



**Universidad
Europea de Madrid**

LAUREATE INTERNATIONAL UNIVERSITIES

ANÁLISIS SINTÁCTICO I

GRAMÁTICAS

© Todos los derechos de propiedad intelectual de esta obra pertenecen en exclusiva a la Universidad Europea de Madrid, S.L.U. Queda terminantemente prohibida la reproducción, puesta a disposición del público y en general cualquier otra forma de explotación de toda o parte de la misma.

La utilización no autorizada de esta obra, así como los perjuicios ocasionados en los derechos de propiedad intelectual e industrial de la Universidad Europea de Madrid, S.L.U., darán lugar al ejercicio de las acciones que legalmente le correspondan y, en su caso, a las responsabilidades que de dicho ejercicio se deriven.

Índice

Presentación	4
Introducción	5
Gramáticas independientes del contexto	7
Árboles de derivación	9
¿Qué es una derivación?	9
¿Qué es un árbol de derivación?	9
Ejemplo de árbol de derivación	11
Limpieza de gramáticas	15
Símbolos superfluos	15
Símbolos inaccesibles	15
Orden de limpieza de gramáticas	16
Ambigüedades	18
Eliminar la recursividad a izquierdas	18
Eliminar varias alternativas que comienzan igual	18
Gramáticas ambiguas	20
Precedencia y asociatividad	21
El problema del else ambiguo	23
Resumen	25

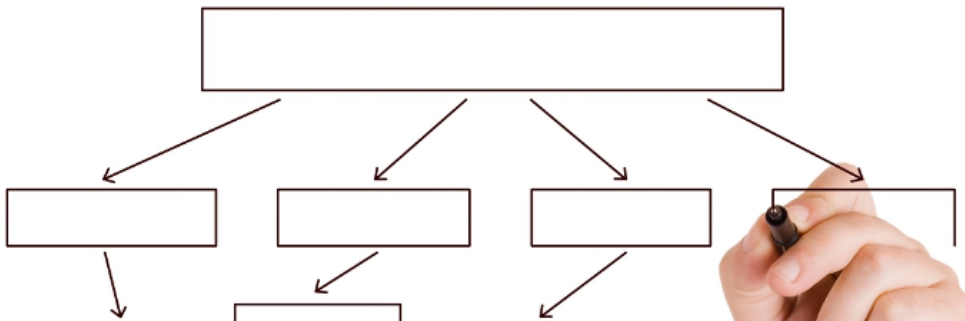
Presentación

El objetivo de este tema es aprender a **diseñar gramáticas a partir de un conjunto de reglas que estas deben cumplir**. Para ello, vamos a ampliar algunos conceptos necesarios para las gramáticas tipo 2 de la jerarquía de Chomsky, a la vez que se revisan conceptos sobre ambigüedades vistos anteriormente.

También es importante **aprender a validar una sentencia para saber si pertenece a una gramática determinada o no**, además de aprender a reconocer cuándo una gramática no está limpia y qué orden hay que seguir para limpiarla.

Los objetivos a conseguir en este tema son:

- Entender la motivación de las Gramáticas Independientes del Contexto (GIC).
- Conocer el funcionamiento de los árboles de derivación.
- Cómo limpiar una gramática.
- Cuál es el orden de limpieza de gramáticas.
- Identificar las ambigüedades y las gramáticas ambiguas.
- Conocer los conceptos de precedencia y asociatividad.
- El problema del e/se ambiguo.



Introducción

Es importante dominar conceptos como **qué es una gramática, cómo se especifica de una manera formal, qué es el lenguaje generado por una gramática, los tipos de gramáticas que existen** (jerarquía de Chomsky) para construir los conceptos que veremos en este tema. También necesitamos conceptos como el de *token* o componentes léxico junto con el de las expresiones regulares (gramáticas tipo 3 de la jerarquía de Chomsky).

La manera formal de describir una gramática es mediante cuatro términos: el alfabeto de los símbolos terminales (ΣT), el alfabeto de los símbolos no terminales (ΣN), el axioma o símbolo inicial de la gramática (S) y un conjunto de producciones (P).

