



Instituto Politécnico Nacional

Escuela Superior de Cómputo



Especialidad: Sistemas computacionales

Materia: Compiladores

Práctica 4.

Analizador léxico del lenguaje C

Alumno: Rubio Morín Raymundo Marcos
Agustín

Boleta: 2014630445

Maestro: Saucedo Delgado Rafael Norman

Contenido

Introducción	3
Desarrollo	4
Metodología	4
Ejemplificar el lenguaje	4
Identificar las clases léxicas.	4
Expresiones regulares para las clases léxicas.....	5
Código en LEX.....	8
Pruebas.....	10
Conclusiones	10
Referencias.....	10
Anexo	11

Introducción

A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma ^[1]:

(nombre-token, valor-atributo)

Para llevar a cabo la elaboración de esta práctica se eligió como proyecto el compilador de lenguaje C a Ensamblador, ya que no cuento con el tiempo necesario para la elaboración de algún otro proyecto.

En esta práctica se usa el generador de analizadores léxico Flex 2.6.0, el cual va recorriendo la entrada estándar hasta encontrar una concordancia, es decir, un lexema correspondiente al lenguaje de algunas de las expresiones regulares representadas por patrones ^[2]. En esta practica se usa para identificar a que clase o categoría pertenece cada lexema de un lenguaje C.

C es un lenguaje de programación de propósito general originalmente desarrollado por Dennis Ritchie entre 1969 y 1972 en los Laboratorios Bell, como evolución del anterior lenguaje B, a su vez basado en BCPL.

C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistema, aunque también se utiliza para crear aplicaciones.

Se trata de un lenguaje de tipos de datos estáticos, débilmente tipificado, de medio nivel, ya que dispone de las estructuras típicas de los lenguajes de alto nivel, pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos ^[3].

Desarrollo

Metodología

Ejemplificar el lenguaje

Ejemplo 1:

- Importación de bibliotecas, declaración de variables y su tipo (int, float , ... ,etc).
- Declaración de la función principal Main()
- Tipo de retorno return (parámetro).

```
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

Figura 1. Programa sencillo en C

Ejemplo 2:

- Declaración de variables
 - Tipo_dato nombre_de_variable = valor
- Asignación
 - Nombre_de_variable = valor

