Gramáticas independientes del contexto

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante **gramáticas independientes del contexto.** Estas gramáticas han desarrollado un importante papel en la tecnología de implementación de compiladores.

Vamos a presentar la notación de la gramática independiente del contexto y a mostrar cómo las gramáticas definen los lenguajes. Veremos los "árboles de análisis sintácticos", que representan la estructura que aplica una gramática a las cadenas de su lenguaje.

El árbol de análisis es el resultado que proporciona el analizador sintáctico de un lenguaje de programación y es la forma en la que se suele representar la estructura de los programas.

Existe una notación similar a la de los autómatas, denominada "autómata a pila", que también describe todos y sólo los lenguajes independientes del contexto.

Cada lenguaje de programación tiene reglas precisas, las cuales prescriben la estructura sintáctica de los programas bien formados. Por ejemplo, en "C" un programa está compuesto de funciones, una función de declaraciones e instrucciones, una instrucción de expresiones, y así sucesivamente.

La sintaxis de las construcciones de un lenguaje de programación puede especificarse mediante gramáticas independientes del contexto usando la notación **BNF** (**Forma de Backus-Naur**). Las gramáticas ofrecen beneficios considerables, tanto para los diseñadores de lenguajes como para los escritores de compiladores.

- Una gramática proporciona una especificación sintáctica precisa, pero fácil de entender, de un lenguaje de programación.
- A partir de ciertas clases de gramáticas, podemos construir de manera automática un analizador sintáctico eficiente que determine la estructura sintáctica de un programa fuente. **(*)**
- Como beneficio adicional, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y puntos problemáticos que podrían haberse pasado por alto durante la fase inicial del diseño del lenguaje.

- La estructura impartida a un lenguaje mediante una gramática diseñada en forma apropiada es útil para traducir los programas fuente en código objeto correcto, y para detectar errores.
- Una gramática permite que un lenguaje evolucione o se desarrolle en forma iterativa, agregando nuevas construcciones para realizar nuevas tareas. Estas nuevas construcciones pueden integrarse con más facilidad en una implementación que siga la estructura gramatical del lenguaje.

(*) Una herramienta utilizada para la generación automática de analizadores sintácticos es GNU Bison el que convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto, en un programa en C, C++, o Java que realiza análisis sintáctico. Es utilizado para crear analizadores para muchos lenguajes, desde simples calculadoras hasta lenguajes complejos. Para utilizar Bison, es necesaria experiencia con la sintaxis usada para describir gramáticas.
https://www.gnu.org/software/bison/

Definición formal de las gramáticas independientes del contexto

Existen cuatro componentes importantes en una descripción gramatical de un lenguaje:

1. Un conjunto finito de símbolos que forma las cadenas del lenguaje que se está definiendo. Denominamos a este conjunto alfabeto terminal o **alfabeto de símbolos terminales.** (usaremos este color para estos símbolos)
2. Un conjunto finito de variables, denominado también en ocasiones **símbolos no terminales** o categorías sintácticas (usaremos este color para estos símbolos). Cada variable representa un lenguaje; es decir, un conjunto de cadenas.
3. Una de las variables representa el lenguaje que se está definiendo; se denomina **símbolo inicial** (usaremos este color para este símbolo). Otras variables representan las clases auxiliares de cadenas que se emplean para definir el lenguaje del símbolo inicial.
4. Un conjunto finito de **producciones o reglas** que representan la definición recursiva de un lenguaje. Cada producción consta de:

a) Una variable a la que define (parcialmente) la producción. Esta variable a menudo se denomina cabeza de la producción.

b) El símbolo de producción →

c) Una cadena formada por cero o más símbolos terminales y variables no terminales. Esta cadena, denominada cuerpo de la producción, representa una manera de formar cadenas pertenecientes al lenguaje.

De este modo, dejamos los símbolos terminales invariables y sustituimos cada una de las variables del cuerpo por una cadena que sabemos que pertenece al lenguaje de dicha variable.

Los cuatro componentes que acabamos de describir definen una gramática independiente del contexto, (GIC), o simplemente una gramática, o en inglés CFG, *context-free grammar*. Representaremos una GIC **G** mediante sus cuatro componentes:

G = (N,T,P,S)

Donde **N** es el conjunto de variables **no terminales**, **T** son los símbolos **terminales**, **P** es el conjunto de producciones y **S** es el símbolo inicial que pertenece al conjunto **N**.

En resumen la definición formal de una gramática libre de contexto (o simplemente gramática) consiste en **terminales**, **no terminales**, **un símbolo inicial** y **producciones**.

1. Los **terminales** son los símbolos básicos a partir de los cuales se forman las cadenas. El término "**nombre de token**" es un sinónimo de "**terminal**"; con frecuencia usaremos la palabra "**token**" en vez de terminal, cuando esté claro que estamos hablando sólo sobre el nombre del token, algunos terminales pueden ser palabras reservadas en algún determinado lenguaje de programación, como por ejemplo **if**, **else**, y los símbolos "**(**" y "**)**".

2. Los **no terminales** son **variables sintácticas** que denotan **conjuntos de cadenas**. En por ejemplo **instr** y **expr** son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.

3. En una gramática, un no terminal se distingue como el **símbolo inicial**, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las

producciones para el símbolo inicial se listan primero, , en ocasiones también se lo denomina "axioma" .

4. Las **producciones** de una gramática especifican la forma en que **pueden combinarse los terminales y los no terminales para formar cadenas**. Cada producción consiste en:

a) Un no terminal, conocido como **encabezado o lado izquierdo** de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.

b) El símbolo →. Algunas veces se ha utilizado **::=** en vez de la flecha.
Este símbolo se lee como: "**deriva en**", o bien "**produce**".

c) Un **cuerpo o lado derecho**, que consiste en cero o más terminales y no terminales.

Ejemplo 1: La gramática en la **Figura 1** define expresiones aritméticas simples. En esta gramática, los símbolos terminales **T** son: **id + − * / ( )**

Los símbolos de los no terminales **N** son **expresión**, **term** y **factor.**

expresión es el símbolo inicial, **S**

Las producciones **P** son:

expresión → expresión + term

expresión → expresión − term

expresión → term

term → term * factor

term → term / factor

term → factor

factor → (expresión)

factor → id

Figura 1: Gramática para las expresiones aritméticas simples

Convenciones de notación

Para evitar siempre tener que decir que “éstos son los terminales”, “éstos son los no terminales”, etcétera, utilizaremos las siguientes convenciones de notación para las gramáticas:

1. Estos símbolos son **terminales**:

a. Las primeras letras minúsculas del alfabeto, como **a**, **b**, **c**.

b. Los símbolos de operadores como **+**, ***** , etcétera.

c. Los símbolos de puntuación como paréntesis, coma, etcétera.
d. Los dígitos **0**, **1**, …, **9**.
e. Las cadenas en negrita como **id** o **if**, cada una de las cuales representa un solo símbolo terminal.

2. Estos símbolos son **no terminales**:

a. Las primeras letras mayúsculas del alfabeto, como **A**, **B**, **C**, ...
b. La letra **S** que, al aparecer es, por lo general, **el símbolo inicial**.
c. Los nombres en cursiva y minúsculas, como ***expr*** o ***instr***.
d. Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar **no terminales**. Por ejemplo, los no terminales para las **expresiones**, los **términos** y los **factores** se representan a menudo mediante **E**, **T** y **F**, respectivamente.

3. Las últimas letras mayúsculas del alfabeto, como **X**, **Y**, **Z**, representan símbolos gramaticales; es decir, pueden ser **no terminales o terminales**.

4. Las últimas letras minúsculas del alfabeto, como **u**, **v**, …, **z**, representan **cadenas de terminales** (posiblemente vacías).

5. Las letras griegas minúsculas **α, β, γ**, por ejemplo, representan cadenas (posiblemente vacías) de **símbolos gramaticales**. Por ende, una producción genérica puede escribirse como **A** → **α**, en donde **A** es el encabezado y **α** el cuerpo.

6. Un conjunto de producciones **A** → α_1, **A** → α_2, …, **A** → α_k con un encabezado común **A** (las llamaremos producciones **A**), puede escribirse como **A** → α_1 | α_2 | … | α_k. a α_1,α_2, …, α_k les llamamos las alternativas para **A**.

