Tema 2 | Análisis de Léxico

Contenidos

2.1 Descripción funcional.
2.2 Conceptos de token, lexema y patrón.
2.3 Fundamentos: álgebra de lenguajes.
- 2.3.1 Alfabeto y lenguaje.
- 2.3.2 Operaciones con lenguajes.
- 2.3.3 Especificación de los tokens y patrones.
- 2.3.4 Propiedades y notación taquigráfica de las expresiones regulares.
- 2.3.5 Autómata finito no determinista (AFN) asociado a una expresión regular.
- 2.3.6 AFN reconocedor de patrones (sin y con anticipación).

2.4 Tratamiento de errores.
- 2.4.1 Estrategias para la recuperación ante el error.
- 2.4.2 Reparación de errores.

2.5 LEX (Generador de Analizadores de Léxico).
- 2.5.1 Introducción.
- 2.5.2 Especificación Lex.
- 2.5.3 Consideraciones finales.
- 2.5.4 Ejemplos Lex.

Capítulo 2 — **Análisis de Léxico**

Contenidos

2.1	**Descripción funcional**	**33**
2.2	**Conceptos básicos**	**34**
2.3	**Fundamentos. Álgebra de lenguajes**	**34**
2.3.1	Especificación de los tokens y patrones	35
2.3.2	Propiedades y notación taquigráfica de las expresiones regulares	36
2.3.3	Autómata finito no determinista (AFN) asociado a una expresión regular	37
2.3.4	Reconocimiento de patrones	41
2.4	**Pasos para el diseño del analizador de léxico**	**43**
2.5	**Tratamiento de errores**	**44**
2.5.1	Tratamiento de errores en el análisis léxico	46
2.5.2	Estrategias para la recuperación ante el error	46
2.6	**Concepto de tabla de símbolos**	**47**

Bibliografía básica

[Aho90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compiladores. Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana 1990.

[Broo93] J.G. Brookshear. *Teoría de la Computación. Lenguajes formales, autómatas y complejidad*. Addison-Wesley Iberoamericana 1993

Capítulo 3 — **LEX (Generador de Analizadores de Léxico)**

Contenidos

3.1	**Introducción**	**51**
3.2	**Especificación LEX**	**53**
3.2.1	Declaraciones o definiciones	54
3.2.2	Reglas o acciones	55
3.2.2.1	Acciones por defecto	55
3.2.2.2	Acciones especiales	56
3.2.2.3	Condiciones START (sensibilidad al contexto izquierdo)	57
3.2.2.4	Condiciones START exclusivas	57
3.2.3	Procedimientos auxiliares	58
3.2.4	Variables y funciones LEX	58
3.3	**Consideraciones finales**	**59**
3.4	**Ejemplos LEX**	**59**

Bibliografía básica

[Aho90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compiladores. Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana 1990.

[Lesk75] M.E. Lesk, E. Schmidth. *Lex. A Lexical Analyzer Generator en UNIX Programmer's Manual 2*. AT&T Bell Laboratories 1975.

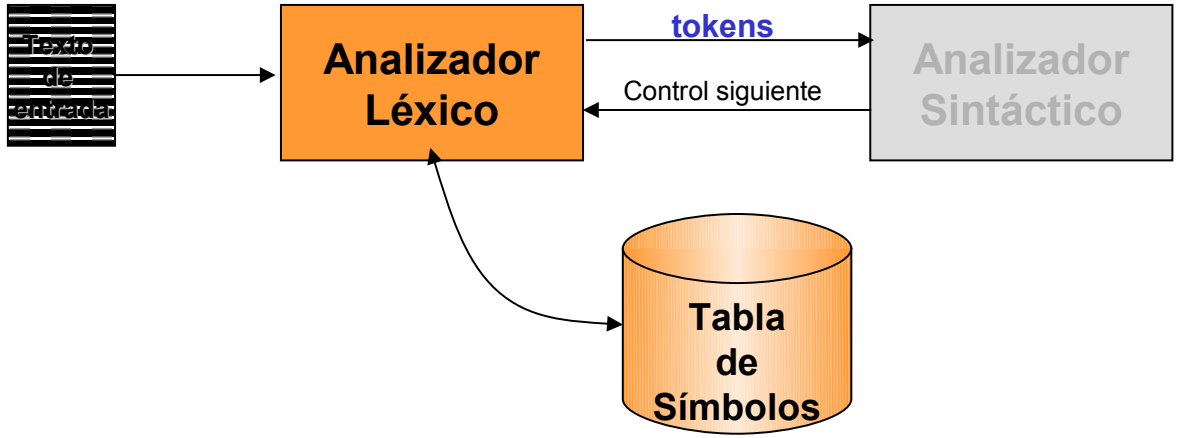
[Levi92] J.R. Levine, T. Manson, D. Brown. *Lex & Yacc*. O'Reilly & Associates, Inc. 1992.

[Benn90] J.P. Bennet. *Introduction to compiling techniques: A first course using ANSI C, LEX y YACC*. McGraw-Hill. 1990.

24/09/13

2.1 Descripción funcional

Lee, carácter a carácter, del documento de entrada (texto fuente) y genera una secuencia de patrones léxicos denominados **tokens** y, en su caso, junto con sus atributos.



Aportaciones del Analizador de Léxico

- Simplificar el diseño del analizador sintáctico, confiriéndole una mayor eficacia.
- Portabilidad del traductor.

2.2 Conceptos Básicos

- **TOKEN**: Conjunto de secuencias de caracteres con la misma misión sintácticamente.
- **LEXEMA**: Secuencia de caracteres que forman un token.
- **PATRÓN**: Regla o reglas que describen a los lexemas pertenecientes a un token.

Ejemplo 2.1: Dada la especificación de este lenguaje.

$$
\begin{array}{rcl}
P & \rightarrow & S \mid P\,S \\
S & \rightarrow & \textit{repeat exp } S; \\
 & \mid & \textit{while exp } S; \\
 & \mid & \textit{if exp } S; \\
 & \mid & \textit{id} := \textit{exp}; \\
\textit{exp} & \rightarrow & \textit{exp} * \textit{exp} \\
 & \mid & \textit{exp} / \textit{exp} \\
 & \mid & \textit{exp} + \textit{exp} \\
 & \mid & + \textit{exp} \\
 & \mid & * \textit{exp} \\
 & \mid & (\textit{exp}) \\
 & \mid & \textit{constante} \\
 & \mid & \textit{id} \\
\textit{constante} & \rightarrow & \textit{constante digito} \mid \textit{digito} \\
\textit{digito} & \rightarrow & [0-9] \\
\textit{id} & \rightarrow & \textit{id letra} \mid \textit{letra} \\
\textit{letra} & \rightarrow & [a-z]
\end{array}
$$

Inicialmente, tomamos los siguientes lexemas: ***repeat***, ***while***, ***if***. ¿qué tienen en común?... Forman parte de una sentencia ***S*** y van seguidos de "***exp S;***"

COINCIDEN EN TODO LO QUE LE PRECEDE Y SUCEDE EN UN TEXTO

Es decir, es posible agrupar estos tres lexemas en otro de mayor abstracción para constituir un **token**.

