Tema 4. Lenguajes Formales y Gramáticas

Universidad Europea

LAUREATE INTERNATIONAL UNIVERSITIES

Autor: Prof. Leopoldo Santos 2020

- **Noam Chomsky** empezó con el estudio del **lenguaje natural,** con la idea de estructurarlo, y consiguió una revolución en este campo.

Gramáticas Tipo 0	**Sin restricciones.** Estas gramáticas tienen que tener en su parte izquierda al menos un símbolo no terminal.
Gramáticas Tipo 1	**Dependientes del contexto**. Se las denomina **dependientes del contexto** porque hay que tener en cuenta los símbolos que vienen antes y después del que queremos sustituir (su contexto).
Gramática Tipo 2	**Independientes del contexto. Generan lenguajes independientes del contexto y se caracterizan porque en la parte izquierda de una producción solo pueden tener un símbolo no terminal.**
Gramática Tipo 3	**Expresiones regulares**. Estas **son las gramáticas más restrictivas** y generan lenguajes regulares. En su parte izquierda tienen solo un **no terminal** y **en su parte derecha tienen solo un terminal**.

Indice

- Introducción
- Gramáticas independientes del Contexto (GIC)
- Árboles de Derivación
- Limpieza de gramáticas
- Ambigüedades

- **Una GIC tiene cuatro componentes**:

  – Terminales o Componentes Léxicos. Son los símbolos básicos con los que se forman las cadenas.

    Ejemplo: Palabras clave *IF*, *THEN*, *ELSE*.

  – No terminales. Son variables sintácticas que denotan conjuntos de cadenas.

    Ejemplo: *prop* y *expr* son no terminales.

  – Un no terminal considerado como símbolo inicial.

  – Producciones. Especifican cómo se pueden combinar los terminales y no terminales. Consta de:

    Un no terminal, lado izquierdo de la producción.

    Una flecha.

    Una cadena de terminales y no terminales.

- Convenciones de notación:
  - Símbolos terminales:
    - Dígitos.
    - Operadores (+, -, etc.).
    - Símbolos de puntuación.
    - Cadenas de caracteres en minúscula.
  - Símbolos no terminales:
    - Cadenas de caracteres en mayúscula.
    - La letra *S* suele ser el símbolo inicial.
  - Las letras griegas minúsculas (por ejemplo: α, β, γ), representan cadenas de símbolos gramaticales (terminales y no terminales).
    - Ejemplo: a → α.

- Si a → α_1, a → α_2, ..., a → α_k son todas las producciones con a a la izquierda (producciones de a), se puede escribir:

  a → α_1 | α_2 | ... | α_k

- Si no se indica lo contrario, las primeras producciones corresponden al símbolo inicial.

- La cadena que contiene cero símbolos terminales y no terminales recibe el nombre de cadena vacía, y se representa por λ .

- ¿Por qué no utilizar GIC para definir componentes léxicos?
  - Etapa inicial modular ➔ Separación de análisis léxico y sintáctico.
  - Los lenguajes regulares representan un subconjunto de los lenguajes independientes de contexto.
  - Las reglas léxicas pueden ser expresadas por medio de expresiones regulares pero no las sintácticas.
  - Las expresiones regulares utilizan una notación concisa y más fácil de entender.
  - Los analizadores léxicos son más eficientes a partir de expresiones regulares.

- Ejemplo: Una GIC para expresiones aritméticas simples.

  – Ejemplos:

    82 + 57

    25 + 36 * 92

    (12 + 34) * (65 * -(26 + 12))

  – Terminales o tokens:

    num para todos los números.

    op_suma (‘+’), **op_neg** (‘-’), **op_mult** (‘*’), **parent_izq** (‘(’), **parent_der** (‘)’).

- ¿Gramática que reconozca todas las posibles expresiones?

  E → E O E

  E → (E)

  E → - E

  E → num

  O → +

  O → *

  O → -

- ¿Cumple todas las convenciones de notación?

