# Análisis del Código: Analizador Léxico en Java

## 1. Descripción General

Este código implementa un analizador léxico para expresiones matemáticas con notación tipo LaTeX. Su objetivo es identificar tokens válidos, clasificar funciones y operadores, y detectar errores léxicos en una expresión leída desde un archivo de texto.

## 2. Estructura del Código

• enum Estado: define los estados posibles del autómata (inicio, lectura de funciones, error, etc.).
• class Token: representa un lexema reconocido con su tipo y posición.
• class ErrorLexico: representa un error detectado en la expresión, incluyendo una posible solución.
• FUNCIONES_VALIDAS: conjunto de funciones matemáticas válidas del lenguaje.
• transiciones: tabla de transiciones que define cómo cambia el estado del autómata.

## 3. Funcionamiento Paso a Paso

Se utiliza el método 'analizarLinea' para procesar una línea del archivo de entrada:

1. Se inicia en el estado INICIO.
2. Se recorre cada carácter de la línea.
3. Se clasifica el carácter según su tipo (letra, dígito, símbolo, función, etc.).
4. Se actualiza el estado del autómata según la tabla de transiciones.
5. Se registra cada token en una tabla de símbolos.
6. Si hay errores, se guardan en una tabla de errores con detalles.
7. Al finalizar, se valida si se llegó al estado FIN y no hubo errores para aceptar la cadena.

## 4. Detección de Errores

Los errores se detectan en los siguientes casos:
• Caracteres no reconocidos por el lenguaje.
• Variables con más de una letra.
• Funciones escritas sin la barra invertida inicial.
• Funciones inválidas (no incluidas en el conjunto de funciones válidas).
• Transiciones inválidas en el autómata (no hay camino posible desde el estado actual).

Cada error se clasifica con un código (E1, E2, E3, etc.), incluye una descripción, el valor incorrecto, su posición y una sugerencia de solución.

## 5. Tablas Generadas

• Tabla de Símbolos: muestra todos los tokens reconocidos en la expresión, incluyendo su tipo y posición.
• Tabla de Errores: muestra cada error encontrado con su código, descripción y solución sugerida.
• Si la cadena termina en el estado FIN y no hay errores, se acepta; de lo contrario, se rechaza.

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.*;


// Clase principal del analizador léxico

public class AnalizadorLexico {


  // Definición de los estados del autómata finito determinista (AFD)

  enum Estado {

    INICIO,          // Estado inicial

    PESO,            // Se ha leído el símbolo '$'

    FUNCION_INICIO,    // Inicio de una función (como '\')

    FUNCION_INVALIDA,  // Función mal escrita o no reconocida

    LETRA,           // Identificadores válidos de una sola letra

    DIGITO,          // Constantes enteras

    SIMBOLO,         // Operadores y símbolos válidos

    TRIGFUNC, LOGFUNC, OTRASFUNC, SIMBFUNC, // Subcategorías de funciones (no se usan directamente)
```

```
    CONTENIDO,        // Contenido general permitido después del inicio
    FIN,              // Estado final válido
    ERROR             // Estado de error
  }

  // Clase para representar un token válido o inválido
  static class Token {
    String lexema, tipo;
    int posicion;

    Token(String lexema, String tipo, int posicion) {
      this.lexema = lexema;
      this.tipo = tipo;
      this.posicion = posicion;
    }
  }

  // Clase para representar errores léxicos con toda su información
  static class ErrorLexico {
    String id, nombre, descripcion, valor, solucion;
    int posicion;

    ErrorLexico(String id, String nombre, String descripcion, String valor, int posicion,
String solucion) {
      this.id = id;
      this.nombre = nombre;
```

```
        this.descripcion = descripcion;
        this.valor = valor;
        this.posicion = posicion;
        this.solucion = solucion;
    }
}

