Principles of Programming Languages 192
Assignment 1

Responsible TA: Naama Dayan

Submission Date: 24/3/2019

Part 1: Theoretical Questions

Submit the solution to this part as `Q1.pdf`.

1. What is "*referential transparency*"? Give an advantage and an example.

2. Convert the following function to adhere to the FP paradigm (using the functions we saw in class: `map`, `filter`, `reduce`):

```
function averageSalaryOver9000(employees) {
    let salarySum = 0;
    let counter = 0;
    for (let i = 0; i < employees.length; i++) {
        if (employees[i].salary > 9000) {
            salarySum += employees[i].salary;
            counter++;
        }
    }
    return salarySum / counter;
}
```

Example input:

```
[{name: "Moshe", salary: 5600}, {name: "Dorit", salary: 7800},
 {name: "Naama", salary: 10000},{name: "Dani", salary: 9600}]
```

Example output: 9800

3. Write the most specific TypeScript type expression associated to each of the following expressions:

3.1. `[{name: 'Sara', degrees: [{name: 'CS', years: 3}, {name: 'Biology', years: 4}]}]`

3.2. `(f, g, h) => x => f(g(h(x + 1)))`

3.3. `(pred, arr) => arr.map(pred).reduce((acc, cur) => acc || cur, false)`

3.4. `(f, a) => a.reduce((acc,cur) => acc + f(cur), 0)`

Part 2: TypeScript Programming

Complete the following functions in TypeScript in the file `Q2.ts`. Make sure to write your code using type annotations, and adhering to the FP paradigm.

Question 1

Given the following interface:

```
interface NumberTree {
    root: number;
    children: NumberTree[];
}
```

Complete the function `sumTreeIf`. This function gets a NumberTree and a predicate as parameters and returns the sum of all tree nodes which satisfy the predicate.

Examples:

```
const t1: NumberTree = {
    root: 1,
    children: [
        {
            root: 2,
            children: [
                {
                    root: 5,
                    children: []
                }
            ]
        },
        {
            root: 3,
            children: []
        },
        {
            root: 4,
            children: []
        }
    ]
};

sumTreeIf(t1, n => true); // ==> 15
sumTreeIf(t1, n => n % 2 === 0); // ==> 6
```

Question 2

Given the following interface:

```
interface WordTree {
    root: string;
    children: WordTree[];
}
```

Complete the function `sentenceFromTree`. This function gets a WordTree as a parameter and returns a string made from all the words in the tree, concatenated to each other, in depth-first order.

Example:

```
const t2: WordTree = {
    root: "Hello",
    children: [
        {
            root: "students,",
            children: [
                {
                    root: "how",
                    children: []
                }
            ]
        },
        {
            root: "are",
            children: [
                {
                    root: "you",
                    children: [
                        {
                            root: "doing",
                            children: []
                        }
                    ]
                },
                {
                    root: "this",
                    children: [
                        {
                            root: "fine",
                            children: []
                        }
                    ]
                }
            ]
        },
        {
            root: "assignment?",
            children: []
```

```
        }
    ]
};

sentenceFromTree(t2); // ==> "Hello students, how are you doing this fine assignment?"
```

Question 3

Given the following interfaces:

```
interface Grade {
    course: string;
    grade: number;
}

interface Student {
    name: string;
    gender: string;
    grades: Grade[];
}

interface SchoolClass {
    classNumber: number;
    students: Student[];
}

type School = SchoolClass[];
```

Complete the following functions:

1. Write a function `hasSomeoneFailedBiology` that gets a School and checks if there is a student who failed biology (failure means a grade < 56).
2. Write a function `allGirlsPassMath` that gets a School and verifies all girls at school passed the course math (grade ≥ 56).

Examples:

```
let school1 = [
    {
        classNumber: 1,
        students: [
            {
                name: "Moshe",
                gender: "Male",
                grades: [
                    {course: "math", grade: 38},
                    {course: "literature", grade: 68},
                    {course: "biology", grade: 57}
                ]
            },
            {
                name: "Ziva",
                gender: "Female",
                grades: [
                    {course: "math", grade: 67},
                    {course: "literature", grade: 68},
                    {course: "biology", grade: 100}
                ]
            }
        ]
    },
    {
        classNumber: 2,
        students: [
            {
                name: "Ifat",
                gender: "Female",
                grades: [
                    {course: "math", grade: 68},
                    {course: "literature", grade: 68},
                    {course: "biology", grade: 90}
                ]},
            {
                name: "Tomer",
                gender: "Male",
                grades: [
                    {course: "math", grade: 70},
                    {course: "literature", grade: 68},
                    {course: "biology", grade: 100}
                ]
            }
        ]
    }
]

hasSomeoneFailedBiology(school1); // ==> false
allGirlsPassMath(school1); // ==> true
```

Question 4

We saw in lectures the "*disjoint union*" idiom with the `Shape` type. Reminder:

```
interface Point2D {
    x: number;
    y: number;
}

type Shape = Circle | Rectangle | Triangle;

interface Circle {
    tag: "circle";
    center: Point2D;
    radius: number;
}

interface Rectangle {
    tag: "rectangle";
    upperLeft: Point2D;
    lowerRight: Point2D;
}

interface Triangle {
    tag: "triangle";
    p1: Point2D;
    p2: Point2D;
    p3: Point2D;
}
```

