Curso: Compiladores
Tema 2. Análisis léxico
(Parte 3)

M.C. Laura Sandoval Montaño

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

En la actualidad, existen varios generadores de analizadores léxicos como lex y Jlex.

En este curso nos enfocaremos al uso de lex (en Unix) o flex (en Linux). Este último, se distribuye como parte del paquete Gnu que produce la Free Software Foundation y también se encuentra disponible gratuitamente en muchos sitios de Internet.

Lex es un compilador que genera automáticamente analizadores léxicos; el programa fuente a compilar debe estar escrito en lenguaje lex, el cual es un lenguaje orientado a acciones. Este tipo de lenguajes identifica un conjunto de **patrones**, y realiza una tarea específica para cada cadena que pertenezca a cierto patrón.

Con esta herramienta podremos definir un patrón por cada clase de componente léxico que deseemos que nuestro analizador léxico reconozca. Como ya vimos, cada clase de componente léxico está definido por una **expresión regular**, por lo que esa expresión regular será un patrón en el programa fuente de lex.

En otras palabras, lex es un programa que toma como entrada un archivo de texto, con extensión .l, que contiene expresiones regulares, junto con las acciones que se tomarán cuando se iguale cada expresión.

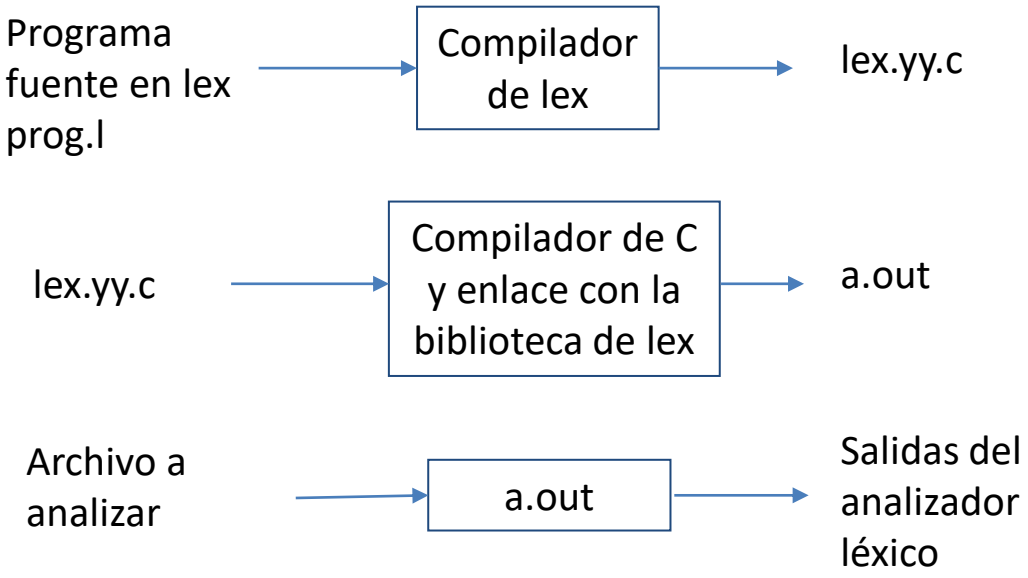
Este programa produce un archivo de salida que contiene un código fuente en C que define un procedimiento yylex() que es una implementación de la tabla de transición del AFD que identificará las cadenas del lenguaje definido a su vez, por las expresiones regulares del archivo de entrada.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

El archivo de salida de lex, denominado por lo general **lex.yy.c** o **lexyy.c**, se compila y liga entonces a un programa principal para obtener un programa ejecutable.

Las acciones o tareas asociadas al reconocimiento de una cadena que cumple con un patrón, son partes de código en C y se transfieren directamente a lex.yy.c.

De manera esquemática, el proceso para obtener un analizador léxico utilizando la herramienta lex, se muestra a continuación.



Proceso para obtener un analizador léxico con lex.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Especificaciones en lex.

