ANÁLISIS SINTÁCTICO

Analizador Léxico

↓ tokens

Analizador Sintáctico

↓ Árbol sintáctico

Comprueba si la secuencia de tokens que va recibiendo tiene una sintaxis correcta

¿Cómo?

Construyendo el árbol sintáctico, usando las reglas de la **Gramática de Contexto Libre**

IF x THEN a := 0

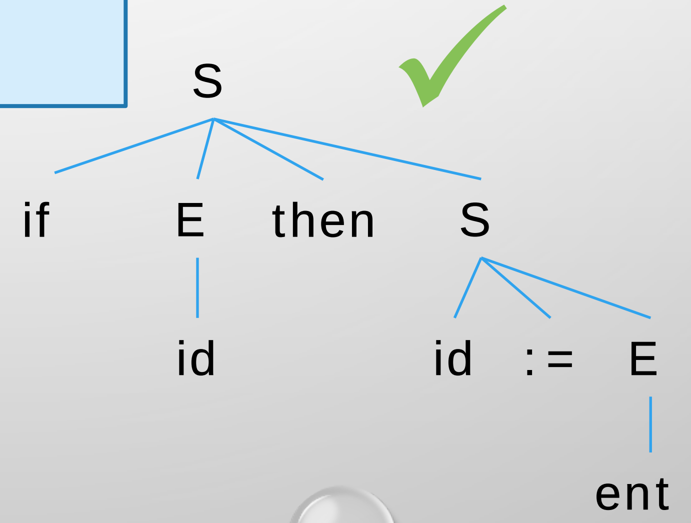
Analizador Léxico

tokens

<if, > <id, 12> <then, > <id, 3> <asig, > <ent, 0>

Analizador Sintáctico

Árbol sintáctico



Comprueba si la secuencia de tokens que va recibiendo tiene una sintaxis correcta

¿Cómo?

Construyendo el árbol sintáctico, usando las reglas de la **Gramática de Contexto Libre**

GCL

S → if E then S
S → if E then S else S
S → id := E
E → id
E → ent

IF x ELSE a := 0

Analizador Léxico

tokens

<if, > <id, 12> <else, > <id, 3> <asig, > <ent, 0>

Analizador Sintáctico

Árbol sintáctico

S

if E then S

id

GCL

S → if E then S
S → if E then S else S
S → id := E
E → id
E → ent

Comprueba si la secuencia de tokens que va recibiendo tiene una sintaxis correcta

¿Cómo?

Construyendo el árbol sintáctico, usando las reglas de la **Gramática de Contexto Libre**

Gramática formal

$G = (N, T, P, S)$

N: conjunto de símbolos no terminales
T: conjunto de símbolos terminales (alfabeto de entrada)
P: conjunto de reglas de producción
S: axioma ($S \in N$)

El alfabeto de entrada (conjunto T de la GCL del A. Sintáctico) son los tokens

Lenguaje Generado por una Gramática

$L(G) = \{ \omega \mid \quad \omega \in T^* \quad S \Rightarrow^+ \omega \}$

Formas Sentenciales

$F(G) = \{ \sigma \mid \quad \sigma \in (N \cup T)^* \quad S \Rightarrow^+ \sigma \}$

Gramática Tipo 2 o Gramática de Contexto Libre (GCL)

Sus reglas son de la forma: $A \rightarrow \alpha \qquad A \in N \qquad \alpha \in (N \cup T)^*$

Clasificación de gramáticas de Chomsky

$GR \subseteq GCL \subseteq G\_Tipo1 \subseteq G\_Tipo0$

A. Léx. A. Sint.

Ejemplo de GCL, G

N = { D, T , L}

T = { integer, real, id }

P =

D → T L
T → integer
T → real
L → id , L
L → id

Axioma = D

Cadenas de entrada correctas

(se puede construir su árbol con G)

ω_1 = real id , id

ω_2 = real id , id , id

ω_3 = integer id

Cadenas de entrada incorrectas

(no se puede construir su árbol con G)

ω_4 = real id ,

ω_5 = real , id , id

ω_6 = integer id ;

