Expresiones Regulares – Analizador Léxico - FLEX

1. Escribir expresiones regulares para los siguientes elementos devolviendo un par (token/lexema) por cada una de ellas. ¿Existen tokens que generen lexemas diferentes?s
   a. Códigos postales. (Por ej.: 1234, 8366, etc., pero no 422 o 0027)

      S -> ABBB
      A -> [1-9]
      B -> [0-9]

   b. Número de Patentes mercosur

      S -> AAABBBAA
      A->[A-Z]
      B->[0-9]

   c. Comentarios acotados por /* y */. = {letras, *, /}

      C -> "/*".F."*/"
      F -> [a-zA-Z0-9]/*

   d. Identificadores de cualquier longitud que comiencen con una letra y contengan letras, dígitos o guiones. No pueden terminar con guión

      I -> L F* L+
      L -> [a-zA-Z]
      F -> [a-zA-Z0-9]-

   e. Idem anterior pero que no contengan dos guiones seguidos

      I -> L (F? | L | N)* (L | N)+ (F? | L | N)* (L | N)+
      L -> [a-zA-Z]
      F -> [a-zA-Z0-9]-
      N -> [0-9]

   f. Constantes en otras bases como las del lenguaje C

      N -> (C | H)+
      C -> [0-9]
      H -> [A-F]

   g. Constantes aritméticas enteras. Controlar el rango permitido

      C -> [0-9]+

   h. Constantes reales con formato xx.xx Controlar el rango permitido.

      N -> C* "." C+
      C -> [0-9]+

i. Constantes string de la forma “texto “

S -> [a-zA-Z]+

j. Palabras reservadas (IF-WHILE-DECVAR-ENDDEC-INTEGER-FLOAT-WRITE)

S -> I | W | D | E | N | F | R
I -> “IF”
W -> “WHILE”
D -> “DECVAR”
E -> “ENDDEC”
N -> “INTEGER”
F -> “FLOAT”
R -> “WRITE”

k. Operadores lógicos y operadores aritméticos básicos

OPL -> MEN | MAY | MEI | MAI | DIS | EQL
ARI -> MAS | MEN | POR | DIV
MAS -> “+”
MEN -> “-”
POR -> “*”
DIV -> “/”
MEN -> “<”
MAY -> “>”
MEI -> “<=”
MAI -> “>=”
DIS -> “<>”
EQL -> “==”

l. Constantes en otras bases como en el lenguaje C

2. Dado el siguiente fragmento de código, determinar los elementos léxicos y escribir una expresión regular para cada uno de éstos

```
DECVAR
	contador: Integer;
	promedio: Float;
	actual, suma: Float;
ENDDEC
write "Hola mundo!";
contador: 0;
actual: 999;
suma: 02;
contador: contador+1;
while (contador <= 92) {
	contador: contador + 1;
	actual: (contador/0.342) + (contador*contador);
	suma: suma + actual;
}
write "La suma es: ";
write suma;
if (actual > 2){
		write "2 > 3";
	}
	if (actual < 3){
	if(actual >= 3){
```

```
                write "soy true";
        }
        if(actual <= 3){
                write "soy true";
        }
        if(actual != 3){
                write "soy true";
        }
        if(actual == 3){
                write "soy true";
        }
}else{
        actual:333.3333;
}
```

integer -> [0-9]+
float -> [0-9]*”.”[0-9]+
cte_integer -> integer
cte_float -> float
id -> letra (letra | digito)*
operacion -> < | >= | <= | <> | ==
palabra -> if | else | while | decvar | endvar
operador -> + | - | * | /
p_a -> (
p_c ->)
letra -> [a-zA-Z]
digito -> [0-9]
pyc -> ;
asig -> :=

3. Sobre el fragmento anterior, escribir una tabla de símbolos para los lexemas que correspondan

integer -> [0-9]+
float -> [0-9]*”.”[0-9]+
cte_integer -> integer
cte_float -> float
id -> letra (letra | digito)*
operacion -> < | >= | <= | <> | ==
palabra -> if | else | while | decvar | endvar
operador -> + | - | * | /
p_a -> (
p_c ->)
letra -> [a-zA-Z]
digito -> [0-9]
pyc -> ;
asig -> :=

4. Generar un archivo FLEX para probar el fragmento de código del ejercicio anterior.

Gramáticas - Analizador Sintáctico - BISON

5. Desarrollar una gramática en formato BNF para reconocer expresiones aritméticas simples (sumas, restas, multiplicaciones y divisiones) que operan con constantes enteras e identificadores (Utilizar tokens para los elementos léxicos)

P´ -> P	**<programa´>::=<programa>**
P -> E	**<programa>::=<expresion>**
E -> E + F	**<expresion>::=<expresion>+<factor>**
E -> E - F	**<expresion>::=<expresion>-<factor>**
E -> E * F	**<expresion>::=<expresion>*<factor>**
E -> E / F	**<expresion>::=<expresion>/<factor>**
E -> F	**<expresion>::=<factor>**
F -> id	**<factor>::=id**
F -> cte	**<factor>::=cte**
F -> (E)	**<factor>::=(<expresion>)**

6. Desarrollar una gramática igual a la anterior donde la multiplicación y división tenga prioridad sobre la suma y la resta.

