



EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO®

Instituto Tecnológico de Matamoros

Unidad 4. Analizador Léxico

Nombre de los estudiantes

Nicole Rodríguez González
Víctor Hugo Vázquez Gómez

Asignatura

Lenguajes y Autómatas I

Carrera

INGENIERÍA EN SISTEMAS COMPUTACIONALES

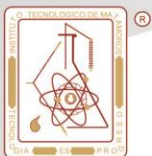
Profesor

Hernández Compeán María Guadalupe

H. Matamoros Tamaulipas

25 marzo 2020

Excelencia en Educación Tecnológica®
Tecnología es progreso®



Índice

Contenido

Índice	1
Introducción	2
Funciones del analizador léxico	3
Componentes léxicos, patrones y lexemas.....	4
Creación de Tabla de Tokens	6
Errores léxicos	7
Generadores de analizadores Léxicos.....	8
Aproximaciones para construir un analizador lexicográfico.....	8
Ejemplo	9
Aplicaciones (Caso de estudio).....	15
Conclusión grupal.....	16
Referencias bibliográficas	16

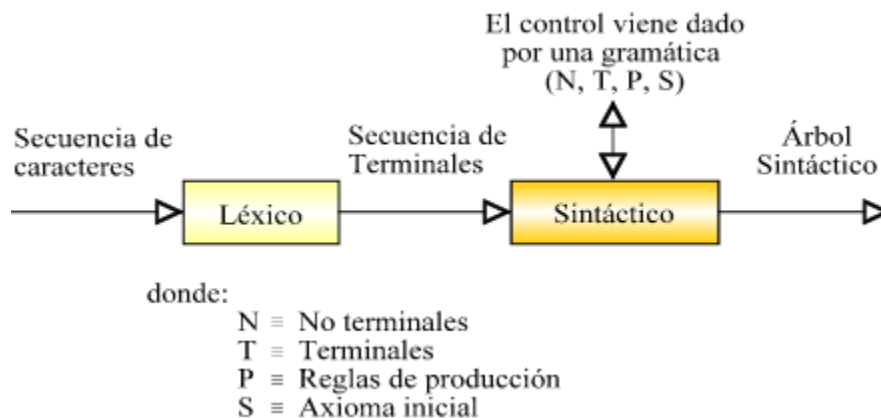
Introducción

En este reporte nos daremos a la tarea de investigar la primera fase de un compilador, el analizador léxico. Intentaremos comprender las técnicas utilizadas para construir dichos analizadores, investigaremos también sus aplicaciones, qué tipo de errores podemos cometer a la hora de realizar el analizador léxico.

Para empezar a estudiar el tema tenemos que tener claro el concepto de qué es un analizador léxico. Para eso nos basamos en el libro de compiladores escrito por Sergio Gálvez Rojas y Miguel Ángel Mora Mata. En él nos dice:

El analizador léxico es el que se encarga de buscar componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones.

La entrada en sí es una secuencia de caracteres que se encuentra con una codificación tal como la UTF-8 o la ASCII. El analizador lo que hace con esta secuencia es dividir en palabras con significado propio para después convertirlas en una secuencia que el analizador sintáctico pueda entender.