Un programa en lex consta de tres partes:

declaraciones
%%
reglas de traducción (acciones)
%%
procedimientos auxiliares

La sección de *declaraciones* incluye declaración de variables, constantes y definición de expresiones regulares.

Las *reglas de traducción* son proposiciones de la forma:

exp1 {acción1}
exp2 {acción2}
⋮
expn {acciónn}

Donde *expi* es una expresión regular y cada *acción* es código en C que se realizará cuando la cadena pertenezca al lenguaje definido por dicha expresión regular.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

La tercera sección contiene todos los procedimientos auxiliares que pueden necesitar las acciones. Si se incluye la función main( ) en esta sección, ésta deberá invocar a la función yylex(), que es, como lo mencioné, generada por el compilador lex. Si no definimos la función main() en esta sección, el compilador lex la genera.

Las instrucciones de las zonas de declaraciones y de reglas de traducción, deberán iniciar en la columna 1; al igual que los %% que dividen las secciones. Las líneas donde se encuentren los %% deberán contener exclusivamente dichos caracteres.

Ejemplo 2.6.1 Realicemos un primer programa en lex/flex que reconozca a los identificadores del lenguaje C:

```
dig     [0-9]
let     [A-Za-z_]
ident   {let}({let}|{dig})*
%%
{ident}   {printf(" %s es un identificador\n", yytext);}
%%
main( )
{
    yylex();
}
```

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Ahora revisemos sección por sección del programa.

- Sección de declaraciones.

dig [0-9]	**dig** es el nombre de la expresión regular que define a los dígitos del 0 al 9.
let [A-Za-z_]	**let** es el nombre de la expresión regular que define a las letras mayúsculas y minúsculas, también incluye al guion bajo.
ident {let}({let}|{dig})*	**ident** es el nombre de la expresión regular que define al nombre de los identificadores. let y dig van encerrados entre llaves { } porque son nombres de expresiones regulares antes definidas.

Las líneas de declaraciones de expresiones regulares, tienen la siguiente sintaxis:

nombre *expresión regular*

donde *nombre* es el nombre de la expresión regular anotada al frente. El *nombre* debe tener las mismas características que el nombre de los identificadores de C.

expresión regular debe ocupar la notación lex. Más adelante veremos a detalle cuál es dicha notación.

Entre el *nombre* y la *expresión regular* debe haber al menos un espacio. NO USAR TABULADORES. Tampoco dejar espacios después de la expresión regular.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

- Sección de reglas de traducción.

En el programa sólo hay una sola instrucción en esta sección:

```
{ident}   {printf(" %s es un identificador\n", yytext);}
```

Observemos que la expresión regular que define el nombre de los identificadores, ya la definimos en la sección de declaraciones y le dimos el nombre de “ident”; por lo que en esta sección se utiliza sólo su nombre, el cual debe ir encerrado entre llaves { }.

Si en el archivo de entrada encuentra una cadena que pertenece al lenguaje de los nombres de identificadores, la apunta con la variable (char *)**yytext** (que el mismo compilador la define para este propósito) y entonces realiza la acción:

```
{printf(" %s es un identificador\n", yytext);}
```

Nótese que está escrita en lenguaje C. Al ser una sola instrucción, estrictamente no requiere de las llaves { } que la encierren, pero es buena práctica hacerlo.

Se puede poner directamente la expresión regular de los nombres de identificadores:

```
{let}({let}|{dig})*    {printf(" %s es un identificador\n", yytext);}
```

Pero las buenas prácticas nos recomiendan usar nombres de expresiones regulares ya definidos.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

- Sección de procedimientos auxiliares.

El programa sólo tiene un procedimiento, la función main():

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

Observemos que lo único que realiza es llamar a la función yylex(), la cual fue creada por el compilador lex y es la función que realiza el reconocimiento de los componentes léxicos a través de la implementación de su correspondiente AFD.

Obtención del programa del analizador léxico.

