

# Class Notes

## CLASS NOTES

Contenidos de la Pizarra + Diapositivas + Apuntes.

[Formato pdf \(Última Actualización\)](#)



*Apuntes bien hechos a partir del temario del segundo examen.*

## INDEX GITHUB

- [1.-Basic-Concepts](#)
- [2. NFA to DFA](#)
- [3. Epsilon Closures, Accessibility and Co-Accessibility](#).
- [4. Minimization](#)
- [5. Properties and Operations](#)
- [6. Pumping](#)
- [7. Grammar](#)
- [8. ER a Automata](#)
- [9. Automata a ER](#)
- [10. ER a Gramatica POR HACER](#)
- [11. Gramáticas de Contexto Libre - GCL](#)
- [12. Sintaxis](#)



*Ciertas marcaciones se verán mal o distintas fuera de Obsidian.*

*Recomendable descargar [Obsidian](#) y abrir este archivo haciendo un **clon** del repositorio.*

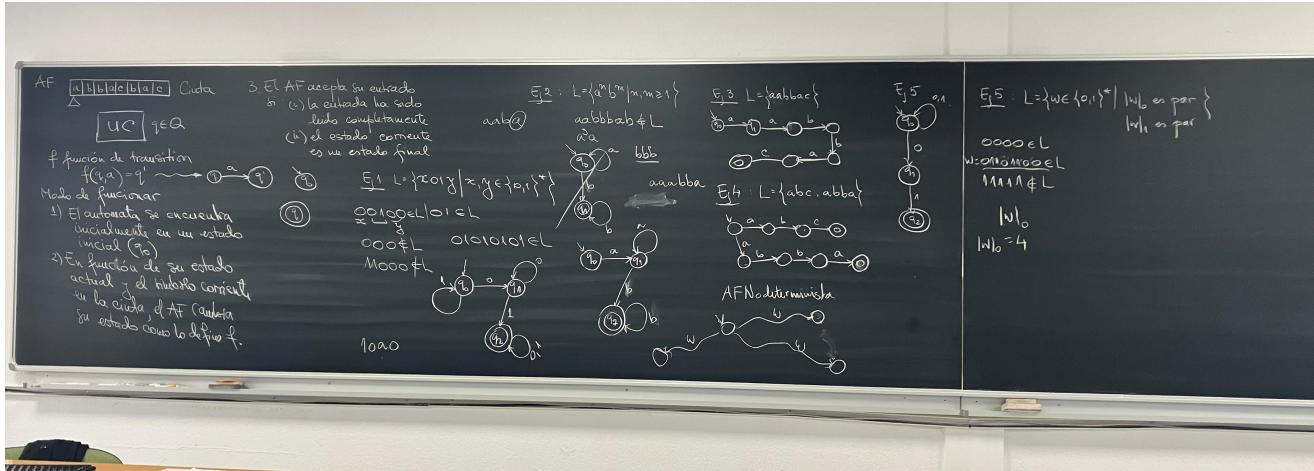
## INDEX OBSIDIAN

- [1. Basic Concepts](#)
- [2. NFA to DFA](#)
- [3. Epsilon Closures, Accessibility and Co-Accessibility](#).
- [4. Minimization](#)
- [5. Properties and Operations](#)
- [6. Pumping](#)

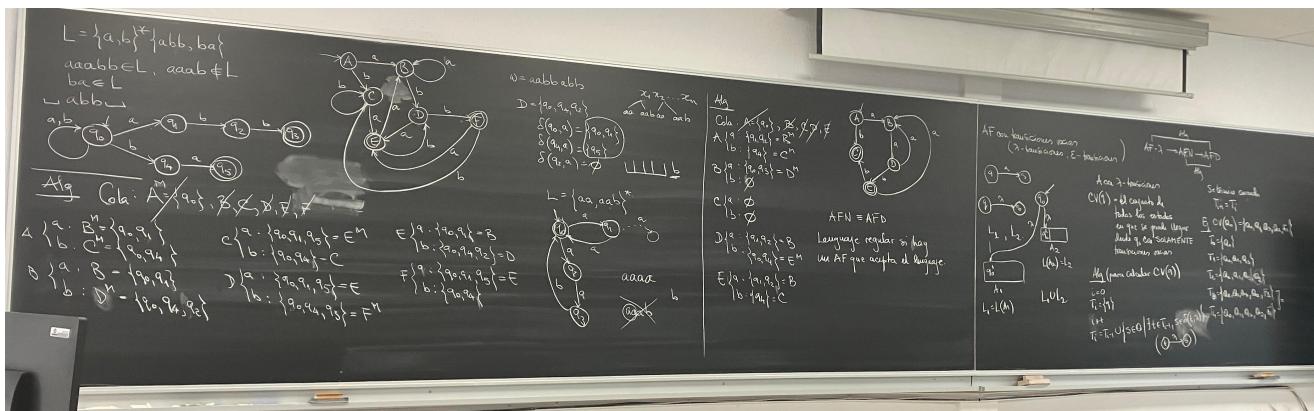
- [7. Grammar](#)
  - [8. ER a Automata](#)
  - [9. Automata a ER](#)
  - [10. ER a Gramatica POR HACER](#)
  - [11. Gramáticas de Contexto Libre - \\*GCL\\*](#)
  - [12. Sintaxis](#)

## **NOTES**

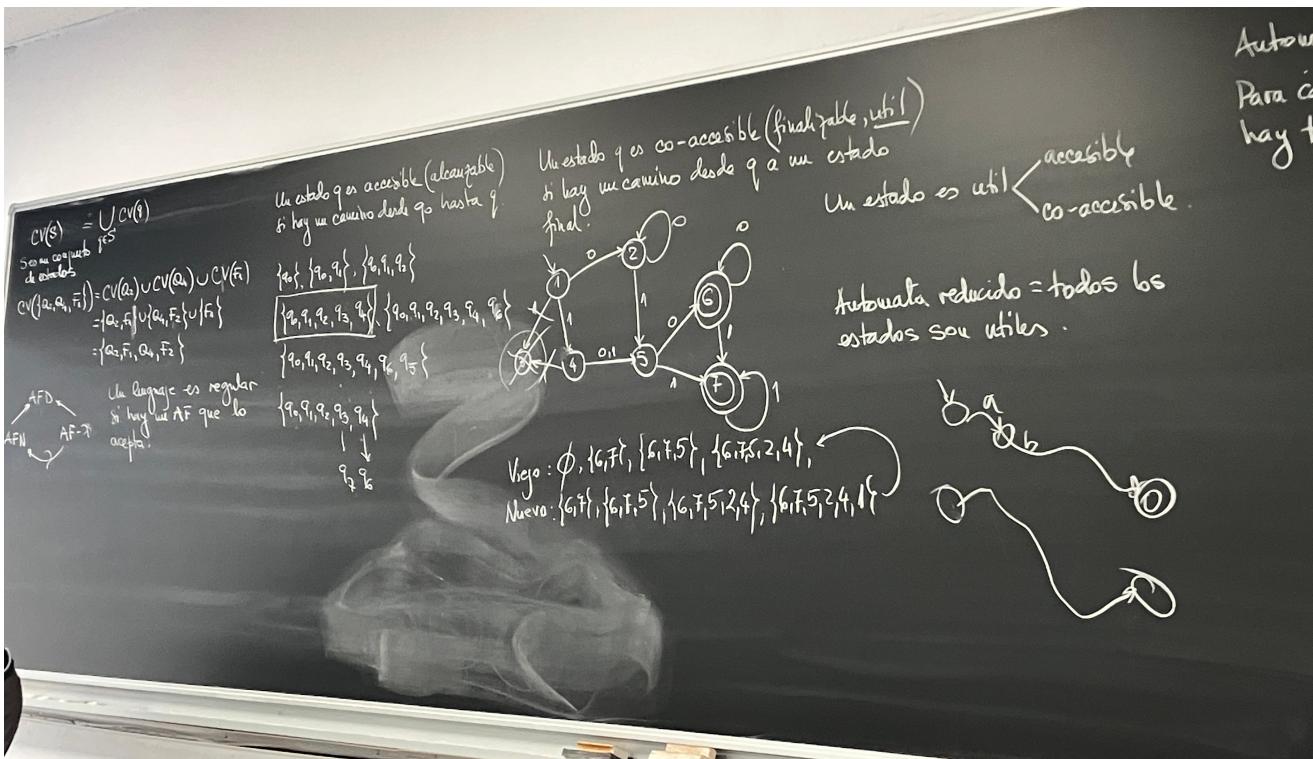
# 1. Basic Concepts



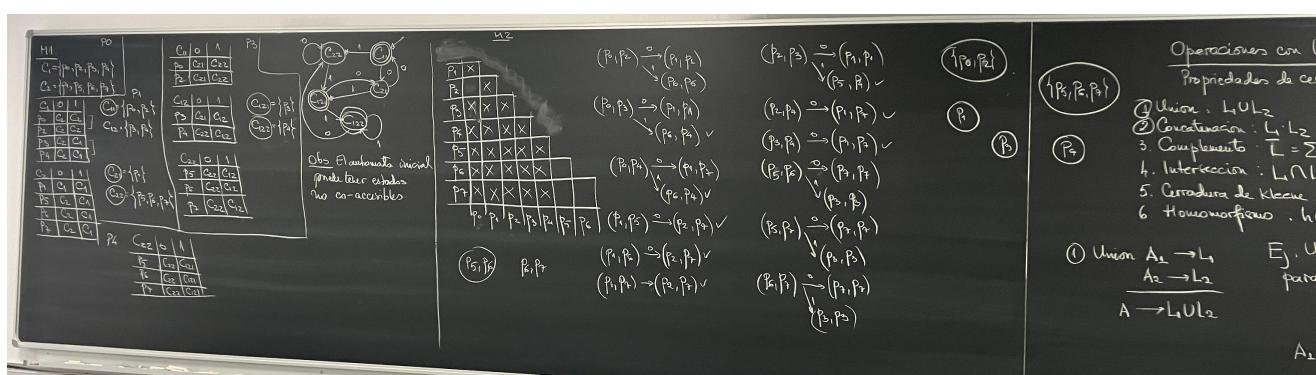
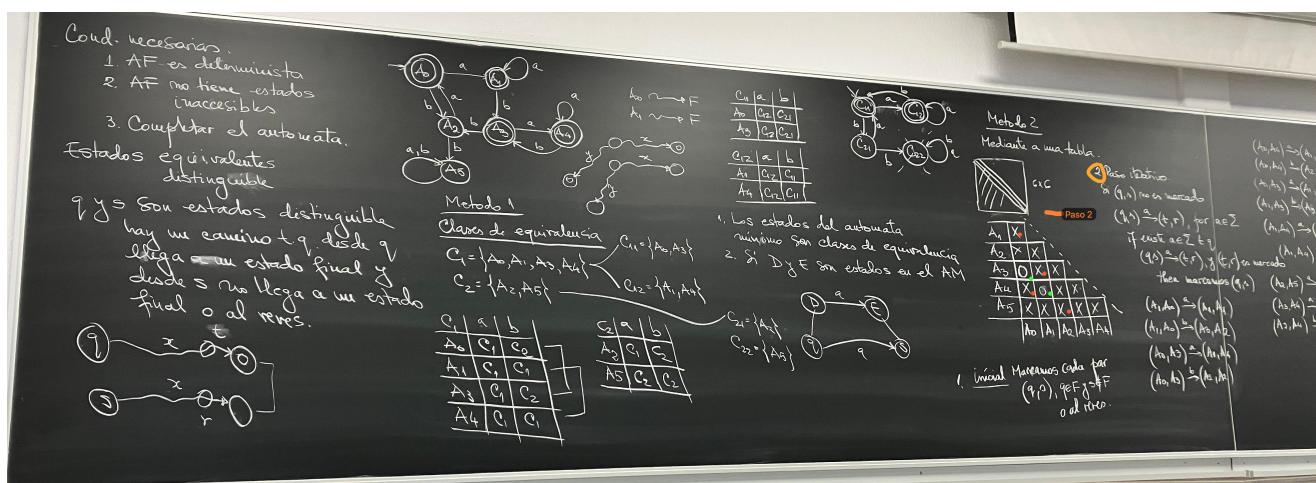
## 2. NFA to DFA



### 3. Epsilon Closures, Accessibility and Co-Accessibility

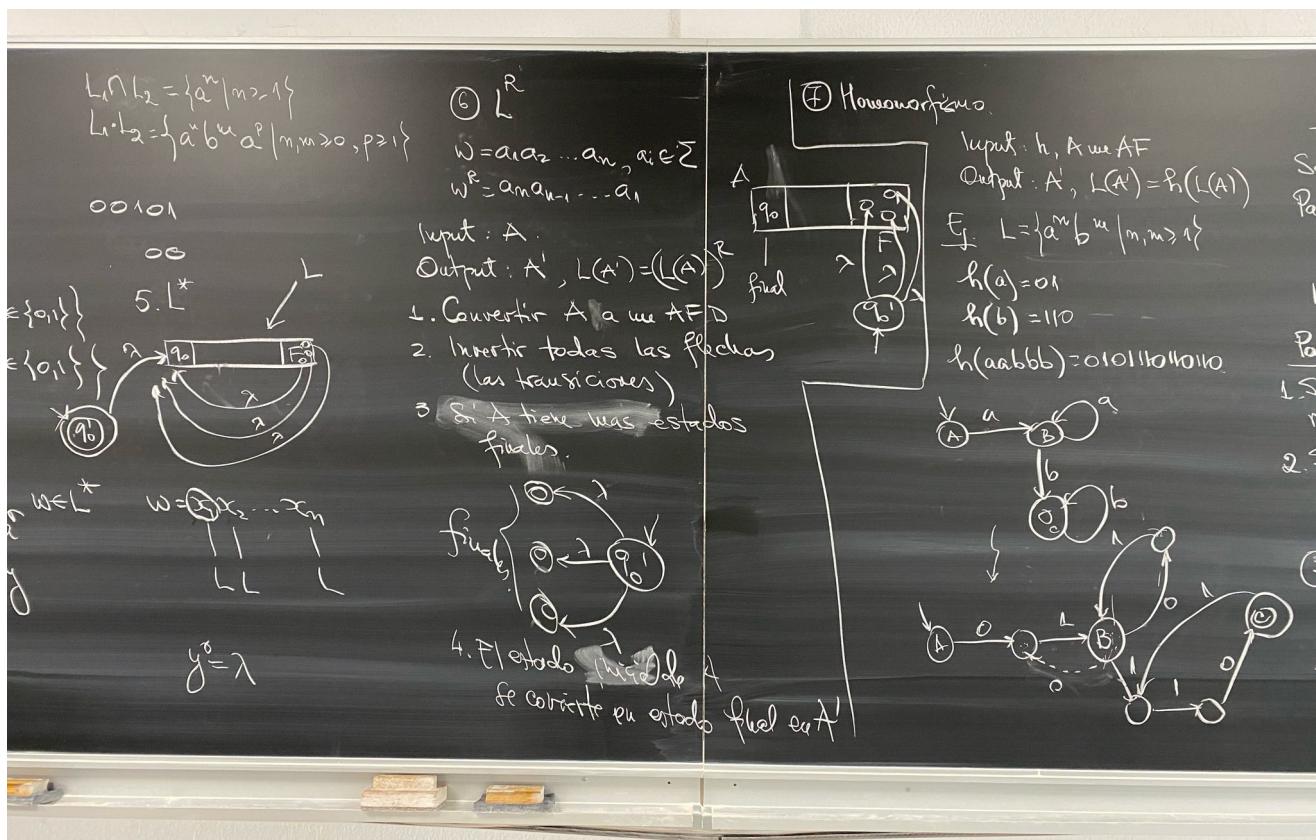
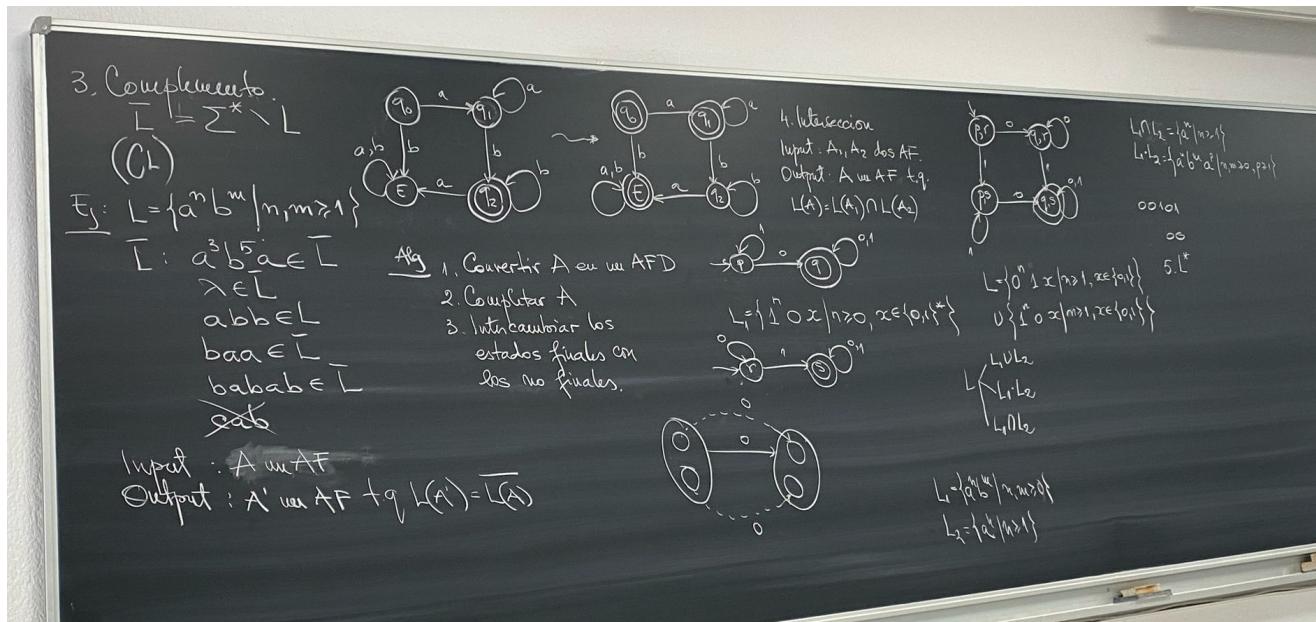
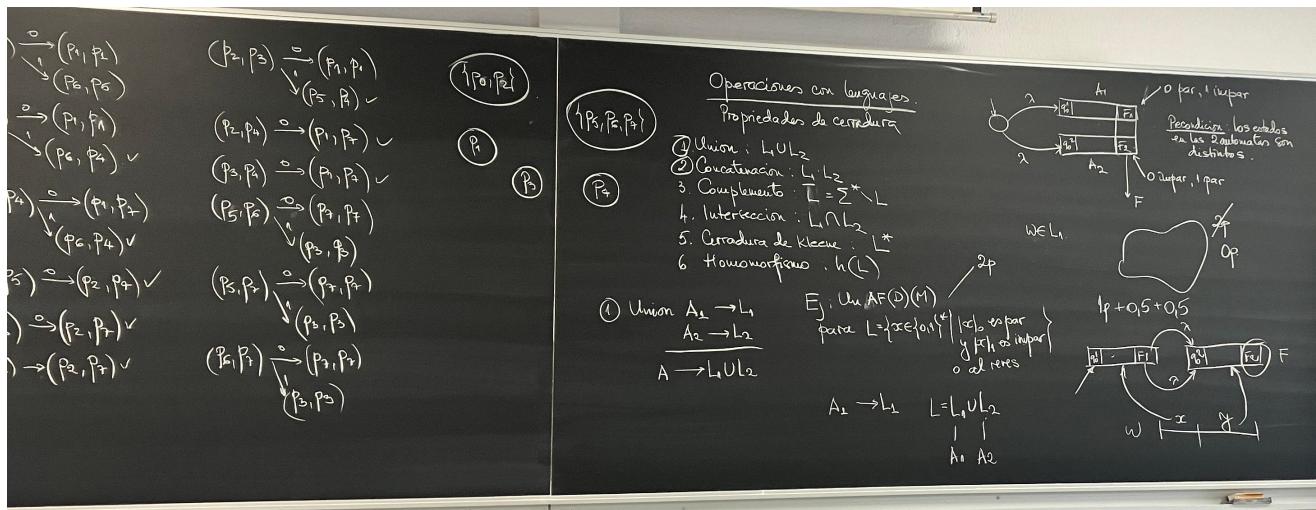