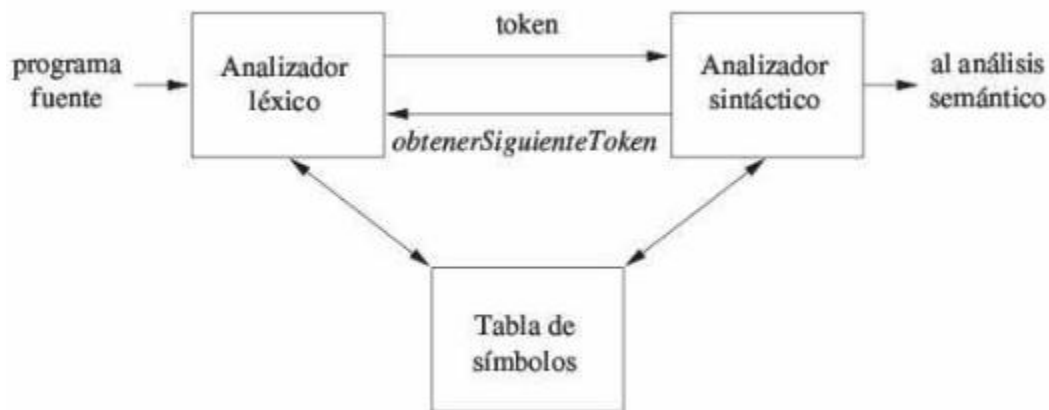


El analizador léxico entonces lo que hace es reconocer las palabras en función a una gramática regular dado el alfabeto Σ donde la gramática son los símbolos sobre los que el ordenador trabaja (que son el conjunto de símbolos terminales) mientras que sus no terminales son las categorías léxicas en que se integran las distintas secuencias de caracteres.

Funciones del analizador léxico

La función del analizador léxico es la de leer los caracteres de la entrada del programa fuente, agruparlos en lexemas y producir una salida de tokens para cada lexema en el programa. El flujo de tokens se envía al analizador sintáctico para que lo analice.

Cuando el analizador léxico descubre un lexema que constituye a un identificador, debe introducir ese lexema a la tabla de símbolos. En algunos casos, el analizador léxico puede leer la información relacionada con el tipo de información de la tabla de símbolos, como ayuda para determinar el token apropiado que debe pasar al analizador sintáctico.



En esta imagen se realizan dichas interacciones. Por lo regular la interacción se implementa haciendo que el analizador sintáctico llame al analizador léxico. La llamada hace que el analizador léxico lea los caracteres de la entrada para que identifique el siguiente lexema y mandarlo como un token, el cual se lo da al analizador sintáctico.

El analizador léxico es la parte del compilador que lee el texto de origen, y como tal realiza otras tareas además de las ya mencionadas. Tales como:

- Eliminar comentarios.
- Eliminar espacios en blanco (como el tabulador, nueva línea u otros caracteres que se usan para separar tokens de entrada).
- Correlacionar mensajes de error generados por el compilador con el programa fuente.

Algunas veces, los analizadores léxicos se dividen en una cascada de dos procesos:

- El *escaneo*, que consiste en los procesos simples que no requieren la determinación de tokens de entrada, como la eliminación de comentarios y la compactación de los caracteres de espacio en blanco consecutivos en uno solo.
- El propio análisis léxico es la porción más compleja, en donde escanear produce la secuencia de tokens como salida.

Componentes léxicos, patrones y lexemas

Desde un punto de vista muy general, podemos abstraer el programa que implementa un análisis léxico gráfico mediante una estructura como:

(Expresión regular)₁ {acción a ejecutar}₁
(Expresión regular)₂ {acción a ejecutar}₂
(Expresión regular)₃ {acción a ejecutar}₃
...
(Expresión regular)_n {acción a ejecutar}_n

donde cada acción a ejecutar es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando la secuencia de entrada coincida con la expresión regular. Normalmente esta acción suele finalizar con la devolución de una categoría léxica.

Al hablar sobre el análisis léxico, utilizamos tres términos distintos, pero relacionados:

- Un token es un par que consiste en un nombre de token y un valor de atributo opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico. A partir de este momento, en general escribiremos el nombre de un token en **negrita**. Con frecuencia nos referiremos a un token por su nombre.
- Un patrón es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.

- Un lexema es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

Una vez detectado que un grupo de caracteres coincide con un patrón, se considera que se ha detectado un lexema. A continuación, se le asocia el número de su categoría léxica, y dicho número o token se le pasa al sintáctico junto con información adicional, si fuera necesario.

