Manual Técnico

CARLOS DÍAZ - 202400245

Menú

```
<div class="container">
    <div class="main-content">
        <aside class="sidebar">
            <div class="controls">
                <div class="file-upload" id="fileUpload">
                    <>
                    <h3>Cargar Archivo</h3>
                    <p>Haga clic o arrastre un archivo .txt</p>
                    <input type="file" id="fileInput" class="file-input" accept=".txt">
                </div>

                <button class="btn" id="btnAnalizar" disabled>Analizar Torneo</button>
                <button class="btn" id="btnReporte" disabled>Generar Reporte</button>
                <button class="btn" id="btnBracket" disabled>Mostrar Bracket</button>
                <button class="btn btn-danger" id="btnLimpiar">Limpiar Todo</button>
            </div>
             <div class="reportes-container" id="reportesContainer">
                    <h4>📊 Reportes Disponibles</h4>
                    <div class="reportes-grid">
                        <button class="btn-reporte" onclick="descargarReporteEquipos()">
                            Reporte de Equipos
                        </button>
```

El menú principal de la aplicación está construido en HTML y utiliza la librería Bootstrap, lo que permite darle un estilo más moderno y organizado a la interfaz.

¿Por qué se usa Bootstrap?

Bootstrap es un framework de diseño que proporciona componentes ya estilizados, como botones, barras de navegación y contenedores. Gracias a esto:

- Los botones del menú tienen un aspecto uniforme y atractivo.
- La disposición de los elementos es responsiva, es decir, se adapta automáticamente a distintos tamaños de pantalla (computadora, tablet o móvil).
- Se pueden aplicar fácilmente colores, espaciados y efectos visuales que mejoran la experiencia del usuario.

Cargar Archivo

```
async function analizarTorneo() {
    const codigo = codeEditor.value.trim();
    if (!codigo) {
        alert('Por favor, ingrese o cargue un código de torneo');
        return;
    }

    mostrarLoading(true);

    try {
        // Análisis léxico
        const lexer = new Lexer(codigo);
        const { tokens, errores } = lexer.analizar();

        // Mostrar tokens y errores
        mostrarTokens(tokens);
        mostrarErrores(errores);

        if (errores.length === 0) {
            // Análisis semántico
            const procesador = new ProcesadorTorneo(tokens);
            resultadosAnalisis = procesador.procesar();

            if (resultadosAnalisis.errores && resultadosAnalisis.errores.length > 0) {
                mostrarErroresSemanticos(resultadosAnalisis.errores);
            }
```

Este código implementa la lógica principal para analizar la entrada del torneo, que puede ser escrita en el área de texto o cargada desde un archivo. La función es asíncrona (async) porque involucra operaciones que pueden tomar cierto tiempo (como el análisis del código), y se maneja con try...catch para controlar errores.

Tokens y Errores

```
function mostrarErrores(errores) {
    const errorsTable = document.getElementById('errorsTable');

    if (errores.length === 0) {
        errorsTable.innerHTML = '<p>✅ No se encontraron errores léxicos</p>';
        return;
    }

    let html = `
        <div class="result-section">
            <h3>❌ Errores Léxicos Encontrados</h3>
            <p>Se encontraron <strong style="color: 🟥#e74c3c;">${errores.length}</strong> errores:</p>

            <div style="overflow-x: auto;">
                <table>
                    <thead>
                        <tr>
                            <th>#</th>
                            <th>Lexema</th>
                            <th>Tipo de Error</th>
                            <th>Descripción</th>
                            <th>Línea</th>
                            <th>Columna</th>
                        </tr>
                    </thead>
                    <tbody>
```

