Guia P1 - Prog. Funcional

Ejercicio: De Imperativo a Funcional

Refactoriza el siguiente código imperativo para usar programación funcional (map, filter, reduce):

```
const productos = [
    { nombre: "Laptop", precio: 1200, categoria: "tecnologia" },
    { nombre: "Mouse", precio: 25, categoria: "tecnologia" },
    { nombre: "Silla", precio: 150, categoria: "muebles" },
    { nombre: "Teclado", precio: 75, categoria: "tecnologia" }
];

// Código imperativo
let productosTecnologia = [];
for (let i = 0; i < productos.length; i++) {
    if (productos[i].categoria === "tecnologia") {
        productosTecnologia.push(productos[i]);
    }
}

let preciosConDescuento = [];
for (let i = 0; i < productosTecnologia.length; i++) {
    preciosConDescuento.push(productosTecnologia[i].precio * 0.9);
}

let total = 0;
for (let i = 0; i < preciosConDescuento.length; i++) {
    total += preciosConDescuento[i];
}

console.log(total);
```

Requisitos:

- Usa `filter`, `map` y `reduce`
- Todo debe hacerse en una sola expresión encadenada
- El código debe ser más legible que el original

Solución

Ejercicio: Operaciones Simples en Árbol Binario

Usando el tipo `Node`:

`type Node = {value: number; left: Node | null; rigth: Node | null;}`

Implemente las siguientes tres funciones independientes:

```
// 1. Retorna la suma de todos los valores del árbol
function sum(node: Node | null): number

// 2. Retorna el valor máximo del árbol
function max(node: Node | null): number

// 3. Retorna la cantidad total de nodos del árbol
function count(node: Node | null): number
```

Considere los casos borde (árbol vacío, un solo nodo, etc.)

Solución

Ejercicio: Transformación de Árbol

Usando el tipo `Node`:

`type Node = {value: number; left: Node | null; rigth: Node | null;}`

Implemente:

```
function transform(node: Node | null, f: (value: number) => number): Node | null
```

Esta función debe crear un **nuevo árbol** con la misma estructura, pero aplicando la función `f` a cada valor.

Ejemplos:

- Si `f = (x) => x * 2`, duplica todos los valores
- Si `f = (x) => x % 2 === 0 ? x : 0`, reemplaza impares con 0

Importante: No debe modificar el árbol original, debe crear uno nuevo.

Solución

Ejercicio: Árboles N-arios

Considere el siguiente tipo que modela un árbol n-ario (cada nodo puede tener cualquier cantidad de hijos):

```ts
type TreeNode = {
  value: number;
  children: TreeNode[];
}
```

Implemente la función:

```ts
function fold(node: TreeNode, f: (current: number, accumulated: number) => numbe
```

Esta función permite realizar operaciones como:

- Si `f = (curr, acc) => curr + acc`, calcula la suma de todos los valores
- Si `f = (curr, acc) => Math.max(curr, acc)`, encuentra el valor máximo
- Si `f = (curr, acc) => acc + 1`, cuenta la cantidad de nodos

Solución

Ejercicio: Composición de Funciones

Componer funciones para crear pipelines de transformación.

Implementa dos funciones:

- `compose`, que combina funciones de derecha a izquierda.
- `pipe`, que combina funciones de izquierda a derecha.

Ambas funciones deben trabajar específicamente con funciones que operan sobre números (`(x: number) => number`).

Ejemplo:

```ts
const duplicar = (x: number) => x * 2;
const sumar3 = (x: number) => x + 3;
const elevarAlCuadrado = (x: number) => x ** 2;

// Operación con compose: de derecha a izquierda
const operacion1 = compose([duplicar, sumar3, elevarAlCuadrado]);
console.log(operacion1(2));
// duplicar(sumar3(elevarAlCuadrado(2))) = duplicar(sumar3(4)) = duplicar(7) = 1

// Operación con pipe: de izquierda a derecha
const operacion2 = pipe([elevarAlCuadrado, sumar3, duplicar]);
console.log(operacion2(2));
// duplicar(sumar3(elevarAlCuadrado(2))) = 14
```

Requisitos:

1. Las funciones deben aceptar únicamente funciones que operen sobre números.
2. Deben funcionar con cualquier cantidad de funciones.
3. Ambas implementaciones (`compose` y `pipe`) deben ser independientes.

Solución

Ejercicio: Sistema de Notificaciones con Strategy

Tipo: Implementación desde planteamiento

Planteamiento:
Diseña un sistema de notificaciones para una aplicación que puede enviar mensajes por:

- Email
- SMS
- Push notifications
- Slack

Requisitos:

1. Usa el patrón Strategy funcional (sin clases)
2. Cada estrategia debe recibir un destinatario y un mensaje
3. Implementa una función `enviarNotificacion` que acepte la estrategia como parámetro
4. Simula el envío imprimiendo en consola
5. Crea un ejemplo donde se envíe la misma notificación por diferentes canales

Bonus: Implementa un "NotificationManager" que pueda enviar por múltiples canales simultáneamente.

Solución

Conceptos Clave

Funciones como Ciudadanos de Primera Clase

```
// 1. Asignar a variables
const saludar = (nombre: string) => `Hola ${nombre}`;

// 2. Pasar como argumentos
const ejecutar = (fn: Function, valor: any) => fn(valor);
ejecutar(saludar, "Italo");
```

```
// 3. Retornar desde funciones
const crearSumador = (x: number) => (y: number) => x + y;

// 4. Almacenar en estructuras
const operaciones = [suma, resta, multiplicacion];
```

Higher-Order Functions

```
// Función que RECIBE función(es):
const aplicarDosVeces = (fn: (x: number) => number, valor: number) => {
    return fn(fn(valor));
};

// Función que RETORNA función:
const multiplicadorPor = (factor: number) => {
    return (numero: number) => numero * factor;
};
```