GRAMÁTICAS

LENGUAJES : Un conjunto de cadenas formadas a partir de un alfabeto dado.

Aspecto Léxico: es el modo en que se forman las palabras o cadenas que pertenecen al lenguaje, para esto se usan patrones (expresiones regulare).

Las palabras o cadenas del lenguaje se clasifican en distintos “componentes léxico”, en ocasiones llamados tokens.

Aspecto Sintáctico: es la forma en que se combinan las palabras para formar oraciones (sentencias) o bloques de un lenguaje de programación.

Así como los “patrones” definen el aspecto léxico (la formación de las palabras).

Los aspectos sintácticos se definen mediante “Gramáticas”

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.

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

- El proceso de construcción del analizador sintáctico **puede revelar ambigüedades sintácticas** y puntos problemáticos.

- La estructura impartida a un lenguaje mediante una gramática 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.

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 (tokens).**

- **2.** Un conjunto finito de **variables**, denominado también en ocasiones **símbolos no terminales** o categorías sintácticas. 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**. 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 →. (se suele leer como “deriva en”)

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.

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
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**.

Ejemplo 1: La gramática siguiente 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

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**:

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

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

  Los símbolos de puntuación como paréntesis, coma, etcétera.

  Los dígitos **0**, **1**, …, **9**.

  Las cadenas en negrita como **id** o **if**, cada una de las cuales representa un solo símbolo terminal.

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

  Las primeras letras mayúsculas del alfabeto, como **A**, **B**, **C**, ...

  La letra **S** que, al aparecer es, por lo general, **el símbolo inicial**.

  Los nombres en cursiva y minúsculas, como ***expr*** o ***instr***.

- 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.

- **<u>Ejemplo 2</u>**: Mediante estas convenciones, la gramática del **ejemplo 1** puede rescribirse en forma concisa como:

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

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

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

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.**
La construcción del árbol sintáctico puede hacerse en forma **descendente** o **ascendente**.

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:

$$E \Rightarrow -E$$

El símbolo ⇒ se lee como que E deriva en –E en uno o más pasos

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.

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

Así:

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

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

Si $S \overset{*}{\Rightarrow} \alpha$, en donde **S** es el símbolo inicial de una gramática **G**, decimos que **α** 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 S $\overset{*}{\Rightarrow}$ 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:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$$

Pueden realizarse dos derivaciones para Comprobar que **–(id * id)** es un enunciado de **E**

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id) \quad (1)$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id) \quad (2)$$

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 **α ⇒ β** es un paso en el que se sustituye el no terminal por la izquierda en **α**, escribimos **α** $\underset{lm}{\Rightarrow}$ **β**.
2. **En las derivaciones por la derecha**, siempre se elige el no terminal por la derecha; en este caso escribimos **α** $\underset{rm}{\Rightarrow}$ **β**

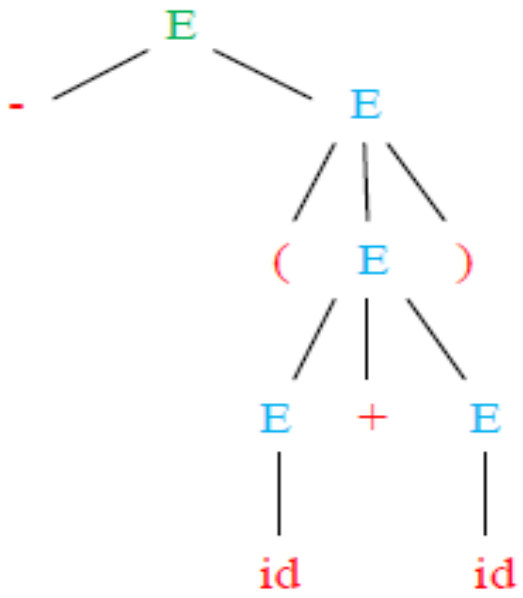
La derivación **E ⇒ −E ⇒ −(E) ⇒ −(E + E) ⇒ −(id + E) ⇒ −(id + id)** es por la **izquierda**

Árboles de análisis sintáctico y derivaciones

- 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**.

Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción.

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



- 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 ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

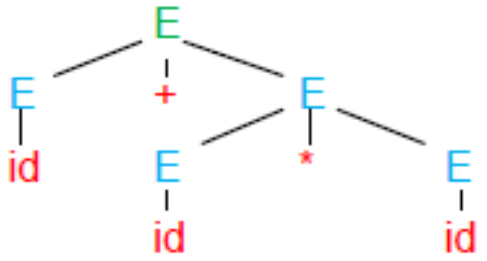
E→ E + E | E * E | (E) | id **(3)**

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

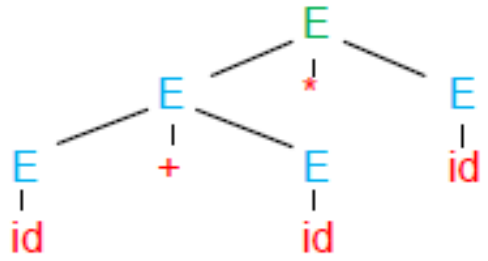
E→ E + E | E * E | (E) | id

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**



(a)



(b)

Factorización por la izquierda

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.