Funcionamiento del archivo de análisis léxico

1. Lectura del código de entrada
   - El archivo recibe el texto escrito en el área de código o cargado desde un archivo externo.
   - Este texto se analiza carácter por carácter y palabra por palabra, para identificar si cada elemento corresponde a un token válido o a un error.
2. Clasificación en listas
   - Durante la lectura, el archivo va almacenando los elementos reconocidos en listas separadas:
     - Lista de Tokens: contiene todos los símbolos válidos encontrados (ej. identificadores, nombres de equipos, separadores, palabras reservadas).
     - Lista de Errores: contiene aquellos fragmentos que no cumplen con las reglas del lenguaje definido (ej. caracteres desconocidos, estructuras mal escritas).
3. Disponibilidad para la interfaz
   - Una vez finalizado el análisis, estas listas quedan disponibles para ser recorridas y mostradas al usuario en la aplicación:
     - Los tokens se visualizan en la pestaña correspondiente del navbar.
     - Los errores se listan en la pestaña de errores, indicando claramente lo que debe corregirse.

Lexer

```
import { PALABRAS_RESERVADAS, ATRIBUTOS_VALIDOS, Token, TIPOS_SIMBOLOS, POSICIONES_VALIDAS,FASES_TORNEO } from "./Tokens.js";

export class Lexer {
    constructor(texto) {
        this.texto = texto;
        this.posicion = 0;
        this.linea = 1;
        this.columna = 1;
        this.tokens = [];
        this.errores = [];
        this.estado = "INICIO";
        this.buffer = "";
        this.inicioCol = 1;
        this.inicioLinea = 1;
    }

    analizar() {
        while (this.posicion < this.texto.length) {
            let char = this.texto[this.posicion];

            switch (this.estado) {
                case "INICIO":
                    this.procesarEstadoInicio(char);
                    break;
                case "IDENT":
                    this.procesarEstadoIdent(char);
                    break;
```

Este archivo define la clase encargada de leer el texto de entrada y convertirlo en tokens o errores, según las reglas establecidas.
Constructor constructor(texto)
Cuando se crea un objeto de la clase Lexer, se inicializan las variables necesarias para el análisis:

- this.texto → almacena el texto completo que se va a analizar.
- this.posicion → índice actual dentro del texto (sirve para recorrerlo carácter por carácter).
- this.linea y this.columna` → ayudan a llevar control de la ubicación exacta en el archivo, útil para reportar errores.
- this.tokens → lista donde se guardan los tokens reconocidos.
- this.errores → lista donde se almacenan los errores detectados.

Tokens

```
export class Token {
    constructor(lexema, tipo, linea, columna) {
        this.lexema = lexema;
        this.tipo = tipo;
        this.linea = linea;
        this.columna = columna;
    }
}

// Palabras reservadas actualizadas
export const PALABRAS_RESERVADAS = [
    "TORNEO", "EQUIPOS", "ELIMINACION", "equipo", "jugador",
    "partido", "resultado", "goleador", "vs", "goleadores"
];
export const FASES_TORNEO = [
    "cuartos", "semifinal", "final"
];

// Atributos válidos actualizados
export const ATRIBUTOS_VALIDOS = [
    "nombre", "sede", "equipos", "posicion", "numero",
    "edad", "minuto"
];

// Símbolos actualizados
export const TIPOS_SIMBOLOS = {
    "{": "Llave izquierda",
```

- Propósito: Representa una unidad léxica identificada en un texto de entrada (por ejemplo, un archivo de definición de torneo).
- Atributos:
- lexema: La cadena de caracteres que forma el token (ej. "TORNEO", "nombre").
- tipo: La categoría del token (puede ser palabra reservada, atributo, símbolo, etc.).
- linea: Número de línea en el archivo donde se encontró el token.
- columna: Posición dentro de la línea donde inicia el token.
- Uso: Cada vez que el analizador léxico identifica un token, crea una instancia de esta clase para guardar su información y luego procesarlo.

