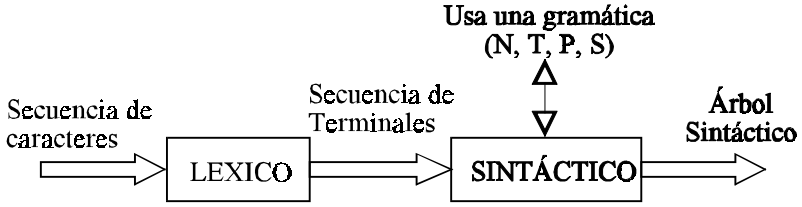
Tema 2.
Análisis lexicográfico

Este capítulo estudia la primera fase de un compilador, es decir su análisis lexicográfico, o más concisamente análisis léxico. Las técnicas utilizadas para construir analizadores léxicos también se pueden aplicar a otras áreas, como, por ejemplo, a lenguajes de consulta y sistemas de recuperación de información. En cada aplicación, el problema de fondo es la especificación y diseño de programas que ejecuten las acciones activadas por palabras que siguen ciertos patrones dentro de las cadenas a reconocer. Como la programación dirigida por patrones es de mucha utilidad, se introduce un lenguaje de patrón-acción, llamado LEX, para especificar los analizadores léxicos. En este lenguaje, los patrones se especifican por medio de expresiones regulares, y un compilador de LEX puede generar un reconocedor de las expresiones regulares mediante una autómata finito eficiente.

Por otro lado, una herramienta *software* que automatiza la construcción de analizadores léxicos permite que personas con diferentes conocimientos utilicen la concordancia de patrones en sus propias áreas de aplicación.

✯ ¿Que es un analizador léxico?

Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones.
La entrada del analizador léxico podemos definirla como una secuencia de caracteres.



gramática (N, T, P, S)

- N ➪ Símbolos no terminales.
- T ➪ Símbolos terminales
- P ➪ Reglas de producción
- S ➪ Axioma inicial

El analizador léxico tiene que dividir la secuencia de caracteres en palabras con significado propio y después convertirlo a una secuencia de terminales desde el punto de vista del analizador sintáctico, que es la entrada del analizador sintáctico.

El analizador léxico reconoce las palabras en función de una gramática regular de manera que sus NO TERMINALES se convierten en los elementos de entrada de fases posteriores. En LEX, por ejemplo, esta gramática se expresa mediante expresiones regulares.

Funciones del analizador léxico

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción, suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden "Dame el siguiente componente léxico"del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

