Q1:

Imperative:

imperative programming is a programming paradigm of software that uses statements that change a program's state, the control flow in Imperative programming is an explicit sequence of commands - mainly defined in contrast to “declarative”, where commands (or statements) are executed, and through their effect, state is modified (mutated).

Procedural:

It’s a kind of Imperative programming organized around hierarchies of nested procedure calls. It 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, 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 to this paradigm are procedures, local variables. These 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 where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values.

Computation proceeds by (nested) function calls that avoid any global state mutation and through the definition of function composition.

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.

In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions.

Procedural programming is more improved than Imperative programming by adding layers of abstraction in the form of procedures., also the code becomes reusable, and it is easier to keep track of the control flow. Functional programming is more improved than Procedural programming, because in Functional programming we can use recursion and concurrency unlike the Procedural programming.

Q2:

```
function averageGradesOver60(grades: number[]):number {
  let gradesOver60 : number[] = grades.filter((num:number) => num >=60);
  let count = gradesOver60.length;
  return gradesOver60.reduce((acc:number, curr: number) => acc + curr, 0) / count;
 }
```

Q3:

a. (x: number[], y : (a : number) => Boolean) => Boolean

b. (x: number[]) => number

c. <T>(x: boolean, y: T[]) => T

d. <T,V>(f: (a: T) => V, g: (a: number) =>T) = > (a: number) => V

Q4:

a way to hide the intricacies of data structures made from cons cells. In other words, abstraction barriers isolate different levels of the system. The implementer of the high-level system doesn't have to know about the low-level details. basically, instead of doing some complex operations inline, we move them, we extract them out into a function with a given name. That's called a barrier, where we don't really have to think about the internals of how this thing gets calculated when we use it. And we have an operation that's got a clear, meaningful name.