```
#include <stdio.h>

main() /* Suma dos valores */
{
    int num1=4,num2,num3=6;
    printf("El valor de num1 es %d",num1);
    printf("\nEl valor de num3 es %d",num3);
    num2=num1+num3;
    printf("\nnum1 + num3 = %d",num2);
}
```

Figura 2. Declaración de variables

Identificar las clases léxicas.

Se identificaron las clases léxicas:

- Palabra reservada
- Tipo de dato

- Ciclo
- Id
- Constante hexadecimal
- Constante octal
- Constante
- Operador aritmético
- Operador de comparación
- Operador a nivel de bit
- Salto de línea
- Modificador
- Especificador de almacenamiento
- Sentencia de control
- Directiva
- Operador
- Modificador de acceso
- Constante double
- Constante long double
- Asignación
- Predecremento
- Preincremento
- Miembro de puntero
- Separador
- Operador ternario

Expresiones regulares para las clases léxicas

Expresiones regulares usando Flex ^[4]

'x'

coincidir con el caracter 'x'

'.'

cualquier caracter (byte) excepto nueva línea

'[xyz]'

una "clase de caracter"; en este caso, el patrón coincide con un 'x', una 'y' o un 'z'

'[abj-oZ]'

una "clase de carácter" con un rango en ella; coincide con un 'A', una 'b', luego una 'j' mediante 'O' o un 'Z'

'[^ AZ]'

una "clase de caracter negado", es decir, cualquier carácter excepto los de la clase. En este caso, cualquier carácter EXCEPTO una letra mayúscula.

``[^ AZ \n]``
cualquier carácter EXCEPTO una letra mayúscula o una nueva línea

`' r *'`
cero o más *r* , donde *r* es cualquier expresión regular

`' r +'`
uno o más *R* 's

`' r ?'`
cero o una *r* (es decir, "una *r* opcional ")

`` r {2,5} ``
en cualquier lugar de dos a cinco *r* s'

`` r {2,} ``
dos o más *R* 's

`` r {4} ``
exactamente 4 *r* 's

`"{ nombre }"`
la expansión de la definición de "*nombre* "

``" [xyz] \ "foo" ``
la cadena literal: ``[xyz]" foo ``

``\ x ``
si *x* es un `'A '`, `'b'`, `'f'`, `n`, `'r'`, `'t'` o `'v'`, luego la interpretación ANSI-C de `\ x` . De lo contrario, un literal `' x '` (usado para escapar de operadores como `'*'`)

``\ 0 ``
un caracter NUL (código ASCII 0)

``\ 123 ``
el caracter con valor octal 123

``\ x2a'`
el caracter con valor hexadecimal 2ª

`'(r)'`
coincidir con una *r* ; los paréntesis se utilizan para anular la precedencia

`' r s '`
la expresión regular *r* seguida de la expresión regular *s* ; llamado "concatenación"

`` r | s ``

ya sea una r o una s

`' r / s '`

una r , pero solo si va seguida de una s . El texto que coincide con s se incluye al determinar si esta regla es la *coincidencia más larga*, pero luego se devuelve a la entrada antes de que se ejecute la acción. Entonces, la acción solo ve el texto que coincide con r . Este tipo de patrón se denomina *contexto final*.

`^ r '`

una r , pero solo al principio de una línea (es decir, que recién comienza a escanear o justo después de escanear una nueva línea).

`' r $'`

una r , pero solo al final de una línea (es decir, justo antes de una nueva línea). Equivalente a `" r / \ n"`. Tenga en cuenta que la noción de flex de "nueva línea" es exactamente lo que el compilador de C utilizado para compilar flex interprete como `" \ n"`; en particular, en algunos sistemas DOS debe filtrar las `\ r` en la entrada usted mismo, o usar explícitamente `r / \ r \ n` para `"r $"`.

`` < s > r '`

una r , pero solo en las condiciones de inicio s (ver más abajo para la discusión de las condiciones de inicio) `< s1 , s2 , s3 > r` igual, pero en cualquiera de las condiciones de inicio `s1 , s2` o `s3`

`` < * > r '`

una r en cualquier condición de arranque, incluso una exclusiva.

`` < < EOF > > '`

un final de archivo

`` < s1 , s2 > < < EOF > > '`

un final de archivo cuando está en la condición de inicio `s1` o `s2`

Código en LEX

```
lexico.l
1  D      [0-9]
2  L      [a-zA-Z_]
3  H      [a-zA-F0-9]
4  E      [Ee][+-]?{D}+
5  FS     (f|F|l|L)
6  IS     (u|U|l|L)*
7
8  %{
9  #include <stdio.h>
10
11  %}
12
13  %%
14
15  "auto"      { printf("<Especificador de almacenamiento>"); }
16  "break"     { printf("<Sentencia de control>"); }
17  "case"      { printf("<Palabra reservada>"); }
18  "char"      { printf("<Tipo de dato>"); }
19  "const"     { printf("<Modificador>"); }
20  "continue"  { printf("<Sentencia de control>"); }
21  "default"   { printf("<Palabra reservada>"); }
22  "do"        { printf("<Ciclo>"); }
23  "double"    { printf("<Tipo de dato>"); }
24  "else"      { printf("<Directiva>"); }
25  "enum"      { printf("<Tipo de dato>"); }
26  "extern"    { printf("<Especificador de almacenamiento>"); }
27  "float"     { printf("<Tipo de dato>"); }
28  "for"       { printf("<Ciclo>"); }
29  "goto"      { printf("<Palabra reservada>"); }
30  "if"        { printf("<Directiva>"); }
31  "int"       { printf("<Tipo de dato>"); }
32  "long"      { printf("<Tipo de dato>"); }
33  "register"   { printf("<Especificador de almacenamiento>"); }
34  "return"    { printf("<Palabra reservada>"); }
35  "short"     { printf("<Tipo de dato>"); }
36  "signed"    { printf("<Tipo de dato>"); }
37  "sizeof"    { printf("<Operador>"); }
38  "static"    { printf("<Modificador de almacenamiento>"); }
39  "struct"    { printf("<Palabra reservada>"); }
40  "switch"    { printf("<Sentencia de control>"); }
41  "typedef"   { printf("<Palabra reservada>"); }
42  "union"     { printf("<Palabra reservada>"); }
43  "unsigned"  { printf("<Tipo de dato>"); }
44  "void"      { printf("<Tipo de dato>"); }
45  "volatile"  { printf("<Modificador de acceso>"); }
46  "while"     { printf("<Ciclo>"); }
47
48  {L}({L}|{D})* { printf("<ID>"); }
```

Figura 4. Léxico1.l