Se denota mediante $G = \{\Sigma T, \Sigma N, S, P\}$, y su forma de representarla es mediante la notación denominada Forma de Backus-Naur (notación BNF), como vemos en la siguiente producción $E \rightarrow E + E$, donde el único símbolo terminal es $+$, el no terminal es E , que a su vez es el axioma de la gramática.

 [La independencia de una gramática respecto del contexto](#)
En detalle

También es importante validar que esa gramática es capaz de reconocer sentencias del lenguaje que se ha definido previamente, y para ello utilizamos unas estructuras denominadas **árboles de derivación** o **árboles sintácticos**.

Por todo lo anterior, las gramáticas nos proporcionan las siguientes ventajas (Aho et al, 1986):

1. Proporcionan una **especificación sintáctica precisa y fácil** de entender de un lenguaje de programación.
2. A partir de algunas clases de gramáticas, **se puede construir automáticamente un analizador sintáctico eficiente** que determine si un programa fuente está sintácticamente bien formado o no.
3. Imparte una estructura a un lenguaje de programación útil para la traducción de programas fuente a código objeto correcto y para la detección de errores.
4. Se pueden **añadir mejoras al lenguaje fácilmente**.

En detalle**La independencia de una gramática respecto del contexto**

Principalmente, debemos ser capaces de construir una gramática a partir de la especificación de un lenguaje y, una vez obtenida, comprobar que no tiene errores ni ambigüedades y dejarla preparada para su utilización por un analizador sintáctico.

Para ello, **debemos saber distinguir cuándo una gramática es independiente del contexto** (gramáticas tipo 2 de la jerarquía de Chomsky) y cuándo no lo es.

Gramáticas independientes del contexto

Una gramática independiente del contexto (GIC) es una especificación para la estructura sintáctica de un lenguaje de programación (Louden, 2004). Dependiendo del autor, también se las denomina **gramáticas libres de contexto**.

 [¿Por qué se llaman independientes del contexto?](#)

En detalle

Un ejemplo de gramática tipo 1 es el siguiente: $aBc \rightarrow aJc$, donde para transformar B en J, tienen que tener a su izquierda a y a su derecha c, por tanto tienen que tener una parte común, es decir, importa el contexto puesto que dependen de él.

En las GIC, la parte izquierda de las producciones solo pueden tener un símbolo no terminal y para transformar una palabra en otra, el símbolo no terminal que se sustituye no depende de lo que haya a su izquierda o a su derecha. Un ejemplo de producción puede ser: $A \rightarrow Bc$.

Otro ejemplo de producción podría ser $A \rightarrow Bc \mid aBd$. A las producciones de la gramática también se les denomina reglas gramaticales.

 [Notación BNF](#)

Ejemplo

Veamos otro ejemplo mas concreto. Una declaración de variables está formada por el tipo de la variable y las variables, que pueden ser una variable o varias (una lista de identificadores), es decir uno o varios identificadores. Los tipos de las variables pueden ser enteros (int), reales (real) o de tipo caracter (char).

¿Cómo lo especificamos en esta notación?

La declaración la sustituimos por el no terminal D, las variables por el no terminal V y los tipos posibles por el no terminal T. Los tipos posibles serán int, real o char.

 [Resultado](#)

En detalle

En detalle**¿Por qué se llaman independientes del contexto?**

Son las gramáticas tipo 2 de la jerarquía de Chomsky y se denominan independientes del contexto, en contraposición con las de tipo 1, que se denominan dependientes del contexto o sensibles al contexto.

Algunos autores a las gramáticas de tipo 1 las denominan gramáticas de contexto libre, que es justo lo contrario de lo que significan.

Ejemplo**Notación BNF**

Para especificar esta última producción hemos utilizado la notación BNF, en la que el primer símbolo es un no terminal (A) que es el nombre de la estructura o producción, el segundo símbolo es el metasímbolo "→" y este símbolo viene seguido por una cadena de símbolos que pueden ser:

- No terminales (en este caso B, puesto que establecimos un convenio por el que las letras mayúsculas representaban no terminales).
- Símbolos terminales (o símbolos del alfabeto y se representan mediante letras minúsculas, en esta producción c).
- El metasímbolo "|", indicando que hay otra alternativa.

