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, 2 DE MAYO DEL 2,025

ÍNDICE

Contenido

INTRODUCCIÓN

El presente manual técnico documenta el desarrollo del sistema Generador Visual de Mapas Narrativos, una herramienta diseñada para transformar descripciones textuales estructuradas en representaciones gráficas interactivas. Este proyecto fue desarrollado con el objetivo de aplicar conocimientos teóricos sobre análisis léxico, análisis sintáctico, gramáticas libres de contexto (GLC), autómatas finitos deterministas (AFD), generación de reportes y visualización mediante Graphviz.

El sistema implementa un analizador léxico y sintáctico desarrollado completamente desde cero, sin el uso de herramientas automáticas como JFlex o CUP, cumpliendo así con las restricciones establecidas. El análisis se basa en una gramática formal cuidadosamente definida, validando estructuras como mundos, lugares, conexiones y objetos especiales. Además, se genera una representación visual a partir de los datos procesados, permitiendo observar la disposición espacial y relaciones entre los elementos narrativos.

Este manual describe detalladamente los componentes internos del sistema, incluyendo las estructuras de datos, el AFD utilizado en el análisis léxico, el método del árbol para el análisis sintáctico y la construcción de la GLC. Asimismo, se detallan los módulos responsables de la generación de reportes y del graficado automático en formato .dot.

OBJETIVOS

1. GENERAL

Desarrollar una herramienta en Java capaz de analizar, validar y graficar descripciones narrativas estructuradas mediante un analizador léxico y sintáctico propio, generando visualizaciones automáticas e informes detallados de tokens y errores.

2. ESPECÍFICOS

- Implementar un analizador léxico basado en un AFD de diseño propio para identificar tokens válidos y errores léxicos en el archivo de entrada.
- Diseñar e implementar un analizador sintáctico mediante el método del árbol, validando la estructura del texto conforme a una gramática libre de contexto definida

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 models**

  - **Mundo.java:**
    Representa un mundo independiente definido en el archivo .lfp. Contiene las listas de lugares, conexiones y objetos especiales:

    - Atributos: nombre (String), lugares (List<Lugar>), conexiones (List<Conexion>), objetos (List<ObjetoEspecial>).
    - Métodos:
      - Constructor y getters para cada lista
      - Métodos auxiliares como agregarLugar(), agregarConexion() y agregarObjeto() si se desea extender funcionalidad.

  - **Lugar.java:**
    Representa una ubicación dentro de un mundo, con nombre, tipo y coordenadas:
    - Atributos: nombre (String), tipo (String), x (int), y (int).
    - Métodos:
      - Constructor, getters y toString().

  - **Conexion.java:**
    Representa una conexión dirigida entre dos lugares:
    - Atributos: origen (String), destino (String), tipo (String).
    - Métodos:
      - Constructor, getters y toString().

  - **ObjetoEspecial.java:**
    Define un objeto especial dentro del mundo, ya sea colocado en un lugar o por coordenadas:
    - Atributos: nombre (String), tipo (String), lugar (String opcional), x (int), y (int).
    - Métodos:
      - Constructores sobrecargados según ubicación.
      - Getters y toString().

- **Paquete parser**
  - **AnalizadorLexico.java:**
    Se encarga de realizar el análisis léxico del texto de entrada, identificando tokens válidos y registrando errores.

    - **Atributos:**
      - *entrada* : texto fuente.
      - *entrada*: texto fuente.
      - *errores*: lista de errores léxicos encontrados
      - *posicion, linea, columna*: seguimiento de posición en el análisis.
    - **Métodos:**
      - *analizar():* recorre el texto carácter por carácter y clasifica cada componente léxico.
      - Métodos auxiliares como *escanearPalabra(), escanearNumero(), escanearCadena() y agregarToken()*
      - Getters para obtener la lista de tokens y errores

  - **AnalizadorSintactico.java:**
    Realiza el análisis sintáctico sobre los tokens generados, validando estructuras según una gramática libre de contexto diseñada a mano.

