MANUAL Técnico

Java Bridge: Traductor de Lenguajes Java a Python

Mario Rodrigo Balam

Correo electrónico: 2908263140708

Carné: 202200147

Curso: Lenguajes Formales

Sección: A-

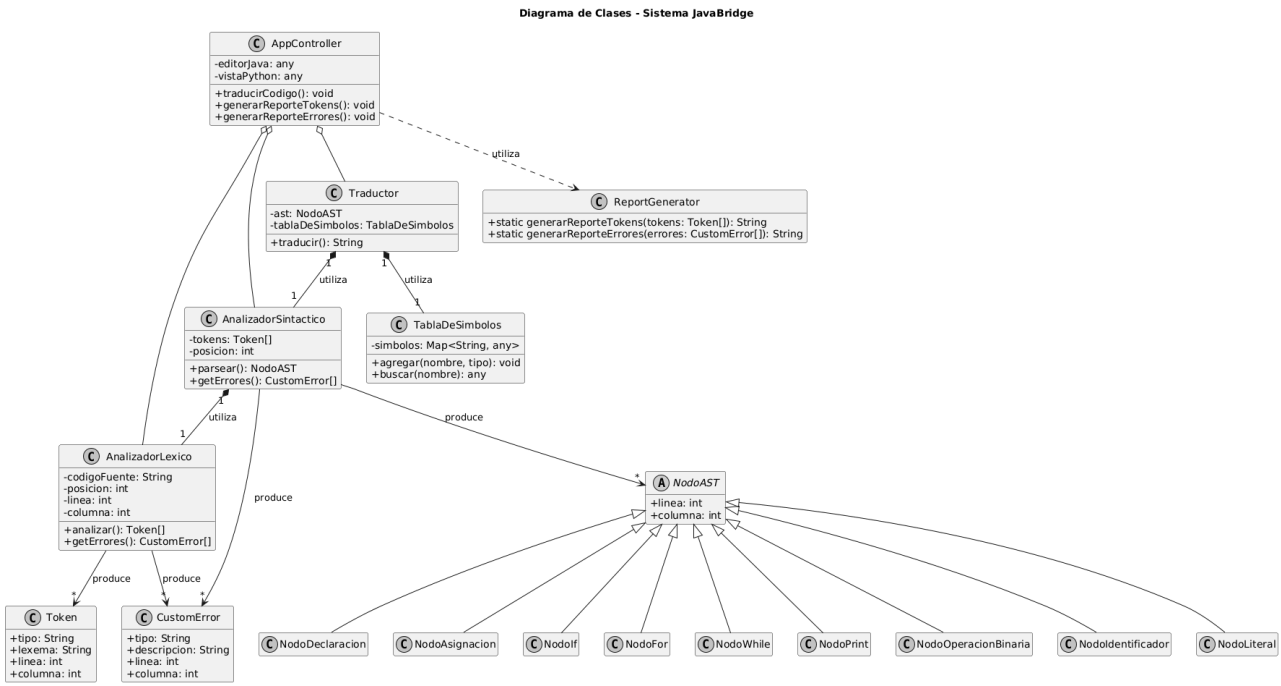
Octubre 2025

Ing Vivian Damaris Campos González

Aux: Carlos Javier Cox Bautista

TABLA DE CONTENIDO

Introduccion _____ 3
Objetivo GENERAL _____ 4
tECNOLOGIAS UTILIZADAS _____ 5
IV. Especificación técnicas _____ 5
V. Estructura del proyecto _____ 5
Arquitectura del sistema _____ 7
Descripción: _____ 7
Diagrama de Flujo de Datos: _____ 7
2. Analizador Léxico (scanner.js) _____ 9
Diseño del AFD _____ 9
Diagramad de clases. _____ 12



_____ 12
Arquitectura Orientada a objetos para el sistema JavaBridge. _____ 12

INTRODUCCION

1.1. Propósito del Documento

Explica que este manual describe en detalle la arquitectura, el diseño, la implementación y las pruebas del proyecto JavaBridge. Está dirigido a evaluadores, desarrolladores y cualquier persona interesada en comprender el funcionamiento interno del traductor.

