

18/03/23

OLC1

Manual Técnico



Universidad de San Carlos de Guatemala

Organización de Lenguajes y Compiladores

*Carlos Eduardo Soto Marroquín
201902502*

Primer Semestre marzo 2022

Contents

Introducción

Dicho proyecto analiza un lenguaje específico ya detallado ya sea por medio de la lectura de un archivo con extensión “olc”, o por medio del ingreso del área de texto que posee la aplicación, en el archivo o cuadro de texto se tienen conjuntos, expresiones regulares y lexemas, que nos ayudarán a construir árboles de expresiones, autómatas tanto finitos deterministas como no deterministas, implementando el método del árbol, método de Thompson, etc. Finalmente mostrando los resultados en salidas como imágenes, archivos dot, y json.

Especificación Técnica

Los requerimientos mínimos para el posterior uso del programa son:

- Lenguajes implementados: JAVA, HTML, CSS
- IDE usado: NetBeans 8.0
- Editor de código: visual studio code
- Sistema operativo: Windows 10 PRO (64 bits)
- Interfaz gráfica: AWT, SWING, HTML, CSS
- Librerías implementadas: AWT, SWING, JFLEX, JCUP

Objetivos:

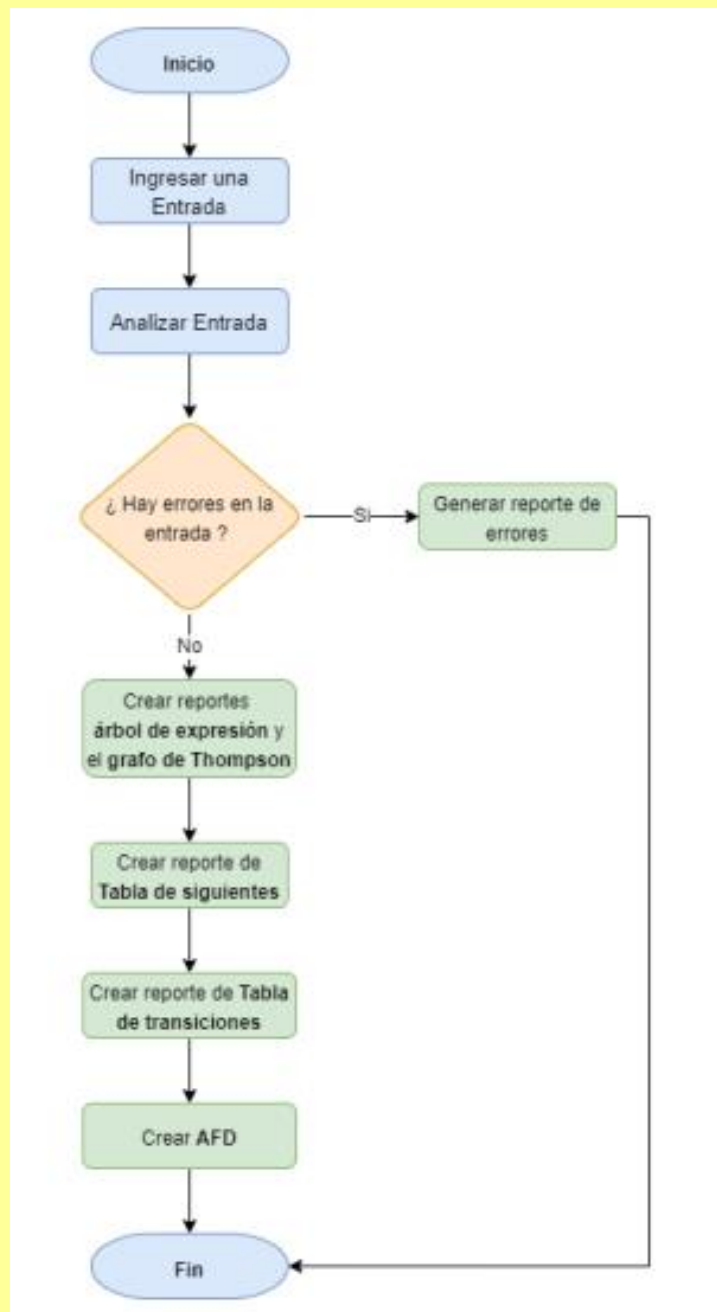
- Aplicación de conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para construir una solución de software que permita generar análisis por medio del método del árbol.
- Reforzamiento y aplicación del método del árbol de expresiones regulares en Autómatas Finitos Deterministas (AFD).
- Reforzamiento y aplicación del método de Thompson de expresiones regulares en Autómatas Finitos No Deterministas (AFND).
- Identificar y programar el proceso de reconocimiento de lexemas mediante el uso de Autómatas Finitos Deterministas.

