# Gramáticas BNF y EBNF para Definir lenguajes de Programación

Por Kirstein Gätjens S.

## Contexto para estudiar estas gramáticas

La definición de un lenguaje de programación contiene como componente formal para su sintaxis el desarrollo de una gramática. Existen varios modelos para efectuar estas definiciones, de los cuáles el más conocido tal vez es el de BNF creado por Backus y por Naur a inicios de los 60´s de la centuria anterior. Para po-der comprender de una mejor forma como definir un lenguaje es necesario estudiar brevemente las generalidades de una gramática.

## Gramáticas Libres de Contexto

Una gramática libre de contexto es la herramienta que se utiliza para definir lenguajes libres de contexto, es decir en palabras informales, lenguajes que no tienen partes que contengan significados diferentes en dependencia con la ubicación o uso que se les de a dichas partes. La sintaxis típica de un lenguaje de programación entra dentro de esta categoría, junto con una cantidad infinita de lenguajes, algunos muy simples y otros más complejos. Los lenguajes naturales por el contrario están fuera de esta categoría junto con otra cantidad infinita de lenguajes.

El proceso de generación que siguen las gramáticas es la forma de corroborar si un texto pertenece o no al lenguaje que define la gramática particular de dicho lenguaje. A ese texto se le suele llamar hilera, programa, frase, oración, etc. dependiendo del contexto en que se esté trabajando. El proceso de generación se detalla en la siguiente sección.

Para que una gramática sea considerada libre de contexto debe cumplir ciertas normas mínimas que se discuten en la sección de errores en las gramáticas, pero primero se debe estudiar alguna notación que ayude al lector a tener un modelo en común sobre el cual trabajar. El sentido común dicta que esa notación sea BNF.

## Backus-Naur Form

Como se puede apreciar las siglas de BNF están resumiendo el nombre de los autores del modelo que se estudia en esta sección.

Como toda gramática, una gramática de este tipo contiene cuatro partes fundamentales:

a) Un conjunto finito de símbolos conocido como Terminales. Son los símbolos pertenecientes al lenguaje que genera la gramática, estos pueden ser palabras de lenguaje, símbolos, garabatos, etc. Cualquier unidad léxica que sea parte del lenguaje.

b) Un conjunto finito de símbolos intermedios conocidos como No terminales que se utilizan en el proceso de generación de la gramática. A estos también se les suele conocer como variables o símbolos auxiliares. La nomenclatura típica para diferenciar los no terminales de los terminales es encerrar los primeros entre brackets, por ejemplo **<T>** es un No Terminal y T es un terminal.

c) Un No Terminal especial único con el que se comienza el proceso de generación. Debe de ser un símbolo perteneciente al conjunto de No Terminales. No tiene un nombre estándar pero suele llamarse **<S>** o **<Start>** pero cualquier nombre es válido.

d) Un conjunto finito de reglas conocido como producciones de la forma: **<A> ::= secuencia**

Donde el símbolo **::=** se lee como "está definido por" o "genera a", el lado izquierdo de la producción **debe** ser un No Terminal y el lado derecho **debe** ser una forma sentencial, es decir una secuencia finita de símbolos terminales y No terminales que mantienen un orden particular. Este lado derecho puede ser una secuencia vacía en cuyo caso se le conoce como **épsilon** y se utiliza el símbolo ε para referenciarlo. Para las formas sentenciales se suele utilizar letras griegas como nomenclatura estándar.

Una vez que se tiene claramente definidas las cuatro partes que contiene una gramática, este define un lenguaje. Este len-guaje es un conjunto (probablemente infinito) de textos. Para po-der saber si un texto particular pertenece o no a este conjunto hay que seguir el proceso de generación que consiste en lo siguiente:

Se tiene como forma sentencial inicial al no terminal inicial de la gramática. Con el objetivo de tratar de llegar a producir el texto del que se desea probar su pertenencia al lenguaje, se debe seleccionar algún no terminal de la forma sentencial actual y cambiarlo por alguna de las producciones de la gramática que lo definen a él. Este proceso se debe seguir iterando hasta que se genere el texto deseado o se compruebe la imposibilidad de hacerlo.

Al escribir la gramática de un lenguaje en BNF se puede llegar a notar que es un poco largo y tedioso el describir algunas de las reglas. Existe una forma para simplificar el trabajo que se analiza en una sección posterior.

## Errores en las gramáticas

Las reglas para desarrollar una gramática son muy sencillas como se logra apreciar en la sección anterior, sin embargo existen ciertas normas adicionales que al no cumplirlas hacen que la gramática contenga errores y no pueda considerarse libre de contexto. Las principales de estas reglas son:

* Todo no terminal debe estar definido en la gramática, es decir debe existir al menos una posibilidad (producción) de que cada no terminal que aparezca en la gramática genere alguna forma sentencial. A este error se le llama gramática incompleta.
* Todo no terminal debe ser alcanzable a partir del proceso de generación desde el no terminal inicial. No tiene sentido tener producciones que definan terminales que nunca llegarán a utilizarse en el proceso de generación. A esto se le suele llamar tener no terminales inalcanzables en la gramática.
* Debe ser posible cambiar todo no terminal por una forma sentencial que no necesariamente lleve a producir ese mismo no terminal. En ese caso sería imposible que una vez que aparece ese no terminal en una forma sentencial, se logre desaparecer, por lo que no se podría llegar a tener solo terminales para generar un texto. Se dice que una gramática con ese error es una gramática que no aterriza.

Existen algunos errores adicionales, que hacen que la gramática aunque sea libre de contexto no sea útil para un compilador típico, entre esos errores tenemos necesitar factori-zación o tener recursividad izquierda sea directa o indirectamente, pero ese tipo de errores se dejarán para otra ocasión.

