# Curso: Compiladores

# Tema 2. Análisis léxico

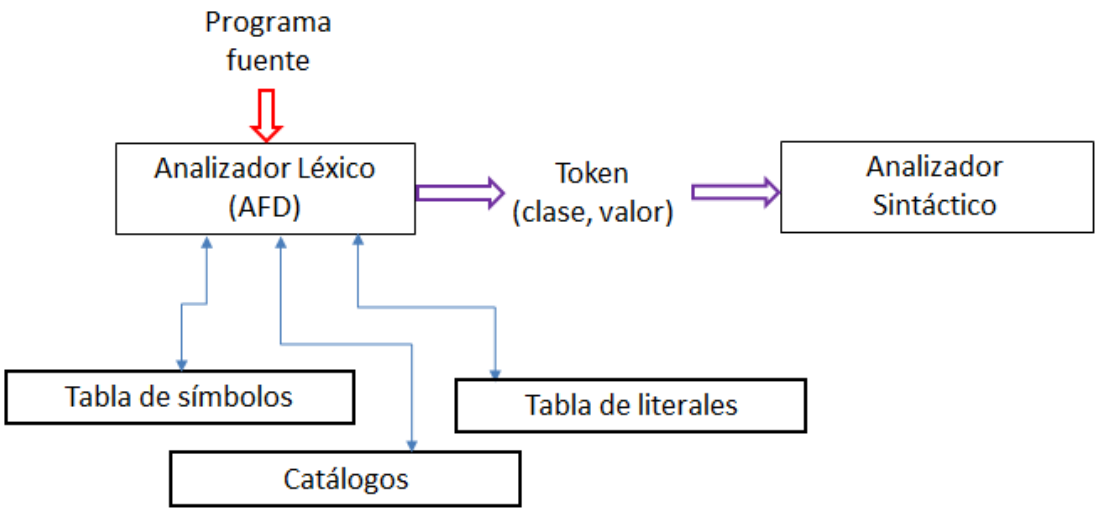
**Objetivo**.
El alumno construirá un analizador léxico a partir de la definición de clases de componentes léxicos.

**Contenido**.
2.1 Funciones de un Analizador léxico
2.2 Identificación de clases léxicas.
2.3 Estructura de las tablas de símbolos
2.4 Manejo de errores léxicos
2.5 Programación de un analizador léxico (scanner).
2.6 Generación automática de analizadores léxicos

## 2.1 Funciones de un analizador léxico.

Recordemos que un Analizador Léxico agrupa caracteres, tomados del programa fuente, en "palabras" o cadenas para identificar si se tratan de componentes léxicos del lenguaje a compilar; por cada componente léxico identificado, genera un ***token***.

De manera esquemática observemos qué elementos maneja el Analizador Léxico.




Componentes en el proceso de Análisis Léxico.

A manera de pasos secuenciales, presentaré cómo trabaja el Analizador Léxico:

1. Crea las tablas. Las de tipo catálogo, las llena; a las otras, sólo define su estructura.

2. Lee carácter por carácter del programa fuente hasta formar una cadena que sea identificada como un componente léxico, como un error o el carácter de fin de archivo.

3. Si se trata de un componente léxico, dependiendo de la clase a la que pertenezca, realiza lo siguiente:

   - Si es un identificador, lo busca en la tabla de símbolos. Si lo encuentra, entonces toma la posición del identificador en la tabla de símbolos, la cual va a ser el campo "valor" del token. Si el identificador no está en la tabla de símbolos, lo coloca ahí y guarda, en el campo "valor" del token, la posición donde fue colocado. Finalmente genera el token con los campos clase (identificador) y valor (posición en la tabla de símbolos), y se lo entrega al analizador sintáctico.

   - Si es una cadena o una constante, la busca en la tabla de literales. Y sigue el mismo procedimiento del identificador, sólo que utiliza la tabla de literales. El token que genera tendrá como clase: cadena o tipo de constante(entera, real, etc), y como valor, la posición dentro de la tabla de literales.

   - Si es palabra reservada, busca en qué posición, del catálogo correspondiente, se encuentra y la guarda en el campo "valor" del token. Crea el token correspondiente con clase (pal. reservada) y valor (posición en el catálogo) y se lo entrega al analizador sintáctico.

