ANÁLISIS LÉXICO

José Sánchez Juárez

Septiembre, 2018

II

Índice general

Índice de figuras

Índice de cuadros

Analizadores léxicos

No me dan pena los burgueses
vencidos. Y cuando pienso que
van a darme pena, aprieto bien
los dientes y cierro bien los ojos.
Pienso en mis largos días sin
zapatos ni rosas. Pienso en mis
largos días sin sombrero ni nubes.
Pienso en mis largos días sin
camisa ni sueños. Pienso en mis
largos días con mi piel prohibida.
Pienso en mis largos días ...
En fin, que todo lo recuerdo. Y
como todo lo recuerdo,
¿qué carajo me pide usted que
haga? Pero además, pregúnteles.
Estoy seguro de que también
recuerdan ellos.

Nicolás Guillén.
1902−1889

El analizador léxico es también llamado escaner, su función es reconocer una cadena de caracteres para identificar las palabras del lenguaje en estudio. La forma de ir reconociendo el carácter que forma a la palabra, se hace acoplando la cadena de caracteres en estudio a una gramática por medio de un autómata. Al final el autómata reconoce si la cadena de caracteres es una palabra válida. Las palabras que debe reconocer el escaner, son: los números, los identificadores, las palabras reservadas, los operadores y los delimitadores.

Definición de clases léxicas por medio de expresiones regulares

Todo lenguaje de programación se expresa por medio de cinco tokens, los cuales están agrupados en clases léxicas. Las clases léxicas son los tipos de palabras que se usan en programación, tales son: los números, los identificadores, las palabras reservadas, los operadores y los delimitadores. Estas clases léxicas, también llamadas tokens, se pueden expresar por medio de expresiones regulares.

Clases léxicas

Todo lenguaje tiene tipos de palabras que ayudan a expresar las ideas. El lenguaje de programación también tiene tipos de palabras, que permiten expresar las ideas por medio de algoritmos. Los tipos de palabras que tiene cada lenguaje de programación se pueden clasificar, en: números, identificadores, palabras clave, operadores y delimitadores.

Se muestra en el siguiente listado ejemplos de tokens:

1. **Números**: 1, 2, 3, 4, 0.5, 0.6, 0.7, etc.
2. **Identificadores**: a, b, c, d, a1, b2, A, A1, etc.
3. **Palabras clave**: int, float, main, new, etc.
4. **Operadores**: +, -, *, /, etc.
5. **Delimitadores**: , , [,], etc.

Expresiones regulares

El término **letra** y **carácter** se usará como sinónimo de **símbolo** para denotar un elemento de un **alfabeto**. Si se pone una secuencia de símbolos lado a lado, se tiene una **cadena de símbolos**. Por ejemplo, 01011 es una cadena del alfabeto binario {0, 1 }. El término **sentencia** y **palabra** son frecuentemente usadas como sinónimos de **cadena** [**?**] .

Se formaliza la idea de una gramática y cómo se usa. Para este propósito, sea V_T un conjunto finito de símbolos no vacío llamado alfabeto terminal. Los símbolos en V_T se llaman símbolos terminales. El metalenguaje que se usa para generar cadenas en el lenguaje se supone que contiene un conjunto de clases sintácticas o variables llamadas símbolos no terminales. El conjunto de símbolos no terminales es denotado por V_N y los elementos de V_N se utilizan para definir la sintaxis (estructura) del lenguaje. Además, se supone que los conjuntos V_N y V_T son disjuntos [7].

DEFINICIÓN 1 (Vocabulario.) *Es el conjunto $V_N \cup V_T$ que consiste en símbolos no terminales y terminales [7].*

DEFINICIÓN 2 (Lenguaje.) *Es un conjunto de cadenas de longitud finita sobre algún alfabeto finito Σ [**?**] .*

La expresión regular se define como la representación de las palabras por medio de cadenas de símbolos, donde se utiliza la notación algebraica, la cual es reducida y fácil de asimilar. Esta representación se hace por medio de tres operaciones algebraicas de los símbolos: la concatenación, la concatenación de los caracteres de cero a más veces (cerradura) y la alternativa.

DEFINICIÓN 3 (Conjunto regular.) *Sea Σ un alfabeto finito. Un conjunto regular es recursivo sobre el alfabeto Σ de la siguiente manera:*

1. *Φ es una conjunto regular sobre Σ.*
2. *$\{e\}$ es una conjunto regular sobre Σ.*
3. *$\{a\}$ es una conjunto regular sobre Σ..*
4. *Si P y Q son conjuntos regulares sobre Σ, entonces:*
   a) *$P \cup Q$.*
   b) *PQ.*
   c) *P^*.*
5. *Nada también es un conjunto regular.*

[?] .

DEFINICIÓN 4 (Expresión regular.) *Sea Σ un alfabeto finito. Una expresión regular es una definición recursiva, como se presenta en el siguiente listado:*

1. *Φ es una expresión regular que denota el conjunto regular $\{\Phi\}$.*
2. *e es una expresión regular que denota el conjunto regular $\{e\}$.*
3. *$a \in \Sigma$ es una expresión regular que denota el conjunto regular $\{a\}$.*
4. *Si p y q son expresiones regulares que denota el conjunto regular P y Q respectivamente, entonces :*
   a) *$(p + q)$ es una expresión regular que denota $P \cup Q$.*
   b) *(pq) es una expresión regular que denota PQ.*
   c) *$(p)^*$ es una expresión regular que denota P^*.*
5. *Nada también es una expresión regular.*

[?] .

La concatenación

La concatenación es la consecución de símbolos, uno después de otro. Así que la concatenación de los símbolos a y b se expresa, como:

$$ab$$

Sean R y S dos cadenas de símbolos, la concatenación de las dos cadenas se denota como: RS. Se define por medio de la notación de conjuntos, como a continuación se muestra: $\{xy|x \in R \ \wedge \ Y \in S\}$.

Un solo símbolo es por si solo una expresión regular, ya que es la concatenación del símbolo con la cadena vacía una o más veces. Sea ϵ la representación de la cadena vacía. El símbolo a, se expresa como la concatenación de la cadena vacía una o más veces con el símbolo, de la siguiente manera:

$$a = a\epsilon = a\epsilon\epsilon = a\epsilon\epsilon\epsilon\cdots$$

Por lo que, una cadena con un sólo símbolo, es en sí, una expresión regular. Reafirmando: a, b, ϵ, son expresiones regulares.

Las cerraduras

La concatenación de símbolos cero o más veces hasta el infinito se hace por medio de las cerraduras de Kleene y la concatenación de una o más veces hasta el infinito se hace por medio de la cerradura positiva. La cerradura de Kleene o transitiva se representa de la siguiente manera:

$$\bigcup_{i=0}^{\infty} a^i = a^*$$

La cerradura positiva se representa como sigue:

$$\bigcup_{i=1}^{\infty} a^i = a^+$$

La alternativa

Cuando se quiere reconocer un símbolo u otro símbolo, la notación algebraica que lo determina, es la siguiente:

$$a|b$$

Esto quiere decir que se lee a o se lee b. Sean R y S dos cadenas de símbolos, la alternativa de dos cadenas se define como $\{x|x \in R \vee x \in S\}$. Su representación, es:

$$R|S$$

Así que observemos un fragmento de código en un lenguaje de programación. En el lenguaje C++ o en Java:

Listing 1: Código en Java.

```
1 Public class MinTest
2 {
3   public static void main(string[] args)
4   {
```

```
5       int a=3;
6       int b=7;
7
8       System.out.println(min(a,b));
9      }
10
11     {
12      return x<y?x:y;
13     }
14
15 }
```

Del código en Java del listado 1 se extraen los tokens, tales como las palabras reservadas: int, return, public, estatic, etc. Los identificadores: a, b, x, etc. Los números, 3, 7, etc. Los operadores: <, ?. Los delimitadores: (,), ;, etc.

