Example questions (fall 2019)

This is a subset of the questions that were asked in Java on 6.031 Fall 2019 Quiz 2. They have been translated to TypeScript here.

The problems in this quiz refer to the code for mutable `MutInfoEntry` and immutable `ImInfoEntry`, at the end of this quiz.

You may detach the code pages.

Train stations, airports, and other transit hubs often have displays that show upcoming departures or arrivals along with other information: a track or gate number, delays, cancellations, etc.

For this quiz, an *information board* is made of several *information board entries*. Each entry has limited space: 16 characters to display a *destination* and 12 characters for a *status*. Both are restricted to upper-case letters, digits, colons, and spaces. For example, a board with three entries:

```
WASHINGTON DC    11:05 AM

LONDON HEATHROW  11:55 AM

HONG KONG        DELAYED
```

In order to show more information, the board cycles each entry through a looping sequence of up to four statuses. For example, if WASHINGTON DC and LONDON HEATHROW have 2-status loops, and HONG KONG has a 3-status loop, then every few seconds the board will update:

```
WASHINGTON DC    ON TIME
LONDON HEATHROW  ON TIME
HONG KONG        NEW DEPRTURE
```

```
WASHINGTON DC    11:05 AM
LONDON HEATHROW  11:55 AM
HONG KONG         1:40 PM
```

```
WASHINGTON DC    ON TIME
LONDON HEATHROW  ON TIME
HONG KONG        DELAYED
```

```
WASHINGTON DC    11:05 AM
LONDON HEATHROW  11:55 AM
HONG KONG        NEW DEPRTURE
```

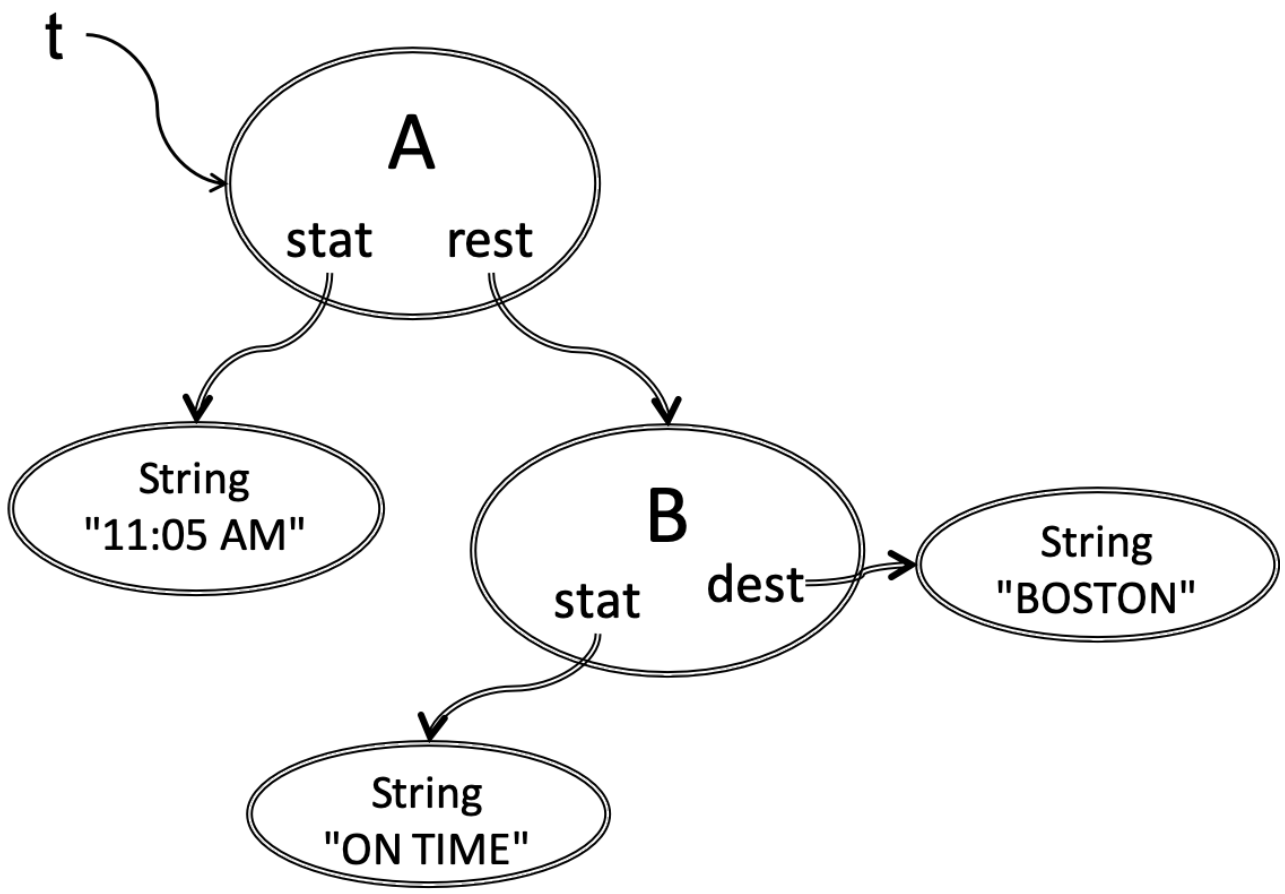
```
WASHINGTON DC    ON TIME
LONDON HEATHROW  ON TIME
HONG KONG         1:40 PM
```