Por ejemplo, si se necesita construir un analizador léxico que reconozca los números enteros, los números reales y los identificadores de usuario en minúsculas, se puede proponer una estructura como:

<u>Expresión Regular</u>	<u>Terminal asociado</u>
$(0 \dots 9)^+$	NUM_ENT
$(0 \dots 9)^* \cdot (0 \dots 9)^+$	NUM_REAL
$(a \dots z)(a \dots z 0 \dots 9)^*$	ID

Asociado a la categoría gramatical de número entero se tiene el token **NUM_ENT** que puede equivaler, p.ej. al número 280; asociado a la categoría gramatical número real se tiene el token **NUM_REAL** que equivale al número 281; y la categoría gramatical identificador de usuario tiene el token ID que equivale al número 282. Así, la estructura expresión regular-acción sería la siguiente:

$(0 \dots 9)^+$	{ return 280;}
$(0 \dots 9)^* \cdot (0 \dots 9)^+$	{ return 281;}
$(a \dots z)^*(a \dots z 0\dots 9)$	{ return 282;}
" "	{ }

De esta manera, un analizador léxico que obedeciera a esta estructura, si durante su ejecución se encuentra con la cadena:

95.7 99 cont

intentará leer el lexema más grande de forma que, aunque el texto "95" encaja con el primer patrón, el punto y los dígitos que le siguen ".7" hacen que el analizador decida reconocer "95.7" como un todo, en lugar de reconocer de manera independiente "95" por un lado y ".7" por otro; así se retorna el token **NUM_REAL**. Resulta evidente que un comportamiento distinto al expuesto sería una fuente de problemas. A continuación el patrón " " y la acción asociada permiten ignorar los espacios en blanco. El "99" coincide con el patrón de **NUM_ENT**, y la palabra "cont" con **ID**.

Creación de Tabla de Tokens

Existen diferentes tipos de tokens y a cada uno se le puede asociar un tipo y, en algunos casos, un valor. Los tokens se pueden agrupar en dos categorías:

Cadenas específicas, como las palabras reservadas (if, while, ...), signos de puntuación (., ,, =, ...), operadores aritméticos (+, *, ...) y lógicos (AND, OR, NOT, ...), etc. Habitualmente, las cadenas específicas no tienen asociado ningún valor, sólo su tipo.

Cadenas no específicas, como los identificadores o las constantes numéricas o de texto. Las cadenas no específicas siempre tienen tipo y valor. Por ejemplo, si dato es el nombre de una variable, el tipo del token será identificador y su valor será dato.

La tabla de tokens tiene dos funciones principales:

- Efectuar chequeos semánticos.
- Generación de código.

La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: nombre, tipo, dirección de localización, tamaño, etc. La gestión de la tabla de símbolos es muy importante, ya que consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica. Aunque también sirve para guardar información referente a los tipos creados por el usuario, tipos enumerados y, en general, a cualquier identificador creado por el usuario.

Consideraciones sobre la tabla de tokens

La tabla de tokens puede iniciarse con cierta información útil como:

- **Constantes:** PI, E, etc.
- **Funciones de librería:** EXP, LOG, etc.
- **Palabras reservadas:** Esto facilita el trabajo lexicográfico que tras reconocer un identificador lo busca en la tabla de tokens y si es la palabra reservada devuelve el token asociado.

Conforme van apareciendo nuevas declaraciones de identificadores, el analizador léxico, o el analizador sintáctico según la estrategia que sigamos, insertará nuevas entradas en la tabla de símbolos, evitando siempre la existencia de entradas repetidas.

La tabla de símbolos consta de una estructura llamada símbolo. Las operaciones que puede realizar son:

- **Crear:** Crea una tabla vacía.

- **Insertar:** Parte de una tabla de símbolo y de un nodo, lo que hace es añadir ese nodo a la cabeza de la tabla dado un par clave-valor.
- **Buscar:** Busca el nodo que contiene el nombre que le paso por parámetro, es decir, dada la clave de un elemento, encontrar su valor.
- **Imprimir:** Devuelve una lista con los valores que tiene los identificadores de usuario, es decir recorre la tabla de símbolos. Este procedimiento no es necesario, pero se añade por claridad, y a efectos de resumen y depuración
- **Cambio de valor:** Buscar el elemento y cambiar su valor.
- **Borrado:** Eliminar un elemento de la tabla.
- **Longitud de búsqueda** (o tiempo de acceso)

Procedimiento de creación:

El final de la tabla de símbolos se usa como una lista al revés. Cuando aparece {se añade un bloque y se pone el puntero al final de la lista. Cuando llega} se vacía la lista correspondiente al bloque que terminó, copiando al principio de la tabla.

