II

GRAMÁTICAS.

2.1 INTRODUCCIÓN A LAS GRAMÁTICAS *34*

2.2 ESTRUCTURAS DE LAS GRAMÁTICAS *37*

2.3 CLASIFICACIÓN DE LAS GRAMÁTICAS *41*

2.4 REPRESENTACIÓN DE GRAMÁTICAS *44*

2.5 EJERCICIOS PROPUESTOS *84*

Palabras clave : gramáticas, clasificación de gramáticas. Gramáticas de contexto libre.Lenguajes y autómatas. Derivaciones, lenguaje generado.

2.1 INTRODUCCIÓN A LAS GRAMÁTICAS.

En la sección 1.3 se mostró como las primeras fases de análisis en un proceso de compilación interactúan para efectuar ciertas tareas sobre el programa fuente. El analizador léxico que tiene de entrada al programa fuente, identifica a los tokens y los envía al analizador sintáctico, fig 2.1.

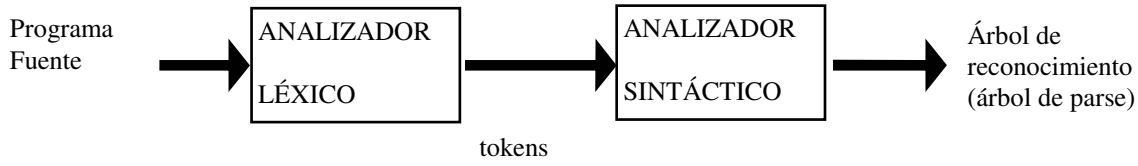
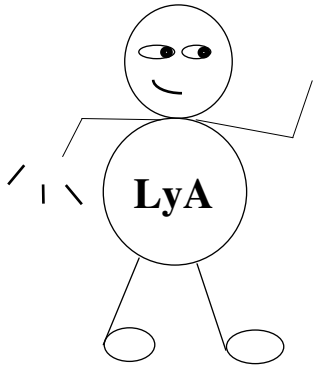


Fig 2.1 Análisis léxico y análisis sintáctico de un programa.

¿ Cuántos tokens envía el analizador léxico al analizador sintáctico ?; los que requiera el propio analizador sintáctico para el reconocimiento de la sintáxis de una instrucción. La tarea fundamental del análisis sintáctico es recibir los tokens que juntos (concatenados) constituyen en escencia, a una instrucción. De acuerdo a ciertas reglas de sintáxis para cada instrucción , decide si éstas -las instrucciones-, están bien construidas; es decir, respetan y cumplen dichas reglas de sintáxis.
Las *gramáticas* consisten de un conjunto de reglas, que nos permiten especificar formalmente la sintáxis de las instrucciones de un lenguaje de programación.



Expresiones regulares = especificación de tokens.

Gramáticas = especificación de sintáxis de instrucciones.

Los programas analizadores sintácticos que se basan en gramáticas para reconocer las instrucciones residentes en el programa fuente se denominan *Parser's (reconocedores).* Las dos clases de parser más comunes son :

- *Parser Descendente (Top Down)*
- *Parser Ascendente (Bottom Up)*

INSTRUCCIONES	
Especificación	***Reconocimiento***
Gramáticas	1. Parser Descendente (Top Down) 2. Parser Ascendente (Bottom Up)

Fig. 2.2 Especificación y reconocimiento de instrucciones.

Veamos con más detalle el fin de una gramática. Hemos dicho que una gramática se utiliza para especificar de manera formal, la sintáxis de una instrucción (o de varias) . El uso de una gramática le permite a un reconocedor (Parser), saber si una instrucción está bien construida. Si no está adecuadamente construida, el reconocedor lo hará saber mediante el envío de un mensaje "error de sintáxis" .
Supongamos la instrucción *scanf* que en lenguaje ***C*** es usada para leer datos desde el teclado. En la figura 2.3 se muestran algunas posibles formas en las que puede ser encontrada en un programa fuente, la instrucción *scanf.*

```
1)      scanf ("%d",&iNum);
2)      scanf ("%f",pfNum);
3)      scanf ("%s %d",sNom,&iTen);
4)      scanf ("%c",&asCar[i]);
5)      scanf ("%d %f %d",piEnt1,pfReal,piEnt2);
            . . . . .
```

Fig 2.3 Algunas instancias de la instrucción *scanf.*

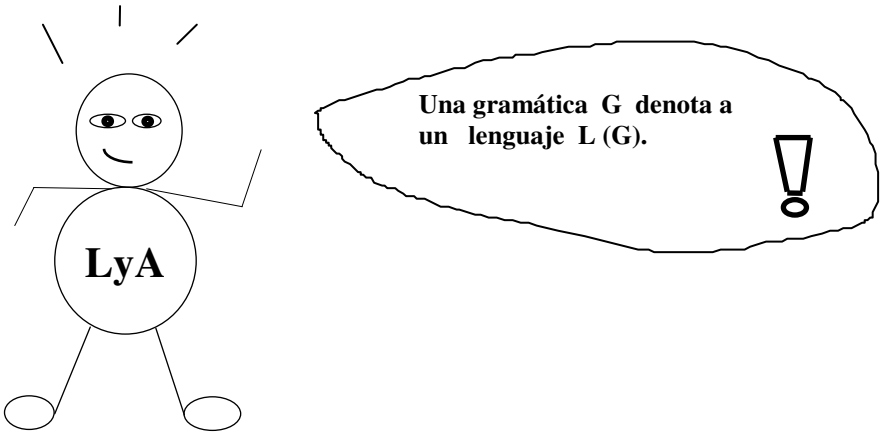
Podemos llegar a ciertas conclusiones hechándole un vistazo a la fig. 2.3. Entre las cuestiones que mas nos interesan, están :

1. Las **n** instancias son un conjunto de tokens concatenados. La instancia 1 tiene 8 tokens debidamente concatenados :

scanf	Palabra reservada
(	Separador
”%d”	CteLit
,	Separador
&	Operador de Dirección
iNum	Identificador
)	Separador
;	Terminador de Instrucción

2. Las **n** instancias de la instrucción constituyen un lenguaje, donde las cadenas que lo constituyen, precisamente, son *tokens concatenados.*

3. Si el analizador sintáctico (Parser descendente o ascendente) se basa en gramáticas para efectuar el reconocimiento de una instrucción bien construida, entonces la gramática debe especificar al lenguaje compuesto por todas las cadenas que son una instancia para dicha instrucción.



2.2 ESTRUCTURA DE LAS GRAMÁTICAS.

Una gramática consiste de 4 componentes, y es denotada por $G = (V_T, V_N, S, \Phi)$ donde :

V_T Es el conjunto de *símbolos terminales* a partir de los cuales las cadenas son formadas. Si la gramática es usada para denotar un lenguaje de programación , entonces los tokens son los símbolos terminales.

V_N Es el conjunto de *símbolos no terminales* , también llamados *variables sintácticas*. Las variables sintácticas denotan un conjunto de cadenas. Los símbolos no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática.

S Es un símbolo no terminal de la gramática que es distinguido como *símbolo de inicio*. El conjunto de cadenas que este símbolo de inicio denota, es el lenguaje definido por la gramática G.

Φ Es el conjunto de *producciones o reglas gramaticales*. Las producciones de una gramática especifican la manera en que los tokens (terminales) y las variables sintácticas (no terminales) pueden ser combinados para formar cadenas.

Ejemplo 2.1 *Dada la siguiente gramática :*

R ⟶ *read*
R ⟶ *readln*
R ⟶ *read (P)*
R ⟶ *readln (P)*
P ⟶ *P, id*
P ⟶ *id*

Encuentra sus componentes V_T, V_N, S y Φ.

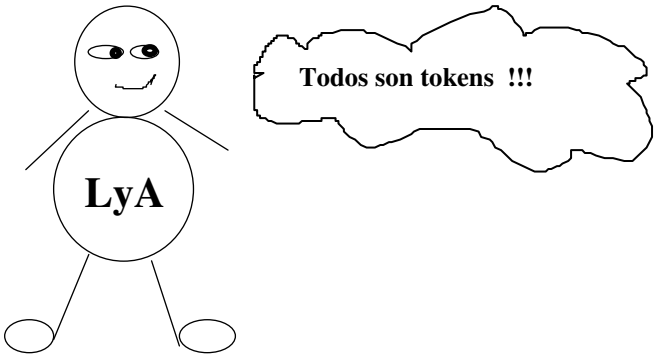
Usemos la siguiente convención de notación :

- Las letras mayúsculas servirán para denotar variables sintácticas (Símbolos no terminales).
- El lado izquierdo de la primera producción es el símbolo de inicio.
- Si $A \longrightarrow \alpha_1$, $A \longrightarrow \alpha_2$, . . . $A \longrightarrow \alpha_K$ son producciones teniendo a la no terminal A en el lado izquierdo, podemos escribir $A \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_K \mid$ donde $\alpha_1, \alpha_2, \ldots, \alpha_K$ se les llama las alternativas para A.

Encontremos el conjunto de símbolos terminales V_T.

V_T = { read, readln, (,) , , , id }

ya que *read* y *readln* son palabras reservadas, *(,)* y la coma son separadores, e *id* es un identificador.



V_N = { R, P } únicamente dos símbolos no terminales.

El símbolo de inicio es *R*, aplicando la 2da. convención de notación, y el conjunto Φ de producciones es condensado, aplicando la 3a. convención :

Φ = { R ⟶ read | readln | read (P) | readln (P)
P ⟶ P, id | id }

La gramática es :

G = ({ read, readln, (,) , , , id }, {R,P},R,Φ)

Ejemplo 2.2 *Supongamos la gramática que especifica a la instrucción de asignación en Pascal, con expresiones aritméticas +, -, * y / únicamente.*

1) $A \longrightarrow id := E;$
2) $E \longrightarrow E + T$
3) $E \longrightarrow E - T$
4) $E \longrightarrow T$
5) $T \longrightarrow T * F$
6) $T \longrightarrow T / F$
7) $T \longrightarrow F$
8) $F \longrightarrow id$

9) $F \rightarrow num$
10) $F \rightarrow (E)$
11) $F \rightarrow - E$

La gramática $G = (V_T, V_N, S, \Phi)$ tiene los siguientes componentes :

V_T = { id, :=, ; , +, - , * , / , num, (,) }
V_N = { A,E,T,F }
S = A

Usando la convención de notación (3), la gramática puede representarse como :

A → id := E;
E → E + T | E - T | T
T → T * F | T / F | F
F → id | num | (E) | -E

Ejemplo 2.3 *Veamos una gramática que nos genera el lenguaje de todos los números enteros pares :*

N → *KP*
N → *L*
K → *KD*
K → *Z*
L → *2*
L → *4*
L → *6*
L → *8*
P → *0*
P → *2*
P → *4*
P → *6*
P → *8*
Z → *1*
Z → *2*
...
...
Z → *9*
D → *0*

$D \rightarrow 1$
$D \rightarrow 2$
...
...
$D \rightarrow 9$

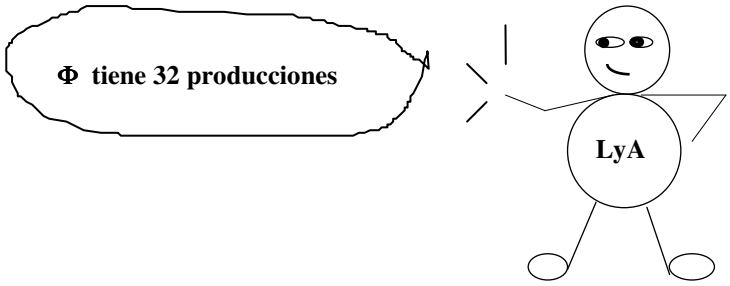
$V_T = \{ 0, 1, 2..., 9 \}$ // *Los dígitos del 0-9.*

