202111718

Manual Técnico

Proyecto 1

Walter Javier Santizo Mazariegos

Introducción

El proyecto se centra en el análisis léxico y sintáctico utilizando expresiones regulares y autómatas finitos deterministas para validar cadenas. Fue creado en JAVA con Jflex y Cup para analizar el archivo de entrada y generar archivos de salida que muestran las cadenas y métodos utilizados para crear el AFD con Graphviz. Este proyecto es innovador en el uso de tecnologías para la gestión eficiente en la entrada de datos que recibe nuestro analizador ya que reconoce n tipo de expresiones según los requerimientos solicitados

El propósito de la aplicación es cumplir con los requisitos establecidos en el curso de Organización de Lenguajes y Compiladores 1, que es parte de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala. Estos requisitos incluyen la capacidad de analizar datos de entrada con un formato específico y generar informes, entre otras funciones. Además, la aplicación está diseñada para implementar el método del árbol y el método de Thompson con los datos ingresados, para que los estudiantes del curso puedan verificar la corrección de las respuestas en tareas y exámenes. La aplicación ofrece diversas funciones para la gestión de información. Finalmente, cuenta con la opción de generar informes en formato HTML y JSON.

Resumen

Un compilador es un programa informático que se encarga de transformar el código fuente de un programa escrito en un lenguaje de programación de alto nivel (como C++, Java o Python) en un código ejecutable en una máquina específica (como un ordenador o un teléfono móvil).

un compilador es un programa que traduce el código de programación escrito en un lenguaje de alto nivel que los humanos pueden entender, a un lenguaje de bajo nivel que la máquina puede entender. Esto se hace para que el código pueda ser ejecutado en la computadora de destino. El proceso de compilación involucra varias fases, como análisis léxico, análisis sintáctico, análisis semántico, generación de código y optimización de código. Cada una de estas fases se encarga de realizar una tarea específica para transformar el código de entrada en código ejecutable.

Los compiladores tienen una amplia variedad de aplicaciones en el campo de la informática. Aquí te presento algunas de las aplicaciones más comunes:

Desarrollo de software: Los compiladores son una herramienta fundamental para los desarrolladores de software, ya que les permiten escribir programas en lenguajes de programación de alto nivel, como C++, Java, Python, entre otros, y luego compilarlos para ser ejecutados en diferentes plataformas.

Optimización de código: Los compiladores también se utilizan para optimizar el código fuente de un programa, lo que puede mejorar su rendimiento y velocidad de ejecución.

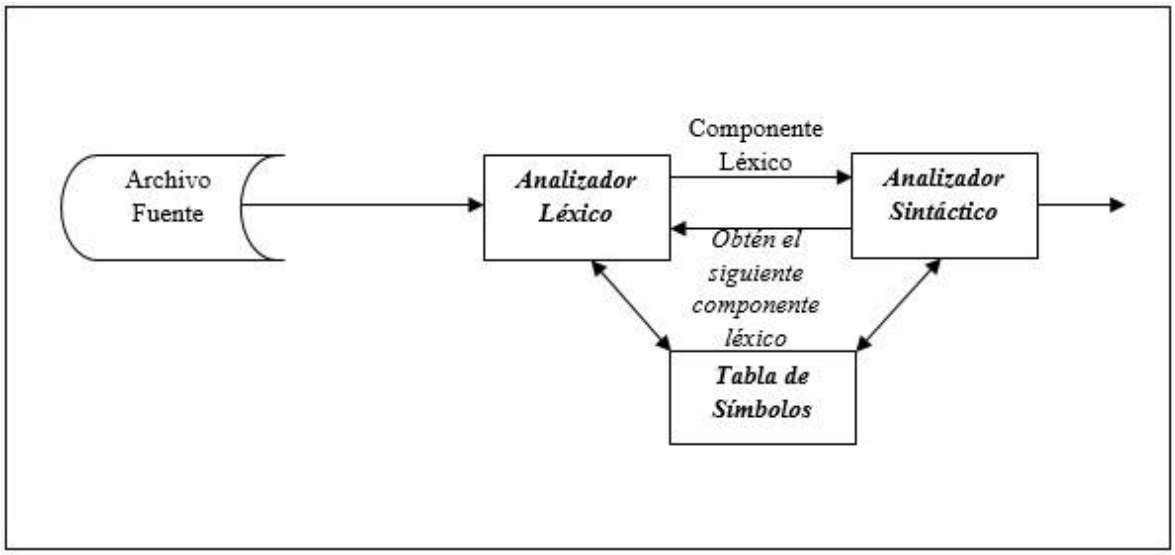
Seguridad: Los compiladores también se utilizan para mejorar la seguridad de los programas, ya que pueden detectar vulnerabilidades en el código fuente, como errores de programación y vulnerabilidades de seguridad.

