UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
INTRODUCCIÓN A LA PROGRAMACIÓN Y COMPUTACIÓN 2
CATEDRÁTICO: ING. DAVID ESTUARDO MORALES AJCOT
TUTOR ACADÉMICO: HERBERTH ABISAI AVILA RUIZ

MANUAL TÉCNICO

Josué Daniel Fuentes Díaz
CARNÉ: 202300668
SECCIÓN: B+

GUATEMALA, 28 DE MARZO DEL 2,025

ÍNDICE

Contenido

INTRODUCCIÓN

El presente documento constituye el manual técnico del proyecto AFDGraph, desarrollado como parte del proyecto 1. El objetivo principal de este proyecto es la construcción de una herramienta capaz de analizar, validar y graficar Autómatas Finitos Deterministas (AFD) definidos en archivos con extensión .lfp.

La aplicación fue desarrollada utilizando Java, haciendo uso exclusivo de Swing para la interfaz gráfica, conforme a los lineamientos establecidos por el curso. El sistema implementa un analizador léxico personalizado que procesa los archivos de entrada carácter por carácter, sin el uso de generadores automáticos como ANTLR, JFlex o similares. Además, se implementó un módulo de análisis estructural para validar la sintaxis de los AFDs y detectar posibles errores de forma detallada.

El proyecto también permite la visualización gráfica de los autómatas utilizando coordenadas calculadas con algoritmos de recorrido en anchura (BFS), y cuenta con la capacidad de generar reportes en formato HTML de los tokens válidos y errores léxicos detectados.

Este manual técnico documenta la estructura del sistema, las clases utilizadas, su funcionalidad principal y las decisiones de diseño tomadas durante el desarrollo, con el fin de facilitar su comprensión, mantenimiento y extensión futura.

OBJETIVOS

1. GENERAL

Desarrollar una aplicación en Java que permita el análisis léxico, validación estructural y representación gráfica de Autómatas Finitos Deterministas (AFD) definidos en archivos .lfp, cumpliendo con los lineamientos establecidos por el curso de Lenguajes Formales y de Programación.

2. ESPECÍFICOS

- Implementar un analizador léxico personalizado que procese carácter por carácter sin herramientas externas, generando tokens válidos y detectando errores léxicos en los archivos de entrada.
- Diseñar una interfaz gráfica con Java Swing que permita al usuario cargar archivos, visualizar los AFD definidos, graficar su estructura y generar reportes HTML de los análisis realizados.

ESPECIFICACIÓN TÉCNICA

- **REQUISITOS DE HARDWARE**
    - Procesador Mínimo 1 GHz o superior
    - Memoria RAM mínima: 512 MB
    - Al menos 100 MB de espacio disponible para la instalación y ejecución del programa

- **REQUISITOS DE SOFTWARE**
    - Compatible con Windows, Linux y macOS
    - Se puede utilizar cualquier IDE como Eclipse, IntelliJ IDEA o NetBeans para compilar y ejecutar el programa.
    - El software utiliza únicamente las librerías estándar de Java (como java.io, java.util, javax.swing, java.awt, etc.), sin requerir librerías externas adicionales
    - Java Runtime Environment (JRE) o Java Development Kit (JDK) versión 8 o superior

LÓGICA DEL PROGRAMA

➢ Paquete model

- **AFD.java:**
  Define la estructura y comportamiento de un Autómata Finito Determinista:

  - **Atributos**: nombre, descripción, conjunto de estados, alfabeto, estado inicial, estados finales y mapa de transiciones.
  - **Métodos:**
    - agregarEstado(), agregarSimbolo(), agregarFinal(), agregarTransicion(): permiten construir el autómata.
    - getSiguienteEstado(): obtiene el estado destino dada una transición.
    - validar(): verifica la consistencia del AFD (que el estado inicial y finales existan, y que todos los estados tengan transiciones)
    - Métodos toString() y toFormalString() para representar el AFD en texto

```java
package model;

import java.util.*;
import java.util.stream.Collectors;

public class AFD {
    private final String nombre;
    private String descripcion;
    private final Set<String> estados;
    private final Set<String> alfabeto;
    private String estadoInicial;
    private final Set<String> estadosFinales;
    private final Map<String, Map<String, String>> transiciones;

    public AFD(String nombre) {
        this.nombre = Objects.requireNonNull(nombre, "El nombre del AFD no puede ser nulo");
        this.estados = new LinkedHashSet<>(); // Mantiene orden de inserción
        this.alfabeto = new LinkedHashSet<>();
        this.estadosFinales = new LinkedHashSet<>();
        this.transiciones = new LinkedHashMap<>();
    }

    // Métodos de acceso
    public String getNombre() {
        return nombre;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
```

- **Estado.java:**

Representa un estado individual del autómata:

  - Atributos: nombre del estado, bandera si es estado inicial, y bandera si es estado final
  - Métodos:
    - setEsInicial(), setEsFinal(): controlan la configuración del estado.
    - Métodos equals() y hashCode() permiten comparar estados por nombre.
    - toString(): devuelve una representación legible del estado.

```
package model;

public class Estado {
    private final String nombre;
    private boolean esFinal;
    private boolean esInicial;

    public Estado(String nombre) {...8 lines }

    // Getters
    public String getNombre() {...3 lines }

    public boolean esFinal() {...3 lines }

    public boolean esInicial() {...3 lines }

    // Setters con validación
    public void setEsFinal(boolean esFinal) {...6 lines }

    public void setEsInicial(boolean esInicial) {...6 lines }


    //Representación textual del estado

    @Override
    public String toString() {...4 lines }


    // Comparación basada en el nombre del estado

    @Override
    public boolean equals(Object obj) {...6 lines }
```

