Análisis léxico

Pedro O. Pérez M., PhD.

Desarrollo de aplicaciones avanzadas de ciencias computacionales
Módulo 3: Compiladores
Tecnológico de Monterrey

pperezm@tec.mx

02-2024

1. 3.1 - 3.3
   - 3.1 El papel del analizador léxico
   - 3.2 Búfer de entrada
   - 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
   - Diseño de un generador de analizadores léxicos

- 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.

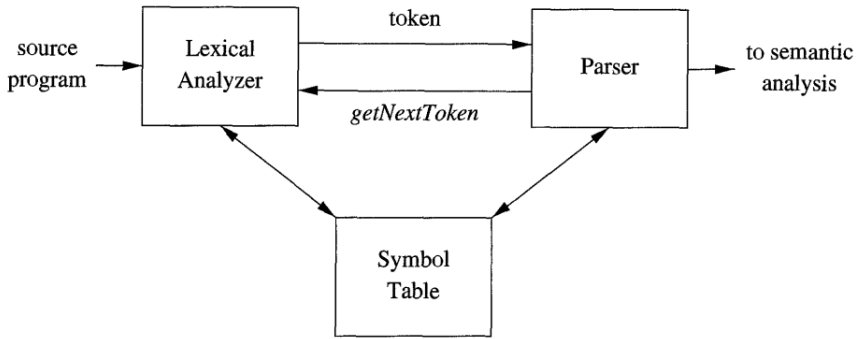


Figure 3.1: Interactions between the lexical analyzer and the parser

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.

Tokens, patrones y lexemas

- Un token es una 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.

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.

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

Figure 3.2: Examples of tokens

- 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.

Ejemplo

Divide el siguiente código en tokens del lenguaje C/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.

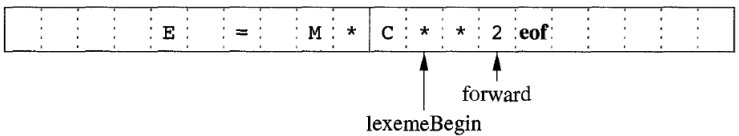


Figure 3.3: Using a pair of input buffers

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`.

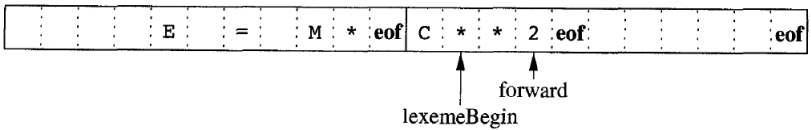


Figure 3.4: Sentinels at the end of each buffer

Por cada carácter leído, debemos hacer dos pruebas: uno para determinar si es el final de búfer y otro para determinar que carácter es leído. Podemos combinar las dos pruebas usando un centinela (un carácter que no aparece en la gramática).

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

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

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

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

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

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

- ϵ es una expresión regular, y $L(\mathbf{r})$ *es* ϵ*, esto es, el lenguaje cuyo único miembro es la cadena vacía.*
- *Si* a *es un símbolo de* Σ*, entonces* $\mathbf{a}$ *es una expresión regular, y* $L(\mathbf{a}) = a$*, esto es , el lenguaje con una cadena de longitud uno, con* a *es esa posición.*

Sean r y L *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 $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ 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: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, 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 $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ 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

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

TABLE 2.2 Extended Regular Expression Operations

Operation	Symbol	Example	Regular Expression
concatenation		ab	ab
		[a-c][AB]	$aA \cup aB \cup bA \cup bB \cup cA \cup cB$
Kleene star	*	[ab] *	$(a \cup b)^*$
disjunction	|	[ab]*|A	$(a \cup b)^* \cup A$
zero or more	+	[ab]+	$(a \cup b)^+$
zero or one	?	a?	$(a \cup \lambda)$
one character	.	a.a	$a(a \cup b)a$ if $\Sigma = \{a, b\}$
n-times	{n}	a{4}	$aaaa = a^4$
n or more times	{n,}	a{4,}	$aaaaa^*$
n to m times	{n,m}	a{4,6}	$aaaa \cup aaaaa \cup aaaaaa$

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

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{[A-Za-z\_]} \\
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter} \mid \mathit{digit}\ )^*
\end{array}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{digits} & \rightarrow & \mathit{digit}^+ \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )?
\end{array}
$$

Resuelve los siguiente ejercicios:

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

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

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

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

d) $\mathbf{a}^*\mathbf{ba}^*\mathbf{ba}^*\mathbf{ba}^*$.

