FILTER`
`JOIN FILTER teams SELECT teamname, division WITH mascots`
`WHERE division = "AL East" SELECT *`
	teamname **division** **mascotname**
	Red Sox — AL East — Wally the Green Monster

Notice that some query expressions have recursive structure, and notice that `FILTER` supports only one `WHERE` predicate but multiple `SELECT` columns.

(✂a) Complete a ParserLib grammar to parse this textual language, such that all the examples above are valid. Here are a few invalid queries your grammar should reject:

```
FILTER teams                                    (missing SELECT...)
FILTER teams SELECT teamname SELECT place       (double SELECT...)
JOIN teams                                      (missing WITH...)
JOIN teams WITH mascots WHERE division="AL East" (WHERE without FILTER...)
```

Assume data file names and columns names must match the provided **name** nonterminal, and use the provided **predicate** and **columns** nonterminals. Include a nonterminal called **query** that is the top-level production of the grammar.

Design your grammar to be SFB/ETU/RFC, and to support AST-building code that is SFB/ETU/RFC.

```
@skip whitespace {

   query ::= name | filter | join ;
   filter ::= 'FILTER' query ('WHERE' predicate)? 'SELECT' ('*' | columns) ;
   join ::= 'JOIN' query 'WITH' query ;

    predicate ::= name '=' value ;
    columns ::= name (',' name)* ;
}
name ::= [a-z] [a-z0-9]* ;
value ::= '"' ([^"\] | '\' ["\])* '"' ;
whitespace ::= [ \t\r\n]+ ;
```

Valid expressions are tables, so **Table** will be the interface for our AST, and concrete variants will implement `Table`. We will also make use of two more types that do **not** implement `Table`.

(✂b) First, `Predicate` represents the predicate in a filtering query. Write **just the signature** for the single most crucial operation we will need from `Predicate`, where the input is a Row:

```
/** Represents a query row-filtering predicate. */
export interface Predicate {

    e.g. matches(row: Row): boolean;

}
```

Be sure to choose good names and types. Do not write out the spec.

(✂c) Write a data type definition where `Predicate` has exactly **two** variants that cover the two kinds of predicates found in the query language examples above:

```
=

    +

    e.g. Predicate = HasValue(column: string, value: string) + True()
```

Be sure to use good names and include parameter names and types.

(✂d) Second, `ColumnSpec` represents the column list in a filtering query. Write **just the signature** for the single most crucial operation we will need from `ColumnSpec`, where the input is a `Row`:

```
/** Represents a query column-filtering specification. */
export interface ColumnSpec {

}
```

     e.g. `narrowed(row: Row): Row;`

Be sure to choose good names and types. Do not write out the spec.

(✂e) Write a data type definition where `ColumnSpec` has exactly **two** variants that cover the two kinds of column lists found in the query language examples above:

  `=`

    `+`

    e.g. `ColumnSpec = OnlyCols(columns: Array<string>) + AllCols()`

Be sure to use good names and include parameter names and types.

(✂f) Finally, with the help of `Predicate` and `ColumnSpec`, complete a recursive data type definition for `Table` where it has exactly **three** variants:

  `=`

    `DataFile(`*...correct parameters...*`)`

    `+`

    `+`

    `Table = DataFile(data: Array<Row>) + Filter(table: Table, predicate: Predicate, columnSpec: ColumnSpec) + Join(left: Table, right: Table)`

Be sure to use good names and include parameter names and types.

Problem ⨯4

A database certainly ought to support *concurrent* operation, for example because executing queries may take time, and many clients may be accessing the database simultaneously.

To focus on support for asynchronous queries, let's set aside using `Generator`: JavaScript combines async and generators into *async generators*, but we haven't practiced using those.

