



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Análisis sintáctico

Felipe Restrepo Calle

ferestrepoca@unal.edu.co

Departamento de Ingeniería de Sistemas e Industrial
Facultad de Ingeniería
Universidad Nacional de Colombia
Sede Bogotá

- 1. Conceptos básicos**
- 2. Análisis sintáctico**
- 3. Diseño de gramáticas**

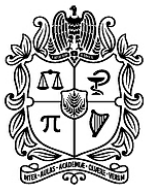


Gramática BNF

La notación formal usada para describir la gramática de un lenguaje es la **Forma Backus-Naur (BNF)**.

$$G = (V_N; V_T; S; P)$$

- V_N es el conjunto de símbolos **no terminales** (variables)
- V_T es el conjunto de símbolos **terminales** (todos pertenecen al alfabeto)
- S es el **símbolo inicial** de la gramática
- P es el conjunto de **producciones o reglas** de la gramática



Gramáticas

Normalmente, solamente se especifica el conjunto de producciones **P**, y se asume que el símbolo inicial de la gramática es la parte izquierda de la primera producción.

Ejemplo:

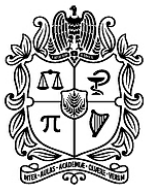
$A \rightarrow a B C$

$B \rightarrow b \text{ bas}$

$B \rightarrow \text{big } C \text{ boss}$

$C \rightarrow \varepsilon$

$C \rightarrow c$



Gramáticas - Derivaciones

Una **derivación** es una secuencia de cadenas de símbolos en la que cada cadena es el resultado de la aplicación de una regla de la gramática a la cadena anterior.

Una **derivación válida** es aquella en la que la primera cadena de la secuencia es el símbolo inicial, y la última es una cadena de terminales.

Ejemplo:

A \rightarrow **a** B C
B \rightarrow **b** **bas**
B \rightarrow **big** C **boss**
C \rightarrow ϵ
C \rightarrow **c**

Derivación por la derecha

A \rightarrow **a** B C
 \rightarrow **a** B **c**
 \rightarrow **a** **big** C **boss** **c**
 \rightarrow **a** **big** **boss** **c**

Derivación por la izquierda

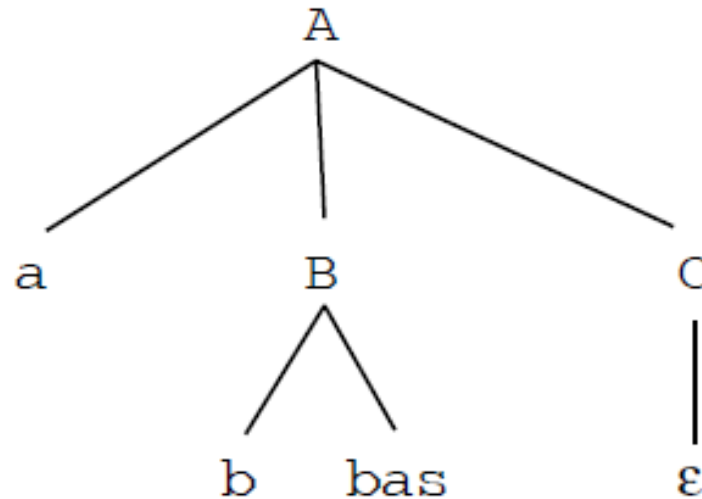
A \rightarrow **a** B C
 \rightarrow **a** **b** **bas** C
 \rightarrow **a** **b** **bas**

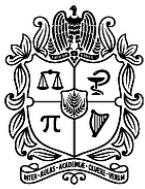


Gramáticas – Árbol de derivación

Un **árbol de derivación** es un árbol en el que se representa una derivación válida de una cadena (pero no se especifica el orden en que se han aplicado las reglas).

