# Contenido

## 3.2 GRAMÁTICAS LIBRES DE CONTEXTO

Una gramática libre de contexto es una especificación para la estructura sintáctica de un lenguaje de programación. Una especificación así es muy similar a la especificación de la estructura léxica de un lenguaje utilizando expresiones regulares, excepto que una gramática libre de contexto involucra reglas de recursividad. Como ejemplo de ejecución utilizaremos expresiones aritméticas simples de enteros con operaciones de suma, resta y multiplicación. Estas expresiones se pueden dar mediante la gramática siguiente:

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{\texttt{número}}$$
$$op \rightarrow + \mid - \mid *$$

### 3.2.1 Comparación respecto a la notación de una expresión regular

Consideremos cómo se compara la gramática libre de contexto de muestra anterior con las reglas de la expresión regular dada por ***número*** del capítulo anterior:

```
número = dígito dígito*
dígito = 0|1|2|3|4|5|6|7|8|9
```

En las reglas de la expresión regular básica tenemos tres operaciones: selección (dada por el metasímbolo de la barra vertical), concatenación (dada sin un metasímbolo) y repetición (dada por el metasímbolo del asterisco). También empleamos el signo de igualdad para representar la definición de un nombre para una expresión regular, y escribimos el nombre en itálicas para distinguirlo de una secuencia de caracteres reales.

Las reglas gramaticales utilizan notaciones semejantes. Los nombres se escriben en cursivas o itálicas (pero ahora con una fuente diferente, de modo que podamos distinguirlas de los nombres para las expresiones regulares). La barra vertical todavía aparece como el metasímbolo para selección. La concatenación también se utiliza como operación estándar. Sin embargo, no hay ningún metasímbolo para la repetición (como el * de las expresiones regulares), un punto al cual regresaremos en breve. Una diferencia adicional en la notación es que ahora utilizamos el símbolo de la flecha → en lugar del de igualdad para expresar las definiciones de los nombres. Esto se debe a que ahora los nombres no pueden simplemente ser reemplazados por sus definiciones, porque está implicado un proceso de definición más complejo, como resultado de la naturaleza recursiva de las definiciones.[1] En nuestro ejemplo, la regla para *exp* es recursiva, en el sentido que el nombre *exp* aparece a la derecha de la flecha.

Advierta también que las reglas gramaticales utilizan expresiones regulares como componentes. En las reglas para *exp* y *op* se tienen en realidad seis expresiones regulares

1. Pero vea más adelante en el capítulo el análisis sobre reglas gramaticales como ecuaciones.

representando tokens en el lenguaje. Cinco de éstas son tokens de carácter simple: `+`, `-`, `*`, `(` y `)`. Una es el nombre ***número***, el nombre de un token representando secuencias de dígitos.

De manera similar a la de este ejemplo, las reglas gramaticales se usaron por primera vez en la descripción del lenguaje Algol60. La notación fue desarrollada por John Backus y adaptada por Peter Naur para el informe Algol60. De este modo, generalmente se dice que las reglas gramaticales en esta forma están en la **forma Backus-Naur**, o **BNF** (**Backus-Naur Form**).

## 3.2.2 Especificación de las reglas de una gramática libre de contexto

Como las expresiones regulares, las reglas gramaticales están definidas sobre un alfabeto, o conjunto de símbolos. En el caso de expresiones regulares, estos símbolos por lo regular son caracteres. En el caso de reglas gramaticales, los símbolos son generalmente tokens que representan cadenas de caracteres. En el último capítulo definimos los tokens en un analizador léxico utilizando un tipo enumerado en C. En este capítulo, para evitar entrar en detalles de la manera en que los tokens se representan en un lenguaje de implementación específica (como C), emplearemos las expresiones regulares en sí mismas para representar los tokens. En el caso en que un token sea un símbolo fijo, como en la palabra reservada `while` o los símbolos especiales como `+` o `:=`, escribiremos la cadena en sí en la fuente de código que se utilizó en el capítulo 2. En el caso de tokens tales como identificadores y números, que representan más de una cadena, utilizaremos la fuente de código en itálicas, justo como si el token fuera un nombre para una expresión regular (lo que por lo regular representa). Por ejemplo, representaremos el alfabeto de tokens para el lenguaje TINY como el conjunto

```
{if,then,else,end,repeat,until,read,write,
identificador,número,+,-,*,/,=,<,(,),;,:=}
```

en lugar del conjunto de tokens (como se definieron en el analizador léxico de TINY):

```
{IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,ID,NUM,
PLUS,MINUS,TIMES,OVER,EQ,LT,LPAREN,RPAREN,SEMI,ASSIGN}
```