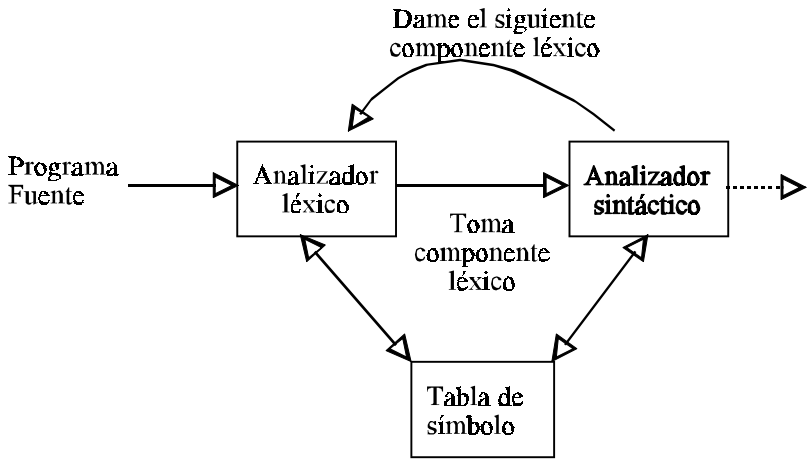


Figura 2 Interacción de un analizador léxico con el analizador sintáctico

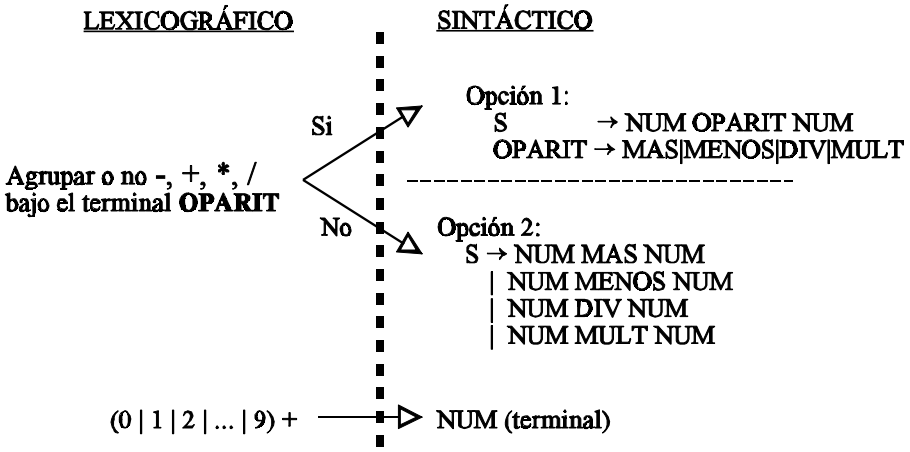
Otras funciones que realiza :

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, ..., y tratarlos correctamente con respecto a la tabla de símbolos (solo en los casos que debe de tratar con la tabla de símbolos).
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información sobre donde se ha producido.
- Avisar de errores léxicos. Por ejemplo, si @ no pertenece al lenguaje, avisar de un error.
- Puede hacer funciones de preprocesador.

Necesidad del analizador léxico

Un tema importante es el porqué se separan los dos análisis lexicográfico y sintáctico, en vez de realizar sólo el análisis sintáctico, del programa fuente, cosa perfectamente posible aunque no plausible. Algunas razones de esta separación son:

- Un diseño sencillo es quizás la consideración más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una u otra de dichas fases. El analizador léxico nos permite simplificar el analizador sintáctico.



Si el sintáctico tuviera la gramática de la Opción 1 , el lexicográfico sería:

Opción 1: (0 | 1 | 2 | ... | 9) + ⟹ NUM
(“+” | “-” | ”*“ | ”/“) ⟹ OPARIT

Si en cambio el sintáctico toma la Opción 2, el lexicográfico sería:

Opción 2: (0 | 1 | 2 | ... | 9) + ⟹ NUM
“+” ⟹ MAS
“-” ⟹ MENOS
“*” ⟹ MULT
“/” ⟹ DIV

Es más, si ni siquiera hubiera análisis léxico, el propio análisis sintáctico vería incrementado su número de reglas:

NUM ⟹ 0
| 1
| 2
| 3
....
| NUM NUM

Necesidad del analizador léxico

A modo de conclusión, diremos que tenemos dos gramáticas, una que se encarga del análisis léxico y otra que se encarga del análisis sintáctico. ¿Que consideramos componente básico?, ¿Donde ponemos el punto divisor de qué se encarga cada gramática?. Si las divisiones se hacen muy pequeñas estamos complicando la gramática, por ejemplo, en la opción 2, la gramática sintáctica se nos complica un poco. Seguiremos dos reglas para que no se nos complique. La primera es que tendremos que hacer divisiones de forma que no perdamos información, esto quedará más claro en capítulos posteriores, y nos veremos ayudados por el concepto de *atributo*. La segunda es que por regla general el analizador lexicográfico debe de encargarse de la parte que involucra una gramática regular (que nosotros expresaremos mediante expresiones regulares).

- Se mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para esa función. Gran parte del tiempo se consume en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de buffers para la lectura de caracteres de entrada y procesamiento de componentes léxicos se puede mejorar significativamente el rendimiento de un compilador.

- Se mejora la portabilidad del compilador. Las peculiaridades del alfabeto de entrada y otras anomalías propias de los dispositivos pueden limitarse al analizador léxico. La representación de símbolos especiales o no estándares, como ↑ en Pascal, pueden ser aisladas en el analizador léxico.

