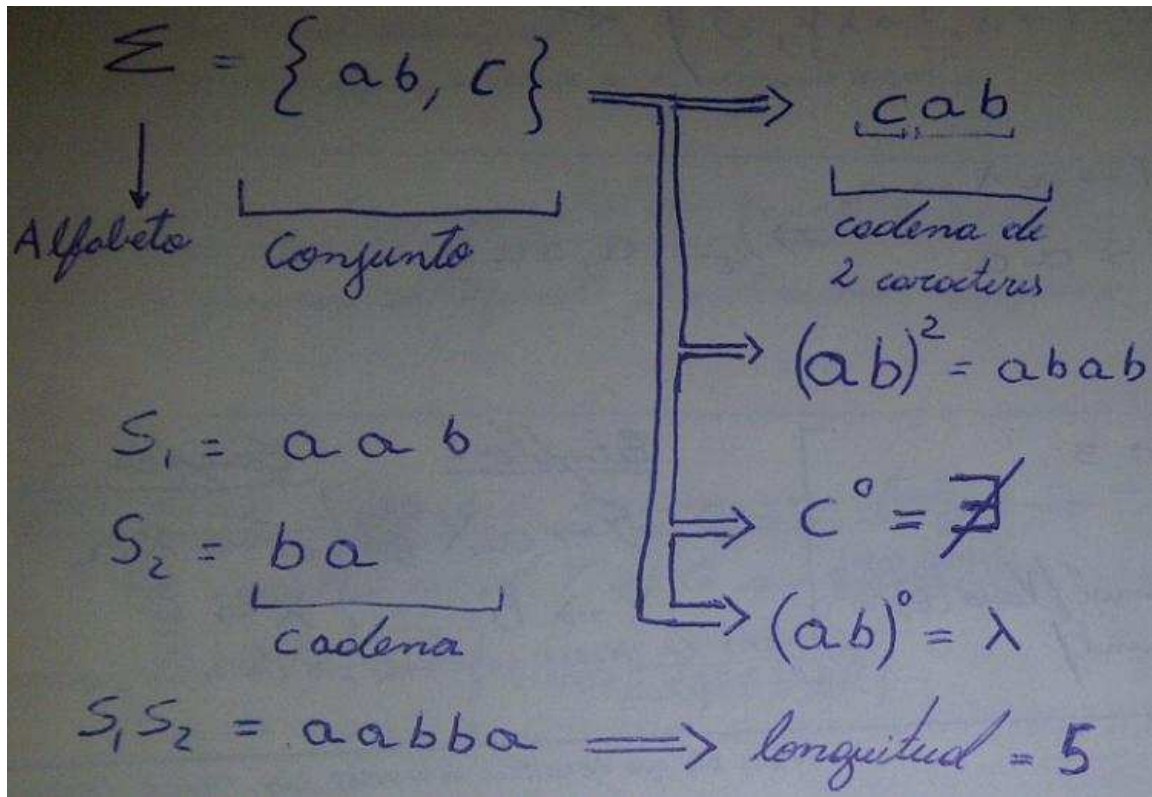


INDICE

INTRODUCCIÓN LENGUAJES	2
LENGUAJES FORMALES	3
LENGUAJES INFINITOS	3
GRAMÁTICA FORMAL	4
GRAMÁTICA REGULAR (GR) → TIPO 3	4
GRAMÁTICA REGULAR INFINITAS (GRI)	4
GRAMÁTICA QUASI-REGULAR (GQR)	5
GRAMÁTICA INDEPENDIENTE DEL CONTEXTO (GIC) → TIPO 2	5
GRAMÁTICA INDEPENDIENTE DEL CONTEXTO INFINITAS (GICI)	5
GRAMÁTICA SENSIBLES AL CONTEXTO (GSC) → TIPO 1	6
GRAMÁTICA IRRESTRICITAS (GI) → TIPO 0	6
DERIVACIÓN A IZQUIERDA Y DERECHA DE UNA GRAMÁTICA	7
BNF	8
EXPRESIONES REGULARES	9
Metalenguaje para Expresiones Regulares (metaERs)	11
AUTOMATAS	12
AUTÓMATA FINITO	13
Cláusula Positiva y Cláusula Potencia (kleene)	14
AUTÓMATA FINITO DETERMINÍSTICO	15
AUTÓMATA FINITO DETERMINÍSTICO COMPLETO	16
COMPLEMENTO DE UN AUTÓMATA FINITO DETERMINÍSTICO	17
INTERSECCIÓN DE DOS DE UN AUTÓMATAS FINITOS DETERMINÍSTICOS	18
AUTÓMATA FINITO NO DETERMINÍSTICO	20
AUTÓMATA FINITO NO DETERMINÍSTICO CON TRANSICIONES (ϵ)	21
ALGORITMO DE THOMPSON	22
ALGORITMO DE CLAUSURA (ϵ) ó CONSTRUCCION DE SUBCONJUNTOS	23
CONJUNTO "HACIA"	24
CONVERTIR UN AUTÓMATA FINITO NO DETERMINISTICO A UN AUTÓMATA FINITO DETERMINÍSTICO (AFN → AFD)	25
CONVERTIR UN AUTÓMATA FINITO A LA EXPRESIÓN REGULAR (AF → ER)	28
OBTENCIÓN AUTÓMATA FINITO MINIMO (AFD MINIMO)	30
MAQUINA DE TURING	35
AUTÓMATA FINITO CON PILA (AFP)	37
AUTÓMATA FINITO CON PILA DETERMINÍSTICO (AFPD)	38
PROCESOS DE COMPILACION	40
ANÁLISIS LÉXICO	41
ANÁLISIS SINTACTICO	42
ANÁLISIS SEMANTICO	43
TABLA DE SÍMBOLOS	44
CONSTRUCCIÓN DE PROCEDIMIENTOS DE ANÁLISIS SINTÁCTICO (PAS)	45
CONVERTIR UNA GIC EN GIC LL(1)	47
CONJUNTO PRIMERO	48
ANSI C	49

INTRODUCCIÓN LENGUAJES



Σ^* = todas las cadenas que se puedan formar

$a^+ = a \cdot a^*$ = aparece al menos una vez

LENGUAJE FORMAL

$$L_1 = \{ ccccp, cccpp, ppccc, ppppcc \}$$

LENGUAJE POR EXTENSIÓN

$$L_1 = \{ c^4p, c^3p^2, p^2c^3, p^4c \}$$

LENGUAJE POR COMPRENSIÓN

$$L_2 = \{ b^n / 0 \leq n \leq 8 \}$$

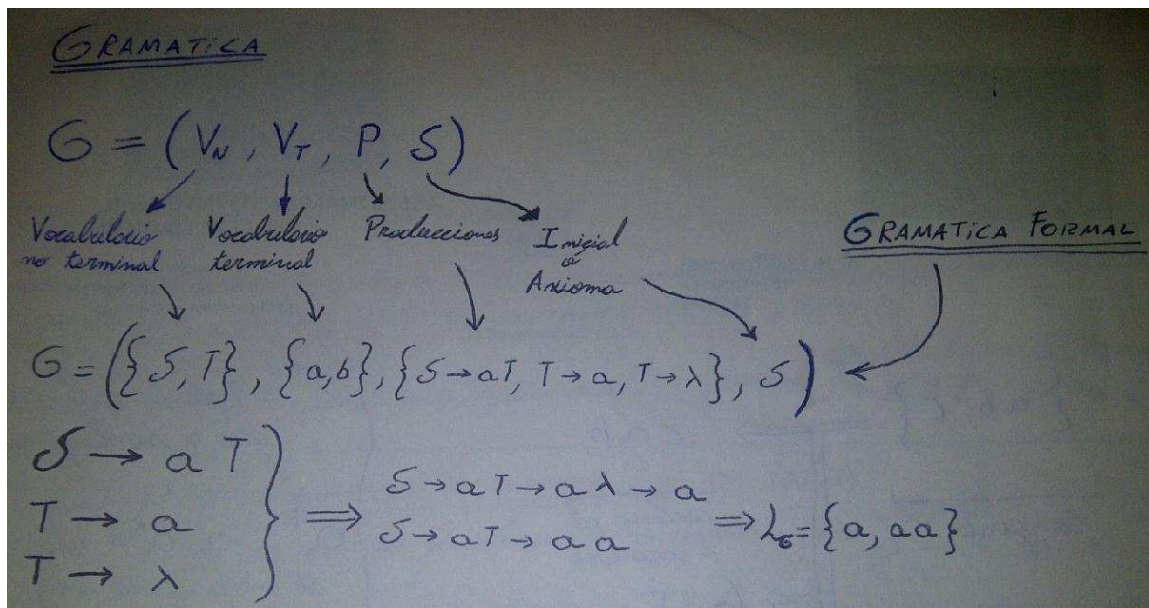
Por extensión sería $\rightarrow L_2 = \{ \lambda, b, b^2, \dots, b^8 \}$

LENGUAJE REGULAR INFINITO

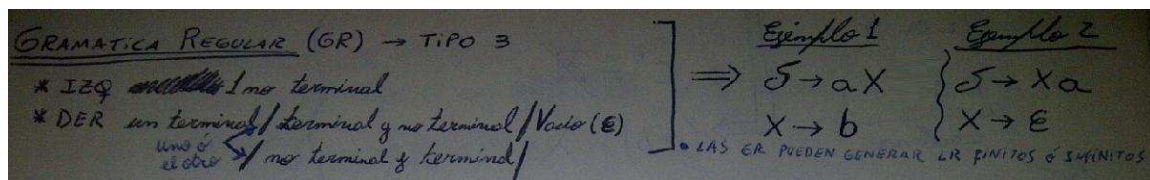
$$L_3 = \{ a^n b^n / n > 0 \}$$

- Un sublenguaje de un Lenguaje Regular Infinito (LRI) puede ser finito o infinito

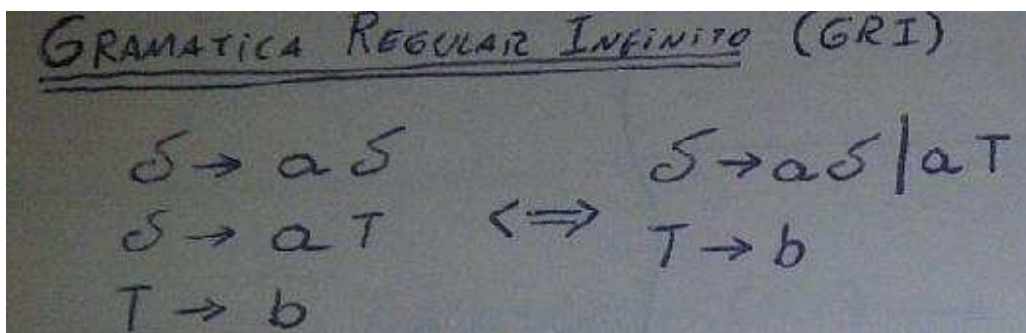
GRAMÁTICA FORMAL



GRAMÁTICA REGULAR



GRAMÁTICA REGULAR INFINITA



Las gramáticas regulares que generan lenguajes infinitos tienen producciones recursivas

GRAMÁTICA QUASI-REGULAR

GRAMÁTICA QUASI-REGULAR (GQR)
Son aquellos que no cumplen con una GR pero que al reescribirse sí lo son.
• Una GQR SIEMPRE PUEDE SER REESCRITA COMO UNA GR

\Rightarrow

Ejemplo (GQR)
 $S \rightarrow N | NS$
 $N \rightarrow a | b | c$
• Los gramáticas son equivalentes si generan el mismo lenguaje