Dado un alfabeto, una **regla gramatical libre de contexto en BNF** se compone de una cadena de símbolos. El primer símbolo es un nombre para una estructura. El segundo símbolo es el metasímbolo "→". Este símbolo está seguido por una cadena de símbolos, cada uno de los cuales es un símbolo del alfabeto, un nombre para una estructura o el metasímbolo "|".

En términos informales, una regla gramatical en BNF se interpreta como sigue. La regla define la estructura cuyo nombre está a la izquierda de la flecha. Se define la estructura de manera que incluya una de las selecciones en el lado derecho separada por las barras verticales. Las secuencias de símbolos y nombres de estructura dentro de cada selección definen el diseño de la estructura. Por ejemplo, considere las reglas gramaticales de nuestro ejemplo anterior:

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{\textit{número}}$$
$$op \rightarrow + \mid - \mid *$$

La primera regla define una estructura de expresión (con nombre *exp*) compuesta por una expresión seguida por un operador y otra expresión, por una expresión dentro de paréntesis, o bien por un número. La segunda define un operador (con nombre *op*) compuesto de uno de los símbolos `+`, `-` o `*`.

Los metasímbolos y convenciones que usamos aquí son semejantes a los de uso generalizado, pero es necesario advertir que no hay un estándar universal para estas convenciones. En realidad, las alternativas comunes para el metasímbolo de la flecha "→" incluyen "=" (el

signo de igualdad), ":" (el signo de dos puntos) y "::=" (signo de dos puntos doble y el de igualdad). En archivos de texto normales también es necesario hallar un reemplazo para el uso de las itálicas. Esto se hace frecuentemente encerrando los nombres de estructura con "picoparéntesis" < . . . > y escribiendo los nombres de token en itálicas con letras mayúsculas. De este modo, con diferentes convenciones, las reglas gramaticales anteriores podrían aparecer como

```
<exp> ::= <exp> <op> <exp> | ( <exp> ) | NÚMERO
<op> ::= + | - | *
```

Cada autor también tendrá otras variaciones en estas notaciones. Varias de las más importantes (algunas de las cuales utilizaremos ocasionalmente) se analizarán más adelante en esta sección. Sin embargo, vale la pena analizar de inmediato dos pequeñas cuestiones adicionales sobre la notación.

En ocasiones, aunque los paréntesis son útiles para reasignar la precedencia en las expresiones regulares, conviene incluir paréntesis en los metasímbolos de la notación BNF. Por ejemplo, se puede volver a escribir las reglas gramaticales anteriores como una sola regla gramatical de la manera siguiente:

$$exp \rightarrow exp\ (\text{“+”} \mid \text{“-”} \mid \text{“*”})\ exp \mid \text{“(“}\ exp\ \text{“)”} \mid \textbf{\textit{número}}$$

En esta regla los paréntesis son necesarios para agrupar las opciones de los operadores entre las expresiones en el lado derecho, puesto que la concatenación tiene precedencia sobre la selección (como en las expresiones regulares). De este modo, la regla siguiente tendría un significado diferente (e incorrecto):

$$exp \rightarrow exp\ \text{“+”} \mid \text{“-”} \mid \text{“*”}\ exp \mid \text{“(“}\ exp\ \text{“)”} \mid \textbf{\textit{número}}$$

Advierta también que, cuando se incluyen los paréntesis como metasímbolo, es necesario distinguir los tokens de paréntesis de los metasímbolos, lo que hicimos poniéndolos entre comillas, como hacíamos en el caso de expresiones regulares. (Para mantener la consistencia también encerramos los símbolos de operador entre comillas.)