Inserción: se añade a la lista del final de la tabla de símbolos, se corre el puntero y se aumenta en 1 el número de elementos de ese bloque.

Búsqueda: se busca en el bloque presente sólo (si es una declaración) o en él y sus antepasados en otro caso.

Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente. Por ejemplo, si la cadena **fi** se encuentra por primera vez en un programa en C en el siguiente contexto:

fi (a = f (x)) . . .

un analizador léxico no puede saber si **fi** es una palabra clave **if** mal escrita, o un identificador de una función no declarada. Como **fi** es un lexema válido para el token **id**, el analizador léxico debe regresar el token **id** al analizador sintáctico y dejar que alguna otra fase del compilador (quizá el analizador sintáctico en este caso) mande un error debido a la transposición de las letras.

Sin embargo, suponga que surge una situación en la cual el analizador léxico no puede proceder, ya que ninguno de los patrones para los tokens coincide con algún prefijo del resto de la entrada. La estrategia de recuperación más simple es la recuperación en “modo de pánico”. Eliminamos caracteres sucesivos del resto de la entrada, hasta que el analizador léxico pueda encontrar un token bien formado al principio de lo que haya quedado de entrada. Esta técnica de recuperación puede confundir al analizador sintáctico, pero en un entorno de computación interactivo, puede ser bastante adecuado.

Otras de las posibles acciones de recuperación de errores son:

1. Eliminar un carácter del resto de la entrada.
2. Insertar un carácter faltante en el resto de la entrada.
3. Sustituir un carácter por otro.
4. Transponer dos caracteres adyacentes.

Las transformaciones como éstas pueden probarse en un intento por reparar la entrada. La estrategia más sencilla es ver si un prefijo del resto de la entrada puede transformarse en un lexema válido mediante una transformación simple. Esta estrategia tiene sentido, ya que en la práctica la mayoría de los errores léxicos involucran a un solo carácter. Una estrategia de corrección más general es encontrar el menor número de transformaciones necesarias para convertir el programa fuente en uno que consista sólo de lexemas válidos, pero este método se considera demasiado costoso en la práctica como para que valga la pena realizarlo.

Generadores de analizadores Léxicos

Aproximaciones para construir un analizador lexicográfico

Hay tres mecanismos básicos para construir un analizador lexicográfico:

- Ad hoc. Consiste en la codificación de un programa reconocedor que no sigue los formalismos propios de la teoría de autómatas. Este tipo de construcciones es muy propenso a errores y difícil de mantener.
- Mediante la implementación manual de los autómatas finitos. Este mecanismo consiste en construir los patrones necesarios para cada categoría léxica, construir sus autómatas finitos individuales, fusionarlos por opcionalidad y, finalmente, implementar los autómatas resultantes. Aunque la construcción de analizadores mediante este método es sistemática y no propensa a errores, cualquier actualización de los patrones reconocedores implica la modificación del código que los implementa, por lo que el mantenimiento se hace muy costoso.
- Mediante un meta compilador. En este caso, se utiliza un programa especial que tiene como entradas pares de la forma (expresión regular, acción). El meta compilador 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 analizador léxico de nuestro lenguaje. Por supuesto, existen metas compiladores que generan código Java, Pascal, etc. en lugar de C.

Ejemplo

A continuación, vamos a usar Flex para crear un analizador léxico llamado Flex.

