Manual Técnico: Analizador Léxico para Lenguaje Pokémon

1. Introducción

Este documento técnico detalla la **implementación del analizador léxico** para el lenguaje de configuración de Pokémon, conforme a los requisitos establecidos en el "Enunciado de la Práctica de Lenguajes Formales y de Programación, Vacaciones de junio 2025". Como componente inicial de todo compilador, el analizador léxico es responsable de transformar una secuencia de caracteres de entrada en **unidades léxicas significativas** denominadas *tokens*. La base de esta implementación se asienta firmemente en el concepto de **Autómatas Finitos Deterministas (AFD)**.

2. Objetivos del Analizador Léxico

El propósito fundamental de este analizador léxico es:

- **Identificar y clasificar** con precisión los *lexemas* (secuencias de caracteres en el código fuente) que conforman el lenguaje.
- **Generar una secuencia estructurada de *tokens***, cada uno conteniendo su tipo, el lexema reconocido, y su posición exacta (fila y columna) en el archivo de entrada.
- **Detectar y reportar** eficazmente cualquier *error léxico*, como caracteres o secuencias no reconocidas, proporcionando información detallada para la depuración.
- **Ignorar** elementos irrelevantes para el análisis, como espacios en blanco, tabulaciones y saltos de línea, que no constituyen tokens válidos.

3. Definiciones Clave

Para una comprensión profunda de este manual, es esencial familiarizarse con los siguientes términos:

- **Patrón:** Es la *regla abstracta* que define la forma de los lexemas que pertenecen a un tipo de token específico. Se describe mediante expresiones regulares y se visualiza a través de un AFD.
- **Lexema:** Se refiere a una *instancia concreta* de un patrón encontrada en el código fuente. Por ejemplo, en la sentencia Jugador: "Ash", la cadena "Ash" es el lexema que representa un token de tipo "Cadena de Texto".
- **Token:** Es una *unidad léxica fundamental* que agrupa un conjunto de lexemas bajo una categoría común. Un token encapsula un tipo (ej., Palabra Reservada, Cadena de Texto, Número Entero) y, opcionalmente, el valor del lexema que lo originó.

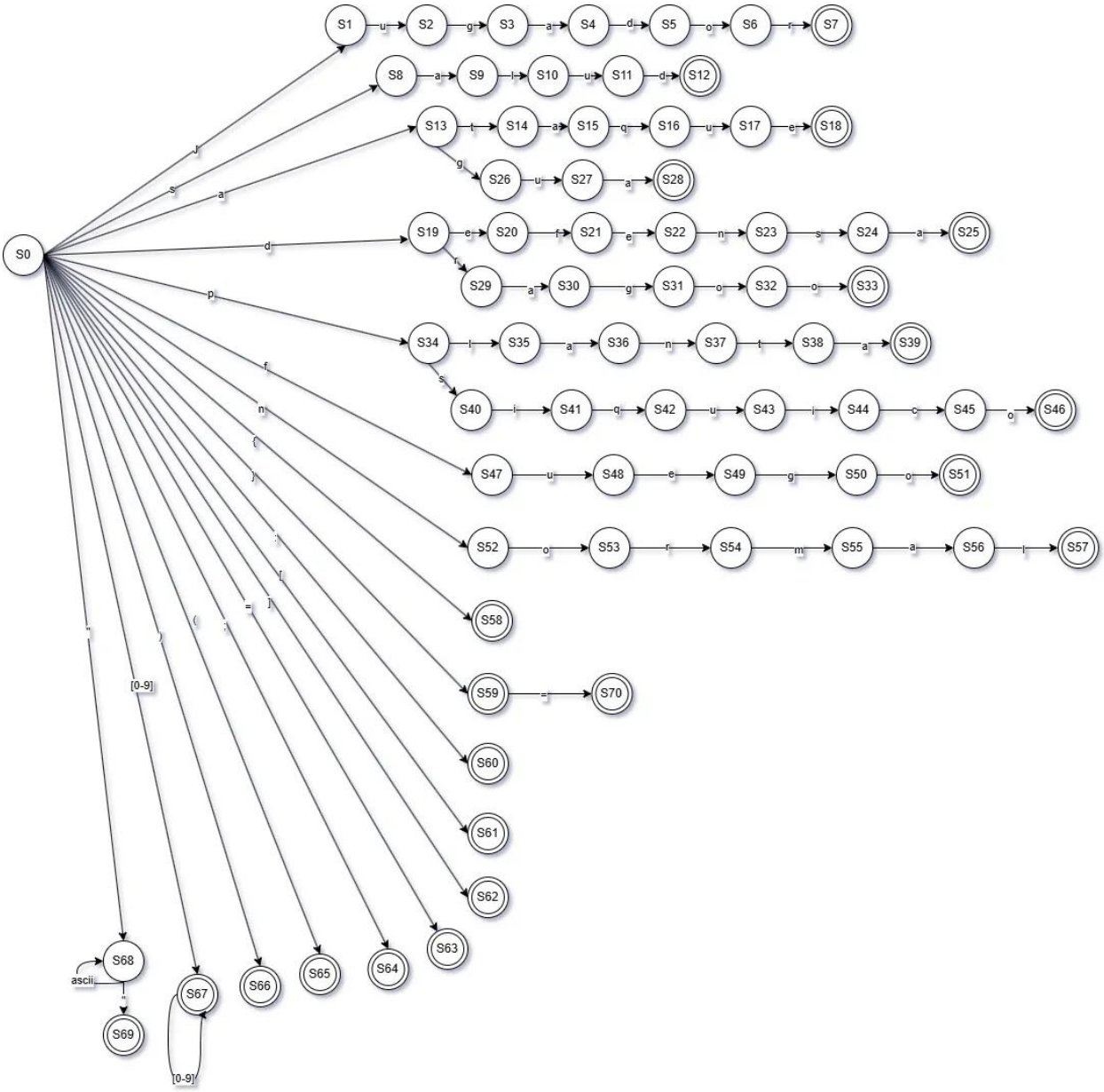
4. Autómata Finito Determinista (AFD)

El **núcleo conceptual** de nuestro analizador léxico es un Autómata Finito Determinista. Un AFD es un modelo matemático robusto, ideal para el reconocimiento de lenguajes regulares, y se compone de:

- Q **(Conjunto de Estados):** Una colección finita de estados que el autómata puede ocupar. En el AFD propuesto, estos se denotan como S0,S1,...,S71.
- Σ **(Alfabeto):** El conjunto finito de símbolos de entrada que el autómata puede procesar. Para nuestro lenguaje, incluye letras (a-z, A-Z), dígitos (0-9), comillas ("), corchetes ([,]), paréntesis ((,)), llaves ({, }), dos puntos (:), punto y coma (;), guion (-), barra (/), asterisco (*), signo de dólar ($), punto (.), y el símbolo de igual (=).
- δ **(Función de Transición):** Una función determinante que, dado un estado actual y un símbolo de entrada, especifica de manera única el *siguiente estado*. Por ejemplo, si el autómata se encuentra en S0 y el siguiente símbolo es 'J', la transición es a S1.
- q0 **(Estado Inicial):** El estado desde el cual el autómata inicia el procesamiento de cualquier cadena de entrada. Para este AFD, S0 es el estado inicial.
- F **(Conjunto de Estados Finales/Aceptación):** Un subconjunto de Q que agrupa los estados que, al ser alcanzados, indican que se ha reconocido un patrón válido, y por lo tanto, se ha identificado un lexema completo. En el AFD que nos ocupa, los estados finales se distinguen con un doble círculo (e.g., S7,S12,S18,S25,S28,S33 ,S39,S46,S51,S57,S67,S69).