- **Transicion.java:**
  Modela una transición entre dos estados:

  - **Atributos**: estado origen, símbolo de transición y estado destino
  - **Métodos:**
    - getEstadoOrigen(), getSimbolo(), getEstadoDestino(): acceso a los componentes.
    - toString(): representa la transición de forma entendible (q0 --a--> q1).
    - toFormalString(): exporta la transición en el formato requerido ("a" -> q1
    - Métodos equals() y hashCode() para comparación de transiciones

```
package model;

import java.util.Objects;

public class Transicion {
    private final String estadoOrigen;
    private final String simbolo;
    private final String estadoDestino;

    /** Constructor principal ...7 lines */
    public Transicion(String estadoOrigen, String simbolo, String estadoDestino) {...15 lines }

    // Getters
    public String getEstadoOrigen() {
        return estadoOrigen;
    }

    public String getSimbolo() {...3 lines }

    public String getEstadoDestino() {...3 lines }

      //Representación textual de la transición
    @Override
    public String toString() {...3 lines }

     //Comparación basada en estado origen, símbolo y estado destino

    @Override
    public boolean equals(Object obj) {...8 lines }

    /** Hash code basado en estado origen, símbolo y estado destino ...3 lines */
    @Override
    public int hashCode() {...3 lines }
```

- **Paquete controller**
  - **Analizador.java:**
    Gestiona la lectura, validación y procesamiento estructural de archivos .lfp que definen AFDs:

    - **Proceso:**
      - Abre el archivo seleccionado por el usuario.
      - Divide el contenido en bloques por AFDs.
      - Para cada bloque, analiza propiedades como descripción, estados, alfabeto, inicial, finales y transiciones.
      - Crea objetos AFD con la información válida y registra los errores por cada uno.
    - **Resultado:** Devuelve un mapa con todos los AFDs válidos y registra una lista de errores para cada AFD.

  - **AnalizadorLexico.java:**
    Realiza el análisis carácter por carácter del archivo .lfp, identificando tokens válidos y errores léxicos:

    - **Proceso:**
      - Recorre el texto carácter por carácter.
      - Clasifica los lexemas como identificadores, palabras reservadas, símbolos, cadenas, números o flechas.
      - Identifica errores léxicos como símbolos no válidos o cadenas sin cerrar.

      Resultado: Retorna una lista de objetos Token y retorna una lista de objetos ErrorLexico si se encontraron errores.

  - **Token.java:**
    Representa cada lexema reconocido durante el análisis léxico

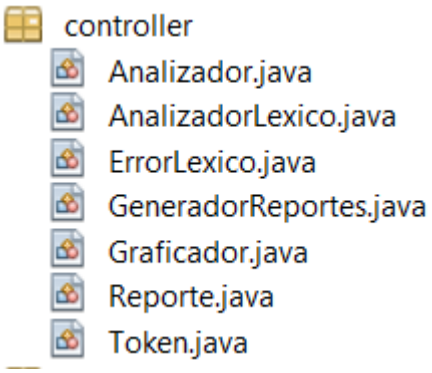
    - **Proceso:**
      - Guarda el tipo de token (simbolo, identificador, etc.), el lexema, y su posición (línea y columna)
      - Provee métodos para exportar el token en texto plano o formato tabla HTML.
    - **Resultado:** Instancias de Token que son utilizadas en el análisis y generación de reporte.

- **ErrorLexico.java:**
  Contiene los datos de errores encontrados durante el escaneo léxico del archivo .lfp:

    - **Proceso:**
        - Almacena el carácter problemático, su línea y columna, y una descripción.
        - Permite representar el error en texto o en HTML para reportes.
    - **Resultado:** Lista de errores que se utiliza para mostrar mensajes al usuario y generar reportes léxicos.

- **Reporte.java:**
  Genera archivos .html con tablas estilizadas para visualizar tokens y errores léxicos:

    - **Proceso:**
        - Toma listas de Token y ErrorLexico.
        - Crea tablas con encabezados, estilos visuales, y contenido ordenado.
        - Genera reportes individuales por AFD o un reporte general.
    - **Resultado:** Crea archivos tokens.html y errores.html en la carpeta reportes/ del proyecto.



➢ Paquete view

- **Interfaz.java:**
  Encapsula la interfaz gráfica principal de la aplicación. Permite al usuario cargar archivos, seleccionar y graficar AFDs, y generar reportes:

.

- **Componentes principales:**
  - JButton btnAnalizarArchivo: Abre el archivo .lfp y ejecuta el análisis.
  - JButton btnAnalizarArchivo: Abre el archivo .lfp y ejecuta el análisis.
  - JPanel panelDibujo: Área donde se grafica visualmente el autómata
  - JTextArea areaTexto: Muestra el contenido original del archivo leído
  - JButton btnGraficar: Grafica el AFD seleccionado del ComboBox
  - JButton btnGenerarReporte: Genera los reportes de tokens y errores léxicos
- **Resultado**: Interfaz amigable que permite visualizar los autómatas, ver errores y exportar reportes con un solo clic

```
package view;

import ...10 lines

/**...4 lines */
public class Interfaz extends JFrame {

    private final Analizador analizador;
    private final JButton btnAnalizarArchivo;
    private final JButton btnGraficar;
    private final JButton btnGenerarReporte;
    private final JComboBox<String> comboAFDs;
    private final JPanel panelDibujo;
    private final JTextArea areaTexto;
    private String contenidoOriginal = "";

    public Interfaz() {...62 lines }

    private void actualizarInterfazDespuesDeAnalisis() {...20 lines }

    private void graficarAFDSeleccionado() {...30 lines }

    private void mostrarErroresAFD(String nombreAFD) {...27 lines }

    private void actualizarComboBox() {...11 lines }

}
```