1.2. Alcance del Proyecto

Resume el objetivo principal: traducir un subconjunto definido de Java a Python, realizando análisis léxico, sintáctico y semántico de forma manual. Menciona las características principales soportadas (tipos de datos, estructuras de control, etc.).

OBJETIVO GENERAL

El objetivo principal de TourneyJS es proporcionar una plataforma que:

1. Realice el análisis léxico de archivos que contengan definiciones de torneos deportivos
2. Identifique y clasifique tokens según una gramática definida
3. Detecte y reporte errores léxicos con precisión
4. Genere estructuras de datos organizadas a partir de la información procesada
5. Ofrezca reportes detallados y visualizaciones intuitivas de los datos analizados

TECNOLOGIAS UTILIZADAS

IV. Tecnologías y Entorno de Desarrollo

- Lenguaje de Programación: JavaScript (ECMAScript 6+).
- Entorno de Ejecución: Navegador web (para la interfaz) y Node.js
- Interfaz Gráfica: HTML5, CSS3.
- Control de Versiones: Git y GitHub.
- Herramientas de Diagramación: PlantUML / Mermaid.

V. Estructura del proyecto

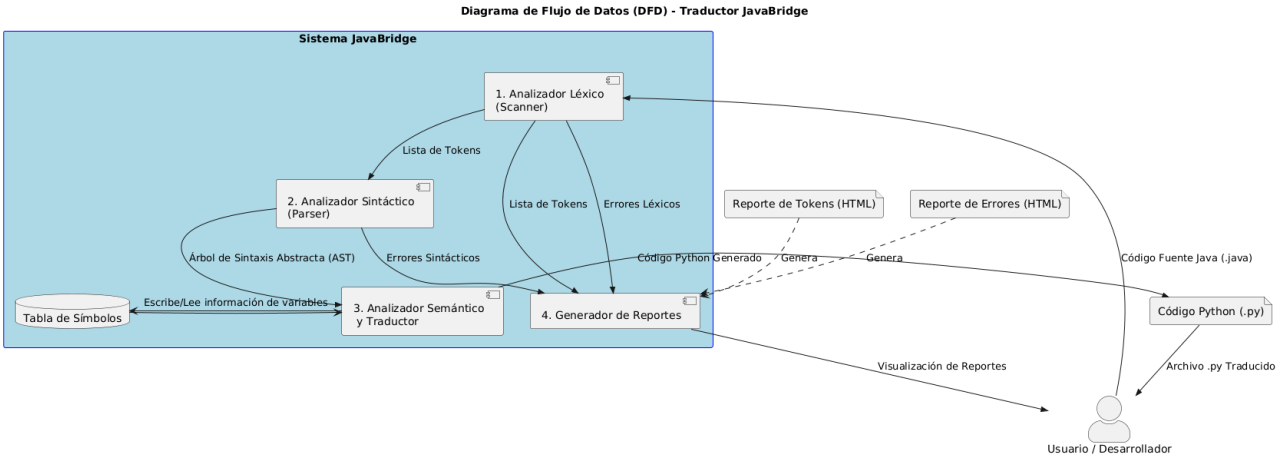
```
javabridgetranslator/
├── docs/
│   └── (Aquí van mi manual técnico, diagramas, etc.)
│
├── public/
│   ├── css/
│   │   └── style.css
│   ├── js/
│   │   └── app.js          // Lógica de la interfaz (botones, DOM)
│   └── index.html          // La interfaz gráfica
│
├── src/
│   ├── analyzer/
│   │   ├── Lexer.js        // Clase para el Analizador Léxico
│   │   ├── Parser.js       // Clase para el Analizador Sintáctico
│   │   └── Translator.js   // Clase para el Traductor
│   │
│   └── utils/
│       └── ReportGenerator.js // Para crear el HTML de los reportes
│
├── test-cases/
│   ├── caso1_valido.java
│   └── caso2_error_lexico.java
│
├── .gitignore
├── package.json
└── server.js               // Nuestro servidor web con Node.js y Express
```

