Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 1 A

Manual de Técnico

Wilber Steven Zúñiga Ruano

Carné: 202006629

Guatemala, marzo 2023

Contenido

Introducción

El presente informe describe los aspectos técnicos de la aplicación con el objetivo de familiarizar al personal técnico con la codificación y funcionalidad.

La codificación de la aplicación se realizó en el lenguaje de JAVA, se utilizó el paradigma de la Programación Orientada a Objetos para poder encapsular los diversos métodos a utilizar en las diversas clases y estructuras de datos utilizadas.

Objetivos

Describir el diseño, funcionamiento y correcto uso de la aplicación para gestionar su uso, mostrando las especificaciones de los archivos de entrada y partes más relevantes del mismo.

Requerimientos

El sistema puede ser instalado en cualquier versión de Windows que cumpla con los siguientes requerimientos:

- **Versión de Java**: 1.8.1-291 o superior
- **JDK**: 1.8.0_111 o superior
- **JRE**: 1.8.0_291 o superior

Manual Técnico

1. Análisis Léxico

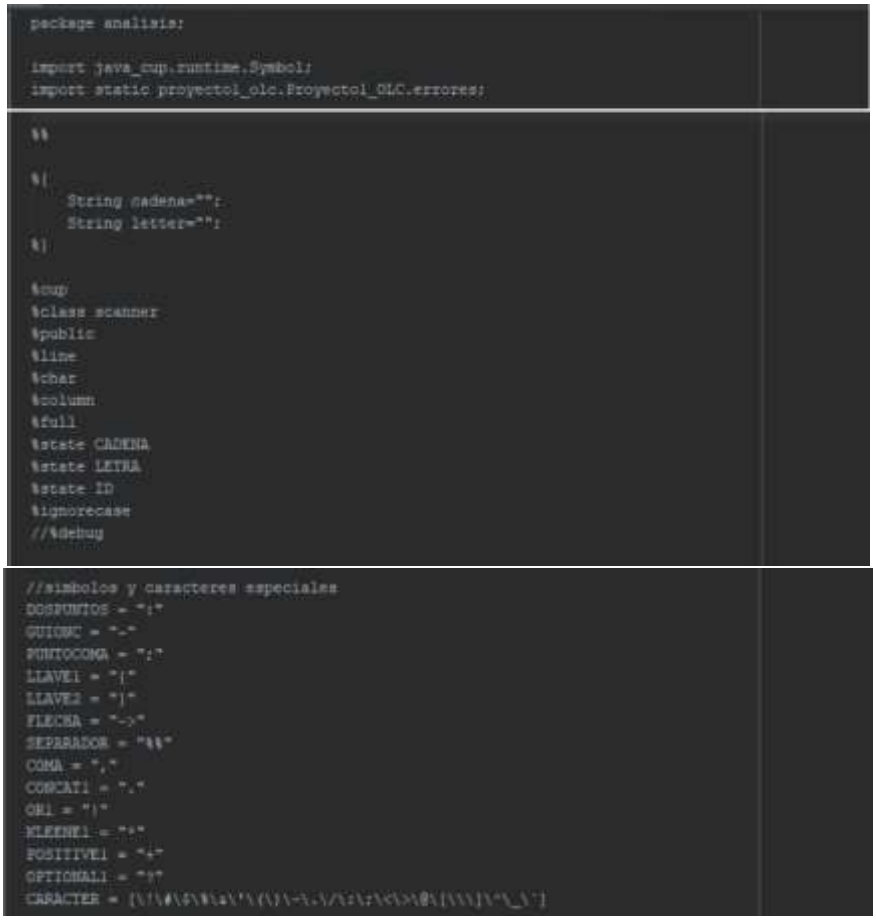
   La aplicación realiza un análisis léxico con la ayuda de la librería JFLEX para poder reconocer los lexemas de la cadena y convertirlos en tokens que se utilizaran para el análisis sintáctico. Para reconocer los tokens se utilizan expresiones regulares, las cuales definirán que token será cada lexema analizado.