Logística extraída

```
procesarSeccionTorneo() {
    console.log("🎯 INICIANDO PROCESAMIENTO TORNEO - Índice inicial:", this.indice);
    this.avanzar(); // TORNEO
    this.avanzar(); // {
    console.log("Después de consumir TORNEO y { - Índice:", this.indice);

    // ✅ PROTECCIÓN CONTRA BUCLE INFINITO
    let contadorSeguridad = 0;
    const MAX_ITERACIONES = 500;

    while (this.indice < this.tokens.length && contadorSeguridad < MAX_ITERACIONES) {
        contadorSeguridad++;
        const token = this.tokens[this.indice];
        console.log("📄 Token actual:", token, "| Índice:", this.indice);

        // ✅ ACEPTAR AMBOS DELIMITADORES: } y )
        if (token.lexema === "}" || token.lexema === ")") {
            console.log("✅ ENCONTRÉ DELIMITADOR DE CIERRE:", token.lexema);
            this.avanzar(); // Consume } o )
            break;
        }

        if (token.tipo === "Atributo") {
            console.log("✅ ENCONTRÉ ATRIBUTO:", token.lexema);
            const atributo = token.lexema;
            this.avanzar(); // Consume el atributo
```

El archivo ProcesarTorneo tiene como objetivo recibir y validar la información de un torneo, tal como se define en los archivos que el usuario ingresa a la app. Se encarga de:

1. Validar cada sección del archivo: torneo, equipos, jugadores, partidos, resultados, fases, etc.
2. Almacenar toda la información relevante en estructuras de datos internas.
3. Preparar la información para generar estadísticas y reportes (goleadores, resultados, posiciones, etc.).

En otras palabras, es el núcleo lógico que organiza y procesa los datos del torneo.

```
procesar() {
    console.log("🔄 INICIANDO PROCESAMIENTO COMPLETO");

    try {
        while (this.indice < this.tokens.length) {
            const token = this.tokens[this.indice];
            console.log("🔍 Token en procesar():", token.lexema, "| Tipo:", token.tipo, "| Índice:", this.indice);

            if (token.lexema === "TORNEO" && token.tipo === "Palabra reservada") {
                console.log("🎯 Encontré TORNEO, llamando procesarSeccionTorneo()");
                this.procesarSeccionTorneo();
            } else if (token.lexema === "EQUIPOS" && token.tipo === "Palabra reservada") {
                console.log("👥 Encontré EQUIPOS, llamando procesarSeccionEquipos()");
                this.procesarSeccionEquipos();
            } else if (token.lexema === "ELIMINACION" && token.tipo === "Palabra reservada") {
                console.log("🏆 Encontré ELIMINACION, llamando procesarSeccionEliminacion()");
                this.procesarSeccionEliminacion();
            } else {
                console.log("➡️ Avanzando al siguiente token");
                this.indice++;
            }
```

Descargar Reporteria

```
function descargarReporteEquipos() {
    if (!resultadosAnalisis) {
        alert('Primero debe analizar un torneo válido');
        return;
    }
    const contenido = generarHTMLReporteEquipos();
    descargarHTML(contenido, 'reporte_equipos.html');
}

function descargarReporteGoleadores() {
    if (!resultadosAnalisis) {
        alert('Primero debe analizar un torneo válido');
        return;
    }
    const contenido = generarHTMLReporteGoleadores();
    descargarHTML(contenido, 'reporte_goleadores.html');
}

function descargarReporteGeneral() {
    if (!resultadosAnalisis) {
        alert('Primero debe analizar un torneo válido');
        return;
    }
    const contenido = generarHTMLReporteGeneral();
    descargarHTML(contenido, 'reporte_general.html');
}
function descargarReporteBracket() {
```

Este conjunto de funciones tiene como objetivo permitir al usuario descargar reportes HTML (y Graphviz) del torneo que fue analizado en la aplicación. Cada función valida que los datos existan y luego genera un archivo listo para descargar.