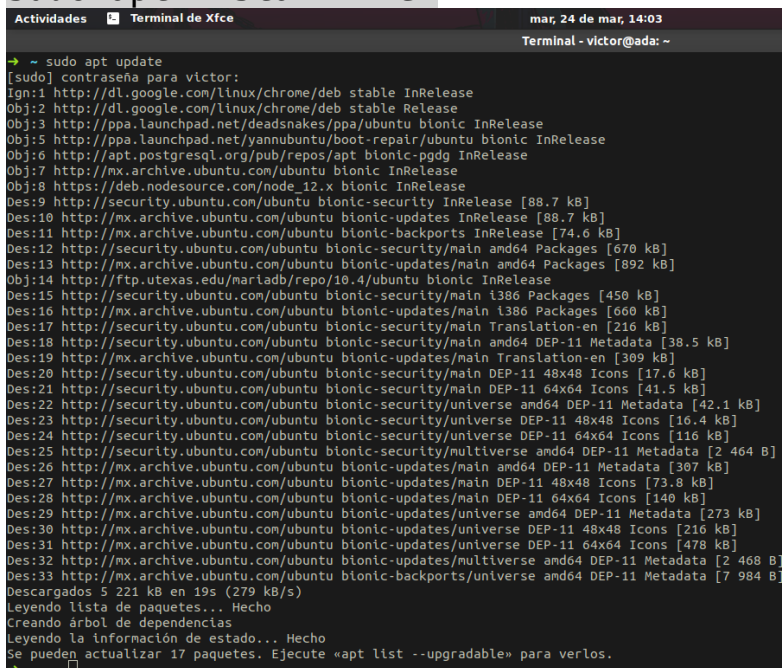
Flex (fast lexical analyzer generator) es una herramienta para crear analizadores léxicos. Escrito por Vern Paxson en C alrededor de 1987. Es usado junto al Berkley Yacc parser generator o el GNU bison parser generator. En sí Flex y Bison son más rápidos que Flex y Yacc juntos.

Bison produce un analizador del archivo de entrada que provee el usuario. La función **yylex()** es automáticamente generado por flex cuando se le da un archivo con extensión **.l** y esta función **yylex()** es esperada por el analizador para llamar para recuperar los tokens del archivo actual.

Vamos a hacer este ejercicio en Ubuntu:

Para ello primero tenemos que instalar flex en ubuntu.

```
sudo apt update
sudo apt install flex
```