... and so on.

2. (26 points) Recursive Datatypes

Suppose we want to implement `ImInfoEntry` (an *immutable* information board entry) as a recursive data type with two variants. The two variants are called A and B.

The snapshot diagram below shows how the datatype represents an information board entry `t` with destination "BOSTON" and two statuses "11:05 AM" and "ON TIME", whose current status is "11:05 AM".



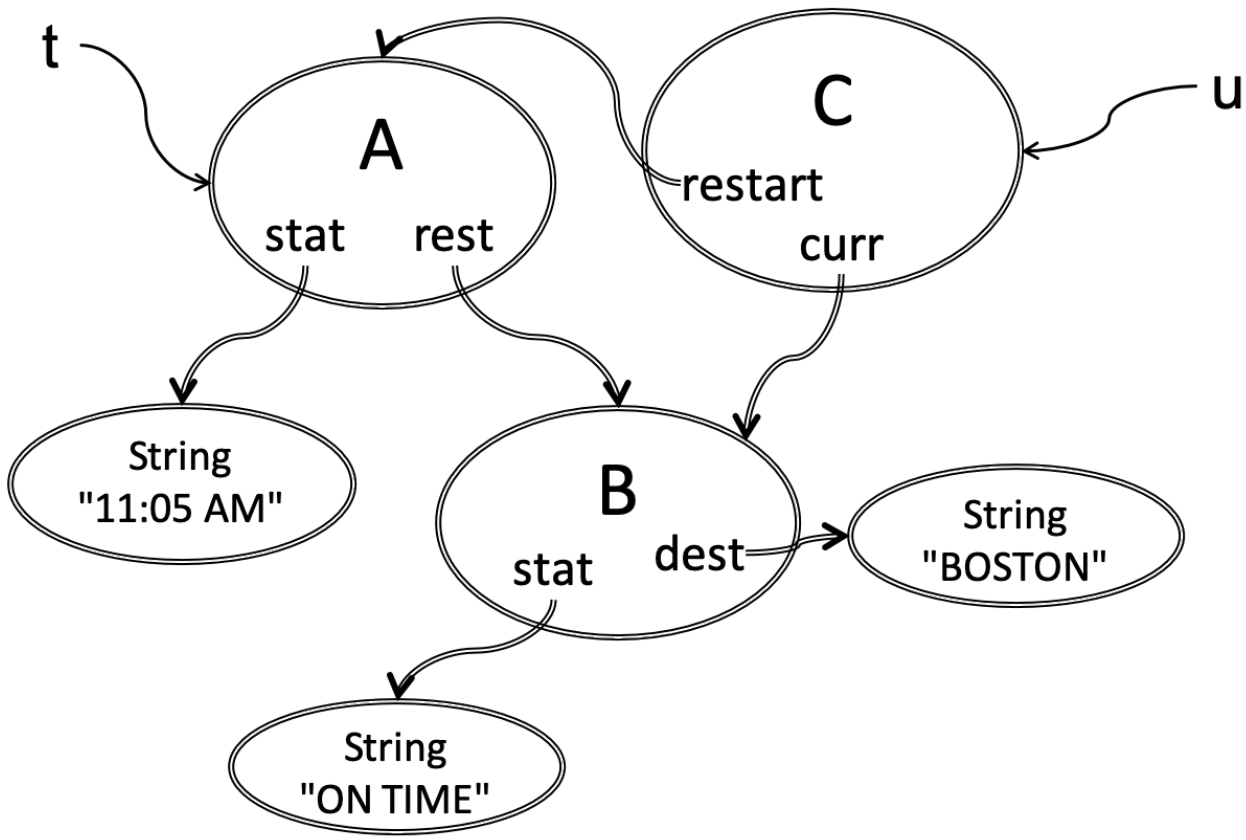
(a) Write a datatype definition that corresponds to the snapshot diagram and implements `ImInfoEntry`.

```
ImInfoEntry =
```