Instead, let's replace our use of generators with the following *iterator* type called `Cursor`, which is generic in the type of items it iterates over:

```
export interface Cursor<Item> {
    /**
     * @returns (a promise of) the next item or undefined if there are no more items,
     *          where the returned promises will resolve in order
     */
    next(): Promise<Item|undefined>
}
```

Now the table interface could look like:

```
export interface CursorTable {
    // this operation is the same:
    columns(): Array<string>;

    // this operation returns a Cursor instead of a Generator:
    rows(): Cursor<Row>;
}
```

Suppose we write a new version of `parseData`:

```
function cursorParseData(filename: string): Cursor<Row> {
    // each line in the data file stores one row of the table
    const lines: Cursor<string> = // ... get a Cursor for the lines of text in filename ...
    return new MappingCursor<string, Row>(lines, (line) => new DataRow(line));
}
```

Assume the following about `DataRow`:

```
export class DataRow implements Row {
    public constructor(dataline: string) { ... }
    // ... colNames() and value(..) ...
}
```

Complete the implementation of `MappingCursor`. Note that it has two generic type parameters. You do *not* need to write specs or internal documentation.

```
export class MappingCursor<Input, Output> implements Cursor<Output> {

    // fields (if any; you do not need to write internal docs)

    // constructor (you do not need to write the spec)

    constructor(private readonly source: Cursor<Input>,
                private readonly fn: (input: Input) => Output) { }

    /** @inheritdoc */
    public async next(): Promise<Output|undefined> {

        const next = await this.source.next();
        if (next === undefined) { return undefined; }
        return this.fn(next);

    }
}
```

Problem ⋊5

Using `cursorParseData` from the previous problem to once again implement a table where the data are stored in a file on the filesystem, suppose we write:

```
export class CursorDataFile implements CursorTable {
    private readonly data: Cursor<Row>;
    constructor(filename: string, /* other parameters */) {
        this.data = cursorParseData(filename);
        // ... other fields ...
    }
    public columns(): Array<string> { /* ... */ }
    public rows(): Cursor<Row> { return this.data; }
}
```

(⋊a) What is wrong with this approach for `rows()`? Say what kind of bug this is, and explain with a simple, specific scenario:

Aliasing mutability bug: if we call rows() and iterate over the results, calling rows() again returns the same Cursor which will not return those rows again.

To fix the problem, we try writing this class for use in `CursorDataFile`:
(You can [open this code in a separate tab](#).)

```
export class ReplayCursor<Item> implements Cursor<Item> {

    // public factory function to create ReplayCursors
    public static make<Item>(source: Cursor<Item>) {
        return new ReplayCursor([], source);
    }

    private idx = 0;

    private constructor(
        private readonly seen: Array<Item>,
        private readonly source: Cursor<Item>
    ) {
    }

    public fromBeginning(): ReplayCursor<Item> {
        // instances of ReplayCursor deliberately share rep values!
        return new ReplayCursor(this.seen, this.source);
    }

    /** @inheritdoc */
    public async next(): Promise<Item|undefined> { // to refer to these lines in questions below:
        if (this.idx < this.seen.length) {          // A
            return this.seen[this.idx++];           // B
        }
        const item = await this.source.next();      // C
        if (item !== undefined) {                   // D
            this.seen[this.idx++] = item;           // E
        }
        return item;                                // F
    }
}
```

(Remember that "this.idx++" is a post-increment: it evaluates to the current value of this.idx, then increases it by one.)

We pat ourselves on the back for our cleverness, and the code sometimes works. It also sometimes doesn't.

(✕b) Write an excellent AF, explaining a complete abstract value. *Ignore* the concurrency bug but pay attention to the entire rep and all operations:

> AF(idx, seen, source) = the Cursor over the items in seen followed by the items from source, where the next item
> will be: seen[idx] if it exists, or the next item from source otherwise

(✕c) What is the concurrency issue? Assume we're running at the top level and have the following variables:

```
const yum: Cursor<string> = // some Cursor that produces: "apple", "banana", "coconut"
const rc1 = ReplayCursor.make(yum);
const rc2 = rc1.fromBeginning();
```

Remember that the promises from yum will resolve in order.

