# Compiladores
# Unidad 2: Análisis Léxico
# Sintaxis léxica

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota introducimos, formalizamos y ejemplificamos la sintaxis léxica como el primer nivel de la sintaxis concreta en el diseño de lenguajes de programación. Partimos de la separación clásica entre nivel léxico y nivel gramatical, fijamos el alfabeto, las categorías léxicas y la función $\mathsf{Lex}$ que asigna a cada categoría el conjunto de lexemas válidos, y mostramos cómo el lenguaje léxico total se construye como una unión disjunta de dichos conjuntos. Para aterrizar las ideas, trabajaremos con un fragmento de MiniLisp y cerraremos con un ejemplo de analizador léxico ad-hoc en Haskell que refleja fielmente la especificación teórica.

En el proceso de traducción, la sintaxis léxica responde a la pregunta "¿qué cadenas son palabras del lenguaje y de qué tipo?", preparando el terreno para que la sintaxis libre de contexto se ocupe después de "¿cómo se combinan esas palabras?". En términos formales, fijamos $\Sigma$ como alfabeto, un conjunto finito $C$ de categorías (identificadores, literales, delimitadores, reservadas, etc.) y una aplicación $\mathsf{Lex} : C \to \mathcal{P}(\Sigma^*)$ que determina, para cada categoría, el lenguaje regular de sus lexemas. Con esta base, modelamos cada token como un par $\langle cat, atrib \rangle$ y justificamos por qué el reconocimiento léxico puede realizarse con autómatas finitos (o, pedagógicamente, con un *scanner* manual), manteniendo separación de preocupaciones y modularidad en el compilador.

## 1. Sintaxis Léxica de un Lenguaje de Programación

En el estudio y la construcción de lenguajes de programación, resulta esencial fijar con precisión cómo deben escribirse los programas: qué símbolos están permitidos, qué "palabras" del lenguaje existen y de qué manera pueden combinarse para formar expresiones y sentencias. Este conjunto de reglas textuales se denomina sintaxis concreta. Dentro de ella distinguimos dos niveles: (1) la sintaxis léxica, que determina las unidades mínimas significativas (componentes léxicos o *tokens*) a partir de símbolos; y (2) la sintaxis libre de contexto (o gramatical), que describe cómo esos componentes léxicos se organizan en estructuras mayores como expresiones, comandos o módulos. Esta separación conceptual aporta claridad, modularidad y una base rigurosa para el diseño de traductores (intérpretes y compiladores).

**Definición 1. (Sintaxis concreta)**

*Llamamos* ***sintaxis concreta*** *de un lenguaje a un par* $(L, G)$ *donde:*

- $L$ *es la* ***parte léxica*** *que fija el universo de componentes léxicos posibles (sus clases o categorías y las cadenas de caracteres que cada clase admite). Esta parte corresponde a los* ***lenguajes regulares (Tipo 3 en la jerarquía de Chomsky)****, ya que puede describirse con expresiones*

*regulares y modelarse con autómatas finitos.*

- *$G$ es la **parte gramatical** que fija las formas textuales válidas a nivel de expresiones, sentencias y programas completos, **tratanto los componentes léxicos como unidades**. Esta parte corresponde a los **lenguajes libres de contexto (Tipo 2 en la jararquía de Chomsky)**, que pueden describirse con gramáticas libres de contexto y analizarse mediante autómatas de pila*

Intuitivamente $L$ responde a "qué piezas mínimas existen y cómo se escriben"; $G$ responde a "cómo se ensamblan esas piezas". En la práctica, un traductor debe verificar primero que un texto fuente cumpla con la sintaxis concreta: (1) que sus piezas mínimas estén bien formadas (nivel léxico) y (b) que esas piezas estén en un orden permitido (nivel gramatical). Sólo después tiene sentido pasar a aspectos semánticos o de traducción de significados.

**Ejemplo 1**

Consideremos la siguiente expresión en un lenguaje aritmético mínimo: `3 + 42 + 7`

**(a) Qué afirma la sintaxis léxica en este caso (en palabras):**
- La cadena contiene cinco "fragmentos" significativos: `3`, `+`, `42`, `+`, `7`
- Cada `3`, `42`, `7` **cuenta como número entero** (una categoría léxica).
- Cada `+` **cuenta como operador de suma** (otra cateogoría léxica).

La secuencia completa es: `[Numero, +, Número, +, Numero]`

**(b) Qué afirma la sintaxis gramatical en este caso (en palabras):**
- Un **número** por sí sólo es una **expresión** válida.
- Una **expresión** seguida de `+` y de un **númmero** forma una expresión mayor.
- Por lo tanto, encadenar "`número + numero + número` produce una **expresión aritmética** correcta.