- Otra razón por la que se separan los dos análisis es para que el analizador léxico se centre en el reconocimiento de componentes básicos complejos. Por ejemplo en FORTRAN, existen el siguiente par de proposiciones :

  DO 5 I = 2.5 (Asignación de 2.5 a la variable DO5I)
  DO 5 I = 2,5 (Bucle que se repite para I = 2, 3, 4, 5)

  En éste lenguaje los espacios en blancos no son significativos fuera de los comentarios y de un cierto tipo de cadenas, de modo que supóngase que todos los espacios en blanco eliminables se suprimen antes de comenzar el análisis léxico. En tal caso, las proposiciones anteriores aparecerían al analizador léxico como

  DO5I = 2.5
  DO5I = 2,5

  El analizador léxico no sabe si DO es una palabra reservada o es el prefijo de una variable hasta que llegue a la coma. El analizador ha tenido que mirar más allá de la propia palabra a reconocer haciendo lo que se denomina lookahead (o prebúsqueda).

Conceptos de tokens, patrones y lexemas

El analizador lexicográfico puede tener la siguiente estructura:

```
(Expresión regular 1)      { acción 1}
(Expresión regular 2)      { acción 2}
(Expresión regular 3)      { acción 3}
        ⋮                      ⋮

(Expresión regular n)      { acción n}
```

Donde cada acción es un fragmento de programa que describe cual ha de ser la acción del analizador léxico cuando la secuencia de entrada coincida con la expresión regular.

- **Patrón** : es una expresión regular.
- **Token** : es el terminal asociado a un patrón. Cada token se convierte en un número que es un código identificativo de cada patrón. En algunos casos, cada número tiene asociado un puntero a la tabla de símbolos. Utilizamos la palabra terminal desde el punto de vista de la gramática utilizada por el analizador sintáctico.
- **Lexema** : Es cada secuencia de caracteres concreta que encaja con un patrón, es decir, es como una instancia de un patrón.

  Ej: 8, 23, 50 (son lexemas que encajan con el patrón (0 | 1 | 2 | ... | 9) +)

Una vez detectado que un grupo de caracteres coincide con un patrón, se ha detectado un lexema. A continuación se le asocia un número, que se le pasará al sintáctico, y, si es necesario, información adicional, como puede ser una entrada en la tabla de símbolos.

La tabla de símbolos suelen ser listas encadenadas de registros con parte variable: listas ordenadas, árboles binarios de búsqueda, tablas hash, etc.

Ejemplo: Hacer un analizador léxico que nos reconozca los números enteros, los números reales y los identificadores de usuario. Vamos a hacer este ejemplo en C.

Expresión Regular	⟹	Terminales
(0 ... 9) +	⟹	NUM_ENT
(0 ... 9)*. (0 ... 9) +	⟹	NUM_REAL
(a ... z) (a ... z 0 ... 9) *	⟹	ID

Asociado a la categoría gramatical de número entero tendremos el token NUM_ENT que puede equivaler por ejemplo al número 280; asociado a la categoría gramatical número real tendremos el token NUM_REAL que equivale al número 281; asociado a la categoría gramatical identificador de usuario tendremos el token ID que equivale al número 282.

```
( 0 ... 9) +              { return 280;}
(0 ... 9)*. (0 ... 9) +   { return 281;}
(a ...z) (a ...z 0...9) * { return 282;}
```

Conceptos de tokens, patrones y lexemas

Si tuviéramos como texto de entrada el siguiente:

95.7 99 cont

El analizador léxico intenta leer el lexema más grande; el 95 encaja con el primer patrón, pero sigue, al encontrarse el punto, se da cuenta de que también encaja con el segundo patrón, entonces como este es más grande, toma la acción del segundo patrón, return NUM_REAL. El 99 coincide con el patrón NUM_ENT, y la palabra con ID. Los espacios en blanco no coinciden con ningún patrón, y veremos más adelante como tratarlos.

En vez de trabajar con los números 280, 281, 282, se definen mnemotécnicos.

```
# define NUM_ENT   280
# define NUM_REAL   281
# define NUM_ID   282

(“ ”\t \n)
(0 ... 9) +                  {return NUM_ENT;}
(0 ... 9) *. (0 ... 9) +    {return NUM_REAL;}
(a ... z) (a ... z 0 ... 9)* {return ID;}
```

Las palabras que entran por el patrón (“ ”\t \n) no tienen acción asociada, por lo que , por defecto, se consideran meros espaciadores.