ARQUITECTURA DEL SISTEMA

Descripción:

TourneyJS sigue una arquitectura modular en el lado del cliente (frontend), separando la lógica de análisis de la presentación. El flujo de datos es unidireccional: desde la entrada del usuario, a través de las capas de análisis, hasta la renderización de los resultados en la interfaz de usuario

Diagrama de Flujo de Datos:



Explicación:

Este diagrama ilustra el flujo de datos a través del sistema **JavaBridge**. Se divide en tres partes principales: entidades externas (el usuario), procesos internos del sistema y almacenes de datos (archivos de salida).

1. Entidades

- **Usuario / Desarrollador:** Es la entidad externa que interactúa con el sistema. Proporciona el archivo .java de entrada y recibe como salida el archivo .py traducido y los reportes HTML.

2. Procesos Internos (Componentes del Sistema JavaBridge)

1. **Analizador Léxico (Scanner):**
    - **Entrada:** Recibe el Código Fuente Java como una cadena de texto.
    - **Función:** Lee el código fuente carácter por carácter y lo agrupa en componentes léxicos siguiendo las reglas del AFD.
    - **Salidas:**
        - Una Lista de Tokens si el proceso es exitoso.
        - Una lista de Errores Léxicos si encuentra caracteres o secuencias no válidas.

2. **Analizador Sintáctico (Parser):**
    - **Entrada:** Recibe la Lista de Tokens del analizador léxico.
    - **Función:** Verifica si la secuencia de tokens cumple con las reglas de la Gramática Libre de Contexto (GLC) del lenguaje. Si la estructura es correcta, construye una representación jerárquica del código, conocida como Árbol de Sintaxis Abstracta (AST).
    - **Salidas:**
        - Un Árbol de Sintaxis Abstracta (AST) que representa la estructura del programa.
        - Una lista de Errores Sintácticos si la secuencia de tokens es inválida (ej: falta un punto y coma).
3. **Analizador Semántico y Traductor:**
    - **Entrada:** Recibe el Árbol de Sintaxis Abstracta (AST).
    - **Función:** Recorre el AST para realizar validaciones semánticas (ej: que una variable sea declarada antes de usarse). Para esto, utiliza una Tabla de Símbolos para registrar y consultar información de las variables. Si todas las validaciones son correctas, traduce cada nodo del árbol a su equivalente en Python.
    - **Salidas:**
        - El Código Python Generado como una cadena de texto.
4. **Generador de Reportes:**
    - **Entradas:** Recibe la Lista de Tokens, los Errores Léxicos y los Errores Sintácticos de los procesos de análisis.
    - **Función:** Formatea los datos recibidos y genera archivos HTML legibles para el usuario.
    - **Salidas:**
        - Reporte de Tokens (HTML).
        - Reporte de Errores (HTML).

3. Almacenes y Archivos de Salida

- **Tabla de Símbolos:** Es una estructura de datos interna y temporal, crucial para el análisis semántico. Almacena información sobre los identificadores (variables), como su tipo y alcance.
- **Código Python (.py):** El archivo final con el código traducido, generado únicamente si no se encontraron errores en ninguna de las fases de análisis.
- **Reportes (HTML):** Archivos generados para que el usuario pueda inspeccionar los tokens reconocidos y los errores detectados durante la compilación.

