Contenido

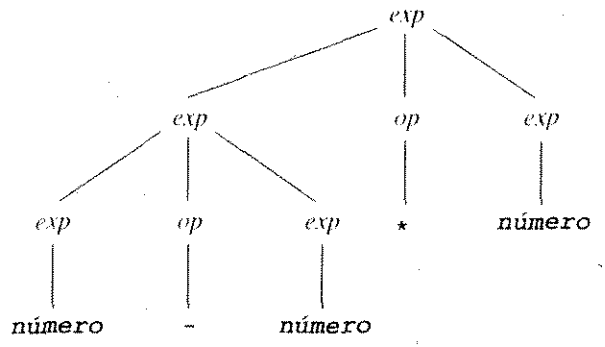
3.4 AMBIGÜEDAD

3.4.1 Gramáticas ambiguas

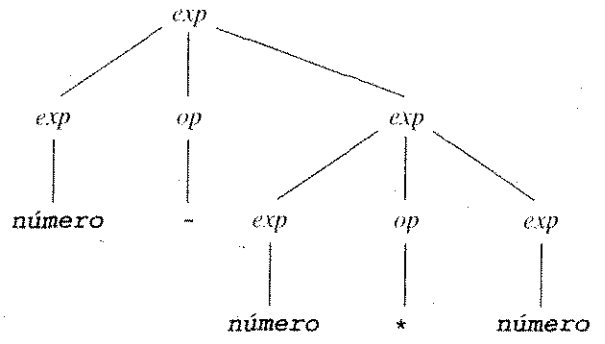
Los árboles de análisis gramatical y los árboles sintácticos expresan de manera única la estructura de la sintaxis, y efectúan derivaciones por la izquierda y por la derecha, pero no derivaciones en general. Desgraciadamente, una gramática puede permitir que una cadena tenga más de un árbol de análisis gramatical. Considere, por ejemplo, la gramática de la aritmética entera simple que hemos estado utilizando como ejemplo estándar

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

y considere la cadena `34-3*42`. Esta cadena tiene dos árboles de análisis gramatical diferentes



y



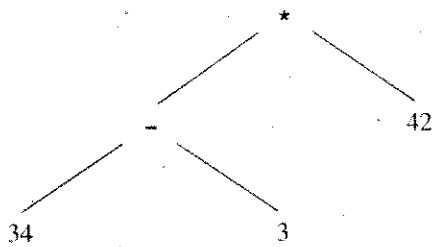
correspondientes a las dos derivaciones por la izquierda

$exp \Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
$\Rightarrow exp\ op\ exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
$\Rightarrow$ **número** $op\ exp\ op\ exp$	$[exp \rightarrow$ **número** $]$
$\Rightarrow$ **número** - $exp\ op\ exp$	$[op \rightarrow$ - $]$
$\Rightarrow$ **número** - **número** $op\ exp$	$[exp \rightarrow$ **número** $]$
$\Rightarrow$ **número** - **número** * exp	$[op \rightarrow$ * $]$
$\Rightarrow$ **número** - **número** * **número**	$[exp \rightarrow$ **número** $]$

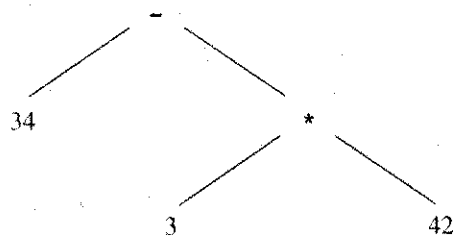
y

$exp \Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
$\Rightarrow$ **número** $op\ exp$	$[exp \rightarrow$ **número** $]$
$\Rightarrow$ **número** - exp	$[op \rightarrow$ - $]$
$\Rightarrow$ **número** - $exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
$\Rightarrow$ número - número $op\ exp$	$[exp \rightarrow$ **número** $]$
$\Rightarrow$ **número** - **número** * exp	$[op \rightarrow$ * $]$
$\Rightarrow$ número - número * número	$[exp \rightarrow$ **número** $]$