- También se puede escribir como:

  E → E O E | (E) | - E | num

  O → + | * | -

- Derivaciones.
  - Una derivación es la secuencia de producciones que hay que aplicar, partiendo del símbolo inicial, para generar una cadena del lenguaje.
  - Una cadena pertenece a un lenguaje si se puede construir una derivación que la genere.
  - Paso simple de derivación: Si a → γ, es una producción y α y β son cadenas arbitrarias de símbolos gramaticales, terminales y no terminales:

    $\alpha a \beta \rightarrow \alpha \gamma \beta$.

- Ejemplo:
  - Gramática:

    E → E O E | (E) | - E | num

    O → + | * | -
  - Entrada:

    34 * (26 + 12)
  - Cadena de tokens:

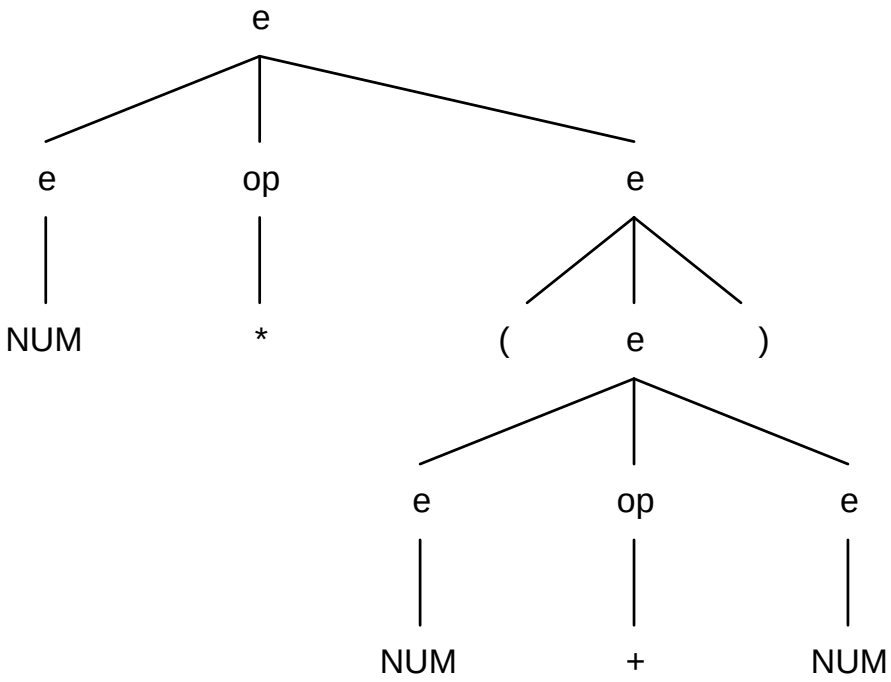
    num * (num + num)

– Secuencia de derivaciones.

		E
E → E O E	⇒	E O E
E → num	⇒	num O E
O → *	⇒	num * E
E → (E)	⇒	num * (E)
E → E O E	⇒	num * (E O E)
E → num	⇒	num * (num O E)
O → +	⇒	num * (num + E)
E → num	⇒	num * (num + num)

- **Arbol de Análisis Sintáctico**.
  - Es una representación gráfica de la estructura sintáctica.
  - Muestra la secuencia de derivaciones realizada, **pero no el orden**.
    Los nodos interiores se corresponden con no terminales.
    Las hojas son terminales.
    Cada nodo padre es la parte izquierda de una producción, y sus hijos son la parte derecha, de izq a dcha.

- Ejemplo: num * (num + num)



- **¿Qué es una derivación?**

Se representa por **x ==> y** y es la aplicación de una regla de producción a→b a una cadena x para convertirla en otra cadena o palabra y.

Cuando lo que se aplica es una **secuencia de producciones** a una cadena se representa por **x ==>* y**, queriendo indicar que llegamos a otra cadena en más de un paso. En el ejemplo anterior, D → T V ==>* int a, b, c;, y realizamos, por tanto, derivaciones en más de un paso para obtener la cadena final

- **Derivación más a la izquierda frente a derivación más a la derecha.**
  - Derivación más a la izquierda.
    Derivar el no terminal más a la izquierda.
    El ejemplo anterior utilizaba derivación más a la izquierda.
  - Derivación más a la derecha.
    Derivar el no terminal más a la derecha.

