# Procesadores de lenguaje

## Tema 2 – Análisis léxico

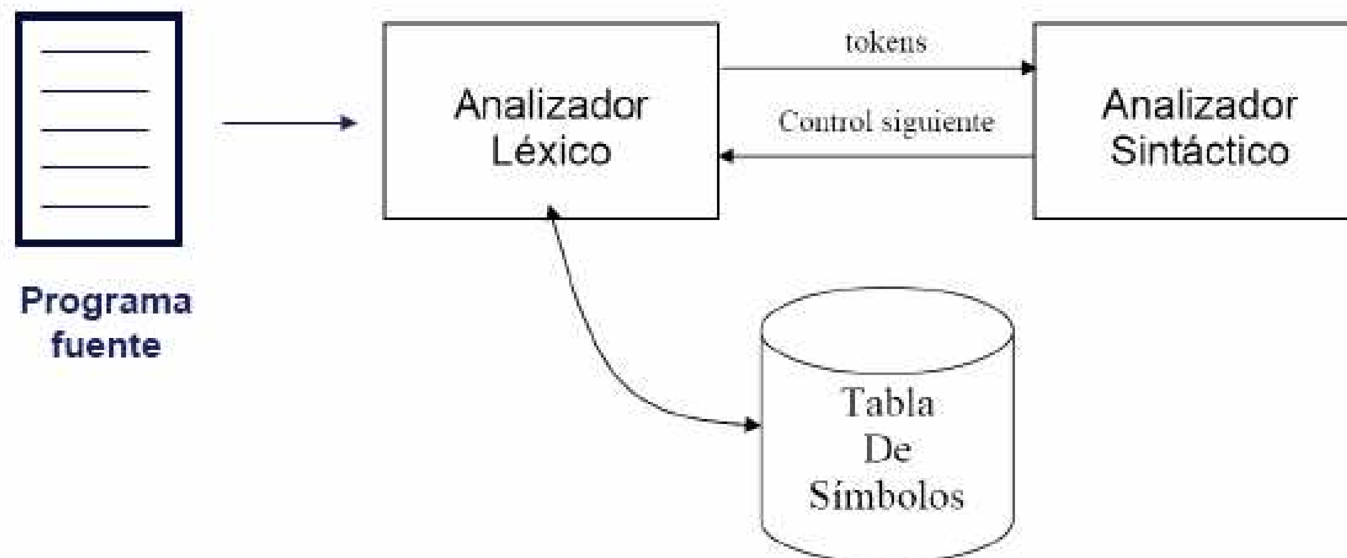
Ing. Edgard Pilco Apaza

# Resumen del tema

- Descripción de un analizador léxico
- Formalismos de especificación léxica:
  - Expresiones regulares
  - Autómatas finitos
- Errores léxicos
- Construcción de un analizador léxico
  - Generadores automáticos de analizadores léxicos

# → Descripción de un analizador léxico

- Un analizador léxico lee carácter a carácter del programa fuente y genera una secuencia de componentes léxicos (*tokens*) que corresponden a unos patrones a los que asocia, si es necesario, unos atributos.



# Necesidad de un analizador léxico

- Al comparar las expresiones

```
aØ:=ØbØ*Ø7Ø+Ø4Ø;
aØØØ:=ØØb*7+4ØØØ;
```

  - La estructura de las dos expresiones es equivalente,
  - La posición de los caracteres que las componen, aunque siguen el mismo orden, son diferentes.

- Si las distintas fases del compilador tuvieran que trabajar con los caracteres directamente sería más complicado descubrir la estructura de un programa.

# Independencia de léxico y sintáctico

- El analizador léxico suele convertirse en una **subrutina** del analizador sintáctico.

- Varias razones para su independencia:
  - Se simplifica el diseño del analizador sintáctico.
  - Se consigue un compilador más eficiente
    - Un sistema de entrada optimizado aumenta la velocidad de lectura.
  - Añade portabilidad al compilador: independencia del alfabeto.

- Se comunican mediante:
  - Buffer intermedio.
  - Llamada a subrutina.