7. A menos que se indique lo contrario, el encabezado de la primera producción es el símbolo inicial.

Ejemplo 2: Mediante estas convenciones, la gramática del **ejemplo 1** puede rescribirse en forma concisa como:

E → **E + T** | **E - T** | **T**
T → **T * F** | **T / F** | **F**
F → **(E)** | **id**

Las convenciones de notación nos indican que **E**, **T** y **F** son no terminales, y **E** es el símbolo inicial. El resto de los símbolos son terminales.

Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura. **Empezando con el símbolo inicial**, cada paso de rescritura sustituye a un **no terminal por el cuerpo de una de sus producciones**. Esta vista derivacional corresponde a la construcción descendente de un **árbol de análisis sintáctico**, pero la precisión que ofrecen las derivaciones será muy útil cuando hablemos del análisis sintáctico ascendente. Como veremos, el análisis sintáctico ascendente se relaciona con una clase de derivaciones conocidas como derivaciones de "**más a la derecha**", en donde el no terminal por la derecha se rescribe en cada paso.

Por ejemplo, considere la siguiente gramática, con un solo no terminal **E**:

E → E + E | E * E | − E | (E) | id

La producción **E → − E** significa que si **E** denota una expresión, entonces **− E** debe también denotar una expresión. La sustitución de una sola **E** por **− E** se describirá escribiendo lo siguiente:

E ⇒ − E

lo cual se lee como "**E** deriva a **− E** ". La producción **E → (E)** puede aplicarse para sustituir cualquier instancia de **E** en cualquier cadena de símbolos gramaticales por **(E)** por ejemplo,

E * E ⇒ (E) * E o E * E ⇒ E * (E). Podemos tomar una sola **E** y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

A dicha secuencia de sustituciones la llamamos una **derivación de −(id) a partir de E**. Esta derivación proporciona la prueba de que la cadena **−(id)** es una instancia específica de una expresión.

Para una definición general de la derivación, considere un no terminal **A** en la mitad de una secuencia de símbolos gramaticales, como en **αAβ**, en donde **α** y **β** son cadenas arbitrarias de símbolos gramaticales. Suponga que **A → γ** es una producción. Entonces, escribimos **αAβ ⇒ αγβ**. El símbolo **⇒** significa, "**se deriva en uno o más pasos**".

Cuando una secuencia de pasos de derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ se rescribe como α_1 a α_n, decimos que α_1 deriva a α_n.

Con frecuencia es conveniente poder decir, "**deriva en cero o más pasos**". Para este fin, podemos usar el símbolo $\overset{*}{\Rightarrow}$

Así,

1. $\boldsymbol{\alpha} \overset{*}{\Rightarrow} \boldsymbol{\alpha}$, para cualquier cadena α.

2. Si $\boldsymbol{\alpha} \overset{*}{\Rightarrow} \boldsymbol{\beta}$ y $\boldsymbol{\beta} \Rightarrow \boldsymbol{\gamma}$, entonces $\boldsymbol{\alpha} \overset{*}{\Rightarrow} \boldsymbol{\gamma}$.

Si $\mathbf{S} \overset{*}{\Rightarrow} \boldsymbol{\alpha}$, en donde **S** es el símbolo inicial de una gramática **G**, decimos que $\boldsymbol{\alpha}$ es una forma de frase de **G**. Observe que una forma de frase puede contener tanto terminales como no terminales, y puede estar vacía. Un **enunciado** de **G** es una forma de frase **sin símbolos no terminales**.

El lenguaje generado por una gramática es su conjunto de "oraciones". Por ende, una cadena de terminales **w** está en **L(G),** el lenguaje generado por **G**, **si y sólo si w es un enunciado de G (o** $\mathbf{S} \overset{*}{\Rightarrow} \mathbf{w}$**)**

Un lenguaje que puede generarse mediante una gramática se considera un lenguaje libre de contexto. Si dos gramáticas generan el mismo lenguaje, se consideran como equivalentes.

La cadena **−(id + id)** es un enunciado de la gramática

$$\mathbf{E} \rightarrow \mathbf{E + E \mid E * E \mid - E \mid (E) \mid id}$$

ya que hay una derivación:

$$\mathbf{E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)} \qquad \mathbf{(1)}$$

Las cadenas **E**, **−E**, **−(E),** …, **−(id + id)** son todas formas de frases de esta gramática. Escribimos $\mathbf{E} \overset{*}{\Rightarrow} \mathbf{-(id + id)}$ para indicar que **−(id + id)** puede derivarse de **E**.

En cada paso de una derivación, hay dos elecciones por hacer. Debemos elegir qué **no terminal** debemos sustituir, y habiendo realizado esta elección, debemos elegir una producción con ese no terminal como encabezado. Por ejemplo, la siguiente derivación alternativa de **−(id + id)** difiere de la derivación **(1)** en los últimos dos pasos:

$$\mathbf{E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)} \qquad \mathbf{(2)}$$

Cada no terminal se sustituye por el mismo cuerpo en las dos derivaciones, pero el orden de las sustituciones es distinto.

Para comprender la forma en que trabajan los analizadores sintácticos, debemos considerar las derivaciones en las que el no terminal que se va a sustituir en cada paso se elige de la siguiente manera:

1. **En las derivaciones por la izquierda**, siempre se elige el no terminal por la izquierda en cada de frase. Si $\alpha \Rightarrow \beta$ es un paso en el que se sustituye el no terminal por la izquierda en α, escribimos $\alpha \underset{lm}{\Rightarrow} \beta$.

2. **En las derivaciones por la derecha**, siempre se elige el no terminal por la derecha; en este caso escribimos $\alpha \underset{rm}{\Rightarrow} \beta$

La derivación $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$ es por la **izquierda**, por lo que puede rescribirse de la siguiente manera:

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E)) \underset{lm}{\Rightarrow} -(id+id)$$

Mientras que $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$ es una derivación por la **derecha**.

Si utilizamos nuestras convenciones de notación, cada paso por la izquierda puede escribirse $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$ en donde **w** consiste sólo de terminales, $A \rightarrow \delta$ es la producción que se aplica, y γ es una cadena de símbolos gramaticales. Para enfatizar que α deriva a β mediante una derivación por la izquierda, escribimos $\alpha \underset{lm}{\Rightarrow} \beta$.

Si $S \underset{lm}{\Rightarrow} \alpha$, decimos que α es una forma de frase izquierda de la gramática en cuestión.

Si análogas definiciones son válidas para las derivaciones por la derecha. A estas derivaciones se les conoce algunas veces como **derivaciones canónicas**.

Árboles de análisis sintáctico y derivaciones

Existe una representación de árbol para las derivaciones que ha demostrado ser extremadamente útil. Este árbol muestra claramente cómo se agrupan los símbolos de

una cadena terminal en subcadenas, que pertenecen al lenguaje de una de las variables de la gramática. Pero lo más importante es que el árbol, conocido como "**árbol de derivación**", cuando se emplea en un compilador, es la estructura de datos que representa el programa fuente.

En un compilador, la estructura del árbol del programa fuente facilita la traducción del programa fuente a código ejecutable permitiendo que el proceso de traducción sea realizado por funciones naturales recursivas.

En esta sección vamos a presentar los árboles de derivación y a demostrar que están estrechamente ligados a la existencia de las derivaciones y las inferencias recursivas. Posteriormente, estudiaremos la cuestión de la ambigüedad en las gramáticas y lenguajes, la cual constituye una importante aplicación de los árboles de derivación.

Ciertas gramáticas permiten que una cadena terminal tenga más de un árbol de análisis. Esta situación hace que esa gramática sea inadecuada para un lenguaje de programación, ya que el compilador no puede decidir la estructura sintáctica de determinados programas fuentes y, por tanto, no podría deducir con seguridad cuál será el código ejecutable apropiado correspondiente al programa.

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. **Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción**. El nodo interior se etiqueta con el no terminal **A** en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta **A** durante la derivación.

Por ejemplo, el árbol de análisis sintáctico para **−(id + id)** en la Figura 2 resulta de la derivación **(1)**, así como de la derivación **(2)**.

Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama producto o frontera del árbol.

Para ver la relación entre las derivaciones y los árboles de análisis sintáctico, considere cualquier derivación **α1 ⇒ α2⇒ … ⇒ αn**, en donde **α1** es un sólo no terminal **A**. Para cada forma de frase **αi** en la derivación, podemos construir un árbol de análisis sintáctico cuyo producto sea **αi**. El proceso es una inducción sobre **i**.

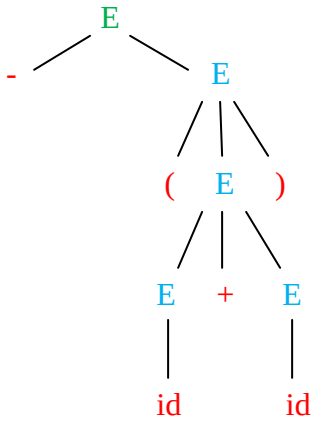


Figura 2: Árbol de análisis sintáctico para **−(id + id)**

Ejemplo 3: La secuencia de árboles de análisis sintáctico que se construyen a partir de la derivación **(1)** se muestra en la Figura 2. En el primer paso de la derivación, **E** ⇒ **−E**. Para modelar este paso, se agregan dos hijos, etiquetados como **−** y **E**, a la raíz **E** del árbol inicial. El resultado es el segundo árbol.

En el segundo paso de la derivación, **−E** ⇒ **−(E)**. Por consiguiente, agregamos tres hijos, etiquetados como **(**, **E** y **)**, al nodo hoja etiquetado como **E** del segundo árbol, para obtener el tercer árbol con coséchale producto **−(E)**. Si continuamos de esta forma, obtenemos el árbol de análisis sintáctico completo como el sexto árbol.

Como un árbol de análisis sintáctico ignora las variaciones en el orden en el que se sustituyen los símbolos en las formas de las oraciones, hay una relación de varios a uno entre las derivaciones y los árboles de análisis sintáctico. Por ejemplo, ambas derivaciones **(1)** y **(2)** se asocian con el mismo árbol de análisis sintáctico final de la Figura 3.

En lo que sigue, realizaremos con frecuencia el análisis sintáctico produciendo una derivación por la izquierda o por la derecha, ya que hay una relación de uno a uno entre los árboles de análisis sintáctico y este tipo de derivaciones. Tanto las derivaciones por la izquierda como las de por la derecha eligen un orden específico para sustituir símbolos en las formas de las oraciones, por lo que también filtran las variaciones en orden. No es difícil mostrar que todos los árboles sintácticos tienen asociadas una derivación única por la izquierda y una derivación única por la derecha.

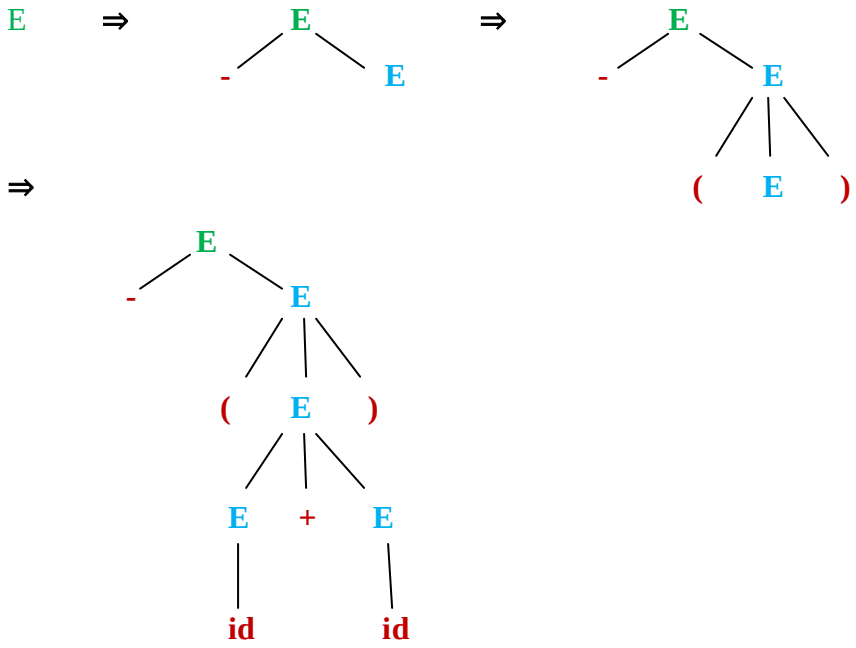


Figura 3: Secuencia de árboles de análisis sintáctico para la derivación **(1)**

En resumen, si tenemos una gramática **G = (N,T,P,S)**. Los árboles de derivación para **G** son aquellos árboles que cumplen las condiciones siguientes:

1. Cada nodo interior está etiquetado con una variable de **N**.

2. Cada hoja está etiquetada bien con una variable, un símbolo terminal o **ε** . Sin embargo, si la hoja está etiquetada con **ε** , entonces tiene que ser el único hijo de su padre.

3. Si un nodo interior está etiquetado como **A** y sus hijos están etiquetados como: **X1,X2, . . . ,Xk** respectivamente, comenzando por la izquierda, entonces **A→X1X2 · · Xk** es una producción de **P**. Observe que el único caso en que una de las **X** puede reemplazarse por **ε** es cuando es la etiqueta del único hijo y **A→ε** es una producción de **G**.

Ambigüedad

Una gramática que produce más de un árbol de análisis sintáctico para cierto enunciado es ambigua. Dicho de otra forma, una gramática ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

Ejemplo 4: La gramática de expresiones aritméticas:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id \qquad (3)$$

permite dos derivaciones por la izquierda distintas para el enunciado **id + id * id**:

E ⇒ E + E
⇒ id + E
⇒ id + E * E
⇒ id + id * E
⇒ id + id * id

E ⇒ E * E
⇒ E + E * E
⇒ id + E * E
⇒ id + id * E
⇒ id + id * id

Los árboles de análisis sintáctico correspondientes aparecen en la Figura 4.

Observe que el árbol de análisis sintáctico de la Figura 4(a) refleja la precedencia que se asume comúnmente para + y *, mientras que el árbol de la Figura 4(b) no. Es decir, lo común es tratar al operador * teniendo mayor precedencia que +, en forma correspondiente al hecho de que, por lo general, evaluamos la expresión **a + b * c** como **a + (b * c)**, en vez de hacerlo como **(a + b) * c**.

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única qué árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con reglas para eliminar la ambigüedad, las cuales "descartan" los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.

Figura 4: Dos árboles de análisis sintáctico para **id+id*id**

Factorización por la izquierda

La factorización por la izquierda es una transformación gramátical, útil para producir una gramática adecuada para el análisis sintáctico predictivo, o descendente. Cuando la elección entre dos producciones **A** alternativas no está clara, tal vez podamos rescribir las producciones para diferir la decisión hasta haber visto la suficiente entrada como para poder realizar la elección correcta.

En general, si **A → αβ1 | αβ2** son dos producciones **A**, y la entrada empieza con una cadena no vacía derivada de **α**, no sabemos si debemos expandir **A** a **αβ1** o a **αβ2**. No obstante, podemos diferir la decisión si expandimos **A** a **αA'**. Así, después de ver la entrada derivada de **α**, expandimos **A'** a **β1** o a **β2**. Es decir, si se factorizan por la izquierda, las producciones originales se convierten en:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta 1 \mid \beta 2$$

Recursividad por la izquierda

Una gramática es recursiva por la izquierda si tiene un no terminal **A** tal que haya una derivación $A \Rightarrow A\alpha$ para cierta cadena **α**. Los métodos de análisis sintáctico descendentes no pueden manejar las gramáticas recursivas por la izquierda, por lo que se necesita una transformación para eliminar la recursividad por la izquierda.

El caso general de recursividad por la izquierda se puede mostrar con producciones del tipo: $A \rightarrow A\alpha \mid \beta$

La cual puede sustituirse mediante las siguientes producciones no recursivas por la izquierda:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

sin cambiar las cadenas que se derivan de **A**. Esta regla por sí sola basta para muchas gramáticas.