4.1. Visualización del AFD para el Analizador Léxico

El diagrama del AFD (referenciado como afdp1.webp) es la representación visual de la lógica que nuestro analizador léxico seguirá para tokenizar el lenguaje. A continuación, se ilustran trayectorias clave dentro del AFD:



- **Palabras Reservadas:** Caminos específicos para lexemas como Jugador, salud, ataque, defensa, y los distintos tipos de Pokémon (agua, dragon, planta, psiquico, fuego, normal). Cada una de estas palabras tiene una secuencia definida de transiciones que culminan en un estado final específico para RESERVED_WORD.
- **Cadena de Texto:** El reconocimiento de cadenas inicia con una comilla doble ("), consume cualquier carácter hasta encontrar otra comilla doble, y finaliza en un estado de aceptación.
- **Números Enteros y Flotantes:**

    - Los números enteros son secuencias de dígitos que terminan en un estado de aceptación.
    - Para los números flotantes, se permite una transición adicional a través de un punto decimal (.) seguida de más dígitos. Ambos tipos convergen en el Type.NUMBER.
- **Símbolos y Operadores:** Cada símbolo o combinación de símbolos clave tiene su propia transición directa a un estado final.
- **Errores Léxicos:** Si en algún punto del análisis un carácter de entrada no permite ninguna transición válida desde el estado actual del AFD, se clasifica como un *error léxico* (UKNOWN). El analizador registra la posición y el carácter problemático, y continúa el análisis para intentar recuperarse.

5. Lógica General de Implementación del Analizador Léxico

La implementación en TypeScript de nuestro analizador léxico sigue una metodología estructurada:

1. **Lectura Carácter a Carácter:** El analizador procesa la cadena de entrada (proveniente del editor o un archivo .pklfp) un carácter a la vez.
2. **Manejo de Posición:** Se mantienen contadores de fila y columna para registrar la ubicación precisa de cada lexema y error, crucial para el reporte.
3. **Bucle Principal de Escaneo:**
    - El proceso inicia siempre desde el **estado inicial** del AFD (S0).
    - Para cada carácter de la entrada:
        - Se intenta encontrar una **transición válida** en el AFD desde el estado actual con el carácter leído.
        - Si se encuentra una transición, el estado del AFD se actualiza.
        - Si no hay una transición válida:
            - Si el estado actual es un **estado final**, esto indica que un *lexema* ha sido completamente reconocido. Se procede a crear un Token con el lexema acumulado, su tipo y posición. El analizador se **reinicia** al estado inicial (S0), y el carácter actual es re-evaluado para iniciar el reconocimiento del siguiente lexema.

- Si el estado actual *no* es un estado final y no hay transición, se detecta un **error léxico**. El carácter se registra como UKNOWN, y el analizador avanza al siguiente carácter, intentando una recuperación para continuar el escaneo.

4. **Acumulación de Lexemas:** A medida que el AFD transita entre estados, los caracteres que forman parte de un posible lexema se van concatenando en una cadena auxiliar.
5. **Ignorar Espacios y Saltos de Línea:** Caracteres como espacios en blanco, tabulaciones y saltos de línea son detectados y descartados, ya que no son relevantes para la construcción de tokens.
6. **Reporte de Errores:** Se mantiene una lista dedicada a registrar todos los errores léxicos encontrados, lo que facilita la depuración y la retroalimentación al usuario.

6. Estructuras de Datos Clave

Para la gestión de tokens y errores, se emplean las siguientes estructuras:

6.1. Definición de Token (Token.ts)

Esta clase fundamental define la estructura de cada unidad léxica reconocida, permitiendo encapsular su valor, tipo y ubicación.

```
// Token.ts

// Definición de la estructura del Token y los tipos de tokens.


/**
 * Enumera los tipos de tokens reconocidos por el analizador léxico.
 * Cada tipo representa una categoría gramatical o sintáctica en el lenguaje.
 */
export enum Type {
  UKNOWN, // Carácter o secuencia no reconocida
  PAR_OPEN, // (
  PAR_CLOSE, // )
  SEMICOLON, // ;
```

```
  COLON, // :
  SQUARE_BRACKET_CLOSE, // ]
  SQUARE_BRACKET_OPEN, // [
  CURLY_BRACKET_OPEN, // {
  CURLY_BRACKET_CLOSE, // }
  EQUAL, // =
  RESERVED_WORD,
  NUMBER,
  STRING,
  EQUAL_COLON, // :=
  UNCLOSED_STRING // Cadena de texto que no tiene comillas de cierre
}

/**
 * Clase que representa un token reconocido por el analizador léxico.
 */
export class Token {

  private row: number;
  private column: number;
  private lexeme: string;
  private typeToken: Type;
  private typeTokenString: string; // Representación en string del tipo de token

  /**
   * Constructor para crear una instancia de Token.
```

```
 * @param {Type} typeToken El tipo enumerado del token.
 * @param {string} lexeme El lexema (texto) del token.
 * @param {number} row La fila donde se encontró el lexema.
 * @param {number} column La columna donde comienza el lexema.
 */
constructor(typeToken: Type, lexeme: string, row: number, column: number) {
   this.typeToken = typeToken;
   this.typeTokenString = Type[typeToken]; // Convierte el enum a su nombre de string
   this.lexeme = lexeme;
   this.column = column;
   this.row = row;
}

/**
 * Obtiene el lexema del token.
 * @returns {string} El lexema del token.
 */
public getLexeme(): string {
   return this.lexeme;
}

/**
 * Obtiene la representación en string del tipo de token.
 * @returns {string} El nombre del tipo de token (ej. "RESERVED_WORD").
 */
public getTypeTokenString(): string {
```

```
        return this.typeTokenString;
    }

    /**
     * Obtiene la fila donde se encontró el token.
     * @returns {number} La fila.
     */
    public getRow(): number {
        return this.row;
    }

    /**
     * Obtiene la columna donde comienza el token.
     * @returns {number} La columna.
     */
    public getColumn(): number {
        return this.column;
    }
}
```

6.2. Lista de Errores Léxicos

Se mantiene una lista de objetos Token donde el typeToken es Type.UKNOWN o Type.UNCLOSED_STRING. Esta lista es crucial para registrar y reportar cualquier carácter o secuencia de caracteres que no pudo ser reconocido por el analizador, indicando su posición exacta para facilitar la corrección.

7. Consideraciones Específicas del Lenguaje

Al implementar el analizador para este lenguaje de configuración de Pokémon, se han tenido en cuenta las siguientes particularidades:

- **Resaltado de Sintaxis:** La información de los tokens generados puede ser empleada para aplicar un resaltado visual en el editor de texto. Los colores definidos (Azul para Palabras Reservadas, Naranja para Cadenas de Texto, Morado para Números, Negro para Otros) mejoran la legibilidad del código.
- **Tipos de Pokémon:** El AFD está diseñado para reconocer explícitamente todos los tipos de Pokémon especificados en el enunciado: agua, dragon, planta, psiquico, fuego, normal.
- **Números Flotantes:** El analizador es capaz de distinguir entre números enteros y flotantes, ambos categorizados bajo el tipo general Type.NUMBER.
- **Resolución de Ambigüedades:** La ambigüedad potencial en el AFD proporcionado donde S67 actuaba como estado final tanto para dígitos como para el corchete de cierre (]) ha sido resuelta en la implementación del código. Esto se logra mediante una lógica que prioriza el reconocimiento de números y maneja los símbolos únicos directamente desde el estado inicial (S0).