En detalle**Resultado**

- $D \rightarrow TV$.
- $T \rightarrow \text{int} \mid \text{real} \mid \text{char}$.
- $V \rightarrow \text{id}; \mid \text{id}, V$.

Nota1: El punto y coma ";" y la coma "," son terminales, al igual que int o id.

Nota 2: Es importante entender el concepto de **recursividad** que encierra la producción $V \rightarrow \text{id}, V$. La recursividad es una característica esencial de las GIC y nos permite expresar el concepto de un identificador seguido de uno o más identificadores. Este concepto de recursividad no lo hubiéramos podido expresar con una expresión regular.

Árboles de derivación

Ahora necesitamos validar que una sentencia pertenece a esta gramática y para ello necesitamos una herramienta denominada árbol de derivación o árbol de análisis sintáctico.

¿Qué es una derivación?

Se representa por $x \Rightarrow y$ y es la aplicación de una regla de producción $a \rightarrow b$ a una cadena x para convertirla en otra cadena o palabra y .

En el ejemplo anterior, partiendo de $D \rightarrow T V$, si aplicamos la producción $T \rightarrow \text{int}$, **se deriva directamente** $D \rightarrow \text{int } V$.

Es importante entender que decimos directamente porque es lo que se obtiene a partir de la aplicación directa de la producción $T \rightarrow \text{int}$.

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

 [Derivaciones por la izquierda y la derecha](#)
En detalle

¿Qué es un árbol de derivación?

Es una representación utilizada para describir el proceso de derivación de una sentencia, donde se cumple lo siguiente:

- La raíz del árbol es el símbolo inicial de la gramática.
- Los nodos intermedios del árbol son los no terminales de las producciones que se utilizan. Estos nodos intermedios tendrán tantos hijos como elementos tenga el lado derecho de la producción.
- Los nodos hoja serán los terminales.

 [Ejemplo de representación](#)
Gráfico

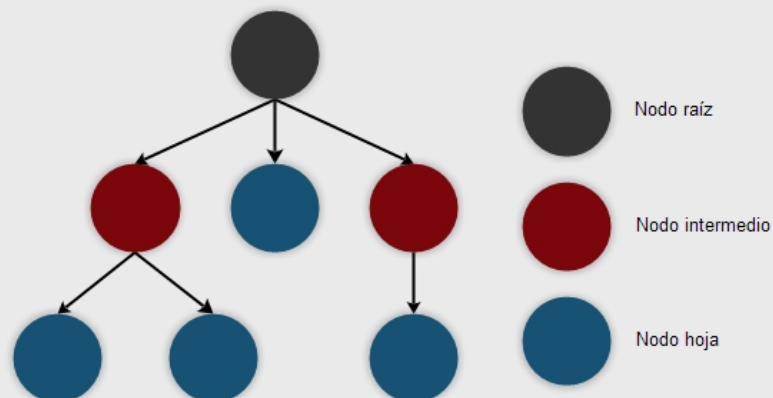
En detalle**Derivaciones por la izquierda y la derecha**

Las derivaciones se pueden realizar empezando por el símbolo no terminal que está más a la izquierda de la cadena, y a estas derivaciones se las denomina **derivación por la izquierda**, o bien se pueden hacer sustituciones en la cadena inicial empezando por el símbolo no terminal que está mas a la derecha y a estas derivaciones se las denomina **derivaciones por la derecha**.

Dependiendo de cuál escojamos se obtiene un resultado u otro.