Ejemplo (GR)
 $S \rightarrow a | b | c | aS | bS | cS$

GRAMÁTICA INDEPENDIENTE DEL CONTEXTO

GRAMÁTICA INDEPENDIENTE DEL CONTEXTO (GIC) \rightarrow TIPO 2
* IEQ. / no terminal
* DER. cualquier cosa en cualquier orden

\Rightarrow

Ejemplo 1
 $S \rightarrow ZXZ$

Ejemplo 2
 $S \rightarrow aEZbR$

GRAMÁTICA INDEPENDIENTE DEL CONTEXTO INFINITAS

GRAMÁTICAS INDEPENDIENTES DE CONTEXTO INFINITAS (GICI)
• * Se generan utilizando producciones recursivas

\Rightarrow

Ejemplo
 $S \rightarrow aSb | a$

Gramática Sensibles al Contexto

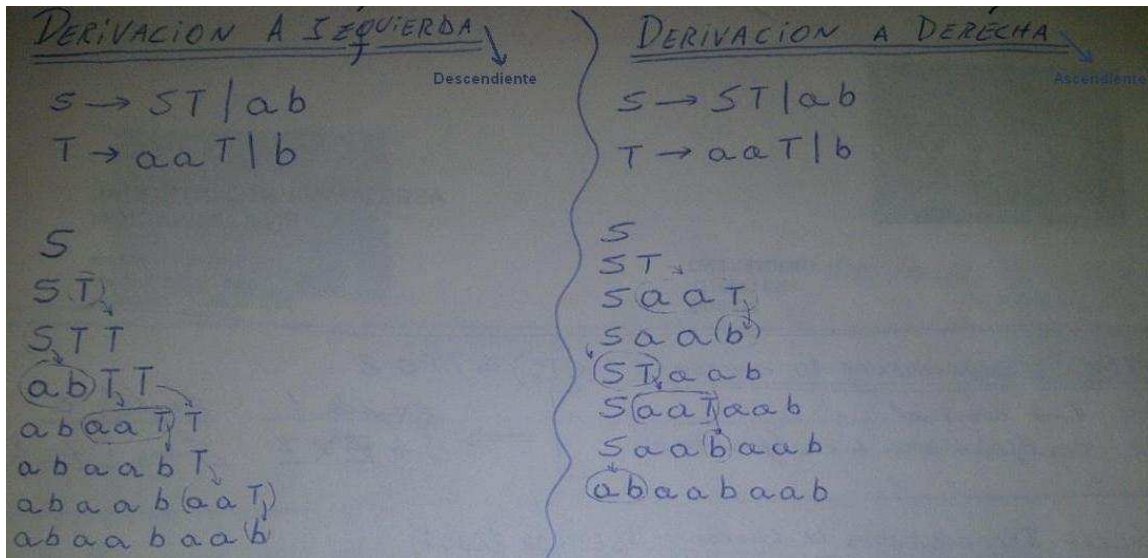
GRAMÁTICA SENSIBLE AL CONTEXTO (GSC) \rightarrow TIPO 1
* Son igual a los de TIPO 0 con ~~$|Izq| \leq |DER|$~~ $|Izq| \leq |DER|$
$$\left. \begin{array}{l} S \rightarrow ACaB \\ Ca \rightarrow aac \end{array} \right\} \Rightarrow |Izq| \leq |DER|$$

 ~~$AE \rightarrow \epsilon$~~

Gramática Irrestriccta

GRAMÁTICA IRRESTRICCTA (GI) \rightarrow TIPO 0
$$\begin{array}{l} S \rightarrow ACaB \\ Ca \rightarrow aac \\ CB \rightarrow DB | E \\ aD \rightarrow Da \\ AD \rightarrow AC \\ aE \rightarrow Ea \\ AE \rightarrow \epsilon \end{array}$$

Derivación de una Gramática



- Tanto la derivación a izquierda como a derecha generan una misma palabra
- La derivación es un proceso que permite obtener cada una de las palabras de un Lenguaje Formal a partir del axioma
- La derivación sirve para determinar si una cadena pertenece o forma parte del Lenguaje

Ejemplo:

Construya la Tabla de Derivación Vertical a Izquierda y la Tabla de Evaluación correspondiente para la expresión $2 + 4 * 6$. Utilizando las producciones de la siguiente GIC.

N° Producción	Producción
1	$E \rightarrow T$
2	$E \rightarrow E + T$
3	$E \rightarrow F$
4	$E \rightarrow T * F$
5	$E \rightarrow 2 4 6$

Tabla de Derivación Vertical a Izquierda

Producción Aplicada	Cadena de Derivación Obtenida
Axioma	E
2	E+T
1	T+T
3	F+T
5	2+T
4	2+T*F
3	2+F*F
5	2+4*F
5	2+4*6

Tabla de Evaluación

Cadena de Derivación a Reducir	Producción a Aplicar	Operación
2+4*6	5	
2+4*F6	5	
2+F4+F6	3	
2+T4+F6	4	$4 * 6 = 24$
2+T24	5	
F2+T24	3	
T2+T24	1	
E2+T24	2	$2 + 24 = 26$
E26	Axioma	Resultado Final

BNF

¿Como se construye una BNF?

- Los no terminales van encerrados entre los signos < >
- La flecha \rightarrow se cambia por dos veces dos punto y un igual $::=$
- Lo que esta encerrado entre llaves aparece cero o mas veces { }

Ejemplo de BNF ALGOL

`<identificador> ::= letra {<letra o digito>}`

`<letra o digito> ::= <letra> | <digito>`

`<letra> ::= a | b | c | A | B | C`

`<digito> ::= 0 | 1 | 2 | 3`

EXPRESIONES REGULARES

Importante: los lenguajes finitos siempre son regulares.

La Expresión Regular representa al lenguaje.

LENGUAJE	ER
$L = \emptyset$	\emptyset
$L = \{\epsilon\}$	ϵ
$L = \{a\}$	a
$L = \{aaaaaaa\}$	$aaaaaaa$ ó a^7
$L = \{a, \epsilon\}$	$a+\epsilon$
$L = \{aa, ba\}$	$aa+ba$
$L = \{a^n / 0 \leq n \leq 5000\}$	se representa por comprensión
$L = \{a^n / n \geq 0\}$	a^*
$L = \{a^n / n \geq 1\}$	a^+

Las ERs se construyen utilizando los operadores básicos *unión (+)*, *concatenación (.)* y *clausura de Kleene (*)*.

Los operadores potencia y clausura positiva no son básicos y se utilizan para simplificar la escritura de ERs.

Los operadores potencia y clausura tienen máxima prioridad, luego el operador concatenación y por último el operador unión.

La EXPRESIÓN REGULAR UNIVERSAL es la ER que representa al Lenguaje Universal sobre un alfabeto dado, representa al LR que contiene la palabra vacía y todas las palabras que se pueden formar con símbolos del alfabeto dado. Tienen una aplicación muy importante en la construcción de ERs que representan a un gran número de LR infinitos que se describen, por ejemplo, de la siguiente forma: "El LR formado por todas las palabras sobre el alfabeto $\{a, b\}$ que comienzan con a " o "El LR formado por todas las palabras sobre el alfabeto $\{a, b, c\}$ que tienen una única a y terminan con b ó c ".

Recordar que "terminar" no significa que deba existir al menos un carácter antes de b ó c .

Operaciones sobre LR_s y las ER_s correspondientes

Sean L_1 y L_2 dos LR_s representados por R_1 y R_2

Operación	Expresión Regular que la representa
$L_1 \cup L_2$ (unión)	$R_1 + R_2$
$L_1 L_2$ (concatenación)	$R_1 R_2$
L^* (clausura de Kleene)	R^*
L^+ (clausura positiva)	R^+
L^c (complemento de un LR con respecto al LU)	No existen operadores para describir el complemento de una ER, por ahora la escribiremos analizando el lenguaje
$L_1 \cap L_2$ (intersección)	No existen operadores para describir la intersección de dos ER _s , escribiremos la ER que represente a las palabras que pertenecen a los dos lenguajes.

Definición Regular

Algunas herramientas que trabajan con ER_s como datos, describen a las ER_s en la forma vista hasta el momento. Otras herramientas, en cambio, trabajan con lo que se llama DEFINICIONES REGULARES.

La Definición Regular es semejante a una expresión regular pero en lugar de los caracteres del alfabeto se utilizan nombres de expresiones regulares auxiliares.

Si hay que trabajar con una ER compleja o varias ER_s, se descompone la expresión total en expresiones más simples, cada una de las cuales recibe un nombre que la identifica.

Finalmente, la ER general es descripta usando estos nombres como si fueran símbolos del alfabeto.

Ejemplos:

Expresión Regular	Definición Regular (una posible)
$a + b$	$LA = a$ $LB = b$ $LA + LB$
$(a+b+c) (aa)^*$	$L = a+b+c$ $D = aa$ $L D^*$
$(1+2+3+4+5) * a + \epsilon$	$N = 1+2+3+4+5$ $L = a$ $N^* L + \epsilon$
Número real sin signo formado uno o mas dígitos y que finaliza con un punto	$N = 0+1+2+3+4+5+6+7+8+9$ $P = .$ $N^+ P$

Metalinguaje para Expresiones Regulares (metaERs)

Un metalinguaje es un lenguaje que se usa para describir otro lenguaje. El metalinguaje para las expresiones regulares utiliza operandos (caracteres del alfabeto), operadores (clausura, concatenación, unión, potencia y ?) y metacaracteres.

Recordar:

- ✓ Los operadores deben ser representados en la misma línea (no se permiten supraíndices ni subíndices)
- ✓ Los espacios en blanco son opcionales, aunque no pueden existir en una concatenación
- ✓ Los operadores unarios (*clausura de Kleene*, *clausura positiva*, *potencia* y *?*) tienen máxima prioridad.
- ✓ Si un símbolo utilizado como metacarácter es un carácter de una ER, se lo debe preceder de una "barra invertida" (ejemplo: \+).

Operadores

- | operador *unión* de expresiones
- { } operador *potencia*, repetición determinada
- {, } operador *potencia* extendido a un intervalo
- ? operador que indica cero o una ocurrencia de la expresión que lo precede
- * operador *clausura de Kleene*
- + operador *clausura positiva*

Metacaracteres

- . (punto) para representar un carácter
- [] clase de caracteres (ejemplo: [abc] representa la ER a+b+c)
- [-] clase de caracteres en un intervalo (ejemplo: [a-d] representa la ER a+b+c+d)
- () para agrupar una ER

El . (punto) y los [] se utilizan con caracteres, todos los demás se utilizan con expresiones.

Ejemplos:

ER	metaER	ER	metaER
a+b	a b	bbb + ε	b{3}?
a ⁺	a+ ó aa*	a + b*	a b *
(2+3+4+5).	([2-5]) \.	0 ⁺ 1 ⁺	0+1+
aaa+aa+a	a{1,3}	(ab)(ab)*	(ab) +

AUTOMATAS

Los autómatas son modelos matemáticos que reconocen palabras de un lenguaje y se lo define como una maquina abstracta que contiene estados y transiciones.

Aceptar: un autómata acepta una cadena una vez que pasa por las transiciones y llega al estado final.

Rechazar: un autómata rechaza una cadena cuando esa cadena no puede ser formada por el autómata.

Reconocer: un autómata reconoce una cadena cuando el autómata acepta la cadena y rechaza todas las que no.

Tipos de Autómatas

Gramática Formal	Lenguaje Formal	Autómata
GR	LR	Autómata Finito
GIC	LIC	Autómata Finito con Pila
GSC	LSC	Máquina de Turing
G. Irrestricta	L. Irrestricto	Máquina de Turing

AUTÓMATA FINITO

Un autómata finito es una herramienta abstracta que se utiliza para reconocer un único Lenguaje Regular.

