Teoría de autómatas y lenguajes formales

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

pperezm@tec.mx

06-2022

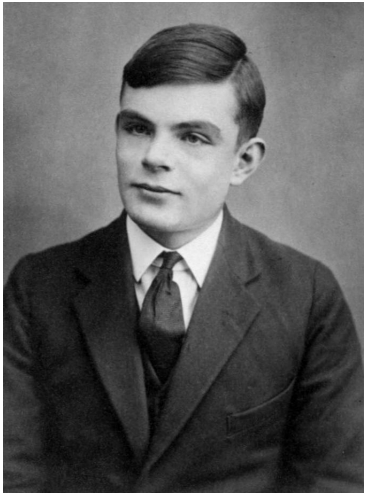
¿Qué es la teoría de la computación?

La teoría de la computación es el campo de las ciencias computacionales que trata con los problemas que pueden ser resueltos en un modelo de computación, usando un algoritmo, con qué eficiencia se pueden resolver o en qué grado (soluciones aproximadas o precisas). Este campo se divide en tres ramas:

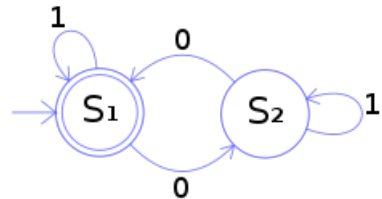
- Teoría de autómatas y lenguajes formales.
- Teoría de la computabilidad.
- Teoría de la complejidad computacional.

La teoría de autómatas es el estudio de dispositivos de cálculo abstractos (máquinas). Antes de que existieran las computadora, en la década de los treinta, Dr. Alan Turing estudió una máquina abstracta que tenía todas las capacidades de las computadoras de hoy en día.

El objetivo de Turing era describir de forma precisa los límites entre lo que una máquina de cálculo podía y no podía hacer.

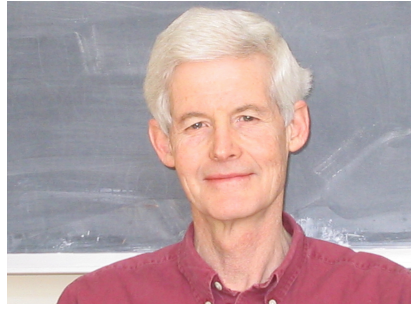


En las décadas de los cuarentas y cincuentas, una serie de investigadores estudiaron las máquinas más simples, las cuales todavía hoy denominamos "autómatas finito". Originalmente, estos autómata se propusieron para modelar el funcionamiento del cerebro y, posteriormente, resultado extremadamente útiles para muchos otros propósitos.



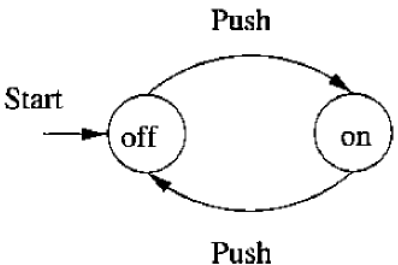
Fue en la década de los cincuenta que el lingüista N. Chomsky inició el estudio de las "gramática" formales. Aunque no son máquinas estrictamente, estas gramáticas están estrechamente relacionadas con los autómatas abstractos y sirven como base de algunos importantes componentes de software.

En 1969, S. Cook amplió el estudio realizado por Turing sobre lo que se podía y no se podía calcular. Cook fue capaz de separar aquellos problemas que se podían resolver de forma eficiente mediante computadora (Problemas P) de aquellos problemas que, en principio, pueden resolverse, pero que en la práctica consumen tanto tiempo que las computadoras son inútiles para todo, excepto para casos muy simples (Problemas NP).

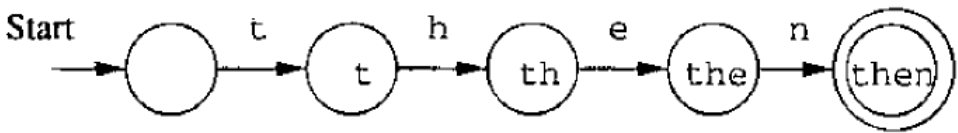


¿Por qué estudiar la teoría de autómatas?