    - **Atributos:**
      - tokens: lista de tokens a procesar.
      - tokens: lista de tokens a procesar.
      - mundos: lista de objetos Mundo generados al reconocer bloques válidos
    - **Métodos:**
      - *iniciar():* comienza el análisis iterando por cada bloque world.
      - analizarMundo(): analiza la estructura de un mundo completo, incluyendo lugares, conexiones y objetos
      - Métodos auxiliares: analizarLugar(), analizarConexion(), analizarObjeto(), verificar(), extraerLexema() y sincronizar()
      - Getters para acceder a los mundos analizados y errores

- **Paquete utils**
  - **GeneradorMapa.java:**
    Genera el archivo .dot y la imagen del mapa (.png) correspondiente a cada mundo analizado, usando Graphviz:
    - **Métodos principales:**
      - generarArchivoDot(Mundo mundo): crea el archivo DOT y ejecuta el comando dot para generar la imagen.
      - Usa las coordenadas definidas en el archivo .lfp para posicionar los nodos
      - Diferencia entre lugares y objetos, asignando formas, colores y emojis según su tipo
    - **Métodos auxiliares:**
      - obtenerFormaLugar(String tipo) y obtenerColorLugar(String tipo).
      - obtenerFormaObjeto(String tipo), obtenerColorObjeto(String tipo) y obtenerEmojiObjeto(String tipo).
      - obtenerEstiloConexion(String tipo) y obtenerColorConexion(String tipo).

  - **Reporte.java:**
    Se encarga de generar los reportes en formato HTML:
    - **Métodos:**
      - generarReporteTokens(List<Token> tokens): genera un archivo con una tabla de tokens válidos.
      - generarReporteErroresSeparados(List<Token> lexicos, List<Token> sintacticos): genera dos tablas separadas para errores léxicos y sintácticos
      - Ambos reportes se almacenan automáticamente en la carpeta reportes/

- **Paquete views**
  - **VentanaPrincipal.java:**
    Representa la interfaz gráfica principal del sistema. Permite cargar archivos, analizarlos, generar reportes y visualizar los mapas generados:
    - **Componentes principales:**
      - JTextArea txtAreaTexto: área de texto para cargar y editar el contenido del archivo .lfp.
      - JComboBox comboMapas: permite seleccionar el mundo a graficar
      - JLabel lblImagenMapa: área donde se muestra el mapa generado
    - **Botones funcionales:**
      - Cargar archivo: permite seleccionar y cargar un archivo desde el sistema.
      - Limpiar área: limpia tanto el área de texto como la imagen del mapa.
      - Analizar archivo: ejecuta el análisis léxico y sintáctico.
      - Generar reportes: produce reportes HTML de tokens y errores.
    - **Lógica relevante:**
      - analizarArchivo() busca y carga los mundos válidos al ComboBox.
      - procesarEntradaYMostrarMundos() ejecuta el análisis completo y guarda los mundos válidos.
      - generarMapa(String nombreMundo) llama al generador de mapas y muestra la imagen correspondiente.
      - Limpia la imagen si el nuevo archivo contiene errores para evitar confusión visual.

- **Paquete parser**
  - **AnalizadorLexico.java:**
    Implementa un analizador léxico propio, encargado de leer el texto ingresado y separar cada componente significativo como tokens:
    - **Funciones principales:**
      - Identifica tokens como palabras clave *(world*, *place*, *connect, etc.*), símbolos (*{, :, (, etc.),* identificadores, números y cadenas.
      - Maneja errores léxicos, como caracteres inesperados o cadenas mal cerradas
      - Utiliza un sistema manual sin herramientas externas como JFlex, empleando estructuras while, switch, y StringBuilder
    - **Resultados:**
      - Lista de Token válidos.
      - Lista de errores léxicos con línea y columna.
      - Soporta múltiples mundos dentro del mismo archivo .lfp.
  - **AnalizadorSintactico.java:**
    Se encarga de validar la estructura del archivo con base en una Gramática Libre de Contexto definida manualmente, sin herramientas automáticas como CUP:
    - **Funciones principales:**
      - Verifica la estructura completa de mundos, lugares, conexiones y objetos especiales.
      - Detecta errores sintácticos como llaves no cerradas, orden incorrecto de palabras clave, o estructuras mal formadas
      - Construye una lista de objetos Mundo, que contienen internamente Lugar, Conexion y ObjetoEspecial
    - **Soporte adicional:**
      - Mecanismo de sincronización para continuar el análisis después de un error.
      - Incluye mensajes detallados para cada tipo de error encontrado.