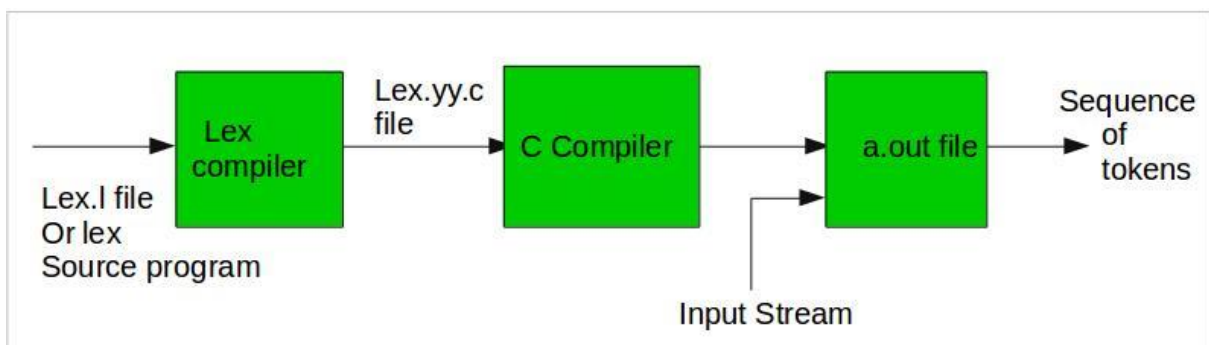


```
Actividades  Terminal de Xfce  mar, 24 de mar, 14:03
Terminal - victor@ada: ~

➜ ~ sudo apt update
[sudo] contraseña para victor:
Ign:1 http://dl.google.com/linux/chrome/deb stable InRelease
Obj:2 http://dl.google.com/linux/chrome/deb stable Release
Obj:3 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu bionic InRelease
Obj:5 http://ppa.launchpad.net/yannubuntu/boot-repair/ubuntu bionic InRelease
Obj:6 http://apt.postgresql.org/pub/repos/apt bionic-pgdg InRelease
Obj:7 http://mx.archive.ubuntu.com/ubuntu bionic InRelease
Obj:8 https://deb.nodesource.com/node_12.x bionic InRelease
Des:9 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Des:10 http://mx.archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Des:11 http://mx.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Des:12 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [670 kB]
Des:13 http://mx.archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [892 kB]
Obj:14 http://ftp.utexas.edu/mariadb/repos/10.4/ubuntu bionic InRelease
Des:15 http://security.ubuntu.com/ubuntu bionic-security/main i386 Packages [450 kB]
Des:16 http://mx.archive.ubuntu.com/ubuntu bionic-updates/main i386 Packages [660 kB]
Des:17 http://security.ubuntu.com/ubuntu bionic-security/main Translation-en [216 kB]
Des:18 http://security.ubuntu.com/ubuntu bionic-security/main amd64 DEP-11 Metadata [38.5 kB]
Des:19 http://mx.archive.ubuntu.com/ubuntu bionic-updates/main Translation-en [309 kB]
Des:20 http://security.ubuntu.com/ubuntu bionic-security/main DEP-11 48x48 Icons [17.6 kB]
Des:21 http://security.ubuntu.com/ubuntu bionic-security/main DEP-11 64x64 Icons [41.5 kB]
Des:22 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 DEP-11 Metadata [42.1 kB]
Des:23 http://security.ubuntu.com/ubuntu bionic-security/universe DEP-11 48x48 Icons [16.4 kB]
Des:24 http://security.ubuntu.com/ubuntu bionic-security/universe DEP-11 64x64 Icons [116 kB]
Des:25 http://security.ubuntu.com/ubuntu bionic-security/multiverse amd64 DEP-11 Metadata [2 464 B]
Des:26 http://mx.archive.ubuntu.com/ubuntu bionic-updates/main amd64 DEP-11 Metadata [307 kB]
Des:27 http://mx.archive.ubuntu.com/ubuntu bionic-updates/main DEP-11 48x48 Icons [73.8 kB]
Des:28 http://mx.archive.ubuntu.com/ubuntu bionic-updates/main DEP-11 64x64 Icons [140 kB]
Des:29 http://mx.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 DEP-11 Metadata [273 kB]
Des:30 http://mx.archive.ubuntu.com/ubuntu bionic-updates/universe DEP-11 48x48 Icons [216 kB]
Des:31 http://mx.archive.ubuntu.com/ubuntu bionic-updates/universe DEP-11 64x64 Icons [478 kB]
Des:32 http://mx.archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 DEP-11 Metadata [2 468 B]
Des:33 http://mx.archive.ubuntu.com/ubuntu bionic-backports/universe amd64 DEP-11 Metadata [7 984 B]
Descargados 5 221 kB en 19s (279 kB/s)
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se pueden actualizar 17 paquetes. Ejecute «apt list --upgradable» para verlos.
➜ ~
```

```
Actividades Terminal de Xfce mar, 24 de mar, 14:05
Terminal - sudo apt install flex
→ ~ sudo apt install flex
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Los paquetes indicados a continuación se instalaron de forma automática y ya no son necesarios.
 libgit2-26 libhttp-parser2.7.1 libqt5serialport5 libqt5sql5 libqt5sql5-sqlite libqt5xml5
Utilice «sudo apt autoremove» para eliminarlos.
Se instalarán los siguientes paquetes adicionales:
 libfl-dev libfl2
Paquetes sugeridos:
 bison flex-doc
Se instalarán los siguientes paquetes NUEVOS:
 flex libfl-dev libfl2
0 actualizados, 3 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
Se necesita descargar 334 kB de archivos.
Se utilizarán 1 113 kB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n] s
```

Ya que lo tenemos instalado veamos cómo es que funciona Flex con este diagrama:



- En el primer paso un archivo de entrada describe que el analizador léxico que se generará llamado **lex.l** está escrito en lenguaje lex. El compilador lex transforma el programa **lex.l** en C, en un archivo que siempre se llama **lex.yy.c**.
- En el segundo paso el compilador de C compila el archivo **lex.yy.c** en un archivo ejecutable llamado **a.out**.
- Por último, el archivo de salida **a.out** toma una secuencia de caracteres de entrada y produce una secuencia de tokens.

Estructura del programa:

En el archivo de entrada hay 3 secciones:

1. **La sección de definición:** Contiene la declaración de variables, definiciones regulares, constantes. En esta sección el texto está encerrados en "%{ }". Todo lo que esté dentro de esos brackets será copiado al

archivo **lex.yy.c**

Sintaxis:

```
%{  
  // Definiciones  
}%
```

2. **La sección de reglas:** Contiene una serie de reglas de la forma: *patron accion*. el patrón debe estar sin indentación y acción empieza en la misma línea que los {}. La regla es que estén encerrados en “%% %%”.