P´ -> P	**<programa´>::=<programa>**
P -> E	**<programa>::=<expresion>**
E -> E + T	**<expresion>::=<expresion>+<termino>**
E -> E - T	**<expresion>::=<expresion>-<termino>**
E -> T	**<expresion>::=<termino>**
T -> T * F	**<termino>::=<termino>*<factor>**
T -> T / F	**<termino>::=<termino>/<factor>**
T -> F	**<termino>::=<factor>**
F -> id	**<factor>::=id**
F -> cte	**<factor>::=cte**

7. Agregar a la gramática anterior la posibilidad de incorporar expresiones entre paréntesis.

P´ -> P	**<programa´>::=<programa>**
P -> E	**<programa>::=<expresion>**
E -> E + T	**<expresion>::=<expresion>+<termino>**
E -> E - T	**<expresion>::=<expresion>-<termino>**
E -> T	**<expresion>::=<termino>**
T -> T * F	**<termino>::=<termino>*<factor>**
T -> T / F	**<termino>::=<termino>/<factor>**
T -> F	**<termino>::=<factor>**
F -> id	**<factor>::=id**
F -> cte	**<factor>::=cte**
F -> (E)	**<factor>::=(<expresion>)**

8.

   a. Desarrollar una BNF/GLC para reconocer asignaciones simples. Estas deben tener un Left Side y un Right Side. El Left Side debe ser un identificador y el Right Side una expresión aritmética simple

P´ -> P P -> A; A -> id=E E -> E + T E -> E - T E -> T T -> T * F T -> T / F T -> F F -> id F -> cte F -> (E)	<programa´>::=<programa> <programa>::=<asignacion>; <asignacion>::=id=<expresion> <expresion>::=<expresion>+<termino> <expresion>::=<expresion>-<termino> <expresion>::=<termino> <termino>::=<termino>*<factor> <termino>::=<termino>/<factor> <termino>::=<factor> <factor>::=id <factor>::=cte <factor>::=(<expresion>)

b. Hacer un árbol de parsing para el siguiente programa

actual:= (contador/342) + (contador*contador);

Estado 00 P´->.P P->.A; A->.id=E	Estado 01 (0 P) P´->P.	Estado 02 (0 A) P->A.;	Estado 03 (0 id) A->id.=E
Estado 04 (2 ;) P->A;.	Estado 05 (3 =) A->id=.E E -> .E+T E -> .E-T E -> .T T -> .T*F T -> .T/F T -> .F F -> .id F -> .cte F -> .(E)	Estado 06 (5 E) A->id=E. E -> E.+T E -> E.-T	Estado 07 (5 T)(11 T) E -> T. T -> T.*F T -> T./F
Estado 08 (5 F) (11 F) (12 F) (13 F) T -> F.	Estado 09 (5 id) (11 id) (12 id) (13 id) (14 id) (15 id) F -> id.	Estado 10 (5 cte) (11 cte) (12 cte) (13 cte) (14 cte) (15 cte) F -> cte.	Estado 11 (5 () (11 () (12 () (13 () (14 () (15 () F -> (.E) E -> .E+T E -> .E-T E -> .T T -> .T*F T -> .T/F T -> .F F -> .id F -> .cte F -> .(E)
Estado 12 (6 +) (16 +) E -> E+.T T -> .T*F T -> .T/F T -> .F F -> .id F -> .cte F -> .(E)	Estado 13 (6 -)(16 -) E -> E-.T T -> .T*F T -> .T/F T -> .F F -> .id F -> .cte F -> .(E)	Estado 14 (7 *)(17 *)(18 *) T -> T*.F F -> .id F -> .cte F -> .(E)	Estado 15 (7 /)(17 /)(18 /) T -> T/.F F -> .id F -> .cte F -> .(E)
Estado 16 (11 E) F -> (E.) E -> E.+T E -> E.-T	Estado 17 (12 T) E -> E+T. T -> T.*F T -> T./F	Estado 18 (13 T) E -> E-T. T -> T.*F T -> T./F	Estado 19 (14 F) T -> T*F.
Estado 20 (15 F) T -> T/F.	Estado 21 (16)) F -> (E).	PRIM(P)= PRIM(A)= PRIM(E)= PRIM(T)=	SGT(P)=$ SGT(A)=; SGT(E)=;,+,-,) SGT(T)=;,+,-,*,/,)

		PRIM(F)=	SGT(F)=;,+,-,*,/,)
GOTO	**DESPLAZAMIENTOS**		**GRAMÁTICA Y ESTADOS**
01 (00 P) 02 (00 A) 06 (05 E) 07 (05 T) 07 (11 T) 08 (05 F) 08 (11 F) 08 (12 F) 08 (13 F) 16 (11 E) 17 (12 T) 18 (13 T) 19 (14 F) 20 (15 F)	03 (00 id) 04 (02 ;) 05 (03 =) 09 (05 id) 09 (11 id) 09 (12 id) 09 (13 id) 09 (14 id) 09 (15 id) 10 (05 cte) 10 (11 cte) 10 (12 cte) 10 (13 cte) 10 (14 cte) 10 (15 cte) 11 (05 () 11 (11 ()	11 (11 () 11 (12 () 11 (13 () 11 (14 () 11 (15 () 12 (06 +) 12 (16 +) 13 (06 -) 13 (16 -) 14 (07 *) 14 (17 *) 14 (18 *) 15 (07 /) 15 (17 /) 15 (18 /) 21 (16))	R01 E01 P´ -> P R02 E02 P -> A; R03 E06 A -> id=E R04 E17 E -> E + T R05 E18 E -> E - T R06 E07 E -> T R07 E19 T -> T * F R08 E20 T -> T / F R09 E08 T -> F R10 E09 F -> id R11 E10 F -> cte R12 E21 F -> (E)