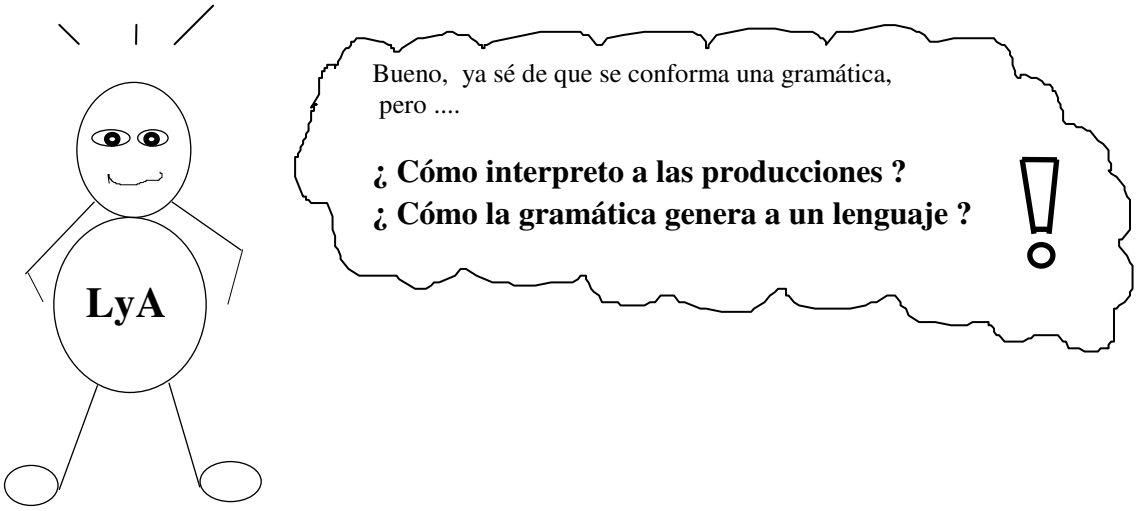
$V_N = \{ N, K, L, Z, D, P \}$

$S = N$

La gramática con la agrupación de las alternativas para cada variable sintáctica es :

$N \rightarrow KP \mid L$
$K \rightarrow KD \mid Z$
$P \rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8$
$L \rightarrow 2 \mid 4 \mid 6 \mid 8$
$D \rightarrow 0 \mid 1 \mid ... \mid 9$
$Z \rightarrow 1 \mid 2 \mid ... \mid 9$





Sección 2.4 resuelve tus preguntas ! !

2.3 CLASIFICACIÓN DE LAS GRAMÁTICAS.

Chomsky clasificó las gramáticas en 4 clases, imponiendo un conjunto de restricciones sobre las producciones. La clasificación es la siguiente :

- *Gramáticas no restringidas.*
- *Gramáticas sensibles al contexto.*
- *Gramáticas de contexto libre.*
- *Gramáticas regulares.*

Gramáticas no restringidas.- Son aquellas en las cuales las producciones no están sujetas a ninguna clase de restricción en su composición.

Gramáticas sensibles al contexto.- Son aquellas que contienen únicamente producciones de la forma :

$\alpha \longrightarrow \beta$, donde $|\alpha| \leq |\beta|$, y $|\alpha|$ denota la longitud de la cadena α.

Ejemplo : *Sea la gramática con producciones Φ :*

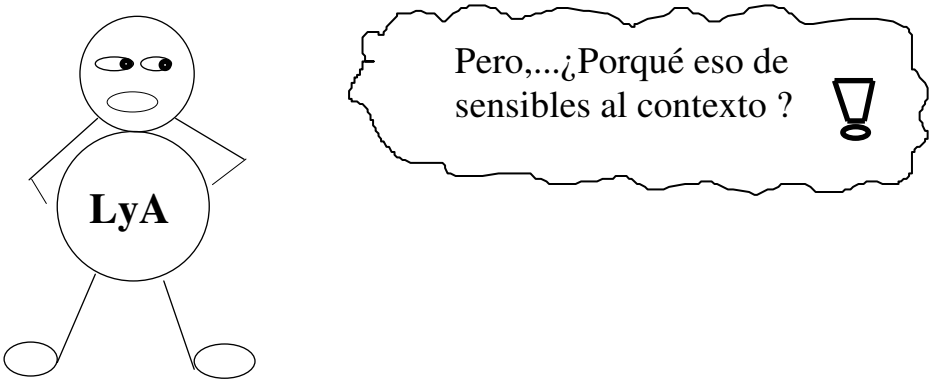
(1) S ⟶ aSBC
(2) S ⟶ abC
(3) bB ⟶ bb
(4) bC ⟶ bc
(5) CB ⟶ BC
(6) cC ⟶ cc

En la figura 2.4 mostramos la producción y los diferentes parámetros α,β, $|\alpha|$, $|\beta|$.

No de Producción	α	β	$\lvert\alpha\rvert$	$\lvert\beta\rvert$
1	S	aSBC	1	4
2	S	abC	1	3
3	bB	bb	2	2
4	bC	bc	2	2
5	CB	BC	2	2
6	cC	cc	2	2

Fig. 2.4 Longitudes de α y β.

Para la gramática G = ({a,b,c}, {S,B,C}, S, Φ), siempre se cumple $|\alpha| \leq |\beta|$.



Una producción puede leerse de la siguiente forma :

$A \longrightarrow \alpha$ *" A produce a alfa" ó bien, " A es definido por alfa".*

Si analizamos más detenidamente la gramática del ejemplo, encontramos producciones como las no. 3, 4, 5 y 6 que tienen la siguiente interpretación :

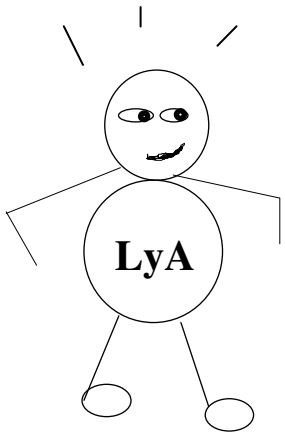
bB ⟶ bb *"B produce ó es definido por bb sólo cuando a su izquierda tiene el token b"*

bC ⟶ bc *"C produce ó es definido por bc sólo cuando a su izquierda tiene el token b"*

CB ⟶ BC *"C produce ó es definido por BC sólo cuando a su derecha tiene la no terminal B"*
ó
"B produce ó es definido por BC sólo cuando a su izquierda tiene la no terminal C".

Lo mismo sucede para la producción 6. La interpretación se deja de ejercicio al alumno.

Los ejemplos anteriores se refieren a que una no terminal genera o produce una cadena, sólo cuando su contexto (a su izquierda ó a su derecha) está restringido a la aparición de cierto token o de un no terminal. Este contexto está contenido en el lado izquierdo de la propia producción.



Entonces : la primera producción
S → aSBC
Nada a la izquierda, nada a la derecha

¿ CONTEXTO LIBRE ?

Gramática de contexto libre.- Son aquellas que contienen solamente producciones de la forma :

$\alpha \longrightarrow \beta$, donde $|\alpha| \leq |\beta|$ y $\alpha \in V_N$.

Ejemplo : *Sea la gramática con producciones Φ:*

$S \longrightarrow aCa$
$C \longrightarrow aCa$
$C \longrightarrow b$

Sus componentes son : $V_T = \{ a,b \}$, $V_N = \{ S,C \}$, $S = S$.

Analizando el lado izquierdo de cada producción, se encuentra un símbolo no terminal, *libre en su contexto*, es decir, a su izquierda y derecha aparece la cadena $\in$.

Gramáticas de contexto libre : su característica principal, el lado izquierdo de las producciones sólo un símbolo no terminal :

NO MÁS

Gramáticas Regulares.- Son aquellas que contienen sólo producciones de la forma $\alpha \longrightarrow \beta$, donde $|\alpha| \leq |\beta|$ y $\alpha \in V_N$ y β tiene la forma aB ó a, donde $a \in V_T$ y $\beta \in V_N$.

Ejemplo : Sea la gramática con producciones Φ :

$S \rightarrow aS$
$S \rightarrow aB$
$B \rightarrow bC$
$C \rightarrow aC$
$C \rightarrow a$

Sus componentes son : $V_T = \{ a, b \}$, $V_N = \{ S,B,C \}$, $S = S$.
Observando cada una de la reglas o producciones, todas tienen en su lado izquierdo sólo una no terminal. Las gramáticas regulares son también gramáticas de contexto libre, o sea :

gramáticas regulares $\subset$ ***gramáticas de contexto libre.***

Además, el lado izquierdo de cada producción cumple con la forma aB ó a, por ejemplo :

S	→ aS	a = a	,	B = S
S	→ aB	a = a	,	B = B
B	→ bC	a = b	,	B = C
C	→ aC	a = a	,	B = C
C	→ a	a = a		

Los lenguajes de programación tienen la característica agradable que consiste en que la mayoría de sus instrucciones pueden ser especificadas, por *gramáticas de contexto libre.*

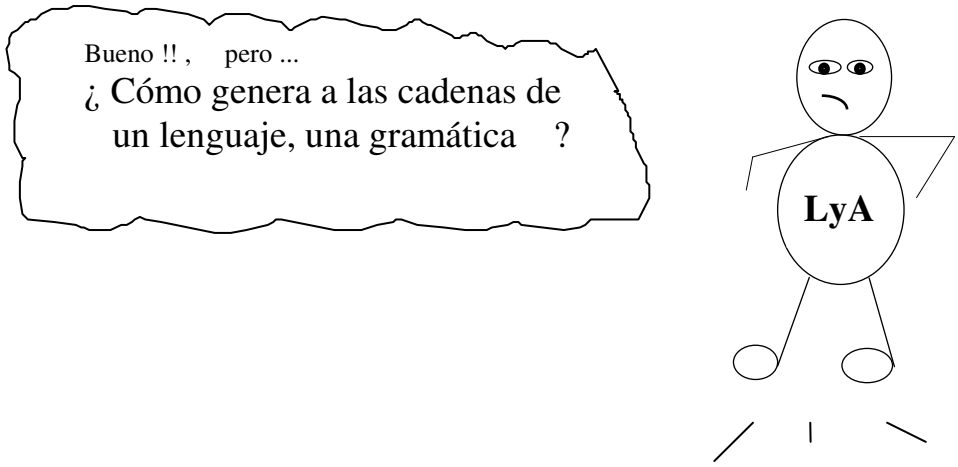
2.4 REPRESENTACIÓN DE GRAMÁTICAS.

Dedicaremos esta sección al estudio de varios temas que se relacionan con una gramática y su representación. A continuación listamos los temas.

- *Lenguaje generado por una gramática.*
- *Derivaciones*
- *Arboles de reconocimiento (Arboles de Parse)*
- *Ambigüedad*
- *Escritura de gramática*
- Precedencia y Asociatividad de operadores.
- Recursividad a la izquierda. Recursividad a la derecha.
- *Notación de Backus Naur (BNF).*

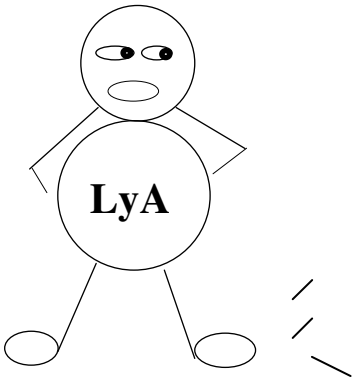
Lenguaje generado por una gramática.

En la sección 2.1 concluimos que una gramática genera un lenguaje L(G) , donde G es la gramática. En lenguajes de programación cada cadena del lenguaje, es un conjunto de tokens concatenados. *Las gramáticas también pueden usarse para representar tokens*, pero esta tarea es mejor delegarla a las expresiones regulares. En algunos ejemplos utilizaremos una gramática para representar a un token, como en el ejemplo 2.3. Utilizaremos a las gramáticas para especificar las reglas de sintáxis de instrucciones o sentencias, pertenecientes a un lenguaje determinado.



Existen varias formas de visualizar el proceso mediante el cual una gramática genera o define a un lenguaje. A continuación veremos dos de ellas :

- *Derivaciones*
- *Arboles de Reconocimiento (Arboles de Parse).*



Herramientas para generar un lenguaje utilizando una gramática :

- **Derivaciones**
- **Arboles de parse**

Derivaciones.

