1:

A

Imperative Programming:

In the imperative programming paradigm, programs are structured as sequences of statements that change the program's state. This paradigm emphasizes direct control over the computer's execution flow and mutable state.

Key features include:

1. **State Manipulation**: Programs maintain mutable state, which is modified by executing imperative statements. This direct manipulation of state allows precise control over program behavior.
2. **Sequential Execution**: Instructions are executed sequentially, with each statement directly influencing the subsequent state of the program. This sequential execution facilitates clear and deterministic program flow.
3. **Procedural Abstraction**: Imperative programming encourages the decomposition of tasks into reusable procedures or functions. These procedures encapsulate sequences of imperative statements, promoting code organization and modularity.
4. **Control Flow Constructs**: Common constructs like loops, conditionals, and subroutine calls enable the expression of complex control flow patterns within imperative programs. These constructs facilitate conditional branching, iterative processes, and modular code composition.
5. **Mutability**: Mutable data structures and variables are prevalent in imperative programming, allowing for in-place modifications and dynamic state management. While mutable state provides flexibility, it can also introduce challenges related to side effects and state consistency.

Languages like C, Python, and Java exemplify the imperative paradigm, offering constructs for expressing sequential logic and manipulating state. Imperative programming is well-suited for tasks requiring fine-grained control over execution flow and state manipulation, but it can lead to verbose code and challenges in managing mutable state at scale.

Procedural Programming:

In the procedural programming paradigm, the focus is on organizing code into procedures or routines, which are sequences of instructions to be executed. Programs consist of procedures that can call each other to solve a problem step by step.

Key features include:

1. **Procedure-Centric Design**: Programs are structured around procedures or subroutines, which encapsulate a series of steps to perform a specific task.
2. **Global State**: Data is often shared globally among procedures, potentially leading to issues with modularity and data integrity.
3. **Emphasis on Control Flow**: Procedural programming languages provide constructs like loops and conditionals to control the flow of execution within procedures.
4. **Data Abstraction**: While procedural languages may support data abstraction to some extent, the primary focus is on organizing code around procedures rather than abstract data types.

Languages like C and Pascal exemplify the procedural paradigm, offering constructs for defining procedures, manipulating data, and controlling program flow. Procedural programming facilitates structured and modular code design, making it suitable for tasks where step-by-step procedures are predominant.

Functional Programming:

Functional programming revolves around the evaluation of mathematical functions and emphasizes immutable data and function purity. Programs are composed of functions that operate on immutable data structures, avoiding side effects and mutable state.

Key features include:

1. **Function-Centric Design**: Programs are structured around functions, which are pure mathematical mappings from inputs to outputs, devoid of side effects.
2. **Immutable Data**: Data structures are immutable, meaning they cannot be modified after creation. Functions create new data structures rather than modifying existing ones.
3. **Higher-Order Functions**: Functions can accept other functions as arguments and return functions as results, enabling powerful abstractions and expressive code.

4. **Recursion**: Recursion is a common technique for solving problems in functional programming, replacing mutable state and loops with self-referential function calls.

Languages like Haskell, Lisp, and Scala embody the functional paradigm, providing robust support for functions as first-class citizens. Functional programming promotes concise, declarative code that is easier to reason about and test, particularly in concurrent and distributed systems. However, mastering functional programming concepts may require a shift in mindset for developers accustomed to imperative or procedural paradigms.

B

The procedural paradigm builds upon the imperative paradigm by organizing code into reusable procedures or functions. These procedures encapsulate sequences of imperative statements, promoting modularity, code reusability, and easier maintenance. By breaking down tasks into smaller, more manageable chunks, procedural programming improves code organization and readability. Additionally, procedural abstraction allows developers to focus on solving specific problems within isolated contexts, reducing complexity and enhancing code maintainability. Overall, the procedural paradigm offers a structured approach to programming that facilitates better code organization and modular design compared to the more linear approach of the imperative paradigm.

C

The functional paradigm improves upon the procedural paradigm by emphasizing the use of pure functions and immutable data structures. In functional programming, functions are treated as first-class citizens, meaning they can be passed as arguments to other functions and returned as results. This enables higher-order functions, which facilitate powerful abstractions and concise code.

By using immutable data structures, functional programming avoids the complexities associated with mutable state and side effects. This leads to more predictable and easier-to-reason-about code, as functions only depend on their inputs and produce deterministic outputs, without modifying any external state.

Functional programming also encourages a declarative style of programming, where the focus is on expressing what the program should do rather than how it should do it. This leads to code that is often more concise, expressive, and maintainable compared to the imperative or procedural approaches.

Overall, the functional paradigm promotes modularity, code reuse, and easier reasoning about program behavior by leveraging pure functions, immutable data, and declarative programming style, thereby improving upon the procedural paradigm.

2:

```
const getDiscountedProductAveragePrice = (inventory: Product[]): number => {
    const discountedProducts :Product[]  = inventory.filter(product => product.discounted);
    const discountedPrices : number[] = discountedProducts.map(product => product.price);

    if (discountedPrices.length === 0) {
        return 0;
    }

    const discountedPriceSum : number  = discountedPrices.reduce((sum, price) => sum + price, 0);
    return discountedPriceSum / discountedPrices.length;
}
```

2.1:

```
type somefunctiontype= <T>(x:T[],f:( y:T )=>boolean)=>boolean
const f1:somefunctiontype  = (x, y) => x.some(y);
```

2.2:

```
type somefunctiontype2=(x: number[])=>number
const f2 :  somefunctiontype2 =  x => x.reduce((acc, cur) => acc + cur, 0)
```

2.3:

```
type somefunctiontype3=<T>(x:boolean,y:T[])=>T
const f3: somefunctiontype3 = <T> (x:boolean, y:T[]) => x ? y[0] : y[1]
```

2.4:

```
type ComposedFunction = <U, V>(f: (arg: U) => V, g: (arg: number) => U) => (x: number) => V;
```