Consecuencia: Reducimos tres reglas gramaticales en una sola:

S → **CONDCICLO** *exp S* ;

| *id* = *exp* ;

Donde, **CONDCICLO** es un token definido por el siguiente patrón:

```
("repeat"|"while"|"if")
```

2.3 Fundamentos

Alfabeto: Conjunto de símbolos. $A = a, b, \ldots, z, A, B, \ldots, Z$

Lenguaje: Conjunto de secuencias de símbolos de un alfabeto. Para un alfabeto **A** le corresponde un *lenguaje elemental* que es **A**.

$$L = a, b, \ldots, z, A, B, \ldots, Z$$

Operaciones con Lenguajes

- **Unión**: $L_1 \bigcup L_2 = \{x/x \in L_1 \ o \ x \in L_2\}$
- **Intersección**: $L_1 \bigcap L_2 = \{x/x \in L_1 \ y \ x \in L_2\}$
- **Concatenación**: $L_1 L_2 = \{xz/x \in L_1 \ y \ z \in L_2\}, L^0 = \{\lambda\}$
- **Clausura**: $L^* = \bigcup_{i=0}^{\infty} L^i$
- **Semigrupo**: $L^+ = \bigcup_{i=1}^{\infty} L^i$

Formas de Especificar un Lenguaje

$L = \{a, aa, aba, abbaa, \ldots\}$

Especificación de los tokens

Token	Identificador	Atributos	Patrón/Expresión regular
CONDCICLO	255	0: `repeat` 1: `while` 2: `if`	**("repeat"|"while"|"if")**

Expresiones Regulares

Dado un alfabeto Σ. Las reglas que definen expresiones regulares de Σ son:

- λ es una expresión regular que representa la secuencia vacía: $\{\lambda\}$
- Si, $a \in \sum$ entonces $\{a\}$ es una expresión regular.
- Sea y dos expresiones regulares que denotan al lenguaje $L(r)$ y $L(s)$. Se cumple:
  - a) $(r)|(s)$ es una expresión regular notada por $L(r) \cup L(s)$.
  - b) $(r)(s)$ es una expresión regular notada por $L(r)L(s)$.
  - c) $(r)^*$ es una expresión regular notada por $L(r)^*$.
  - d) (r) es una expresión regular notada por $L(r)$.

Propiedades Algebraicas de las Expresiones Regulares

Con las definiciones anteriores se cumples las siguientes propiedades:

- Conmutativa: $r|s = s|r$.
- Asociativa: $r|(s|t) = (r|s)|t$.
- Asociativa sobre la concatenación: $(rs)t = r(st)$.
- Distributiva de la concatenación y unión: $r(s|t) = rs|rt$ y $(r|t)s = rs|ts$.
- Elemento neutro: $\lambda r = r$ y $r\lambda = r$, siendo λ el elemento neutro y también se puede expresar $r^* = (r|\lambda)^*$ y $r^{**} = r^*$.

También se define la siguiente notación taquigráfica para la representación de expresiones regulares:

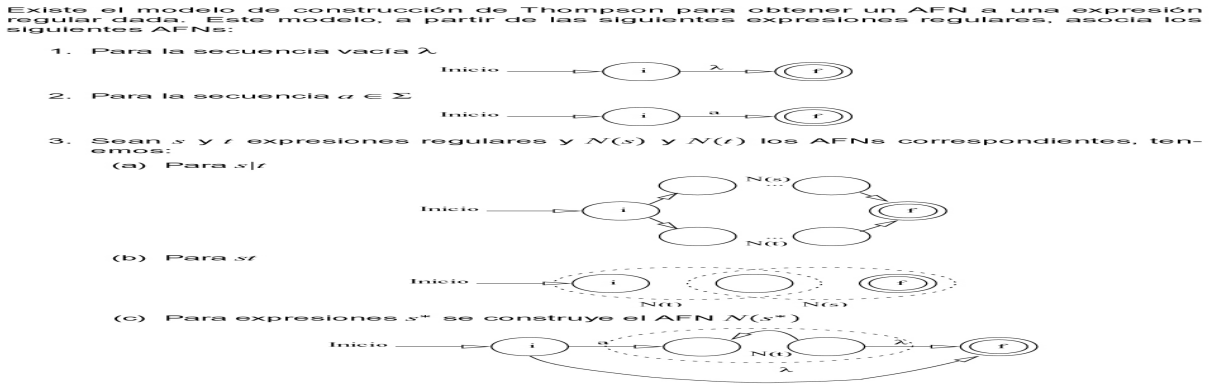
1. Uno o más veces: operador $+$, $(r^* = r^+|\lambda)$.
2. Cero o una vez: operador $?$.
3. Cero o más veces: operador $*$.
4. Una forma cómoda de definir *clases de caracteres* es de la siguiente forma:

$$a|b|c|\cdots|z = [a-z]$$

AFN Asociado a una Expresión Regular

Teorema: A toda expresión regular le corresponde un AFN (ver demostración en [Broo93]).

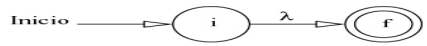
Existe el modelo de construcción de Thompson para obtener el AFN asociado a una expresión regular dada. Este modelo, a partir de las siguientes expresiones regulares, asocia los siguientes AFNs:



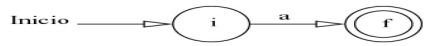
AFN Asociado a una Expresión Regular (2)

Existe el modelo de construcción de Thompson para obtener un AFN a una expresión regular dada. Este modelo, a partir de las siguientes expresiones regulares, asocia los siguientes AFNs:

1. Para la secuencia vacía λ

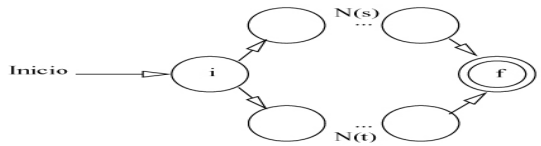


2. Para la secuencia $a \in \Sigma$

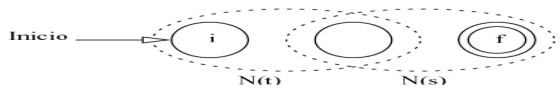


3. Sean s y t expresiones regulares y $\mathcal{N}(s)$ y $\mathcal{N}(t)$ los AFNs correspondientes, tenemos:

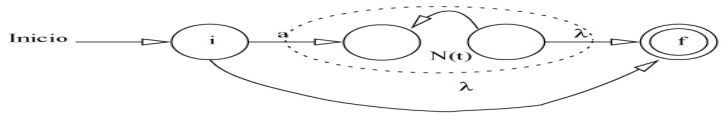
   (a) Para $s|t$



   (b) Para st



   (c) Para expresiones s^* se construye el AFN $\mathcal{N}(s^*)$



Existe el modelo de construcción de Thompson para obtener un AFN a una expresión regular dada. Este modelo, a partir de las siguientes expresiones regulares, asocia los siguientes AFNs:

1. Para la secuencia vacía λ
2. Para la secuencia $a \in \Sigma$
3. Sean s y t expresiones regulares y $\mathcal{N}(s)$ y $\mathcal{N}(t)$ los AFNs correspondientes, tenemos:
   (a) Para $s|t$
   (b) Para st
   (c) Para expresiones s^* se construye el AFN $\mathcal{N}(s^*)$

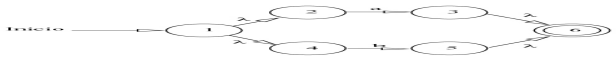
Ejemplo de Construcción de Thompson (1)

Supongamos la siguiente expresión regular

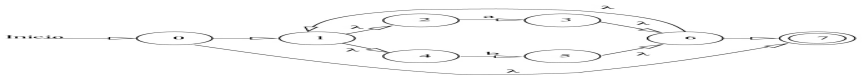
$r = (a|b)^*abb$

La forma de construir el AFN es por pasos. En primer lugar el de los símbolos a y b por separado:

Posteriormente $a|b$



Seguimos con la expresión $(a|b)^*$



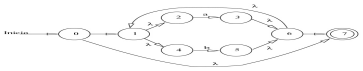
Supongamos la siguiente expresión regular

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

La forma de construir el AFN es por pasos. En primer lugar el de los símbolos a y b por separado:

Posteriormente $a|b$

Seguimos con la expresión $(a|b)^*$



$a|b$

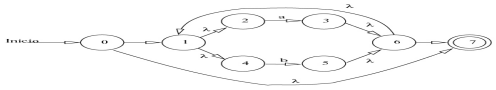
Supongamos la siguiente expresión regular