Ejemplo:





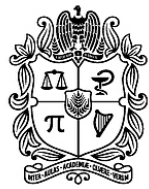
Gramáticas - Recursividad

- Una gramática se dice que es **recursiva por la izquierda** si tiene al menos una regla de esta forma:

$$E \rightarrow E \text{ opsuma } T$$

- También, una gramática puede presentar **recursividad por la derecha**:

$$E \rightarrow T \text{ opsuma } E$$



Gramáticas – Factores comunes

Una gramática tiene **factores comunes por la izquierda** si tiene símbolos comunes al principio de la parte derecha de la regla en al menos dos reglas:

$$A \rightarrow B \mathbf{a} C$$
$$A \rightarrow B \mathbf{a} d$$

...



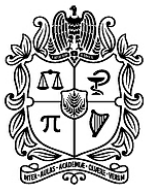
Gramática ambigua

Una gramática es **ambigua** cuando es posible encontrar más de un árbol de derivación para una cadena.

Ejemplo:

$$\begin{aligned} E &\rightarrow E \text{ op } E \\ E &\rightarrow \text{num} \end{aligned}$$

La cadena "2+3-4" tiene dos árboles de derivación, y la cadena "2+3-4+5" tiene más de dos árboles.



Gramática ambigua

IMPORTANTE: la única forma de saber si una gramática es ambigua es encontrando una cadena con más de un árbol de derivación. Sin embargo, si la gramática tiene alguna de las siguientes características, es sencillo encontrar este tipo de cadenas:

- Gramáticas con ciclos simples o menos simples:

$$S \rightarrow A$$
$$S \rightarrow \mathbf{a}$$
$$A \rightarrow \mathbf{S}$$

- Alguna regla de la forma:

$$E \rightarrow \mathbf{E} \dots \mathbf{E}$$

Con cualquier cadena de terminales y no terminales entre las dos E



Gramática ambigua

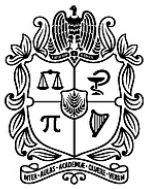
IMPORTANTE: la única forma de saber si una gramática es ambigua es encontrando una cadena con más de un árbol de derivación. Sin embargo, si la gramática tiene alguna de las siguientes características, es sencillo encontrar este tipo de cadenas:

- Un conjunto de reglas que ofrezcan caminos alternativos entre dos puntos:

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

- Producciones recursivas en las que las variables no recursivas de la producción puedan derivar a la cadena vacía:

$$S \rightarrow H R S$$
$$S \rightarrow s$$
$$H \rightarrow h \mid \epsilon$$
$$R \rightarrow r \mid \epsilon$$



