PPL Assignment 1

<u>Part 1</u>: Theoretical Questions

1) **Imperative paradigm :**
   Based on commands (statements).
   Control flow is an explicit sequence of those commands .
   mainly defined in contrast to “declarative” .
   In general, in the Imperative paradigm, commands are executed, and through their effect, state is modified.

   Procedural programming:
   encourages the use of small units of codes, called procedures, which encapsulate well-defined commands. Procedures interact through well defined interfaces published by each procedure (the contract of the procedure, including the signature of which arguments it expects, and which return value it returns), and local variables are used inside each procedure without affecting the state of the program outside the scope of the procedures. Key programming constructs associated with this paradigm are procedures, local variables. Procedural programming founded to avoid copy code and to help the developer to have local variables .
   Procedural programming improves on these weaknesses by introducing:
   - Procedures — commands with a well defined interface of input parameters / output parameters and expected behavior.
   - Local variables — variables which are defined only within the scope of the procedure.

   Functional paradigm :
   Computation proceeds by (nested) function calls that avoid any global state mutation and through the definition of function composition.
   This paradigm is an example of programming that is most similar to evaluation of expressions in mathematics.
   In functional programming, a program is viewed as an expression, which is evaluated by successive applications of functions to their arguments, and substitution of the result for the functional expression.
   Computation in functional programming has no side-effects
   FP supports high-order functions, thus:

     - Functions can be passed as arguments to other functions
     - Functions can be returned as a result of a function
     - Functions can be anonymous (no names)

2) (a) **<T>**(x: **T[]**, y: **(a: T) => boolean**) => x.some(y) **:boolean**

   (b) x: **number[]** => x.reduce((acc: **number**, cur: **number**) => acc + cur **(number)**, 0) **:number**

   (c) **<T>**(x: **boolean**, y: **T[]**) => x ? y[0] : y[1] **:T** **//Generic can replaced by 'any' in a case of Etrogenic-array**

3) **Abstraction Barriers**

   What it is basically, is instead of doing some complex operations inline, you move them, you extract them out into a function, you name that function. Now you have a barrier, where you don't really have to think about the internals of how this thing gets calculated, you have an operation that's got a nice, clear, meaningful name that you're working on.

   Abstraction barriers are simply taking operations that you're doing on some data structure or some piece of data. You're repeating it. It doesn't have enough meaning, so you extract it out and give it a name.

   We can see a case of abstraction barriers in the example 'printCubes ': the procedure printCubes iterates over an array, and applies the function cube on each element. The client of the printCubes procedure does not directly invoke the cube function - it is encapsulated inside the printCubes procedure. If such discipline is applied systematically, we can enforce abstraction barriers between collections of procedures.