Módulo 2 Lenguajes

Challenges

1. Aplanando arrays

Apartado A

Dado un array multidimensional, construye una función inmutable que devuelva el mismo array aplanado, esto es, con un único nivel de profundidad. Por ejemplo, el siguiente array:

```
const sample = [1, [2, 3], [[4], [5, 6, [7, 8, [9]]]]];
```

quedaría aplanado como:

```
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Apartado B

¿Has resuelto el ejercicio anterior? Suponiendo que los arrays multidimensionales del ejercicio anterior no serán de naturaleza mixta, es decir, sus elementos siempre serán del mismo tipo ¿Serías capaz de proporcionar un tipado adecuado a dicha función de aplanamiento?

2. Acceso en profundidad

Apartado A

Implementa un mecanismo `deepGet` para acceder en profundidad a objetos anidados, de modo que podamos recuperar una propiedad en cualquiera de sus niveles. Mira a continuación el comportamiento que debería seguir:

```
const myObject = {
  a: 1,
  b: {
    c: null,
    d: {
      e: 3,
      f: {
        g: "bingo",
      }
    }
  }
};

const deepGet = ¿..?

console.log(deepGet(myObject, "x")); // undefined
console.log(deepGet(myObject, "a")); // 1
console.log(deepGet(myObject, "b")); // { c: null, d: {....}}
console.log(deepGet(myObject, "b", "c")); // null
console.log(deepGet(myObject, "b", "d", "f", "g")); // bingo
console.log(deepGet(myObject));  // {a: 1, b: {...}}
```

Apartado B

Ahora implementa el complementario, `deepSet`, que permita guardar valores en profundidad. Su comportamiento debería ser:

```
const myObject = {};

const deepSet = ¿..?

deepSet(1, myObject, "a", "b");
console.log(JSON.stringify(myObject));  // {a: { b: 1}}
deepSet(2, myObject, "a", "c");
console.log(JSON.stringify(myObject));  // {a: { b: 1, c: 2}}
deepSet(3, myObject, "a");
console.log(JSON.stringify(myObject));  // {a: 3}
deepSet(4, myObject);
console.log(JSON.stringify(myObject));  // Do nothing // {a: 3}
```

3. Árbol

¿Cómo generarías con TypeScript un tipado para estructuras en forma de árbol? Un árbol es una estructura que parte de un nodo raiz, a partir del cual salen más nodos. Cada nodo en un árbol puede tener hijos (más nodos) o no tenerlos (convirtiendose en un nodo final o una "hoja").

4. Trazas por consola

Ejecuta el siguiente código:

```
const delay = ms => new Promise(resolve => setTimeout(resolve, ms));

const showMessage = async ([time, message]) => {
  await delay(time);
  console.log(message);
};

const triggers = [
  async () => await showMessage([200, "third"]),
  async () => await showMessage([100, "second"])
];

const run = triggers => {
  console.log("first");
  triggers.forEach(t => t());
};

run(triggers);
```

Las trazas resultante en consola son:

```
first;
second;
third;
```

El ejercicio consiste en reordenar las trazas para que se muestren invertidas, es decir, con el siguiente orden:

```
third;
second;
first;
```

Pero para ello **tan solo podrás modificar la función** `run`.

Queda prohibido modificar cualquier otro código asi como el contenido de `triggers`.

5. Memoization

Apartado A

Implementa un mecanismo de *memoización* para funciones costosas y tipalo con TypeScript. La memoización optimiza sucesivas llamadas del siguiente modo:

```
const expensiveFunction = () => {
  console.log("Una única llamada");
  return 3.1415;
}

const memoize = ¿..?;

const memoized = memoize(expensiveFunction);
console.log(memoized()); // Una única llamada // 3.1415
console.log(memoized()); // 3.1415
console.log(memoized()); // 3.1415
```

NOTA: Puedes suponer que las funciones que van a ser memoizadas no llevan argumentos y tampoco devuelven valores `null` o `undefined`.

Apartado B

¿Podrías hacerlo en una sola línea?

Apartado C

Contempla ahora la posibilidad de que la función a memoizar pueda tener argumentos. Por simplicidad supongamos sólo argumentos primitivos: `string`, `number` o `boolean` y que no sean `undefined`. ¿Podrías hacer una versión aceptando argumentos? ¿Cómo la tiparías con TS? Un ejemplo de comportamiento podría ser:

```
let count = 0; // Comprobacion de nº de ejecuciones
const repeatText = (repetitions: number, text: string): string =>
  (count++, `${text} `.repeat(repetitions).trim())

const memoize = ¿..?;

const memoizedGreet = memoize(repeatText);

console.log(memoizedGreet(1, "pam"));   // pam
console.log(memoizedGreet(3, "chun"));  // chun chun chun
console.log(memoizedGreet(1, "pam"));   // pam
console.log(memoizedGreet(3, "chun"));  // chun chun chun
console.log(count);                     // 2
```