TP N°1 - PARADIGMAS IMPERATIVOS

Ejercicio 1

Parte A

Considera el lenguaje JavaScript acotado al paradigma de programación estructurada y analízalo en términos de los cuatro componentes de un paradigma mencionados por Kuhn.

1. Generalización simbólica: ¿Cuáles son las reglas escritas del lenguaje?

   Las reglas escritas de **JavaScript (ECMAScript)** son las siguientes:

   #### Reglas léxicas (alfabeto del lenguaje)

   - **Sensibilidad a mayúsculas/minúsculas**: let ≠ Let.
   - **Conjunto de caracteres**: Unicode (puede usarse en identificadores y cadenas).
   - **Comentarios**:
     - Línea: // comentario
     - Bloque: /* comentario */
   - **Separadores**: espacios, tabuladores, saltos de línea.

   #### Identificadores

   - Deben empezar con una letra, $ o _.
   - Luego pueden contener letras, dígitos, $ o _.
   - No pueden ser palabras reservadas (ej: if, while, class, function, etc.).

   Ejemplo válido:

```
let _variable1 = 10;
```

Tipos de datos primitivos

- number (enteros y reales, IEEE 754), bigint, string, boolean, undefined, null, symbol

Literales

- Numéricos: `10`, `3.14`, `0xFF`, `0b1010`, `1e3`
- Cadenas: `"texto"`, `'texto'`, `` `plantilla ${exp}` ``
- Booleanos: `true`, `false`
- Objetos: `{ clave: "valor" }`
- Arreglos: `[1, 2, 3]`

Operadores

- Aritméticos: `+ - * / % **`
- Comparación: `== != === !== > >= < <=`
- Lógicos: `&& || !`
- Asignación: `= += -= *= /= %= **=`
- Otros: `typeof`, `instanceof`, `in`, `? :` (ternario)

Expresiones y sentencias

- **Expresiones**: combinaciones de valores y operadores (`x + y`, `f(2)`).
- **Sentencias**:
  - Declarativas: `let`, `const`, `var`
  - Condicionales: `if`, `switch`
  - Iterativas: `for`, `while`, `do...while`
  - Control: `break`, `continue`, `return`, `throw`
  - Bloques: `{ ... }`

Funciones y clases

- Declaración de funciones:

```
function suma(a, b) {
  return a + b;
}
```

- Funciones flecha:

```
const suma = (a, b) => a + b;
```

- Clases:

```
class Persona {
  constructor(nombre) {
    this.nombre = nombre;
  }
}
```

Reglas de alcance y contexto

- Ámbito de bloque con let y const.
- Ámbito de función con var.
- Contexto this depende de cómo se invoque la función.

Objetos y prototipos

- Todo objeto hereda de un prototipo.
- Se pueden crear objetos con {} o con new.

Reglas de ejecución

- Modelo basado en eventos y single-thread.
- Manejo de promesas y async/await para asincronía.

2. Creencias de los profesionales: ¿Qué características particulares del lenguaje se

cree que sean "mejores" que en otros lenguajes?

Las características particulares del lenguaje JS que cree que son mejores son:

Ubicuidad

- Está en **todos los navegadores** modernos sin necesidad de instalar nada.
- Permite hacer aplicaciones que corren tanto en cliente (frontend) como en servidor (con Node.js).

  👉 Esto lo hace único: muy pocos lenguajes tienen esa presencia universal.

Flexibilidad

- Es **dinámico** y de **tipado débil**: no obliga a declarar tipos rígidamente como Java o C.
- Permite mezclar estilos: funcional, imperativo y orientado a objetos (con prototipos o clases).

  👉 Los profesionales creen que esto lo hace muy rápido para prototipar.

Curva de aprendizaje baja

- Su sintaxis es parecida a C/Java, pero menos estricta.
- Se puede empezar con pocas líneas y lograr resultados visibles en el navegador.

  👉 Esto se valora porque es más accesible para principiantes.

Ecosistema masivo

- Librerías y frameworks (React, Angular, Vue, Node.js, Express, etc.).
- Herramientas de desarrollo modernas (NPM, Webpack, Babel, etc.).

  👉 Muchos profesionales creen que esta comunidad y cantidad de recursos lo hacen “mejor” que lenguajes más cerrados.

