Tema 2

Análisis Léxico

Bibliografía:

- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 3, páginas: 85-158.
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 2, páginas: 31-93.

Contenido:

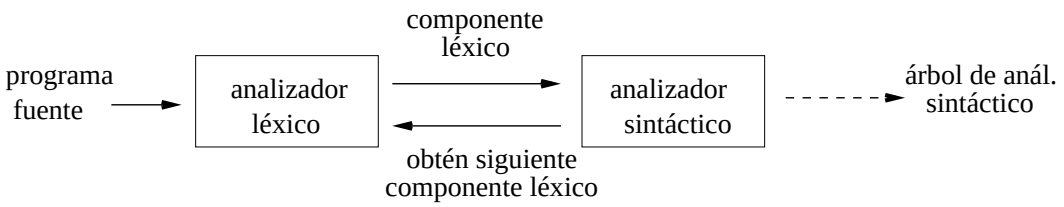
1. Funciones del analizador léxico.
2. Especificación de los componentes léxicos: expresiones regulares.
3. Reconocimiento de los componentes léxicos: autómatas finitos.
4. Implementación de un analizador léxico:
   a) Dirigido por tabla
   b) Mediante bucles anidados.
5. Aspectos prácticos en la implementación de un análisis léxico:
   a) Principio de máxima longitud.
   b) Palabras reservadas versus identificadores.
   c) Gestión del buffer de entrada.
6. Errores léxicos y su tratamiento.
7. Generadores automáticos de analizadores léxicos: *Lex.*

2.1. Funciones del analizador léxico.

Analizador léxico (scanner): lee la secuencia de caracteres del programa fuente, caracter a caracter, y los agrupa para formar unidades con significado propio, *los componentes léxicos* (*tokens* en inglés). Estos componentes léxicos representan:

- palabras reservadas: `if, while, do`, ...
- identificadores: asociados a variables, nombres de funciones, tipos definidos por el usuario, etiquetas,... Por ejemplo: `posicion, velocidad, tiempo`, ...
- operadores: = * + - / == > < & ! = ...
- símbolos especiales: ; ( ) [ ] { } ...
- constantes numéricas: literales que representan valores enteros, en coma flotante, etc, 982, 0xF678, -83.2E+2,...
- constantes de caracteres: literales que representan cadenas concretas de caracteres, "hola mundo",...

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática. Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden obtén el siguiente componente léxico, el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico.



Otras funciones secundarias:

- Manejo del fichero de entrada del programa fuente: abrirlo, leer sus caracteres, cerrarlo y gestionar posibles errores de lectura.
- Eliminar comentarios, espacios en blanco, tabuladores y saltos de línea (caracteres no válidos para formar un *token*).
- Inclusión de ficheros: # include ...
- La expansión de macros y funciones *inline*: # define ...
- Contabilizar el número de lineas y columnas para emitir mensajes de error.
- Reconocimiento y ejecución de las directivas de compilación (por ejemplo, para depurar u optimizar el código fuente).

Ventajas de separar el análisis léxico y el análisis sintáctico:

- Facilita transportabilidad del traductor (por ejemplo, si decidimos en un momento dado cambiar las palabras reservadas *begin* y *end* de inicio y fin de bloque, por { y }, sólo hay que cambiar este modulo.
- Se simplifica el diseño: el analizador es un objeto con el que se interactúa mediante ciertos métodos. Se localiza en un único módulo la lectura física de los caracteres, por lo que facilita tratamientos especializados de E/S.

Componentes Léxicos, Patrones, Lexemas

Patrón: es una regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico (una expresión regular).

Lexema: cadena de caracteres que concuerda con un patrón que describe un componente léxico. Un componente léxico puede tener uno o infinitos lexemas. Por ejemplo: palabras reservadas tienen un único lexema. Los números y los identificadores tienen infinitos lexemas.

Compon. léxico	Lexema	Patrón
identificador	indice, a, temp	letra seguida de letras o dígitos
num_entero	1492, 1, 2	dígito seguido de más dígitos
if	if	letra i seguida de letra f
do	do	letra d seguida de o
op_div	/	carácter /
op_asig	=	carácter =

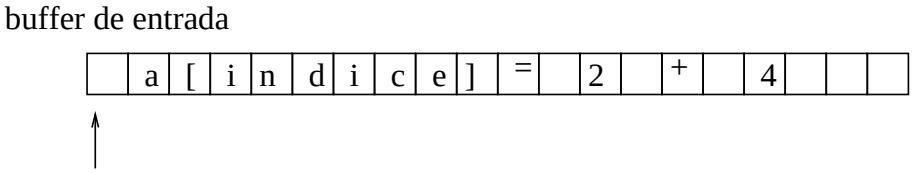
Los componentes léxicos se suelen definir como un tipo enumerado. Se codifican como enteros. También se suele almacenar la cadena de caracteres que se acaba de reconocer (el lexema), que se usará posteriomenete para el análisis semántico.

```
typedef enum{ TKN_IF, TKN_THEN, TKN_NUM, TKN_ID, TKN_OPADD,...} TokenType;
```