Alcances del proyecto:

La intención del proyecto es que el encargado del desarrollo de esta aplicación aprenda a utilizar las herramientas que nos proveen las librerías de JFlex, y JCup, que se encargan de analizar un lenguaje en específico. Aplicando los conocimientos de lenguajes formales, y de organización de lenguajes y compiladores.

Lógica del programa

El programa sigue un flujo de trabajo, para el desarrollo de la aplicación, a continuación, se muestra el flujo de forma general:










Clases utilizadas:

	AFND.java
	Comparacion.java
	Conjuntos.java
	Container.java
	EXREGAN.java
	Encadenado.java
	Errores.java
	Estado.java
	Expresiones.java
	NodeType.java
	Nodo.java
	Reporte.java
	Transicion.java
	Ventana.form
	Ventana.java

Dichas clases fueron usadas, para la creación completa del proyecto.

Distribución de Paquetes:

Analizadores	
	Analizadores.java
	Lexico.java
	Lexico.java~
	Lexico.jflex
	Sintactico.cup
	parser.java
	sym.java

Archivo lexico.jflex:

Archivo donde se definen los tokens que se producirán con el archivo de entrada.














Archivo Sintactico.cup:

Archivo donde se define la sintaxis del lenguaje usado.

Clase Analizadores.java:

Ejecuta los archivos jflex y jcup y automáticamente se escriben los archivos léxico.java, parser.java y sym.java.

Clases principales

	AFND.java
	Comparacion.java
	Conjuntos.java
	Container.java
	EXREGAN.java
	Encadenado.java
	Errores.java
	Estado.java
	Expresiones.java
	NodeType.java
	Nodo.java
	Reporte.java
	Transicion.java
	Ventana.form
	Ventana.java

Creación del método del árbol

Clase Nodo.java

```
public class Nodo {

    public Nodo hizq;
    public Nodo hder;
    public String valor;
    public int id;
    public int identificador;
    public String anulable;
    public String primero;
    public String ultimo;
    public NodeType.Types type;

    public Nodo(Nodo hizq, Nodo hder, String valor, int id, int identificador, String anulable, String primero, String ultimo, NodeType.Types type) {
        this.hizq = hizq;
        this.hder = hder;
        this.valor = valor;
        this.id = id;
        this.identificador = identificador;
        this.anulable = anulable;
        this.primero = primero;
        this.ultimo = ultimo;
        this.type = type;
    }
}
```

En la clase nodo crearemos cada uno de los nodos del árbol que se divide por nodo izquierdo, nodo derecho, el valor, un id (no sirve para conectar nodos), identificador (para numerar las hojas no anulables y anulables), un string que identificará si el nodo es anulable, y string para definir los primeros y otro que define los últimos.

A través del archivo de cup se irán creando los nodos y definiendo su identificador si es anulable sus primeros y sus últimos.

```
NOTACIONER ::= or NOTACIONER: a NOTACIONER: b {
    String an;
    if(a.getAnulable()=="A" || b.getAnulable()=="A"){
        an="A";
    }else{
        an="N";
    }
    String prim= a.getPrimero()+b.getPrimero();
    String ult= a.getUltimo()+b.getUltimo();
    Nodo nuevaor = new Nodo(a, b, "|", parser.contId,0,an,prim,ult, Types.DISYUNCTION);

    parser.contId++;
    //revision
    String primero;
    String segundo;
    String ex;
    primero=re.pop();
    segundo= re.pop();
    ex="("+segundo+"|"+ primero+")";
    re.add(ex);
    RESULT = nuevaor;
};}
```