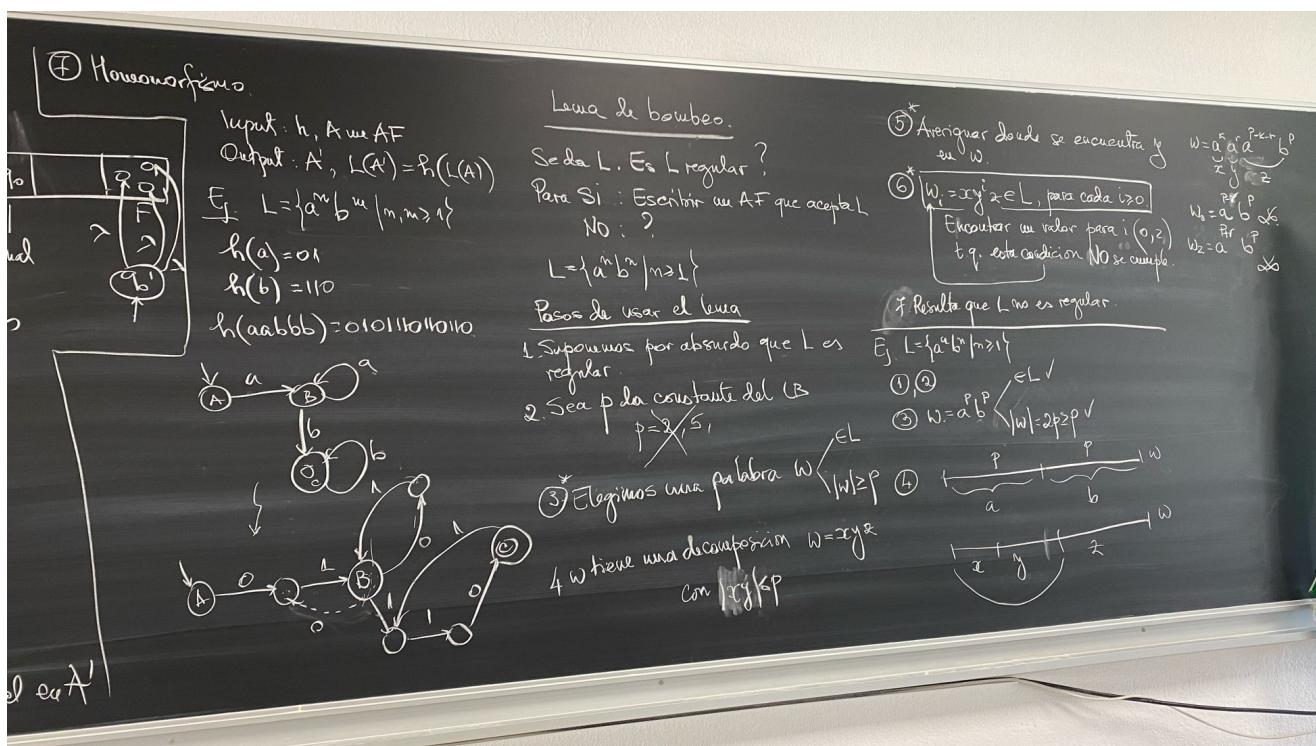
## 4. Minimization



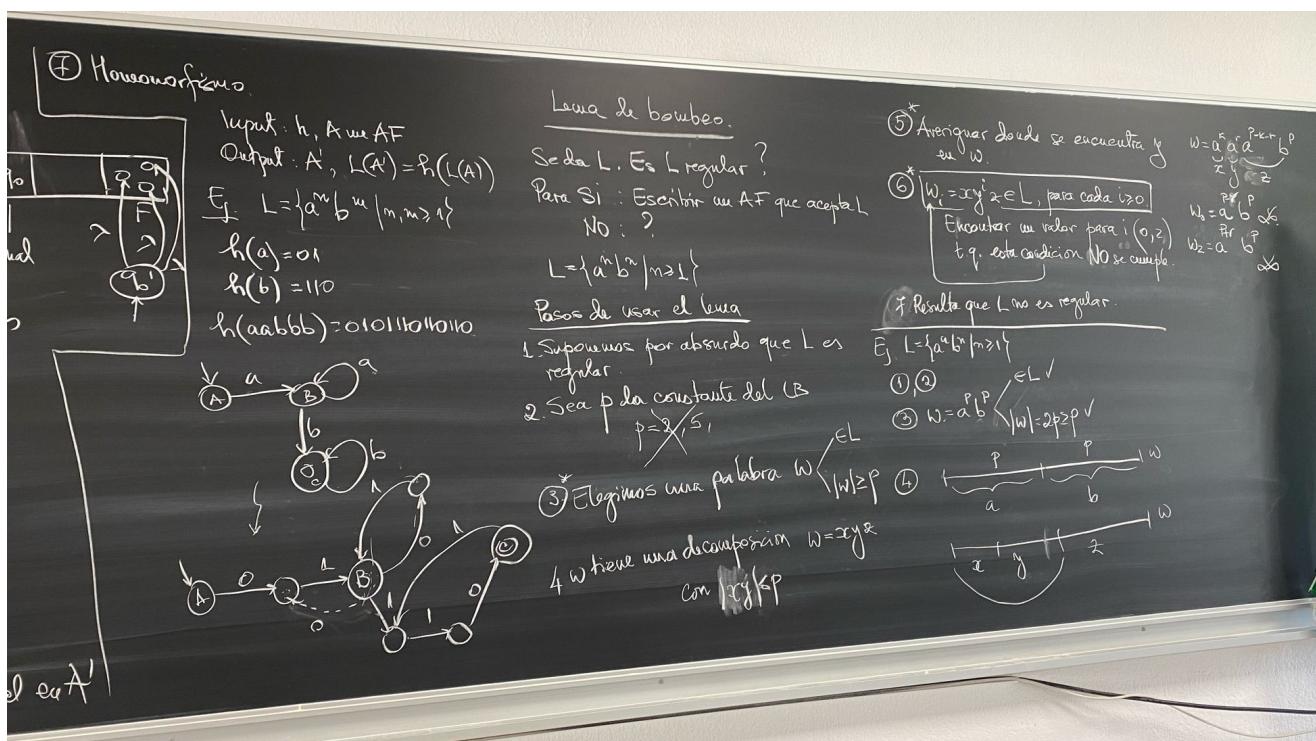
## ► Notes

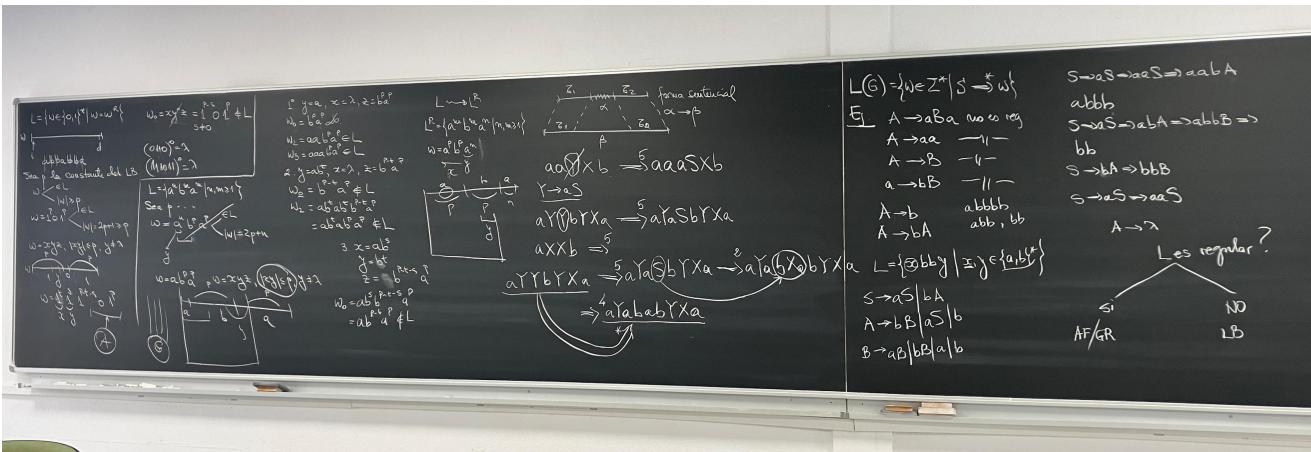
## 5. Properties and Operations





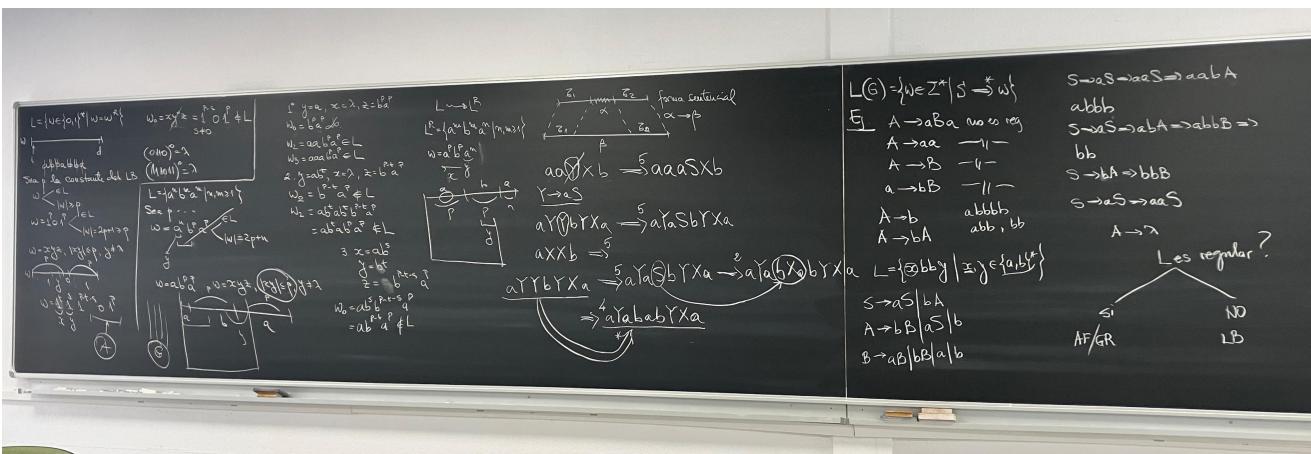
## 6. Pumping





## ► Apuntes

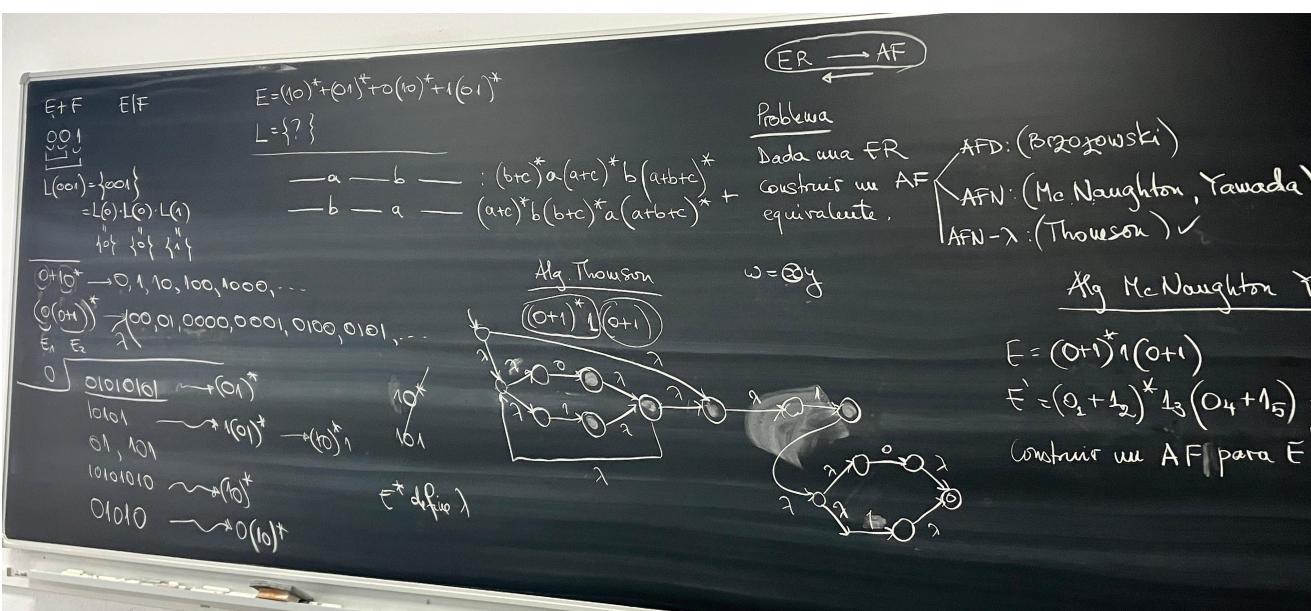
## 7. Grammar



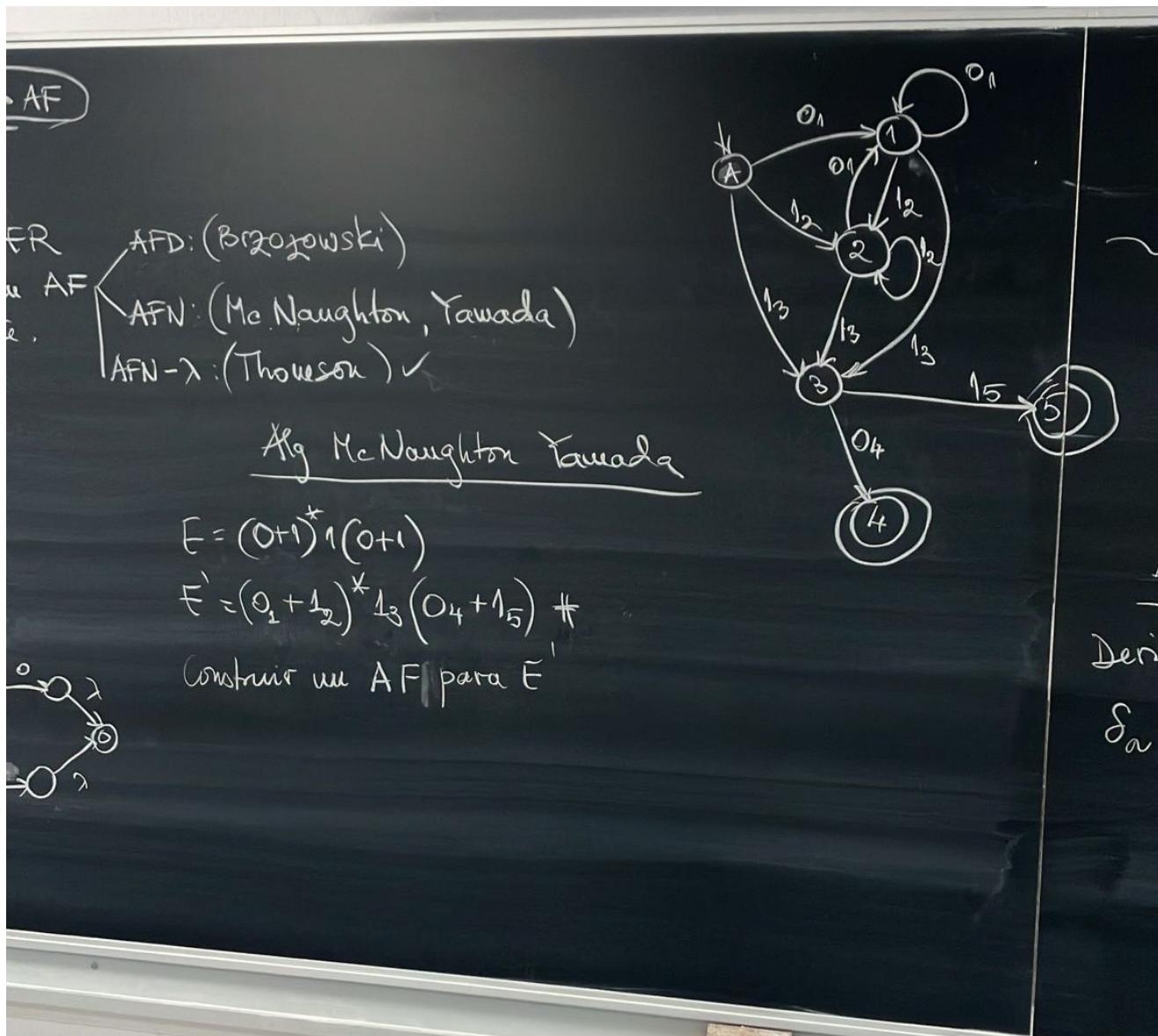
## ► Apuntes

## 8. ER a Automata

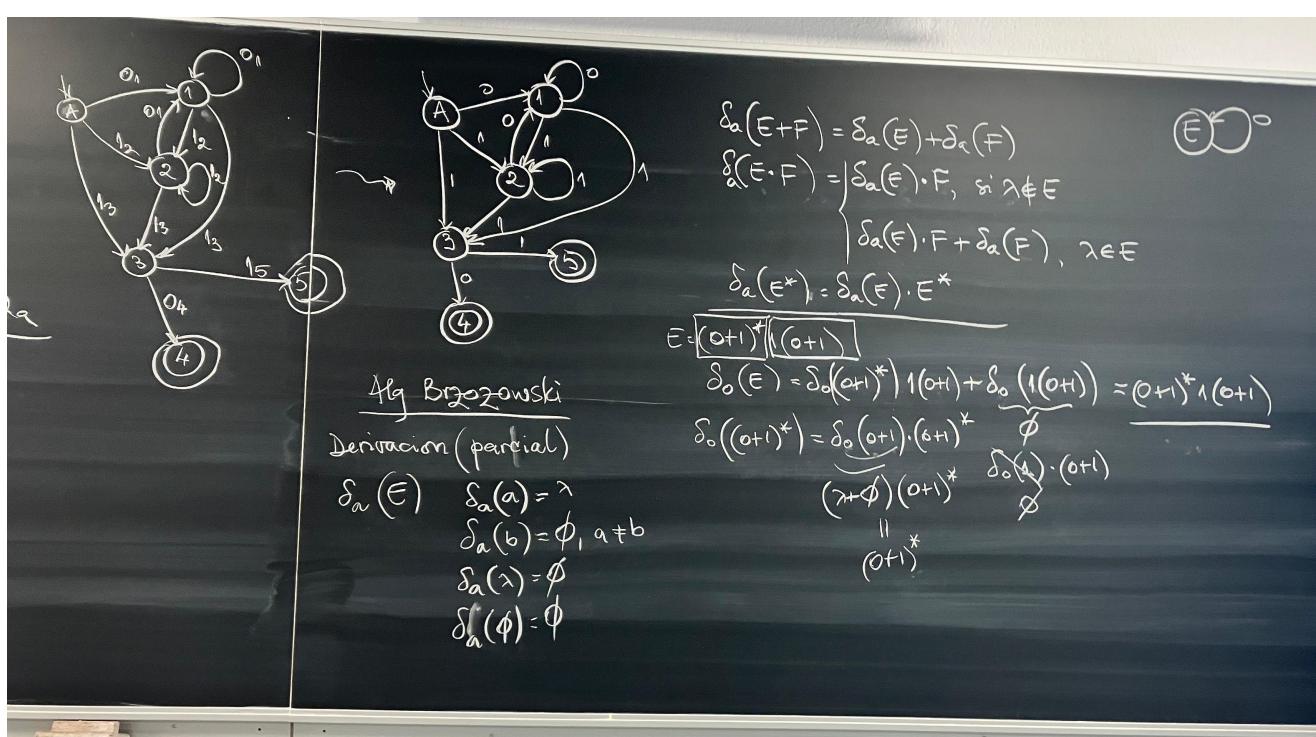
### 8.1. Thomson (NFA- $\epsilon$ )



### 8.2. M'Naghten Yamada (NFA)



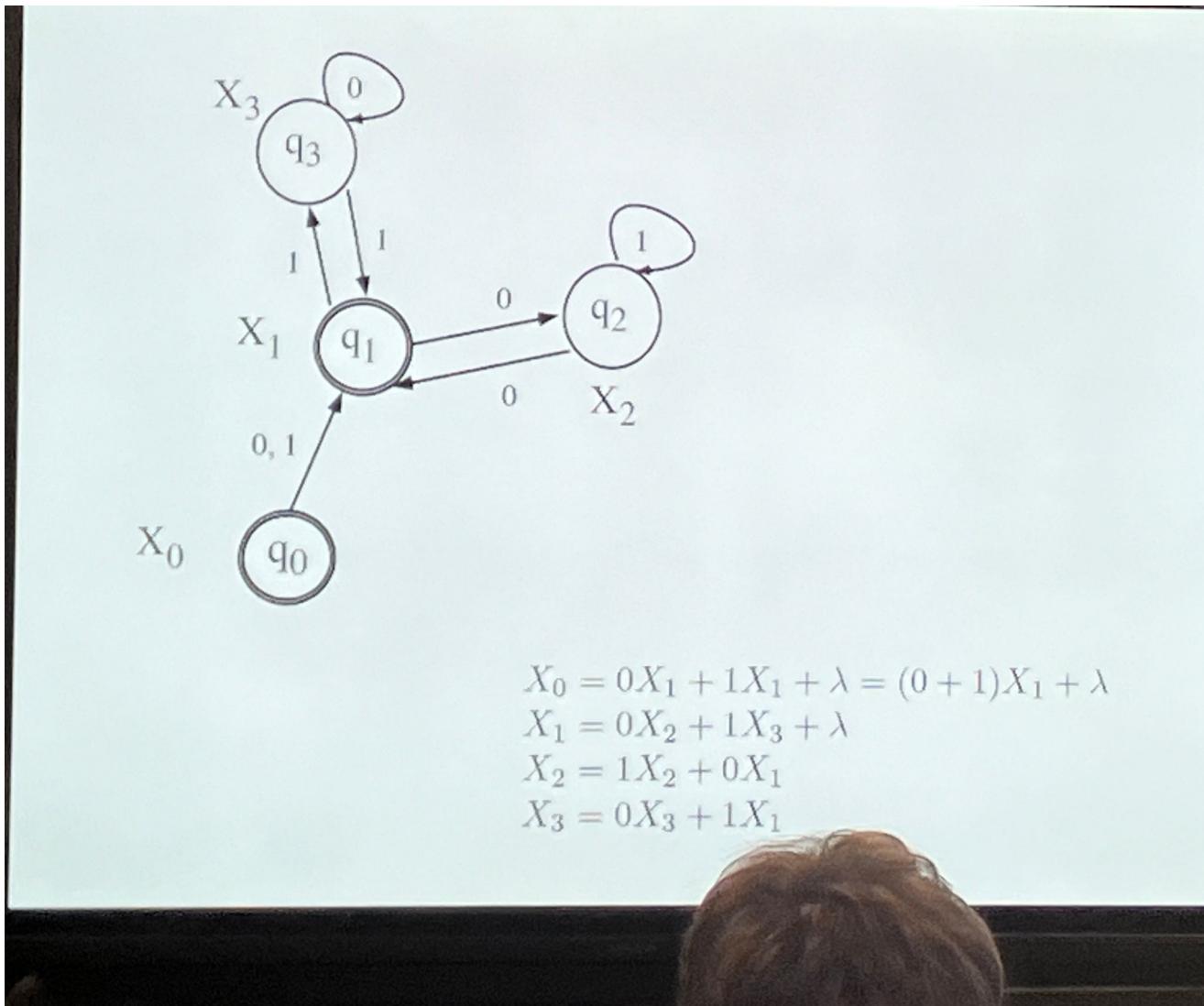
### 8.3 Brzozowski (DFA)



## ► Apuntes

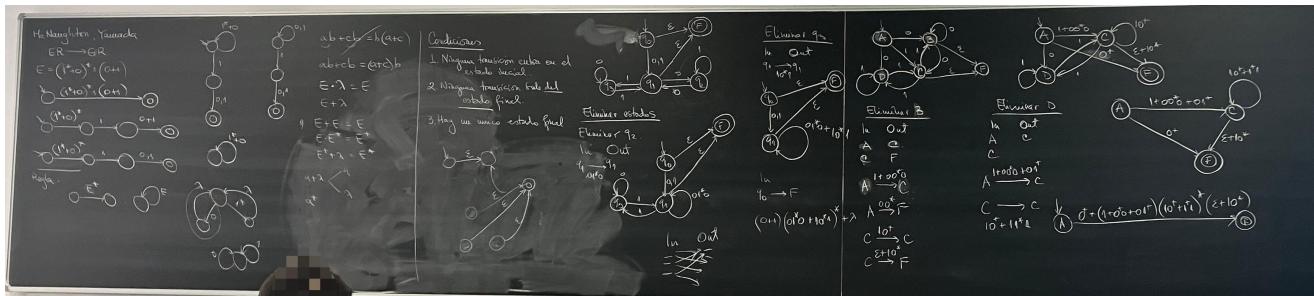
## 9. Automata a ER

## 9.1. Mediante Sistema de Ecuaciones



## ► Apuntes

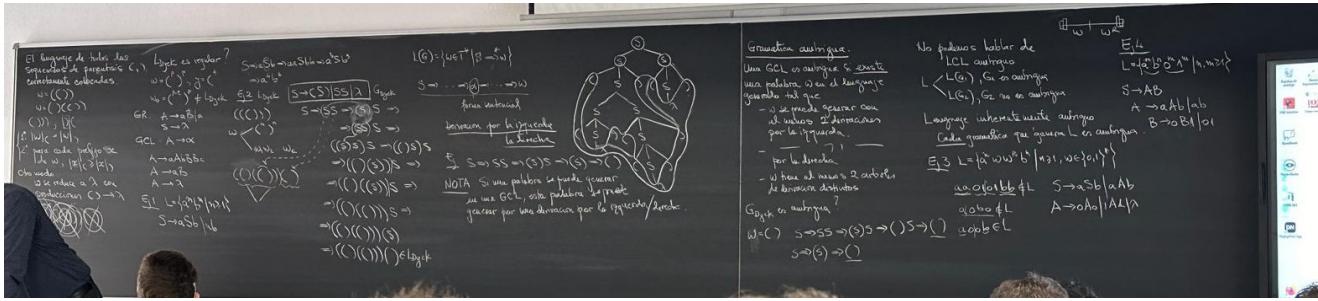
## 9.2. Mediante Eliminación de Estados



## ► Apuntes

## **10. ER a Gramatica POR HACER**

## **11. Gramáticas de Contexto Libre - GCL**



## 11.1. Árbol de Derivación

## *Representación gráfica del proceso de derivación de una gramática.*

- **Raiz:** Nodo inicial
  - **Nodos:** Símbolos No Terminales
  - **Hojas:** Símbolos Terminales