Gramática ambigua

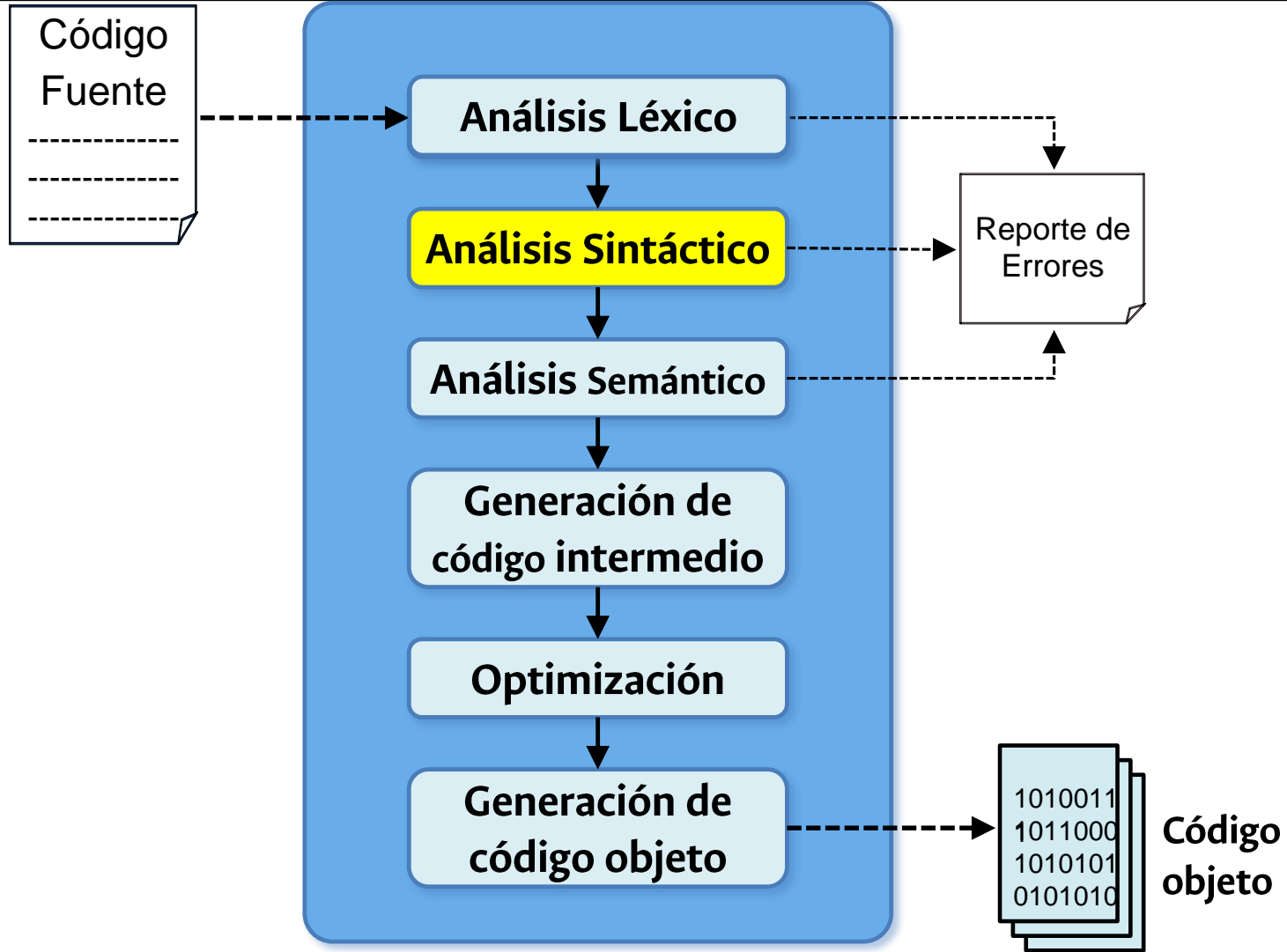
IMPORTANTE: la única forma de saber si una gramática es ambigua es encontrando una cadena con más de un árbol de derivación. Sin embargo, si la gramática tiene alguna de las siguientes características, es sencillo encontrar este tipo de cadenas:

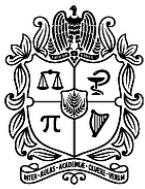
- Símbolos no terminales que puedan derivar a la cadena vacía y a la misma cadena de terminales, y que aparezcan juntas en la parte derecha de una regla:

$$S \rightarrow H \ R$$
$$H \rightarrow h \mid \epsilon$$
$$R \rightarrow r \mid h \mid \epsilon$$



1. Conceptos básicos
- 2. Análisis sintáctico**
3. Diseño de gramáticas





¿Qué es el análisis sintáctico?

Proceso que permite **decidir si una cadena dada pertenece o no a una gramática independiente del contexto (GIC).**

- **Tipo de errores:**

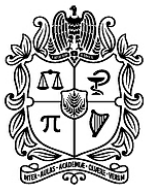
```
int foo(int x, int y)
{
  for(int i=0; x i)
  {
    fi(i>y)
    return x
  }
  return y;
}
```

No se esperaba “)”

