

# Análisis léxico

Pedro O. Pérez M., PhD.

Diseño de compiladores  
Tecnológico de Monterrey

*pperezm@tec.mx*

02-2023

## ① 3.1 - 3.3

3.1 El papel del analizador léxico

3.2 Búfer de entrada

3.3 Especificación de tokens

## ② 3.4 - 3.6

3.4 Reconocimiento de tokens

3.5 Lex, un generador de analizadores léxicos

3.6 Autómata finito

## ③ 3.7 - 3.9

3.7 De expresiones regulares a autómatas

3.8 Diseño de un generador de analizadores léxicos

3.9 Optimización de un emparejador basado en AFD

- La tarea principal de un analizador léxico es leer los caracteres de entrada de un programa fuente, agruparlos en lexemas, y producir como salida una secuencia de tokens por cada lexema.

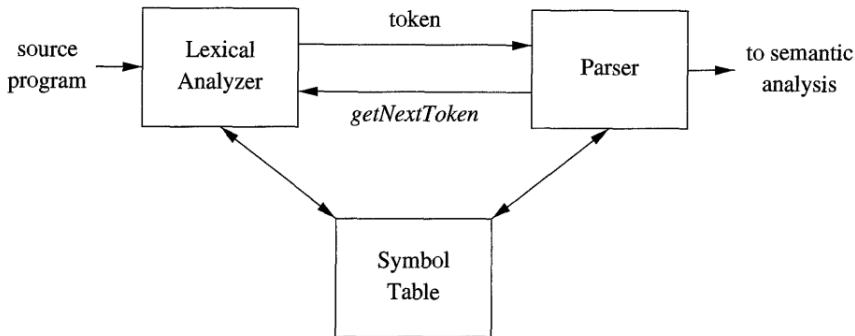


Figure 3.1: Interactions between the lexical analyzer and the parser

## 3.1.1 Análisis léxico vs. sintáctico

Existen tres razones principales por las cuales se separa el análisis léxico del sintáctico.

- Simplicidad del diseño.
- La eficiencia del compilador es mejorada.
- La portabilidad del compilador es aumentada.

## 3.1.2 Tokens, patrones y lexemas

- Un token es un dupla que consiste del nombre del token y un atributo opcional. El nombre del token es un símbolo abstracto que representa un TIPO de unidad léxica.
- Un patrón es una descripción de la forma que los lexemas de un token pueden tomar. En el caso de una palabra reservada, el patrón es la secuencia de caracteres que forman la palabra reservada.
- Un lexema es la secuencia específica de caracteres en el programa fuente que se emparejan con el patrón de un token determinado.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
<b>if</b>	characters <code>i</code> , <code>f</code>	<code>if</code>
<b>else</b>	characters <code>e</code> , <code>l</code> , <code>s</code> , <code>e</code>	<code>else</code>
<b>comparison</b>	<code>&lt;</code> or <code>&gt;</code> or <code>&lt;=</code> or <code>&gt;=</code> or <code>==</code> or <code>!=</code>	<code>&lt;=</code> , <code>!=</code>
<b>id</b>	letter followed by letters and digits	<code>pi</code> , <code>score</code> , <code>D2</code>
<b>number</b>	any numeric constant	<code>3.14159</code> , <code>0</code> , <code>6.02e23</code>
<b>literal</b>	anything but <code>"</code> , surrounded by <code>"</code> 's	<code>"core dumped"</code>

Figure 3.2: Examples of tokens

En la mayoría de los lenguajes de programación, las siguientes clases cubren la mayoría, sino todas, de tokens:

- Un token para cada palabra reservada.
- Token para operadores (individuales o grupales)
- Un token representando todos los identificadores.
- Uno o más tokens representan constantes (números o literales).
- Tokens para cada símbolo de puntuación.

### 3.1.3 Atributos de los tokens

Entre los atributos más comunes asociados a los tokens podemos encontrar su lexema, su tipo y la localización de ese token en la tabla de símbolos. Por ejemplo, para la cadena en Fortran  $E = M * C ** 2$ , tendríamos...

<**id**, pointer to symbol-table entry for **E**>

<**assign\_op**>

<**id**, pointer to symbol-table entry for **M**>

<**mult\_op**>

<**id**, pointer to symbol-table entry for **C**>

<**exp\_op**>

<**number**, integer value 2>



- Es difícil para un analizador léxico detectar un error en el programa fuente sin la ayuda de otros componentes.
- Sin embargo, supongamos que un analizador léxico es incapaz de continuar porque ninguno de los patrones de los token se emparejan con ningún prefijo de la cadena de entrada, ¿qué hacemos? Entramos en “modo pánico”, ignora todos los caracteres de entrada hasta que pueda detectar un token bien formado.

Existen otras estrategias:

- Borrar un carácter de la entrada restante.
- Insertar un carácter omitido en la entrada restante.
- Reemplazar un carácter por otro.
- Intercambiar dos caracteres adyacentes.

Divide el siguiente código en C++:

```
float limitedSquare(x) float x {  
    /* returns x-squared, but never more than 100 */  
    return (x<=-10.0||x>=10.0)?100:x*x;  
}
```

Antes de hablar sobre el problema de reconocimiento de lexemas, es conveniente examinar algunas estrategias utilizados para agilizar la lectura de un programa fuente.