2. Analizador Léxico (scanner.js)

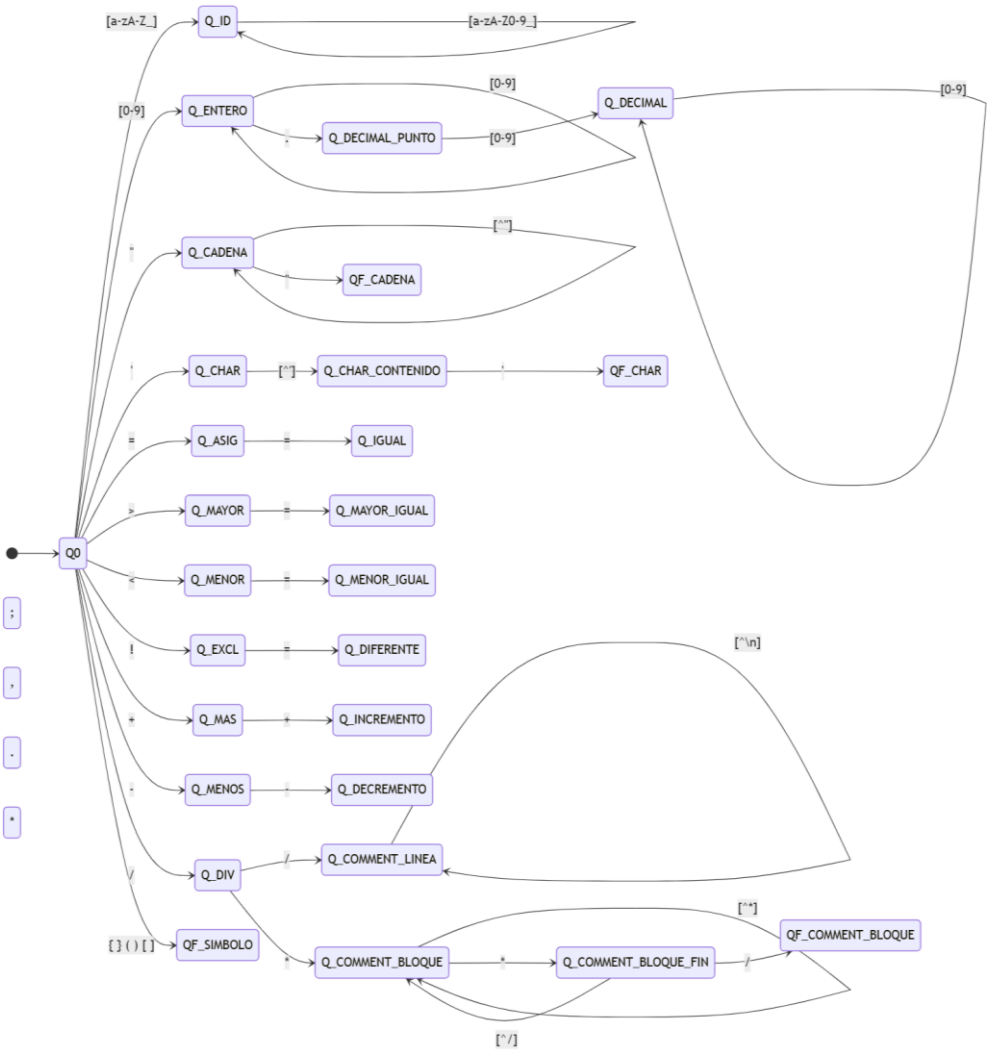
El analizador léxico, o scanner, es responsable de leer el archivo de entrada como una secuencia de caracteres y convertirla en una secuencia de tokens. Se implementó utilizando un Autómata Finito Determinista (AFD) simulado a través de una máquina de estados explícita en JavaScript.