   - Para las otras clases de componentes léxicos, que están definidas en catálogos, se realiza el mismo procedimiento de las palabras reservadas. Crea el token con la clase correspondiente (operador relacional, lógico, aritmético, etc.) y valor (posición en el catálogo correspondiente) y se lo entrega al analizador sintáctico.

   - Se regresa al paso 2.

4. Si es una cadena o carácter no reconocido envía un mensaje de error y se regresa al paso 2.

5. Si es el carácter es el fin de archivo (EOF), termina el proceso de análisis léxico.

Este análisis se realiza a través de un Autómata Finito Determinístico, ya que reconoce cadenas que pertenecen a un lenguaje.

Entonces, de forma puntual, se puede decir que las funciones del Analizador Léxico son las siguientes:

- Crear las tablas (de símbolos, de literales y catálogos).
- Actualizar las tablas de símbolos y de literales conforme identifica dichos componentes léxicos.
- Crear los tokens. Formados por el par ordenado: campo y valor.
- Calcular el valor de las constantes numéricas.

- Ignorar los espacios en blanco, saltos de línea y tabuladores (no generan token)
- Deberá detectar el fin de archivo (EOF)
- Además, reconocer y descartar comentarios.

Consideraciones de diseño del Analizador Léxico:

- No debe interrumpir su análisis por datos o caracteres extraños que contenga el programa fuente que está analizando.
- Debe hacer su análisis con eficiencia, la cual incluye rapidez y uso adecuado de memoria principal.

Veamos con más detalle cómo es que el analizador léxico va identificando las clases de componentes léxicos.

Como había mencionado, las clases de componentes léxicos las definen los diseñadores de lenguajes. Es así que para crear el analizador léxico que reconozca dichos componentes léxicos, debemos conocer cuáles son las clases y cómo se definen.

Por ejemplo, en el lenguaje ANSI C, un identificador, ya sea un nombre de variable, de función o de constante simbólica, se define como:

“Una cadena compuesta por letras y dígitos; el primer carácter debe ser una letra. El “guion bajo” ( _ ) cuenta como una letra, sin embargo no se recomienda usar este carácter al inicio de la cadena, puesto que las rutinas de biblioteca con frecuencia lo utilizan para iniciar los nombres de variables y funciones. Las letras mayúsculas y minúsculas son distintas, de tal manera que **x** y **X** son dos identificadores diferentes. La práctica tradicional de C es usar letras minúsculas para nombres de variables o funciones, y todo en mayúsculas para constantes simbólicas. El tamaño mínimo de la cadena es 1”

Sabemos que una **expresión regular** es una notación que define un **lenguaje regular**, y debido a que los componentes léxicos pertenecen a un lenguaje regular, definiremos una expresión regular para cada clase de componente léxico.

**Ejercicio 2.2.1**

Elaborar la expresión regular que defina a los identificadores del lenguaje ANSI C, con las características antes descritas.

Usaremos la notación ampliada, más cercana a la empleada en lex/flex.

Basándonos en que una expresión regular está compuesta, a su vez, de expresiones regulares más simples, definiré primeramente, expresiones regulares más simples y les asignaré un nombre.

<dig> = [0-9] expresión regular que define a los dígitos; le asigno el nombre <dig>

<letra> = [A-Za-z] expresión regular que define a las letras mayúsculas y minúsculas; le asigno el nombre <letra>

Al nombre de la expresión regular lo encierro entre < >, para distinguirlo de una expresión regular. Ahora, usando esas expresiones simples, la expresión regular que define al lenguaje de los identificadores es:

<ident> = (<letra>|_ )(<letra>|_|<dig>)*

**Ejercicio 2.2.2**
Elaborar la expresión regular que defina al lenguaje de las constantes numéricas enteras en base 10 (decimal), del lenguaje ANSI C: "Una constante numérica entera en base 10 inicia con un dígito diferente a 0 y le siguen cero o más dígitos. Considere la constante 0 como decimal".