   1.1. Crear Analizador léxico con JFLEX

   Para poder crear el analizador léxico con JFLEX es necesario crear un archivo con extensión jflex. Este archivo se divide en 5 segmentos los cuales son:

   - Importaciones
   - Declaración de variables globales
   - Declaración de opciones de jflex
   - Declaración de tokens y expresiones regulares
   - Definición de estados

   1.2. Ejemplo de archivo JFLEX



```
package analisis;

import java_cup.runtime.Symbol;
import static proyecto1_olc.Proyecto1_OLC.errores;
```

```
%%

%{
    String cadena="";
    String letter="";
%}

%cup
%class scanner
%public
%line
%char
%column
%full
%state CADENA
%state LETRA
%state ID
%ignorecase
//%debug
```

```
//simbolos y caracteres especiales
DOSPUNTOS = ":"
GUIONC = "~"
PUNTOCOMA = ";"
LLAVE1 = "{"
LLAVE2 = "}"
FLECHA = "->"
SEPARADOR = "%%"
COMA = ","
CONCAT1 = "."
OR1 = "|"
KLEENE1 = "*"
POSITIVE1 = "+"
OPTIONAL1 = "?"
CARACTER = [\!\#\$\%\&\'\(\)\-\.\/\:\;\<\>\@\[\\\]\^\_\`]
```

```
<YYINITIAL> {CONJ1}      { return new Symbol(sym.CONJ1, yyline, yycolumn,yytext());}
<YYINITIAL> {DOSPUNTOS}      { return new Symbol(sym.DOSPUNTOS, yyline, yycolumn,yytext());}
<YYINITIAL> {PUNTOCOMA}      { return new Symbol(sym.PUNTOCOMA, yyline, yycolumn,yytext());}
<YYINITIAL> {GUIONC}     { return new Symbol(sym.GUIONC, yyline, yycolumn,yytext());}
<YYINITIAL> {LLAVE1}     { return new Symbol(sym.LLAVE1, yyline, yycolumn,yytext());}
<YYINITIAL> {LLAVE2}     { return new Symbol(sym.LLAVE2, yyline, yycolumn,yytext());}
<YYINITIAL> {FLECHA}     { return new Symbol(sym.FLECHA, yyline, yycolumn,yytext());}
<YYINITIAL> {SEPARADOR}      { return new Symbol(sym.SEPARADOR, yyline, yycolumn,yytext());}
<YYINITIAL> {COMA}     { return new Symbol(sym.COMA, yyline, yycolumn,yytext());}
<YYINITIAL> [\"]       { yybegin(CADENA); cadena="_"; }
<YYINITIAL> {CONCAT1}      { return new Symbol(sym.CONCAT1, yyline, yycolumn,yytext());}
<YYINITIAL> {OR1}     { return new Symbol(sym.OR1, yyline, yycolumn,yytext());}
<YYINITIAL> {KLEENE1}      { return new Symbol(sym.KLEENE1, yyline, yycolumn,yytext());}
<YYINITIAL> {POSITIVE1}      { return new Symbol(sym.POSITIVE1, yyline, yycolumn,yytext());}
<YYINITIAL> {OPTIONAL1}      { return new Symbol(sym.OPTIONAL1, yyline, yycolumn,yytext());}
<YYINITIAL> {OPTIONAL1}      { return new Symbol(sym.OPTIONAL1, yyline, yycolumn,yytext());}
<YYINITIAL> {CARACTER}     { return new Symbol(sym.CARACTER, yyline, yycolumn,yytext());}
```

2. Análisis Sintáctico

   Luego del análisis léxico, la aplicación realiza un análisis sintáctico con la ayuda de la librería CUP, la cual permite realizar el análisis sintáctico en base a una gramática.