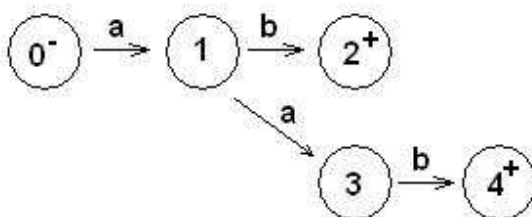
Un autómata finito esta compuesto por un único estado inicial (representado con el signo -), una serie de transiciones (\rightarrow) y uno o varios estados finales (representado con el signo +).

Cada transición tiene un estado de partida y un estado de llegada. En un CICLO el estado de partida coincide con el de llegada.

Autómata Finito que reconoce Lenguajes Regulares Finitos

Expresión Regular = $ab + aab$

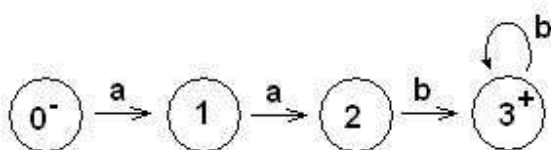
Lenguaje Regular = $a(b+ab)$



Autómata Finito que reconoce Lenguajes Regulares Infinitos

Expresión Regular = a^2b^+

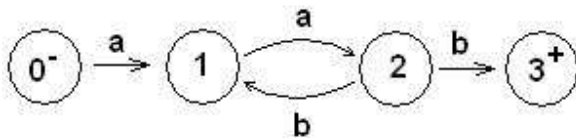
Lenguaje Regular = aab^+



Cláusula Positiva

La cláusula positiva (+) significa que el carácter o caracteres deben aparecer al menos una vez.

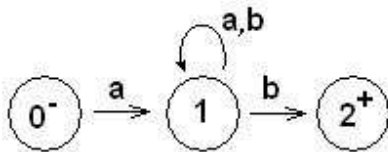
Expresión Regular = $a(ab)^+ab$



Cláusula Potencia (kleene)

La cláusula de potencia (*) significa que el carácter o caracteres puede no aparecer o aparecer una o varias veces.

Expresión Regular = $a(a+b)^+b$



AUTÓMATA FINITO DETERMINÍSTICO

Los autómatas finitos determinísticos son aquellos que para cualquier estado en que se encuentre el autómata en un momento dado, la lectura de un carácter determina (sin ambigüedades) cuál será el estado de llegada en la próxima transición.

Dos autómatas finitos determinísticos son equivalentes si reconocen el mismo lenguaje regular.

Definición Formal

$$M = (Q, \Sigma, T, q_0, F)$$

Q = conjunto finito de estados

Σ = alfabeto de caracteres

T = transiciones

q_0 = único estado inicial

F = conjunto de estados finales

Las transiciones puede ser representadas mediante una tabla de transiciones (TT)

Ej.

$$M = (Q, \Sigma, T, q_0, F)$$

$$Q = \{0, 1, 2, 3\}$$

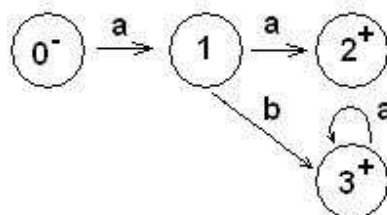
$$\Sigma = \{a, b\}$$

$$T = \{0 \Rightarrow a \Rightarrow 1, 1 \Rightarrow a \Rightarrow 2, 1 \Rightarrow b \Rightarrow 3, 3 \Rightarrow a \Rightarrow 3\}$$

$$q_0 = 0$$

$$F = \{2, 3\}$$

TT	a	b
0 ⁻	1	-
1	2	3
2 ⁺	-	-
3 ⁺	3	-



AUTÓMATA FINITO DETERMINÍSTICO COMPLETO

Un autómata finitos determinísticos es completo si cada estado tiene exactamente una transición por cada carácter del alfabeto, es decir, que su tabla de transiciones no tiene huecos (signo – ó vacío) en tal caso se denomina incompleto.

¿Cómo completar un Autómata Finito Determinístico Incompleto?

Para completar un AFD Incompleto hay que eliminar los “huecos” de su tabla de transiciones. Para ello se deben realizar los siguientes pasos:

- 1) Se agrega un nuevo estado, denominado estado de rechazo.
- 2) Se reemplaza cada “hueco” de la tabla de transición por el estado de rechazo.

Ej.

AFD Incompleto	AFD Completo																																	
$M = (Q, \Sigma, T, q_0, F)$ $Q = \{0, 1, 2, 3\}$ $\Sigma = \{a, b\}$ $T =$ representadas por TT $q_0 = 0$ $F = \{2, 3\}$	$M = (Q, \Sigma, T, q_0, F)$ $Q = \{0, 1, 2, 3, 4\}$ $\Sigma = \{a, b\}$ $T =$ representadas por TT $q_0 = 0$ $F = \{2, 3\}$																																	
<table><tr><th>TT</th><th>a</th><th>b</th></tr><tr><td>0⁻</td><td>1</td><td>-</td></tr><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>2⁺</td><td>-</td><td>-</td></tr><tr><td>3⁺</td><td>3</td><td>-</td></tr></table>	TT	a	b	0 ⁻	1	-	1	2	3	2 ⁺	-	-	3 ⁺	3	-	<table><tr><th>TT</th><th>a</th><th>b</th></tr><tr><td>0⁻</td><td>1</td><td>4</td></tr><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>2⁺</td><td>4</td><td>4</td></tr><tr><td>3⁺</td><td>3</td><td>4</td></tr><tr><td>4</td><td>4</td><td>4</td></tr></table>	TT	a	b	0 ⁻	1	4	1	2	3	2 ⁺	4	4	3 ⁺	3	4	4	4	4
TT	a	b																																
0 ⁻	1	-																																
1	2	3																																
2 ⁺	-	-																																
3 ⁺	3	-																																
TT	a	b																																
0 ⁻	1	4																																
1	2	3																																
2 ⁺	4	4																																
3 ⁺	3	4																																
4	4	4																																

COMPLEMENTO DE UN AUTÓMATA FINITO DETERMINÍSTICO

El complemento de un Autómata Finito Determinístico da como resultado un autómata finito determinístico y se obtiene de la siguiente forma:

- 1) Al estado inicial agregarlo también como estado final
- 2) Invertir los estados finales por los no finales

Ej.

Sea $M_2 = (\{5,6,7,8\}, \{a,b\}, T_2, 5, \{8\})$, con la función T_2 representada por la TT:

TT	a	b
5⁻	-	6
6	7	7
7	8	8
8⁺	8	8

Para hallar el AFD complementado de M_2 , primero debemos completar el autómata dado agregando el estado de rechazo (en este caso el estado 9)

AFD = M_2			AFD Complementado = M_2^C		
TT₁	a	b	TT₂	a	b
5⁻	9	6	5⁻	9	6
6	7	7	6⁺	7	7
7	8	8	7⁺	8	8
8⁺	8	8	8	8	8
9	9	9	9⁺	9	9

$$M_2^C = (\{5,6,7,8,9\}, \{a,b\}, T, 5, \{5,6,7,9\})$$

INTERSECCIÓN DE DOS DE UN AUTÓMATAS FINITOS DETERMINÍSTICOS

La intersección de dos Autómata Finito Determinístico da como resultado un autómata finito determinístico y reconoce las palabras comunes a los lenguajes regulares reconocidos por ambos autómatas. Los estados del AFD intersección se obtiene de los pares ordenados de los estados de ambos autómatas.

Ej.

Sea $M_1 = (\{0,1,2,3,4\}, \{a,b\}, T_1, 0, \{2,4\})$, con la función T_1 representada por la TT:
 Sea $M_2 = (\{5,6,7,8\}, \{a,b\}, T_2, 5, \{8\})$, con la función T_2 representada por la TT:

AFD (M_1)			AFD (M_2)		
TT	a	b	TT	a	b
0⁻	1	3	5⁻	-	6
1	2	1	6	7	7
2⁺	2	1	7	8	8
3	3	4	8⁺	8	8
4⁺	3	4			

1) Se colocan en la tabla como pares ordenados los estados iniciales.

TT	a	b
(0,5)⁻	-	(3,6)
(3,6)	?	?

Nota: Como se observa $0 \rightarrow a \rightarrow 1$ pertenece a M_1 mientras que en M_2 $5 \rightarrow a$ no contiene elemento de transición es por ello que no se agrega a la tabla TT intersección.

2) Seguir completando los pares ordenados

TT	a	b
(0,5)⁻	-	(3,6)
(3,6)	(3,7)	(4,7)
(3,7)	?	?
(4,7)	?	?

3) Una vez completada la tabla se deben hallar los estados que tienen el mismo comportamiento y simplificarlo por uno de ellos.

Dos o mas estados tienen el mismo comportamiento si:

- a) Todos son estados no finales \ Todos son estados finales
- b) Sus filas son idénticas.

TT	a	b
(0,5)⁻	-	(3,6)
(3,6)	(3,7)	(4,7)
(3,7)	(3,8)	(4,8)
(4,7)	(3,8)	(4,8)
(3,8)	(3,8)	(4,8)
(4,8)⁺	(3,8)	(4,8)

Simplificamos los estados que tienen el mismo comportamiento y obtenemos la siguiente tabla:

TT	a	b
(0,5)⁻	-	(3,6)
(3,6)	(3,7)	(3 ,7)
(3,7)	(3, 7)	(4,8)
(4,8)⁺	(3,8)	(4,8)

Nota: Al simplificar hay que cambiar (forzar) los valores para que coincidan.

4) Renombramos los estados para hallar el AFD mínimo

TT	a	b
0⁻	-	1
1	2	2
2	2	3
3⁺	3	3

AUTÓMATA FINITO NO DETERMINÍSTICO

Un autómata finito no determinístico es aquel que para algún o algunos estados hay mas de una transición por un mismo carácter, o si tiene alguna o algunas transiciones ϵ .

La diferencia entre la definición formal de un AFND y la de un AFD está en la función de transiciones.

Para todo Autómata Finito No Determinístico existe un Autómata Finito Determinístico equivalente.

Definición Formal

$$M = (Q, \Sigma, T, q_0, F)$$

Q = conjunto finito de estados

Σ = alfabeto de caracteres

T = transiciones que puede ser representadas mediante una tabla (TT)

q_0 = único estado inicial

F = conjunto de estados finales

Ej.

$$M = (Q, \Sigma, T, q_0, F)$$

$$Q = \{0, 1, 2\}$$

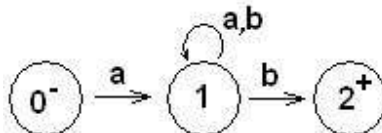
$$\Sigma = \{a, b\}$$

T = representada por TT

$$q_0 = 0$$

$$F = \{2\}$$

TT	a	b
0 ⁻	{1}	-
1	{1}	{1,2}
2 ⁺	-	-



AUTÓMATA FINITO NO DETERMINÍSTICO CON TRANSICIONES (ϵ)

Los Autómatas finitos no determinísticos con transiciones " ϵ " son AFNs y se caracterizan por la existencia de una o mas transiciones que ocurren sin que el autómata lea el próximo carácter de la cadena que está analizando.

Una transición (ϵ) representa un cambio de estado repentino, sin que intervenga ningún carácter del alfabeto.

Si un AF tiene transiciones (ϵ) entonces en la tabla hay una columna para el símbolo ϵ .

