Procesadores de Lenguajes

Tema 2

Análisis léxico

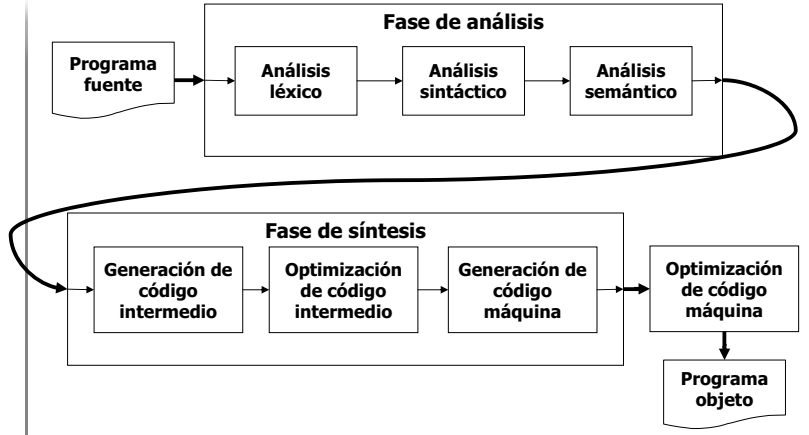
Tema 2 – Análisis Léxico

Sesión 1: Conceptos básicos del análisis léxico

Sesión 1: Conceptos básicos del análisis léxico

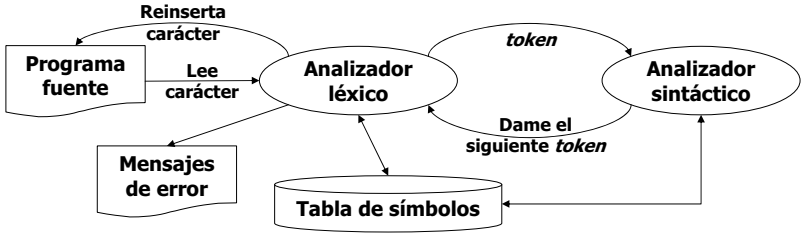
- El análisis léxico en contexto
- Tareas del análisis léxico
- ¿Por qué separar análisis léxico y sintáctico?
- Análisis léxico: definiciones
- Tipos de *tokens*:
  - Invariables
  - Construidos por el usuario
- Ejemplo: *tokens* de Pascal
- Especificación de *tokens*
  - En el analizador sintáctico
  - En el analizador léxico
    - El campo **tipo**
    - El campo **atributo**
- Un ejemplo simple de análisis léxico

El análisis léxico en contexto (I)



El análisis léxico en contexto (II)

- Primera fase de un compilador.
- Contexto de operación:



Tareas del análisis léxico

- Tareas principales:
  - Apertura y cierre del fichero de entrada (código fuente).
  - Lectura de los caracteres del fichero de entrada.
  - Generación de la secuencia de componentes léxicos (*tokens*) detectados en la entrada.
- Tareas secundarias:
  - Eliminación de los comentarios del programa fuente, así como de los blancos, saltos de página, tabuladores, retornos de carro, y demás caracteres propios del dispositivo de entrada.
  - Reconocimiento de los distintos identificadores y asignación de una posición en la tabla de símbolos.
  - Preproceso de las macros del programa fuente, si éste las contiene.
  - Detección e información de los errores lexicográficos del código fuente.
  - Relación de los mensajes de error que produce el compilador en sus diversas fases con la parte correspondiente del programa fuente (el número de la línea en la que aparecen).

¿Por qué separar análisis léxico y sintáctico?

- Se simplifica el diseño del analizador sintáctico:
  - No ha de preocuparse de leer el fichero de entrada, ni procesar blancos, ni comentarios, ni de recibir caracteres inesperados.
- El diseño general del compilador se hace más claro y comprensible.
- Se mejora la eficiencia del compilador en su conjunto:
  - La lectura del programa fuente suele requerir gran parte del tiempo de compilación, que se ve reducido si el analizador lexicográfico incorpora técnicas especiales de lectura, o está realizado en ensamblador.
- Aumenta la portabilidad del compilador:
  - Abstrae al resto del compilador de todas las diferencias que se produzcan en el alfabeto de entrada, o en el dispositivo de almacenamiento. Ejemplos:
  - Migrar programas fuente en ASCII al estándar EBCDIC.
  - Compilar ficheros TXT, HTML, Excel, Word, PDF, etc., con independencia del formato de entrada.