Siguiendo las siguientes reglas:

8.1.1.3 Determinar los nodos anulables

Se debe basar utilizando las siguientes reglas:

Terminal	No anulable
C_1^*	Anulable
C_1^+	No Anulable (Si C_1 es no anulable)
$C_1?$	Anulable
$C_1 C_2$	(Anulable(C_1) Anulable(C_2))?Anulable:No Anulable
C_1C_2	(Anulable(C_1)&&Anulable(C_2))?Anulable:No Anulable

8.1.1.4 Determinar los nodos primeros

Se debe basar utilizando las siguientes reglas:

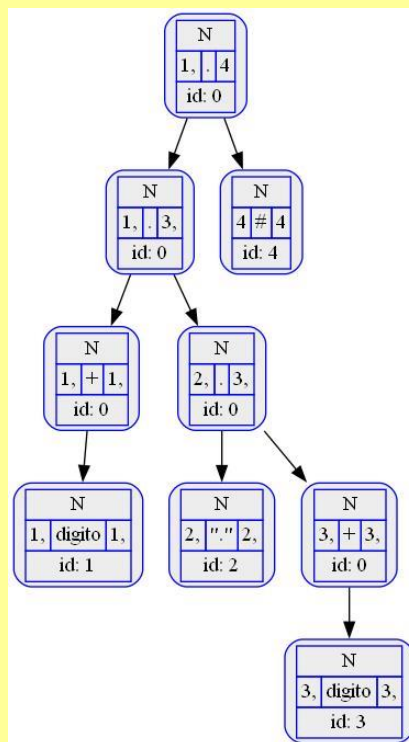
terminal	Terminal
C_1^*	Primeros(C_1)
C_1^+	Primeros(C_1)
$C_1?$	Primeros(C_1)
$C_1 C_2$	Primeros(C_1) + Primeros(C_2)
C_1C_2	(Anulable(C_1))?(Primeros(C_1) + Primeros(C_2)):Primeros(C_1)

8.1.1.5 Determinar los nodos últimos

Se debe basar utilizando las siguientes reglas:

terminal	Terminal
C_1^*	ultimo(C_1)
C_1^+	ultimo(C_1)
$C_1?$	ultimo(C_1)
$C_1 C_2$	ultimo(C_1) + ultimo(C_2)
C_1C_2	(anulable(C_2))?(ultimo(C_1) + ultimo(C_2)):ultimo(C_2)

Dando como resultado el árbol:



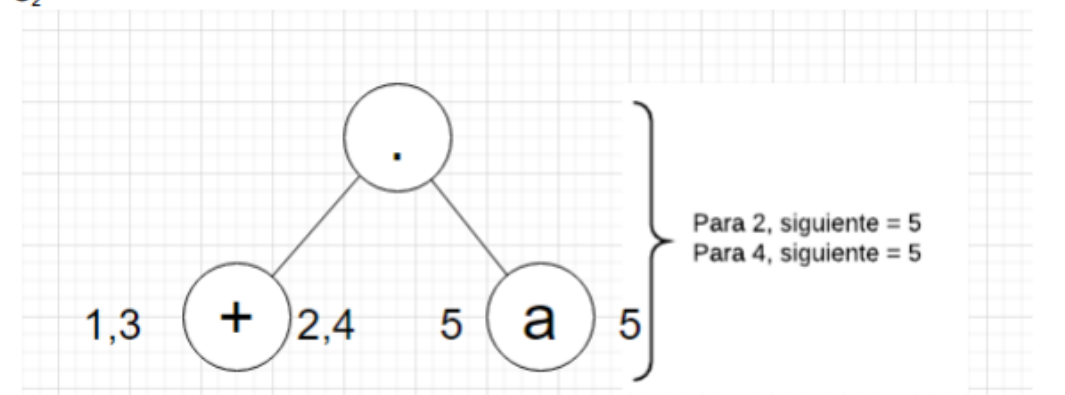
Clase Container.java:

Nuevamente utilizando el archivo de cup se irá añadiendo los siguientes a una estructura de datos hash table, creado dentro de la misma clase utilizando el método añadirSiguiente(), retornando los primeros y últimos de los nodos de la hoja.