Generación de código para hardware específico: Los compiladores pueden generar código para ser ejecutado en hardware específico, como procesadores de gráficos, sistemas embebidos y dispositivos móviles.

Desarrollo

Las tecnologías utilizadas para la implementación de este proyecto fueron

En esta etapa del proyecto seguimos el esquema de las 2 primeras fases de análisis del compilador las cuales son el análisis léxico y sintáctico y se hizo por medio de este esquema que representa la entrada de caracteres que recibe el analizador y la salida de tokens que luego analizara el analizador sintáctico siempre utilizando la tabla de símbolos en las dos fases como medida de prevención de errores y de uso posterior



Jflex

JFlex es una herramienta de generación de analizadores léxicos (scanners) escritos en Java. Un analizador léxico es una parte importante de un compilador, ya que su función es leer el código fuente y separar los distintos elementos léxicos, como palabras clave, identificadores, números y símbolos, para que el compilador pueda entenderlos y convertirlos en un programa ejecutable.

JFlex se encarga de generar un analizador léxico a partir de una especificación en un archivo de texto que describe los patrones de caracteres que deben ser reconocidos y las acciones que deben ser ejecutadas cuando se encuentran esos patrones. La especificación se realiza utilizando expresiones regulares, que permiten definir patrones de búsqueda complejos y flexibles.

```
//package e importaciones
%%
//configuraciones para el análisis
%%
//reglas lexicas
```

Cup

CUP (Constructor of Useful Parsers) es una herramienta de generación de analizadores sintácticos (parsers) para Java. Un analizador sintáctico es una parte importante de un compilador, ya que su función es analizar la estructura del código fuente y verificar si cumple con la sintaxis definida en el lenguaje de programación.

CUP se encarga de generar un analizador sintáctico a partir de una especificación en un archivo de texto que describe las reglas gramaticales del lenguaje. La especificación se realiza utilizando una notación llamada "gramática libre de contexto", que permite definir la estructura del lenguaje de programación mediante la definición de reglas gramaticales.

La herramienta CUP utiliza el algoritmo de análisis sintáctico LR (left-to-right, rightmost derivation), que es capaz de procesar gramáticas más complejas y ambigüedades en la definición de la estructura del lenguaje.

```
//package e importaciones

parser code
{:
:}
//terminales
terminal String ENTERO;
//no terminales
non terminal instrucciones;
//precedencias
precedence left MAS,MENOS;
//producción de inicio
start with ini;
//producciones
ini::=instrucciones;
```

Flujo de Caracteres

```
<!
     4R(H1V0 D3 PRU3B4 M3D10
!>
{

//                        ----DEFINIENDO CONJUNTOS----
CONJ: mayus - > A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z;
CONJ: minus -     > a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z;
CONJ: letra -  > a~z;
CONJ: digito -  > 0~9;

//                        ----DEFINIENDO EXPRESIONES----
frase -> ."C"."O"."M"."P"."I"."1" ? + | | {letra} {digito} " ";
cadena  -   > . \' . + | | | | \n {minus} {mayus} {digito} " " \';
%%
%%
cadena : "\'cadena entre comilla simple\'"; //bueno
frase : "COMPI1 sale con 100"; // bueno

}

<!
     12·$333544·%&//
          ·····NO TE DESANIMES ESTO SALE CON 100
     12·$333544·%&//
!>
```

Este será un flujo de caracteres estándar y convertirá en tokens mediante la siguiente tabla que reconoce ciertos patrones y mediante el análisis léxico del archivo de jflex los convertirá en tokens que luego el parser se encargara de convertir en un AST o árbol de análisis sintáctico