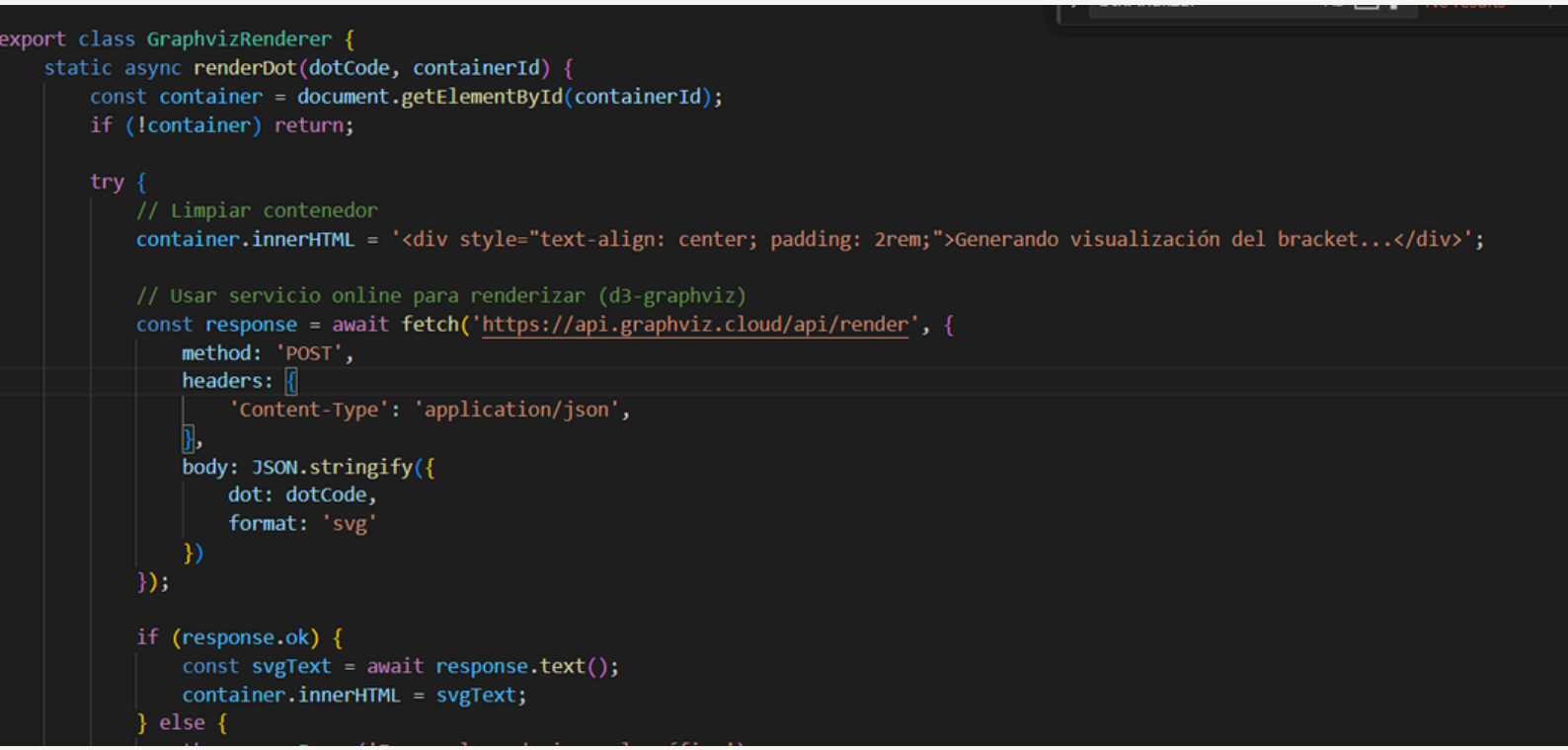
Flujo:
-Genera el contenido HTML del reporte llamando a generarHTMLReporteEquipos().
-Llama a descargarHTML(contenido, 'nombre_archivo') para iniciar la descarga.
-Resultado: Un archivo reporte_equipos.html con la información de los equipos.