Se muestran ejemplos de expresiones regulares en la siguiente lista:

1. **Números**: 1, 2, 3, 4, 0.5, 0.6, 7, etc.
2. **Identificadores**: $l(l|d)^*$
3. **Palabras clave**: int, float, main, new, etc.
4. **Operadores**: +, -, *, /, etc.
5. **Delimitadores**: , , [,], etc.

Análisis Léxico

Cuando el compilador analiza un programa escrito en un lenguaje de alto nivel. El primer paso es reconocer las palabras contenidas en el programa, de acuerdo a las clases léxicas [3]. Las palabras se van leyendo símbolo por símbolo y al agrupar todos los caracteres, se forman las palabras, las cuales se comparan con un conjunto de reglas, que se representan por medio de expresiones regulares o por medio de gramáticas. Si la palabra es válida, de acuerdo al lenguaje de programación usado acoplado con las reglas de una expresión regular o una gramática, el escaner asigna una clase léxica o categoría sintáctica, o la reconoce como parte del programa.

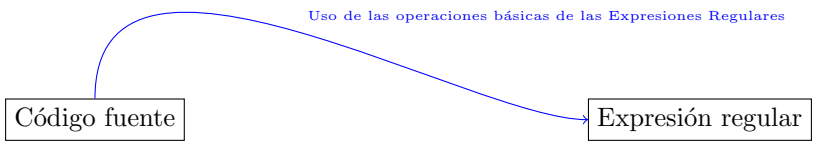

Figura 1: Conversión de código fuente a expresión regular.

de esta manera se sabe si en el programa las palabras estan bien escritas

Existen herramientas automáticas para generar escaners. El proceso de la herramienta es una descripción matemática de la sintaxis léxica del lenguaje.

Para construir los escaners artesanales y los escaneres generados, se aplican las mismas técnicas en ambos. Los compiladores comerciales y los compiladores de código abierto usan escaneres artesanales [5]. Los escaners artesanales son más rápidos que los escaners generados, porque la implementación optimiza una porción del encabezado que no se puede evitar en un escaner generado.

El modelo de reconocedor, es un programa que identifica las palabras de una cadena de caracteres.

Reconocedor de palabras, la explicación más simple de un algoritmo para reconocer palabras es reconocer carácter por carácter. Por ejemplo reconocer la palabra clave new. Considerando la rutina SiguienteCaracter() que regresa el siguiente carácter, se implementa como sigue:

Algorithm 0.1: SiguienteCaracter(), donde p es una lista enlazada simple, que contiene la palabra a analizar.

```
1  TIPE c:char ;
2  TIPE ↑ p:Lista ;
3  while p ↑.sig <> NULL do
4  |  c := p ↑ .dato ;
5  |  ↑ p := p ↑.sig ;
6  |  return c ;
```

Algorithm 0.2: Lectura de la palabra new.

```
1  c ← SiguienteCaracter() ;
2  if c == 'n' then
3  |  c ← SiguienteCaracter() ;
4  |  if c == 'e' then
5  |  |  c ← SiguienteCaracter() ;
6  |  |  if c == 'w' then
7  |  |  |  Correcto ;
8  |  else
9  |  error ;
10 else
11 error ;
```

Creación de AFN por medio de la construcción de Thompson

Las clases léxicas, se representan con las siguientes operaciones básicas, las cuales son: la concatenación, la alternativa, la cerradura de Kleene y la cerradura positiva. A partir de estas operaciones básicas se construye el autómata AFN, que va a reconocer a los tokens que forman el lenguaje de programación.

Autómata finito no determinístico (AFN)

El autómata finito no determinista se abrevia como AFN. Este autómata se define como aquel que tiene transiciones con la cadena vacía ϵ y que tiene transiciones de un estado con el mismo símbolo a dos estados diferentes. Y el número de los estados que componen al AFN son finitos.

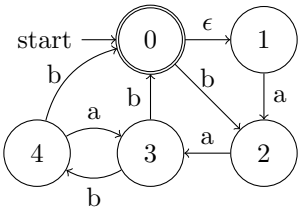

Figura 2: Ejemplo 1

El autómata de la figura 2, tiene una transición del estado 0 al estado 1 con la cadena vacía ϵ. Además tiene dos transiciones con el símbolo b del estado 3 a los estados 0 y 4. Por lo que este autómata es AFN.

Autómata finito determinístico (AFD)

El autómata finito determinístico se abrevia como AFD. Este autómata no debe tener transiciones con la cadena vacía ϵ, también no debe tener transiciones de un estado con un mismo símbolo a dos estados diferentes. Y el número de estados debe ser finito.

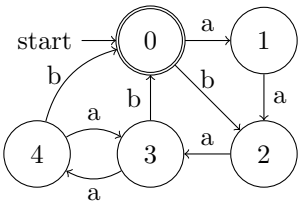

Figura 3: Ejemplo 2

El autómata de la figura 3 es AFD, ya que no tiene transiciones ϵ y tampoco tiene transiciones de un estado con el mismo símbolo a dos estados diferentes.

Construcción de Thompson

Tomando en cuenta las operaciones básicas de las expresiones regulares, se plantea la construcción del AFN: para la concatenación ab, se representa por el grafo de la figura 4 [5].

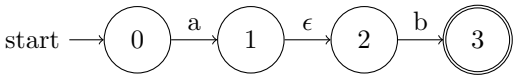

Figura 4: Construcción de Thompson para la concatenación de ab.

La construcción de thompson para la cerradura de Kleene, en notación algebraica es a^*, el grafo es el de la figura 5 [5]:

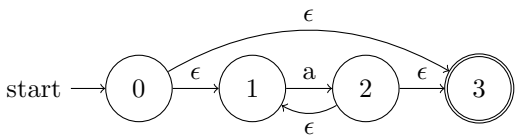

Figura 5: Construcción de Thompson para la cerradura de Kleene a^*.

La construcción de thompson para la cerradura de positiva, en notación algebraica es a^+, el grafo es el de la figura 6 [4]:

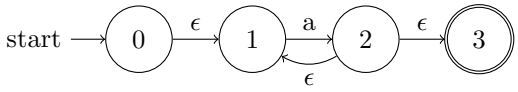

Figura 6: Construcción de Thompson para la cerradura de positiva a^+.

La construcción de Thompson para la alternativa, en notación algebraica es $a|b$, el grafo es el de la figura 7 [5]:

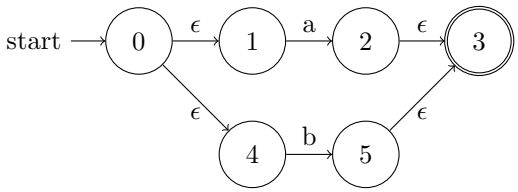

Figura 7: Construcción de Thompson para la alternativa $a|b$.