A partir de este punto, nos centraremos exclusivamente en la parte léxica: qué es, cómo se define y qué unidades típicas (componentes léxicos) encontramos en los lenguajes de programación. En la siguiente unidad retomaremos los temas asociados a la sintaxis gramatical.

**Definición 2. (Sintaxis léxica)**

*Sea $\Sigma$ un **alfabeto finito** (el conjunto de símbolos permitidos en los textos fuente de un lenguaje $\mathcal{L}$). La sintais léxica establece:*

1. *Un conjunto finito $C$ de **categorías léxicas** (p. ej., Identificador, Entero, Real, Cadena, OperadorSuma, ParéntesisAbre, ParéntesisCierra, PalabraReservada, etc.).*

2. *Para cada $c \in C$, un **conjunto de lexemas válidos** $\mathsf{Lex}(c) \subseteq \Sigma^\star$ que describen exactamente que **cadenas de símbolos** pertenecen a esa categoría.*

**Observación 1**

Es importante remarcar que cada conjunto $\mathsf{Lex}(c)$ es un **lenguaje regular**, y por tanto puede definirse utlizando formalismos bien establecidos como **expresiones regulares** y/o **autómatas finitos**.

Con ello, podemos reunir la parte léxica en la estructura:

$$L = (\Sigma, C, \mathsf{Lex})$$

donde $\mathsf{Lex}$ es la asignación $c \mapsto \mathsf{Lex}(c)$.

El lenguaje léxico total es la unión disjunta de los conjuntos de lexemas:

$$L_{lex} = \biguplus_{c \in C} \mathsf{Lex}(c)$$

entendida como "totas las cadenas que el sistema reconoce como algún componente léxico, sabiendo a qué categoría pertenecen.

**Ejemplo 2**

En un lenguaje hipotético, podemos definir:

- $\Sigma = \{0, 1, 2, \ldots, 9, +, -\}$
- $C = \{\mathit{Número}, \mathit{Operador}\}$
- $\mathsf{Lex}(\mathit{Número}) = \{x \ : \ x \text{ es una cadena de uno o más dígitos}\}$
- $\mathsf{Lex}(\mathit{Operador}) = \{+, -\}$

De este modo, la cadena `25 - 7 + 3` se reconoce léxicamente como:

```
[(Número, 25), (Operador, -), (Numero, 7), (Operador, +), (Numero, 3)]
```

La lista de parejas generada en el ejemplo anterior usa la definición de sintaxis léxica y genera una clasificación con la función $\mathsf{Lex}$. Llamamos a cada una de las parejas de esta lista componente léxico.

**Definición 3. (Componente léxico)**

*Una vez fijado $L_{lex}$, modelamos cada **unidad reconocida** como un **componente léxico**:*

$$\langle cat, atrib \rangle$$

- *$cat \in C$ es la **categoría léxica** (qué "tipo de palabra" del lenguaje es).*
- *$atrib$ es un **conjunto (o registro) de atributos** asociados al componente léxico. En este caso consideraremos únicamente el **valor**, es decir, la interpretación interna que el compilador asigna a la cadena reconocida.*

**Ejemplo 3**

Retomemos el caso del Ejemplo 2 con la cadena `25 - 7 + 3`. Cada uno de los componentes léxicos tiene su categoría y valor:

- `25` $\rightarrow \langle \textit{Número}, \{\textit{valor } = 25\}\rangle$
- `-` $\rightarrow \langle \textit{Operador}, \{\textit{valor } = -\}\rangle$
- `7` $\rightarrow \langle \textit{Número}, \{\textit{valor } = 7\}\rangle$
- `+` $\rightarrow \langle \textit{Operador}, \{\textit{valor } = +\}\rangle$
- `3` $\rightarrow \langle \textit{Número}, \{\textit{valor } = 3\}\rangle$

Dado que sólo tenemos un atributo (valor), la salida final puede omitir la notación de registro. Lo cuál explica la salida del ejemplo anterior:

```
[(Número, 25), (Operador, -), (Numero, 7), (Operador, +), (Numero, 3)]
```

De esta forma, un compilador ya no trabaja directamente con cadenas de símbolos, sino con unidades abstractas que encapsulan la categoría y el valor de cada componente léxico.

**Observación 2**

En esta nota, y por motivos de claridad, consideraremos como atributo únicamente al **valor**. Sin embargo, en un compilador real es común incluir otros atributos, como:

- Posición: Línea y columna donde aparece el componente, útil para reportar errores.
- Tipo: si es entero, flotante, caractere, etc.
- Información adicional: En identificadores, por ejemplo, puede incluir referencias a la tabla de símbolos.