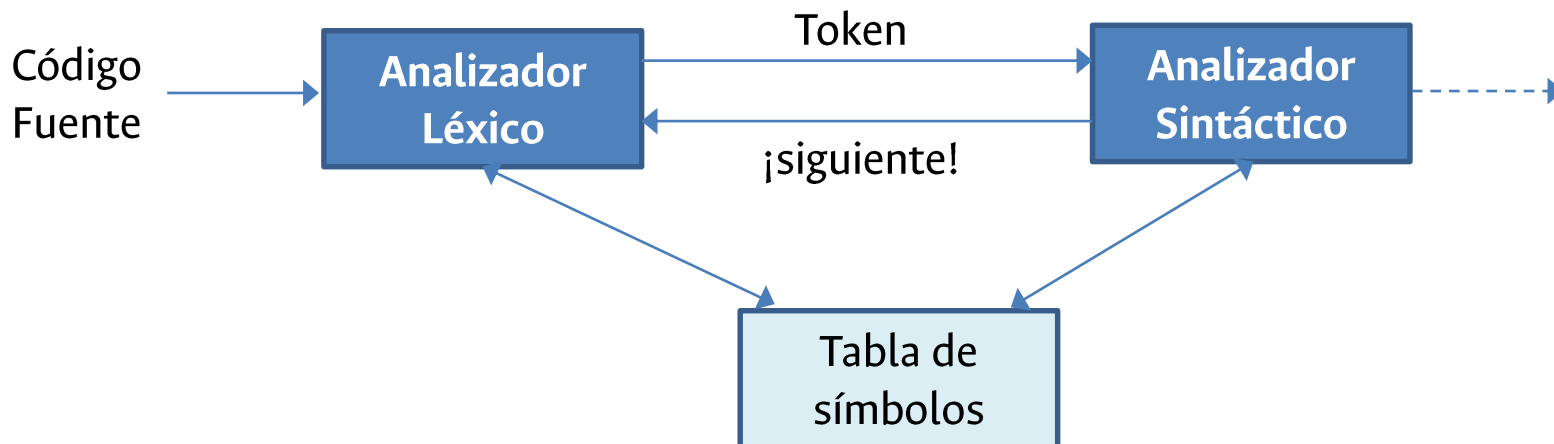
No es una expresión válida

Falta “;”

Después del identificador “fi” se esperaba ...



¿Cómo funciona el analizador sintáctico?



- La primera acción de un analizador sintáctico es obtener un **token** de la entrada, llamando al analizador léxico (que trabaja como un subprograma).
- El analizador va leyendo **tokens** del analizador léxico a la vez que va generando la **traducción**, comprobando que la **sintaxis** es correcta y comprobando las **restricciones semánticas**.

IMPORTANTE: Las tres tareas (traducción, sintaxis, semántica) se realizan de forma simultánea, aunque a veces es necesario acumular varios tokens para realizar alguna comprobación o generar la traducción.



Ejemplo: gramática de expresiones simples

$E \rightarrow E \text{ opsuma } T$

$E \rightarrow T$

$T \rightarrow T \text{ opmul } F$

$T \rightarrow F$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

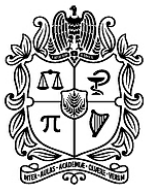
$F \rightarrow \text{pari } E \text{ pard}$

Cadenas:

$2 + 3 * 4$

$2 + 3 - 4$

$2 + 3 * (4 - 5)$

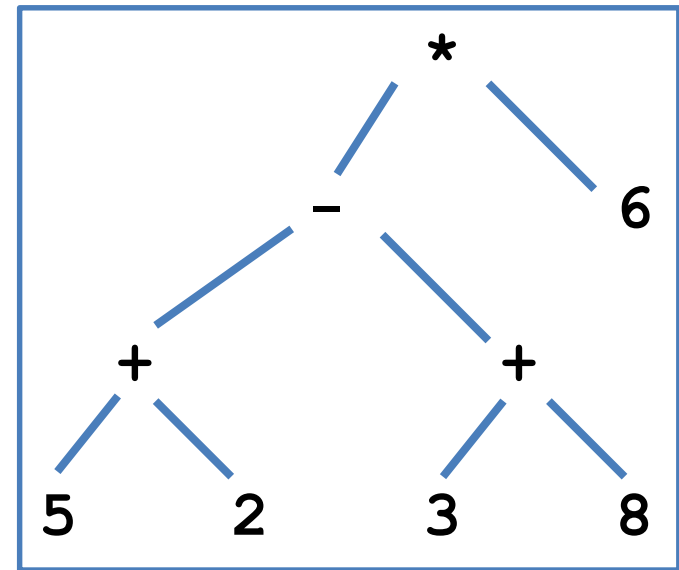


Árbol AST (*Abstract Syntax Tree*)

Árbol de Sintaxis Abstracta (AST)

