UNViMe

Trabajo Práctico 1

Alumno: Tula E. Fernando

Carrera: Ing. de Sistemas de Información

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?

   Como lenguaje acotado a programación estructurada, posee todas las reglas escritas de un lenguaje de este estilo, ya que tiene:

   - Estructuras basicas: do, do-while, for, if-else.
   - Variables flexibles que se pueden manejar mediante las palabras reservadas `var` o `let` las cuales no requieren el tipo de dato al momento de ser definidas, sólo el nombre de la variable.
   - Una línea de código puede terminar en `;` o no.
   - Los arreglos y estructuras se tratan como objetos.
   - Tipos y operadores: primitivas (`number, string, boolean, null, undefined, symbol, bigint`), objetos literales, operadores aritméticos, lógicos, comparación estricta `===`.
   - Control de flujo: `break, continue`, manejo de errores `try/catch/finally`.
   - Reglas de estilo del subconjunto estructurado: entrada única y salida controlada por `return`; sin herencia prototipal ni this/clases; sin metaprogramación; evitar efectos globales.
   - Contexto de ejecución: modo estricto `use strict` recomendado; evaluación determinada por orden textual; sin dependencias de host para la semántica del núcleo.

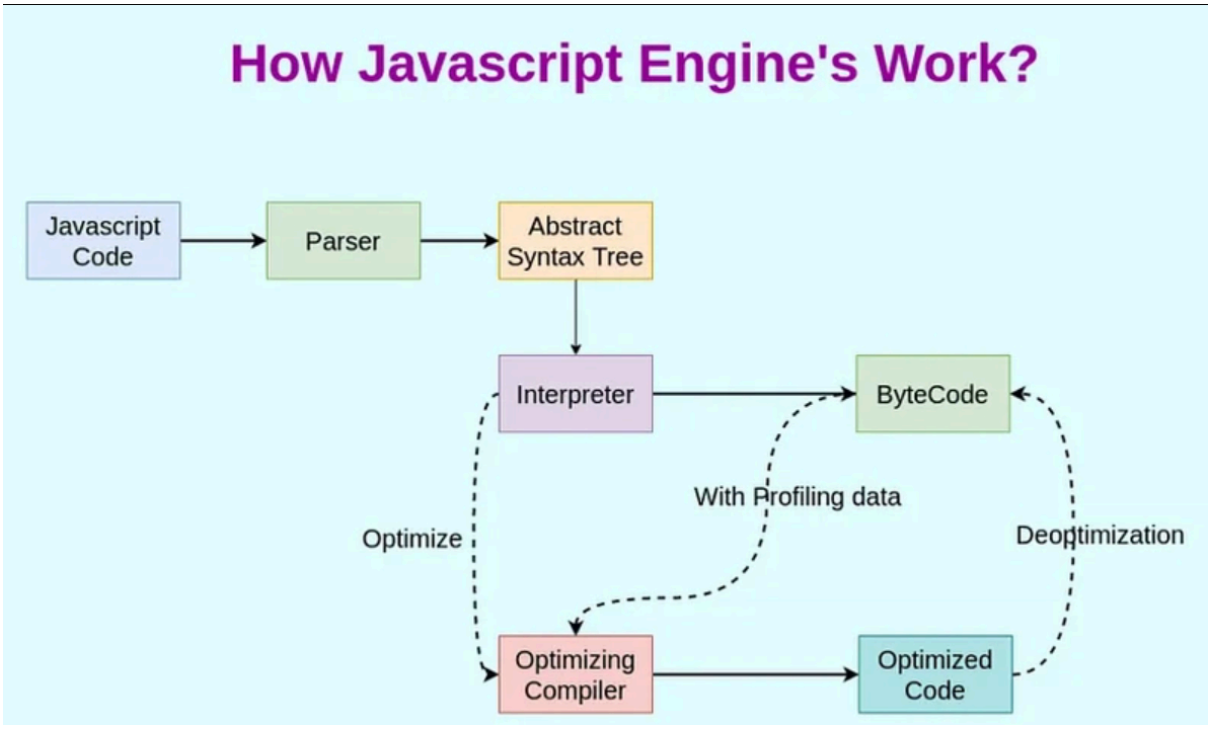
2. Creencias de los profesionales: ¿Qué características particulares del lenguaje se cree que sean "mejores" que en otros lenguajes?

