UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO
FACULTAD DE INGENIERIA

“ANALISADOR LEXICO EN FLEX”

PROFESOR:

Adrian Ulises Mercado Martinez

NOMBRE:

Almaguer Rioja Carlos Israel

Ocaña Paredes Kidcia Mireya

Villareal Silva Jose Antonio

MATERIA:

Compiladores

GRUPO:

2

CICLO ESCOLAR: 2020-2

Gramatica

sin: significa sin tipo, car: tipo caracter

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

-Análisis del problema:

Dada la gramática anterior elaboraremos un analizador léxico en “Flex” el cual formara parte de un programa mas grande que funcionara como compilador, para esta primera parte nos encargaremos de analizar a grandes rasgos que estén bien escritas las palabras para esta gramática y que funcione con el lenguaje, lo primero que tenemos que hacer es identificar los terminales de los no terminales ya que vamos a ocupar mayormente los terminales para el análisis Léxico.

-Terminales:

“estructura”,”inicio", “def”, “fin”, “ent”, “real”, “car”, “dreal”, “sin”, “si”, "entonces" “sino”, "mientras" , ”hacer”, “según”,”escribir", “leer”, “devolver”, “terminar”, "caso" “pred”, “o”, “y”, “no”, “verdadero”, “falso”, “,”, “;”, “:”, “:=“, “.”, “(“, “)”, “[“, “]”, ">" “<“, “<=“, “>=“, “<>”, “=“, “+”, “-“, “*”, “/“, “%”, " “, {num},{id}, {caracter}, {cadena}.

-No terminales:

Programa, declaraciones, tipo_registro, tipo, base, tipo_arreglo, lista_var, funciones, argumentos, lista_arg, arg, tipo_arg, param_arr, sentencias, sentencia, casos, predeterminado, e_bool, relaciónal, oprel, expresion, oparit, variable, dato_est_sim, arreglo, parámetros, lista_param.

-Expresiones regulares:

Para el Lexer no sera necesario que todos los terminales tengan expresiones regulares ya que la mayoría son palabras reservadas pero sí necesitaremos de algunas expresiones para poder escribir el código y que lo acepte nuestro analizador.

1.-E.regular para las letras: [a-zA-Z] = letra

2.-E.regular para los dígitos: [0-9] = dígito

1.- E.regular para los “id” : {letra}({letra}|[_]|{dígito})*

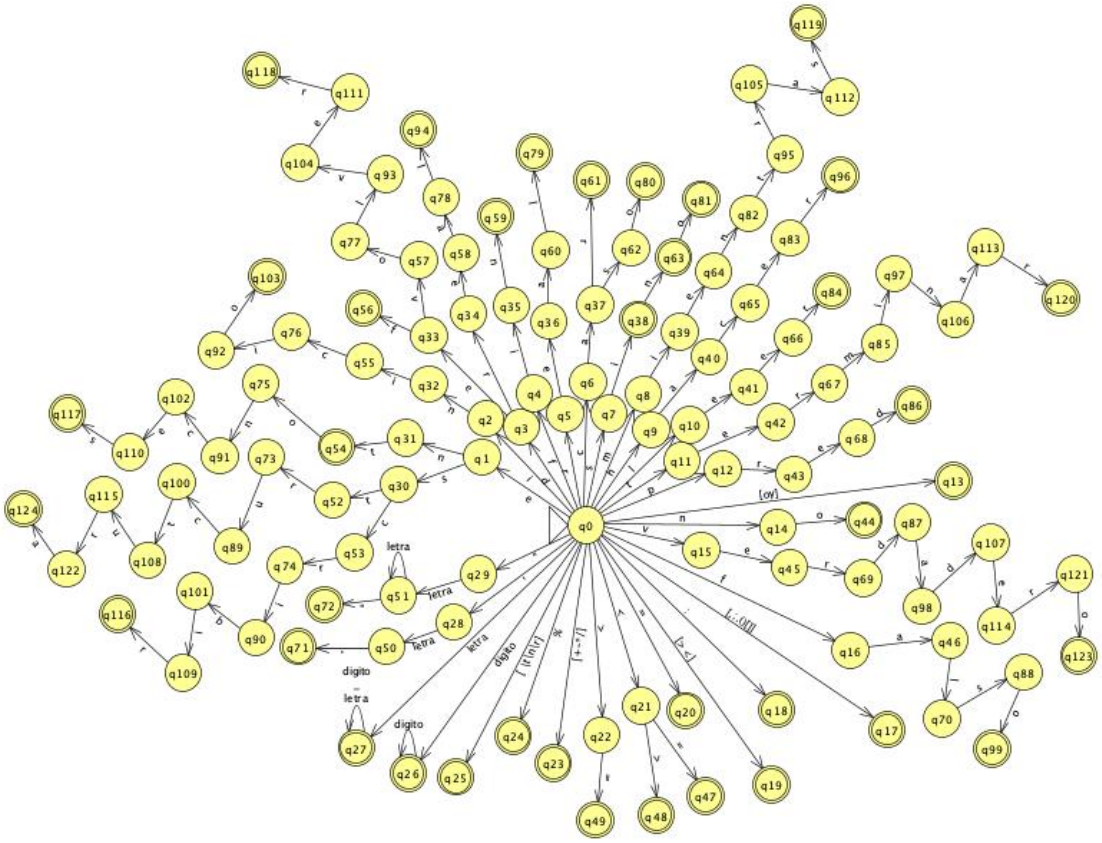
2.- E.regular para las “cadenas” : [“]{letra}+[“]