Gráfico**Ejemplo de representación**

Un ejemplo de representación se ve en la siguiente figura:

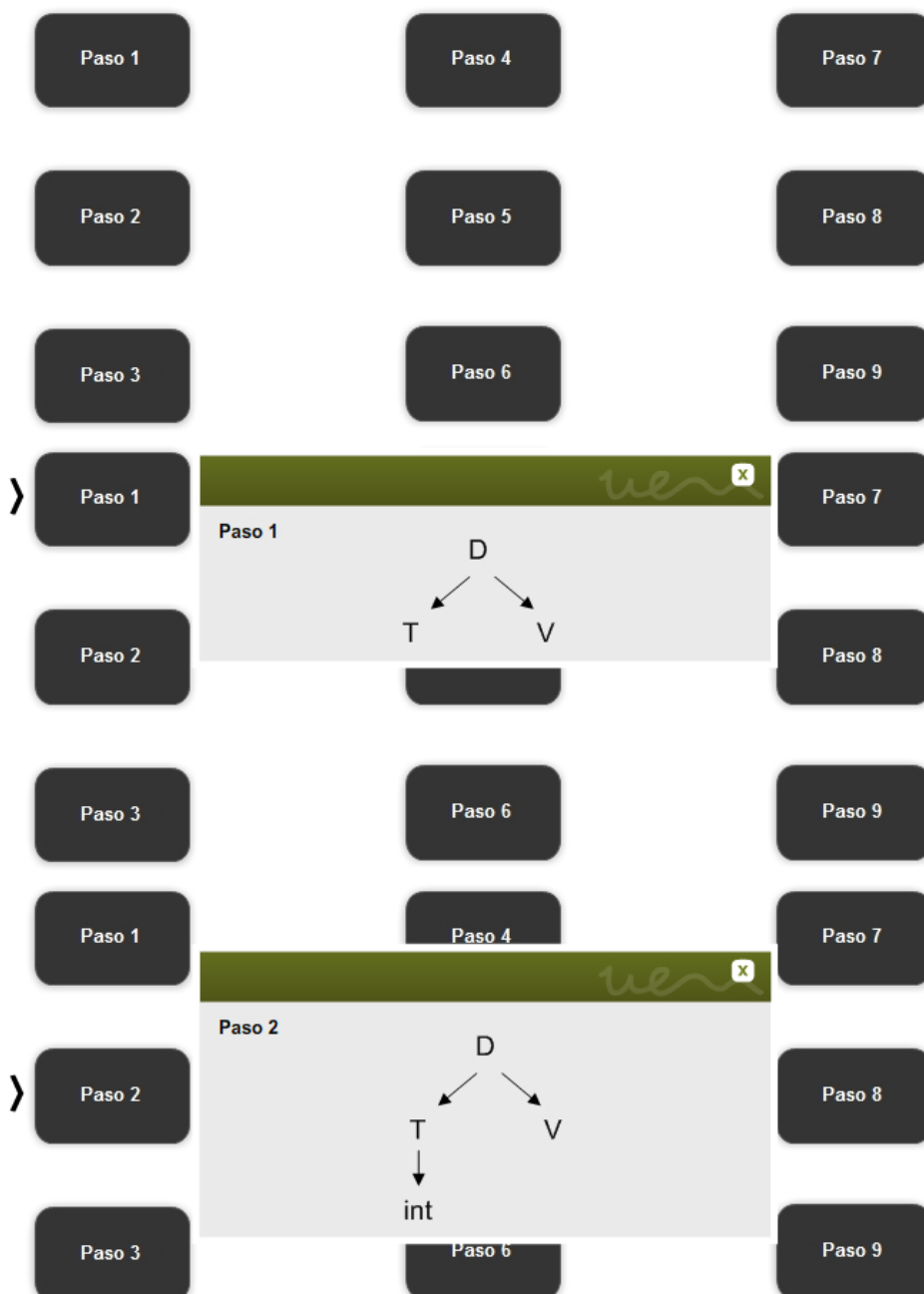


Ejemplo de árbol de derivación

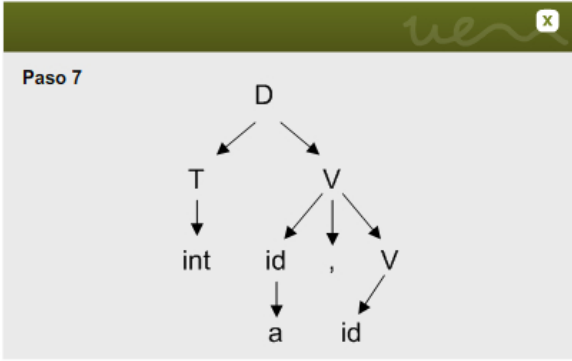
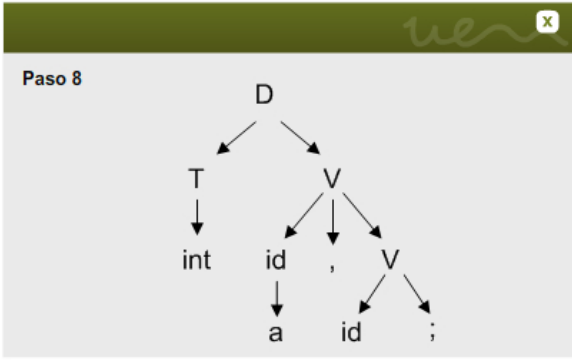
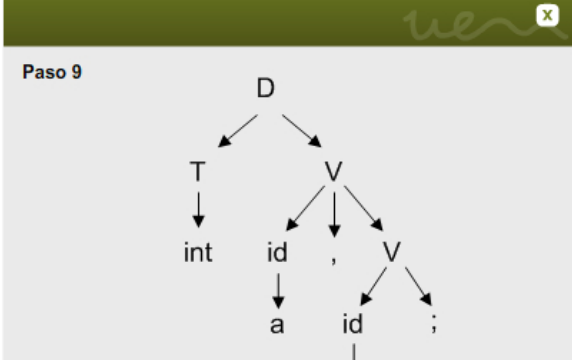
El objetivo principal de un árbol de derivación, también denominado árbol sintáctico, es representar una sentencia del lenguaje y validar de esta forma que pertenece o no a la gramática.

 [Sentencia para la gramática de declaración de variables](#)
En detalle

Para validar esta sentencia vamos a hacer derivaciones por la izquierda, que puedes observar paso a paso en el modelo interactivo o conjuntamente en el documento adjunto:



Paso 1	Paso 4	Paso 7
Paso 2		Paso 8
Paso 3		Paso 9
Paso 1	Paso 4	Paso 7
Paso 2		Paso 8
Paso 3		Paso 9
Paso 1	Paso 4	Paso 7
Paso 2		Paso 8
Paso 3	Paso 6	Paso 9
Paso 1	Paso 4	Paso 7
Paso 2		Paso 8
Paso 3		Paso 9

Paso 1		Paso 7
Paso 2		Paso 8
Paso 3		Paso 9
Paso 1		Paso 7
Paso 2		Paso 8
Paso 3		Paso 9
Paso 1		Paso 7
Paso 2		Paso 8
Paso 3		Paso 9

 [Derivaciones](#)
 [Conclusión](#)
 Documentos En detalle

En detalle

Sentencia para la gramática de declaración de variables

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

- $D \rightarrow TV$.
- $T \rightarrow \text{int} \mid \text{real} \mid \text{char}$.
