PPL – Assignment1 – Part1

By Yair Derry & Gal Noy

Question 1:

(a)

i. Imperative – The program is a series of commands/steps that should be executed to reach a specific goal.
ii. Functional – The program is an expression/a series of expressions and not a series of commands. Executing the program means finding expression's value and not performing actions. Functions are also expressions, means there are no "side-effects" – everything is Immutable and there are no variable assignments.
iii. Procedural – The program is broken up into small, easily testable procedures/units, which can be called from different parts of the code.

(b)

As opposed to imperative paradigm, the procedural paradigm adds a layer of structure and organization to the code, by breaking it to smaller, manageable pieces of code. This allows easier testing, cleaner and organized code, and encapsulation (parts of the code may be hidden while others not).

(c)

As opposed to procedural paradigm, the functional paradigm emphasizes immutability, pure functions, higher-order functions, lazy evaluation, and concurrency. This approach results in code that is more robust, maintainable, and scalable than procedural programming.

Question 2:

```
function sumEven(numbersAsString: string[]): number {
    const stringToNumber = (x: string) : number => parseInt(x, 10);
    const mapNumbers = (strings : string[]) : number[] => map(stringToNumber, strings);
    const isEven = (x: number) : boolean => x%2===0;
    const filterEvens = (numbers: number[]) : number[] => filter(isEven, numbers);
    const sum = (numbers : number[]) : number => reduce((acc: number, curr: number) :
number => acc + curr, 0, numbers);

    return sum(filterEvens(mapNumbers(numbersAsString)));
}
```

Question 3:

```
<T>(x: T[], y: (z: T) => boolean): boolean => x.some(y); // (a)
(x: number[]): number => x.reduce((acc, cur) => acc + cur, 0); // (b)
<T>(x: boolean, y: T[]): T => (x ? y[0] : y[1]); // (c)
<T, U>(f: (a: T) => U, g: (b: number) => T) => (x: number): U => f(g(x + 1)); // (d)
```

Question 4:

The concept of abstraction barriers refers to a technique in software engineering that aims to separate different levels of abstraction within a system. By using abstraction barriers, software developers can create well-defined interfaces between different parts of a system. This allows them to change the implementation of one part of the system without affecting other parts of the system that rely on it. That helps ensure that different parts of a system can be developed, tested, and maintained independently of each other.