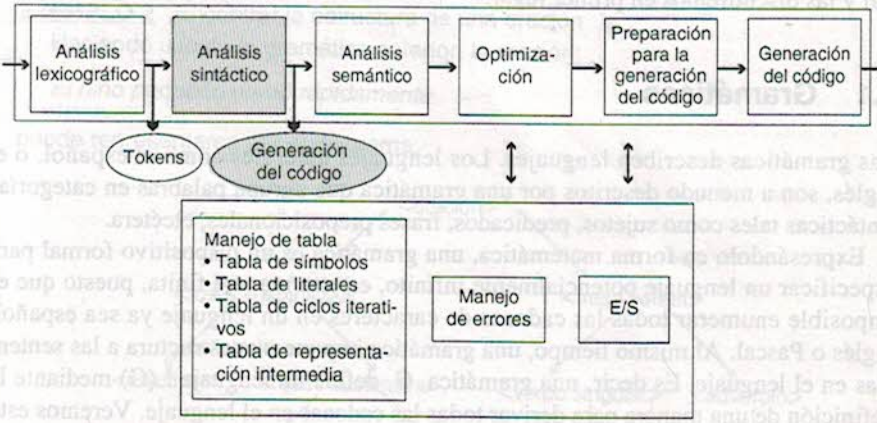
3

Introducción a las gramáticas y al análisis gramatical

El tiempo vuela como una flecha;
la fruta vuela como un plátano
–Groucho Marx



3.0 Introducción

El *análisis gramatical,* también conocido como *análisis sintáctico,* es la fase que intenta agrupar secuencias de tokens en categorías sintácticas. Si el analizador sintáctico no puede encontrar un agrupamiento así, entonces se informa que ha ocurrido un *error sintáctico.*

EJEMPLO 1 Caracteres, tokens y categorías sintácticas
Considérese la secuencia:

```
4 * Máx
```

Ésta consiste de siete caracteres (contando los espacios en blanco). Un analizador lexicográfico puede encontrar tres tokens *(4, * y Máx).* Sin embargo, el analizador sintáctico (analizador gramatical) puede agrupar todo esto en una categoría sintáctica, la *expresión.*

Muchos compiladores son *controlados por el analizador sintáctico,* lo que significa que este analizador llama al rastreador cada vez que necesita un nuevo token en lugar de recibir un flujo completo de éstos sobre el cual trabajar.

Los analizadores sintácticos se escriben para reconocer programas sintácticamente correctos en algún lenguaje. Tradicionalmente, la sintaxis de un lenguaje se define por medio de una *gramática,* esto es, un conjunto de reglas que definen si un programa es aparentemente correcto.

Otra forma de definir la sintaxis de un lenguaje es mediante un *aceptor* o *reconocedor* que decide si una cadena de caracteres particular pertenece a un lenguaje.

Para encontrar tokens, los aceptores recurren a los autómatas finitos ya utilizados (capítulo 2) aunque se discutieron otros formalismos. Las expresiones regulares son útiles en la generación automática de analizadores lexicográficos.

En el análisis sintáctico, las *gramáticas de libre contexto* son el formalismo más útil y las discutiremos en primer lugar.

3.1 Gramáticas

Las gramáticas describen lenguajes. Los lenguajes naturales como el español, o el inglés, son a menudo descritos por una gramática que agrupa palabras en categorías sintácticas tales como sujetos, predicados, frases preposicionales, etcétera.

Expresándolo en forma matemática, una gramática es un dispositivo formal para especificar un lenguaje potencialmente infinito, en una manera finita, puesto que es imposible enumerar todas las cadenas de caracteres en un lenguaje ya sea español, inglés o Pascal. Al mismo tiempo, una gramática impone una estructura a las sentencias en el lenguaje. Es decir, una gramática, G, define un lenguaje L(G) mediante la definición de una manera para derivar todas las cadenas en el lenguaje. Veremos esto primero para un (muy) pequeño subconjunto del idioma español.

3.1.1 Gramática de libre contexto para el idioma español

Noam Chomsky, en 1957, empleó la siguiente notación, denominada *producciones,* para definir la sintaxis del inglés (válida para el español). Los términos *oración, frase sustantiva,* etc., más las reglas siguientes describen un conjunto pequeño de oraciones en español. Los artículos *un* y *el, la, los, las* se han descrito aquí como adjetivos, por simplicidad.

<oración>	→ <frase sustantiva><frase verbal>
<frase sustantiva>	→ <adjetivo>< frase sustantiva> | <sustantivo singular><adjetivo>
<frase verbal>	→ <verbo singular><adverbio>
<adjetivo>	→ Un | El | pequeño
<sustantivo singular>	→ niño
<verbo singular>	→ corrió
<adverbio>	→ rápidamente

Aquí, la flecha, $\rightarrow$, debe leerse como "se define como" y la barra vertical, "|", como "o". De este modo, una frase sustantiva se define como un adjetivo seguido por otra frase sustantiva, o como un adjetivo seguido por un sustantivo singular. Esta definición de frase sustantiva es recursiva porque <frase sustantiva> se presenta en ambos lados de la producción. Las gramáticas a menudo son recursivas para permitir cadenas de caracteres de longitud infinita.