# → Conceptos básicos

- **Token (o componente léxico)**: Secuencia de caracteres con significado sintáctico propio.
- **Lexema**: Secuencia de caracteres cuya estructura se corresponde con el patrón de un token.
- **Patrón**: Regla que describe los lexemas correspondientes a un token.

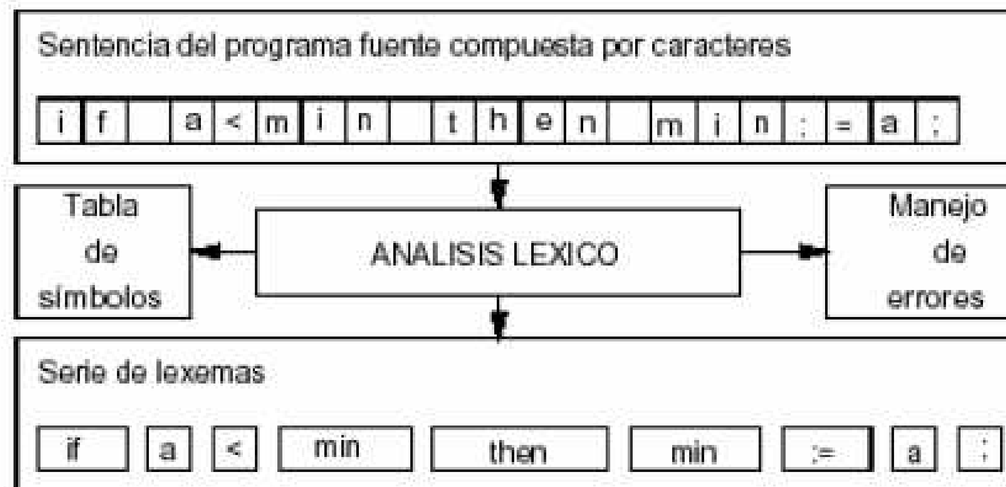
| COMPONENTE LEXICO | LEXEMAS EJEMPLO | DESCRIPCION DEL PATRON | OBSERVACIONES |
|---|---|---|---|
| Identificador | X, Y valor, X2 | <letra>(<letra> \| <dígito>)* | Identificacores |
| CADENA | "Una cadena" | caracteres entre " y " | Constante de Cadena |

# → Componentes léxicos

- En la mayoría de los lenguajes de programación, se consideran componentes léxicos (tokens):
  - palabras reservadas
  - operadores (de comparación, asignación, lógicos, aritméticos …)
  - identificadores
  - constantes
  - signos de puntuación (paréntesis, punto y coma …)
  - marcas de comienzo y fin de bloque
- Los delimitadores no serán considerados, en general, tokens.
- Cuando un patrón puede concordar con más de un lexema es necesario conocer información adicional:
  - Esta información se almacena como atributos del token.

# Descripción de un analizador léxico

- El análisis léxico es un análisis de los caracteres
  - Parte de éstos y por medio de patrones reconoce los lexemas
  - Envía al analizador sintáctico el componente léxico y sus atributos
  - Puede hacer tareas adicionales: eliminar blancos, control líneas …



# El analizador léxico como interfaz

- El analizador léxico y el sintáctico forman un par *productor-consumidor*.
- En algunas situaciones, el analizador léxico tiene que leer algunos caracteres por adelantado para decidir de qué *token* se trata.

# Ejemplo

- Para la sentencia en Pascal:

```
IF x < 10 THEN x := x + y
```

- El analizador lexicográfico daría como resultado inicialmente la siguiente cadena de caracteres:

```
<79> <12> <28> <13> <80> <12> <65> <12> <34> <12>
```

- O lo que es lo mismo:

```
if identificador op_menor literal_entero then identificador
     asignación identificador op_suma identificador
```