En PCLEX le introducimos una especificación parecida a la anterior y a la salida obtendremos un analizador lexicográfico en C.

Hay tres formas para construir un analizador lexicográfico.

- Ad hoc (a pelo)
- Autómatas finitos (como en teoría de autómatas y lenguajes formales)
- Metacompilador (más fácil) : Le pasamos pares (exp regular, {acción}) . El metacompilador genera todos los autómatas finitos, los convierte a autómata finito determinista, y lo implementa en C. El programa C así generado se compila y se genera un ejecutable que es el análisis léxico de nuestro lenguaje.

El generador de analizadores lexicográfico: PCLEX

Se han desarrollado algunas herramientas para construir analizadores léxicos a partir de notaciones de propósito especial basadas en expresiones regulares.

En esta sección se describe una herramienta concreta, llamada LEX, muy utilizada en la especificación de analizadores léxicos para varios lenguajes. Esta herramienta se denomina PCLEX, y la especificación de su entrada, lenguaje LEX. El estudio de una herramienta existente permitirá mostrar cómo, utilizando expresiones regulares, se puede combinar la especificación de patrones con acciones, por ejemplo, haciendo entradas de una tabla de símbolos cuya ejecución se pueda pedir a un analizador léxico.

Las reglas de conversión son de la forma:

p_1 {*acción*$_1$}
p_2 {*acción*$_2$}
... ...
p_n {*acción*$_n$}

donde p_i es una expresión regular y cada *acción* es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando el patrón p_i concuerda con un lexema. En LEX, las acciones se escriben en C, en general, sin embargo, pueden estar en cualquier lenguaje de implantación.

Un analizador léxico creado por LEX se comporta en sincronía con un analizador sintáctico como sigue. Cuando es activado por el analizador sintáctico, el analizador léxico crea una función llamada yylex, que una vez llamada, comienza a leer la entrada, un carácter a la vez, hasta que encuentre el mayor prefijo de la cadena que concuerde con una de las expresiones regulares p_i. Entonces, ejecuta la *acción*$_i$. Generalmente, *acción*$_i$ devolverá el control al analizador sintáctico. Sin embargo, si no lo hace, el analizador léxico se dispone a encontrar más lexemas, hasta que una acción hace que el control regrese al analizador sintáctico. La búsqueda repetida de lexemas hasta encontrar una instrucción **return** explícita permite al analizador léxico procesar espacios en blanco y comentarios de manera apropiada.

El analizador léxico devuelve un único lexema al analizador sintáctico que estará almacenado en la variable yytext. Si queremos retornar más información además del token, podemos usar la variable global yylval.

Los programas que se obtienen en LEX son muy grandes, (aunque muy rápidos también), lo cual a veces resulta perjudicial. Aunque su ventaja principal es que permite hacer analizadores complejos con bastante rapidez.

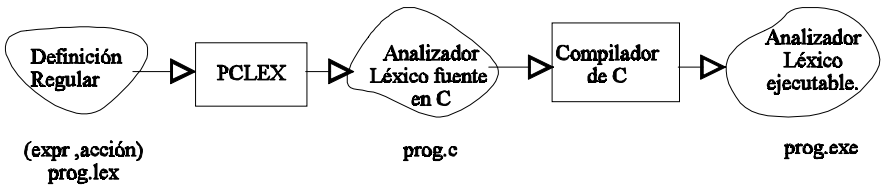
LEX tiene su propio lenguaje, al que llamaremos LEX.

El generador de analizadores lexicográfico: PCLEX

El lenguaje LEX permite expresar expresiones regulares, y la acción a tomar al encontrar cada una de ellas.

Pasos para crear un analizador léxico:

- Construir el fuente en LEX.
- Compilarlo con LEX. Se obtiene un fuente en C. Algunas veces hay que efectuar modificaciones directas en este código.
- Compilarlo con un compilador C.



Si ejecutamos prog.exe se quedará esperando a que le introduzca datos por teclado para analizarlos hasta que le introduzcamos ctrl+z. El programa va buscando entre los patrones si coincide con el lexema que le hemos metido. Cuando reconoce uno ejecuta la acción que tiene asociada.