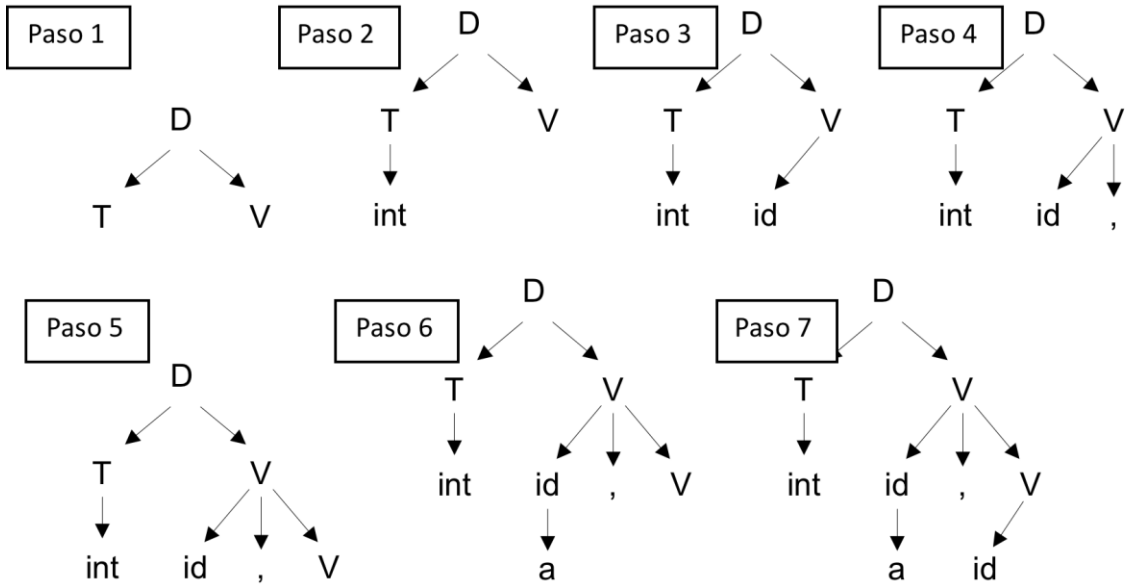
Supongamos la siguiente sentencia para la gramática de declaración de variables de la gramática que ya hemos utilizado anteriormente: **int a, b**;

D → T V

T → int | real | char

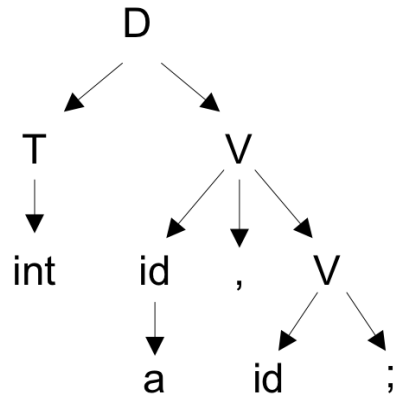
V → id; | id, V

Para validar esta sentencia vamos a hacer derivaciones por la izquierda

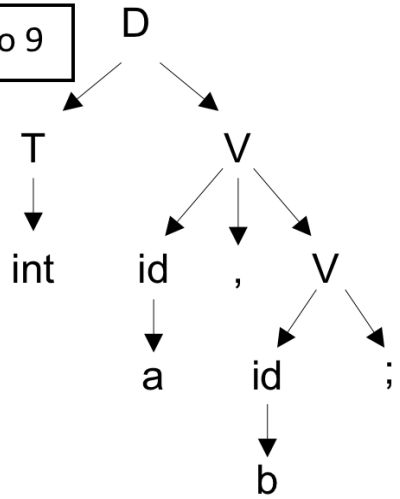


Arboles de Derivación

Paso 8



Paso 9



- Ejemplo:
  - Entrada:

    25 + 32 * 17
  - Cadena de tokens:

    num + num * num
  - Dos posibles derivaciones:

Gramática:

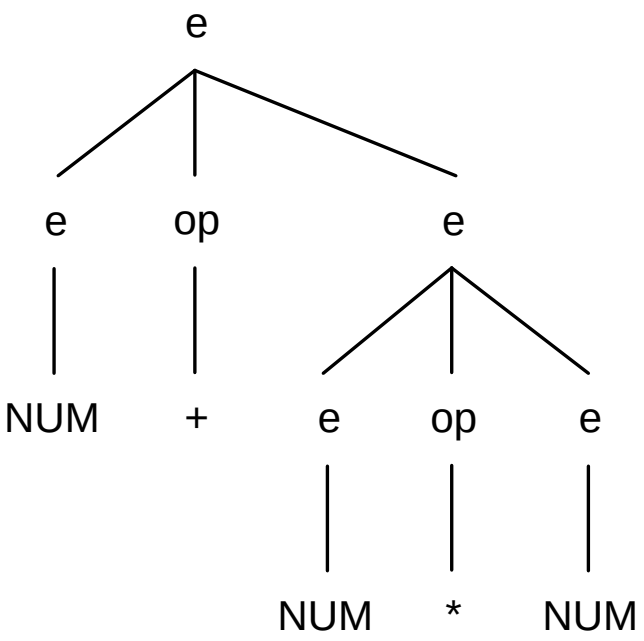
E → E O E | (E) | - E | num

O → + | *

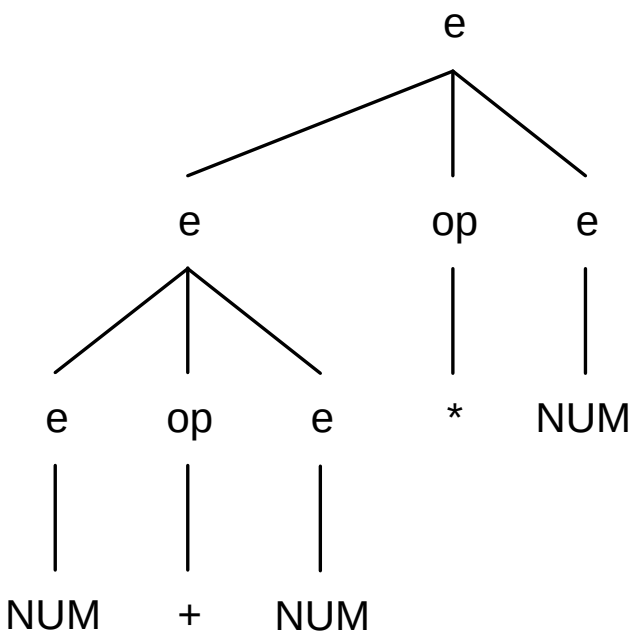
E
E O E
num O E
num + E
num + E O E
num + num O E
num + num * E
num + num * num

E
E O E
E O num
E * num
E O E * num
E O num * num
E + num * num
num + num * num

Arboles de Derivación



25 + (32 * 17) = 569



(25 + 32) * 17 = 969

- Las gramáticas que nos permiten definir lenguajes de programación están formadas por una gran cantidad de reglas en notación BNF que puede contener errores, bien porque no son necesarias, bien porque tienen símbolos superfluos o innecesarios. Al proceso de corregir la gramática se le denomina **“limpieza de la gramática”**. El objetivo es que tanto los símbolos como las reglas que se utilizan en la gramática sirvan para algo.

- Hay dos tipos de errores que se cometen con los símbolos: símbolos superfluos y símbolos inaccesibles.
- **Símbolos superfluos**: Depende del conjunto al que pertenezcan (terminales o no terminales)
  - **Símbolo no terminal superfluo**: Es aquel del que se derivan palabras que no contienen ningún terminal o bien incluyen al mismo no terminal.
  - **Símbolo terminal superfluo**: Es aquel que no puede ser alcanzado por derivación desde el axioma.
- **Símbolos inaccesibles**: Son aquellos símbolos no terminales que no pueden ser alcanzados por derivaciones desde el axioma de la gramática.