# → Implementación

```
int explorador (void)
{   c = nuevocaracter (); /*función del sistema de entrada */
    switch (c)
    {   case ' ':
        case '\t':
        case '\n': break; /*no hacer nada para los separadores */
        case '+': return (OPERADOR_SUMA);
        case '-':
        case '*': return (OPERADOR_MULTIPLICAR);
        case '/':
        /*.... aqui vendrian el resto de los operadores y
         simbolos de puntuación por orden de precedencia */
        default:     if (esnumero(c))
                    {/*se leen caracteres mientras sean
                        numeros y se retorna al flujo de entrada el último
                        caracter leido, se guarda el lexema leido
                        y se retorna con el testigo NÚMERO_ENTERO */
                        return(NÚMERO_ENTERO);
                    }
                    else if (esletra(c))
                    {/*se leen caracteres mientras sean letras o números
                        y se retorna al flujo de entrada el último carácter
                        leido, se guarda el lexema leido y se comprueba si
                        es una palabra clave. Se retorna IDENTIFICADOR o
                        la palabra clave*/
                         return(testigo);
                    }
    } /* del switch */
} /* del explorador */
```

# Ejemplo

- El resultado final sería:

```
<79,-> <12,32> <28,-> <13,10> <80,-> <12,32>
<65,-> <12,32> <34,-> <12,33>
```

- Es decir:

```
<if,-> <identificador,32> <op_menor,->
<literal_entero,10> <then,-> <identificador,32>
<asignación,-> <identificador,32> <op_suma,->
<identificador,33>
```

# Atributos

- Los identificadores tienen asociados como atributos el lugar de la tabla de símbolos donde se han guardado:

```
< identificador, 32 >
```

- A veces el analizador sintáctico es el único que maneja la tabla de símbolos, en ese caso llevan asociado el propio lexema:

```
< identificador, x >
```

- Los literales tiene asociados como atributos el propio lexema:

```
< literal_entero, 10 >
```

# Errores léxicos

- El analizador léxico rechaza texto con caracteres ilegales (no recogidos en el alfabeto) o combinaciones ilegales.

- Ejemplos para un analizador léxico de C++:
  - "ñ", "é" (caracteres que no pertenecen al alfabeto del lenguaje)
  - ":=", ":::" (no coinciden con ningún patrón de los *tokens* posibles)

- Posibles acciones del analizador léxico para detectar errores, recuperarse y seguir trabajando:
  - *Modo de pánico*: Ignorar los caracteres no válidos hasta un carácter en el cual se considera que podría empezar un lexema correcto.

# Errores léxicos

- Algunas rutinas de recuperación "inteligente" llevan a cabo correcciones como las siguientes:
  - Borrar los caracteres extraños.
  - Insertar un carácter que pudiera faltar.
  - Reemplazar un carácter presuntamente incorrecto por uno correcto.
  - Conmutar las posiciones de dos caracteres adyacentes.

# → Especificación de componentes léxicos

- **Alfabeto:** Conjunto finito de símbolos.

$A = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$

- **Cadena** sobre un alfabeto**:** secuencia finita de símbolos de ese alfabeto.

$s = 543$ (También llamado Cuerda)

  - Cadena vacía: ε
  - Operaciones con cadenas: concatenación y exponenciación

- **Lenguaje:** conjunto de cadenas sobre un alfabeto.
  - Operaciones con lenguajes: Unión, Concatenación, Cerradura de Kleene (Cierre *) y Cerradura positiva (Cierre +)

$L = \{ 543, 547, 548, 542 \}$

# → Expresiones regulares

- Notación que facilita la especificación de lenguajes.
- ε: cadena vacía
- a: cadena "a" si "a" Є alfabeto.
- Si *r* y *s* son cadenas que pertenecen respectivamente a los lenguajes L(r) y L(s), son expresiones regulares:
  - r|s ≡ L(r) U L(s)
  - rs ≡ L(r)L(s)
  - r* ≡ (L(r))*
- Abreviaturas:
  - 0 ó más → a*
  - 1 ó más → aa* ≡ a+
  - 0 ó 1 → ε | a ≡ a?
  - clases de caracteres → a | b | c ≡ [abc] . También: a | b | c | ... | z ≡ [a-z]
  - Los paréntesis agrupan subexpresiones: (a|b|c)*

