PROGRAMA 1

“Analizador léxico”

INTEGRANTES:

- *Aguilar Castro Carlos Alfonso*
- *Bustamante Piza Karla Mireli*
- *Ugalde Vivo José Francisco*

Facultad de Ingeniería, UNAM

Grupo: 01

Compiladores

ANÁLISIS DEL PROBLEMA

Se debe desarrollar el análisis léxico de la gramática mostrada en la mostrada a continuación, deberá contener sus declaraciones correspondientes con Lex.

1. programa → declaraciones funciones
2. declaraciones → tipo lista_var; declaraciones
   | tipo_registro lista_var; declaraciones
   | ε
3. tipo_registro → **estructura inicio** declaraciones **fin**
4. tipo → base tipo_arreglo
5. base → **ent** | **real** | **dreal** | **car** | **sin**
6. tipo_arreglo → **(num)** tipo_arreglo | ε
7. lista_var → lista_var, **id** | **id**
8. funciones → **def** tipo **id(**argumentos**) inicio** declaraciones sentencias **fin** funciones
   | ε
9. argumentos → listar_arg | **sin**
10. lista_arg → lista_arg, arg | arg
11. arg → tipo_arg **id**
12. tipo_arg → base param_arr
13. param_arr → **( )** param_arr | ε
14. sentencias → sentencias sentencia | sentencia
15. sentencia → **si** e_bool **entonces** sentencia **fin**
    | **si** e_bool **entonces** sentencia **sino** sentencia **fin**
    | **mientras** e_bool **hacer** sentencia **fin**
    | **hacer** sentencia **mientras** e_bool;
    | **segun (**variable**) hacer** casos predeterminado **fin**
    | variable **:=** expresion **;**
    | **escribir** expresion **;**
    | **leer** variable **;** | **devolver;**
    | **devolver** expresion;
    | **terminar;**
    | **inicio** sentencias **fin**
16. casos → **caso num:** sentencia casos | **caso num:** sentencia
17. predeterminado → **pred:** sentencia | ε
18. e_bool → e_bool **o** e_bool | e_bool **y** e_bool | **no** e_bool | **(** e_bool **)**
    | relacional | **verdadero** | **falso**
19. relacional → relacional oprel relacional | expresion
20. oprel → > | < | >= | <= | <> | =
21. expresion → expresion oparit expresion
    | expresion **%** expresion | **(**expresion **)** | **id**
    | variable | **num** | **cadena** | **caracter** | **id(**parametros**)**
22. oparit → + | − | ∗ | /
23. variable → dato_est_sim | arreglo
24. dato_est_sim → dato_est_sim **.id** | **id**
25. arreglo → **id (** expresion **)** | arreglo **(** expresion **)**
26. parametros → lista_param | ε
27. lista_param → lista_param, expresion | expresion

DISEÑO DE LA SOLUCI´ON I.

- Separar los terminales de los no terminales

TERMINALES	NO TERMINALES
• **ent** • **real** • **dreal** • **car** • **sin** • **estructura** • **def** • **si** • **sino** • **mientras** • **hacer** • **inicio** • **fin** • **escribir** • **leer** • **devolver** • **según** • **caso** • **pred** • **o** • **y** • **no** • **verdadero** • **falso** • **terminar** • **(** • **)** • **+** • **-** • ***** • **/** • **%** • **<** • **<=** • **>** • **>=** • **=**	• declaraciones • tipo_arreglo • lista_var • funciones • argumentos • arg • tipo_arg • param_arr • sentencias • sentencia • predeterminado • e_bool • relacional • expresión • variable • variable_comp • dato_est_sim • arreglo • parámetros • lista_param

• := • ; • .	