EJEMPLO 1 (Construcción de Thompson.) *La entrada del analizador léxico es el código fuente y la salida son los tokens reconocidos en el código fuente de la entrada. Así que para reconocer los tokens, se hace de* ***dos formas****: por medio de las expresiones regulares o por medio de los autómatas.*

Si tenemos la ***palabra reservada if****, la expresión regular de esta palabra reservada es* ***Itálica{if}****.*

El AFN de la expresión regular **Itálica{if}**, es el que se muestra en la figura 11 , el cuál representa la concatenación del símbolo **i** con el símbolo **f**.

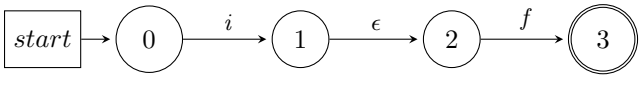

Figura 8: AFN para la expresión regular Itálica{if

.

EJEMPLO 2 (Construcción de Thompson.) *Ahora la entrada del analizador léxico es una alternativa y la salida son los tokens reconocidos en el código fuente de la entrada. Así que para reconocer los tokens, se hace de* ***dos formas****: por medio de las expresiones regulares o por medio de los autómatas.*

Si tenemos la expresión regular ***Itálica{(a | b)}****.*

Su AFN es el que se muestra en la figura 9 .

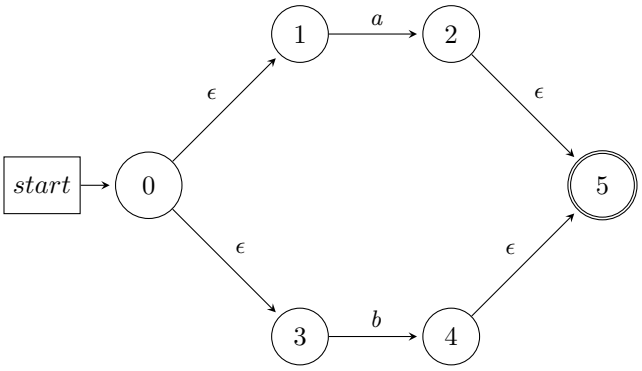

Figura 9: AFN para la expresión regular Itálica$\{a|b\}$.

EJEMPLO 3 (Construcción de Thompson.) *Ahora la entrada del analizador léxico es un identificador.*

La expresión regular del identificador es ***Itálica{****$l(l|d)^*$***}****.*

Su AFN es el que se muestra en la figura 10 .

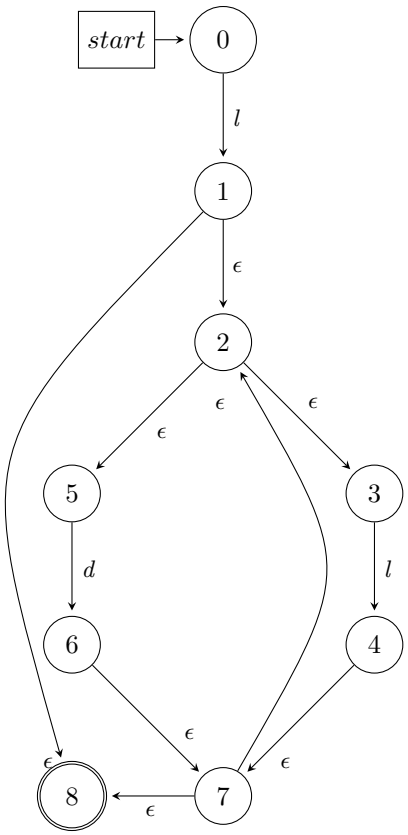

Figura 10: AFN obtenido de los patrones de la construcción de Thompson de la expresión regular **Itálica{**$l(l|d)^*$**}**.

Conversión de AFN a AFD

Existe un AFD equivalente a un AFN con transiciones ϵ, el que se puede obtener de la siguiente manera:

Sea M=(Q, Σ, δ, q_0, F) un AFN con transiciones ϵ. Un AFD equivalente a un AFN será como sigue:

M'=(Q_1, Σ, δ_1, q_1, F_1), donde Q_1 es subconjunto de 2^Q, lo que quiere decir que cada estado de un AFD corresponde a un subconjunto de Q.

Para cada subconjunto de estados de M habrá un solo estado de M'. Donde M' simulará el comportamiento del autómata M, haciendo transiciones entre sus estados de la forma que M hace las transiciones entre sus subconjuntos.

DEFINICIÓN 5 (Cerradura-ϵ.) *La cerradura-ϵ($\{S\}$) es el conjunto de todos los estados que reciben una transición con el carácter vacío ϵ, incluido el conjunto de estados $\{S\}$ de donde salen las transiciones ϵ.*

El primer estado q_1 = cerradura-ϵ($\{q_0\}$), es el estado inicial del AFD. Se agregará q_1 a Q_1, y luego se encuentra la transición de q_1 de la siguiente manera:

δ (q_1, a) = cerradura-ϵ (δ (representación del subconjunto q_1, a)).

Si esta transición genera un nuevo subconjunto de Q, entonces este será agregado a Q_1; y la siguiente transición de este se deberá encontrar, se continuará de esta manera hasta que no se pueda agregar un nuevo estado a Q_1.

Después se identificará los estados del AFD que contengan al menos un estado que pertenezca al conjunto de estados F. Si la cerradura-ϵ($\{q_0\}$) no contiene un miembro de F y el conjunto de los estados del AFD constituye F_1, pero si la cerradura-ϵ($\{q_0\}$) contiene un miembro de F.

De [3] la transformación de AFN a AFD por medio de subconjuntos, se hace de la siguiente manera.

Aplicando los siguientes conceptos:

Cerradura-ϵ($\{s\}$), es el conjunto de estados del AFN alcanzables desde el estado s del AFN con transiciones ϵ solamente.

Cerradura-ϵ($\{T\}$), es el conjunto de estados del AFN alcanzables desde el estado s en el conjunto T con transiciones ϵ solamente; $\cup_{s \in T}\ cerradura - \epsilon(s)$.

mover(T, a), es el conjunto de estados del AFN a los cuales hay una transición con un símbolo de entrada a desde algún estado s en T.

Se exploran los conjuntos de estados de N que pueden ver después las cadenas de entrada. Como base, antes de leer el primer símbolo de entrada, N puede estar en cualquiera de los estados de la cerradura-ϵ($\{s_0\}$), donde s_0 es el estado inicial. Se supone que N puede estar en el conjunto de estados T después de leer la cadena de entrada x. Si hay un lectura a, entonces N puede inmediatamente ir a cualquiera de los estados mover(T, a). Sin embargo después de leer a este puede hacer algunas transiciones ϵ; así que N podría estar en algunos de los estados cerradura-ϵ((mover(T, a)) después de leer la entrada xa. Siguiendo estas ideas, la construcción del conjunto de estados D's, $D_{estados}$, y su función de transición

$D_{transicion}$ se muestra en el siguiente algoritmo:

Algorithm 0.3: Transiciones

```
1 while Hay un estado no marcado T en D_estados do
2     Marcar T ;
3     for Para cada símbolo de entrada a do
4         U := cerradura-ϵ(mover(T, a)) ;
5         if U no esta en D_estados then
6             Agregar U como un estado no marcado en D_estados ;
7         D_transicion[T, a] := U ;
```

El estado inicial de D es cerradura-$\epsilon(\{s_0\})$, y el estado de aceptación de D son todos los conjuntos de estados N's que incluyen al menos un estado de aceptación N.

