Principles of Programming Languages 212
Assignment 1

Responsible TA: Ruth Hochman

Submission Date: 25/3/2021

Part 0: Preliminaries

Structure of a TypeScript Project

Every TypeScript assignment will come with two very important files:

- `package.json` - lists the dependencies of the project.
- `tsconfig.json` - specifies the TypeScript compiler options.

Before starting to work on your assignment, open a command prompt in your assignment folder and run `npm install` to install the dependencies.

What happens when you run `npm install` and the file `package.json` is present in the folder is the following:

1. `npm` will download all required modules and their dependencies from the internet into the folder `node_modules`.
2. A file `package-lock.json` is created which lists the exact version of all the packages that have been installed.

What `tsconfig.json` controls is the way the TypeScript compiler (`tsc`) analyzes and typechecks the code in this project. We will use for all the assignments the strongest form of type-checking, which is called the "strict" mode of the `tsc` compiler.

Do not delete or change these files (*e.g.*, install new packages or change compiler options), as we will run your code against our own copy of those files, exactly the way we provide them.

If you change these files, your code may run on your machine but not when we test it, which may lead to a situation where you believe your code is correct, but you would fail to pass compilation when we grade the assignment (which means a grade of zero).

Testing Your Code

Every TypeScript assignment will have Mocha and Chai as dependencies for testing purposes. In order to run the tests, save your tests in the `test` directory in a file ending with `.test.ts` and run `npm test` from a command prompt. This will activate the execution of the tests you have specified in the test file and report the results of the tests in a very nice format.

An example test file `assignmentX.test.ts` might look like this:

```
import { expect } from "chai";
import { sum } from "../src/assignmentX";

describe("Assignment X", () => {
  it("sums two numbers", () => {
    expect(sum(1, 2)).to.equal(3);
  });
});
```

Every function you want to test must be `export`-ed, for example, in `assignmentX.ts`, so that it can be `import`-ed in the `.test.ts` file (and by our automatic test script when we grade the assignment).

```
export const sum = (a: number, b: number) => a + b;
```

You are given some basic tests in the `test` directory, just to make sure you are on the right track during the assignment.

What to Submit

You should submit a zip file called `<id1>_<id2>.zip` which has the following structure:

```
/
├── Part1.pdf
└── src
    ├── part2
    │   └── part2.ts
    └── part3
        ├── find.ts
        ├── state.ts
        ├── queue.ts
        └── stack.ts
```

Make sure that when you extract the zip (using `unzip` on Linux), the result is flat, *i.e.*, not inside a folder. This structure is crucial for us to be able to import your code to our tests. Also, make sure the file is a `.zip` file – not a RAR or TAR or any other compression format.

Part 1: Theoretical Questions

Submit the solution to this part as `Part1.pdf`. We can't stress this enough: the file *has to be a PDF file*.

1. Explain the following programming paradigms:
   (a) Imperative
   (b) Procedural
   (c) Functional

   How does the procedural paradigm improve over the imperative paradigm? How does the functional paradigm improve over the procedural paradigm?

2. Write the most specific types for the following expressions:
   (a) `(x, y) => x.some(y)`
   (b) `x => x.reduce((acc, cur) => acc + cur, 0)`
   (c) `(x, y) => x ? y[0] : y[1]`

3. Explain the concept of "abstraction barriers".

Part 2: Fun with TypeScript

Complete the following functions in TypeScript in the file `src/part2/part2.ts`. One of the assignment's dependencies is the Ramda library shown in class, and you may use it freely. Make sure to write your code using type annotations, and adhering to the Functional Programming paradigm, *i.e.*, use only `const` for variable declarations (which also means no loops), and no using `push`, `pop`, `shift`, `unshift`, `splice`, `sort`, `reverse`, `fill` on arrays.

You may use helper functions as much as you want, but they must follow the same constraints as above.

