PPL TASK 1 – Ben Kapon 206001547 and Ori Cohen 314804741

Part 1: (Each task is in different page)

1.1

1)

a) **Imperative Paradigm**: In the imperative paradigm, a program is written as a sequence of commands. Running the program means executing these commands one after another, modifying the program's state step by step.
Examples: Java, C++, Python.

b) **Procedural Paradigm**: The procedural paradigm builds on the imperative paradigm by organizing code into procedures (functions).
These procedures can be called from different parts of the program, enabling reuse of logic.
This improves readability, modularity, and structure.
Examples: C, Java, Python.

c) **Function Paradigm**: The functional paradigm views a program as a set of expressions, rather than a sequence of commands. Execution is seen as computing results, not as performing a sequence of commands.
Functions are considered expressions as well, and calling a function means evaluating an expression.
This paradigm avoids side effects and mutable state, leading to clearer, more predictable programs.
Examples: Scheme, Haskell, JavaScript (partially).

2) The procedural paradigm improves on the imperative paradigm by organizing code into reusable procedures (functions). This makes the program modular, improving readability, maintainability, and reusability of the code. By breaking down the program into smaller, more manageable parts, it reduces code duplication and increases clarity.

3) In The functional paradigm improves over the procedural paradigm by eliminating side effects and mutable state. This makes the code more predictable, easier to reason about, and easier to test.
It also enables higher-order functions and function composition, which leads to more abstract and concise code.

1.2

```
type Transaction = {
        category: string;
        price: number;
        quantity: number;
}

function  calculateRevenueByCategoryFP(transactions: Transaction[]): Record <string, number> {
        const moreThanFive = (t: Transaction): boolean => t.quantity > 5;

        const afterDiscount:{ category: string; total: number }[] =
                transactions.map((t: Transaction): { category: string,  total: number }=> ({
                category: t.category,
                total: (moreThanFive(t) ? 0.9 : 1) * t.price * t.quantity
        }));

        const filteredFromUnder50: { category: string; total: number }[] =
                afterDiscount.filter((t: { category: string; total: number }) : boolean => t.total >= 50 );

        const revenueByCategory: Record<string, number> =
                filteredFromUnder50.reduce((acc: Record<string, number>, t: { category: string; total: number })
                : Record<string, number> => ({
                        ...acc,
                        [t.category]: (acc[t.category] ?? 0) + t.total
                        }),
                        {}
        );
        return revenueByCategory;
}
```

* The spread operator (...) copies all properties from an existing object into a new object, allowing non-destructive updates.

** The nullish coalescing operator (??) returns the right-hand value only if the left-hand value is null or undefined.

1.3

1. (x, y) => x.some(y) ➔ <T> (x:T[], y:(item:T) => boolean) => boolean
2. x => x.map(y => y * 2) ➔ (x:number[]) => number[]
3. (x, y) => x.filter(y) ➔ <T>(x:T[], y:(item:T) => Boolean) => T[]
4. x => x.reduce((acc, cur) => acc + cur, 0) ➔ (x: number[]) : => number
5. (x, y) => x ? y[0] : y[1] ➔ <T>(x:boolean, y:T[]) => T
6. (f,g) => x => f(g(x+1)) ➔ <T1,T2> (f: (a: T2) => T1, g: (b: number) => T2) => (x: number) => T1