Supongamos la gramática del ejemplo 2.1, que define al lenguaje de la instrucción de lectura en Pascal.

R ⟶ *read* | *readln* | *read(P)* | *readln(P)*
P ⟶ *P,id* | *id*

A menudo, nos hacemos la siguiente pregunta :

R *⇒ readln (iNum) *" ¿ El símbolo de inicio R deriva en cero ó más etapas a la sentencia **readln (iNum)** ? "*

El símbolo ⇒ es el utilizado para denotar una derivación , es decir, la generación de una cadena (sentencia) del lenguaje. El símbolo ⇒ se puede combinar con otro para indicar :

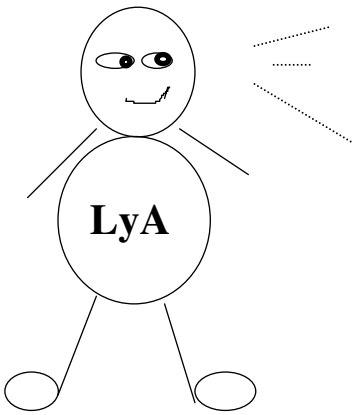
*⇒ *" deriva en 0 o más etapas a ... "*
+⇒ *" deriva en una o más etapas a ... "*
⇒ *" deriva en una etapa a ... "*

La *derivación* consiste en utilizar una producción, con el fin de ir efectuando sustituciones; una sustitución indica que la no terminal -variable sintáctica- situada al lado izquierdo de una producción, es reemplazada por la cadena que constituye la parte derecha de dicha producción. Así, para la anterior cuestión :

R *⇒ *readln (iNum)*

el proceso de derivación es :
R ⇒ readln (P) ⇒ readln (id)

(1) (2)
La generación es exitosa, ya que el analizador léxico no envía el lexema *iNum*, sino el token *id* (identificador). Asimismo, la derivación (1) utiliza la producción R→ readln(P) y la derivación (2), utiliza la producción P⟶ id .



¿ Porqué iniciamos la derivación con R y no con P ? .

¿ Es lo mismo ?

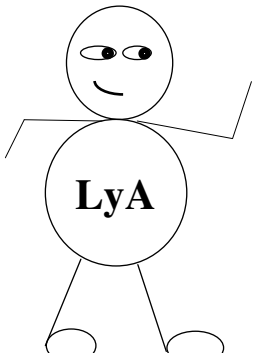
La generación (derivación) de una sentencia o cadena de un lenguaje, sólo se puede lograr si dicha derivación comienza, *a partir del símbolo de inicio o distinguido* de la gramática.
Analicemos ahora, las producciones de la gramática de este ejemplo. El símbolo⟶ es interpretado como *" Es definido por "*. En las producciones :

R ⟶ read | readln | read(P) | readln(P)

decimos que *R* es definido por las cadenas *read* o *readln* o *read(P)* o bien *readln(P).* La definición del símbolo no terminal *R* está en función de los tokens *read*, *readln*, *(,)* y la variable sintáctica -símbolo no terminal- *P*. Para la definición de *P*, se utilizan dos producciones :

P ⟶ P,id | id

O sea , *P* es definido por *P, id* o bien por el token *id* solamente. Aquí *P* el símbolo no terminal, está definido en función de los tokens *id, la coma y por el mismo !!*. Esta regla es claramente **recursiva.** El uso de la *recursividad* al escribir reglas o producciones, es la característica principal que distingue a una gramática con respecto a una expresión regular. Una expresión regular no tiene definiciones recursivas.



Escritura de :

Expresiones regulares ... definiciones no recursivas.

Gramáticas ... definiciones recursivas.

La recursividad en una regla es para una gramática, lo que la operación de cerradura es para una expresión regular (r* ó r^+). Para mostrar lo anterior, obtengamos la derivación para la cadena *readln (x,y,z).*

R $*\Rightarrow$ readln (x,y,z)

R $\Rightarrow$ readln (P) // R $\longrightarrow$ readl (P)
R $\Rightarrow$ readln (P,id) // P $\longrightarrow$ P,id
R $\Rightarrow$ readln (P,id,id) // P $\longrightarrow$ P,id
R $\Rightarrow$ readln (id,id,id) // P $\longrightarrow$ id

La aplicación de la regla recursiva P $\longrightarrow$ P,id nos permite obtener la cadena *,id* cuantas veces se encuentre en la sentencia un *id*. Obviamente el símbolo no terminal *P*, genera el lenguaje :

L(P) = { *id* $(,id)^{\mathbf{n}}$ | n ≥ 0 } , que es una concatenación de *id* con las cadenas $(,id)^n$ fig. 2.5.

n	**Cadena de L(P)**
0	id (,id) ^{0} = id ∈ = id
1	id (,id) ^{1} = id,id
2	id (,id) ^{2} = id(,id,id) = id,id,id
...	...

Fig 2.5 Lenguaje generado por la variable sintáctica P.

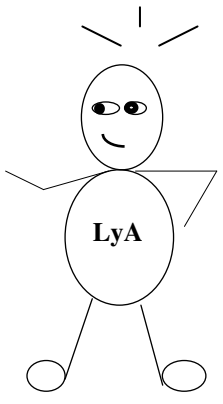
Podemos también concluir con este ejemplo, que *P* nunca podrá derivar por si misma, una sentencia del lenguaje especificado por la gramática :

R $\longrightarrow$ read | readln | read(P) | readln(P)
P $\longrightarrow$ P,id | id

cuyo símbolo de inicio es *R* .

¿ P $*\Rightarrow$ readln (x) ?

NUNCA !



Ejemplo 2.4 *Suponga la gramática :*

$A \longrightarrow id = E$
$E \longrightarrow E+T \mid E-T \mid T$
$T \longrightarrow T*F \mid T/F \mid F$
$F \longrightarrow id \mid num \mid -E \mid (E)$

con elementos : $V_T = \{ id, =, +, -, *, /, num, (,) \}$, $V_N = \{ A, E, T, F \}$, $S = A$.

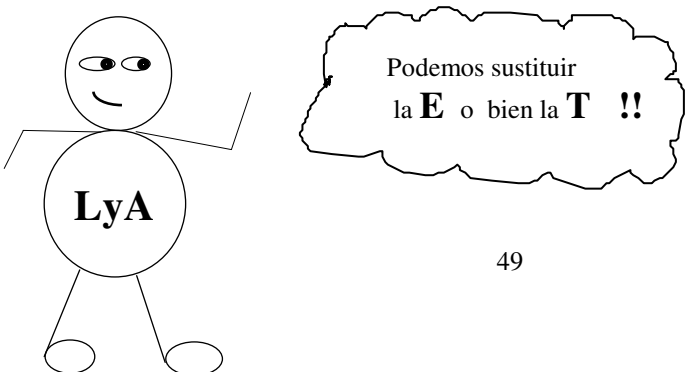
Obtengamos la derivación de x = y + 5 * z, es decir, $A \overset{*}{\Rightarrow} x = y + 5 * z$.

Derivación	Producción Aplicada
(1) $A \Rightarrow id = E$	$A \longrightarrow id = E$
(2) $\Rightarrow id = E+T$	$E \longrightarrow E+T$
(3) $\Rightarrow id = T+T$	$E \longrightarrow T$
(4) $\Rightarrow id = F+T$	$T \longrightarrow F$
(5) $\Rightarrow id = id+T$	$F \longrightarrow id$
(6) $\Rightarrow id = id+T*F$	$T \longrightarrow T*F$
(7) $\Rightarrow id = id+F*F$	$T \longrightarrow F$
(8) $\Rightarrow id = id+num*F$	$F \longrightarrow num$
(9) $\Rightarrow id = id+num*id$	$F \longrightarrow id$

Derivación con éxito.

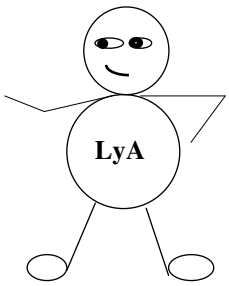
En la anterior derivación de 9 etapas, observamos que siempre sustituímos el no terminal situado *"más a la izquierda"* de la forma sentencial. Una *forma sentencial* es una cadena que contiene al menos un símbolo no terminal -variable sintáctica-. Una *sentencia* es una cadena que contiene solamente símbolos terminales -tokens-. En la anterior tabla las derivaciones numeradas del (1) al (8) derivan formas sentenciales. La derivación (9) produce una sentencia. Veamos la derivación (1) : $A \Rightarrow id = E$. La única no terminal que podemos sustituir es la *E,* empleando la alternativa $E \longrightarrow E+T$, obtenemos la derivación (2) :

$A \Rightarrow id = E \Rightarrow id = E+T$



A ⇒ id = E ⇒ id = E+T ⇒ id = T+T

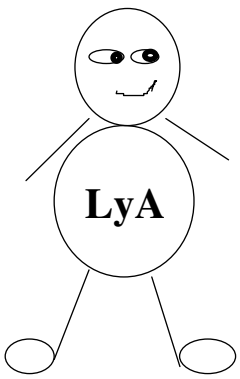
E ⟶ T
En una derivación "más a la izquierda" siempre sustituímos a la no terminal que se encuentra más a la izquierda.



Llamemos a esta manera de derivar una sentencia como *Derivación a la Izquierda*. Un *reconocedor descendente (Parser Top Down)* es un programa que reconoce instrucciones utilizando una gramática y derivaciones a la izquierda.

A continuación mostraremos el proceso de derivación a la izquierda de la cadena :
x = - y * z + 1

Derivación a la izquierda	Producción Aplicada
A ⇒ id = E	A ⟶ id = E
⇒ id = E+T	E ⟶ E+T
⇒ id = T+T	E ⟶ T
⇒ id = T*F+T	T ⟶ T*F
⇒ id = F*F+T	T ⟶ F
⇒ id = -E*F+T	F ⟶ -E
⇒ id = -T*F+T	E ⟶ T
⇒ id = -F*F+T	T ⟶ F
⇒ id = -id*F+T	F ⟶ id
⇒ id = -id*id+T	F ⟶ id
⇒ id = -id*id+F	T ⟶ F
⇒ id = -id*id+num	F ⟶ num



Caray !! El proceso de derivación de una sentencia o instrucción, requiere de una significativa cantidad de etapas, aún para una cadena tan simple como :
*x = - y * z + 1*

Pobre Parser !! ....

¿ Será que todo lo anterior lo hace a mano ?!!

Afortunadamente, existen técnicas que permiten que el algoritmo utilizado por un *reconocedor* (*Parser*) sea eficiente.

Ejemplo 2.5 *Tomemos la misma gramática y también la misma sentencia a reconocer. Obtengamos la derivación :*

$$A \ *\Rightarrow \ x = y + 5 * z$$

	Derivación a la derecha	Producción Aplicada
(1)	$A \Rightarrow$ id = E	A → id = E
(2)	$\Rightarrow$ id = E+T	E → E+T
(3)	$\Rightarrow$ id = E+T*F	T → T*F
(4)	$\Rightarrow$ id = E+T*id	F → id
(5)	$\Rightarrow$ id = E+F*id	T → F
(6)	$\Rightarrow$ id = E+num*id	F → num
(7)	$\Rightarrow$ id = T+num*id	E → T
(8)	$\Rightarrow$ id = F+num*id	T → F
(9)	$\Rightarrow$ id = id+num*id	F → id

Comparando con la derivación en el ejemplo 2.4, observamos que tenemos el mismo número de etapas de derivación, pero siempre se ha sustituído el no terminal situado *"más a la derecha"*. La derivación (2) tiene dos no terminales que pueden sustituirse:

$$A \Rightarrow \ id = E \quad \Rightarrow \ id = E + T$$

En una *derivación a la derecha*, la no terminal seleccionada debe ser la *T*. Esto sucede, precisamente al obtener la 3a. derivación, en que *T* es sustituída por la producción o alternativa $T \longrightarrow T*F$.

$$A \Rightarrow \ id = E \quad \Rightarrow \ id = E+T \Rightarrow \ id = E+T*F$$

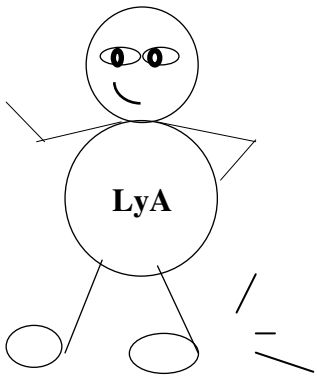
Un *reconocedor ascendente (Parser Bottom Up)* es un programa que reconoce instrucciones utilizando una gramática y derivaciones a la derecha. Al proceso de derivación a la derecha, también se le conoce como *Derivación Canónica*. La siguiente tabla nos muestra la derivación a la derecha de la cadena : $x = -y * z + 1$.