Definición Formal

$$M = (Q, \Sigma, T, q_0, F)$$

Q = conjunto finito de estados

Σ = alfabeto de caracteres

T = transiciones que pueden ser representadas en una tabla (TT)

q_0 = único estado inicial

F = conjunto de estados finales

Ej.

$$M = (Q, \Sigma, T, q_0, F)$$

$$Q = \{A, B\}$$

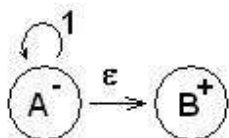
$$\Sigma = \{1\}$$

T = representada por TT

$$q_0 = A$$

$$F = \{B\}$$

TT	1	ϵ
A ⁻	{A}	{B}
B ⁺	-	-



ALGORITMO DE THOMPSON

El algoritmo de Thompson consiste en 3 partes:

- 1) Descomponer la Expresión Regular en sus componentes básicos (caracteres, operadores y ϵ)
- 2) Generar un Autómata Finito básico por cada carácter o símbolo ϵ .
- 3) Componer los autómatas básicos según los operadores existentes en la Expresión Regular, hasta lograr el Autómata Finito que reconoce a la Expresión Regular dada. Y por ultimo se debe completar el autómata para que tenga un UNICO estado final


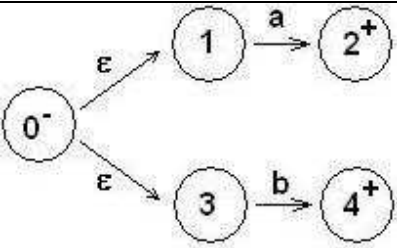
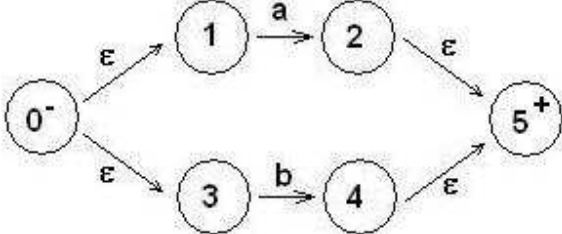
¿Para que sirve el Algoritmo de Thompson?

El Algoritmo de Thompson sirve para construir un Autómata Finito en forma mas rápida y menos compleja, pero no algorítmica. Facilitando la construcción de un Autómata Finito No Determinístico a partir de una Expresión Regular.

Nota:

- > Todos los autómatas contruidos siguiendo el algoritmo de Thompson tienen las mismas características fundamentales.
- > El algoritmo de Thompson trabaja únicamente con los operadores oficiales de las Expresiones Regulares (unión, concatenación y clausura de Kleene *)

Ejemplo: Expresión Regular = $a+b$

1) Descomposición	
2) Generar autómata finito básico	
3) Componer los autómatas básicos y completarlo para que tenga un único estado final.	

ALGORITMO DE CLAUSURA (ϵ) ó CONSTRUCCION DE SUBCONJUNTOS

El conjunto de Clausura (ϵ) de un estado nunca puede ser vacío porque contiene como mínimo a su propio estado, es decir, si un estado no tiene transiciones (ϵ), la clausura (ϵ) de ese estado solo contiene al mismo estado.

¿Cómo determinar las clausuras a partir de una TT?

Ej.

TT	a	ϵ
0 ⁻	{1}	{3,6}
1	-	{2}
2 ⁺	-	{4}
3	-	{4}
4	{5}	-
5 ⁺	-	-
6 ⁺	{6}	-

1) Comenzamos por aquellas que no tienen transiciones en la columna (ϵ)

$$\text{Clausura-}\epsilon(4) = \{4\}$$

$$\text{Clausura-}\epsilon(5) = \{5\}$$

$$\text{Clausura-}\epsilon(6) = \{6\}$$

2) Analizamos aquellas que tengan una única transición en la columna (ϵ)

Explicación: El estado 1 tiene una transición- ϵ al estado 2, quien a su vez tiene una transición- ϵ al estado 4. Entonces del estado 1 se puede llegar al estado 2 y 4 utilizando transición- ϵ por lo que la Clausura- ϵ (1) queda compuesta por 1, 2 y 4.

$$\text{Clausura-}\epsilon(1) = \{1,2,4\}$$

$$\text{Clausura-}\epsilon(2) = \{2,4\}$$

$$\text{Clausura-}\epsilon(3) = \{3,4\}$$

3) Analizamos aquellas que tengan mas de una transición en la columna (ϵ)

$$\text{Clausura-}\epsilon(0) = \{0,3,4,6\}$$

CONJUNTO “HACIA”

Sea R un conjunto de estados y sea “ x ” un símbolo del alfabeto. Entonces, $hacia(R,x)$ es el conjunto de estados a los cuales se transita por el símbolo x .
Ósea el conjunto de estados de llegada para (R,x) .

Ej.

Si R es un conjunto que está formado por los estados 1, 2 y 3 compuestas por las siguientes transiciones: $1 \rightarrow a \rightarrow 4$, $2 \rightarrow a \rightarrow 5$, $3 \rightarrow b \rightarrow 6$

$Hacia(R,a) = \{4,5\}$

$Hacia(R,b) = \{6\}$

CONVERTIR UN AUTÓMATA FINITO NO DETERMINÍSTICO A UN AUTÓMATA FINITO DETERMINÍSTICO (AFN → AFD)

Expresión Regular → Algoritmo de Thompson → AFN →

Algoritmo de Clausura (ϵ) → Conjunto "Hacia" → AFD →

Algoritmo de Clases → AFD Mínimo

Ejemplo: Expresión Regular = $(a + b)^* ab$

1) Partiendo de la Expresión Regular, aplicamos el Algoritmo de Thompson para obtener un Autómata Finito No Determinístico, el cual nos muestra la siguiente tabla de transiciones:

TT	a	b	ϵ
0 ⁻	-	-	{1,7}
1	-	-	{2,3}
2	{4}	-	-
3	-	{5}	-
4	-	-	{6}
5	-	-	{6}
6	-	-	{1,7}
7	-	-	{8}
8	{9}	-	-
9	-	{10}	-
10	-	-	{11}
11 ⁺	-	-	-

2) Aplicamos el Algoritmo de Clausura (ϵ) y una vez obtenidas las clausuras armamos la tabla de estados de transición conservando el estado inicial.

CLAUSURAS APARTIR DE LA TT	TABLA DE ESTADO											
Clausura-ε (2) = {2}	<table><tr><th>Estado</th><th>a</th><th>b</th></tr><tr><td>{0,1,2,3,7,8} ^</td><td>?</td><td>?</td></tr><tr><td>?</td><td>?</td><td>?</td></tr></table>			Estado	a	b	{0,1,2,3,7,8} ^	?	?	?	?	?
Estado				a	b							
{0,1,2,3,7,8} ^				?	?							
?				?	?							
Clausura-ε (3) = {3}												
Clausura-ε (8) = {8}												
Clausura-ε (10) = {10,11}												
Clausura-ε (11) = {11}												
Clausura-ε (4) = {1,2,3,4,6,7,8}												
Clausura-ε (5) = {1,2,3,5,6,7,8}												
Clausura-ε (7) = {7,8}												
Clausura-ε (9) = {9}												
Clausura-ε (0) = {0,1,2,3,7,8}												
Clausura-ε (1) = {1,2,3}												
Clausura-ε (6) = {1,2,3,6,7,8}												

3) Aplicamos el método de conjuntos “hacia” para cada uno de los caracteres del alfabeto (en este caso ‘a’ y ‘b’).

Como se observa en la tabla 2 $2 \rightarrow a \rightarrow 4$, $8 \rightarrow a \rightarrow 9$ entonces el conjunto ‘hacia’ queda formado por:

$$\text{Hacia}(\{0,1,2,3,7,8\},a) = \{4,9\}$$

Como se observa en la tabla 3 $3 \rightarrow b \rightarrow 5$, $9 \rightarrow b \rightarrow 10$ entonces el conjunto ‘hacia’ queda formado por:

$$\text{Hacia}(\{0,1,2,3,7,8\},b) = \{5,10\}$$

4) Hallamos la de Clausura (ϵ) de cada conjunto ‘Hacia’ hallado en el punto 3 para obtener los estado de llegada de las respectivas transiciones.

Clausura- ϵ ($\text{Hacia}(\{0,1,2,3,7,8\},a)$) es lo mismo que escribir Clausura- ϵ ($\{4,9\}$) que a su vez es lo mismo que Clausura- ϵ (4) \cup Clausura- ϵ (9) = $\{1,2,3,4,6,7,8\} \cup \{9\} = \{1,2,3,4,6,7,8,9\} = \{1-4,6-9\}$

Clausura- ϵ ($\text{Hacia}(\{0,1,2,3,7,8\},b)$) es lo mismo que escribir Clausura- ϵ ($\{5,10\}$) que a su vez es lo mismo que Clausura- ϵ (5) \cup Clausura- ϵ (10) = $\{1,2,3,5,6,7,8\} \cup \{10,11\} = \{1,2,3,5,6,7,8,10,11\}$

5) Completamos la fila del estado inicial y las columnas de la tabla de transiciones con los valores hallados en el punto anterior.

Estado	a	b
$\{0,1,2,3,7,8\}$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8,10,11\}$
$\{1-4, 6-9\}$?	?
$\{1,2,3,5,6,7,8,10,11\}$?	?

6) Volvemos a repetir los pasos 3 y 4 para completar la tabla.

> Hacia($\{1-4,6-9\},a$) = $\{4,9\}$ = Clausura- ϵ (4) U Clausura- ϵ (9) = **$\{1-4, 6-9\}$**

> Hacia($\{1-4,6-9\},b$) = $\{5,10\}$ = Clausura- ϵ (5) U Clausura- ϵ (10) = **$\{1,2,3,5,6,7,8,10,11\}$**

Estado	a	b
$\{0,1,2,3,7,8\}^-$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8,10,11\}$
$\{1-4, 6-9\}$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8,10,11\}$
$\{1,2,3,5,6,7,8,10,11\}$?	?