Tabla de Tokens

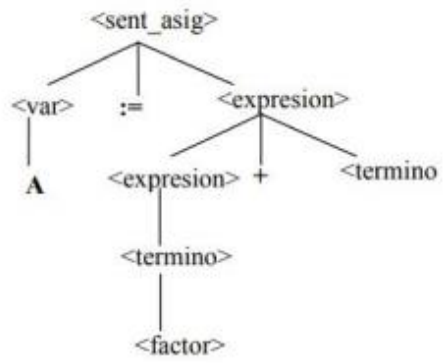
Terminal	Patron
LETRA	[a-zA-Z]
CONJUNTO	“CONJ”
IDENTIFICADOR	[a-zA-Z_]+([a-zA-Z_]+|[0-9]+)*
DIGITO	[0-9]
PTCOMA	“;”
DOSPTO	“:”
VIRG	“~”
BRAIZQ	“{”
BRADER	“}”
ASIGN	“-”[\r\t]*”>”
PERCEN	“%%”(\n”%%”)?
CONCAT	“.”
DIS	“|”
KLE	“*”
MAS	“+”
INTER	“?”
COMA	“,”
SPECHAR	"\\""n" | "\\""\"" | "\\""\""
ASCII	[!-/] | [:-@] | [\[-`] | [\{-\}]
CADENA	"\"" [^"\\\""]* "\\\""* [^\"]*[^"\\\""]* "\\\""* "\""
COMENTARIO_S	"//" [^\r\n]* [\r|\n|\r\n]?
COMENTARIO_M	"<!" [^!]* "!>"
BLANCOS	[\r\t]+

Gramática Libre de Contexto

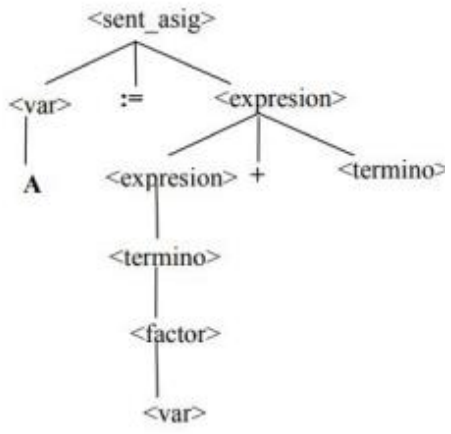
```
<sent_asig> ::= <var> = <expresion>

<expresion> ::= <expresion> + <termino>
              | <expresion> - <termino>
              | <termino>

<termino> ::= <termino> * <factor>
            | <termino> / <factor>
            | <factor>

<factor> ::= ( <expresion> )
           | <var>
           | <num>

<var> ::= [a-zA-Z]

<num> ::= [0-9]
```

Árbol de Análisis sintáctico de una asignación A=A+B

5)



6)



Método del árbol

Una vez Reconocidos los Tokens y convertidos se utilizo el método del árbol que construye un AFD a partir de una expresión regular

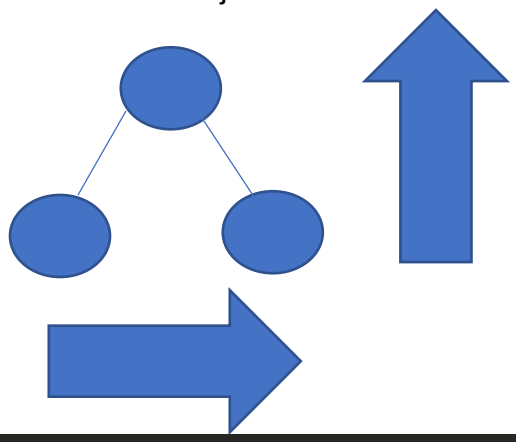
1. Agregar símbolo de finalización: Se le agrega un símbolo de finalización a la expresión regular

   Ejemplo:

   1*(0(0*(1|0)*))

   1*(0(0*(1|0)*))#

2. Construir árbol sintáctico: Se genera el árbol binario asociado a la expresión regular enumerando sus hojas. Como cup tiene una gramática LR o ascendente construimos el árbol de esta forma de izquierda a derecha de abajo a arriba



```
public class NodeArbol {
    public String token;
    public String lexema,Expresion,NombreExpresion;
    public int id;
    public NodeArbol hijoIzq;
    public NodeArbol hijoDer;
    public Boolean anulable=false;


    public ArrayList<NodeArbol> hijos = new ArrayList<NodeArbol>();
    public ArrayList<Integer> primeros = new ArrayList<Integer>();
    public ArrayList<Integer> ultimos = new ArrayList<Integer>();
```