Los árboles sintácticos asociados son



y



Una gramática que genera una cadena con dos árboles de análisis gramatical distintos se denomina **gramática ambigua**. Una gramática de esta clase representa un serio problema para un analizador sintáctico, ya que no especifica con precisión la estructura sintáctica de un programa (aun cuando las mismas cadenas legales, es decir, los miembros del lenguaje de la gramática, estén completamente determinados). En cierto sentido, una gramática ambigua es como un autómata no determinístico en el que dos rutas por separado pueden aceptar la misma cadena. Sin embargo, la ambigüedad en las gramáticas no se puede eliminar tan fácilmente como el no determinismo en los autómatas finitos, puesto que no hay algoritmo para hacerlo así, a diferencia de la situación en el caso de los autómatas (la construcción del subconjunto analizada en el capítulo anterior).[4]

Una gramática ambigua debe, por lo tanto, considerarse como una especificación incompleta de la sintaxis de un lenguaje, y como tal debería evitarse. Afortunadamente las gramáticas ambiguas nunca pasan las pruebas que presentamos más adelante para los algoritmos estándar de análisis sintáctico, y existe un conjunto de técnicas estándar para tratar con ambigüedades típicas que surgen en lenguajes de programación.

Para tratar con las ambigüedades se utilizan dos métodos básicos. Uno consiste en establecer una regla que especifique en cada caso ambiguo cuál de los árboles de análisis gramatical (o árboles sintácticos) es el correcto. Una regla de esta clase se conoce como **regla de no ambigüedad** o de **eliminación de ambigüedades**. La utilidad de una regla de esta naturaleza es que corrige la ambigüedad sin modificar (con la posible complicación que esto implica) la gramática. La desventaja es que ya no sólo la gramática determina la estructura sintáctica del lenguaje. La alternativa es cambiar la gramática a una forma que obligue a construir el árbol de análisis gramatical correcto, de tal manera que se elimine la ambigüedad. Naturalmente, en cualquier método primero debemos decidir cuál de los árboles es el correcto en un caso ambiguo. Esto involucra de nueva cuenta al principio de la traducción dirigida por sintaxis. El árbol de análisis gramatical (o árbol sintáctico) que buscamos es aquel que refleja correctamente el significado posterior que aplicaremos a la construcción a fin de traducirlo a código objeto.

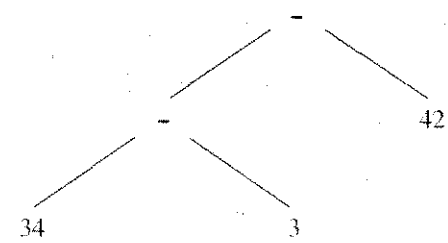
---

4. La situación en realidad es aún peor, puesto que no hay algoritmo para determinar si una gramática es ambigua en primera instancia. Véase la sección 3.2.7.

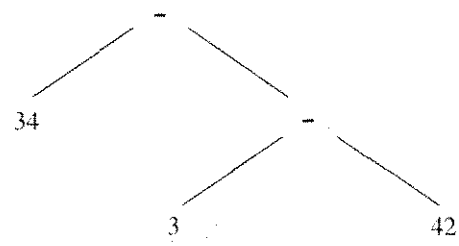
¿Cuál de los dos árboles sintácticos antes presentados representan la interpretación correcta de la cadena `34-3*42`? El primer árbol indica, al hacer el nodo de resta un hijo del nodo de multiplicación, que tenemos la intención de evaluar la expresión evaluando primero la resta (34 − 3 = 31) y después la multiplicación (31 * 42 = 1302). El segundo árbol, por otra parte, indica que primero se realizará la multiplicación (3 * 42 = 126) y después la resta (34 −126 = −92). La cuestión de cuál árbol elegiremos depende de cuál de estos cálculos visualicemos como correcto. La convención matemática estándar dicta que la segunda interpretación es la correcta. Esto se debe a que se dice que la multiplicación tiene **precedencia** sobre la resta. Por lo regular, tanto la multiplicación como la división tienen precedencia tanto sobre la suma como sobre la resta.

Para eliminar la ambigüedad dada en nuestra gramática de expresión simple, ahora podríamos simplemente establecer una regla de eliminación de ambigüedad que establezca las precedencias relativas de las tres operaciones representadas. La solución estándar es darle a la suma y a la resta la misma precedencia, y proporcionar a la multiplicación una precedencia más alta.

Desgraciadamente, esta regla aún no elimina por completo la ambigüedad de la gramática. Considere la cadena `34-3-42`. La cadena también tiene dos posibles árboles sintácticos:



y



El primero representa el cálculo (34 − 3) − 42 = −11, mientras el segundo representa el cálculo 34 − (3 − 42) = 73. De nueva cuenta, la cuestión de cuál cálculo es el correcto es un asunto de convención. La matemática estándar dicta que la primera selección es la correcta. Esto se debe a que la resta se considera como **asociativa por la izquierda**; es decir, que se realizan una serie de operaciones de resta de izquierda a derecha.