Es importante conocer el lexema (para construir la tabla de símbolos). Los componentes léxicos se representan mediante una estructura registro con tipo de *token* y lexema:

```
typedef struct {
      TokenType token;
      char *lexema; //se reserva memoria dinámicamente
} TokenRecord;
TokenRecord getToken(void);
```

a[indice]= 2 + 4



Cada componente léxico va acompañado de su lexema:

```
<TKN_ID, a>
<TKN_CORAPER, [>
<TKN_ID, indice>
<TKN_CORCIERRE, ]>
<TKN_NUM, 2>
<TKN_OPADD, +>
<TKN_NUM, 4>
```

2.2. Especificación de los componentes léxicos: expresiones regulares.

Los componentes léxicos se especifican haciendo uso de expresiones regulares. Además de las tres operaciones básicas: concatenación, repetición (*) y alternativas (|) vamos a usar los siguientes metasímbolos:

Una o más repeticiones +

$r+$ indica una o más repeticiones de r

$(0|1)+ = (0|1)(0|1)^*$

Cualquier carácter .

.*b.* indica cualquier cadena que contiene una letra b

Un rango de caracteres [ ] (clase)

[a-z] indica cualquier carácter entre la a y z minúsculas

[a-zA-Z] indica cualquier letra del abecedario minúscula o mayúscula

[0-9] indica cualquier dígito de 0 a 9

[abc] indica $a|b|c$

Cualquier carácter excepto un conjunto dado ~

~ (a|b) indica cualquier carácter que no sea una a ó b

Opcionalidad ?

r? indica que la expresión r puede aparecer o no. En el caso de que aparezca sólo lo hará una vez.

Cuando queremos usar estos símbolos con su significado tenemos que usar la barra de escape, [a\-z] significaría cualquier letra que sea a, guión o z.

Algunos ejemplos:

Números

```
nat = [0-9]+
signedNat = ( + | -)? nat
number = signedNat(''.''nat)? (E signedNat)?
```

Identificadores

```
letter = [a-zA-Z]
digit = [0-9]
```

```
identifier = letter (letter | digit)*
```

Palabras Reservadas

```
tkn_if = ‘‘if’’
tkn_while = ‘‘while’’
tkn_do = ‘‘do’’
```

Comentarios

`{ (~ })* }` comentarios en Pascal

Delimitadores

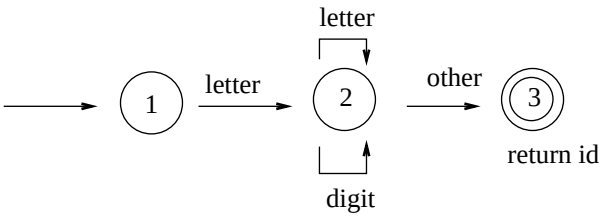
```
delimitadores = (newline | blank | tab | comment)+
```

2.3. Reconocimiento de los componentes léxicos: autómatas finitos.

Los AFD se pueden utilizar para reconocer las expresiones regulares asociadas a los componentes léxicos.

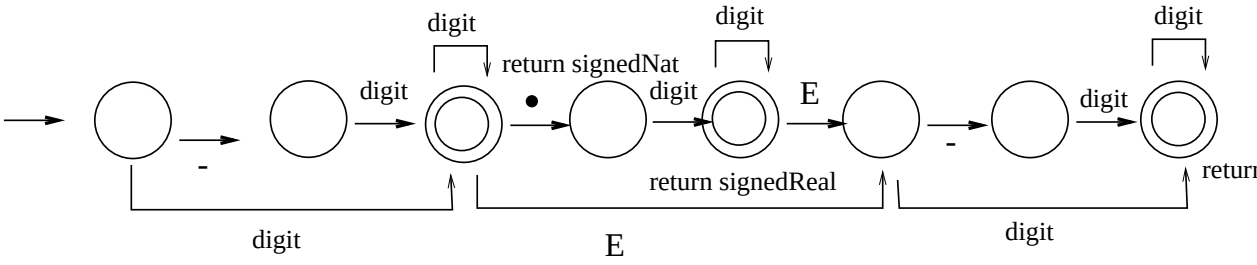
Identificadores

```
identifier = letter (letter | digit)*
```