- **Paquete models**
  - **Mundo.java:**
    Clase principal que representa un mundo narrativo. Contiene los atributos:
    - nombre (String) — identificador del mundo.
    - Listas de objetos Lugar, Conexion y ObjetoEspecial, que se agrupan para formar la estructura completa del mundo.
  - **Lugar.java:**
    Representa un lugar dentro del mundo:
    - Atributos: nombre, tipo, x, y.
    - Se usan para graficar nodos con coordenadas específicas y forma/color según el tipo
  - **Conexion.java:**
    Define una conexión entre dos lugares del mundo:
    - Atributos: origen, destino, tipo.
    - Se representan como aristas dirigidas, con estilo visual diferente según el tipo (camino, puente, etc.)
  - **ObjetoEspecial.java:**
    Representa objetos que pueden estar ubicados en un lugar o en coordenadas:
    - Atributos: nombre, tipo, lugar (opcional), x, y (opcionales).
    - Cada tipo se grafica con un ícono (emoji), forma y color específicos

Construcción de tokens, expresiones regulares y método del árbol

El proceso de análisis léxico se desarrolló completamente de forma manual, sin emplear herramientas automáticas como JFlex o ANTLR, asegurando así un control detallado del reconocimiento de lexemas y errores.

➢ Tokens

Se diseñó una clase Token que encapsula el tipo de token, el lexema asociado y su posición en el texto fuente (línea y columna). Los tipos de token definidos se enumeran en el archivo TipoToken.java y cubren:

- Palabras clave: world, place, connect, object, to, with, at.
- Símbolos: {, }, (,), :, ,, " (para cadenas).
- Identificadores alfanuméricos
- Números (enteros).
- Errores léxicos (símbolos no válidos).

```java
package parser;

public enum TipoToken {
    // Palabras clave
    WORLD, PLACE, CONNECT, OBJECT, TO, WITH, AT,

    // Tipos
    IDENTIFICADOR, NUMERO, CADENA,

    // Símbolos
    LLAVE_IZQ, LLAVE_DER, PARENTESIS_IZQ, PARENTESIS_DER, COMA, DOS_PUNTOS,

    // Fin
    EOF,

    // Error
    ERROR
}
```

```java
package parser;

public class Token {
    public TipoToken tipo;
    public String lexema;
    public int linea;
    public int columna;

    public Token(TipoToken tipo, String lexema, int linea, int columna) {
        this.tipo = tipo;
        this.lexema = lexema;
        this.linea = linea;
        this.columna = columna;
    }

    @Override
    public String toString() {
        return tipo + " → \"" + lexema + "\" (línea " + linea + ", col " + columna + ")";
    }
```

➢ Expresiones Regulares

Las expresiones regulares se implementaron como condiciones en código Java dentro del método analizar() del AnalizadorLexico. No se usaron expresiones regulares escritas como tal, sino que el reconocimiento se dio a través de:

- **Letras**: inicio de palabras reservadas o identificadores.
- **Dígitos**: inicio de números enteros.
- **Comillas**: inicio de cadenas
- **Otros símbolos fijos**: comparación directa por carácter.

Ejemplo de equivalentes lógicos:

- **Identificadores:** [a-zA-Z][a-zA-Z0-9_]*.
- **Números**: [0-9]+
- **Cadenas**: \"[^\"]*\"

```
public void analizar() {
    while (!estaAlFinal()) {
        char actual = avanzar();

        // Saltar espacios en blanco
        if (Character.isWhitespace(actual)) {
            if (actual == '\n') {
                linea++;
                columna = 1;
            } else {
                columna++;
            }
            continue;
        }

        // Ignorar líneas que comienzan con @ (errores intencionales o comentarios)
        if (actual == '@') {
            errores.add(new Token(TipoToken.ERROR, "@", linea, columna));
            columna++;
            continue;
        }
```