Análisis léxico: definiciones

- **Componente léxico** (***token***):
  - Un par ordenado cuyo primer elemento representa el **tipo** de componente léxico detectado en la entrada y cuyo segundo elemento (el **atributo** del *token*) da información adicional del mismo (subclasificándolo, dando un valor numérico, un puntero a una entrada de la tabla de símbolos, etc.).
- **Patrón** asociado a un (tipo de) *token* :
  - Descripción, mediante una regla o una expresión regular, de las cadenas de caracteres del lenguaje de entrada para las cuales se genera un mismo (tipo de) *token*.
- **Lexema**:
  - Aquella secuencia de caracteres del programa fuente con la que concuerda (*matches*) el patrón de un *token*.

Tokens invariables

- Operadores:
  - Símbolos que representan operaciones entre variables y constantes.
- Signos de puntuación:
  - Como en el lenguaje natural, son *tokens* que delimitan otros *tokens*, así como las sentencias del lenguaje (la coma, el punto, el punto y coma, los dos puntos, los paréntesis, etc.)
- Palabras claves:
  - Son cadenas indivisibles de caracteres, con una semántica predefinida en el lenguaje.
  - Palabras reservadas:
    - Aquellas que no pueden usarse para ningún otro propósito que aquél para el que se han definido en el lenguaje (salvo en comentarios).
  - Normalmente, las palabras clave de un lenguaje son también palabras reservadas en ese lenguaje.

Tokens construidos por el usuario

- Identificadores:
  - Nombres simbólicos que se darán a ciertos elementos de programación (por ejemplo: nombres de constantes, de tipos, de variables, de subprogramas, etc.)
  - Deben ser convenientemente declarados, para ser reconocidos por el compilador.
- Constantes:
  - Datos que no cambiarán su valor a lo largo del programa.
  - Tipos: enteras, reales, carácter, cadenas de caracteres, etc.

Ejemplo: *tokens* invariables en Pascal (I)

- Operadores:
  - <operador> ::= <op_matemático> | <op_comparación> | <op_lógico> | := | ↑
  - <op_matemático> ::= + | - | * | / | div | mod | shr | shl | in
  - <op_comparación> ::= < | > | <= | >= | = | <>
  - <op_lógico> ::= not | and | or

    (* independientemente de mayúsculas/minúsculas *)

- Signos de puntuación:
  - <puntuación> ::= . | , | ; | : | ^ | ' | " | (|) | [|] | { | }

Ejemplo: *tokens* invariables en Pascal (II)

- Palabras clave (o reservadas) [Grogono, 1996]

AND	ARRAY	BEGIN	CASE	CONST
DIV	DO	DOWNTO	ELSE	END
FILE	FOR	FUNCTION	GOTO	IF
IN	LABEL	MOD	NIL	NOT
OF	OR	PACKED	PROCEDURE	PROGRAM
RECORD	REPEAT	SET	THEN	TO
TYPE	UNTIL	VAR	WHILE	WITH

Especificación de *tokens* (I)

- Los componentes léxicos (tokens) son tratados como:
  - Símbolos no terminales de la gramática asociada al analizador léxico.
  - Símbolos terminales de la gramática asociada al analizador sintáctico.
    - Generalmente, se notarán en negrita en este caso.
- El **tipo** de *token* se pasa codificado al parser (mediante un tipo enumerado, por ejemplo)

Especificación de *tokens* (II)

- El campo **atributo** del *token* es:
  - Para identificadores:
    - Un puntero a la entrada de la tabla de símbolos donde se almacena la información del lexema (entre otras cosas).
  - Para palabras clave o reservadas:
    - El literal que la constituye (poco eficiente).
    - El valor de un tipo enumerado que la identifique.
    - El índice que la caracteriza dentro de una tabla específica para palabras reservadas dentro de la tabla de símbolos.