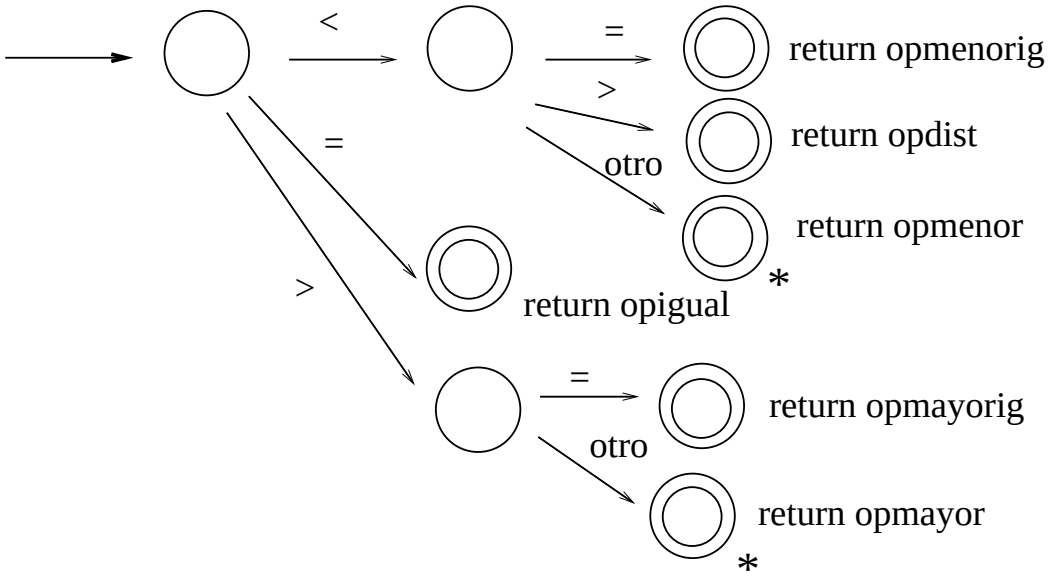
Números naturales y reales

```
nat = [0-9]+
signedNat = (-)? nat
number = signedNat(‘‘.’’nat)? (E signedNat)?
```



Operadores relacionales

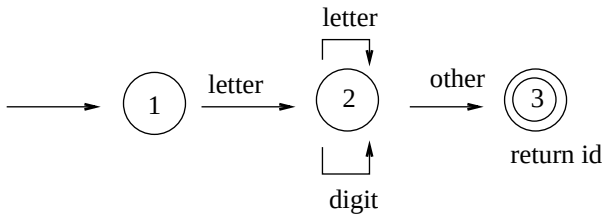
Operadores < | = | <= | <> | >= | >



El símbolo * indica que se debe hacer un retroceso en la entrada pues se ha consumido un símbolo demás, que no pertenece al lexema del componente léxico que se devuelve.

2.4. Implementación de un analizador léxico.

Supongamos que queremos reconocer identificadores:



Mediante bucles anidados

Usamos una variable para almacenar el estado actual y una estructura tipo `case` doble anidada. El primer `case` comprueba el estado actual y el siguiente el carácter en la entrada. Las transiciones se corresponden con asociar un nuevo estado a la variable y avanzar en la entrada.

```
ch=next input char;
state = 1;
while (state == 1 o 2) do
      case state
      1: case ch
           letter : avanzar en la entrada; state=2;
           otro caso : state = error u otro
           fin_case;
      2: case ch
           letter, digit : avanzar en la entrada; state=2 (no necesario);
           otro caso : state = 3;
           fin_case;
      fin_case;
fin_while;
if (state==3) then aceptar else error;
```

El código que se genera es largo y difícil de mantener en el caso de que se introduzcan nuevos caracteres en el alfabeto de entrada o nuevos estados.

Mediante una tabla de transiciones

input/state	letter	digit	other	Accepting
1	2			no
2	2	2	[3]	no
3				yes

Se asume que los campos en blanco son errores. Los estados de aceptación se marcan con una columna adicional. Los corchetes representan que no se tiene que consumir un carácter en la entrada (no avanzar).

```
state = 1;
ch = next_input_character;
while (not Accept[state]) and (not error(state)) do
      newstate = T[state,ch];
      if Advance[state,ch] then ch=next_input_char;
      state=newstate;
end while;
if Accept[state] then accept;
```

`Advance` y `Accept` son dos arrays booleanos indexados por el estado y por el carácter en la entrada. El primero indica si tenemos que avanzar en la entrada. El segundo si tenemos un estado de aceptación.
Ventajas: el código es reducido y fácil de mantener, sirve para cualquier analizador, las modificaciones se realizan sólo en la tabla.

Desventajas: tablas de gran tamaño, se incrementa el espacio necesario (velocidad algo reducida respecto al método anterior). Este método es el que utiliza Flex (Linux) y Lex (Unix).
Coste: $O(|x|)$, es independiente del tamaño del autómata.

2.5. Aspectos prácticos en la implementación de un analizador léxico

Principio de máxima longitud

Se da prioridad al componente léxico de máxima longitud.
Por ejemplo: <> se interpreta como el operador "distinto de", en vez de "menor que" y "mayor que".
Por ejemplo: `ende` se interpreta como el identificador `ende` y no como la palabra reservada `end` y la letra e.

