

**Compiladores**  
**Análisis Léxico**  
**Programa 1**

- A. **Descripción del problema.** Se presenta la siguiente gramática para la cual se requiere hacer un análisis léxico, separando los terminales de los no terminales, realizando las expresiones regulares de cada uno de los terminales así como el AFD resultante de este análisis. Una vez terminado dicho análisis se deberá programar el analizador léxico utilizando flex.

---sin: significa sin tipo, car: tipo caracter----

1. programa  $\rightarrow$  declaraciones funciones
2. declaraciones  $\rightarrow$  tipo lista\_var; declaraciones | tipo\_registro lista\_var; declaraciones |  $\epsilon$
3. tipo\_registro  $\rightarrow$  **estructura inicio** declaraciones **fin**
4. tipo  $\rightarrow$  base tipo\_arreglo
5. base  $\rightarrow$  **ent** | **real** | **dreal** | **car** | **sin**
6. tipo\_arreglo  $\rightarrow$  [**num**] tipo\_arreglo |  $\epsilon$
7. lista\_var  $\rightarrow$  lista\_var, **id** | **id**
8. funciones  $\rightarrow$  **def** tipo **id**(argumentos) **inicio** declaraciones sentencias **fin** funciones |  $\epsilon$
9. argumentos  $\rightarrow$  lista\_arg | **sin**
10. lista\_arg  $\rightarrow$  lista\_arg, arg | arg
11. arg  $\rightarrow$  tipo\_arg **id**
12. tipo\_arg  $\rightarrow$  base param\_arr
13. param\_arr  $\rightarrow$  [ ] param\_arr |  $\epsilon$
14. sentencias  $\rightarrow$  sentencias sentencia | sentencia
15. sentencia  $\rightarrow$  **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  $\rightarrow$  **caso num:** sentencia casos | **caso num:** sentencia
17. predeterminado  $\rightarrow$  **pred:** sentencia |  $\epsilon$
18. e\_bool  $\rightarrow$  e\_bool **o** e\_bool | e\_bool **y** e\_bool | **no** e\_bool | ( e\_bool ) | relacional  
| **verdadero** | **falso**
19. relacional  $\rightarrow$  relacional oprel relacional | expresion
20. oprel  $\rightarrow$  > | < | >= | <= | <> | =
21. expresion  $\rightarrow$  expresion oparit expresion | expresion % expresion | (expresion ) | **id**  
| variable | **num** | **cadena** | **caracter** | **id**(parametros)
22. oparit  $\rightarrow$  + | - | \* | /

23. variable  $\rightarrow$  dato\_est\_sim | arreglo  
 24. dato\_est\_sim  $\rightarrow$  dato\_est\_sim .id | id  
 25. arreglo  $\rightarrow$  id [ expresion ] | arreglo [ expresion ]  
 26. parametros  $\rightarrow$  lista\_param |  $\epsilon$   
 27. lista\_param  $\rightarrow$  lista\_param, expresion | expresion

## B. Diseño de la solución

- I. Separar los terminales de los no terminales

Terminales		No terminales	
estructura	sino	programa	oprel
inicio	mientras	declaraciones	expresion
fin	hacer	tipo_registro	oparit
ent	segun	tipo	variable
real	escribir	base	dato_est_sim
dreal	leer	tipo_arreglo	arreglo
car	devolver	lista_var	parametros
sin	terminar	funciones	lista_param
num	caso	argumentos	
id	pred	lista_arg	
def	o	arg	

( )	y	tipo_arg	
[]	no	param_arr	
,	verdadero	sentencias	
;	falso	sentencia	
.	%	casos	
:	cadena	predeterminado	
si	caracter	e_bool	
entonces		relacional	