## 11.2 Gramáticas Ambiguas

*Generan palabras por más de un árbol de derivación. No hay como detectar gramáticas ambiguas.*

Es decir, se puede generar la misma palabra de distintas formas.

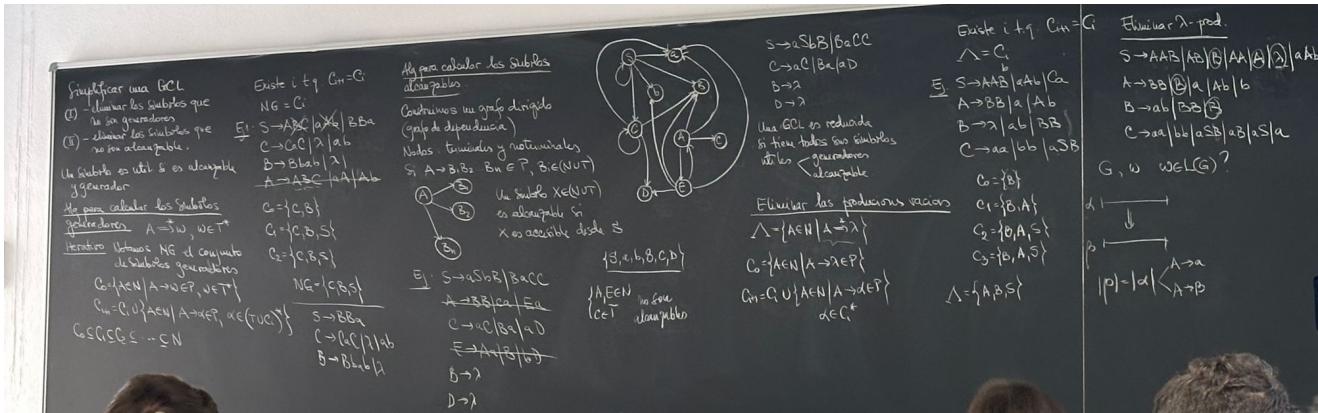
### 11.2.1. Lenguajes inherentemente ambiguos

*Todas gramáticas que generan el lenguaje son ambiguas.*

## 11.2.2. Grado de Ambigüedad

*Cantidad de árboles de derivación que generan una palabra.*

## 11.3 Limpiar Gramáticas



*Consiste en quitar los símbolos inútiles e innaccesibles.*



El profe y las diapositivas ponen nombres distintos a términos de este apartado.

Diapositivas	Clase
Símbolos Inútiles	Símbolos No Generadores
Símbolos Inaccesibles	Símbolos No Alcanzables
Útil y Accesible	Útil

### 11.3.1. Símbolos Inútiles

*Símbolos no terminales a través de los cuales no se puede llegar a una palabra.*

### 11.3.2. Símbolos Inaccesibles

*Símbolos que no se pueden alcanzar desde el símbolo inicial.*

### 11.3.3. Eliminación de Símbolos Inútiles - No Generadores

El símbolo de la parte izquierda de una derivación directa  $A \rightarrow w : w \in \sigma^*$  es útil si:

- Al menos un símbolo de la parte derecha de la derivación es útiles.