Palabras reservadas versus identificadores

Un lenguaje de programación puede tener del orden de 50 palabras reservadas. Para evitar tener un AFD demasiado grande las palabras reservadas se reconocen como identificadores (una palabra reservada también es una letra seguida de más letras) y se comprueba antes de decidir el tipo de *token* si se trata de una palabra reservada o de un identificador consultando una tabla previamente inicializada con las palabras reservadas. Este método es recomendable cuando el número de palabras reservadas es grande.

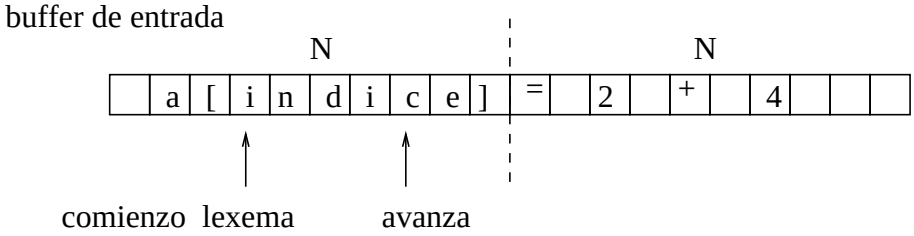
Entrada de los identificadores en la Tabla de Símbolos

En lenguajes sencillos con sólo variables globales y declaraciones, es normal implementar el *scanner* para que introduzca los identificadores en la Tabla de Símbolos conforme los va reconociendo, si es que no han sido ya introducidos. Después, el analizador sintáctico se encarga de introducir información adicional sobre su tipo, etc, conforme la va obteniendo durante el análisis sintáctico. Si se trata de un lenguaje estructurado, el *scanner* no introduce los identificadores en la Tabla de Símbolos, porque en este tipo de lenguajes, los identificadores pueden tener diferentes significados según su contexto (como una variable, como una función, como un campo de una estructura, . . . ,). Es el análisis sintáctico, cuando ya ha recogido información sobre su ámbito, cuando introduce los identificadores en la Tabla de Símbolos.

Gestión del buffer de entrada

El proceso de lectura de los caracteres de la entrada y formar los componentes léxicos es lo más costoso en tiempo en el proceso de

traducción. Es importante implementarlo eficientemente.
Se utiliza un buffer dividido en dos mitades de tamaño N caracteres, donde N es un bloque de disco (1024, 4096). Se leen N caracteres de la entrada en cada mitad del buffer con una única orden de lectura. Se mantienen dos apuntadores. Uno marca el inicio del lexema y el otro el carácter actual que se mueve hasta encontrar una subcadena que corresponde con un patrón. Una vez reconocido un componente léxico ambos apuntadores se colocan en la misma posición y justo detrás del lexema reconocido.




```
if avanza está final primera mitad then
      recargar segunda mitad;
      avanza = avanza+1;
elseif avanza está final segunda mitad then
      recargar primera mitad;
      pasar avanza a principio primera mitad;
else avanza = avanza+1;
```

A tener en cuenta en la implementación:

- El análisis léxico es una subrutina del análisis sintáctico que devuelve el tipo de componente léxico y el lexema cada vez que es llamada.
- Usar un tipo enumerado para los tipos de componentes léxicos. Usar un tipo enumerado para los estados del analizador.
- En el analizador léxico debe haber una función que se encarga de gestionar el buffer de entrada. Se leerá una linea que se almacenará en un vector de caracteres. Cuando haya sido procesada se carga de nuevo.
- Almacenar en variables el número de linea y columna para emitir mensajes de error.

- Las palabras reservadas se reconocen como identificadores y antes de devolver un identificador se comprueba si es una palabra reservada o un identificador consultando en una tabla previamente inicializada con las palabras reservadas.
- Hay casos en los que es necesario reinsertar un carácter en el buffer de entrada.
- Además de los componentes léxicos definidos por el lenguaje es conveniente añadir un par especiales para indicar el final de fichero y la detección de un error.
- Usar las funciones *isdigit()*, *isalpha()* para implementar las transiciones en el AFD.

2.6. Tratamiento de errores léxicos

Los errores léxicos se detectan cuando el analizador léxico intenta reconocer componentes léxicos y la cadena de caracteres de la entrada no encaja con ningún patrón. Son situaciones en las que usa un carácter inválido (@,$,",¿,...), que no pertenece al vocabulario del lenguaje de programación, al escribir mal un identificador, palabra reservada u operador.
Errores léxicos típicos son:

1. *nombre ilegales de identificadores*: un nombre contiene caracteres inválidos.
2. *números incorrectos*: un número contiene caracteres inválidos o no está formado correctamente, por ejemplo `3,14` en vez de `3.14` ó `0.3.14`.
3. *errores de ortografía en palabras reservadas*: caracteres omitidos, adicionales o cambiados de sitio, por ejemplo la palabra `hwile` en vez de `while`.
4. *fin de archivo*: se detecta un fin de archivo a la mitad de un componente léxico.