Se dice que esta gramática es de *libre contexto* debido a que solamente se presenta una categoría sintáctica, p. ej., *<frase verbal>*, a la izquierda de la flecha. Si hubiera más de una categoría sintáctica, esto describiría un contexto y se denominaría como *sensitiva al contexto.*

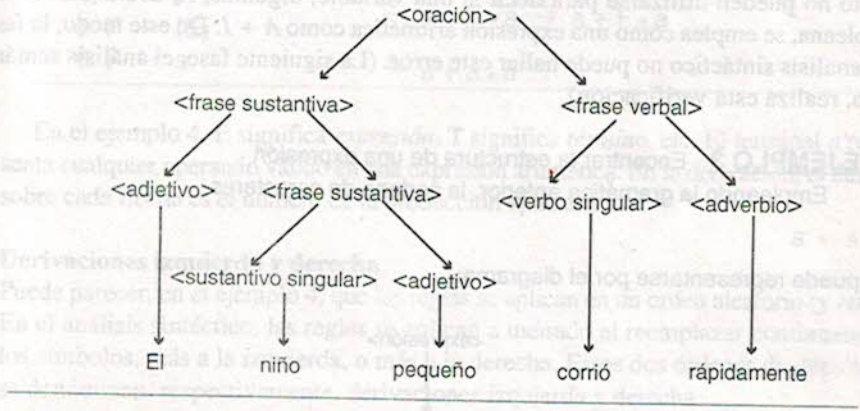
Una gramática es un ejemplo de un *metalenguaje* (un lenguaje utilizado para describir otro lenguaje). Aquí, el metalenguaje es la gramática de libre contexto empleada para describir una parte del lenguaje del idioma español.

EJEMPLO 2 Encontrar la estructura de una oración

Haciendo uso de la gramática anterior, la oración:

El niño pequeño corrió rápidamente

puede representarse por el diagrama:



La mayoría de la gente estaría de acuerdo en que "Rápidamente, el niño corrió" también es una oración sintácticamente correcta, pero no se puede derivar de la gramática anterior. De hecho, es imposible describir todas las oraciones correctas en español empleando una gramática de libre contexto. Por otro lado, es posible, utilizando la gramática mencionada con anterioridad, derivar la cadena de caracteres sintácticamente correcta, pero semánticamente incorrecta, "Pequeño el niño corrió rápidamente".

3.1.2 Gramáticas de libre contexto para lenguajes de programación

La estructura del español está dada en términos de sujetos, verbos, etc. La estructura de un programa de computadora está dada en términos de procedimientos, sentencias, expresiones, etc. Por ejemplo, una expresión aritmética consistente de sólo adición y multiplicación, puede describirse empleando las reglas siguientes:

```
<expresión>  ::= <expresión> + <término> | <término>
<término>    ::= <término> * <factor> |<factor>
<factor>     ::= (<expresión>) | <nombre> | <entero>
<nombre>     ::= <letra> | <nombre> <letra> | <nombre> <dígito>
<entero>     ::= <dígito> | <entero> <dígito>
<letra>      ::= A | B | ... | Z
<dígito>     ::= 0 | 1 | 2 | 3 | 4 | ... | 9
```

Aquí, hemos empleado "::=" para *se define como* en lugar de la flecha, $\rightarrow$, anteriormente usada. El metalenguaje BNF (forma Backus-Naur, por sus siglas en inglés), es una manera de especificar lenguajes de libre contexto y se definió originalmente utilizando ::= en vez de $\rightarrow$. Generalmente se atribuye a John Backus y Peter Naur el crédito por el desarrollo de esta notación como una forma de describir la sintaxis del lenguaje de programación Algol. En tanto que comprendemos lo que esto significa, y lo que son las capacidades de esta descripción gramatical, la notación no es importante.

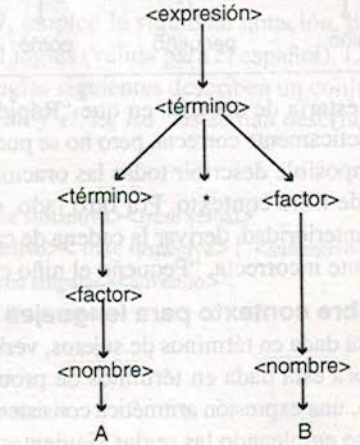
A diferencia de los lenguajes naturales como el español, todas las cadenas de caracteres válidas en un lenguaje de programación pueden especificarse empleando una gramática de libre contexto. Sin embargo, las gramáticas para los lenguajes de programación especifican las cadenas semánticamente incorrectas, así como lo hacen las gramáticas para los lenguajes naturales. Por ejemplo, las gramáticas de libre contexto no pueden utilizarse para decir si una variable, digamos, *A, declarada* como booleana, se emplea como una expresión aritmética como *A + 1*. De este modo, la fase de análisis sintáctico no puede hallar este error. (La siguiente fase, el análisis semántico, realiza esta verificación).