Ej:

```
[0-9] + {printf("numero");}
[A-Z]+ {printf("palabra");}
```

tecleamos : HOLA 23 ^z	Cuando un lexema caza con un patrón, el PCLEX me cede el control.
Salida : palabranumero	Reconoce HOLA, ejecuta la acción asociada y después sigue reconociendo lexemas.

c:>

Si queremos meter como entrada un texto que no proceda de la entrada standard, sino que proceda de un fichero lo que se hace es redirigir la entrada.

```
prog < file.pas > salida.txt
```

< file.pas	redirige la entrada
> salida.txt	redirige la salida.

El generador de analizadores lexicográfico: PCLEX

✯ El formato de un programa LEX es:

```
Área de definiciones LEX
%%                    /* es lo único obligatorio en todo el programa */
Área de Reglas
%%
Área de funciones
```

El mínimo programa que se puede construir en LEX es:

```
%%
```

En el área de reglas vamos a definir los patrones que se quieren buscar a la entrada, y al lado de tales expresiones regulares, se detallan (en C) las acciones a ejecutar tras encontrar una cadena que se adapte al patrón indicado.

En LEX, si una cadena de entrada no encaja con ningún patrón, la acción que se toma es escribir tal entrada en la salida. %% copia la entrada en la salida.

El ejecutable que se obtiene usa la salida standard.

✯ Premisas de LEX para reconocer lexemas

1. LEX toma siempre el lexema más largo posible.
2. En caso de conflicto toma siempre el patrón que aparezca en primera posición.

Como consecuencia de la 2ª premisa: las palabras reservadas se colocan siempre antes que el patrón de identificador de usuario.

```
"TYPE"
"VAR"
[A-Z][A-Z0-9]*
```

Si se introduce el lexema "VAR" ,en este ejemplo hay un conflicto, (ya que puede entrar tanto por el patrón segundo, como por el patrón tercero). Entonces toma el patrón que aparece antes, en nuestro ejemplo sería "VAR". Y reconoce el lexema como palabra reservada.

Si cambio el orden:

```
"TYPE"
[A-Z][A-Z0-9]*
"VAR"
```

Sigue habiendo un conflicto, y esta vez entraría por el patrón de identificador de usuario, nunca se reconocería como una palabra reservada

Las premisas se cumplen en ese orden.

El generador de analizadores lexicográfico: PCLEX

✯ Caracteres especiales de LEX

“ : Sirve para encerrar cualquier cadena de literales. Por regla general no es necesario encerrar los literales entre comillas a no ser que incluyan símbolos especiales.

“(*” Comentario de MODULA2.

\ : Hace literal al siguiente carácter, excepto : \n, \t.

nº octal Indica el carácter cuyo valor ASCII es *nº octal*

Ej: \012 Reconoce el carácter decimal 10 que es el LF

PCLEX tiene el problema de que si se le pone un número ASCII mayor de 128 se bloquea.

[ ] : Indican clases de caracteres, o sea uno de los caracteres que encierra.

[abc] Indica, o la “a”, o la “b”, o la “c”. ([abc] ≡ (a|b|c))

Permite en su interior el uso de:

- : indica rango.

[A-Z0-9] de la “A” a la “Z” o de “0” a “9”.

^ : indica complementación (cuando aparece al comienzo, justo detrás de “[”)

[^abc] Cualquier carácter excepto la “a”, la “b” o la “c”.

[^A-Z] Cualquier carácter excepto de la “A” a la “Z”.

? : Lo que precede es opcional.

a? ≡ a| ε

[A-Z]? Cualquier letra de la “A” a la “Z” o bien ε.

a?b ≡ ab | εb.

. : Representa a cualquier carácter excepto el retorno de carro (\n). Es muy interesante porque nos permite recoger cualquier otro carácter, pero solo entra uno.

| : Indica opción. (OR).

a|b	a o b.
. | \n	Por aquí encaja cualquier carácter.
(. | \n)*	Por aquí entra el programa entero, y como LEX tiene la premisa del lexema más largo, ignorará cualquier otro patrón. Nos daría un desbordamiento. No hacerlo nunca.

***** : Indica repetición 0 o más veces de lo que le precede.

+ : Indica repetición 1 o más veces de lo que le precede.