You are also given a helper function `stringToArray` which takes a string and returns an array of the characters that make up the string. For example:

```
stringToArray("Hello!"); // ==> [ 'H', 'e', 'l', 'l', 'o', '!' ]
```

You are encouraged to use Ramda's `pipe` function, which takes a list of functions and returns a function which "pipes" the functions one after the other. It is similar to `compose`, but the order of applications is reversed. For example:

```
import { pipe } from "ramda";

const f = pipe(
    (x: number) => x * x,
    (x: number) => x + 1,
    (x: number) => 2 * x
);

f(5); // ==> 52
```

Remember that it is crucial you do *not* remove the `export` keyword from the code in the given template.

Question 1

Write a function `countVowels` that takes a string as input and returns the number of vowels in the text. Reminder: vowels are one of 'a', 'e', 'i', 'o', 'u', either uppercase or lowercase. For example:

```
countVowels("This is SOME Text"); // ==> 5
```

Question 2

Write a function `runLengthEncoding` which takes a string as input and returns a "compressed" version of it, where identical consecutive characters appear as the character followed by its count. For example:

```
runLengthEncoding("aaaabbbccd"); // ==> 'a4b3c2d'
```

Question 3

Write a function `isPaired` that takes a string and returns whether the parentheses (`{`, `}`, `(`, `)`, `[`, `]`) in the string are paired. For example:

```
isPaired("This is ([some]) {text}"); // ==> true
isPaired("This is ]some[ (text)"); // ==> false
```

Part 3: A Fistful of Monads

What is a monad? According to Wikipedia: "In functional programming, a **monad** is a design pattern that allows structuring programs generically while automating away boilerplate code needed by the program logic. Monads achieve this by providing their own data type (a particular type for each type of monad), which represents a specific form of computation, along with one procedure to wrap values of any basic type within the monad (yielding a **monadic value**) and another to compose functions that output monadic values (called **monadic functions**)."

During the semester, we will use two such monads: the `Result<T>` monad (used to deal with computations that may fail) and the `Optional<T>` monad (used when a computation might not yield a value).

The main function used with monads is the `bind` function (also called `chain` and `flatmap` in other languages, and by the `>>=` operator in Haskell). `bind` is used to compose two monads in a way that makes sense in the context of the specific monad.

In your solution, in addition to `Result<T>` and `State<S, A>`, use only TypeScript constructs and types that are functional, *i.e.*, under the same constraints as in Part 2, and that were covered in class.

When All Else Fails

We are going to get very familiar with the `Result<T>` monad in our interpreters' code, so to get up and running with using it and getting to know its constructors and its `bind` function, we will convert a function that throws an error to a function that uses `Result<T>`.

1. Read the code in `src/lib/result.ts`. Try to understand how `bind` composes two `Result<T>` values.
2. In `src/part3/find.ts`, you are given this code:

```
/* Library code */
const findOrThrow = <T>(pred: (x: T) => boolean, a: T[]): T => {
    for (let i = 0; i < a.length; i++) {
        if (pred(a[i])) return a[i];
    }
    throw "No element found.";
}

/* Client code */
const returnSquaredIfFoundEven_v1 = (a: number[]): number => {
    try {
        const x = findOrThrow(x => x % 2 === 0, a);
        return x * x;
    } catch (e) {
        return -1;
    }
}
```

   (a) Write a *generic pure function* `findResult` which takes a predicate and an array, and returns an `Ok` of the first element that the predicate returns `true` for, or a `Failure` if no such element exists.

   (b) Only using `bind`, write a function `returnSquaredIfFoundEven_v2` that uses `findResult` to return an `Ok` of the first even value squared, or a `Failure` if no even numbers exist.

   (c) Only using `either`, write a function `returnSquaredIfFoundEven_v3` that uses `findResult` to return the first even value squared, or a -1 if no even numbers exist.

"L'État, c'est moi."

Many computations involve state in some way or another: random number generation, defined variables, mutable data structures, etc. As we already know, managing state with mutation is contrary to the functional paradigm. To avoid managing state with mutation, we introduce the State monad.