EJEMPLO 3 Encontrar la estructura de una expresión

Empleando la gramática anterior, la cadena de caracteres

```
A * B
```

puede representarse por el diagrama:



Árbol de análisis gramatical

El diagrama anterior se conoce como un *árbol de análisis gramatical.* Muestra cómo *se deriva* (genera) una cadena utilizando una gramática. Otras denominaciones para el árbol de análisis gramatical son *árbol sintáctico* y *árbol de derivación.*

Derivaciones

El árbol de análisis sintáctico anterior muestra la estructura de $A * B$, pero no nos dice con exactitud en qué orden fueron aplicadas las reglas enumeradas al principio de esta sección. (Estaremos interesados en el orden en que se aplicaron las reglas para el análisis sintáctico). El ejemplo 4 muestra una versión simplificada de esta gramática, y a su lado, una derivación de $a + a * a$. Utilizaremos esta gramática para definir alguna notación necesaria para describir cómo se analiza sintácticamente una cadena de caracteres.

EJEMPLO 4 La expresión sintáctica (simplificada) y una derivación

Gramática	Derivación de $a + a * a$
1. $E \rightarrow E + T$	$E \xrightarrow{1} E + T \xrightarrow{2} T + T \xrightarrow{3} T + T * F$
2. $E \rightarrow T$	$\xrightarrow{4} F + T * F \xrightarrow{6} F + T * a$
3. $T \rightarrow T * F$	$\xrightarrow{6} a + T * a \xrightarrow{4} a + F * a$
4. $T \rightarrow F$	$\xrightarrow{6} a + a * a$
5. $F \rightarrow (E)$	
6. $F \rightarrow a$	

En el ejemplo 4, E significa *expresión,* T significa *término,* etc. El terminal *a* representa cualquier operando válido en una expresión aritmética. En la derivación, el número sobre cada flecha es el número de la producción aplicada.

Derivaciones izquierda y derecha

Puede parecer, en el ejemplo 4, que las reglas se aplican en un orden aleatorio (y así es). En el análisis sintáctico, las reglas se aplican a menudo al reemplazar continuamente los símbolos, más a la izquierda, o más a la derecha. Estos dos órdenes de derivación se denominan, respectivamente, derivaciones izquierda y derecha.

EJEMPLO 5 Derivación derecha de $a + a * a$

$$E \xrightarrow{1} E + T \xrightarrow{3} E + T * F \xrightarrow{6} E + T * a$$
$$\xrightarrow{4} E + F * a \xrightarrow{6} E + a * a$$
$$\xrightarrow{2} T + a * a \xrightarrow{4} F + a * a$$
$$\xrightarrow{6} a + a * a$$

Las reglas aplicadas son:
1 3 6 4 6 2 4 6
Si aplicamos estas reglas en el orden enumerado, podemos esquematizar un árbol de análisis sintáctico para esta cadena de caracteres. De este modo, la lista de reglas aplicadas representa un árbol de análisis sintáctico.

Terminales
Una gramática se compone de un conjunto de símbolos terminales (tokens) tales como el signo de suma, +, el signo de multiplicación, *, paréntesis derecho e izquierdo, (y), y el identificador *a*. Un terminal es un símbolo que no aparece en el lado izquierdo de ninguna producción. (Los terminales pueden no componerse de un solo caracter.)

No terminales
Los no terminales son los nodos no-hojas en un árbol de análisis sintáctico. En la gramática anterior, E, T y F son no terminales. En nuestra primera y más formal versión de esta gramática, todos los no terminales fueron encerrados en paréntesis angulares o picoparéntesis, < >.

Producciones
Las producciones se introdujeron en la sección 3.1.1, y se pueden pensar como un conjunto de reglas de reemplazo (también conocidas como *reglas de reescritura*). Cada regla puede escribirse:

$$A ::= \alpha$$

o

$$A \rightarrow \alpha$$

donde A es un no terminal y α es una cadena de terminales y no terminales. En la gramática anterior E $\rightarrow$ E + T es una producción.

Símbolo de inicio
El símbolo de *partida* o de *inicio*, también denominado símbolo *meta* es un no terminal especial diseñado como el único a partir del cual todas las cadenas son derivadas. En nuestra gramática, E (por "expresión") es el símbolo designado de inicio.

Forma sentencial
Una *forma sentencial* es cualquier cadena de caracteres derivable desde el símbolo de inicio. De este modo, en la derivación anterior de $a + a * a$, E + T * F, E + F * a , y F + a * a, son todas formas sentenciales como lo es la misma $a + a * a$.

Sentencia
Una *sentencia* es una forma sentencial que consiste solamente de terminales tales como $a + a * a$.

Una sentencia puede derivarse (ser creada) utilizando el algoritmo siguiente:

Algoritmo
Derivar cadena

```
Cadena := Símbolo de inicio
REPITE
   Elija cualquier no terminal en Cadena.
   Encuentre una producción con este no terminal en el
      extremo izquierdo.
   Reemplace el no terminal con una de las opciones en
      el extremo derecho de la producción.
HASTA Cadena contiene sólo terminales.
```

La derivación del ejemplo 4 emplea este algoritmo para derivar $a + a * a$ de la gramática de expresión.

