Universidad Nacional Autónoma de México
Facultad de Ciencias
Compiladores
Práctica Analizadores léxicos con Lex
Profesor: Adrián Ulises Mercado Martínez
Ayudante: Carlos Gerardo Acosta Hernández

Analizadores léxicos con Lex (flex)

Objetivo

Que el alumno conozca y utilice los principios para generar analizadores léxicos utilizando lex.

Introducción

Lex es una herramienta para generar analizadores léxicos, que se deben describir mediante las expresiones regulares de los tokens que serán reconocidas por el analizador léxico (scanner o lexer).

Originalmente fue desarrollado para el sistema operativo Unix, pero con la popularidad de Linux se creo una versión para este sistema llamada flex.

Estructura dde un archivo lex

Un programa en Lex consta de tres secciones:

```
Seccion de declaraciones
%%
Seccion de exprpesiones regulares
%%
Seccion de codigo de usuario (codigo en lenguaje C)

```

Sección de declaraciones

- **Directivas de código c.** Se utiliza para incluir los archivos de biblioteca y definir las variables globales.Su estructura es la siguiente:

```
    %{
      #include <stdio.h>
      int contador;
    }%

```

- **Macros.** Las macros son variables a las que se les asignar una expresión regular. Ejemplo *letra [a-zA-Z]*, la macro se llama letra, separada por un espacio de la definición de la expresión regular.
- **Directivas.** Las directivas le indican a Lex que realice tareas extras al momento de generar el analizador léxico.
  1. %option yylineno genera un contador de líneas automáticamente.
  2. %option noyywrap le indica a lex que debe generar de forma automática la función yywrap.
  3. %x o %s es para declarar estados léxicos.

Sección de expresiones regulares

- **Estructura:**

  Expresión Regular (espacio, nunca salto de línea) {Acción Léxica}
- **Expresión Regular:** debe estar escrita con la sintaxis de Lex
- **Acción Léxica:** se ejecuta cada vez que se encuentra una cadena que coincida con la expresión regular y es escrita en lenguaje C, encerrada por llaves.

Sección de código de usuario

En esta sección se escriben las funciones auxiliares para realizar el análisis léxico, por lo general es donde se agrega a main.

Metacaracteres

Carácter	Descripción
c	Cualquier carácter representado por c que no sea un operador
\c	El carácter c literalmente
"S"	La cadena s, literalmente
·	Cualquier carácter excepto el salto de línea
∧	Inicio de línea
$	Fin de línea
[s]	Cualquier carácter incluido dentro de la cadena s
[∧s]	Cualquier carácter que no esté dentro de s
\n	salto de línea
*	Cerradura de Kleene
+	Cerradura positiva
|	Disyunción
?	Cero o una instancia
{m, n}	entre m y n instancias

Desarrollo:

1. Instalación de flex
   (a) Para Debian o Ubuntu: sudo apt-get install flex
   (b) Para Suse u OpenSuse: sudo zypper in flex
2. para comprobar que se ha instalado correctamente:
   flex −−version
3. Primer programa en lex

```
%{
  #include <stdio.h>
%}

digito [0-9]
letra [a-zA-Z]
espacio [ \t\n}]
esps {espacio}

%%

{esps} {/*Ingnorar los espacios en blanco*/}
{digito}+ {printf("Encontre un numero %s\n", yytext);}
{letra}+  {printf("Encontre una palabra %s\n", yytext);}

%%

int main(){
  yylex();
}


```

Pasos

(a) Transcribir el código anterior a un archivo con extensión .l, .lex o .flex.
(b) Compilar mediante la instrucción: flex archivo.l
(c) Comprobar se genero el archivo lex.yy.c
(d) Compilar mediante: gcc lex.yy.c -o nombreEjecutable -lfl
(e) Ejecutar

Preguntas

(a) ¿Qué ocurre si en la primera sección se quitan las llaves al nombre de la macro espacio?

(b) ¿Qué ocurre si en la segunda sección se quitan las llaves a las macros?

(c) ¿Cómo se escribe un comentario en flex?

(d) ¿Qué se guarda en yytext?

(e) ¿Qué pasa al ejecutar el programa e introducir cadenas de caracteres y de dígitos por la consola?

(f) ¿Qué ocurre si introducimos caracteres como "*" en la consola?