De manera informal, existe muchos sistemas o componentes que pueden ser representados por un número finito de estados. El proceso de un estado es recordar la porción relevante de la historia del sistema. Dado que puede llegar a existir una gran cantidad de estados, el sistema debe ser diseñado para recordar a que es importante. Y, pues que es una cantidad infinita de estados, es posible implementar un sistema con una cantidad fija de recursos. Un ejemplo, sería el autómata de un apagador.



En algunos casos, lo que debe recordar cada estado es más complejo que la opción de encendido/apagado. Por ejemplo, el autómata que reconoce la palabra reservada"then":



Los autómatas finitos constituyen un modelo útil para muchos tipos de hardware y software.

- Software para diseñar y probar el comportamiento de circuitos digitales.
- El "analizador léxico" de un compilador típico, es decir, el componente del compilador que separa el texto de entrada en unidades lógicas, tal como identificadores, palabras clave y signos de puntuación.
- Software para explorar cuerpos de texto largos, como colecciones de páginas web, o para determinar el número de apariciones de palabras, frases u otros patrones.
- Software para verificar sistemas de todo tipo que tengan un número finito de estados diferentes, tales como protocolos de comunicaciones o protocolos para el intercambio seguro de información.

Un autómata de estado finitos es una quíntupla $(Q, \Sigma, \delta, q_0, F)$, tal que Q es un conjunto finito de estados, Σ es un alfabeto de entrada, $q_0 \in Q$ es el estado inicial, $F \subseteq Q$ es el conjunto de estados finales y δ es la función de transición que proyecta $Q \times \Sigma$. Es decir, $\delta(q, a)$ es un estado para cada estado q y cada símbolo de entrada a.

Existe dos importante notaciones que no son las utilizadas normalmente con los autómatas, pero que desempeñan un importante papel en el estudio de los autómatas y sus aplicaciones.

1. Las gramáticas son modelos útiles en el diseño de software que sirve para procesar datos con una estructura recursiva. El ejemplo más conocido es el e una "analizador sintáctico" (parser).
2. Las expresiones regulares también especifican la estructura de los datos, especialmente de las cadenas de texto. Los patrones de cadenas de caracteres que pueden describir expresiones regulares son los mismos que pueden ser descritos por los autómatas finitos.

Los autómatas son esenciales para el estudio de los límites de la computación. Existen dos factores importantes a este respecto:

1. ¿Qué puede hacer una computadora? Esta área de estudio se conoce como "decidibilidad", y los problemas que una computadora que pueden resolver se dice que son "decidibles".
2. ¿Qué puede hacer una computadora de manera eficiente? Esta área se conoce como "intratabilidad", y los problemas que una computadora puede resolver en un tiempo proporcional a alguna función que crezca lentamente como el tamaño de la entrada son de "crecimiento lento", mientras que se considera que las funciones que crecen más rápido que cualquier función polinómica crecen con demasiada rapidez.

Un ejemplo: analizador léxico

- 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.

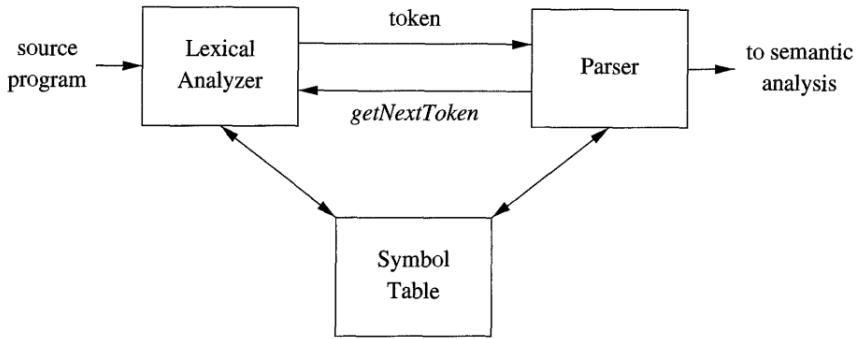


Figure 3.1: Interactions between the lexical analyzer and the parser

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
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

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.