Para responder esta pregunta primero hay que partir de la definicion de JS: “JavaScript es un lenguaje de programación de scripting, interpretado —más precisamente, de compilación just-in-time—, multiparadigma, con funciones de primera clase, de un sólo hilo.”

Entre sus principales ventajas encontramos:

- La primera de ellas es la **simplicidad**. Al no haber concurrencia ni ser necesario sincronizar hilos, es mucho más sencillo de comprender y depurar.
- Otra ventaja, quizás la más importante, es la facilidad con la que permite manejar eventos, tarea en la JavaScript se destaca. Al haber **sólo un hilo**, los eventos sólo pueden suceder en orden secuencial, evitando que dos o más procesos desatados por dos eventos compitan por mutar el estado de algún recurso.
- **Portabilidad**, se puede ejecutar en cualquier navegador gracias a su **tokenización** (descompone el código fuente en unidades más pequeñas llamadas tokens). Una vez que el código dado está tokenizado, el analizador convierte ese código en un AST. A continuación se genera el código de bytes es una abstracción del código máquina. Es lo que permite que Javascript funcione en todas las plataformas, como Windows, Android, macOS, etc. El código de bytes se crea teniendo en cuenta la arquitectura del sistema en el que se ejecuta.
    - **Bytecode** es como un traductor que conoce todos los idiomas (es decir, diferentes arquitecturas) y ayuda al JIT (turista) a entender el lugar donde está (diferentes plataformas), pero el traductor es siempre el mismo amigo del turista, solo que conoce muchos idiomas.
- **Rapidez**: **Just In Time** se compone de un compilador y un intérprete. Por lo tanto, cuando el código JS entra en contacto con el motor JS, se inicia con el intérprete. ¿Por qué? Para que la ejecución pueda comenzar y el usuario no tenga que esperar. Empieza a ejecutar el código. Mientras ejecuta el código, también registra qué funciones o partes del código se invocan repetidamente y las marca como **rutas activas**. Una vez recopilados suficientes datos para llamar a una función como ruta activa , el compilador toma el bytecode y crea una versión optimizada del mismo, almacenándola en caché.
- Se puede utilizar para trabajar tanto para ser ejecutada en el navegador como desde el servidor usando **Node.js**, el cual actúa como un contenedor del motor V8 en el lado del servidor. Introduce características como una arquitectura basada en eventos, un sistema de módulos CommonJS y API integradas para tareas de servidor.
- Es un lenguaje **flexible** y de **tipado dinámico:**
    1. Tipos asociados a los valores, no a las variables.

2. Conversión implícita de tipos: Muchas operaciones realizan coerciones automáticas; por ejemplo, al usar el operador + con un número y una cadena, JavaScript convierte el número en cadena y concatena
3. Estructuras de datos flexibles.
    a. Objetos de estructura abierta: Los objetos en JavaScript se comportan como tablas dinámicas: no tienen una forma fija y sus propiedades pueden añadirse, eliminarse o reordenarse en cualquier momento
    b. Arreglos heterogéneos y dinámicos: Los arrays son objetos especializados que pueden contener valores de cualquier tipo y crecer o encogerse cuando se añaden o eliminan elementos.
4. Funciones de primera clase y parámetros flexibles. Las funciones son objetos y se pueden pasar como argumentos, devolver desde otras funciones o asignar a variables. Pueden declararse con cualquier número de parámetros y llamarse con más o menos argumentos de los que esperan: los faltantes se consideran undefined y los extras se ignoran.
5. Acceso dinámico a propiedades y uso de prototipos. Gracias a la notación con corchetes (obj[clave]), es posible determinar en tiempo de ejecución qué propiedad se lee o se escribe.
6. Variables sin tipo explícito y semicolons opcionales. Las declaraciones var, let y const no requieren indicar el tipo.