# Expresiones regulares

**Alternación:** dadas dos expresiones regulares *M* y *N*, el operador de alternación denotado por | crea una nueva expresión regular *M | N*. Una cuerda está en el lenguaje de *M | N* si está en el lenguaje de *M* o está en el lenguaje de *N*.

a | b "a" o "b"

**Concatenación:** dadas dos expresiones regulares *M* y *N*, el operador de concatenación denotado por 🕐 crea una nueva expresión regular *M* 🕐 *N*. Una cadena está en el lenguaje de *M* 🕐 *N* si es la concatenación de cualesquiera dos cadenas ♋ y ♌ tal que ♋ está en el lenguaje de *M* y ♌ está en el lenguaje de *N*.

a 🕐 b "ab"

# Expresiones regulares

**Epsilon:** La expresión regular *epsilon* representa un lenguaje cuya única cuerda es la cuerda nula.

| | |
|---|---|
| ε | "" |

**Repetición:** Dada una expresión regular *M*, su cerradura de Kleene es *M**. Una cuerda está en *M** si es la concatenación de cero o más cuerdas, todas pertenecientes a *M*.

| | |
|---|---|
| a* | { "", "a", "aa", "aaa", ... } |

# → Expresiones regulares

A las cadenas les llamamos *strings*.

Omitimos el símbolo de concatenación.

a 🕐 b 🕐 c     abc

Omitimos el símbolo de *epsilon*.

( a | ε )     ( a | )

Asumimos que la cerradura de Kleene tiene precedencia sobre la concatenación.

ab*     a 🕐 (b*)

Asumimos que la concatenación tiene precedencia sobre la alternación.

ab | c     (a 🕐 b) | c

# Expresiones regulares

[abcd] ( a | b | c | d)

[b -g] [bcdefg]

[b -gM -Qkr] [bcdefgMNOPQkr]

*M*? ( *M* | ε )

*M*+ *M* 🕐 *M**

# En resumen

| a | Un caracter ordinario. |
|---|---|
| ♏ o ε | El string vacío. |
| | Otra forma de escribir el string vacío. |
| M | N | Alternación, elegir M o N. |
| M 🕐 N | Concatenación, M seguida de N |
| MN | Otra forma de escribir concatenación. |
| M* | Repetición (cero o más veces). |
| M+ | Repetición (una o más veces) |
| M? | Opcional, cero o una ocurrencia de M. |
| [a -zA -Z] | Alternación de un conjunto de caracteres. |
| "a.+*" | Strings entre comillas no son interpretados. |

# Ejemplos

| | |
|---|---|
| ( 0 \| 1 )* 0 | Números binarios múltiplos de 2. |
| b*abb*a? | Strings de a's y b's sin a's consecutivas. |
| [ab]*aa[ab]* | Strings de a's y b's que contienen a's consecutivas. |
| (0\|1\|2\|3\|4\|5\|6\|7\|8\|9) | Un *dígito*. |
| *dígito dígito** | Un entero positivo (*posint*). |
| -?*posint* | Un entero (*int*). |
| *int* (. *posint*)? | Un real |

# Ejemplos

```
0(0|1)0*: 010, 000, 010000, 01, etc.

a(ab)+b*: aab, aababb, aabbbbb, aabababab bb, etc.

[1-9]?0 : 0, 10, 20, 30,..., 90

[a-zA-Z]

[0-9]+

[a-zA-Z]([a-zA-Z]|[0-9])*

[0-9]+(.[0-9]+)?
```

# → Definiciones regulares

- Nombre que se da a una expresión regular por conveniencia, definiéndola como si fuera un símbolo:

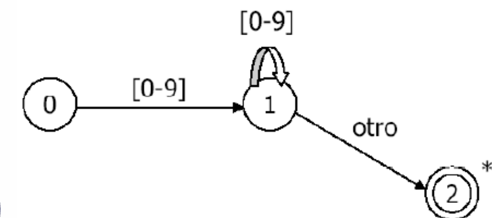
```
Letra → [a-zA-Z]

Dígito → 0|1|2|3|4|5|6|7|8|9

Identificador → Letra(Letra|Digito)*

NúmeroReal → Digito+(.Digito+)?
```