Especificación de *tokens* (III)

- El campo **atributo** del *token* es (cont.):
  - Para constantes:
    - Si es numérica:
      - El resultado obtenido al convertir al cadena de caracteres en su valor numérico correspondiente.
    - Si es un carácter, el propio carácter.
    - Si es una cadena de caracteres:
      - El literal que la constituye (poco eficiente)
      - El índice que la caracteriza dentro de una tabla específica para constantes de este tipo dentro de la tabla de símbolos.
  - Para operadores y signos de puntuación:
    - Si se da toda la información necesaria en el campo tipo, el atributo lleva un valor vacío.
    - En caso contrario:
      - El valor de un tipo enumerado que lo identifique.
      - El índice que lo caracteriza dentro de una tabla específica para operadores (o signos de puntuación) dentro de la tabla de símbolos.

Un ejemplo simple de análisis léxico

- Para la sentencia:

  CONST PI = 3.1416 (* La constante PI *);
- Se generaría la secuencia de *tokens*:

LEXEMA	Descripción (informal) del PATRÓN	TOKEN
CONST	**El literal "CONST"**	**(PAL_RES, 9)**
PI	**<ID> → <letra> | <letra> <R>** **<R> → (<letra> | <dígito>)***	**(ID, ↑T.S. <2>)**
=	**El carácter (aislado) '='**	**(OP_ASIG, *CONST*)**
3.1416	**Cualquier secuencia numérica con formato de coma fija o flotante**	**(REAL, 3'1416)**
;	**El carácter ';'**	**(PUNT, ';')**

Interfaces del analizador léxico con otros módulos (1)

- Con el analizador sintáctico:
  - TipoToken Scan();
    - TipoToken ≡ (TipoCodToken × TipoAtributo)
      - TipoCodToken ≡ (ID, PAL_RES, OP, PUNT, …)
      - TipoAtributo ≡ PUNTERO | ENTERO | REAL | ∅ | …
- Con la tabla de símbolos (TS):
  - ↑ $ENTRADA_{TS}$ Inserta (STRING, TipoCodToken);
  - ↑ $ENTRADA_{TS}$ Busca (STRING);

Interfaces del analizador léxico con otros módulos (2)

- Con el gestor de errores:
  - Si se usa una variable global para gestionar el número de línea:
    - [VOID] Error (INTEGER);
      - Todos los errores se guardan en una tabla global y se pasa la clave que lo identifica.
    - [VOID] Error (STRING);
      - Se pasa la cadena de caracteres que conforma el mensaje informativo.
  - En otro caso:
    - [VOID] Error (STRING, INTEGER);
      - Se pasa la cadena de caracteres que conforma el mensaje informativo, así como el número de línea actual
  - ¿Cómo se comunicaría el número de línea al resto de módulos?

Tema 2 – Análisis Léxico

Sesión 2: Construcción de *ATNs* para el análisis léxico

Sesión 2: Construcción de *ATNs* para el análisis léxico

- Separación analizador léxico ↔ sintáctico
- Ejemplo de construcción de un *ATN*
  - Identificación de los *tokens* que el escáner va a entregar al *parser*
  - Agrupación, según su tipo, de los *tokens* identificados
  - Selección de las reglas de análisis léxico
  - Especificación formal de los *tokens*
  - Construcción del autómata finito
  - Codificación y adición de acciones semánticas
    - El operador de preanálisis (*lookahead*)

Separación analizador léxico ↔ sintáctico

- En principio, en función del diseñador: ha de simplificarse el análisis y desarrollo del procesador del lenguaje al completo
- Analizador sintáctico:
  - Reglas propias de una gramática independiente del contexto, incluyendo:
    - Reglas no lineales
    - Reglas de patrones para estructuras con emparejamiento: BEGIN … END, (…), etc.
- Analizador léxico:
  - Reglas de una gramática regular:
    - Reglas lineales y/o que definan patrones de *tokens*.

