MANUAL TECNICO TOURNEY JS

JEFRY ALEXIS SEMEYÁ LOPEZ

202003562

LENGUAJES FORMALES Y DE PROGRAMACIÓN

SECCION: B+

INTRODUCCIÓN

La aplicación fue construida bajo un enfoque académico y práctico, implementando de manera manual un **analizador léxico y sintáctico** basado en un Autómata Finito Determinista (AFD) y un parser ligero, sin recurrir a herramientas automáticas como ANTLR, JFlex, Lex u otras. Esto garantiza un entendimiento profundo del procesamiento del lenguaje definido para la descripción de torneos, así como un mayor control sobre cada etapa del análisis.

Entre sus principales características técnicas destacan:

- Implementación manual de un **scanner** (analizador léxico) capaz de identificar tokens válidos y reportar errores léxicos.
- Construcción de un **parser ligero**, encargado de transformar la secuencia de tokens en una estructura de datos que modela el torneo.
- Generación de reportes en formato HTML, que incluyen:
    - Bracket de eliminación con visualización del progreso de los equipos.
    - Estadísticas detalladas por equipo.
    - Información general del torneo.
    - Tokens reconocidos y errores encontrados.
- Generación de representaciones gráficas en **Graphviz (DOT)** para la visualización de llaves de eliminación.
- Interfaz web desarrollada con **HTML, CSS y JavaScript**, con un diseño moderno, profesional e intuitivo.

FUNCION (ESCAPEHTML)

```
function escapeHtml(s){
  if (s == null || s == undefined) return "";
  return String(s)
  .replaceAll("&", "&amp;")
  .replaceAll("<", "&lt;")
  .replaceAll(">", "&gt;")
  .replaceAll('"', "&quot;");
}
```

PROPÓSITO: Escapa caracteres especiales HTML para prevenir inyección de código

USO: Seguridad al mostrar contenido dinámico en HTML

CONST TOKENTYPE

```
const TokenType ={
  RESERVED: "RESERVED",        // Palabras reservadas del lenguaje
  IDENT: "IDENT",             // Identificadores de usuario
  STRING:"STRING",            // Cadenas entre comillas
  NUMBER:"NUMBER",            // Números enteros
  VS: "VS",                   // Separador especial "vs"
  LBRACE: "LBRACE",          // Llave izquierda {
  RBRACE: "RBRACE",          // Llave derecha }
  LBRACKET: "LBRACKET",      // Corchete izquierdo [
  RBRACKET: "RBRACKET",      // Corchete derecho ]
  COLON: "COLON",            // Dos puntos :
  COMMA: "COMMA",            // Coma ,
  SEMICOLON: "SEMICOLON",    // Punto y coma ;
};
```

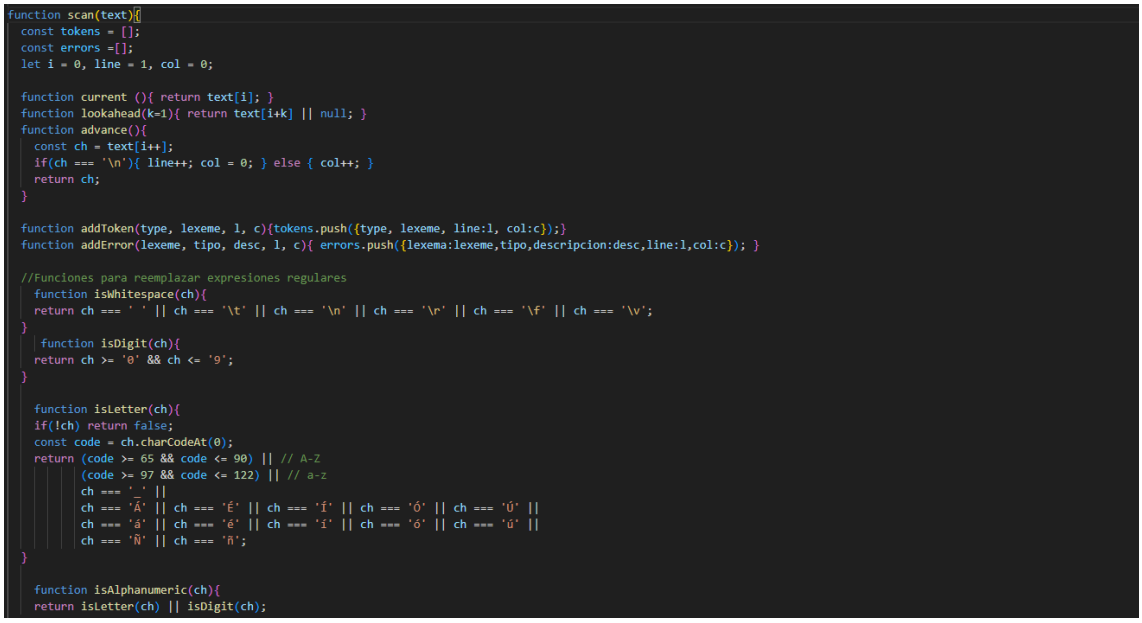
Esta sección define la estructura de tokens que el analizador léxico puede reconocer y el conjunto de palabras reservadas del lenguaje.