4. Modificar al código anterior en un archivo nuevo, de tal manera que reconozca lo siguiente:

   (a) La expresión regular para los hexadecimales en lenguaje C

   (b) Los números decimales sin notación exponencial.

   (c) Los identificadores válidos del lenguaje C, con longitud máxima de 32 caracteres (**Sugerencia:**use el operador {m,n}).

   (d) Los espacios en blanco

   Compile y genere el ejecutable, realice varias pruebas para verificar que su programa funciona.

5. Escriba el siguiente código:

```
%{
#include <stdio.h>
%}

id [a-zA-Z]+
espacio [ \t\n]+

%%

{id}+ {printf("Encontre un identificador %s\n", yytext);}
int {printf("Encontre una palabra reservada %s\n", yytext);}
float {printf("Encontre una palabra reservada %s\n", yytext);}
if  {printf("Encontre una palabra reservada %s\n", yytext);}
else  {printf("Encontre una palabra reservada %s\n", yytext);}
{espacio} {/*Ingnorar los espacios en blanco*/}

%%

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

   Pasos

   (a) Genere el ejecutable.

   (b) Pruebe que el programa funciona introduciendo identificadores de solo letras.

   (c) Pruebe el programa con las palabras reservadas descritas

   (d) Pase la expresión regular {id} al final de las palabras reservadas.

   (e) Compile nuevamente el programa

   (f) Vuelva a probar.

   Preguntas

   (a) ¿Qué pasa con las palabras reservadas?

   (b) ¿Antes de hacer el cambio de la posición de {id} el programa se comporta de forma correcta con las palabras reservadas?

   (c) ¿Qué sucede cuando se cambia el orden de la expresión regular {id}?

   (d) ¿Será importante la precedencia o prioridad de las expresiones regulares en lex?

6. Manejo de errores

```
%{
  #include <stdio.h>
%}

id [a-zA-Z]+
espacio [ \t\n}]+

%%

{id}+ {printf("Encontre un identificador %s\n", yytext);}
int {printf("Encontre una palabra reservada %s\n", yytext);}
float {printf("Encontre una palabra reservada %s\n", yytext);}
if  {printf("Encontre una palabra reservada %s\n", yytext);}
else  {printf("Encontre una palabra reservada %s\n", yytext);}
{espacio} {/*Ingnorar los espacios en blanco*/}
. {printf("Ha ocurrido un error lexico: %s\n", yytext);}
%%

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

Pasos

(a) Genere el programa ejecutable

(b) Ejecute e introduzca símbolos que su programa no debe reconocer

Preguntas

(a) ¿Qué pasa con los errores?

(b) ¿Qué pasa si se cambia de posición la expresión regular "."?

Estados

Lex dispone de un mecanismo para activar expresiones regulares de forma condicional. Cualquier expresión regular a la que se le anteponga <nombre_estado> permanecerá activa únicamente cuando el analizador léxico se encuentre en el estado nombre_estado.

Por ejemplo:

<CADENA>[^"] { hacer_algo(); }

La expresión regular [^"] estará activa exclusivamente durante el estado <CADENA>.

Una expresión regular puede pertenecer a mas de un estado. Por ejemplo:

<CADENA, COMENTARIO>[^"] { }

La expresión regular se activará en los estado: CADENA, y COMENTARIO.

Los estados deben ser declarados en la primera sección de un archivo lex mediante las directivas %x o %s, seguidas por una lista de identificadores para los estados. %s declara una lista de estados inclusivos, mientras que %x hace a los estados exclusivos.

Los estados son activados mediante la acción BEGIN. Un estado permanecerá activo hasta que se ejecute la siguiente acción BEGIN.

Si el estado es inclusivo, las expresiones regulares que no hayan sido etiquetadas con alguno de los estados también permanecerán activas, en cambio; si el estado es exclusivo, solamente las expresiones regulares etiquetadas con el estado podrán estar activas.

Ejemplo:

```
%s inclusivo
%%
<inclusivo>expresion1 { accion1 }
expresion2 { accion2}

```

Tendría una equivalencia con

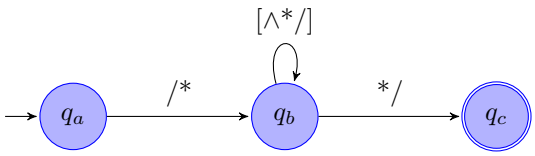
```
%x exclusivo
%%
<exclusivo>expresion1 { accion1 }
<INITIAL,exclusivo>expresion2 { accion2}

```

Si no se incluye *exclusivo* en las etiquetas de la expresion2 entonces esta expresión no estará activa durante el periodo que el estado *exclusivo* sea el estado actual.