Los paréntesis no son absolutamente necesarios como metasímbolos en BNF, ya que siempre es posible separar las partes entre paréntesis en una nueva regla gramatical. De hecho, la operación de selección que da el metasímbolo de la barra vertical tampoco es necesaria en reglas gramaticales, si permitimos que el mismo nombre aparezca cualquier número de veces a la izquierda de la flecha. Por ejemplo, nuestra gramática de expresión simple podría escribirse como se aprecia a continuación:

$$\begin{aligned} exp &\rightarrow exp\ op\ exp \\ exp &\rightarrow \text{(}\ exp\ \text{)} \\ exp &\rightarrow \textbf{\textit{número}} \\ op &\rightarrow \text{+} \\ op &\rightarrow \text{-} \\ op &\rightarrow \text{*} \end{aligned}$$

Sin embargo, por lo regular describiremos reglas gramaticales de manera que todas las selecciones para cada estructura estén enumeradas en una sola regla, y cada nombre de estructura aparezca sólo una vez a la izquierda de la flecha.

En ocasiones, por simplicidad, daremos ejemplos de reglas gramaticales en una notación abreviada. En estos casos utilizaremos letras en mayúsculas para nombres de estructura y letras en minúsculas para símbolos de token individuales (que con frecuencia son sólo caracteres simples). De este modo, nuestra gramática de expresión simple podría escribirse en esta forma abreviada de la manera que sigue

$$E \rightarrow E\ O\ E \mid (\ E\ ) \mid \mathtt{n}$$
$$O \rightarrow \mathtt{+} \mid \mathtt{-} \mid \mathtt{*}$$

En ocasiones también simplificaremos la notación cuando estemos utilizando solamente caracteres como tokens y los estemos escribiendo sin utilizar una fuente de código:

$$E \rightarrow E\ O\ E \mid (\ E\ ) \mid a$$
$$O \rightarrow + \mid - \mid *$$

## 3.2.3 Derivaciones y el lenguaje definido por una gramática

Ahora volvamos a la descripción de cómo las reglas gramaticales determinan un "lenguaje", o conjunto de cadenas legales de tokens.

Las reglas gramaticales libres de contexto determinan el conjunto de cadenas sintácticamente legales de símbolos de token para las estructuras definidas por las reglas. Por ejemplo, la expresión aritmética

```
(34-3)*42
```

corresponde a la cadena legal de siete tokens

```
( número - número ) * número
```

donde los tokens de **`número`** tienen sus estructuras determinadas por el analizador léxico y la cadena misma es legalmente una expresión porque cada parte corresponde a selecciones determinadas por las reglas gramaticales

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \mathtt{número}$$
$$op \rightarrow \mathtt{+} \mid \mathtt{-} \mid \mathtt{*}$$

Por otra parte, la cadena

```
(34-3*42
```

no es una expresión legal, porque se tiene un paréntesis izquierdo que no tiene su correspondiente paréntesis derecho y la segunda selección en la regla gramatical para una expresión *exp* requiere que los paréntesis se generen en pares.

Las reglas gramaticales determinan las cadenas legales de símbolos de token por medio de derivaciones. Una **derivación** es una secuencia de reemplazos de nombres de estructura por selecciones en los lados derechos de las reglas gramaticales. Una derivación comienza con un nombre de estructura simple y termina con una cadena de símbolos de token. En cada etapa de una derivación se hace un reemplazo simple utilizando una selección de una regla gramatical.

Como ejemplo, la figura 3.1 proporciona una derivación para la expresión `(34-3)*42` utilizando las reglas gramaticales como se dan en nuestra expresión gramatical simple. En cada paso se proporciona a la derecha la selección de la regla gramatical utilizada para el reemplazo. (También numeramos los pasos para una referencia posterior.)