El profe se complica demasiado la vida calculando  $C$ , solo hay q ir probando para cada  $A$  si se puede llegar a un terminal de alguna forma.

*Repetir este proceso recursivamente.*

### 11.3.4 Eliminación de Símbolos Inaccesibles - No Alcanzables

El símbolo de la parte derecha de una derivación directa  $A \rightarrow w : w \in \sigma^*$  es accesible si:

- El símbolo de la parte izquierda de la derivación es accesible.

*Repetir este proceso recursivamente.*

### 11.3.5 GCL Reducida

*Todos sus símbolos son:*

- **Alcanzables**
- **Generadores**

Es decir, son *Útiles*.

## 11.4 Transformar Gramáticas de Tipo 2

Existe  $i$  t.q.  $C_{ii} = G$

$$\Delta = \{G\}$$

Ej.

$$S \rightarrow AAB \mid aAb \mid Ca$$

$$A \rightarrow BB \mid a \mid Ab$$

$$B \rightarrow \lambda \mid ab \mid BB$$

$$C \rightarrow aa \mid bb \mid aSB$$

$$Co = \{B\}$$

$$C_1 = \{B, A\}$$

$$C_2 = \{B, A, S\}$$

$$C_3 = \{B, A, S\}$$

$$\Delta = \{A, B, S\}$$

or

Eliminar  $\lambda$ -prod.

Ej.

$$S \rightarrow AAB \mid AB \mid \lambda \mid AA \mid A \mid \lambda \mid aAb \mid ab \mid Ca$$

$$A \rightarrow BB \mid B \mid a \mid Ab \mid b$$

$$B \rightarrow ab \mid BB \mid \lambda$$

$$C \rightarrow aa \mid bb \mid aSB \mid aB \mid aS \mid a$$

$G, w \in L(G) ?$

$\Delta \vdash \downarrow$

$\beta \vdash \downarrow$

$|P| = |\alpha| \begin{cases} A \rightarrow a \\ A \rightarrow B \end{cases}$

Producciones unitarias

$A \rightarrow B$

(i)  $A \rightarrow A$  se elimina directamente  
(ii)  $A \rightarrow B$  añadido a las prod de  $A$   
 $B \rightarrow d \in P$        $A \rightarrow d$

(iii) Si aparece una prod  $A \rightarrow B$  considerada ya se ignora

Ej.

$$S \rightarrow AAB \mid AB \mid ab \mid BB \mid AA \mid \lambda \mid aAb \mid ab \mid Ca \mid a \mid Ab \mid b$$

$$B \rightarrow ab \mid BB$$

$$A \rightarrow BB \mid a \mid Ab \mid b \mid ab$$

$$C \rightarrow aa \mid bb \mid aSB \mid aB \mid aS \mid a$$

Se basa en eliminar **Producciones Unitarias** y **Producciones Vacías**.

#### 11.4.1 Eliminar Producciones Vacías - $\lambda$ -prod.

Se basa en actualizar las derivaciones quitando de cada Símbolo, transiciones lambda.

##### Note

Una forma que me sirve es escribir en una tabla el símbolo y su transición lambda y en filas las transiciones que derivan en el símbolo. Luego teniendo en cuenta que el símbolo ya no puede ser lambda, sacamos todas combinaciones posibles de las transiciones.

Prod. $B \rightarrow \lambda$	Prod. $A \rightarrow \lambda$	Prod. $S \rightarrow \lambda$	Prod $S \rightarrow \lambda$
$B \rightarrow BB$	$B$		
$S \rightarrow AAB$	$AA$	$A \parallel \lambda$	
$A \rightarrow BB$	$B \parallel \lambda$		
$C \rightarrow aSB$	$aS$		$a$
	$S \rightarrow aAb$	$ab$	
	$A \rightarrow Ab$	$b$	

##### Note

Al acabar, juntamos todas descomposiciones.

#### 11.4.2 Eliminar Producciones Unitarias

Se basa en actualizar las derivaciones, de forma que no existan transiciones unitarias - del tipo  $A \rightarrow B$ .

- $A \rightarrow A$  : Borramos.

- $A \rightarrow B$  : Añadimos a  $A$  las derivaciones de  $B$ .

*Forma guay:*  $\forall A \rightarrow B : B \rightarrow \alpha_n$ , pasar a:  $A \rightarrow \alpha_n$

## 11.5 Forma Normal de Chomsky - FNC

Cuando el LCL (Lenguaje de Contexto Libre) se genera por una gramática donde las producciones son de la forma:

- $A \rightarrow BC$
- $A \rightarrow a$

Para ello tenemos que:

1. **Limpiar la Gramática**
  - **Eliminar Prefijos Comunes (Eliminar Ambigüedad)**
    1. **Eliminar Producciones- $\lambda$**
    2. **Eliminar Producciones Unitarias**
    3. **Eliminar Símbolos Inútiles**
2. Producciones con **2 o más implicados** son siempre **no terminales**.
3. Producciones con **3 o más implicados** divididas en producciones de **dos variables**.

### 11.5.1 Eliminar Ambigüedad de GCL

P

~~Gramática no ambigua:~~

Cada palabra en el lenguaje generado tiene una UNICA derivación por la izquierda.

alizador  
retáctico

Prefijos comunes

$$A \rightarrow \alpha p \mid \beta q \dots$$

$$\begin{array}{l} S \rightarrow aT \\ T \rightarrow bX \mid S \end{array}$$

Eliminar los prefijos comunes

$$A \rightarrow \alpha p_1 \mid \alpha p_2 \mid \dots \mid \alpha p_n \mid \beta_1 \mid \dots \mid \beta_m$$

$$\begin{array}{l} A \rightarrow \beta_1 \mid \dots \mid \beta_m \mid \alpha X \\ X \rightarrow p_1 \mid p_2 \mid \dots \mid p_n \end{array}$$

$$\begin{array}{l} S \rightarrow abX \mid aS \\ vX \rightarrow AaB \mid C \end{array}$$

Ej -  $S \rightarrow \underline{ab}AaB \mid \underline{ab}C \mid aS$

$$A \rightarrow bAb \mid bAc$$

$$B \rightarrow X$$

$$C \rightarrow \underline{abc} \mid \underline{acb}$$

$$\begin{array}{l} A \rightarrow bAE \\ E \rightarrow bA \mid C \end{array}$$

$$\begin{array}{l} C \rightarrow aD \\ D \rightarrow bc \mid cb \end{array}$$

Eliminar los prefijos comunes  
Recursividad (inmediata) por la izquierda

$$A \rightarrow A\alpha \quad S \Rightarrow^* x A \beta \xrightarrow{\text{m}} x A \alpha \beta \Rightarrow x A \alpha \alpha \beta$$

Eliminar la recursividad por la izquierda

$$\oplus A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \underbrace{\beta_1 | \beta_2 | \dots | \beta_m}_{\text{recursivas}}$$

$$\rightarrow A \rightarrow \beta_1 X | \beta_2 X | \dots | \beta_m X$$

$$X \rightarrow \alpha_1 X | \alpha_2 X | \dots | \alpha_n X | \lambda \}$$

$$A \Rightarrow A\alpha_i \rightarrow A\alpha_i \alpha_i \rightarrow \underbrace{A\alpha_i \alpha_i \alpha_i}_{\text{prediccion}} \rightarrow \underbrace{\beta_{r+1} \alpha_i \alpha_i}_{\text{prediccion}}$$

$$A \rightarrow \beta_r X \Rightarrow \beta_{r+1} X \Rightarrow \beta_{r+1} \alpha_j X \Rightarrow \beta_{r+1} \alpha_j \alpha_i X$$

Ej.

$$S \rightarrow S(S) | aSBA | (\ ) | S\alpha = \underbrace{\beta_{r+1} \alpha_j \alpha_i}_{\text{prediccion}}$$

$$A \rightarrow B(B) | A(AB) | (b)$$

$$B \rightarrow B\alpha | B\beta$$

$$\left\{ \begin{array}{l} S \rightarrow aSBAX | X \\ X \rightarrow SX | aX | \lambda \\ A \rightarrow BBY | bY \\ Y \rightarrow ABY | \lambda \\ B \rightarrow Z \\ Z \rightarrow aZ | bZ \end{array} \right.$$

$$Z \Rightarrow a$$

$$B \rightarrow \alpha$$

$$\oplus E \rightarrow E+E | E \times E$$

$$\omega = aaaa$$

$$E \Rightarrow E+E$$

$$1,1,$$

Para pasar de una Gramática Ambigua a una no ambigua, buscamos eliminar los Prefijos Comunes.

Para **Prefijos Comunes Normales**:

1. Sacar "Factor Común" del Símbolo.
2. Crear nuevo **No Terminal** con los implicados del original.

### Note

El paso 2 solo se realiza en producciones con más de un símbolo en cada implicado.

Para **Prefijos Comunes Recursivos Inmediatos por la Izquierda**:

**Input:**

- $A \rightarrow A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_n$

**Output:**

- $A \rightarrow \beta_1 B | \dots | \beta_n B$
- $B \rightarrow \alpha_1 B | \dots | \alpha_n B | \lambda$

Donde:

- $A$ : Símbolo No terminal.
- $B$ : Símbolo No terminal.
- $\alpha$ : Subpalabra que procede  $A$ .
- $\beta$ : Subpalabra que no procede  $A$ .

#### Note

##### **Mi razonamiento:**

Sabemos que en una transición  $A \rightarrow pAs|a|b$ ,  $p$  actúa como prefijos,  $s$  como sufijos y que el cuerpo de  $A$  podría ser o  $a$  o  $b$ .

En el caso de **Recursividad Inmediata**, los prefijos no están presentes, quedando:  $A \rightarrow As|a|b$ . Es decir, el cuerpo de  $A$  estará compuesto por o  $a$  o  $b$  y luego sufijos  $s$ , podemos traducir esto a:  $A \rightarrow aB|bB$ , siendo  $B$  los sufijos  $s$ :  $B \rightarrow sB|\lambda$ .

Ahora podemos generalizar para cualquier producción y sacar la fórmula dada.  $s$  será cualquier subpalabra que proceda  $A$  ( $\alpha$ ) y las subpalabras  $a$  y  $b$  serán cualquier subpalabra que no proceda  $A$  ( $\beta$ ).

## 11.5.2 Prefijo Común

Dada una producción  $A \rightarrow X|Y$ , esta producción es ambigua si los símbolos iniciales de  $X$  e  $Y$  son iguales.

#### Note

##### *Ejemplo:*

$A \rightarrow ab|a$

$A \rightarrow Ab|Aa|a$  - Recursividad Inmediata por la Izquierda!

## 11.5.3 Paso 2

Para realizar el paso 2:

1. Creamos una nueva producción para cada producción existente donde haya dos o más implicados donde no son todos no terminales.
2. Intercambiamos en producciones originales los símbolos terminales por los nuevos símbolos no terminales.
3. Las producciones nuevas producirán los símbolos terminales.

#### Note

##### *Ejemplo:*

- $A \rightarrow aAB|B$
- $B \rightarrow b$

Como  $A$  produce tanto no terminales como terminales, crearemos una nueva producción que produzca  $a$ , quedando:

- $A \rightarrow CAB|B$
- $B \rightarrow b$
- $C \rightarrow a$

#### 11.5.4 Paso 3

Para realizar el paso 3:

1. Creamos una nueva producción para cada producción existente donde se produzca más de 2 terminales.
2. Intercambiamos en las producciones originales dos implicados por la nueva producción.
3. La producción nueva tendrá 2 de los implicados de la producción original.



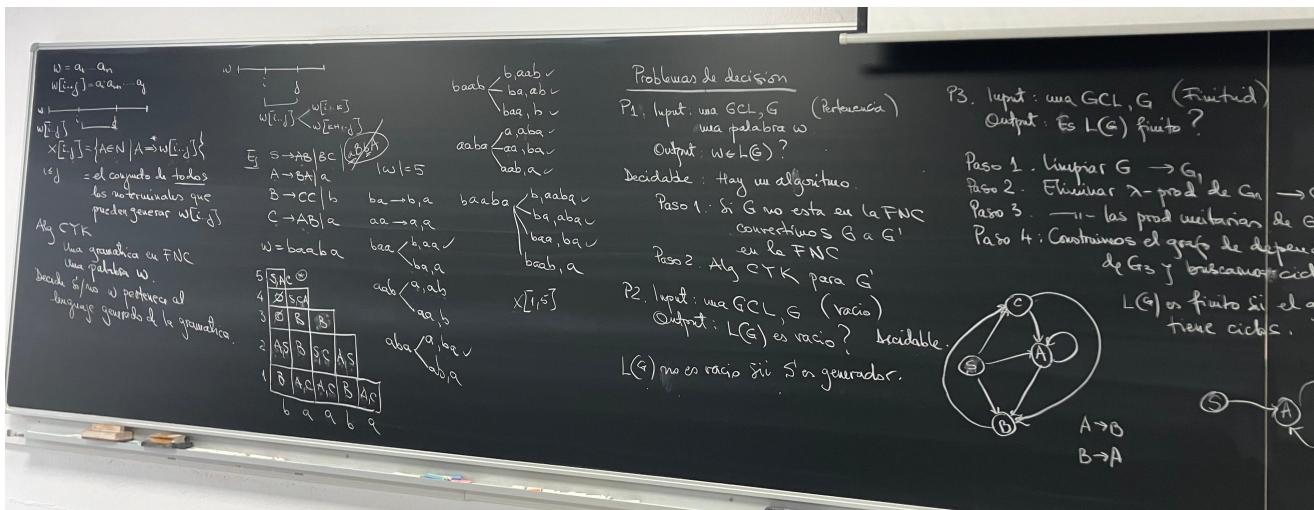
**Ejemplo:**

- $A \rightarrow CAB|B$
- $B \rightarrow b$
- $C \rightarrow a$

Como  $CAB$  tiene tres implicados, crearemos una nueva producción que produzca 2 de ellos - elegimos  $CA$  -, quedando::

- $A \rightarrow DB|B$
- $B \rightarrow b$
- $C \rightarrow a$
- $D \rightarrow CA$

#### 11.6 Algoritmo CYK



Dado una gramática en FNC y una palabra  $w$ , nos dice si la palabra pertenece al lenguaje generado por la gramática.

### ⚠ Warning

La gramática debe estar en FNC.

1. Construimos una matriz triangular inferior de  $n \times x$ .
2. En el eje X escribimos carácter por carácter, la palabra a probar ( $w$ ).
3. Escribimos en la fila más baja el conjunto de símbolos que nos posibilita llegar directamente al carácter de abajo.
4. Escribimos en la fila arriba de la anterior, el conjunto de símbolos que nos posibilita llegar a la palabra actual - *teniendo en cuenta todas combinaciones posibles*. Para esto tenemos en cuenta los resultados que hemos sacado anteriormente de cada conjunto de caracteres.

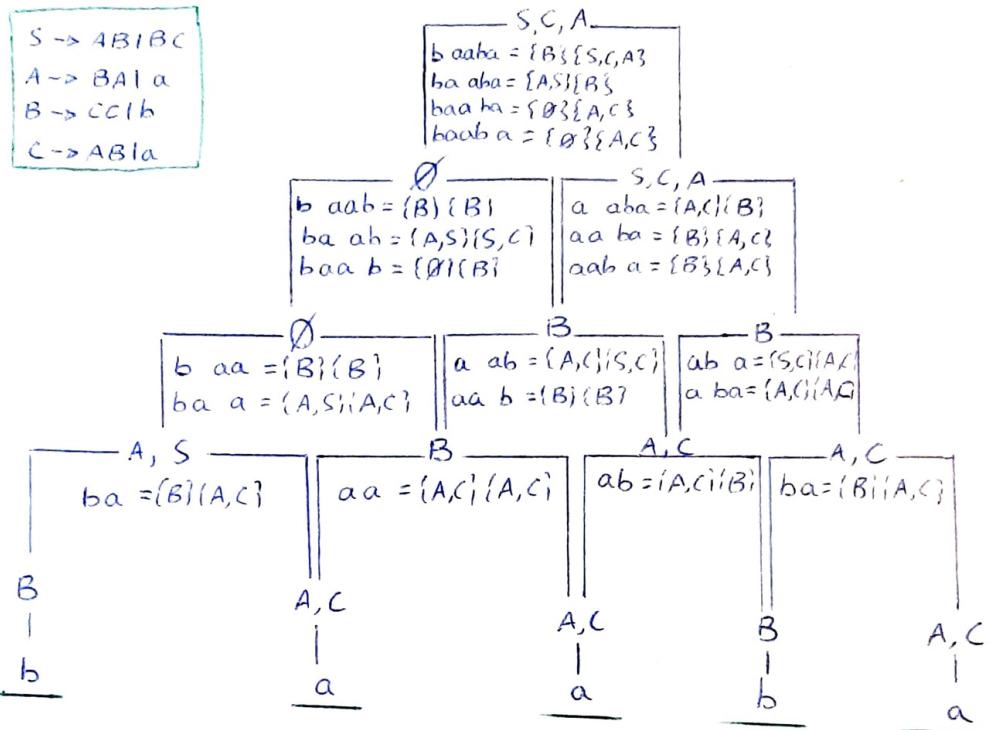
### >Note

Personalmente, prefiero hacer una pirámide que un triángulo. Es decir, poner el resultado de la fila superior entre las columnas de abajo.