## II. Las expresiones regulares para los terminales

caracter-> [a-zA-Z]+

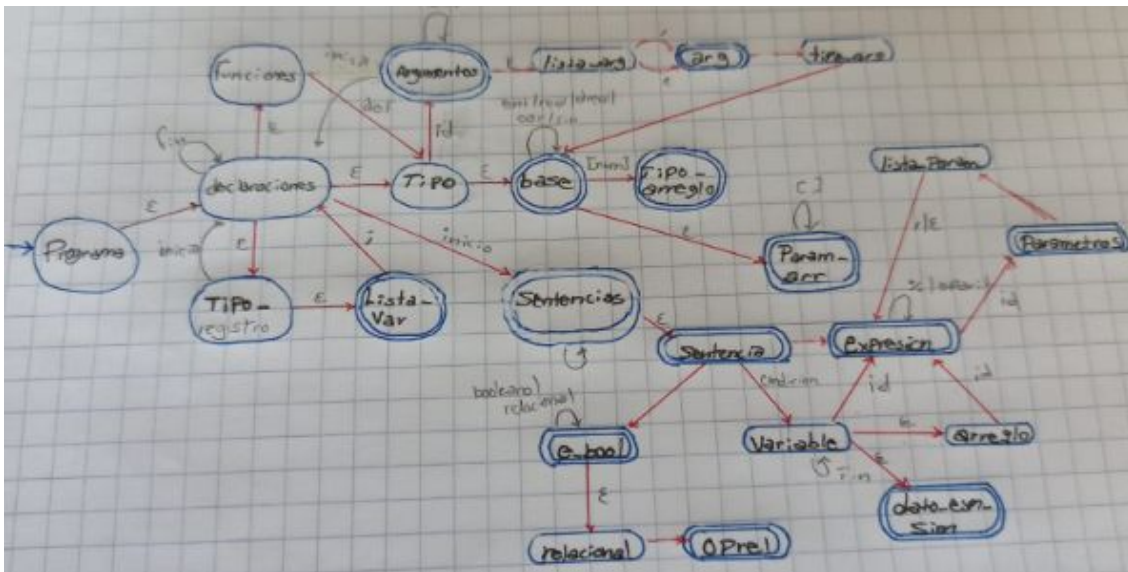
num-> [0-9]+

digito->[0-9]

id-> caracter\_(digito)\*

cadena->(caracter)+

### III. El AFD resultante(Imagen)



C. Implementación. Describir cómo está implementado su programa, las partes que lo componen (no es todo el código).

Lo primero es la parte de la declaración de los encabezados o delimitadores:

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "y.tab.h"
void error(char *s);
%}
```

Esta instrucción es para indicarle al software que debe de leer solo un archivo de entrada.

```
%option noyywrap
```

Después de la instrucción previamente mencionada, sigue la declaración de las expresiones regulares:

```
id [a-zA-Z_][a-zA-Z_0-9]*
num [0-9]+
```

Luego se definen las reglas o procedimientos auxiliares, las cuales indican el comportamiento de nuestro analizador léxico.

```
"estructura" {return ESTRUCT;}
"inicio" {return INICIO;}
```

```
{character} {
    strcpy(yylval.dir,yytext);
    return CHARACTER;
}
```

La siguiente línea de código es de gran utilidad, pues le indicamos al software que queremos ignorar los espacios, además, posteriormente se hacen declaraciones para no perder de vista los comentarios, pues esto podría causar conflictos con el compilador.

```
[ \n\r\t] {}
```

Por último, la siguiente función nos imprime cualquier error léxico que encuentre:

```
void error(char *s){
    printf("Error léxico: %s\n\n",s);
}
```

NOTA: Los bloques de código mostrados no son los bloques completos, esto por temas de simplicidad.

D. Forma de ejecutar el Programa.

**Windows:** Desde "CMD" debemos llegar a la locación del archivo, posteriormente teclear la siguiente instrucción:

```
C:\Users\      \      \ \Compiladores\Programas Equipo>flex "archivo".l
```

Donde "archivo".l es el archivo que deseamos ejecutar con la respectiva extensión .l.

NOTA: en windows se requiere una variable declarada como:

```
int yylineno = 1;
```