Ejemplo de análisis sintáctico de 2 cadenas de entrada

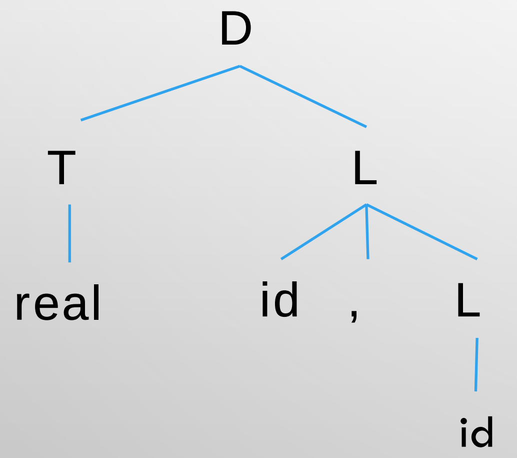
N = { D, T , L}

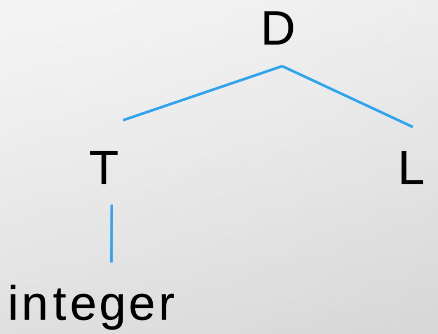
T = { integer, real, id }

P =

$$\boxed{\begin{array}{l} D \rightarrow T\ L \\ T \rightarrow \text{integer} \\ T \rightarrow \text{real} \\ L \rightarrow \text{id}\ ,\ L \\ L \rightarrow \text{id} \end{array}}$$

Axioma = D





ω_1 = real id , id ✓

ω_7 = integer ✗

El Analizador Sintáctico ha de obtener **siempre el mismo árbol para una misma cadena de entrada**, y aplicando la misma secuencia de pasos

→ Ha de ser determinista

→ La GCL no puede ser ambigua

TIPOS DE ANALIZADORES SINTÁCTICOS:

- Descendente o Ascendente

Descendente. Construye el árbol desde la raíz hacia las hojas

Ascendente. Construye el árbol desde las hojas hacia la raíz

- Con retroceso o Sin retroceso

Con retroceso. Si hay varias reglas candidatas para expandir un nodo del árbol, se elige una; si no se consigue terminar el árbol, se retrocede hasta ese nodo y se elige otra regla, y así hasta terminar el árbol o agotar las reglas candidatas → Ineficientes

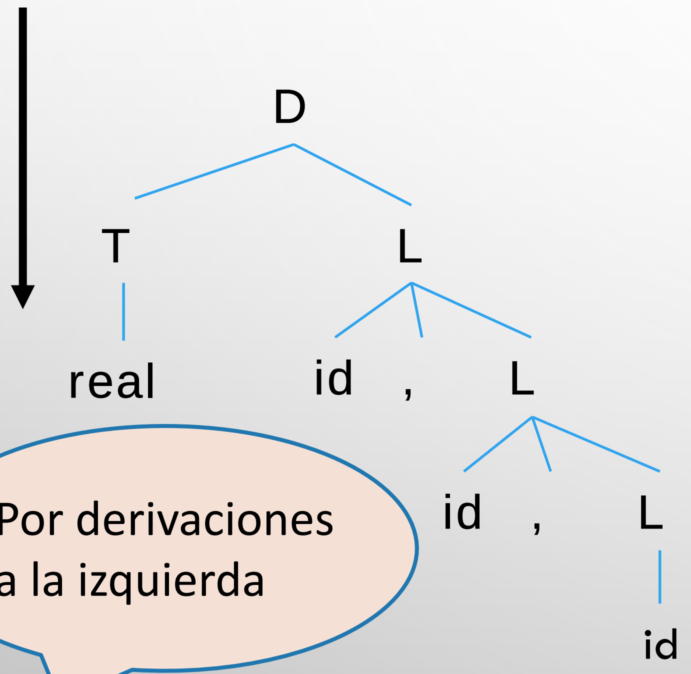
Sin **retroceso**. Utiliza algún criterio para saber con certeza cuál es la siguiente regla a aplicar en cada instante → ¿¿Cómo?? **Gramáticas LL y LR**