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

class AnalizadorLexicoTorneos {
  constructor() {
    this.listaTokens = []; this.listaError = [];
    this.palabrasEstructurales = {
      'torneo': "PALABRA_RESERVADA", 'equipos': "PALABRA_RESERVADA", 'eliminacion': "PALABRA_RESERVADA",
      'equipo': "PALABRA_RESERVADA", 'jugador': "PALABRA_RESERVADA", 'partido': "PALABRA_RESERVADA",
      'goleador': "PALABRA_RESERVADA", 'goleadores': "PALABRA_RESERVADA", 'vs': "PALABRA_RESERVADA",
      'semifinal': "PALABRA_RESERVADA", 'final': "PALABRA_RESERVADA", 'cuartos': "PALABRA_RESERVADA"
    };
    this.palabrasDePropiedad = {
        'nombre': 'NOMBRE', 'sede': 'SEDE', 'posicion': 'POSICION',
        'numero': 'NUMERO_PROP', 'edad': 'EDAD', 'resultado': 'RESULTADO', 'minuto': 'MINUTO'
    };
  }
```

Diseño del AFD

Este diagrama representa la lógica central del AFD. Muestra los estados principales y las transiciones basadas en el tipo de carácter leído.



Lógica de Implementación Detallada (El Proceso que Muestra el Diagrama)

Tomemos como analogía que el analizador procesa la cadena: int var1;

1. **Inicio**: El puntero de lectura está en la 'i'. El estado actual es Q0. El buffer del lexema está vacío.
2. **Procesando int:**
    - Lee 'i': Es una letra. Transición Q0 -> Q_ID. Buffer: "i".
    - Lee 'n': Es una letra/dígito. Permanece en Q_ID. Buffer: "in".
    - Lee 't': Es una letra/dígito. Permanece en Q_ID. Buffer: "int".
    - Lee (espacio): Este es el lookahead no válido. El espacio no es parte de un identificador.
    - Acción: El lexema ha terminado.
        - Emitir Token: El estado actual es Q_ID y el buffer es "int". Se verifica si "int" es una palabra reservada. Lo es. Se crea el token

{tipo: "PALABRA_RESERVADA_INT", lexema: "int", ... } y se añade a la lista.

- Reiniciar: El espacio no fue consumido, sigue siendo el carácter actual. El estado vuelve a Q0. El buffer se vacía.

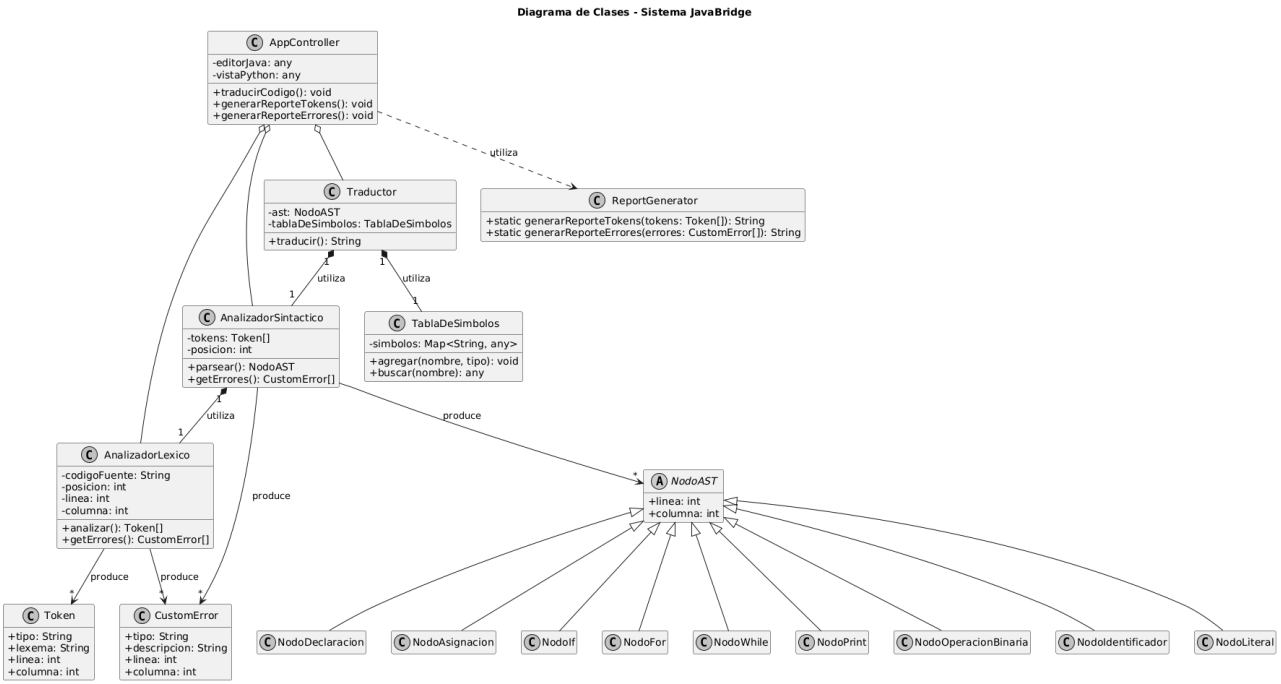
3. **Procesando el Espacio:**
   - El puntero sigue en el (espacio). El estado es Q0.
   - Acción: Es un espacio en blanco. Se ignora. Se avanza el puntero. El estado sigue siendo Q0.
4. **Procesando var1:**
   - Lee 'v': Letra. Q0 -> Q_ID. Buffer: "v".
   - Lee 'a': Letra/dígito. Permanece en Q_ID. Buffer: "va".
   - Lee 'r': Letra/dígito. Permanece en Q_ID. Buffer: "var".
   - Lee '1': Letra/dígito. Permanece en Q_ID. Buffer: "var1".
   - Lee ;: Lookahead no válido. El punto y coma no es parte de un identificador.
   - Acción: **El lexema ha terminado**.
   - Emitir Token: El estado es Q_ID y el buffer es "var1". No es una palabra reservada. Se crea el token { tipo: "IDENTIFICADOR", lexema: "var1", ... }.
   - Reiniciar: El ; no se consumió. El estado vuelve a Q0. El buffer se vacía.
5. **Procesando ;:**
   - Lee ;: Es un símbolo. Q0 -> QF_PUNTO_COMA.
   - Acción: Este es un token de un solo carácter. No se necesita lookahead.
   - Emitir Token: Se crea el token {tipo: "PUNTO_Y_COMA", lexema: ";", ...}.
   - Reiniciar: Se avanza el puntero. El estado vuelve a Q0.