CONST RESERVED_SET

```
const RESERVED_SET = new Set([
  "torneo", "equipos", "eliminacion",
  "equipo", "jugador", "partido", "resultado", "goleadores",
  "cuartos", "semifinal", "final", "nombre", "posicion", "numero",
  "edad", "vs", "goleador", "minuto", "sede"

]);
```

Define todas las palabras reservadas del lenguaje de torneos

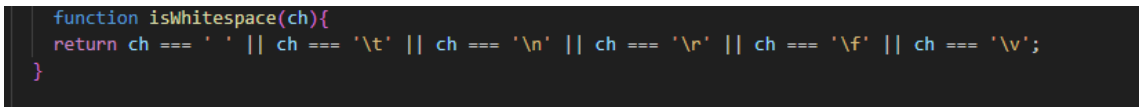
FUNCION (FUCTION SCAN)



```
function scan(text){
  const tokens = [];
  const errors =[];
  let i = 0, line = 1, col = 0;

  function current (){ return text[i]; }
  function lookahead(k=1){ return text[i+k] || null; }
  function advance(){
    const ch = text[i++];
    if(ch === '\n'){ line++; col = 0; } else { col++; }
    return ch;
  }

  function addToken(type, lexeme, l, c){tokens.push({type, lexeme, line:l, col:c});}
  function addError(lexeme, tipo, desc, l, c){ errors.push({lexema:lexeme,tipo,descripcion:desc,line:l,col:c}); }

  //Funciones para reemplazar expresiones regulares
    function isWhitespace(ch){
    return ch === ' ' || ch === '\t' || ch === '\n' || ch === '\r' || ch === '\f' || ch === '\v';
  }
     function isDigit(ch){
    return ch >= '0' && ch <= '9';
  }

    function isLetter(ch){
    if(!ch) return false;
    const code = ch.charCodeAt(0);
    return (code >= 65 && code <= 90) || // A-Z
           (code >= 97 && code <= 122) || // a-z
           ch === '_' ||
           ch === 'Á' || ch === 'É' || ch === 'Í' || ch === 'Ó' || ch === 'Ú' ||
           ch === 'á' || ch === 'é' || ch === 'í' || ch === 'ó' || ch === 'ú' ||
           ch === 'Ñ' || ch === 'ñ';
  }

    function isAlphanumeric(ch){
    return isLetter(ch) || isDigit(ch);
```

Analiza léxicamente el texto de entrada y genera tokens, dentro de esta fuctionscan se implementaron funciones para reemplazar las expresiones regulares.

FUNCION (ISWHITSPACE)



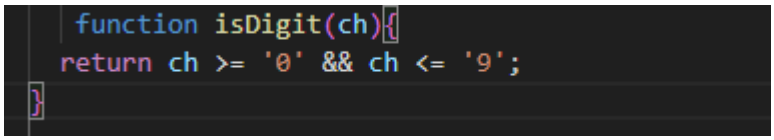
```
    function isWhitespace(ch){
    return ch === ' ' || ch === '\t' || ch === '\n' || ch === '\r' || ch === '\f' || ch === '\v';
  }
```

PROPÓSITO: Determina si un carácter es espacio en blanco

PARÁMETROS: ch (char) - Carácter a evaluar

RETORNA: boolean - true si es espacio en blanco

FUNCION (ISDIGIT)



```
    function isDigit(ch){
    return ch >= '0' && ch <= '9';
  }
```

PROPÓSITO: Determina si un carácter es un dígito (0-9)