   2.1. Crear el analizador Sintáctico con CUP

   Para poder crear el analizador sintáctico con CUP es necesario crear un archivo con extensión cup. Este archivo se divide en 5 segmentos los cuales son:

   - Importaciones
   - Métodos de errores (recuperables y no recuperables)
   - Declaración de variables globales
   - Declaración de terminales y no terminales
   - Gramática

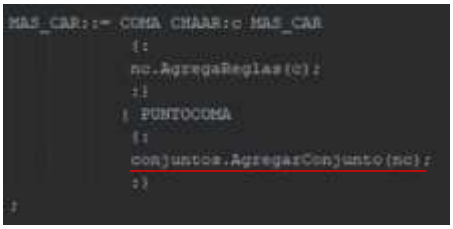
   2.2. Ejemplo de archivo CUP

   2.3. Compilación de archivos jflex y cup

   Una vez terminados los archivos jflex y cup es necesario compilarlos con la librería para que estos generen las clases que realizaran el análisis léxico y sintáctico. Para compilarlos se utilizó una clase auxiliar la cual contiene los métodos necesarios para su compilación.

2.4. Observaciones importantes

Durante el análisis sintáctico, se almacenan los conjuntos, expresiones regulares y valores de estas en dos listas simples enlazadas. Esto con el objetivo de poder capturar la información y poder utilizarla en la realización de los métodos.



3. Método del Árbol

El método del árbol nos sirve para la creación de autómatas finitos deterministas recibiendo una expresión regular. La expresión regular se transforma en un árbol y se realizan las funciones de anulable, primeros, y últimos en el árbol, posteriormente, se utiliza esta información para la realización de la tabla de siguientes y finalmente se utiliza la tabla de siguientes para realizar la tabla de transiciones la cual representa el autómata finito determinista con sus transiciones y estados de aceptación.

La implementación de dicho método se realizó por medio de estructuras de datos y objetos, siendo el objeto que encapsula todo “MetodoArbol”.

3.1. Objeto MetodoArbol

Objeto que encapsula toda la ejecución del método del árbol. Este posee 6 atributos los cuales son:

```
public class MetodoArbol {

    public String identificador;
    public ListaExpresiones expresiones;
    public Arbol a;
    public ListaSiguientes siguientes;
    public ListaTransiciones transiciones;
    public String cadena;
```

El constructor de esta clase recibe la lista de expresiones que se obtuvo del análisis sintáctico y utiliza dicha información para cargar el árbol.

3.1.1. Método “CargarArbol”

Este método realiza la inserción de nodos al árbol, insertando primero una concatenación y al final un carácter de fin de cadena que servirán para la elaboración del árbol.

3.1.2. Método “Ejecutar”

Este método manda a llamar a todos los métodos necesarios para la ejecución del método del árbol. Siendo estos:

```
public void Ejecutar() throws IOException, FileNotFoundException, DocumentException {
    this.a.Identifica_Hojas(this.a.raiz);
    this.a.Anulables(this.a.raiz);
    this.a.Primeros(this.a.raiz);
    this.a.Ultimos(this.a.raiz);
    this.siguientes.CrearNodos(this.a.raiz);
    this.siguientes.AgregandoSiguientes(this.a.raiz);
    this.a.GenerarReporteGraphviz();
    this.siguientes.GraficaPDF();
    this.transiciones.CargaSiguientes(this.siguientes, this.a.raiz);
    this.transiciones.ReporteTransiciones();
    this.transiciones.AFD_Graphviz();

}
```

3.1.3. Otros métodos

Adicionalmente a los métodos ya mencionados, se posee el método booleano que escanea la cadena con la ayuda del AFD resultante del método del árbol. De la salida de este método se ejecuta el método GenerarJSON y Salida los cuales generan las salidas del análisis, siendo el archivo JSON y el texto que se mostrara en la interfaz respectivamente.

```
public void CargarArbol() {...10 lines }

public boolean EscaneaCadena() {...49 lines }

public String Salida(boolean valida) {...15 lines }

public String GenerandoJSON(boolean valida) {...14 lines }
```