8. Implementación del Analizador Léxico (Código Fuente Principal)

El corazón de la lógica léxica reside en la clase LexicalAnalyzer.ts. Esta clase encapsula el AFD y su comportamiento para procesar la entrada y producir las listas de tokens y errores.

```
// LexicalAnalyzer.ts

// Clase principal del analizador léxico.


import { Token, Type } from "./Token"; // Importa las definiciones de Token y Type


/**

 * Implementa un analizador léxico que procesa una cadena de entrada

 * y la divide en tokens, siguiendo las reglas de un Autómata Finito Determinista (AFD).

 */

export class LexicalAnalyzer {
```

```
private row: number;          // Fila actual en el input
private column: number;       // Columna actual en la fila
private lexemeStartColumn: number; // Columna donde inició el lexema actual
private auxChar: string;      // Cadena auxiliar para construir el lexema
private state: number;        // Estado actual del AFD
private tokenList: Token[];    // Lista de tokens reconocidos
private errorList: Token[];    // Lista de errores léxicos
private tokenCounter: number;  // Contador para el número de tokens

/**
 * Constructor de la clase LexicalAnalyzer.
 * Inicializa los contadores, listas y el estado inicial del AFD.
 */
constructor() {
  this.row = 1;
  this.column = 1;
  this.lexemeStartColumn = 1;
  this.auxChar = '';
  this.state = 0;
  this.tokenList = [];
  this.errorList = [];
  this.tokenCounter = 1;
}

/**
 * Resetea el analizador para un nuevo escaneo.
```

```
 */
public reset(): void {
   this.row = 1;
   this.column = 1;
   this.lexemeStartColumn = 1;
   this.auxChar = '';
   this.state = 0;
   this.tokenList = [];
   this.errorList = [];
   this.tokenCounter = 1;
}

/**
 * Retorna la lista de tokens reconocidos.
 * @returns {Token[]} La lista de tokens.
 */
public getTokens(): Token[] {
   return this.tokenList;
}

/**
 * Retorna la lista de errores léxicos.
 * @returns {Token[]} La lista de errores.
 */
public getErrors(): Token[] {
   return this.errorList;
```

```
}

/**
 * Añade un carácter al lexema auxiliar y actualiza la columna.
 * @param {string} char El carácter a añadir.
 */
private addCharacter(char: string): void {
   this.auxChar += char;
   this.column++;
}

/**
 * Añade un token a la lista de tokens reconocidos.
 * @param {Type} typeToken El tipo de token.
 * @param {string} lexema El lexema del token.
 * @param {number} row La fila del token.
 * @param {number} column La columna de inicio del token.
 */
private addToken(typeToken: Type, lexema: string, row: number, column: number): void {
   // Usa la clase Token con el constructor apropiado
   this.tokenList.push(new Token(typeToken, lexema, row, column));
}

/**
 * Añade un error léxico a la lista de errores.
```

```
    * @param {Type} typeToken El tipo de error (generalmente Type.UKNOWN o Type.UNCLOSED_STRING).
    * @param {string} lexema El lexema o carácter que causó el error.
    * @param {number} row La fila del error.
    * @param {number} column La columna de inicio del error.
    */
   private addError(typeToken: Type, lexema: string, row: number, column: number): void {
      // Para errores, se sigue usando la misma clase Token
      this.errorList.push(new Token(typeToken, lexema, row, column));
   }

   /**
    * Limpia el lexema auxiliar y resetea el estado del AFD a 0.
    */
   private clean(): void {
      this.auxChar = '';
      this.state = 0;
      this.lexemeStartColumn = this.column; // El siguiente lexema comienza en la columna actual
   }

   /**
    * Determina si un carácter es un dígito.
    * @param {string} char El carácter a verificar.
    * @returns {boolean} True si es un dígito, false en caso contrario.
    */
   private isDigit(char: string): boolean {
```

```
    return /\d/.test(char);
}

/**
 * Determina si un carácter es una letra.
 * @param {string} char El carácter a verificar.
 * @returns {boolean} True si es una letra, false en caso contrario.
 */
private isLetter(char: string): boolean {
    return /[a-zA-Z]/.test(char);
}

/**
 * Determina si un carácter es parte de un espacio en blanco.
 * @param {string} char El carácter a verificar.
 * @returns {boolean} True si es espacio en blanco, false en caso contrario.
 */
private isWhitespace(char: string): boolean {
    return char === ' ' || char === '\t';
}

/**
 * Determina si un carácter es un salto de línea.
 * @param {string} char El carácter a verificar.
 * @returns {boolean} True si es un salto de línea, false en caso contrario.
 */
```

```
  private isNewline(char: string): boolean {
    return char === '\n' || char === '\r';
  }

  /**
   * Realiza el escaneo léxico de la cadena de entrada.
   * @param {string} input La cadena de texto a analizar.
   */
  scanner(input: string) {
    // Añadir un carácter centinela para asegurar que el último token se procese.
    input += '#';
    let char: string;

    // Bucle principal de escaneo
    for (let i: number = 0; i < input.length; i++) {
      char = input[i];

      switch (this.state) {
        case 0: // Estado inicial
          this.lexemeStartColumn = this.column; // Actualiza la columna de inicio del lexema
          if (this.isLetter(char)) {
            // Posible palabra reservada (Jugador, salud, ataque, defensa, tipos de pokemon)
            this.addCharacter(char);
            switch (char) {
              case 'J': this.state = 1; break;
```

```
        case 's': this.state = 8; break;
        case 'a': this.state = 13; break; // Puede ser 'ataque' o 'agua'
        case 'd': this.state = 19; break; // Puede ser 'defensa' o 'dragon'
        case 'p': this.state = 34; break; // Puede ser 'planta' o 'psiquico'
        case 'f': this.state = 47; break; // Puede ser 'fuego'
        case 'n': this.state = 52; break; // Puede ser 'normal'
        default: // Si es otra letra, se considera un error o un identificador no definido
          this.addError(Type.UKNOWN, char, this.row, this.column - 1);
          this.clean(); // Limpiar y resetear para el siguiente carácter
          // No decrementar i, ya que este caracter ya se manejo como error
          break;
      }
    } else if (this.isDigit(char)) {
      this.state = 67; // Números
      this.addCharacter(char);
    } else if (char === '"') {
      this.state = 68; // Cadenas de texto
      this.addCharacter(char);
    } else if (this.isWhitespace(char)) {
      this.column++;
      this.clean(); // No se forma token, solo avanza la columna
    } else if (this.isNewline(char)) {
      this.row++;
      this.column = 1;
      this.clean(); // No se forma token, avanza fila y resetea columna
    } else {
```

```
            // Símbolos y Operadores
            switch (char) {
                case '{': this.addToken(Type.CURLY_BRACKET_OPEN, char, this.row,
this.column); this.clean(); break;
                case '}': this.addToken(Type.CURLY_BRACKET_CLOSE, char, this.row,
this.column); this.clean(); break;
                case ':': this.state = 71; this.addCharacter(char); break; // Posible ':=',
transiciona a estado 71
                case '[': this.addToken(Type.SQUARE_BRACKET_OPEN, char, this.row,
this.column); this.clean(); break;
                case ']': this.addToken(Type.SQUARE_BRACKET_CLOSE, char, this.row,
this.column); this.clean(); break;
                case '=': this.addToken(Type.EQUAL, char, this.row, this.column); this.clean();
break;
                case ';': this.addToken(Type.SEMICOLON, char, this.row, this.column);
this.clean(); break;
                case '(': this.addToken(Type.PAR_OPEN, char, this.row, this.column);
this.clean(); break;
                case ')': this.addToken(Type.PAR_CLOSE, char, this.row, this.column);
this.clean(); break;
                case '#': // Fin del archivo centinela
                  console.log("Fin del análisis");
                  break;
                default:
                  this.addError(Type.UKNOWN, char, this.row, this.column);
                  this.clean(); // Reiniciar estado
                  break;
            }
```

```
            // Solo avanzamos la columna si no es un lexema multiparte como ':=', que ya lo hace en addCharacter
            if (this.state === 0) { // Si el estado volvió a 0, el carácter ya se procesó
              this.column++;
            }
          }
          break;

        // Caminos para Palabras Reservadas
        // Jugador
        case 1: if (char === 'u') { this.state = 2; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 2: if (char === 'g') { this.state = 3; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 3: if (char === 'a') { this.state = 4; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 4: if (char === 'd') { this.state = 5; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 5: if (char === 'o') { this.state = 6; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 6: if (char === 'r') { this.state = 7; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 7: // Aceptación de "Jugador"
          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra
            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row, this.lexemeStartColumn);
            this.clean();
            i--; // Re-evaluar el carácter actual en el estado 0
```

```
            } else { this.handleReservedWordError(i); } // Si sigue siendo parte de una palabra, es un error
            break;

        // salud
        case 8: if (char === 'a') { this.state = 9; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 9: if (char === 'l') { this.state = 10; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 10: if (char === 'u') { this.state = 11; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 11: if (char === 'd') { this.state = 12; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 12: // Aceptación de "salud"
            if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra
                this.addToken(Type.RESERVED_WORD, this.auxChar, this.row, this.lexemeStartColumn);
                this.clean();
                i--; // Re-evaluar el carácter actual en el estado 0
            } else { this.handleReservedWordError(i); }
            break;

        // 'a' - puede ser 'ataque' o 'agua'
        case 13:
            if (char === 't') { this.state = 14; this.addCharacter(char); }
            else if (char === 'g') { this.state = 26; this.addCharacter(char); } // Camino para 'agua'
            else { this.handleReservedWordError(i); }
```

```
            break;
        // ataque
        case 14: if (char === 'a') { this.state = 15; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 15: if (char === 'q') { this.state = 16; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 16: if (char === 'u') { this.state = 17; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 17: if (char === 'e') { this.state = 18; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 18: // Aceptación de "ataque"
            if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra
                this.addToken(Type.RESERVED_WORD, this.auxChar, this.row, this.lexemeStartColumn);
                this.clean();