- Captura estructuras anidadas
- Abstrae la sintaxis específica
- Compacto y fácil de usar

Ejemplo: $((5+2)-(3+8))*6$

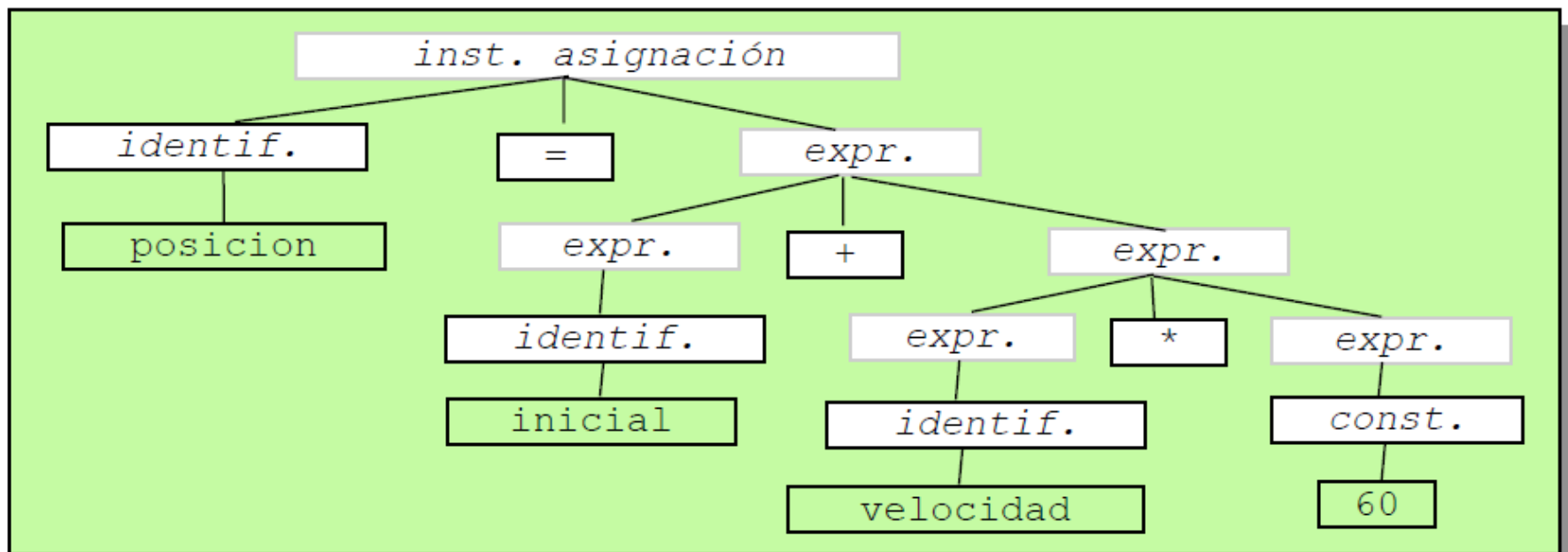




Árbol Sintáctico (*parse tree*)

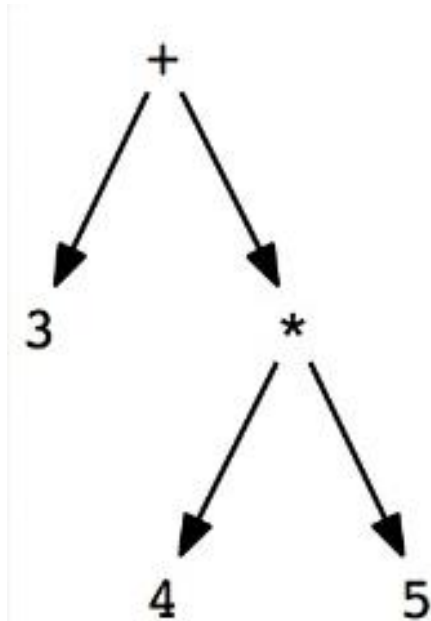
- El árbol sintáctico correspondiente es