# Definiciones regulares

```
Dígito → [0-9]

Dígitos → Digito+

Fracción → (.Dígitos)?

Exponente → (E(+|-)?Dígitos)?

NúmeroReal → Dígitos Fracción Exponente
```

# Diagramas de transiciones

- Compuestos por estados y transiciones entre éstos
  - Círculos: estados
  - Arcos etiquetados: transiciones
- Sirven para representar lo que sucede a medida que se toman caracteres de la entrada
  - Cada círculo = un estado
  - Cada arista = un carácter en la entrada
  - Doble círculo = aceptación de un elemento léxico
  - Asterisco = el último carácter se debe devolver a la entrada (retroceso)
  - Otro = cualquier carácter no asignado a otra flecha



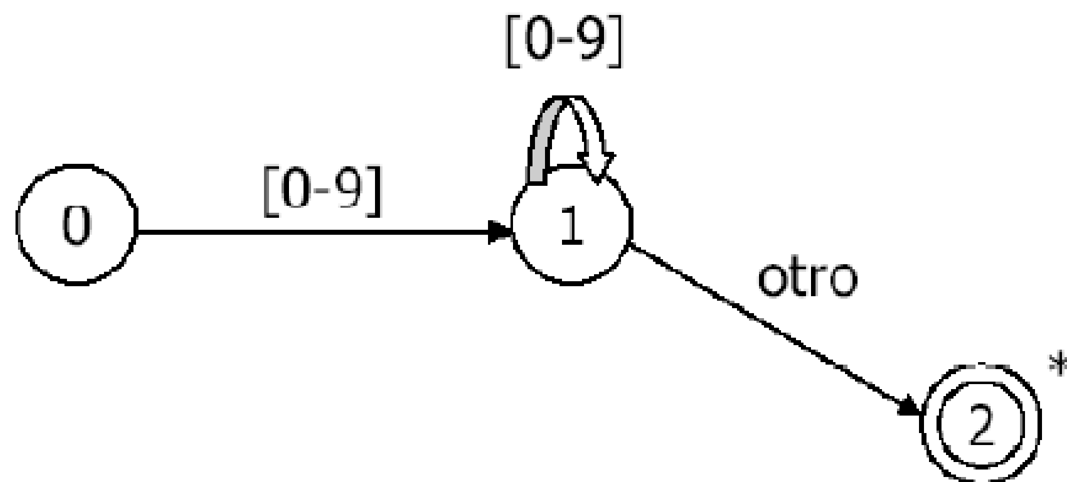

- No tiene estados de error (sin transición = error)

# Autómatas finitos deterministas (AFD)

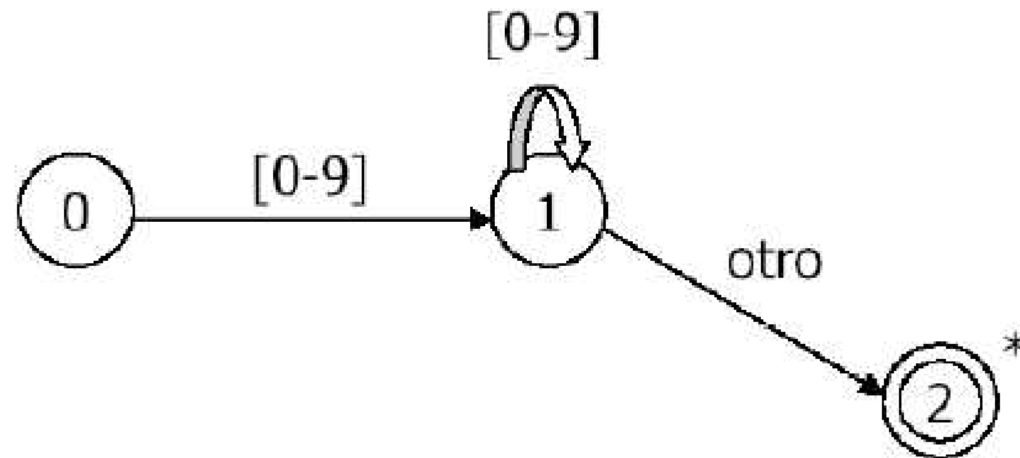
- Son diagramas de transiciones generalizados.
- Un reconocedor de un lenguaje es un programa que dice si una cadena pertenece o no a un lenguaje.
- Toda expresión regular puede reconocerse mediante un autómata finito determinista (AFD)
- Se emplean autómatas por la sencillez de programar código que simule su funcionamiento.