!! e) $(\mathbf{aa|bb})^*((\mathbf{ab|ba})(\mathbf{aa|bb})^*(\mathbf{ab|ba})(\mathbf{aa|bb})^*)^*$.

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

Diagramas de transición

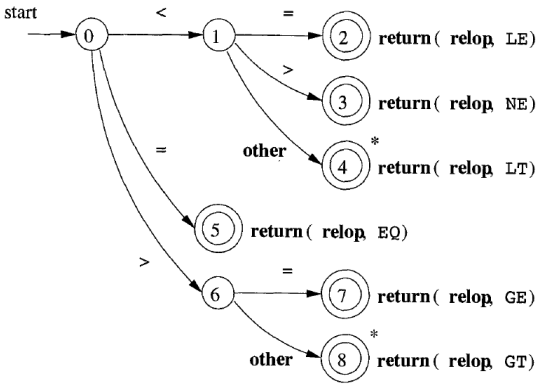


Figure 3.13: Transition diagram for **relop**

- Estado de aceptación o final.
- Regresar el apuntador "`forward`".
- Estado inicial.

Reconocimiento de palabras reservadas e identificadores

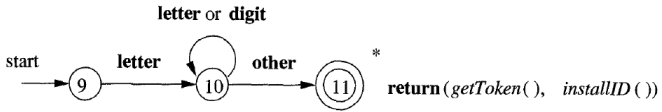


Figure 3.14: A transition diagram for **id**'s and keywords

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

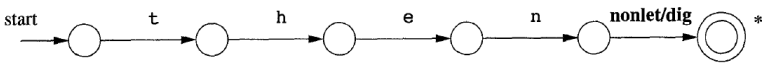


Figure 3.15: Hypothetical transition diagram for the keyword `then`

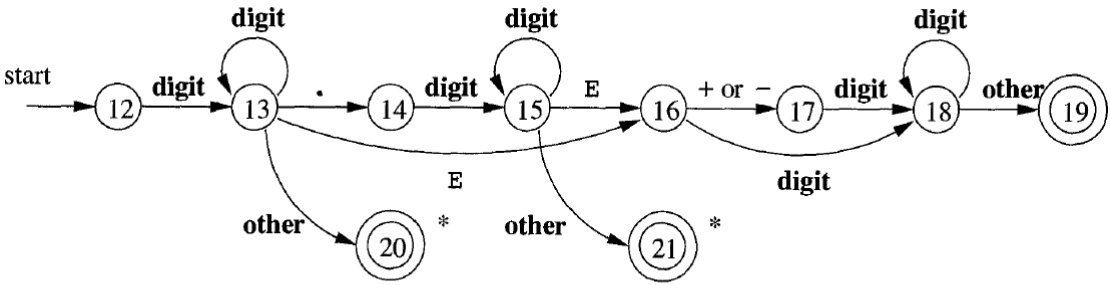


Figure 3.16: A transition diagram for unsigned numbers

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 ϵ 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.

Un autómata finito no determinista (AFN) consiste de:

- Un conjunto finito de estados, S.
- Un conjunto de símbolos de entrada Σ (alfabeto de entrada). Asumimos que ϵ nunca es miembro de Σ.
- 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).

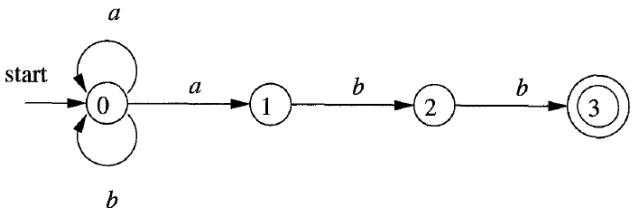


Figure 3.24: A nondeterministic finite automaton

STATE	a	b	ϵ
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

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.