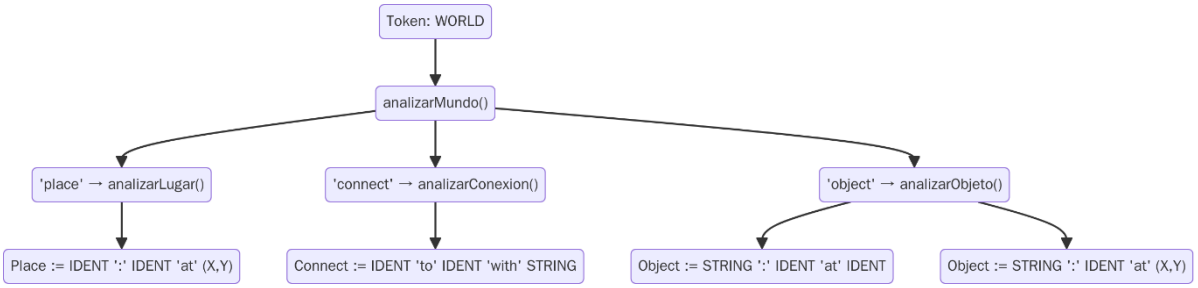
```
    // Reconocer símbolos simples
    switch (actual) {
        case '{':
            agregarToken(TipoToken.LLAVE_IZQ, "{");
            break;
        case '}':
            agregarToken(TipoToken.LLAVE_DER, "}");
            break;
        case '(':
            agregarToken(TipoToken.PARENTESIS_IZQ, "(");
            break;
        case ')':
            agregarToken(TipoToken.PARENTESIS_DER, ")");
            break;
        case ',':
            agregarToken(TipoToken.COMA, ",");
            break;
        case ':':
            agregarToken(TipoToken.DOS_PUNTOS, ":");
            break;
        case '"':
            escanearCadena();
            break;
        default:
            if (Character.isLetter(actual)) {
                escanearPalabra(actual);
            } else if (Character.isDigit(actual)) {
                escanearNumero(actual);
            } else {
                errores.add(new Token(TipoToken.ERROR, String.valueOf(actual), linea, columna));
                columna++;
            }
            break;
    }
```

- **Método del Árbol**

  Se utilizó un método tipo árbol descendente recursivo para la construcción del analizador sintáctico (AnalizadorSintactico). Cada nodo del árbol representa una regla gramatical de alto nivel como:

  - **Mundo:** world "nombre" { (Place | Connect | Object)* }.
  - **Place**: place ID : ID at (NUM, NUM)
  - **Connect** → connect ID to ID with "tipo"
  - **Object** → object "nombre" : ID at (ID | (NUM, NUM))

  Cada uno de estos no terminales tiene un método correspondiente (analizarMundo(), analizarLugar(), etc.) que consume los tokens esperados de forma ordenada y construye objetos de modelo (Mundo, Lugar, etc.), reflejando así una derivación estructurada del texto fuente.



➢ Construcción del AFD

Para implementar el analizador léxico del proyecto, se construyó un Autómata Finito Determinista (AFD) manualmente. Este autómata permite reconocer los distintos tokens válidos definidos en nuestro lenguaje, como palabras reservadas (*place, connect, object, etc*.), símbolos *({, }, :, ,, (,))*, identificadores, números y cadenas entre comillas.

- **Definición del alfabeto:**
  El alfabeto de entrada incluye:
  - Letras (a–z, A–Z).
  - Dígitos (0–9)
  - Caracteres especiales (", :, ,, {, }, (,))
  - Caracteres de espacio en blanco (, \t, \n)
- **Identificación de patrones léxicos:**
  Para cada tipo de token se definió una expresión regular. Algunos ejemplos:
  - **Palabras reservadas**: place, object, connect, world, etc.
  - **Identificadores**: letras o guiones bajos seguidos por letras o dígitos: [a-zA-Z_][a-zA-Z0-9_]*
  - **Números**: \d+
  - **Cadenas**: "[^"\n]*"
- **Diseño del AFD:**
  A partir de las expresiones regulares, se construyó el AFD con una serie de estados, transiciones y estados de aceptación. Por ejemplo:
  - Estado inicial q0 transiciona con "p" a un estado intermedio que puede conducir a place.
  - Con comillas " inicia una cadena hasta encontrar la comilla de cierre
  - Los números se aceptan si todos los caracteres son dígitos consecutivos