Hay tres tipos de errores que se cometen con la reglas y son aquellas que no generan derivaciones útiles: **reglas innecesarias**, **reglas no generativas** y **reglas de redenominación**

- **Reglas innecesarias:** Son reglas del tipo A → A, donde A pertence a los no terminales

- **Reglas no generativas:** Son reglas del tipo A → λ, cuando A es distinta de S (el axioma de la gramática)

- **Reglas de redenominación:** Son reglas del tipo A → B, donde tanto A como B pertenecen a los no terminales.

Una gramática **está bien formada si está limpia**, es decir:

- Sin reglas innecesarias (A → A)
- Sin símbolos inaccesibles
- Sin símbolos superfluos
- Sin reglas no generativas (A → λ)
- Sin reglas de redenominación (A → B)

Ejemplo: G: ({a, b, c, d},{X, Y, V, W, Z}, X, P}, donde P está compuesto por las siguientes reglas o producciones

X→ Wa| Zba | λ

Y → aVa

V → Y

W → bX

Z → Za

Observando la gramática se puede determinar:

- No hay reglas innecesarias.
- Los símbolos Y y V son inaccesibles, puesto que no podemos llegar a ellos mediante derivaciones desde el axioma de la gramática (X).
- El símbolo Z es un símbolo no terminal superfluo Z → Za puesto que cuando se hagan derivaciones siempre aparece el propio símbolo Z.
- Los símbolos c y d son símbolos terminales superfluos.
- No hay reglas no generativas
- La regla V → Y es una regla de redenominación (este error es muy común)
- Para limpiar esta gramática eliminamos tanto los símbolos inaccesibles y superfluos como las producciones en las que estos aparecen. Lo mismo se hace con las reglas innecesarias y com ambas eliminaciones no se altera el lenguaje generado por la gramática.

Si limpiamos esta gramática nos queda G: ({a, b, },{X, W}, X, P}

X→ Wa~~| Zba~~ | λ

~~Y → aVa~~

~~V → Y~~

W → bX

~~Z → Za~~

Obteniéndose finalmente:

X→ Wa| λ

W → bX

NOTA: El objetivo, no es tanto el dominar los algoritmos de limpieza de gramáticas, sino el reconocer los errores y no cometerlos cuando diseñemos una gramática

- Ejemplo: Construir una GIC que reconozca secuencias de proposiciones separadas por símbolos de punto y coma en los bloques *begin-end* de Pascal.
  - Debe existir al menos una proposición.

    bloque → BEGIN lista_props END

    lista_props → prop ; lista_props | prop

  - Puede existir una secuencia de proposiciones vacía.

    bloque → BEGIN lista_props END

    lista_props_opc → lista_props | λ

    lista_props → prop ; lista_props | prop

La gramática anterior es ambigua.

•La aplicación de diferentes órdenes de derivación produce diferentes árboles sintácticos.

•Los resultados obtenidos con cada orden de derivación son diferentes ➔ Sólo uno de ellos puede ser el correcto.

•En ocasiones se puede eliminar la ambigüedad rescribiendo la gramática con nuevos símbolos no terminales.

▪Ejemplo: Modificar la gramática de las expresiones aritméticas para que no exista ambigüedad.