- $V \rightarrow \text{id}; \mid \text{id}, V$.

En detalle**Conclusión**

Como vemos por los sucesivos pasos, se ha reconocido la sentencia, puesto que ha ido aplicando sustituciones de la gramática a partir del nodo raíz, pasando por los nodos intermedios para llegar a los nodos hoja y hemos obtenido todos los terminales que componen la sentencia que queremos reconocer.

Limpieza de gramáticas

Las gramáticas que nos permiten definir lenguajes de programación están formadas por una gran cantidad de reglas en notación BNF, que pueden contener errores (porque no son necesarias, o 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.
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 las reglas que no generan derivaciones útiles: reglas innecesarias, reglas no generativas y reglas de red denominación.

Reglas innecesarias	Son reglas del tipo $A \rightarrow A$, donde A pertenece a los no terminales.
Reglas no generativas	Son reglas del tipo $A \rightarrow \lambda$, cuando A es distinta de S (el axioma de la gramática).
Reglas de red denominación	Son reglas del tipo $A \rightarrow B$, donde tanto A como B pertenecen a los no terminales.

Orden de limpieza de gramáticas

Una gramática está bien formada si:

- Está limpia.
- Sin reglas no generativas ($A \rightarrow \lambda$).
- Sin reglas de red denominación ($A \rightarrow B$).