- Las expresiones regulares para los terminales

```
caracter        [a-zA-Z]
digito          [0-9]
estructura      (e|E)(s|S)(t|T)(r|R)(u|U)(c|C)(t|T)(u|U)(r|R)(a|A)
inicio          (i|I)(n|N)(i|I)(c|C)(i|I)(o|O)
fin             (f|F)(i|I)(n|N)
ent             (e|E)(n|N)(t|T)(" ")
real            (r|R)(e|E)(a|A)(l|L)
dreal           (d|D)(r|R)(e|E)(a|A)(l|L)
car             (c|C)(a|A)(r|R)
sin             (s|S)(i|I)(n|N)
num             {digito}+
nreal           ({digito})+"."({digito})*|"."({digito})*
def             (d|D)(e|E)(f|F)
entonces        (e|E)(n|N)(t|T)(o|O)(n|N)(c|C)(e|E)(s|S)
sino            (s|S)(i|I)(n|N)(o|O)
si              (s|S)(i|I)
hacer           (h|H)(a|A)(c|C)(e|E)(r|R)
mientras        (m|M)(i|I)(e|E)(n|N)(t|T)(r|R)(a|A)(s|S)
segun           (s|S)(e|E)(g|G)(u|U)(n|N)
escribir        (e|E)(s|S)(c|C)(r|R)(i|I)(b|B)(i|I)(r|R)
leer            (l|L)(e|E)(e|E)(r|R)
devolver        (d|D)(e|E)(v|V)(o|O)(l|L)(v|V)(e|E)(r|R)
terminar        (t|T)(e|E)(r|R)(m|M)(i|I)(n|N)(a|A)(r|R)
caso            (c|C)(a|A)(s|S)(o|O)
pred            (p|P)(r|R)(e|E)(d|D)
o               (o|O)
y               (y|Y)
no              (n|N)(o|O)
verdadero       (v|V)(e|E)(r|R)(d|D)(a|A)(d|D)(e|E)(r|R)(o|O)
falso           (f|F)(a|A)(l|L)(s|S)(o|O)
lib             ({caracter}+)(".h")
id              ({caracter}|"_"|"@")({caracter}|"_"|"@"|{digito})+
cadena          ("\/")([^\n\/]*([.])*)*("\/")
```

- El AFD resultante

IMPLEMENTACIÓN

Como se sabe, un programa en Lex se compone de tres secciones, la cuales son:

1. Definiciones: Contiene declaraciones de definiciones de nombres sencillas para simplificar la especificación del escáner, y declaraciones de condiciones de arranque. Donde se colocó la siguiente parte de código.

```
%{
   int yylineno;
   #include    <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   #include "tokens.h"
%}

%option yylineno
%option noyywrap

caracter       [a-zA-Z]
digito         [0-9]
estructura     (e|E)(s|S)(t|T)(r|R)(u|U)(c|C)(t|T)(u|U)(r|R)(a|A)
inicio         (i|I)(n|N)(i|I)(c|C)(i|I)(o|O)
fin            (f|F)(i|I)(n|N)
ent            (e|E)(n|N)(t|T)
real           (r|R)(e|E)(a|A)(l|L)
dreal          (d|D)(r|R)(e|E)(a|A)(l|L)
car            (c|C)(a|A)(r|R)
sin            (s|S)(i|I)(n|N)
nentero        [1-9]({digito})*
nreal          ({digito})+"."({digito})*|"."({digito})*
num            {nreal}
def            (d|D)(e|E)(f|F)
entonces       (e|E)(n|N)(t|T)(o|O)(n|N)(c|C)(e|E)(s|S)
sino           (s|S)(i|I)(n|N)(o|O)
si             (s|S)(i|I)
hacer          (h|H)(a|A)(c|C)(e|E)(r|R)
mientras       (m|M)(i|I)(e|E)(n|N)(t|T)(r|R)(a|A)(s|S)
segun          (s|S)(e|E)(g|G)(u|U)(n|N)
escribir       (e|E)(s|S)(c|C)(r|R)(i|I)(b|B)(i|I)(r|R)
leer           (l|L)(e|E)(e|E)(r|R)
devolver       (d|D)(e|E)(v|V)(o|O)(l|L)(v|V)(e|E)(r|R)
terminar       (t|T)(e|E)(r|R)(m|M)(i|I)(n|N)(a|A)(r|R)
```