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**.

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 → αA‘
A' → β1 | β2

Recursividad por la izquierda

Una gramática es recursiva por la izquierda si tiene un no terminal **A** tal que haya una derivación **A ⇒ Aα** para cierta cadena **α**.

Los métodos de análisis sintáctico **descendentes no pueden manejar las gramáticas recursivas por la izquierda**, por lo que es necesario eliminarla.

El caso general de recursividad por la izquierda se puede mostrar con producciones del tipo: **A → Aα | β**

La cual puede sustituirse mediante las siguientes producciones no recursivas por la izquierda sin cambiar las cadenas que se derivan de **A**. Esta regla por sí sola basta para muchas gramáticas.

A → βA'
A' → αA' | ε

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

E → T E'
E' → + T E'
T → F T'
T' → * F T'
F → (E) | id

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

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

A → βA'
A' → αA' | ε

El par recursivo por la izquierda de las producciones **E → E + T | T** se sustituye mediante **E → T E'** y **E' → + T E' | ε**.
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).**

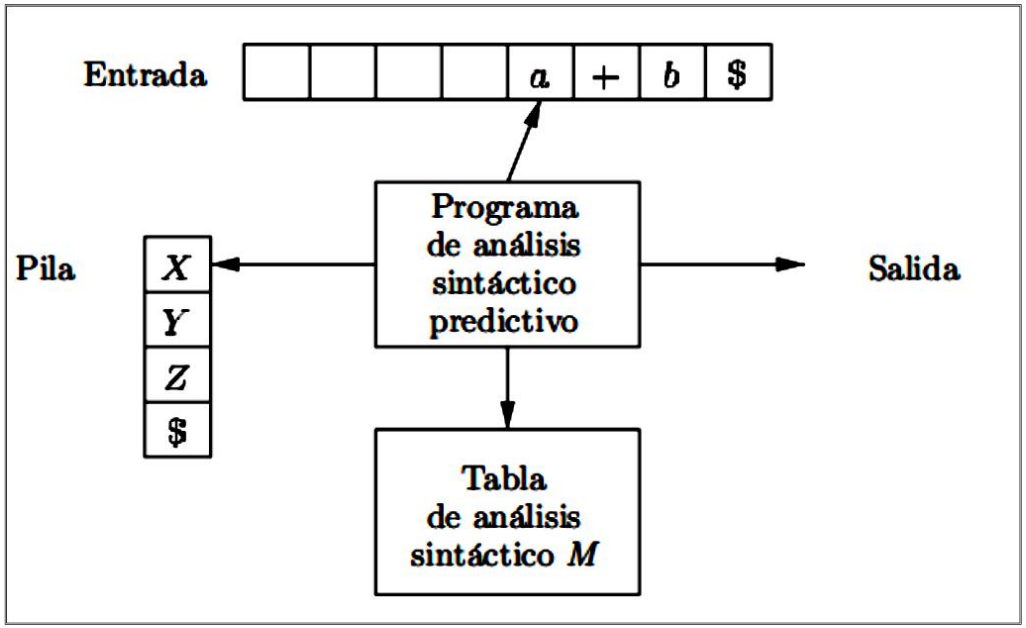
La primera "**L**" en **LL(1)** es para explorar la entrada de izquierda a derecha (por left en inglés), la segunda "**L**" para producir una derivación por la izquierda, y el "**1**" para usar un símbolo de entrada de anticipación en cada paso, para tomar las decisiones de acción del análisis sintáctico.

Las gramáticas **LL(1)** son 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

Se puede construir un analizador sintáctico predictivo no recursivo mediante El mantenimiento de una pila.
El analizador sintáctico es controlado por una tabla, tiene una **entrada**, una **pila** que contiene una secuencia de símbolos gramaticales, una **tabla** de análisis sintáctico, y un **flujo de salida**.



Ejemplo: Para la gramática de expresiones:

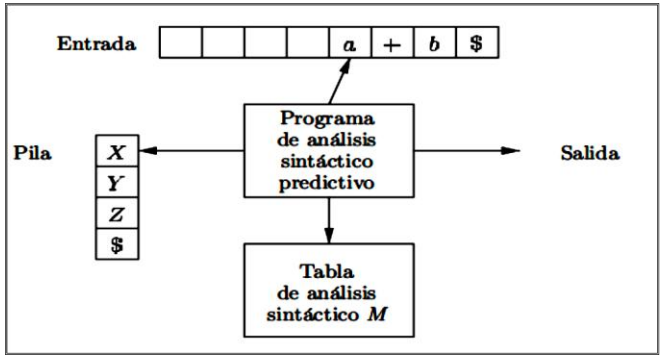
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Se tiene la tabla de análisis sintáctico siguiente, donde 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)		