### Ejemplo:

$$G = S \rightarrow AB|BC, A \rightarrow BA|a, B \rightarrow CC|b, C \rightarrow AB|a$$

$$w = baaba$$



1. Escribimos los caracteres.
2. Para cada carácter, buscamos símbolos que les produzcan. *Ej. Para "a", los símbolos que le producen son: A y C, por lo tanto lo apuntamos.*
3. Para cada palabra, para cada posible descomposición, buscamos símbolos que les produzcan. *Ej. Para la palabra "ba", su descomposición es: "b a", por lo tanto su conjunto de símbolos implicados será la combinatoria entre los símbolos que producen "b" y de los que producen "a", es decir:  $\{B\} \times \{A, C\} = \{BA, BC\}$ , ahora buscamos producciones que contienen estos implicados y apuntamos los implicantes.*
4. Repetimos el paso 3 hasta que toda file de  $\emptyset$  o que lleguemos, a un valor.

### Note

Podemos comprobar que  $w \in L$ , ya que tenemos una serie de símbolos de la gramática que juntos generan esa palabra. En este caso:  $S, C, A$ .

## 11.7 Ejemplo: $w \in L(G)$

Para probar que sí, tenemos que ser capaces de crear un árbol de derivación por la izquierda.

**Ej. FNC**

$S \rightarrow A$   
 $A \rightarrow a$   
 $B \rightarrow b$   
 $C \rightarrow c$   
 $D \rightarrow d$

**Ej. ⑤**

$G \xrightarrow{①, ②, ③, ④, ⑤} G'$   
 (en FNC)

**Input:** una gramática de contexto libre  $G$   
 una cadena  $w$   
**Output:**  $WF(G) ?$

Si  $\rightarrow$  Árbol de derivación  
 Derivación por la izquierda

$G$  es una GCL arbitraria

① Limpia/simplificar  $G$   
 • eliminar los símbolos no generadores  
 • eliminar los símbolos inaccesibles

② Eliminar las producciones vacías ( $\lambda$ -prod)

③ Eliminar las producciones unitarias

Forja Normal Chomsky

$S \rightarrow a, A \in N, a \in T$   
 $A \rightarrow BC, A, B, C \in N$

**④**  $A \rightarrow a \quad \alpha \in N \quad (\alpha \neq \lambda)$   
 $\alpha \in NT^*$

$A \rightarrow abBAab \rightarrow \left\{ \begin{array}{l} A \rightarrow XYBAXB \\ X \rightarrow a \\ Y \rightarrow b \end{array} \right.$

$A \rightarrow a \quad | \quad A \in N$

$A \rightarrow B \xrightarrow{\lambda}$

**⑤**  $A \rightarrow B_1 \dots B_n \quad n \geq 3$

$A \rightarrow B_1 C_1$   
 $C_1 \rightarrow B_2 C_2$   
 $C_2 \rightarrow \dots$   
 $C_{n-2} \rightarrow B_{n-1} B_n$

**Árbol:** transformar las prod no permitidas de la FNC Ieu

$A \rightarrow B_1 B_2 \dots B_n, B_i \in N$

**Ej. FNC**

$A \rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} B \xrightarrow{\lambda} C \xrightarrow{\lambda} D$

$A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$   
 $A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$

$\Rightarrow A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$

$A \rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} Z \xrightarrow{\lambda} V$

$Z \rightarrow c$

$A \rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} C_1 \xrightarrow{\lambda} C_2 \dots \xrightarrow{\lambda} C_n \xrightarrow{\lambda} V$

$C_1 \rightarrow B_1 \dots B_n$

$C_2 \rightarrow B_2 \dots B_n$

$C_n \rightarrow B_n$

**Árbol:** transformar las prod no permitidas de la FNC Ieu

$A \rightarrow B_1 B_2 \dots B_n, B_i \in N$

$A \rightarrow a \quad | \quad a \in T$

$A \rightarrow B \xrightarrow{\lambda}$

$A \rightarrow aabbabc \rightarrow \left\{ \begin{array}{l} A \rightarrow XXYYZZV \\ X \rightarrow a \\ Y \rightarrow b \\ Z \rightarrow c \end{array} \right.$

**Árbol:** transformar las prod no permitidas de la FNC Ieu

$A \rightarrow B_1 B_2 \dots B_n, B_i \in N$

$A \rightarrow a \quad | \quad a \in T$

$A \rightarrow B \xrightarrow{\lambda}$

$A \rightarrow aabbabc \rightarrow \left\{ \begin{array}{l} A \rightarrow XXYYZZV \\ X \rightarrow a \\ Y \rightarrow b \\ Z \rightarrow c \end{array} \right.$

**Ej. ⑤**

$A \rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} BAXB$

$A \rightarrow XC_1$   
 $C_1 \rightarrow YC_2$   
 $C_2 \rightarrow BC_3$   
 $C_3 \rightarrow AC_4$   
 $C_4 \rightarrow XB$

$A \rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} C_2$

$\Rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} BC_3 \Rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} BAC_4$

$\Rightarrow X \xrightarrow{\lambda} Y \xrightarrow{\lambda} BAXB$

**① Li**  
**Gene**  
**Acc**

**S-**  
**A**  
**B**

**Ej. FNC**

$S \rightarrow ASB | AB | CD$   
 $A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$   
 $A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$

$\Rightarrow A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$

**② Eliminar  $\lambda$ -prod.**

$\lambda = \{a, b\}$

$S \rightarrow ASB | AB | AS | \lambda | \lambda | \lambda | \lambda$   
 $A \rightarrow aAS \quad | \quad a \in T$   
 $B \rightarrow bSb \quad | \quad S \in N$

$\Rightarrow S \rightarrow ASB | AB | AS | a | b | a | a$

**③ Limpieza**

$A \rightarrow a$   
 $B \rightarrow b$   
 $C \rightarrow SB$   
 $C_1 \rightarrow AS$   
 $C_2 \rightarrow YC_3$   
 $C_3 \rightarrow SY$

$A \rightarrow XC_2 | XS | XA | a$   
 $B \rightarrow YY | SC_3 | YC_4 | C_4 Y | XC_2 | XS | XA | a$   
 $X \rightarrow a$   
 $Y \rightarrow b$

**④ Eliminar las prod unitarias**

$A \rightarrow A$   
 $B \rightarrow b$   
 $C \rightarrow SB$   
 $C_1 \rightarrow AS$   
 $C_2 \rightarrow YC_3$   
 $C_3 \rightarrow SY$

$A \rightarrow XC_2 | XS | XA | a$   
 $B \rightarrow YY | SC_3 | YC_4 | C_4 Y | XC_2 | XS | XA | a$   
 $X \rightarrow a$   
 $Y \rightarrow b$

**⑤**  $S \rightarrow AC_1 | SB | AB | AS | \lambda | XC_2 | XS | XA | a | YY | SC_3 | YC_4 | C_4 Y$

$C_1 \rightarrow SB$   
 $C_2 \rightarrow AS$   
 $C_3 \rightarrow YC_4$   
 $C_4 \rightarrow SY$

$A \rightarrow XC_2 | XS | XA | a$   
 $B \rightarrow YY | SC_3 | YC_4 | C_4 Y | XC_2 | XS | XA | a$

$X \rightarrow a$   
 $Y \rightarrow b$

**Algoritmo:** CYK

\* Cocke - Younger - Kasami

**Argentina CYK**

**Cocke - Younger -**

**⑤**  $S \rightarrow AC_1 | SB | AB | AS | \lambda | XC_2 | XS | XA | a | YY | SC_3 | YC_4 | C_4 Y$

$C_1 \rightarrow SB$   
 $C_2 \rightarrow AS$   
 $C_3 \rightarrow YC_4$   
 $C_4 \rightarrow SY$

$A \rightarrow XC_2 | XS | XA | a$   
 $B \rightarrow YY | SC_3 | YC_4 | C_4 Y | XC_2 | XS | XA | a$

$X \rightarrow a$   
 $Y \rightarrow b$

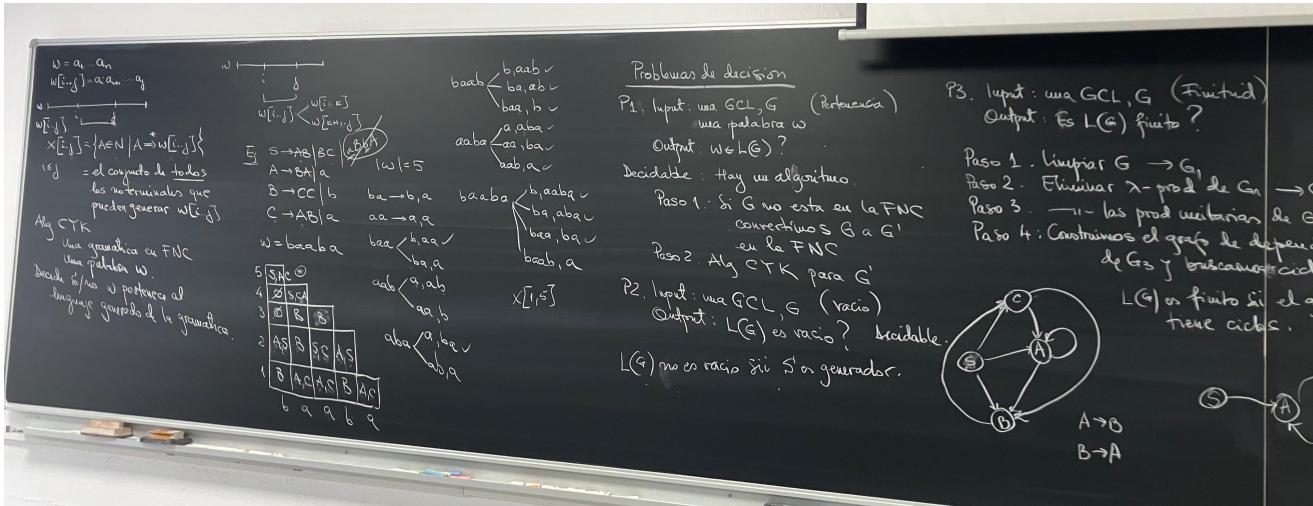
**Algoritmo:** CYK

\* Cocke - Younger - Kasami

**Diagrama:**

W → S

## 11.8 Problemas de Decisión



### 11.8.1 Problema - Pertenencia de Palabra

$w \in L(G)$  ?

Para resolver este problema:

1. Pasar  $G$  a FNC  $\rightarrow G_1$ .
2. Realizar CYK para  $w$  en  $G_1$ .

### 11.8.2 Problema - Lenguaje Vacío

$L(G)$  vacío?

Para resolver este problema:

- Probar si  $S$  es **generador**.

### 11.8.3 Problema - Lenguaje Finito

$L(G)$  finito?

Para resolver este problema:

1. Limpiar:  $G \rightarrow G_1$ .
2. Eliminar  $\lambda$ -prod:  $G_1 \rightarrow G_2$ .
3. Eliminar prod. unitarias:  $G_2 \rightarrow G_3$ .
4. Construir grafo de dependencia de  $G_3$ .

$L(G)$  será infinito cuando exista al menos un ciclo - Ej.  $A \rightarrow AB$ .

### 11.8.4 Problemas no Decidibles

- GCL **ambigua**?
- LLC **inherente amibiguo**?
- $GLC \cap GLC = \emptyset$ ?
- $GLC = GLC$ ?

## 11.9 Propiedades de Cerradura

(Perfección) P3. Input: una GCL,  $G$  (Finito)  
Output: Es  $L(G)$  finito?

Algoritmo:  
esta en la FNC  
nodos  $G$  a  $G'$   
 $G'$  (vacío)  
vacío? Decidible.  
Sí es generador.

Paso 1. Limpiar  $G \rightarrow G_1$   
Paso 2. Eliminar  $\rightarrow$ -prod de  $G_1 \rightarrow G_2$   
Paso 3.  $\rightarrow$ -prod unitario de  $G_2 \rightarrow G_3$   
Paso 4. Construir el grafo de dependencias de  $G_3$  y buscar ciclos  
 $L(G)$  es finito si el grafo no tiene ciclos.

$A \rightarrow B$   
 $B \rightarrow A$

P4 Input: dos GCL,  $G_1, G_2$ .  
Output:  $L(G_1) = L(G_2)$ ?  
No decidable!

Propiedades de cerradura

1º Unión:  $G_1, G_2, L(G_1) \cup L(G_2)$  es LCL  
 $G_1 = (N_1, T_1, P_1)$   $N_1 \cap N_2 = \emptyset$   
 $G_2 = (N_2, T_2, P_2)$   
 $L(G) = L(G_1) \cup L(G_2)$

2º Concatenación  
 $G_1 = (N_1, T_1, P_1)$   $N_1 \cap N_2 = \emptyset$   
 $G_2 = (N_2, T_2, P_2)$   
 $G = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\})$

3º Kleene  
 $G = (N, T, P)$   
 $G^* = (N \cup \{S\}, T \cup \{S\}, P \cup \{S \rightarrow S\}^*)$   
 $S \rightarrow S_1 \mid S_2$   
 $S \rightarrow S_1 \mid S_2 \rightarrow S \rightarrow S \rightarrow S \rightarrow S \rightarrow S$   
 $\Rightarrow SSS$

Sean  $L_1, L_2$  LLC's (Lenguajes de Libre Contexto) y  $L_3$  LR (Lenguaje Regular):

PROPIEDAD	OPERACIÓN	RESULTADO
Unión	$L = L_1 \cup L_2$	LLC
Intersección - LLC	$L = L_1 \cap L_2$	LLC
Intersección - LR	$L = L_1 \cap L_3$	?
Concatenación	$L = L_1 L_2$	LLC
Kleene	$L = L_1^*$	LLC
Inversión	$L = L_1^R$	LLC
Complementario	$L = L_1^C$	?
Diferencia	$L = L_1 - L_2$	?

## 11.10 Lema de Bombeo - GCL [POR HACER]

Lema de bombeo

Los que LCL.  
Existe una constante  $p \geq 1$  tq para cada palabra  $z \in L$  con  $|z| \geq p$

- $z = uvwxyz$
- $|vwx| \leq p$
- $|wx| \neq \emptyset$
- Para cada  $i \geq 0$   $zi \in L$

Ej:  $L = \{a^n b^n c^n \mid n \geq 1\}$

P1 Sea  $p$  la constante del LB

P2 Elegir  $z \in L$  con  $|z| \geq p$

P3. Examinar todas las posiciones donde se colocan  $v$  y  $w$ .