Ejemplo: La gramática de expresiones no recursivas por la izquierda:

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow *\ F\ T'$$

$$F \rightarrow (\ E\ ) \mid id$$

se obtiene mediante la eliminación de la recursividad inmediata por la izquierda de la gramática de expresiones:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (\ E\ ) \mid id$$

El par recursivo por la izquierda de las producciones $E \rightarrow E + T \mid T$ se sustituye mediante $E \rightarrow T\ E'$ y $E' \rightarrow +\ T\ E' \mid \varepsilon$.

Las nuevas producciones para **T** y **T'** se obtienen de manera similar, eliminando la recursividad inmediata por la izquierda.

Gramáticas LL(1)

Los analizadores sintácticos predictivos, es decir, los analizadores sintácticos de descenso recursivo, pueden construirse para una clase de gramáticas llamadas **LL(1).** La primera "**L**" en **LL(1)** es para explorar la entrada de izquierda a derecha (por left en inglés), la segunda "**L**" para producir una derivación por la izquierda, y el "**1**" para usar un símbolo de entrada de anticipación en cada paso, para tomar las decisiones de acción del análisis sintáctico.

La clase de gramáticas **LL(1)** es lo bastante robusta como para cubrir la mayoría de las construcciones de programación, aunque hay que tener cuidado al escribir una gramática adecuada para el lenguaje fuente. Por ejemplo, **ninguna gramática recursiva por la izquierda** o **ambigua** puede ser **LL(1).**

Análisis sintáctico predictivo no recursivo

Podemos construir un analizador sintáctico predictivo no recursivo mediante el mantenimiento explícito de una pila.

El analizador sintáctico imita una derivación por la izquierda. Si **w** es la entrada que se ha relacionado hasta ahora, entonces la pila contiene una secuencia de símbolos gramaticales **α** de tal forma que:

$$S \underset{lm}{\Rightarrow} w\alpha,$$

El analizador sintáctico controlado por una tabla, que se muestra en la figura 5, tiene una **entrada**, una **pila** que contiene una secuencia de símbolos gramaticales, una **tabla** de análisis sintáctico, y un **flujo de salida**.

La entrada contiene la cadena que se va a analizar, seguida por el marcador final **$**. Reutilizamos el símbolo **$** para marcar la parte inferior de la pila, que al principio contiene el símbolo inicial de la gramática encima de **$**.

El analizador sintáctico se controla mediante un programa que considera a **X**, el símbolo en la **parte superior de la pila**, y a **a**, el símbolo de entrada actual. Si **X** es un **no terminal**, el analizador sintáctico elige una producción **X** mediante una consulta a la entrada **M[X, a]** de la tabla de análisis sintáctico **M** (aquí podría ejecutarse código adicional; por ejemplo, el código para construir un nodo en un árbol de análisis sintáctico). En cualquier otro caso, verifica si hay una coincidencia entre el terminal **X** y el símbolo de entrada actual **a**.

El comportamiento del analizador sintáctico puede describirse en términos de sus configuraciones, que proporcionan el contenido de la pila y el resto de la entrada.
El siguiente algoritmo describe la forma en que se manipulan las configuraciones.

ALGORITMO: Análisis sintáctico predictivo, controlado por una tabla.
ENTRADA: Una cadena **w** y una **tabla de análisis sintáctico M para la gramática G.**
SALIDA: Si **w** está en **L(G)**, una derivación por la izquierda de **w**; en caso contrario, una indicación de error.

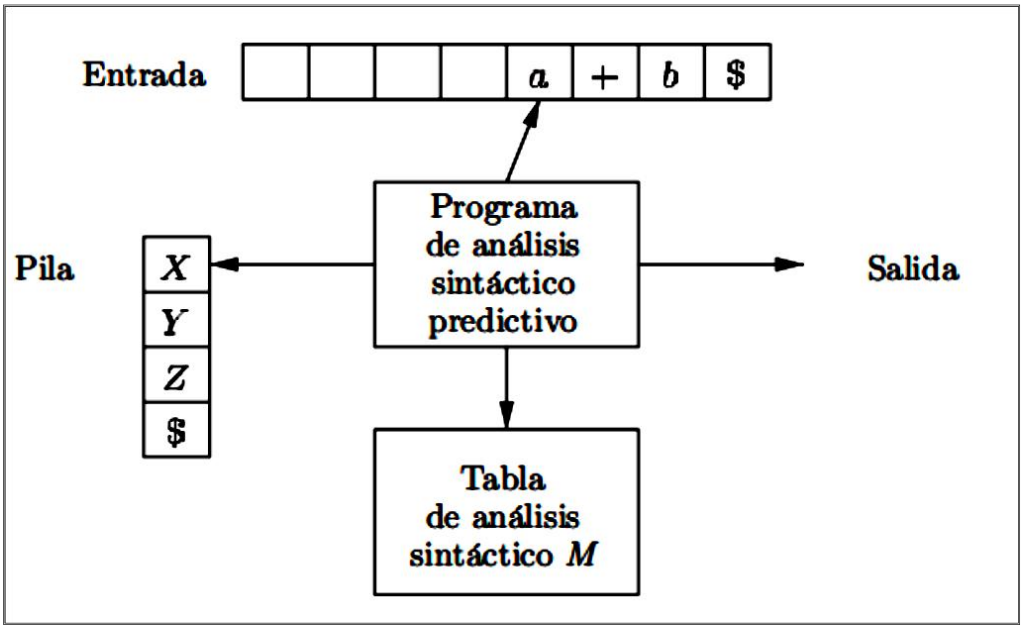


Figura 5: Modelo de un analizador sintáctico predictivo, controlado por una tabla

Para la gramática de expresiones:

E → T E'

E' → + T E' | ε

T → F T'

T' → * F T' | ε

F → (E) | id

Se tiene la tabla de análisis sintáctico en la figura 6. Los espacios en blanco son entradas de error; los espacios que no están en blanco indican una producción con la cual se expande un no terminal.

No terminal	Símbolo de entrada					
	id	**+**	*****	**(**	**)**	**$**
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F → (E)		

Figura 6: Tabla de análisis sintáctico

(La construcción de esta tabla puede encontrarse en la bibliografía)

MÉTODO: Al principio, el analizador sintáctico se encuentra en una configuración con w$ en el búfer de entrada, y el símbolo inicial **S** de **G** en la parte superior de la pila, por encima de **$**.

El programa en la figura 7 utiliza la tabla de análisis sintáctico predictivo **M** para producir un análisis sintáctico predictivo para la entrada.

establecer **ip** para que apunte al primer símbolo de w;
establecer X con el símbolo de la parte superior de la pila;
while (X ≠ $) { /* la pila no está vacía */
    if (X es **a**) sacar de la pila y avanzar ip;
    else if (X es un terminal) error();
    else if (M[X, a] es una entrada de error) error();
    else if (M[X, a] = X → Y1Y2 …Yk) {
        enviar de salida la producción X → Y1Y2 … Yk;
        sacar de la pila;
        meter Yk, Yk−1, …, Y1 en la pila, con Y1 en la parte superior;
    }
    establecer X con el símbolo de la parte superior de la pila;
}

Figura 7: Algoritmo de análisis sintáctico predictivo

Ejemplo: Considere la gramática para expresiones vista; se muestra la tabla de análisis sintáctico en la figura 6. Con la entrada **id + id * id**, el analizador predictivo sin

recursividad realiza la secuencia de movimientos en la figura 8. Estos movimientos corresponden a una derivación por la izquierda para la derivación completa:

Pila	Entrada	Salida
$E	id + id * id$	
$E' T	id + id * id$	E→ T E'
$E' T' F	id + id * id$	T→ F T'
$E' T' id	id + id * id$	F→ id
$E' T'	+ id * id$	
$E'	+ id * id$	T'→ ε
$E' T +	+ id * id$	E'→ + TE'
$E' T	id * id$	
$E' T' F	id * id$	F→ FT'
$E' T' id	id * id$	F→ id
$E' T'	* id$	
$E' T' F *	*id$	T'→ *FT'
$E' T' F	id$	
$E' T' id	id$	F→ id
$E' T'	$	
$E'	$	T'→ ε
$	$	E'→ ε