Ejemplo de construcción del árbol

1. D → T L
2. T → integer
3. T → real
4. L → id , L
5. L → id

ω = `real id , id , id`

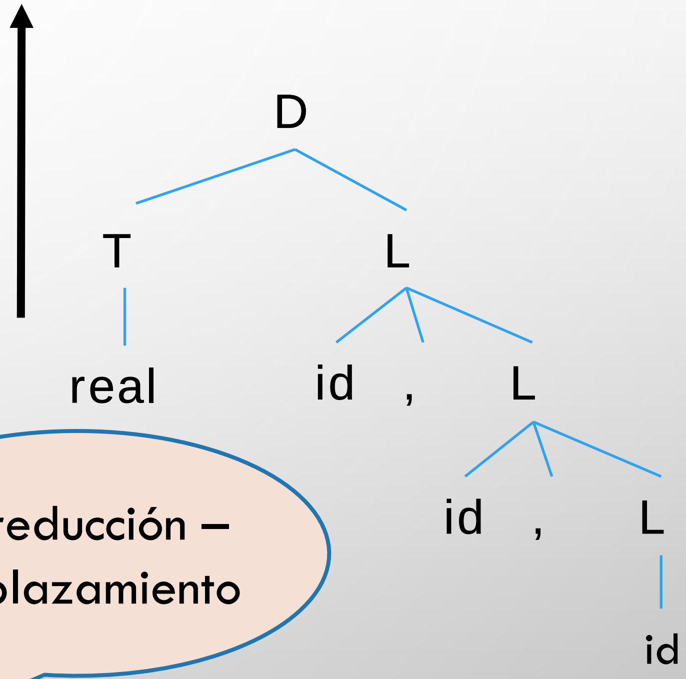
A. St. Descendente



Por derivaciones a la izquierda

Parse: 1 3 4 4 5

A. St. Ascendente



Por reducción – desplazamiento

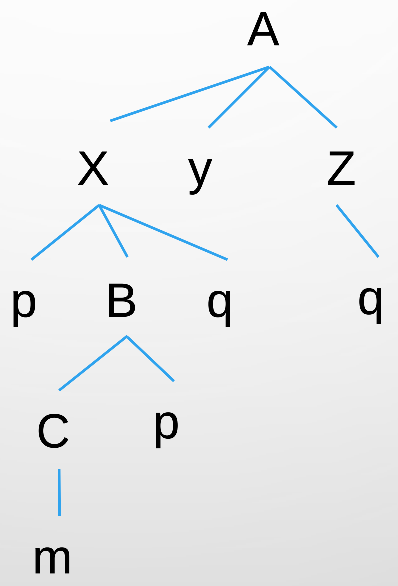
Parse: 3 5 4 4 1

ANÁLISIS SINTÁCTICO DESCENDENTE SIN RETROCESO (GRAMÁTICAS LL)

Descendente

- Construye el árbol desde la raíz (axioma de la GCL) hacia las hojas
- Funciona por derivaciones a la izquierda

1. A → X y Z
2. X → p B q
3. B → C p
4. C → m
5. Z → q



La GCL no puede ser recursiva por la izquierda → bucle infinito!!