PARÁMETROS: ch (char) - Carácter a evaluar

RETORNA: boolean - true si es dígito

FUNCION (ISLETTER)

```
function isLetter(ch){
if(!ch) return false;
const code = ch.charCodeAt(0);
return (code >= 65 && code <= 90) || // A-Z
       (code >= 97 && code <= 122) || // a-z
       ch === '_' ||
       ch === 'Á' || ch === 'É' || ch === 'Í' || ch === 'Ó' || ch === 'Ú' ||
       ch === 'á' || ch === 'é' || ch === 'í' || ch === 'ó' || ch === 'ú' ||
       ch === 'Ñ' || ch === 'ñ';
}
```

PROPÓSITO: Determina si un carácter es una letra (incluye acentos y ñ)

PARÁMETROS: ch (char) - Carácter a evaluar

RETORNA: boolean - true si es letra válida para identificadores

FUNCION (ISALPHANUMERIC)

```
function isAlphanumeric(ch){
  return isLetter(ch) || isDigit(ch);
}
```

PROPÓSITO: Determina si un carácter es alfanumérico (letra o dígito)

PARÁMETROS: ch (char) - Carácter a evaluar

RETORNA: boolean - true si es letra o dígito

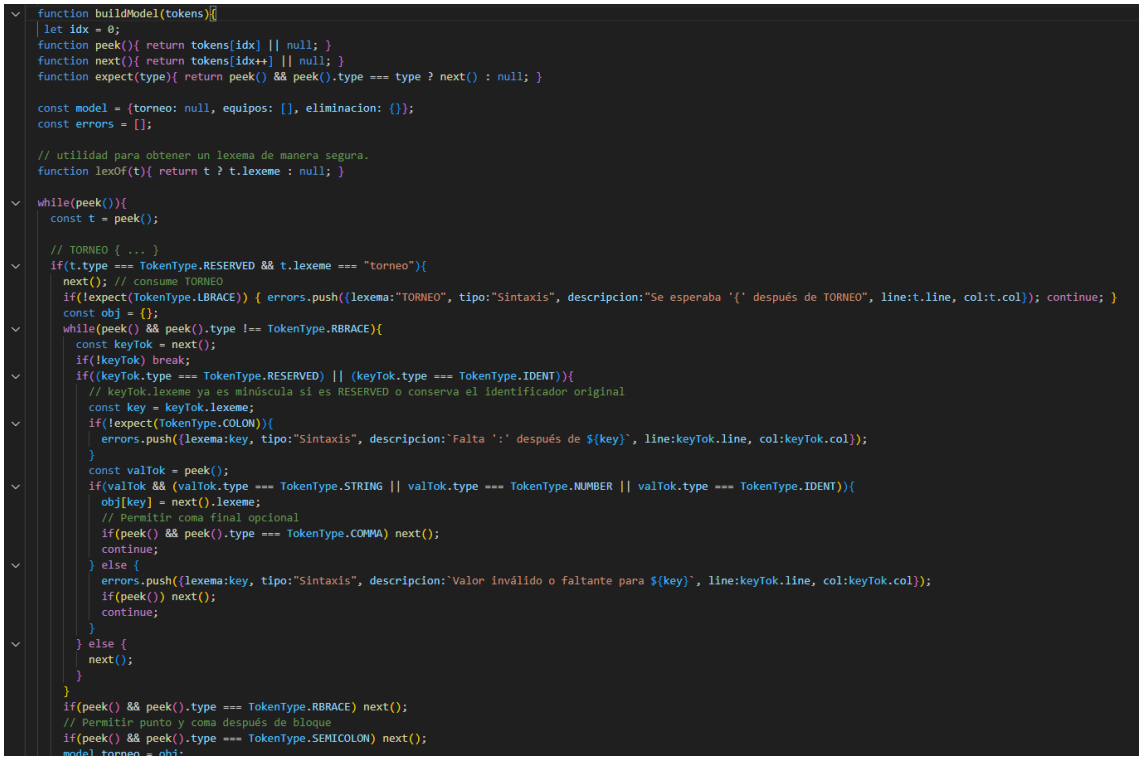
USO: Para continuar leyendo identificadores después del primer carácter