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

Figure 3.37: Simulating an NFA

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

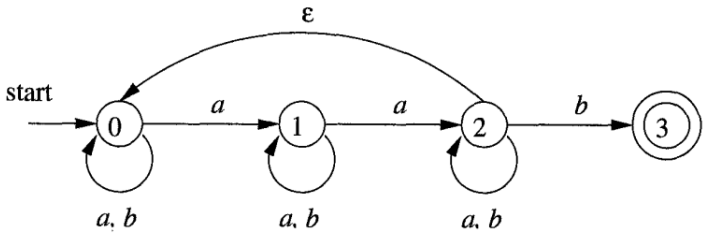


Figure 3.29: NFA for Exercise 3.6.3

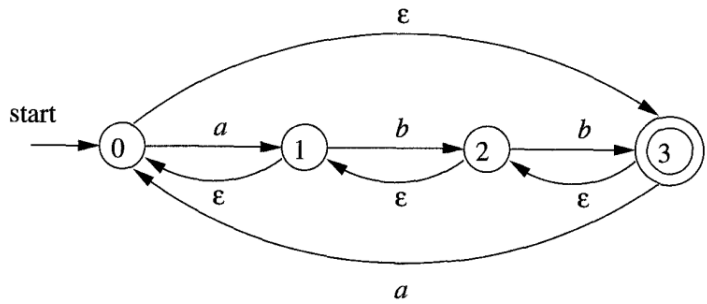


Figure 3.30: NFA for Exercise 3.6.4

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.

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

- No existen movimientos bajo ϵ.
- 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

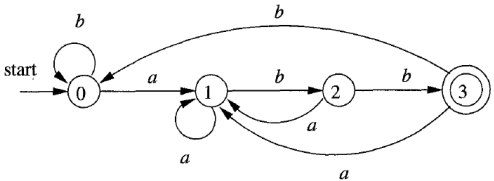
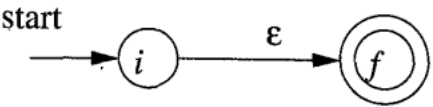