// Conjunto de funciones válidas del lenguaje
static final Set<String> FUNCIONES_VALIDAS = new HashSet<>(Arrays.asList(
    "\\sin", "\\cos", "\\tan", "\\arcsin", "\\arccos", "\\arctan", "\\cot", "\\csc", "\\sec",
"\\sinh", "\\cosh", "\\tanh", "\\coth",
    "\\log", "\\ln", "\\lg", "\\frac", "\\cdot", "\\exp", "\\deg", "\\det", "\\dim", "\\gcd",
"\\hom", "\\inf", "\\sup", "\\ker",
    "\\lim", "\\liminf", "\\limsup", "\\max", "\\min", "\\Pr", "\\to", "\\arg",
    "\\theta", "\\alpha", "\\beta", "\\gamma", "\\delta", "\\epsilon", "\\pi", "\\mu",
"\\nu", "\\rho", "\\sigma", "\\phi", "\\psi", "\\omega"
));

// Mapa de transiciones: estado actual → (tipo de entrada → estado siguiente)
static Map<Estado, Map<String, Estado>> transiciones = new HashMap<>();

// Inicialización del autómata: se definen las transiciones posibles
static {
    add(Estado.INICIO, "$", Estado.PESO);
    add(Estado.INICIO, "otro", Estado.ERROR);

    add(Estado.PESO, "(", Estado.FUNCION_INICIO);
```

```
        add(Estado.PESO, "letra", Estado.LETRA);
        add(Estado.PESO, "digito", Estado.DIGITO);
        add(Estado.PESO, "simbolo", Estado.SIMBOLO);
        add(Estado.PESO, "funcion_valida", Estado.TRIGFUNC);
        add(Estado.PESO, "funcion_invalida", Estado.FUNCION_INVALIDA);
        add(Estado.PESO, "invalido", Estado.ERROR);
        add(Estado.PESO, "cualquier", Estado.CONTENIDO);

        // Transiciones desde estados intermedios al contenido
        for (Estado s : List.of(Estado.LETRA, Estado.DIGITO, Estado.SIMBOLO,
Estado.FUNCION_INICIO, Estado.TRIGFUNC, Estado.LOGFUNC, Estado.OTRASFUNC,
Estado.SIMBFUNC)) {
            add(s, "cualquier", Estado.CONTENIDO);
        }

        // Dentro de CONTENIDO, se aceptan más letras, dígitos o símbolos
        add(Estado.CONTENIDO, "letra", Estado.CONTENIDO);
        add(Estado.CONTENIDO, "digito", Estado.CONTENIDO);
        add(Estado.CONTENIDO, "simbolo", Estado.CONTENIDO);
        add(Estado.CONTENIDO, "funcion_valida", Estado.CONTENIDO);
        add(Estado.CONTENIDO, "$final", Estado.FIN);
        add(Estado.CONTENIDO, "invalido", Estado.ERROR);
    }

    // Método auxiliar para añadir una transición al AFD
    private static void add(Estado from, String inputType, Estado to) {
        transiciones.putIfAbsent(from, new HashMap<>());
```

```
        transiciones.get(from).put(inputType, to);
    }

    // Método principal que analiza una línea del archivo
    public static void analizarLinea(String linea) {
        Estado estado = Estado.INICIO;
        List<Token> tablaSimbolos = new ArrayList<>();
        List<ErrorLexico> errores = new ArrayList<>();

        int i = 0;
        while (i < linea.length()) {
            String tipoEntrada = null;
            char c = linea.charAt(i);

            // Delimitadores $
            if (c == '$') {
                tipoEntrada = (i == 0) ? "$" : "$final";
                tablaSimbolos.add(new Token("$", i == 0 ? "DELIMITADOR_INI" :
"DELIMITADOR_FIN", i));

            // Letras (identificadores o funciones sin barra)
            } else if (Character.isLetter(c)) {
                int j = i;
                while (j < linea.length() && Character.isLetter(linea.charAt(j))) j++;
                String lexema = linea.substring(i, j);
                boolean esFuncionEsperada = FUNCIONES_VALIDAS.contains("\\" + lexema);
```

```
            if (lexema.length() > 1) {
                if (esFuncionEsperada) {
                    errores.add(new ErrorLexico("E3", "Falta barra inversa", "Una función debe comenzar con '\\'", lexema, i, "Agregue '\\' antes de la función"));
                } else {
                    errores.add(new ErrorLexico("E2", "Variable inválida", "Una variable debe tener solo una letra", lexema, i, "Use una sola letra"));
                }
                tablaSimbolos.add(new Token(lexema, "ERROR", i));
            } else {
                tipoEntrada = "letra";
                tablaSimbolos.add(new Token(lexema, "IDENTIFICADOR", i));
            }
            i = j - 1;