BUCLE PRINCIPAL

```
while(i < text.length){
  let ch = current();
  const startLine = line, startCol = col + 1;

  if(isWhitespace(ch)){ advance(); continue; }

  if(ch === '/' && lookahead() === '/'){
    // consumir hasta nueva linea
    advance(); advance();
    while(i < text.length && current() !== '\n') advance();
    continue;
  }
  if(ch === '/' && lookahead() === '*'){
    advance(); advance();
    let closed = false;
    while(i < text.length){
      if(current() === '*' && lookahead() === '/'){ advance(); advance(); closed = true; break; }
      advance();
    }
    (local function) addError(lexeme: any, tipo: any, desc: any, l: any, c: any): void
    if(!closed) addError("/*", "Comentario no cerrado", "Comentario de bloque sin cerrar '*/'", startLine, startCol);
    continue;
  }

  // string "..."
  if(ch === '"'){
    advance(); // consumir "
    let lex = "";
    let closed = false;
    while(i < text.length){
      const c = current();
      if(c === '"'){ advance(); closed = true; break; }
      if(c === '\n'){ addError(lex, "Cadena no cerrada", "Cadena sin comillas de cierre en la misma línea", startLine, startCol); break; }
      lex += advance();
    }
    if(closed) addToken(TokenType.STRING, lex, startLine, startCol);
    continue;
  }
```

Este bucle implementa el AFD procesando carácter por carácter hasta completar todo el texto de entrada

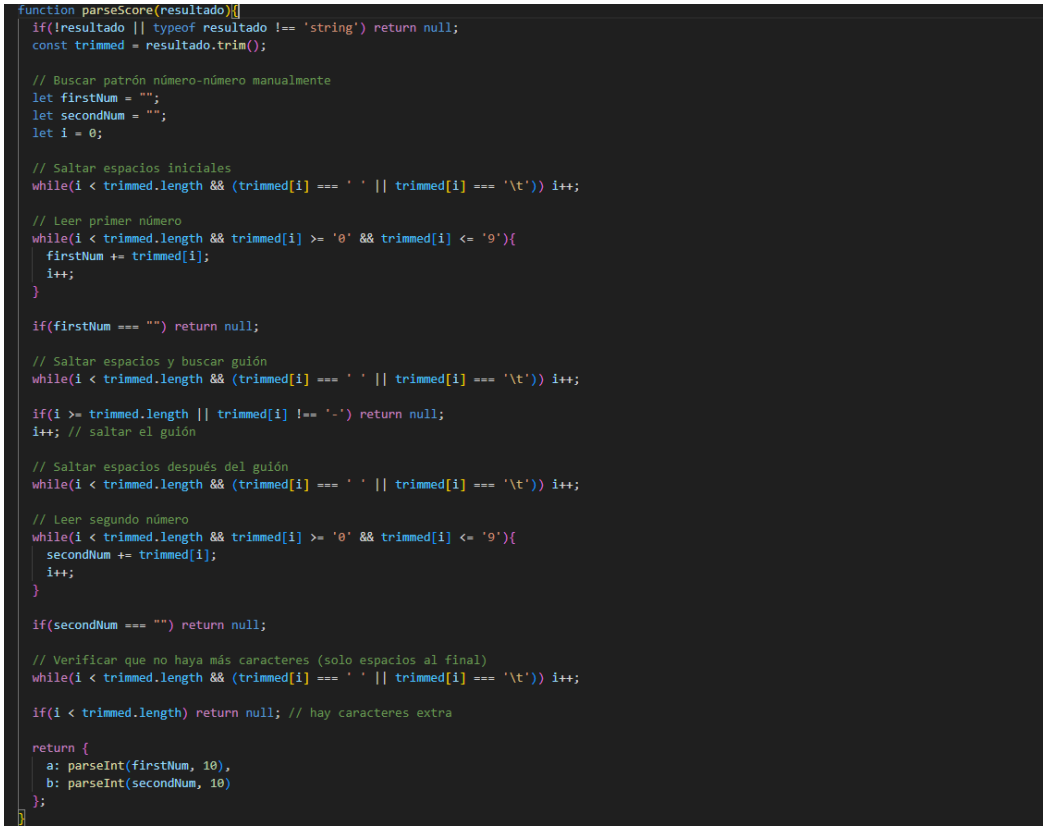
FUNCION (BUILDMODEL)



```
function buildModel(tokens){
  let idx = 0;
  function peek(){ return tokens[idx] || null; }
  function next(){ return tokens[idx++] || null; }
  function expect(type){ return peek() && peek().type === type ? next() : null; }

  const model = {torneo: null, equipos: [], eliminacion: {}};
  const errors = [];

  // utilidad para obtener un lexema de manera segura.
  function lexOf(t){ return t ? t.lexeme : null; }

  while(peek()){
    const t = peek();

    // TORNEO { ... }
    if(t.type === TokenType.RESERVED && t.lexeme === "torneo"){
      next(); // consume TORNEO
      if(!expect(TokenType.LBRACE)) { errors.push({lexema:"TORNEO", tipo:"Sintaxis", descripcion:"Se esperaba '{' después de TORNEO", line:t.line, col:t.col}); continue; }
      const obj = {};
      while(peek() && peek().type !== TokenType.RBRACE){
        const keyTok = next();
        if(!keyTok) break;
        if((keyTok.type === TokenType.RESERVED) || (keyTok.type === TokenType.IDENT)){
          // keyTok.lexeme ya es minúscula si es RESERVED o conserva el identificador original
          const key = keyTok.lexeme;
          if(!expect(TokenType.COLON)){
            errors.push({lexema:key, tipo:"Sintaxis", descripcion:`Falta ':' después de ${key}`, line:keyTok.line, col:keyTok.col});
          }
          const valTok = peek();
          if(valTok && (valTok.type === TokenType.STRING || valTok.type === TokenType.NUMBER || valTok.type === TokenType.IDENT)){
            obj[key] = next().lexeme;
            // Permitir coma final opcional
            if(peek() && peek().type === TokenType.COMMA) next();
            continue;
          } else {
            errors.push({lexema:key, tipo:"Sintaxis", descripcion:`Valor inválido o faltante para ${key}`, line:keyTok.line, col:keyTok.col});
            if(peek()) next();
            continue;
          }
        } else {
          next();
        }
      }
      if(peek() && peek().type === TokenType.RBRACE) next();
      // Permitir punto y coma después de bloque
      if(peek() && peek().type === TokenType.SEMICOLON) next();
      model.torneo = obj;
```