BNF extendida

Los procedimientos recursivos en programación pueden reescribirse haciendo uso de la iteración (y de una pila o "stack"). En forma similar, reescribimos producciones recursivas empleando la iteración. Las llaves tipográficas, { }, se utilizan a menudo para representar 0 o más ocurrencias de los elementos en el extremo izquierdo de una regla, y los corchetes, [ ], para representar elementos opcionales. De este modo, haciendo uso de la BNF extendida, podemos escribir la "gramática de expresión" anterior como:

```
E → T { + T }
T → F { * F }
F → (E) | a
```

La primera regla aquí deriva las formas sentenciales T, $T + T$, $T + T + T$, etcétera. Por supuesto, ya que F puede derivar una E en F → (E), esta gramática es todavía indirectamente recursiva.

Producciones épsilon

Una producción puede tener del lado derecho sólo cadenas vacías, $\in$. La siguiente gramática es otra versión de la llamada gramática de "expresiones":

```
E  → T E'
E' → + T E' | ∈
T  → F T'
T' → * F T' | ∈
F  → (E) | a
```

Aquí, se han introducido dos nuevos términos, E′ y T′.

Relación entre las gramáticas y el análisis sintáctico

En cierto sentido, el análisis sintáctico revierte el proceso de derivación en el sentido que tenemos una cadena de entrada y debemos "descubrir" el árbol de análisis sintáctico (si lo hay) para ella.

3.2 Ambigüedad

Si una oración en idioma inglés tiene más de un significado, se dice que es ambigua. Con frecuencia tales oraciones pueden analizarse sintácticamente en más de una forma. La oración,

Time flies like an arrow
(El tiempo vuela como una flecha)

puede interpretarse con *time (tiempo)* como un sustantivo, *flies (vuela)* como un verbo, y *like an arrow (como una flecha)* como una frase adverbial. Esta interpretación es un comentario acerca del rápido paso del tiempo. Sin embargo, si *time* se interpreta como un adjetivo, *flies* como un sustantivo, *like* como un verbo y *arrow* como un sujeto directo, la oración se convertiría en un comentario acerca de la vida amorosa de alguna especie conocida como *time fly (¡"las moscas de tiempo quieren a una flecha"!)*. Existen otras interpretaciones de esta oración, basadas en su sintaxis (véase el ejercicio 4).

En forma similar, se asigna significado a las construcciones en lenguaje de programación con base en sus sintaxis. Por consiguiente, preferimos que las gramáticas de los lenguajes de programación describan programas sin ambigüedad alguna.

Una sentencia es *ambigua* si hay más de una derivación distinta. Si una sentencia es ambigua, el árbol de análisis sintáctico no es único; podemos crear más de un árbol de análisis sintáctico para la misma sentencia.

Una gramática es *ambigua* incluso si puede generar una oración ambigua.

Los ejemplos 6 y 7 muestran construcciones comunes de lenguajes de programación, (1) expresiones y (2) la sentencia SI-ENTONCES-OTRO haciendo uso de gramáticas ambiguas.

EJEMPLO 6. Gramática ambigua para expresiones

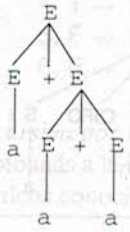
Considérese la siguiente gramática de expresiones:

```
E → E + E
E → E * E
E → (E)
E → a
```

y la entrada: *a* + *a* + *a*. Podemos hallar las derivaciones siguientes:

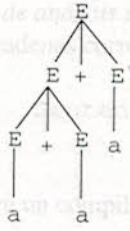
Derivación 1:

E → E + E → a + E → a + E + E
→ a + a + E → a + a + a



Derivación 2:

E → E + E → E + E + E → a + E + E
→ a + a + E → a + a + a



De este modo, en el ejemplo 6 hay dos árboles de análisis sintáctico enteramente diferentes para la misma expresión empleando la gramática. Por esta razón, se dice que la gramática es ambigua. Además, no muestra los niveles de precedencia como sí lo hace la gramática original (E → E + T, etc.) aunque existen otras maneras de incorporar la precedencia.

