Part 1: Theoretical questions

a.

i. The imperative paradigm is about solving the problem step by step. Meaning, there is no use in models or methods that were written before to help the programmer solve parts from his problem on the way to the final code. The focus of imperative programming is how the problem should be solved and the programmer must code each step. The procedural and the object-oriented paradigms are examples for imperative paradigms.

ii. The procedural programming paradigm is about separating your code to a 'group' of small code scopes. Every scope of code is called "Procedure" and the main idea is to create a reasonable order for the program and make every procedure independent. The procedures consist of local variables, methods and orders which are defined by published interfaces which tell the programmer the basic laws of the procedure he wishes to make, like signatures, types and more. Those interfaces are being written with no influence on the programmer's code directly.

iii. Functional programming is a programming paradigm focused on using pure functions, immutable data, and higher-order functions. Pure functions always produce the same output for the same input, avoiding surprises and complications. Immutable data cannot change after creation, encouraging clarity and simplicity. Higher-order functions allow functions to be passed around like any other value, promoting flexibility and abstraction. Other principles include recursion for breaking down problems, lazy evaluation for efficiency and treating functions as fundamental elements.

b. The procedural paradigm improves over the imperative paradigm by making the code easier to understand. Because of the separation of the code to procedures, it becomes more organized and because of that, it's more simple to debug it, maintain it, and change it in case needed. In addition, by using methods and interfaces that have been written before instead of solving any subproblem step by step, the programmer saves a lot of time by avoiding 'black work' and repeating code.

c. Functional code is written in a way that is very similar to math. Because of that, and that there are no side-effects, the correctness of code is much easier to check and the code is also much more readable, compared to procedural code. Another advantage is that The final result of a computation of expressions does not depend on the order in which sub-expressions are evaluated, making it easier to exploit parallelism and improve performance with less correctness issues and concerns. In addition, the abstraction and the separation between how data is represented from how it is used, make it easier to manage complex types, structures and systems.

2.

```
import { map } from "ramda";
import { filter } from "ramda";
import { reduce } from "ramda";

type Product = {
    name: string;
    price: number;
    discounted: boolean;
}

const DiscountedProductAveragePrice = (inventory: Product[]) : number =>
    (
        inventory.length === 0 ? 0 : (((inventory.filter((item: Product) => item.discounted))
        .reduce((sum : number, item : Product) => sum + item.price , 0))
        / (inventory.filter((item: Product) => item.discounted)).length)
    );
```

3.

```
const a = <T>(x : T[] , y : (value: T) => boolean) : boolean => x.some(y);

const b = (x : number[]) => (x.reduce((acc : number, cur : number) : number  => acc + cur), 0);

const c = <T>(x : boolean, y : T[]) : T => x ? y[0] : y[1];

const d = <T, S>(f: (value: T) => S, g: (value: number) => T) => (x : number) : S => f(g(x+1));
```