Construye la estructura de torneo a partir de tokens, Devuelve modelo, errores

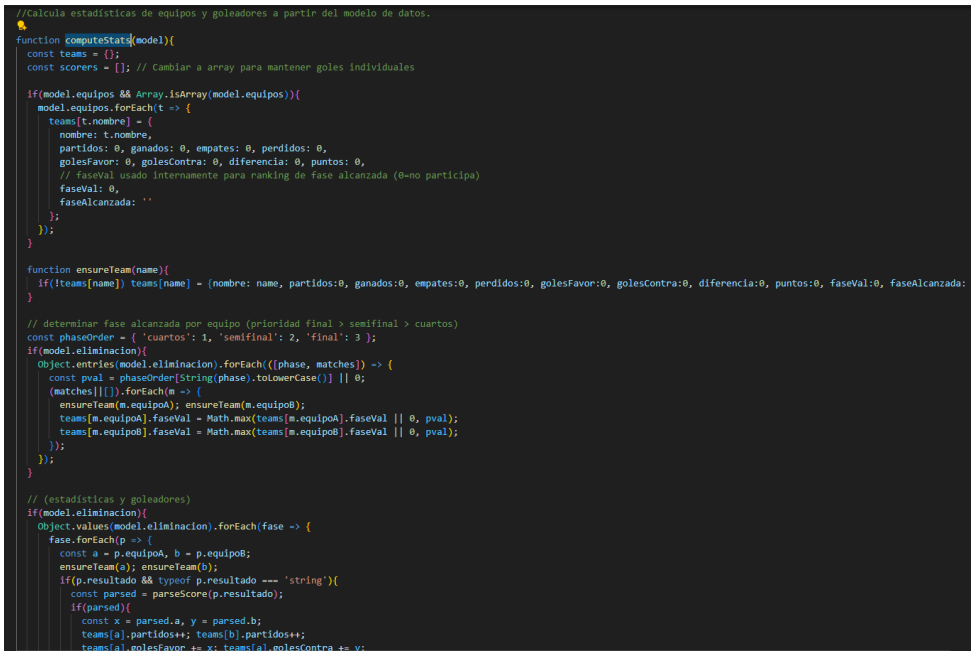
FUNCION (PARSESCORE)



```
function parseScore(resultado){
  if(!resultado || typeof resultado !== 'string') return null;
  const trimmed = resultado.trim();

  // Buscar patrón número-número manualmente
  let firstNum = "";
  let secondNum = "";
  let i = 0;

  // Saltar espacios iniciales
  while(i < trimmed.length && (trimmed[i] === ' ' || trimmed[i] === '\t')) i++;

  // Leer primer número
  while(i < trimmed.length && trimmed[i] >= '0' && trimmed[i] <= '9'){
    firstNum += trimmed[i];
    i++;
  }

  if(firstNum === "") return null;

  // Saltar espacios y buscar guión
  while(i < trimmed.length && (trimmed[i] === ' ' || trimmed[i] === '\t')) i++;

  if(i >= trimmed.length || trimmed[i] !== '-') return null;
  i++; // saltar el guión

  // Saltar espacios después del guión
  while(i < trimmed.length && (trimmed[i] === ' ' || trimmed[i] === '\t')) i++;

  // Leer segundo número
  while(i < trimmed.length && trimmed[i] >= '0' && trimmed[i] <= '9'){
    secondNum += trimmed[i];
    i++;
  }

  if(secondNum === "") return null;

  // Verificar que no haya más caracteres (solo espacios al final)
  while(i < trimmed.length && (trimmed[i] === ' ' || trimmed[i] === '\t')) i++;

  if(i < trimmed.length) return null; // hay caracteres extra

  return {
    a: parseInt(firstNum, 10),
    b: parseInt(secondNum, 10)
  };
}
```