Los errores léxicos se deben a descuidos del programador. En general, la recuperación de errores léxicos es sencilla y siempre se traduce en la generación de un error de sintaxis que será detectado más tarde por el analizador sintáctico cuando el analizador léxico devuelve un componente léxico que el analizador sintáctico no espera en esa posición.
Los métodos de recuperación de errores léxicos se basan bien en saltarse caracteres en la entrada hasta que un patrón se ha podido reconocer; o bien usar otros métodos más sofisticados que incluyen la inserción, borrado, sustitución de un carácter en la entrada o intercambio de dos caracteres consecutivos. Una buena estrategia para la recuperación de errores léxicos:

- si en el momento de detectar el error ya hemos pasado por algún estado final ejecutamos la acción correspondiente al último estado final visitado con el lexema formado hasta que salimos de él; el resto de caracteres leídos se devuelven al flujo de entrada y se vuelve al estado inicial;
- si no hemos pasado por ningún estado final, advertimos que el carácter encontrado no se esperaba, lo eliminamos y proseguimos con el análisis.

Por ejemplo, ante la entrada $73.a$:

- devolvemos el componente léxico entero con lexema 73;
- devolvemos al buffer de entrada los caracteres $.a$ y
- volvemos al estado inicial.

En la siguiente llamada al analizador léxico se producirá un nuevo error, en este caso descartamos el carácter leído (el punto) y seguimos con el análisis. En la siguiente llamada detectamos el identificador a.
Si se sabe que el siguiente componente léxico es una palabra reservada (en la gramática esperamos una palabra reservada) es posible corregir la palabra mal escrita. Por ejemplo, si se ha escrito `hwile` en vez de `while` o `fi` en vez de `if`, intercambiando caracteres adyacentes.

2.7. Generadores automáticos de analizadores léxicos: *Lex*

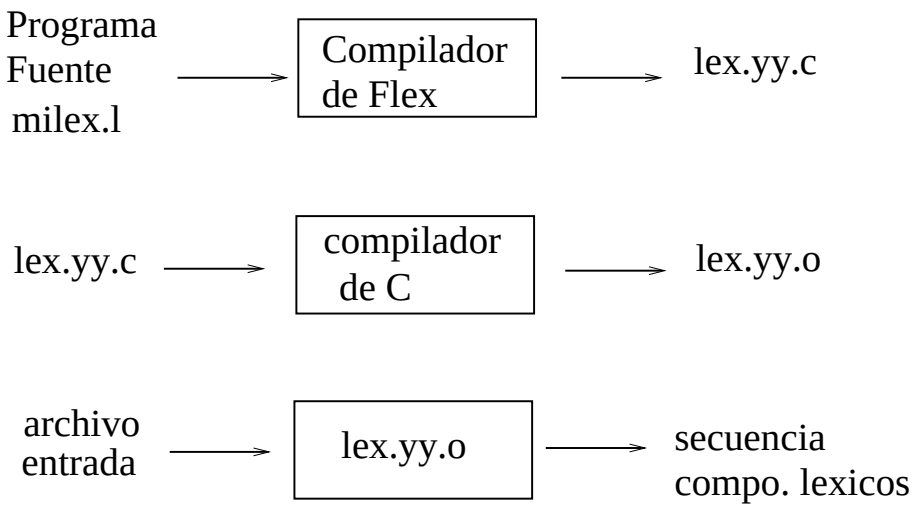
Todos los analizadores léxicos realizan la misma función (se implementan de igual forma) excepto en los tokens que reconocen, las expresiones regulares que los definen. Resulta entonces natural y una forma de ahorar esfuerzo, utilizar generadores automáticos de analizadores léxicos. Estos generadores sólo necesitan conocer la especificación de tokens a reconocer.

Los analizadores léxicos (el AFD) se pueden implementar a mano para reconocer los componentes léxicos de un determinado lenguaje. Este opción tiene sentido sólo cuando el aprender a utilizar una determinada herramienta nos llevaría más tiempo que el hacerlo nosotros a mano. Originalmente, esta era la forma habitual, pues los realizados a mano eran más rápidos que los generados por las herramientas automáticas (basados en tablas). (el proceso de análisis léxico conlleva la mayor parte del tiempo del proceso de compilación). Las últimas herramientas con las técnicas de compresión de tablas son más eficientes.
Flex (*Fast Lexical Analyzer Generator*) es un generador automático de analizadores léxicos en lenguaje C. Es software de GNU.

- *Entrada*: un fichero texto con la especificación de los componentes léxicos, las expresiones regulares que los definen.
- *Salida*: un programa en C que implementa dicho analizador, preparado para ser compilado y utilizado.