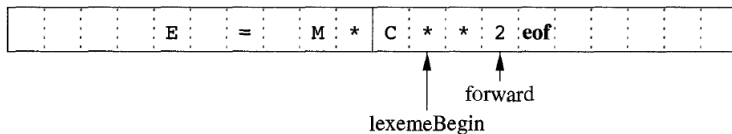


Figure 3.3: Using a pair of input buffers

Una importante estrategia para agilizar la lectura es emplear dos búferes que, alternativamente, son recargados. Cada búfer es un tamaño  $N$  (usualmente 4096 bytes). Dos punteros son mantenidos: `lexemeBegin` y `forward`.



```

switch ( *forward++ ) {
    case eof:
        if ( forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if ( forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}

```

Figure 3.5: Lookahead code with sentinels

- Un *alfabeto* es un conjunto finito de símbolos.
- Un *string* de un alfabeto es una secuencia finita de símbolos obtenidos de ese alfabeto. La longitud de una string  $s$ , usualmente escrita  $|s|$ , el número de símbolos en  $s$ .
- Un *lenguaje* es cualquier conjunto contable de strings sobre un alfabeto determinado.
- Si  $x$  y  $y$  son strings, entonces la concatenación de  $x$  y  $y$ , indicada como  $xy$ , es el string que se forma al agregar  $y$  a  $x$ . El string vacío,  $\epsilon$ , es identidad bajo la concatenación.
- Concatenación se define como sigue:  $s^0$  es  $\epsilon$ , y para  $i > 0$ ,  $s^i$  es  $s^{(i-1)}s$ .



Existe algunos términos relacionados con strings que debemos tener en cuenta:

- El *prefijo* de un string  $s$  se obtiene al remover cero o más símbolos del final de  $s$ . Por ejemplo, *ban*, *banana* y  $\epsilon$  son prefijos de *banana*.
- El *sufijo* de un string  $s$  se obtiene al remove cero o más símbolos del inicial de  $s$ . Por ejemplo *nana*, *banana* y  $\epsilon$  son sufijos de *banana*.
- Un *substring* de  $s$  se obtiene borrando cualquier prefijo y sufijo de  $s$ . Por ejemplo, *banana*, *nan* y  $\epsilon$  son substrings de *banana*.
- Los prefijos, sufijos o substring *propios* de un string  $s$  son todos aquellos prefijos, sufijos o substrings de  $s$  que no son  $\epsilon$  o el mismo string.
- Una *subsecuencia* de  $s$  es cualquier string formado a partir de la eliminación de cero o más símbolo no necesariamente consecutivos. Por ejemplo, *baan* es una subsecuencia de *banana*.

## 3.3.2 Operaciones sobre lenguajes

OPERATION	DEFINITION AND NOTATION
<i>Union of <math>L</math> and <math>M</math></i>	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
<i>Concatenation of <math>L</math> and <math>M</math></i>	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
<i>Kleene closure of <math>L</math></i>	$L^* = \bigcup_{i=0}^{\infty} L^i$
<i>Positive closure of <math>L</math></i>	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

Sea  $L$  el conjunto de caracteres  $a..z, A..Z$  y  $D$  el conjunto de dígitos  $0..9$ :

- $L \cup D$  es el conjunto de caracteres y dígitos.
- $LD$  es conjunto de string de longitud 2 integrados por una letra y un dígito.
- $L^4$  el conjunto de string de 4 letras.
- $L^*$  es el conjunto de todos los string de letras, incluyendo  $\epsilon$ .
- $L(L \cup D)^*$  es el conjunto de todos los strings de letras y dígitos que empiezan con una letra.
- $D^+$  es el conjunto de todos los string de uno o más dígitos.

Las expresiones regulares son construidas recursivamente a partir de expresiones regulares más pequeñas.

- $\epsilon$  es una expresión regular, y  $L(r)$  es  $\epsilon$ , esto es, el lenguaje cuyo único miembro es el string vacío.
- Si  $a$  es un símbolo de  $\Sigma$ , entonces  $a$  es una expresión regular, y  $L(a) = a$ , esto es, el lenguaje con un string de longitud uno, con  $a$  es esa posición.

Sean  $r$  y  $s$  expresiones regulares denotando los lenguajes  $L(r)$  y  $L(s)$  respectivamente:

- $(r)|(s)$  es una expresión regular que denota el lenguaje  $L(r) \cup L(s)$ .
- $(r)(s)$  es una expresión regular que denota  $L(r)L(s)$ .
- $(r)^*$  es una expresión regular que denota  $(L(r))^*$ .
- $(r)$  es una expresión regular que denota  $L(r)$ .

Podemos quitar ciertos paréntesis si adoptamos las siguiente convenciones:

- El operador unitario  $*$  tiene la más alta prioridad y es asociativo por la izquierda.
- La concatenación es segundo en precedencia y asociativo por la izquierda.
- $|$  tiene la precedencia más baja y es asociativo por la izquierda.