3.2. Objeto Árbol

Estructura de datos basada en el Árbol Binario de Búsqueda, la cual genera el árbol necesario para realizar el método del árbol. Dicha estructura utiliza dos métodos de inserción, el primero permite la inserción de los nodos necesarios para poder realizar el método del árbol y el segundo es un método recursivo que permite la inserción en los nodos del árbol. Ambos métodos se auxilian de una lista de tipo PILA (first in, last out). La pila permite realizar la correcta inserción de los nodos del árbol, almacenando un apuntador al nodo que tenga mayor prioridad para llenarse. Al finalizar cada inserción se verifica que los nodos de la pila estén completos, en caso de estarlo se eliminan de la pila.

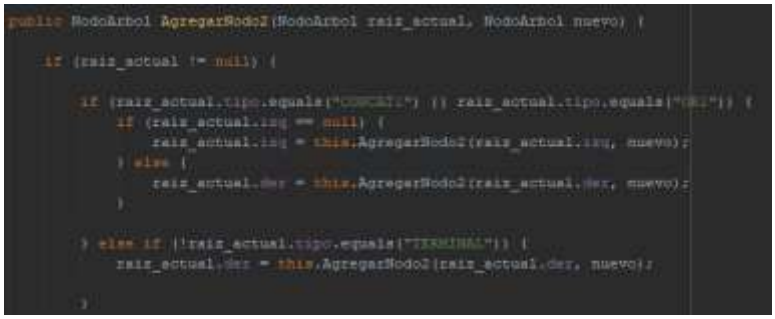
```
public class Arbol {

    public String identificador;
    public NodoArbol raiz;
    public boolean flag;
    public ListaPila NonTerm;
    public int contador;
    private int contador2;

    public Arbol(String id) {
        this.raiz = null;
        this.flag = false;
        this.NonTerm = new ListaPila();
        this.contador = 1;
        this.contador2 = 1;
        this.identificador = id;

    }
```

3.2.1. Método AgregarNodo

Primer método de inserción el cual permite la correcta inserción de los nodos necesarios para realizar el método del árbol. Este realiza el llamado al segundo método de inserción cuando sea necesario y hace la verificación de los nodos de la pila, viendo si están completos recorriéndolos a la derecha (valor tomado por defecto para los nodos que solo poseen un hijo y que permite verificar que los nodos de concatenación y or estén completos).

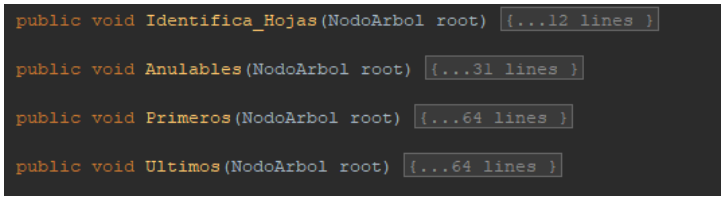
3.2.2. Método AgregarNodo2

Segundo método de inserción el cual se auxilia de la pila de inserción para poder realizar la correcta inserción de los nodos que tengan una prioridad mayor. En caso de que la pila este vacía recorre el árbol de manera recursiva para realizar la inserción (basado en el método recursivo de inserción del Árbol Binario de Búsqueda).



3.2.3. Métodos auxiliares para ejecutar el método del árbol

Dicha estructura posee 4 métodos auxiliares para la realización del árbol. Estos métodos utilizan el recorrido post-order (izquierda derecha raíz) para recorrer el árbol. El primer método se encarga de identificar con un número cada una de las hojas del árbol. El segundo método se encarga de verificar si son anulables en base a comparaciones y colocando la anulabilidad en el atributo anulable, siendo verdadero cuando lo se. El tercer método se encarga de colocar los valores primera posición en cada nodo, para esto se auxilia de una lista simple de números enteros. El cuarto método se encarga de colocar los valores de última posición, utilizando la misma lista de números enteros.