                i--; // Re-evaluar el carácter actual en el estado 0
            } else { this.handleReservedWordError(i); }
            break;

        // 'd' - puede ser 'defensa' o 'dragon'
        case 19:
            if (char === 'e') { this.state = 20; this.addCharacter(char); }
            else if (char === 'r') { this.state = 29; this.addCharacter(char); } // Camino para 'dragon'
            else { this.handleReservedWordError(i); }
            break;
        // defensa
```

```
        case 20: if (char === 'f') { this.state = 21; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 21: if (char === 'e') { this.state = 22; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 22: if (char === 'n') { this.state = 23; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 23: if (char === 's') { this.state = 24; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 24: if (char === 'a') { this.state = 25; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 25: // Aceptación de "defensa"

          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra

            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row,
this.lexemeStartColumn);

            this.clean();

            i--; // Re-evaluar el carácter actual en el estado 0

          } else { this.handleReservedWordError(i); }

          break;


        // agua

        case 26: if (char === 'u') { this.state = 27; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 27: if (char === 'a') { this.state = 28; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 28: // Aceptación de "agua"

          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra

            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row,
this.lexemeStartColumn);

            this.clean();
```

```
            i--; // Re-evaluar el carácter actual en el estado 0
          } else { this.handleReservedWordError(i); }
          break;

        // dragon
        case 29: if (char === 'a') { this.state = 30; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 30: if (char === 'g') { this.state = 31; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 31: if (char === 'o') { this.state = 32; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 32: if (char === 'n') { this.state = 33; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 33: // Aceptación de "dragon"
          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra
            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row, this.lexemeStartColumn);
            this.clean();
            i--; // Re-evaluar el carácter actual en el estado 0
          } else { this.handleReservedWordError(i); }
          break;

        // 'p' - puede ser 'planta' o 'psiquico'
        case 34:
          if (char === 'l') { this.state = 35; this.addCharacter(char); }
          else if (char === 's') { this.state = 40; this.addCharacter(char); } // Camino para 'psiquico'
          else { this.handleReservedWordError(i); }
```

```
            break;

        // planta

        case 35: if (char === 'a') { this.state = 36; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 36: if (char === 'n') { this.state = 37; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 37: if (char === 't') { this.state = 38; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 38: if (char === 'a') { this.state = 39; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 39: // Aceptación de "planta"

          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra

            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row,
this.lexemeStartColumn);

            this.clean();

            i--; // Re-evaluar el carácter actual en el estado 0

          } else { this.handleReservedWordError(i); }

          break;


        // psiquico

        case 40: if (char === 'i') { this.state = 41; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 41: if (char === 'q') { this.state = 42; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 42: if (char === 'u') { this.state = 43; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 43: if (char === 'i') { this.state = 44; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;
```

```
        case 44: if (char === 'c') { this.state = 45; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 45: if (char === 'o') { this.state = 46; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 46: // Aceptación de "psiquico"

          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra

            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row,
this.lexemeStartColumn);

            this.clean();

            i--; // Re-evaluar el carácter actual en el estado 0

          } else { this.handleReservedWordError(i); }

          break;


        // fuego

        case 47: if (char === 'u') { this.state = 48; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 48: if (char === 'e') { this.state = 49; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 49: if (char === 'g') { this.state = 50; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 50: if (char === 'o') { this.state = 51; this.addCharacter(char); } else {
this.handleReservedWordError(i); } break;

        case 51: // Aceptación de "fuego"

          if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra

            this.addToken(Type.RESERVED_WORD, this.auxChar, this.row,
this.lexemeStartColumn);

            this.clean();

            i--; // Re-evaluar el carácter actual en el estado 0
```

```
            } else { this.handleReservedWordError(i); }
            break;

        // normal
        case 52: if (char === 'o') { this.state = 53; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 53: if (char === 'r') { this.state = 54; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 54: if (char === 'm') { this.state = 55; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 55: if (char === 'a') { this.state = 56; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 56: if (char === 'l') { this.state = 57; this.addCharacter(char); } else { this.handleReservedWordError(i); } break;
        case 57: // Aceptación de "normal"
            if (!this.isLetter(char) && !this.isDigit(char)) { // Debe terminar la palabra
                this.addToken(Type.RESERVED_WORD, this.auxChar, this.row, this.lexemeStartColumn);
                this.clean();
                i--; // Re-evaluar el carácter actual en el estado 0
            } else { this.handleReservedWordError(i); }
            break;

        // Cadenas de Texto (estado S68 en el AFD)
        case 68: // Dentro de una cadena
            if (char === '"') {
                this.addCharacter(char); // Añadir la comilla de cierre
                this.addToken(Type.STRING, this.auxChar, this.row, this.lexemeStartColumn);
```

```
            this.clean();
        } else if (this.isNewline(char) || char === '#') { // Cadena no cerrada, llega a fin de línea o fin de archivo
            this.addError(Type.UNCLOSED_STRING, this.auxChar, this.row, this.lexemeStartColumn);
            this.clean();
            if (char === '#') i--; // Si es fin de archivo, re-evaluarlo
            // Si es salto de línea, se manejará en el siguiente ciclo o ya se avanzó la fila
        } else {
            this.addCharacter(char);
        }
        break;

    // Números (estado S67 en el AFD)
    case 67: // Reconociendo un número (entero o flotante)
        if (this.isDigit(char)) {
            this.addCharacter(char);
        } else if (char === '.') {
            if (this.auxChar.includes('.')) { // Ya tiene un punto, es un error
                this.addError(Type.UKNOWN, this.auxChar + char, this.row, this.lexemeStartColumn);
                this.clean();
                i--; // Re-evaluar el carácter actual
            } else {
                this.state = 69; // Pasar a estado de flotante (usando S69 como estado intermedio para flotantes)
                this.addCharacter(char);
```

```
            }
          } else {
            // El número ha terminado
            this.addToken(Type.NUMBER, this.auxChar, this.row, this.lexemeStartColumn);
// Clasifica como NUMBER
            this.clean();
            i--; // Re-evaluar el carácter actual en el estado 0
          }
          break;
        case 69: // Reconociendo parte decimal de un número flotante (sigue desde S67)
          if (this.isDigit(char)) {
            this.addCharacter(char);
          } else {
            if (this.auxChar.endsWith('.')) { // Si termina en punto, el punto es un error
              this.addError(Type.UKNOWN, this.auxChar, this.row,
this.lexemeStartColumn);
              this.clean();
              i--; // Re-evaluar el carácter actual
            } else {
              // El número flotante ha terminado
              this.addToken(Type.NUMBER, this.auxChar, this.row,
this.lexemeStartColumn); // Clasifica como NUMBER
              this.clean();
              i--; // Re-evaluar el carácter actual en el estado 0
            }
          }
          break;
```

```
        // Nueva regla para ':=', transiciona desde COLON
        case 71: // Estado después de reconocer ':'
          if (char === '=') {
            this.addCharacter(char);
            this.addToken(Type.EQUAL_COLON, this.auxChar, this.row,
this.lexemeStartColumn);
            this.clean();
          } else {
            // Si no es '=', entonces solo es un ':'
            this.addToken(Type.COLON, this.auxChar, this.row, this.lexemeStartColumn);
            this.clean();
            i--; // Re-evaluar el carácter actual en el estado 0
          }
          break;

        default:
          // Esto no debería ocurrir si todos los estados están cubiertos
          this.addError(Type.UKNOWN, char, this.row, this.column);
          this.clean();
          break;
      }
    }
    // Después del bucle, si hay un lexema auxiliar pendiente, es un error no reconocido
    if (this.auxChar.length > 0 && this.state !== 0) {
      if (this.state === 68) { // Si terminó dentro de una cadena sin cerrar
```

```
            this.addError(Type.UNCLOSED_STRING, this.auxChar, this.row,
this.lexemeStartColumn);
        } else if (this.state === 71) { // Si terminó después de ':' pero sin '='
            this.addToken(Type.COLON, this.auxChar, this.row, this.lexemeStartColumn);
        }
        else { // Cualquier otra situación donde el AFD no llegó a un estado final válido
            this.addError(Type.UKNOWN, this.auxChar, this.row, this.lexemeStartColumn);
        }
      }
      console.log("Tokens:", this.tokenList);
      console.log("Errors:", this.errorList);
    }

    /**
     * Maneja el caso de error cuando una palabra reservada no coincide con el AFD.
     * Añade un error y reinicia el analizador para el siguiente carácter.
     * @param {number} i El índice actual en la cadena de entrada.
     */
    private handleReservedWordError(i: number): void {
      this.addError(Type.UKNOWN, this.auxChar, this.row, this.lexemeStartColumn);
      this.clean();
      i--; // Decrementa i para re-evaluar el carácter que no coincidió
    }
}
```