Figura 3.1
Una derivación para la expresión aritmética `(34-3)*42`

| | | |
|---|---|---|
| (1) | *exp* ⇒ *exp op exp* | [*exp* → *exp op exp*] |
| (2) | ⇒ *exp op* **`número`** | [*exp* → **`número`**] |
| (3) | ⇒ *exp* * **`número`** | [*op* → * ] |
| (4) | ⇒ ( *exp* ) * **`número`** | [*exp* → ( *exp* )] |
| (5) | ⇒ ( *exp op exp* ) * **`número`** | [*exp* → *exp op exp*] |
| (6) | ⇒ (*exp op* **`número`**) * **`número`** | [*exp* → **`número`**] |
| (7) | ⇒ (*exp* - **`número`**) * **`número`** | [*op* → - ] |
| (8) | ⇒ (**`número`** - **`número`**)*`número` | [*exp* → **`número`**] |

Advierta que los pasos de derivación utilizan una flecha diferente al metasímbolo de flecha que se emplea en las reglas gramaticales. Esto se debe a que existe una diferencia entre un paso de derivación y una regla gramatical: las reglas gramaticales **definen**, mientras que los pasos de derivación **construyen** mediante reemplazo. En el primer paso de la figura 3.1, la *exp* simple se reemplaza por la cadena *exp op exp* del lado derecho de la regla *exp* → *exp op exp* (la primera selección en la BNF para *exp*). En el segundo paso, la *exp* del extremo derecho en la cadena *exp op exp* se reemplaza por el símbolo **`número`** del lado derecho de la selección *exp* → **`número`** para obtener la cadena *exp op* **`número`**. En ese paso el *op* (operador) se reemplaza por el símbolo * del lado derecho de la regla *op* → * (la tercera de las selecciones en la BNF para *op*) para obtener la cadena *exp* * **`número`**. Y así sucesivamente.

El conjunto de todas las cadenas de símbolos de token obtenido por derivaciones del símbolo *exp* es el **lenguaje definido por la gramática** de expresiones. Este lenguaje contiene todas las expresiones sintácticamente legales. Podemos escribir esto de manera simbólica como

$$L(G) = \{ s \mid exp \Rightarrow^* s \}$$

donde $G$ representa la gramática de expresión, $s$ representa una cadena arbitraria de símbolos de token (en ocasiones denominada **sentencia**), y los símbolos $\Rightarrow^*$ representan una derivación compuesta de una secuencia de reemplazos como se describieron anteriormente. (El asterisco se utiliza para indicar una secuencia de pasos, así como para indicar repetición en expresiones regulares.) Las reglas gramaticales en ocasiones se conocen como **producciones** porque "producen" las cadenas en $L(G)$ mediante derivaciones.

Cada nombre de estructura en una gramática define su propio lenguaje de cadenas sintácticamente legales de tokens. Por ejemplo, el lenguaje definido por *op* en nuestra gramática de expresión simple define el lenguaje {+, -, *} compuesto sólo de tres símbolos. Por lo regular estamos más interesados en el lenguaje definido por la estructura más general en una gramática. La gramática para un lenguaje de programación a menudo define una estructura denominada *programa*, y el lenguaje de esta estructura es el conjunto de todos los programas sintácticamente legales del lenguaje de programación (advierta que aquí utilizamos la palabra "lenguaje" en dos sentidos diferentes).

Por ejemplo, un BNF para Pascal comenzará con reglas gramaticales tales como

*programa* → *encabezado-programa* ; *bloque-programa* .
*encabezado-programa* → . . .
*bloque-programa* → . . .
. . .

(La primera regla dice que un programa se compone de un encabezado de programa, seguido por un signo de punto y coma, seguido por un bloque de programa, seguido por un punto.) En lenguajes con compilación separada, como C, la estructura más general se conoce a menudo como una *unidad de compilación.* En todo caso, suponemos que la estructura más general se menciona primero en las reglas gramaticales, a menos que lo especifiquemos de otra manera. (En la teoría matemática de las gramáticas libres de contexto esta estructura se denomina **símbolo inicial.**)

Otro segmento de terminología nos permite distinguir con más claridad entre los nombres de estructura y los símbolos del alfabeto (los cuales hemos estado llamando símbolos de token, porque por lo regular son tokens en aplicaciones de compilador). Los nombres de estructuras también se conocen como **no terminales**, porque siempre se debe reemplazar de más en una derivación (no terminan una derivación). En contraste, los símbolos en el alfabeto se denominan **terminales**, porque éstos terminan una derivación. Como los terminales por lo regular son tokens en aplicaciones de compilador, utilizarán ambos nombres, los que, en esencia, son sinónimos. A menudo, se hace referencia tanto a los terminales como a los no terminales como símbolos.

Consideremos ahora algunos ejemplos de lenguajes generados por gramáticas.