# Diagramas de transiciones

**NúmeroEntero → Digito⁺**



# → Diagramas de transiciones

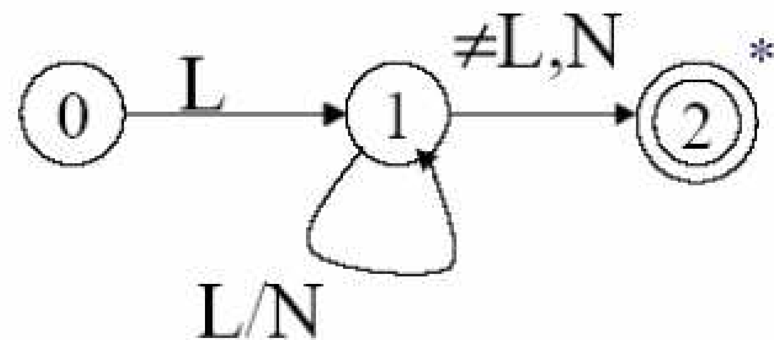



| Estado | 0-9 | otro | Token | Retroceso |
|---|---|---|---|---|
| 0 | 1 | - | - | - |
| 1 | 1 | **Error** | - | - |
| 2 | - | - | Num_entero | 1 |

# → Diagramas de transiciones

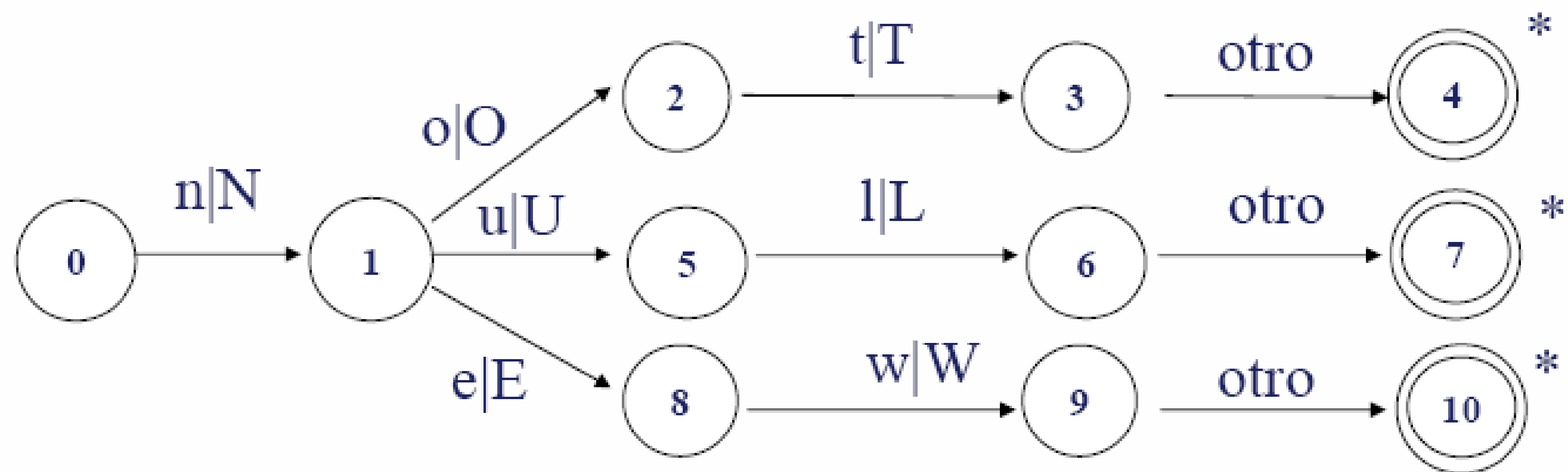
**Identificador → Letra(Letra|Digito)***

L → [a-zA-Z]
N → [0-9]
Id → L (L | N)*



# Diagramas de transiciones

## Not-Nul-New (Pascal)




3 – Reconocer "not"
6 – Reconocer "nul"
9 – Reconocer "new"

# Tratamiento de palabras reservadas

- Son aquellas que los lenguajes de programación "reservan" para usos particulares.