De este modo, una ambigüedad adicional que requiere de una regla de eliminación de ambigüedades es la asociatividad de cada una de las operaciones de suma, resta y multiplicación. Es común especificar que estas tres operaciones son asociativas por la izquierda. Esto en realidad elimina las ambigüedades restantes en nuestra gramática de expresión simple (aunque no podremos demostrar esto sino hasta más adelante).

También se puede elegir especificar que una operación es **no asociativa**, en el sentido de que una secuencia de más de un operador es una expresión que no está permitida. Por ejemplo, podríamos haber escrito nuestra gramática de expresión simple de la manera que se muestra a continuación:

$$
\begin{aligned}
&exp \rightarrow factor\ op\ factor \mid factor\\
&factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \texttt{número}\\
&op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}
\end{aligned}
$$

En este caso cadenas como `34-3-42` e incluso `34-3*42` son ilegales, y en cambio se deben escribir con paréntesis, tal como `(34-3)-42` y `34-(3*42)`. Estas expresiones **completamente entre paréntesis** no necesitan de la especificación de asociatividad o, en realidad, de precedencia. La gramática anterior carece de ambigüedad como está escrita. Naturalmente, no sólo cambiamos la gramática, sino que también cambiamos el lenguaje que se está reconociendo.

Ahora volveremos a métodos para volver a escribir la gramática a fin de eliminar la ambigüedad en vez de establecer reglas de no ambigüedad. Advierta que debemos hallar métodos que no modifiquen las cadenas básicas que se están reconociendo (como se hizo en el ejemplo de las expresiones completamente entre paréntesis).

3.4.2 Precedencia y asociatividad

Para manejar la precedencia de las operaciones en la gramática debemos agrupar los operadores en grupos de igual precedencia, y para cada precedencia debemos escribir una regla diferente. Por ejemplo, la precedencia de la multiplicación sobre la suma y la resta se puede agregar a nuestra gramática de expresión simple como se ve a continuación:

$$
\begin{aligned}
&exp \rightarrow exp\ opsuma\ exp \mid term\\
&opsuma \rightarrow \texttt{+} \mid \texttt{-}\\
&term \rightarrow term\ opmult\ term \mid factor\\
&opmult \rightarrow \texttt{*}\\
&factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \texttt{número}
\end{aligned}
$$

En esta gramática la multiplicación se agrupa bajo la regla *term*, mientras que la suma y la resta se agrupan bajo la regla *exp*. Como el caso base para una *exp* es un *term*, esto significa que la suma y la resta aparecerán "más altas" (es decir, más cercanas a la raíz) en los árboles de análisis gramatical y sintáctico, de manera que reciben una precedencia más baja. Una agrupación así de operadores en diferentes niveles de precedencia es un método estándar para la especificación sintáctica utilizando BNF. A una agrupación de esta clase se le conoce como **cascada de precedencia**.

Esta última gramática para expresiones aritméticas simples todavía no especifica la asociatividad de los operadores y aún es ambigua. La causa es que la recursión en ambos lados del operador permite que cualquier lado iguale repeticiones del operador en una derivación (y, por lo tanto, en los árboles de análisis gramatical y sintáctico). La solución es reemplazar una de las recursiones con el caso base, forzando las coincidencias repetitivas en el lado en que está la recursión restante. Por consiguiente, reemplazando la regla

$$exp \rightarrow exp\ opsuma\ exp \mid term$$

por

$$exp \rightarrow exp\ opsuma\ term \mid term$$

se hace a la suma y a la resta asociativas por la izquierda, mientras que al escribir

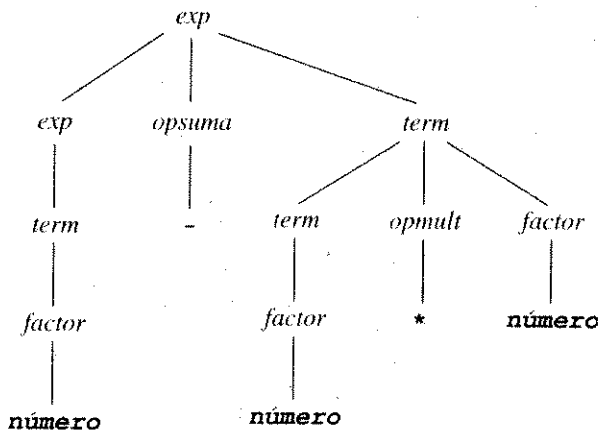
$$exp \rightarrow term\ opsuma\ exp \mid term$$

se hacen asociativas por la derecha. En otras palabras, una regla recursiva por la izquierda hace a sus operadores asociados a la izquierda, mientras que una regla recursiva por la derecha los hace asociados a la derecha.