Un fichero Lex consiste de tres partes: definiciones, reglas y rutinas auxiliares, separadas por %%

```
{ definiciones }
%%
{ reglas }
patrón 1   { código C }
patrón 2   { código C }
...
patrón n   { código C }
%%
```




```
{ código auxiliar }
```

La sección de definiciones incluye declaraciones de variables, constantes y definiciones regulares que pudieran ser utilizadas más adelante.
La sección de reglas describe cada patrón y cada acción es un fragmento de código que indica la acción que se debe realizar cuando se encuentra dicho patrón.
La sección de código auxiliar contienen las funciones que pueden ser llamadas por las acciones de la sección de reglas.
Algunas consideraciones:

- Cualquier código C que se desea insertar va comprendido entre %{ y %}.
- Flex hace correspondencia siempre con la cadena más larga.
- Si se verifican dos patrones, se elige la regla que aparece primero (colocar las palabras reservadas antes que los identificadores).
- Para definir las expresiones regulares se utiliza los símbolos:

- Existen los siguientes nombres internos:

Símbolo	Uso	Ejemplo
[ ]	definir clase	[a-z]
-	definir rangos	[a-z]
\	caracter de “escape” para representar símb. especiales	$'\backslash t'$
^	negación	[^xy]
“ “	representar una cadena de caracteres	“while”
* + ( ) |	operadores habituales para expre. regulares	[0-9]+
?	opcionalidad	[0-9]+(”.”[0-9]+)?
{ }	expansión macro definida en seccion primera	{digit}

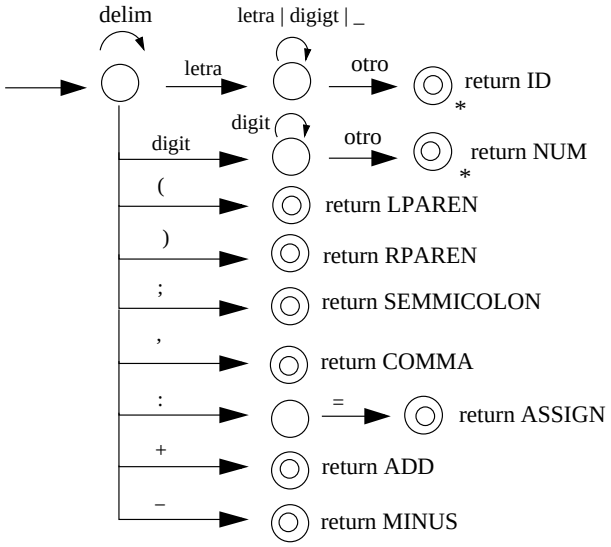
char * yytext;	// almacena el lexema reconocido
int yyleng	longitud del lexema
int yylong ;	// longitud del lexema
int yylex();	// llamada al analizador, devuelve el tipo de token (0 si EOF)
FILE * yyin;	//fichero de entrada, por defecto stdin
FILE * yyout;	//fichero de salida, por defecto stdout
char input();	devuelve el caracter actual en buffer
void output(c);	escribe un carracter en la salida
void unput(c);	devuelve un caracter al buffer de entrada
yywrap();	to wrap up input processing

Para compilar:
flex ejemplo1.l
cc lex.yy.c -o milex -lfl
milex < ficheroentrada

```
%{ /* Este programa cuenta el número de identificadores, reales y enteros
que aparecen en un programa */
int nlineas=0;
int nid=0;
int nentero=0;
int nreal=0;
%}
letra [a-zA-Z]
digito [0-9]
%%
{letra}({letra}|{digito})*    {printf("identificador%s \n", yytext);
nid++;}
{digito}+                {printf("entero%d \n", atoi(yytext)); nentero++;}
{digito}+ ".” {digito}+       [\ n]                        {nlineas++;}
.                          {/* otra cosa, no hacer nada */}
%%
int main(int argc, char **argv)
{
   yylex();
   printf (”Número de lineas:%d \n", nlineas);
   printf (”Número de identificadores:%d \n", nid);
   printf (”Número de enteros:%d \n", nentero);
   printf (”Número de reales:%d \n", nreal);
   exit(0);
}
```

2.8. El Scanner para Micro

Veamos un ejemplo de la implementación del análisis léxico del lenguaje Micro descrito en el tema anterior. El AFD es:



- El analizador léxico (scanner) lee un programa fuente de un fichero y produce una secuencia de componentes léxicos. El scanner es de hecho una función (`GetToken( )` que devuelve un token cada vez que es llamada por el analizador sintáctico.
- Se ha utilizado tipos enumerados para definir los tipos de tokens y estados del AFD.
- Se ha implementado una tabla de palabras reservadas, de manera que cuando se encuentra un **id** buscamos en la tabla (función `LookUpResrvedWords()`) para ver si se trata efectivamente de un identificador o de una palabra reservada.
- El AFD se ha implementado mediante bucles anidados.
- El buffer de entrada es un vector de caracteres de longitud fija que se actualiza de linea en linea. Hay que tener en cuenta a veces en "no leer demasiado", en ocasiones es necesario leer el principio del siguiente token para saber que el actual ha terminado (esta situación la marcamos con *), de ahí el hecho de implementar la función `UnGetChar()`.