Derivación a la derecha	**Producción Aplicada**
A ⇒ id = E	A → id = E
⇒ id = E+T	E → E+T
⇒ id = E+F	T → F
⇒ id = E+num	F → num
⇒ id = T+num	E → T
⇒ id = T*F+num	T → T*F
⇒ id = T*id+num	F → id
⇒ id = F*id+num	T → F
⇒ id = -E*id+num	F → -E
⇒ id = -T*id+num	E → T
⇒ id = -F*id+num	T → F
⇒ id = -id*id+num	F → id

- Las derivaciones nos permiten visualizar el proceso de generación de cadenas, al usar una gramática.
- *Derivaciones a la izquierda* ... utilizadas por :

  Reconocedores descendentes.
- *Derivaciones a la derecha* ... utilizadas por :

  Reconocedores ascendentes.



A continuación se muestran algunos ejemplos típicos, donde se obtiene el lenguaje generado por una gramática dada.

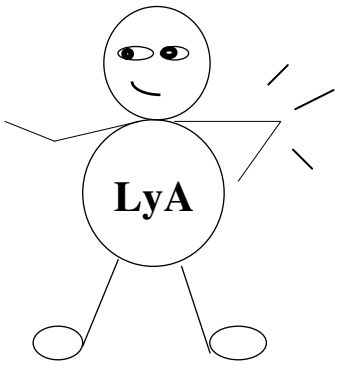
Ejemplo 2.6 *Encontrar el lenguaje generado L(G) por la gramática cuyo conjunto de producciones Φ es :*

$S \longrightarrow aCa$
$C \longrightarrow aCa$
$C \longrightarrow b$

Los componentes de la gramática son : $V_T = \{ a,b \}$, $V_N = \{ S,C \}$, $S = S$.
Agrupemos en alternativas, las producciones cuyo lado izquierdo sea la misma no terminal :

$S \longrightarrow aCa$
$C \longrightarrow aCa \mid b$

Paso 1.
Agrupar las producciones.

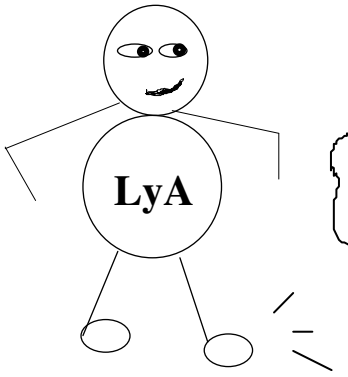


Luego, apliquemos una derivación utilizando sólo reglas no recursivas :

$S \underset{S \rightarrow aCa}{\Rightarrow} aCa \underset{C \rightarrow b}{\Rightarrow} aba$

Paso 2.
Obtener las sentencias simples, empleando sólo combinaciones de producciones no recursivas.

Derivación	**Producción utilizada**
$S \Rightarrow aCa$	$S \longrightarrow aCa$
$\Rightarrow aba$	$C \longrightarrow b$



Ohh !! ...
Existe un símbolo **b** y en su contexto, observo que a la izquierda y a la derecha se tiene un símbolo **a .**

Sentencia derivada *aba*

Ahora, usemos la regla recursiva C⟶aCa para observar qué cadenas produce el uso de esta regla recursiva.

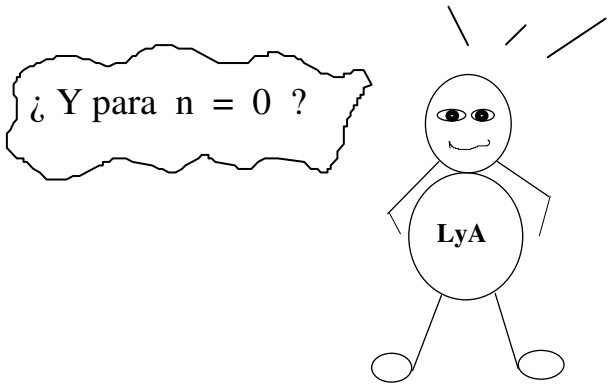
$$S \underset{S \to aCa}{\Rightarrow} aCa \underset{C \to aCa}{\Rightarrow} aaCaa \underset{C \to aCa}{\Rightarrow} aaaCaaa \underset{C \to aCa}{\Rightarrow} aaaaCaaaa$$

$$\Rightarrow aabaa \quad (C \to b) \qquad \Rightarrow aaabaaa \quad (C \to b) \qquad \Rightarrow aaaabaaaa \quad (C \to b)$$

La regla C ⟶ aCa produce el mismo número de *a's* a la izquierda y a la derecha de la *b*.

Podemos concluir, que el lenguaje generado por G= ({a,b}, {S,C}, S, Φ) es :

L (G) = { $a^n b a^n$ | n > 0 }



Ejemplo 2.7 *Supongamos la gramática del read y readln en Pascal.*

R ⟶ read | readln | read(P) | readln(P)
P ⟶ P,id | id

Obtener el lenguaje generado por la gramática.

La gramática tiene las alternativas agrupadas. Únicamente existe una producción recursiva :

P ⟶ P,id

Obtengamos las cadenas más simples que genera el símbolo de inicio *R* . Ésto se logra, derivando sólo con reglas no recursivas :

$R \underset{R \to read}{\Rightarrow}$ read // 1a. cadena

$R \Rightarrow$ readln // 2a. cadena

$R \rightarrow readln$

$R \Rightarrow_{R \rightarrow read(P)} read(P) \Rightarrow_{P \rightarrow id} read(id)$ // 3a. cadena

$R \Rightarrow_{R \rightarrow readln(P)} readln(P) \Rightarrow_{P \rightarrow id} readln(id)$ // 4a. cadena

Ahora, apliquemos la regla recursiva para observar el conjunto de cadenas que produce :

$R \Rightarrow_{R \rightarrow read(P)} read(P) \Rightarrow_{P \rightarrow P,id} read(P,id) \Rightarrow_{P \rightarrow P,id} read(P,id,id) \Rightarrow_{P \rightarrow P,id} read(P,id,id,id)$

$\Downarrow$ $\Downarrow$ $\Downarrow$

$\Rightarrow_{P \rightarrow id} read(id,id)$ $\Rightarrow_{P \rightarrow P,id} read(id,id,id)$ $\Rightarrow_{P \rightarrow P,id} read(id,id,id,id)$

Juntemos las sentencias anteriormente derivadas en una tabla, y observemos las partes que las componen :

Sentencia	Parte 1	Parte 2
read	read	$\in$
readln	readln	$\in$
read(id)	read	(id)
readln(id)	readln	(id)
read(id,id)	read	(id,id)
readln(id,id)	readln	(id,id)
read(id,id,id)	read	(id,id,id)
readln(id,id,id)	readln	(id,id,id)

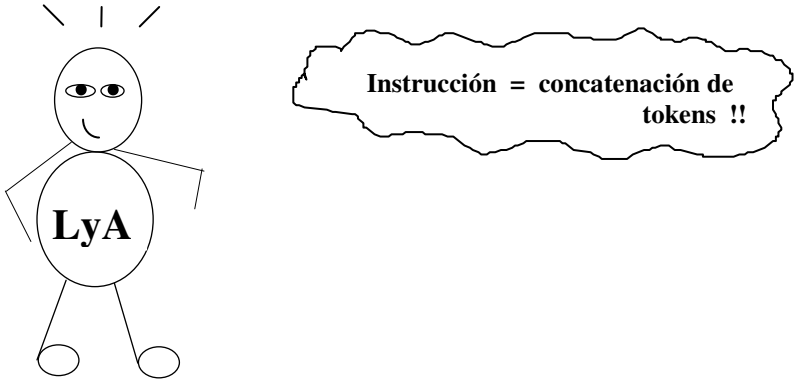
El lenguaje generado es la concatenación de parte1 con parte 2 :

$\mathbf{L(G)} = \{ (\mathbf{read}^{\mathbf{r}}\ \mathbf{readln}^{\mathbf{s}})\ (\ (\ \mathbf{id}\ (\ \mathbf{,id}\ )^{\mathbf{n}})\ )^{\mathbf{m}} \mid n >= 0,\ 0 \le m \le 1,\ 0 \le r \le 1, 0 \le s \le 1,\ r \neq s \}$

Ejemplo 2.8 *Supongamos la gramática con producciones :*

$S \longrightarrow aS$
$S \longrightarrow aB$
$B \longrightarrow bC$
$C \longrightarrow aC$
$C \longrightarrow a$

Obtener el lenguaje generado L(G).
Volviendo a los ejemplos 2.6 y 2.7, el lenguaje generado siempre ha sido expresado en función de los símbolos terminales. Una sentencia o instrucción de un lenguaje, es una concatenación de tokens.



El aplicar inicialmente las reglas o producciones recursivas, tiene como objetivo encontrar las cadenas o sentencias más simples. Agrupemos la gramática :

S ⟶ aS | aB
B ⟶ bC
C ⟶ aC | a

Las reglas no recursivas son :

S ⟶ aB
B ⟶ bC
C ⟶ a

Obtengamos una derivación con estas producciones :

$S \underset{S \to aB}{\Rightarrow} aB \underset{B \to bC}{\Rightarrow} abC \underset{C \to a}{\Rightarrow} \mathbf{aba}$

Sentencia más simple !!!

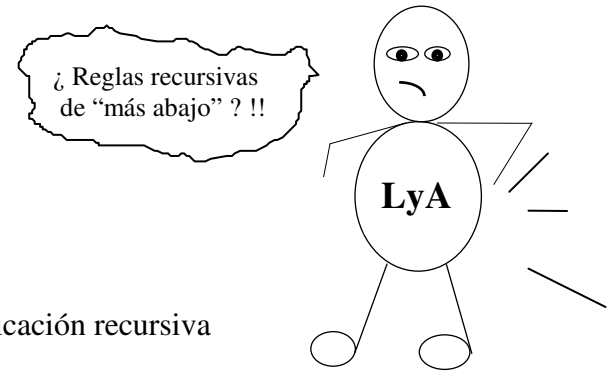
Ahora, utilicemos la recursividad de *"más abajo",* es decir C⟶ aC .

$S \underset{S \to aB}{\Rightarrow} aB \underset{B \to bC}{\Rightarrow} abC \underset{C \to aC}{\Rightarrow} abaC \underset{C \to aC}{\Rightarrow} abaaC \underset{C \to aC}{\Rightarrow} abaaaC$

$\Rightarrow abaa_{C \to a} \quad \Rightarrow abaaa_{C \to a} \quad \Rightarrow abaaaa_{C \to a}$

Esta regla produce ***a's*** a la derecha de la ***b*** solamente !!!

La regla recursiva "más abajo" vista desde el símbolo de inicio de la gramática es precisamente C $\longrightarrow$ aC. La regla recursiva siguiente "más abajo", se busca ahora en las alternativas para la no terminal B. En este caso B no produce recursividad, por lo que la regla recursiva siguiente "más abajo" es : S $\longrightarrow$ aS .



Observemos que es lo que produce la aplicación recursiva de esta regla.