parse: 1 2 3 4 5

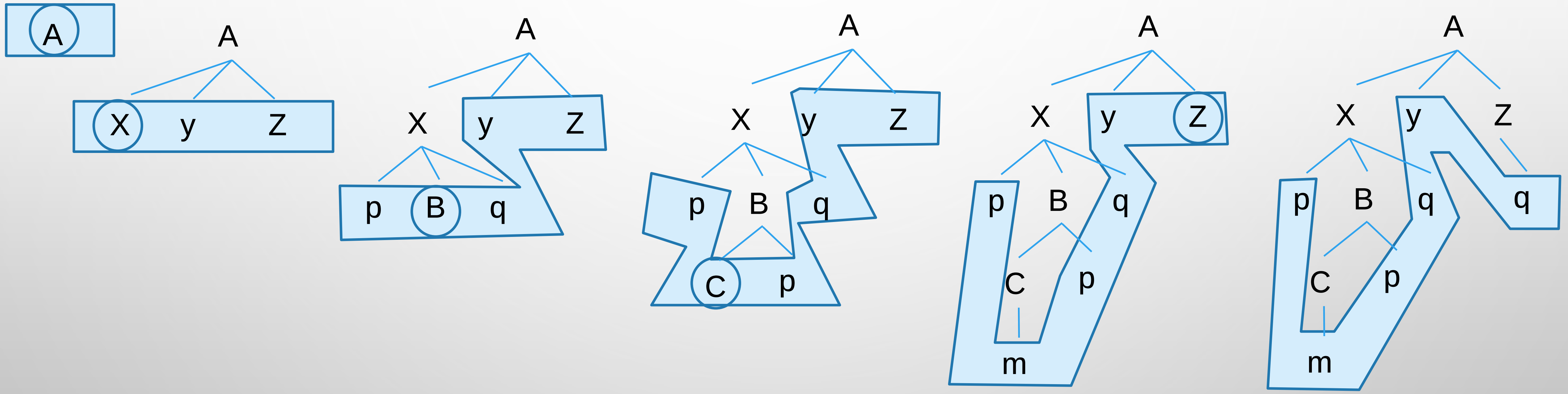
Descendente sin retroceso

- En cada instante solo hay una regla válida. ¿Qué propiedad de la GCL garantiza esto? → **Gramáticas LL**