Este ciclo se repite hasta que se consumen todos los caracteres del archivo de entrada. La clave es siempre "avanzar hasta que el patrón se rompa, y luego emitir el token sin consumir el carácter que rompió el patrón".

3. Analizador Sintáctico (structures.js)

Descripción: El analizador sintáctico, o parser, recibe la secuencia de tokens del lexer y verifica si esta sigue las reglas gramaticales del lenguaje de torneos. Se implementó utilizando la técnica de Descenso Recursivo, donde cada regla gramatical principal (ej. bloque_torneo, bloque_equipos) corresponde a una función en el parser.

Diagramad de clases.



Arquitectura Orientada a objetos para el sistema JavaBridge.

Este diagrama presenta una arquitectura orientada a objetos para el sistema **JavaBridge**. La estructura se basa en el principio de separación de responsabilidades.

1. **Clases de Datos**
    - **Token:** Representa una unidad léxica individual. Es una estructura de datos simple que contiene el tipo de token (ej: IDENTIFICADOR), el lexema (miVariable), y su ubicación en el código fuente.}
    - **CustomError:** Representa un error léxico o sintáctico, almacenando su descripción y ubicación para facilitar la generación de reportes.
    - **NodoAST (y sus subclases):** Representan el Árbol de Sintaxis Abstracta. Cada tipo de construcción del lenguaje (una declaración, una sentencia if, una operación matemática) tiene su propia clase de nodo. Esto permite un procesamiento estructurado en la fase de traducción utilizando patrones como el Visitor.
2. **Componentes Principales**
    - **AnalizadorLexico:** Su única responsabilidad es convertir el código fuente en una lista de Tokens y una lista de CustomError léxicos. Encapsula toda la lógica del AFD.
    - **AnalizadorSintactico:** Recibe la lista de tokens. Su responsabilidad es validar la estructura gramatical y, si es correcta, construir el NodoAST raíz. También genera errores sintácticos si la gramática no se cumple.

- **TablaDeSimbolos:** Una estructura de datos utilizada por el traductor para llevar un registro de las variables que han sido declaradas, su tipo y otros metadatos relevantes para el análisis semántico.
- **Traductor:** Su responsabilidad es recorrer el AST proporcionado por el parser, realizar validaciones semánticas con la ayuda de la TablaDeSimbolos, y generar la cadena de texto final con el código Python.

3. Clases de Utilidad y Control

- **ReportGenerator:** Una clase de utilidad (static) que se encarga exclusivamente de tomar listas de tokens o errores y convertirlas en formato HTML.
- **AppController:** Actúa como el orquestador de todo el proceso. Esta clase estaría vinculada a la interfaz de usuario. Cuando el usuario presiona el botón "Traducir", una instancia de AppController crea los analizadores, invoca sus métodos en la secuencia correcta (analizar ()-> parsear () -> traducir ()) y utiliza el ReportGenerator para mostrar los resultados.

Métodos Adicionales

- **Motor de Lógica (engine.js):**
  - **Descripción:** La clase TournamentEngine desacopla la lógica de negocio del análisis sintáctico. Recibe el árbol de sintaxis abstracto (el objeto estructuras) y lo procesa para generar datos estadísticos.