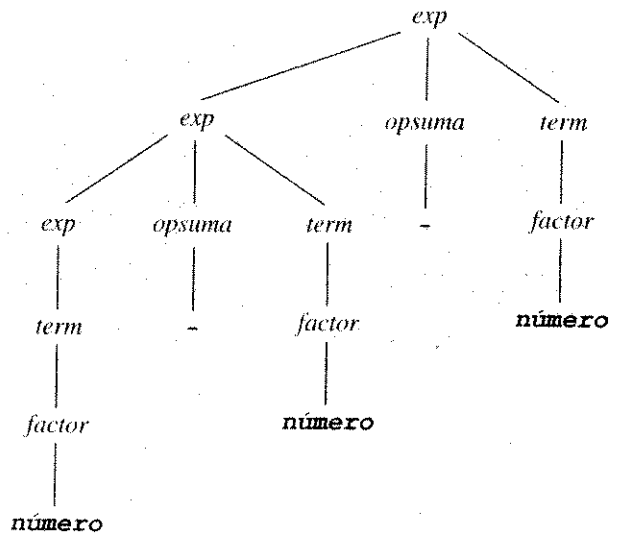
Para completar la eliminación de ambigüedades en las reglas BNF en el caso de nuestras expresiones aritméticas simples escribimos las reglas que permiten hacer todas las operaciones asociativas por la izquierda:

$$exp \rightarrow exp\ opsuma\ term \mid term$$
$$opsuma \rightarrow \texttt{+} \mid \texttt{-}$$
$$term \rightarrow term\ opmult\ factor \mid factor$$
$$opmult \rightarrow \texttt{*}$$
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{\texttt{número}}$$

Ahora el árbol de análisis gramatical para la expresión `34-3*42` es



y el árbol de análisis gramatical para la expresión `34-3-42` es



Advierta que las cascadas de precedencia provocan que los árboles de análisis gramatical se vuelvan mucho más complejos. Sin embargo, no afectan a los árboles sintácticos.

3.4.3 El problema del else ambiguo

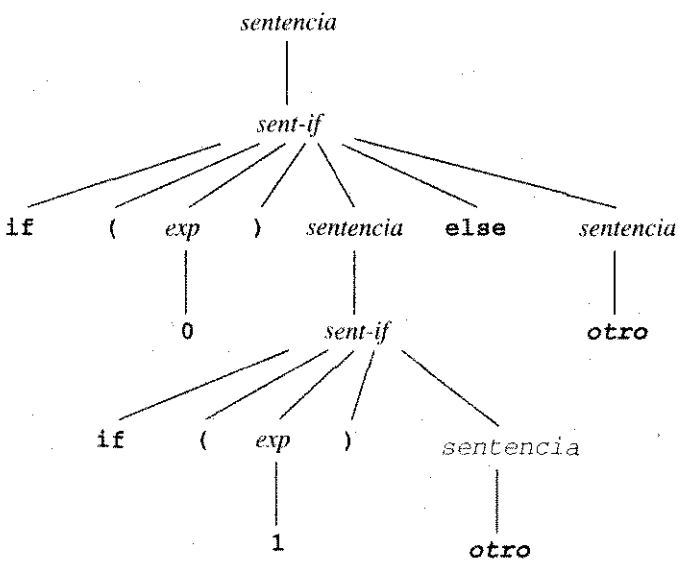
Considere la gramática del ejemplo 3.4 (página 103):

sentencia → *sent-if* | **otro**
sent-if → **if** **(** *exp* **)** *sentencia*
| **if** **(** *exp* **)** *sentencia* **else** *sentencia*
exp → **0** | **1**