- Muestra nombre de torneo, sede y cantidad de equipos.

1. Tabla de fases y partidos

- Recorre cada fase (torneo.fases) y luego cada partido dentro de la fase.
- Muestra:
  - Nombre de la fase.
  - Equipos que juegan (equipoLocal vs equipoVisitante).
  - Resultado del partido.
  - Ganador, con clase CSS para estilo (ganador o pendiente).

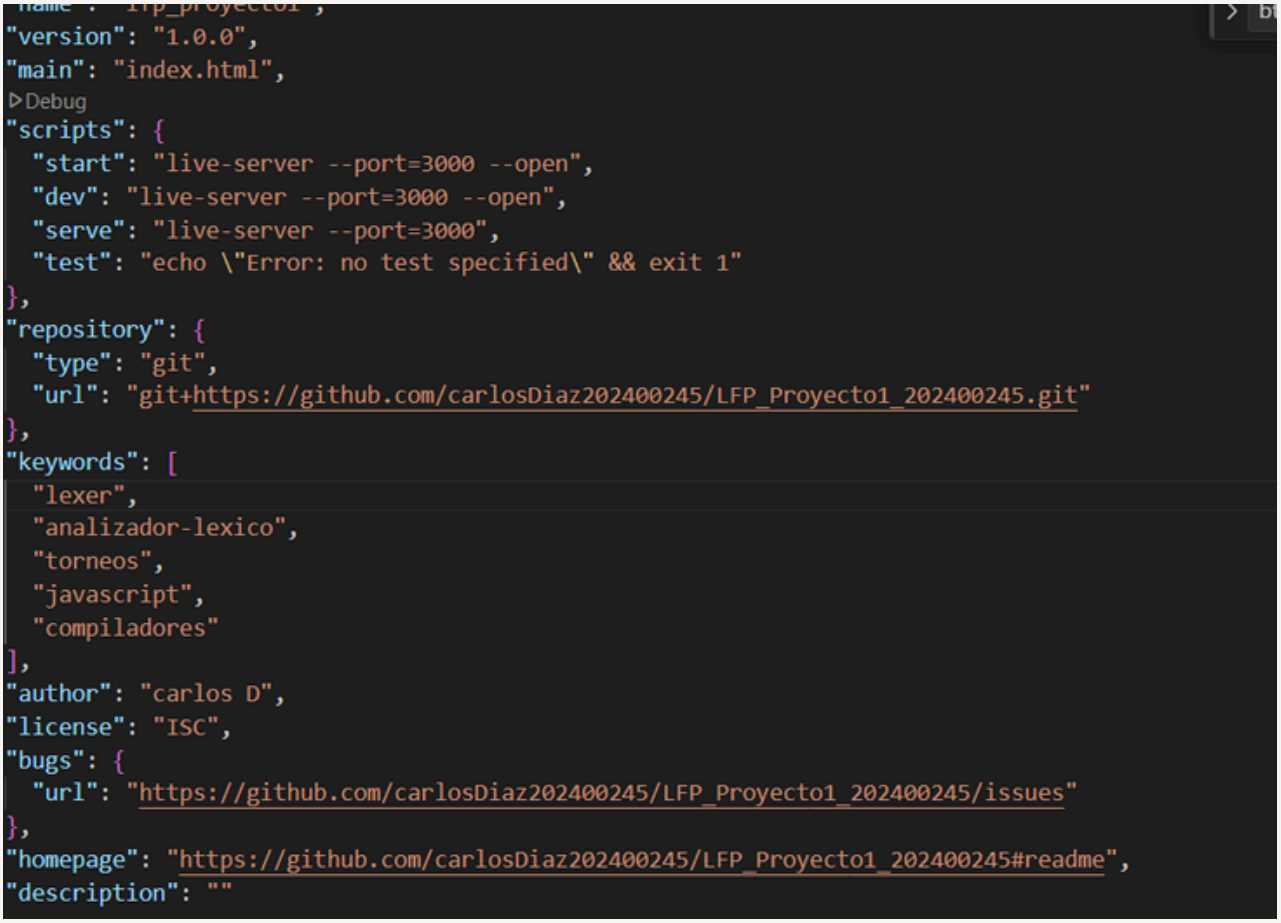
Generador Graphviz



```
export class GraphvizRenderer {
    static async renderDot(dotCode, containerId) {
        const container = document.getElementById(containerId);
        if (!container) return;

        try {
            // Limpiar contenedor
            container.innerHTML = '<div style="text-align: center; padding: 2rem;">Generando visualización del bracket...</div>';

            // Usar servicio online para renderizar (d3-graphviz)
            const response = await fetch('https://api.graphviz.cloud/api/render', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    dot: dotCode,
                    format: 'svg'
                })
            });

            if (response.ok) {
                const svgText = await response.text();
                container.innerHTML = svgText;
            } else {
```