El programa escrito en lenguaje lex, se compila con flex (fast lex) bajo plataforma Linux:

$flex prog.l

Con esto genera el programa en C con nombre lex.yy.c, el cual se compila en C enlazando las bibliotecas de flex:

$gcc lex.yy.c –lfl

Finalmente se ejecuta el archivo a.out obtenido:

$./a.out

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Al ejecutar el archivo a.out, el cursor estará esperando a que se le dé una cadena por la entrada estándar, en este caso el teclado. Esto es porque en el programa no se le indicó que tomara como entrada un archivo. La salida será la salida estándar, o sea, el monitor; se puede indicar que la salida sea hacia un archivo.

Mientras se esté ejecutando el archivo, estará solicitando cadenas, entonces para indicarle que ya no le daremos cadenas, se le da *ctrl C* o *ctrl D*.

Las cadenas que no reconoce como identificadores, hace un eco a la salida.

Ejemplo 2.6.2 Veamos el programa en donde las cadenas se leen desde un archivo:

```
dig     [0-9]
let     [A-Za-z_]
ident   {let}({let}|{dig})*
%%
{ident}   {printf(" %s es un identificador\n", yytext);}
%%
main(int argc, char *argv[])
{
    yyin = fopen(argv[1],"r");
    yylex();
}
```

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

En este caso, cuando se ejecute, en la línea de comandos se debe poner el nombre del archivo a analizar:

$./a.out datos.txt

Esto es porque la función main() tiene argumentos, y redirigimos la entrada al archivo apuntándolo por **yyin**, la cual es otra variable definida por el compilador para que apunte a un archivo de entrada.

```
main(int argc, char *argv[])
{
    yyin = fopen(argv[1],"r");

    yylex();
}
```

argc indica cuántas cadenas hay en la línea de comandos al ejecutar el programa.
argv es un arreglo de apuntadores a cadenas, contiene todas las cadenas dadas en la línea de comandos al ejecutar el programa.

Hacemos que la entrada sea el archivo cuyo nombre es la segunda cadena de la línea de comandos dada. Lo debe apuntar **yyin**

Se invoca a la función **yylex()** que es la que hace el reconocimiento de las cadenas de entrada.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Ejemplo 2.6.3 Ahora, agreguemos al ejemplo anterior, las instrucciones para dirigir la salida a un archivo. Además comentaremos el programa.

```
%{
  /* Programa que lee un programa y hace el reconocimiento de
       identificadores del lenguaje C
  */

  FILE *archSal;
%}
dig     [0-9]
let     [A-Za-z_]
ident   {let}({let}|{dig})*
%%
{ident}   {fprintf(archSal," %s es un identificador\n", yytext);}
%%
main(int argc, char *argv[])
{
    yyin = fopen(argv[1],"r");
    archSal = fopen("salida.txt","w");
    yylex();
    close(archSal);
}
```

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Observamos nuevos elementos:

Al inicio del program hay líneas escritas en lenguaje C que están encerradas entre una línea inicial con **%{** y una línea final con **%}**. Todo lo que se ponga entre estas líneas es transcrito tal cual al programa yy.lex.c :

```
%{
  /* Programa que lee un programa y hace el reconocimiento de
       identificadores del lenguaje C
  */

  FILE *archSal;
%}
```

Observamos que primero hay un comentario indicando la función del programa, después la declaración de un apuntador a archivo nombrado archSal, el cual será utilizado para dirigir la salida a un archivo.

Las líneas con **%{** y **%}** deben iniciar en la **columna 1** y solo deben contar con dichos caracteres.

Para poner comentarios en lex se pueden usar /* …*/ o // pero deben ocupar líneas nuevas.