	P	A	E	T	F	id	;	=	cte	(	)	+	-	*	/	$
00	01	02				D03										
01																OK
02							D04									
03								D05								
04																R02
05			06	07	08	D09			D10	D11						
06							R03					D12	D13			
07							R06				R06	R06	R06	D14	D15	
08							R09				R09	R09	R09	R09	R09	
09							R10				R10	R10	R10	R10	R10	
10							R11				R11	R11	R11	R11	R11	
11			16	07	08	D09			D10	D11						
12				17	08	D09			D10	D11						
13				18	08	D09			D10	D11						
14					19	D09			D10	D11						
15					20	D09			D10	D11						
16											D21	D12	D13			
17							R04				R04	R04	R04	D14	D15	
18							R05				R05	R05	R05	D14	D15	
19							R07				R07	R07	R07	R07	R07	
20							R08				R08	R08	R08	R08	R08	
21							R12				R12	R12	R12	R12	R12	

id=(id/cte)+(id*id);

00id03=05(11id09 00id03=05(11F08 00id03=05(11T07/15cte10 00id03=05(11T07/15F20 00id03=05(11T07 00id03=05(11E16)21 00id03=05F08	00id03=05T07 00id03=05E06+12(11id09 00id03=05E06+12(11F08 00id03=05E06+12(11T07*14id09 00id03=05E06+12(11T07*14F19 00id03=05E06+12(11T07*14F19 00id03=05E06+12(11T07	00id03=05E06+12(11E16)21 00id03=05E06+12F8 00id03=05E06+12T17 00id03=05E06 00A02;04 00P01OK

Lista de Reglas

10,09,11,08,06,12,09,06,10,09,10,07,06,12,09,04,03,02,01

9. En algunos lenguajes de programación se permite la asignación múltiple, en la que varias variables pueden recibir el mismo valor. Los que siguen son algunos ejemplos de sentencias en los que se hace uso de esta forma de escritura:

```
a:=inc:=minimo:=expresion;
total:=precio:=expresión;
```

   a. Escriba en BNF la sintaxis de este tipo de asignaciones con las siguientes condiciones: la sentencia debe terminar con ";" y el valor de la derecha debe ser una expresión como la desarrollada en el ejercicio 7

P´ -> P	**<programa´>::=<programa>**
P -> L;	**<programa>::=<lista>;**
L -> L=E	**<lista>::=<lista>=<expresion>**
L -> E	**<lista>::=<expresion>**
E -> E+T	**<expresion>::=<expresion>+<termino>**
E -> E-T	**<expresion>::=<expresion>-<termino>**
E -> T	**<expresion>::=<termino>**
T -> T*F	**<termino>::=<termino>*<factor>**
T -> T/F	**<termino>::=<termino>/<factor>**
T -> F	**<termino>::=<factor>**
F -> id	**<factor>::=id**
F -> cte	**<factor>::=cte**
F -> (E)	**<factor>::=(<expresion>)**

Estado 00 **P´-> .P** **P -> .L;** **L -> .L=E** **L -> .E** **E -> .E+T** **E -> .E-T** **E -> .T** **T -> .T*F** **T -> .T/F** **T -> .F** **F -> .id** **F -> .cte** **F -> .(E)**	**Estado 01 (00 P)** **P´-> P.**	**Estado 02 (00 L)** **P -> L.;** **L -> L.=E**	**Estado 03 (00 E)** **L -> E.** **E -> E.+T** **E -> E.-T**
Estado 04 (00 T)(08 T)(10 T) **E -> T.** **T -> T.*F** **T -> T./F**	**Estado 05 (00 F)(08 F)(10 F)(11 F)(12 F)** **T -> F.**	**Estado 06 (00 id)(08 id)(10 id)(11 id)(12 id)(13 id)(14 id)** **F -> id.**	**Estado 07 (00 cte)(08 cte)(11 cte)(12 cte)(13 cte)(14 cte)** **F -> cte.**
Estado 08 (00 ()(10 ()(11 ()(12 ()(13 ()(14 () **F -> (.E)** **E -> .E+T** **E -> .E-T** **E -> .T** **T -> .T*F** **T -> .T/F** **T -> .F** **F -> .id** **F -> .cte** **F -> .(E)**	**Estado 09 (02 ;)** **P -> L;.**	**Estado 10 (02 =)** **L -> L=.E** **E -> .E+T** **E -> .E-T** **E -> .T** **T -> .T*F** **T -> .T/F** **T -> .F** **F -> .id** **F -> .cte** **F -> .(E)**	**Estado 11 (03 +)(15 +)(16 +)** **E -> E+.T** **T -> .T*F** **T -> .T/F** **T -> .F** **F -> .id** **F -> .cte** **F -> .(E)**
Estado 12 (03 -)(15 -)(16 -) **E -> E-.T** **T -> .T*F** **T -> .T/F**	**Estado 13 (04 *)(17 *)(18 *)** **T -> T*.F** **F -> .id** **F -> .cte**	**Estado 14 (04 /)(17 /)(18 /)** **T -> T/.F** **F -> .id** **F -> .cte**	**Estado 15 (08 E)** **F -> (E.)** **E -> E.+T** **E -> E.-T**