**Componentes léxicos clásicos en lenguajes de programación**

1. Identificadores: Nombres dados por la persona programadora (`x, contador, toString`)
2. Palabras reservadas: Cadenas con significado fijo (`if, while, return`)
3. Literales numéricas: Valores escritos directamente (`42, 3.14, 0xFF`)
4. Literales de cadena y carácter: `"hola"`, `'a'`
5. Operadores y delimitadores: `+,-,*,(,)`.
6. Espacios en blanco y comentarios: Útiles como separadores o anotaciones, aunque algunos lenguajes los tratan como componentes léxicos especiales (ej. PYTHON con la indentación).

Cada uno de estos casos puede representarse como componente léxico $\langle \textit{cat, atr}\rangle$, garantizando así que los programas se construyen con unidades mínimas claras y bien definidas.

# 2. Analizadores léxicos

Hemos visto que el objetivo de la sintaxis léxica es describir qué cadenas se aceptan como "palabras" del lenguaje (componentes léxicos), cómo se clasifican en categorías léxicas y cómo cada uno puede portar atributos. Ese marco establece qué debe reconocerse. En esta sección explicamos cómo se reconoce de manera sistemática en un traductor: mediante un analizador léxico, el módulo que materializa la sintaxis léxica y produce la secuencia de componentes léxicos que se usarán en las fases siguientes.

Un analizador léxico (también *lexer* o *scanner*) es el componente de un traductor que lee una cadena (el programa fuente, un elemento de $\Sigma^\star$) y emite una lista de componentes léxicos (*tokens*), cada uno con su categoría y (al menos) un atributo (en este curso el valor). Conceptualmente:

- **Entrada:** Una única cadena de caracteres (el texto del programa).
- **Salida:** Una secuencia (lista) de componentes léxicos $\langle cat, atr \rangle$ , en el orden en que aparecen en el código.

**Observación 3**

A partir de este punto emplearemos *token* en lugar de componente léxico con el fin de acoplarnos a la literatura estándar sobre el tema. Sin embargo, es importante que quede clara su traducción al español y que estamos hablando de lo mismo.

**Observación 4**

En la práctica moderna **scanner** y **lexer** suelen usarse como **sinónimos** del mismo módulo. Algunas fuentes reservan *scanner* para el acto de **recorrer** el texto (saltando espacios, comentarios, etc.) y *lexer* para la **clasificación** de lexemas en categorías. En compiladores reales ambas tareas forman un único subsistema el **analizador léxico**. Por comodidad usaremos *lexer* en estas notas, pero es importante entender la diferencia.

Esta fase "realiza" la sintaxis léxica que formalizamos en la sección anterior (las $\mathsf{Lex}(c)$ que describen los lexemas de cada categoría) y prepara el terreno para que la sintaxis libre de contexto opere sobre *tokens* y no sobre símbolos.

### Dos enfoques principales de implementación

#### Enfoque ad-hoc "a manita"

Se programa un algoritmo que recorre la cadena símbolo por símbolo, agrupa dígitos para formar números, letras/digitos/... para identificadores, reconoce símbolos individuales, consulta una tabla de palabras reservadas, etc. Es directo y útil pedagógicamente, pero escala peor a lenguajes con detalles léxicos complejos (comentarios anidados, Unicode completo, cadenas con escape, *raw strings*, *f-strings*, etc.).

#### Enfoque formal (Teoría de lneguajes regulares)

Se especifican los *tokens* como Expresiones Regulares (ER); luego, por construcciones estándar, se obtienen Autómatas Finitos No Deterministas (AFN) equivalentes y, mediante determinación, Autómatas Finitos

Deterministas (AFD) eficientes (tiempo lineal en el tamaño de la entrada). No desarrollaremos en esta nota aún estos conceptos; basta subrayar que este método sistematiza el reconocimiento léxico y conecta la práctica del compilador con la teoría de lenguajes regulares.

**Ejemplo 4**

A continuación ejemplificamos cómo construir un analizador léxico ad-hoc para *MiniLisp*. Primero se presenta la gramática objetivo y después el código del *lexer* que reconoce los *tokens* correspondientes.