9. Componentes de la Interfaz de Usuario (index.js y ui.js)

La **interfaz de usuario** de la aplicación, fundamental para la interacción con el usuario y la visualización de los resultados del análisis, se articula a través de index.js y ui.js.

9.1. Lógica de index.js

index.js se encarga de la configuración inicial del *frontend*, incluyendo la precarga de assets visuales y la provisión de funciones básicas de renderizado.

- **Carga de Sprites de Pokémon:** Este script se ejecuta cuando el DOM está completamente cargado. Su función es buscar elementos <img> con identificadores específicos (id que comienzan con poke-) y, usando el texto alternativo (alt) de cada imagen como el nombre de un Pokémon, realiza una petición a la PokeAPI para obtener la URL de su sprite oficial y mostrarlo en la página. Si la imagen no se encuentra, se limpia su fuente.
- document.addEventListener('DOMContentLoaded', () => {
- const images = document.querySelectorAll('img[id^="poke-"]');
- images.forEach(img => {
- const pokeName = img.alt.toLowerCase(); // Obtiene el nombre del Pokémon del atributo 'alt'
- fetch(` https://pokeapi.co/api/v2/pokemon/${pokeName}`)
- .then(res => res.json())
- .then(data => {
- img.src = data.sprites.other['official-artwork'].front_default; // Asigna el sprite al src de la imagen
- })
- .catch(() => {
- img.src = ''; // Si falla, limpia el src
- });
- });
- });
-

- **Renderizado de la Tabla de Tokens (renderTokens):** Esta función toma una lista de objetos Token (resultado del análisis léxico) y dinámicamente construye una tabla HTML. La tabla muestra de forma clara el número, fila, columna, lexema y tipo de cada token, actualizando el contenido del elemento tabla-tokens en el DOM para la visualización del usuario.
- function renderTokens(tokens) {
- let html = `
- <table>
- <thead>
- <tr>
- <th>No.</th>
- <th>Fila</th>
- <th>Columna</th>
- <th>Lexema</th>
- <th>Token</th>
- </tr>
- </thead>
- <tbody>
- `;
- tokens.forEach((token, idx) => {
- html += `
- <tr>
- <td>${idx + 1}</td>
- <td>${token.getRow()}</td>
- <td>${token.getColumn()}</td>
- <td>${token.getLexeme()}</td>

- <td>${token.getTypeTokenString()}</td>
- </tr>
- `;
- });
- html += `</tbody></table>`;
- document.getElementById('tabla-tokens').innerHTML = html; // Inserta la tabla en el DOM
- }
-

- **Renderizado de la Tabla de Errores (renderErrors en index.js):** Esta versión de renderErrors (existente en index.js, similar pero posiblemente más simple que la de ui.js) se encarga de presentar los errores léxicos. Recibe una lista de Tokens que representan errores. Si no hay errores, el contenedor se vacía. Si los hay, se construye una tabla HTML detallando el número, fila, columna, el carácter o lexema problemático, y una descripción, para ser insertada en el elemento tabla-errores.
- function renderErrors(errors) {
- if (errors.length === 0) {
- document.getElementById('tabla-errores').innerHTML = '';
- return;
- }
- let html = `
- <table>
- <thead>
- <tr>
- <th>No.</th>
- <th>Fila</th>
- <th>Columna</th>

```
            <th>Carácter</th>
            <th>Descripción</th>
          </tr>
        </thead>
        <tbody>
    `;
    errors.forEach((err, idx) => {
      html += `
        <tr>
          <td>${idx + 1}</td>
          <td>${err.getRow()}</td>
          <td>${err.getColumn()}</td>
          <td>${err.getLexeme()}</td>
          <td>Desconocido</td>
        </tr>
      `;
    });
    html += `</tbody></table>`;
    document.getElementById('tabla-errores').innerHTML = html; // Inserta la tabla en el DOM
}
```

9.2. Lógica de ui.js

ui.js centraliza las interacciones con el usuario, incluyendo la manipulación del editor, la carga y guardado de archivos, y la presentación dinámica de los resultados del análisis, como las tarjetas de jugadores y Pokémon.