Ejemplo de construcción de un *ATN* (1)

- Construcción del analizador léxico para el lenguaje generado por la gramática:
  - sent → **for** cond asign R_1
  - asign → id **:=** num **+** num **;** R_2
  - cond → id **:=** num **to** num **by** num R_3
  - num → d d* [**.** d d*] R_4
  - id → l (l | d)* R_5
  - l → **a** | … | **z** | **A** | … | **Z** R_6
  - d → **0** | … | **9** R_7

Ejemplo de construcción de un *ATN* (2)

- PASO 1: Identificación de los *tokens* que el escáner va a entregar al *parser*:
  - R_1: PAL_CLAVE_FOR
  - R_2: ID, OP_ASIG, NUM, OP_SUMA, SEP_PUNTO_COMA
  - R_3: ID, OP_ASIG, NUM, PAL_CLAVE_TO , PAL_CLAVE_BY
  - R_4: NUM_REAL, NUM_ENTERO
  - R_5: ∅
  - R_6: ∅
  - R_7: ∅
- PASO 2: Agrupación, según su tipo, de los *tokens* identificados:
  - PALABRAS CLAVES: FOR, TO, BY.
  - IDENTIFICADOR (ID).
  - OPERADORES: ASIGNACIÓN (OP_ASIG), SUMA (OP_SUMA).
  - NUMÉRICOS: NÚMERO (NUM) o REAL y ENTERO.
  - SEPARADORES: PUNTO_COMA

Ejemplo de construcción de un *ATN* (2)

- PASO 3: Selección de las reglas de análisis léxico (las que se van a codificar en el *ATN*):
  - R_1 , R_2 , y R_3 describen secuencias válidas de *tokens*: se codifican en el analizador sintáctico.
  - R_6 y R_7 no se codifican explícitamente en el *ATN*: se admiten **pequeños** abusos de notación
  - Las reglas que se codifican de forma explícita en el *ATN* son R_4 y R_5

Ejemplo de construcción de un *ATN* (3)

- PASO 4: Especificación formal de los *tokens* generados por el *ATN*:
  - Opción 1 ⇒ {ID, NUM, PAL_RES (FOR | TO | BY), OP_ASIG, SUMA, PUNT_COMA}
  - Opción 2 ⇒ {ID, ENTERO, REAL, PAL_RES (FOR | TO | BY), OP (ASIG | SUMA), PUNT_COMA}
  - Opción 3 ⇒ {ID, ENTERO, REAL, FOR, TO, BY, OP (ASIG | SUMA), PUNT_COMA}
  - Opción 4 ⇒ {ID, ENTERO, REAL, PAL_RES (FOR | TO | BY), OP_ASIG, SUMA, PUNT_COMA}

Ejemplo de construcción de un *ATN* (3)

- PASO 5: Construcción del autómata finito del *ATN*:
  - PASO 5.1: Definición del conjunto de delimitadores de palabras reservadas, números e identificadores (DELIM):
    - DELIM = { ' ' (blanco), EOL, EOF, TAB, '**:**', '**+**', '**;**' }
  - PASO 5.2: Identificación de los estados y las transiciones
- PASO 6: Codificación y adición de las acciones semánticas

Diagrama de transiciones del autómata finito (1)

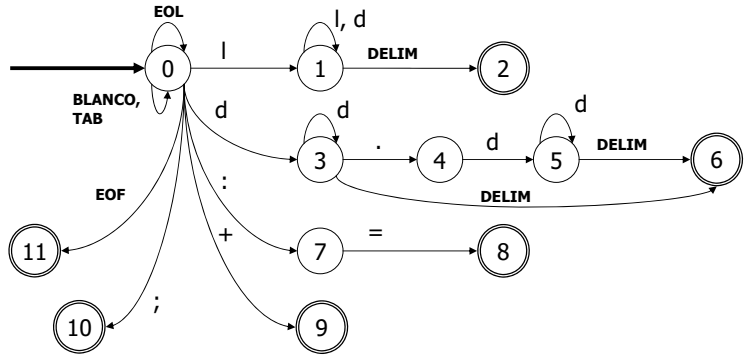
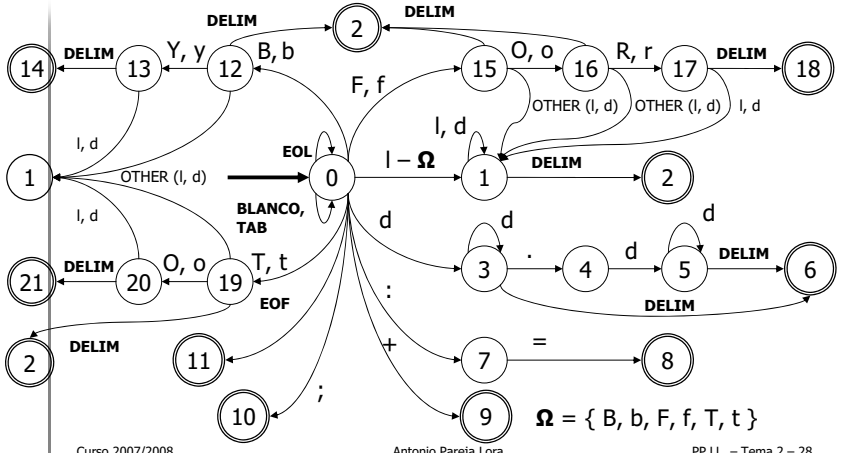
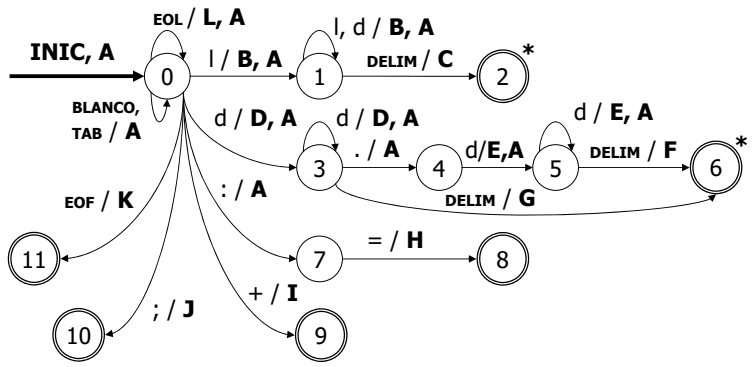