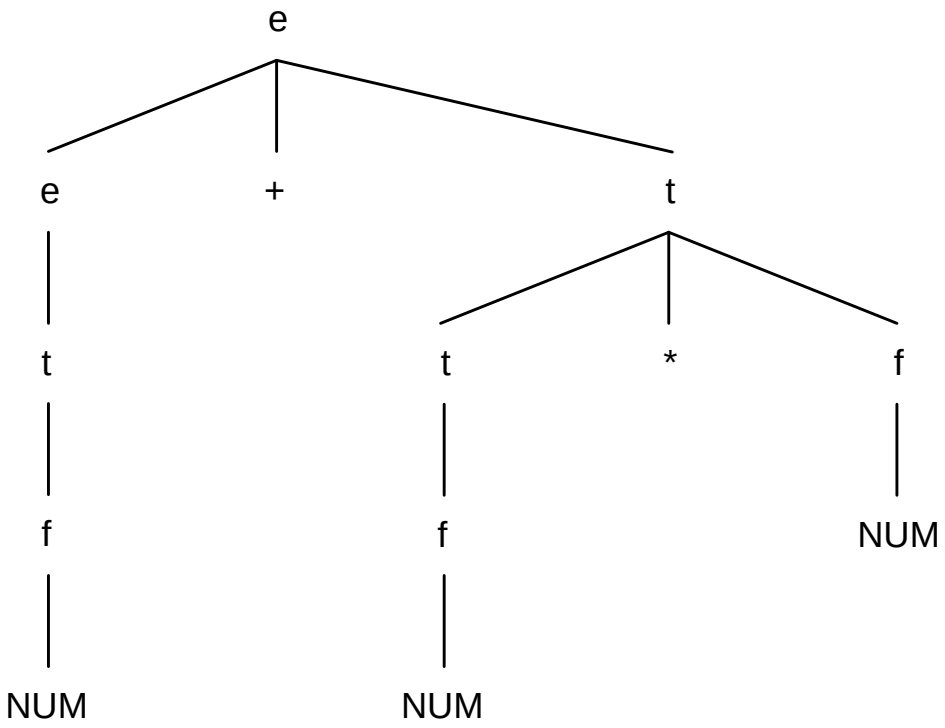
Prioridad de operadores.

E → E + T | T

T → T * F | F

F → num | (E)

Arbol de análisis sintáctico para la cadena: num + num * num



- Ejemplo: Razonar si la siguiente gramática es ambigua.

  prop → IF exp THEN prop

  prop → IF exp THEN prop ELSE prop

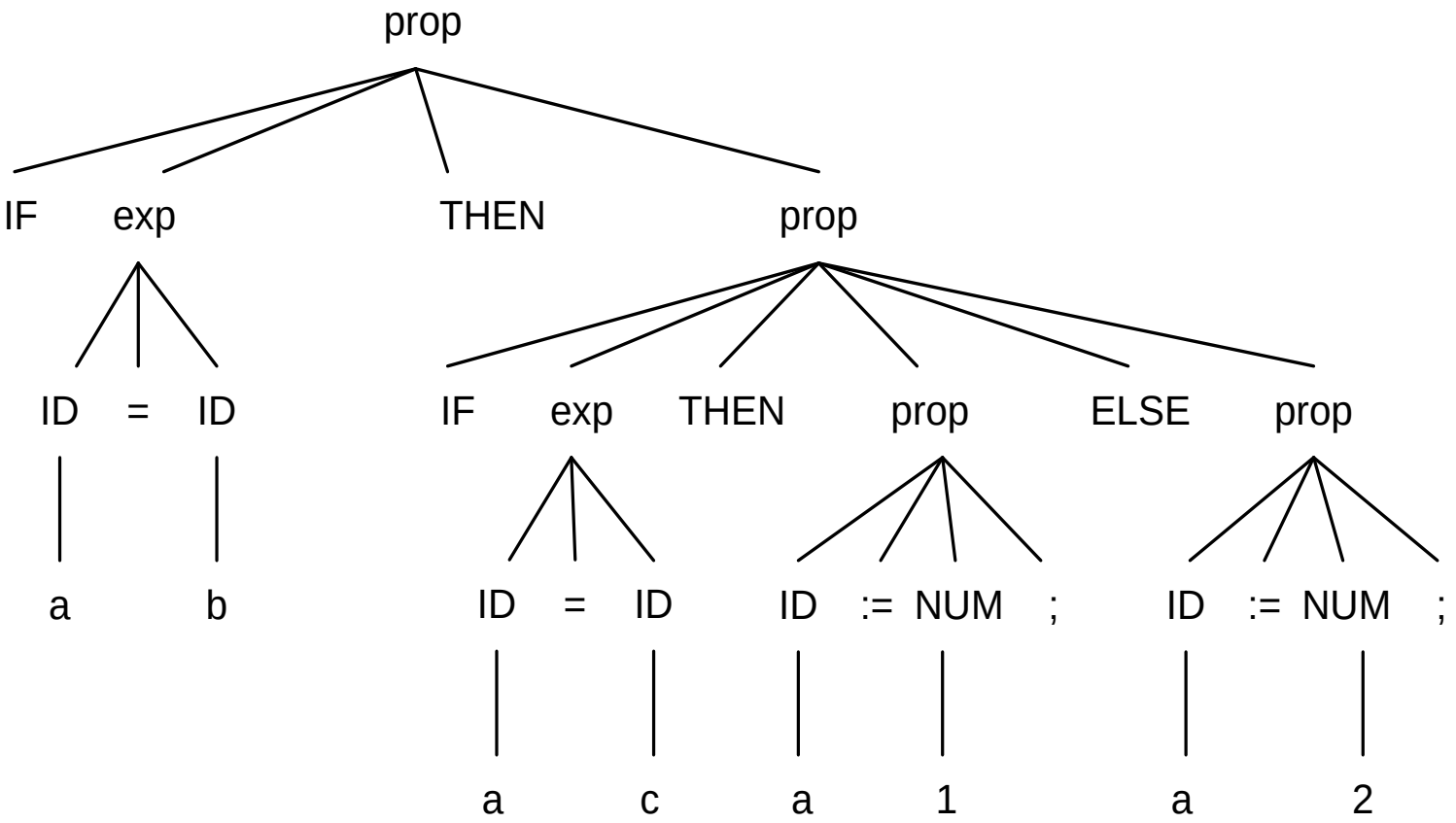
  prop → IDENTIFICADOR := NUMERO ;