Respuesta:

<digNo0> = [1-9] expresión regular que define a los dígitos sin incluir al 0; le asigno el nombre <digNo0>

Entonces:
<cteEntDec> = <digNo0>(<digNo0>|0)*|0

**Construcción de un Analizador Léxico.**

Como ya había mencionado, el Analizador léxico es un Autómata finito determinístico (AFD) que reconoce las cadenas de un programa fuente, que pertenecen a una clase de componente léxico. Por lo que el procedimiento a seguir para elaborar un Analizador léxico es primeramente definir la expresión regular de cada clase de componente léxico y con base en ellas construir el AFD.

**Ejercicio 2.2.3**
Construir un analizador léxico, es decir, un autómata finito determinístico que reconozca tanto identificadores como constantes numéricas enteras decimales del Lenguaje C.

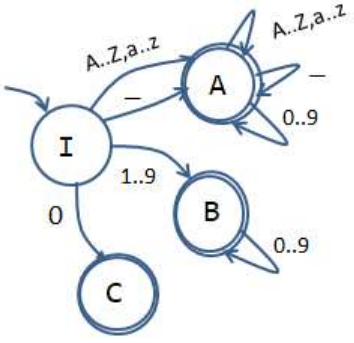
*Procedimiento.*

- Unimos las dos expresiones regulares antes obtenidas:

(<letra>|_ )(<letra>|_|<dig>)*|<digNo0>(<digNo0>|0)*|0

*Observa que sólo usamos paréntesis y no [ ] para definir precedencia de operadores.

- Ahora elaboramos el diagrama de transiciones del AFD




El estado I es el estado inicial
Los estados A, B y C son estados finales:

- El estado A reconoce a los identificadores.
- Los estados B y C reconocen a las constantes enteras decimales

*Observa que sí es determinístico

- Por último, elaboramos su tabla de transición.

| Estado | A..Z | a..z | _ | 0 | 1..9 | Reconoce |
|---|---|---|---|---|---|---|
| I | A | A | A | C | B | |
| A | A | A | A | A | A | identificador |
| B | | | | B | B | cte entera decimal |
| C | | | | | | cte entera decimal |

Observaciones importantes de la tabla:

- Las columnas deben ser únicas, es decir, un elemento que esté en una columna no debe estar en otra columna. Por eso en la tabla están las columnas 0 y 1..9, y no hay otra con 0..9. Entonces en el estado B, aunque en el diagrama está la transición 0..9, en la tabla se indica en las columnas 0 y 1..9. Lo mismo sucede con la transición 0..9 del estado A.
- También es muy importante que un estado final sólo deba reconocer una sola clase de componente léxico, para que no exista ambigüedad. En ocasiones, cuando se quiere minimizar el AFD, ocurre la ambigüedad; en estos casos no se debe obtener el AFD mínimo.

La tabla de símbolos es una estructura de datos que tiene un registro para cada identificador, con campos para sus atributos. La estructura de datos debe diseñarse de tal forma que permita al compilador buscar el registro para cada identificador y almacenar u obtener datos de ese registro con rapidez.

Generalmente se manejan los siguientes campos:

- Un campo para almacenar el nombre del identificador
- Campos para indicar atributos del identificador como su tipo (entero, real, entero largo, carácter, etc.), el alcance, valores estáticos, etc.

El tamaño del nombre del identificador varía de *1* a *n* caracteres, por lo que lo más conveniente es manejar el campo para su almacenamiento, como un apuntador a cadena. Debemos recordar que esta tabla debe ocupar eficientemente la memoria.

El otro campo (el de los atributos), podría ser una lista ligada que nos indique por cada atributo, qué tipo de atributo es y en su caso su valor.

**El analizador léxico deberá crear la estructura de datos para la tabla de símbolos y sólo irá actualizando el campo donde se almacenará el nombre del identificador.**

Los otros campos se irán actualizando en la etapa de análisis semántico.

En la etapa de análisis léxico la detección de errores puede darse cuando en la tabla de transición de su correspondiente AFD se llega a una celda vacía y no es un estado final.

Entonces, se reporta que toda la cadena, hasta el momento formada, no es un componente léxico válido. Por ejemplo, de la siguiente línea de código en C, ¿qué cadena no sería reconocida?

x_1= 0xG; 

| Cadena | Clase de componente léxico |
|---|---|
| x_1 | Identificador |
| = | Operador de asignación |
| 0xG | Error |
| ; | Símbolo especial |

Actividad 2.4.1 Del AFD que reconoce los operadores relacionales y de corrimiento elaborado como ejercicio en la clase pasada, ¿qué ocurriría si tratara de reconocer el operador **=!** ?