La cerradura se calcula con el siguiente algoritmo:

Algorithm 0.4: Algoritmo para la cerradura

```
1 Poner todos los estados de T en una pila ;
2 inicializar cerradura-ϵ(T) a T ;
3 while pila <> NULL do
4     pop(t) ;
5     for Cada estado u con un arco de t a u etiquetado con ϵ do
6         if No esta en cerradura-ϵ({T)} then
7             Agregar u a cerradura-ϵ({T}) ;
8             push(u, pila) ;
```

Algoritmo de subconjuntos

La transformación de un AFN a un AFD se hace aplicando el algoritmo de subconjuntos, el cual comienza al aplicar cerradura-ϵ al primer estado del AFN. El concepto de cerradura, es: al estado al que se le aplica la cerradura-ϵ se incluye en el conjunto y además todos los estados que se accedan por medio de un transición ϵ a partir del estado al que se le aplica la cerradura-ϵ. Se van formando subconjuntos de estados, los cuales también se van formado del movimiento de todos los caracteres que acepta el AFN desde el estado al que se obtuvo, así si los subconjuntos de estados son diferentes se marcan como un estado nuevo para el AFD.

DEFINICIÓN 6 (Movimiento(Estado, Símbolo).) *Es el movimiento de un estado marcado como* ***Estado*** *con cada símbolo del alfabeto* Σ *hacia otro estado. El movimiento se puede expresar también como* ***mov(estado, símbolo)****.*

Observar el siguiente ejemplo:

EJEMPLO 4 (Conversión de AFN a AFD.) *La siguiente expresión regular* ***Itálica{***$(a|b)^*ba$***}*** *transformarla en AFN, aplicando la construcción de Thompson. Para después aplicar el algoritmo de subconjuntos y así transformar el AFN a AFD.*

El AFN que se obteniene aplicando los patrones de la construcción de Thompson, es el que aparece en la figura 11 :

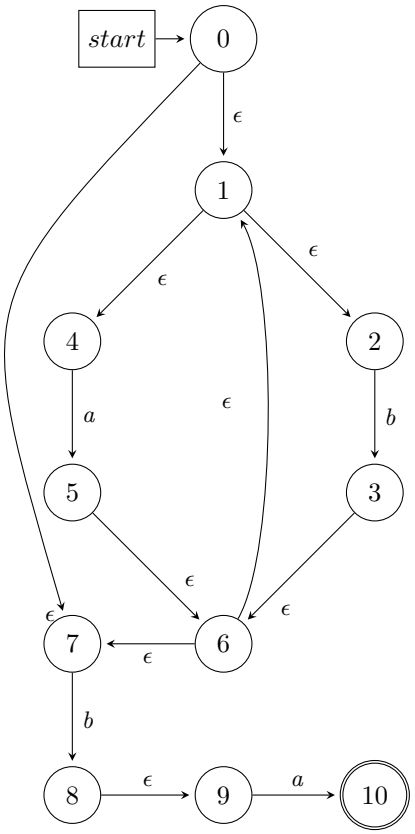

Figura 11: AFN obtenido de los patrones de la construcción de Thompson de la expresión regular **Itálica{**$(a|b)^*ba$**}**.

Para transformar el AFN de la figura 11 a AFD, se aplica el algoritmo de subconjuntos. Se comienza aplicando la operación de cerr-ϵ al estado 0, este primer subconjunto de estados será el primer estado del AFD, el que se etiquetará como S_0:

$$cerr - \epsilon(\{0\}) = \{0, 1, 2, 4, 7\} = S_0$$

Para obtener el estado S_1 se aplica la operación de movimiento por todo el alfabeto $\Sigma = \{a, b\}$, también a cada movimiento se le aplica la operación de cerr-ϵ, como se muestra a continuación:

$$mov(S_0, a) = \{5\}$$

Se le aplica la operación de cerr-ϵ al movimiento de S_0 con el símbolo a, por lo que se obtiene:

$$cerr - \epsilon(mov(S_0, a)) = \{1, 2, 4, 5, 6, 7\} = S_1$$

Se hacen las siguientes operaciones:

$$mov(S_0, b) = \{3, 8\}$$

$$cerr - \epsilon(mov(S_0, b)) = \{1, 2, 3, 4, 6, 7, 8, 9\} = S_2$$

Se hacen las siguientes operaciones:

$$mov(S_1, a) = \{5\}$$

$$cerr - \epsilon(mov(S_1, a)) = S_1$$

Se hacen las siguientes operaciones:

$$mov(S_1, b) = \{3, 8\}$$

$$cerr - \epsilon(mov(S_1, b)) = S_2$$

Se hacen las siguientes operaciones:

$$mov(S_2, a) = \{5, 10\}$$

$$cerr - \epsilon(mov(S_2, a)) = \{1, 2, 4, 5, 6, 7, 10\} = S_3$$

Se hacen las siguientes operaciones:

$$mov(S_2, b) = \{3, 8\}$$

$$cerr - \epsilon(mov(S_2, b)) = S_2$$

Se hacen las siguientes operaciones:

$$mov(S_3, a) = \{5\}$$

$$cerr - \epsilon(mov(S_3, a)) = S_1$$

Se hacen las siguientes operaciones:

$$mov(S_3, b) = \{3, 8\}$$

$$cerr - \epsilon(mov(S_3, b)) = S_2$$

El siguiente paso es obtener de las operaciones anteriores el AFD, como se muestra en la figura 12 .

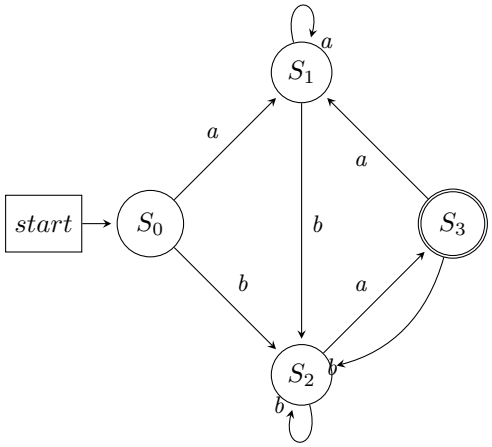

Figura 12: AFD que se obtiene de aplicar el algoritmo de subconjuntos al AFN de la figura 11 .

Creación de un AFD a partir de una expresión regular

El proceso de construcción de un AFD de una expresión regular, es: aplicar la construcción de Thompson a la expresión regular para obtener un autómata AFN, para después aplicar el algoritmo de subconjuntos al AFN para obtener un AFD no mínimo, como se muestra en la figura 13.

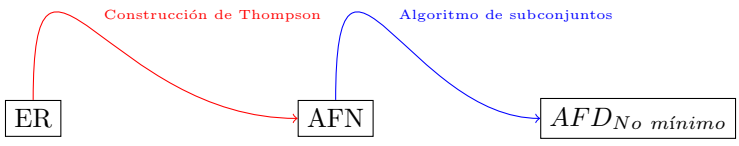

Figura 13: Conversión de una Expresión Regular a un Autómata AFD.