P4 Para cada posición elegir un valor de  $i \geq 0$  tq  $zi \notin L$  (En general  $i = 0, 2$  vale)

S1 Sea  $z = a^p b^p c^p$ ,  $|z| = 3p \geq p$

z  $\xrightarrow{\quad a \quad} \xrightarrow{\quad b \quad} \xrightarrow{\quad c \quad}$   
 $\uparrow \quad \uparrow \quad \uparrow$   
 Caso 1:  $\xrightarrow{\quad a \quad} \xrightarrow{\quad b \quad} \xrightarrow{\quad c \quad}$   
 $\uparrow \quad \uparrow \quad \uparrow$   
 Caso 2:  $\xrightarrow{\quad a \quad} \xrightarrow{\quad b \quad} \xrightarrow{\quad c \quad}$   
 $\uparrow \quad \uparrow \quad \uparrow$   
 Caso 3:  $\xrightarrow{\quad a \quad} \xrightarrow{\quad b \quad} \xrightarrow{\quad c \quad}$   
 $\uparrow \quad \uparrow \quad \uparrow$   
 Caso 4:  $\xrightarrow{\quad a \quad} \xrightarrow{\quad b \quad} \xrightarrow{\quad c \quad}$   
 $\uparrow \quad \uparrow \quad \uparrow$   
 Caso 5:  $\xrightarrow{\quad a \quad} \xrightarrow{\quad b \quad} \xrightarrow{\quad c \quad}$   
 $\uparrow \quad \uparrow \quad \uparrow$

Caso 1.  $v = a^p$  ( $vwx = a^r$ )  
 $x = a^q$   
 $w = v^i w x^j = a^i b^p c^p \notin L$

Caso 2.  $v = a^p$  ( $vwx = a^r$ )  
 $x = a^q$   
 $w = v^i w x^j = a^i b^p c^p \notin L$

Caso 3.  $v = a^p$  ( $vwx = b^r$ )  
 $x = b^q$   
 $w = v^i w x^j = b^i c^p \notin L$

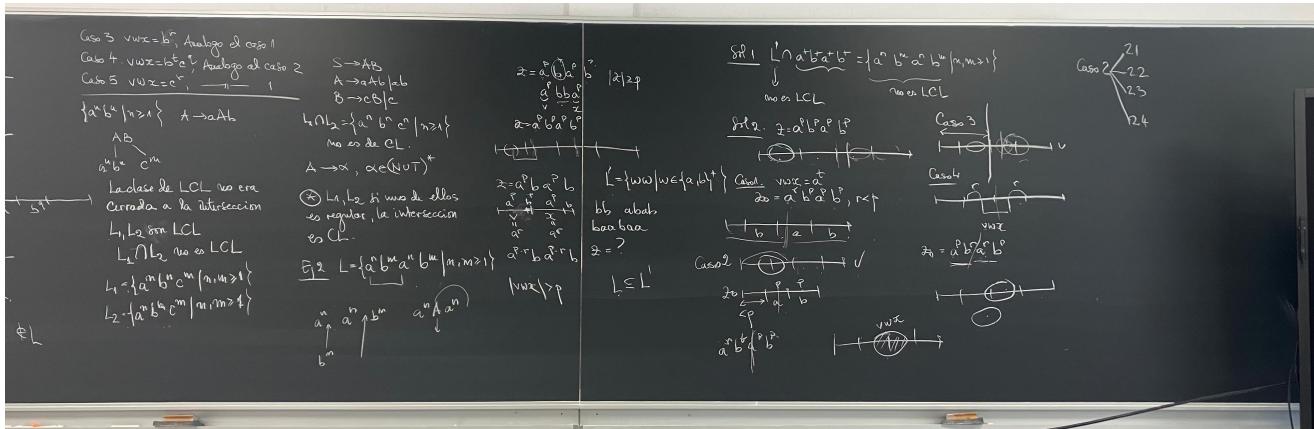
Caso 4.  $v = b^p$  ( $vwx = b^r$ )  
 $x = b^q$   
 $w = v^i w x^j = a^p b^i c^p \notin L$

Caso 5.  $v = c^p$  ( $vwx = c^r$ )  
 $x = c^q$   
 $w = v^i w x^j = a^p b^p c^i \notin L$

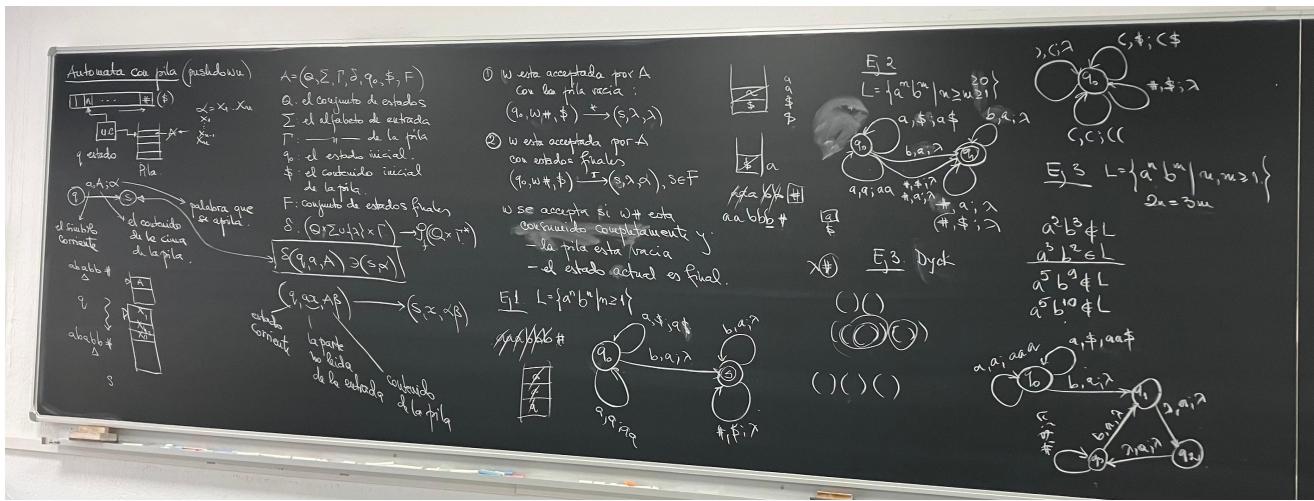
AB  
 $a^m b^n$   
 $c^m$

La clase de LCL no era cerrada a la intersección.  
 $L_1, L_2$  son LCL  
 $L_1 \cap L_2$  no es LCL

$L_1 = \{a^m b^n c^m \mid m, n \geq 1\}$   
 $L_2 = \{a^n b^m c^m \mid m, n \geq 1\}$



## 11.11 Autómata de Pila - AP



*Se define como:*

$$AP = (\Sigma, Q, \Gamma, \delta, q_0, ], F)$$

- $\Sigma$  : Alfabeto de las palabras.
  - $Q$  : Conjunto de todos estados.
  - $\Gamma$  : Alfabeto de la pila.
  - $\delta$  : Función de transición -  $(q_i, a, A) \rightarrow (q_j, BB)$ .
    - $q_i$  : Estado actual.
    - $a$  : Símbolo de entrada.
    - $A$  : Elemento que se quita de la pila.
    - $q_j$  : Estado destino.
    - $BB$  : Elementos para poner en la pila.
  - $q_0$  : Estado inicial del autómata.
  - $]$  : Símbolo inicial de la pila.
  - $F$  : Estados finales del autómata.

Note

En mis apuntes los "Centinelas" - caracteres que delimitan un valor - son:

- Centinela de Entrada : #→]

- Centinela de Pila :  $\$ \rightarrow \}$

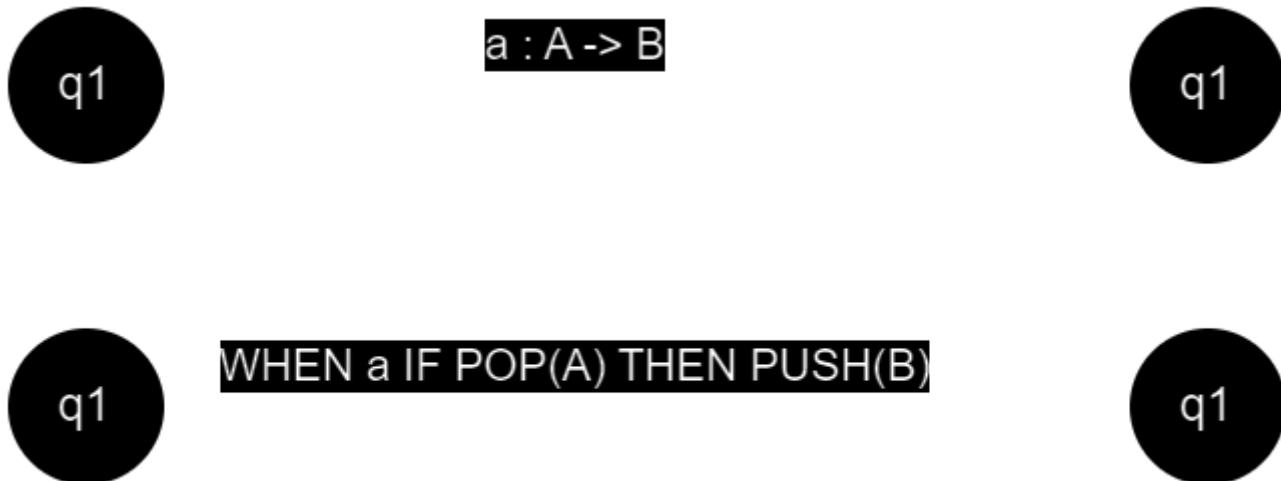
### 11.11.1 Representación Gráfica - Grafos

Se hace igual a que en grafos de expresiones regulares, excepto:

- Transiciones ( $\delta$ ) : Flechas. Con etiquetas  $a; A; BB$ 
  - $a$  : Símbolo de entrada.
  - $A$  : Elemento que se quita de la pila.
  - $BB$  : Elementos para poner en la pila.



Unas formas más intuitivas - para mí - de entender las transiciones en los grafos son:



### 11.11.2 Inicialización

Antes de empezar a consumir entradas, el AP inicializa la:

- Entrada, añadiendo al final de la palabra el **Centinela de Entrada**.
- Pila, añadiendo - pusheando - el **Centinela de Pila**.

### 11.11.3 Consumición

Un AP funciona de la siguiente forma:

1. Recibe una palabra para que sea consumida y consume la primera entrada.
2. Busca transiciones en el estado actual que esperan esa entrada.
3. De las transiciones encontradas, prueba a consumir su símbolo definido. Si puede consumir:
  1. Se realiza la transición.
  2. Se elimina de la pila el símbolo definido.
  3. Se añade a la pila el símbolo definido.
  4. Si puede avanzar la entrada, pasa a 2.
  5. Pasa a **Aceptación**.

## 4. Pasa a Aceptación.



O en forma de código:

```

stack;
current_state;

FUNCTION consume (entry) // entry es una letra
    transitions = current_state.transitions_with(entry)
    FOR transition IN transitions DO
        IF transition.stack_top_value == stack.peek() THEN // peek devuelve
            top
                stack.pop() // pop quita
            top
                stack.push(transition.stack_new_top_values) // push añade
            a top
                current_state = transition.next_state
        END
    END
END

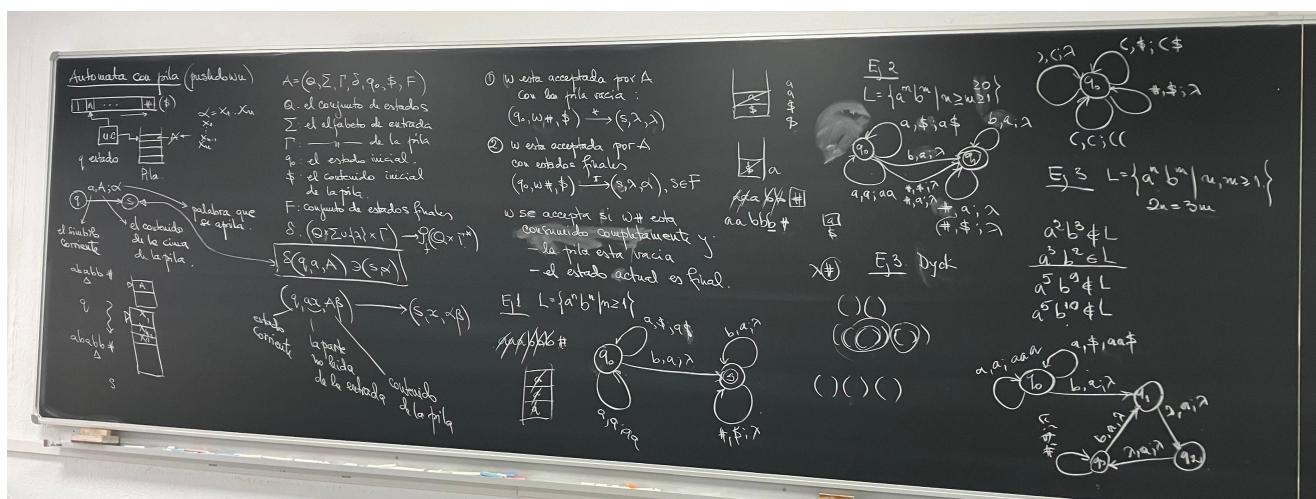
```

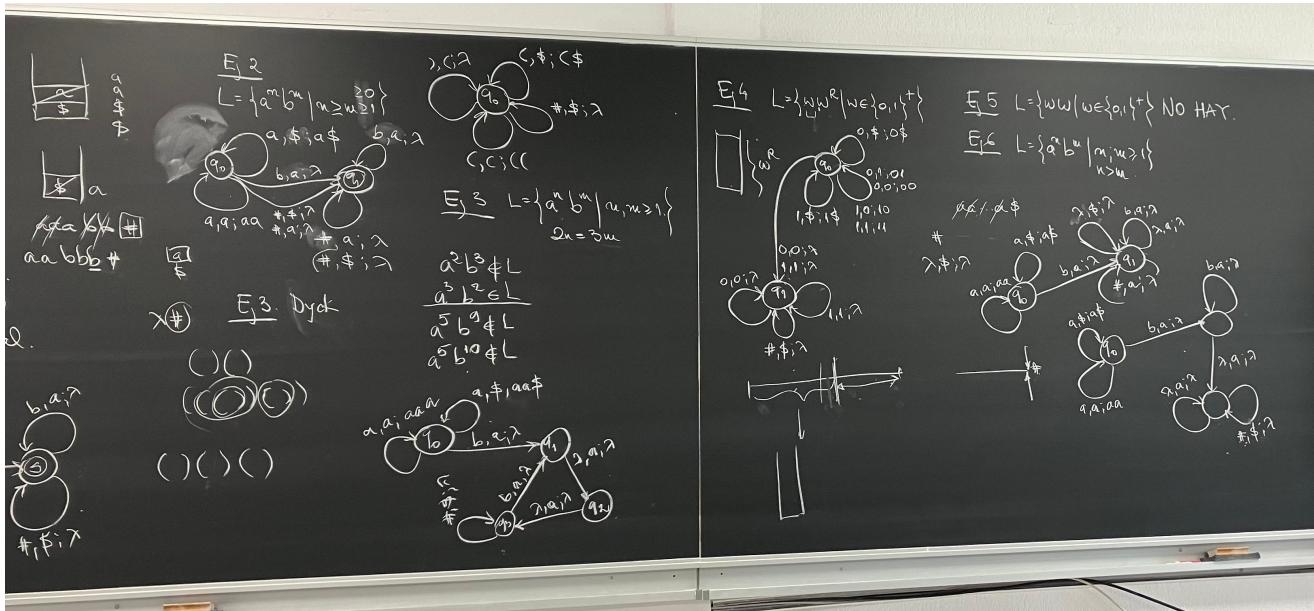
## 11.11.4 Aceptación

Un **Pushdown Automaton** acepta palabras -  $w$  - cuando:

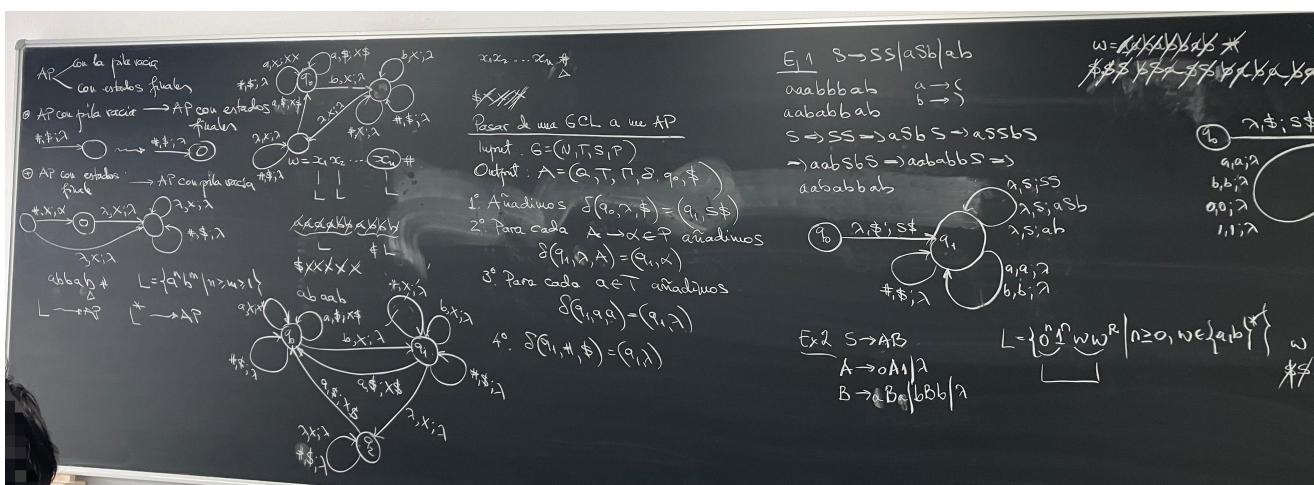
- Al consumir totalmente  $w$ , tiene la **Pila Vacía**.
- Al consumir totalmente  $w$ , está en un **Estado Final**.

## 11.12 Ejemplos: AP





## 11.13 Transformación: GCL a AP



Pasos:

- **Añadir Estado Inicial** de inicialización.
- **Añadir Estado** para consumición.