- **Codificación manual del AFD:**
  El AFD fue implementado directamente en código Java dentro de la clase AnalizadorLexico, usando condicionales (if, switch) y estructuras de control para simular los estados y transiciones. Por ejemplo:

```
// Reconocer símbolos simples
switch (actual) {
    case '{':
        agregarToken(TipoToken.LLAVE_IZQ, "{");
        break;
    case '}':
        agregarToken(TipoToken.LLAVE_DER, "}");
        break;
    case '(':
        agregarToken(TipoToken.PARENTESIS_IZQ, "(");
        break;
    case ')':
        agregarToken(TipoToken.PARENTESIS_DER, ")");
        break;
    case ',':
        agregarToken(TipoToken.COMA, ",");
        break;
    case ':':
        agregarToken(TipoToken.DOS_PUNTOS, ":");
        break;
    case '"':
        escanearCadena();
        break;
    default:
        if (Character.isLetter(actual)) {
            escanearPalabra(actual);
        } else if (Character.isDigit(actual)) {
            escanearNumero(actual);
        } else {
            errores.add(new Token(TipoToken.ERROR, String.valueOf(actual), linea, columna));
            columna++;
        }
        break;
}
```

  Cada método representa una **rama del árbol** de análisis léxico, y simula el comportamiento del AFD al procesar secuencias de caracteres

- **Resultado:**
  Gracias al AFD implementado manualmente, el programa es capaz de:
    - Reconocer todos los tokens definidos
    - Identificar errores léxicos
    - Generar una lista ordenada de tokens con posición (línea y columna)
    - Proveer una base robusta para el análisis sintáctico posterior

Estado	Entrada esperada	Acción / Transición	Estado siguiente	Token resultante
q0	"	Inicia cadena	q_cadena	—
q0	letra	Inicia palabra reservada/ID	q_palabra	—
q0	dígito	Inicia número	q_numero	—
q0	{	Reconoce llave izquierda	—	LLAVE_IZQ
q0	}	Reconoce llave derecha	—	LLAVE_DER
q0	:	Reconoce dos puntos	—	DOS_PUNTOS
q0	,	Reconoce coma	—	COMA
q0	(	Reconoce paréntesis izquierdo	—	PARENTESIS_IZQ
q0	)	Reconoce paréntesis derecho	—	PARENTESIS_DER
q0	@	Carácter inválido	—	ERROR
q_cadena	cualquier carácter ≠ "	Agrega carácter a la cadena	q_cadena	—
q_cadena	"	Fin de cadena	—	CADENA
q_palabra	letra/dígito/_	Continua palabra	q_palabra	—
q_palabra	otro símbolo	Finaliza palabra (revisar si es palabra clave o ID)	—	PLACE, OBJECT, etc. o IDENTIFICADOR
q_numero	dígito	Continúa número	q_numero	—
q_numero	otro símbolo	Finaliza número	—	NUMERO

- **Gramática Libre de Contexto:**

  La gramática fue implementada para reconocer la estructura del lenguaje de entrada utilizado en el sistema. Se usó para construir el analizador sintáctico del proyecto y permite validar archivos .lfp estructurados correctamente

  - **No terminales:**
    - S → símbolo inicial
    - MUNDO → definición de un mundo
    - SENTENCIAS → una o más sentencias dentro del mundo
    - LUGAR → definición de lugar
    - CONECCION → definición de conexión
    - OBJETO → definición de objeto especial
    - COORDS → coordenadas (x, y)
  - **Terminales (tokens):**
    - Palabras clave: world, place, connect, object, to, with, at
    - Símbolos: {, }, :, ,, (,), "
    - Identificadores: cadenas alfanuméricas
    - Cadenas: CADENA (entre comillas)
    - Números: NUM