# Principles of Programming Languages – Assignment1

Question 1.1:

1)
   A) Imperative languages known as languages where program is a sequence of commands, and the execution of the program involves performing these commands one after the other. In other words, you specify the computer a series of commands. Also, imperative Programming focuses on changing the program's state using statements like loops and variables. We can think of it like writing a recipe (first do this and then do that…).
   B) Procedural languages use functions (procedures) that can be called from various locations within the code to organize it. It allows breaking the program into smaller chunks of code and make the code more readable. Furthermore, it allows for local variables and introduces scopes. Like Imperative languages the program follows a top-down structure.
   C) Functional Programming focuses on the "What to do" and not the "How to do" things. Its uses pure functions (meaning there are no side effects) that do not change any of the variables or the state of the program. In another words, all the variables in the program are immutable and the functions return a new instance of something without changing any variable existed. Like the Procedural languages it allows for using functions (pure once obviously) and follows a top-down structure.
2) The procedural paradigm improves over the imperative paradigm in few aspects:
   - Code organization and reusability:
     Procedural paradigm allows encapsulating blocks of code into reusable procedures breaking larger task into smaller manageable units. Makes the code easier to understand, maintain and debug.
   - Abstraction:
     Functions allow to abstract away specific sequences of instruction behind a function name instead of repeating the same set of commands multiple times. A programmer can define a procedure once and then call it whenever that functionality is needed.
   - Modularity:
     Breaking a program into functions simplifies the development process of complex programs. Also, each procedure can be developed and tested independently and that allows easier debugging and maintaining
3) The functional paradigm improves over the procedural paradigm in few aspects:
   - Absence of side effects:
     Because of using only pure functions that can't modify the state of the program, it allows for more predictable code. It means that for the same input it will always produce the same output and that makes the code easier to debug and verify.
   - Enhanced Parallelism and Concurrency:
     Pure functions do not modify shared data or state, so there is no need for complex locking mechanisms to prevent race conditions.

- Abstraction and composition:
  composing smaller, reusable pure functions allows for functions to take other functions as arguments creating Higher-Order functions. That enables powerful abstractions and more concise and expressive code and can lead to more modular and maintainable programs.

Question 1.2:

```
function calculateRevenueByCategoryFP(transactions: Transaction[]) : Record<string, number> {
    const mapData: (transaction: Transaction) => [category: string, price: number] =
        (transaction: Transaction)  : [category: string, price: number] =>
            [transaction.category, transaction.quantity > 5 ? transaction.price * transaction.quantity * 0.9 : transaction.price * transaction.quantity];
    const calculatedData = transactions.map(mapData);
    const filteredData = calculatedData.filter((transactionData: [category: string, price: number]) => transactionData[1] >= 50)
    const reduce = (acc: Record<string, number>, [category, price]: [category: string, price: number]) => {
        acc[category] = (acc[category] || 0) + price;
        return acc;
    }

    return filteredData.reduce(reduce, {});
}
```

Question 1.3:

```
1.  <T>(x: T[], y: (item: T) => boolean) => boolean
2.  (x: number[]) => number[]
3.  <T>(x: T[], y: (item: T) => boolean) => T[]
4.  (x: number[]) => number
5.  <T>(x: boolean, y: [T, T, ... T[]]) => T
6.  <T,U>(f: (arg: T) => U, g: (x: number) => T) => (x: number) => U
```