Una clase Node árbol Modela el nodo con sus atributos siguientes primeros, data y enumeración en caso de ser una hoja. L a clase Árbol se encarga de almacenar todos estos nodos que luego recorreremos para hacer todas las operaciones necesarias para calcular los atributos

```java
public class Arbol {

    public NodeArbol raiz;
    public HashMap<Integer, List<Integer>> Siguientes = new HashMap<Integer, List<Integer>>();
    public ArrayList<NodeArbol> leaves = new ArrayList<NodeArbol>();
    public ArrayList<String> alfabeto = new ArrayList<String>();
    public ArrayList<T_Siguientes> t_Siguientes = new ArrayList<T_Siguientes>();
    public ArrayList<Estado> AFD = new ArrayList<Estado>();
    public Arbol(NodeArbol raiz) {
        this.raiz = raiz;
    }
```

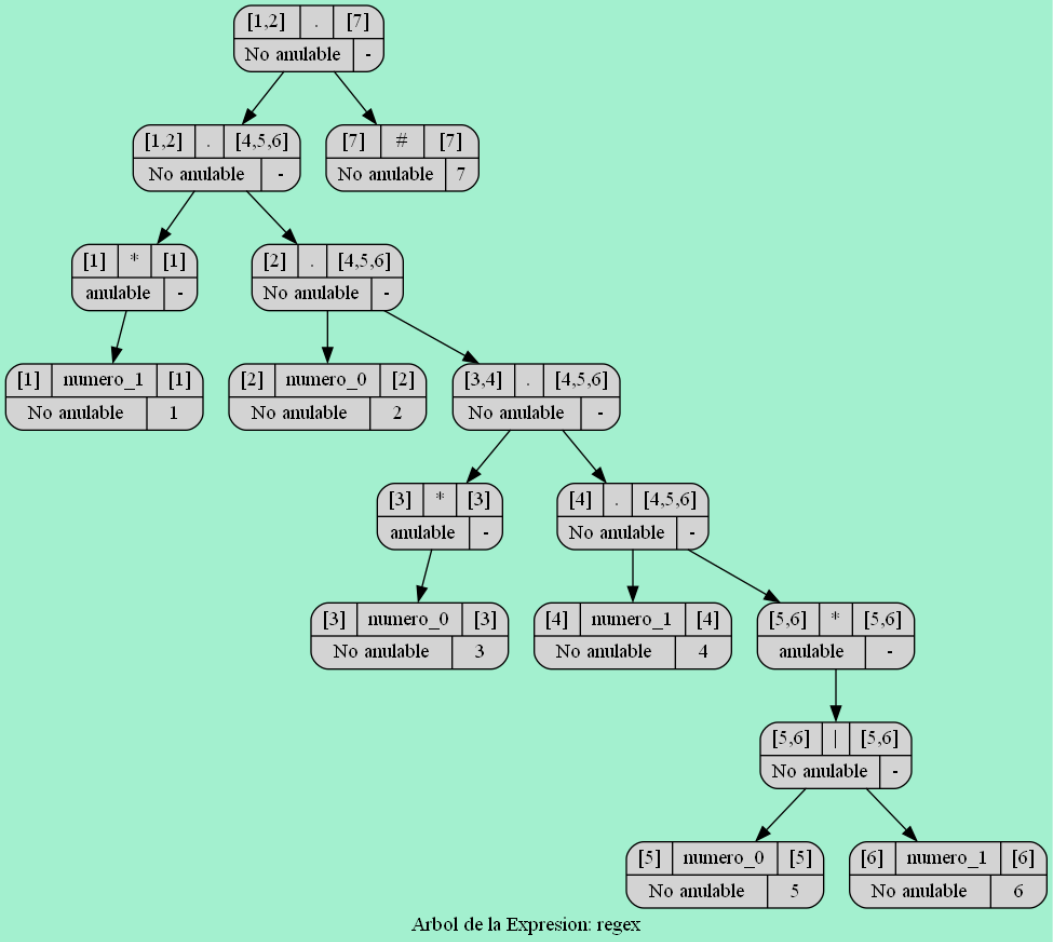
3. Calcular Anulables: se define que nodos son anulables o no, basado en reglas estipuladas.