Figura 8: Movimientos que realiza un analizador sintáctico predictivo con la entrada **id + id * id**

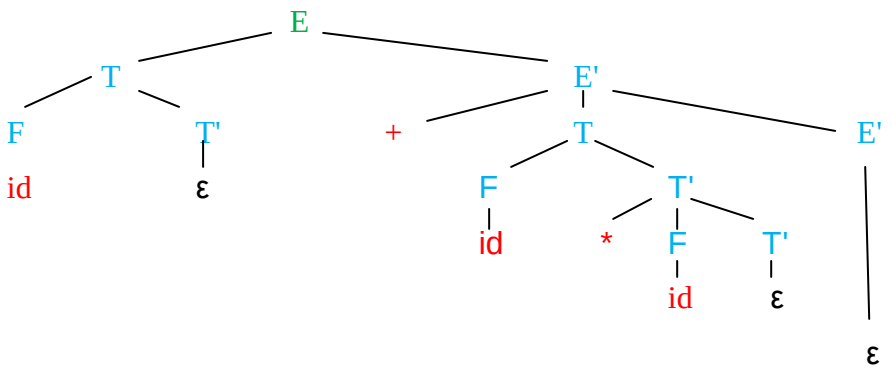


Figura 9: Arbol sintáctico construido por el analizador sintáctico del ejemplo

Observe que en el árbol sintáctico de la figura 9, las hojas corresponden a la cadena de entrada o bien a la cadena vacía. La raiz del árbol es el símbolo inicial de la gramática.

Análisis sintáctico ascendente

El tipo más frecuente de analizador sintáctico ascendentes en la actualidad se basa en un concepto conocido como análisis sintáctico **LR(k)**; la "**L**" indica la exploración de izquierda a derecha de la entrada, la "**R**" indica la construcción de una **derivación por la derecha**, y la **k** para el número de símbolos de entrada de preanálisis que se utilizan al hacer decisiones del análisis sintáctico. El caso k = 1 es de interés práctico, por lo que aquí sólo consideraremos los analizadores sintácticos LR con k = 1. Cuando se omite (k), se asume que k es 1.

Los analizadores sintácticos **LR son controlados por tablas, en forma muy parecida a los analizadores sintácticos LL** no recursivos visto anteriormente. Para que una gramática sea **LR**, basta con poder construir un analizador sintáctico de **desplazamiento-reducción** de izquierda a derecha que pueda reconocer "***mangos***" de las formas de frases derechas, cuando éstas aparecen en la parte superior de la pila.

El análisis sintáctico LR es atractivo por una variedad de razones:

• Pueden construirse analizadores sintácticos LR para reconocer prácticamente todas las construcciones de lenguajes de programación para las cuales puedan escribirse gramáticas libres de contexto.

• El método de análisis sintáctico LR es el método de análisis sintáctico de desplazamiento - reducción más general que se conoce a la fecha.

• Un analizador sintáctico LR puede detectar un error sintáctico tan pronto como sea posible en una exploración de izquierda a derecha de la entrada.

• La clase de gramáticas que pueden analizarse mediante los métodos **LR** es un **superconjunto** propio de la clase de gramáticas que pueden analizarse con métodos predictivos o **LL**.

Para que una gramática sea **LR(k)**, debemos ser capaces de reconocer la ocurrencia del lado derecho de una producción en una forma de frase derecha, con **k** símbolos de entrada de preanálisis. Este requerimiento es mucho menos estricto que para las gramáticas **LL(k)**, en donde debemos ser capaces de reconocer el uso de una producción, viendo sólo los primeros símbolos **k** de lo que deriva su lado derecho. Por

ende, no debe sorprender que las **gramáticas LR puedan describir más lenguajes** que las gramáticas **LL**.

La **principal desventaja** del método **LR** es que es demasiado trabajo construir un analizador sintáctico **LR** en forma manual para una gramática común de un lenguaje de programación.

Se necesita una herramienta especializada: un generador de analizadores sintácticos **LR**. Hay varios generadores disponibles, uno se los que se utilizan con más frecuencia es **Yacc**. (*) Este generador recibe una gramática libre de contexto y produce de manera automática un analizador para esa gramática. Si la gramática contiene ambigüedades u otras construcciones que sean difíciles de analizar en una exploración de izquierda a derecha de la entrada, entonces el generador de analizadores sintácticos localiza estas construcciones y proporciona mensajes de diagnóstico detallados.

(*) **Yacc** es un programa para generar analizadores sintácticos. Las siglas del nombre significan *Yet Another Compiler-Compiler*, es decir, "Otro generador de compiladores más". Genera un analizador sintáctico (la parte de un compilador que comprueba que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje) basado en una notación similar a la BNF. Yacc genera el código para el analizador sintáctico en el lenguaje de programación "C"

Un análisis sintáctico ascendente corresponde a la construcción de un árbol de análisis sintáctico para una cadena de entrada **que empieza en las hojas** (la parte inferior) y avanza hacia **la raíz** (la parte superior).

Es conveniente describir el análisis sintáctico como el proceso de construcción de árboles de análisis sintáctico, aunque de hecho un *front-end* de usuario podría realizar una traducción directamente, sin necesidad de construir un árbol explícito.

La secuencia

id * id

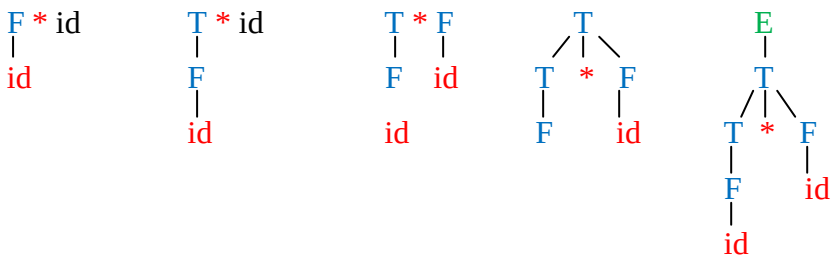


Figura 10 Un análisis sintáctico ascendente para **id * id**

La figura 10 ilustra un análisis sintáctico ascendente del flujo de tokens **id * id**, con respecto a la gramática de expresiones

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id \qquad \textbf{(4)}$$

Esta sección presenta un estilo general de análisis sintáctico ascendente, conocido como análisis sintáctico de **desplazamiento-reducción**. Luego hablaremos sobre las gramáticas **LR**, la clase más extensa de gramáticas para las cuales pueden construirse los analizadores sintácticos de **desplazamiento-reducción**.

Reducciones

Podemos considerar el análisis sintáctico ascendente como el proceso de "**reducir**" una cadena **w** al **símbolo inicial de la gramática**. En cada paso de reducción, se sustituye una subcadena específica que coincide con el cuerpo de una producción por el no terminal que se encuentra en el encabezado de esa producción.

Las decisiones clave durante el análisis sintáctico ascendente son acerca de cuándo reducir y qué producción aplicar, a medida que procede el análisis sintáctico.

Durante una exploración de izquierda a derecha de la entrada, el análisis sintáctico ascendente construye una derivación por la derecha en forma inversa. De manera informal, un **"mango"** es una subcadena que coincide con el cuerpo de una producción, y cuya reducción representa un paso a lo largo del inverso de una derivación por la derecha.

Por ejemplo, si agregamos subíndices a los ***tokens id*** para mejorar la legibilidad, los mangos durante el análisis sintáctico de $\mathbf{id_1 * id_2}$, de acuerdo con la gramática de expresiones **(4)** son como en la figura 11. Aunque **T** es el cuerpo de la producción **E → T**, el símbolo **T** no es un mango en la forma de frase $\mathbf{T * id_2}$. Si **T** se sustituyera por **E**, obtendríamos la cadena $\mathbf{E * id_2}$, lo cual no puede derivarse del símbolo inicial **E**. Por ende, la subcadena por la izquierda que coincide con el cuerpo de alguna producción no necesita ser un mango.