**Example 3.4:** Let  $\Sigma = \{a, b\}$ .

1. The regular expression  $\mathbf{a|b}$  denotes the language  $\{a, b\}$ .
2.  $\mathbf{(a|b)(a|b)}$  denotes  $\{aa, ab, ba, bb\}$ , the language of all strings of length two over the alphabet  $\Sigma$ . Another regular expression for the same language is  $\mathbf{aa|ab|ba|bb}$ .
3.  $\mathbf{a^*}$  denotes the language consisting of all strings of zero or more  $a$ 's, that is,  $\{\epsilon, a, aa, aaa, \dots\}$ .
4.  $\mathbf{(a|b)^*}$  denotes the set of all strings consisting of zero or more instances of  $a$  or  $b$ , that is, all strings of  $a$ 's and  $b$ 's:  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$ . Another regular expression for the same language is  $\mathbf{(a^*b^*)^*}$ .
5.  $\mathbf{a|a^*b}$  denotes the language  $\{a, b, ab, aab, aaab, \dots\}$ , that is, the string  $a$  and all strings consisting of zero or more  $a$ 's and ending in  $b$ .

LAW	DESCRIPTION
$r s = s r$	$ $ is commutative
$r (s t) = (r s) t$	$ $ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s t) = rs rt; (s t)r = sr tr$	Concatenation distributes over $ $
$\epsilon r = r\epsilon = r$	$\epsilon$ is the identity for concatenation
$r^* = (r \epsilon)^*$	$\epsilon$ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Figure 3.7: Algebraic laws for regular expressions



**Example 3.6:** Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$\begin{aligned}
 \textit{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\
 \textit{digits} &\rightarrow \textit{digit} \textit{digit}^* \\
 \textit{optionalFraction} &\rightarrow . \textit{digits} \mid \epsilon \\
 \textit{optionalExponent} &\rightarrow ( \textit{E} ( + \mid - \mid \epsilon ) \textit{digits} ) \mid \epsilon \\
 \textit{number} &\rightarrow \textit{digits} \textit{optionalFraction} \textit{optionalExponent}
 \end{aligned}$$

### 3.3.5 Extensiones de expresiones regulares

- $+$ : Una o más instancias.
- $?$ : Cero o una instancia.
- Clases de caracteres: Una expresión regular del tipo  $a_1|a_2|\dots|a_n$  donde  $a_i$ 's son símbolos del alfabeto puede ser reemplazado por  $[a_1a_2\dots a_n]$ .

**Example 3.7:** Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$\begin{aligned} \textit{letter\_} &\rightarrow [\textbf{A-Za-z\_}] \\ \textit{digit} &\rightarrow [0-9] \\ \textit{id} &\rightarrow \textit{letter\_} ( \textit{letter} \mid \textit{digit} )^* \end{aligned}$$

The regular definition of Example 3.6 can also be simplified:

$$\begin{aligned} \textit{digit} &\rightarrow [0-9] \\ \textit{digits} &\rightarrow \textit{digit}^+ \\ \textit{number} &\rightarrow \textit{digits} ( . \textit{digits} )? ( \textbf{E} [+-]? \textit{digits} )? \end{aligned}$$

Resuelve los siguiente ejercicios:

**Exercise 3.3.2:** Describe the languages denoted by the following regular expressions:

a)  $a(a|b)^*a$ .

b)  $((\epsilon|a)b^*)^*$ .

c)  $(a|b)^*a(a|b)(a|b)$ .

d)  $a^*ba^*ba^*ba^*$ .

!! e)  $(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$ .

Incisos a, b, c

**Exercise 3.3.5:** Write regular definitions for the following languages:

- a) All strings of lowercase letters that contain the five vowels in order.
- b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
- c) Comments, consisting of a string surrounded by `/*` and `*/`, without an intervening `*/`, unless it is inside double-quotes `"`.

Incisos a, b

**Exercise 3.3.6:** Write character classes for the following sets of characters:

- a) The first ten letters (up to “j”) in either upper or lower case.
- b) The lowercase consonants.
- c) The “digits” in a hexadecimal number (choose either upper or lower case for the “digits” above 9).
- d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

Incisos a, c

$$\begin{array}{ll}
 stmt & \rightarrow \text{if } expr \text{ then } stmt \\
 & | \text{if } expr \text{ then } stmt \text{ else } stmt \\
 & | \epsilon \\
 expr & \rightarrow term \text{ relop } term \\
 & | term \\
 term & \rightarrow id \\
 & | number
 \end{array}$$

Figure 3.10: A grammar for branching statements

<i>digit</i>	→	[0-9]
<i>digits</i>	→	<i>digit</i> <sup>+</sup>
<i>number</i>	→	<i>digits</i> ( . <i>digits</i> )? ( E [+ -]? <i>digits</i> )?
<i>letter</i>	→	[A-Za-z]
<i>id</i>	→	<i>letter</i> ( <i>letter</i>   <i>digit</i> )*
<i>if</i>	→	if
<i>then</i>	→	then
<i>else</i>	→	else
<i>relop</i>	→	<   >   <=   >=   =   <>

Figure 3.11: Patterns for tokens of Example 3.8



LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	–	–
<b>if</b>	<b>if</b>	–
<b>then</b>	<b>then</b>	–
<b>else</b>	<b>else</b>	–
Any <i>id</i>	<b>id</b>	Pointer to table entry
Any <i>number</i>	<b>number</b>	Pointer to table entry
<b>&lt;</b>	<b>relop</b>	<b>LT</b>
<b>&lt;=</b>	<b>relop</b>	<b>LE</b>
<b>=</b>	<b>relop</b>	<b>EQ</b>
<b>&lt;&gt;</b>	<b>relop</b>	<b>NE</b>
<b>&gt;</b>	<b>relop</b>	<b>GT</b>
<b>&gt;=</b>	<b>relop</b>	<b>GE</b>