Demonstrate the bug by writing at most four lines of code that do **not** refer to yum and do **not** use loops or control structures. The code, when followed by the subsequent line console.log(val1, val2) given, should demonstrate the bug:

```
    const promise1 = rc1.next();
    const promise2 = rc2.next();
    const val1 = await promise1;
    const val2 = await promise2;

  console.log(val1, val2); // not the fruits we expected!
```

State what the console output will be:

```
    apple, banana
```

And explain specifically the sequence of events and what happened to the reps of `rc1` and/or `rc2` to cause this bug. Be as specific as possible about what happens at any points where control is given up:

rc1.next() calls source.next() and returns a promise at its await.
rc2.next() does the same, and we then give up control at await promise1.
The first source.next() resolves with "apple", which is assigned into seen[0] and returned.
The second source.next() resolves with "banana", which is again assigned into seen[0] and returned.

(✂**d**) Fix the bug by reimplementing `next()` in at most eight lines of code, without changing the rep:
(Lines with only closing braces are not counted. Answers with more than eight code lines will receive no credit.)

```
    public async next(): Promise<Item|undefined> {

    }
```

e.g.

```
    const item = await this.source.next();
    if (item !== undefined) {
        this.seen.push(item);
    }
    if (this.idx < this.seen.length) {
        return this.seen[this.idx++];
    }
    return undefined;
```

In at most two sentences, why is your implementation correct? Why does it no longer have the concurrency bug? We will not run your code — even if your code is correct, this argument must be clear and convincing in order to receive any credit!

We no longer have an await that separates establishing idx === seen.length from later using idx and seen with that assumption; instead we always append to seen, and we use idx and seen without giving up control.

Note that your implementation may require a different RI than `ReplayCursor` originally assumed and/or a revised AF. You do **not** need to update your AF above.

Problem ⋊6

To support many clients accessing the database over a network, suppose the system supports a message-passing protocol with the following messages:

Message	Parameters	Specification
add_rows	tablename, list of lists of values	Add the given rows to the given table, and respond to indicate the rows have been added
delete_where	tablename, predicate	Delete the rows from the given table that match the predicate, and respond to indicate they have been deleted
start_query	query string	Start evaluating the query, and respond with a unique ID for retrieving the results
get_batch	unique query ID	Respond with more result rows from the given query, with the guarantees below...

Notice that clients are now working with mutable abstract tables.

The idea of start_query and get_batch is that a query may take a long time to run and may have many results. After getting a unique ID from start_query, the client calls get_batch multiple times, and the server will guarantee:

1. get_batch returns at least one row *if and only if* there are more result rows to return, otherwise it returns nothing.
2. Every result row is returned in exactly one get_batch.

You would like to add a third guarantee:

3. The returned rows represent the result of the query at the time of start_query, they do not reflect later changes.

(⋊a) What is guarantee (3)? Pick one and justify your answer in one sentence:

Postcondition, this is only a condition on the returned results.

(⋊b) Explain in one clear sentence using concepts we have studied: what makes guarantee (3) difficult to add?

Concurrent requests interleave and mutate shared data, so we need to defend against race conditions.

So let's go ahead and add guarantee (3)...

(⋊c) If we modify the system so that add_rows and delete_where wait until the relevant rows can be added/deleted without disrupting concurrent work: how can one client, using the protocol, prevent others from ever making progress? Give a specific scenario:

Start queries for everything in the database and never make any get_batch requests, so that all additions and deletions must wait indefinitely.

(⋈**d**) Suppose `get_batch` works by calling an internal `getNextBatch(..)` operation and responding with its output. We modify the system so that:

- Taken all together, the results from `getNextBatch` always include all the rows that are correct to return according to all three guarantees — but may also include result rows that should be omitted according to (3).
- Before returning each batch in a `get_batch` response, we correctly filter out any result rows in that batch that don't meet guarantee (3).

What additional change must we make so the system works as intended? Explain clearly:

> To satisfy (1), if we filter all the rows from a batch, we must keep calling getNextBatch until we have a nonempty filtered result or get an empty batch.