  exp → IDENTIFICADOR = IDENTIFICADOR

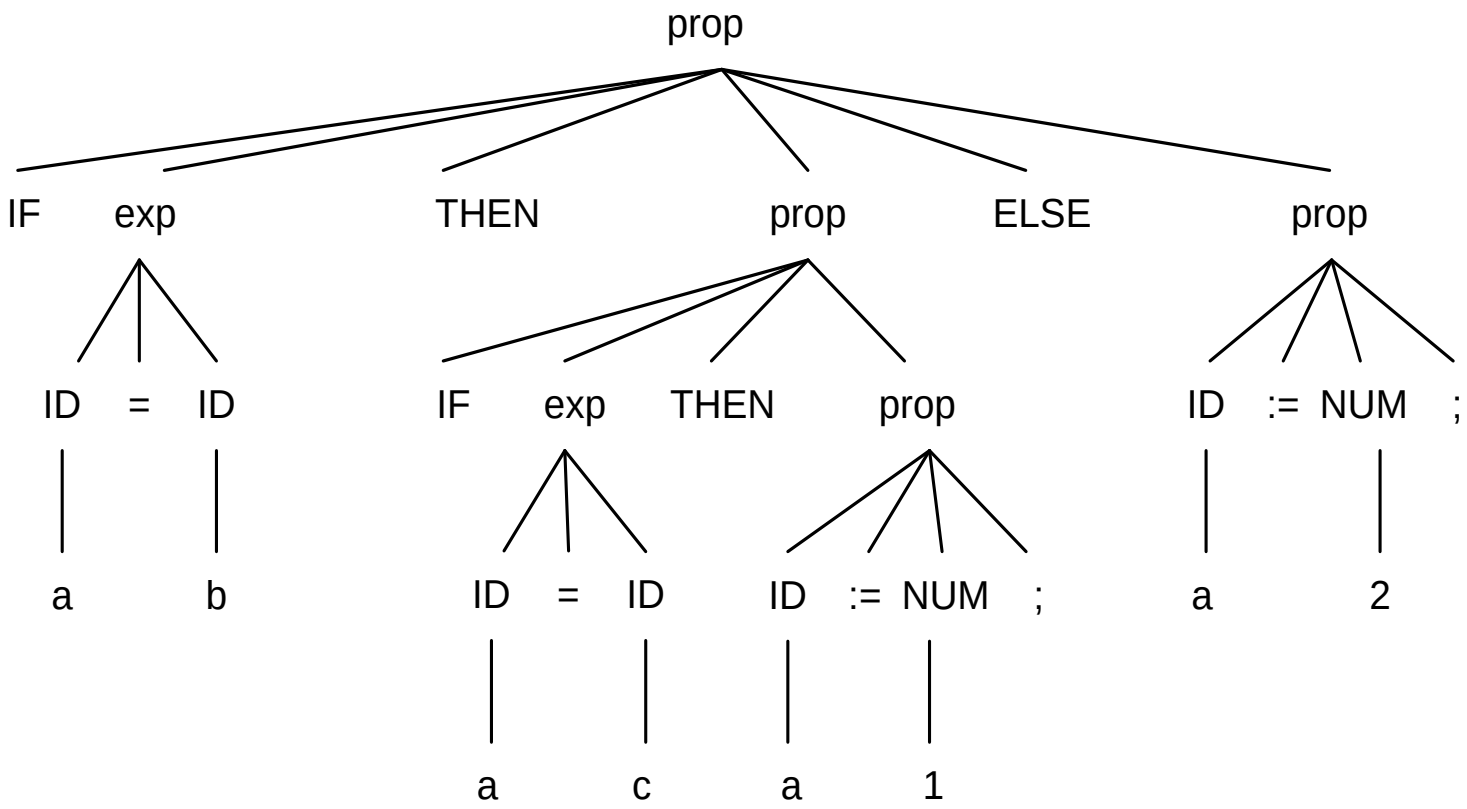
  – ¿Existe un único árbol para la siguiente cadena?

```
IF a=b THEN
        IF a=c THEN
                a:=1;
        ELSE
                a:=2;
```