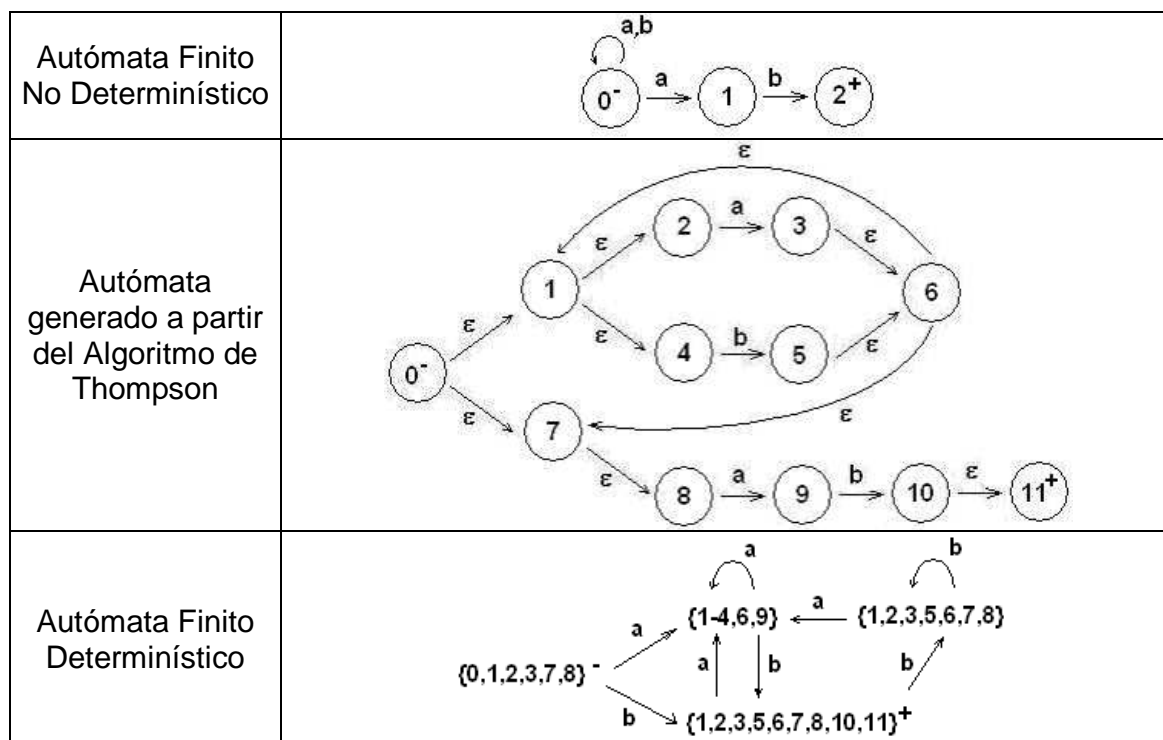
> Hacia($\{1,2,3,5,6,7,8,10,11\},a$) = $\{4,9\}$ = Clausura- ϵ (4) U Clausura- ϵ (9) = **$\{1-4, 6-9\}$**

> Hacia($\{1,2,3,5,6,7,8,10,11\},b$) = $\{5\}$ = Clausura- ϵ (5) = **$\{1,2,3,5,6,7,8\}$**

> Hacia($\{1,2,3,5,6,7,8\},a$) = $\{4,9\}$ = Clausura- ϵ (4) U Clausura- ϵ (9) = **$\{1-4, 6-9\}$**

> Hacia($\{1,2,3,5,6,7,8\},b$) = $\{5\}$ = Clausura- ϵ (5) = **$\{1,2,3,5,6,7,8\}$**

Estado	a	b
$\{0,1,2,3,7,8\}^-$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8,10,11\}$
$\{1-4, 6-9\}$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8,10,11\}$
$\{1,2,3,5,6,7,8,10,11\}^+$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8\}$
$\{1,2,3,5,6,7,8\}$	$\{1-4, 6-9\}$	$\{1,2,3,5,6,7,8\}$



CONVERTIR UN AUTÓMATA FINITO A LA EXPRESIÓN REGULAR (AF → ER)

Existe un método que permite obtener la Expresión Regular de un lenguaje tomando como punto de partida la Tabla de Transiciones de un Autómata Finito (puede ser determinístico o no determinístico, completo o incompleto) que reconoce las palabras del lenguaje.

El método consta de 4 pasos:

- 1) Depuración
- 2) Planteo del sistema de ecuaciones
- 3) Reducción
- 4) Resolución del sistema de ecuaciones

Ej.

Sea la siguiente, la TT de un AF:

Estado	a	b
0 ⁺	1	2
1 ⁺	1	4
2	3	4
3 ⁺	3	4
4	4	4
5	3	4

1 - Depuración

Se deben eliminar los estados inalcanzables y de rechazo.

El 5 es un estado inalcanzable porque ningún otro estado puede llegar a él.

El 4 es un estado de rechazo porque la fila esta completa por el mismo valor.

Estado	a	b
0^+	1	2
1^+	1	4
2	3	4
3^+	3	4

2 - Planteo del sistema de ecuaciones

$0 = a1 + b2 + \epsilon$ (a los estados finales se les agrega una producción por ϵ)

$1 = a1 + \epsilon$

$2 = a3$

$3 = a3 + \epsilon$

3 - Reducciones

Se deben describir las producciones recursivas utilizando el operador *

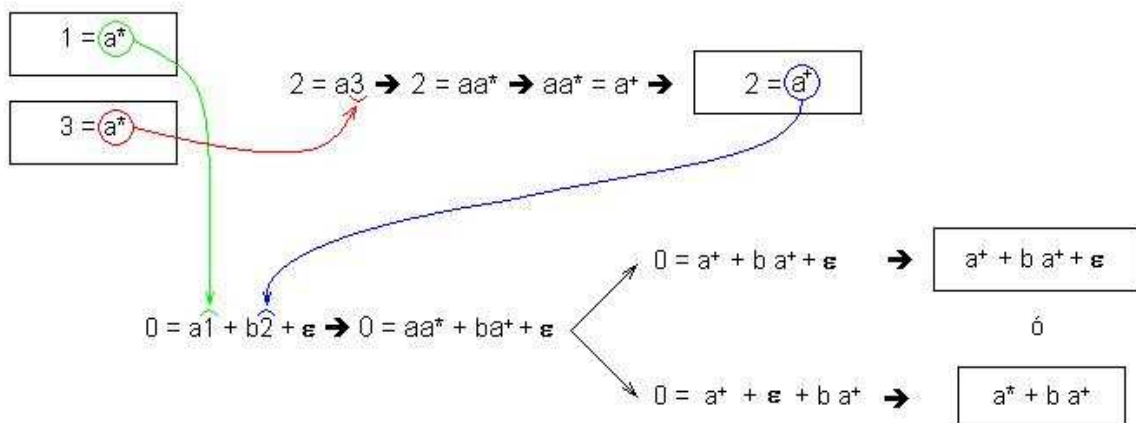
$1 = a^* \cdot \epsilon = a^* \rightarrow 1 = a^*$

$3 = a^* \cdot \epsilon = a^* \rightarrow 3 = a^*$

4 - Resolución del sistema

La ecuación del estado inicial es la última que se resuelve.

DATOS



OBTENCIÓN AUTÓMATA FINITO MINIMO (AFD MINIMO)

El autómata finito determinístico mínimo es el AFD con la mínima cantidad de estados que reconoce a un Lenguaje Regular.

> Dos o mas AFDs son equivalentes si el AFD mínimo que se obtiene a partir de ellos es el mismo.

> Dos o mas Expresiones Regulares son equivalentes si son reconocidas por el mismo AFD mínimo.

Ej.

Expresión Regular = $a^*ba+aba^*$

TT	a	b
0 ⁻	1	2
1	3	4
2	7	-
3	3	2
4 ⁺	5	-
5 ⁺	6	-
6 ⁺	6	-
7 ⁺	-	-

1) Se debe completar la tabla con el estado de rechazo...

TT	a	b
0 ⁻	1	2
1	3	4
2	7	8
3	3	2
4 ⁺	5	8
5 ⁺	6	8
6 ⁺	6	8
7 ⁺	8	8
8	8	8

2) Se debe particionar el conjunto de estados en dos clases: una para los estados no finales y otra para los estados finales.

TT	a	b	CLASES
0 ⁻	1	2	Clase C0
1	3	4	
2	7	8	
3	3	2	
8	8	8	
4 ⁺	5	8	Clase C1
5 ⁺	6	8	
6 ⁺	6	8	
7 ⁺	8	8	

3) Verificamos si existen estados con el mismo comportamiento y los simplificamos.

TT	a	b	CLASES
0 ⁻	1	2	Clase C0
1	3	4	
2	7	8	
3	3	2	
8	8	8	
4 ⁺	5	8	Clase C1
5 ⁺	5	8	
7 ⁺	8	8	

Nota: Al simplificar ha quedaría $5 \rightarrow a \rightarrow 6$ pero 6 fue simplificado por ello $5 \rightarrow a \rightarrow 5$.

Como se observa en la tabla al simplificar y modificar los valores se puede volver a simplificar los estados 4 y 5, entonces la tabla queda de la siguiente forma:

TT	a	b	CLASES
0 ⁻	1	2	Clase C0
1	3	4	
2	7	8	
3	3	2	
8	8	8	
4 ⁺	4	8	Clase C1
7 ⁺	8	8	

4) Construimos la tabla de transiciones por clases (TTC).

0⁻ → a → 1 ... el estado 1 pertenece a la Clase C0 por lo que se debe reemplazar el estado 1 por su clase.

2⁻ → a → 7 ... el estado 7 pertenece a la Clase C1 por lo que se debe reemplazar el estado 7 por su clase.

TTC	a	b	CLASES
0 ⁻	C0	C0	Clase C0
1	C0	C1	
2	C1	C0	
3	C0	C0	
8	C0	C0	
4 ⁺	C1	C0	Clase C1
7 ⁺	C0	C0	

3) Reordenamos la TTC de acuerdo a las clases y volvemos a particionar si es necesario.

TTC	a	b	CLASES
0 ⁻	C0	C0	Clase C0
3	C0	C0	
8	C0	C0	
1	C0	C1	Clase C2
2	C1	C0	Clase C3
4 ⁺	C1	C0	Clase C1
7 ⁺	C0	C0	Clase C4

Nota: Las Clases C1 y C3 están divididas ya que 2 es un elemento no terminal y 4⁺ es un elemento terminal

4) Actualizamos la TTC con los valores de la TT.

TTC	a	b	CLASES
0 ⁻	C2	C3	Clase C0
3	C0	C3	
8	C0	C0	
1	C0	C1	Clase C2
2	C1	C0	Clase C3
4 ⁺	C1	C0	Clase C1
7 ⁺	C0	C0	Clase C4

TT	a	b	CLASES
0 ⁻	1	2	Clase C0
1	3	4	
2	7	8	
3	3	2	
8	8	8	
4 ⁺	4	8	Clase C1
7 ⁺	8	8	

0⁻ → a → 1 ... el estado 1 pertenece a la Clase C0 para la tabla TT pero en la tabla TTC el estado 1 se encuentra en la Clase C2 por lo que ahora 0⁻ → a → **C2**

3 → b → 2 ... el estado 2 pertenece a la Clase C0 para la tabla TT pero en la tabla TTC el estado 2 se encuentra en la Clase C3 por lo que ahora 3 → b → **C3**

4) Reconstruimos la TT para obtener el AFD mínimo sin completar (sin el estado de rechazo)

TT	a	b
0⁻	1	2
1	3	4
2	7	-
3	3	2
4⁺	4	-
7⁺	-	-

MAQUINA DE TURING

La clase de autómatas que ahora se conoce como máquinas de Turing fue propuesta por Alan Turing en 1936. Las máquinas de Turing (MT) se asemejan a los AF en que constan de un mecanismo de control y un flujo de entrada que concebimos como una cinta; la diferencia es que las MT pueden mover sus cabezas de lectura hacia delante y hacia atrás y pueden leer o escribir en la cinta. La cinta tiene un extremo izquierdo pero se extiende indefinidamente hacia la derecha. Si la MT intenta ir mas allá del extremo izquierdo la ejecución de la máquina sufrirá una terminación anormal.

Una MT puede emplear su cinta como almacenamiento auxiliar y además de hacer operaciones de inserción y extracción, puede rastrear los datos de la cinta y modificar las celdas que desee sin alterar las demás.

La MT trabaja con dos alfabetos: el alfabeto de la máquina que son los símbolos en el que están codificados los datos de entrada iniciales y el alfabeto de la cinta que son marcas especiales.

El espacio en blanco pertenece al conjunto de símbolos de la cinta, si la MT tiene que borrar una celda escribe en ella un espacio en blanco. Usaremos el símbolo para representar el espacio en blanco

La MT contiene un mecanismo de control que en cualquier momento puede encontrarse en uno de entre un número finito de estados.