In order to accomodate for polymorphic functions, we add to `Shape` type constructors:

```
const makeCircle = (center: Point2D, radius: number): Circle =>
    ({tag: "circle", center: center, radius: radius});
const makeRectangle = (upperLeft: Point2D, lowerRight: Point2D): Rectangle =>
    ({tag: "rectangle", upperLeft: upperLeft, lowerRight: lowerRight});
const makeTriangle = (p1: Point2D, p2: Point2D, p3: Point2D): Triangle =>
    ({tag: "triangle", p1: p1, p2: p2, p3: p3});
```

and type predicates:

```
const isCircle = (x: any): x is Circle => x.tag === "circle";
const isRectangle = (x: any): x is Rectangle => x.tag === "rectangle";
const isTriangle = (x: any): x is Triangle => x.tag === "triangle";
```

Here is an example of the `area` function we saw in class, using type predicates:

```
const area = (s: Shape): number =>
    isCircle(s) ? s.radius * s.radius * 3.14 :
    isRectangle(s) ? (s.upperLeft.x - s.lowerRight.x) * (s.upperLeft.y - s.lowerRight.y) :
    isTriangle(s) ? 0.5 * (s.p1.x * s.p2.y + s.p2.x * s.p3.y + s.p3.x * s.p1.y
                         - s.p2.x * s.p1.y - s.p3.x * s.p2.y - s.p1.x * s.p3.y) :
    0;
```

`Q2.java` includes an interface `PaymentMethod` which declares one method `charge` and three implementing classes: `Cash`, `DebitCard`, and `Wallet`. There is also a `YMDDate` class with a static function to check whether a date comes before another date. The method `charge` takes a `PaymentMethod` and deducts the specified amount from the payment method (or the most it can, in the case there isn't enough money). The method returns how much money is left to charge. For example, if we have a debit card which is not yet expired, and has 500 shekels on it, calling `charge` on it with today's date and 700 shekels will return 200 shekels, and leave the debit card with 0 shekels.

This Java implementation illustrates how polymorphism is achieved in the Object-Oriented paradigm - using language constructs such as Java Interface and Classes, virtual functions and mutation of private data members in Classes.

Your task is to re-write the same program, but adopting it to the FP paradigm. In this paradigm, we achieve polymorphism by using the disjoint union idiom for types and we avoid mutation altogether by adopting the persistent data types method (see here). In this approach, instead of mutating the internal state of compound data structures (the equivalent of classes in Java), the operations that would mutate (and would be a function that returns a void value) is instead a constructor that returns a new value of the object.

Using the *disjoint union* idiom, implement in `Q2.ts` the `PaymentMethod`, `Cash`, `DebitCard`, and `Wallet` types, and write a *functional* version of `charge` which returns a `ChargeResult`. A `ChargeResult` holds how much money is left to charge, and a new wallet. Using the previous example, the functional version of `charge` will return a `ChargeResult` with 200 shekels, and a new wallet with a debit card that has 0 shekels on it.

The interfaces for `YMDDate` and `ChargeResult`, as well as the implementation for `comesBefore`, are given in `Q2.ts`.

Examples:

For the payment method:

```
const wallet1 = makeWallet([
    makeCash(4500),
    makeDebitCard(3000, {year: 2020, month: 7, day: 31}),
    makeDebitCard(300, {year: 2020, month: 7, day: 31})
]);
```

The call `console.log(JSON.stringify(charge(wallet1, 7000, {year: 2019, month: 3, day: 7})));` will print:

```
{
    "amountLeft":0,
    "wallet":{
        "tag":"wallet",
        "paymentMethods":[
            {
                "tag":"cash",
                "amount":0
            },
            {
                "tag":"dc",
                "amount":500,
                "expirationDate":{
                    "year":2020,
                    "month":7,
                    "day":31
                }
            },
            {
                "tag":"dc",
                "amount":300,
                "expirationDate":{
                    "year":2020,
                    "month":7,
                    "day":31
                }
            }
        ]
    }
}
```

For the payment method:

```
const wallet2 = makeWallet([
    makeCash(4500),
    makeDebitCard(3000, {year: 2010, month: 7, day: 31}), // note the expiration date
    makeDebitCard(300, {year: 2020, month: 7, day: 31})
]);
```

The call `console.log(JSON.stringify(charge(wallet2, 7000, {year: 2019, month: 3, day: 7})));` will print:

```
{
    "amountLeft":2200,
    "wallet":{
        "tag":"wallet",
        "paymentMethods":[
            {
                "tag":"cash",
                "amount":0
            },
            {
                "tag":"dc",
                "amount":3000,
                "expirationDate":{
                    "year":2010,
                    "month":7,
                    "day":31
                }
            },
            {
                "tag":"dc",
                "amount":0,
                "expirationDate":{
                    "year":2020,
                    "month":7,
                    "day":31
                }
            }
        ]
    }
}
```

Part 3: Fun with Scheme

Complete the following functions in Scheme in the file `Q3.rkt`.

Question 1

The function `ngrams` takes a list of symbols and a number $n \leq$ the length of the list and returns a list of consecutive n symbols. For example:

- `(ngrams '(the cat in the hat) 3) ;; ==> '((the cat in) (cat in the) (in the hat))`
- `(ngrams '(the cat in the hat) 2) ;; ==> '((the cat) (cat in) (in the) (the hat))`

Question 2

The function `ngrams-with-padding` takes a list of symbols and a number $n \leq$ the length of the list and returns a list of consecutive n symbols, padding with `*` if necessary. For example:

- `(ngrams-with-padding '(the cat in the hat) 3)`
  `;; ==> '((the cat in) (cat in the) (in the hat) (the hat *) (hat * *))`
- `(ngrams-with-padding '(the cat in the hat) 2)`
  `;; ==> '((the cat) (cat in) (in the) (the hat) (hat *))`

Have Fun and Good Luck!