 [Gramática](#)
Ejemplo

Conclusiones sobre la gramática
1. No hay reglas innecesarias.
2. Los símbolos Y y V son inaccesibles, puesto que no podemos llegar a ellos mediante derivaciones desde el axioma de la gramática (X).
3. El símbolo Z es un símbolo no terminal superfluo $Z \rightarrow Za$, puesto que cuando se hagan derivaciones siempre aparece el propio símbolo Z.
4. Los símbolos c y d son símbolos terminales superfluos.
5. No hay reglas no generativas.
6. La regla $V \rightarrow Y$ es una regla de red denominación (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 con ambas eliminaciones no se altera el lenguaje generado por la gramática.

 [Limpieza de la gramática](#)  [Objetivo](#)
En detalle Reflexión

Gramática limpia

- Sin reglas innecesarias ($A \rightarrow A$).
- Sin símbolos inaccesibles.
- Sin símbolos superfluos.

Ejemplo**Gramática**

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

- $X \rightarrow Wa \mid Zba \mid \lambda.$
- $Y \rightarrow aVa.$
- $V \rightarrow Y.$
- $W \rightarrow bX.$
- $Z \rightarrow Za.$

En detalle**Limpieza de la gramática**

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

- $X \rightarrow Wa \mid Zba \mid \lambda.$
- $Y \rightarrow aVa.$
- $V \rightarrow Y.$
- $W \rightarrow bX.$
- $Z \rightarrow Za.$

Obteniéndose finalmente:

- $X \rightarrow Wa \mid \lambda.$
- $W \rightarrow bX.$

Reflexión**Objetivo**

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

Ambigüedades

Una gramática es ambigua si permite construir dos o más árboles de derivación distintos para la misma cadena. Es importante saber reconocer en una gramática los dos tipos fundamentales de ambigüedades que nos interesa eliminar:

1. Recursividad por la izquierda.
2. Varias alternativas de una sentencia comienzan igual.

Eliminar la recursividad a izquierdas

Partimos de la siguiente producción $A \rightarrow A\alpha \mid \beta$, donde tanto α como β representan conjuntos de terminales y no terminales.



Eliminar varias alternativas que comienzan igual

A esto se le llama **factorizar** y se resuelve de la siguiente forma: puesto que parte de alternativas de una producción que comienzan igual, tenemos que determinar la parte común más larga entre estas alternativas y a partir de ahí se sustituye por un no terminal que actuará de discriminante. Si tenemos $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \theta$, donde θ representa otras alternativas que no comienzan con α , se hace lo siguiente:

- $A \rightarrow \alpha A' \mid \theta$.
- $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$.

Ejemplo:

- $V \rightarrow id; \mid id, V$ donde la parte común, α , es igual a id .

Por tanto, aplicando la regla anterior:

- $V \rightarrow id A'$.
- $A' \rightarrow ; \mid , \mid V$.

En detalle

Regla a aplicar

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

La regla a aplicar es la siguiente:

- $A \rightarrow \beta A'$.
- $A' \rightarrow \alpha A \mid \lambda$.

Ejemplo**Resultado**

Ejemplo:

- $V \rightarrow V, T \mid T$, donde $\alpha = , T$ y $\beta = T$.

Por tanto el resultado sería:

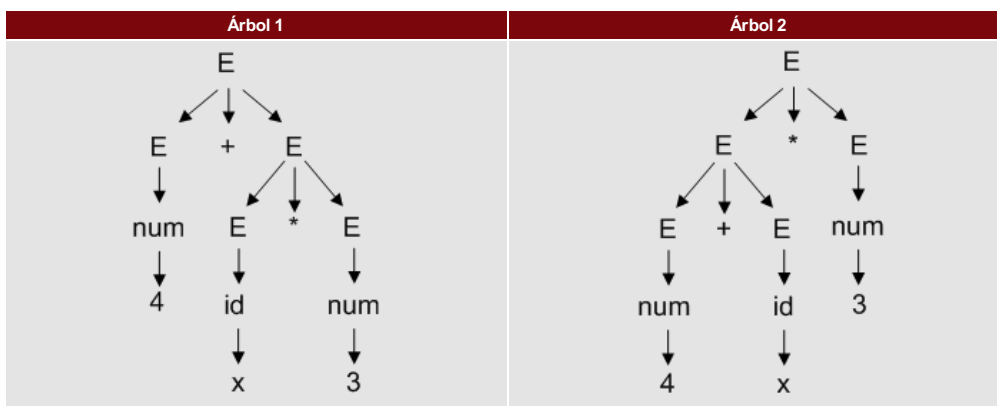
- $V \rightarrow T V'$.
- $V' \rightarrow , TV \mid \lambda$.