**Ejemplo 3.1** Considere la gramática $G$ con la regla gramatical simple

$$E \rightarrow ( E ) \mid a$$

Esta gramática tiene un no terminal $E$ y tres terminales (,) y $a$. Genera, además, el lenguaje $L(G) = \{a,(a), ((a)), (((a))), \ldots\} = \{({}^n\, a)^n \mid n \text{ un entero} \geq 0\}$, es decir, las cadenas compuestas de 0 o más paréntesis izquierdos, seguidos por una $a$, seguida por el mismo número de paréntesis derechos que de paréntesis izquierdos. Como ejemplo de una derivación para una de estas cadenas ofrecemos una derivación para $((a))$:

$$E \Rightarrow ( E ) \Rightarrow (( E )) \Rightarrow ((a)) \qquad \S$$

**Ejemplo 3.2** Considere la gramática $G$ con la regla gramatical simple

$$E \rightarrow ( E )$$

Ésta resulta ser igual que la gramática del ejemplo anterior, sólo que se perdió la opción $E \rightarrow a$. Esta gramática no genera ninguna cadena, de modo que su lenguaje es vacío: $L(G) = \{\}$. La causa es que cualquier derivación que comienza con $E$ genera cadenas que siempre contienen una $E$. Así, no hay manera de que podamos derivar una cadena compuesta sólo de terminales. En realidad, como ocurre con todos los procesos recursivos (como las demostraciones por inducción o las funciones recursivas), una regla gramatical que define

una estructura de manera recursiva siempre debe tener por lo menos un caso no recursivo (al que podríamos denominar **caso base**). La regla gramatical de este ejemplo no lo tiene, y cualquier derivación potencial está condenada a una recursión infinita. §

**Ejemplo 3.3**

Considere la gramática G con la regla gramatical simple

$$E \rightarrow E + a \mid a$$

Esta gramática genera todas las cadenas compuestas de $a$ separadas por signos de "más" (+):

$$L(G) = \{ a, a + a, a + a + a, a + a + a + a, \ldots \}$$

Para ver esto (de manera informal), considere el efecto de la regla $E \rightarrow E + a$: esto provoca que la cadena $+ a$ se repita sobre la derecha en una derivación:

$$E \Rightarrow E + a \Rightarrow E + a + a \Rightarrow E + a + a + a \Rightarrow \ldots$$

Finalmente, debemos reemplazar la $E$ a la izquierda utilizando el caso base $E \rightarrow a$.

Podemos demostrar esto más formalmente mediante inducción como sigue. En primer lugar, mostramos que toda cadena $a + a + \cdots + a$ está en $L(G)$ por inducción en el número de las $a$. La derivación $E \Rightarrow a$ muestra que $a$ está en $L(G)$; suponga ahora que $s = a + a + \cdots + a$, con $n - 1$ $a$, está en $L(G)$. De este modo, existe una derivación $E \Rightarrow^* s$: ahora la derivación $E \Rightarrow E + a \Rightarrow^* s + a$ muestra que la cadena $s + a$, con $n + a$, está en $L(G)$. A la inversa, también mostramos que cualquier cadena $s$ de $L(G)$ debe ser de la forma $a + a + \cdots + a$. Mostramos esto mediante inducción sobre la longitud de una derivación. Si la derivación tiene longitud 1, entonces es de la forma $E \Rightarrow a$, así que $s$ es de la forma correcta. Ahora, suponga la veracidad de la hipótesis para todas las cadenas con derivaciones de longitud $n - 1$, y sea $E \Rightarrow^* s$ una derivación de longitud $n > 1$. Esta derivación debe comenzar con el reemplazo de $E$ por $E + a$, y de esta manera será de la forma $E \Rightarrow E + a \Rightarrow^* s' + a = s$. Entonces, $s'$ tiene una derivación de longitud $n - 1$, y de este modo será de la forma $a + a + \cdots + a$. Por lo tanto, $s$ misma debe tener esta misma forma. §