```
/**
 * Procesa todos los partidos para calcular las estadísticas de cada equipo.
 */
calcularEstadisticas() {
    const equiposStats = {};

    // 1. Inicializar estadísticas para cada equipo
    this.estructuras.equipos.forEach(equipo => {
        equiposStats[equipo.name] = {
            PJ: 0, G: 0, P: 0, GF: 0, GC: 0, DG: 0, faseAlcanzada: "No jugó"
        };
    });

    // 2. Iterar sobre cada fase y cada partido para actualizar las estadísticas
    const fases = this.estructuras.eliminacion || {};
    for (const nombreFase in fases) {
        const partidos = fases[nombreFase];
        partidos.forEach(partido => {
            const { equipo1, equipo2, resultado } = partido;
            const goles = resultado.split('-').map(g => parseInt(g));
            const goles1 = goles[0];
            const goles2 = goles[1];

            if (equiposStats[equipo1] && equiposStats[equipo2]) {
                // Actualizar partidos jugados
                equiposStats[equipo1].PJ++;
                equiposStats[equipo2].PJ++;

                // Actualizar goles
                equiposStats[equipo1].GF += goles1;
                equiposStats[equipo1].GC += goles2;
                equiposStats[equipo2].GF += goles2;
                equiposStats[equipo2].GC += goles1;
```

  - **Algoritmos:**

- calcularEstadisticas(): Itera sobre todos los partidos en la estructura de eliminacion. Para cada partido, actualiza un objeto de estadísticas para los dos equipos involucrados, incrementando partidos jugados (PJ), victorias (G), derrotas (P), goles a favor (GF) y goles en contra (GC).

```
calcularEstadisticas() {
    const equiposStats = {};

    // 1. Inicializar estadísticas para cada equipo
    this.estructuras.equipos.forEach(equipo => {
        equiposStats[equipo.name] = {
            PJ: 0, G: 0, P: 0, GF: 0, GC: 0, DG: 0, faseAlcanzada: "No jugó"
        };
    });

    // 2. Iterar sobre cada fase y cada partido para actualizar las estadísticas
    const fases = this.estructuras.eliminacion || {};
    for (const nombreFase in fases) {
        const partidos = fases[nombreFase];
        partidos.forEach(partido => {
            const { equipo1, equipo2, resultado } = partido;
            const goles = resultado.split('-').map(g => parseInt(g));
            const goles1 = goles[0];
            const goles2 = goles[1];

            if (equiposStats[equipo1] && equiposStats[equipo2]) {
                // Actualizar partidos jugados
                equiposStats[equipo1].PJ++;
                equiposStats[equipo2].PJ++;

                // Actualizar goles
                equiposStats[equipo1].GF += goles1;
                equiposStats[equipo1].GC += goles2;
                equiposStats[equipo2].GF += goles2;
                equiposStats[equipo2].GC += goles1;

                // Actualizar victorias y derrotas
                const ganador = goles1 > goles2 ? equipo1 : equipo2;
                const perdedor = goles1 > goles2 ? equipo2 : equipo1;

                equiposStats[ganador].G++;
                equiposStats[perdedor].P++;
```

- getGoleadores(): Recorre todos los partidos y todos los goleadores, creando una lista plana. Luego, utiliza reduce() para agrupar los goles por nombre de jugador y finalmente ordena la lista resultante.