EJEMPLO 7 Otro colgante

Considere la siguiente gramática para las sentencias: SI-ENTONCES-OTRO:

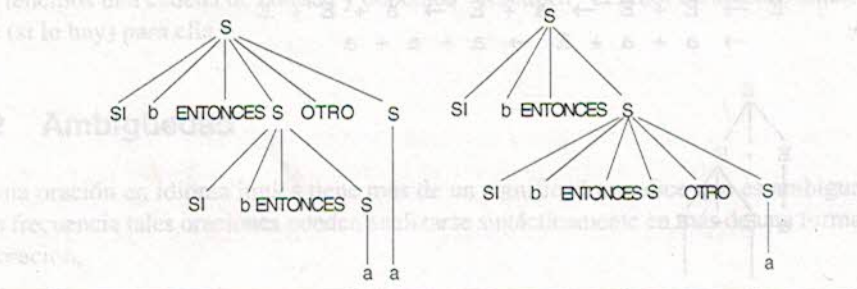
```
S → SI b ENTONCES S OTRO S
  | SI b ENTONCES S
  | a
```

donde *b* representa una condición y *a* representa otra secuencia de sentencias:

Considérese la siguiente cadena de caracteres derivable de esta gramática:

```
SI b ENTONCES SI b ENTONCES a OTRO a
```

donde *b* es una condición y *a* es una sentencia o conjunto de sentencias. Esta cadena tiene dos árboles de análisis sintáctico:



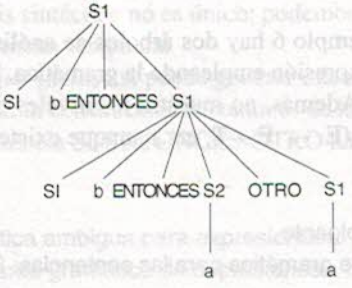
Para la gramática del ejemplo 7, el segundo árbol de análisis sintáctico a menudo se considera el único a elegir, es decir, el *otro* debería asociarse con el *si* "más cercano". Podemos revisar la gramática del ejemplo 7 a:

```
S1 → SI b ENTONCES S1 | SI b ENTONCES S2 OTRO S1 | a
S2 → SI b ENTONCES S2 OTRO S2 | a
```

Aquí, S1 es el símbolo de inicio. Ahora

```
SI b ENTONCES SI b ENTONCES a OTRO a
```

tiene un árbol de análisis sintáctico:



La ambigüedad no siempre es una propiedad inherente de los lenguajes; existen gramáticas ambiguas y no ambiguas para algunas construcciones. Sabemos que esas expresiones pueden describirse sin ambigüedad, puesto que lo hemos hecho en ejemplos previos.

Otros problemas de lenguaje

La ambigüedad no es el único problema que se encuentra en las gramáticas. Otras propiedades de éstas hacen difícil el análisis sintáctico utilizando técnicas particulares.

Por ejemplo, se dice que la gramática de expresiones:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → a

es *recursiva por la izquierda.* E puede generar una forma sentencial con E como el símbolo más a la izquierda. Como veremos, este es un problema para los analizadores sintácticos conocidos como analizadores sintácticos *descendentes.*

3.3 El problema del análisis sintáctico

Planteado en forma breve, el problema del análisis sintáctico es tomar una *cadena de símbolos* en un lenguaje y una gramática para ese lenguaje, y de éstos, construir el *árbol de análisis sintáctico,* o informar que esa oración es sintácticamente incorrecta. Para cadenas correctas:

```
Sentencia + Gramática  →  Árbol de análisis
                          sintáctico
```

Para un compilador, una sentencia es un programa:

```
Programa + Gramática  →  Árbol de análisis
                         sintáctico
```

Escribir una gramática para un lenguaje de programación es con frecuencia una tarea difícil. En el proyecto descrito al final de este capítulo, se muestra una gramática para un (muy) pequeño subconjunto de Ada. Se harán adiciones a esta gramática en capítulos posteriores.

Buscaremos los analizadores sintácticos que leen la cadena de entrada de izquierda a derecha buscando a lo máximo un token hacia adelante.

Utilizaremos la gramática siguiente que describe cadenas de dígitos decimales para ilustrar los diferentes métodos y problemas de análisis sintáctico.

```
N → D | N D
D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

3.3.1 Análisis sintáctico descendente

El análisis sintáctico descendente crea una derivación más a la izquierda. Los pasos son: 1) Comenzar con el símbolo de *inicio* (para nuestra gramática, *N*) como la "raíz" del árbol

de análisis sintáctico. (¡Es interesante que a la parte superior del árbol se le denomine raíz!)

2) En cada paso, reemplace el no terminal *V* del lado izquierdo en la forma sentencial actual.

```
xVy
by u
```

donde hay una regla (una producción)

```
V → u
```

de modo que tenemos

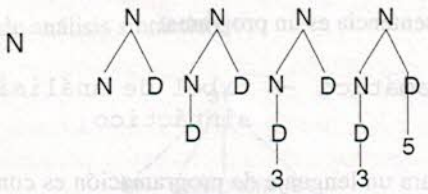
```
xuy
```

EJEMPLO 8 Análisis sintáctico descendente de 3 5

Empleando la gramática anterior, la entrada 3 5 debería ser analizada en forma descendente como:

```
N → N D → D D → 3 D → 3 5
```

El árbol de análisis sintáctico se construye paso a paso:



Describiremos dos métodos de análisis sintáctico descendente en el capítulo 4.

3.3.2 Análisis sintáctico ascendente

El análisis sintáctico ascendente crea el inverso de una derivación *derecha.*

Los pasos son:

1) Comience con la cadena de caracteres (p. ej., las hojas del árbol de análisis sintáctico que será creado).

2) Intente *reducir* al símbolo de *inicio* encontrando el *controlador* actual: El controlador es

i) la colección más extensa de terminales y no terminales en la parte extrema izquierda de la entrada que pueda encontrarse en el lado derecho de alguna producción y

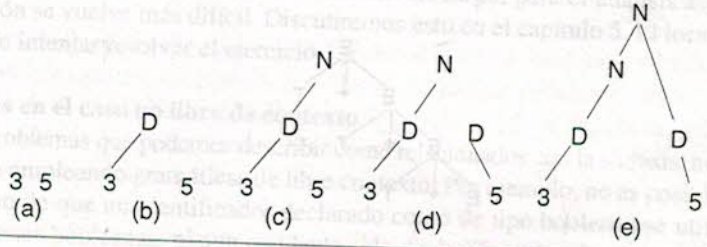
ii) tal que todos los símbolos a la derecha del manejador sean terminales y
iii) tal que el reemplazo del controlador con el lado izquierdo de la producción eventualmente (al encontrar más controladores) conduce de regreso al símbolo de *inicio*.

EJEMPLO 9 Análisis sintáctico ascendente de 3 5

La cadena de caracteres 3 5 se analizaría en forma ascendente como:

```
3 5 ← D 5 ← N 5 ← N D ← N
```

Los pasos para la construcción del árbol son:



En la figura del ejemplo 9, denotada como el inciso (*a*), 3 es el controlador puesto que todos los símbolos a su derecha (el número 5) son terminales y el 3 es la cadena más extensa encontrada en el lado derecho de una producción. En (*b*), D es el controlador; en (*c*), 5 es el controlador, y en (*d*), N D es el controlador. Se alienta al lector a justificar la elección del controlador en cada caso (véase el ejercicio 5).

3.3.3 Cuestiones del análisis sintáctico

Tanto el análisis sintáctico descendente como el ascendente involucran cuestiones por resolver.

Problemas en el caso descendente

***1)* Más de una elección** Supóngase que la gramática contiene dos opciones para el no terminal V:

```
V → u1 | u2
```

¿Cuál debería escogerse?

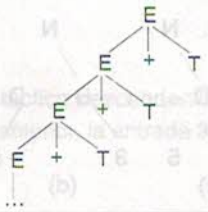
Una solución posible es elegir aleatoriamente; recuérdese el "estado del análisis" y cuál se eligió. Si se equivoca, el analizador sintáctico deberá regresar e intentar la otra opción. Esto se denomina análisis sintáctico inverso o backtracking, y no es muy eficiente. Una mejor manera, pero más complicada, es verificar el(los) siguiente(s) token(s) de entrada en la cadena y hacerla coincidir con el(los) primero(s) terminal(es) deriva-

do(s) de u1 o u2. Aunque es más complicado, esto es más eficiente.

2) **Recursividad izquierda** Considérese la gramática de expresión estándar:

```
E → E + T | T
T → T * F | F
F → (E) | a
```

y supóngase que queremos analizar $a + a + a + \ldots$ sin buscar más allá de uno o dos tokens hacia adelante. (En la práctica, los analizadores sintácticos buscan cuando mucho un token hacia adelante). Sin conocer cuántos signos "+" hay en la cadena, el analizador sintáctico no sabe cuántas veces aplicar la regla E → E + T. El signo "+" generado por el primer E → E + T puede estar 100 tokens más allá del principio de la cadena (!):



Es decir, sin el conocimiento de cuántos "+" existen en la cadena (¡y no es práctico buscar en toda la cadena para encontrarlos!), un analizador sintáctico descendente no sabe cuántas veces emplear la producción E → E + T antes de usar la regla E → T.

Si se utiliza el análisis descendente, las gramáticas recursivas izquierdas deben modificarse (véanse los ejercicios 9, 10 y 11).

3) **Factorización izquierda** Algunas veces dos producciones comparten un(os) símbolo(s) común(es) al comienzo de sus lados derechos. Éste es un problema para los analizadores sintácticos descendentes que deben extender una producción o la otra.

Por ejemplo,

```
Sentencia → SI C ENTONCES S OTRO S | SI C ENTONCES S
```

contiene los mismos tokens iniciales, *SI C ENTONCES S* (*C* y *S* representan muchos tokens).

Si cambiamos esto a:

```
S  → SI C ENTONCES S S'
S' → OTRO S | ∈
```

entonces un analizador sintáctico descendente no tiene ningún problema. Existe un método formal para la factorización izquierda de una gramática (véase ejercicio 12).

Cuestiones en el caso ascendente

El análisis sintáctico ascendente no acarrea tantos problemas como el caso descendente. El primer problema para el análisis sintáctico ascendente es análogo a un problema para análisis descendente.

Supóngase que la gramática contiene ambas producciones:

$$U \rightarrow \omega$$
$$V \rightarrow \omega$$

y ω es el controlador en la cadena que está siendo analizada; ¿se reduciría ω a U o a V? Como con el caso descendente, el análisis sintáctico hace una elección aleatoria y retrocede si se equivoca. Una vez más, esta forma es ineficiente y se emplea generalmente un método de búsqueda adelantada. Sin embargo, para el análisis ascendente dicha acción se vuelve más difícil. Discutiremos esto en el capítulo 5. El lector interesado puede intentar resolver el ejercicio 13.

Cuestiones en el caso no libre de contexto

Algunos problemas que podemos describir como relacionados con la sintaxis, no pueden describirse empleando gramáticas de libre contexto. Por ejemplo, no es posible expresar el hecho de que un identificador declarado como de tipo booleano se utilice sólo en expresiones booleanas, ni aun que haya sido declarado antes de que se utilice. Un problema similar es el de hacer coincidir el número y tipo de argumentos en una llamada de procedimiento, para aquellos declarados en la definición del procedimiento. Aunque estos aspectos parecen ser "sintácticos", se resuelven en la fase de análisis semántico de un compilador.

3.4 Generadores de analizadores sintácticos

Los dos capítulos siguientes describen detalles de varios métodos y algoritmos de análisis sintáctico. Sin embargo, es posible evitar la escritura de estos programas mediante el uso de uno de los muchos programas generadores de analizadores sintácticos que existen en el medio.

Los generadores de analizadores sintácticos, como los generadores de rastreadores, con frecuencia vienen en dos partes: un generador de tablas, que lee en la BNF y crea una tabla, y un programa controlador que lee la entrada, consulta la tabla, y crea el árbol de análisis sintáctico. Las tablas y el controlador varían de acuerdo con el método de análisis si es descendente (descrito en el capítulo 4) o ascendente (descrito en el capítulo 5). Muchos de los generadores de analizadores sintácticos de la actualidad incluyen a la tabla y al controlador dentro de un programa, haciendo la herramienta resultante un poco más fácil de personalizar.

Un programa generador de analizadores sintácticos muy conocido es YACC, *Yet Another Compiler Compiler*, una herramienta que viene con el sistema operativo UNIX.

3.5 Resumen

Este capítulo introduce el problema de encontrar la estructura gramatical de programas de entrada. El código será creado en última instancia basado en su estructura sintáctica, así que la fase de análisis sintáctico es una fase extremadamente importante.

Las funciones principales de un analizador sintáctico son: 1) la construcción de un árbol de análisis sintáctico para una cadena de caracteres de entrada o 2) el informe de que la cadena de entrada no es gramaticalmente correcta.

Hay dos métodos principales de análisis sintáctico: descendente, el cual crea un árbol de análisis sintáctico desde la raíz hacia abajo, y ascendente, que lo crea desde las hojas hacia arriba.

Los lenguajes se describen utilizando gramáticas. En particular, la sintaxis de un lenguaje se describe empleando gramáticas de libre contexto y escribiendo hacia abajo usando la forma Backus-Naur (BNF).

Las gramáticas de libre contexto a menudo tienen que reescribirse para poder instrumentarlas en un analizador sintáctico. Algunas de sus dificultades incluyen la ambigüedad, la recursividad izquierda y la necesidad de factorización izquierda.

Muchos de los términos y conceptos introducidos en este capítulo se utilizarán en los capítulos 4 (análisis sintáctico descendente), 5 (análisis sintáctico ascendente) y 6 (manejo de errores sintácticos).

EJERCICIOS

1. Derive la cadena $a + a * a$ utilizando (*a*) la gramática del ejemplo 4, (*b*) la gramática ambigua del ejemplo 6, (*c*) la gramática BNF extendida de la sección 3.1.2, y (*d*) la gramática con las producciones épsilon de la sección 3.1.2. En cada caso, dibuje tantos árboles de análisis sintáctico como pueda.
2. Para la cadena de caracteres del ejercicio 1 y las cuatro gramáticas, muestre (*a*) una derivación derecha y (*b*) una derivación izquierda.
3. Estudie los dos primeros ejercicios y diseñe un algoritmo que derive una cadena de caracteres dada en forma automática, buscando solamente el siguiente símbolo de entrada. Es decir, diseñe un algoritmo para análisis sintáctico descendente.
4. Encuentre otra interpretación (véase la sección 3.2) para la oración ambigua: *Time flies like an arrow* (el tiempo vuela como una flecha).
5. Justifique la elección de los controladores en los pasos (*b*), (*c*) y (*d*) del ejemplo 9.
6. Utilizando la gramática de expresiones "estándar":

```
E → E + T
E → T
T → T * F
T → F
F → (E) | Id
```

identifique el controlador de:

a) `a * a + a`
b) `T * F + a`
c) `T + F`
d) `E + F`
e) `E + T`
f) `E + T * F`

7. Demuestre que la gramática:

```
Número → Dígito | Número Número
Dígito →  0 | 1 | 2 |. . . |9
```

es ambigua. Modifíquela para que no sea ambigua, pero que además genere el mismo conjunto de cadenas de caracteres.

8. Considere la gramática siguiente:

```
S → (S) S | ∈
```

a) Describa el conjunto de cadenas de caracteres que genera.
b) ¿Es ambigua?
c) ¿Es recursiva izquierda?

9. Eliminación de la recursividad izquierda. Parte I: Existe una técnica formal para eliminar la recursividad izquierda de las producciones.
Para cada regla que contiene una opción de recursividad izquierda,

```
A → A α | β
```

introduce un nuevo no terminal A′ y reescribe la regla como

```
A → β A′
A′ → ∈ | α A′
```

Así, la producción:

```
E → E + T | T
```

es de recursividad izquierda con "E" jugando el papel de A, "+ T" jugando el papel de α, y "T" jugando el papel de β. Al introducir el nuevo no terminal E′, la producción puede reemplazarse por:

```
E  → T E′
E′ → ∈ | + T E′
```

(Adviértase, sin embargo, que hemos perdido la estructura "asociativa por la izquierda" original.) Utilice esta regla para eliminar la recursividad izquierda de las siguientes producciones:

a) `T → T * F | F`
b) `A → A X Y z | z`
c) `ListaSentencia → ListaSentencia; Sentencia | Sentencia`

10. Eliminación de la recursividad izquierda. Parte II: El método del ejercicio 9 puede extenderse a más de dos selecciones en el lado derecho. La regla general es reemplazar

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_m$$

por

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$
$$A' \rightarrow \in \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A'$$

Emplee esta regla para eliminar la recursividad izquierda en los casos siguientes:

i) `A → AXYz | z | By`
ii) `A → AXYz | Az | z | By`

11. Eliminación de la recursividad izquierda. Parte III: El ejercicio 10 describe una regla para la eliminación directa de la recursividad izquierda de una producción general. Eliminar la recursividad izquierda de una gramática completa puede ser más difícil debido a la recursividad izquierda indirecta. Por ejemplo:

```
A → B x y | x
B → C D
C → A | c
D → d
```

es indirectamente recursiva. (¿Por qué?) El algoritmo siguiente elimina por completo la recursividad izquierda.

Organice los no terminales en algún orden, A_1, A_2, A_3, ..., A_n.

```
PARA i := 1 HASTA n HACER
COMIENZA {PARA i}
    PARA j := 1 HASTA n HACER
    COMIENZA {PARA j}
```

1) Reemplace cada producción de la forma $A_i \rightarrow A_j\gamma$ por las producciones:

$A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$
donde:
$A_j \rightarrow \delta_1 \mid \delta_2 \mid ... \mid \delta_k$

son todas las producciones A_j actuales.
2) Elimine la recursividad izquierda directa de las producciones A_i:

```
  TERMINA {PARA j}
TERMINA {PARA i}
```

Elimine la recursividad izquierda de la gramática anterior.

12. Factorización izquierda: Cuando una gramática tiene dos o más producciones de la forma $A \rightarrow \beta\alpha_1$, $A \rightarrow \beta\alpha_2$, un analizador sintáctico descendente tiene que buscar hacia atrás todos los tokens en α en la cadena de caracteres que se está analizando para decidir cuál producción utilizar. En vez de esto la gramática puede cambiarse. La técnica formal es modificar

$A \rightarrow \beta\alpha_1 \mid \rightarrow \beta\alpha_2$
a
$A \rightarrow \beta\ A'$
$A' \rightarrow \alpha_1 \mid \alpha_2$

Aplique esta técnica a:

a) `S → SI C ENTONCES S OTRO S|SI C ENTONCES S`
b) `S → REPITE S HASTA C |REPITE S PARA SIEMPRE`

13. Diseñe un método de búsqueda (y de este modo un requerimiento para gramáticas que se analizarán sintácticamente en forma ascendente) para el análisis sintáctico ascendente que resolverá el problema del controlador que se presenta en el lado derecho de dos producciones diferentes.

Proyecto de compilador parte II

Gramática BNF

Lo que se ve a continuación es un BNF extendido para un subconjunto de Ada:

```
Programa                → comienza SecuenciaDeSentencias termina ;
SecuenciaDeSentencias   → Sentencia {Sentencia}
```

```
Sentencia                → SentenciaSimple
SentenciaSimple          → SentenciaDeAsignación
SentenciaDeAsignación    → Nombre := Expresión ;
Nombre                   → NombreSimple
NombreSimple             → Identificador
Expresión                → Relación
Relación                 → ExpresiónSimple
ExpresiónSimple          → Término {Término OperadorAdición}
Término                  → Factor{OperadorMultiplicación Factor}
Factor                   → Primario
Primario                 → Nombre|LiteralNumérica|(Expresión)
OperadorSuma             → + | -
OperadorMultiplicación   → * | / | mod | rem
LiteralNumérica          → LiteralDecimal
LiteralDecimal           → Entero
```

Asignación

Cree un árbol de análisis sintáctico, a mano, empleando la gramática anterior, y el programa del proyecto asignado en el capítulo 2 que se repite enseguida. No necesita incluir todos los nodos, puesto que el árbol es extenso.

```
comienza
    a := b3;
    xyz := a + b + c
        - p / q;
    a := xyz * (p + q) ;
    p := a - xyz - p;
termina;
```