Parsea resultados deportivos del formato "X-Y" sin usar expresiones regulares.

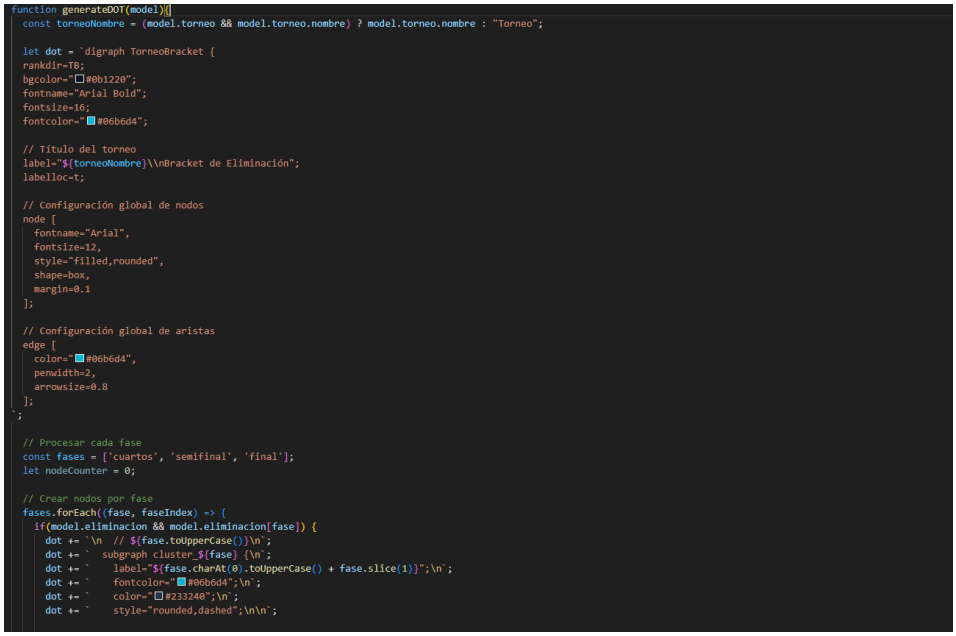
FUNCION (computeStats)



```
//Calcula estadísticas de equipos y goleadores a partir del modelo de datos.

function computeStats(model){
  const teams = {};
  const scorers = []; // Cambiar a array para mantener goles individuales

  if(model.equipos && Array.isArray(model.equipos)){
    model.equipos.forEach(t => {
      teams[t.nombre] = {
        nombre: t.nombre,
        partidos: 0, ganados: 0, empates: 0, perdidos: 0,
        golesFavor: 0, golesContra: 0, diferencia: 0, puntos: 0,
        // faseVal usado internamente para ranking de fase alcanzada (0=no participa)
        faseVal: 0,
        faseAlcanzada: ''
      };
    });
  }

  function ensureTeam(name){
    if(!teams[name]) teams[name] = {nombre: name, partidos:0, ganados:0, empates:0, perdidos:0, golesFavor:0, golesContra:0, diferencia:0, puntos:0, faseVal:0, faseAlcanzada:
  }

  // determinar fase alcanzada por equipo (prioridad final > semifinal > cuartos)
  const phaseOrder = { 'cuartos': 1, 'semifinal': 2, 'final': 3 };
  if(model.eliminacion){
    Object.entries(model.eliminacion).forEach(([phase, matches]) => {
      const pval = phaseOrder[String(phase).toLowerCase()] || 0;
      (matches||[]).forEach(m => {
        ensureTeam(m.equipoA); ensureTeam(m.equipoB);
        teams[m.equipoA].faseVal = Math.max(teams[m.equipoA].faseVal || 0, pval);
        teams[m.equipoB].faseVal = Math.max(teams[m.equipoB].faseVal || 0, pval);
      });
    });
  }

  // (estadísticas y goleadores)
  if(model.eliminacion){
    Object.values(model.eliminacion).forEach(fase => {
      fase.forEach(p => {
        const a = p.equipoA, b = p.equipoB;
        ensureTeam(a); ensureTeam(b);
        if(p.resultado && typeof p.resultado === 'string'){
          const parsed = parseScore(p.resultado);
          if(parsed){
            const x = parsed.a, y = parsed.b;
            teams[a].partidos++; teams[b].partidos++;
            teams[a].golesFavor += x; teams[a].golesContra += y;
```