(b) Fill in the blanks to implement `destination()`, `status()`, and `size()` for variants A and B:

```
export class A implements ImInfoEntry {
  ...
  public destination(): string { return ____________________________ ; }


  public status(): string      { return ____________________________ ; }


  public size(): number        { return ____________________________ ; }
}

export class B implements ImInfoEntry {
  ...
  public destination(): string { return ____________________________ ; }


  public status(): string      { return ____________________________ ; }


  public size(): number        { return ____________________________ ; }
}
```

To help implement the `nextEntry` operation, we add one more variant `C`. The result of `u = t.nextEntry()` is shown in the snapshot diagram below.



(c) Fill in the blanks to implement `nextEntry()` for all three variants.

```
export class A implements ImInfoEntry {
  ...
  public nextEntry(): ImInfoEntry {
      return new C(this, this.rest);
  }
}

export class B implements ImInfoEntry {
  ...
  public nextEntry(): ImInfoEntry {

      return ____________________________________________________ ;

  }
}

export class C implements ImInfoEntry {
  ...
  public nextEntry(): ImInfoEntry {
    if (this.curr.size() === 1) { // curr has reached the end of the list

      return ____________________________________________________ ;

    } else {

      return ____________________________________________________ ;
    }
  }
}
```

3. (22 points) Grammars

(a) Which of these regular expressions accept (fully match) every legal status and destination string, and reject (fail to fully match) at least one illegal string? Circle YES or NO.

```
[A-Z0-9: ]+
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO
```

```
([A-Z]*|[0-9]*|:*| *)+
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO
```

```
[A-Z]*[0-9]*[:]*[ ]*
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO
```

```
.*[A-Z0-9: ]*
        matches every legal string?              YES      NO

        rejects at least one illegal string?     YES      NO
```

(b) Suppose an information board entry is represented as a string of text as in this example:

```
WASHINGTON|NEW DEPRTURE,TRACK 2,11:35AM
```

Complete the grammar below so that it can be used to parse an information board entry, with starting nonterminal `infoentry`. Your grammar must use the `destination` and `status` nonterminals shown, which you can assume have been defined with a correct answer from part (a).

For the purpose of this grammar, assume that statuses and destinations have **no maximum length**, and an information board entry has **no maximum number of statuses**.

```
destination ::= *a correct regular expression from part (a)*
status ::= *a correct regular expression from part (a)*
```

```

```

4. (26 points) Map/Filter and Callbacks

Suppose we add `map` and `filter` operations to `ImInfoEntry`, to transform the (cyclic) stream of status messages that an information board entry displays:

```
map: ImInfoEntry x (string -> string) -> ImInfoEntry
filter: ImInfoEntry x (string -> boolean) -> ImInfoEntry
```

These operations affect only the statuses of an `ImInfoEntry`, not its destination.

(a) Of the four kinds of ADT operations, what kind(s) of operations is `ImInfoEntry.map`? Leave extra boxes blank:

(b) Use `map` to replace every English status message found in the `translations` map below with its corresponding French translation.

```
let translations: Map<string, string> = new Map([["ON TIME",   "A LHEURE"
],
                                          ["CANCELED", "SUPPRIME"]]);

let train1: ImInfoEntry = parseImInfoEntry("MONTREAL|ON TIME,11:05 AM");
// train1 has statuses "ON TIME", "11:05 AM"

let train2: ImInfoEntry = train1.map((...MAP...));
// train2 has statuses "A LHEURE", "11:05 AM"
```

Write a function to replace **`(...MAP...)`** in the code above:

(c) Write a function that, if passed to **`filter`** (not `map`), would transform the stream of status messages in a way that cannot be a legal abstract value of the `ImInfoEntry` type.