NODO n	$anulable(n)$
Una hoja etiquetada como ϵ	**true**
Una hoja con la posición i	**false**
Un nodo-o $n = c_1 \mid c_2$	$anulable(c_1)$ **or** $anulable(c_2)$
Un nodo-concat $n = c_1 c_2$	$anulable(c_1)$ **and** $anulable(c_2)$
Un nodo-asterisco $n = c_1*$	**true**

4. Calcular Primeros y últimos Se calcula los primero de cada nodo basado en reglas estipuladas

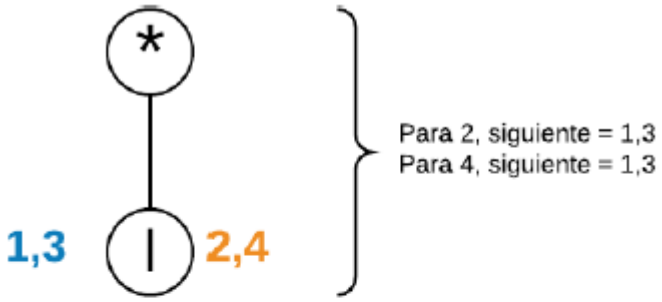
NODO n	$primerapos(n)$	ultimapos
Una hoja etiquetada como ϵ	$\emptyset$	-
Una hoja con la posición i	$\{i\}$	i
Un nodo-o $n = c_1 \mid c_2$	$primerapos(c_1) \cup primerapos(c_2)$	ultimapos(c1) U ultimapos(c2)
Un nodo-concat $n = c_1 c_2$	**if** $(anulable(c_1))$ $primerapos(c_1) \cup primerapos(c_2)$ **else** $primerapos(c_1)$	if anulable c2 ultimapos(c1) U ultimapos(c2) else ultimapos(c2)
Un nodo-asterisco $n = c_1*$	$primerapos(c_1)$	ultimapos(c1)

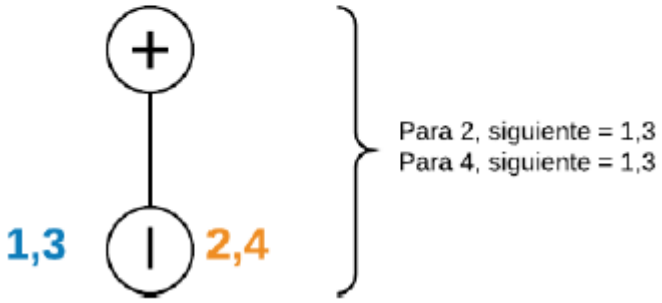
El árbol que se genera es el siguiente con todas sus características previamente calculadas se vería de la siguiente forma

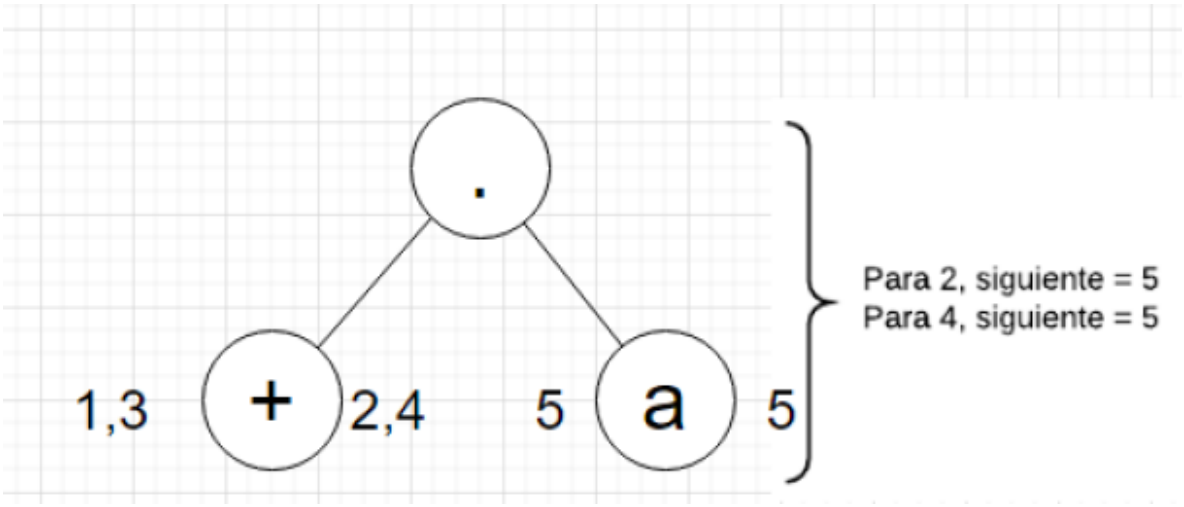


Arbol de la Expresion: regex

5. Calcular los siguientes: Se calcula los siguientes de cada hoja basado en reglas estipuladas. Solamente Concatenación (·), Cero o Muchas veces (*) y Una o muchas veces (+) tienen SiguienteEn la concatenación los Siguientes para los últimos de C1 son los primeros del Nodo C2