Gramáticas ambiguas

Una gramática es ambigua cuando para reconocer una sentencia podemos utilizar más de un árbol sintáctico. Es interesante destacar que esta ambigüedad se produce si reconoce al menos una sentencia ambigua.

Veamos el caso de una gramática diseñada para reconocer operaciones aritméticas mediante expresiones, donde estas expresiones se sustituyen por el no terminal E y donde para una sentencia dada (en este caso: $4 + x * 3$) obtenemos más de un árbol de derivación:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{num} \mid \text{id}.$$



 [Ambigüedad de la gramática](#)
En detalle

En detalle

Ambigüedad de la gramática

Usando la lógica aprendida en matemáticas, el árbol válido es el árbol 1, puesto que tiene en cuenta la precedencia del signo de multiplicar sobre el de suma, por lo que hace la multiplicación $x*3$ y después le suma el 4.

En cualquier caso, con la gramática que tenemos esto no está resuelto por lo que el árbol 2 es tan válido como el árbol 1, reflejando que la gramática es ambigua.

Es importante destacar que no todas las ambigüedades se resuelven sintácticamente y debe ser el analizador semántico mediante controles semánticos el que las resuelva.

Precedencia y asociatividad

Como hemos visto en la anterior pantalla, el problema de esa gramática era de precedencia de operadores, es decir, qué operador debe ser evaluado antes para que la operación tenga sentido desde el punto de vista matemático.

Hay varias soluciones a este problema de prioridad a la hora de seleccionar las operaciones y, por tanto, los operadores.

Cascada de precedencia (Louden, 2004)	Se agrupan los operadores en diferentes niveles de precedencia. Al colocarlos en la gramática más cerca del axioma de la gramática (en definitiva más cerca de la raíz del árbol de análisis) obtienen una precedencia menor.
De forma explícita en el analizador sintáctico	Los generadores de análisis sintáctico tienen directivas para definir la precedencia de los operadores.

Como nos indica Luengo Díez et al (2005), el orden de precedencia de mayor a menor precedencia:

1. Los paréntesis ().
2. Menos unario (-).
3. Potenciación (^).
4. Multiplicación (*) y división (/).
5. Suma (+) y resta (-).

Por asociatividad entendemos la forma en que los operadores agrupan a los operandos para hacer las operaciones. Es útil cuando los operadores tienen la misma precedencia y hay que seleccionar que operación se hace primero.

 [Asociatividad por derecha e izquierda](#)
En detalle

Utilizando de una forma apropiada estas dos propiedades, precedencia y asociatividad, podemos conseguir que una gramática ambigua, como la que hemos utilizado en el ejemplo (gramática de operadores) se convierta en no ambigua.

En detalle**Asociatividad por derecha e izquierda**

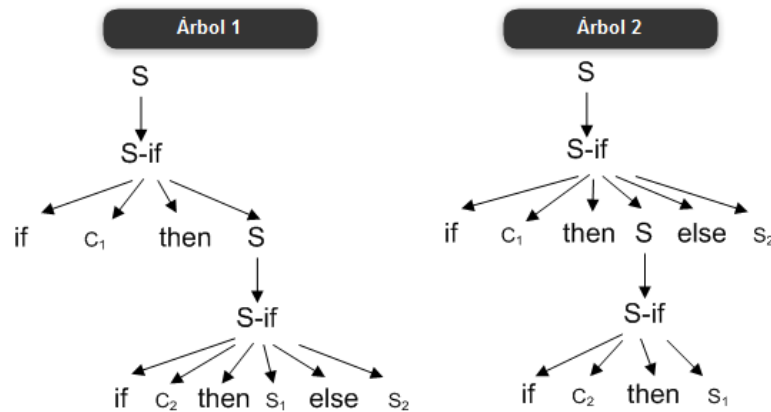
Se habla de asociatividad por la izquierda para los operadores suma (+), resta (-), multiplicación (*) y división (/), mientras que para la potenciación (^) o para el signo de igualdad (=) la asociatividad es por la derecha.