Calcula estadísticas de equipos y goleadores a partir del modelo de datos.

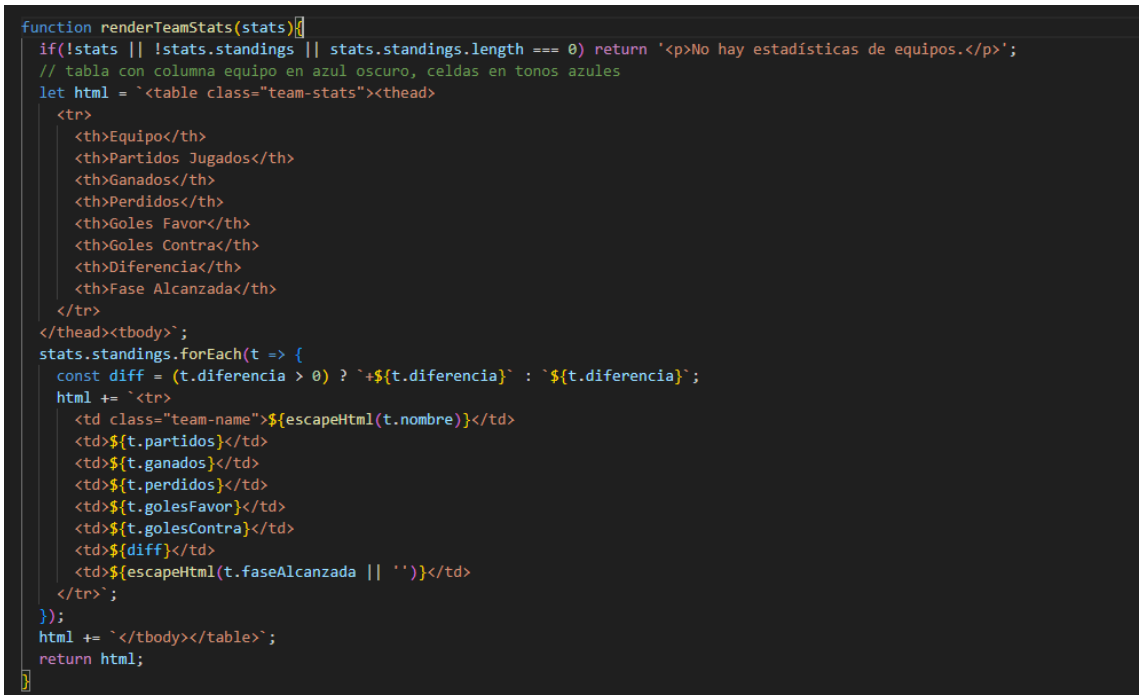
FUNCION(GENERATEDOT)



```
function generateDOT(model){
  const torneoNombre = (model.torneo && model.torneo.nombre) ? model.torneo.nombre : "Torneo";

  let dot = `digraph TorneoBracket {
  rankdir=TB;
  bgcolor="#0b1220";
  fontname="Arial Bold";
  fontsize=16;
  fontcolor="#06b6d4";

  // Título del torneo
  label="${torneoNombre}\\nBracket de Eliminación";
  labelloc=t;

  // Configuración global de nodos
  node [
    fontname="Arial",
    fontsize=12,
    style="filled,rounded",
    shape=box,
    margin=0.1
  ];

  // Configuración global de aristas
  edge [
    color="#06b6d4",
    penwidth=2,
    arrowsize=0.8
  ];
`;

  // Procesar cada fase
  const fases = ['cuartos', 'semifinal', 'final'];
  let nodeCounter = 0;

  // Crear nodos por fase
  fases.forEach((fase, faseIndex) => {
    if(model.eliminacion && model.eliminacion[fase]) {
      dot += `\n  // ${fase.toUpperCase()}\n`;
      dot += `  subgraph cluster_${fase} {\n`;
      dot += `    label="${fase.charAt(0).toUpperCase() + fase.slice(1)}";\n`;
      dot += `    fontcolor="#06b6d4";\n`;
      dot += `    color="#233240";\n`;
      dot += `    style="rounded,dashed";\n\n`;
```