Simbolo	Nodo	Siguientes
numero_1	1	[1, 2]
numero_0	2	[3, 4]
numero_0	3	[3, 4]
numero_1	4	[5, 6, 7]
numero_0	5	[5, 6, 7]
numero_1	6	[5, 6, 7]
#	7	------

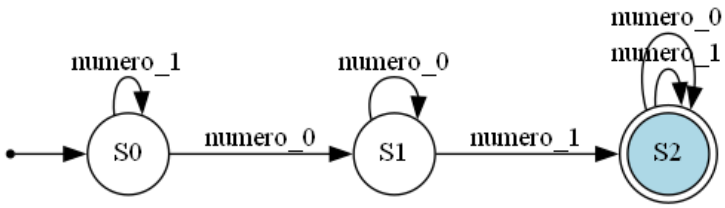
Tabla de Siguientes de la Expresion: regex

6. Tabla de transiciones: Se genera una tabla de transición de estados basados en reglas estipuladas. El estado inicial son los identificadores que estén en los primeros del nodo raíz Un estado de aceptación es todo aquel que tenga el identificador del símbolo # agregado inicialmente. Hay que calcular cerraduras

	Terminales			
Estado	numero_1	numero_0	#	
S0 [1, 2]	S0	S1	----	
S1 [3, 4]	S2	S1	----	
S2 [5, 6, 7]	S2	S2	----	Aceptacion

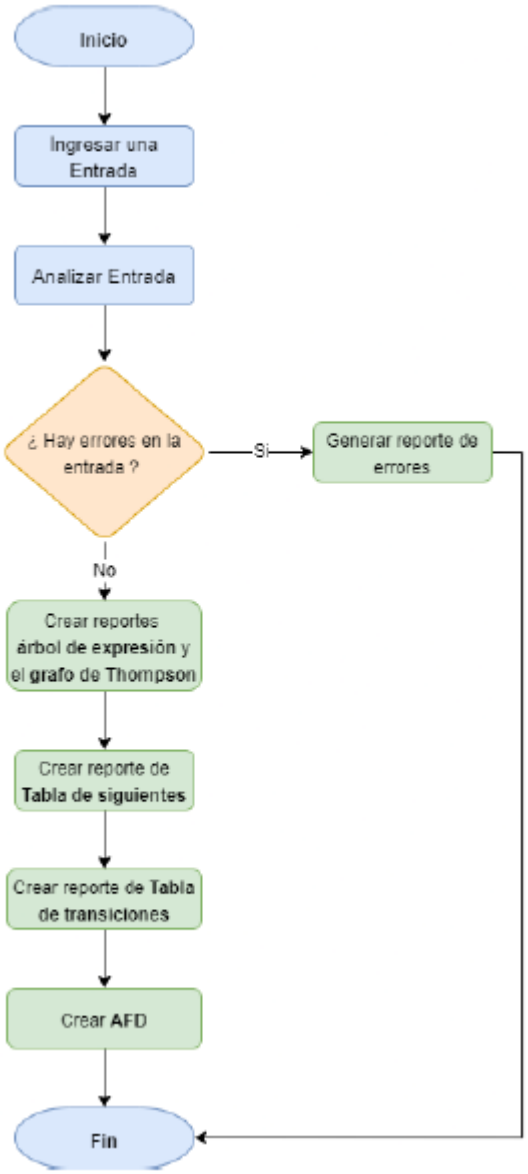
Transiciones de la expresion: regex

7. Graficamos el Grafo de transiciones



AFD de la expresion:regex

Se Acaba la construcción del autómata que verificara las cadenas en caso de que el flujo de caracteres este malo generara un tabla de errores léxicos sobre los caracteres que el lenguaje OLC no reconoce



Flujo General del programa los errores están modelados por medio de una clase de errores de la siguiente forma

```
public class ErrorLexico {
    public String Lexema;
    public String Descripcion="Caracter No Reconocido por el Lenguaje";
    public int Linea;
    public int Columna;
```

Método de Thompson

El método de Thompson es un algoritmo utilizado en teoría de lenguajes formales para convertir una expresión regular en un autómata finito no determinista (AFN) equivalente. Este proceso se conoce como construcción de Thompson.