- Como delimitadores se ha usado: el espacio en blanco, el tabulador y final de línea.
- Para los tokens **id** y **num** es necesario recoger el lexema para poder después realizar comprobaciones semánticas (por eje: declaración antes de uso). El lexema de los componentes léxicos se almacena en el campo `lexema` de la estructura. Por comodidad se ha almacenado para todos los tipos de tokens.
- Para gestionar el final de fichero se ha creado el token `TKN_EOF`, que es lo que devuelve el analizador léxico cuando encuentra el caracter de final de fichero, y así verificar que se ha procesado todo el fichero de entrada.

21 abr 03 12:52 **MicroScanner.c** Página 1/4

```
/*   Este programa implementa un alnalizador lexico para el lenguaje MICRO.
La Entrada es una cadena de caracteres (fichero fuente) y la Salida son tokens.
Suponemos que el buffer de entrada es simplemente un vector de caracteres de lon
gitud fija, que vamos recargando conforme vamos agotando. Leemos una linea cada
vez, hasta encontrar el final de fichero (EOF)
EL AFD está implementado mediante bucles anidados.
La distinción entre IDs y Palabras reservadas se hace mediante la inicialización
 de una tabla de palabras reservadas. Los identificadores tienen una longitud in
ferior a 32 caracteres.
Variables globales: contador de linea y columna, y el buffer de entrada. */

#include <stdio.h>
#include <string.h>
#include <ctype.h> //para funciones isdigit, isalpha ...
#define MAXLENID 32
#define MAXLENBUF 1024
#define MAXRESWORDS 4

typedef enum {
  TKN_BEGIN, TKN_END, TKN_READ, TKN_WRITE, TKN_ID, TKN_NUM, TKN_LPAREN, TKN_RPAR
EN, TKN_SEMICOLON, TKN_COMMA, TKN_ASSIGN, TKN_ADD, TKN_MINUS, TKN_EOF, TKN_ERROR
 }
token_types;

typedef enum {
  IN_START, IN_ID, IN_NUM, IN_LPAREN, IN_RPAREN, IN_SEMICOLON, IN_COMMA, IN_ASSI
GN, IN_ADD, IN_MINUS, IN_EOF, IN_ERROR, IN_DONE}
States;

typedef struct {
  token_types type;
  char lexema[MAXLENID];
} Token;

Token ReservedWords[MAXRESWORDS]={ {TKN_BEGIN, "begin"}, {TKN_END, "end"}, {TKN_R
EAD, "read"}, {TKN_WRITE, "write"} };

int nline=0;
int ncol=0;
char buffer[MAXLENBUF];

Token LookUpReservedWords(char *);
char GetChar(FILE *);
void UnGetChar(void);
Token GetToken(FILE *);
int isdelim(char c) ; //devuelve 1/0 si c es un blanco, tab, \n

Token LookUpReservedWords(char *s)
{
  int i=0;
  Token tok;

  for (i=0; i< MAXRESWORDS; i++) {
      if (strcmp(s, ReservedWords[i].lexema)==0) {
        return(ReservedWords[i]);
      }
    }
  strcpy(tok.lexema,s);
  tok.type=TKN_ID;
  return(tok);
}
```