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>

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;
}
```

- 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, 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 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 ϵ 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 ϵ son sufijos de $banana$.
- *Un substring* de s se obtiene borrando cualquier prefijo y sufijo de s. Por ejemplo, $banana$, nan y ϵ 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 ϵ 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 $banana$.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEME
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`, `6.02e2`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 3.2: Examples of tokens

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 ϵ.
- $L\,(L \cup D)^*$ 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.

- ϵ es una expresión regular, y $L(\mathbf{r})$ *es* ϵ, *esto es, el lenguaje cuyo único miembro es el string vacío.*
- *Si* a *es un símbolo de* Σ, *entonces* $\mathbf{a}$ *es una expresión regular, y* $L(\mathbf{a}) = a$, *esto es , el lenguaje con un string 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

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

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

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:

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

En resumen

Hagamos un resumen de los diferentes operadores existentes en una expresión regular

Resuelve los siguiente ejercicios:

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

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

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

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

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

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

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, c

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, b

Diagramas de transición

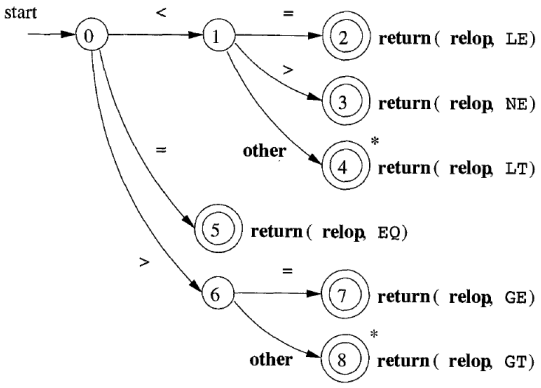


Figure 3.13: Transition diagram for **relop**

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

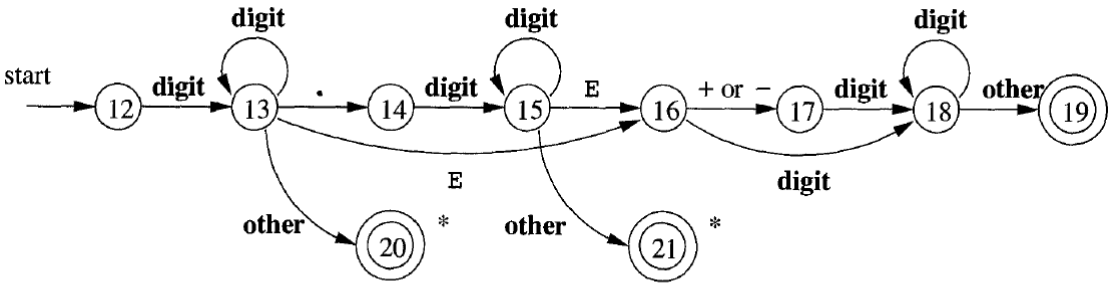


Figure 3.16: A transition diagram for unsigned numbers

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

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).

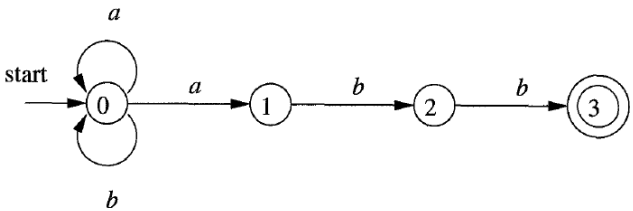


Figure 3.24: A nondeterministic finite automaton

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.

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

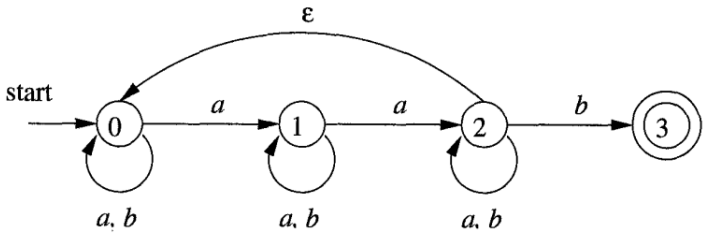


Figure 3.29: NFA for Exercise 3.6.3

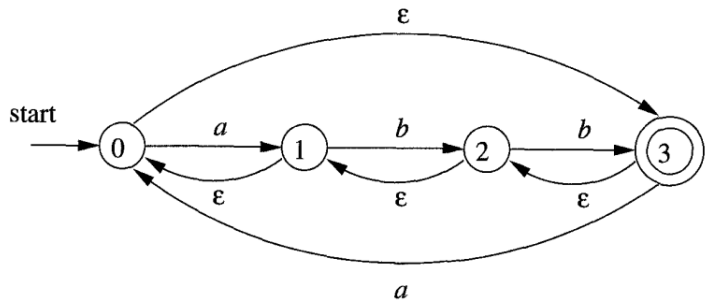


Figure 3.30: NFA for Exercise 3.6.4