[Transformación de expresión regular a AFD.] Otro método es usar cuatro funciones: anulable, primerapos, ultimapos y siguientepos, haciendo recorridos sobre un árbol. Por último se construye el AFD a partir de siguientepos. Las funciones anulable, primerapos y ultimapos se definen sobre los nodos del árbol sintáctico y se usan para calcular siguientepos, que esta definida en el conjunto de posiciones [3].

DEFINICIÓN 7 (Función siguientepos().) *Es el conjunto de posiciones* ***j*** *tales que hay alguna cadena de entrada* ***cd*** *tal que* ***i*** *corresponde a la aparición de* ***c*** *y* ***j*** *a la aparición de d [3] .*

DEFINICIÓN 8 (Función primerapos().) *La que proporciona el conjunto de posiciones que pueden concordar con el primer símbolo de una cadena generada por la subexpresión con raíz en n [3] .*

DEFINICIÓN 9 (Función últimapos().) *La que proporciona el conjunto de posiciones que pueden concordar con el último símbolo en esa cadena [3] .*

DEFINICIÓN 10 (Función anulable().) *Es necesario conocer qué nodos son las raíces de las subexpresiones que generan lenguajes que incluyen la cadena vacía. A dichos nodos se les denomina anulables, y la función anulable(n) se define como verdadera si el nodo n es anulable, y falso en caso contrario [3] .*

DEFINICIÓN 11 (Construcción de un AFD de una expresión regular.)
Entrada:

Una expresión regular r.

Salida:

Un AFD ***D*** *que reconoce a L(r).*

Método:

1. *Constrúyase un árbol sintáctico para la expresión regular aumentada (r)#, dónde # es un marcador de final único que se añade a (r).*
2. *Constrúyanse las funciones anulable(), primerapos(), útimapos() y siguientepos() haciendo recorridos en profundidad en el árbol T.*
3. *Contrúyanse los* ***estadosD****, el conjunto de estados* ***D****, y* ***tranD****, la tabla de transiciones para* ***D****. Los estados dentro de* ***estadosD*** *son conjuntos de posiciones; al principio, cada estado esta "no marcado", y un estado se convierte en "marcado" justo antes de considerar sus transiciones de salida. El estado de inicio de* ***D*** *es primerapos(raíz), y los estados de aceptación son todos los que contienen la posición asociada con el marcador de final #.*

[3] .

Listing 2: Construcción de la tabla de transiciones del estado D.

```
1  al principio, el unico estado no marcado en estadosD es
2  primerapos(raiz), donde raiz es la raiz del arbol de
3  sintaxis para (r)#;
4
5  while hay un estado sin marcar T en estadosD do begin
6      marcar T;
7      for cada simbolo de entrada a do begin
8          sea U el conjunto de posiciones que estan en
9              siguientepos(p) para alguna posicion p en T,
10             tal
11             que el simbolo en la posicion p es a;
12         if U no esta vacio y no esta en estadosD then
13             aniadir U como estado no marcado a estadosD;
14         tranD[T, a] := U
15     end
16 end
```

Ejercicio

Convertir la expresión regular aumentada **Itálica {** $(a|b)^*abb\#$ **}** en un AFD, utilizando las funciones primerapos, últimapos, siguientepos. Lo primero es colocar en un árbol sintáctico la expresión regular. Existen los nodos: nodo-ast que es el nodo para una cerradura de Kleene, el nodo-cat que es el nodo para la concatenación, el nodo-o que es el nodo para la alternativa y también el nodo-pos que es el nodo para la cerradura positiva.

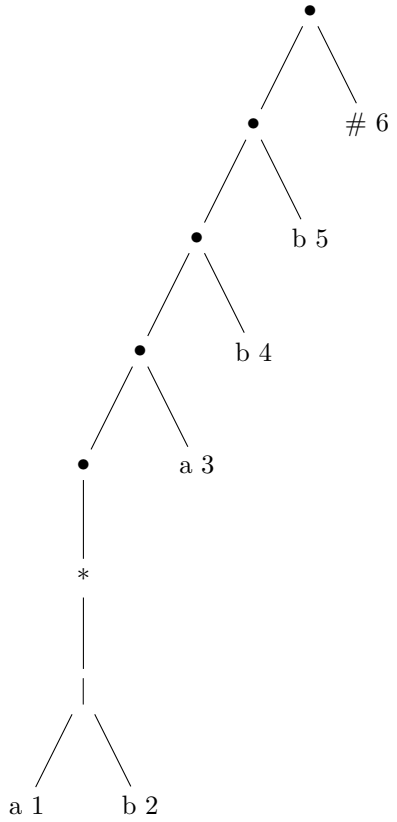

Figura 14: Árbol de análisis sintáctico aumentado.

En la expresión regular **Itálica{** $(a|b)^*abb$ **}** se enumeran los símbolos del alfabeto incluido el símbolo #, como se muestra en el cuadro 2 :

(	a	|	b	)	*	a	b	b	#
	1		2			3	4	5	6

Cuadro 1: Posiciones de los caracteres.

En el árbol se marcan los nodos en la parte izquierda del nodo se colocan los primeros y en la parte derecha del nodo se colocan los últimos.

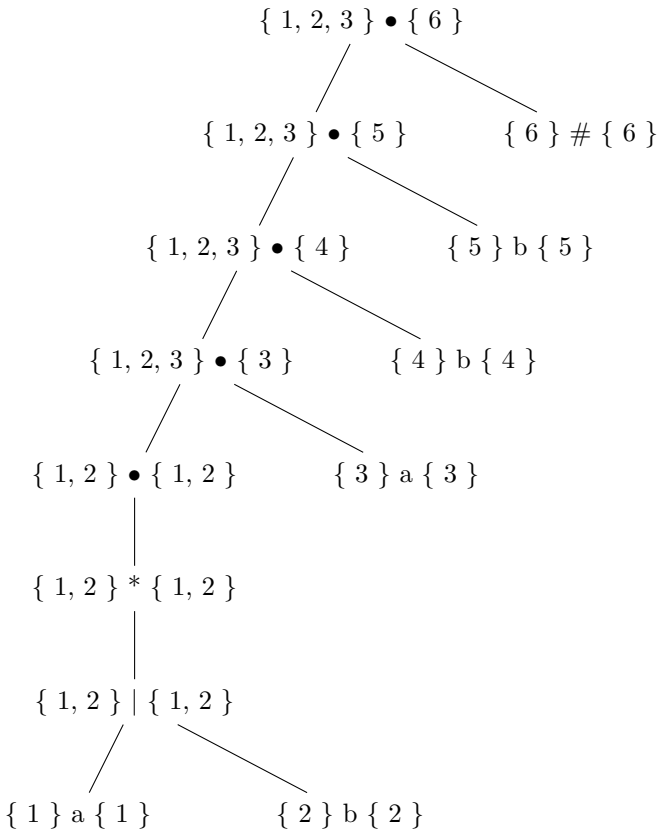

Figura 15: Las funciones primerapos y últimapos para los nodos del árbol de análisis sintáctico aumentado para la expresión regular **Itálica{** $(a|b)^*abb$ **}**.