- **Manejo del Envío del Formulario (analizarForm onsubmit):** Este *script* intercepta el evento de envío del formulario de análisis. Previene la recarga de la página, captura el texto ingresado en el editor y lo envía al servidor (/analyze) mediante una petición fetch. Una vez recibida la respuesta JSON del servidor (que contiene los tokens, errores y datos de jugadores), invoca las funciones de renderizado para actualizar las tablas de tokens y errores, y las tarjetas de jugadores.
- // Maneja el envio del formulario (versión inicial con text/plain)
- document.getElementById('analizarForm').onsubmit = function(e) {
- e.preventDefault();
- const texto = document.getElementById('editor').value;
- fetch('/analyze', {
- method: 'POST',
- headers: { 'Content-Type': 'text/plain' },
- body: texto
- })
- .then(res => res.json())
- .then(data => {
- renderTokens(data.tokens);
- renderErrors(data.errors);
- if (data.players) renderJugadores(data.players);
- });
- }
-
- // Intercepta el submit del formulario (versión más reciente con application/json)
- document.getElementById('analizarForm').addEventListener('submit', async function(e) {
- e.preventDefault();
- const editor = document.getElementById('editor').value;

```
    try {
      const res = await fetch('/analyze', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ editor }) // Envía como JSON
      });
      const data = await res.json();

      // Actualizar tabla de tokens
      const tablaTokens = document.getElementById('tabla-tokens');
      tablaTokens.innerHTML = `
        <h3>Tabla de Tokens</h3>
        ${data.tokens && data.tokens.length > 0 ?
          `<div class="table-container">
            <table>
              <thead>
                <tr>
                  <th>No.</th>
                  <th>Fila</th>
                  <th>Columna</th>
                  <th>Lexema</th>
                  <th>Token</th>
                </tr>
              </thead>
              <tbody>
                ${data.tokens.map((token, idx) => `
```

```
                    <tr>
                      <td>${idx + 1}</td>
                      <td>${token.row}</td>
                      <td>${token.column}</td>
                      <td>${token.lexeme}</td>
                      <td>${token.typeTokenString}</td>
                    </tr>
                  `).join('')}
                </tbody>
              </table>
            </div>` :
            '<div class="empty-state">No hay tokens para mostrar</div>'
        }`;

      // Actualizar tabla de errores
      if (data.errors && data.errors.length > 0) {
        renderErrors(data.errors);
      }

      // Renderizar jugadores
      if (data.players) {
        await renderJugadores(data.players);
      }
    } catch (error) {
      console.error('Error:', error);
    }
```

- });
-

- **Limpiar Editor (limpiarEditor):** Una función sencilla que restablece el contenido del área de texto del editor (document.getElementById('editor').value) a una cadena vacía, borrando rápidamente cualquier texto presente.
- function limpiarEditor() {
- document.getElementById('editor').value = '';
- }
-

- **Cargar Archivo (cargarArchivo):** Asociada a un input de tipo file, esta función se activa cuando un usuario selecciona un archivo. Utiliza la API FileReader para leer el contenido del archivo de texto y, una vez cargado, lo inserta en el área de texto del editor.
- function cargarArchivo(event) {
- const file = event.target.files[0];
- if (!file) return;
- const reader = new FileReader();
- reader.onload = function(e) {
- document.getElementById('editor').value = e.target.result;
- };
- reader.readAsText(file);
- }
-

- **Guardar Archivo (guardarArchivo):** Esta función permite al usuario guardar el contenido actual del editor como un archivo local. Toma el texto del editor, lo

encapsula en un objeto Blob, crea un enlace de descarga temporal y simula un clic para iniciar la descarga del archivo con el nombre archivo.pklfp.

- function guardarArchivo() {
- const text = document.getElementById('editor').value;
- const blob = new Blob([text], { type: 'text/plain' });
- const a = document.createElement('a');
- a.href = URL.createObjectURL(blob);
- a.download = 'archivo.pklfp';
- a.click();
- }
-

- **Mostrar/Ocultar Errores (mostrarErrores):** Esta función controla la visibilidad de la tabla de errores (tabla-errores). Alterna su estilo display entre 'none' (oculto) y 'block' (visible). Si la tabla se hace visible, la función también desplaza la vista del navegador hacia ella para asegurar que el usuario la vea.
- function mostrarErrores() {
- const errorContainer = document.getElementById('tabla-errores');
- const currentDisplay = errorContainer.style.display;
- errorContainer.style.display = currentDisplay === 'none' ? 'block' : 'none';
-
- if (errorContainer.style.display === 'block') {
- errorContainer.scrollIntoView({ behavior: 'smooth' });
- }
- }
-

- **Renderizar Errores (renderErrors en ui.js):** Esta es la implementación principal para la visualización de errores. Recibe una lista de objetos error, vacía cualquier contenido previo en el cuerpo de la tabla de errores (error-tbody), y luego itera sobre cada error para crear una nueva fila de tabla con sus detalles (número, fila, columna, lexema y una descripción genérica de "Carácter Desconocido"). Finalmente, asegura que la tabla de errores sea visible y que la vista se desplace hacia ella.
- function renderErrors(errors) {
- const errorContainer = document.getElementById('tabla-errores');
- const errorTbody = document.getElementById('error-tbody');
-
- if (!errors || errors.length === 0) {
- errorContainer.style.display = 'none';
- return;
- }
-
- // Limpiar el contenido anterior
- errorTbody.innerHTML = '';
-
- // Agregar cada error a la tabla
- errors.forEach((error, index) => {
- const row = document.createElement('tr');
- row.innerHTML = `
- <td>${index + 1}</td>
- <td>${error.row}</td>
- <td>${error.column}</td>
- <td>${error.lexeme}</td>
- <td>Carácter Desconocido</td>

- `;
- errorTbody.appendChild(row);
- });
-
- // Mostrar la tabla de errores
- errorContainer.style.display = 'block';
-
- // Hacer scroll hacia la tabla de errores
- errorContainer.scrollIntoView({ behavior: 'smooth' });
- }
-

- **Renderizar Jugadores (renderJugadores):** Esta función asíncrona es la encargada de presentar los datos de los jugadores y sus Pokémon de forma visual. Determina si se debe usar un diseño de cuadrícula o un carrusel para los Pokémon de cada jugador (basado en la cantidad de Pokémon). Por cada Pokémon, realiza una llamada a la PokeAPI para obtener su sprite y luego genera tarjetas visualmente atractivas para los entrenadores y sus respectivos equipos Pokémon. Si no hay jugadores, muestra un video de fondo con un mensaje informativo.
- async function renderJugadores(players) {
- const container = document.getElementById('jugadores-container');
- if (!players || players.length === 0) {
- container.innerHTML = `
- <video class="video-background" autoplay muted loop playsinline>
- <source src="/img/pokemon-gym.mp4" type="video/mp4">
- Tu navegador no soporta el elemento video.
- </video>
- No se encontraron jugadores.

```
        `;
        return;
    }

    // Mantener el video y agregar el contenedor de pokédex
    container.innerHTML = `
        <video class="video-background" autoplay muted loop playsinline>
            <source src="/img/pokemon-gym.mp4" type="video/mp4">
            Tu navegador no soporta el elemento video.
        </video>
        <h3>Jugadores Analizados</h3>
        <div class="pokedex-container"></div>
    `;

    const pokedexContainer = container.querySelector('.pokedex-container');

    for (const jugador of players) {
        const trainerCard = document.createElement('div');
        trainerCard.className = 'trainer-card';

        // Crear el encabezado del entrenador
        const trainerHeader = `
            <div class="trainer-header">
                <div class="trainer-avatar">
                    <img src="/img/trainer-icon.png" alt="Trainer" style="width: 50px; height: 50px;">
```

