Práctico 4 - Gramáticas libres de contexto

Ejercicio 1
Para cada uno de los siguientes lenguajes definidos sobre el alfabeto A = {a, b, c, d, e, h, x, y, z, 0, 1, 2, 3, 4} diseñar y definir formalmente una GLC que lo reconozca:

a) $L_1 = \{a^{2k}\ b^{2n}\ c^k\ d^j / k, n, j >= 0\}$

k	n	j	Cadena resultante
0	0	0	λ
0	0	1	d
0	1	0	bb
1	0	0	aac
0	1	1	bbd
1	1	1	aabbcd

Base = $aa^k\ bb^n\ c^k\ d^j$

S => λ | C | A | AC

A => aa A c | aac | B
B => bbB | bb
C => dC | d

b) $L_2 = \{x^r\ y^s\ z^t / t = r+s, r, s >= 1\}$

Posibles: xyzz, xxyyzzzz
Base = $x^r\ y^s\ z^r\ z^s$

S => xAz

A => xAz | B
B => yBz | yz

c) $L_3 = \{x^r\ y^s\ z^t / s = r+t, r,s >= 1\}$

Despeje del exponente s = r + t <==> 1 = 1 + t <==> 1 - 1 = t <==> t = 0 ====> t >= 0.

Posibles:

r	s	t	Cadena resultante
0	0	0	λ
1	1	0	xy
0	1	1	yz
1	2	1	xyyz

Base = $x^r\ y^r\ y^t\ z^t$

S => λ | A | B | AB

A => xAy | xy
B => yBz | yz

d) L_4 = { x / x = a Y e, donde Y = b^{3n} c d^{3n}, n >= 1}.

Posibles: abbbcddde, abbbbbbcddddddde.

Base: a $(bbb)^n$ c $(ddd)^n$ e

S => abbbAddde

A => bbb A ddd | c

e) L_5 = {1^n 0^k / n >= 0 y k = 3n}.

Posibles = 1000, 11000000, 111000000000.

Base = 1^n $(000)^n$

S => 1 S 000 | 1000

f) L_6 = { $(ab)^j$ c^{2i} b^{i+1} c^k d^n / i, k, n >= 0 y n < j}.

Posibles:

j	i	k	n	Cadena resultante
1	0	0	0	abb
1	0	0	1	abbd
1	0	1	0	abbc
1	0	1	1	abbcd
1	1	0	0	abccbb
1	1	0	1	abccbbd
1	1	1	0	abccbbc
1	1	1	1	abccbbcd

Base = $(ab)^j$ $(cc)^i$ b^i **b** c^k d^n

S => Ab | AbD | AbC | AbCD

A => abA | ab | B
B => ccBb | ccb
C => cC | c
D => dD | d

g) L_7 = { a^n b^{n+2} a^m e^k b^{m+1} / n, m >= 1 y k >= 0}.

Posibles:

n	m	k	Cadena resultante
1	1	0	ab**bb**ab**b**
1	1	1	ab**bb**aeb**b**

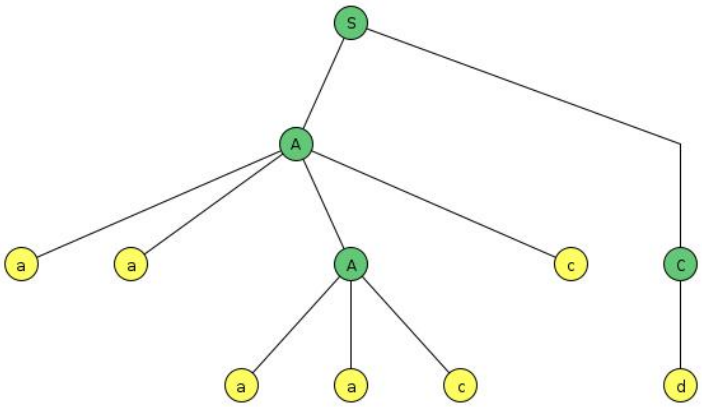
Base = a^n b^n **bb** a^m e^k b^m **b**

S => AbbBb

A => aAb | ab
B => aBb | aCb | ab
C => eC | e

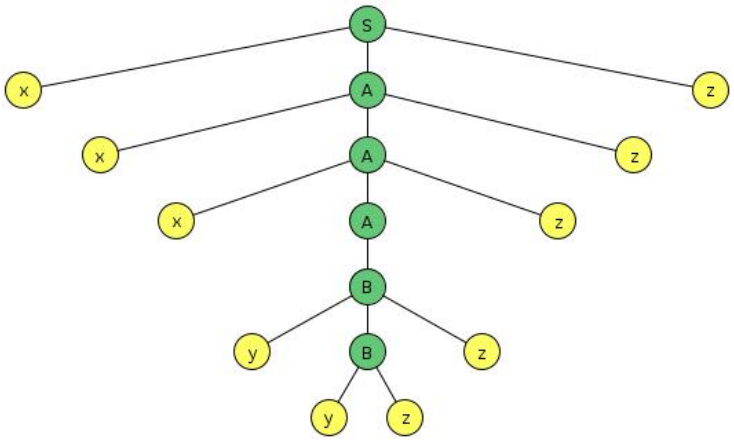
Ejercicio 2
Para cada una de las siguientes hileras armar el árbol de derivación en cada una de las gramáticas definidas en el ejercicio anterior.

a) h_1 = aaaacccd.



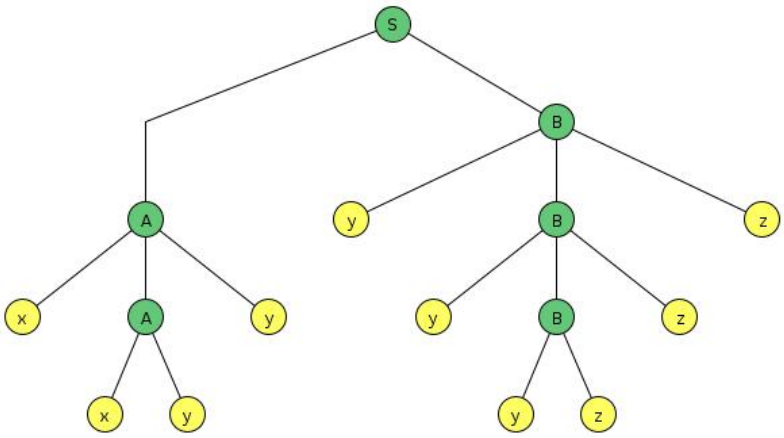
h_1 = aaaacccd => No pertenece a la G definida y por lo tanto no se puede hacer su árbol de derivación.
Resultado más cercano = aaaaccd.
Sobra una “c” en h_1.

b) h_2 = xxxyyzzzzz.



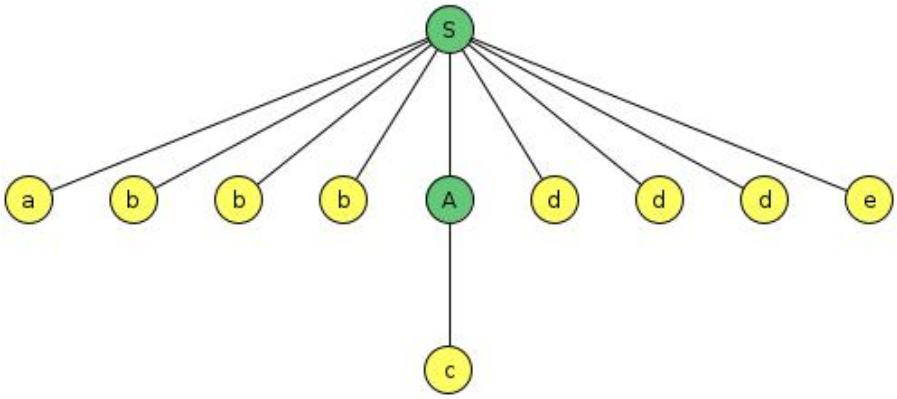
La cadena pertenece a la G definida.

c) h_3 = xxyyyyyzzz.



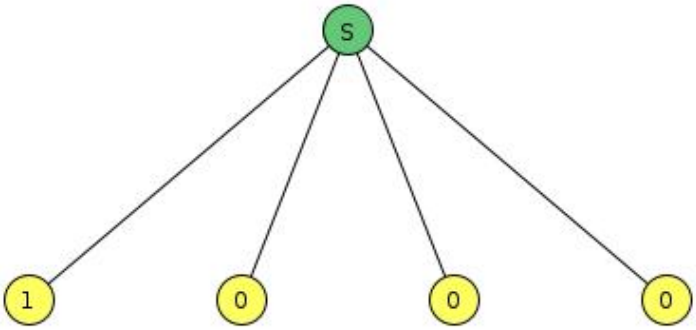
La cadena pertenece a la G definida.

d) h_4 = abbbcddde.



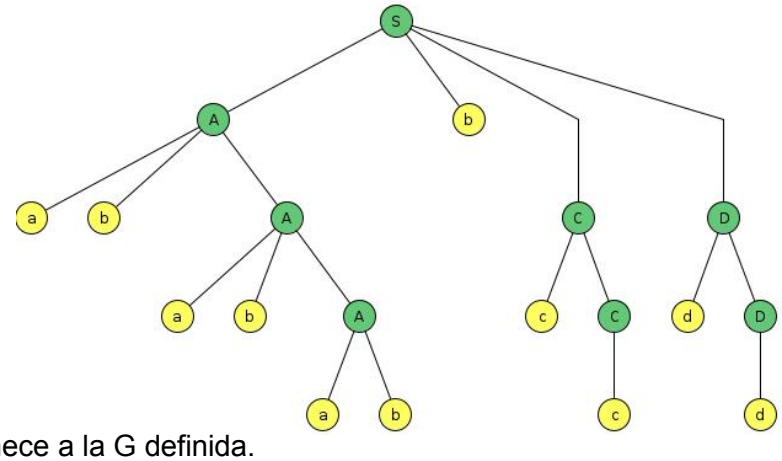
La cadena pertenece a la G definida.

e) h_5 = 11000.



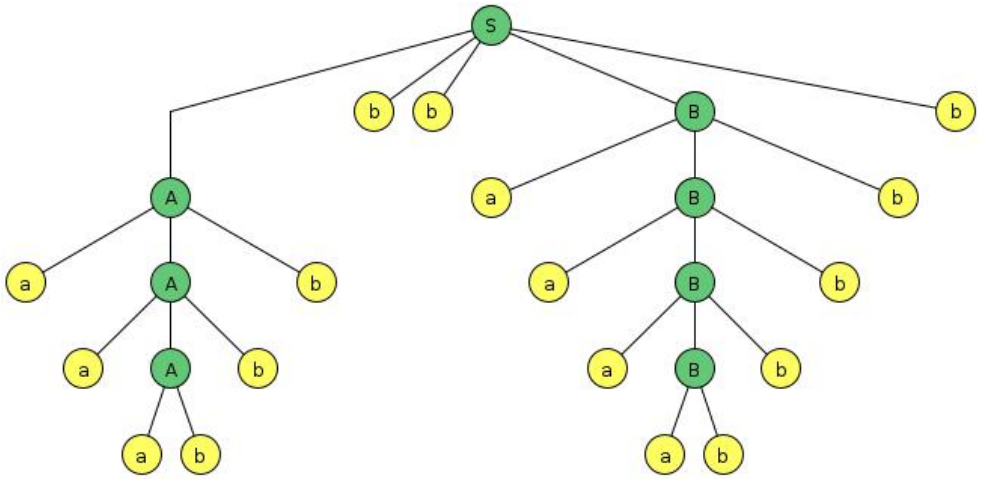
h_5 = 11000 => No pertenece a la G definida y por lo tanto no se puede hacer su árbol de derivación.
Resultado más cercano = 1000 .
Sobra un “1” en h_5.

f) h_6 = abababbccdd.



La cadena pertenece a la G definida.

g) h_7 = aaabbbbbaaaabbbbb.



La cadena pertenece a la G definida.

Ejercicio 3:
Dada la siguiente gramática ambigua:

A => 0BB
B => 1A | 0A | 0

a) Mostrar los diferentes árboles de derivación para una hilera adecuada.

b) Clasificar el lenguaje que describe esta gramática.

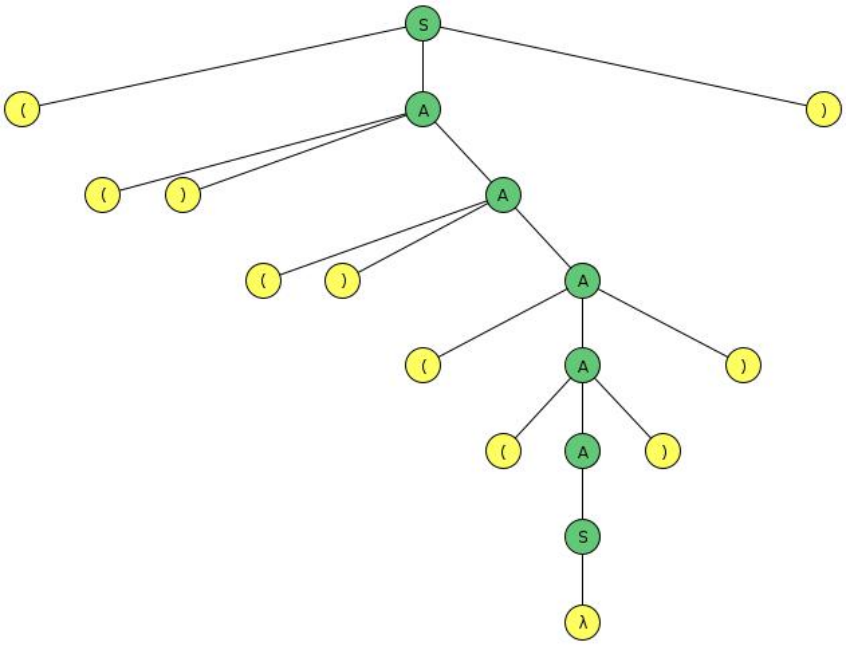
El lenguaje descripto por G son los números binarios pares.

Ejercicio 4:

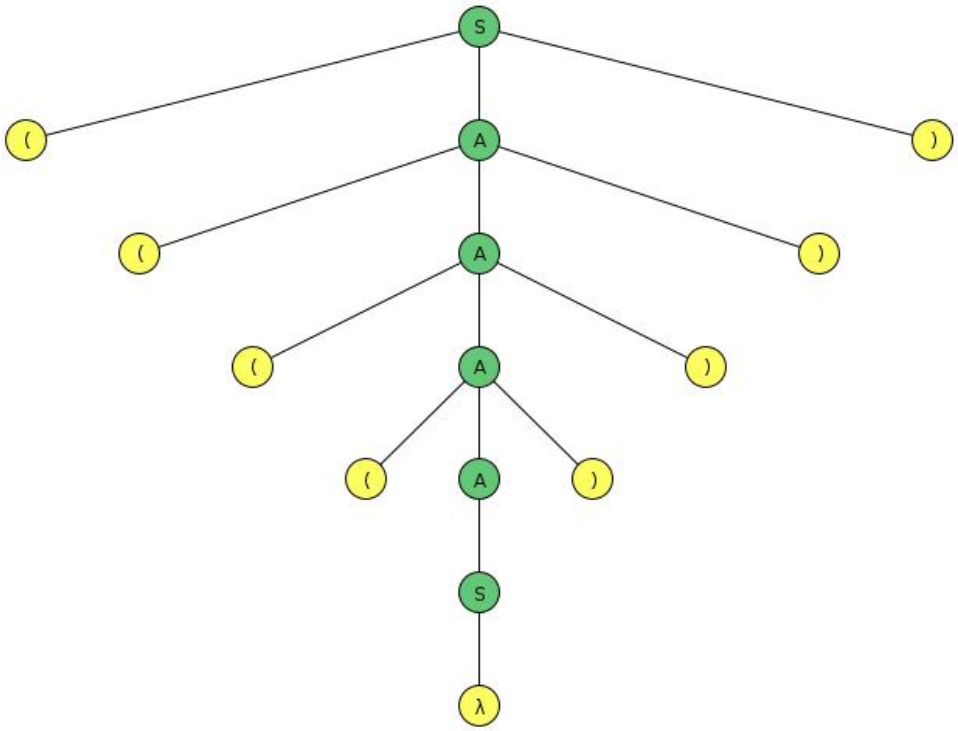
1. Definir una GLC que genere las hileras de la forma: x= (); y = λ ; z=(() ()); y cualquier otra donde los paréntesis se encuentren balanceados. Observar que el primer paréntesis que abre se aparea con el último que cierra; por ejemplo x = ( ) ( ) ( ) es una hilera no válida.

   S => λ | (A)
   A => (A) | ()A | S

2. Generar el árbol de derivación más a la izquierda de las siguientes hileras
   a) x = (( ) ( ) (( )))



b) x = (((( )))).



Ejercicio 5
Definir una GLC que genere cabeceras de métodos en Java:
`tipo_de_retorno nombre_del_metodo ( lista_parametros )`

Donde:
- Lista de parámetros será una enumeración de cero o más parámetros, separados por comas.
- Los parámetros tendrán la forma:
  - `tipo nombre`.
- `nombre` y `nombre_del_metodo`: será una sucesión de letras y dígitos que comienza con una letra.
- Tipo_de_retorno y tipo: Considerarlos como terminales.

```
<h> ::= tipo_de_retorno <nom_metodo>(<lista_param>) | tipo_de_retorno<nom_metodo>( )

<nom_metodo> ::= <nombre>

<lista_param> ::= <param> | <lista_param>,<param>

<param> ::= tipo <nombre>

<nombre> ::= <letra> | <digito> | <nombre><digito> | <nombre><letra>

<letra> ::= A | B | ... | Z | a | b | ... | Z

<digito> ::= 0 | 1 | ... | 9.
```

Ejercicio 6:
Definir una GLC que genere cabeceras de procedimientos de la forma:
procedure nombre **(** lista de parámetros **)**

Donde:
- Lista de parámetros será una enumeración de parámetros, separados por comas.
- Los parámetros tendrán la forma:
  - nombre tipo ent/sal [**default** cte].
- nombre: será una sucesión de letras y dígitos que comienza con una letra.
- tipo: puede ser char, number, string o bool.

- cte: es una constante numérica o un string.
- ent/sal: puede ser IN, OUT, INOUT.
- los corchetes no son parte del lenguaje, indican que esa parte de la cabecera es opcional.
- Lo que figura en negrita es terminal: procedure, (,), default.

Ejemplos de hileras válidas:

- procedure mostrarlista (lista char IN,
  cant number IN default 0,
  salida char OUT).

- procedure vector(vec1 string IN default‘ ’,
  vec2 string INOUT).

```
<h> ::= procedure <nombre> ( <list_param> )
<nombre> ::= <letra> | <nombre> <letra> | <nombre> <digito>

<letra> ::= A | B | ... | Z | a | b | ... | z
<digito> ::= 0 | 1 | 2 | ... | 9
<numero> ::= <digito> | <numero> <digito>

<list_param> ::= <param> | <list_param> , <param>

<param> ::= <nombre> <tipo> <ent_sal> | <nombre> <tipo> <ent_sal> default <cte>

<tipo> ::= char | number | string | bool

<ent_sal> ::= IN | OUT | INOUT

<cte> ::= <nombre> | <numero>
```

Ejercicio 7

Dado el siguiente BNF para expresiones lógicas:

```
<expresion_logica> :: = <expresion_logica> or <expresion_logica>
                      | <expresion_logica> and <expresion_logica>
                      | (<expresion_logica>)
                      | not <expresion_logica>
                      | true | false | <variable_logica>.
<variable_logica> : : = A | B | C | ... | Z.
```

a) Determinar, usando árboles de derivación, si las siguientes son expresiones lógicas están bien definidas de acuerdo al BNF dado:
   i. A or ((B and not (B or A)))and true.
   ii. ((A and B) or C) and not A or B.
   iii. A and (C or B) or not (true and false).

b) Determinar si la gramática es o no ambigua. Justifique. En caso de ser ambigua, definir si es posible una gramática no ambigua que genere el mismo lenguaje.

Ejercicio 8

En Java las declaraciones pueden ser con o sin inicialización, de la forma:

```
tipo nombre = cte, nombre = cte, ............, nombre = cte;
tipo nombre, nombre, nombre,......., nombre;
```

Donde tipo es algun nombre de tipos válido en el lenguaje y nombre es un identificador java.
Constante puede ser una constante entera o de cualquier tipo.
Generarla GLC correspondiente.

```
<declaracion> ::= <tipo> <con_ini>; | <tipo> <sin_ini>;
<con_ini> ::= <nombre> = <cte> | <con_ini> , <nombre> = <cte>
<sin_ini> ::= <nombre> | <sin_ini> , <nombre>
<cte> ::= <tipo>.
```

```
<tipo> ::= integer | string | float | boolean.
<nombre> ::= <letra> | <nombre><letra> | <nombre><digito>.
<letra> := a | b | ... | z | A | B | ... | Z.
<digito> ::= 0 | 1 | ... | 9.
```

Ejercicio 9
Escribir una BNF para describir los números naturales. ¿Es necesario escribir reglas para este lenguaje?

```
<n> ::= <digito> | <n> <digito>

<digito> ::= 0 | 1 | ... | 9
```

Ejercicio 10
a) Desarrollar una gramática en formato BNF para reconocer expresiones aritméticas simples (sumas, restas, multiplicaciones y divisiones) que operan con constantes enteras e identificadores.

En BNF:
```
<expr> ::= <termino> | <termino><operacion><expr>
<operacion> ::= + | - | * | /
<termino> ::= id | cte.
```

En GLC:
E => E + E | E - E | E * E | E / E
E => id | cte

b) Agregar a la gramática anterior la posibilidad que la multiplicación y división tengan prioridad sobre la suma y la resta.

En BNF:
<expr> ::= <expr> + <termino> | <expr> - <termino> | <termino>
<termino> ::= <termino> * <factor> | <termino> / <factor> | <factor>
<factor> ::= id | cte

En GLC:
E => E + T | E - T | T
T => T * F | T / F | F
F => id | cte

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

En BNF:
<expr> ::= <expr> + <termino> | <expr> - <termino> | <termino>
<termino> ::= <termino> * <factor> | <termino> / <factor> | <factor>
<factor> ::= id | cte | (<expr>)

En GLC:
E => E + T | E - T | T
T => T * F | T / F | F
F => id | cte | (E)

Ejercicio 11
a) Desarrollar una BNF 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 que soporte sumas, restas, multiplicaciones, divisiones y expresiones entre paréntesis (basarse en la gramática del ejercicio 10).

<expr> es igual a la GLC del punto anterior.

```
<sentencia>::= <id> := <expr>;
<id> ::= <letra> | <id><letra> | <id><numero>
<letra> ::= a | b | ... | z | A | B | ... | Z
<numero> ::= <digito> | <numero><digito>
<digito> ::= 0 | 1 | ... 9
```

b) Probar el siguiente ejemplo indicando la lista de reglas. Dibujar el árbol de parsing.

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

Ejercicio 12

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:=expresion;
```

Escribiren BNF/GLC la sintaxis de este tipo de asignaciones con las siguientes condiciones: la sentencia debe terminar con ";" y el valor de la derecha debe ser expresión como la desarrollada en el ejercicio 10.

<expr> es igual a la GLC del punto 10.

```
<sentencia> ::= <id>:=<valor>
<valor> ::= <expr>:=<valor> | <expr>;

<id> := <letra> | <id><letra> | <id><numero>
<letra> ::= a | b | ... | z | A | B | ... | Z
<numero> ::= <digito> | <numero><digito>
<digito> ::= 0 | 1 | ... 9
```

Ejercicio 13

Definir una BNF/GLC que soporte programas que tengan asignaciones simples como las del ejercicio 11 y asignaciones múltiples como las del ejercicio 12. 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.

<expr> es igual a la GLC del punto 10.

<asignacion> es igual a la GLC del punto 12, que acepta asignaciones única y múltiples.

<programa> ::= INICIO <lista_sentencias> FIN
<lista_sentencias> ::= <sentencia><lista_sentencia> | <sentencia>

Ejercicio 14

a) Definir una BNF/GLC para un lenguaje que soporta:
   i. Asignaciones simples de expresiones.
   ii. Sentencia IF.
   iii. Sentencia WHILE.
   iv. Sentencia WRITE (constante string).
   v. Comienza y Termine con las palabras INICIO y FIN.
   vi. Separador de sentencias ;.
   vii. Separador de bloques { }.
   viii. Operador de asignación :
   ix. Condiciones : dos expresiones con un solo operador.

   Agregar al lenguaje del ejercicio anterior la posibilidad de definir sentencias en las que se pueda incorporar la función standard AVERAGE.

Formato de AVERAGE : `AVG( Lista de expresiones )` donde Lista de expresiones es una lista de expresiones aritméticas separadas por comas dentro de dos corchetes. AVERAGE debe operar dentro de una expresión.

```
<program> ::= INICIO <lista_sentencia> FIN

<lista_sentencia> ::= <sentencia> <lista_sentencia> | <sentencia>

<sentencia> ::= <asignacion>;| <if> | <while> | <write>;

<asignacion> ::= <lista_id> = <exp_sr>

<lista_id> ::= <id> = <lista_id> | <id>

<if> ::= if (<condicion>) <bloque>

<while> ::= while (<condicion>) <bloque>

<write> ::= write <cte_str>| write <id>

<bloque> ::= { <lista_sentencia> }

<condicion> ::= <exp_lobi> Corregir: 2 expresiones con 1 solo operador lógico.
<exp_lobi> ::= <exp_lobi> or <comp_lobi> | <exp_lobi> and <comp_lobi>| <comp_lobi> ??

<comp_lobi> ::= <comp_lobi> <op_lobi><valor>| <valor>

<op_lobi> ::= > | >= | < | <= | != | ==

<valor> ::= (<valor>) | -<valor> | <exp_lobi> | <exp_sr> | <id> | <cte>| <funcion> Acepta (id) > (4) Revisar. El not no se pide, solo condiciones lógicas binarias.

<exp_sr> ::= <exp_sr> + <exp_md> | <exp_sr> -<exp_md> | <exp_md>

<exp_md> ::= <exp_md> * <valor> | <exp_md> / <valor> | <valor>

<id> ::=<letra><cadena_caracter> | <letra>
<cte> ::= <cte_num> | <cte_str> | <cte_log>
<cte_num> ::= <numero><cte_num> | <numero>
<cte_str> ::= "" | "<cadena_caracter>"
<cte_log> ::= true | false
<cadena_caracter> ::= <caracter><cadena_caracter> | <caracter>
<caracter> ::= <letra> | <numero>| -| _<letra> ::= a | b | c | ... | z | A | B | C | ...
| Z
<numero> ::= 0 | 1 | 2 | ... | 9

<funcion> ::= <avg>
<avg> ::= avg([<lista_valores>])
<lista_valores> ::= <valor>, <lista_valores> | <valor>
```

No es valor, ojo conel start symbol de la gramática de expresiones. Va a permitir avg([(id)]) Y como valor deriva en <exp_lobi> se mezclan expresiones lógicas.

<programa>::=INICIO <contenido>FIN|INICIO FIN

<contenido>::=<sentencia><cont_programa contenido? >|<sentencia_if><contenido>|

<sentencia_while><contenido>|<sentencia_write><contenido>|<sentencia>|<sentencia_if>
|<sentencia_while>|<sentencia_write>

Corregida: <contenido>

<sentencia><sentencia_while>|<sentencia_write|<sentencia_if>|<sentencia_while>|
<asignación><separador_de_sentencias>

<asignación>::=<id> =<valor> En lugar de valor, armar la gramática de expresiones.

<valor>::=<id>|<numero>|<id><operador>< valor >|<numero><operador><valor>

<sentencia_if>::=if(<condicion>)<bloque>

<bloque> ::= <separador_bloquesA><lista_sentencia> <separador_bloquesC>

<lista_sentencia>::=<sentencia><lista_sentencia>|<sentencia>

Ya está definida sentencia, como todas las sentencias del programa. Puede usar la regla <contenido>, en lugar de lista_sentencia.

<sentencia>::=<asignacion><separador_de_sentencias>|<sentencia_if>|<sentencia_while>
|<sentencia_write>

<sentencia_while>::=while (<condición>) <bloque>

<sentencia_write>::=write (<cte_str>)|write (<id>)

<condición>::=<expresion><op_logico><expresion>

<expresion>::=<id>|<numero>

<op_logico>::= <|<=|==|!=|>|>=

<separador_bloquesA>::={

<separador_bloquesC>::=} Sino, puede armar bloque }{

<separador_de_sentencias>::=;

<operador>::=+|-|*|/

<id>::=<letra><cadena_caracter>|<letra>

<cadena_caracter> ::= <caracter> <cadena_caracter> | < caracter>

<caracter>::= <letra>| <numero> | -|_

<letra>::= a | b | c | ... | z | A | B | C | ... | Z

<numero>::= 0 | 1 | 2 | ... | 9

<cte_str> ::= “” | “<cadena_caracter>”

b) Probar con FLEX y JavaCupel 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
```

c) 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;` ó `id,id,.....,id: Tipo`

d) Probar con JFLEX y JavaCup 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
```

e) Generar una tabla de símbolos en el sector de declaraciones del ejercicio anterior. Indique claramente como quedan conformadas las columnas de la Tabla.

f) Indique como sería el error si alguna variable dentro del programa no estuviese declarada. Probar.