3.- E.regular para el tabulador, salto de linea etc. : [\n\t\r]+

4.- E.regular para los números : {dígito}+

5.- E.regular para los caracteres : [‘]{letra}[‘]

-AFD resultante:



-Implementación:

Como nuestro programa sera en Flex hay que recordar que el código se divide en tres secciones separadas por “%%” cada una, una vez ya identificados los terminales y no terminales dentro de los terminales hacemos una segunda separación entre las palabras reservadas que vamos a reconocer tal cual están en la gramática y los termínales que pueden ser una serie de opciones que vamos a englobar por medio de expresiones regulares como por ejemplo el “ID”

En la primer sección de definiciones vamos a definir las librerías que utilizaremos así como las expresiones regulares que incluirá nuestro lenguaje principalmente (recuerda que para incluir las librerías es importante poner %{ y %} para incluirlas la principio de nuestro archivo Flex.

En la segunda sección de reglas pondremos todos los terminales de nuestra gramática que previamente seleccionamos y le indicaremos al analizador que hacer en caso de leer alguno de ellos, también debemos de incluir los terminales que definimos en la primera sección pero hay que denotarlos con { y } para que el analizador sepa que tiene que reconocer la expresión regular y no el String como en las palabras reservadas, ejem: {id} {return ID;}.

Finalmente, la sección de código de usuario simplemente se copia a lex.yy.c literalmente, esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por este. La presencia de esta sección es opcional según sea el caso, en nuestro caso únicamente la utilizaremos para llamar a yywrap y le retornaremos un 1.

-Forma de ejecutar el programa:

Para compilar un programa en Flex necesitamos abrir la terminal del ordenador y posicionarnos en la carpeta o dirección donde se encuentre nuestro archivo de Flex, una vez ahi colocamos el siguiente comando: flex lexerF2.l .

Si no existe ningún error esto nos compilara correctamente el archivo, aunque hay que recordar que para ejecutar el programa este tiene que estar junto con un segundo programa .y que se encarga del análisis sintáctico, para fines demostrativos supondremos que tenemos un programa llamado ejemplo.y, una vez teniendo este necesitamos compilarlo con Byacc o Bison según lo hallamos hecho, para ello colocaremos el siguiente comando en terminal: byacc -d ejemplo.y.
Esto nos generara el fichero y.tab.h que contiene los identificadores de los tokens en byacc.

Finalmente si no tenemos errores en los programas debemos de compilarlos con gcc lo cual nos generara un ejecutable el cual contendrá a nuestro analizador.