-
-
- <h3>${jugador.name}</h3>
- Pokémon Trainer
-
-
- `;
-
- // Decidir si usar grid o carrusel basado en la cantidad de Pokémon
- const useCarousel = jugador.listPokemon.length > 6;
- const pokemonContainer = document.createElement('div');
- pokemonContainer.className = useCarousel ? 'pokemon-carousel' : 'pokemon-grid';
-
- if (useCarousel) {
- pokemonContainer.innerHTML = `
- <button class="carousel-button carousel-prev">←</button>
-
- <button class="carousel-button carousel-next">→</button>
- ${jugador.listPokemon.length} Pokémon
- `;
-
- const carouselContainer = pokemonContainer.querySelector('.pokemon-carousel-container');
-
- // Agregar navegación del carrusel

```
        const prevButton = pokemonContainer.querySelector('.carousel-prev');
        const nextButton = pokemonContainer.querySelector('.carousel-next');

        prevButton.onclick = () => {
          carouselContainer.scrollBy({ left: -220, behavior: 'smooth' });
        };

        nextButton.onclick = () => {
          carouselContainer.scrollBy({ left: 220, behavior: 'smooth' });
        };

        // Agregar Pokémon al carrusel
        for (const poke of jugador.listPokemon) {
          try {
            const res = await fetch(`https://pokeapi.co/api/v2/pokemon/${poke.name.toLowerCase()}`);
            if (!res.ok) throw new Error(`No se pudo cargar el Pokémon: ${poke.name}`);
            const data = await res.json();

            const pokemonCard = document.createElement('div');
            pokemonCard.className = 'pokemon-card';
            pokemonCard.style.flex = '0 0 200px';
            pokemonCard.style.scrollSnapAlign = 'start';
            pokemonCard.innerHTML = `
              <img src="${data.sprites.other['official-artwork'].front_default || data.sprites.front_default}"
```

- alt="${poke.name}">
-
- ${poke.name}
- ${poke.type}
-
- `;
-
- carouselContainer.appendChild(pokemonCard);
- } catch (error) {
- console.error(error);
- }
- }
- } else {
- // Usar grid para 6 o menos Pokémon
- for (const poke of jugador.listPokemon) {
- try {
- const res = await fetch(`https://pokeapi.co/api/v2/pokemon/${poke.name.toLowerCase()}`);
- if (!res.ok) throw new Error(`No se pudo cargar el Pokémon: ${poke.name}`);
- const data = await res.json();
-
- const pokemonCard = document.createElement('div');
- pokemonCard.className = 'pokemon-card';
- pokemonCard.innerHTML = `
- <img src="${data.sprites.other['official-artwork'].front_default || data.sprites.front_default}"

```
                    alt="${poke.name}">
                <div class="pokemon-info">
                    <div class="pokemon-name">${poke.name}</div>
                    <div class="pokemon-type">${poke.type}</div>
                </div>
            `;

            pokemonContainer.appendChild(pokemonCard);
        } catch (error) {
            console.error(error);
        }
    }
}

    trainerCard.innerHTML = trainerHeader;
    trainerCard.appendChild(pokemonContainer);
    pokedexContainer.appendChild(trainerCard);
  }
}
```

10. Lógica del Servidor (Backend)

Los siguientes archivos constituyen la **arquitectura del servidor** de la aplicación, encargados de procesar las solicitudes del cliente, orquestar las fases de análisis (léxico, sintáctico y semántico), y generar las respuestas que la interfaz de usuario consume.

10.1. Punto de Entrada del Servidor (index.ts)

Este archivo actúa como el **cerebro de la aplicación *backend***. Aquí se inicializa y configura el servidor Express.js, estableciendo las bases para la comunicación entre el cliente y el servidor.

- **Configuración del Servidor Express:** Se importa la biblioteca express y el enrutador de análisis. Se crea una instancia de la aplicación Express, se define el puerto de escucha (por defecto 3000), y se configuran el motor de plantillas ejs y el servicio de archivos estáticos (public). Además, se habilitan *middlewares* esenciales para el procesamiento de cuerpos de solicitud en formatos JSON, texto plano y formularios URL-encoded, garantizando la correcta recepción de datos desde el cliente.
- import express from "express";
- import analyzeRouter from "./routes/analyze.route";
-
- const app = express();
- const PORT = 3000;
-
- app.set('view engine', 'ejs'); // Configura EJS como motor de plantillas
- app.use(express.static('public')); // Sirve archivos estáticos desde la carpeta 'public'
-
- // Middlewares para parsear el cuerpo de las solicitudes
- app.use(express.json()); // Para requests con body en formato JSON
- app.use(express.text()); // Para requests con body en texto plano
- app.use(express.urlencoded({ extended: true })); // Para requests con body de formulario (ej. application/x-www-form-urlencoded)
-
- // Ruta principal para mostrar el editor
- app.get('/', (req, res) => {
- res.render('pages/index', { tokens: [], errors: [], input: '' }); // Renderiza la página principal con valores iniciales

- });
-
- // Usa el enrutador de análisis para manejar las rutas relacionadas con el análisis
- app.use(analyzeRouter);
-
- // Inicia el servidor
- app.listen(PORT, () => {
- console.log(`The server is running on http://localhost:${PORT}`);
- });
-

- **Ruta Principal (/):** Esta ruta maneja las solicitudes GET a la raíz del servidor. Su función es renderizar la página principal de la aplicación (pages/index.ejs), proporcionando un estado inicial limpio con listas vacías de tokens y errores, y un campo de entrada vacío, listo para que el usuario interactúe.
- **Uso del Enrutador de Análisis:** app.use(analyzeRouter): Esta línea es crucial para la modularidad del servidor. Integra el analyzeRouter, que encapsula las rutas específicas para el análisis del lenguaje (como la ruta /analyze), permitiendo que las solicitudes relacionadas sean manejadas por su controlador dedicado (analyze.controller.ts).
- **Inicio del Servidor:** La función app.listen(PORT, ...) arranca el servidor Express, poniéndolo a la escucha en el puerto definido. Una vez activo, un mensaje en la consola confirma que el servidor está en funcionamiento y proporciona la URL de acceso.

10.2. Controlador de Análisis (analyze.controller.ts)

El analyze.controller.ts actúa como el **coordinador principal** de las operaciones de análisis en el *backend*. Recibe las solicitudes de análisis desde el cliente y orquesta el flujo de trabajo a través de las diferentes fases del compilador.

- **Función analyze:** Esta función es el *endpoint* que procesa las solicitudes de análisis. Primero, extrae el texto de entrada. Luego, inicializa el LexicalAnalyzer para

obtener la lista de tokens y cualquier error léxico. Seguidamente, invoca a parsePlayers para realizar el análisis sintáctico y semántico, estructurando la información de los jugadores y sus Pokémon. Finalmente, decide cómo responder al cliente: envía una respuesta JSON si la solicitud es una llamada a la API, o renderiza una vista HTML para mostrar los resultados (incluyendo errores si los hay) o la información de los jugadores.