Derivaciones a la izquierda y Forma Sentencial

1. A → X y Z
2. X → p B q
3. Z → q
4. B → C p
5. C → m

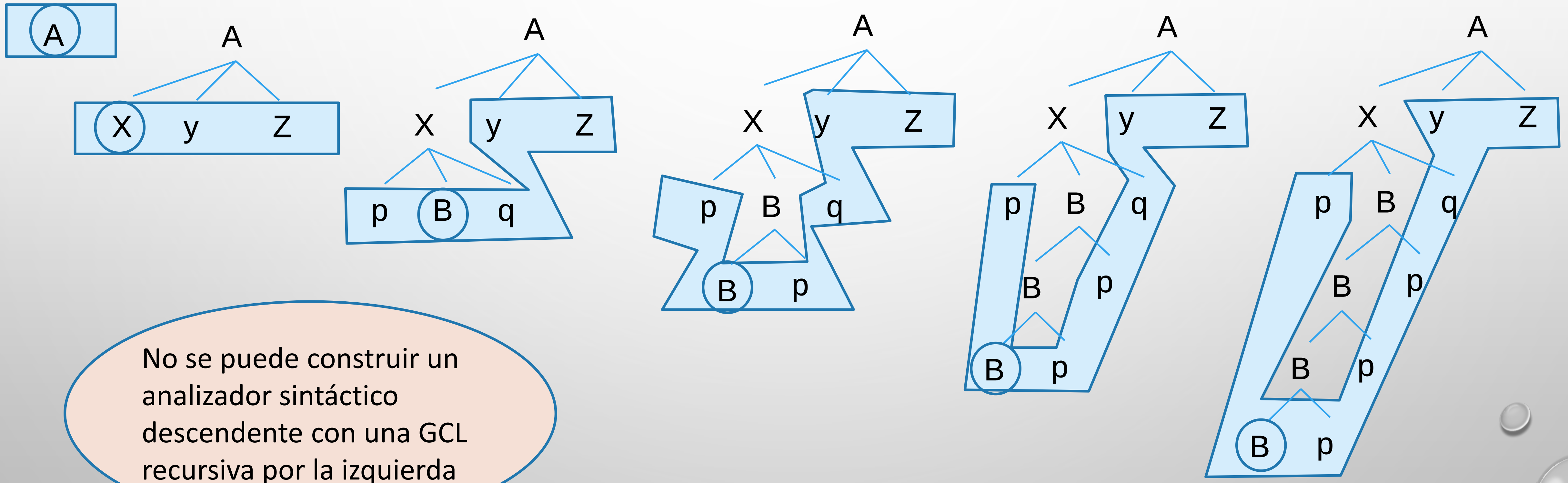
ω = p m p q y q



Problema con las GCL recursivas por la izquierda

1. A → X y Z
2. X → p B q
3. Z → q
4. B → B p
5. B → m

ω = p m p q y q



No se puede construir un analizador sintáctico descendente con una GCL recursiva por la izquierda

¡¡ Bucle infinito !!

¿Cómo **eliminar la recursividad por la izquierda**?

G:

$$\boxed{\begin{array}{l} A \rightarrow A\ \alpha \\ A \rightarrow \beta \end{array}}$$

¿Qué lenguaje genera esta gramática?

¿Cómo **eliminar la recursividad por la izquierda**?

G:

$A \rightarrow A\ \alpha$
$A \rightarrow \beta$

$L(G) = \{\beta, \beta\ \alpha, \beta\ \alpha\ \alpha, \beta\ \alpha\ \alpha\ \alpha, \beta\ \alpha\ \alpha\ \alpha\ \alpha, \ldots\} = \{\beta\ \alpha^*\}$

G':

$A \rightarrow \beta\ A'$
$A' \rightarrow \alpha\ A'$
$A' \rightarrow \lambda$

$L(G') = L(G)$

¿Cómo **eliminar la recursividad por la izquierda**?

G:

$A \rightarrow A\ \alpha$
$A \rightarrow \beta$

$L(G) = \{\beta, \beta\ \alpha\ , \beta\ \alpha\ \alpha\ , \beta\ \alpha\ \alpha\ \alpha\ , \beta\ \alpha\ \alpha\ \alpha\ \alpha, \ldots \} = \{\beta\ \alpha^*\}$

G':

$A \rightarrow \beta\ A'$
$A' \rightarrow \alpha\ A'$
$A' \rightarrow \lambda$

- Ejemplo G_1:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow id$

¿Cómo **eliminar la recursividad por la izquierda**?

G:

$A \rightarrow A\ \alpha$
$A \rightarrow \beta$

$L(G) = \{\beta, \beta\ \alpha, \beta\ \alpha\ \alpha, \beta\ \alpha\ \alpha\ \alpha, \beta\ \alpha\ \alpha\ \alpha\ \alpha, \dots\} = \{\beta\ \alpha^*\}$

G':

$A \rightarrow \beta\ A'$
$A' \rightarrow \alpha\ A'$
$A' \rightarrow \lambda$

- Ejemplo G_1:

$E \rightarrow E\ \boxed{+\ T}$ (α)
$E \rightarrow \boxed{T}$ (β)
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow id$

$E \rightarrow \boxed{T}\ E'$
$E' \rightarrow \boxed{+\ T}\ E' \mid \lambda$
$T \rightarrow T\ \boxed{*\ F}$ (α)
$T \rightarrow \boxed{F}$ (β)
$F \rightarrow id$

G'_1:

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \lambda$
$T \rightarrow \boxed{F}\ T'$
$T' \rightarrow \boxed{*\ F}\ T' \mid \lambda$
$F \rightarrow id$

¿Cómo **eliminar la recursividad por la izquierda**?