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

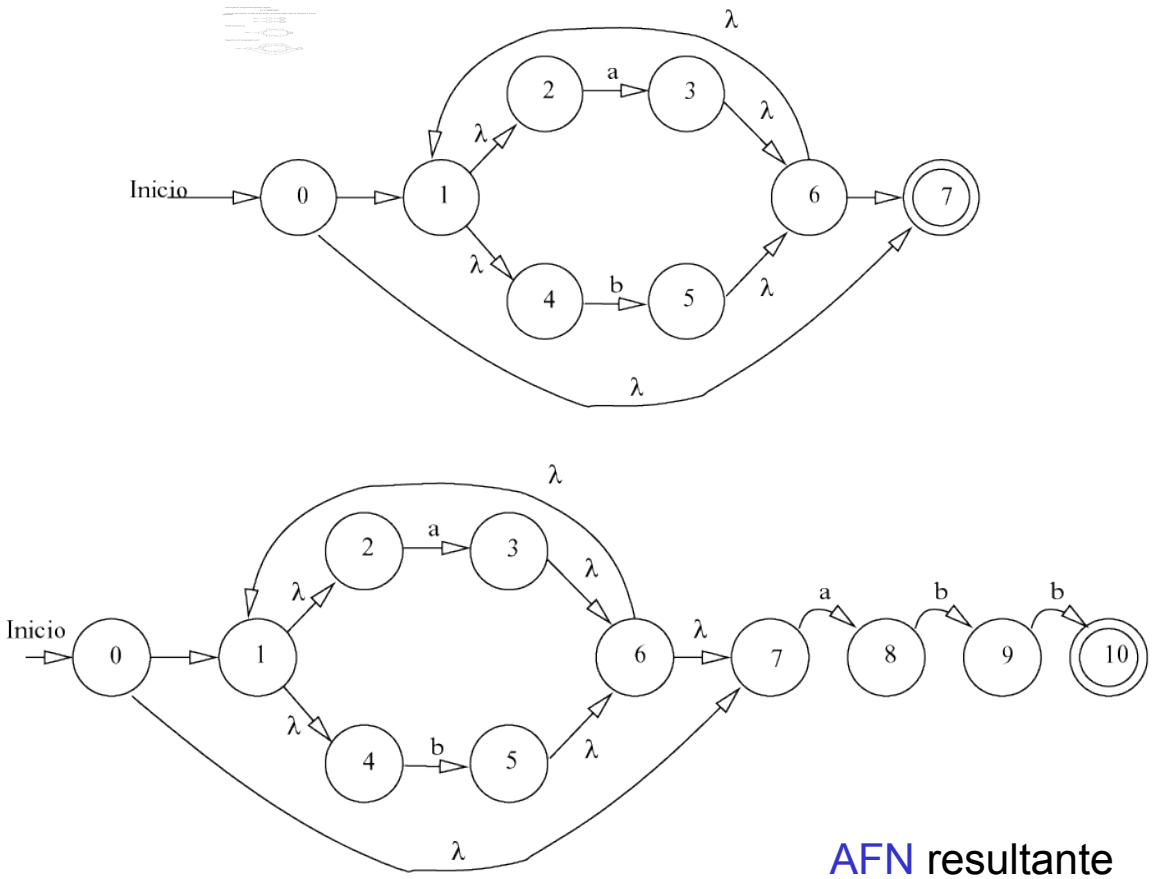
La forma de construir el AFN es por pasos. En primer lugar el de los símbolos a y b por separado:

Posteriormente $a|b$

Seguimos con la expresión $(a|b)^*$



Ejemplo de Construcción de Thompson (2)



AFN resultante

Paso de un AFN a un AFD equivalente

Existe un teorema en teoría de autómatas por el que a todo AFN se le puede asociar un **AFD** (Autómata Finito Determinista) equivalente.

Un **AFD** es un caso especial de AFN en el que se cumplen las siguientes condiciones:

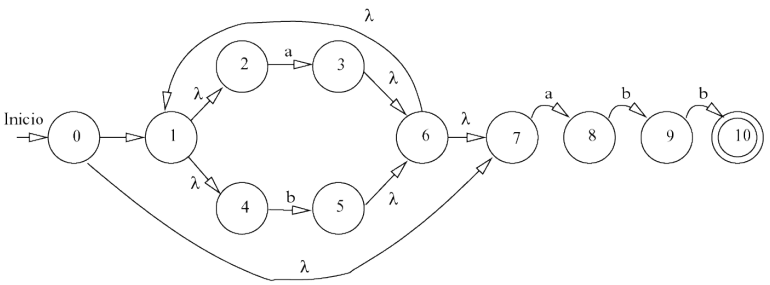
1. Ningún estado tiene una **λ-transición**.
2. Para cada estado *s* y cada símbolo de entrada *a*, hay a lo sumo una arista etiquetada *a* que sale de *s*.

Para la conversión emplearemos tres funciones:

- **λ-transición(*s*)**: Conjunto de estados del AFN alcanzables desde el estado *s* únicamente con λ-transiciones.
- **λ-clausura(T)**: Conjunto de estados del AFN alcanzables desde algún estado *s* en *T* únicamente con **λ-transiciones** (siendo *T* un conjunto de estados).
- **Move(*T*,*a*)**: Conjunto de estados del AFN hacia los cuales hay una transición con el símbolo de entrada *a* desde algún estado *s* en *T*.

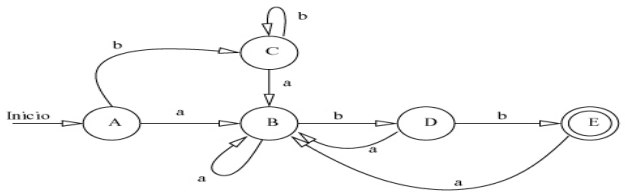
Ejemplo 2.2: Construcción del AFD equivalente a partir del AFN anterior. El alfabeto de entrada es {a,b}.

$\lambda\text{-clausura}(0) = \{0,1,2,4,7\} = \boldsymbol{A}$
$\lambda\text{-clausura}(Move(A,a)) = \lambda\text{-clausura}(\{3,8\}) = \{1,2,3,4,6,7,8\} = \boldsymbol{B}$
$Trans(A,a) = B$
$\lambda\text{-clausura}(Move(A,b)) = \lambda\text{-clausura}(\{5\}) = \{1,2,4,5,6,7\} = \boldsymbol{C}$
$Trans(A,b) = B$
$\lambda\text{-clausura}(Move(B,a)) = \lambda\text{-clausura}(\{3,8\}) = \boldsymbol{B}$
$Trans(B,a) = B$
$\lambda\text{-clausura}(Move(B,b)) = \lambda\text{-clausura}(\{5,9\}) = \{1,2,4,5,6,7,9\}) = \boldsymbol{D}$
$Trans(B,b) = D$
$\lambda\text{-clausura}(Move(C,a)) = \lambda\text{-clausura}(\{3,8\}) = \boldsymbol{B}$
$Trans(C,a) = B$
$\lambda\text{-clausura}(Move(C,b)) = \lambda\text{-clausura}(\{5\}) = \boldsymbol{C}$
$Trans(C,b) = C$
$\lambda\text{-clausura}(Move(D,a)) = \lambda\text{-clausura}(\{3,8\}) = \boldsymbol{B}$
$Trans(D,a) = B$
$\lambda\text{-clausura}(Move(D,b)) = \lambda\text{-clausura}(\{5,10\}) = \{1,2,4,5,6,7,10\} = \boldsymbol{E}$
$Trans(D,b) = E$
$\lambda\text{-clausura}(Move(E,a)) = \lambda\text{-clausura}(\{3,8\}) = \boldsymbol{B}$
$Trans(E,a) = B$
$\lambda\text{-clausura}(Move(E,b)) = \lambda\text{-clausura}(\{5\}) = \boldsymbol{C}$
$Trans(E,b) = \boldsymbol{C}$



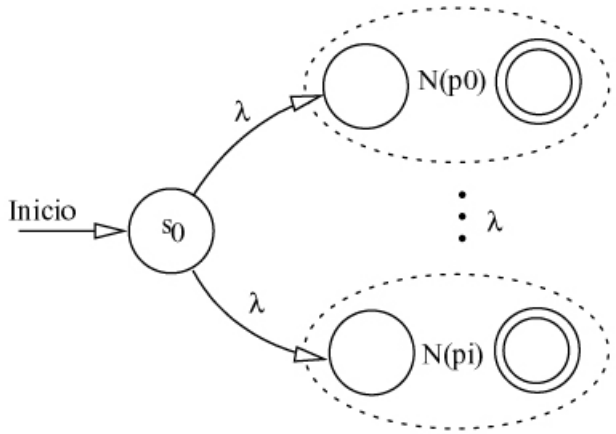
Estado	Símbolo *a*	Símbolo *b*
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Tabla 2.1: Tabla de transiciones *Trans* para el AFD.



