1a.

The imperative paradigm focuses on describing how a program should execute by using statements that change the program's state. It relies on sequential execution, loops, and conditionals to manipulate variables directly.

1b.

The procedural paradigm is a type of imperative programming that organizes code into functions (procedures) to improve modularity and reusability. Instead of writing everything sequentially, programs are divided into reusable blocks, reducing code duplication.

1c.

The functional paradigm treats computation as the evaluation of mathematical functions, avoiding mutable state and side effects. It promotes pure functions, immutability, and higher-order functions, leading to more predictable and parallelizable code.

2.

The procedural paradigm improves imperative programming by introducing structured code organization using functions. This makes programs more readable, reusable, and maintainable, reducing redundancy and making debugging easier.

3.

The functional paradigm eliminates mutable state and side effects, making programs more predictable, easier to test, and inherently parallelizable. It also promotes higher-order functions and recursion, reducing reliance on loops and conditionals. This leads to cleaner, more maintainable code with fewer bugs.

```
type Transaction = {
  category: string;
  price: number;
  quantity: number;
};

const calculateRevenueByCategory = (
  transactions: Transaction[]
): Record<string, number> =>
  transactions
    .map((transaction) => ({
      ...transaction,
      price:
        transaction.quantity > 5 ? transaction.price * 0.9 : transaction.price,
    }))
    .filter((transaction) => transaction.price >= 50)
    .reduce(
      (acc, transaction) => ({
        ...acc,
        [transaction.category]:
          (acc[transaction.category] || 0) +
          transaction.price * transaction.quantity,
      }),
      {} as Record<string, number>
    );

type type1 = <T>(x: T[], y: (value: T) => boolean) => boolean;
const test1: type1 = (x, y) => x.some(y);

type type2 = (x: number[]) => number[];
const test2: type2 = (x) => x.map((y) => y * 2);
```

```
type type3 = <T>(x: T[], y: (value: T) => boolean) => T[];
const test3: type3 = (x, y) => x.filter(y);

type type4 = (x: number[]) => number;
const test4: type4 = (x) => x.reduce((acc, cur) => acc + cur, 0);

type type5 = <T>(x: boolean, y: T[]) => T;
const test5: type5 = (x, y) => (x ? y[0] : y[1]);

type type6 = (
  f: (x: number) => number,
  g: (x: number) => number
) => (x: number) => number;
const test6: type6 = (f, g) => (x) => f(g(x + 1));
```