Diagrama de transiciones del autómata finito (2)



Red de transición extendida (*ATN*)



El operador de preanálisis (*lookahead*)

- Motivación:
  - En el reconocimiento del fin de palabras reservadas, números e identificadores:
    - Hay que encontrar un carácter en la entrada que no sea una letra (ni un dígito, para identificadores y números)
    - ¿Qué se hace con ese carácter, si es un ';', por ejemplo?
- Solución genérica:
  - Se usa la variable preanálisis (*lookahead*) para leer el siguiente carácter en la entrada.
  - Si se ha leído un carácter de más en algún momento, se vuelve a depositar en la entrada – acción semántica (*)
    - Así está disponible para la siguiente llamada al escáner.

Listado de acciones semánticas (1)

INIC: Código de inicialización de variables, etc.:

```
{   lexema = ""; parteEntera = 0;          parteDecimal = 0.0;
    base = 10;   pesoDecimal = 1.0 / base; numLineas =0;
}
```

A: Lee el siguiente carácter de la entrada (código fuente):

```
– {  preanalisis = getchar();                }
– {  preanalisis = getc(stdin);              }
```

B: Añade el carácter leído al final del lexema ya acumulado:

```
{   Concatena(lexema, preanalisis);          }
```

C: Búsqueda/inserción en la tabla de símbolos y generación del *token* identificador para el analizador sintáctico:

```
{   token = TS.Busca(lexema);
    if (token == NULL)
        token = GeneraToken(ID, TS.Inserta(lexema));
    return token;
}
```

```
TOK->tipo = ID;
TOK->atributo = … ; // puntero a la TS
```


Listado de acciones semánticas (2)

* : Retrocede una posición el puntero de lectura del fichero fuente (devuelve el último carácter leído a la entrada):

```
{  ungetc(preanalisis, stdin);        }
```

D: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número:

```
{  parteEntera = base * parteEntera + Digito(preanalisis); }
```

E: Conversión y adición del carácter de preanálisis al valor acumulado de la parte decimal del número:

```
{  parteDecimal += (Digito(preanalisis) * pesoDecimal);
   pesoDecimal = pesoDecimal/base;
}
```

F: Genera un *token* de número real para el analizador sintáctico:

```
{  token = GeneraToken (REAL, parteEntera + parteDecimal);
   return token;
}
```

Listado de acciones semánticas (3)

G: Genera un *token* de número entero para el analizador sintáctico:

```
{  token = GeneraToken (INTEGER, parteEntera);
   return token;
}
```

H: Genera un *token* de operador de asignación para el analizador sintáctico:

```
{  token = GeneraToken (OP_ASIG, NULL);
   return token;
}
```

GeneraToken (OPERADOR, ASIG)