Asincronía y modelo de eventos

- Manejo nativo de asincronía con **callbacks, Promesas y async/await**.
- El **event loop** simplifica la concurrencia (aunque no siempre sea fácil de dominar).

  👉 Se considera mejor que en lenguajes donde hay que manejar hilos manualmente.

Ejecución multiplataforma

- Con Node.js, se puede usar para **backend, frontend, apps móviles (React Native), escritorio (Electron)**.
- Un mismo lenguaje en varios dominios reduce la curva de aprendizaje y costos.

  👉 Muchos creen que esta "versatilidad" es mejor que usar distintos lenguajes para cada capa.

Rapidez de desarrollo

- Al no tener compilación pesada (se interpreta), se prueba el código al instante.
- Perfecto para prototipado rápido y desarrollo ágil.

  👉 Profesionales valoran que ahorra tiempo frente a lenguajes compilados como C++ o Java.

Parte B

Considera el lenguaje JavaScript acotado al paradigma de programación estructurada y analízalo en términos de los ejes propuestos para la elección de un lenguaje de programación ( ) y responde:

1. ¿Tiene una sintaxis y una semántica bien definida? ¿Existe documentación oficial?

Sí, tiene una sintaxis y una semántica bien definida y su documentacion es:

- La definición oficial está en la especificación **ECMAScript (ECMA-262)**, mantenida por **ECMA International**.
- La sintaxis y semántica están formalizadas en esa norma, aunque con cierta flexibilidad en la interpretación de implementaciones.

- La documentación de referencia más usada es la de **MDN Web Docs (Mozilla Developer Network)**.

2. ¿Es posible comprobar el código producido en ese lenguaje?

Sí, pero con matices.

- Existen **herramientas de linters** (como ESLint) y **tipado estático opcional** con TypeScript o Flow que permiten comprobar el código antes de ejecutarlo.
- En JavaScript puro, el chequeo es en **tiempo de ejecución**, no en compilación (lo que lo hace menos seguro comparado con Java o C).

3. ¿Es confiable?

En general sí, pero depende:

- Los navegadores y Node.js son entornos **muy probados y seguros**.
- Sin embargo, por ser dinámico y de tipado débil, el programador puede introducir fácilmente errores difíciles de detectar.
- La confiabilidad aumenta usando **buenas prácticas** (tests, tipado opcional con TS, linters).

4. ¿Es ortogonal?

No completamente.

- JavaScript **no es totalmente ortogonal**:
  - Ejemplo: `typeof null` devuelve `"object"` (inconsistencia histórica).
  - `NaN !== NaN` es verdadero.
  - Algunas operaciones funcionan distinto según el tipo (`+` suma números, pero concatena strings).
- Hay excepciones que rompen esa uniformidad.

5. ¿Cuáles son sus características de consistencia y uniformidad?

- **Consistencia parcial**: la sintaxis base (estructuras de control, funciones, objetos) es relativamente coherente.

- **Inconsistencias históricas**: coerción de tipos (`'5' - 2` da `3`, pero `'5' + 2` da `"52"`), comparación con `==` vs `===`, y manejo de valores especiales (`null`, `undefined`, `NaN`).

6. ¿Es extensible? ¿Hay subconjuntos de ese lenguaje?

Sí.

- Extensible: se pueden definir objetos, prototipos, clases y hasta modificar el comportamiento de objetos nativos (aunque no siempre recomendable).
- Subconjuntos:
  - **Strict mode** (`"use strict";`) restringe ciertas conductas para mejorar seguridad y claridad.
  - **Subconjuntos prácticos**: muchos equipos imponen estilos o convenciones que restringen el uso del lenguaje (ej. "JavaScript seguro" o "subset de Node").
  - TypeScript se podría ver como una extensión que mejora JavaScript sin romper compatibilidad.

7. El código producido, ¿es transportable?

Sí, y es una de sus grandes fortalezas.

- El mismo código JS funciona en cualquier navegador moderno, en Node.js, e incluso en apps móviles (React Native) o escritorio (Electron).
- La transportabilidad está casi garantizada gracias a la estandarización ECMAScript.
- **Matiz**: pueden existir pequeñas diferencias en implementaciones antiguas de navegadores, pero hoy la compatibilidad es muy alta.