Para calcular la función siguientepos se aplican las siguientes dos reglas:

1. Si n es un nodo-cat con hijo izquierdo c_1 e hijo derecho c_2, e i es una posición dentro de últimapos(c_1), entonces todas las posiciones de primerapos(c_2) están en siguientepos(i).

2. Si n es un nodo-ast, e i es una posición dentro de últimapos((n), entonces todas las posiciones de primerapos(n) están en siguientepos(i).

En el cuadro **??** se listan los conjuntos primeros y últimos:

	(	a	|	b	)	*	a	b	b	#
PRMERAPOS		1		2			3	4	5	6
ÚLTIMAPOS		1		2			3	4	5	6

Cuadro 2: Conjuntos primeros y últimos del árbol de la figura 15 .

Se obtiene el autómata AFD de la figura 2:

Minimización de estados de un AFD

Después de transformar un autómata AFN a AFD, es conveniente que el AFD tenga el mínimo de estados con la finalidad de que cuando se programe el AFD sea con un número de estados pequeño.

Para minimizar un AFD existen algoritmos de Minimización, los cuales, son: El algoritmo de Hopcroft [5], el algoritmo de consistencia [4].

El algoritmo de Hopcroft

Se forma un conjunto de particiones P del autómata a minimizar. La construcción del conjunto de particiones se basa en el principio de que un estado se comporta de la misma manera que otro estado, si tienen el mismo comportamiento ante la misma cadena de entrada, entonces los estados son equivalentes. Así es, se expresa de la siguiente manera. Ante la entrada de la cadena de caracteres *abc*, el comportamiento de un estado $p \rightarrow q$ es el mismo que el de otro estado $p_1 \rightarrow q_1$, entonces los estados q y q_1 son equivalentes. Representado gráficamente:

$$p \xrightarrow{abc} q$$

Se tiene que el estado q_1 tiene el mismo comportamiento que q, ante la misma cadena de entrada, lo que se representa de la siguiente forma:

$$p_1 \xrightarrow{abc} q_1$$

El algoritmo de consistencia

La manera de minimizar el número de estados es determinar las salidas de cada uno de los estados basados en que hacen de acuerdo a la cadena de caracteres que aceptan los estados.

Dado un AFD D que acepta el alfabeto Σ con estado S donde $F \subseteq S$ es el conjunto de los estados de aceptación, construimos un AFD D' donde cada estado es un grupo de estados de D. Los grupos en el AFD mínimo son consistentes: para cada par de estados s_1, s_2 en el mismo grupo G y cualquier símbolo c, mover(s_1, c) esta en el mismo grupo G_1 como mover(s_2, c) o ambos son indefinidos. En otras palabras, no podemos decir s_1 y s_2 aparte por buscar sus tradiciones.

Se forman dos grupos de estados: los estados finales y los estados no finales.

Se verifica la consistencia de los grupos formados. Si no son consistentes se divide el grupo, tomando los grupos de estados que si sean consistentes y se reemplaza el grupo que no fué consistente por los nuevos grupos formados.

Esto se repite hasta que todos los grupos sean consistentes.

Ejemplo

Se tiene el siguiente autómata AFD no mínimo: con el conjunto de estados $S = 0, 1, 2, 3, 4$, donde la tabla de transiciones se representa en el cuadro 3:

Estados	Símbolo	Símbolo
S	a	b
0	1	2
1	2	-
2	3	-
3	4	0
4	3	0

Cuadro 3: Tabla de transiciones de un AFD no mínimo.

El autómata AFD no mínimo de la tabla de transiciones 3, esta representado en la figura 16 : los estados finales $F = \{0\}$ y el conjunto de los estados no finales $G = \{1, 2, 3, 4\}$.

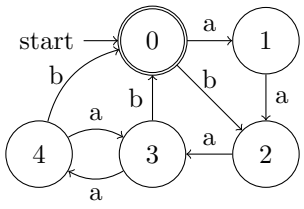

Figura 16: Autómata AFD no mínimo.

Para minimizar el autómata de la figura 16, se aplica el algoritmo de consistencia a los dos grupos formados: el grupo de estados no finales G y el grupo de estados finales F.

Se presentan los grupos G y F en forma de tabla, como se muestra en el cuadro 4 y en el cuadro 5, respectivamente:

Estados	Símbolo a	Símbolo b
1	2	-
2	3	-
3	4	0
4	3	0

Cuadro 4: Tabla de transiciones del grupo G.

Al aplicar la consistencia al grupo G representado en el cuadro 4 , se determina que todos los estados son inconsistentes, pues tienen diferentes transiciones. Por lo que cada estado se divide en un grupo diferente, de manera que se obtienen cuatro grupos.

Al aplicar la consistencia al grupo F representado en el cuadro 5 , y ya que tiene un sólo estado se considera que el grupo F es consistente, por lo que se obtiene un grupo.

Estados	Símbolo a	Símbolo b
0	1	2

Cuadro 5: Tabla de transiciones del grupo F.

En total se obtuvieron cinco grupos consistentes y ya que el AFD no mínimo tiene cinco estados. Entonces, se concluye que el AFD no mínimo es el AFD mínimo.

Lista de ejercicios

1.- Dado el autómata $AFD_{No\ mínimo}$ de la figura 17, minimizarlo aplicando los algoritmos de minimización:

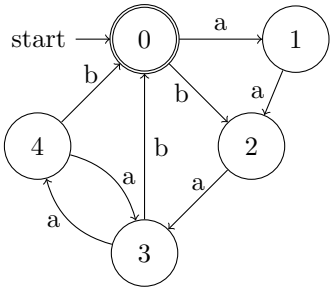

Figura 17: Autómata AFD no mínimo.

2.- Para el $AFD_{No\ mínimo}$ de la figura 18, minimizarlo aplicando los algoritmos de minimización:

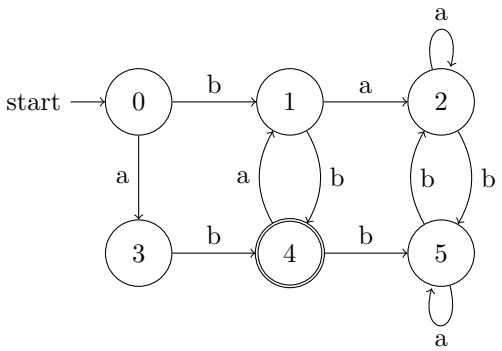

Figura 18: Autómata AFD no mínimo.

Construcción de analizadores léxicos definidos por medio de expresiones regulares

Para construir un analizador léxico se parte de la expresión regular, la cual se transforma en autómata AFN por medio de la construcción de Thompson. Después, el autómata AFN se convierte en autómata AFD utilizando el algoritmo de subconjuntos, posteriormente se minimiza el número de estados, utilizando un algoritmo de minimización. Cuando ya se tiene el AFD mínimo, este se programa en un lenguaje de programación de alto nivel, ya sea C++, Java, etc. La programación del AFD mínimo, es la implementación del analizador léxico.

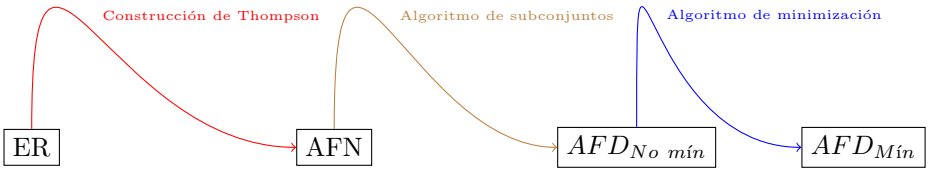

Figura 19: Pasos para la construcción de un analizador léxico desde una expresión regular.