- GraphvizRenderer es una clase estática que se encarga de generar visualizaciones de brackets de torneos a partir de código DOT (formato usado por Graphviz).
- Permite renderizar gráficos directamente en la página web.
- Ofrece mecanismos de recuperación (fallback) si la visualización falla, permitiendo descargar el código DOT y abrirlo en herramientas externas.
- Propósito: Renderiza un gráfico DOT en un contenedor HTML específico.
- Parámetros:
- dotCode: Código DOT que describe la estructura del bracket.
- containerId: ID del elemento HTML donde se insertará el gráfico.
- Flujo del método Propósito: Proporciona una alternativa visual cuando el renderizado directo falla o no está disponible.
- Flujo:
- Muestra un mensaje indicando que la visualización no está disponible.
- Ofrece enlaces a herramientas externas para abrir el código DOT:
- Edotor.net
- Graphviz.it
- Muestra el código DOT en un textarea para que el usuario pueda copiarlo.
- Incluye un botón para descargar el archivo DOT directamente desde la página.

Live Server



```
"version": "1.0.0",
"main": "index.html",
▷Debug
"scripts": {
  "start": "live-server --port=3000 --open",
  "dev": "live-server --port=3000 --open",
  "serve": "live-server --port=3000",
  "test": "echo \"Error: no test specified\" && exit 1"
},
"repository": {
  "type": "git",
  "url": "git+https://github.com/carlosDiaz202400245/LFP_Proyecto1_202400245.git"
},
"keywords": [
  "lexer",
  "analizador-lexico",
  "torneos",
  "javascript",
  "compiladores"
],
"author": "carlos D",
"license": "ISC",
"bugs": {
  "url": "https://github.com/carlosDiaz202400245/LFP_Proyecto1_202400245/issues"
},
"homepage": "https://github.com/carlosDiaz202400245/LFP_Proyecto1_202400245#readme",
"description": ""
```

- Live Server es una extensión para Visual Studio Code que permite ejecutar proyectos web directamente en el navegador.
- Proporciona un servidor local y actualización automática cada vez que se guardan cambios en los archivos HTML, CSS o JavaScript.

2. Por qué se usa en este proyecto

En tu proyecto de torneos:

1. Se requiere un entorno web: Los reportes se generan en HTML, se aplican estilos CSS y se ejecutan funciones JavaScript.
2. Live Server evita abrir archivos localmente con “file://”: Algunos métodos de JavaScript, como descargar archivos o usar módulos import/export, requieren que los archivos se sirvan desde un servidor web, no directamente desde el explorador de archivos.
3. Permite ver cambios en tiempo real: Cada vez que modificas un script o estilo, el navegador se actualiza automáticamente, facilitando el desarrollo y la prueba de reportes.

Diagrama de flujo

https://miro.com/app/board/uXjVJFKnXvE=/?share_link_id=228169535975

Diagrama UML

https://miro.com/app/board/uXjVJFKspDI=