Se puede dar otra situación. Al estar analizando una cadena, en el AFD, se llega a una celda vacía pero en un estado final.

Entonces, pueden suceder dos cosas:

a) El último carácter leído es el inicio de otro componente léxico. Esto ocurre porque los componentes léxicos no están separados por espacios en blanco, tabuladores o salto de línea que pueden ser utilizados como delimitadores. Veamos un ejemplo en el que sí deben reconocerse como componentes léxicos válidos.

x_1=6.7+y_1;

Cuando se está haciendo la revisión de la primera cadena: se lee x, se lee _, se lee 1 y al leer el carácter =, en el AFD se llegaría a una celda vacía pero en un estado donde se reconoce a un identificador. Por lo que se reconoce al identificador x_1 y no avanza en la lectura del siguiente carácter de entrada, sino que se mantiene en el carácter = y se inicia con el reconocimiento, es decir, se va al estado inicial.

De manera análoga, se reconocerían los siguientes componentes léxicos como:

"=" operador de asignación, "6.7" constante real, "+" operador aritmético, "y_1" identificador y ";" símbolo especial.

b) El último carácter leído es un carácter que no pertenece al alfabeto. En este caso, no llega a ninguna celda del AFD por lo que si ya se tenía una cadena, se observa el estado en el que está; si es un estado final, la cadena sin el carácter último leído, se reconoce; si el estado no es final, entonces la cadena que se tenía junto con el último carácter leído no se

reconoce. Si no se había tenido una cadena, significa que es sólo un símbolo que no pertenece a ningún componente léxico del lenguaje.

**Ejercicio 2.4.1** Para las siguientes cadenas, indica cómo las reconocería el analizador léxico del lenguaje C.

a) 2.2.3
   - El analizador léxico podría reconocer dos constantes de punto flotante: 2.2 y .3
   - O bien, indicar que no es un componente léxico válido. En este segundo caso el AFD sería más inteligente. Es decir, requiere de realizar más operaciones que simplemente ver si la celda a la que llega con el segundo punto (.) está vacía.

b) inta,b;
   - El analizador léxico reconocería: "inta" como identificador, "," como símbolo especial, "b" como identificador, y ";" como símbolo especial. En este caso sí es indispensable poner un espacio entre "int" y "a" para que reconozca que "int" es palabra reservada y "a" es un identificador.

Si bien, un Analizador Léxico es en sí un Autómata Finito Determinístico (AFD), además de hacer el reconocimiento de cadenas como componentes léxicos, realiza otras funciones, entre ellas la generación de tokens y la creación de las diferentes tablas.

Vamos a realizar un ejercicio completo donde se reflejen más funciones que realiza el Analizador Léxico.

**Ejercicio 2.5.1** Sobre el siguiente código en C, realiza el análisis léxico, mostrado las tablas que se crean y actualizan, así como la generación de tokens.

```
void main( )
{
   int a,b=5;
   printf("Dame un valor entero:  ");
   scanf("%d",&a);
   if (a<b)
      printf("\nEl valor dado es menor a 5");
   else
      printf("\nEl valor dado es igual o mayor a 5");
}
```

Respuesta

a) Creamos los catálogos. Por practicidad, sólo crearemos los catálogos cuyos elementos se encuentran en el código.

   - Catálogo de clases de componentes léxicos (en orden de presentación)

| Clase | Descripción |
|---|---|
| 0 | palabras reservadas |
| 1 | identificadores |
| 2 | símbolos especiales |
| 3 | operadores de asignación |
| 4 | constantes enteras |
| 5 | constantes cadenas |
| 6 | operadores relacionales |

Podemos observar que de este catálogo de clases, identificamos que el analizador léxico requiere los catálogos de: palabras reservadas, símbolos especiales, operadores de asignación y operadores relacionales; las cuales las definiremos a continuación.