Now suppose that a mutable information board entry `MutInfoEntry` also has a `map` operation:

```
map: MutInfoEntry x (string => string) => void
```

`MutInfoEntry.map` transforms all statuses subsequently returned by the entry, as shown in this example:

```
1  const toFrench: string => string = ...MAP...; // a correct answer to p
art (b) above
2  const train: MutInfoEntry = new MutInfoEntry("MONTREAL");
3  train.nextStatus();  // returns ""
4  train.map(toFrench);
5  train.setStatuses(["ON TIME","11:05 AM"]);
6  train.nextStatus();  // returns "A LHEURE"
7  train.nextStatus();  // returns "11:05 AM"
8  train.setStatuses(["CANCELED"]);
9  train.nextStatus(); // returns "SUPPRIME"
```

(d) What kind(s) of operation is `MutInfoEntry.map` ? Leave extra boxes blank:

To implement `map` , the rep of `MutInfoEntry` now has a third field:

```
private f: string => string;
```

and its abstraction function is (only relevant parts shown):

> AF(`destination` , `statuses` , `f`) = the info board entry with current status `f(statuses[0])` and looping through future statuses `f(statuses[1])` , ..., `f(statuses[statuses.length-1])` , `f(status[0])` , and so on... [rest of AF elided]

The `MutInfoEntry` methods are implemented to obey this AF and behave as shown in the code above.

(e) Write a function for the initial value of `f` for a new `MutInfoEntry` object.

(f) During which of the numbered lines in the example code above (d) is the `toFrench`

function called? List all line numbers that apply, or write NEVER if `toFrench` is never called. Note that this question is asking about **`toFrench`**.

(g) What should `MutInfoEntry`'s rep invariant comment say about `f`? Note that this question is asking about **`f`**.

Code

```
/**
 *  An information board entry that shows a destination (e.g. "WASHINGTON D
 *  and current status (e.g. "DELAYED") in a cycle of 1 to 4 statuses
 *  (e.g. [ "DELAYED", "NEW DEPRTURE", "11:55 AM" ]).
 *
 *  A valid destination is up to 16 characters, consisting only of
 *  upper-case letters A-Z, digits, colons, or spaces.
 *
 *  A valid status is up to 12 characters, consisting only of
 *  upper-case letters A-Z, digits, colons, or spaces.
 */
export interface ImInfoEntry {

    /** @return the destination*/
    destination(): string;

    /** @return the currently-shown status */
    status(): string;

    /** @return the entry with the same destination and statuses,
     *          showing the next status in the cycle  */
    nextEntry(): ImInfoEntry;

    /** @return number of statuses in the cycle, integer from 1 to 4 */
    size(): number;
}

/**
 * @param entry information board entry represented as a string according
 *        to the grammar in Problem 3
```

```
 * @return corresponding information board entry value
 */
export function parseImInfoEntry(entry: string): ImInfoEntry { ... }

/**
 *  An information board entry that shows a destination (e.g. "WASHINGTON D
 *  and cycles through a list of 1 to 4 statuses (e.g. [ "11:05 AM", "ON TI
 *  or [ "NOW BOARDING", "TRACK 3" ]).
 *
 *  A valid destination is up to 16 characters, consisting only of
 *  upper-case letters A-Z, digits, colons, or spaces.
 *
 *  A valid status is up to 12 characters, consisting only of
 *  upper-case letters A-Z, digits, colons, or spaces.
 */
export class MutInfoEntry {

    private statuses: string[] = [];

    // Abstraction function:
    //   <elided>
    // Rep invariant:
    //   - destination is a valid destination (defined above)
    //   - statuses has 1-4 elements, each of which is a valid status (defi

    /** Create a new information board entry with the given destination and
     *  a single empty status.
     *  @param destination a valid destination (defined above) */
    public constructor(public readonly destination: string) {
        statuses.push("");
    }

    /**  @return the destination */
    public destination(): string { return destination; }

    /** @return the next status to display, infinitely cycling through this
     *          info board entry's statuses in order  */
    public nextStatus(): string {
        const status: string = statuses.shift()
        statuses.push(status); // put it back on end so that statuses cycle
        return status;
    }

    /** Set the statuses. The first status in the list will be displayed ne
     *  @param statuses new statuses, a 1- to 4-item list of valid statuses
```

```
    public setStatuses(statuses: string[]): void {
        this.statuses = [...statuses];
    }
}
```