Forma de Frase Dereca	Mango	Reducción de la Producción
$id_1 * id_2$	id_1	F → id
F * id_2	F	T → F
T * id_2	id_2	F → id
T * F	T * F	E → T * F

Figura 11: Mangos durante un análisis sintáctico de **$id_1 * id_2$**

Análisis sintáctico por desplazamiento-reducción

El análisis sintáctico de desplazamiento-reducción es una forma de análisis sintáctico ascendente, en la cual una pila contiene símbolos gramaticales y un búfer de entrada contiene el resto de la cadena que se va a analizar. Como veremos, el mango siempre aparece en la parte superior de la pila, justo antes de identificarla como el mango.

Utilizamos el **$** para marcar la parte inferior de la pila y también el extremo derecho de la entrada. Por convención, al hablar sobre el análisis sintáctico ascendente, mostramos la parte superior de la pila a la derecha.

Al principio la pila está vacía, y la cadena **w** está en la entrada

Pila	Entrada	Acción
$	**$id_1 * id_2$**	Desplazar
$ **id_1**	**$* id_2$**	Reducir F → id
$F	**$* id_2$**	Reducir T → F
$T	**$* id_2$**	Desplazar
$T **	**id_2**	Desplazar
$T ** **$id_2$**	$	Reducir F → id
$T * F	$	Reducir T → T * F
$T	$	Reducir E → T
$E	$	Aceptar

Figura 12: Configuraciones de un analizador sintáctico de desplazamiento-reducción, con una entrada **id1 * id2**

Aunque las operaciones primarias son desplazar y reducir, en realidad hay cuatro acciones posibles que puede realizar un analizador sintáctico de desplazamiento-reducción:
(1) desplazar,
(2) reducir,
(3) aceptar
(4) error.

1. ***Desplazar***. Desplazar el siguiente símbolo de entrada y lo coloca en la parte superior de la pila.
2. ***Reducir***. El extremo derecho de la cadena que se va a reducir debe estar en la parte superior de la pila. Localizar el extremo izquierdo de la cadena dentro de la pila y decidir con qué terminal se va a sustituir la cadena.
3. ***Aceptar*.** Anunciar que el análisis sintáctico se completó con éxito.
4. ***Error*.** Descubrir un error de sintaxis y llamar a una rutina de recuperación de errores.

El uso de una pila en el análisis sintáctico de desplazamiento-reducción se justifica debido a un hecho importante: **el mango siempre aparecerá en algún momento dado en la parte superior de la pila, nunca en el interior.**

Algoritmos de análisis sintáctico LR

El análisis LR, se basa en tablas de análisis sintáctico, para construir estas tablas existen tres métodos:
1) El más sencillo se denomina **SLR**, es el más fácil de implementar, pero el menos poderoso.
2) El segundo se denomina **LR canónico**, es el más poderoso y costoso
3) El tercero se denomina LR con examen anticipado (**LALR**) y está entre los dos anteriores en cuanto a poder y costo.
El método **LALR** funciona con las gramáticas de la mayoría de los lenguajes de programación, y con poco esfuerzo se puede implementar de forma sencilla.

En la figura 13: se muestra un diagrama de un analizador sintáctico LR. Este diagrama consiste en una entrada, una salida, una pila, un programa controlador y una tabla de análisis sintáctico que tiene dos partes (ACCION y el ir_A). El programa controlador es

igual para todos los analizadores sintácticos LR; sólo la tabla de análisis sintáctico cambia de un analizador sintáctico a otro. El programa de análisis sintáctico lee caracteres de un búfer de entrada, uno a la vez. En donde un analizador sintáctico de desplazamiento-reducción desplazaría a un símbolo, un analizador sintáctico LR desplaza a un estado. Cada estado sintetiza la información contenida en la pila, debajo de éste.

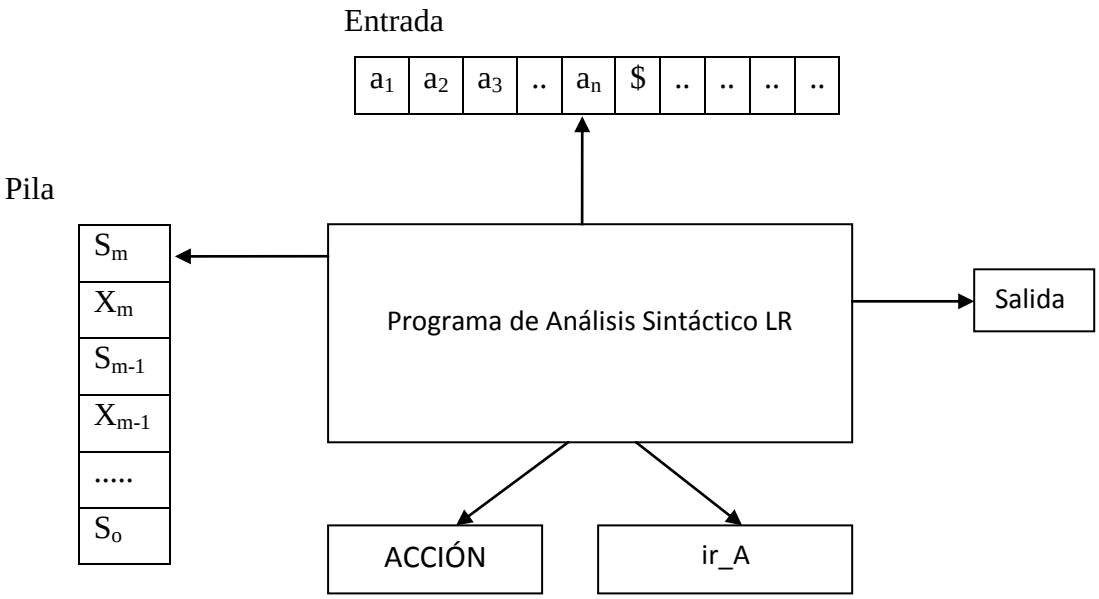


Figura 13: Modelo de un analizador sintáctico LR

El programa de control lee la cadena de entrada de uno en un caracter. En la pila se almacenan pares de X_m S_m donde S_m es un símbolo llamado "estado" y X_m es un *símbolo gramatical* (terminal o no terminal).

Se usan el símbolo y estado de la cima de la pila en combinacíon con el caracter de entrada para indexar la tabla de análisis sintáctico del analizador y determinar que acción tomar, si desplazamiento o reducción.

La acciones que puede tomar el analizador, de la tabla de análisis es una de las cuatro siguientes:

1. Desplazar S, donde S es un estado
2. Reducir por una producción gramatical $A \rightarrow \boldsymbol{\beta}$
3. Aceptar
4. Error

En resumen el algoritmo LR consta de:

Entrada: Una cadena de entrada **w** y una tabla de análisis sintáctico **LR** con las funciones de **ir_A** y **Acción** para una gramática **G.**

Salida: Si **w** está el **L(G)** un análisis sintáctico de **w**, de lo contrario se tiene una condición de error.

Método: Inicialmente S_0 está en la pila del analizador donde S_0 es el estado inicial y **w** está en el *buffer* de entrada. El analizador ejecuta el siguiente programa hasta llegar a un estado de aceptación o de error.

Apunta un puntero (por ejemplo **p**) al primer símbolo se ***w$***

```
for(; ;) {
        sea S el estado de la cima de la pila y a el apuntado por p
        if ( Acción [S,a] = desplazar S' ) {
                meter a y despues S' en la cima de la pila;
                avanzar p al siguiente símbolo de entrada;
        }
        else if ( Acción [S,a] = reducir A → β ) {
                sacar 2*|β| símbolos de la pila;
                sea S' el estado ahora en la cima de la pila meter A y despúes ir_A[S'A]
                en la    cima de la pila, emitir la producción A → β
        }
        else if ( Acción [S,a] = aceptar ) return;
        else error():
}
```

Ejemplo:

Tenemos la siguiente gramática para operadores binarios **+** y *****

(1) E → E + T

(2) E → T

(3) T → T * F

(4) T → F

(5) F → (E)

(6) F → id

La tabla de análisis sintáctico para un analizador LR es:

Estado	Acción						Ir_A		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	D5			D4			1	2	3
1		D6				aceptar			
2		R2	D7		R2	R2			
3		R4	R4		R4	R4			
4	D5			D4			8	2	3
5		R6	R6		R6	R6			
6	D5			D4				9	3
7	D5			D4					10
8		D6			D11				
9		R1	D7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Di significa desplazar y meter en la pila en estado **i**.