# Alternativas que simplifican el trabajo: EBNF

Algún tiempo después de que comenzó a extenderse el modelo BNF se le agregaron varias herramientas que hacían más fácil el desarrollo de gramáticas en dicho modelo a este nuevo modelo que las incluye se le suele llamar Extended Backus-Naur Form o simplemente EBNF. Sin embargo es importante hacer notar que las herramientas adicionales que aporta BNF no le dan un mayor poder computacional, es decir cualquier lenguaje que puede ser generado por una gramática EBNF tiene a su vez una gramática BNF que lo genera. Esto es demostrado en la sección siguiente.

Las herramientas adicionales a BNF que facilitan el trabajo en el desarrollo de gramáticas y que convierten el modelo en EBNF son las siguientes:

* Se puede utilizar el | como separador de alternativas para una forma sentencial, por ejemplo si se tiene la producción:

**<T> ::= 00 | 10 | 01** se puede decir que el no terminal **<T>** genera los textos **00**, **10** o **01**. pero solo uno de ellos a la vez.

Esta herramienta permite añadir una característica importante a EBNF, solo puede haber una única producción para definir cada no terminal de la gramática, de forma tal que la cardinalidad de los conjuntos de producciones y no terminales sea siempre la misma.

* Los corchetes que encierran una forma sentencial indican que dicha forma sentencial puede o no estar. En otras palabras es opcional. Por ejemplo la forma sentencial: **0** [ **1** ] **0** puede producir solo dos textos: **00** y **010**

* El uso del operador * como operador posfijo indica que se pueden tener de cero a N repeticiones del elemento sobre el que trabaja. Por ejemplo la forma sentencial: **0 1* 0** puede producir los textos: **00**, **010, 0110**, **01110**, **011110**, **0111110**, etc. Se puede utilizar las llaves como un sinónimo de esta herramienta, el ejemplo anterior se podría escribir **0 { 1 } 0** y generaría el mismo conjunto de textos.

* El operador $^+$ es similar al operador anterior solo que no permite que la forma sentencial que le precede no aparezca del todo, por ejemplo si se tiene **0 1$^+$ 0** se generarán los textos **010**, **0110**, **01110**, **011110**, etc. pero no se generará el texto **00**

* El uso de paréntesis para agrupar formas sentenciales. Se utiliza para poder alterar la precedencia de las otras herramientas (que es la usual) y mezclarlas en una sola forma sentencial.

* Como hay varios símbolos que se utilizan como parte del modelo, si se desea que alguno de ellos sea un terminal de la gramática debe encerrarse entre comillas simples.

# De EBNF a BNF y viceversa

La forma sencilla de demostrar que BNF y EBNF tienen igual poder computacional es mostrar dos algoritmos de conversión. Uno que reciba como entrada una gramática BNF y que genere una EBNF equivalente y otro que funcione en sentido contrario.

Si se tiene una gramática en BNF es muy sencillo convertirla a EBNF, de hecho la única característica que no satisface una gramática BNF para ser EBNF es la posibilidad de múltiples definiciones de un no terminal mediante diferentes producciones, pero es sencillo convertirlas todas en una sola producción que tiene como lado derecho los lados derechos de las producciones originales separados por el operador |. Por ejemplo si se tienen las producciones:

<A> ::= $\alpha$ <A> ::= $\delta$ y <A> ::= $\beta$

se pueden cambiar por una sola equivalente en EBNF como sigue:

<A> ::= $\alpha$ | $\delta$ | $\beta$

Sin embargo el camino contrario no es tan simple, pero se puede realizar sin demasiado esfuerzo. Basta con encontrar formas equivalentes en BNF para cada una de las herramientas adicionales que provee EBNF. He aquí una descripción de dichas formas.

* Si se tiene una forma sentencial dentro de una secuencia opcional, se puede sustituir por reglas de la siguiente forma:

<A> ::= $\alpha$ [ $\delta$ ] $\beta$

se sustituiría por:

<A> ::= $\alpha$ <B> $\beta$

<B> ::= $\delta$ y <B> ::= $\varepsilon$

* Si se tiene una separación por un pipe | se sustituiría así:

<A> ::= $\alpha$ | $\beta$

sería substituido por:

<A> ::= $\alpha$ y <A> ::= $\beta$

* Si se tienen unas llaves o un asterisco, se trabajaría:

<A> ::= $\alpha$ { $\delta$ } $\beta$ o bien <A> ::= $\alpha$ $\delta$* $\beta$

se sustituiría por:

<A> ::= $\alpha$ <B> $\beta$

<B> ::= $\delta$ <B> y <B> ::= $\varepsilon$

* Si se desea sustituir el $^+$, es similar a sustituir las llaves solo que se conserva el $\delta$ en la producción original:

<A> ::= $\alpha$ $\delta^+$ $\beta$

se sustituiría por:

<A> ::= $\alpha$ $\delta$ <B> $\beta$

<B> ::= $\delta$ <B>

<B> ::= $\varepsilon$

* Los paréntesis redondos que se utiliza para agrupar van dando indicaciones de cuáles de las reglas anteriores se deben de utilizar, para mantener la equivalencia entre las reglas. Una vez que ya no son indispensables los paréntesis se pueden eliminar.

* Como último paso, si lo único que resta de EBNF son las comillas simples alrededor de algunos terminales, estas pueden desaparecer, recordando que debe ser lo último en eliminarse.

# Anotaciones finales

El escribir la gramática que genere un lenguaje es similar al arte de la programación, no se puede enseñar una “receta de cocina” que permita escribir la gramática, pero puede aprenderse con mucha práctica y revisión de ejercicios ilustrativos.

Sin embargo el manejo de las herramientas BNF y EBNF y las transformaciones entre ellas si se puede aprender muy fácil y algorítmicamente.