Ambigüedad


prop
IF
exp
THEN
prop
ID
=
ID
IF
exp
THEN
prop
ELSE
prop
a
b
ID
=
ID
ID
:=
NUM
;
ID
:=
NUM
;
a
c
a
1
a
2

Ambigüedad


prop
IF
exp
THEN
prop
ELSE
prop
ID
=
ID
IF
exp
THEN
prop
ID
:=
NUM
;
a
b
ID
=
ID
ID
:=
NUM
;
a
2
a
c
a
1

– ¿Cómo podríamos modificar la anterior gramática para que no sea ambigua?

prop → prop2

prop → prop3

prop2 → IF exp THEN prop

prop3 → IDENTIFICADOR := NUMERO ;

prop3 → IF exp THEN prop3 ELSE prop

- Una gramática recursiva por la izquierda puede hacer que un AS por descenso recursivo, con o sin retroceso, entre en un bucle infinito.
  - Cuando se intenta expandir *a*, puede que de nuevo se intente expandir *a* sin haber consumido ningún símbolo de la entrada.
  - Ejemplo:

    s → rA | B
    r → rC | D

  - Debemos eliminar la recursividad directa por la izquierda. Para ello, reescribiremos la producción de otra forma.

    $r \rightarrow r\ \alpha \mid \beta$

    se sustituye por:

    $r \rightarrow \beta\ r'$

    $r' \rightarrow \alpha\ r' \mid \lambda$

- Ejemplo: Gramática para las expresiones aritméticas.

  E → E + T | T
  T → T * F | F
  F → (F) | id

  – Esta gramática contiene algunas producciones recursivas por la izquierda. Debemos reescribir dichas producciones.

E → T E’
E’→ + T E’ | λ
T → F T’
T’ → * F T’ | λ
F → (E) | id

- Hemos comprobado anteriormente que en ocasiones nos encontramos con situaciones en las que no podemos decidir entre varias producciones a elegir.
  - En esos casos, se debe producir retroceso si la entrada no concuerda con el árbol generado por la primera producción elegida.
  - Podemos rescribir las producciones para retrasar la decisión hasta haber analizado lo suficiente de la entrada como para elegir la opción correcta.

    $a \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$

    Se convierte en:

    $a \rightarrow \alpha\ b$

    $b \rightarrow \beta_1 \mid \beta_2$

Universidad Europea

LAUREATE INTERNATIONAL UNIVERSITIES

Madrid Valencia Canarias