2. Reglas: Es la serie de reglas que contendrá:

```
%%
{ent}           {return INT;}
{real}          {return FLOAT;}
{dreal}         {return DOUBLE;}
{car}           {return CHAR;}
{sin}           {return SIN;}

{estructura}    {return STRUCT;}
{def}           {return DEF;}
{si}            {return IF;}
{sino}          {return ELSE;}
{mientras}      {return WHILE;}
{hacer}         {return DO;}
{inicio}        {return START;}
{fin}           {return END;}
{escribir}      {return WRITE;}
{leer}          {return READ;}
{devolver}      {return RETURN;}
{segun}         {return SWITCH;}
{caso}          {return CASE;}
{pred}     {return DEFAULT;}

{o}             {return OR;}
{y}             {return AND;}
{no}            {return NOT;}
{verdadero}     {return TRUE;}
{falso}         {return FALSE;}

{id}            {return ID;}

"("  {return LPAR;}
")"  {return RPAR;}
"+"  {return PLUS;}
"-"  {return MINUS;}
"*"  {return MUL;}
```

3. Código de usuario: Esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por este. Aquí colocamos nuestro main.

```
%%

FILE *yyin;
char *yytext;

int main(int text, char **arch){
  if(text <2){
    printf("Falta el nombre del archivo\n");
  }
  yyin = fopen(arch[1], "r");

  FILE *yyout;
  yyout = fopen("tokensSal.txt", "w+"); //archivo donde se escriben los tokens

  int tokenVal;
  char tokenChar[3];
  while(tokenVal = yylex()){
      printf("%s valor del token %i\n",yytext,tokenVal);
      fputs(yytext,yyout);
      fputs("  ",yyout);
      sprintf(tokenChar,"%d",tokenVal);
      fputs(tokenChar,yyout);
      fputs("\n",yyout);
  }
    return 0;
  printf("\n");

  fclose(yyin);
  fclose(yyout);
  return 0;
}
```

FORMA DE EJECUTAR

1. Primero se deberá ejecutar el siguiente comando:

   # lex alexv3.l

   Este generara nuestro .c para así poder realizar el siguiente paso

2. Se compilará el .c obtenido en el paso anterior con el comando:

   #gcc lex.yy.c

3. Se obtuvo nuestro ejecutable de nombre “compilar” por lo que se utilizará el siguiente comando:

   #./a.out entrada.txt

   Se añade “entrada.txt” debido a que nuestro programa espera que se le dé un archivo .txt para poder funcionar.

```
[kmireli@karlamireli program1]$ ls
alex.l  gramatica.txt  tokens.h  tokensSal.txt
[kmireli@karlamireli program1]$ lex alex.l
[kmireli@karlamireli program1]$ ls
alex.l  gramatica.txt  lex.yy.c  tokens.h  tokensSal.txt
[kmireli@karlamireli program1]$ gcc lex.yy.c -o compilar
[kmireli@karlamireli program1]$ ls
alex.l  compilar  gramatica.txt  lex.yy.c  tokens.h  tokensSal.txt
[kmireli@karlamireli program1]$ ./compilar gramatica.txt
progra valor del token 41
m →  de valor del token 41
clara valor del token 41
cio valor del token 17
ne valor del token 41
s funcio valor del token 17
ne valor del token 41
s
de valor del token 41
clara valor del token 41
cio valor del token 17
ne valor del token 41
s → ti valor del token 41
po valor del token 17
 li valor del token 41
sta_ valor del token 41
var valor del token 41
; valor del token 34
de valor del token 41
clara valor del token 41
cio valor del token 17
ne valor del token 41
s | valor del token 41
 ti valor del token 41
po valor del token 17
 regi valor del token 41
```