Let's explore the concrete example of random number generation. Suppose we have a *pure* function `random` for generating pseudo-random numbers given a seed value:

```
const random = (seed: number): number =>
    (80189 * seed + 190886) % 1052010;
```

To use it, we just need to keep track of the new seed coming out of the function:

```
const seed0 = 42;
const seed1 = random(seed0);
console.log(seed1); // ==> 402794
const seed2 = random(seed1);
console.log(seed2); // ==> 1027932
```

This isn't so bad, but consider the following client code to roll a die:

```
const randomDie = (seed: number): number => {
    const newSeed = random(seed);
    return 1 + Math.floor(newSeed / 1052010 * 6);
};

const seed = 42;
const die1 = randomDie(seed);
const die2 = randomDie(seed);
console.log(die1, die2); // ==> 3 3
```

This is no good, we are getting the same die roll, because the seed didn't change. What we need to do is keep track of the new seed, which is the *state* of the pseudo-random number generator. We can change the `randomDie` function like so:

```
const randomDie = (seed: number): [number, number] => {
    const newSeed = random(seed);
    const die = 1 + Math.floor(newSeed / 1052010 * 6);
    return [newSeed, die];
};

const seed0 = 42;
const [seed1, die1] = randomDie(seed0);
const [seed2, die2] = randomDie(seed1);
console.log(die1, die2); // ==> 3 6
```

Almost perfect! We are getting different die rolls, but passing the seeds around when we compose multiple calls of the function is cumbersome and risk-prone. We want to abstract the way successive calls to a state are threaded from one call to the next in a *state composition* function. This is what is defined in the `bind` function associated with this monad.

To this end, we will define the State monad to be a function from the initial state of type `S` to a pair of the new state and result of the computation: `type State<S, A> = (initialState: S) => [S, A];`

Note: the type `[S, A]` is a type that represents an array of size 2 where the first element is of type `S` and the second element is of type `A`. This is known in TypeScript as a "tuple".

The name "State monad" can be a little confusing, as the `State<S, A>` instance is actually a "state processor", and `S` is the type of the internal state we are keeping.

Back to our die-rolling function: assume we have implemented `die` of type `State<number, number>`, and the monadic composition function `bind` (we are not showing you how to do this here). We can now implement a function to roll two dice without explicitly passing around the seed like so:

```
const rollTwoDice: State<number, [number, number]> =
    bind(die, die1 => bind(die, die2 => s => [s, [die1, die2]]));

const seed = 42;
const [newSeed, dice] = rollTwoDice(seed);
console.log(dice); // ==> [ 3, 6 ]
```

Excellent! We have the correct values for the dice as before, and we didn't have to interleave the seed between the computations: we only needed to give the initial seed value.

Now that we have the motivation for the State monad, implement in `src/part3/state.ts` the function `bind`. Its signature is:

```
const bind: <S, A, B>(state: State<S, A>, f: (x: A) => State<S, B>) => State<S, B>
```

Think how `bind` should behave, and let its type guide your implementation.

Now that we have the State monad and its `bind` function defined, we can put them to good use. We saw in class how to implement a functional stack, but that required keeping track of intermediate stacks. We will now implement a functional queue and stack using the State monad.

Implement the following in `src/part3/queue.ts`:

1. `enqueue` is a function which takes a number `x` and returns a State that adds `x` to the queue.
2. `dequeue` is a State which dequeues a number from the queue and returns it.

Also in `src/part3/queue.ts`, implement `queueManip` which does the following *only using* `bind`, `enqueue`, *and* `dequeue`!

1. Dequeues a number `x` from the queue
2. Enqueues `2 * x`
3. Enqueues `x / 3`
4. Dequeues

For example (remember that `queueManip` is a State):

```
queueManip([6, 7, 8]); // ==> [ [ 8, 12, 2 ], 7 ]
```

In `src/part3/stack.ts`, implement the following:

1. `push` is a function which takes a number `x` and returns a State that adds `x` to the stack.
2. `pop` is a State which pops a number from the stack and returns it.

Same as with our queue, add a definition for `stackManip` in `src/part3/stack.ts` which does the following *only using* `bind`, `push`, *and* `pop`!

1. Pops a number `x`
2. Pushes `x * x`
3. Pops a number `y`
4. Pushes `x + y`

For example:

```
console.log(stackManip([4, 5, 6])); // ==> [ [ 20, 5, 6 ], undefined ]
```

Note: we won't check cases of dequeuing an empty queue, or popping an empty stack.

Good Luck and Have Fun!