( ) :Indica agrupación (igual que en las expresiones aritméticas).

{ } : Indica rango de repetición. También nos permite crear expresiones regulares, y asignarle un nombre.

a{1,5} ≡ aa?a?a?a?a? (Es el * restringido)

★ **Caracteres de sensibilidad al contexto**

$: El patrón que le precede solo se reconoce si está al final de la línea. No incluye \n como parte del lexema.

(a|b|cd)$ a \n, o b \n, o cd \n

^ : Fuera de los corchetes indica que el patrón que le sucede sólo se reconoce si está al comienzo de la línea

^"casa"	"casa"
"casa"	^"casa"

Ver la importancia de las dos premisas de LEX: En el segundo ejemplo no entraría nunca por ^"casa". Los casos especiales se ponen siempre antes.

/ : Reconoce el patrón que le precede si y sólo si es prefijo de una secuencia simple como la que le sucede.

ab/c

ejemplo el lexema 'abcd'. Por ese patrón entra 'ab' porque está sucedido de 'c', sino no entraría (c no entra).

ab/c+ Esto es un error. No se permite 'c+' detrás, porque no es simple.

<id n> : Permiten expresar condiciones START. Se declaran en el área de definiciones:

% START id_1, $id_{2,...}$

Se activan en las acciones (código C asociado a cada patrón), con:

BEGIN id_i ;

Cuando un patrón está precedido por:

<id_i>

donde id_i es una condición START, el patrón sólo se reconoce si está activa la condición START.

El generador de analizadores lexicográfico: PCLEX

Ejemplo : Escribe el nombre de todos los procedimientos y funciones de un programa MODULA-2.

```
% START PROC
%%
“PROCEDURE” {BEGIN PROC;}
<PROC> [a-zA-Z][a-zA-Z0-9]* {printf (“-------”);
                             BEGIN 0 ;}
```

Cuando un lexema entra por un patrón, la variable yytext contiene ese lexema, de forma que en el printf del ejemplo podríamos poner:

```
printf (“%s\n”, yytext);
```

El núcleo básico de lo generado es una función yylex que se encarga de buscar un lexema y ejecutar su acción asociada. Así sucesivamente hasta que en una de las acciones se encuentre un return, o se acabe la entrada.

La zona de definiciones tiene tres utilidades fundamentales:

a) Poder dar un nombre a los patrones más frecuentes.
b) Poner código C que será global a todo el programa.
c) Para definir las condiciones START.

En la zona de definiciones podemos crear expresiones regulares auxiliares. Por ejemplo:

```
D       [0-9]
L       [a-zA-Z]
%%
{D}+
{L}({L}|{D})*
```

Todo se debe comenzar en la primera columna. Si se comienza algo no en la primera columna, se pasa directamente al programa C generado por LEX.

Así podemos crear definiciones de variables, etc. Sin embargo, para ello se recomienda poner, ya que de esta forma se pueden poner incluso directivas del procesador, que deben comenzar obligatoriamente en la primera columna.

```
%{
definiciones C.
%}
```

Antes del primer %%, con lo cual la definición será global.

Poner %{ y %} al principio es igual que poner código C al final (en la zona de funciones).

El generador de analizadores lexicográfico: PCLEX

En MS-DOS cuando se compila no hay función main. Para ponerlo, en el área de funciones basta con poner:

```
void main()
{ yylex();
};
```

El main tiene que tener una llamada al analizador lexicográfico mediante la función yylex. El analizador sintáctico, cada vez que pida token tiene que hacer una llamada a yylex, y el yylex tiene que hacer un return para devolverselo al analizador sintáctico. Ahora no hará return hasta que no lo enlacemos con el YACC para que el analizador sintáctico pueda ir construyendo el árbol sintáctico.

Ejemplo: Programa que me dice cuantas veces aparece la variable casa.

```
%{
int cont=0;
%}
%%
“casa”  {cont ++;}
. | \n    {;}
%%
void main(){
        yylex();
        printf(“casa aparece %d veces”, cont);
}
```

✯ **Funciones y variables que nos suministra PCLEX**

yylex ( ) : Analizador lexicográfico.

yytext : Coincide con el lexema actual

yyleng : Longitud del lexema actual.