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Para poner comentarios en lex se pueden usar /* */ o // pero deben ocupar líneas nuevas. Por ejemplo, en la sección de declaraciones:

```
/* se definen los dígitos */
dig     [0-9]
// se definen las letras
let     [A-Za-z_]
ident   {let}({let}|{dig})*   // se define nombre de identificador
```

Aquí marcaría error porque no reconocería la regla

En la sección de reglas de traducción, también deben estar en líneas nuevas y **no en la columna 1**. Los comentarios pueden estar como parte del código a realizar escrito en C:

```
%%
  /* única acción  */
{ident}   { /* reconoce un identificador */
            fprintf(archSal," %s es un identificador\n", yytext); // se imprime el identificador
          }
%%
```

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Notación en lex para escribir expresiones regulares.

La notación es muy similar a la usada en las clases anteriores y las que usaron en su curso de Lenguajes formales y autómatas. Por lo que haremos algunas precisiones.

Los operadores que se emplean en las expresiones regulares son:

\ ” . ^ $ [ ] * + { } | ( ) / ?

además del guion **–** , cuando está dentro de los corchetes para definir un rango.

También se les llaman *metacaracteres* debido a que pueden pertenecer a una cadena de una clase de componentes léxicos.

Es indispensable identificarlos para que en el momento de generar la expresión regular y si alguno de estos operadores son parte de la cadena a reconocer hay que ‘escaparlos’ anteponiéndole \. Veamos algunos casos:

Ejercicio 2.6.1 Definir los operadores aritméticos de C con expresiones regulares en lex.

Respuesta 1:

ope_arit \+|-|*|\/|% ⟹ Hay que ‘escapar’ los operadores +, * y /

Respuesta 2:

ope_arit [+\-*/%] ⟹ Hay que ‘escapar’ sólo el operador -

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

La siguiente tabla presenta en orden decreciente de precedencia, la conformación de expresiones regulares en lex. (***c*** representa cualquier carácter simple, ***r*** una expresión regular, y ***s*** una cadena)

Expresión	Significado	Ejemplo
c	cualquier carácter *c* que no sea operador	a
c	el carácter *c* literalmente	\+
″*s*″	la cadena *s* literalmente	″adios…″
.	cualquier carácter excepto de nueva línea	a.*b
^	el comienzo de línea	^int
$	el fin de línea	a;$
[*s*]	cualquier carácter en s excepto -	[*+ab]
[^*s*]	cualquier carácter que no esté en *s*	[^1234567890]
*r**	cero o más ocurrencias de *r*	a*
r+	una o más ocurrencias de *r*	a+
r?	cero o una ocurrencia de *r*	a?
r{*m*,*n*}	*m* a *n* ocurrencias de *r*	a{1,8}
r_1r_2	r_1 y entonces r_2	ab
$r_1 \mid r_2$	r_1 o r_2	a | b
(*r*)	*r*	(a | b)
r_1 / r_2	r_1 cuando va seguida de r_2	a/--

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Hagamos algunos ejercicios para definir expresiones regulares en lex.

Ejercicio 2.6.2 Escribe la expresión regular que defina los siguientes lenguajes:

a) Los números reales con parte entera y de 0 a 5 dígitos decimales; los números reales deben tener forzosamente el ‘.’

Respuesta.

num_real [0-9]+\.[0-9]{0,5}

b) Nombres de identificadores formados por letras mayúsculas y minúsculas; que inicien con letra mayúscula, le sigan puras minúsculas y termine con letra mayúscula. El tamaño del nombre es mínimo de 4 y máximo de 8

Respuesta.

IdentiF [A-Z][a-z]{2,6}[A-Z]

c) Los operadores relacionales del lenguaje C

Respuesta.

op_rel (>|<)=?|(!|=)=

2. Análisis léxico
2.6 Generación automática de analizadores léxicos

Actividad 2.6.1 Elabora diferentes expresiones regulares en flex que ocupen los diferentes operadores. Inclúyelos en un programa escrito en flex, obtén el ejecutable y prueba cada expresión regular.

Actividad 2.6.2 Elabora un programa en flex que reconozca los nombres de identificadores en C, así como todos los operadores de relación y de corrimiento de C; como salida debe indicar qué componente léxico encontró o si la cadena no es reconocida.