21 abr 03 12:52 **MicroScanner.c** Página 2/4

```
char GetChar(FILE *fp) {

  char c;
  static int n; //longitud línea leida, se guarda valor de llamada a llamada

  if (( ncol==0) || (ncol==n) ) {
    if (NULL!=fgets(buffer, MAXLENBUF, fp)) /* lee hasta el salto de linea */
      {
        n=strlen(buffer);
        ncol=0;
        nline++;
      }
    else {
      return(EOF);
    }
  }
  c=buffer[ncol++];
  return (c);
}

void UnGetChar() {

  ncol--;
}

int isdelim(char c) {

  char delim[3]={' ', '\t', '\n'};
  int i;

  for (i=0;i<3;i++) {
    if (c==delim[i]) {
      return(1);
    }
  }
  return(0);
}

Token GetToken(FILE *fp) {

  char c;
  States state=IN_START;
  Token token;
  int index=0; //indice al caracter actual del lexema

  while (state!=IN_DONE) {
    switch (state) {
    case IN_START: {
      c=GetChar(fp);
      while ( isdelim(c)) {
        c=GetChar(fp);
      }
      if (isalpha((int) c)) {
        state=IN_ID;
        token.lexema[index++]=c;
      }
      else if (isdigit((int) c)) {
        state=IN_NUM;
        token.lexema[index++]=c;
      }
      else if (c=='(') {
        token.type=TKN_LPAREN;
```

```
        state=IN_DONE;
        token.lexema[index++]=c;
      }
      else if (c==')') {
        token.type=TKN_RPAREN;
        state=IN_DONE;
        token.lexema[index++]=c;
      }
      else if (c==';') {
        token.type=TKN_SEMICOLON;
        state=IN_DONE;
        token.lexema[index++]=c;
      }
      else if (c==',') {
        token.type=TKN_COMMA;
        state=IN_DONE;
        token.lexema[index++]=c;
      }
      else if (c==':') {
        state=IN_ASSIGN;
        token.lexema[index++]=c;
      }
      else if (c=='+') {
        token.type=TKN_ADD;
        state=IN_DONE;
        token.lexema[index++]=c;
      }
      else if (c=='-') {
        token.type=TKN_MINUS;
        state=IN_DONE;
        token.lexema[index++]=c;
      }
      else if (EOF==c) {
        token.type=TKN_EOF;
        state=IN_DONE;
        token.lexema[index++]=c;
      }
     else {
        token.type=TKN_ERROR;
        state=IN_DONE;
      }
    break;
    }
    case IN_NUM: {
        c=GetChar(fp);
        token.lexema[index++]=c;
        if (!isdigit((int) c) ) {
          token.type=TKN_NUM;
          state=IN_DONE;
          UnGetChar();
          index--;
        }
        break;
    }
    case IN_ID: {
        c=GetChar(fp);
        token.lexema[index++]=c;
        if (! ( (isalnum( (int) c) ) || (c=='_') ) ) {
          token.type=TKN_ID;
          state=IN_DONE;
          UnGetChar();
          index--;
```



```
          token.lexema[index]='\0';
          token=LookUpReservedWords(token.lexema);
        }
        break;
    }
    case IN_ASSIGN: {
        c=GetChar(fp);
        token.lexema[index++]=c;
        if (c=='=') {
          token.type=TKN_ASSIGN;
          state=IN_DONE;
          index--;
        }
        break;
    }
  default: {
      token.type=TKN_ERROR;
      state=IN_DONE;
      token.lexema[index++]=c;
    }
  } //end switch
}//end while

   if (token.type==TKN_ERROR) {
     fprintf(stderr, "\nLine %d:%d, Error: Character %c does not match any token\n", nline, ncol,
 c);
   }
   token.lexema[index]='\0'; //para acabar la cadena
   return(token);
}

int main(int argc, char **argv){

  FILE *fp;
  Token token;

  if (argc==2) {
    if ( NULL== (fp= fopen(argv[1],"r")) ) {
      fprintf(stderr, "File %s cannot be open\n", argv[1]);
      return(0);
    }
    else {
      token=GetToken(fp);
      while (TKN_EOF != token.type) {
        fprintf(stderr,"(%d,%s) ", token.type, token.lexema);
        token=GetToken(fp);
      }
      printf("\n%d lineas analizadas", nline);
      return(0);
    }
  }
  else {
    fprintf(stderr, "Usage: program_name file_name\n");
    return(0);
  }
}
```

2.9. Ejercicios

1. (0.3 ptos) Diseña una expresión regular y construye el AFD para las siguientes especificaciones en lenguaje natural:

   a) Los números enteros expresados en notación decimal, octal o hexadecimal. En notación decimal, un entero consiste en uno o más dígitos. En notación octal hay al menos dos dígitos, el primero de los cuales es siempre un cero y los restantes quedan comprendidos entre el cero y el siete. En notación hexadecimal, hay al menos tres caracteres, los dos primeros caracteres son un cero y una equis y los restantes son dígitos del cero al nueve o letras entre la A y la F.

   b) La dirección de un correo electrónico.

   c) Las palabras clave: `char, integer, float`.

   d) La dirección de una página web.

   e) Identificadores y las palabras reservadas: `case, char, const, continue`.

   f) Comentarios tipo C y C++ /* esto es un comentario */, //esto es un comentatio.

   g) Los identificadoes compuestos de letras, digitos y subrayados, que empiezan por una letra y no puede haber más de un subrayado seguido.

   h) Los comentarios delimitados por ##, que permiten un # dentro del cuerpo del comentario.

2. (0.3 ptos) Construye el AFD para reconocer los operadores menor que (<), menor o igual que (<=), igual que (==), mayor que (>), mayor o igual que (>=), distinto de (<>). Escribe en pseudocódigo el código correspondiente para su implementación.

3. (0.2 ptos) Modifica el AFD y el código de la implementación del AFD para el lenguaje Micro para que reconozca comentarios. Recordad: un comentario comienza por – y acaba al final de la línea actual.

4. (0.3 ptos) Escribe en pseudocódigo la implementación del AFD para el caso 1.3 y 1.6, bien usando bucles anidados o una tabla de transiciones.

5. (0.2 ptos) Escribe un programa en Lex que cuente el número de veces que aparece las palabras reservadas `while, if`, el operador de asignación `=` y el número de identificadores en un programa en C.

6. (0.2 ptos) Escribe un programa en Lex que sustituya las palabras reservadas `while, do, if, else` a mayúsculas. Debe generar un fichero igual al de la entrada donde se han cambiado esas palabras reservadas.

7. (0.2 ptos) Supongamos que hay una pagina web donde se quiere cambiar todas las referencias `http:\\bugs.uv.es` por `http:\\informatica.uv.es`. Escribe un programa Lex que lo haga de forma automática.