Algunos estados predefinidos en lex son: INITIAL, STRING, COMMENT, entre otros.

En ocasiones es más sencillo diseñar el AFD que la expresión regular, por ejemplo en el caso de los comentarios podríamos tener el siguiente autómata.




```
%x comentario

%%

"/*"          BEGIN(comentario);

<comentario>[^*\n]*        {/* ignora todo lo que no sea '*' ni salto de linea*/}
<comentario>"*"+[^*/\n]*   {/* ignora '*'s no seguidos por '/' y el salto de linea */}
<comentario>\n             {/* Ignora los saltos de linea*/}
<comentario>"*"+"/"        {BEGIN(INITIAL);}

```

yyin y yyout

La variable global yyin sirve para indicarle a lex que va a leer el código fuente ya sea de la entrada estándar o de un archivo de texto

La variable global yyout sirve para escribir en un archivo de texto que puede ser utilizado como salida

Este código se usa para abrir el archivo, aunque falta hacer las comprobaciones para validar el número de argumentos que son pasados a main y que en realidad se abra el archivo.

```
int main(inr argc, char **argv){
FILE *f, *o;
f = fopen(argv[1], "r");
yyin = f;
...
fclose(yyin);
}

```

De forma similar se puede generar un archivo para yyout

```
int main(inr argc, char **argv){
FILE *f;
f = fopen("salida", "w");
yyout = f;
...

fclose(yyout);
}

```

Función yywrap

La función yywrap sirve para indicarle a lex si debe leer un archivo más, si ésta retorna un 1 ya no se leerán mas archivos, si retorna un 0 la función yylex se quedará en espera de un nuevo archivo.

La función yywrap se puede programar con el siguiente código

```
%%
int yywrap(){
return 1;
}

```

En caso de que no se quiera programar la función yywrap se puede utilizar la directiva %option noyywrap

```
%option noyywrap

```

Contador de líneas

Para contar las líneas del archivo fuente que va a leer la función yylex se puede definir una expresión regular como sigue:

```
%{
int linea;
%}
...
%%

"\n"    { linea++;}

```

Sin embargo lex tiene la opción de generar una variable que cuente líneas de forma automática.

```
%option yylineno

```

Ejercicios

(a) Hacer un programa en lex que reciba como entrada un archivo con palabras escritas en minúsculas y genere como salida un archivo con todas las palabras escritas en mayúsculas. (considere una palabra como aquella cadena que la separan espacios)

(b) Hacer un programa en lex que se comporte como el comando wc de Linux, es decir dado un archivo de entrada debe arrojar a la salida cuántas palabras tiene, el número de líneas, el número de caracteres.

(c) Hacer un programa en lex que reciba un archivo con palabras y espacios en blanco y genere un archivo en donde se eliminaron los espacios en blanco.

(d) Hacer un analizador léxico que genere los siguientes tokens

i. Números enteros
ii. Números reales
iii. Identificadores del tipo lenguaje C
iv. Cadenas de caracteres que están encerradas entre comillas
v. Palabras reservadas: if, then, else, while, do, case, is, void , true,false, begin, end y not.
vi. Los espacios en blanco.
vii. Los comentarios de una sola línea que empiezan con $--$ y termina con el salto de línea
viii. Los comentarios multilínea que abren con $< *$ y cierran con $* >$.
ix. Los operadores aritméticos: +, -, *, /, %
x. Los operadores relacionales: ¿, ¡, ¡¿, =
xi. El operador de asignación ":="
xii. Los símbolos especiales: (,), {, }, ;, ,
xiii. En caso de ocurrir un error debe escribir el caracter que lo produjo y el número de línea donde ocurrió.

(e) (**Opcional**)Hacer un programa en lex que reciba un archivo que contiene líneas de números enteros separadas por espacios, la líneas pueden terminar en A o en B, para las líneas que terminen en A imprimir la suma de los números de esa línea, para las líneas que terminen en B imprimir la línea tal como aparece en el programa fuente.

(f) Escriba sus resultados y conclusiones.