S $\Rightarrow$ aS $\Rightarrow$ aaB $\Rightarrow$ aabC $\Rightarrow$ **aaba**
S $\Rightarrow$ aaS $\Rightarrow$ aaaB $\Rightarrow$ aaabC $\Rightarrow$ **aaaba**
S $\Rightarrow$ aaaS $\Rightarrow$ aaaaB $\Rightarrow$ aaaabC $\Rightarrow$ **aaaaba**
S $\Rightarrow$ aaaaS $\Rightarrow$ aaaaaB $\Rightarrow$ aaaaabC $\Rightarrow$ **aaaaaba**

La regla S $\longrightarrow$ aS produce *a's* a la izquierda de la *b* **!!** . Así, el lenguaje generado es identificado como :

$$\mathbf{L\ (G)\ =\ \{\ a^m\ b\ a^n\ \mid\ m,n > 0\ \}}$$

Ejemplo 2.9 *Sea la gramática con producciones :*

Φ

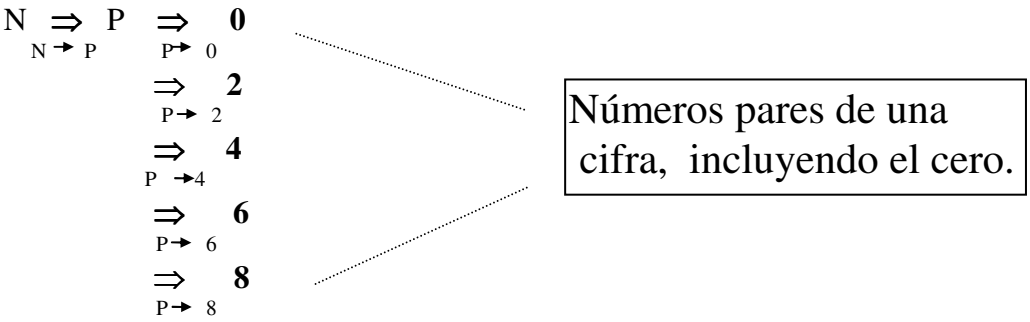
$N \longrightarrow DN$
$N \longrightarrow P$
$P \longrightarrow 0$
$P \longrightarrow 2$
$P \longrightarrow 4$
$P \longrightarrow 6$
$P \longrightarrow 8$
$D \longrightarrow 0$
$D \longrightarrow 1$
$D \longrightarrow 2$
$D \longrightarrow 3$
.
$D \longrightarrow 9$

Obtener el lenguaje generado L(G), donde G = ({0,1,2..,9}, {N,P,D}, S, Φ) .

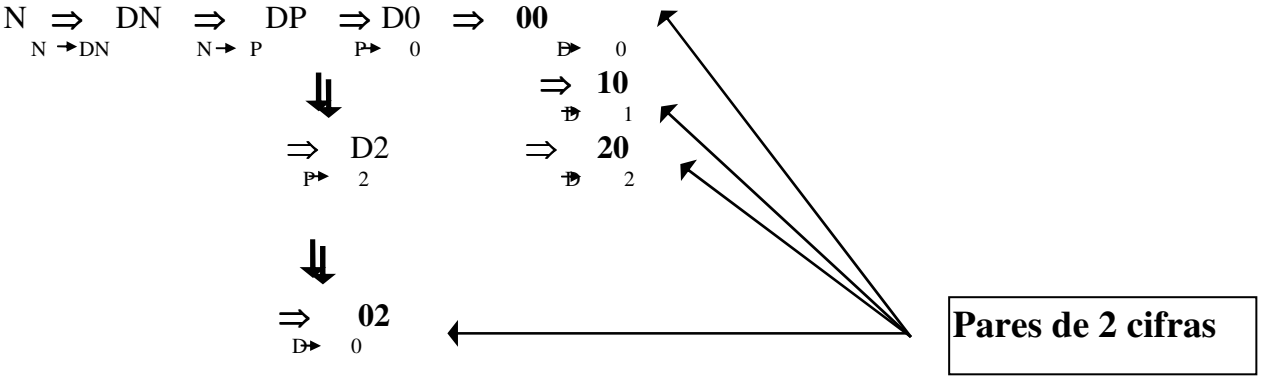
Agrupamos las alternativas para cada una de las no terminales -variables sintácticas- en

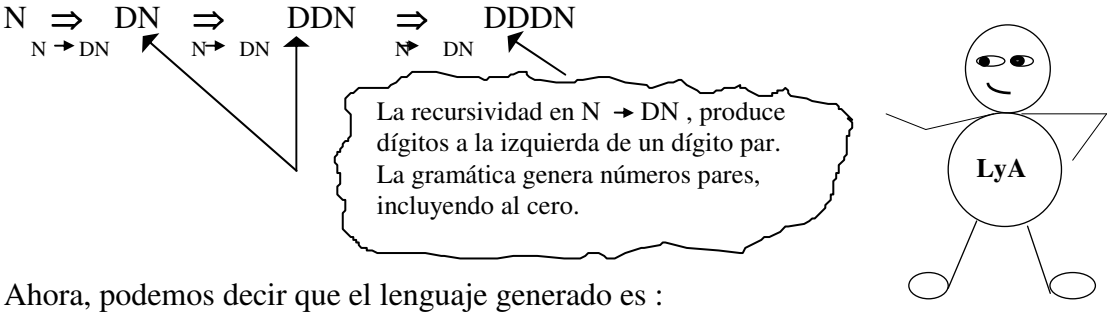
G :
N ⟶ DN | P
P ⟶ 0 | 2 | 4 | 6 | 8
D ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

El número de producciones de Φ es 17, no obstante que después de la agrupación pudieran parecer sólo 3. Únicamente existe una regla recursiva N⟶ DN. Las sentencias más simples, se obtienen con una derivación utilizando sólo reglas no recursivas.



Ahora, aplicamos la regla recursiva *"más abajo",* que resulta ser : N ⟶ DN.





Ahora, podemos decir que el lenguaje generado es :

L (G) = { dig^n Par | *dig* ∈ *{0,1,2,....9}* , *par* ∈ *{0,2,4,6,8}* , *n >= 0* }

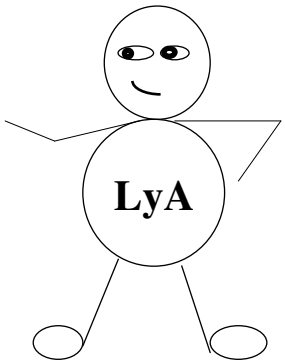
Ejemplo 2.10 *Sea el conjunto de producciones Φ :*

S → *aS*
S → *bA*
A → *aA*
A → *aB*
B → *b*

Obtener el lenguaje generado por la gramática G = ({a,b} , {S,A,B}, S, Φ).

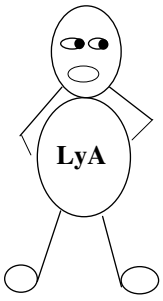
S → aS | bA
A → aA | aB
B → b

Recuerda !!
El primer paso es agrupar las producciones



Recursivas	No recursivas
S → aS A → aA	S → bA A → aB B → b

Luego, identifica las reglas recursivas y las no recursivas.



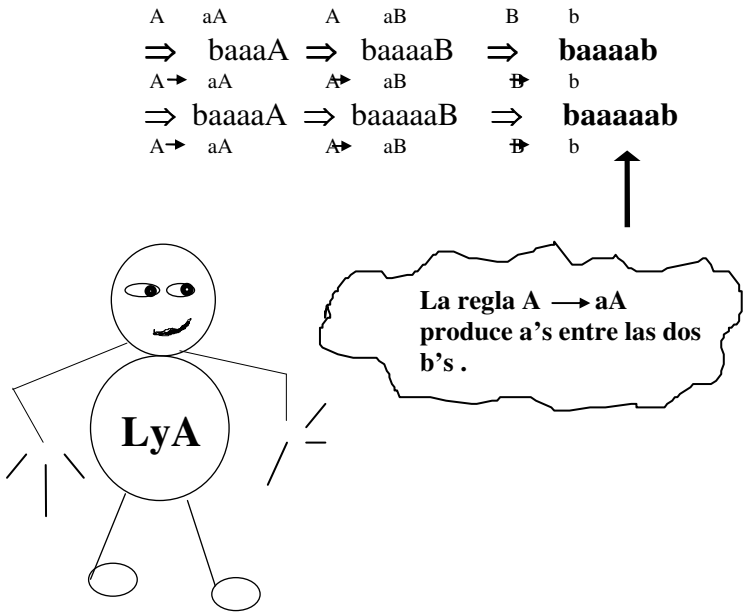
Ahora, encuentra las cadenas más simples, derivando sólo con producciones no recursivas.

S ⇒ bA ⇒ baB ⇒ **bab**

Cadena más simple

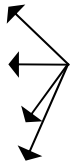
S ⇒ bA ⇒ baA ⇒ baaB ⇒ **baab**
(S → bA, A → aA, A → aB, B → b)
⇒ baaA ⇒ baaaB ⇒ **baaab**
(→, →, →)

Busca la regla recursiva "más abajo", y aplicala para observar las cadenas que produce :
A → aA



Sólo falta aplicar la otra regla recursiva. Observemos las cadenas que produce :

S ⇒ aS ⇒ abA ⇒ abaB ⇒ **abab**
⇒ aaS ⇒ aabA ⇒ aabaB ⇒ **aabab**
⇒ aaaS ⇒ aaabA ⇒ aaabaB ⇒ **aaabab**
⇒ aaaaS ⇒ aaaabA ⇒ aaaabaB ⇒ **aaaabab**



S → aS agrega **a's** a la izquierda de la primera **b**

De acuerdo a las anteriores sentencias derivadas, el lenguaje generado puede escribirse como :

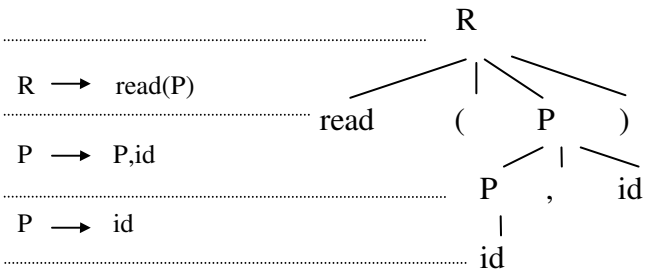
$$\mathbf{L(G) = \{\ a^m\ b\ a^n\ b\ \mid\ m >= 0\ ,\ n > 0\ \}}$$

Arboles de Reconocimiento (Arboles de Parse).

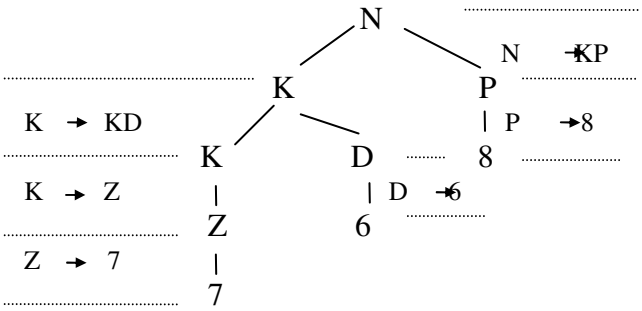
Un *árbol de parse* puede ser visto como una representación gráfica de una derivación, donde el orden en que son aplicadas las producciones es mostrado, debido a la naturaleza jerárquica del árbol.
Un *árbol de parse* tiene las siguientes características :

- La raíz del árbol es el símbolo de inicio de la gramática.
- Las hojas (nodos sin hijos) son símbolos terminales, es decir, tokens.
- Los nodos intermedios son etiquetados siempre por un símbolo no terminal.
- Un nodo y sus hijos constituyen una producción.

El árbol de parse que representa la derivación $R \overset{*}{\Rightarrow} read(x,y)$ es mostrado enseguida. La gramática es la del ejemplo 2.7.