`posicion = inicial + velocidad * 60`



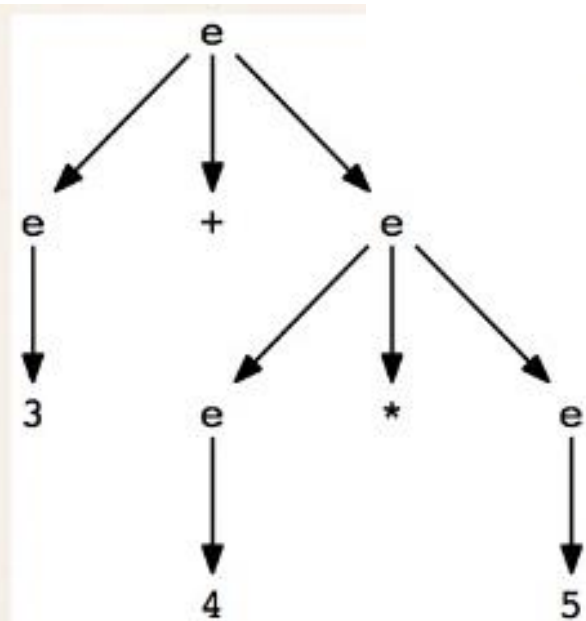


Árbol de Sintaxis Abstracta (AST)



Vs.

Árbol sintáctico (*parse tree*)





Algoritmos de análisis sintáctico

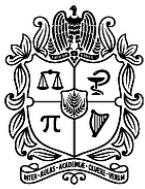
- Para cualquier GIC:
 - Cocke-Younger-Kasami (CYK), Earley, Tomita, ... **$O(n^3)$**
- Si se desea costo temporal lineal, **$O(n)$** , es necesario poner restricciones a las GIC, es decir, usar subconjuntos del conjunto de las GIC. Hay dos estrategias:
 - **Análisis sintáctico descendente (ASD)**
 - **Análisis sintáctico ascendente (ASA)**

Implementación de analizadores sintácticos:

- ✓ A mano (para gramáticas simples)
- ✓ Usando generadores automáticos: yacc/bison, ANTLR, PCCTS, ...

Características **no** deseables para el análisis lineal





- Análisis sintáctico descendente:
 - Recursividad por la izquierda
 - Factores comunes por la izquierda
 - Ambigüedad
- Análisis sintáctico ascendente:
 - Ambigüedad








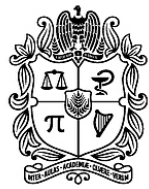
Análisis sintáctico descendente (ASD)

Reproduce una derivación por la **izquierda** de la cadena de entrada

Ejemplo: `int a,b,c;`

D → *Tipo id L* 
Tipo → **int** 
Tipo → **float**
L → **coma id L** 
L → **pyc** 

D → *Tipo id L* 
→ **int id(a) L** 
→ **int id(a) coma id(b) L** 
→ **int id(a) coma id(b) coma id(c) L** 
→ **int id(a) coma id(b) coma id(c) pyc** 



Análisis sintáctico ascendente (ASA)

Reconstruye la inversa de una derivación por la **derecha** de la cadena de entrada.

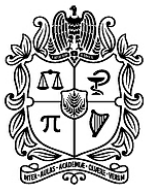
Ejemplo: `int a,b,c;`

<i>D</i>	→	<i>Tipo id L</i>	←
<i>Tipo</i>	→	<code>int</code>	←
<i>Tipo</i>	→	<code>float</code>	
<i>L</i>	→	<code>coma id L</code>	←
<i>L</i>	→	<code>pyc</code>	←

<code>int id(a) coma id(b) coma id(c) pyc</code>	←
<code>Tipo id(a) coma id(b) coma id(c) pyc</code>	←
<code>Tipo id(a) coma id(b) coma id(c) L</code>	←
<code>Tipo id(a) coma id(b) L</code>	←
<code>Tipo id(a) L</code>	←
<i>D</i>	←



1. Conceptos básicos
2. Análisis sintáctico
- 3. Diseño de gramáticas**



Diseño de gramáticas

- Un buen diseño de la gramática nos permitirá **reflejar de forma natural características semánticas** del lenguaje en el árbol de derivación, y esto permitirá que la **traducción sea más sencilla**.
- Es importante diseñar una buena gramática, pero luego es posible que se tenga que **modificar según el tipo de analizador sintáctico** que se desee utilizar.
- Es posible que al diseñar el **proceso de traducción sea necesario rediseñar la gramática** para facilitar el diseño del traductor.