**Gramática de MiniLisp (BNF)**

```
<expr> ::= <id>
         | <num>
         | <bool>
         | (+ <expr> <expr>)
         | (- <expr> <expr>)
         | (not <expr>)
         | (if0 <expr> <expr> <expr>)
         | (let (<id> <expr>) <expr>)
         | (letrec (<id> <expr>) <expr>)
         | (lambda (<id>) <expr>)
         | (<expr> <expr>)


<id> ::= a | b | foo | var1 | ...

<num> ::= ... | -1 | 0 | 1 | ...

<bool> ::= #t | #f
```

**Lexer ad-hoc en Haskell**

```haskell
data Token = TokenId String
           | TokenNum Int
           | TokenBool Bool
           | TokenSuma
           | TokenResta
           | TokenNot
           | TokenPA
           | TokenPC
           | TokenLet
           | TokenLetRec
           | TokenIf0
           | TokenLambda
          deriving(Show)

lexer :: String -> [Token]
lexer [] = []
lexer (' ' : xs) = lexer xs
lexer ('(' : xs) = TokenPA : lexer xs
lexer (')' : xs) = TokenPC : lexer xs
lexer ('+' : xs) = TokenSuma : lexer xs
lexer ('-' : xs) = TokenResta : lexer xs
lexer ('n':'o':'t':xs) = TokenNot : lexer xs
```

```
lexer ('#':'t':xs) = TokenBool True  : lexer xs
lexer ('#':'f':xs) = TokenBool False : lexer xs
lexer ('l':'e':'t':'r':'e':'c':xs) = TokenLetRec : lexer xs
lexer ('l':'e':'t':xs) = TokenLet : lexer xs
lexer ('i':'f':'0':xs) = TokenIf0 : lexer xs
lexer ('l':'a':'m':'b':'d':'a':xs) = TokenLambda : lexer xs
lexer (x:xs)
  | isDigit x = lexNum  (x:xs)
  | isAlpha x = lexAlph (x:xs)

lexNum :: String -> [Token]
lexNum cs = TokenNum (read num) : lexer rest
  where (num, rest) = span isDigit cs

lexAlph :: String -> [Token]
lexAlph cs = TokenId var : lexer rest
  where (var, rest) = span isAlpha cs
```

La implementación *ad-hoc* resulta ilustrativa, pero también compleja de mantener en lenguajes grandes. Por ello, la teoría de lenguajes regulares ofrece un enfoque sistemático que permite automatizar el proceso de construcción de analizadores léxicos. Los pasos son:

1. Definir expresiones regulares (ER) para cada categoría léxica.
2. Convertir cada ER en un AFN-$\varepsilon$ (autómata finito con transiciones épsilon)
3. Eliminar $\varepsilon$-transiciones para obtener un AFN equivalente.
4. Construir un AFD (autómata determinista) equivalente mediante determinización.
5. Minimizar el AFD para obtener la versión más eficiente.
6. Implementar como MDD (máquina discriminadora determinista), es decir, un programa eficiente que reconoce *tokens* en tiempo lineal.

Este proceso no se hace "a mano" en compiladores modernos: existen generadores de analizadorez léxicos automáticos, que implementan estas transformaciones internamente. El programa sólo provee un archivo de especificación (con reglas en forma de ER y acciones asociadas), y la herramienta produce el código fuente de una función `lexer`. Ejemplos clásicos son Lex (en C) y Alex (en Haskell).

El beneficio es evidente: Se elimina la necesidad de codificar manualmente cada detalle del reconocimiento léxico, se obtienen analizadores correctos y eficientes de forma autoomática, y se acelera el desarrollo de compiladores.

## 3. Conclusión

Al finalizar, contamos con una caracterización precisa del nivel léxico: qué símbolos y palabras reconoce el lenguaje, cómo se clasifican y cómo se materializan en tokens con atributos útiles para las fases posteriores. La construcción $L_{\text{lex}} = \biguplus_{c \in C} \mathsf{Lex}(c)$ garantiza que toda cadena aceptada llegue anotada con su

categoría, y el ejemplo de MiniLisp demuestra la traducción directa entre la teoría (lenguajes regulares y función Lex) y la práctica (un lexer ad-hoc en Haskell). Con esta infraestructura, la siguiente etapa (la sintaxis libre de contexto) puede operar sobre una secuencia bien tipada de *tokens*, simplificando el análisis sintáctico y preparando el camino para la verificación semántica y la generación de código.

## Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Boston, MA, USA: Addison-Wesley, 2006.

[2] J. E. Hopcroft, R. Motwani y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd. Boston, MA, USA: Addison-Wesley, 2007.

[3] T. W. Parsons, *Introduction to Compiler Construction*. New York, NY, USA: W. H. Freeman, 1992.

[4] M. S. Romero, *Lenguajes de Programación: Unidad 2, Sintaxis Concreta*, `https://lambdasspace.github.io/LDP/notas/ldp_n04.pdf`, Facultad de Ciencias, UNAM, 2024.

[5] M. S. Romero, *MiniLisp*, `https://github.com/lambdasspace/MiniLisp/tree/main`, Semestre 2025-1, 2025.