No utilizar nombres de funciones que empiecen por yy------, y tampoco que tengan un solo carácter.

yylval : es una variable global (inicialmente de tipo entero), que permite retornar información adicional. Nos permite comunicarnos con el sintáctico.

yyerror ( ) : Es una función que se encarga de emitir y controlar errores (saca mensajes de error por pantalla).

El generador de analizadores lexicográfico: PCLEX

yywrap ( ) : Se llama a esta función una vez que se ha leído todo el texto de entrada. Se puede usar para visualizar tablas de resúmenes.
Es una macro que hay que redefinir. Para ello hay que borrarla antes. ¿Cómo se borra una definición previa? - con la directiva #undef.
Yywrap es llamada por yylex al encontrar un EOF. Puede retornar dos valores:
- Falso (0): es que el EOF que se ha encontrado no es el verdadero y tiene que seguir leyendo. Se supone que se suministrará de alguna forma más texto de entrada.
- True (1) : El EOF encontrado es el definitivo y para.

yyless(n) : Se queda con los n primeros caracteres del lexema actual. El resto los devuelve a la entrada .yyleng se modifica.

Ejemplo: Estos tres patrones son equivalentes.

```
abc*$
abc* / \n
abc* \n          {yyless(yyleng-1);}
```

input ( ) : Consume el siguiente carácter de la entrada y lo añade al lexema actual

```
%%
abc     { printf (“%s”, yytext);
          input( );
          printf(“%s”, yytext);}
```

Introducimos como entrada : abcde
El lexema antes del input (en el primer printf) sería ‘abc’, después del input (en el segundo printf) será abcd.

output (c): Emite el carácter c por la salida standard.

unput © : Coloca el carácter c al comienzo de la entrada
Ejemplo:

```
%%
abc     {printf (“%s”,yytext);}
tal
```

La acción que tiene asociada el patrón ‘abc’, es la acción por defecto, sino la pongo sale lo mismo. ECHO es una macro que nos suministra PCLEX.

Otra forma de ponerlo

```
%%
abc     {ECHO; unput(‘t’);}
tal     {ECHO;}
```

Si la entrada es “abcal”, Los tres primeros caracteres del lexema coincide con el primer patrón. Después nos queda en la entrada el lexema”al“, pero como se ha hecho un unput(‘t’), el lexema que queda es ‘tal’, que coincide con el segundo patrón.

ECHO : Macro que copia la entrada en la salida (es la acción que hace por defecto).

yymore ( ): Concatenar el siguiente lexema al contenido actual de yytext. Sirve para reconocer los literales entrecomillados.

```
\" [^ \"] * \"    Lexemas que empiece por comillas, cualquier cosa, y termine en
                  comillas
“Hola”
“Ho;*”
“Hola,\”tío\” “        Da problema porque la primera comilla de “tío”, la
                       considera como las comillas de cierre.
```

¿Que hay que hacer para que reconozca el lexema entero?

```
\" [^ \"] * \"    {if (yytext[yyleng-1]==’\\’)
                        yymore();
                   else
                        input();}
```

Lo que se ponga detrás de yymore nunca se ejecutará, ya que yymore es una especie de GOTO.

REJECT : Rechaza el lexema actual y busca otro patrón. REJECT le dice al LEX que el lexema encontrado no corresponde realmente a la expresión regular en curso, y que busque la siguiente expresión regular a que corresponda. La macro REJECT debe ser lo último de una acción. Si hay algo detrás, no se ejecuta.
Por ejemplo: Para buscar cuantas veces aparecen las letras ‘teclado’ y ‘lado’, se haría:

```
%{
int t=0, l=0;
%}
%%
teclado        {t ++; REJECT;}
lado           {l ++;}
```

Si la entrada es ‘teclado ‘^Z
1.- Entra por el lexema más largo que sería ‘teclado’ y ejecuta la acción asociada. Incrementa la variable ‘t’; y rechaza el lexema actual.
2.- Saca por pantalla ‘tec’ que es la acción por defecto si no coincide el lexema con ningún patrón.
3.- El lexema ‘lado’ coincide con el segundo patrón y ejecuta la acción asociada. Incrementa la variable ‘l’