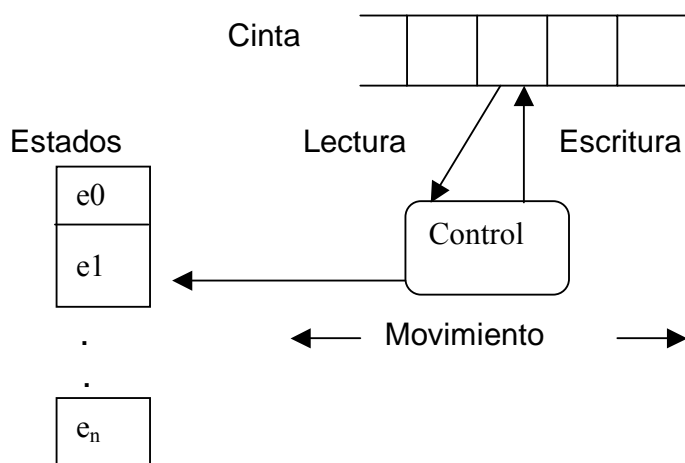
Uno de estos estados es el estado inicial y representa el estado en el cual la MT comienza los cálculos.

Otro de los estados se conoce como estado de parada, una vez que la máquina llega a ese estado terminan todos los cálculos. El estado de parada de una MT difiere de los estados de aceptación de los AF en que éstos pueden continuar sus cálculos después de llegar a un estado de aceptación, mientras que una MT debe detenerse en el momento en que llegue a su estado de parada.

El estado inicial de una MT no puede ser a la vez el estado de parada, por lo tanto toda MT debe tener cuando menos dos estados.

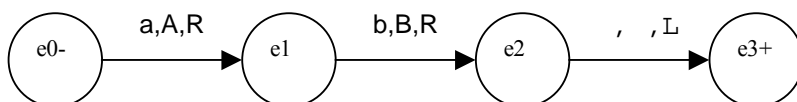
Las acciones específicas que puede realizar una MT consisten en operaciones de escritura y de movimiento. La operación de escritura consiste en reemplazar un símbolo en la cinta con otro símbolo (puede ser el mismo símbolo que se leyó) y luego cambiar a un nuevo estado (el cual puede ser el mismo donde se encontraba antes). La operación de movimiento comprende mover la cabeza una celda a la derecha o a la izquierda y luego pasar a un nuevo estado (que puede ser igual al de partida).

El siguiente esquema representa la arquitectura de la Máquina de Turing:



a) Diagrama de transiciones de una MT que reconoce $L = \{ab\}$.

El autómata mas apropiado para reconocer palabras de este lenguaje es el Autómata Finito. Para este ejemplo usaremos la MT que tiene la capacidad de reconocer palabras de cualquier lenguaje formal.



Los elementos que constituyen esta MT son:

- 1) un alfabeto A de símbolos del lenguaje a reconocer;
 $A = \{a, b\}$
- 2) una cinta infinita con la cadena ab ;
- 3) una cabeza de cinta ;
- 4) un alfabeto A' de símbolos que pueden ser escritos en la cinta por la cabeza de cinta;
 $A' = \{A, B\}$
- 5) un conjunto finito de estados
- 6) un conjunto de reglas que representan las transiciones

b) Reglas del programa de la MT que reconoce $L = \{a^n b^n / n \geq 1\}$.

El autómata mas apropiado para reconocer palabras de este lenguaje es el Autómata Finito con Pila. Como en el ejemplo anterior usaremos la MT que tiene la capacidad de reconocer palabras de cualquier lenguaje formal.

$e0 - a, A, R - e1$
 $e0 - B, B, R - e3$
 $e1 - a, a, R - e1$
 $e1 - B, B, R - e1$
 $e1 - b, B, L - e2$
 $e2 - a, a, L - e2$
 $e2 - B, B, L - e2$
 $e2 - A, A, R - e0$
 $e3 - B, B, R - e3$
 $e3 - , , L - e4+$

c) Reglas del programa de la MT que reconoce $L = \{a^n b^n c^n / n \geq 1\}$.

Esta MT está desarrollada en el libro.

AUTOMATAS FINITOS CON PILA (AFP)

$M = (E, A, A', T, e_0, p_0, F)$

E	Conjunto de estados
A	Alfabeto de entrada (cadena a analizar)
A'	Alfabeto de la pila
T	Transiciones
e_0	Estado inicial de la pila
p_0	Símbolo inicial de la pila
F	Conjunto de estados finales

Ejemplo de Transiciones

$T(4, a, Z) = \{(4, RPZ), (5, \epsilon)\}$

T(Se encuentra en el estado 4, lee el carácter 'a', en la cabeza de la pila se encuentra 'Z')
= {(Se queda en el estado 4 , agrega 'RP' a la cabeza de la pila) ó (se mueve al estado 5 ,
quita el símbolo Z de la cabeza de la pila)}

Nomenclaturas

fdc = fin de cadena

\$ = pila lógicamente vacía

ϵ = no se agrega elemento a la pila o quitar elemento de la pila (pop)

NOTA:

Un AFP puede reconocer un LIC de dos maneras, por estado final (como en los autómatas finitos) o por pila vacía.

AUTÓMATA FINITO CON PILA DETERMINÍSTICO (AFPD)

- Los AFPD reconocen el lenguaje de los operadores en ANSI C.
- Los AFPD permiten hacer movimientos sin leer del flujo de entrada.

$$T(e, a, R) = (e', \alpha)$$

T(se encuentra en el estado "e", lee el símbolo de entrada "a", tiene el símbolo "R" en la cabeza de la pila) entonces (pasa por el estado e', reemplaza en la pila el símbolo "R" por la secuencia de símbolos α) y adelanta una posición en el flujo de entrada.

$$T(e, \epsilon, R) = (e', \alpha)$$

T(se encuentra en el estado "e", lee el símbolo de entrada " ϵ ", tiene el símbolo "R" en la cabeza de la pila) entonces (pasa al estado e', reemplaza en la pila el símbolo "R" por la secuencia de símbolos α) y no adelanta ninguna posición en el flujo de entrada.

NOTA:

Para que un AFP sea determinístico, solo uno de los dos tipos de movimientos se puede dar para cualquier par (estado, símbolo en la cabeza de la pila)

Ejemplo de Transiciones

$$e0, \$ \Rightarrow a \Rightarrow e1, R\$$$

Comienza en el estado inicial "e0" con la pila vacía "\$".

Lee el carácter "a".

Se mueve al estado "e1" y agrega una "R" a la cabeza de la pila.

$$e1, R \Rightarrow b \Rightarrow e2, \epsilon$$

Comienza en el estado inicial "e1" y en la cabeza de la pila una "R".

Lee el carácter "b".

Se mueve al estado "e2" y quita el primer elemento de la pila "R".

Diagrama de Transiciones

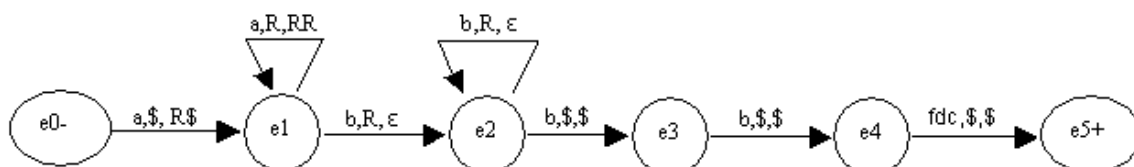
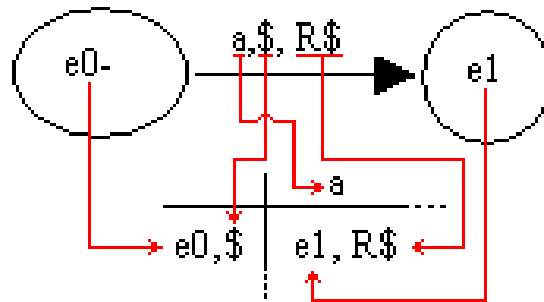


Tabla de Movimientos (TM)

Así se completa la tabla...



Así quedaría completa la tabla de movimientos del diagrama de transiciones de la hoja anterior.

TM	a	b	fdc
e0, \$	e1, R\$		
e1, R	e1, RR	e2, ε	
e2, R		e2, ε	
e2, \$		e3, \$	
e3, \$		e4, \$	
e4, \$			e5, \$

Proceso de compilación (1º Parte)

Un compilador es un programa que lee un programa en un lenguaje y lo traduce a un programa equivalente en otro lenguaje y además informa al usuario sobre la presencia de errores en el programa de entrada.

Se divide en dos fases: una parte que analiza la entrada y genera estructuras intermedias y otra parte que sintetiza la salida.

La fase de ANÁLISIS comprende al análisis Léxico, Sintáctico y Semántico.

La fase de SÍNTESIS se ocupa de la Generación de código intermedio, la Optimización de código y la Generación de código de máquina.

Proceso de compilación (2º Parte)

Sabemos que en el proceso de compilación, el Parser invoca al Scanner cada vez que necesita un nuevo token y que verifica si los tokens que recibe sucesivamente forman sentencias válidas. **El Parser es el módulo que realiza el análisis sintáctico**; hay dos formas de análisis: el Descendente y el Ascendente.

El **análisis sintáctico descendente recursivo** (ASDR) es un método descendente en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada. A cada no terminal de una gramática se asocia un procedimiento. Una forma especial de análisis sintáctico descendente recursivo, llamado **análisis sintáctico predictivo**, utiliza el símbolo de preanálisis para determinar sin ambigüedad el procedimiento seleccionado para cada no terminal. La secuencia de procedimientos llamados en el procesamiento de la entrada define implícitamente un árbol de análisis sintáctico para la entrada. El análisis sintáctico predictivo depende de la información sobre los primeros símbolos que pueden ser generados por el lado derecho de una producción

Para utilizar el ASDR, la gramática no puede ser recursiva a Izquierda. Se necesita una gramática LL. Para trabajar con análisis sintáctico predictivo la gramática debe ser LL(1). Estas gramáticas necesitan un solo "símbolo de preanálisis" (si un no terminal tiene varias producciones, el parser puede decidir cuál aplicar con solo conocer el próximo símbolo de entrada).

Análisis léxico

Consiste en detectar palabras de los Lenguajes Regulares que forman parte del Lenguaje de Programación, esta tarea es realizada por el analizador léxico o Scanner.



La función primordial es agrupar caracteres de la entrada en tokens

Estos tokens son suministrados ("bajo demanda") al analizador sintáctico

Los tokens se pasan como valores "simples", por ejemplo: Identificador, Operador, Constante, CaracterPuntuación, etc.

Algunos tokens requieren algo más que su propia identificación, por ejemplo, las constantes requieren su valor, los identificadores requieren el String; es decir que el scanner debe realizar, a veces, una doble función: identificar el token y evaluarlo.

En el proceso para detectar tokens, el scanner puede encontrar errores; a estos errores se los denomina Errores Léxicos.

Errores léxicos típicos son:

1. nombre ilegales de identificadores: un nombre contiene caracteres inválidos.
2. números incorrectos: un número contiene caracteres inválidos o no está formado correctamente, por ejemplo 3,14 en vez de 3.14 o 0.3.14

Los errores léxicos se deben a descuidos del programador. En general, la recuperación de errores léxicos es sencilla y siempre se traduce en la generación de un error de sintaxis que será detectado más tarde por el analizador sintáctico cuando el analizador léxico devuelve un componente léxico que el analizador sintáctico no espera en esa posición.

Los métodos de recuperación de errores léxicos se basan bien en saltarse caracteres en la entrada hasta que un patrón se ha podido reconocer; o bien usar otros métodos más sofisticados que incluyen la inserción, borrado, sustitución de un carácter en la entrada o intercambio de dos caracteres consecutivos.

Análisis sintáctico

Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados.

Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas independientes de contexto (GIC).

El análisis sintáctico impone una estructura jerárquica a la cadena de componentes léxicos, que se representará por medio de árboles sintácticos.

