Assignment 1

Question 1

A. Imperative :

imperative programming is a programming paradigm that uses statements that change a program's state. An imperative program consists of commands for the computer to perform. Control flow is an explicit sequence of commands - mainly defined in contrast to “declarative". Imperative programming focuses on describing how a program operates.

B. Procedural:

Procedural programming is derived from imperative programming. In procedural programming we can define procedures that can be used in different parts of the program. It is essentially Imperative programming organized around hierarchies of nested procedure calls.

C. Functional:

The program is a phrase, or series of phrases, and not a sequence of commands. Running the program is a calculation of the expressions, i.e. finding its value, rather than executing the commands.

Functional programming strongly adheres to the concept of immutability. Computation proceeds by (nested) function calls that avoid any global state mutation and through the definition of function composition.

The procedural programming improves over the imperative programming by allowing us to organize the code better, avoid redundant code repetition and make the program more modular and adaptive. Separation of the code to procedures improves its readability and makes it easier to understand.

The Functional programming improves over the procedural programming in that functional programming has no side effects, therefore we do not change the state of the program or memory. This allows safe use of concurrency. In addition, the functions behave like mathematical functions so we can prove and justify the correctness of a program somewhat easily. Another improvement is that FP provides a level of abstraction, it emphasizes the isolation of abstract types that clearly separate implementation from interface.

Question 2

```
const filterOver60: (grades: number[]) => number[] =
        (grades: number[]) => grades.filter((x) => x > 60)

const sumArray: (grades: number[]) => number = (grades: number[]) =>
        grades.reduce((num1, num2) => (num1 + num2), 0)

const averageGradesOver60: (grades: number[]) => number =
(grades: number[]) =>
        sumArray(filterOver60(grades)) / filterOver60(grades).length;
```

Question 3

a. $< T > (\ x: T[], y: (k: T) => \ boolean) => boolean.$
b. $(x: number[]) \ => \ number.$
c. $< T > (\ x: boolean, y: T[]) \ => \ T.$
d. $< T, E > \ (f: (T) => E\ , g: (number) => \ T) => (x: number\ ) => E$

$$= f \circ g \circ h(x) \mid h: (x: number) => number = (x: number) => x + 1$$

Question 4

Abstraction barrier : the concept of Abstraction barrier is the separation between the use of a function and its implementation. Complex function can be "broken down" to small parts that can be used without necessarily knowing their implementation. Abstraction barrier enable complex system to be built with higher confidence and productivity. Lower layers are tested in isolation from higher layers and provide a sturdier foundation than trying to build a complex system.