En el árbol se aprecian 6 nodos sin hijos, es decir, hojas. Todos ellos etiquetados por un token. La raíz es el símbolo de inicio *R* de la gramática. Los nodos intermedios son 2 y ambos son etiquetados por la variable sintáctica *P*.
Tomemos la gramática del ejemplo 2.3 cuyo lenguaje generado son los números pares excluyendo al cero. El número par nunca empieza con cero. La derivación para la sentencia $N \overset{*}{\Rightarrow} 768$ y el árbol de parse son los siguientes :



$$N \Rightarrow KP \Rightarrow KDP \Rightarrow ZDP \Rightarrow 7DP \Rightarrow 76P \Rightarrow \mathbf{768}$$
$$N \to KP \quad K \to KD \quad K \to Z \quad Z \to 7 \quad D \to 6 \quad P \to 8$$

La raíz del árbol es etiquetado con el símbolo de inicio *N*. Existen 3 hojas etiquetadas por los tokens 7,6 y 8. Los nodos intermedios son 5, etiquetados por los símbolos no terminales *K, Z, D y P*.

Ambigüedad.

Se dice que una *gramática es ambigua*, si ésta produce más de un árbol de reconocimiento (parse) para una misma sentencia. O de otra manera, cuando la

gramática produce más de una derivación -ya sea izquierda ó derecha- para una misma sentencia.
Supongamos la gramática :

E ⟶ E+E | E-E | E*E | E/E | id | num y

G = ({ +, -, *, /, id, num }, { E }, E, Φ)

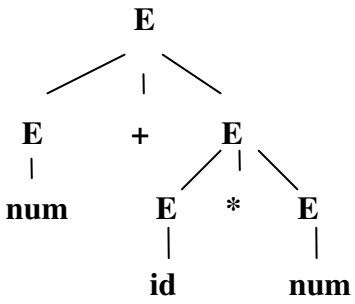
Obtengamos la derivación para la sentencia : *5 + id * 9*

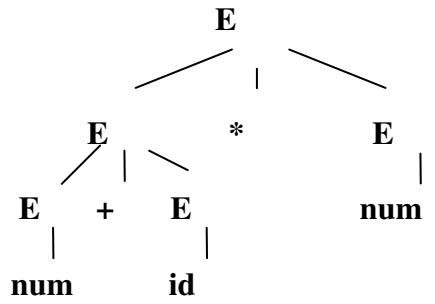
(1) $E \underset{E \to E+E}{\Rightarrow} E+E \underset{E \to num}{\Rightarrow} num+E \underset{E \to E*E}{\Rightarrow} num+E*E \underset{E \to id}{\Rightarrow} num+id*E \underset{E \to num}{\Rightarrow}$ **num+id*num**

Pero, existe otra derivación para la misma sentencia :

(2) $E \underset{E \to E*E}{\Rightarrow} E*E \underset{E \to E+E}{\Rightarrow} E+E*E \underset{E \to num}{\Rightarrow} num+E*E \underset{E \to id}{\Rightarrow} num+id*E \underset{E \to num}{\Rightarrow}$ **num+id*num**

Los arboles de parse para las dos derivaciones anteriores son :

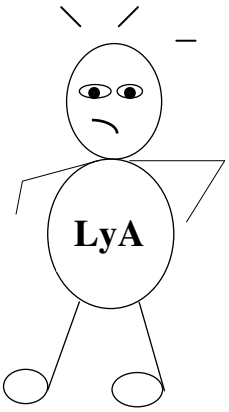




(1) *(2)*

¿ Será adecuado ... escribir gramáticas ambiguas ?

Y las gramáticas utilizadas anteriormente en los diferentes ejemplos ¿ son ambiguas ?



Hemos pretendido que las gramáticas anteriormente utilizadas no sean ambiguas. Además, para cierto tipo de reconocedores es deseable que las gramáticas no sean ambiguas, y para algunas aplicaciones podemos considerar métodos que pueden usar gramáticas ambiguas, pero conllevan la utilización de reglas para eliminar las derivaciones o arboles de parse no deseados, dejando sólo una opción de derivación ó de árbol, para cada una de las sentencias ó instrucciones. Como podemos ver, lo anterior traería consigo, una tarea de programación que puede llevar a una mayor complejidad o bien, a una mayor cantidad de código, cuando construimos un reconocedor. Generalmente, un *reconocedor con gramáticas ambiguas*, es *menos eficiente* que un reconocedor con gramáticas no ambiguas, cuando se trata de reconocer un mismo lenguaje (conjunto de sentencias).

Existe un ejemplo típico para indicar como una gramática ambigua puede traer "problemas" al derivar una sentencia en más de una forma no deseada. Nos referimos a la instrucción **if** . Propongamos la siguiente gramática para dicha instrucción en el lenguaje *C :*

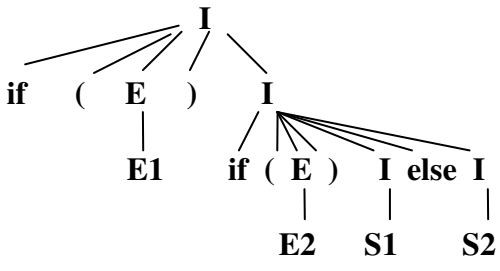
I ⟶ if (E) I | if (E) I else I | S

donde :

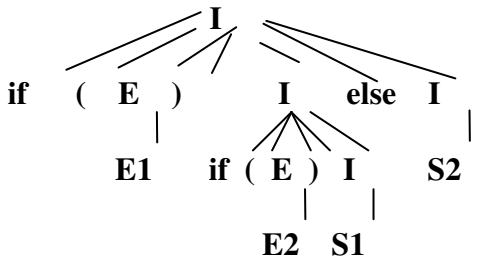
S es el símbolo de inicio que genera sentencias diferentes al *if* , y

E es el símbolo de inicio que genera sentencias de la clase *expresiones*.

De acuerdo a esta gramática, la instrucción : *if (E1) if (E2) S1 else S2* tiene dos arboles de parse, y por lo tanto la gramática es ambigua fig 2.6.



(a)



(b)

Fig 2.6 Arboles de parse para la sentencia *if (E1) if(E2) S1 else S2.*

Es claro, que el árbol que se prefiere es el de la fig 2.6(a), ya que la regla general es que *"el else pertenece al if más cercano"*. Modifiquemos la gramática para que no exista entre el "entonces" y el "sino" del **if**, una instrucción **if** que no tenga la rama else (sino).

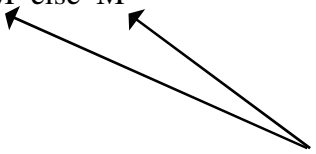
$$I \longrightarrow \underset{(1)}{M} \mid \underset{(2)}{N}$$

$$M \longrightarrow \text{if } (E) \underset{(3)}{M} \text{ else } M \mid \underset{(4)}{S}$$

$$N \longrightarrow \text{if } (E) \underset{(5)}{I} \mid \text{if } (E) \underset{(6)}{M} \text{ else } N$$

Tratemos de obtener más de una derivación para la sentencia : *if (E1) if (E2) S1 else S2* :

$$I \underset{(1)}{\Rightarrow} M \underset{(3)}{\Rightarrow} \text{if } (E)\ M \text{ else } M$$



Falla !!
Obliga a utilizar de nuevo *if (E) M else M* oó terminar con *S* .

Ahora, vayamos por otro camino es decir, utilicemos la alternativa (6) :

$$I \underset{(2)}{\Rightarrow} N \underset{(6)}{\Rightarrow} \text{if } (E)\ M \text{ else } N \Rightarrow \text{if } (E1)$$

También falla !!
Obliga a terminar con *M* → *S* ... (4), o bien a utilizar *M* → *if (E) M else M* ... (3), que sería fatal, ya que sólo existe un *else* en la sentencia .

Así, encontramos que la sentencia sólo es derivable siguiendo las etapas de sustitución que se muestran a continuación :

$$I \underset{(2)}{\Rightarrow} N$$

$$\underset{(5)}{\Rightarrow} \text{if } (E)\ I$$

$$\underset{(1)}{\Rightarrow} \text{if } (E1)\ M$$

$$\underset{(3)}{\Rightarrow} \text{if } (E1) \text{ if } (E)\ M \text{ else } M$$

$$\underset{(4)}{\Rightarrow} \text{if } (E1) \text{ if } (E2)\ S1 \text{ else } M$$

Sentencia reconocida !!

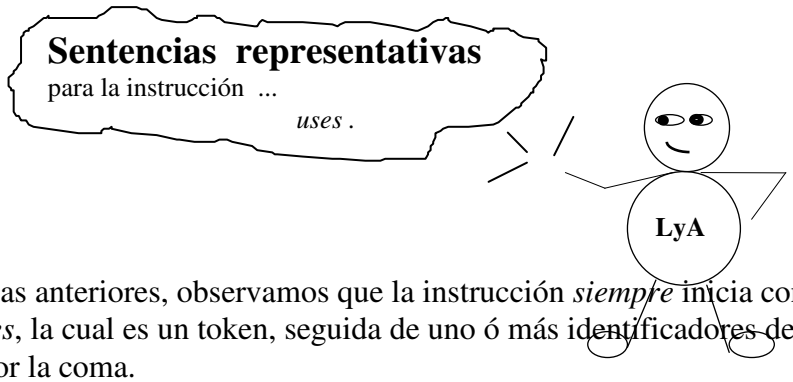
$\Rightarrow$ (5) *if (E1) if (E2) S1 else S2* $\longleftarrow$

Escritura de Gramáticas.

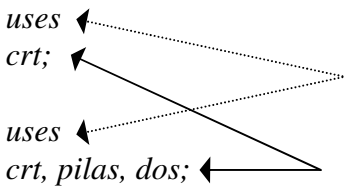
La *escritura de gramáticas* es una tarea muy parecida a la de escribir definiciones regulares. No existe una metodología definida para efectuar dicho trabajo, pero un buen empiezo es examinar las *sentencias más representativas* del lenguaje que deseamos generar con la gramática.

Ejemplo 2.11 *Deseamos escribir una gramática para especificar la sintáxis de la instrucción* ***uses*** *en Pascal. Esta instrucción se utiliza para declarar las bibliotecas ya sea predefinidas o bien definidas por el usuario, al inicio de un programa en Turbo Pascal. Restringiremos las bibliotecas predefinidas a : crt, printer, dos y graph.*

```
(1)     uses
          crt ;
(2)     uses
          crt, pilas, dos;
(3)     uses
          colas, listas;
        ...
        ...
```



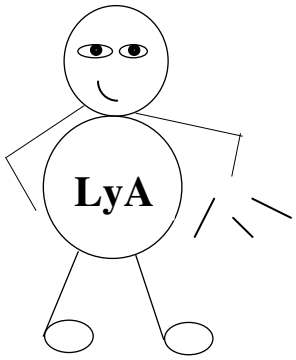
Analizando las sentencias anteriores, observamos que la instrucción *siempre* inicia con la palabra reservada *uses*, la cual es un token, seguida de uno ó más identificadores de bibliotecas separados por la coma.



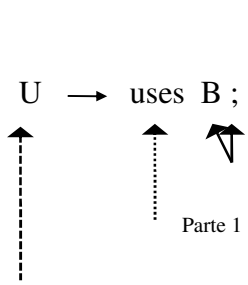
La instrucción tiene dos componentes :

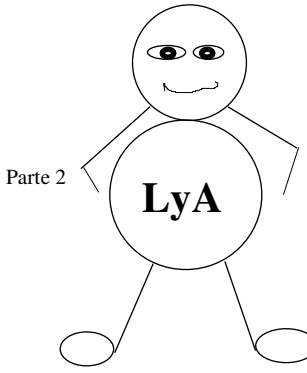
parte 1 ..... token *uses*

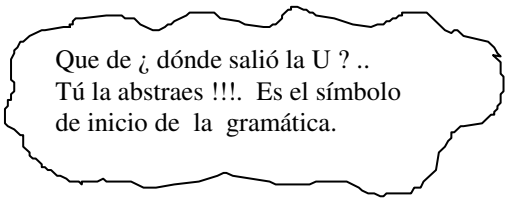
parte 2 ..... bibliotecas separadas por coma y la terminación punto y coma.