Ejemplo

EJEMPLO 5 (Construcción de un analizador léxico.) *A partir de la siguiente expresión regular construir un analizador léxico:* ***la expresión regular*** $(a|b)^*$ ba***, se transforma en AFN aplicando la construcción de Thompson para tres concatenaciones****, la primera concatenación es una cerradura de Kleene y dentro de la cerradura de Kleene una alternativa, por lo que se obtiene el siguiente autómata:*

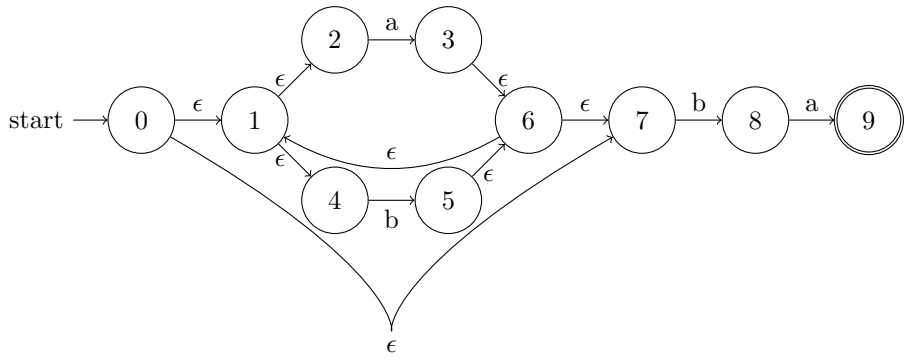

Figura 20: Autómata AFN obtenido de la expresión regular $(a|b)^*ba$.

El siguiente paso es convertir el autómata AFN en un autómata AFD aplicando el **algoritmo de subconjuntos**. De donde se obtiene el el autómata AFD no mínimo, que se muestra en seguida:

cerradura-$\epsilon(\{0\})$ = { 0, 1, 2, 4, 7 } = S_0

cerradura-ϵ(mov(S_0,a)) = cerradura-ϵ({ 3 }) = { 3, 6, 7, 1, 2, 4 } = S_1
cerradura-ϵ(mov(S_0,b)) = cerradura-ϵ({ 5 }) = { 5, 6, 7, 1, 2, 4 } = S_2

cerradura-ϵ(mov(S_1,a)) = cerradura-ϵ({ 3 }) = S_1
cerradura-ϵ(mov(S_1,b)) = cerradura-ϵ({ 8, 5 }) = { 8, 5, 6, 7, 1, 2, 4 } = S_3

cerradura-ϵ(mov(S_2,a)) = cerradura-ϵ({ 3 }) = S_1
cerradura-ϵ(mov(S_2,b)) = cerradura-ϵ({ 8, 5 }) = S_3

cerradura-ϵ(mov(S_3,a)) = cerradura-ϵ({ 9, 3 }) = { 9, 3, 6, 7, 1, 2, 4 } = S_4
cerradura-ϵ(mov(S_3,b)) = cerradura-ϵ({ 8, 5 }) = S_3

cerradura-ϵ(mov(S_4,a)) = cerradura-ϵ({ 3 }) = S_1
cerradura-ϵ(mov(S_4,b)) = cerradura-ϵ({ 8, 5 }) = S_3

De las operaciones de cerradura se obtiene el AFD:

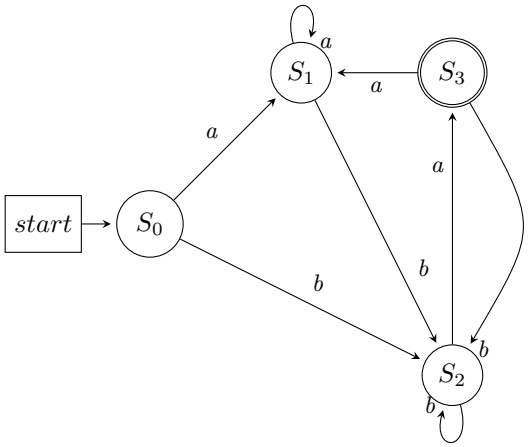

Figura 21: AFD obtenido de la expresión regular **Itálica{**$(a|b)^*ba$**}**.

Se aplica el algoritmo de minimización del cual se obtiene el autómata AFD mínimo:

Se forma el grupo de estados no finales $G_0 = \{S_0, S_1, S_2, S_3\}$ y el grupo de estados finales $G_1 = \{S_4\}$. El grupo G_1 por tener un sólo estado es de por si consistente. Al grupo G_0 se le aplica la consistencia y este se divide en dos grupos: $G_2 = \{S_0, S_1, S_2\}$ y $G_3 = \{S_3\}$, así que el autómata AFD mínimo se forma como en la figura 22 .

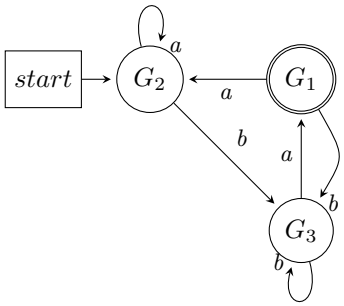

Figura 22: AFD mínimo obtenido de la expresión regular $(a|b)^*ba$.

EJEMPLO 6 (Construcción de un analizador léxico.) *A partir de la siguiente expresión regular construir un analizador léxico:* ***la expresión regular*** $(a|b)^*$ *abb****, se transforma en AFN aplicando la construcción de Thompson para cuatro concatenaciones****, la primera concatenación es una cerradura de Kleene y dentro de la cerradura de Kleene una alternativa, por lo que se obtiene el siguiente autómata:*

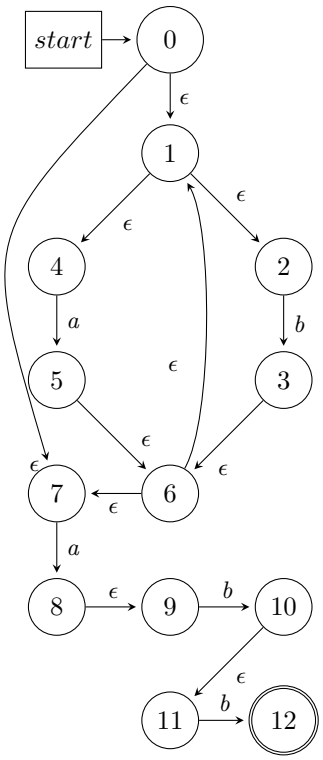

Figura 23: AFN obtenido de los patrones de la construcción de Thompson de la expresión regular **Itálica{**$(a|b)^*abb$**}**.

El autómata AFD de la figura 24 se obtiene aplicando el algoritmo de subconjuntos al autómata AFN de la figura 23 .