En el ejemplo $3 + 2 - 5$, en el que los dos operadores tienen la misma precedencia, primero se hace la suma puesto que es asociativa por la izquierda ($3+2$) y posteriormente se hace la resta ($5-5$).

El problema del *else* ambiguo

Veamos la siguiente gramática, utilizada por el lenguaje de programación C y que produce una ambigüedad a la hora de decidir a que *if* está asociado un *else* determinado:

Supongamos la siguiente sentencia de entrada, a la que vamos a añadir subíndices a los no terminales para que sea más fácil identificarlos: *if C₁ then if C₂ then S₁ else S₂*.



En el árbol 1 el *else* lo hemos asociado con el *if* más cercano, mientras que en el árbol 2 lo hemos asociado con el primer *if*. Dependiendo del lenguaje de programación, esto se resuelve de una manera o de la otra, aunque la mayoría de los lenguajes utilizan el primer árbol, donde el *else* se empareja con el *if* más cercano.

 [¿Cómo lo resolvemos?](#)
En detalle

Ambigüedad de la gramática

sentencia \rightarrow sentencia-*if* | otras-sentencias.

sentencia-*if* \rightarrow **if** condición **then** sentencia.

| **if** condición **then** sentencia **else** sentencia.

En negrita están los terminales de la gramática, y el resto son no terminales, donde condición será una expresión booleana (valdrá 0 o 1) y sentencia será al menos una sentencia seguida de cero o más sentencias. Utilizando las convenciones que hemos adoptado la gramática quedaría:

$S \rightarrow S\text{-if} \mid O\text{-}S.$

$S\text{-if} \rightarrow \mathbf{if} C \mathbf{then} S.$

| $\mathbf{if} C \mathbf{then} S \mathbf{else} S.$

En detalle**¿Cómo lo resolvemos?**

Se pueden utilizar llaves para agrupar los *if* con sus *else* correspondiente o bien se pueden utilizar palabras reservadas.

La utilización de palabras reservadas la hace Ada, añadiendo **end if**, lo que permite saber dónde acaba y dónde empieza una sentencia *if*, quedando la gramática de la siguiente forma:

sentencia-*if* \rightarrow **if** condición **then** sentencia **end if**.

| **if** condición **then** sentencia **else** sentencia **end if**.

Resumen

En este tema hemos aprendido con algunos ejemplos a **definir una GIC**, distinguiéndola de otros tipos de gramáticas, además de a reconocer cuándo esta gramática no está limpia y en qué orden debería limpiarse o si contiene ambigüedades.

Una vez sabemos reconocer estos problemas, también aprendemos a corregir problemas de **recursividad a izquierdas** o de múltiples alternativas que comienzan igual en una regla o producción (factorización).

En el caso de las gramáticas "sucias", hemos aprendido a **reconocer los símbolos superfluos**, tanto terminales como no terminales e inaccesibles, así como las **reglas innecesarias, no generativas o de red denominación**.

Más adelante hemos aprendido a tratar con gramáticas ambiguas y a detectar esa ambigüedad al obtener más de un árbol de derivación para una sentencia. También vimos conceptos como la derivación por la izquierda y por la derecha.

Por otro lado, hemos entendido los **conceptos de precedencia y asociatividad**, y cómo pueden resolver problemas de ambigüedad en las gramáticas de operadores.

Para finalizar hemos visto un ejemplo de un problema concreto que tienen algunos lenguajes antiguos, como es el del *e/se* ambiguo y cómo se puede corregir.