3.2.4. Métodos auxiliares para los reportes

Además de los métodos ya mencionados, esta estructura posee 4 métodos que permiten la generar el reporte del árbol con ayuda de Graphviz. Este reporte se realiza dos métodos principales, uno recibe el resultado de los que generan la cadena para él .dot y el segundo genera el archivo .dot y lo convierte en PDF. El primer método auxiliar se encarga de crear los nodos del árbol, recorriéndolo, usando recorrido post-order. El segundo método auxiliar se encarga de realizar las conexiones entre los nodos, nuevamente realizando el recorrido post-order.

```
public String GenerarDot() {...12 lines }

public String GenerarNodos(NodoArbol na) {...26 lines }

public String EnlazarNodos(NodoArbol na) {...15 lines }

public void GenerarReporteGraphviz() {...53 lines }
```

3.3. Objeto ListaSiguientes

Estructura de datos lineal que permite la realización de siguientes del método del árbol. Esta recibe la raíz del árbol y añade los terminales que este posee (las hojas del árbol). Cada nodo de esta lista posee una lista de enteros que almacena los identificadores de sus nodos siguientes. El cálculo de nodos siguientes se realiza en los nodos de concatenación, cerradura de Kleene y positiva. Utilizando el recorrido post-order, se navega dentro del árbol y al encontrar alguno de los nodos anteriormente mencionados, se realiza el cálculo de siguientes y se añaden los siguientes de cada nodo correspondiente.

3.3.1. Método CrearNodos

Este método recibe el nodo raíz del árbol, y lo recorre utilizando el recorrido post-order para poder encontrar las hojas (terminales) y poder añadirlos a esta lista. Adicionalmente, este se auxilia del método AgregarTerminales para este proceso.

```
public void CrearNodos(NodoArbol root) {
    if (root != null) {

        this.CrearNodos(root.izq);
        this.CrearNodos(root.der);

        if (root.EsHoja()) {
            this.AgregarTerminales(root.lexema, root.id);
        }
    }
}
```

```java
public void AgregarTerminales(String t, int i) {
    NodoSiguientes nuevo = new NodoSiguientes(t, i);
    if (this.inicio == null) {
        this.inicio = nuevo;
    }
    if (this.fin != null) {
        this.fin.sig = nuevo;
    }
    this.fin = nuevo;
    this.size += 1;
}
```

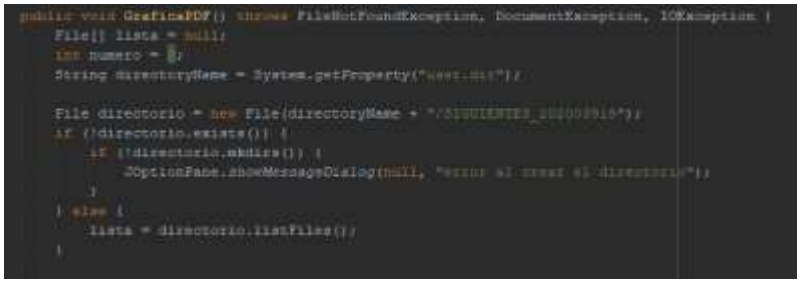
3.3.2. Método AgregarSiguiente

Este método busca el nodo del terminal en cuestión y agrega a su lista de enteros el identificador del nodo siguiente correspondiente.

```java
public void AgregarSiguientes(int id, int sig) {
    NodoSiguientes aux = this.inicio;
    while (aux != null) {
        if (aux.id_hoja == id) {
            aux.siguientes.AgregarPosiciones(sig);
            break;
        }
        aux = aux.sig;
    }
}
```