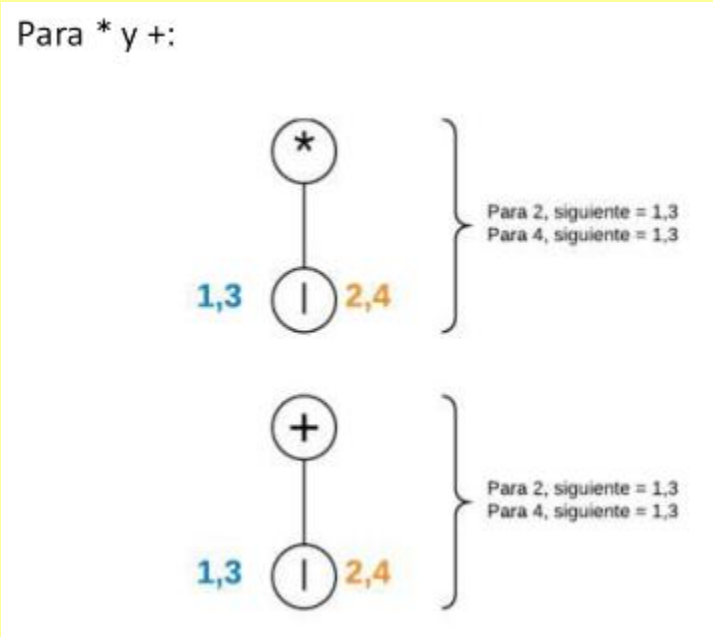
```
| suma NOTACIONER:a{:  
    String prim= a.getPrimero();  
    String ult= a.getUltimo();  
    String []aux= ult.split(",");  
    for (int i = 0; i < aux.length; i++) {  
        th.añadirSiguiente(aux[i],prim);  
    }  
}
```

Siguiendo las siguientes reglas:

En la concatenación los Siguietes para los últimos de C_1 son los primeros del Nodo C_2



Para * y a:



Y con el método GraficarSiguiente() graficará la tabla de siguientes:

Hoja		Siguientes
a	1	2,3,4
a	2	2,3,4
b	3	2,3,4
b	4	5
#	5	--

Tabla de transiciones

Clase Container.java:

Con el método Transición() de forma recursiva recorrerá cada estado para añadir a una lista todas las transiciones.

8.1.1.7 Construir la tabla de Transiciones

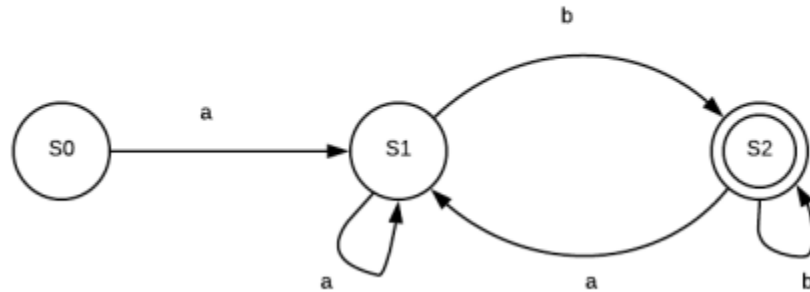
Estado	Terminales	
	a	b
S0 {1}	S1	--
S1 {2,3,4}	S1	S2
S2 {2,3,4,5}	S1	S2

Graficar Autómata

Clase Container.java:

Con el método GraficarAutomata() creará la imagen del autómata con las transiciones creadas con anterioridad.

8.1.1.8 Ilustración del AFD construido utilizando la tabla de Transiciones



Comparaciones

Clase Comparacion.java

Clase Encadenado.java

Clase Conjunto.java

En la clase comparación, con el método comparación() , retornaremos la lista de transiciones creadas con el método transición(), de la clase Container, y si encuentra match la cadena es aprobada de lo contrario la cadena es reprobada y con el método JSON(), se creará un reporte con las comparaciones que se realizaron y en el área de salida de la ventana aparecerá la cadena de la expresión donde se evaluó y si fue aprobada o reprobada.