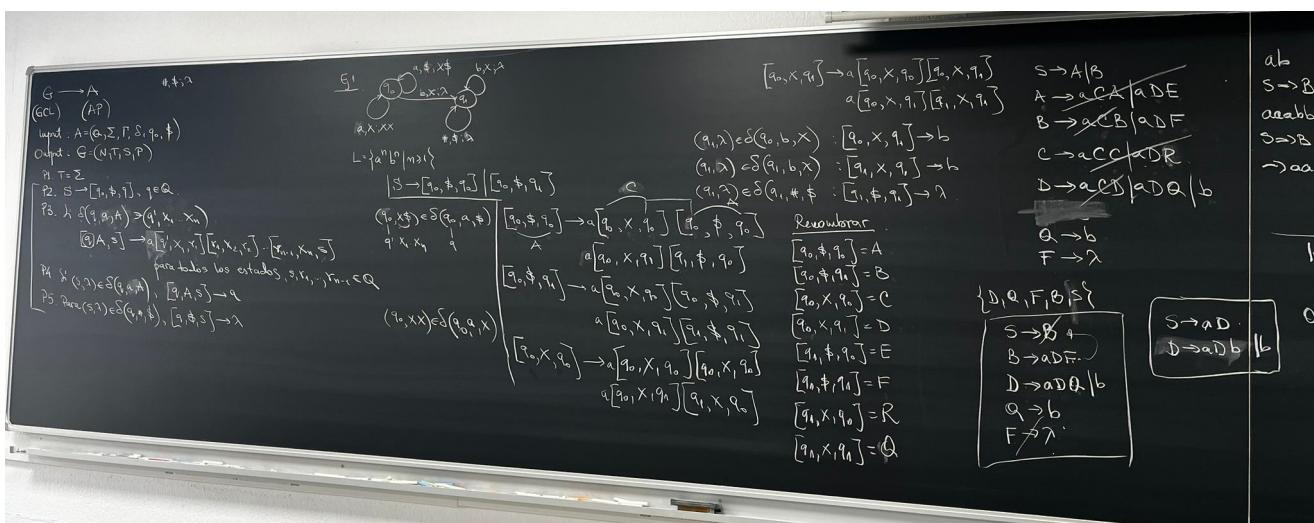
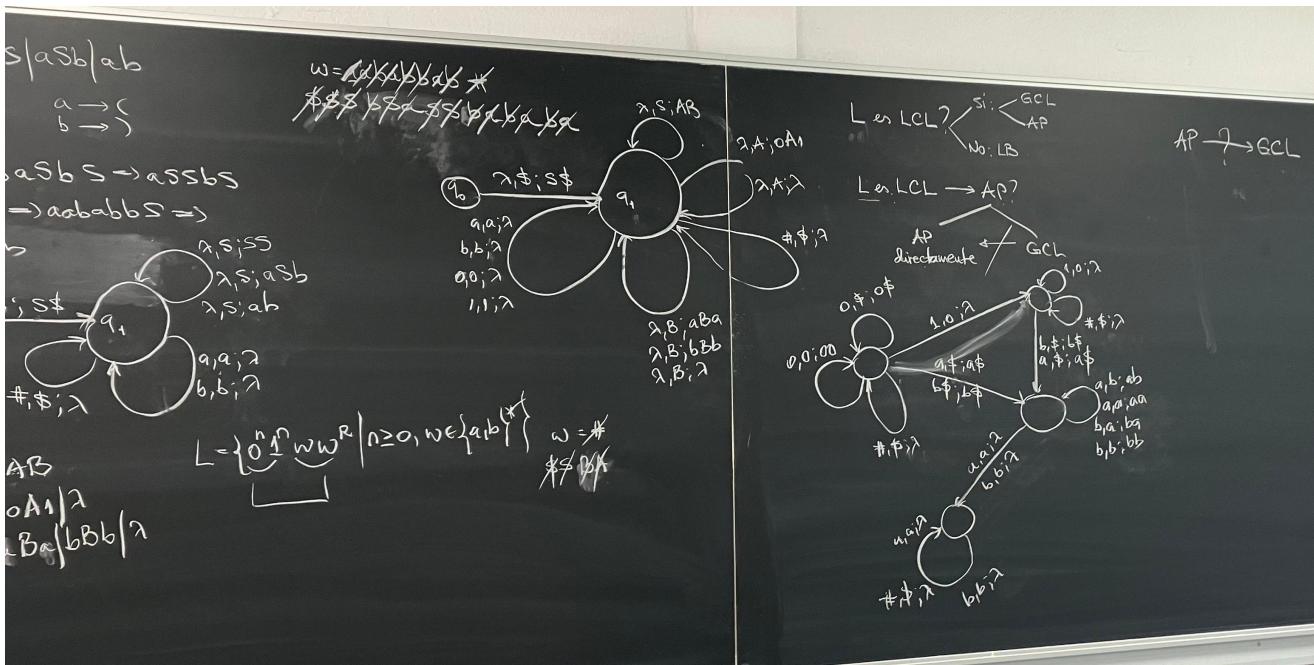
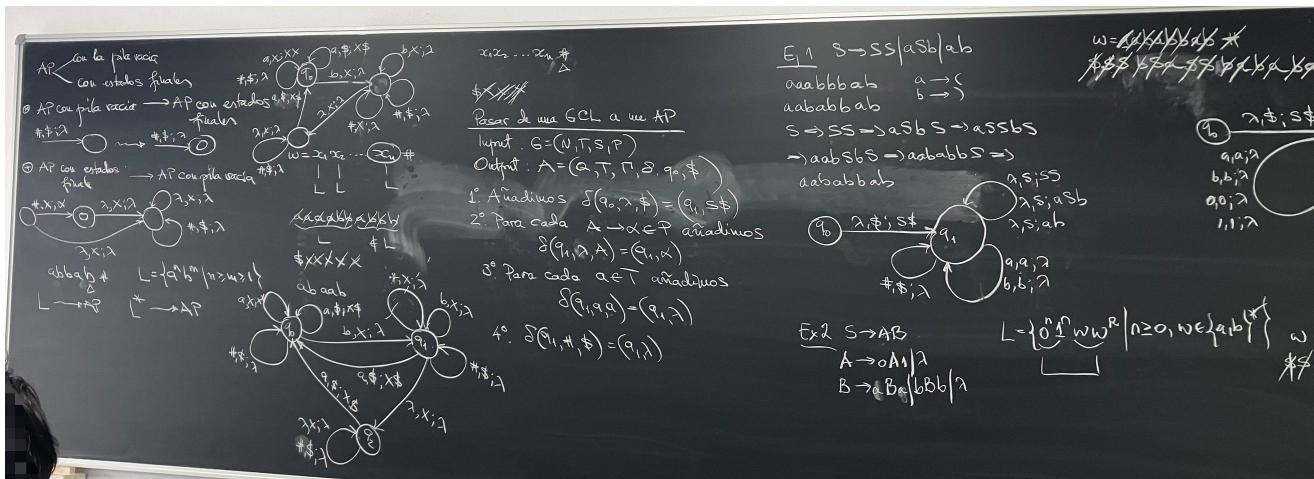
1. **Pushear el Centinela de Pila** con transición  $(\lambda, \}, S\}$ .
2.  $\forall A \rightarrow X : X$  es un implicado cualquiera, **creamos transición**  $(\lambda, A, X)$ .
3.  $\forall a \in \Sigma$ , **creamos transición**  $(a, a, \lambda)$ .
4. **Creamos transición para Vaciado de Pila**  $([], \}, \lambda)$ .

### Note

Solo con dos estados ya nos sirve, ya que definiremos todas entradas como lambda y solo se aceptará la palabra si al consumirla la pila está vacía. De esta forma construimos un AP lo suficientemente general como para aceptar cualquier gramática.

El inconveniente es tener varias transiciones lambda y generar un AP No Determinista.

## 11.14 Ejemplos: GLC a AP



## 12. Sintaxis

Las características sintácticas de un LP se pueden especificar mediante una GLC.

Esa especificación puede ser en formato BNF o BNFA.

 Note

- **LP:** Lenguaje de Programación.
- **EBNF:** BNFA.
- **AS:** Analizador Sintáctico.

## 12.1 Gramáticas en BNF y BNFA

### 12.1.1 BNF

*Notación:*

- **Símbolos No Terminales (A):** <no\_term>
- **Símbolos Terminales (a):** term
- **Producción (→):** ::=

### 12.1.1 BNFA

*Facilita el entendimiento de gramáticas BNF, sobre todo la optionalidad y repetición de elementos.*

*Notación (BNF + Nuevas):*

- **Subpalabra Repetida:** {subpalabra}
  - | Subpalabra aparece **0 o más** veces.
- **Subpalabra Opcional:** [subpalabra]
  - | Subpalabra aparece **0 o 1** vez.
- **Subpalabra Variante:** (subpalabra\_1, subpalabra\_2)
  - | Aparece **subpalabra\_1 o subpalabra\_2, etc.**

### 12.1.2 Ejemplo: BNF

*Gramática en notación BNF que define la sintaxis de las declaraciones de variables en C.*

```
<declaraciones> ::= <una_declaracion> <declaraciones> | e
<una_declaracion> ::= <tipo> <lista_id_var> ;
<lista_id_var> ::= id <mas_id>
<mas_id> ::= , id <mas_id> | e
<tipo> ::= int | float
```

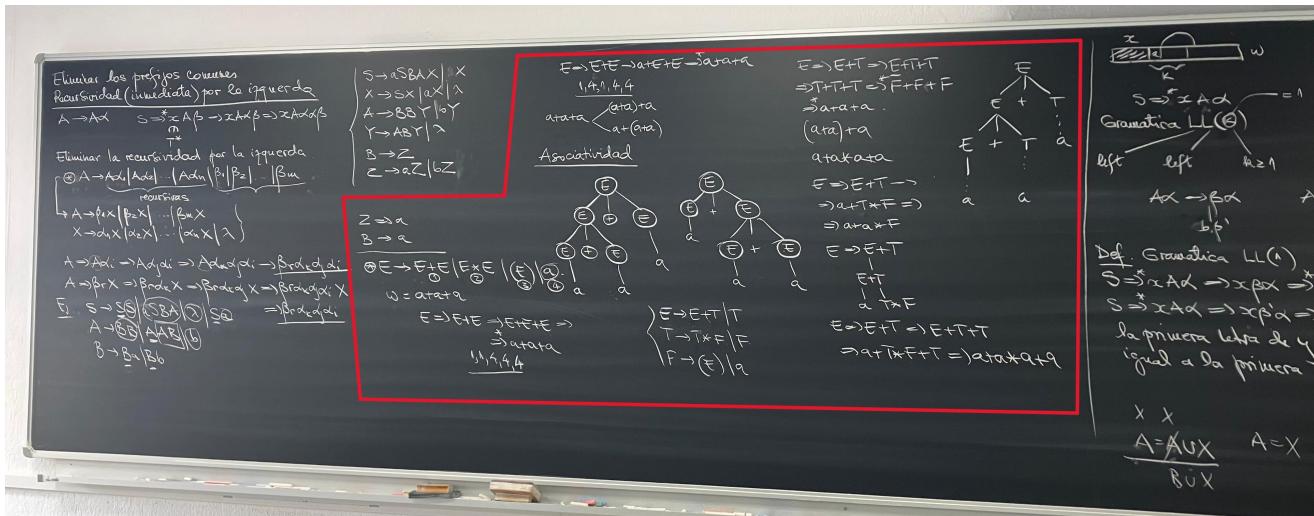
### 12.1.3 Ejemplo: BNFA

*Gramática en notación BNF que define la sintaxis de las declaraciones de variables en C.*

```

<declaraciones> ::= {<una_declaracion> ;}
<una_declaracion> ::= <tipo> id {, id}
<tipo> ::= int | float
  
```

## 12.2 Expresiones Aritméticas



Pg. 15 - 19 : Tema 6

Partimos de una gramática en BNF:

```

<expr> ::= <expr> + <expr> | <expr> * <expr> | (<expr>) | a
  
```

Su equivalente:

- $E \rightarrow E + E | E * E | (E) | a$

Tenemos que resolver el problema de la **Precedencia entre Operadores**, para ello:

- $\forall p, A_p \rightarrow O_p$

Es decir, para cada nivel de precedencia, creamos un **Símbolo No Terminal** para sus operadores.



**Menos Precedencia:** Más cerca al **Símbolo Inicial**.

**Más Precedencia:** Más cerca a los **Operandos**.

Quedamos con la siguiente gramática en BNF:

```

<expr> ::= <expr> + <expr> | <term>
<term> ::= <term> * <func> | <func>
  
```

```
<func> ::= (<expr>) | a
```

 Note

**Paréntesis:** Se trata como un Operando.

**Función** Se trata también como un Operando.

**Unarios (-):** Tienen mayor precedencia.

La gramática es ambigua por lo tanto tenemos que guitarle la ambigüedad, quedando:

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <func> | <func>
<func> ::= (<expr>) | a
```

La gramática en BNF es equivalente a:

- $E \rightarrow E + T | T$
- $T \rightarrow T * F | F$
- $F \rightarrow (E) | a$

## 12.3 Analizadores Sintácticos

Pag. 24 - : Tema 6

Tiene como fin reconstruir y comprobar si los **tokens** proporcionados por el analizado léxico pueden ser generados por la gramática sintáctica que define el lenguaje.

### 12.3.1 Símbolos por Adelantado

Usados para resolver la indeterminación que se presente en el proceso de reconstrucción del árbol de derivación.

 Note

Es el numero de símbolos de la entrada que leemos a la vez.

### 12.3.2 Reconocimiento Descendente - *LL*

Forma de los **Analizadores Sintácticos** de reconstruir el árbol **desde la raíz a las hojas**.

 Note

**LL(K)** : Left Reading, Left Derivation, con **K Símbolos por Adelantado**.

### *Ejemplo: Pag 26 : Tema 6*

w= id \* cte + id

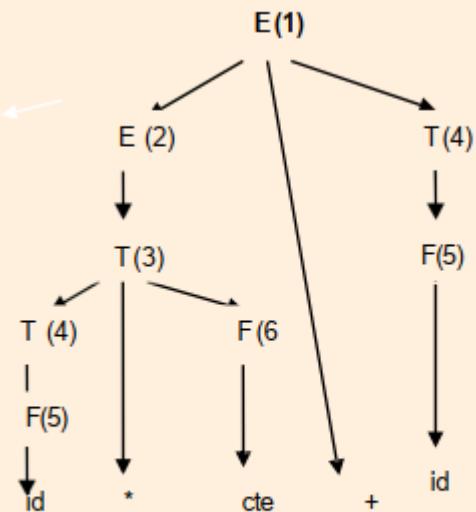
## Ejemplo sencillo

- 1)  $E \rightarrow E + T$  |
  - 2)  $T$
  - 3)  $T \rightarrow T^* F$  |
  - 4)  $F$
  - 5)  $F \rightarrow id$  |
  - 6)  $cte$  |
  - 7)  $(E)$

## Reconstrucción descendente

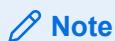
### Derivación izquierda

Producciones: 1-2-3-4-5-6-4-5



### 12.3.2.1 Definición LL(1)

$\vdash T$ $F + F$ $E \vdash E + T$ $E \vdash E + T \quad a$ $a \quad a$ $\Rightarrow E + T + T$ $F + T = aata + a + a$	<p style="text-align: center;"><math>x</math></p>  <p><math>S \xrightarrow{*} x A x \xrightarrow{*} 1</math></p> <p>Gramática LL(0)</p> <p>left left <math>\alpha \geq 1</math></p> <p><math>A\alpha \rightarrow \beta\alpha \quad \alpha \rightarrow \beta</math>  <math>b\beta'</math></p> <p>Def. Gramática LL(1)</p> <p><math>S \xrightarrow{*} x A x \rightarrow x \beta x \xrightarrow{*} x y \in x</math></p> <p><math>S \xrightarrow{*} x A x \rightarrow x \beta' x \Rightarrow x z c \in x</math></p> <p>la primera letra de y es igual a la primera letra de z</p> <p><math>\beta = \beta'</math></p> <p><math>X \quad X</math>  <math>A = \cancel{AUX} \quad A = X</math>  <math>\underline{\quad B \cup X \quad}</math></p>
	<p>Como podemos comprobar que una GLC es LL(1)?</p> <p>Def. una función <math>im</math></p> <p><math>im(x) = \{ \alpha T \mid \alpha \xrightarrow{*} \alpha \beta \}</math></p> <p>G: (a) <math>im(\alpha) = \emptyset</math>  (b) <math>im(\alpha) = \{ \alpha \}</math>  (c) <math>im(\alpha) = \bigcup_{\alpha \rightarrow x \beta} im(x)</math>  (d) <math>im(\alpha \beta) = im(\alpha), \alpha \beta \text{ no es ambiguo}</math>  <math>im(\alpha) \cup im(\beta), \alpha \beta \text{ es ambiguo}</math></p> <p>Ex: <math>S \xrightarrow{*} SAB \mid \lambda \mid aB</math>  <math>A \xrightarrow{*} \lambda \mid Ab \mid AA</math>  <math>B \xrightarrow{*} b \mid ABB</math>  <math>B \xrightarrow{*} ABb \Rightarrow \{b\}</math></p> <p><math>im(A) = im(a) = \{a\}</math>  <math>im(A) =</math></p> <p><math>im(A) = im(a) \cup im(ab) \cup im(ba)</math>  <math>= \emptyset \cup im(a) \cup im(b) \cup im(ab)</math></p> <p><math>im(A) = im(a) \cup \{b\} = \{b\}</math></p> <p><math>im(SAb) = im(s) \cup im(ab) =</math>  <math>= im(s) \cup im(A) \cup im(b)</math>  <math>= im(s) \cup \{b\} =</math>  <math>= im(s) \cup im(ab) \cup \{b\}</math>  <math>= im(s) \cup im(A) \cup \{a, b\}</math>  <math>\{b\} \cup \{b\} \cup im(ABB) \cup \{b\}</math>  <math>\{ab\} \cup im(A) \cup im(bb) =</math>  <math>= \{a, b\} \cup \{b\} \cup im(b) = \{a, b\}</math></p>



En mis palabras: "Una **GCL** está en **LL(1)** si puedes elegir la siguiente producción con solo leer 1 entrada".

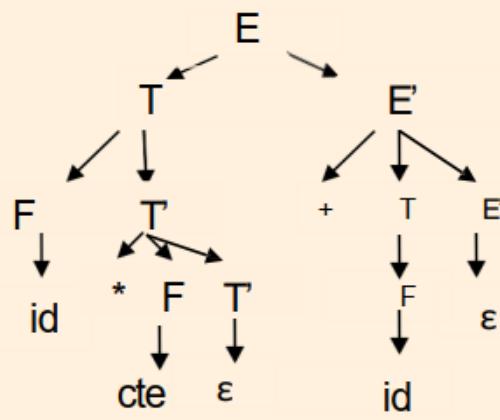
Sobre esta idea podemos asegurar que **si GCL es No Determinista, no es LL(1)**. Pero no podemos asegurar que un GCL Determinista sea siempre LL(1).

Ejemplo LL(1): Pag 27 : Tema 6

$w = \text{id} * \text{cte} + \text{id}$

Gramática LL(1)

$$\begin{aligned} E &\rightarrow TE' \\ E &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow \text{cte} \mid \text{id} \mid (E) \end{aligned}$$



Los pasos para saber si una **Sintaxis** definida por una **GCL** - o por un **Lenguaje** definido por una **GCL** - está en **LL(1)** se explican en el apartado [12.5 Sintaxis en LL\(1\)?](#)

## 12.4 Calculo de Símbolos Directores

### 12.4.1 ¿Qué realmente Representan?

### 12.4.2 Palabras Anulables

Pueden o no tomar un valor en una producción.



Será **Anulable** siempre y cuando no exista ninguna **Subpalabra** en  $\alpha$  que lleve a un **Símbolo Terminal**  $a$ .

Ejemplo:

- $A \rightarrow a|Bb$
- $B \rightarrow aC|\lambda$
- $C \rightarrow B|BA$



Es la palabra  $a$  de la producción  $A \rightarrow a$  anulable?  
Obviamente no, ya que  $a$  ya es en sí **Símbolo Terminal**.

Es la palabra  $Bb$  de la producción  $A \rightarrow Bb$  anulable?  
No, ya que contiene un terminal -  $b$ .

**Es la palabra  $aC$  de la producción  $B \rightarrow aC$  anulable?**

*No, ya que contiene un terminal - a*

**Es la palabra  $\lambda$  de la producción  $B \rightarrow \lambda$  anulable?**

Obviamente sí, ya que  $\lambda$  es en sí la definición de nulo.

**Es la palabra  $B$  de la producción  $C \rightarrow B$  anulable?**

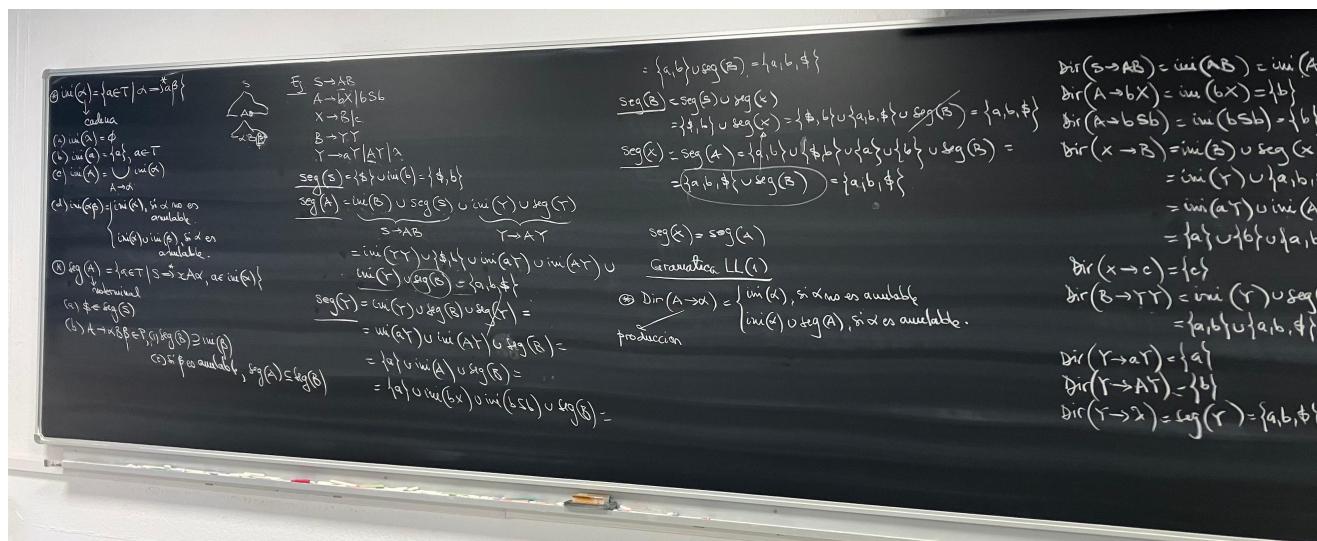
Sí, ya que  $B$  podría ser nulo.

**Es la palabra  $BA$  de la producción  $C \rightarrow BA$  anulable?**

*No, ya que A no es anulable.*

$\begin{aligned} \text{Dir}(S \rightarrow AB) &= \text{ini}(AB) = \text{ini}(A) = \{a\} \\ \text{Dir}(A \rightarrow bX) &= \text{ini}(bX) = \{b\} \\ \text{Dir}(A \rightarrow b SB) &= \text{ini}(b SB) = \{b\} \end{aligned}$	$\begin{aligned} \text{Dir}(x \rightarrow B) &= \text{ini}(B) \cup \text{seg}(x) \\ &= \text{ini}(r) \cup \{a, b, \$\} \\ &= \text{ini}(aT) \cup \text{ini}(AT) \cup \{a, b, \$\} \\ &= \{a\} \cup \{b\} \cup \{a, b, \$\} = \{a, b, \$\} \end{aligned}$	<p><u>Es LL(1) si y solo si</u></p> $\text{Dir}(A \rightarrow x) \cap \text{Dir}(A \rightarrow \beta) = \emptyset$ <p>para cada par de prod.</p> $A \rightarrow x   \beta$
$\text{Dir}(x \rightarrow c) = \{c\}$	$\text{Dir}(B \rightarrow YY) = \text{ini}(Y) \cup \text{seg}(B)$	<p><u>Ej. 2. S → AA</u></p> $\begin{aligned} &A \rightarrow Ab \quad A \in \Sigma \\ &B \rightarrow aA \mid \$ \end{aligned}$
$\text{Dir}(Y \rightarrow aT) = \{a\}$	$= \{a, b\} \cup \{a, b, \$\} = \{a, b, \$\}$	<p>1. Es G una gramática LL(1)? NO</p>
$\text{Dir}(Y \rightarrow AT) = \{b\}$	$\text{Dir}(Y \rightarrow \lambda) = \text{seg}(Y) = \{a, b, \$\}$	<p>2. Eliminar las causas que impiden que G sea LL(1)</p> <p>Eliminar prefijo comunes</p>
$\boxed{\begin{array}{l} S \rightarrow AA \\ A \rightarrow AX[\lambda] \\ X \rightarrow b[\lambda] \\ B \rightarrow aY \\ T \rightarrow AT \end{array}}$	$\begin{array}{l} A \rightarrow Z \\ Z \rightarrow ZX[\lambda] \\ B \rightarrow aY \\ T \rightarrow AT \end{array}$	<p>3. Es la nueva gramática LL(1)?</p> <p>X2 //</p>

$$\begin{aligned}
 \text{seg}(s) &= \{a, b\} \cup \text{seg}(r) = \{a, b, c\} \\
 \text{seg}(A) &= \text{ini}(A) \cup \text{seg}(s) = \text{ini}(x) \cup \text{seg}(s) = \text{ini}(x) \cup \text{seg}(z) \\
 &= \{a, b\} \cup \text{ini}(x) \cup \text{seg}(z) \cup \text{seg}(r) \\
 &= \{a, b, c\} \cup \text{seg}(r) = \{a, b, c\} \cup \{a, b, c\} \cup \text{seg}(x) = \{a, b, c\} \\
 \text{seg}(z) &= \text{seg}(x) = \{a, b, c\} \\
 \text{seg}(x) &= \text{ini}(z) \cup \text{seg}(z) = \text{ini}(x) \cup \text{seg}(z) = \\
 &= \{a, b\} \cup \text{seg}(z) = \{a, b\} \cup \text{seg}(A) = \{a, b, c\} \\
 \text{seg}(r) &= \text{seg}(x) = \{a, b, c\} \\
 \text{seg}(z) &= \text{seg}(A) = \{a, b, c\} \\
 \text{dir}(x \rightarrow xz) &= \text{ini}(xz) = \text{ini}(x) = \{a, b\} \\
 \text{dir}(z \rightarrow z) &= \text{seg}(z) = \{a, b, c\} \quad \boxed{\text{f} \neq \emptyset}
 \end{aligned}$$



#### 12.4.2 Calcular: Iniciales

*Representación: Inic( $\alpha$ )*

### *En los apuntes: $I(\alpha)$*

*a : Subpalabra.*

$$I(\alpha) = \begin{cases} a & \text{si el primer Símbolo de } \alpha \text{ es Terminal} \\ I(\beta) \cup I(\alpha - \beta) & \text{en el caso contrario} \end{cases}$$

 Note

En código:

```

FUNCTION init(alpha)
    x = first_symbol_of(alpha)
    IF x IS TERMINAL THEN
        x
    ELSE
        init(x) UNION init(alpha - x)
    END
END

```

### 12.4.3 Calcular: Segidores

Representación:  $\text{Seg}(X)$

En los apuntes:  $S(X)$

|  **$X : \text{No Terminal.}$**

$$S(X) = \forall \beta \text{ que procede } X \begin{cases} I(\beta) & \text{si } \beta \text{ es No Anulable} \\ I(\beta) \cup S(A) & \text{en el caso contrario} \end{cases}$$

 Note

En código:

```

FUNCTION seg(X)
    FOR beta EN LA DERECHA DE X DO
        IF beta IS NOT NULLABLE THEN
            inic(beta)
        ELSE
            inic(beta) UNION seg(A)
        END
    END
END

```

### 12.4.4 Calcular: Directores

Representación:  $\text{Dir}(A \rightarrow \alpha)$

En los apuntes:  $D(A \rightarrow \alpha)$

|  **$A : \text{No Terminal.}$**

|  **$\alpha : \text{Subpalabra.}$**

## Class Notes

$$D(A \rightarrow \alpha) = \begin{cases} I(\alpha) & \text{si } \alpha \text{ es No Anulable} \\ I(\alpha) \cup S(A) & \text{en el caso contrario} \end{cases}$$



*En código:*

```

FUNCTION dir(A, alpha)
    IF alpha IS NOT NULLABLE THEN
        inic(alpha)
    ELSE
        inic(alpha) UNION seg(A)
    END
END

```

## 12.5 Comprobar si Sintaxis está en LL(1)

### 12.5.1 ¿Qué realmente Significa estar en LL(1)?

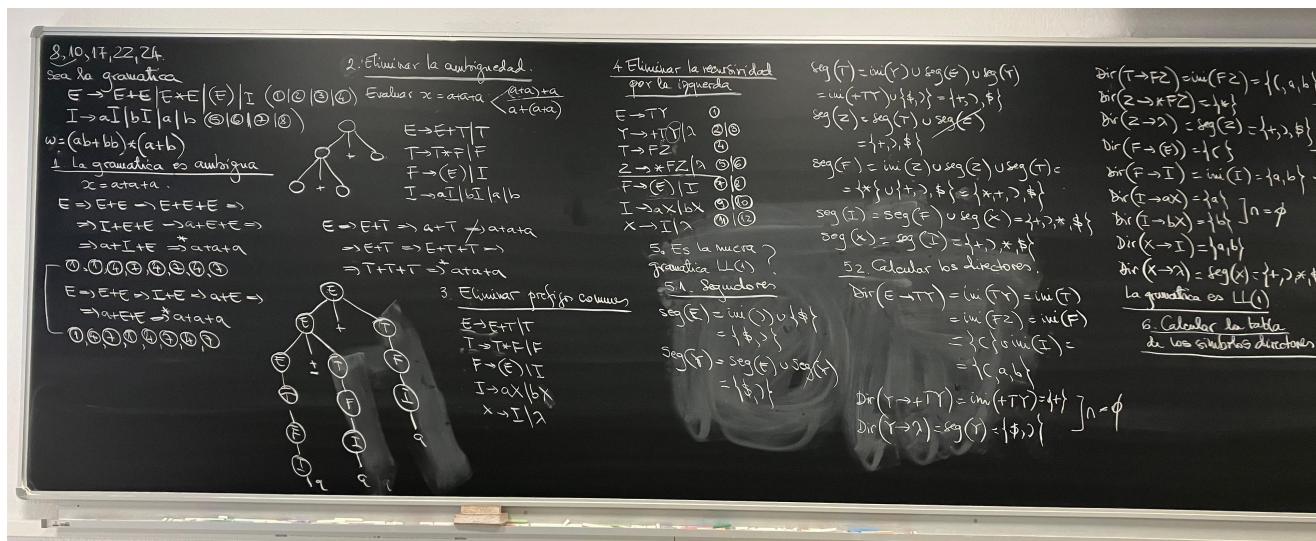
### 12.5.2 ¿Qué tiene que ver los Símbolos Directores con LL(1)?

### 12.5.3 Pasos

## 12.6 Analizadores Descendentes Predictivos Recursivos

## 12.7 Actividad - Analizadores Léxico-Sintácticos Automáticos

### [POR HACER]



## [DUDAS]

Eliminar los prefijos comunes

Recursividad (inmediata) por la izquierda

$$A \rightarrow Ad \quad S \Rightarrow^* x \underset{\text{M}}{\underset{T^*}{\beta}} A \beta \rightarrow x A x \beta \Rightarrow x A d d \beta$$

Eliminar la recursividad por la izquierda

$\oplus A \rightarrow Ad_1 | Ad_2 | \dots | Ad_n | \beta_1 | \beta_2 | \dots | \beta_m$

recursivas

$\rightarrow A \rightarrow \beta_1 X | \beta_2 X | \dots | \beta_m X$

$X \rightarrow \alpha_1 X | \alpha_2 X | \dots | \alpha_n X | \lambda$

$A \rightarrow Ad_i \rightarrow Ad_i \alpha_i \rightarrow Ad_i \alpha_i \beta_i \rightarrow \underline{\beta_i ad_i \alpha_i \beta_i}$

$A \Rightarrow \beta_i X \Rightarrow \underline{\beta_i ad_i} X \Rightarrow \underline{\beta_i ad_i \alpha_i} X \Rightarrow \underline{\beta_i ad_i \alpha_i} X$

Ej.  $S \rightarrow SS | SBA | \lambda | SQ \Rightarrow \underline{\beta_i ad_i \alpha_i}$

$A \rightarrow BB | AAB | \lambda$

$B \rightarrow Ba | Bb$

$\left. \begin{array}{l} S \rightarrow aSBAX \\ X \rightarrow sx | ax | \lambda \\ A \rightarrow BBY | bY \\ Y \rightarrow ABY | \lambda \\ B \rightarrow Z \\ C \rightarrow aZ | bZ \end{array} \right\} X$

$Z \Rightarrow a$

$B \rightarrow a$

---

$\oplus E \rightarrow E+E | E \times E$

$\omega = aaaa$

$E \Rightarrow E+E$

151,

Como S ya nos lleva a aS y de S podríamos seguir por X, por transitividad se podría eliminar la producción  $X \rightarrow aX$  ?