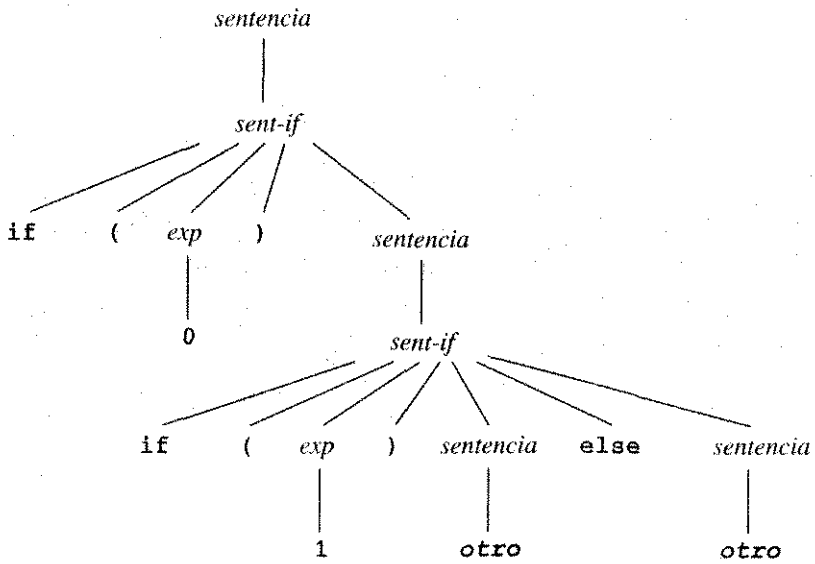
Esta gramática es ambigua como resultado del else opcional. Para ver esto considere la cadena

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

Esta cadena tiene los dos árboles de análisis gramatical:



y



La cuestión de cuál es el correcto depende de si queremos asociar la parte else con la primera o la segunda sentencia if: el primer árbol de análisis gramatical asocia la parte else con la primera sentencia if; el segundo árbol de análisis gramatical la asocia con la segunda

sentencia if. Esta ambigüedad se denomina **problema del else ambiguo**. Para ver cuál árbol de análisis gramatical es correcto, debemos considerar las implicaciones para el significado de la sentencia if. Para obtener una idea más clara de esto considere el siguiente segmento de código en C

```
if (x != 0)
   if (y == 1/x) ok = TRUE;
   else z = 1/x;
```

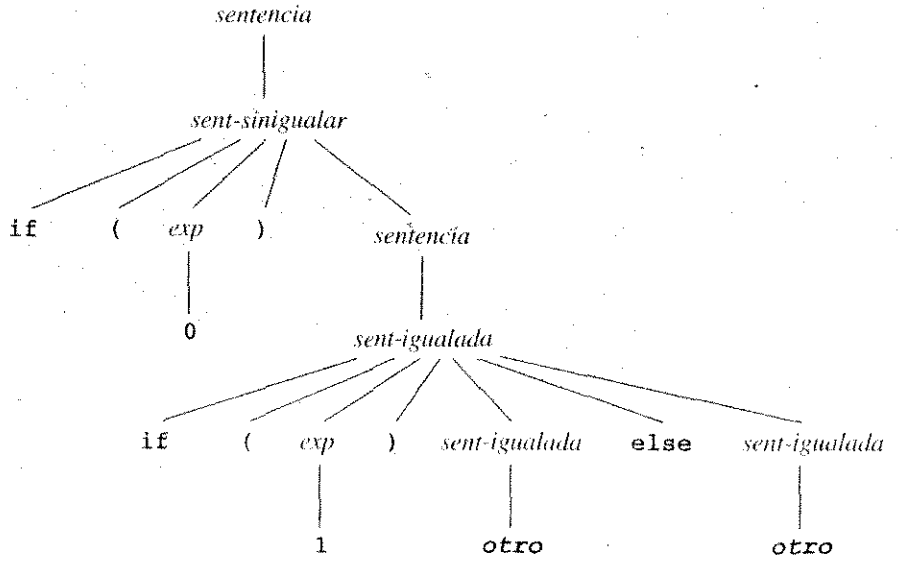
En este código, siempre que **x** sea 0, se presentará un error de división entre 0 si la parte else está asociada con la primera sentencia if. De este modo, la implicación de este código (y en realidad la implicación de la sangría de la parte else) es que una parte else siempre debería estar asociada con la sentencia if más cercana que todavía no tenga una parte else asociada. Esta regla de eliminación de ambigüedad se conoce como **regla de la anidación más cercana** para el problema del else ambiguo, e implica que el segundo árbol de análisis gramatical anterior es el correcto. Advierta que, si quisiéramos, *podríamos* asociar la parte else con la primera sentencia if mediante el uso de llaves {...} en C, como en

```
if (x != 0)
  {if (y == 1/x) ok = TRUE;}
else z = 1/x;
```