Figure 3.12: Tokens, their patterns, and attribute values

## 3.4.1 Diagramas de transición

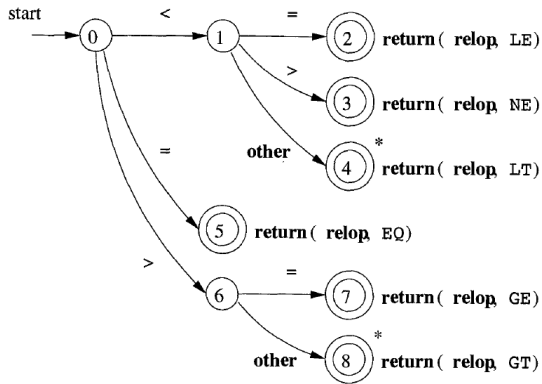


Figure 3.13: Transition diagram for **relop**

- Estado de aceptación o final.
- Regresar el apuntador “forward”.
- Estado inicial.

# Reconocimiento de palabras reservadas e identificadores

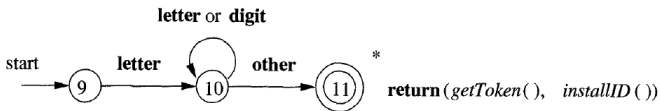


Figure 3.14: A transition diagram for **id**'s and keywords

- Agregar las palabras reservadas a tabla de símbolos antes de empezar el proceso de análisis.
- Crear diagramas de transición separados para cada palabra reservada.

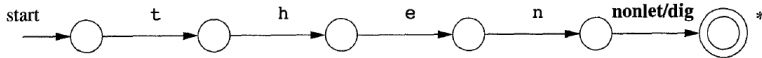


Figure 3.15: Hypothetical transition diagram for the keyword **then**

# Terminando de analizar el ejemplo actual

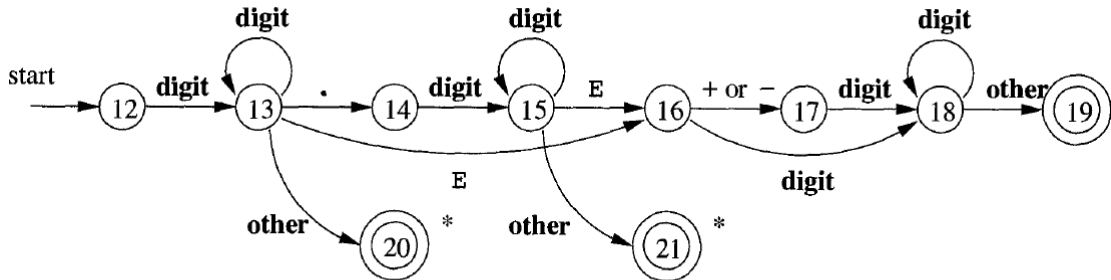


Figure 3.16: A transition diagram for unsigned numbers

## 3.4.4 Arquitectura del un analizador léxico basado en diagramas de transiciones

- Podemos organizar que los diagramas de transición para cada token sean tratados secuencialmente. De esta forma, la función `fail` reinicia el apuntador “forward” e inicia el siguiente diagrama de transición.
- Podemos ejecutar varios diagramas de transición “en paralelo”, alimentado el siguiente carácter de entrada para todos ellos y permitiendo que cada uno haga la transición que se requiera (prefijo más largo).
- Podemos combinar todos los diagramas de transición en uno. De esta forma permitimos que el diagrama de transición lea la entrada y tome el lexema más largo que se empareje con un patrón.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
               or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

**Exercise 3.4.1:** Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.

Incisos a, b, c

**Exercise 3.4.2:** Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

Incisos a, b

Revisa la sección 3.5 de libro [AHO]

Un autómata finito es esencialmente un grafo, como los diagramas de transición, pero con algunas diferencias:

- Los autómatas finitos son “reconocedores”.
- Pueden ser de dos tipos:
  - Autómata Finito No Determinista, AFN (Non-deterministic finite automata, NFA). No tiene restricciones en las etiquetas de los arcos. Un símbolo puede aparecer en diferentes arcos que salgan del mismo estado y  $\epsilon$  es una posible etiqueta.
  - Autómata Finito Determinista, AFD (Deterministic finite automata, DFA). Tiene para cada estado y para cada símbolo del alfabeto exactamente un arco saliendo de ese estado.



## 3.6.1 Autómata Finito No Determinista

Un autómata finito no determinista (AFN) consiste de:

- Un conjunto finito de estados,  $S$ .
- Un conjunto de símbolos de entrada  $\Sigma$  (alfabeto de entrada). Asumimos que  $\epsilon$  nunca es miembro de  $\Sigma$ .
- Una *función de transición* que, para cada estado, y para cada símbolo en  $\Sigma \cup \epsilon$  un conjunto de siguientes estados.
- Un estado  $s_0$  del  $S$  considerado como estado inicial.
- Un conjunto de estados,  $F$ , subconjunto de  $S$ , que son identificados como estado de aceptación (o estados finales).