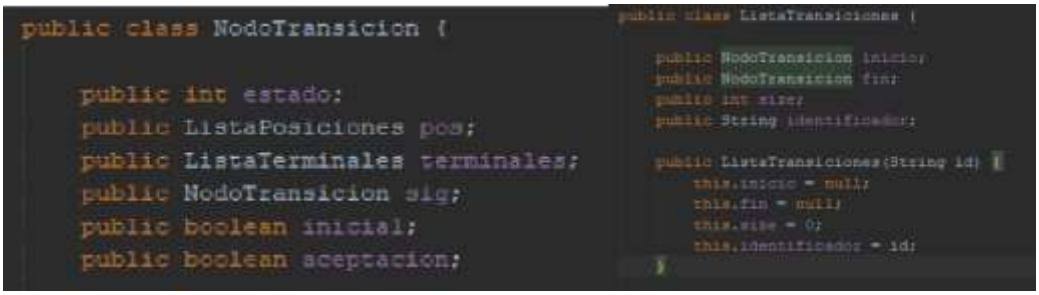
3.3.3. Método GraficaPDF

Para poder generar el PDF con la lista de siguientes, se utiliza la librería ITEXTPDF. Se recorre la lista principal (terminales) y en este recorrido se recorre la lista de siguientes de cada terminal y se genera la tabla fila por fila. Al finalizar la librería genera el PDF.



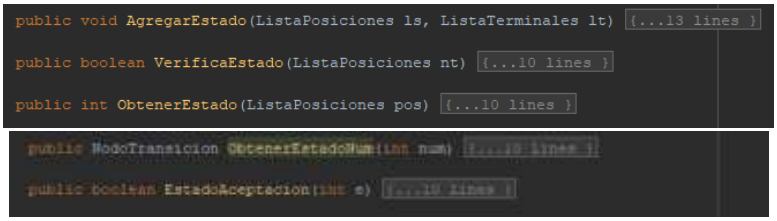
3.4. Objeto ListaTransiciones

Estructura de datos en la cual se crea la lista de transiciones. Esta estructura posee una lista de terminales, una de enteros, booleano de inicial y de aceptación; y un entero que representa el número de estado. Inicialmente esta recibe el nodo raíz del árbol, ya que los primeros de este es el estado inicial y para cada estado se crea una lista de terminales que apoyara al proceso a realizar.



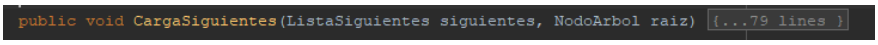
3.4.1. Métodos auxiliares para generar lista de transiciones

Para poder realizar dicho proceso, se utilizan 5 métodos auxiliares. El primero permite agregar un estado a la lista principal. El segundo verifica el estado si las listas de posiciones son iguales o no. El tercero obtiene el número de estado de una lista de posiciones dada. El cuarto retorna el nodo del estado que se busca por su número. Finalmente, el quinto busca un estado por su identificador y valida si es estado de aceptación.



3.4.2. Método CargarSiguientes

Este método recibe la lista de siguientes y el nodo raíz del árbol. Por defecto, agrega como primer estado se colocan los primeros del nodo raíz. Posteriormente empieza el proceso de creación de la lista. Mediante un ciclo se recorre la lista de estados (que inicialmente solo contara con el estado inicial), posteriormente se evaluara cada número que posee la lista de posiciones del nodo en cuestión, al encontrar coincidencias con algún terminal se le agrega a su propia lista de posiciones las posiciones resultantes del método. Luego de recorrer todos los terminales se vuelve a recorrer la lista de terminales para verificar si algún terminal posee una lista de posiciones que no se encuentre en la lista de estados, de ser así se crea el nuevo estado. Este proceso se realiza hasta llegar al final de la lista principal (se llegará al final cuando se hayan creado todos los estados necesarios de esta). Finalmente vuelve a recorrer la lista de estados, pero verificando en cada terminal que posea una lista de posiciones a que estado pertenece.



```
public void CargaSiguientes(ListaSiguientes siguientes, NodoArbol raiz) {
    ListaTerminales lt = new ListaTerminales();
    NodoSiguientes aux = siguientes.inicio;

    while (aux != null) {
        lt.AgregaTerminal(aux.Terminal);
        aux = aux.sig;
    }

    this.AgregarEstado(raiz.primeros, lt);

    NodoTransicion aux2 = this.inicio;
    while (aux2 != null) {


        NodoPosicion position = aux2.pos.inicio;
        while (position != null) {
            String ayuda = siguientes.BuscaTerminal(position.posicion);
            NodoTerminal termina = aux2.terminales.BuscaTerminal(ayuda);
```