Salida:

La cadena: "hola_soy_id_1" es incorrecta para la expresion: identificador
La cadena: "HoLA" es incorrecta para la expresion: identificador
La cadena: "301.59" es incorrecta para la expresion: decimales
La cadena: "1200.31" es incorrecta para la expresion: decimales

Reporte de Errores

Clase Reporte.java

Del lado de jflex y jcup, se creó un método para añadir a una lista de tipo Reporte con todos los errores léxicos que se encuentren en el archivo.

```
%{  
    public void AddError(String tipo, String lexema, int fila, int columna){  
        /*Aqui va la clase de Errores*/  
        Errores newError = new Errores(tipo, lexema, fila, columna);  
        Main.Reporte.listaErrores.add(newError);  
    }  
}%
```

AFND

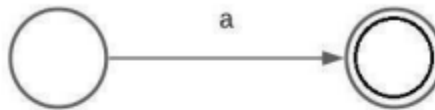
Clase AFND.java

```
public class AFND {  
  
    private Estado initial_state;  
    private ArrayList<Transicion> transiciones;  
  
    private final String epsilon = "ε";  
  
    public AFND(Nodo tree) {  
        transiciones = Buscar(tree);  
        //System.out.println("-----Codigo-----");  
        //System.out.println(tree.getCodigoInterno());  
        //System.out.println("-----Fin-Codigo-----");  
        transiciones.isEmpty();  
    }  
}
```

La clase afnd se le pasa el árbol que se genera en el Sintactico.cup, para poder realizar el procedimiento para construir el AFND.

Nos regimos a las siguientes reglas para crear el afnd:

9.2.1 Transición "a"



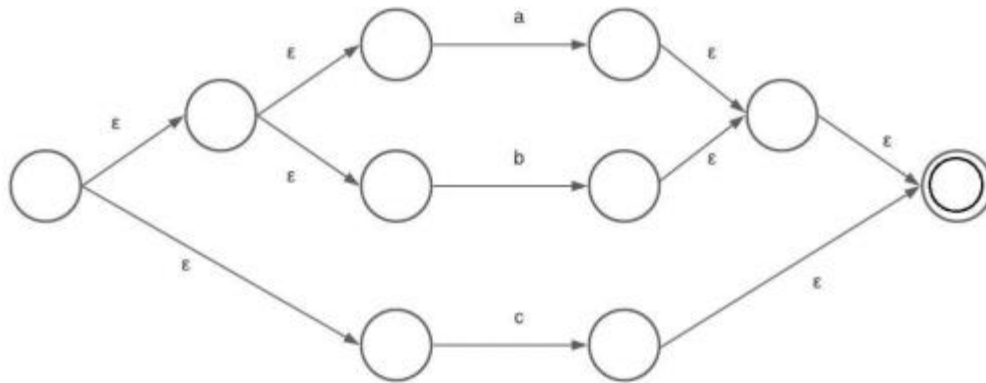
9.2.2 Transición epsilon "ε"



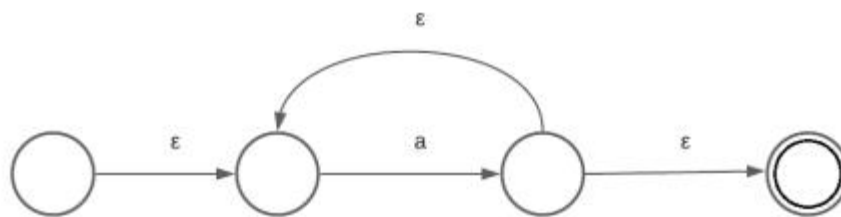
9.2.3 Transición "ab"