Una solución para la ambigüedad del else ambiguo en la BNF misma es más difícil que las ambigüedades previas que hemos visto. Una solución es como sigue:

sentencia → *sent-sinigualar* | *sent-igualada*
sent-sinigualar → `if` `(` *exp* `)` *sent-sinigualar* `else` *sent-sinigualar* | `otro`
sent-igualada → `if` `(` *exp* `)` *sentencia*
| `if` `(` *exp* `)` *sent-sinigualar* `else` *sent-igualada*
exp → `0` | `1`

Esto funciona al permitir que llegue solamente una *sentencia-igualada* antes que un `else` en una sentencia if, con lo que se obliga a que todas las partes else se empaten tan pronto como sea posible. Por ejemplo, el árbol de análisis gramatical asociado para nuestra cadena de muestra se convierte ahora en



lo que en realidad asocia la parte else con la segunda sentencia if.

Por lo regular no se emprende la construcción de la regla de la anidación más cercana en la BNF. En su lugar, se prefiere la regla de eliminación de ambigüedad. Una razón es la complejidad agregada de la nueva gramática, pero la razón principal es que los métodos de análisis sintáctico son fáciles de configurar de una manera tal que se obedezca la regla de la anidación más cercana. (La precedencia y la asociatividad son un poco más difíciles de conseguir automáticamente sin volver a escribir la gramática.)

El problema del else ambiguo tiene sus orígenes en la sintaxis de Algol60. La sintaxis se puede diseñar de tal manera que no aparezca el problema del else ambiguo. Una forma es *requerir* la presencia de la parte else, un método que se ha utilizado en LISP y otros lenguajes funcionales (donde siempre se debe devolver un valor). Otra solución es utilizar una **palabra clave de agrupación** para la sentencia if. Los lenguajes que utilizan esta solución incluyen a Algol60 y Ada. En Ada, por ejemplo, el programador escribe

```
if x /= 0 then
  if y = 1/x then ok := true;
  else z := 1/x;
  end if;
end if;
```

para asociar la parte else con la segunda sentencia if. De manera alternativa, el programador escribe

```
if x /= 0 then
  if y = 1/x then ok := true;
  end if;
else z := 1/x;
end if;
```

para asociarla con la primera sentencia if. La correspondiente BNF en Ada (algo simplificada) es

sent-if → **if** *condición* **then** *secuencia-de-sentencias* **end if**
| **if** *condición* **then** *secuencia-de-sentencias*
else *secuencia-de-sentencias* **end if**

De este modo, las dos palabras clave **end if** son la palabra clave de agrupación en Ada. En Algol68 la palabra clave de agrupación es **fi** (**if** escrito al revés).

3.4.4 Ambigüedad no esencial

En ocasiones una gramática puede ser ambigua y aún así producir siempre árboles sintácticos abstractos únicos. Considere, por ejemplo, la gramática de secuencia de sentencias del ejemplo 3.9 (página 113), donde podríamos elegir una simple lista de hermanos como el árbol sintáctico. En este caso una regla gramatical, ya fuera recursiva por la derecha o recursiva por la izquierda, todavía daría como resultado la misma estructura de árbol sintáctico, y podríamos escribir la gramática ambiguamente como

secuencia-sent → *secuencia-sent* **;** *secuencia-sent* | *sent*
sent → **s**

y aún obtener árboles sintácticos únicos. Una ambigüedad tal se podría llamar **ambigüedad no esencial**, puesto que la semántica asociada no depende de cuál regla de eliminación de ambigüedad se emplee. Una situación semejante surge con los operadores binarios, como los de la suma aritmética o la concatenación de cadenas, que representan **operaciones asociativas** (un operador binario $\cdot$ es asociativo si $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ para todo valor a, b y c). En este caso los árboles sintácticos todavía son distintos, pero representan el mismo valor semántico, y podemos no preocuparnos acerca de cuál utilizar. No obstante, un algoritmo de análisis sintáctico necesitará aplicar alguna regla de no ambigüedad que el escritor del compilador puede necesitar suministrar.