UNViMe

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?
   Si tiene sintaxis y semantica bien definidas por la organización europea de estándares ECMA en 1996. El estándar ECMAScript empezó a regir la forma en la que funciona el lenguaje.
2. ¿Es posible comprobar el código producido en ese lenguaje?
   - Cumplimiento de la intención del programador
     - Se puede comprobar con tests y revisiones de código.
     - Problema: al ser tipado dinámico, es más difícil garantizar ausencia de errores antes de la ejecución (ejemplo: sumar un número con un string).
   - Compilador / intérprete traduce bien
     - En JS, el motor (V8, SpiderMonkey, Chakra, etc.) actúa como intérprete/compilador JIT.
     - La semántica está definida por el estándar ECMAScript, y cada motor tiene que cumplirlo.
     - Por eso se puede afirmar que la traducción es comprobable en la medida en que sigue la especificación.
   - Correctitud de la máquina
     - Depende del hardware y del sistema operativo.
     - Si la CPU ejecuta bien y la memoria funciona, entonces la ejecución de JS será la esperada.
3. ¿Es confiable?

   Se comporta como se anuncia

   a. La semántica de JavaScript está bien definida por **ECMAScript**, y los motores modernos (V8, SpiderMonkey, etc.) la cumplen.
   b. En general, el código hace lo que el programador espera **si se entienden bien las reglas del lenguaje**.
   c. Problema: el **tipado dinámico** y la **coerción implícita** pueden llevar a resultados inesperados → baja confiabilidad si el programador no aplica buenas prácticas.

Ejemplo:

```
"5" + 1  // "51"

"5" - 1  // 4
```

No siempre es intuitivo.

2. **Detección y corrección de errores**
    - **Errores de sintaxis**: el motor los detecta inmediatamente en la ejecución y corta el programa.
    - **Errores de tiempo de ejecución**: JavaScript lanza **excepciones**, que pueden ser atrapadas con `try...catch`.

```
try {
  let x = JSON.parse("texto no válido");
} catch (err) {
  console.log("Error detectado:", err.message);
}
```

Esto permite que el programa no se interrumpa por completo y sea más confiable.

4. **Aplicaciones en tiempo real**
    a. JavaScript **no está diseñado para aplicaciones de tiempo real duro** (ejemplo: control de un marcapasos o sistemas críticos industriales).
    b. Sí se usa con éxito en **tiempo real blando** (chats, juegos online, servidores web concurrentes) porque el modelo de eventos y el bucle asíncrono permiten manejar miles de conexiones sin fallar.
5. ¿Es ortogonal?

Independencia de componentes

- **A favor:**
    - Las estructuras de control (`if`, `for`, `while`, `switch`) funcionan con cualquier expresión válida, sin importar el tipo de dato.

- Las funciones pueden devolver casi cualquier valor: números, strings, objetos, incluso otras funciones.

Los tipos son flexibles: podés pasar cualquier tipo como argumento a cualquier función.

```
function identidad(x) { return x; }

console.log(identidad(5));       // 5

console.log(identidad("hola"));  // "hola"

console.log(identidad([1,2,3])); // [1,2,3]
```

→ Esto muestra **alta ortogonalidad**.

- **En contra:**
  - El **tipado dinámico y coerciones implícitas** hacen que no siempre el resultado sea intuitivo.

    ```
    console.log("5" * 2);   // 10  (string convertido a número)

    console.log("5" + 2);   // "52" (número convertido a string)
    ```

  - El operador `+` no se comporta igual que `*`, lo cual rompe la uniformidad.

Combinar características sin perder significado

- **A favor:** podés anidar funciones, pasar funciones como argumentos, usar arrays en bucles, todo de forma consistente.
- **En contra:** existen muchas **excepciones históricas** que restan ortogonalidad:
  - **Hoisting de `var`:** variables disponibles antes de declararse.
  - **Falsy/truthy**: valores como `0`, `""`, `null`, `undefined`, `NaN` se comportan como `false`, mientras que `[]` y `{}` son `true`.