El algoritmo de Thompson funciona de la siguiente manera:

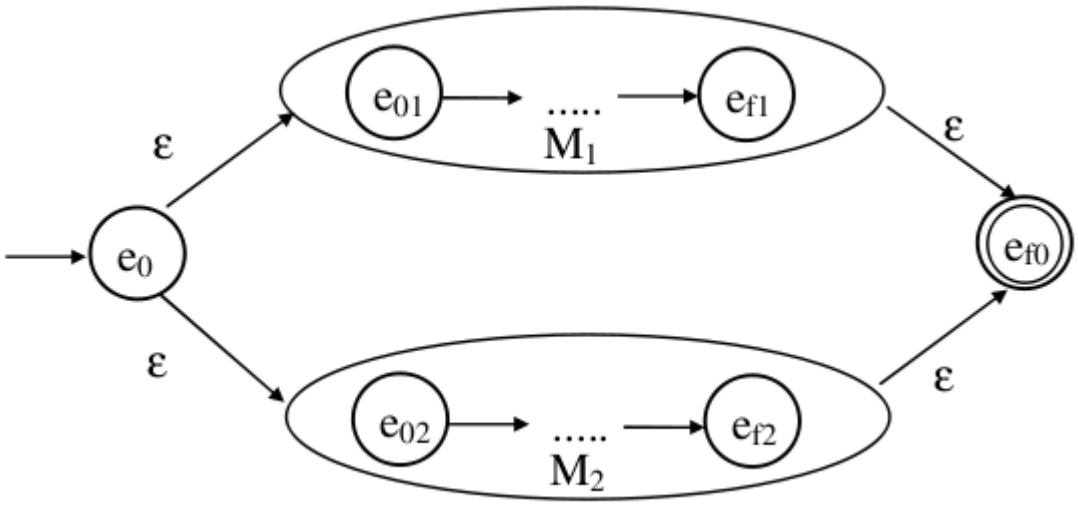
1. Se toma la expresión regular como entrada y se divide en subexpresiones más pequeñas, utilizando las operaciones básicas de concatenación, unión y clausura de Kleene.

2. Se crea un AFN para cada subexpresión utilizando las siguientes reglas:

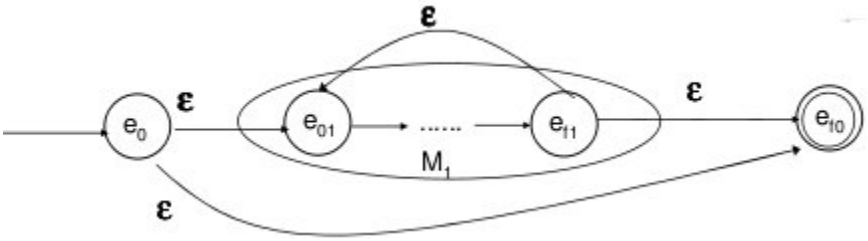
    - Para una letra o símbolo, se crea un AFN con dos estados, uno de inicio y otro final, conectados por una transición etiquetada con el símbolo.

    - Para la concatenación de dos subexpresiones, se conecta el estado final del primer AFN con el estado inicial del segundo AFN.
    - Para la unión de dos subexpresiones, se crea un nuevo estado inicial y dos transiciones, una desde el nuevo estado inicial a cada uno de los AFN originales.

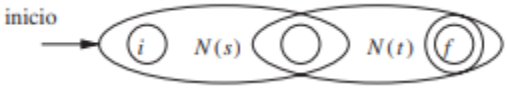
    - Para la clausura de Kleene de una subexpresión, se crea un nuevo estado inicial y final, y se conecta el estado final de la subexpresión con el nuevo estado inicial y final mediante transiciones epsilon. Además, se conecta el nuevo estado inicial con el estado inicial de la subexpresión y el estado final de la subexpresión con el nuevo estado final.