Sintaxis:

```
%%  
patron accion  
%%
```

La tabla de abajo muestra algunos de los patrones que coinciden.

Patrón	Puede coincidir con
[0-9]	todos los dígitos del 0 al 9
[0+9]	ya sea 0, + o 9
[0, 9]	ya sea 0, ‘,’ o 9
[0 9]	ya sea 0, ‘ ’ o 9
[-09]	ya sea -, 0 o 9
[-0-9]	ya sea - o todos los dígitos del 0 al 9
[0-9]+	uno o más dígitos entre 0 al 9
[^a]	todos los demás caracteres excepto a
[^A-Z]	todos los demás caracteres excepto el alfabeto en mayúsculas
a{2,4}	ya sea aa, aaa o aaaa
a{2,}	dos o más ocurrencias de a
a{4}	exactamente 4 a's. aaaa
.	cualquier carácter a excepción de nueva línea
a*	0 o más ocurrencias de a
a+	1 o más ocurrencias de a
[a-z]	todas las letras minúsculas
[a-zA-Z]	cualquier letra del alfabeto
w(x y)z	ya sea wxz o wyz

3. **Sección del código del usuario:** Esta sección contiene declaraciones en C y funciones adicionales. También podemos compilar esas funciones separadamente y cargarlas con un analizador léxico.

Estructura básica del programa:

```
%{  
// Definiciones  
%}
```

```
%%  
Reglas  
%%
```

Sección de código del usuario

Cómo correr el programa:

Para correr el programa primero debe ser guardado con la extensión **.l** o **.lex**. Corre los comandos de abajo en la terminal para correr el programa.

1. lex filename.l o lex filename.lex
2. gcc lex.yy.c
3. ./a.out
4. Proporciona la entrada del programa si es requerido

Primer ejemplo: Contar los caracteres de una cadena

```
/**La seccion de definicion tiene una variable  
la cual puede ser accesada dentro de yylex()  
y main()***/  
%{  
int count = 0;  
%}  
  
/**La seccion de reglas tiene tres reglas, la primera  
regla hace que coincida con letras mayusculas, la segunda  
coincide con cualquier caracter a excepción de la nueva linea y  
la tercera regla no toma la entrada despues del enter***/  
%%  
[A-Z] {printf("%s capital letter\n", yytext);  
      count++;}  
.      {printf("%s not a capital letter\n", yytext);}  
\n {return 0;}  
%%  
  
/**La seccion del codigo imprime el numero de  
letras mayusculas que presenta la entrada dada***/  
int yywrap(){}  
int main(){
```

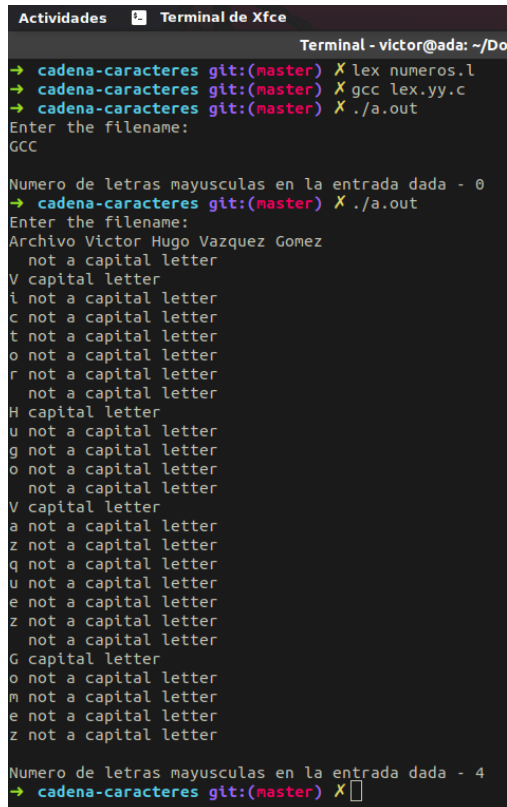
```
// Explicacion:
// yywrap() - envuelve la seccion de reglas de arriba
/* yyin - toma el apuntador del archivo
    el cual contiene la entrada*/
/* yylex() - esta es la funcion principal flex
    la cual ejecuta la seccion de reglas*/
// yytext es el texto en el buffer

FILE *fp;
char filename[50];
printf("Enter the filename: \n");
scanf("%s",filename);
fp = fopen(filename,"r");
yyin = fp;

yylex();
printf("\nNumero de letras mayusculas "
      "en la entrada dada - %d\n", count);

return 0;
}
```