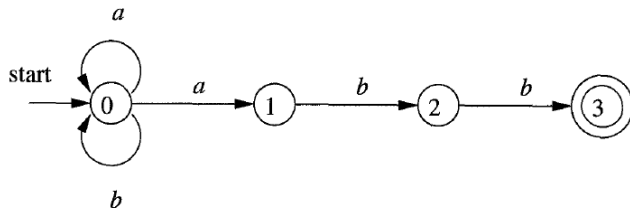


Figure 3.24: A nondeterministic finite automaton

STATE	$a$	$b$	$\epsilon$
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Figure 3.25: Transition table for the NFA of Fig. 3.24

### 3.6.3 Aceptación de una cadena de entrada por un autómata

Un AFN acepta una cadena de entrada  $x$ , sí y solo si existe un camino en el grafo de transición que parta del estado inicial y llegue a un estado de aceptación, tal que los símbolos a lo largo de ese camino formen  $x$ .

## 3.6.4 Autómata Finito Determinista

Un autómata finito determinista (AFD) es un caso especial de AFN donde

- No existen movimientos bajo  $\epsilon$ .
- Para cada estado  $s$  y para símbolo de entrada  $a$ , existe exactamente un arco que sale de  $s$  etiquetado como  $a$ .

```

s = s0;
c = nextChar();
while ( c != eof ) {
    s = move(s, c);
    c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";

```

Figure 3.27: Simulating a DFA

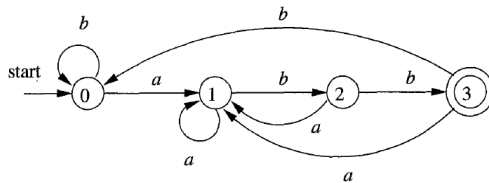


Figure 3.28: DFA accepting  $(a|b)^*abb$

**Exercise 3.6.3:** For the NFA of Fig. 3.29, indicate all the paths labeled  $aabb$ . Does the NFA accept  $aabb$ ?

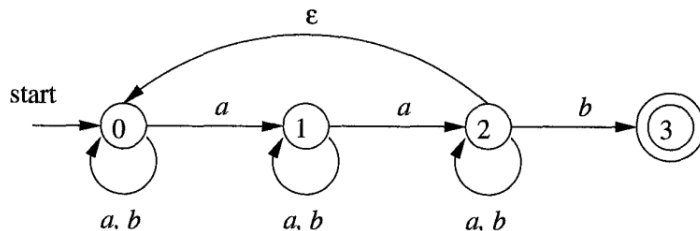


Figure 3.29: NFA for Exercise 3.6.3

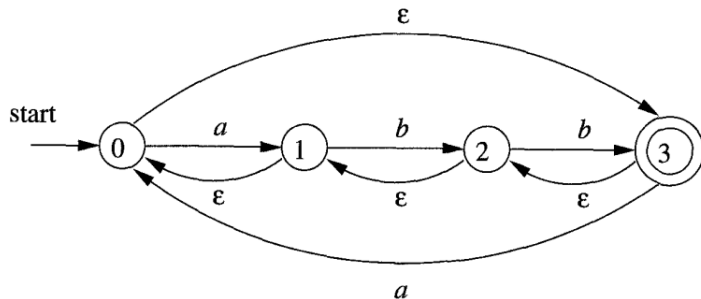


Figure 3.30: NFA for Exercise 3.6.4

**Exercise 3.6.4:** Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

**Exercise 3.6.5:** Give the transition tables for the NFA of:

a) Exercise 3.6.3.

b) Exercise 3.6.4.



## 3.7 De expresiones regulares a autómatas

Las expresiones regulares son la notación empleada para describir un analizador léxico, así como otros software de procesamiento de patrones. Sin embargo, la implementación de este software requiere de una simulación de un AFD o, quizás, de un AFN.

## 3.7.1 Algoritmo de conversión de AFN a AFD

OPERATION	DESCRIPTION
$\epsilon\text{-closure}(s)$	Set of NFA states reachable from NFA state $s$ on $\epsilon$ -transitions alone.
$\epsilon\text{-closure}(T)$	Set of NFA states reachable from some NFA state $s$ in set $T$ on $\epsilon$ -transitions alone; $= \bigcup_{s \text{ in } T} \epsilon\text{-closure}(s)$ .
$\text{move}(T, a)$	Set of NFA states to which there is a transition on input symbol $a$ from some state $s$ in $T$ .

Figure 3.31: Operations on NFA states

```

while ( there is an unmarked state  $T$  in  $Dstates$  ) {
    mark  $T$ ;
    for ( each input symbol  $a$  ) {
         $U = \epsilon\text{-closure}(\text{move}(T, a))$ ;
        if (  $U$  is not in  $Dstates$  )
            add  $U$  as an unmarked state to  $Dstates$ ;
         $Dtran[T, a] = U$ ;
    }
}

```

Figure 3.32: The subset construction

```

push all states of  $T$  onto stack;
initialize  $\epsilon\text{-closure}(T)$  to  $T$ ;
while ( stack is not empty ) {
    pop  $t$ , the top element, off stack;
    for ( each state  $u$  with an edge from  $t$  to  $u$  labeled  $\epsilon$  )
        if (  $u$  is not in  $\epsilon\text{-closure}(T)$  ) {
            add  $u$  to  $\epsilon\text{-closure}(T)$ ;
            push  $u$  onto stack;
        }
}