Genera código DOT (Graphviz) para visualizar el bracket del torneo.

FUNCTION (RENDERTEAMSTATS)



```
function renderTeamStats(stats){
  if(!stats || !stats.standings || stats.standings.length === 0) return '<p>No hay estadísticas de equipos.</p>';
  // tabla con columna equipo en azul oscuro, celdas en tonos azules
  let html = `<table class="team-stats"><thead>
    <tr>
      <th>Equipo</th>
      <th>Partidos Jugados</th>
      <th>Ganados</th>
      <th>Perdidos</th>
      <th>Goles Favor</th>
      <th>Goles Contra</th>
      <th>Diferencia</th>
      <th>Fase Alcanzada</th>
    </tr>
  </thead><tbody>`;
  stats.standings.forEach(t => {
    const diff = (t.diferencia > 0) ? `+${t.diferencia}` : `${t.diferencia}`;
    html += `<tr>
      <td class="team-name">${escapeHtml(t.nombre)}</td>
      <td>${t.partidos}</td>
      <td>${t.ganados}</td>
      <td>${t.perdidos}</td>
      <td>${t.golesFavor}</td>
      <td>${t.golesContra}</td>
      <td>${diff}</td>
      <td>${escapeHtml(t.faseAlcanzada || '')}</td>
    </tr>`;
  });
  html += `</tbody></table>`;
  return html;
}
```

Le mostrara al usuario las estadísticas de equipo.

FUNCION (RENDERSCORESHTML)

```
function renderScorersHtml(scorers){
  if(!scorers || scorers.length === 0) return '<p>No hay goleadores registrados.</p>';

  let html = '<table class="scorers-table"><thead><tr><th>Posición</th><th>Jugador</th><th>Equipo</th><th>Goles</th><th>Minutos de Gol</th></tr></thead><tbody>';

  // Agrupar goles por jugador para calcular totales y minutos
  const groupedScorers = {};
  scorers.forEach(s => {
    const key = s.jugador;
    if(!groupedScorers[key]) {
      groupedScorers[key] = {
        jugador: s.jugador,
        equipo: s.equipo,
        goles: 0,
        minutos: []
      };
    }
    groupedScorers[key].goles++;
    if(s.minuto) {
      groupedScorers[key].minutos.push(s.minuto);
    }
  });

  // Convertir a array y ordenar por goles (descendente)
  const sortedScorers = Object.values(groupedScorers).sort((a,b) => b.goles - a.goles);

  // Renderizar tabla
  sortedScorers.forEach((s,i) => {
    const minutosText = s.minutos.length > 0 ? s.minutos.join(', ') + "'" : 'N/A';
    html += `<tr>
      <td class="position">${i+1}</td>
      <td class="player-name">${escapeHtml(s.jugador)}</td>
      <td class="team-name">${escapeHtml(s.equipo)}</td>
      <td class="goals">${s.goles}</td>
      <td class="minutes">${minutosText}</td>
    </tr>`;
  });

  html += '</tbody></table>';
  return html;
}
```

Le mostrará al usuario los goleadores del torneo.