        // Funciones con barra invertida
        } else if (c == '\\') {
            int j = i + 1;
            while (j < linea.length() && Character.isLetter(linea.charAt(j))) j++;
            String funcion = linea.substring(i, j);
            boolean valida = FUNCIONES_VALIDAS.contains(funcion);
            tipoEntrada = valida ? "funcion_valida" : "funcion_invalida";
            tablaSimbolos.add(new Token(funcion, valida ? "FUNCION" : "FUNCION_INVALIDA", i));
            if (!valida) {
                errores.add(new ErrorLexico("E4", "Función inválida", "La función no existe en el lenguaje", funcion, i, "Verifique el nombre de la función"));
```

```
            }
            i = j - 1;

        // Dígitos
        } else if (Character.isDigit(c)) {
            tipoEntrada = "digito";
            tablaSimbolos.add(new Token(String.valueOf(c), "CONST_ENTERA", i));

        // Símbolos válidos
        } else if ("^_{}=+-*/().,".indexOf(c) != -1) {
            tipoEntrada = "simbolo";
            tablaSimbolos.add(new Token(String.valueOf(c), "OPERADOR/SIMBOLO", i));

        // Carácter inválido
        } else {
            tipoEntrada = "invalido";
            errores.add(new ErrorLexico("E1", "Símbolo desconocido", "Carácter no reconocido por el lenguaje", String.valueOf(c), i, "Escriba un carácter válido"));
            tablaSimbolos.add(new Token(String.valueOf(c), "ERROR", i));
        }

        // Determinar el siguiente estado
        Map<String, Estado> posibles = transiciones.getOrDefault(estado, new HashMap<>());
        Estado siguiente = posibles.getOrDefault(tipoEntrada, posibles.get("cualquier"));

        if (siguiente == null) {
```

```
            errores.add(new ErrorLexico("E5", "Transición inválida", "No hay transición desde el estado actual", String.valueOf(c), i, "Verifique el orden de los símbolos"));
            estado = Estado.ERROR;
            break;
        }

        estado = siguiente;
        if (estado == Estado.ERROR) break;
        i++;
    }

    // Imprimir resultados
    System.out.println("Expresión: " + linea);

    // Tabla de símbolos
    System.out.println("\nTabla de símbolos:");
    System.out.printf("%-4s | %-8s | %-22s | %-8s\n", "Nº", "Lexema", "Token", "Posición");
    System.out.println("-----|----------|------------------------|----------");
    int n = 1;
    for (Token t : tablaSimbolos) {
        System.out.printf("%-4d | %-8s | %-22s | %-8d\n", n++, t.lexema, t.tipo, t.posicion);
    }

    // Evaluación de aceptación o errores
    if (estado == Estado.FIN && errores.isEmpty()) {
        System.out.println("\n✔ Cadena aceptada.\n");
```

```
        } else {
            System.out.println("\n❌ Cadena rechazada.");
            System.out.println("\nTabla de errores:");
            System.out.printf("%-4s | %-22s | %-28s | %-5s | %-4s | %-30s\n", "ID", "Nombre",
"Descripcion", "Val", "Pos", "Solucion");
            System.out.println("-----|----------------------|------------------------------|------|------|--------
----------------------");
            for (ErrorLexico e : errores) {
                System.out.printf("%-4s | %-22s | %-28s | %-5s | %-4d | %-30s\n", e.id, e.nombre,
e.descripcion, e.valor, e.posicion, e.solucion);
            }
        }

        System.out.println("\n----------------------------------------\n");
    }
    // Método principal que lee el archivo y analiza cada línea
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("src/expresiones.txt")))
{
            String linea;
            while ((linea = br.readLine()) != null) {
                analizarLinea(linea.trim());
            }
        } catch (IOException e) {
            System.out.println("❌ Error al leer el archivo: " + e.getMessage());
        }
    }
}
```