```

Figure 3.33: Computing  $\epsilon\text{-closure}(T)$

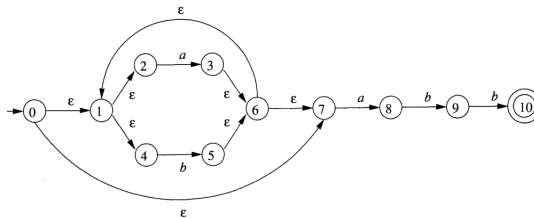


Figure 3.34: NFA  $N$  for  $(a|b)^*abb$

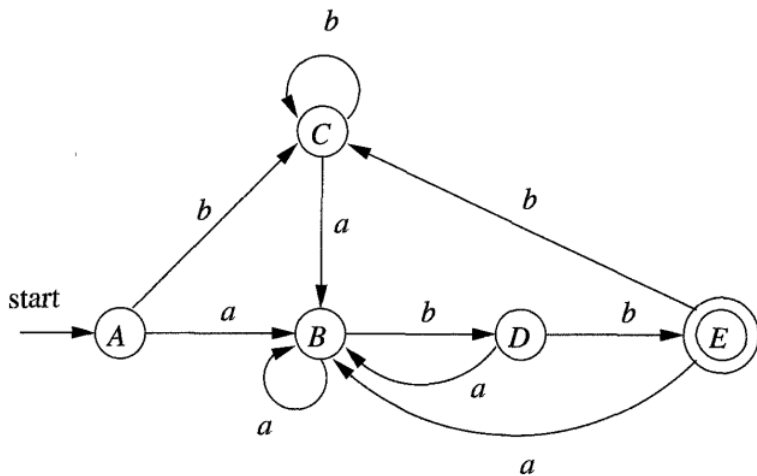


Figure 3.36: Result of applying the subset construction to Fig. 3.34

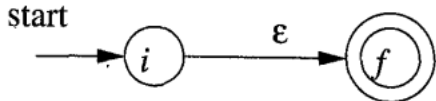
```
1)  $S = \epsilon\text{-closure}(s_0);$   
2)  $c = \text{nextChar}();$   
3) while (  $c \neq \text{eof}$  ) {  
4)      $S = \epsilon\text{-closure}(\text{move}(S, c));$   
5)      $c = \text{nextChar}();$   
6) }  
7) if (  $S \cap F \neq \emptyset$  ) return "yes";  
8) else return "no";
```

Figure 3.37: Simulating an NFA

### 3.7.4 Construcción de un AFN a partir de una expresión regular

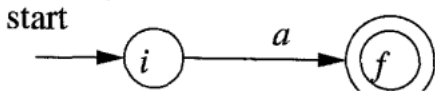
$$r = \epsilon$$

**BASIS:** For expression  $\epsilon$  construct the NFA



$$r = a$$

For any subexpression  $a$  in  $\Sigma$ , construct the NFA



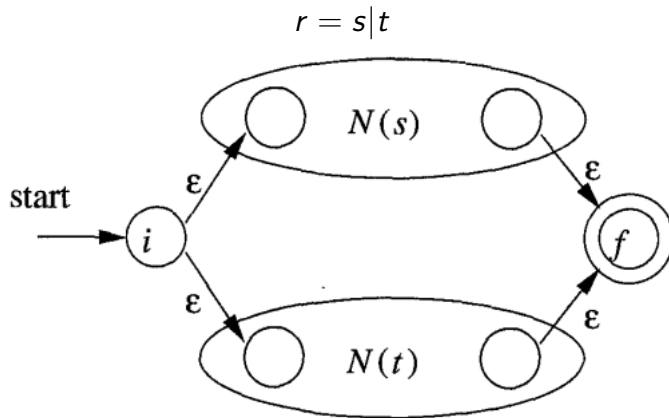


Figure 3.40: NFA for the union of two regular expressions

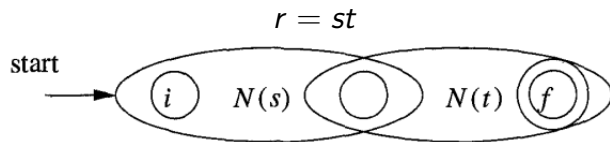


Figure 3.41: NFA for the concatenation of two regular expressions



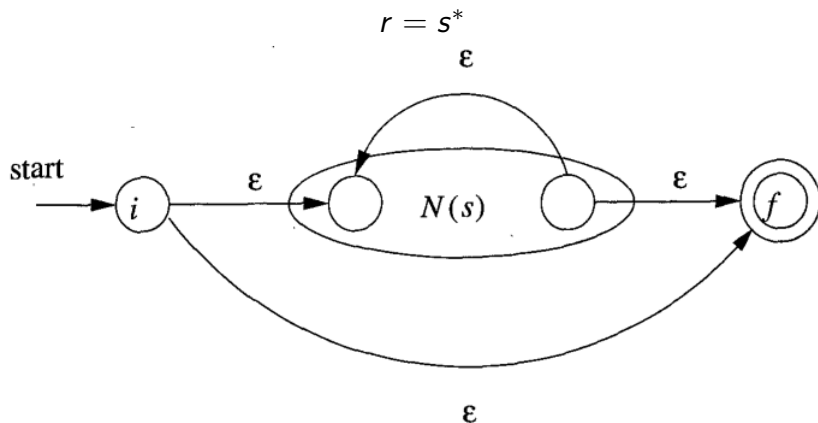


Figure 3.42: NFA for the closure of a regular expression

Construye el AFN de la siguiente expresión:

$$r = (a|b)^*abb$$

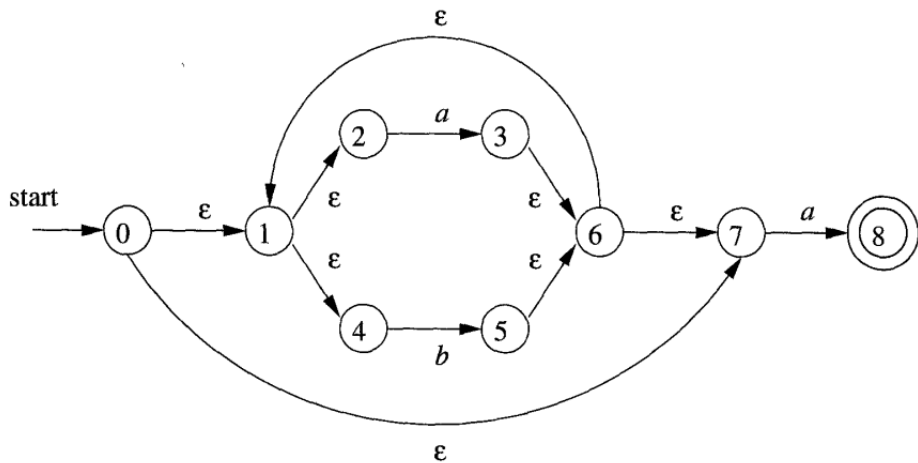


Figure 3.46: NFA for  $r_7$

## Ejercicios de la sección 3.7

**Exercise 3.7.3:** Convert the following regular expressions to deterministic finite automata, using algorithms 3.23 and 3.20:

a)  $(\mathbf{a|b})^*$ .

b)  $(\mathbf{a^*|b^*})^*$ .

c)  $((\epsilon|\mathbf{a})\mathbf{b^*})^*$ .

d)  $(\mathbf{a|b})^*\mathbf{abb(a|b)^*}$ .

Revisa la sección 3.8 del libro **[AHO]**.

Existen tres algoritmos que pueden ser usados para la implementación y optimización de emparejadores de patrones a partir de expresiones regulares:

- El primer algoritmo construye directamente el AFD sin construir la AFN intermedio.
- El segundo algoritmo minimiza el número de estados de cualquier AFD a través de la combinación de estados que tiene el mismo comportamiento futuro.
- El tercer algoritmo produce representaciones más compactas de las tablas de transiciones.

## 3.9.2 Funciones generadas a partir del árbol sintáctico extendido

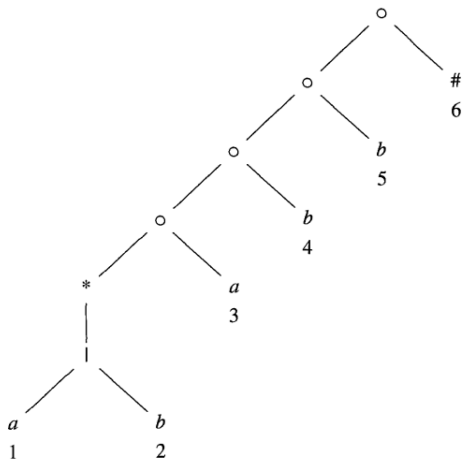


Figure 3.56: Syntax tree for  $(a|b)^*abb\#$

Para construir un AFD directamente a partir de una expresión regular, debemos construir el árbol sintáctico y calcular 4 conjuntos (o funciones): *nullable*, *firstpos*, *lastpos* y *followpos*.

- *nullable*( $n$ ) es verdadero para el nodo  $n$  de un árbol sintáctico sí y solo si la subexpresión representado por  $n$  tiene  $\epsilon$  en su lenguaje.
- *firstpos*( $n$ ) es el conjunto de posiciones de un subárbol con raíz en  $n$  que corresponde con el primer símbolo de al menos un string en el lenguaje de la subexpresión con raíz en  $n$ .
- *lastpos*( $n$ ) es el conjunto de posiciones de un subárbol con raíz en  $n$  que corresponde con el último símbolo de al menos un string en el lenguaje de la subexpresión con raíz en  $n$ .
- *followpos*( $n$ ), para una posición  $p$ , es el conjunto de posiciones  $q$  en todo el árbol sintáctico tal que existe una cadena  $x = a_1 a_2 \dots a_n$  en  $L((r)\#)$  tal que para alguna  $i$ , hay una camino que explica la membresía de  $x$  en  $L((r)\#)$  emparejando  $a_i$  a la posición  $p$  en el árbol sintáctico y  $a_{i+1}$  a la posición  $q$ .