I: Genera un *token* de operador de suma para el analizador sintáctico:

```
{  token = GeneraToken (SUMA, NULL);
   return token;
}
```

GeneraToken (OPERADOR, SUMA)

Listado de acciones semánticas (4)

J: Genera un *token* de punto y coma para el analizador sintáctico:

```
{  token = GeneraToken (PUNT_COMA, NULL);
   return token;
}
```

GeneraToken (PUNTUAC, PUNT_COMA)

K : Genera un *token* de fin de la entrada para el analizador sintáctico:

```
{  token = GeneraToken (FIN, NULL);
   return token;
}
```

L : Incrementa el contador del número de líneas:

```
{  numLineas++;  }
```


Tema 2 – Análisis Léxico

Sesión 3: Implementación del analizador léxico

Sesión 3: Implementación del analizador léxico

- Determinación de los errores léxicos
  - Tipos de errores léxicos
  - Metodología (a partir de la matriz de transiciones del *ATN*)
- Ejemplo de implementación del analizador léxico de un *ATN*

Determinación de los errores léxicos (1)

- Tipos de errores léxicos:
  - Operador inexistente (‘=:’)
  - Carácter no admitido (‘ñ’, ‘á’, etc.)
  - Constantes no válidas:
    - Falta la comilla de cierre en (cadenas de) caracteres
    - Con formato incorrecto o inconsistente (‘ab’, 3.14.16, etc.)
  - Identificadores no válidos:
    - Con formato incorrecto (“#dni”, etc.)
    - Límite máximo de caracteres excedido

Determinación de los errores léxicos (2)

- Método práctico:
  - PASO 1: Construcción de la matriz de transiciones del autómata / *ATN*
  - PASO 2: Cada casilla en blanco es un posible error:
    - Se determina el mensaje informativo del error
    - Se codifica como una acción semántica más en la implementación

Determinación de los errores léxicos – PASO 1

EST. / TRANS.	l	d	DELIM	.	:	=	+	;	OTHER	λ
0	1	3	[0]		7		9	10		
1	1	1	2		2		2	2		
2 (*)										0
3		3	6	4	6		6	6		
4		5								
5		5	6		6		6	6		
6 (*)										0
7						8				
8 (*)										0
9 (*)										0
10 (*)										0

Determinación de los errores léxicos – PASO 2 (A)

EST. / TRANS.	l	d	DELIM	.	:	=	+	;	OTHER	λ
0	1	3	[0]	IV	7	IV	9	10	I	
1	1	1	2	IV	2	IV	2	2	I	
2 (*)										0
3	II	3	6	4	6	II	6	6	I	
4	II	5	II	II	II	II	II	II	I	
5	II	5	6	II	6	II	6	6	I	
6 (*)										0
7	III	III	III	III	III	8	III	III	I	
8 (*)										0
9 (*)										0
10 (*)										0

Determinación de los errores léxicos – PASO 2 (B)

I: Error léxico: Carácter de entrada no permitido

II: Error léxico: Constante numérica mal formada

III: Error léxico: Falta el carácter '='

IV: Error léxico: Carácter inesperado en este contexto

Implementación del analizador léxico (1)

```
TipoToken Scan()
{   int state = 0;        /* codifica el estado actual del ATN */
    TipoToken token;

    /* INIC */
    char * lexema = "";
    int parteEntera = 0;
    double parteDecimal = 0.0;
    int base = 10;
    double pesoDecimal = 1.0 / base;
    static int numLinea = 1;  /* Sólo se inicializa la primera vez */

    /* A */
    char preanalisis = getchar();
```