Rj significa reducir por la producción con número **j**.

Ejemplo para la entrada **id * id + id$**

Pila	Entrada	Acción
(1) 0	id * id + id$	Desplazar 5
(2) 0 id 5	* id + id$	Reducir por (6) F → id
(3) 0 F 3	* id + id$	Reducir por (4) T → F
(4) 0 T 2	* id + id$	Desplazar 7
(5) 0 T 2 * 7	id + id$	Desplazar 5
(6) 0 T 2 * 7 id 5	+ id$	Reducir por (6) F → id
(7) 0 T 2 * 7 F 10	+ id$	Reducir por (3) T → T * F
(8) 0 T 2	+ id$	Reducir por (2) E → T
(9) 0 E 1	+ id$	Desplazar 6
(10) 0 E 1 + 6	id$	Desplazar 5
(11) 0 E 1 + 6 id 5	$	Reducir por (6) F → id
(12) 0 E 1 + 6 F 3	$	Reducir por (4) T → F
(13) 0 E 1 + 6 T 9	$	Reducir por (1) E → E + T
(14) 0 E 1	$	Aceptar

Con la entrada **id * id + id$** la secuencia que se sigue es en (1) el analizador está en el estado 0 (cero) y la entrada en **id**, la acción en la fila 0, columna **id** de la tabla de análisis sintáctico es **D5**, es lo que en se ve en (2), donde se han desplazado a la pila **id** y el estado **5**.

***** es ahora el símbolo de entrada y de la tabla se obtiene **R6**, es decir reducir por (6) **F** → **id**, se extraen dos símbolos de la pila **(2* |id|)** quedando el estado **0** en el tope de la pila y se ve **ir_A[0,F]** que es **3**, por lo que se introduce **F** y **3** en la pila, lo que se ve en la fila (3). Los demás movimientos se realizan de la misma forma.

Aplicaciones de las gramáticas independientes del contexto

Las gramáticas independientes del contexto originalmente fueron concebidas por **N**. **Chomsky** como una forma de describir los lenguajes naturales. Pero esta posibilidad no ha llegado a cumplirse. Sin embargo, a medida que en las Ciencias de la Computación el uso de conceptos definidos recursivamente se ha multiplicado, se ha tenido la necesidad de emplear las **GIC** como una forma de describir instancias de estos conceptos. Vamos a describir ahora dos de estos usos, uno antiguo y otro nuevo.

1. Las gramáticas se utilizan para describir lenguajes de programación. Lo más importante es que existe una forma mecánica de convertir la descripción del lenguaje como GIC en un analizador sintáctico, el componente del compilador que descubre la estructura del programa fuente y representa dicha estructura mediante un árbol de derivación. Esta aplicación constituye uno de los usos más tempranos de las GIC; de hecho, es una de las primeras formas en las que las ideas teóricas de las Ciencias de la Computación pudieron llevarse a la práctica.

2. El desarrollo del XML (Extensible Markup Language) facilita el comercio electrónico permitiendo a los participantes compartir convenios, independientemente de los pedidos, las descripciones de los productos y de otros muchos tipos de documentos. Una parte fundamental del XML es la DTD (Document Type Definition, definición de tipo de documento), que principalmente **es una gramática independiente del contexto** que describe las etiquetas permitidas y las formas en que dichas etiquetas pueden anidarse. Las etiquetas son las palabras clave encerradas entre corchetes triangulares como en HTML, como por ejemplo, <EM> y </EM> para indicar que el texto que encierran tiene que escribirse en cursiva. **Sin embargo, las etiquetas XML no se ocupan de dar formato al texto, sino del significado del mismo**. Por ejemplo, la pareja de etiquetas XML <TELEFONO> y </TELEFONO> marcaría que la secuencia de caracteres encerrada entre ellas debe interpretarse como un número de teléfono.

Ejemplo de XML y las DTD

El hecho de que el HTML se describa mediante una gramática no es en sí extraño. Prácticamente todos los lenguajes de programación se pueden describir mediante sus

propias GIC, por tanto, lo que sí sería sorprendente es que no pudiéramos describir HTML.

Sin embargo, si examinamos otro importante lenguaje de marcado, el XML (eXtensible Markup Language), comprobaríamos que las GIC desempeñan un papel fundamental comoparte del proceso de uso de dicho lenguaje.

El objetivo del XML no es describir el formato del documento; ése es el trabajo del HTML. En lugar de ello, XML intenta describir la **"semántica"** del texto. Por ejemplo, un texto como "Arístides 12" parece una dirección, pero ¿lo es realmente? En XML, ciertas etiquetas encierran una frase que representa una dirección, por ejemplo:

<ADDR> Arístides 12</ADDR>

Sin embargo, no es evidente de forma inmediata que <ADDR> indique que se trata de una dirección. Por ejemplo, si el documento tratara sobre la asignación de memoria, podríamos pensar que la etiqueta <ADDR> hace referencia a una dirección de memoria. Con el fin de clarificar qué indican las distintas clases de etiquetas y qué estructuras pueden aparecer entre pares de estas etiquetas, se espera que las personas con intereses comunes desarrollen estándares en la forma de DTD (Document-Type Definition, definición de tipo de documento).

Una DTD es prácticamente una gramática independiente del contexto, con su propia notación para describir las variables y producciones.

En el siguiente ejemplo, veremos una DTD simple y presentaremos parte del lenguaje utilizado para describir las DTD. El lenguaje de DTD en sí tiene una gramática independiente del contexto, pero no estamos interesados en describir esa gramática. En lugar de ello, el lenguaje para describir las DTD es, en esencia, una notación de GIC y lo que deseamos ver es cómo se expresan las GIC en este lenguaje.

El formato de una DTD es:

<!DOCTYPE nombre-de-DTD [lista de definiciones de elementos]>

A su vez, una definición de elemento utiliza el formato

<!ELEMENT nombre-elemento (descripción del elemento)>

Las descripciones de los elementos son fundamentalmente **expresiones regulares**. Las bases de estas expresiones son:

1. Otros nombres de elementos, que representan el hecho de que los elementos de un tipo pueden aparecer dentro de elementos de otro tipo, al igual que en HTML podemos encontrar texto en cursiva dentro de una lista.

2. El término especial #PCDATA, que especifica cualquier texto que no utiliza etiquetas XML.
Los operadores permitidos son:

1. | para la operación de unión, como en la **notación de las expresiones regulares**
2. La coma indica la **concatenación**.
3. Tres variantes del operador de clausura, Éstas son *****, el operador habitual que especifica "**cero o más apariciones de**" ,**+**, que especifica "**una o más apariciones de**" y **?**, que indica "**cero o una aparición de**". (note la semejanza con los operadores anteriormente vistos en expresiones regulares).

Los paréntesis pueden agrupar los operadores con sus argumentos; en cualquier otro caso, se aplican las reglas de precedencia habituales de los operadores de las expresiones regulares.

Como ejemplo, imaginemos que una serie de distribuidores de computadoras se reúnen para crear una DTD estándar que utilizarán para publicar en la Web las descripciones de los distintos PC que venden. Cada descripción de un PC contendrá el número de modelo y los detalles acerca de las prestaciones del mismo, por ejemplo, la cantidad de RAM, el número y tamaño de los discos, etc. La Figura 14 muestra una hipotética y muy sencilla DTD para los PC.
El nombre de la DTD es PcSpecs. El primer elemento, que es como el **símbolo inicial de una GIC**, es PCS (lista de las especificaciones del PC). Su definición, **PC***, dice que un PCS son cero o más entradas PC.
A continuación encontramos la definición de un elemento PC. Consta de la concatenación de cinco cosas.
Las cuatro primeras son otros elementos que se corresponden con el modelo, el precio, el tipo de procesador y la RAM del PC. Cada uno de estos elementos tiene que aparecer una vez, en dicho orden, ya que la coma indica **concatenación**. El último elemento, DISK+, nos dice que existirán una o más entradas de disco para un PC.
Muchos de los elementos son simplemente textos; MODELO, PRECIO y RAM son de este tipo. Sin embargo, PROCESADOR tiene más estructura. Vemos a partir de su definición que consta de un fabricante, un modelo y la velocidad, en dicho orden, y cada uno de estos elementos son simplemente texto.

```
<!DOCTYPE PcSpecs [
        <!ELEMENT PCS (PC*)>
        <!ELEMENT PC (MODELO, PRECIO, PROCESADOR, RAM, DISCO+)>
        <!ELEMENT MODEL0 (#PCDATA)>
        <!ELEMENT PRECIO (#PCDATA)>
        <!ELEMENT PROCESADOR (FABRICANTE, MODELO, VELOCIDAD)>
        <!ELEMENT FABRICANTE (#PCDATA)>
        <!ELEMENT MODELO (#PCDATA)>
        <!ELEMENT VELOCIDAD (#PCDATA)>
        <!ELEMENT RAM (#PCDATA)>
        <!ELEMENT DISCO (DISCODURO | CD | DVD)>
        <!ELEMENT DISCODURO (FABRICANTE, MODELO, TAMAÑO)
        <!ELEMENT TAMAÑO (#PCDATA)>
        <!ELEMENT CD (VELOCIDAD)>
        <!ELEMENT DVD (VELOCIDAD)>
]>
```