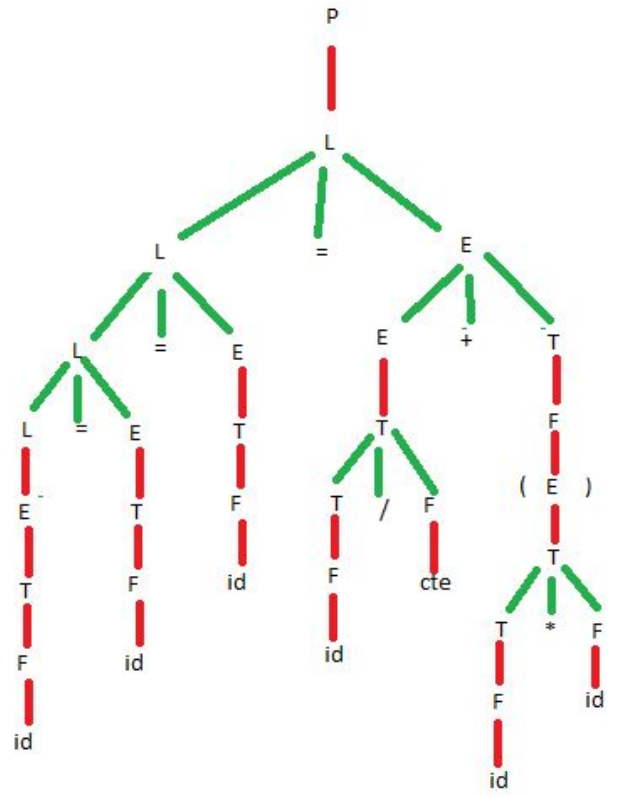
T -> .F F -> .id F -> .cte F -> .(E)	F -> .(E)	F -> .(E)	
Estado 16 (10 E) L -> L=E. E -> E.+T E -> E.-T	Estado 17 (11 T) E -> E+T. T -> T.*F T -> T./F	Estado 18 (12 T) E -> E-T. T -> T.*F T -> T./F	Estado 19 (13 F) T -> T*F.
Estado 20 (14 F) T -> T/F.	Estado 21 (15)) F -> (E).	PRI(P)={id} PRI(L)={id,cte,(} PRI(E)={id,cte,(} PRI(T)={id,cte,(} PRI(F)={id,cte,(	SGT(P)={$} SGT(L)={;,=} SGT(E)={;,=,+,-,)} SGT(T)={;,=,+,-,),*,/} SGT(F)={;,=,+,-,),*,/}
GOTO	DESPLAZAMIENTOS		GRAMÁTICA Y ESTADOS
01 (00 P) 02 (00 L) 03 (00 E) 04 (00 T) 04 (08 T) 04 (10 T) 05 (00 F) 05 (08 F) 05 (10 F) 05 (11 F) 05 (12 F) 15 (08 E) 16 (10 E) 17 (11 T) 18 (12 T) 19 (13 F) 20 (14 F)	06 (00 id) 06 (08 id) 06 (10 id) 06 (11 id) 06 (12 id) 06 (13 id) 06 (14 id) 07 (00 cte) 07 (08 cte) 07 (11 cte) 07 (12 cte) 07 (13 cte) 07 (14 cte) 08 (00 (08 (10 (08 (11 (08 (12 (	08 (13 (08 (14 (09 (02 ;) 10 (02 =) 11 (03 +) 11 (15 +) 11 (16 +) 12 (03 -) 12 (15 -) 12 (16 -) 13 (04 *) 13 (17 *) 13 (18 *) 14 (04 /) 14 (17 /) 14 (18 /) 21 (15))	R01 E01 P´-> P R02 E08 P -> L; R03 E16 L -> L=E R04 E02 L -> E R05 E17 E -> E+T R06 E18 E -> E-T R07 E03 E -> T R08 E19 T -> T*F R09 E20 T -> T/F R10 E05 T -> F R11 E06 F -> id R12 E07 F -> cte R13 E21 F -> (E)