Reconocimiento de Patrones con un AFN

Sea $P_1 \mid P_2 \mid \cdots \mid P_n$ los patrones que deberá reconocer, se le asociará a cada patrón un AFN $P_i \rightarrow N(P_i)$ se añade una **λ-transición** y un estado inicial:



Ejemplo 2.3: Dada la siguiente gramática, determinar el conjunto de tokens con el máximo nivel de abstracción para la construcción de un traductor.

$$
\begin{array}{rcl}
P & \rightarrow & S \mid P\ S \\
S & \rightarrow & \textit{repeat exp } S; \\
 & \mid & \textit{while exp } S; \\
 & \mid & \textit{if exp } S; \\
 & \mid & \textit{id } = \textit{ exp}; \\
\textit{exp} & \rightarrow & \textit{exp} * \textit{exp} \\
 & \mid & \textit{exp} / \textit{exp} \\
 & \mid & \textit{exp} + \textit{exp} \\
 & \mid & + \textit{exp} \\
 & \mid & * \textit{exp} \\
 & \mid & (\textit{exp}) \\
 & \mid & \textit{constante} \\
 & \mid & \textit{id} \\
\textit{constante} & \rightarrow & \textit{constante digito} \mid \textit{digito} \\
\textit{digito} & \rightarrow & [0-9] \\
\textit{id} & \rightarrow & \textit{id letra} \mid \textit{letra} \\
\textit{letra} & \rightarrow & [a-z]
\end{array}
$$

1. **Todas aquellas reglas cuya especificación se corresponda con una expresión regular podría ser, salvo abstracciones posteriores, Lexema.**
2. **Encontrar todas las palabras/símbolos/signos de la gramática descritos por medio de su expresión regular.**
3. **Agruparlos atendiendo a la misión sintáctica, es decir, que desempeñen el mismo papel a nivel sintáctico.**
4. **Repetir el paso anterior con el resto hasta alcanzar la máxima abstracción, es decir, no es posible realizar más agrupaciones**.

Token	Identificador	Atributos	Patrón/Expresión regular

Aplicando el punto 1, tenemos los siguientes elementos que pueden constituir tokens: *constante*, *digito*, *id* y *letra*.

$$
\begin{array}{rcl}
P & \rightarrow & S \mid P\ S \\
S & \rightarrow & \textit{repeat exp } S; \\
 & \mid & \textit{while exp } S; \\
 & \mid & \textit{if exp } S; \\
 & \mid & \textit{id} = \textit{exp}; \\
\textit{exp} & \rightarrow & \textit{exp} * \textit{exp} \\
 & \mid & \textit{exp} / \textit{exp} \\
 & \mid & \textit{exp} + \textit{exp} \\
 & \mid & + \textit{exp} \\
 & \mid & * \textit{exp} \\
 & \mid & (\textit{exp}) \\
 & \mid & \textit{constante} \\
 & \mid & \textit{id} \\
\textit{constante} & \rightarrow & \textit{constante digito} \mid \textit{digito} \\
\textit{digito} & \rightarrow & [0-9] \\
\textit{id} & \rightarrow & \textit{id letra} \mid \textit{letra} \\
\textit{letra} & \rightarrow & [a-z]
\end{array}
$$

Dado que *digito* está incluido dentro de la especificación de *constante*, hace que *dígito* pueda eliminarse. En principio, *constante* es un **lexema**, que constituye un **token**.

De igual modo, sucede con *id* y *letra* respectivamente. Un siguiente **token** sería *id*.

Token	Identificador	Atributos	Patrón/Expresión regular
CONSTANTE			`([0-9])+`
ID			`([a-z])+`

Aplicando los puntos 2 y 3, podemos extraer las siguientes palabras, signos y operadores y agruparlos atendiendo a la misión sintáctica:

$$
\begin{array}{rcl}
P & \rightarrow & S \mid P\ S \\
S & \rightarrow & repeat\ exp\ S; \\
 & | & while\ exp\ S; \\
 & | & if\ exp\ S; \\
 & | & id\ =\ exp; \\
exp & \rightarrow & exp * exp \\
 & | & exp / exp \\
 & | & exp + exp \\
 & | & + exp \\
 & | & * exp \\
 & | & (exp) \\
 & | & constante \\
 & | & id \\
constante & \rightarrow & constante\ digito \mid digito \\
digito & \rightarrow & [0-9] \\
id & \rightarrow & id\ letra \mid letra \\
letra & \rightarrow & [a-z]
\end{array}
$$

De abajo hacia arriba, podemos observar los siguientes:

{ (,) , * , + , / , = , ; , if , while , repeat }

Según se mostraba en el Ejemplo 2.1, if, while y repeat constituyen un **token** dado que cumplen la misma misión sintáctica.

En cuanto a los operadores, hay dos tipos: *unarios/binarios* y *binarios* exclusivos. En cuanto a los primeros tenemos el * y el + que cumplen la misma misión sintáctica. Se agrupan.

Token	Identificador	Atributos	Patrón/Expresión regular
CONDCICLO	257	0: repeat 1: while 2: if	("repeat"|"while"|"if")
PYC	258		";"
ASIGN	259		"="
MULSUM	260	0:*, 1:+	("*"|"+")
DIV	261		"/"
PARIZQ	262		"("
PARDCH	263		")"
CONSTANTE	264		([0-9])+
ID	265		([a-z])+

La gramática resultante resulta notablemente reducida con respecto a la original, tal y como se muestra a continuación:

$$
\begin{array}{rcl}
P & \rightarrow & S \mid P\ S \\
S & \rightarrow & \textit{repeat exp } S; \\
 & \mid & \textit{while exp } S; \\
 & \mid & \textit{if exp } S; \\
 & \mid & \textit{id } = \textit{ exp}; \\
\textit{exp} & \rightarrow & \textit{exp} * \textit{exp} \\
 & \mid & \textit{exp} / \textit{exp} \\
 & \mid & \textit{exp} + \textit{exp} \\
 & \mid & + \textit{exp} \\
 & \mid & * \textit{exp} \\
 & \mid & (\textit{exp}) \\
 & \mid & \textit{constante} \\
 & \mid & \textit{id} \\
\textit{constante} & \rightarrow & \textit{constante digito} \mid \textit{digito} \\
\textit{digito} & \rightarrow & [0-9] \\
\textit{id} & \rightarrow & \textit{id letra} \mid \textit{letra} \\
\textit{letra} & \rightarrow & [a-z]
\end{array}
$$

$$
\begin{array}{rcl}
P & \rightarrow & S \mid P\ S \\
S & \rightarrow & \textbf{CONDCICLO } \textit{exp } S \textbf{ PYC} \\
 & \mid & \textbf{ID ASIGN } \textit{exp } \textbf{PYC} \\
\textit{exp} & \rightarrow & \textit{exp } \textbf{MULSUM } \textit{exp} \\
 & \mid & \textit{exp } \textbf{DIV } \textit{exp} \\
 & \mid & \textbf{MULSUM } \textit{exp} \\
 & \mid & \textbf{PARIZQ } \textit{exp } \textbf{PARDCH} \\
 & \mid & \textbf{CONSTANTE} \\
 & \mid & \textbf{ID}
\end{array}
$$

Conceptos resultantes

GRAMÁTICA ABSTRACTA: Gramática resultante de considerar los tokens como símbolos terminales y eliminar aquellas producciones en las que derivan los tokens.