Diseño de gramáticas para expresiones:

Asociatividad

- La asociatividad indica cómo se agrupan los operandos de un operador cuando aparecen más de dos operandos.

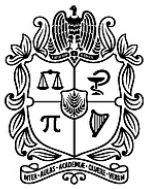
Ejemplo:

El operador “-” tiene **asociatividad por la izquierda**

$$4 - 3 - 2 = -1$$

Si “-” tuviera **asociatividad por la derecha**:

$$4 - 3 - 2 = 3$$



Diseño de gramáticas para expresiones:

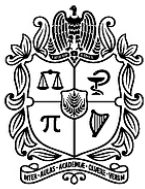
Asociatividad

- Los paréntesis permiten alterar la asociatividad por defecto de un operador.
- ¿Cómo se refleja la asociatividad en una gramática?

Asociatividad izquierda: $E \rightarrow E \text{ operador } T$

Asociatividad derecha: $E \rightarrow T \text{ operador } E$

- Pero... ¿no son todos los operadores asociativos por la izquierda?
Sí, casi todos, pero no todos: “a=b=c=0”



Diseño de gramáticas para expresiones: Precedencia

- En la mayoría de los lenguajes de programación, unos operadores se evalúan antes que otros.

Ejemplo:

$$2 + 3 * 4 = 14$$

El operador $*$ tiene mayor precedencia (prioridad) que el operador $+$.

- Los paréntesis permiten alterar la precedencia:

$$(2 + 3) * 4 = 20$$



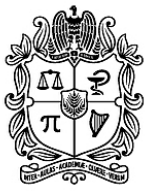
Diseño de gramáticas para expresiones: Precedencia

- ¿Cómo se refleja la precedencia en una gramática? Con un no terminal distinto para cada nivel de precedencia:

```
Expr  → Expr || EBool
Expr  → EBool
Ebool → EBool && ExpRel
Ebool → ExpRel
ExpRel → E oprel E
E      → E opsuma T
E      → T
T      → T opmul F
T      → F
F      → ...
```

↑ **Menor precedencia**

↓ **Mayor precedencia**



Diseño de gramáticas para expresiones:

- Algunos operadores no permiten usar más de dos operandos:

`a < b < c`

→

`a < b && b < c`

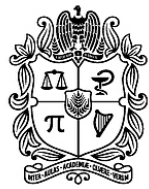


Tabla de asociatividades y precedencias

Es una tabla en la que aparecen los operadores y su asociatividad, ordenados de menor a mayor precedencia:

Ejemplo:

Operador	Asociatividad
@	Izquierda
%	Derecha
#	Izquierda

Menor precedencia

Mayor precedencia

A → A @ B
A → B
B → C % B
B → C
C → C # D
C → D
D → ...



Operadores unarios

Para los operadores unarios se requiere un buen conocimiento del lenguaje y de las gramáticas.

Los ejemplos más conocidos son:

- El operador de negación “!”, que además permite que se repita el operador: “!!!true”

$$\text{ExpRel} \rightarrow ! \text{ExpRel}$$

- El operador de cambio de signo, “-” o “+”, que no permite repeticiones (por ejemplo, “---3” no es correcto)

$$E \rightarrow \text{opsuma } T$$



Ejercicios: asociatividad y precedencia

Diseñar gramáticas no ambiguas para los lenguajes de las expresiones que se pueden construir usando identificadores, paréntesis y los operadores binarios que aparecen en las siguientes tablas (de menor a mayor precedencia):

1.

Operador	Asociatividad
#	Derecha
\$	Izquierda
@ %	Derecha
&	Derecha

2.

Operador	Asociatividad
!	Izquierda
%	Derecha
+ *	Derecha
=	Izquierda

Nota: Los operadores que aparecen en la misma fila tienen la misma precedencia y asociatividad.