	P	L	E	T	F	id	cte	=	;	(	)	+	-	*	/	$	R
00	01	02	03	04	05	D06	D07			D08							
01																OK	R01
02								R04/D 10	R04/D 09								R04
03								R07	R07		R07	R07/D 11	R07/D 12				R07
04														D13	D14		
05								R10	R10		R10	R10	R10	R10	R10		R10
06								R11	R11		R11	R11	R11	R11	R11		R11
07								R12	R12		R12	R12	R12	R12	R12		R12
08			15	04	05	D06	D07			D08						R02	R02
09																	
10			16	04	05	D06	D07			D08							
11				17	05	D06	D07			D08							
12				18	05	D06	D07			D08							
13					19	D06	D07			D08							
14					20	D06	D07			D08							
15												D11	D12			D21	
16								R03	R03			D11	D12				R03
17								R05	R05		R05	R05	R05	D13	D14		R05
18								R06	R06		R06	R06	R06	D13	D14		R06
19								R08	R08		R08	R08	R08	R08	R08		R08
20								R09	R09		R09	R09	R09	R09	R09		R09
21								R13	R13		R13	R13	R13	R13	R13		R13

b. Hacer el árbol de parsing para el siguiente programa:

actual:=promedio:=contador:= promedio/ 342 + (contador*contador);

id=id=id=id/cte+(id*id);

F -> id **T -> F** **E -> T** **L -> E** **F -> id** **T -> F** **E -> T** **L -> L=E** **F -> id** **T -> F** **E -> T** **L -> L=E** **F -> id** **T -> F** **F -> cte** **T -> T/F** **E -> T** **F -> id** **T -> F** **F -> id** **T -> T*F** **E -> T** **F -> (E)** **T -> F** **E -> E+T** **L -> L=E** **P -> L; OK** **Bison OK ¿?**	

10. Definir una BNF/GLC que soporte programas que tengan asignaciones simples como las del ejercicio 8 y asignaciones múltiples como las del ejercicio 9. Estos programas pueden tener una sentencia o muchas sentencias. Cada sentencias finaliza con ; Los programas de este tipo deben comenzar con la palabra reservada INICIO y finalizar con FIN.

P´-> P	**<programa´>::=<programa>**
P -> inicio B fin	**<programa>::=inicio<bloque>fin;**
B -> B S;	**<bloque>:=<bloque><sentencia>;**
B -> S;	**<bloque>:=<sentencia>;**
S -> L	**<sentencia>:=<lista>**
L -> id=L	**<lista>::id=<lista>**
L -> id=E	**<lista>::id=<expresion>**
E -> E + T	**<expresion>::=<expresion>+<termino>**
E -> E - T	**<expresion>::=<expresion>-<termino>**
E -> T	**<expresion>::=<termino>**
T -> T * F	**<termino>::=<termino>*<factor>**
T -> T / F	**<termino>::=<termino>/<factor>**
T -> F	**<termino>::=<factor>**
F -> id	**<factor>::=id**
F -> cte	**<factor>::=cte**
F -> (E)	**<factor>::=(<expresion>)**

11. Utilizar BISON y FLEX para probar el siguiente programa con la gramática del ejercicio anterior

```
INICIO
        contador:= 0;
        actual:=contador:= 999;
        contador: contador+1;
        contador: contador + 1/ (actual *otra);
        actual: (contador/342) + (contador*contador);
        actual:=promedio:=contador:=342;
FIN
```

Reglas que aplica Bison

F -> cte **T -> F** **E -> T** **L -> id=E** **S -> L** **B -> S;** **F -> cte** **T -> F** **E -> T** **L -> id=E** **L -> id=L** **S -> L** **B -> B S;** **F -> id** **T -> F** **E -> T** **F -> cte** **T -> F** **E -> E+T** **L -> id=E** **S -> L** **B -> B S;** **F -> id**	**T -> F** **E -> T** **F -> cte** **T -> F** **F -> id** **T -> F** **F -> id** **T -> T*F** **E -> T** **F -> (E)** **T -> T/F** **E -> E+T** **L -> id=E** **S -> L** **B -> B S;** **F -> id** **T -> F** **F -> cte** **T -> T/F** **E -> T** **F -> (E)** **T -> F** **E -> T**	**F -> id** **T -> F** **F -> id** **T -> T*F** **E -> T** **F -> (E)** **T -> F** **E -> E+T** **L -> id=E** **S -> L** **B -> B S;** **F -> cte** **T -> F** **E -> T** **L -> id=E** **L -> id=L** **L -> id=L** **S -> L** **B -> B S;** **P -> inicio L fin; OK**

Gramáticas – Análisis Sintáctico - BISON

12. Dada la siguiente gramática con elementos terminales fin , := y exp

```
P -> L fin
L -> L , I
L -> I
I -> id := exp
```

a. Hacer el parsing ascendente

P -> L fin **L -> L , I** **L -> I** **I -> id := exp**	**<programa>::=<lista>fin** **<lista>::=<lista>,<indice>** **<lista>::=<indice>** **<indice>::=id=exp**

Estado 00 **P -> .L fin** **L -> .L , I** **L -> .I** **I -> .id = exp**	**Estado 01 (00 L)** **P -> L .fin** **L -> L ., I**	**Estado 02 (00 I)** **L -> I.**	**Estado 03 (00 id)(05 id)** **I -> id .= exp**
Estado 04 (01 fin) **P -> L fin.**	**Estado 05 (01 ,)** **L -> L , .I** **I -> .id = exp**	**Estado 06 (03 =)** **I -> id = .exp**	**Estado 07 (05 I)** **L -> L , I.**

Estado 08 (06 exp) **I -> id = exp.**		**PRIM(P)=id** **PRIM(L)=id** **PRIM(I)=id;,**	**SGT(P)=$** **SGT(L)=fin** **SGT(I)=fin**
GOTO	**DESPLAZAMIENTOS**		**GRAMÁTICA Y ESTADOS**
01 (00 L) **02 (00 I)** **07 (05 I)**	**03 (00 id)** **03 (05 id)** **04 (01 fin)**	**05 (01 ,)** **06 (03 =)** **08 (06 exp)**	**R1 E04 P -> L fin** **R2 E07 L -> L , I** **R3 E02 L -> I** **R4 E08 I -> id := exp**

	P	L	I	id	fin	,	=	exp	$
00		01		D03					
01			02		D04				
02					R3				
03									
04									OK
05			07	D03					
06									
07					R2				
08					R4				

b. Hacer el árbol de parsing y devolver la lista de reglas para la hilera :

id,id,id,id:=exp;

00id03x fail		

Nuevo Ejercicio

P -> L E fin **L -> L , id** **L -> id** **E -> := exp**	**<programa>::=<lista><expresion>fin** **<lista>::=<lista>,id** **<lista>::=id** **<expresion>::=:=exp**

Estado 00 **P -> .L E fin** **L -> .L , id** **L -> .id**	**Estado 01 (00 L)** **P -> L .E fin** **L -> L ., id** **E -> .:= exp**	**Estado 02 (00 id)** **L -> id.**	**Estado 03 (01 E)** **P -> L E .fin**
Estado 04 (01 ,) **L -> L , .id**	**Estado 05 (01 :=)** **E -> := .exp**	**Estado 06 (03 fin)** **P -> L E fin.**	**Estado 07 (04 id)** **L -> L , id.**
Estado 08 (05 exp) **E -> := exp.**		**PRIM(P)=id** **PRIM(L)=id** **PRIM(E)=:=**	**SGT(P)=$** **SGT(L)=,;fin** **SGT(E)=fin**
GOTO	**DESPLAZAMIENTOS**		**GRAMÁTICA Y ESTADOS**
01 (00 L) **03 (01 E)**	**02 (00 id)** **04 (01 ,)** **05 (01 :=)** **06 (03 fin)**	**07 (04 id)** **08 (05 exp)**	**R01 E06 P -> L E fin** **R02 E07 L -> L , id** **R03 E02 L -> id** **R04 E08 E -> := exp**

	P	L	E	id	fin	,	:=	exp	$	reglas
00		1		D2						
01			3			D4	D5			
02					R3	R3				R3
03					D6					
04				D7						

05								D8		
06									OK	R1
07					R2	R2	R2XX			R2
08					R4					R4

id,id,id,id:=exp;

0id2 0L1,4id7 0L1,4id7 0L1,4id7	0L1=5exp8 0L1E3fin6 OK	**Lista de Reglas** **03,02,02,02,04,01**

c. Probar con FLEX y BISON y comparar la lista de reglas obtenida

13. Dada la siguiente gramática

R1: S (L)
R2: S- ( )
R3: L (L)
R4: L L ( )
R5: L ( )

a. Hacer el parsing ascendente

R0 S´-> S **R1 S -> (L)** **R2 S -> ( )** **R3 L -> (L)** **R4 L -> L ( )** **R5 L -> ( )**	**<sentencia>::=(<lista>)** **<sentencia>::=()** **<lista>::=(<lista>)** **<lista>::=<lista>()** **<lista>::=()**

Estado 00 **S -> .(L)** **S -> .( )**	**Estado 01 (00 ()** **S -> (.L)** **S -> (.)** **L -> .(L)** **L -> .L ( )** **L -> .( )**	**Estado 02 (01 L)** **S -> (L .)** **L -> L .( )**	**Estado 03 (01))** **S -> ( ).**
Estado 04 (01 ()(04 () **L -> (.L)** **L -> (.)** **L -> .(L)** **L -> .L ( )** **L -> .( )**	**Estado 05 (02))** **S -> (L).**	**Estado 06 (02 ()(07 ()** **L -> L (.)**	**Estado 07 (04 L)** **L -> (L .)** **L -> L .( )**
Estado 08 (04)) **L -> ( ).**	**Estado 09 (06))** **L -> L ( ).**	**Estado 10 (07))** **L -> (L).**	**PRIM(S)=(** **PRIM(L)=(** **SGT(S)=)** **SGT(L)=)**
GOTO	**DESPLAZAMIENTOS**		**GRAMÁTICA Y ESTADOS**
02 (01 L) **07 (04 L)**	**01 (00 ()** **03 (01))** **04 (01 ()** **04 (04 ()** **05 (02))**	**06 (02 ()** **06 (07 ()** **08 (04))** **09 (06))** **10 (07))**	**R0 E00 S´-> S** **R1 E05 S -> (L)** **R2 E03 S -> ( )** **R3 E10 L -> (L)** **R4 E09 L -> L ( )** **R5 E08 L -> ( )**

b. Hacer el árbol de parsing y devolver la lista de reglas para la hilera : (((( ))) ( ) ( ))

	S	L	(	)	reglas
00			D01	R00	R0
01		02	D04	D03	
02			D06	D05	
03				R02	R2
04		07		D08	
05				R01	R1
06				D09	
07			D06	D10	
08				R05	R5
09				R04	R4
10				R03	R3

(((( ))) ( ) ( ))

00(01(04X	**(( ))) ( ) ( ))**	**Lista de Reglas**

c. Probar con FLEX y BISON y comparar la lista de reglas obtenida

14. Dada la gramática del ejercicio 8
    a. Hacer el parsing ascendente
    b. Hacer el árbol de parsing y devolver la lista de reglas para la hilera

actual:= (contador/342) + (contador*contador);

id=(id/cte)+(id*id);

00id03=05(11id09 00id03=05(11F08 00id03=05(11T07/15cte10 00id03=05(11T07/15F20 00id03=05(11T07 00id03=05(11E16)21 00id03=05F08	00id03=05T07 00id03=05E06+12(11id09 00id03=05E06+12(11F08 00id03=05E06+12(11T07*14id09 00id03=05E06+12(11T07*14F19 00id03=05E06+12(11T07*14F19 00id03=05E06+12(11T07	00id03=05E06+12(11E16)21 00id03=05E06+12F8 00id03=05E06+12T17 00id03=05E06 00A02;04 00P01OK

Lista de Reglas
10,09,11,08,06,12,09,06,10,09,10,07,06,12,09,04,03,02,01

c. Probar con FLEX y BISON y comparar la lista de reglas obtenida

15. Se tiene un lenguaje que soporta 2 tipo de variables: enteras y strings, y soporta también constantes enteras y string. El mismo posee las siguientes características:
    a. Las variables enteras deben terminar con el carácter %
    b. las variables string deben terminar con $
    c. las constantes enteras de mas de un digito no pueden comenzar con 0
    d. las constantes string deben ir entre comillas [“]
    e. Las variables y constantes enteras se pueden sumar, restar, multiplicar y dividir libremente entre si.
    f. Las variables y constantes strings pueden concatenarse libremente entre si con el símbolo “++”

g. No esta permitido mezclar los tipos en las operaciones
h. Las asignaciones deben ser entre elementos del mismo tipo

Escribir una BNF/GLC que describa dicho lenguaje

A->A=ENT	**<asignacion>:=<asignacion>=<entero>**
A->ENT	**<asignacion>:=<entero>**
ENT->CTE	**<entero>:=constante_entero**
ENT->IDE	**<entero>:=id_entero**
S->S++P	**<sentencia>:=<sentencia>++<palabra>**
S->P	**<sentencia>:=<palabra>**
P->IDS	**<palabra>:=<id_string**
P->CTS	**<palabra>:=constante_string**
E->E+T	**<expresion>:=<expresion>+<termino>**
E->E-T	**<expresion>:=<expresion>-<termino>**
E->T	**<expresion>:=<termino>**
T->T*F	**<termino>:=<termino>*<factor>**
T->T/F	**<termino>:=<termino>/<factor>**
T->F	**<termino>:=<factor>**
F->IDE	**<factor>:=id_entero**
F->CTE	**<factor>:=constante_entero**
CTS->”CAD”	**REGEX constante_string**
CTE->DI1NUM	**REGEX constante_entero**
IDS->CAD$	**REGEX id_string**
IDE->NUM%	**REGEX id_entero**
DIG->[0-9]	**REGEX digito**
DI1->[1-9]	**REGEX digitos_sin_cero**
LET->[a-zA-Z]	**REGEX letra**
CAR->[DIG-LET]	**REGEX cadena_regular**
NUM->DIG+	**REGEX numero_entero**
CAD->CAR+	**REGEX cadena**

16. Genere una BNF/GLC para un lenguaje que soporta:
    a. Asignaciones simples de expresiones
    b. Sentencia IF
    c. Sentencia WHILE
    d. Sentencia WRITE (constante string)
    e. Comienza y Termine con las palabras INICIO y FIN
    f. Separador de sentencias ;
    g. Separador de bloques { }
    h. Operador de asignación :
    i. Condiciones : dos expresiones con un solo operador

P´->P	**<programa´>:=<programa>**
P->inicioHfin	**<programa>:=INICIO<hilera>FIN**
H->HS;	**<hilera>:=<asignacion>;**
H->S;	**<hilera>:=<sentencia>;**
S->id:E	**<sentencia>:=id:<expresion>**
S->whileC{H}	**<sentencia>:=WHILE<condicion>{<hilera>}**
S->ifC{H}	**<sentencia>:=IF<condicion>{<hilera>}**
S->write(V)	**<sentencia>:=WRITE(<variable_string>)**
C->FOF	**<condicion>:=<factor><operador><factor>**
O->==	**<operador>:===**
O-><	**<operador>:=<**
O->>	**<operador>:=>**

O-><>	**<operador>:=<></operador>**
O-><=	**<operador>:=<=></operador>**
O->>=	**<operador>:=>=></operador>**
E->E+T	**<expresion>:=<expresion>+<termino></termino></expresion></expresion>**
E->E-T	**<expresion>:=<expresion>-<termino></termino></expresion></expresion>**
E->T	**<expresion>:=<termino></termino></expresion>**
T->T*F	**<termino>:=<termino>*<factor></factor></termino></termino>**
T->T/F	**<termino>:=<termino>/<factor></factor></termino></termino>**
T->F	**<termino>:=<factor></factor></termino>**
F->id	**<factor>:=id</factor>**
F->cte	**<factor>:=cte</factor>**
F->(E)	**<factor>:=(<expresion>)</expresion></factor>**
V->ids	**<variable_string>:=id_string</variable_string>**

17. Probar con FLEX y BISON el siguiente programa. Agregar, si es necesario nuevas expresiones regulares.

```
INICIO
	contador: 0;
	actual: 999;
	suma: 02;
	contador: contador+1;
	while (contador <= 92) {
		contador: contador + 1;
		actual: (contador/0.342) + (contador*contador);
		suma: suma + actual;
	}
	if (actual > 2){
		write "2 > 3";
	}
	if (actual < 3){
		if(actual >= 3){
			write "soy true";
		}
		if(actual <= 3){
			write "soy true";
		}
		if(actual != 3){
			write "soy true";
		}
		if(actual == 3){
			write "soy true";
		}
	}else{
		actual:333.3333;
	}
FIN
```

18. Agregar al lenguaje del ejercicio anterior la posibilidad de definir sentencias en las que se pueda incorporar la función standard AVERAGE
    a. Formato de AVERAGE : AVG(Lista de expresiones) donde Lista de expresiones es una lista de expresiones aritméticas separdas por comas dentro de dos corchetes
    b. AVERAGE debe operar dentro de una expresión

P´->P	**<programa´>:=<programa>**
P->inicioHfin	**<programa>:=INICIO<hilera>FIN**
H->HS;	**<hilera>:=<hilera><sentencia>;**
H->S;	**<hilera>:=<sentencia>;**
S->id:E	**<sentencia>:=id:<expresion>**
S->whileC{H}	**<sentencia>:=WHILE<condicion>{<hilera>}**
S->ifC{H}	**<sentencia>:=IF<condicion>{<hilera>}**
S->write(V)	**<sentencia>:=WRITE(<variable_string>)**
S->avg(L)	**<sentencia>:=AVG(<lista>)**
L-> L,E	**<lista>:=<lista>,<expresion>**
L->E	**<lista>:=<expresion>**
C->FOF	**<condicion>:=<factor><operador><factor>**
O->==	**<operador>:===**
O-><	**<operador>:=<**
O->>	**<operador>:=>**
O-><>	**<operador>:=<>**
O-><=	**<operador>:=<=**
O->>=	**<operador>:=>=**
E->E+T	**<expresion>:=<expresion>+<termino>**
E->E-T	**<expresion>:=<expresion>-<termino>**
E->T	**<expresion>:=<termino>**
T->T*F	**<termino>:=<termino>*<factor>**
T->T/F	**<termino>:=<termino>/<factor>**
T->F	**<termino>:=<factor>**
F->id	**<factor>:=id**
F->cte	**<factor>:=cte**
F->(E)	**<factor>:=(<expresion>)**
V->ids	**<variable_string>:=id_string**

19. Probar con FLEX y BISON el siguiente programa

```
INICIO
	write "Hola mundo!";
	contador: =0;
	actual: 99999999999999;
	suma: 02;
	contador: contador+1;
	while (contador <= 92) {
		contador: contador + 1;
		actual: (contador/0.342) + (contador*contador);
		suma: suma + actual * avg([3,3*4,12,actual,actual*2, actual*actual]);
	}
	write "La suma es: ";
	write suma;
	if (actual > 2){
		write "2 > 3";
	}

	if (actual < 3){
		if(actual >= 3){
			write "soy true";
		}
		if(actual <= 3){
			write "soy true";
		}
		if(actual != 3){
```

```
                write "soy true";
        }
        if(actual == 3){
                write "soy true";
        }
    }else{
        actual:333.3333;
    }
FIN
```

20. Agregar al lenguaje anterior la posibilidad de tener un sector de declaraciones delimitado por las palabras reservadas DECVAR y END. Dentro de este bloque las variables se declaran de la forma: Id: Tipo; o id,id,.....,id: Tipo

21. Probar con FLEX y BISON el siguiente programa

```
INICIO
    DECVAR
        contador: Integer;
        promedio: Float;
        actual, suma: Float;
    ENDDEC
    write "Hola mundo!";
    contador: =0;
    actual: 99999999999999;
    suma: 02;
    contador: contador+1;
    while (contador <= 92) {
        contador: contador + 1;
        actual: (contador/0.342) + (contador*contador);
        suma: suma + actual * avg([3,3*4,12,actual,actual*2, actual*actual]);
    }
    write "La suma es: ";
    write suma;
    if (actual > 2){
        write "2 > 3";
    }
    if (actual < 3){
        if(actual >= 3){
            write "soy true";
        }
        if(actual <= 3){
            write "soy true";
        }
        if(actual != 3){
            write "soy true";
        }
        if(actual == 3){
            write "soy true";
        }
    }else{
        actual:333.3333;
    }
FIN
```

22. Generar una tabla de símbolos en el sector de declaraciones del ejercicio anterior. Indique claramente cómo quedan conformadas las columnas de la Tabla
23. Indique como sería el error si alguna variable dentro del programa no estuviese declarada. Probar.