Los árboles se construyen con los tokens que envía el scanner.

El Parser es el encargado de hacer el análisis sintáctico.

Existen dos formas de análisis sintáctico:

- análisis DESCENDENTE (Top-Down)

 - »construye el árbol desde la raíz (S) hasta llegar a las hojas

- análisis ASCENDENTE(Bottom-Up)

 - »construye el árbol desde las hojas hacia la raíz (S)

El Análisis Sintáctico Descendente Recursivo (ASDR) debe su nombre a que utiliza rutinas que pueden ser recursivas.

El Árbol de Análisis Sintáctico (AAS) que se construye tiene las siguientes propiedades:

La raíz del AAS está etiquetada con el axioma de la GIC.

Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.

Cada nodo interior está etiquetado con un noterminal.

La forma más natural de implementar este analizador descendente es asociar un procedimiento con cada no terminal (PAS)

Cada procedimiento comprueba la corrección del token en estudio con el que el propio procedimiento representa y si es incorrecto (error sintáctico) actúa en consecuencia (mensaje de error y/o estrategia de recuperación).

Análisis semántico

Completa el trabajo del análisis sintáctico. La tarea es realizada por las rutinas semánticas, ellas se encargan de chequear la semántica estática de cada construcción y generar código para una máquina virtual.

El Analizador Semántico finaliza la fase de Análisis del compilador y comienza la fase de Síntesis, en la cual se comienza a generar el código objeto.

Las rutinas semánticas son llamadas por el parser.

El chequeo semántico se encarga, por ejemplo, de comprobar si los tipos de datos que intervienen en las expresiones son compatibles, si los argumentos de una función o procedimiento coinciden en tipo y cantidad con los parámetros, si un identificador ha sido declarado antes de utilizarlo, también verifica que no exista declaración repetida de identificadores.

Los errores que acabamos de mencionar pertenecen a la semántica estática, son los errores que se pueden determinar en tiempo de compilación.

Existen otros errores que sólo se manifiestan durante la ejecución, por ejemplo una división por cero o un subíndice fuera de rango.

Tabla de Símbolos

La tabla de símbolos es una estructura de datos que contiene toda la información relativa a cada identificador que aparece en el programa fuente. Los identificadores pueden ser nombres de variables, tipos de datos, funciones, procedimientos, etc. Evidentemente cada lenguaje de programación tiene unas características propias que se reflejan en la tabla de símbolos.

Cada elemento de la estructura de datos que compone la tabla de símbolos está formado al menos por el identificador y sus atributos. Los atributos son la información relativa de cada identificador.

Algunos de los atributos habituales son:

- ✓ Especificación del identificador: variable, tipo de datos, función, procedimiento, etc.
- ✓ Tipo: en el caso de variables será el identificador de tipo (real, entero, carácter, cadena de caracteres, o un tipo definido previamente). En el caso de las funciones puede ser el tipo devuelto por la función.
- ✓ Dimensión o tamaño. Puede utilizarse el tamaño total que ocupa una variable en bytes, o las dimensiones.

Los atributos pueden variar de unos lenguajes a otros, debido a las características propias de cada lenguaje y a la metodología de desarrollo del compilador.

La tabla de símbolos se crea por cooperación del análisis léxico y el análisis sintáctico.

El análisis léxico proporciona la lista de los identificadores, y el análisis sintáctico permite rellenar los atributos correspondientes a cada identificador.

El analizador semántico también puede rellenar algún atributo.

El analizador semántico y el generador de código obtienen de la tabla de símbolos la información necesaria para realizar su tarea.

Construcción de Procedimientos de Análisis Sintáctico (PAS)

Los PAS reflejan las definiciones dadas por las producciones de la gramática sintáctica.

Recordemos lo visto en el Volumen I sobre GIC's:

"En general: sean v y v'' no terminales y sea t un terminal. Entonces las producciones de una GIC corresponden a este formato general:

$v \rightarrow (v'' + t)^$, donde v y v'' pueden representar el mismo noterminal"*

La estructura de cada PAS sigue el desarrollo del lado derecho de la producción. Las producciones pueden ser recursivas o no.

Producciones no recursivas

Los terminales y no terminales son procesados en el orden en que aparecen.

a) Del lado derecho hay una única sentencia

Ejemplo...

```
<A> -> <E> <C> PUNTO
void A (void) {
    E();
    C();
    Match (PUNTO);
}
```

b) Del lado derecho hay más de una sentencia

En estos casos es necesario averiguar cuál es el próximo token que recibirá el parser para decidir cuál sentencia se deberá procesar. Para averiguar cuál será el próximo token se usa la función ProximoToken().

Ejemplo...

<factor> -> IDENTIFICADOR | CONSTANTE | PARENIZQ <expresión> PARENDER

```
void Factor (void) {
    TOKEN t = ProximoToken();
    switch (t) {
        case IDENTIFICADOR : Match(IDENTIFICADOR); break;
        case CONTANTE : Match (CONSTANTE); break;
        case PARENIZQ : Match (PARENIZQ); Expresion(); Match (PARENDER); break;
        default : ErrorSintactico( t ); break;
    }
}
```

Producciones recursivas

Se utiliza un ciclo infinito del cual se sale cuando se averigua que el token que recibirá el parser no es el esperado.

Ejemplo

```
<listaSentencias> -> <sentencia>
<sentencia> -> t1 ... |
                t2 ...
```

} {<sentencia>} (escrito en Lenguaje Micro)

```
<L> -> <S> | <S> <L>
<S> -> t1... | t2 ...
```

} (escrito en una BNF)

```
void ListaSentencias (void) {
    Sentencia();
    while (1) {
        TOKEN t = ProximoToken();
        if ( t == t1 || t == t2)
            Sentencia();
        else
            return;
    }
}
```

Ejemplo

```
<listaSent> -> <sent> { t1 <sent>}
void ListaSent (void) {
    Sent();
    while (1) {
        TOKEN t = ProximoToken();
        if ( t == t1 ) {
            Match( t );
            Sent();
        }
        else
            return;
    }
}
```

Transformación de una GIC en una GIC LL(1)

La gramática llamada LL(1) puede ser utilizada por un Parser LL con un solo símbolo de preanálisis. Esto significa que si un no terminal tiene varias producciones, el Parser puede decidir cual lado derecho debe aplicar con solo conocer el próximo token del flujo de entrada. Por ejemplo:

$R \rightarrow aR \mid b$, permite construir un LL(1) ya que suponiendo que el flujo de entrada es "aab" con solo conocer el primer token (que coincide con el símbolo de preanálisis) entonces determina inequívocamente que producción aplica.

Si es recursiva a Izquierda

$X \rightarrow Xa \mid b$ se transforma en $X \rightarrow bZ$
 $Z \rightarrow aZ \mid \epsilon$

Si hay prefijo común

$A \rightarrow aB \mid aC \mid \dots$ se transforma en $A \rightarrow aZ$
 $Z \rightarrow B \mid C \mid \dots$

Ejemplo:

$S \rightarrow SbD \mid ab$	se transforma en	$S \rightarrow abZ$
$D \rightarrow a \mid ac$		$Z \rightarrow bDZ \mid \epsilon$
$E \rightarrow S$		$D \rightarrow aR$
		$R \rightarrow c \mid \epsilon$
		$E \rightarrow S$

Conjunto Primero

En la derivación de una producción que del lado derecho tiene varias posibilidades para expandirse se necesita conocer los símbolos de preanálisis para decidir cuál se elige. Se denomina conjunto Primero al conjunto de dichos símbolos.

Si el símbolo de preanálisis está en PRIMERO(α), se usa la producción con lado izquierdo α .

Veamos con ejemplos los distintos casos que se podrían presentar:

1) $S \rightarrow aS \mid bS$ **Primero (S) = {a,b}**

2) $S \rightarrow aS \mid B$ **Primero (S) = {a,c,d}**
 $B \rightarrow c \mid dT$

3) $S \rightarrow aS \mid BT$ **Primero (S) = {a, c, h}**
 $B \rightarrow c \mid \epsilon$
 $T \rightarrow h \mid R$
 $R \rightarrow k$

4) $S \rightarrow BC$ **Primero (S) = {b, c, ϵ }**
 $B \rightarrow b \mid \epsilon$
 $C \rightarrow c \mid \epsilon$

5)

$S \rightarrow TR$	$\text{Primero (S)} = \text{Primero (TR)}$
$R \rightarrow gTR \mid \epsilon$	$\text{Primero (S)} = \text{Primero (T)}$
$T \rightarrow ZQ$	$\text{Primero (S)} = \text{Primero (ZQ)}$
$Q \rightarrow bZQ \mid \epsilon$	$\text{Primero (S)} = \text{Primero (Z)}$
$Z \rightarrow cSc \mid a$	$\text{Primero (S)} = \text{Primero (cSc)} \cup \text{Primero (a)}$
	$\text{Primero (S)} = \text{Primero (c)} \cup \text{Primero (a)}$
	$\text{Primero (S)} = \{c,a\}$

ANSI C

Tipos de Datos

char	-128 a 127
unsigned char	0 a 255
Short	-32768 a 32767
unsigned short	0 a 65535
unsigned long	0 a 4294967295
Flota	$3,9 \times 10^{-38}$ a $3,1 \times 10^{38}$
double	$1,7 \times 10^{-308}$ a $1,7 \times 10^{308}$

Sentencias

<pre>if (....) { ... }else{ ... }</pre>	<pre>while (...){ ... }</pre>
<pre>int i; for (i = 0; i < 10; i++){ ... }</pre>	<pre>Switch (valor){ Case 'A': ...; Case 'B': ...; default: ...; }</pre>

EJEMPLO DE UN PROGRAMA

```
# include <stdio.h>

void producto (int a, int b, int *retorno);

int main(void){
    int resultado;
    producto(2,4,&resultado);
    printf("Multiplicación: %d\n"; resultado);
    return 0;
}

void producto (int a, int b, int *retorno){
    *retorno = a*b;
    return;
}
```

COMANDOS UTILIZADOS DENTRO DEL PRINTF

%c	Char
%d	Int
%f	Flota
%g	Double
%s	String
%o	Octal

\a	Alerta
\b	Retroceso
\t	Tabulación
\n	Nueva línea
\v	Tabulación vertical
\'	Comillas
\?	Signo de interrogación → ?
\0	Fin de línea o centinela

Nota: Los strings están formados por un vector.

Por ejemplo..

```
Char *palabra;
palabra = "utn";
```

Entonces palabra[0] devolvería "u".

TABLA DE PRIORIDADES

Prioridad	Operadores
Más Alta	() [] ->
	! ~ ++ -- (tipo) * & sizeof
	* / %
	+ -
	< < > >
	< <= > >=
	= = ! =
	^
	&&
	?
	.
	= += -= *= /=
Más Baja	,

DEFINICIONES

Definición

- Inicializamos una variable → `int variable1, variable2;`
- Indicamos el prototipo de una función → `int MiFuncion(int a, int b);`

Declaración

- Cuando se desarrolla la función → `int MiFuncion(int a, int b){if(a>b)...}`
- Cuando se declara una estructura de dato → `typedef struct{char a, int b} tEstruc;`

Sentencia

- Son aquellas que terminan con punto y coma (;) → `a+b+c;`

Expresión

- Son aquellas que forman una sentencia → `a+1`

Categorías Léxicas

Palabras Reservadas:

char | do | double | else | float | for | if | int | long | return | sizeof | struct | typedef | void |
while | unsigned | signed | const

Identificadores:

main | VARIABLES | ungetc | bool | cast | NOMBRE DE LAS FUNCIONES

Caracteres de Puntuación:

(|) | , | { | } | [|] | ;

Operadores:

* | / | - | + | ++ | & | % | < | > | = | ! | != | && | <= | >= | += | = | ||

Constantes:

Números o letras entre comillas simples o no | ejemplo: 'A' | ejemplo: 0x23

Ejemplos de algunos conjunto de elementos que representan una constante entera en ANSI C.