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

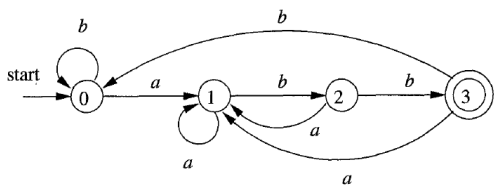


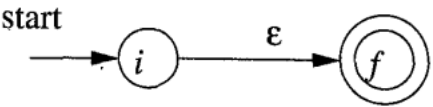
Figure 3.28: DFA accepting **(a|b)*abb**

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.

Construcción de un AFN a partir de una expresión regular

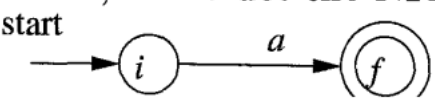
$r = \epsilon$

BASIS: For expression ϵ construct the NFA



$r = a$

For any subexpression a in Σ, construct the NFA



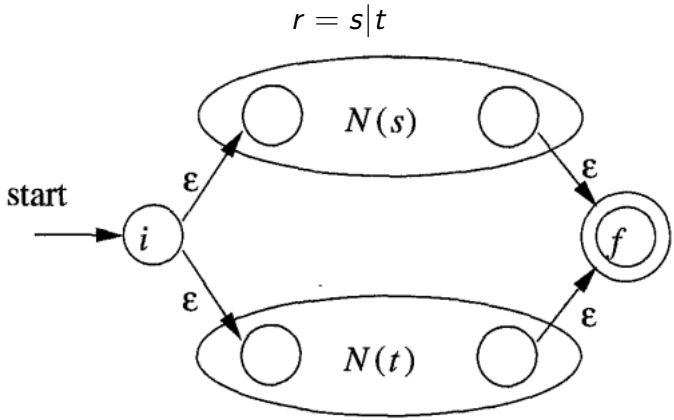


Figure 3.40: NFA for the union of two regular expressions

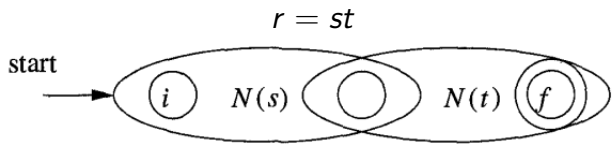


Figure 3.41: NFA for the concatenation of two regular expressions

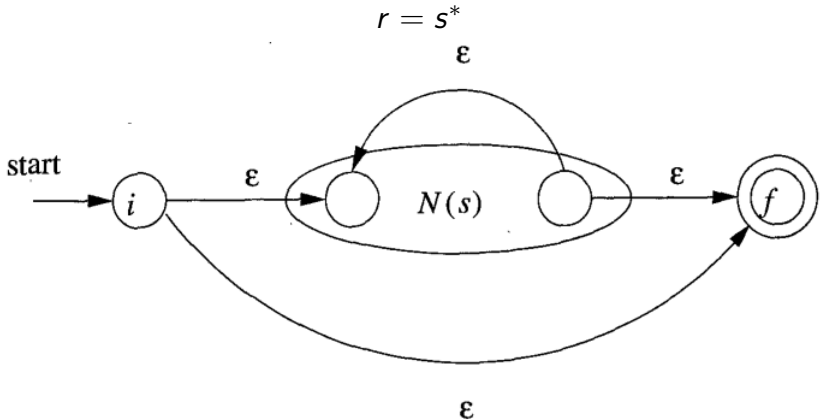


Figure 3.42: NFA for the closure of a regular expression

Construye el AFN de la siguiente expresión:

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

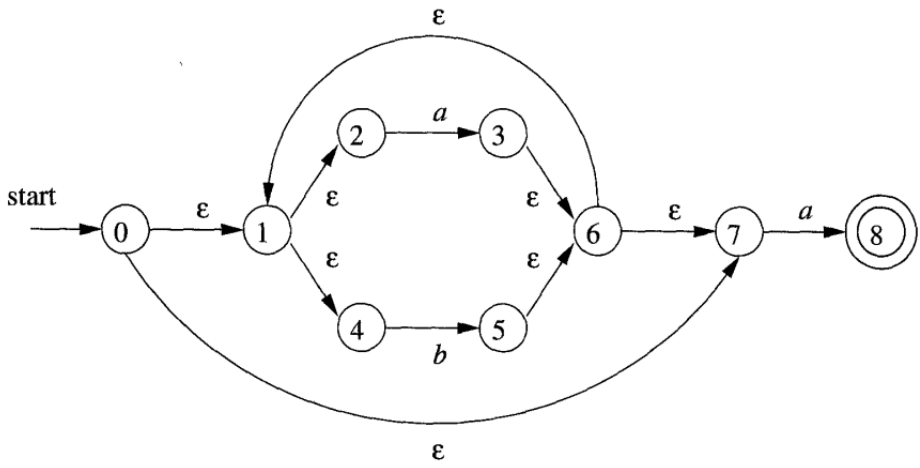


Figure 3.46: NFA for r_7

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

- Realizar incisos b, c, d.

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)

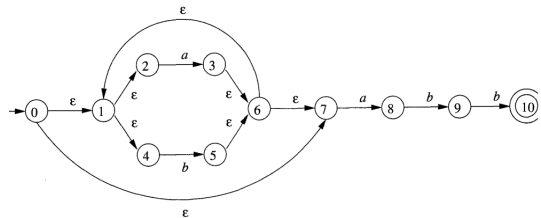


Figure 3.34: NFA N for (**a**|**b**)***abb**

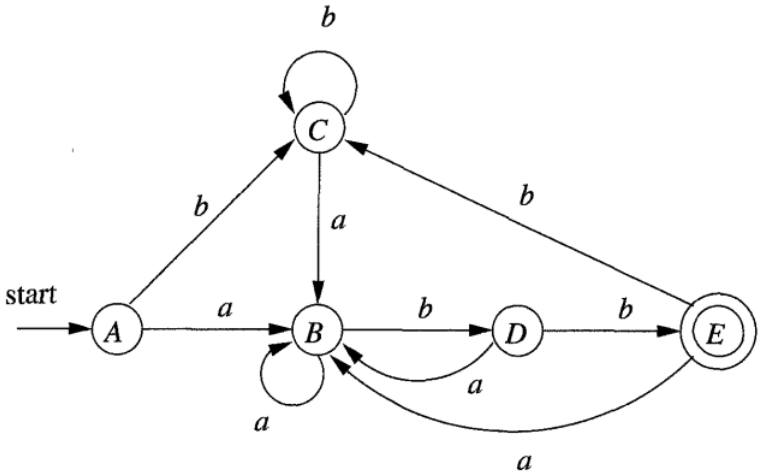


Figure 3.36: Result of applying the subset construction to Fig. 3.34

Ejercicio 6

- Realizar la conversión de los AFN generados en el ejercicio anterior (inciso c) a AFD.

Ver código en el repositorio