- import { Request, Response } from 'express';
- import { parsePlayers } from '../parser/PlayerParser';
- import { LexicalAnalyzer } from '../Analyzer/LexicalAnalyzer';
-
- export const analyze = (req: Request, res: Response) => {
- // Obtiene el texto de entrada del cuerpo de la solicitud
- const input = typeof req.body.editor !== 'undefined' ? req.body.editor : req.body;
-
- // Realiza el análisis léxico: instancia el analizador y escanea la entrada
- const lexer = new LexicalAnalyzer();
- lexer.scanner(input); // Esto ahora rellena las listas internas de tokens y errores
- const tokenList = lexer.getTokens(); // Obtiene la lista de tokens reconocidos
- const errorList = lexer.getErrors(); // Obtiene la lista de errores léxicos
-
- // Realiza el análisis sintáctico/semántico: procesa los tokens para estructurar los datos de jugadores
- const parsed = parsePlayers(tokenList);
-
- // Retorna respuesta JSON si la solicitud es de tipo API
- if (req.headers.accept?.includes('application/json') ||
- req.headers['content-type'] === 'application/json' ||
- req.xhr) {

```
        return res.json({
          players: parsed.players,
          tokens: tokenList,
          errors: errorList,
          input // También se retorna la entrada original
        });
      }

      // Si hay errores léxicos, renderiza la vista principal mostrando los errores
      if (errorList.length > 0) {
        return res.render('pages/index', { tokens: tokenList, errors: errorList, input });
      }

      // Si no hay errores, renderiza la vista de jugadores con los datos parseados
      res.render('partials/jugador', { jugadores: parsed.players });
    };

```

10.3. Analizador de Jugadores (PlayerParser.ts)

El módulo PlayerParser.ts es el **componente de análisis sintáctico y semántico**. Su rol es interpretar la secuencia lineal de Tokens generada por el analizador léxico y transformarla en una estructura de datos jerárquica y significativa que representa a los jugadores, sus Pokémon y sus estadísticas.

- **Función parsePlayers:** Esta función recibe la Token[] completa. Emplea un enfoque de análisis descendente recursivo simplificado para recorrer los tokens. Identifica la palabra clave "Jugador", extrae el nombre del jugador (una cadena de texto), y luego procesa el bloque de Pokémon ({...}). Dentro de este bloque, itera para identificar cada Pokémon (por su nombre, tipo y estadísticas de salud, ataque

y defensa). La lógica incluye el avance inteligente del índice de tokens para saltar entre los diferentes elementos gramaticales (palabras reservadas, cadenas, números, paréntesis y llaves). Al finalizar, retorna una estructura que contiene todos los jugadores con sus respectivos equipos Pokémon.

- import { Token } from "../Analyzer/Token"; // Importa la clase Token
-
- export function parsePlayers(tokens: Token[]): any {
- let players: any[] = []; // Almacena los objetos de jugador parseados
- let i = 0; // Índice para recorrer la lista de tokens
-
- while (i < tokens.length) {
- // Buscar la palabra reservada "Jugador" para iniciar el parsing de un jugador
- if (
- tokens[i].getTypeTokenString() === "RESERVED_WORD" &&
- tokens[i].getLexeme() === "Jugador"
-) {
- // Saltar tokens hasta encontrar el STRING que es el nombre del jugador
- i++;
- while (tokens[i] && tokens[i].getTypeTokenString() !== "STRING") i++;
- let name = tokens[i] ? tokens[i].getLexeme().replace(/"/g, "") : ""; // Extrae el nombre y quita las comillas
-
- // Saltar tokens hasta encontrar la llave de apertura del bloque del jugador
- i++;
- while (tokens[i] && tokens[i].getTypeTokenString() !== "CURLY_BRACKET_OPEN") i++;
- i++; // Avanza después de la llave de apertura

- let listPokemon: any[] = []; // Lista para almacenar los Pokémon de este jugador
-
- // Procesar todos los Pokémon dentro del bloque del jugador
- while (
- tokens[i] &&
- tokens[i].getTypeTokenString() === "STRING" // Un Pokémon empieza con una cadena de texto (su nombre)
-) {
- let pokemonName = tokens[i].getLexeme().replace(/"/g, ""); // Extrae el nombre del Pokémon
-
- // Saltar tokens hasta encontrar el tipo del Pokémon (palabra reservada entre corchetes)
- i++;
- while (tokens[i] && tokens[i].getTypeTokenString() !== "RESERVED_WORD") i++;
- let type = tokens[i] ? tokens[i].getLexeme() : ""; // Extrae el tipo del Pokémon
-
- // Saltar tokens hasta encontrar el paréntesis de apertura del bloque de estadísticas del Pokémon
- while (tokens[i] && tokens[i].getTypeTokenString() !== "PAR_OPEN") i++;
- i++; // Avanza después del paréntesis de apertura
- let health = 0, attack = 0, defense = 0; // Inicializa las estadísticas
-
- // Leer las estadísticas del Pokémon hasta cerrar el paréntesis
- while (
- tokens[i] &&

- tokens[i].getTypeTokenString() !== "PAR_CLOSE"
-) {
- if (
- tokens[i].getTypeTokenString() === "RESERVED_WORD"
-) {
- let stat = tokens[i].getLexema(); // Obtiene el nombre de la estadística (salud, ataque, defensa)
- // Saltar tokens hasta encontrar el número de la estadística
- while (tokens[i] && tokens[i].getTypeTokenString() !== "NUMBER") i++;
- let value = tokens[i] ? parseFloat(tokens[i].getLexema()) : 0; // Extrae el valor numérico
- if (stat === "salud") health = value;
- if (stat === "ataque") attack = value;
- if (stat === "defensa") defense = value;
- }
- i++; // Avanza al siguiente token de estadística
- }
- // Saltar el paréntesis de cierre de las estadísticas del Pokémon
- if (tokens[i] && tokens[i].getTypeTokenString() === "PAR_CLOSE") i++;
- listPokemon.push({ name: pokemonName, type, health, attack, defense }); // Añade el Pokémon a la lista
-
- // Saltar posibles separadores (como punto y coma) o saltos de línea hasta el siguiente STRING (otro Pokémon)
- // o la llave de cierre del bloque del jugador
- while (
- tokens[i] &&

```
                tokens[i].getTypeTokenString() !== "STRING" &&
                tokens[i].getTypeTokenString() !== "CURLY_BRACKET_CLOSE"
            ) {
                i++;
            }
        }
        players.push({ name, listPokemon }); // Añade el jugador completo a la lista de jugadores

        // Saltar la llave de cierre del jugador
        while (tokens[i] && tokens[i].getTypeTokenString() !== "CURLY_BRACKET_CLOSE") i++;
        if (tokens[i] && tokens[i].getTypeTokenString() === "CURLY_BRACKET_CLOSE") i++;
      } else {
        i++; // Si no es "Jugador", avanza al siguiente token
      }
    }
    return { players }; // Retorna la lista de jugadores
}
```

11. Conclusión General

La arquitectura de este proyecto, compuesta por el analizador léxico (LexicalAnalyzer.ts), la definición de tokens (Token.ts), los *scripts* de la interfaz de usuario (index.js, ui.js) y los componentes del *backend* (index.ts, analyze.controller.ts, PlayerParser.ts), conforma un sistema integral. Este sistema está diseñado para tomar un lenguaje de configuración de Pokémon como entrada, tokenizarlo, analizar su estructura y presentar los resultados de manera intuitiva. La sinergia entre estas partes asegura una experiencia de usuario fluida y una base robusta para futuras expansiones del lenguaje o funcionalidades del compilador.