Principles of Programming Languages 232
Assignment 1

Responsible TA: Omri Hefez

Submission Date: 4/4/2023 23:59

Part 0: Preliminaries

Structure of a TypeScript Project

The template of every TypeScript assignment will contain two very important files:

- `package.json` - lists the packagae 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 provided 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 `Jest` as a global dependency for testing purposes (so no need to import it). 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 { sum } from "../src/assignmentX";

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

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

```
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
```

Make sure that when you extract the zip (using `unzip` on Linux), the result is flat, *i.e.*, not inside a folder (the file `part1.pdf` is in the root directory). 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. (a) Explain the following programming paradigms:
      i. Imperative
      ii. Procedural
      iii. Functional

   (b) How does the procedural paradigm improve over the imperative paradigm?

   (c) How does the functional paradigm improve over the procedural paradigm?

2. Convert the following function to adhere to the Functional Programming paradigm, using some or all the functions we saw in class: map, filter, reduce:

```
function sumEven(numbersAsString: string[]): number {
    let sum = 0
    for (let i = 0; i < numbersAsString.length; i++) {
        let num: number = parseInt(numbersAsString[i], 10)
        if (num % 2 == 0) {
            sum += num;
        }
    }
    return sum;
}
```

3. 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]`

   (d) `(f,g) => x => f(g(x+1))`

4. Explain the concept of "abstraction barriers".

Part 2: TypeScript & Functional Programming

In part 2 and part 3, replace every instance of the word `undefined` with your code or typing.

Write the functions for the following questions in TypeScript in the file `src/part2/part2.ts`. One of the assignment's dependencies is the Ramda library shown both in class and practical session, 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 `countLetters` that takes a string as input and returns a map where the key is a letter (lowercase), and the value is the number of times that letter appeared in the string, either uppercase or lowercase. Be sure to exclude whitespaces. For example:

```
countLetters("A banana"); // ==> { a: 4, b: 1, n: 2 }
```

Question 2

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
```

Question 3

Given the following type:

```
type WordTree = {
    root: string;
    children: WordTree[];
}
```

Complete the function `treeToSentence`. This funtion gets a WordTree as an argument and returns a string made of all the words in the tree, concatenated to each other, in depth-first order, separated by a single space.

Example:

```
const t1: WordTree = {
    root: "Hello",
    children: [
        {
            root: "students",
            children: [
                {
                    root: "how",
                    children: []
                }
            ]
        },
        {
            root: "are",
            children: []
        },
        {
            root: "you?",
            children: []
        },
    ]
}

treeToSentence(t1); // ==> Hello students how are you?
```

Part 3: A Gentle Introduction to 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` on 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` on the first even value squared, or a `Failure` if no even numbers exist.

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

Good Luck and Have Fun!