### 3.9.3 Calculando los $nullable(n)$ , $firstpos(n)$ y $lastpos(n)$

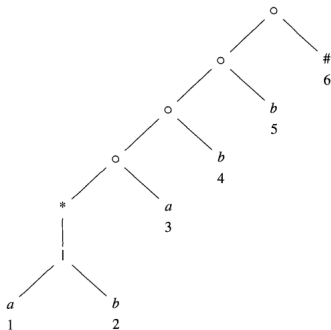


Figure 3.56: Syntax tree for  $(a|b)^*abb\#$

NODE $n$	$nullable(n)$	$firstpos(n)$
A leaf labeled $\epsilon$	<b>true</b>	$\emptyset$
A leaf with position $i$	<b>false</b>	$\{i\}$
An or-node $n = c_1 c_2$	$nullable(c_1)$ <b>or</b> $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
A cat-node $n = c_1c_2$	$nullable(c_1)$ <b>and</b> $nullable(c_2)$	<b>if</b> ( $nullable(c_1)$ ) $firstpos(c_1) \cup firstpos(c_2)$ <b>else</b> $firstpos(c_1)$
A star-node $n = c_1^*$	<b>true</b>	$firstpos(c_1)$

Figure 3.58: Rules for computing  $nullable$  and  $firstpos$

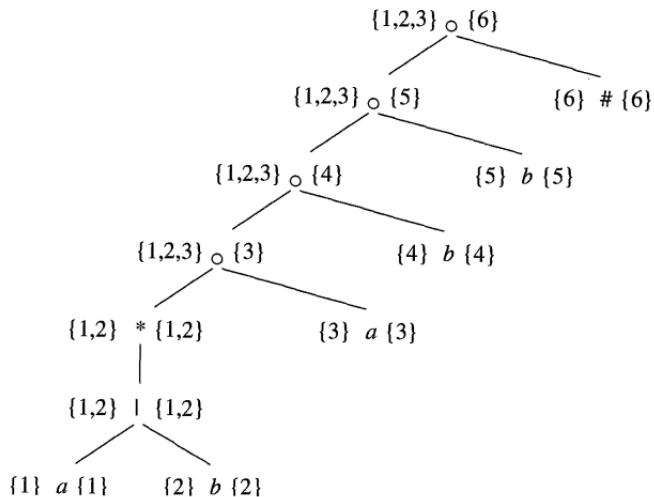


Figure 3.59: *firstpos* and *lastpos* for nodes in the syntax tree for  $(a|b)^*abb\#$

### 3.9.4 Calculando los *followpos*( $n$ )

Finally, we need to see how to compute *followpos*. There are only two ways that a position of a regular expression can be made to follow another.

1. If  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$ , then for every position  $i$  in *lastpos*( $c_1$ ), all positions in *firstpos*( $c_2$ ) are in *followpos*( $i$ ).
2. If  $n$  is a star-node, and  $i$  is a position in *lastpos*( $n$ ), then all positions in *firstpos*( $n$ ) are in *followpos*( $i$ ).

NODE $n$	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Figure 3.60: The function *followpos*

## 3.9.5 Convirtiendo una expresión regular en un AFD

### METHOD:

1. Construct a syntax tree  $T$  from the augmented regular expression  $(r)\#$ .
2. Compute *nullable*, *firstpos*, *lastpos*, and *followpos* for  $T$ , using the methods of Sections 3.9.3 and 3.9.4.
3. Construct  $Dstates$ , the set of states of DFA  $D$ , and  $Dtran$ , the transition function for  $D$ , by the procedure of Fig. 3.62. The states of  $D$  are sets of positions in  $T$ . Initially, each state is “unmarked,” and a state becomes “marked” just before we consider its out-transitions. The start state of  $D$  is  $firstpos(n_0)$ , where node  $n_0$  is the root of  $T$ . The accepting states are those containing the position for the endmarker symbol  $\#$ .

```

initialize  $Dstates$  to contain only the unmarked state  $firstpos(n_0)$ ,
    where  $n_0$  is the root of syntax tree  $T$  for  $(r)\#$ ;
while ( there is an unmarked state  $S$  in  $Dstates$  ) {
    mark  $S$ ;
    for ( each input symbol  $a$  ) {
        let  $U$  be the union of  $followpos(p)$  for all  $p$ 
            in  $S$  that correspond to  $a$ ;
        if (  $U$  is not in  $Dstates$  )
            add  $U$  as an unmarked state to  $Dstates$ ;
         $Dtran[S, a] = U$ ;
    }
}

```

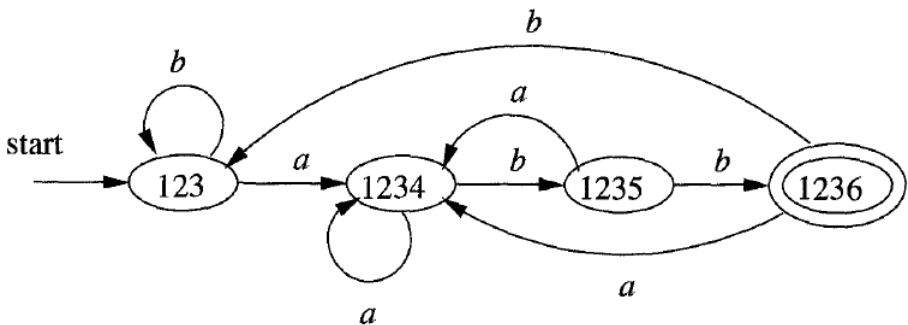


Figure 3.63: DFA constructed from Fig. 3.57

# Revisión de la implementación de un analizador léxico.