- ¿Cómo diferenciarlas de los identificadores?

  - Resolución **implícita**: reconocerlas todas como identificadores, utilizando una tabla adicional con las palabras reservadas que se consulta para ver si el lexema reconocido es un identificador o una palabra reservada.

  - Resolución **explícita**: se indican todas las expresiones regulares de todas las palabras reservadas y se integran los diagramas de transiciones resultantes de sus especificaciones léxicas en la máquina reconocedora.

# Prioridad de los tokens

- Se da prioridad al *token* con el lexema más largo:
  - Si se lee ">=" y ">" se reconoce el primero.
- Si el mismo lexema se puede asociar a dos *tokens*, estos patrones estarán definidos en un orden determinado.
- Ejemplo:
  - *while* → palabra reservada "while"
  - *letra* (*letra* | *digito*)* → identificador
  - Si en la entrada aparece "while", se elegirá la palabra reservada por estar primero.
  - Si estas especificaciones iniciales aparecieran en orden inverso, se reconocería un *token* identificador

# Construcción de un analizador léxico

- Los analizadores léxicos pueden construirse:
  - *Usando generadores de analizadores léxicos:* Es la forma más sencilla pero el código generado por el analizador léxico es más difícil de mantener y puede resultar menos eficiente.
  - *Escribiendo el analizador léxico en un lenguaje de alto nivel:* permite obtener analizadores léxicos con más esfuerzo que con el método anterior pero más eficientes y sencillos de mantener.
  - *Escribiendo el analizador léxico en un lenguaje ensamblador:* Sólo se utiliza en casos específicos debido a su alto coste y baja portabilidad.

# Generador automático (LEX)

- Recibe la especificación de las expresiones regulares de los patrones que representan a los *tokens* del lenguaje y las acciones a tomar cuando los detecte.
- Genera los diagramas de transición de estados en código C , C++ o Java generalmente.
- Ventaja: Comodidad de desarrollo.
- Desventajas:
  - El mantenimiento del código generado resulta complicado.
  - La eficiencia del código generado depende del generador.

# → Generador automático (LEX)

- Parte de un conjunto de reglas léxicas (expresiones regulares) y produce un programa (*yylex)* que reconoce las cadenas que cumplen dichas reglas
  - *yylex* es la implementación del AFD
- A cada regla se asocian un conjunto de acciones.
  - Cuando *yylex* encuentra una cadena que cumple un regla ejecuta las acciones asociadas a esa regla.
- Ejemplo: definiciones de números

```
DIGITO [0-9]
%%
{DIGITO}+                 proc_entero();
{DIGITO}+\.{DIGITO}*      proc_real();
```

# → Implementación del analizador léxico

- Consiste en implementar el diagrama de transiciones mediante la construcción de su tabla de transiciones.
  - filas: estados del diagrama de transiciones
  - columnas: posibles entradas
- Se puede también implementar directamente, utilizando estructuras de selección múltiple (*switch* en C) para leer caracteres hasta completar el *token*.
- Modelo mixto:
  - selección múltiple para los elementos léxicos de estructura más sencilla
  - diagrama de transiciones para el resto (cadenas no específicas, prefijos comunes, etc.)

# El proceso de implementación

- Definir las expresiones regulares del analizador léxico
- Identificar los tokens (códigos y atributos)
- Construir los diagramas de transiciones (AFD)
- Completar los autómatas con acciones semánticas
- Definir todos los posibles errores
- Implementar el AFD y las acciones semánticas usando switch o tablas de transiciones