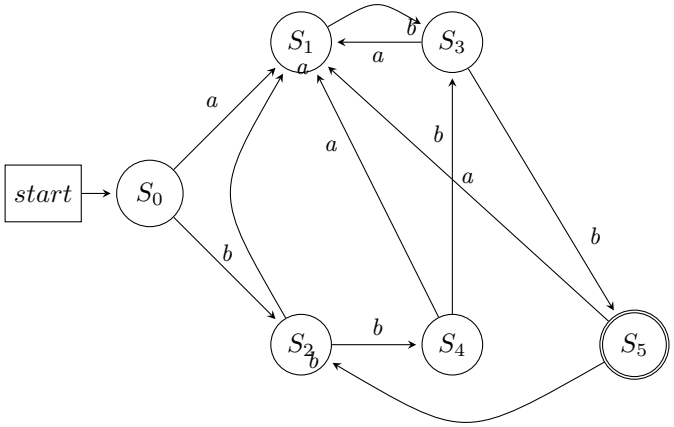

Figura 24: AFD obtenido del AFN de la figura 23 para la expresión regular **Itálica{**$(a|b)^*abb$**}**.

EJEMPLO 7 (Construcción de un analizador léxico.) *A partir de la siguiente expresión regular construir un analizador léxico:* ***la expresión regular*** $(a|b)^*$ bba***, se transforma en AFN aplicando la construcción de Thompson para cuatro concatenaciones****, la primera concatenación es una cerradura de Kleene y dentro de la cerradura de Kleene una alternativa, por lo que se obtiene el siguiente autómata:*

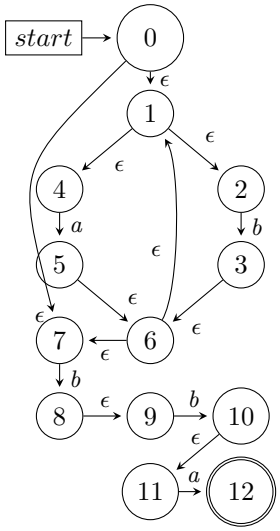

Figura 25: AFN obtenido de los patrones de la construcción de Thompson de la expresión regular **Itálica{**$(a|b)^*bba$**}**.

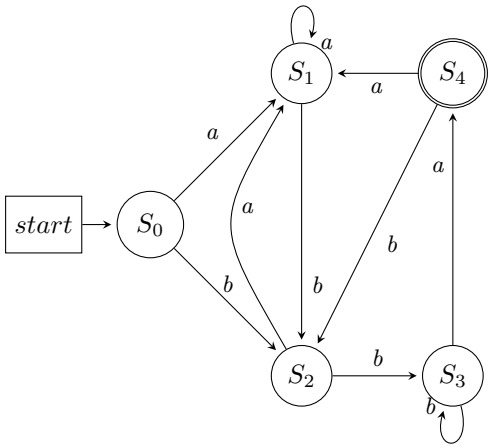

Figura 26: AFD obtenido de la expresión regular **Itálica{**$(a|b)^*bba$**}**.

Aplicando el algoritmo de minimización al AFD de la figura 26 , se obtiene el autómata AFD_M de la figura 27 :

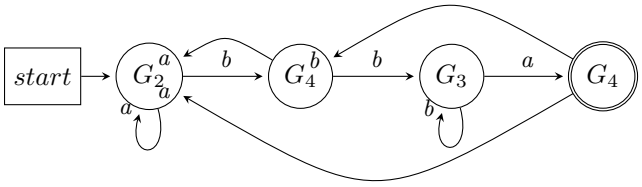

Figura 27: AFD mínimo de la expresión regular **Itálica{**$(a|b)^*bba$**}**.

Se debe construir la función siguiente(), que se muestra a continuación en el algoritmo :

Algorithm 0.5: SiguienteCaracter(), donde p es una lista enlazada simple, que contiene la palabra a analizar.

```
1 TIPE c:char ;
2 TIPE ↑ p:Lista ;
3 while p ↑.sig <> NULL do
4 |   c := p ↑ .dato ;
5 |   ↑ p := p ↑.sig ;
6 |   return c ;
```

Después de haber obtenido la tabla de transiciones del autómata AFD mínimo, se aplica el siguiente código para implementar el analizador léxico, escaner o reconocedor:

Algorithm 0.6: Instrucciones para un reconocedor

```
1 car ← SiguienteCaracter();
2 Estados ← s0;
3 while car <> EOF ∧ Estados <> se do
4 |   Estados := δ[Estados, car];
5 |   car := SiguienteCaracter();
6 if Estados == sa then
7 |   EsEnPan(Aceptado);
8 else
9 EsEnPan(Error);
```

Lista de ejercicios

Dadas las siguientes expresiones regulares construir un analizador léxico:

1. $a^*(a|b)aa$
2. $((a|b)(a|bb))^*$
3. $(ab|ac)^*$
4. $(0|1)^*1100\ 1^*$

5. $(01|10|00)^*11$

Generadores de analizadores léxicos

Los generadores de analizadores léxicos hacen todo el proceso desde la introducción de la expresión regular, estos generan el autómata mínimo ya codificado.

Las especificaciones en FLEX se hace de la siguiente forma:

```
Definiciones
%%
Reglas
%%
Código de usuario
```

En la siguiente tabla 6, se especifican las expresiones regulares en LEX:

Expresión regular	Código
ER	CO
if	{return IF;}
$[a-z][a-z0-9]^*$	{return ID;}
$[0-9]^+$	{return NUM;}
$([0-9]^+"."[0-9]^*)\|([0-9]^*"."[0-9]^+)$	{return REAL;}
$("--"[a-z]^*"")\|(""\|""\|"")^+$	{ /* do nothing */ }
.	{error();}

Cuadro 6: Tabla de código en lex.

El siguiente es un programa en LEX:

```
%{
/* Declaraciones de C: */
#include "tokens.h" /* definiciones de IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Definiciones de Lex: */
digits [0-9]+
%%
/* Expresiones Regulares y Acciones: */
if
{ADJ; return IF;}
[a-z][a-z0-9]*
```

```
{ADJ; yylval.sval=String(yytext);
return ID;}
{digits}
{ADJ; yylval.ival=atoi(yytext);
return NUM;}
({digits}"."[0-9]*)|([0-9]*"."{digits})  {ADJ;
                                          yylval.fval=atof(yytext);
                                          return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+  {ADJ;}
.    {ADJ; EM_error("caracter ilegal");}
```

Bibliografía

[1] Thomas W. Parsons, Introduction to compiler construction, Computer Science Press, 1992.

[2] Ralph Grishman, Computational Linguistics An Introduction, Cambridge University Press, 1986.

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compiladores Principios, técnicas y herramientas, Addison Wesley, 1990.

[4] Torben Aegidius Mogensen, Basics of compiler design, DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF COPENHAGEN, 2009.

[5] Linda Torczon and Keith D. Cooper, Engineering a Compiler, Second Edition, Elsevier 2012.

[6] Bernard Teufel, Stephanie Schmidt, Thomas Teufel, Compiladores, conceptos fundamentales, Addison-Wesley Iberoamericana, 1995.

[7] Jean Paul Tremblay, Paul G. Sorenson, The theory and practice of compiler writing, McGraw Hill Book Company, 1985.

[8] Mark P. Jones, jacc: just another compiler compiler for Java A Reference Manual and User Guide, Department of Computer Science Engineering OGI School of Science Engineering at OHSU 20000 NW Walker Road, Beaverton, OR 97006, USA February 16, 2004.

[9] www.lcc.uma.es/ galvez/ftp/tci/TutorialYacc.pdf