Figura 14 Una DTD para computadoras personales.

```
<PCS>
        <PC>
                <MODEL0>4560</MODELO>
                <PRECIO>$22950</PRECIO>
                <PROCESADOR>
                        <FABRICANTE>Intel</FABRICANTE>
                        <MODEL0>Core I7 - 2600</MODELO>
                        <VELOCIDAD>3.4GHz</VELOCIDAD>
                </PROCESADOR>
                        <RAM>16 Gb</RAM>
                <DISCO><DISCODURO>
                        <FARBICANTE>Maxtor</FABRICANTE>
                        <MODELO>Diamond</MODELO>
                        <TAMAÑO>1 Tb</TAMAÑO>
                </DISCODURO></DISCO>
                        <DISCO><CD>
                        <VELOCIDAD>32x</VELOCIDAD>
                </CD></DISCO>
        </PC>
        <PC>
        ...
        </PC>
</PCS>
```

Figura 15. Parte de un documento que obecede a la estructura de la DTD mostrada en la Figura 14

Es posible que haya observado que las reglas para los elementos de las DTD como la mostrada en la Figura 14 no se parecen a las producciones de las gramáticas independientes del contexto. Muchas de las reglas tienen un formato similar. Por ejemplo,

<!ELEMENT PROCESADOR (FABRICANTE, MODELO, VELOCIDAD)>
es similar a la producción:

Procesador→Fabricante Modelo Velocidad

Sin embargo, la regla:

<!ELEMENT DISCO (DISCODURO | CD | DVD)>

no tiene una definición para DISCO que sea parecida al cuerpo de una producción. En este caso, la extensión es simple: podemos interpretar esta regla como tres producciones, con las barras verticales desempeñandoel mismo papel que las abreviaturas en las producciones que tienen una cabeza común. Así, esta regla es equivalente a las tres producciones

Disco→DiscoDuro | Cd | Dvd

El caso más complicado es:

<!ELEMENT PC (MODELO, PRECIO, PROCESADOR, RAM, DISCO+)>

donde el "cuerpo" incluye un operador de clausura. La solución consiste en sustituir DISCO+ por una nueva variable, por ejemplo **Discos**, que genere, a través de un par de producciones, una o más instancias de la variable Disco. Las producciones equivalentes son por tanto:

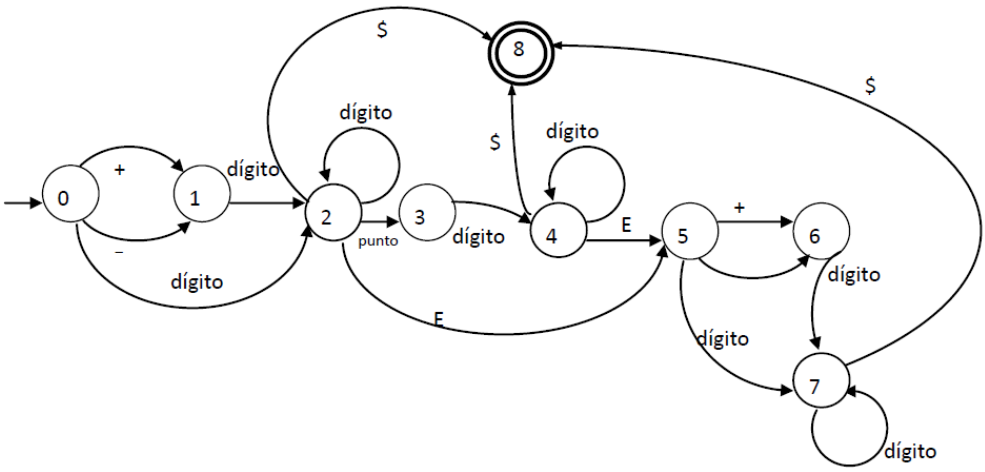
Pc→Modelo Precio Procesador Ram Discos

Discos→Disco | Disco Discos

Si bien hoy en día XML ha sido reemplazado por lenguajes como JSON, este tambíen está basado en su diseño en Gramáticas Independientes del Contexto y en la teoróa de autómatas y sus expreciones regulares asociadas.

Como ejemplo se muestra en la figura 16, el autómata correspondiente a la clase "number", tomada de: https://www.json.org/json-es.html.

Nótese la similitud con el autómata visto anteriormente para números reales:



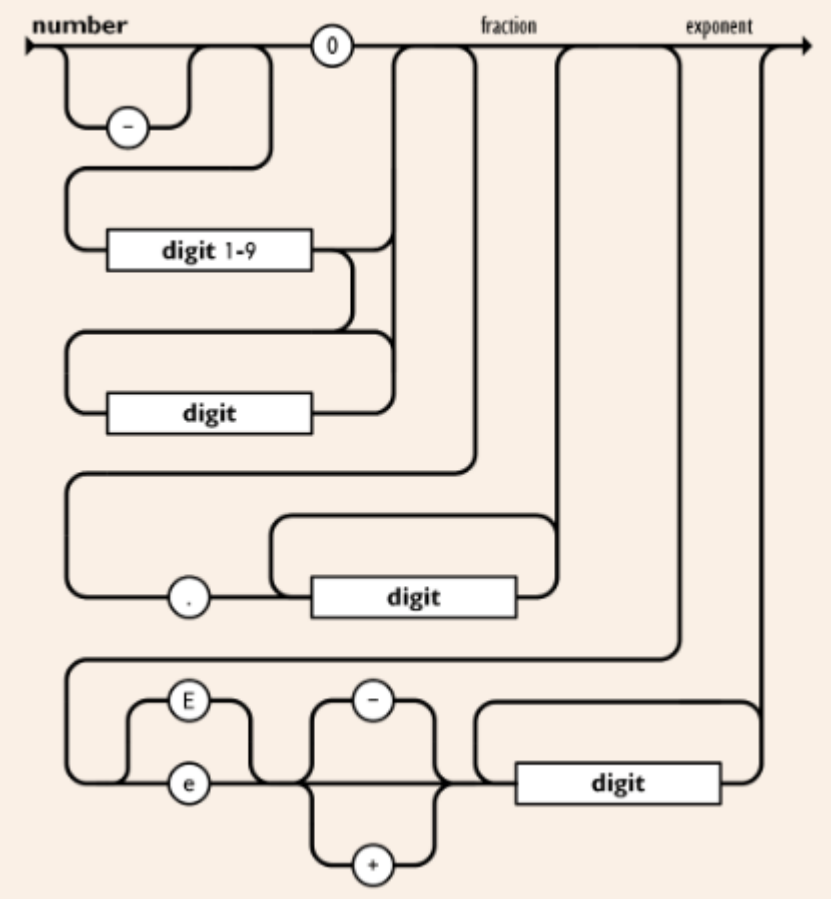


Figura 16: Autómata para la clase "number" en JSON

Bibliografía:

John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, "Teoría de Autómatas, Lenguajes y Computación". *Pearson Addison Wesley.*

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. "Compiladores, Principios, técnicas y herramientas" *Pearson Addison Wesley.*