TABLA DE SÍMBOLOS: Lugar de almacenamiento temporal (durante la fase de compilación y, en lenguajes orientados a objetos, durante la ejecución) debido a la necesidad de identificar los lexemas en la ocurrencia de cada token durante las fases posteriores de traducción.

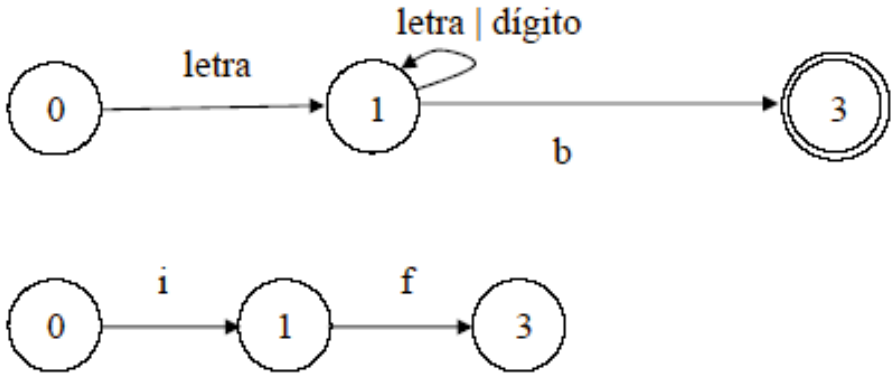
TABLA DE TOKENS: Tabla formada por tantas filas como tokens se hayan identificado. Las columnas tendrán la siguiente información (Nombre, Código, Atributos, Expresión Regular).

IMPLEMENTACIÓN DEL ANALIZADOR DE LÉXICO

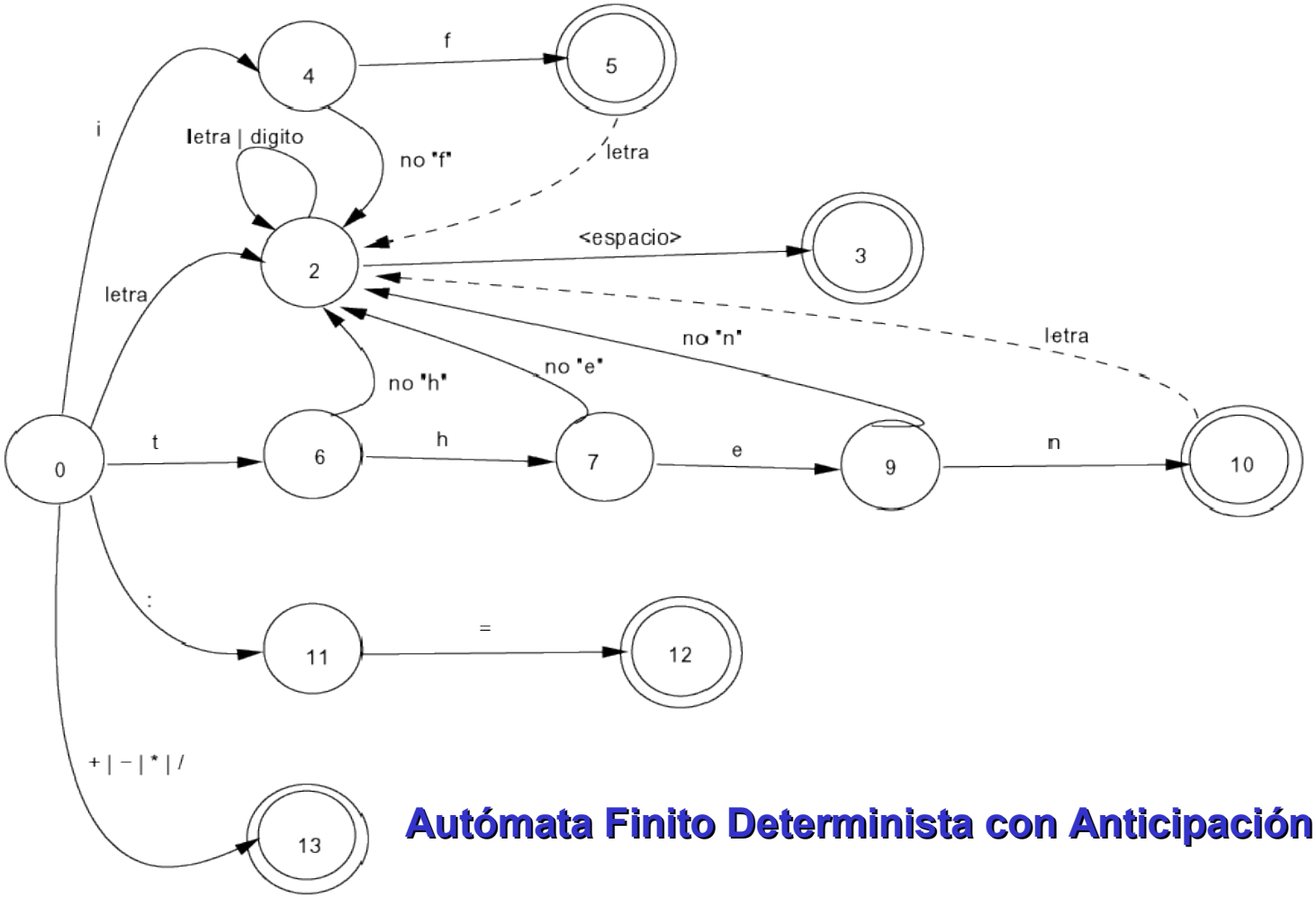
Autómata reconocedor de patrones: Autómata con anticipación.

Autómata reconocedor de palabras: Autómata finito + Tabla de palabras reservadas.

PATRONES: Cada patrón es reconocido por un Autómata finito determinista



Problema de reconocer más de un patrón a la vez…



Autómata Finito Determinista con Anticipación

2.4 Tratamiento de los Errores (1)

Un traductor debe adoptar alguna estrategia para detectar, informar y recuperarse para seguir analizando hasta el final.

Las respuestas ante el error pueden ser:

• **Inaceptables**: Provocadas por fallos del traductor, entrada en lazos infinitos, producir resultados erróneos, y detectar sólo el primer error y detenerse.

• **Aceptables**: Evitar la avalancha de errores (mala recuperación) y, aunque más complejo, informar y reparar el error de forma automática.

2.4 Tratamiento de los Errores (2)

La conducta de un Analizador de Léxico es el de un Autómata finito o “scanner”.

Detección del error

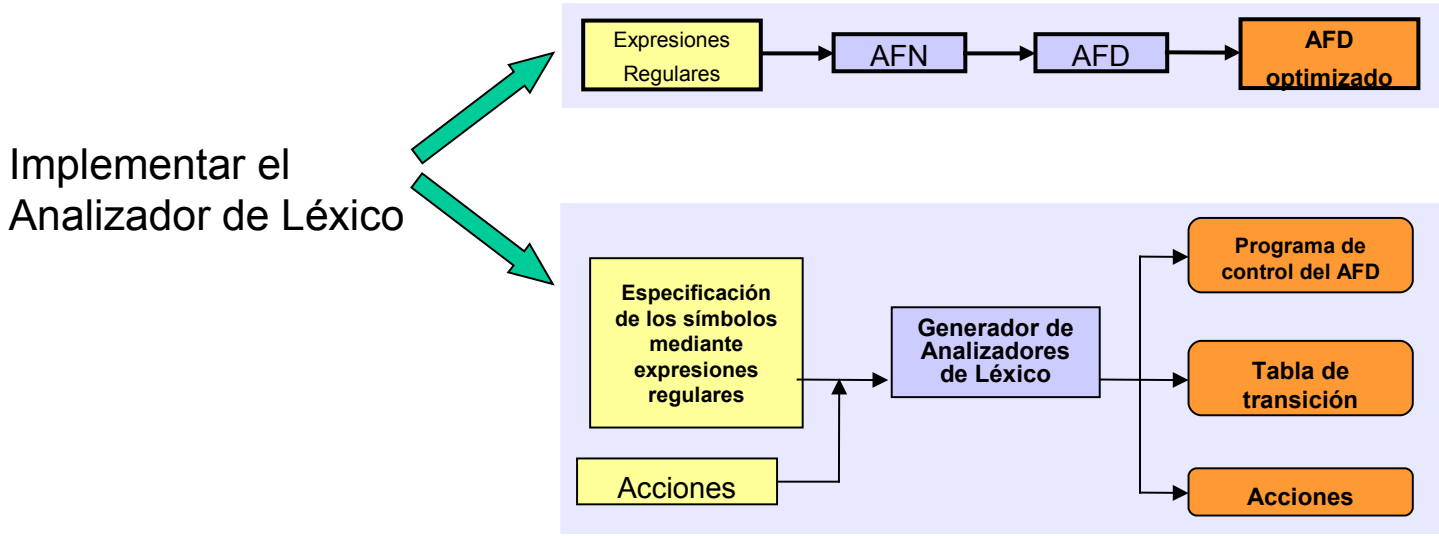
El analizador de Léxico detecta un error cuando no existe transición desde el estado que se encuentra con el símbolo de la entrada. El símbolo en la entrada no es el esperado.