Observando lo anterior, proponemos la primera producción de la gramática :







Símbolo de inicio.

La variable sintáctica B, es necesaria para generar las cadenas de bibliotecas separadas por una coma.

La gramática se complementa añadiendo las siguientes producciones :

U ⟶ uses B ;
B ⟶ B , C | C ..... **(G1)**
C ⟶ id | crt | printer | dos | graph

Veamos el lenguaje producido por *B* :

$B \underset{B \to C}{\Rightarrow} C$ // genera una biblioteca, regla no recursiva.

$B \underset{B \to B,C}{\Rightarrow} B,C \underset{B \to C}{\Rightarrow} C,C$ // genera dos bibliotecas.

$B \underset{B \to B,C}{\Rightarrow} B,C \underset{B \to B,C}{\Rightarrow} B,B,C \underset{B \to C}{\Rightarrow} B,C,C \underset{B \to C}{\Rightarrow} C,C,C$ // tres bibliotecas.

...
...

De acuerdo a lo anterior, B genera las cadenas **C** (**, C**)$^{\mathbf{n}}$ n >= 0, donde el número de bibliotecas generadas es **n+1**.

n	**bibliotecas generadas**
0	1
1	2
2	3
3	4
...	...
n	n+1

La variable sintáctica *C* se utiliza para generar los tokens *id* (bibliotecas definidas por el usuario) y las bibliotecas predefinidas *crt, dos, graph y printer*.

Así, la gramática **G1** tiene :

8 producciones,

VT = { uses, , , ; , id, crt, printer, dos, graph },

VN = { U, B, C } y

el símbolo de inicio S = U .

Las derivaciones y arboles de parse para las sentencias representativas se muestran en la tabla de la figura 2.7.

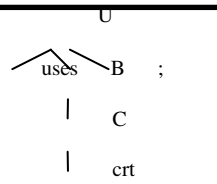
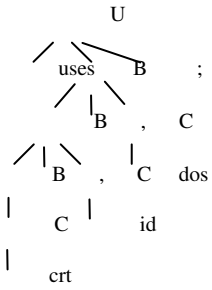
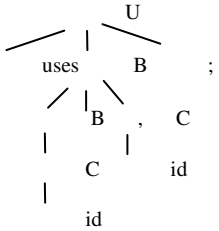
sentencia	derivación	árbol de parse
uses crt;	U ⇒ uses B; ⇒ uses C ; ⇒ uses crt;	
uses crt, pilas, dos;	U ⇒ uses B; ⇒ uses B,C; ⇒ uses B,C,C; ⇒ uses C,C,C; ⇒ uses crt,C,C; ⇒ uses crt,id,C; ⇒ uses crt,id,dos;	
uses colas, listas;	U ⇒ uses B; ⇒ uses B,C ; ⇒ uses C,C; ⇒ uses id,C ; ⇒ uses id,id;	

Fig2.7 Derivaciones y arboles de parse para las sentencias representativas de la instrucción uses en Turbo Pascal.

Ejemplo 2.12 *Supongamos la declaración de variables en Turbo Pascal, con sólo variables simples (no arreglos , no registros) de tipo integer, real y char. Entre las sentencias representativas que podemos tomar en cuenta, se encuentran las siguientes :*

(1)
```
var
  i,j : integer;
```

(2)
```
var
   x   : real;
  c,d  : char;
  k    : integer;
```

En las sentencias observamos que :
(a) Todas empiezan con el token *var* (palabra reservada),
(b) El token de var es seguido de renglones.

De acuerdo a lo anterior tenemos :

D ⟶ var R

D es el símbolo de inicio, y **R** es la variable sintáctica que genera las cadenas que hemos denominado *renglones.*

Dado que pueden existir más de un renglones, las alternativas para el no terminal *R* deben incluir al menos una regla recursiva.

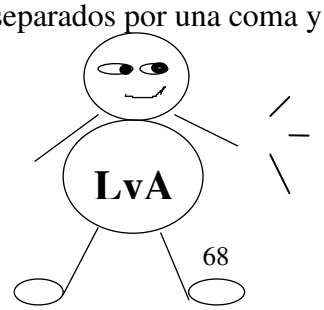
D ⟶ var R
R ⟶ R I : T ; | I : T ;

El lenguaje generado por *R* es encontrado empleando derivaciones de la siguiente forma :

R ⇒ I : T; // 1 renglón de declaraciones
(R → I:T;)
⇒ R I : T; ⇒ I : T; I : T; // 2 renglones
(R → RI:T; ; R → I:T;)
⇒ R I : T; ⇒ R I : T; I : T; ⇒ I : T; I : T; I : T; // 3 renglones
(R → RI:T; ; R → RI:T; ; R → I:T;)
...

R genera el lenguaje : (**I : T;**) ^{n} , n > 0. En todos los casos *I* representa a las cadenas de identificadores separados por una coma y *T* es el no terminal que denota a los tipos de datos.

I ⟶ I , id | id



I genera el lenguaje :

id (,id) ^{n} , n >= 0

Demuéstralo con derivaciones !!!

⟶

y *T* genera :

T char | integer | real

Juntando las producciones, tenemos la gramática que especifica la declaración de variables pedida :

D → *var R*
(1)

R → *R I : T ;* | *I : T ;*
(2) (3)

I → *I , id* | *id*
(4) (5)

T → *char* | *integer* | *real*
(6) (7) (8)

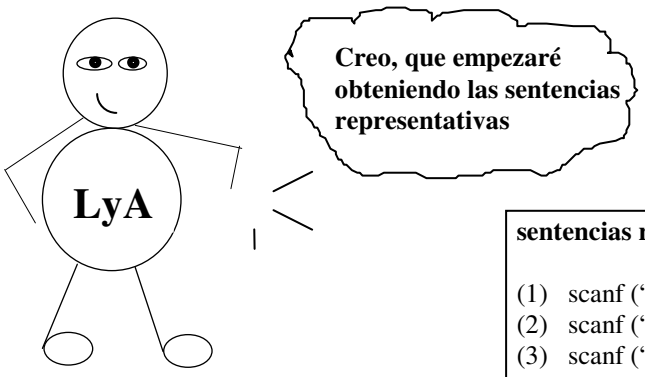
V_T = { *var* , : , ; , **,** , *id* , *char* , *integer* , *real* }
V_N = { *D* , *R* , *I* , *T* }
S = *D*

En la tabla de la fig. 2.8 tenemos las derivaciones a la izquierda para dos de las cadenas representativas.

sentencia	**derivación**
var i,j : integer;	D $\overset{1}{\Rightarrow}$ var R $\overset{3}{\Rightarrow}$ var I : T; $\overset{4}{\Rightarrow}$ var I,id : T; $\overset{5}{\Rightarrow}$ var id,id : T; $\overset{7}{\Rightarrow}$ var id,id : integer;
var x : real; c,d : char; k : integer;	D $\overset{1}{\Rightarrow}$ var R $\overset{2}{\Rightarrow}$ var RI : T ; $\overset{2}{\Rightarrow}$ var RI : T; I : T; $\overset{3}{\Rightarrow}$ var I : T; I : T; I : T; $\overset{5}{\Rightarrow}$ var id : T; I : T; I : T; $\overset{8}{\Rightarrow}$ var id : real; I : T; I : T; $\overset{4}{\Rightarrow}$ var id : real; I,id : T; I : T; $\overset{5}{\Rightarrow}$ var id : real; id,id : T; I : T; $\overset{6}{\Rightarrow}$ var id : real; id,id : char; I : T; $\overset{5}{\Rightarrow}$ var id : real; id,id : char; id : T; $\overset{7}{\Rightarrow}$ var id : real; id,id : char; id : integer;

Fig 2.8 Derivaciones para sentencias representativas : declaración de variables en Pascal.

Ejemplo 2.13 *Encontrar la gramática para la instrucción scanf( ) en lenguaje C. Reconocer la lectura de arreglos n-dimensionales y que sólo acepten en sus índices números y variables; no expresiones. La especificación de formatos debe reconocerse sólo como el token CteLit -constante literal ó cadena-.*



sentencias representativas

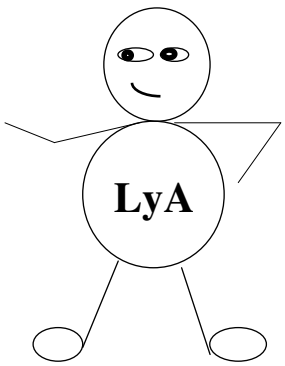
(1) scanf ("%d",&iNum);
(2) scanf ("%f%d",pfReal, piEnt);
(3) scanf ("%s ",sCad);
(4) scanf ("%d %d %d",&x[i],piy[i],&z[4] [1][i]);

Todas las sentencias representativas tienen el prefijo común : *scanf (CteLit,* seguido de los identificadores que pueden presentarse en las formas :

& id	dirección de una variable
id	apuntador a una variable
&id[ ]	dirección del elemento de un arreglo
id[ ]	apuntador al elemento de un arreglo

La primera producción es :

S → scanf (CteLit , V) ;



S es el símbolo de incio, y la no terminal *V* genera las cadenas para uno, dos o más identificadores.

$V \rightarrow V , I \mid I$

El lenguaje generado por V es : $L(V) = \{ \mathbf{I} (\mathbf{,I})^{n} \mid n >= 0 \}$

Puede comprobarse con las derivaciones siguientes :

$V \underset{V \rightarrow I}{\Rightarrow} I$ // 1 identificador

$\underset{V \rightarrow V,I}{\Rightarrow} V , I \underset{V \rightarrow I}{\Rightarrow} I , I$ // 2 identificadores

$\underset{V \rightarrow V,I}{\Rightarrow} V , I , I \underset{V \rightarrow I}{\Rightarrow} I , I , I$ // 3 identificadores

El símbolo no terminal *I* produce directamente con reglas no recursivas, las diferentes formas en que se presenta en la instrucción, un identificador de variable.

$I \rightarrow id \mid \& id \mid id\ K \mid \& id\ K$

$K \rightarrow K [L] \mid [L]$

$L \rightarrow id \mid num$

La variable sintáctica *K* produce las *n* dimensiones en el caso de lectura de arreglos.

$K \underset{K \rightarrow [L]}{\Rightarrow} [L]$ // 1 dimensión

$\underset{K \rightarrow K[L]}{\Rightarrow} K [L] \underset{K \rightarrow [L]}{\Rightarrow} [L] [L]$ // 2 dimensiones

$\underset{K \rightarrow K[L]}{\Rightarrow} K [L] [L] \underset{K \rightarrow [L]}{\Rightarrow} [L] [L] [L]$ // 3 dimensiones

El lenguaje generado por *K* es : $L(K) = \{ (\mathbf{[L]})^{n} \mid n > 0 \}$

Agrupando las producciones, obtenemos la gramática *G3 :*

$S \rightarrow scanf (CteLit , V) ;$ *(1)*

$V \rightarrow V , I \mid I$ *(2) (3)*

$I \rightarrow id \mid \& id \mid id\ K \mid \& id\ K$ ..... *(G3)* *(4) (5) (6) (7)*

$K \rightarrow K [L] \mid [L]$ *(8) (9)*

$L \rightarrow id \mid num$ *(10) (11)*

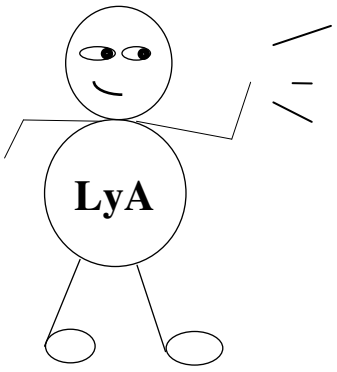
G(3) = ({ scanf , (,) , CteLit , , , ; , id , & , [,] , num } , { S , V , I , K , L } , S , Φ }

La sentencia scanf (“%d %c”, &x[i][0], pcCar); es reconocida aplicando la siguiente derivación a la izquierda :

S $\overset{1}{\Rightarrow}$ scanf (CteLit,V);
$\overset{2}{\Rightarrow}$ scanf (CteLit,V,I);
$\overset{3}{\Rightarrow}$ scanf (CteLit,I,I);
$\overset{7}{\Rightarrow}$ scanf (CteLit,&idK,I);
$\overset{8}{\Rightarrow}$ scanf (CteLit,&idK[L],I);
$\overset{9}{\Rightarrow}$ scanf (CteLit,&id[L][L],I);
$\overset{10}{\Rightarrow}$ scanf (CteLit,&id[id][L],I);
$\overset{11}{\Rightarrow}$ scanf (CteLit,&id[id][num],I);
$\overset{10}{\Rightarrow}$ **scanf (CteLit,&id[id][num],id);**

Precedencia y asociatividad de operadores. Recursividad a la izquierda. Recursividad a la derecha.

