Part1:

- **Imperative** — Control flow is an explicit sequence of commands - mainly defined to achieve the result in contrast to "declarative" which focuses on *what* the program should accomplish without specifying *how* the program should achieve the result.
  The imperative **step by step** instructions describe explicitly which steps are to be performed in what order to obtain the desired solution at the end.

- **Procedural** — Imperative programming organized around hierarchies of nested procedure calls. derived from imperative programming. Procedures simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

- **Functional** —functional programming is a programming paradigm that is most similar to evaluation of expressions. In functional programming, a program is viewed as an expression.
  where programs are constructed by applying and composing functions. It is a declarative programming paradigm has explained above.
  Functions can be passed as arguments to other functions, Functions can be returned as a result of a function.
  Computation proceeds by (nested) function calls that avoid any global state mutation and through the definition of function composition. Furthermore, in this paradigm there is no side effects.

few benefits of procedural programming:

- The code becomes reusable.
- Writing modular code is possible.
- It is easier to keep track of the control flow.

Few benefits of functional programming:

- Code verification – function verification capability, each function separately.
- Parallelism – relatively easy due to the fact that the HEAP is read-only.
- Abstraction/design - with the help of high-order functions (the feature that functions are actually objects) the code can be arranged elegantly.

2)

a) (x, y) => x.some(y)

Type - <T> (x: T[], y: ((value: T, index: number, array: T[]) => boolean) => boolean)

b) x => x.reduce((acc ,curr)=> acc + curr , 0)

Type – x : number []:number => x.reduce((acc : number ,curr : number)=> acc + curr , 0)

c) (x, y) => x ? y[0] : y[1]

Type - <T>(x : boolean , y : T[]) : T => x ? y[0] : y[1]

3)

Abstraction barriers divides the world into two parts – clients and implementors, as the clients are the program consumers and the implementors are the program builders.

While the clients shouldn't have access to the code and just need to trust the software correctness the implementors should have access to the code and must distrust the software and look for bugs.

By this discipline, we can enforce abstraction barriers between procedures – high level procedures only call low level procedures. Now, you no longer have to go down into the data structure and manipulate things at the low level. You can operate it at a higher level.