**Ejemplo 3.4**

Considere la siguiente gramática muy simplificada de sentencias:

$$\begin{aligned} &\textit{sentencia} \rightarrow \textit{sent-if} \mid \texttt{otro} \\ &\textit{sent-if} \rightarrow \texttt{if (}\ \textit{exp}\ \texttt{)}\ \textit{sentencia} \\ &\qquad\qquad \mid \texttt{if (}\ \textit{exp}\ \texttt{)}\ \textit{sentencia}\ \texttt{else}\ \textit{sentencia} \\ &\textit{exp} \rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

El lenguaje de esta gramática se compone de sentencias if anidadas de manera semejante al lenguaje C. (Simplificamos las expresiones de prueba lógica ya sea a 0 o a 1, y agrupamos todas las otras sentencias aparte de las sentencias if en el terminal `otro`.) Ejemplos de cadenas en este lenguaje son

```
otro
if (0) otro
if (1) otro
```

```
if (0) otro else otro
if (1) otro else otro
if (0) if (0) otro
if (0) if (1) otro else otro
if (1) otro else if (0) otro else otro
...
```

Advierta cómo la parte else opcional de la sentencia if está indicada por medio de una selección separada en la regla gramatical para *sent-if*. §

Antes advertimos que se conservan las reglas gramaticales en BNF para concatenación y selección, pero no tienen equivalente específico de la operación de repetición para el * de las expresiones regulares. Una operación así de hecho es innecesaria, puesto que la repetición se puede obtener por medio de recursión (como los programadores en lenguajes funcionales lo saben). Por ejemplo, tanto la regla gramatical

$$A \rightarrow A\ a \mid a$$

como la regla gramatical

$$A \rightarrow a\ A \mid a$$

generan el lenguaje $\{a^n \mid n \text{ un entero} \geq 1\}$ (el conjunto de todas las cadenas con una o más $a$), el cual es igual al que genera la expresión regular `a+`. Por ejemplo, la cadena *aaaa* puede ser generada por la primera regla gramatical con la derivación

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow aaaa$$

Una derivación similar funciona para la segunda regla gramatical. La primera de estas reglas gramaticales es **recursiva por la izquierda**, porque el no terminal $A$ aparece como el primer símbolo en el lado derecho de la regla que define $A$.[2] La segunda regla gramatical es **recursiva por la derecha**.

El ejemplo 3.3 es otro ejemplo de una regla gramatical recursiva por la izquierda, que ocasiona la repetición de la cadena "+ *a*". Éste y el ejemplo anterior se puede generalizar como sigue. Considere una regla de la forma

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

donde $\alpha$ y $\beta$ representan cadenas arbitrarias y $\beta$ no comienza con $A$. Esta regla genera todas las cadenas de la forma $\beta$, $\beta\alpha$, $\beta\alpha\alpha$, $\beta\alpha\alpha\alpha$, . . . (todas las cadenas comienzan con una $\beta$, seguida por 0 o más $\alpha$). De este modo, esta regla gramatical es equivalente en su efecto a la expresión regular $\beta\alpha^*$. De manera similar, la regla gramatical recursiva por la derecha

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

(donde $\beta$ no finaliza en $A$) genera todas las cadenas $\beta$, $\alpha\beta$, $\alpha\alpha\beta$, $\alpha\alpha\alpha\beta$, . . . .

---

2. Éste es un caso especial de recursión por la izquierda denominado **recursión por la izquierda inmediata**. Casos más generales se analizan en el capítulo siguiente.

Si queremos escribir una gramática que genere el mismo lenguaje que la expresión regular `a*`, entonces debemos tener una notación para una regla gramatical que genere la cadena vacía (porque la expresión regular `a*` incluye la cadena vacía). Una regla gramatical así debe tener un lado derecho vacío. Podemos simplemente no escribir nada en el lado derecho, como ocurre en

$$vacío \rightarrow$$

pero emplearemos más a menudo el metasímbolo épsilon para la cadena vacía (como se usó en las expresiones regulares):

$$vacío \rightarrow \varepsilon$$

Una regla gramatical de esta clase se conoce como **producción ε** (una "producción épsilon"). Una gramática que genera un lenguaje que contenga la cadena vacía debe tener por lo menos una producción ε.

