PPL Part 1 Answers

1.A

i.

Imperative Programming is a programming paradigm in which the program consists of commands written and executed one after the other. This type of programming actually gives the computer a set of commands to execute so it “tells” the computer what steps/actions to take, so when executing a task it tells it **how to do something**.

ii.

Procedural programming, like Imperative Programming, relies on commands given to the computer to execute. The main difference from imperative programming is that it allows code to be broken down into procedures (a.k.a functions) and allows those functions to be called from different parts of the code. It allows code reuse, making it more modular and maintainable than imperative programming.

iii.

Functional programming is a programming paradigm in which the program is a set of expressions and running the program is evaluating the expression's value.
it will consist of functions that have no side effects (does not change the state of the program, all it does is get an input and return an output). This is actually programming using functionalities. This type of programming actually gives the computer a set of functionality to execute so it “tells” the computer what tasks to complete, so when executing a program it tells it **what to do**.

1.B

- Allows code reusability, we can use these blocks of code we made again and again.
- Adds Modularity to the code.
- Adds encapsulation because a developer can just use the function/procedure without knowing how that procedure did the task it was meant for.
- Helps code maintainability.
- Helps scalability because we have built abilities/procedures already one after the other and if we want to add something we don’t “start with nothing”, exactly the opposite, we have our tools.
- Prevents “spaghetti code” (or at least allows a better alternative) .
- Makes code more readable.

1.C

- It tells the computer **what to do** rather than **how to do** it, and by that, it adds abstraction to the code.
- Can be parallelized more easily (since there are no side effects, it's like programming with immutable objects,which is thread-safe), so it helps concurrent execution.
- Testing is easier because, functional programming relies purely on functions, each having a certain output according to it’s task, so we’re only checking it’s output.

2.

```
const sum: (inventory: Product[]) => number = (inventory: Product[]) => inventory.filter((x)=> x.discounted).map(x=> x.price)
.reduce((acc,curr) => acc+curr,0);
const amount: (inventory: Product[]) => number  = (inventory: Product[]) => inventory.filter((x)=> x.discounted).length;
const average: (inventory: Product[]) => number = (inventory: Product[]) => sum(inventory)/amount(inventory);
```

3.
(the types are in the rectangle)
A

```
const func: <T>(x:T[],y:(elemnet: T) => boolean) => boolean = (x,y) => x.some(y);
```

B

```
const func2: (x: number[]) => number = x => x.reduce((acc,cur) => acc + cur, 0);
```

C

```
const func3: <T>(x: boolean, y: T[] ) => T = (x,y) => x ? y[0] : y[1];
```

D

```
const func4: <T,S>(f: (input : T) =>  S, g: (input: number) => T) => (x: number) => S = (f,g) => x => f(g(x+1));
```