```

lexico.l
49
50 0[xX]{H}+{IS}?      { printf("<Constante Hexadecimal>"); }
51 0{D}+{IS}?         { printf("<Constante Octal>"); }
52 {D}+{IS}?          { printf("<Constante>"); }
53 L?'(\\.|[^\\"'])*' { printf("<Cadena>"); }
54
55 {D}+{E}{FS}?        { printf("<Float>"); }
56 {D}*"."{D}+({E})?{FS}? { printf("<Double>"); }
57 {D}+"."{D}*({E})?{FS}? { printf("<Long Double>"); }
58
59 L?"(\\.|[^\\""])*\" { printf("<Cadena>"); }
60
61 "...\"              { printf("<Continuara>"); }
62 ">>\"              { printf("<Operador a nivel de bit>"); }
63 "<<\"              { printf("<Operador a nivel de bit>"); }
64 "+=\"              { printf("<Asignacion>"); }
65 "-=\"              { printf("<Asignacion>"); }
66 "*=\"              { printf("<Asignacion>"); }
67 "/=\"              { printf("<Asignacion>"); }
68 "%=\"              { printf("<Asignacion>"); }
69 "&=\"              { printf("<Asignacion>"); }
70 "^=\"              { printf("<Asignacion>"); }
71 "|=\"              { printf("<Asignacion>"); }
72 ">>\"              { printf("<Operador a nivel de bit>"); }
73 "<<\"              { printf("<Operador a nivel de bit>"); }
74 "++\"              { printf("<Pre Incremento>"); }
75 "--\"              { printf("<Pre Decremento>"); }
76 "->\"              { printf("<Miembro de puntero>"); }
77 "&&\"              { printf("<Operador de comparacion>"); }
78 "||\"              { printf("<Operador de comparacion>"); }
79 "<=\"              { printf("<Operador de comparacion>"); }
80 ">=\"              { printf("<Operador de comparacion>"); }
81 "=="              { printf("<Operador de comparacion>"); }
82 "!="              { printf("<Operador de comparacion>"); }
83 ";"              { printf("<Punto y coma>"); }
84 ("{"|"<%")        { printf("<Llave de apertura>"); }
85 ("}"|">%")        { printf("<Llave de termino>"); }
86 ","              { printf("<Separador>"); }
87 ":"              { printf("<Operador ternario>"); }
88 "="              { printf("<Asignacion>"); }
89 "("              { printf("<Parentesis de apertura>"); }
90 ")"              { printf("<Parentesis de termino>"); }
91 ("["|"<:")        { printf("<Corchete de apertura>"); }
92 ("]"|">:")        { printf("<Corchete de termino>"); }
93 "."              { printf("<Miembro>"); }
94 "&\"              { printf("<Operador bit a bit>"); }
95 "!"              { printf("<Operador Not>"); }
96 "~\"              { printf("<Operador bit a bit>"); }

```

Figura 5. Lexico2.1

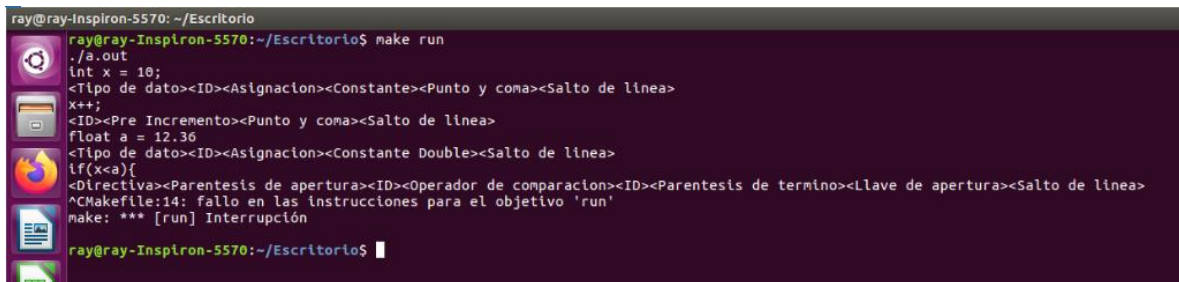
```