Implementación del analizador léxico (2)

```
while (true)
{   switch (state)
    {   case 0:
            if (preanalisis == EOL)
            {   numLinea++;   preanalisis = getchar();   }                    /* L, A */
            if (preanalisis == EOF)
            {   token := GeneraToken (FIN, NULL); return token;   }          /* K */
            else if (isalpha(preanalisis))
            {   state = 1;   Concatena(lexema, preanalisis);  preanalisis = getchar(); } /* B, A */
            else if (isdigit(preanalisis))
            {   state = 3;   parteEntera = base * parteEntera + Digito(preanalisis);   /* D */
                preanalisis = getchar();   }                                 /* A */
            else if (preanalisis == ':'))
            {   state = 7;   preanalisis = getchar();   }                    /* A */
            else if (preanalisis == '+'))
            {   token = GeneraToken (SUMA, NULL);   return token;   }        /* I */
            else if (preanalisis == ';'))
            {   token = GeneraToken (PUNT_COMA, NULL); return token;   }     /* J */
            else if ((preanalisis != ESPACIO) && (preanalisis != TAB) )
            {   GeneraError("Carácter inesperado", numLinea);
                token = GeneraToken (ERROR, NULL);   return token;   }
            else /* ((preanalisis == ESPACIO) || (preanalisis == TAB) ) */
            {   preanalisis = getchar();   }                                 /* A */
            break;
```


Implementación del analizador léxico (3)

```
        case 1:
            if (isalpha(preanalisis) || isdigit(preanalisis))
            {   Concatena(lexema, preanalisis);                  /* B */
                preanalisis = getchar();                         /* A */
            }
            else if  (Pertenece(preanalisis, DELIM))
            {   ungetc(preanalisis, stdin);                      /*  *  */

                token = TS.Busca(lexema);                        /* C */
                if (token == NULL)                               /* C */
                    token = GeneraToken(ID, TS.Inserta(lexema)); /* C */
                return token;                                    /* C */
            }
            else
            {   GeneraError("Carácter inesperado", numLinea);
                token = GeneraToken (ERROR, NULL);
            }
            break;
```

Implementación del analizador léxico (4)

```
case 3:
    if (isdigit(preanalisis))
    {   parteEntera = base * parteEntera + Digito(preanalisis); /* D */
        preanalisis = getchar();                                /* A */
    }
    else if  (Pertenece(preanalisis, DELIM))
    {   ungetc(preanalisis, stdin);                             /*  *  */

        token = GeneraToken (INTEGER, parteEntera);             /* G */
        return token;                                           /* G */
    }
    else if  (preanalisis == '.')
    {    state = 4;      preanalisis = getchar();       }       /* A */
    else
    {   GeneraError("Carácter inesperado", numLinea);
        token = GeneraToken (ERROR, NULL);
    }
    break;
```


Implementación del analizador léxico (5)

```
case 4:
    if (isdigit(preanalisis))
    {   state = 5;

        parteDecimal += (Digito(preanalisis) * pesoDecimal);    /* E */
        pesoDecimal = pesoDecimal/base;                         /* E */

        preanalisis = getchar();                                /* A */
    }
    else
    {   GeneraError("Carácter inesperado", numLinea);
        token = GeneraToken (ERROR, NULL);    }
    break;
case 7:
    if (preanalisis == '=')
    {   token = GeneraToken (OP_ASIG, NULL);  return token;   }   /* H */

    }
    else
    {   GeneraError("Carácter inesperado", numLinea);
        token = GeneraToken (ERROR, NULL);    }
    break;
```


Implementación del analizador léxico (6)

```
            case 5:
                if (isdigit(preanalisis))
                {   parteDecimal += (Digito(preanalisis) * pesoDecimal);   /* E */
                    pesoDecimal = pesoDecimal/base;                        /* E */
                    preanalisis = getchar();                               /* A */
                }
                else if  (Pertenece(preanalisis, DELIM))
                {   ungetc(preanalisis, stdin);                            /*  *  */

                    token = GeneraToken (REAL, parteEntera + parteDecimal);/* F */
                    return token;                                          /* F */
                }
                else
                {   GeneraError("Carácter inesperado", numLinea);
                    token = GeneraToken (ERROR, NULL);  }
                break;
        }
    }
}
```


Bibliografía

- Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: Principles, Techniques and Tools*. Massachusetts: Addison-Wesley Publishing Company, 1986.
- Alfonseca Cubero, E.; Alfonseca Moreno, M.; Moriyón Salomón, R. Teoría de autómatas y lenguajes formales. Madrid: Mc-Graw-Hill/Interamericana de España, S.A.U., 2007.
- Grogono, P. Programación en Pascal. Wilmington, Delaware (EE.UU.):Addison-Wesley Iberoamericana, 1996.
- Sanchís Llorca, F. J. y Galán Pascual, C. Compiladores: Teoría y construcción. Madrid: Editorial Paraninfo, 1986.