Estrategias que se pueden adoptar

- Pasar al estado INICIAL ignorando los símbolos previos del lexema, iniciando de nuevo el proceso de identificación de los tokens.
- Proceso de SINCRONIZACIÓN, también conocido como **modo panic**. Consiste en ignorar los símbolos de la entrada no esperados hasta que aparezca un símbolo esperado. Esta estrategia es buena cuando aparecen símbolos extraños de forma extra.
- Reparar los errores mediante transformación de los lexemas: (borrar caracteres extraños, insertar caracteres que falten, intercambio de caracteres y sustitución de caracteres).

2.5 LEX (Generador de Analizadores de Léxico)

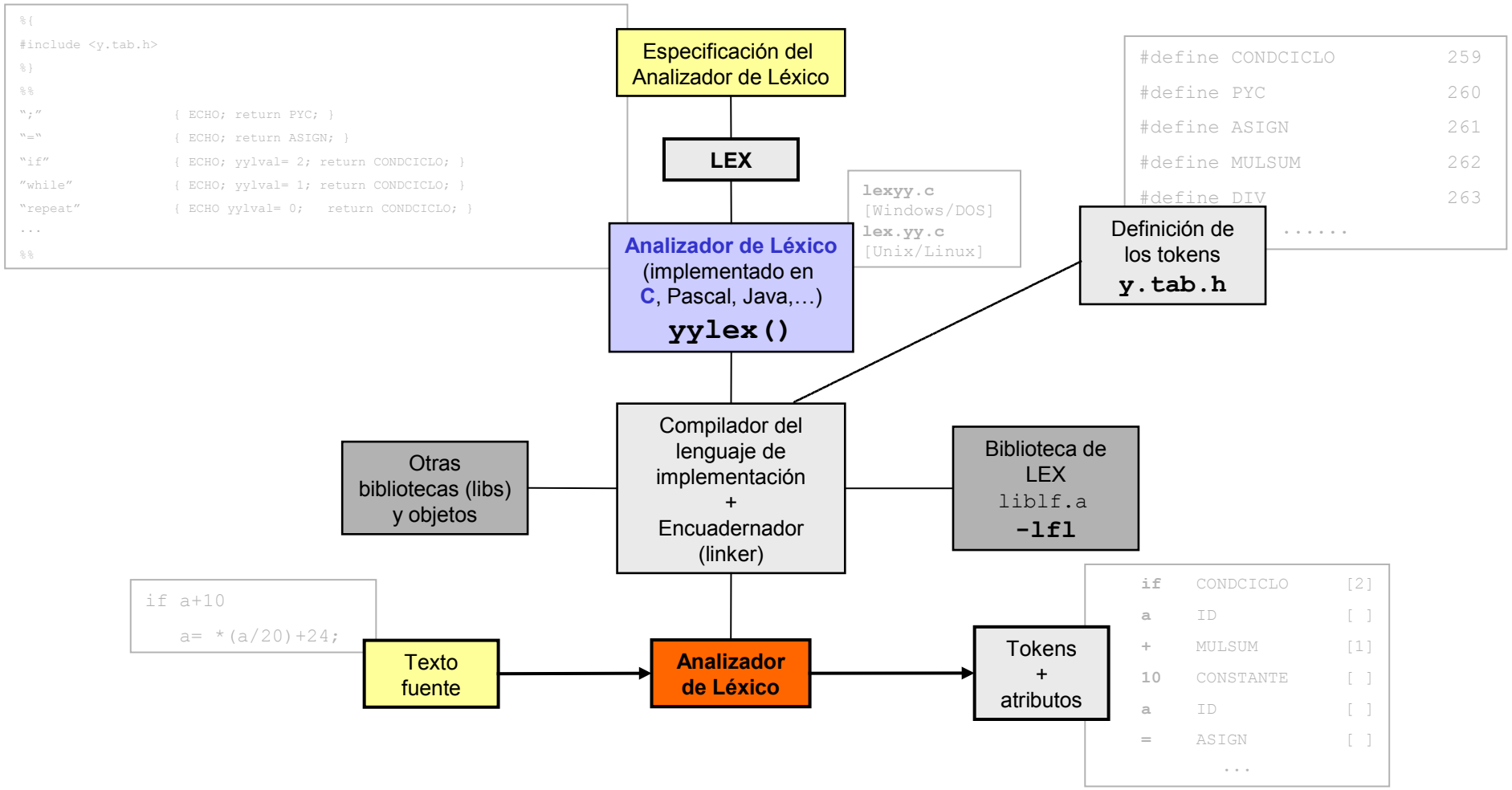
Introducción (1)



Generador de analizadores de léxico: **LEX**

- Construir analizadores de léxico como programas independientes.
- Construir la rutina léxica que utiliza el YACC para generar analizadores sintácticos.
- Obtener programas que realicen análisis, estadísticas o cualquier tipo de transformación a nivel léxico en un texto fuente.

Especificación LEX (1)



Especificación LEX (2)

```
{Definiciones}
%%
{Reglas}
%%
{Procedimientos del usuario}
```

El mínimo programa LEX es:

```
%%
```

Su acción es la de copiar el fichero de entrada en uno de salida.

Definiciones

Pueden contener:

- De forma opcional, **código C** con el siguiente formato:

```
%{
                    /* código C */
%}
```

- **Definición** de un nombre mediante una **expresión regular** con la forma:

Nombre expresión_regular

Ejemplo:

```
letra    [a-zA-Z]
digito   [0-9]
```

NOTA: Su posterior uso requerirá que vaya el nombre entre llaves. Según el ejemplo, se tendrían Que usar del siguiente modo: `{letra}` ó `{dígito}`

Reglas

Esta parte de la especificación **LEX** contiene los patrones que representan todos los símbolos a reconocer por el analizador de léxico, así como las acciones a realizar cuando son identificados.

Patrón (expresión regular) **{ Acción }**

Ejemplo:

```
letra    [a-zA-Z_]
digito   [0-9]
%%
...
{letra}({digito}|{letra})*        { return IDENT; }
...
%%
```

Operadores (1)

Operadores en *lex*	
*	Especifica la repetición de 0 o más veces (ejemplo $a*$).
+	Especifica la repetición de 1 o más veces (ejemplo $a+$).
( )	Útiles para agrupar sub-expresiones.
|	Especifica alternativa (ejemplo $a(bc\|de)$).
?	Especifica la opcionalidad del carácter precedente (ejemplo $ab?c$ reconoce ac o abc).
[]	Define una clase de caracteres, ejemplo $[xy]$. Representa al carácter x o al carácter y.
[m-n]	El operador — entre corchetes define un rango de caracteres entre m y n, según la tabla ascii. Fuera de los corchetes carece de significado, ejemplo: $[x-z]$ representa al carácter x, al carácter y o al carácter z.
[^x]	Indica cualquier carácter menos el indicado o indicados entre corchetes, ejemplo: $[\hat{}abc]$ representa cualquier carácter excepto a, b o c.
[\]	Indica una constante de tipo carácter siguiendo la notación del lenguaje C como rango de caracteres imprimibles.
^	Especifica comienzo de línea.
$	Especifica fin de línea.
.	Especifica cualquier carácter excepto el fin de línea
"x"	Especifica que el carácter entre comillas no es un operador de Lex. (ejemplo "$"no nos indica fin de línea).
\x	Especifica que el carácter que le sigue x no es un operador de Lex (igual que el caso anterior).
{ }	Especifica un patrón definido anteriormente, ejemplo: — digito $[0-9]$. — letra $[a-z, A-Z]$. — identificador6 $\{letra\}(\{letra\}\|\{digito\})*$.
x{m,n}	Especifica que x aparece desde m a n veces, ejemplo: identificador6 $\{letra\}(\{letra\}\|\{digito\}) * \{0,5\}$.
x/y	Reconoce un carácter cuando es seguido por un segundo caracter. En el ejemplo, solo reconoce x si va seguido de y.
\n	Indica fin de línea.
\t	Indica una tabulación.
\\	Indica el símbolo $.
\b	Indica un espacio en blanco.