- Catálogo de palabras reservadas. El lenguaje C sólo cuenta con 32 palabras reservadas.

| Pos. | P. R. |
|---|---|
| 0 | auto |
| 1 | break |
| 2 | case |
| 3 | char |
| 4 | const |
| 5 | continue |
| 6 | default |
| 7 | do |

| Pos. | P.R. |
|---|---|
| 8 | double |
| 9 | else |
| 10 | enum |
| 11 | extern |
| 12 | float |
| 13 | for |
| 14 | goto |
| 15 | if |

| Pos. | P.R. |
|---|---|
| 16 | int |
| 17 | long |
| 18 | register |
| 19 | return |
| 20 | short |
| 21 | signed |
| 22 | sizeof |
| 23 | static |

| Pos. | P.R. |
|---|---|
| 24 | struct |
| 25 | switch |
| 26 | typedef |
| 27 | union |
| 28 | unsigned |
| 29 | void |
| 30 | volatile |
| 31 | while |

- Catálogo de operadores de asignación.

| Pos. | op. asig. |
|---|---|
| 0 | = |
| 1 | += |
| 2 | -= |
| 3 | *= |
| 4 | /= |
| 5 | %= |

| Pos. | op. asig. |
|---|---|
| 6 | &= |
| 7 | ^= |
| 8 | \|= |
| 9 | <<= |
| 10 | >>= |

- Catálogo de operadores relacionales.

| Pos. | op. relac. |
|---|---|
| 0 | > |
| 1 | < |
| 2 | >= |
| 3 | <= |
| 4 | == |
| 5 | != |

- Catálogo de símbolos especiales

| Pos. | simb. esp. |
|---|---|
| 0 | ( |
| 1 | ) |
| 2 | , |
| 3 | ; |
| 4 | { |
| 5 | } |
| 6 | & |

b) Ahora definimos la estructura de las tablas de símbolos y literales.

- La Tabla de símbolos la definiremos de tres campos:

| Pos. | Nombre | Tipo |
|---|---|---|
| | | |

Donde:

**Pos** → posición del identificador dentro de la tabla de símbolos

**Nombre** → nombre del identificador. Para el ejercicio, pondremos el nombre del identificador en este campo. Al implementarlo se deberá poner un apuntador a cadena.

**Tipo** → indicará el tipo de identificador. Lo actualizará el Analizador Semántico. El Analizador Léxico le puede dar un valor inicial de -1

- En la tabla de literales sólo introduciremos a las constantes cadena. Esta tabla sólo tendrá 2 campos :

| Pos. | Cadena |
|---|---|
| | |

El campo cadena, generalmente maneja un apuntador a una cadena. Para este ejercicio, pondremos la cadena.

c) Una vez definidas las tablas y catálogos, se iniciará con la revisión de los componentes léxicos del programa fuente.

- Por cada componente léxico reconocido, generará el *token* correspondiente. El *token* está compuesto por el par ordenado (*clase, valor*).
- En el campo *clase* se colocará el valor numérico de la clase a que corresponda del catálogo de clases.

- El campo *valor* será la posición donde se encuentre el componente léxico en su tabla o catálogo correspondiente. Para las constantes numéricas, el *valor* será el valor numérico correspondiente.

Realicemos el análisis léxico de la primera línea del código

**void main( )**

void → la reconoce como palabra reservada que es la clase 0, y está en la posición 29 en la tabla correspondiente. Por lo que su token es (0,29)

main → la reconoce como identificador que es la clase 1; como la Tabla de símbolos está vacía, lo coloca en la posición 0. Por lo que su token es (1,0)

| Pos. | Nombre | Tipo |
|---|---|---|
| 0 | main | -1 |

( → lo reconoce como símbolo especial que es la clase 2, y está en la posición 0 en la tabla correspondiente. Por lo que su token es (2,0)

) → lo reconoce como símbolo especial que es la clase 2, y está en la posición 1 en la tabla correspondiente. Por lo que su token es (2,1)

Continuemos con la revisión de las siguientes tres líneas:

```
{
   int  a,b=5;
   printf("Dame un valor entero:  ");
```

{ → lo reconoce como símbolo especial que es la clase 2, y está en la posición 4 en la tabla correspondiente. Por lo que su token es (2,4).