97  "-"      { printf("<Operador aritmetico>"); }
98  "+"      { printf("<Operador aritmetico>"); }
99  "*"      { printf("<Operador aritmetico>"); }
100 "/"      { printf("<Operador aritmetico>"); }
101 "%"      { printf("<Operador aritmetico>"); }
102 "<"      { printf("<Operador de comparacion>"); }
103 ">"      { printf("<Operador de comparacion>"); }
104 "^"      { printf("<Operador bit a bit>"); }
105 "|"      { printf("<>Operador bit a bit"); }
106 "?"      { printf("<Operador ternario>"); }
107
108 [\t\v\n\f] {printf("<Salto de linea>\n");}
109 .         { /* ignore bad characters */ }
110
111 %%
112

```

Figura 6. Lexico3.l

Pruebas



```

ray@ray-Inspiron-5570: ~/Escritorio
ray@ray-Inspiron-5570:~/Escritorio$ make run
./a.out
int x = 10;
<Tipo de dato><ID><Asignacion><Constante><Punto y coma><Salto de linea>
x++;
<ID><Pre Incremento><Punto y coma><Salto de linea>
float a = 12.36;
<Tipo de dato><ID><Asignacion><Constante Double><Salto de linea>
if(x<a){
<Directiva><Parentesis de apertura><ID><Operador de comparacion><ID><Parentesis de termino><Llave de apertura><Salto de linea>
^CMakefile:14: fallo en las instrucciones para el objetivo 'run'
make: *** [run] Interrupción
ray@ray-Inspiron-5570:~/Escritorio$

```

Figura 5. Prueba léxico.l

Conclusiones

Me llevo tiempo identificar a que clase pertenecían las expresiones regulares, ya que sabía que hacían mas no como se llaman, así como lograr identificar que hace algunas de expresiones regulares de constantes (por ejemplo: {D}+{E}{FS}?) y tuve que averiguarlo haciendo pruebas con números.

Referencias

- [1] Compiladores principios, técnicas y herramientas. Segunda edición. Alfred V. Aho, Editorial: Pearson, México, 2008.
- [2] Compiladores I. C.P.S. Universidad de Zaragoza-J.Ezpeleta, consultado en 2020. Disponible:

<http://webdiis.unizar.es/~ezpeleta/lib/exe/fetch.php?media=misdatos:compi:2bis.introflex.pdf>

[3] Wikipedia, C (lenguaje de programación), 2020. Disponible:
[https://es.wikipedia.org/wiki/C_\(lenguaje_de_programaci%C3%B3n\)](https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n))

[4] Flex, version 2.5 A fast scanner generator Edition 2.5, March 1995 Vern Paxson
Available:
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjI07rykvvsAhVFXXwKHxQGCNMQFjAlegQICRAC&url=ftp%3A%2F%2Fftp.gnu.org%2Ffold-gnu%2Fmanuals%2Fflex-2.5.4%2Fhtml_mono%2Fflex.html&usg=AOvVaw0XNLHsqJSHyDQGxUfRgCpJ

Anexo

Para poder compilar esta práctica se hace uso de un archivo Makefile y un archivo Main.c dentro de la carpeta que contenga al archivo léxico.l

```
lex.yy.c: lexico.l
    flex lexico.l

main.o: main.c
    gcc -c main.c

a.out: main.o lex.yy.o
    gcc main.o lex.yy.o -lfl

clean:
    run -f a.out main.o lex.yy.o lex.yy.c

run: a.out
    ./a.out
```

Figura 6. Makefile

```
] int main(void){
    yylex();
    return 0;
}
```

Figura 7. Main.c