Operadores (2)

	patrón no especificado
*	Especifica la repetición de 0 o más veces (ejemplo $a*$).
+	Especifica la repetición de 1 o más veces (ejemplo $a+$).
( )	Útiles para agrupar sub-expresiones.
|	Especifica alternativa (ejemplo $a(bc\|de)$).
?	Especifica la opcionalidad del carácter precedente (ejemplo $ab?c$ reconoce ac o abc).
[]	Define una clase de caracteres, ejemplo $[xy]$. Representa al carácter x o al carácter y.
[m-n]	El operador — entre corchetes define un rango de caracteres entre m y n, según la tabla ascii. Fuera de los corchetes carece de significado, ejemplo: $[x-z]$ representa al carácter x, al carácter y o al carácter z.
[^x]	Indica cualquier carácter menos el indicado o indicados entre corchetes, ejemplo: $[^abc]$ representa cualquier carácter excepto a, b o c.
[\]	Indica una constante de tipo carácter siguiendo la notación del lenguaje C como rango de caracteres imprimibles.
^	Especifica comienzo de línea.
$	Especifica fin de línea.
.	Especifica cualquier carácter excepto el fin de línea
"x"	Especifica que el carácter entre comillas no es un operador de Lex. (ejemplo "$"no nos indica fin de línea).
\x	Especifica que el carácter que le sigue x no es un operador de Lex (igual que el caso anterior).
{ }	Especifica un patrón definido anteriormente, ejemplo: — digito $[0-9]$. — letra $[a-z, A-Z]$. — identificador6 $\{letra\}(\{letra\}\|\{digito\})*$.
x{m,n}	Especifica que x aparece desde m a n veces, ejemplo: identificador6 $\{letra\}(\{letra\}\|\{digito\})*\{0,5\}$.
x/y	Reconoce un carácter cuando es seguido por un segundo caracter. En el ejemplo, solo reconoce x si va seguido de y.
\n	Indica fin de línea.
\t	Indica una tabulación.
\\	Indica el símbolo $.
\b	Indica un espacio en blanco.

Ejemplos: Expresiones regulares según la notación de LEX.

`[a-z]`	Representa cualquiera de las letras minúsculas de la `a` a la `z`.
`[-a-z]`	Representa cualquiera de las letras minúsculas de la `a` a la `z` y el signo `-`.
`(ab|cd+)?(ef)*`	Cadenas válidas: `abef`, `cdf`, `cdddd`, `efefef`.

Acciones

Las acciones representan la consecuencia de que el analizador de léxico haya reconocido un determinado lexema perteneciente a un patrón. El formato de las acciones es el siguiente:

<Acción>	::=	**<Acción_Simple>** | "{" **<Acción_Compuesta>** "}"
<Acción_Simple>	::=	";" | **<Acción_Especial>** ";" | <sentencia_C> ";" | "|"
<Acción_Compuesta>	::=	**<Acción_Simple>** { **<Acción_Simple>** }
<Acción_Especial>	::=	ECHO | REJECT | BEGIN

Acciones por defecto

- Para los caracteres **reconocidos** → No realiza ninguna acción.
- Para los caracteres **no reconocidos** → Copiarlos en el fichero de salida (stdout).

Acciones ; y |

- `;` Representa la **acción por defecto** para los caracteres reconocidos.
- `|` Indica que la acción asociada al patrón es la misma que la del siguiente patrón.

Acciones Especiales

- **`ECHO`**: Copia la secuencia de caracteres reconocidos en el fichero de salida.
- **`REJECT`**: Se usa para las reglas ambiguas.
- **`BEGIN`**: Útil cuando se desea tener construcciones sensibles al contexto por la izquierda.

Variables y Funciones LEX

Variables y funciones *lex*	
`char *yytex`	Variable que contiene el lexema reconocido. Al ser una cadena de caracteres, ésta finaliza con el carácter `\0`.
`int yyleng`	Número de caracteres reconocidos.
`int yylineno`	Línea actual del fichero de entrada.
`FILE *yyin`	Fichero de entrada, por defecto es **stdin**.
`FILE *yyout`	Fichero de salida, por defecto es **stdout**.
`void yymore()`	El siguiente token reconocido es añadido al actual en `yytex`, en lugar de reemplazarlo.
`int yywrap()`	Función llamada por el analizador de léxico cuando llega a fin de fichero y que, por defecto, devuelve el valor `1`.
`int yyless(int n)`	Devuelve los n primeros caracteres del token actual al buffer de entrada para que vuelvan a ser analizados y así reajustar `yytex` y `yyleng` convenientemente.
`int input(int *c)`	Lee el siguiente carácter de la entrada.
`void unput(int c)`	Devuelve el carácter `c` a la entrada.
`void output(int c)`	Escribe el carácter `c` en la salida.

Ejemplo 2.1:

```
%%
[a-z]+    printf ("%s", yytext);
%%

main ()
{
   yylex() ;
}
```

- **Ejemplo 2.1 = Ejemplo 2.4**.
- **Ejemplo 2.2**:
  - **Lexemas válidos**: no hace nada.
  - **Lexemas no válidos**: los copia en el archivo de salida.
- **Ejemplo 2.3**:
  - **Lexemas válidos**: los copia en el archivo de salida.
  - **Lexemas no válidos**: no hace nada.

Ejemplo 2.2:

```
%%
[a-z]+    ;
%%

main ()
{
   yylex() ;
}
```

Ejemplo 2.3:

```
%%
[a-z]+    ECHO;
.         ;
%%

main ()
{
   yylex() ;
}
```

Ejemplo 2.4:

```
%%
[a-z]+    ECHO;
%%

main ()
{
   yylex() ;
}
```

Ejemplo 2.8: Contador de palabras, caracteres y líneas de un archivo.

```
%{
static unsigned num_caracteres = 0;          /* # de caracteres */
static unsigned num_palabras = 0;            /* # de palabras   */
static unsigned num_lineas= 0;               /* # de lineas     */
%}

%%

\n          num_caracteres += 2, ++num_lineas;          /* El límite de línea en MS-DOS es CR LF */
[^ \t\n]+   ++num_palabras, num_caracteres+= yyleng;
.           ++num_caracteres ;

%%

main ()
{
   yylex ();
   printf ("%d\t%d\t%d\n", num_caracteres, num_palabras, num_lineas) ;
   exit (0);
}
```

Reglas Ambiguas (Acción REJECT)

El **LEX** permite especificaciones ambiguas, es decir, especificaciones donde dos o más patrones se pueden corresponder con una misma secuencia de caracteres. En este caso la actuación del analizador de léxico será:

1. Elige la secuencia de caracteres más larga que corresponde con un patrón.
2. A igualdad de caracteres opta por el primer patrón, es decir, el definido en primer lugar.