Figure 3.28: DFA accepting **(a|b)*abb**

Construcción de un AFN a partir de una expresión regular

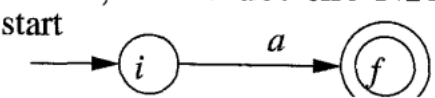
$r = \epsilon$

BASIS: For expression ϵ construct the NFA



$r = a$

For any subexpression a in Σ, construct the NFA



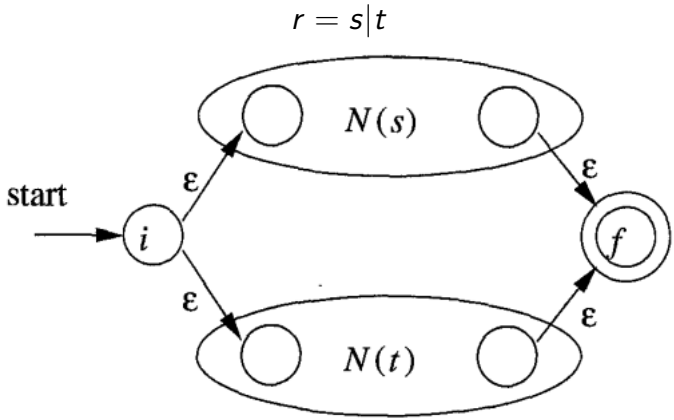


Figure 3.40: NFA for the union of two regular expressions

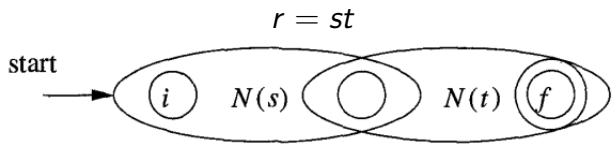


Figure 3.41: NFA for the concatenation of two regular expressions

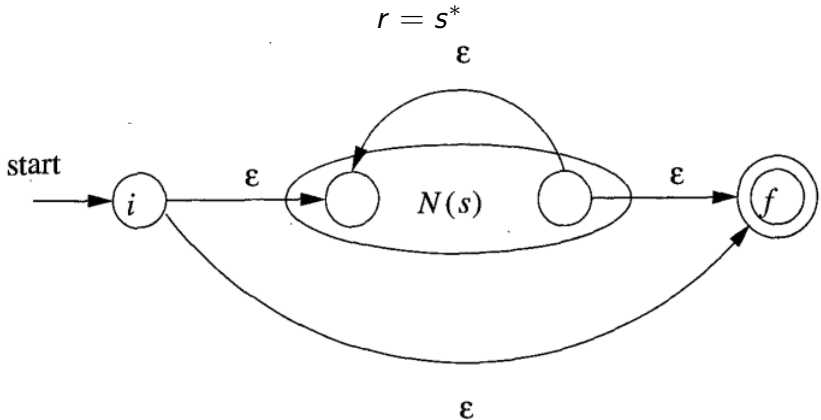


Figure 3.42: NFA for the closure of a regular expression

Construye el AFN de la siguiente expresión:

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

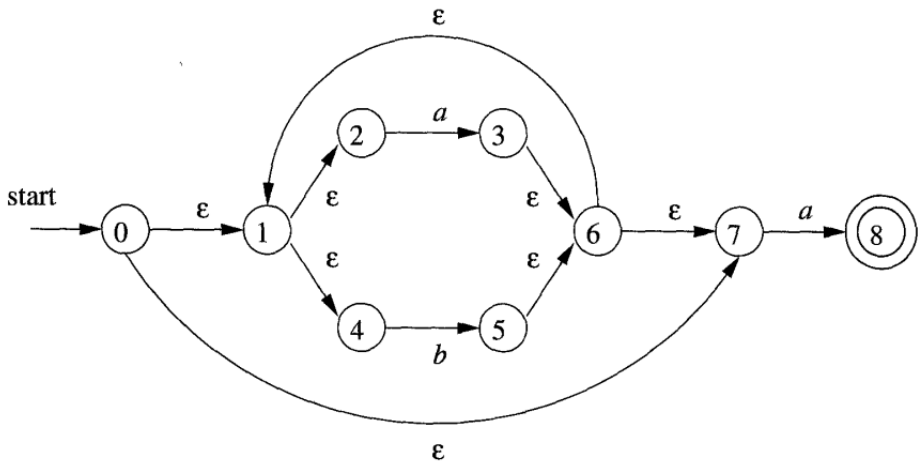


Figure 3.46: NFA for r_7

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

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

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

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

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.

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).
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 = ε-closure(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 ε-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 ε )
                if ( u is not in ε-closure(T) ) {
                        add u to ε-closure(T);
                        push u onto stack;
                }
}
```

Figure 3.33: Computing ϵ-*closure*(T)

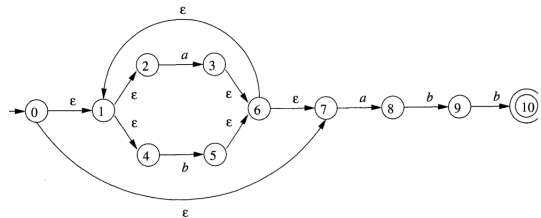


Figure 3.34: NFA N for (**a**|**b**)***abb**

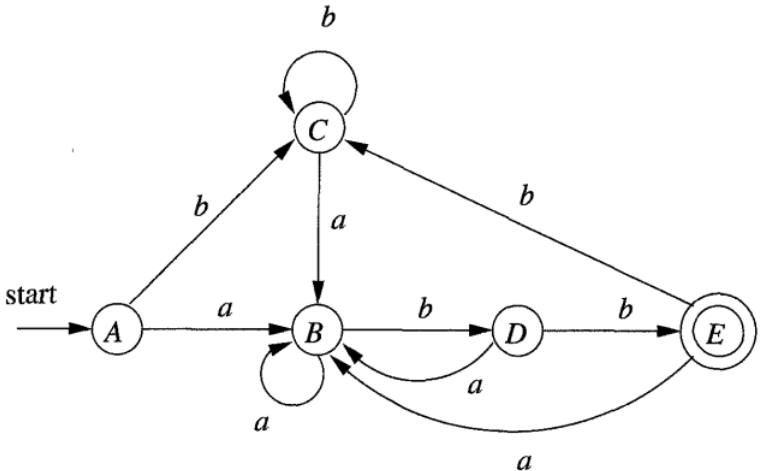


Figure 3.36: Result of applying the subset construction to Fig. 3.34

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

Revisión de la implementación de un analizador léxico:
`https://github.com/Manchas2k4/tc3002b`