3.4.3. Métodos auxiliares para los reportes

De esta estructura se obtienen 2 reportes, los cuales son en la tabla de transiciones y el grafo del AFD resultante del método del árbol. Para la tabla de transiciones se utiliza la librería ITEXTPDF, para la cual es necesario recorrer la lista principal y si lista interna para poder crear la tabla fila por fila y la librería pueda crear el PDF. Para el AFD se utiliza Graphviz, se recorre inicialmente la lista principal para poder crear los nodos y luego se vuelve a recorrer junto a su lista interna para generar las transiciones entre estados con los diferentes terminales. Al terminar este proceso se genera el archivo .dot y se convierte a PDF.

```
public void ReporteTransiciones() throws FileNotFoundException, DocumentException, IOException {...75 lines }

public String GenerarDot() {...37 lines }

public void AFD_Graphviz() {...54 lines }
```

4. Método de Thompson

El método de Thompson nos sirve para la creación de autómatas finitos no deterministas (es decir, que poseen transiciones épsilon). Para los metas caracteres (contactenación, or, cerradura positiva, cerradura de Kleene y la opcional) existe una estructura definida para la creación del autómata finito no determinista. La expresión regular se va transformando, tomando las operaciones más internas y se van uniendo con las mayores (creando primero las estructuras más simples y luego se van uniendo las estructuras) hasta tener un autómata finito no determinista resultante

La implementación de dicho método esta realizada por medio de estructuras de datos auxiliares y la clase principal llamada “MetodoThompson”. Las estructuras de datos utilizadas son: Lista simple enlazada, Lista doblemente enlazada y Grafo (lista de listas)

4.1. Objeto MetodoThompson

Esta clase posee 5 atributos los cuales permiten realizar el método de Thompson, los cuales son:

El constructor de esta clase recibe la lista de expresiones que se obtuvo del análisis sintáctico y se realiza una copia de esta en una lista doblemente enlazada, mediante el método CargaListaDoble.

```
public MetodoThompson(ListaExpresiones expresiones, String id) {
    this.expresiones = expresiones;
    this.identificador = id;
    this.terminales = new ListaDoble();
    this.listaGrafos = new ListaGrafos();
    this.CargarListaDoble();
}

public void CargarListaDoble() {
    NodoExpresion aux = expresiones.inicio;
    while (aux != null) {
        terminales.Insertar(aux.lexema, aux.tipo);
        aux = aux.sig;
    }
}
```

4.1.1. Método Ejecutar

Este es el encargado de realizar el autómata finito no determinista. Este recorre la lista doblemente enlazada de terminales (copia de la obtenida en el análisis sintáctico) y la recorre desde el final hasta el inicio. Al encontrar algo que no sea un terminal (alguno de los meta caracteres ya mencionados), verifica que meta carácter es y realiza un subgrafo. Este subgrafo es almacenado en una lista de grafos y en la lista de terminales se agrega el identificador del grafo en lugar de la operación que se realizó (eliminando de

la expresión tanto el meta carácter como los terminales utilizados con este). Dependiendo del meta carácter en cuestión se realizan las siguientes acciones:

4.1.1.1. Concatenación

Al encontrar una concatenación (.) se poseen cuatro casos, los cuales son:

- Grafo . Grafo: En este caso se toman ambos grafos y se unifican en uno solo. Tomando el inicial del grafo 1 como el inicial del grafo resultante. Se une el nodo final del grafo 1 con el inicio del grafo 2.
- Grafo . Terminal: En este caso se toma el grafo existente y se le añade la transición al terminal en cuestión.
- Terminal . Grafo: En este caso se crea la estructura de la concatenación del terminal y se une al nodo inicial del grafo en cuestión, generando un grafo nuevo y almacenándolo.
- Terminal . Terminal: Caso más sencillo el cual solo genera la estructura de una concatenación simple y la almacena en la lista de grafos.