Ahora podemos escribir una gramática que sea equivalente a la expresión regular `a*`, ya sea como

$$A \rightarrow A\ a \mid \varepsilon$$

o como

$$A \rightarrow a\ A \mid \varepsilon$$

Ambas gramáticas generan el lenguaje $\{a^n \mid n \text{ un entero} \geq 0\} = L(\texttt{a*})$. Las producciones ε también son útiles al definir estructuras que son opcionales, como pronto veremos.

Concluiremos esta subsección con algunos otros ejemplos más.

**Ejemplo 3.5**

Considere la gramática

$$A \rightarrow (\ A\ )\ A \mid \varepsilon$$

Esta gramática genera las cadenas de todos los "paréntesis balanceados". Por ejemplo, la cadena (( )(( ))( ) es generada por la derivación siguiente (la producción ε se utiliza para hacer que desaparezca $A$ cuando sea necesario):

$$\begin{aligned} A &\Rightarrow (\ A\ )\ A \Rightarrow (\ A\ )\ (\ A\ )\ A \Rightarrow (\ A\ )\ (\ A\ ) \Rightarrow (\ A\ )\ (\ ) \Rightarrow (\ (\ A\ )\ A\ )\ (\ ) \\ &\Rightarrow (\ (\ )\ A\ )\ (\ ) \Rightarrow (\ (\ )\ (\ A\ )\ A\ )\ (\ ) \Rightarrow (\ (\ )\ (\ A\ )\ )\ (\ ) \\ &\Rightarrow (\ (\ )\ (\ (\ A\ )\ A\ )\ )\ (\ ) \Rightarrow (\ (\ )\ (\ (\ )\ A\ )\ )\ (\ ) \Rightarrow (\ (\ )\ (\ (\ )\ )\ )\ (\ ) \end{aligned}$$

§

**Ejemplo 3.6**

La gramática de sentencia del ejemplo 3.4 se puede escribir de la siguiente manera alternativa utilizando una producción ε:

$$\begin{aligned} &\textit{sentencia} \rightarrow \textit{sent-if} \mid \texttt{otro} \\ &\textit{sent-if} \rightarrow \texttt{if (}\ \textit{exp}\ \texttt{)}\ \textit{sentencia parte-else} \\ &\textit{parte-else} \rightarrow \texttt{else}\ \textit{sentencia} \mid \varepsilon \\ &\textit{exp} \rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

Advierta cómo la producción ε indica que la estructura *parte-else* es opcional. §

**Ejemplo 3.7**

Considere la siguiente gramática $G$ para una secuencia de sentencias:

$$\textit{secuencia-sent} \rightarrow \textit{sent}\ \texttt{;}\ \textit{secuencia-sent} \mid \textit{sent}$$
$$\textit{sent} \rightarrow \texttt{s}$$

Esta gramática genera secuencias de una o más sentencias separadas por signos de punto y coma (las sentencias se extrajeron del terminal simple `s`):

$$L(G) = \{\ \texttt{s},\ \texttt{s;s},\ \texttt{s;s;s}, \ldots\ \}$$

Si queremos permitir que las secuencias de sentencia también sean vacías, podríamos escribir la siguiente gramática $G'$:

$$\textit{secuencia-sent} \rightarrow \textit{sent}\ \texttt{;}\ \textit{secuencia-sent} \mid \varepsilon$$
$$\textit{sent} \rightarrow \texttt{s}$$

pero esto convierte al signo de punto y coma en un **terminador** de sentencia en vez de en un **separador** de sentencia:

$$L(G') = \{\ \varepsilon,\ \texttt{s;},\ \texttt{s;s;},\ \texttt{s;s;s;}, \ldots\ \}$$

Si deseamos permitir que las secuencias de sentencia sean vacías, pero también mantener el signo de punto y coma como un separador de sentencia, debemos escribir la gramática como se muestra a continuación:

$$\textit{secuencia-sent} \rightarrow \textit{secuencia-sent-no vacía} \mid \varepsilon$$
$$\textit{secuencia-sent-no vacía} \rightarrow \textit{sent}\ \texttt{;}\ \textit{secuencia-sent-no vacía} \mid \textit{sent}$$
$$\textit{sent} \rightarrow \texttt{s}$$

Este ejemplo muestra que debe tenerse cuidado en la ubicación de la producción $\varepsilon$ cuando se construyan estructuras opcionales. §