Ejemplo: **id + id * id**

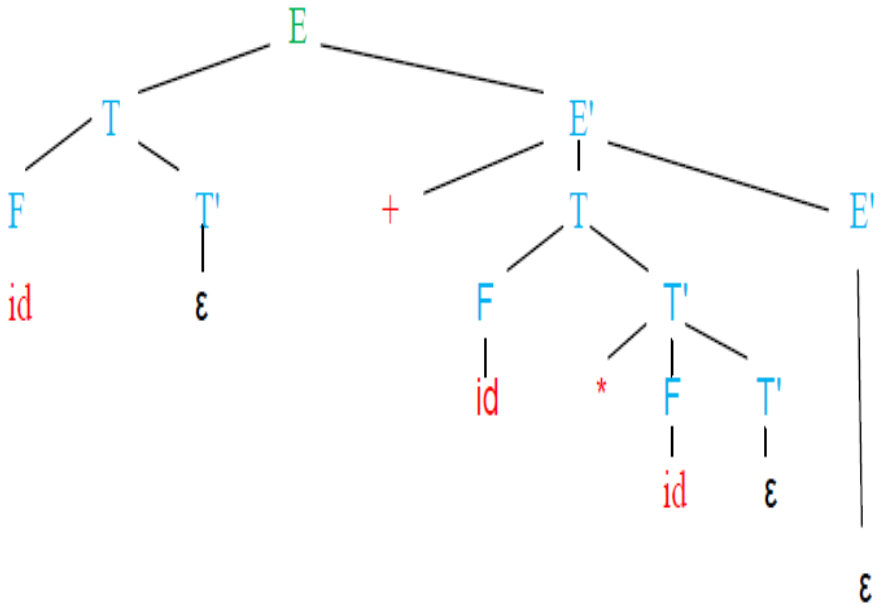
No terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F → (E)		



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

Árbol sintáctico construido por el analizador sintáctico del Ejemplo. Observe que en el árbol sintáctico, las hojas corresponden a la cadena de entrada o bien a la cadena vacía. La raíz del árbol es el símbolo inicial de la gramática.

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



Análisis sintáctico ascendente

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 el de interés práctico

Los analizadores sintácticos **LR son controlados por tablas, en forma muy parecida a los analizadores sintácticos LL** no recursivos.

las **gramáticas LR puedan describir más lenguajes** que las gramáticas **LL**.

- **Temas por ver en apuntes y/o bibliografía**

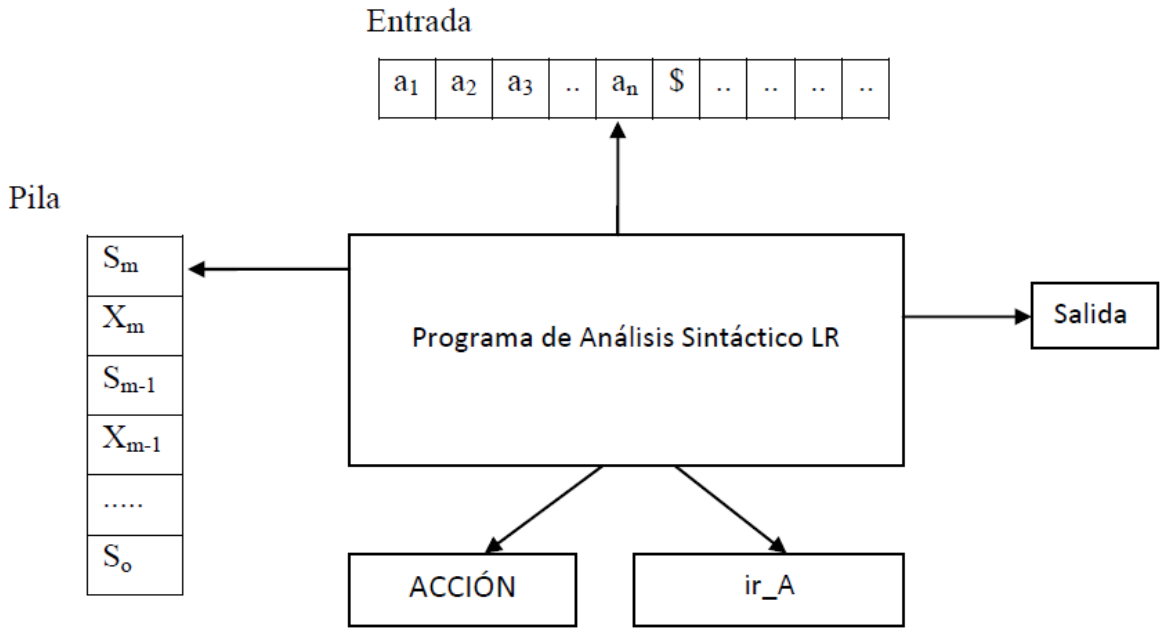
- **Reducciones**
- **Análisis sintáctico por desplazamiento-reducción**
- **Algoritmos de análisis sintáctico LR**

- 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.

Algoritmos de análisis sintáctico LR

En la figura 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.



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

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

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

(1) E → E + T

(2) E → T

(3) T → T * F

(4) T → F

(5) F → (E)

(6) F → id

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 D**5**, 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

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. 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

- **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.*