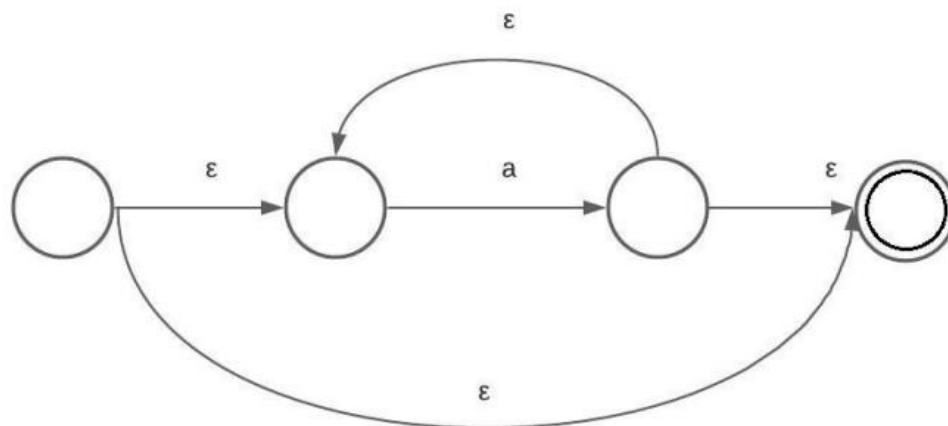
9.2.4 Transición "a|b|c"



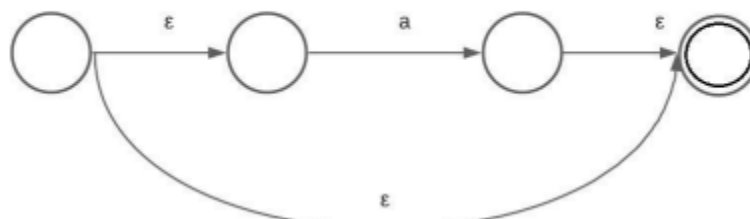
9.2.5 Transición "a⁺"



9.2.6 Transición "a^{*}"



9.2.7 Transición "a?"



Clase NodeType.java

```
public class NodeType {  
  
    public static enum Types {  
        LEAF,  
        CONCATENATION,  
        DISYUNCTION,  
        KLEENE_LOCK,  
        POSITIVE_LOCK,  
        BOOLEAN_LOCK  
    };  
}
```

La clase NodeType contiene un enum, que nos sirve, para identificar los nodos que posee el árbol.

Clase Transición.java

```
public class Transicion {  
  
    public String initialState; //Estado inicial del nodo  
    public String transition; //Transicion en la que se encuentra  
    public String finalState; //Estado hacia donde se dirige el nodo  
  
    public Transicion(String initialState, String transition, String finalState) {  
        this.initialState = initialState;  
        this.transition = transition;  
        this.finalState = finalState;  
    }  
}
```

La clase transición es la que define la forma de los estados de forma general, compuesto como tal en el estado inicial, la transición (número donde se encuentra), y el estado final, ósea hacia donde se dirige.

Clase Estado.java

```
public class Estado {  
  
    public String name;  
    public boolean is_final_state;  
    public Map<String, Estado> next_states = new HashMap<>();  
  
    public Estado(String name, boolean is_final_state) {  
        this.name = name;  
        this.is_final_state = is_final_state;  
    }  
  
    public Estado() {  
        this.name = "0";  
        this.is_final_state = false;  
    }  
}
```

Esta clase nos ayuda a identificar el nombre del estado donde nos encontramos, además de identificar si es el estado final, y de tener un map con los siguientes estados.

Reporte AFND

En la clase Reporte.java se encuentra el método graphAFND, que es el encargado de dibujar y crear la imagen del AFND.

```
public void graphAFND(Nodo tree, String name) {  
    FileWriter fichero = null;  
    PrintWriter pw = null;  
    try {  
        fichero = new FileWriter("src/AFND_201902502/" + name + ".dot");  
        pw = new PrintWriter(fichero);  
        AFND afnd = new AFND(tree);  
  
        String s = "digraph G {\n"  
            + "    node [shape=circle fontsize=13 fontname = \"helvetica\" style=filled fillcolor=\"#CCCCCC\"]; \n"  
            + "    nodesep=0.4;\n"  
            + "    ranksep=0.5;\n"  
            + "    rankdir=LR;\n\n";  
    }
```