SALIDA:



```
Terminal - victor@ada: ~/Do
→ cadena-caracteres git:(master) X lex numeros.l
→ cadena-caracteres git:(master) X gcc lex.yy.c
→ cadena-caracteres git:(master) X ./a.out
Enter the filename:
GCC
Numero de letras mayusculas en la entrada dada - 0
→ cadena-caracteres git:(master) X ./a.out
Enter the filename:
Archivo Victor Hugo Vazquez Gomez
V capital letter
i not a capital letter
c not a capital letter
t not a capital letter
o not a capital letter
r not a capital letter
not a capital letter
H capital letter
u not a capital letter
g not a capital letter
o not a capital letter
not a capital letter
V capital letter
a not a capital letter
z not a capital letter
q not a capital letter
u not a capital letter
e not a capital letter
z not a capital letter
not a capital letter
G capital letter
o not a capital letter
M not a capital letter
e not a capital letter
z not a capital letter
Numero de letras mayusculas en la entrada dada - 4
→ cadena-caracteres git:(master) X
```

Segundo ejemplo: Contar el número de caracteres y el número de líneas de la entrada

```
%{
int no_of_lines = 0;
int no_of_chars = 0;
%}

%%
\n    ++no_of_lines;
.      ++no_of_chars;
end    return 0;
%%

int yywrap(){}
int main(int argc, char **argv)
{
yylex();
printf("numero de lineas = %d, numero de caracteres = %d\n",
      no_of_lines, no_of_chars );

return 0;
}
```

SALIDA:

```
Terminal - victor@ada: ~/Documentos/escuela/sexta-semestre
→ numero-letras git:(master) X lex numeros.l
→ numero-letras git:(master) X gcc lex.yy.c
→ numero-letras git:(master) X ./a.out
Nicole
Rodriguez
Gonzalez
end
numero de lineas = 3, numero de caracteres = 23
→ numero-letras git:(master) X
```

Aplicaciones (Caso de estudio).

Algunas aplicaciones de los analizadores léxicos son:

- El analizador léxico divide la entrada en componentes léxicos.
- Los componentes se agrupan en categorías léxicas.
- Asociamos atributos a las categorías léxicas.
- Especificamos las categorías mediante expresiones regulares.
- Para reconocer los lenguajes asociados a las expresiones regulares empleamos autómatas de estados finitos (AFD).
- se pueden crear los AFD directamente a partir de la expresión regular.
- El analizador léxico utiliza la máquina discriminadora determinista.
- El tratamiento de errores en nivel léxico es muy simple.
- Se pueden emplear las ideas de los analizadores léxicos para facilitar el tratamiento de ficheros de texto.

Conclusión grupal

Los analizadores léxicos son como una aplicación en la que los compiladores que se encargan de verificar si el texto que se escribió en un formato aceptado para todo el programa que se escribe en ese lenguaje de programación y también se encarga de verificar que tenga coherencia, los analizadores léxicos nos sirven para resolver los problemas que pueden ocurrir a causa de que el programa no esté bien estructurado o esté mal escrito.

Referencias bibliográficas

1. Traductores y compiladores con Lex/Yacc, JFLEX/CUP y JavaCC. Edición electrónica. Sergio Gálvez Rojas, Miguel Ángel Mora Mata. páginas 23, 24, 28, 30, 32
2. Compiladores, principios, técnicas y herramientas; segunda edición. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. páginas 109, 110, 111, 113, 140
3. Ejemplos: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>