Ejemplo 2.9: Especificación ambigua. A la derecha se muestra un ejemplo de ejecución.

```
%%
end         { ECHO; printf (" regla 1\n");
endif       { ECHO; printf (" regla 2\n");
a[cd]+      { ECHO; printf (" regla 3\n");
a[bc]+      { ECHO; printf (" regla 4\n");
%%

main ()
{
   yylex() ;
}
```

```
end
end regla 1

endif
endif regla 2

acddd
acddd regla 3

acccc
acccc regla 3

abbbb
abbbb regla 4
```

REJECT significa ir a la siguiente alternativa (patrón) y reexplorar el texto de entrada.

Ejemplo 2.10:

```
%{
int e, a ;
%}

%%

el                  e++ ;
ella                a++ ;
\n                  |
.                   ;

%%

main ()
{
   yylex() ;
   printf ("%d, %d\n", e, a);
}
```

Ejemplo 2.11:

```
%{
int e, a ;
%}

%%

el                  { e++ ; REJECT }
ella                { a++ ; REJECT }
\n                  |
.                   ;

%%

main ()
{
   yylex() ;
   printf ("%d, %d\n", e, a);
}
```

Ejemplo 2.12:

```
%%

a[cd]+   { ECHO; printf ("...regla 1\n"); REJECT }
a[cb]+   { ECHO; printf ("...regla 2\n"); REJECT }
\n       |
.        ;

%%

main ()
{
   yylex() ;
}
```

```
acddd
acddd ...regla 1
acdd ...regla 1
acd ...regla 1
ac ...regla 1
ac ...regla 2
```

```
acbbb
acbbb ...regla 2
acbb ...regla 2
acb ...regla 2
ac ...regla 1
ac ...regla 2
```

Ejemplo 2.13:

```
%%

a[cd]+   { ECHO; printf ("...regla 1\n"); REJECT }
a[cb]+   { ECHO; printf ("...regla 2\n"); }
\n       |
.        ;

%%

main ()
{
   yylex() ;
}
```

```
acddd
acddd ...regla 1
acdd ...regla 1
acd ...regla 1
ac ...regla 1
ac ...regla 2
```

```
acbbb
acbbb ...regla 2
```

Ejemplo 2.14:

```
%{
   int i;
%}

%%

^a      {i='a'; ECHO; }
^b      {i='b'; ECHO; }
^c      {i='c'; ECHO; }
\n      {i=0  ; ECHO; }

letra   { switch (i)
          {  case 'a' : printf (" primera\n"); break ;
             case 'b' : printf (" segunda\n"); break ;
             case 'c' : printf (" tercera\n"); break ;
             default  : ECHO ; break ;
          }
        }

%%

main ()
{
   yylex () ;
}
```

```
acelera que nos vamos letra
acelera que nos vamos  primera

buscando desesperadamente letra
buscando desesperadamente  segunda

cada vez se ve mejor letra
cada vez se ve mejor  tercera
```

Condiciones START (Sensibilidad al contexto izquierdo)

Permiten activar o desactivar patrones dependiendo del entorno en el que se encuentren (contexto).

- Para definir los nombres de las condiciones, situaremos en la parte de las definiciones:

```
%START nombre_de_condicion_1 ... nombre_de_condicion_N
```

- La activación/desactivación de una condición se realizará mediante:
  - **`BEGIN (nombre_condición)`**: Activación.
  - **`BEGIN 0`**: Desactivación de las condiciones.
- En las reglas se hará referencia a una condición de la siguiente forma:

```
<nombre_de_condición>      Patrón
```

Condiciones START exclusivas

En ellas cuando se activa una condición se desactivan todos los patrones que no comiencen por el indicativo **`<nombre_de_condición>`**. El formato solo difiere de las anteriores en la propia definición, es decir:

```
%X nombre_de_condicion_1 ... nombre_de_condicion_N
```

Ejemplo 2.15: Condiciones START.

```
%START A B C

%%

^a          { BEGIN A; ECHO; }
^b          { BEGIN B; ECHO; }
^c          { BEGIN C; ECHO; }
\n          { BEGIN 0; ECHO; }

<A>letra            printf (" primera\n");
<B>letra            printf (" segunda\n");
<C>letra            printf (" tercera\n");

%%

main ()
{
   yylex () ;
};
```

```
acelera que nos vamos letra
acelera que nos vamos  primera

buscando desesperadamente letra
buscando desesperadamente  segunda

cada vez se ve mejor letra
cada vez se ve mejor  tercera
```

Ejemplo 2.16: Condiciones START exclusivas.

```
%X A B C

%%

^a          { BEGIN A; ECHO; }
^b          { BEGIN B; ECHO; }
^c          { BEGIN C; ECHO; }
\n          { BEGIN 0; ECHO; }
Numero      { printf ("##"); }

<A>letra            printf (" primera\n");
<B>letra            printf (" segunda\n");
<C>letra            printf (" tercera\n");

%%

main ()
{
   yylex () ;
};
```

```
acelera que nos vamos letra
acelera que nos vamos  primera

buscando desesperadamente letra
buscando desesperadamente  primera

cada vez se ve mejor letra
cada vez se ve mejor  primera

buscando desesperadamente letra
buscando desesperadamente  segunda

cada vez se ve mejor letra
cada vez se ve mejor  segunda

acelera que nos vamos letra
acelera que nos vamos  segunda

dadas las circunstancias numero
dadas las circunstancias ##

cuando comencemos así letra
cuando comencemos así  tercera

dadas las circunstancias numero
dadas las circunstancias numero
```

Ejemplo 2.17: Analizador léxico para un lenguaje simple.

```
%{
#include <stdlib.h>
#include <string.h>
#include "tabla.h"

int linea_actual=0 ;
%}
letra           [a-zA-Z]
digito          [0-9]
alphanum        [a-zA-Z_0-9]
blanco          [ \t]
otros           .
%%
"("                             return PIZ;
")"                             return PDE;
"+"                             return SUM;
"-"                             return SUB;
"*"                             return MUL;
"/"                             return DIV;
":="                            return ASI;
{letra}{alphanum}*              return ID;
{digito}+                       return ENT;
{blanco}+                       return BL;
\n                              ++linea_actual ;
{otros}                         printf ("\n(Linea %d) Error léxico: token %s\n", yylineno, yytext);
%%
main ()
{
   int val;
   val= yylex() ;
   while (val != 0)
   {
               printf (" %d\n", val);
               val= yylex() ;
   }
   exit (1);
}
```

```
/* tabla.h */

#define SUM   257
#define MUL   258
#define DIV   259
#define SUB   260
#define ENT   261
#define PDE   262
#define PIZ   263
#define ASI   264
#define ID    265
#define BL    266
```

Ejemplo 2.18: Dado el lenguaje descrito por el siguiente conjunto de producciones

```
 Programa → Expresion ; Programa
Expresion → Expresion + Expresion
          | Expresion - Expresion
          | Expresion * Expresion
          | Expresion / Expresion
          | ( Expresion )
          | - Expresion
          | Numero
   Numero → Digito | Numero
   Digito → 0 | 1 | ··· | 9
```

La **tabla de tokens** con máximo nivel de abstracción sería:

Token	Número	Patrón
OPBIN	257	+ | * | /
OPUNA	258	-
PARIZQ	259	(
PARDCH	260	)
PCOMA	261	;
NUMERO	262	digito{digito}

```
%{
#include "tabla.h"
%}

numero      [0-9]+
otros       .

%%

"+"         return OPBIN ;
"*"         return OPBIN ;
"/"         return OPBIN ;
"-"         return OPUNA ;
"("         return PARIZQ ;
")"         return PARDCH ;
";"         return PCOMA ;
[ \t\n]     ;
{numero}    return NUMERO ;
{otros}     printf ("\n(Linea %d) Error léxico: token %s\n",
                    yylineno, yytext);
%%

main ()
{
            int val;

            val= yylex()
            while (val != 0)
                        printf (" %d \n", val) ;
                        val= yylex() ;
            exit (0);
}
```

```
#define   OPBIN     257
#define   OPUNA     258
#define   PARIZQ    259
#define   PARDCH    260
#define   PCOMA     261
#define   NUMERO    262
```