4.1.1.2. OR

Al encontrar un or (|) se poseen cuatro casos, los cuales son:

- Grafo . Grafo: En este caso se crean el nodo inicial y final de la estructura del or, se toma el grafo 1 y 2 y se añaden al nuevo grafo; del nuevo nodo inicial se hace una transición al nodo inicial del grafo 1 y a la inicial del grafo 2 con épsilon. Del nodo final del grafo 1 y 2 se realiza una transición al nuevo estado final con épsilon.
- Grafo . Terminal: En este caso se crean el nodo inicial, concatenación con el terminal y final de la estructura del or, se toma el grafo 1 y se añade al nuevo grafo; del nuevo nodo inicial se hace una transición al nodo inicial del grafo con épsilon. Del nodo final del grafo se realiza una transición al nuevo estado final con épsilon.
- Terminal . Grafo: En este caso se crean el nodo inicial, concatenación con el terminal y final de la estructura del or, se toma el grafo 1 y se añade al nuevo grafo; del nuevo nodo inicial se hace una transición al nodo inicial del grafo con épsilon. Del nodo final del grafo se realiza una transición al nuevo estado final con épsilon.
- Terminal . Terminal: Caso más sencillo el cual solo genera la estructura de un or simple y la almacena en la lista de grafos.

4.1.1.3. Cerradura de Kleene

Al encontrar la cerradura de Kleene (*) existen dos casos, los cuales son:

- Grafo: Se crea el nodo inicial y final de la estructura de Kleene y se une al grafo, tomando como incio->siguiente el nodo inicial del grafo y el final->siguiente con el nodo final del nuevo grafo. Además, se realizan las transiciones con épsilon en los estados respectivos
- Terminal: Caso más sencillo el cual solo genera la estructura de una cerradura de Kleene simple y la almacena en la lista de grafos.

4.1.1.4. Cerradura Positiva

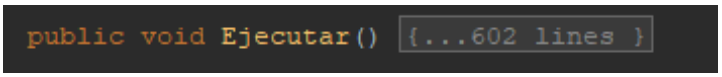
Al encontrar la cerradura Positiva (+) existen dos casos, los cuales son:

- Grafo: Se crea el nodo inicial y final de la estructura de la cerradura Positiva y se une al grafo, tomando como incio->siguiente el nodo inicial del grafo y el final->siguiente con el nodo final del nuevo grafo. Además, se realizan las transiciones con épsilon en los estados respectivos
- Terminal: Caso más sencillo el cual solo genera la estructura de una cerradura Positiva simple y la almacena en la lista de grafos.

4.1.1.5. Cerradura Opcional

Al encontrar la cerradura Opcional (?) existen dos casos, los cuales son:

- Grafo: Se crea el nodo inicial y final de la estructura de la cerradura Opcional y se une al grafo, tomando como incio->siguiente el nodo inicial del grafo y el final->siguiente con el nodo final del nuevo grafo. Además, se realizan las transiciones con épsilon en los estados respectivos
- Terminal: Caso más sencillo el cual solo genera la estructura de una cerradura Opcional simple y la almacena en la lista de grafos.



4.1.2. Métodos para reporte AFND

Se utilizan dos métodos para poder realizar el grafo del autómata finito no determinista. El primero recorre el grafo creando los nodos y transiciones en una cadena. El segundo recibe dicha cadena y genera el archivo .dot y lo convierte a PDF.