Decimal = 436 , 85L , 99UL , 163LU

Octal = 0 , 00

Hexadecimal = 0x42 , 0Xab

ANSI C: matrices

¿Cómo se declaran?

tipo nombre [cantidad de filas] [cantidad de columnas];

Ejemplo: int mat [2] [3];

El primer elemento es mat [0][0], el último es mat [1][2].

¿Cómo se inicializan?

int mat [2] [3] = { {1,2,3} , {4,5,6} };

La matriz como parámetro de una función

int suma_mat (int mat [] [3]) ; ó int suma_mat (int mat [2] [3]) ;

Si una función tiene como parámetro una matriz se debe especificar la cantidad de columnas que posee, pues de lo contrario la función no podría recorrerla.

Otra forma de escribir el prototipo de la función suma_mat es usando typedef:

```
#define FILAS 2
```

```
#define COLUMNAS 3
```

```
typedef int tipoMatriz [FILAS] [COLUMNAS] ;
```

```
int suma_mat (tipoMatriz mat );
```

ANSI C: Archivos binarios.

Un archivo es una colección de bytes en memoria externa que se identifica por un nombre.

La estructura de tipo FILE es el punto de contacto entre el programa y el sistema operativo en lo que a archivos se refiere. FILE contiene cierta información sobre un archivo, por ejemplo un puntero asociado al buffer, un indicador de fin de archivo, etc. El diseño de la estructura FILE está en <stdio.h>.

Cada archivo que se abre tendrá su propia estructura de tipo FILE.

Ejemplo: consideraremos archivos con legajo y nota de ciertos alumnos.

Declaración de archivos binarios

```
typedef struct {  
    int legajo;  
    char nota;  
} TREG;
```

```
FILE *arch1, arch2;
```

Apertura y asignación

```
arch1 = fopen ( "entrada.bin", "rb");  
arch2 = fopen ("salida.bin", "wb");
```

Prototipo de la function fopen (): FILE* fopen (const char* nombre_externo, const char* modo);

Abre el archivo nombre_externo y retorna un puntero a la estructura FILE asociada a nuestro archivo; este puntero se almacena en la variable que representa el nombre_lógico del archivo. Si el archivo nombre_externo no se puede abrir la función fopen() retorna el valor NULL (equivalente a cero).

Cierre de un archivo

`fclose (arch1);`

Prototipo de la function `fclose ()` : `int fclose (FILE *arch);`

El cierre de un archivo tiene 2 efectos fundamentales:

- a) Los bytes que han quedado en el "buffer" son grabados en el disco (si el archivo se abrió para grabar)
- b) Se liberan las áreas de comunicación usadas por el archivo (es decir: su estructura FILE y su buffer). La función `fclose ()` retorna un 0 si tuvo éxito o un EOF si hubo algún error.

Lectura y grabación

`fread (®1, sizeof (treg), 1, arch1);`

`fwrite (®2, sizeof (treg), 1, arch2);`

Los prototipos de las funciones son:

`unsigned fread (void *pbloque, unsigned tamaño, unsigned n, FILE *arch);`

*Lee n datos, cada uno de longitud tamaño bytes, desde arch, y los copia en un bloque apuntado por pbloque. La cantidad de bytes leídos es (n*tamaño). La función fread() retorna n que es la cantidad de ítems (no bytes) leídos, o un valor menor que n si detectó un error o el fin del archivo.*

`unsigned fwrite (void *pbloque, unsigned tamaño, unsigned n, FILE *arch);`

Agrega n datos, cada uno de tamaño bytes, en arch. Los datos grabados comienzan en pbloque. La función fwrite () retorna la cantidad de ítems (no bytes) grabados o un valor menor si detectó algún error.

Fin de Archivo (end-of-file)

En los ciclos de lectura de un archivo se consulta por la marca de fin de archivo. En Pascal se utiliza la función EOF (ARCHIVO). Esta función devuelve TRUE cuando el puntero del archivo esta al final de un archivo y FALSE en caso contrario.

El prototipo es:

```
int feof (FILE *arch);
```

Esta macro retorna un valor distinto de cero si se ha detectado el "eof" al leer previamente desde arch, caso contrario retorna 0. Para activar la función es necesario leer, por lo tanto el ciclo de lectura en ANSI C es el siguiente

```
fread (.....);  
while (! feof (arch1) ) {  
    proceso del reg1  
    fread(.....);  
}
```


ANSI C: Archivos de caracteres.

Declaración

FILE *arch1, arch2;

Apertura y asignación

arch1= fopen ("entrada.txt", "rt"); /* la t es optativa */
arch2= fopen ("salida.txt", "w");

Cierre

fclose (arch1);

Lectura y grabación de un solo carácter por vez

int getc (FILE *arch);

Macro que retorna el próximo carácter del archivo identificado por arch. Si detecta "fin de archivo" o un error, retorna EOF.

int fgetc (FILE *arch);

Función equivalente a la macro getc().

int putc (int ch, FILE *arch);

Macro que graba el carácter ch en el archivo identificado por arch. Retorna el carácter ch si tuvo éxito o EOF si hubo error.

int fputc (int ch, FILE *arch);

Función equivalente a la macro putc().

int ungetc (int c, FILE *arch);

Devuelve el carácter o byte c para la próxima lectura. Retorna c si tuvo éxito ó EOF si hubo error..

Lectura y grabación de strings

char * fgets (char *s, int n, FILE *arch);

Función que lee hasta n-1 caracteres o hasta encontrar '\n', desde el archivo identificado por arch en el string s.

Retiene el '\n' y luego agrega el '\0' a s. (La función gets() transforma el '\n' en '\0').

Si tuvo éxito, fgets() retorna el String apuntado por s; retorna NULL si detectó el "fin de archivo" o hubo error.

int fputs (char *s, FILE *arch);

Función que graba el string s en el archivo identificado por arch.

No agrega el carácter '\n' /(como lo hace puts() */ y tampoco copia el '\0' con que termina el string. El '\n' se debe agregar con otro fputs () si se quiere separar una línea de otra.*

Retorna el último carácter grabado si tuvo éxito o EOF si hubo algún error.

Lectura y grabación con formato

```
int fscanf ( FILE *arch , char *formato, . . . );
```

Función que realiza la lectura con el formato apropiado, desde el archivo identificado por arch, en los objetos cuyas direcciones son dadas.

Retorna la cantidad de datos almacenados si tuvo éxito, EOF si es “fin de archivo”, ó cero si no hay datos.

```
int fprintf ( FILE *arch , char *formato , . . . );
```

Función que envía valores formateados al archivo identificado por arch.

Retorna la cantidad total de bytes grabados, ó EOF si detectó algún error.

Modo “TEXT” y modo “BINARIO”

Del modo de apertura depende como se almacena la señal de nueva línea, como se indica el fin de archivo y con que formato se almacenan los números en el archivo.

En modo text los números son almacenados como sucesión de caracteres, mientras que en el modo binario son almacenados con el mismo formato que tienen en la memoria.

Las siguientes funciones de lectura-grabación pueden utilizarse en archivos binarios y en archivo de texto:

`fgetc()` /*lee un carácter de un archivo ASCII o un byte de un archivo binario*/

`fputc()` /*graba un carácter de un archivo ASCII o un byte de un archivo binario*/

ANSI C: otras funciones de archivo

Función fseek()

int fseek (FILE *arch, long **desplazamiento**, int **desde**);

Permite tratar a los archivos como si fueran vectores, moviéndose directamente a un byte determinado del archivo.

Coloca el “file pointer” (puntero del archivo) asociado con el archivo identificado por **arch** en una posición que está **desplazamiento** bytes desde la ubicación en el archivo dada originalmente por **desde**.

El desplazamiento puede ser positivo (movimiento hacia el final del archivo) o negativo (movimiento hacia el comienzo del archivo) .

El valor de **desde** es 0, 1 o 2 y está definido de la siguiente manera:

#define SEEK_SET 0 (desde el comienzo del archivo)

#define SEEK_CUR 1 (desde la posición actual)

#define SEEK_END 2 (desde el fin del archivo)

La función fseek() retorna 0 si tuvo éxito, o un valor distinto de 0 si detectó algún error (por ejemplo: si se intenta avanzar fuera de los límites del archivo, si el archivo no está abierto, etc).

Función ftell()

long ftell (FILE *arch);

Retorna la posición actual del “file pointer” medido en cantidad de bytes desde el comienzo del archivo. Si hubo algún error retorna (−1).

EJERCICIOS RAROS O JODIDOS

Sea la función ANSI C:

```
unsigned int XX (const char *s, int c) { unsigned int i, n;  
for (i=n=0 ; s[i] != '\0' ; i++) if (s[i]==c) n++ ; return n; }
```

a) Cantidad de ungetc que realiza el scanner?

Respuesta: 29 (los remarcados en rojo)

```
unsigned_int_XX_(const_char_*s, int_c) { unsigned_int_i, n;  
for_(i=n=0_ ; s[i] != '\0' ; i++) if_(s[i]==c) n++ ; return_n; }
```

b) Cantidad de lexemas

```
unsigned int XX (const char *s, int c) {  
    unsigned int i, n;  
    for (i=n=0 ; s[i] != '\0' ; i++)  
        if (s[i]==c)  
            n++;  
  
    return n;  
}
```

IDENTIFICADORES	ID	15
PALABRAS RESERVADAS	PR	10
CONSTANTES	CONST	2
CARACTERES DE PUNTUACION	CP	19
OPERADORES	OP	7
LEXEMAS EN TOTAL:		53

Ejemplos de algunos conjunto de elementos que representan una constante entera en ANSI C.

Decimal = 436 , 85L , 99UL , 163LU

Octal = 0 , 00

Hexadecimal = 0x42 , 0Xab

VERDADEROS O FALSOS

VERDADERAS

- EL AFD y el AFN son modelos equivalentes.
- La maquina de Turing es representada como una memoria por una cinta infinita
- La maquina de Turing no siempre reemplaza el carácter leído por otro distinto.
- El complemento de un lenguaje puede ser el lenguaje con la palabra épsilon.
- La tabla de símbolos comienza a crearse durante el Análisis Léxico.
- Toda gramática formal es una 4-upla (V_n, V_t, P, S) donde S es el axioma.
- El analizador léxico no ignora el carácter espacio.
- El Parser siempre comienza invocando a un PAS
- El LEX no puede detectar y procesar palabras de cualquier lenguaje formal.
- La sintaxis de un LR puede representarse mediante una BNF
- Para construir un Parser basado en el análisis sintáctico descendente recursivo no se requiere un programa especializado tipo yacc.
- En las metaERs, NO se puede usar paréntesis o corchetes para agrupar una ER.
- El Parser tiene la capacidad de reconocer todas las estructuras de un Lenguaje de Programación
- El Algoritmo de Thompson puede utilizarse para construir autómatas que reconozcan palabras de cualquier lenguaje finito.

- ANSI C:
- Las constantes octales comienzan con el dígito cero.
- For (4;4;4) 4; → Es sintacticamente correcta.
- La sentencia while(3)a++; y while(a==a)a++; → son semánticamente equivalentes.
- Printf("%d\n", 'A'); y printf ("%d", 'A'); NO tienen la misma semántica.
- Fi (x>0) a=x; es un error sintáctico.
- Sea int a; entonces la expresión 2==a es semánticamente válida.