```
getGoleadores() {
    const listaGoles = [];
    const fases = this.estructuras.eliminacion || {};

    for (const nombreFase in fases) {
        fases[nombreFase].forEach(partido => {
            const equipoDelGoleador = (nombreGoleador) => {
                const equipo1Info = this.estructuras.equipos.find(e => e.name === partido.equipo1);
                if (equipo1Info && equipo1Info.jugadores.some(j => j.nombre === nombreGoleador)) {
                    return partido.equipo1;
                }
                return partido.equipo2;
            };

            partido.goleadores.forEach(gol => {
                listaGoles.push({
                    nombre: gol.nombre,
                    minuto: gol.minuto,
                    equipo: equipoDelGoleador(gol.nombre)
                });
            });
        });
    }
```

- **Integración con Graphviz (graphviz.js):**

- **Descripción:** La función generarCodigoGraphviz traduce el objeto estructuras.eliminacion a un string en lenguaje DOT.

- **Algoritmo:** Itera sobre cada fase y cada partido, generando un subgraph de Graphviz para cada fase y un node de tipo record para cada partido. Almacena los ganadores de cada partido en un mapa. Luego, en una segunda pasada, conecta los nodos de una fase con los de la siguiente basándose en los ganadores almacenados. La librería Viz.js es luego utilizada en el frontend para renderizar este string DOT en una imagen SVG.

```
function generarCodigoGraphviz(estructuras) {
    if (!estructuras || !estructuras.eliminacion) {
        return `digraph G { label="No hay datos de eliminación para mostrar"; }`;
    }

    let dot = `
digraph TournamentBracket {
    rankdir="LR"; // Dibuja el grafo de izquierda a derecha
    node [shape=record, style=filled, fillcolor="#f8f9fa", color="#adb5bd"];
    edge [color="#495057"];
    graph [bgcolor="transparent", fontname="Arial"];

    // Título del Torneo
    labelloc="t";
    label="${estructuras.torneo.name || 'Torneo sin nombre'}\\n${estructuras.torneo.sede || ''}";
    fontsize=20;
    fontcolor="#343a40";

`;

    const fases = estructuras.eliminacion;
    const nombresFases = Object.keys(fases); // ["semifinal", "final"]
    const ganadores = {}; // Para conectar las fases

    // Crear los nodos de los partidos para cada fase
    nombresFases.forEach(nombreFase => {
        dot += `    subgraph cluster_${nombreFase} {\n`;
        dot += `        label = "${nombreFase.charAt(0).toUpperCase() + nombreFase.slice(1)}";\n`;
        dot += `        color="#dee2e6";\n        style="rounded";\n`;

        fases[nombreFase].forEach((partido, index) => {
            const idPartido = `${nombreFase}_${index}`;
            const goles = partido.resultado.split('-').map(g => parseInt(g));
            const ganadorPartido = goles[0] > goles[1] ? partido.equipo1 : partido.equipo2;
            ganadores[idPartido] = ganadorPartido; // Guardamos el ganador
```

CONCLUSIONES

Se desarrolló completamente un analizador léxico funcional capaz de:

- Identificar y clasificar tokens según la gramática definida
- Detectar errores léxicos con precisión
- Procesar archivos de entrada con definiciones de torneos
- Generar estructuras de datos organizadas a partir del código fuente

El sistema implementa un mecanismo efectivo para:

- Detectar caracteres no válidos
- Identificar cadenas sin cerrar
- Reportar la ubicación exacta de errores (línea y columna)
- Permitir la recuperación y continuación del análisis