    ```
    if ([]) console.log("entra");   // sí

    if ("") console.log("entra");  // no
    ```

Esto no es ortogonal porque no sigue una regla uniforme simple.

Aspectos negativos de la ortogonalidad en JS

a. Como el lenguaje permite **casi cualquier combinación de tipos con operadores**, muchas expresiones compilan y ejecutan, pero el resultado es incoherente o inesperado.

Ejemplo:

```
console.log(true + true);   // 2

console.log(true == 1);     // true

console.log([] == 0);       // true
```

Demasiada ortogonalidad (todo se puede combinar) puede generar **resultados absurdos o ineficientes**.

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

Consistencia con notación matemática

a. **A favor:**
   i. Usa operadores familiares de matemáticas: `+`, `-`, `*`, `/`, `%`.
   ii. Prioridad de operadores es la misma que en álgebra (multiplicación antes que suma).
b. **En contra:**

   Existen casos poco consistentes:

```
"5" + 2   // "52" → concatenación, no suma

"5" - 2   // 3    → resta, convierte a número
```

   i. → El mismo símbolo `+` no siempre es suma: a veces es concatenación, lo que rompe la consistencia con la matemática.

Uniformidad de sintaxis

- **A favor:**
  - Bloques siempre delimitados con `{}`.

- Funciones siempre se declaran con `function` o con la sintaxis de flecha `() => {}`.
- Las declaraciones de variables con `let` y `const` son uniformes en alcance de bloque.

- **En contra:**
  - El **punto y coma (`;`) es opcional**, lo que rompe uniformidad:

```
let x = 5

let y = 6
```

  - Esto funciona, aunque en otros contextos puede dar errores.
  - Las declaraciones antiguas con `var` tienen reglas distintas (hoisting, alcance de función), mientras que `let`/`const` son de bloque.

Nociones similares deberían verse igual

Problemas:

ii. Comparación: `==` hace coerción, `===` no. Dos operadores distintos para lo que conceptualmente debería ser lo mismo → inconsistencia.

iii. Manejo de nulos: `null` y `undefined` existen los dos, con diferencias sutiles y a veces confusas.

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

Subconjuntos en JavaScript

1. **Strict mode**: un subconjunto más seguro del lenguaje, que evita prácticas peligrosas (ej. uso implícito de variables globales).

```
"use strict";

x = 5; // ❌ Error (sin strict sería válido)
```

1. **Versiones ECMAScript (ES3, ES5, ES6, …)**: cada versión puede considerarse un subconjunto del estándar más moderno.
2. **asm.js**: subconjunto optimizado para ejecución de alto rendimiento (usado en navegadores antes de WebAssembly).

3. **Lenguajes host**: algunos entornos limitan el acceso a APIs (por ejemplo, en navegadores no podés usar módulos de `fs` que sí existen en Node).

Extensibilidad en JavaScript

Muy alta:

a. **Core estable (ECMAScript):** el núcleo del lenguaje (sintaxis básica, tipos, funciones, estructuras de control) se mantiene.
b. **Extensiones de versión:** cada nueva edición de ECMAScript agrega características sin romper lo anterior (ej. `let/const`, arrow functions, `async/await`).
c. **Bibliotecas externas:** el ecosistema **npm** permite extender el lenguaje con módulos (ej. `prompt-sync`, `express`, `lodash`).
d. **Supersets como TypeScript:** extienden JavaScript con tipado estático y luego lo transpilan a JS puro.
e. **Web APIs y Node APIs:** cada entorno amplía las capacidades (DOM en navegador, `fs` en Node).

8. El código producido, ¿es transportable?

Si, si te atenés al estándar.

- Corre en **cualquier motor ECMAScript**: navegadores, **Node**, **Deno**.
  Para targets antiguos, **transpilación** (Babel) y **polyfills**.
- Ojo: la **portabilidad de APIs del host** varía (DOM en browser, `fs` en Node).
  Evitá dependencias del entorno si buscás portabilidad máxima.