G:

$A \rightarrow A\ \alpha$
$A \rightarrow \beta$

G':

$A \rightarrow \beta\ A'$
$A' \rightarrow \alpha\ A'$
$A' \rightarrow \lambda$

$L(G) = \{\beta, \beta\ \alpha, \beta\ \alpha\ \alpha, \beta\ \alpha\ \alpha\ \alpha, \ldots\} = \{\beta\ \alpha^*\}$

$L(G') = L(G)$

De nuevo el mismo ejemplo pero detallándolo aún más

Ejemplo G_1:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow id$

$\alpha = + T$
$(E \rightarrow E\ \alpha)$

$\beta = T$
$(E \rightarrow \beta)$

$E \rightarrow T\ E'$
$E' \rightarrow + T\ E' \mid \lambda$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow id$

$E \rightarrow \beta\ E'$
$E' \rightarrow \alpha\ E'$
$E' \rightarrow \lambda$

$\alpha = * F$
$(T \rightarrow T\ \alpha)$

$\beta = F$
$(T \rightarrow \beta)$

G'_1:

$E \rightarrow T\ E'$
$E' \rightarrow + T\ E' \mid \lambda$
$T \rightarrow F\ T'$
$T' \rightarrow * F\ T' \mid \lambda$
$F \rightarrow id$

$T \rightarrow \beta\ T'$
$T' \rightarrow \alpha\ T'$
$T' \rightarrow \lambda$

Descendente sin retroceso

En la construcción del árbol, las tareas a las que ha de hacer frente un descendente son:

1. elegir cuál es el siguiente nodo a expandir (símbolo no terminal a derivar) y
2. elegir qué regla aplicar.

Un descendente elige siempre como siguiente nodo a expandir el del no terminal más a la izquierda en la forma sentencial.

Un descendente **sin retroceso** requiere de alguna estrategia que haga posible que, en cada instante, haya sólo una regla aplicable. Un tipo de gramáticas, las LL(1), tienen la característica que necesitamos: mirando cuál es el siguiente token que se recibe, se determina cuál es la regla a aplicar para expandir el nodo. Es decir, con las gramáticas LL(1), solo hay una regla válida para cada pareja [N, t] (donde N es un nodo no terminal del árbol y t es el token que envía el analizador léxico).

Descendente sin retroceso

Cuando hay que aplicar una derivación, **solo hay una regla posible**

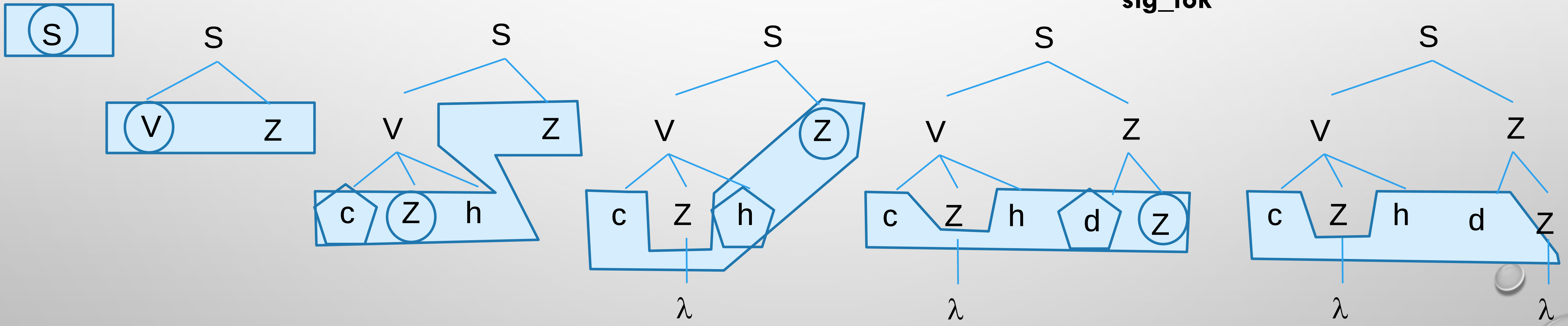
1. ¿No terminal a expandir? → derivaciones a la izquierda
2. ¿Regla a aplicar? → **siguiente token**

S → T V | V Z
T → a T | b T | h
V → λ | c Z h
Z → λ | d Z

ω = c h d

fin ($)

sig_tok



EL SIGUIENTE TOKEN DETERMINA CUÁL ES LA REGLA A APLICAR → Gramáticas LL(1)