OPERADOR OR ”|”

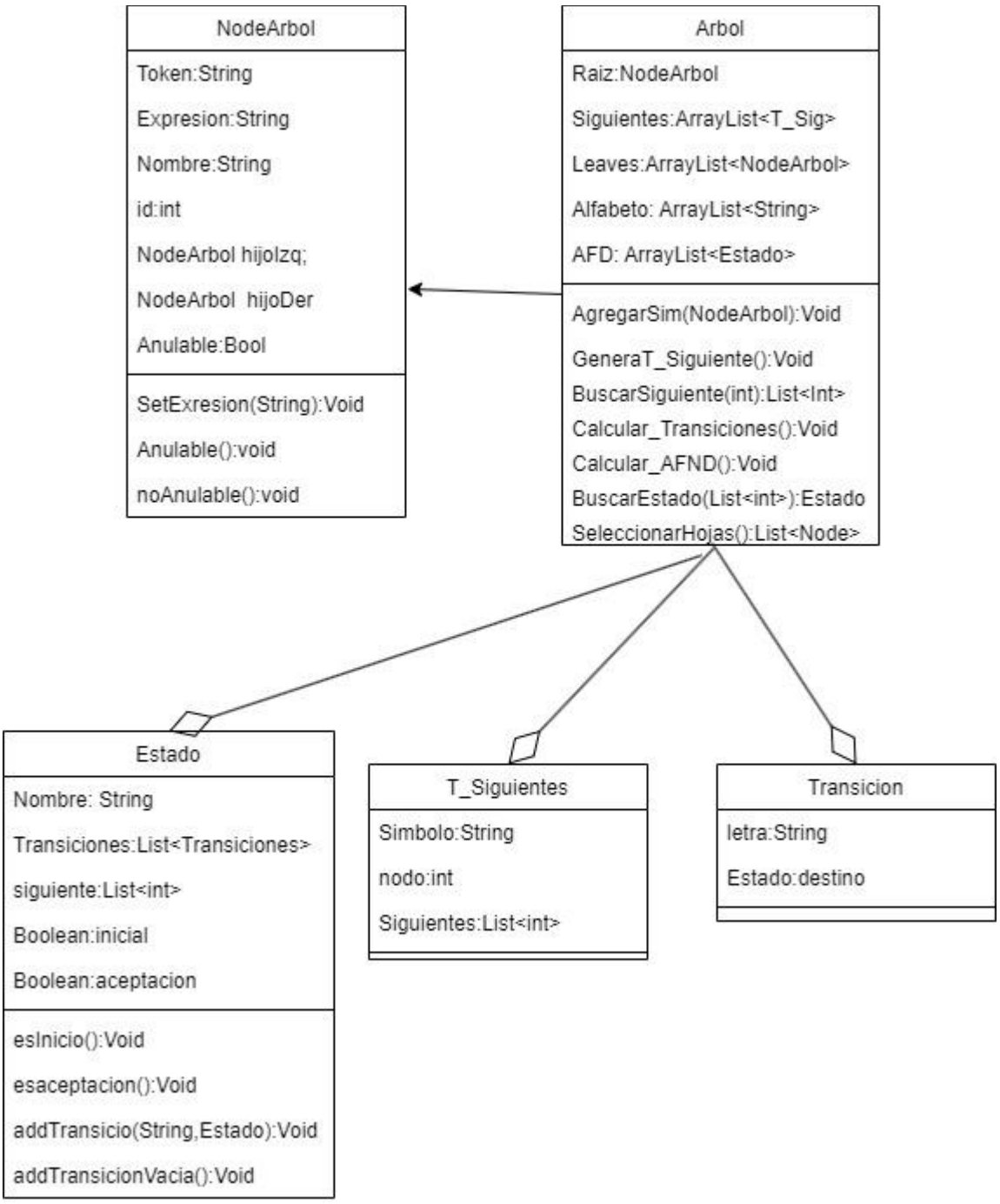


OPERADOR OR ”*”



OPERADOR OR ”.”

Diagrama de Clases



Bibliografías

- Arias Guerra, D. (2008) Estructura de datos Avanzadas (Revisado, ed., Vol. 9). Universidad de las Ciencias Informáticas. https://cutt.ly/eWdkzvt

- Ferris Castell, R. (2004) Algoritmos y Estructura de Datos I (Revisado ed., Vol. 1) Universidad de Valencia http://informatica.uv.es/iiguia/AED/oldwww/2001_02/Teoria/Tema_10.pdf

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson/Addison Wesley.

- Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers.