Ejercicio 1

Primeros				
Regla	**Paso 1**	**Paso 2**	**Paso 3**	**Paso 4**
S' → S			x, y	x, y, z
S → AwS		x, y	x, y, z	
S → xS	x			
A → BC		y, z		
A → y	y			
B → zB	z			
B → z	z			
C → ϵ	ϵ			
C → y	ϵ, y			
C → D		ϵ, y, z		
D→ ϵ	ϵ			
D→ zD	ϵ, z			

Siguientes		
Regla	**Paso 1**	**Paso 2**
S' → S	S': $ S: $	
S → AwS	A: w	
S → xS		
A → BC	B: y, z, w C: w	
A → y		
B → zB		
B → z		
C → ϵ		
C → y		
C → D	D: w	
D→ ϵ		
D→ zD		

Gramática	Primeros	Siguientes	¿Es LL(1)?
S' → S S → A**w**S S → **x**S A → BC A → **y** B → **z**B B → **z** C → **ϵ** C → **y** C → D D→ **ϵ** D→ **z**D	Primeros(S')={x, y, z} Primeros(S)={x, y, z} Primeros(A)={y, z} Primeros(B)={z} Primeros(C)={ϵ, y, z} Primeros(D)={ϵ, z}	Siguientes(S')={$} Siguientes(S)={$} Siguientes(A)={w} Siguientes(B)={y, z, w} Siguientes(C)={w} Siguientes(D)={w}	No es LL(1), porque en la regla B → zB | z, Primeros(zB) ∩ Primeros(z) ≠ ø

	w	**y**	**z**	**x**	**$**
S'		S' → S	S' → S	S' → S	synch
S		S → AwS	S → AwS	S → xS	synch
A	synch	A → y	A → BC		
B	synch	synch	**B → zB** **B → z** **synch**		
C	C → ϵ	C → y	C → D		C → ϵ
D	D → ϵ		D→ zD		D → ϵ

Ejercicio 2

Ejercicio 0							
Se cambió en la regla T a ID por type para que acepte int y float y funcione la cadena							
Gramática	**Primeros**		**Siguientes**				

S → LB A RB A → T ID; A A → ϵ T → type T' T' → * T' T' → ϵ type → int type → float	Primeros(S)={LB} Primeros(A)={ϵ, int, float} Primeros(T)={int, float} Primeros(T')={ϵ, *} Primeros(type)={int, float}		Siguientes(S)={$} Siguientes(A)={RB} Siguientes(T)={ID} Siguientes(T')={ID} Siguientes(type)={*, ID}					
	LB	**RB**	**ID**	**int**	**float**	**;**	*****	**$**
S	S → LB A RB							synch
A		A → ϵ		A → T ID; A	A → T ID; A			
T			synch	T → type T'	T → type T'			
T'			T' → ϵ				T' → * T'	
type			synch	type → int	type → float		synch	

Stack	Entrada	Acción
$ S	{ int x; float *y; } $	S → LB A RB
$ RB A LB	{ int x; float *y; } $	match (LB)
$ RB A	int x; float *y; } $	A → T ID; A
$ RB A ; ID T	int x; float *y; } $	T → type T'
$ RB A ; ID T' type	int x; float *y; } $	match (type)
$ RB A ; ID T'	x; float *y; } $	T' → ϵ
$ RB A ; ID	x; float *y; } $	match (ID)
$ RB A ;	; float *y; } $	match (;)
$ RB A	float *y; } $	A → T ID; A
$ RB A ; ID T	float *y; } $	T → type T'
$ RB A ; ID T' type	float *y; } $	match (type)
$ RB A ; ID T'	*y; } $	T' → * T'
$ RB A ; ID T' *	*y; } $	match (*)
$ RB A ; ID T'	y; } $	T' → ϵ

$ RB A ; ID	y; } $	match (ID)
$ RB A ;	; } $	match (;)
$ RB A	} $	A → ϵ
$ RB	} $	match (RB)
$	$	**Aceptar**

Ejercicio 3

Un lexer consiste en la siguiente gramática para obtener un token en particular

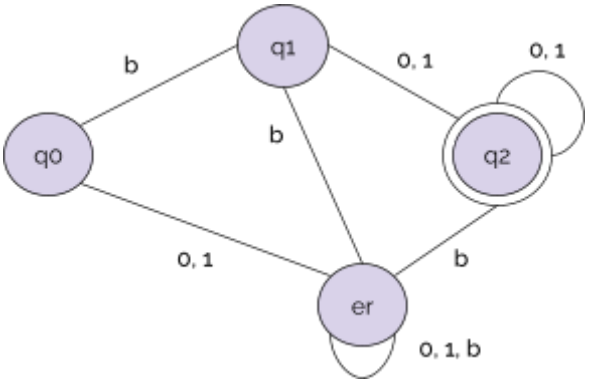
S → b X
X → 0 X
X → 1 X
X → 0 | 1

a) Describa el token aceptado por esta gramática y su expresión regular

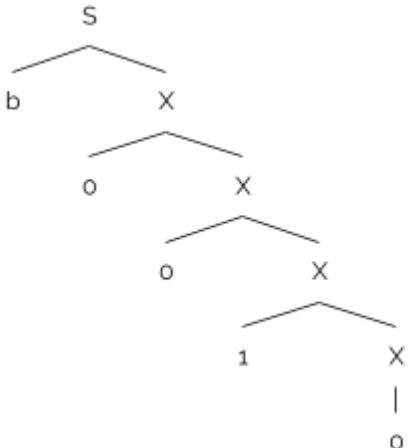
Acepta cadenas que empiezan con b y tiene una secuencia de 0s y 1s a la derecha. Son números en binario.

Expresión regular: b (0+1)+

b) Construya el autómata que representa esta gramática, incluyendo por lo menos un estado de error



c) Construya el árbol gramatical de análisis del lexema b0010



Ejercicio 4

A → ([B])
B → { + B }
B → num

a) Reescriba la regla de A en formato BNF

A → (B) | ()

Puesto que [] indica que es opcional, tenemos dos opciones para la gramática, incluir la B o no incluirla.

b) Reescriba la regla de B en formato BNF, tal que + es asociativa por la izquierda.

Se cambió la regla para poder ser transformada correctamente. Esta nueva regla asegura la derivación de cadenas num(+num)*.

B →num { + num }

Se realiza la recursividad por la izquierda para que haya asociatividad para ese lado.

B → B + num | num

Ejercicio 5

Un lenguaje experimental para mudos consiste en tornar en auditivas palabras de la forma:

p id dig

donde, **id** son palabras consistentes de letras del alfabeto español. La palabra debe estar precedida por una **'p'**, y sucedidas por un dígito(dig). Palabras en este formato, harán que el computador emita un sonido particular por cada palabra leída durante **dig** segundos.

Se asume que id es solo una palabra, es decir, entre p y dig solo habrá un id.

a) Implemente el lexer correspondiente en un pseudocodigo:

La función lexer solo detecta errores léxicos leyendo la cadena y convierte los encontrados a TOKENS. Se asume que TOKEN <token, atributo> transforma la cadena al token correspondiente.

funcion **lexer**:
    INPUT: string w

```
entero i=0
mientras i < tamaño de w
        si w[i] == ' ' entonces
                i += 1
        si w[i] == 'p' entonces
                TOKEN < p, w[i])>
                i += 1
        si w[i] está en el alfabeto
                entero inicio = i
                i += 1
                mientras w[i] está en el alfabeto
                        i += 1
                TOKEN <id, w[inicio, i]>
        si w[i] es un entero entonces
                entero inicio = i
                i += 1
                mientras w[i] es entero
                        i += 1
                TOKEN <dig, w[inicio, i]>
        de lo contrario
                TOKEN <ERROR, w[i]>
                i +=1
```
fin **lexer**;

b) Implemente el parser en un pseudocódigo. Este debe permitir construir oraciones de un máximo de 3 palabras.

El parser lee los TOKENS puestos en la función lexer.

get_next_token() pasa al siguiente token

funcion **parser**:

```
entero palabras = 0
mientras token  = get_next_token()
        si token es tipo p
                token = get_next_token()
                si token es tipo id
                        token = get_next_token()
                        si token es tipo dig
                                token = get_next_token()
                        de lo contrario genere un error en la secuencia
                de lo contrario genere un error en la secuencia
        de lo contrario genere un error en la secuencia
```

fin **parser**;

c) Llame al parser desde el main() y diseñe la interacción entre lexer y parser en este lenguaje.

definir secuencia_tokens
función **main**:
    string w = input
    // el lexer escribe en la secuencia de tokens
    lexer(w)
    // el parser analiza la secuencia_tokens
    parser()
    escribir errores encontrados en la secuencia_tokens
fin **main**;

d) Describa cómo funciona el analizador con un ejemplo