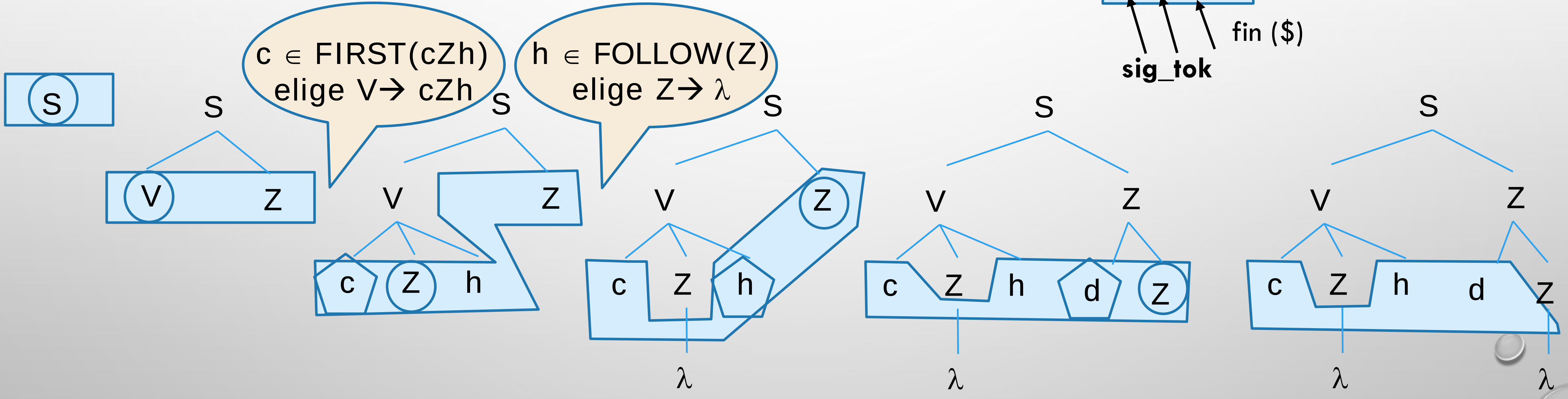
Descendente sin retroceso

Cuando hay que aplicar una derivación, **solo hay una regla posible**

1. ¿No terminal a expandir? → derivaciones a la izquierda
2. ¿Regla a aplicar? → **siguiente token**

S → T V | V Z
T → a T | b T | h
V → λ | c Z h
Z → λ | d Z

ω = c h d



EL SIGUIENTE TOKEN DETERMINA CUÁL ES LA REGLA A APLICAR → Gramáticas LL(1)

Descendente sin retroceso

Cuando hay que aplicar una derivación, **solo hay una regla posible**

- Gramáticas **LL(k)**. Permiten saber qué regla hay que aplicar conociendo, como máximo, los k siguientes tokens de la entrada
- Gramáticas **LL(1)**. Sólo se necesita conocer **un token**.

Construcción del árbol. Se mira el símbolo del nodo activo:

- Si es un terminal, y coincide con **el token actual**, se equiparan (se pide al A. Léx. el siguiente token y se avanza al siguiente nodo del árbol). Si no hubieran coincidido, se habría detectado un error sintáctico.
- Si es un No terminal, se aplica la única regla de derivación que nos llevará a obtener **el token actual**.
  - ✓ a lo sumo, una regla permitirá obtener, desde ese No terminal, el token actual como "primer símbolo terminal más a la izquierda". → **First**
  - ✓ Y ¿qué pasaría si para ese No terminal existe una regla lambda? ¿Con quién se equipara el token actual de la cadena de entrada? Con el "símbolo terminal que vaya a continuación en la forma sentencial". → **Follow**

FIRST

FIRST(X), donde ($X \in \{T \cup N\}$), o
FIRST(α), donde ($\alpha \in \{T \cup N\}^*$)

Conjunto formado por los Terminales que pueden aparecer como **primer símbolo terminal** en las cadenas derivadas a partir de *X* (o a partir de α).

FOLLOW

FOLLOW(A), donde ($A \in N$)

Conjunto formado por los Terminales que pueden aparecer **inmediatamente a continuación** de *A* en alguna forma sentencial.