int → la reconoce como palabra reservada que es la clase 0, y está en la posición 16 en la tabla correspondiente. Por lo que su token es (0,16).

a → lo reconoce como identificador que es la clase 1; lo busca en la Tabla de símbolos y como no está, lo coloca en la posición 1. Por lo que su token es (1,1).

| Pos. | Nombre | Tipo |
|---|---|---|
| 0 | main | -1 |
| 1 | a | -1 |

, → lo reconoce como símbolo especial que es la clase 2, y está en la posición 2 en la tabla correspondiente. Por lo que su token es (2,2).

b → lo reconoce como identificador que es la clase 1; lo busca en la Tabla de símbolos y como no está, lo coloca en la posición 2. Por lo que su token es (1,2).

| Pos. | Nombre | Tipo |
|---|---|---|
| 0 | main | -1 |
| 1 | a | -1 |
| 2 | b | -1 |

= → lo reconoce como operador de asignación que es la clase 3, y está en la posición 0 en la tabla correspondiente. Por lo que su token es (3,0).

5 → la reconoce como constante entera que es la clase 4, se calcula su valor numérico para ponerlo en el campo *valor* del token. Por lo que su token es (4,5)

; → lo reconoce como símbolo especial que es la clase 2, y está en la posición 3 en la tabla correspondiente. Por lo que su token es (2,3)

printf → lo reconoce como identificador que es la clase 1; lo busca en la Tabla de símbolos y como no está, lo coloca en la posición 3. Por lo que su token es (1,3).

| Pos. | Nombre | Tipo |
|---|---|---|
| 0 | main | -1 |
| 1 | a | -1 |
| 2 | b | -1 |
| 3 | printf | -1 |

( → lo reconoce como símbolo especial que es la clase 2, y está en la posición 0 en la tabla correspondiente. Por lo que su token es (2,0)

“Dame un valor entero: ” → la reconoce como constante cadena que es la clase 5, como la tabla de literales está vacía, se sitúa en la posición 0. Por lo que su token es (5,0).

| Pos. | Cadena |
|---|---|
| 0 | “Dame un valor entero: ” |

* Para reconocer a una constante cadena es porque está encerrada entre comillas (“); por lo que las comillas no se manejan como símbolos especiales.
* Muchos diseñadores de compiladores almacenan a las constantes cadena con todo y comillas ya que sirven de delimitadores. Podemos observar que la cadena reconocida tiene un espacio al final.
* En la implementación, en el campo cadena se sitúa un apuntador a cadena, el cual apunta a una cadena bien delimitada (con el carácter ‘\0’ como indicador de fin de cadena).
* También se recomienda revisar si hay en la tabla de literales una cadena idéntica antes de almacenar la cadena reconocida, esto con la intención de ahorrar memoria. A mi parecer creo que la búsqueda de cadenas idénticas propicia uso de más tiempo de procesador. ¿Qué será mejor, ahorrar memoria o ahorrar velocidad?

) → lo reconoce como símbolo especial que es la clase 2, y está en la posición 1 en la tabla correspondiente. Por lo que su token es (2,1)

; → lo reconoce como símbolo especial que es la clase 2, y está en la posición 3 en la tabla correspondiente. Por lo que su token es (2,3)

Entonces, hasta este punto las tablas actualizadas y los tokens generados son:

Tabla de símbolos

| Pos. | Nombre | Tipo |
|---|---|---|
| 0 | main | -1 |
| 1 | a | -1 |
| 2 | b | -1 |
| 3 | printf | -1 |

Tabla de literales

| Pos. | Cadena |
|---|---|
| 0 | "Dame un valor entero: " |

Tokens

(0,29)
(1,0)
(2,0)
(2,1)
(2,4)
(0,16)
(1,1)
(2,2)
(1,2)
(3,0)
(4,5)

(2,3)
(1,3)
(2,0)
(5,0)
(2,1)
(2,3)

Actividad 2.5.1 Terminar de hacer el análisis léxico del programa del ejercicio 2.5.1.

```
void main( )
{
   int a,b=5;
   printf("Dame un valor entero:  ");
   scanf("%d",&a);
   if (a<b)
      printf("\nEl valor dado es menor a 5");
   else
      printf("\nEl valor dado es igual o mayor a 5");
}
```