En una instrucción de asignación tal como *x = 4 + 5 * 8*, esperamos como resultado 44, es decir, que primero se efectúe el producto de 5*8 y luego la suma con 4. Lo anterior debido, a que la prioridad de ejecución de la multiplicación y la división es más alta, con respecto a la resta y la suma. Además, para la instrucción *x = 3+6*4*2-10,* la ejecución de la suma es primero que la ejecución de la resta. Al efectuar la evaluación de la expresión, la computadora realiza las operaciones de izquierda a derecha.



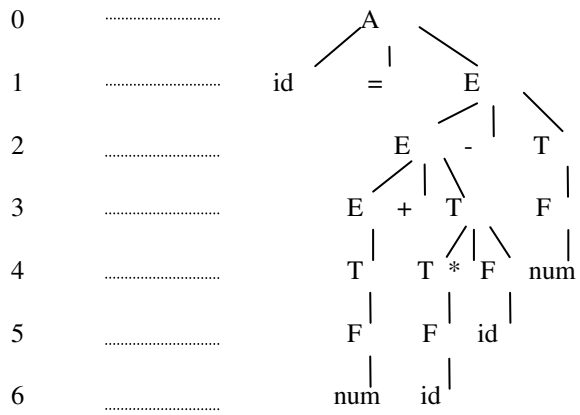
Evaluación de x = 3 + 6 * 4 * 2 - 10

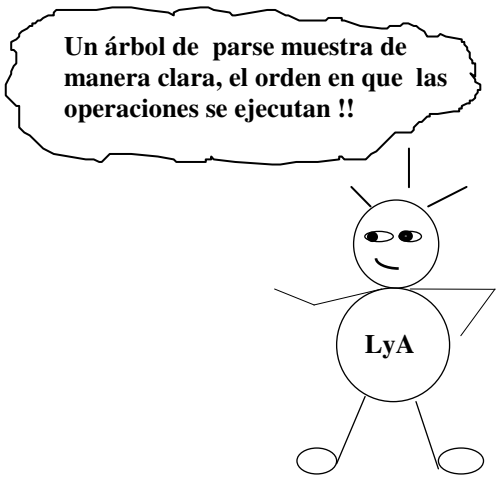
Paso 1 : x = 3 + 48 - 10 // multiplicación
Paso 2 : x = 51 - 10 // suma
Paso 3 : x = 41 // resta

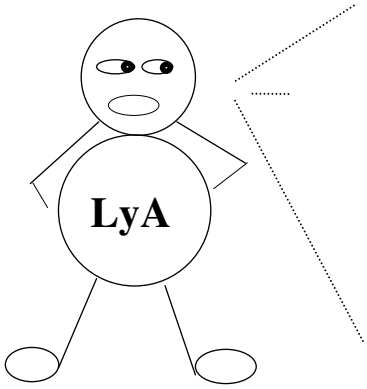
Una gramática que especifique una expresión, debe tomar en cuenta y representar en sus producciones Φ, la precedencia de ejecución de los operadores que intervienen ya sean aritméticos, relacionales, lógicos ó de otra naturaleza.
Por ejemplo, la gramática para la instrucción de asignación y expresiones aritméticas que incluyen los operadores +, -, *, / es típicamente escrita como :

(1) A ⟶ id = E
(2) E ⟶ E + T | E - T | T .... *(G4)*
(3) T ⟶ T * F | T / F | F
(4) F ⟶ id | num

La sentencia *x = 10 + y * z - 20* tiene el siguiente árbol de reconocimiento :







Observaciones :

(1) La asignación de *E* a id en el nivel 1 del árbol, no se puede realizar hasta que no se obtenga el valor de *E* .

(2) El valor de *E* en el nivel 1, se sabe hasta que se efectúa la resta *E - T* en el nivel 2.

(3) La resta en el nivel 2 *E - T* está sujeta a que se obtenga el valor de *E* y éste, es sabido hasta que se efectúe la suma *E + T* en el nivel 3.

(4) La suma *E + T* asimismo, depende de que se realice la multiplicación *T * F* en el nivel 4.

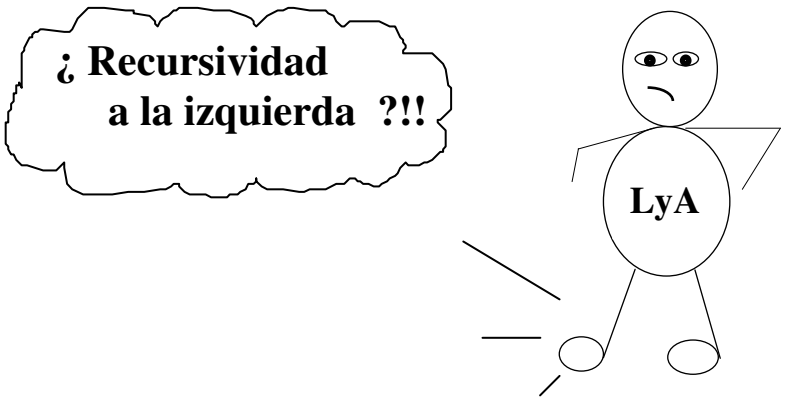
Concluyendo :

id = E sujeta a E - T
E - T sujeta a E + T
*E + T sujeta a T * F*

↑ **prioridad de ejecución.**

Pues como esperabamos, primero es realizada la multiplicación, enseguida la suma y por último la resta, para luego ceder el paso a la asignación !!.

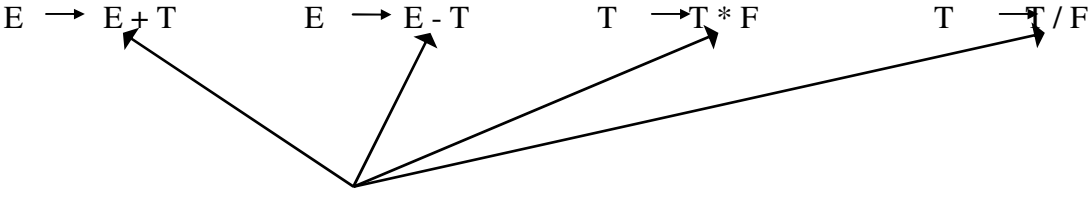
Obviamente, antes de efectuarse una operación, deben de producirse los operandos ya sean *id* o bien *num*. De acuerdo a las observaciones anteriores, terminaremos estipulando, que las operaciones con menos prioridad ocupan las producciones más cercanas (en derivación) a la primera producción -regla que contiene al símbolo de inicio-. Por ejemplo, la suma y resta ocupan el 2o. nivel después de la primera producción y sus reglas están etiquetadas con el (2) en la gramática *G4*. Le siguen las operaciones de multiplicación y división cuya prioridad es mayor, terminando con las producciones →
F id y F num , etiquetadas con el (3) y (4) en *G4*, respectivamente. Lo anteriormente dicho no es una regla, pero es útil su conocimiento.
La ejecución de las operaciones se realiza de izquierda a derecha. Esto es logrado, gracias a que las reglas recursivas han sido escritas con la *recursividad a la izquierda*.



Observemos las producciones para los símbolos no terminales *E* y *T* de la gramatica *G4.*

E → E + T | E - T | T
T → T * F | T / F | F

Las reglas o alternativas recursivas son :



Recursividad a la izquierda

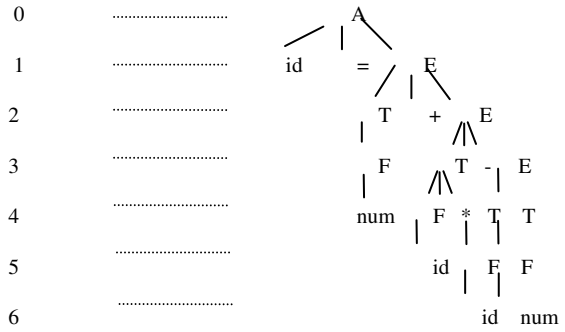
La recursividad aparece a la izquierda de la alternativa.

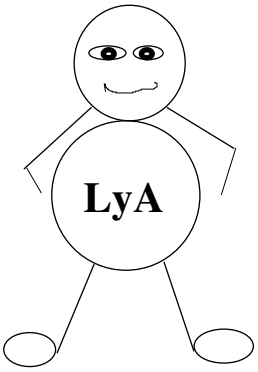
Podemos escribir estas reglas recursivas, conservando la *recursividad a la derecha*, tal y como lo mostramos enseguida :

(1) A → id = E
(2) E → T + E | T - E | T .... *(G5)*
(3) T → F * T | F / T | F
(4) F → id | num

La misma gramática *G4*
pero con .....
recursividad a la derecha !!

Veamos que sucede al obtener el árbol de parse para la sentencia *x = 10 + y * z - 20*, utilizando las producciones de *G5*.





Se conserva la precedencia de la multiplicación sobre la suma y la resta, pero ... se efectúa primero la resta que la suma !!!

Es decir, la ejecución se realiza de derecha a izquierda.

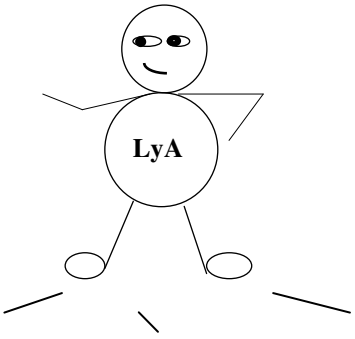
*x = 10 + y * z - 20*

75

ejecución

Exite un ejemplo típico de la aplicación de la *recursividad a la derecha*. Supongamos que deseamos obtener la gramática para especificar la instrucción de asignación transitiva en el lenguaje de programación *C*. En la sentencia *a = b = c = 0*; el cero es asignado a la variable *c*, luego el valor de *c* es asignado a *b* y por último el valor de *b* es asignado a *a*. Es decir, la operación de asignación es realizada de derecha a izquierda. El operador = es asociativo a la derecha. Un operador *asociativo a la derecha* se asocia u opera, sobre el operando situado a su izquierda. Un operador *asociativo a la izquierda* se asocia u opera, sobre el operando situado a su derecha. Son ejemplos de operadores asociativos a la izquierda : +, -, *, /.

En la sentencia - *x* + *y* ,
el - opera sobre la *x*
y el + opera sobre la *y*.



1 2 3

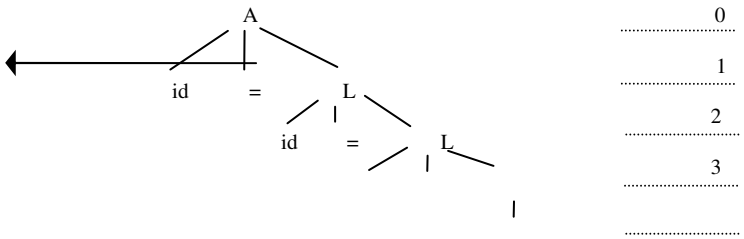
En nuestro ejemplo *a = b = c = 0;* el operador = (3) se asocia a la *c*, el operador = (2) opera sobre la *b* y el operador = (1) se asocia a la *a.*

La escritura de una regla que incluye operadores *asociativos a la derecha*, utiliza la *recursividad a la derecha*. Así, el conjunto de producciones para la gramática de la asignación transitiva es :

A ⟶ *id = L* ..... *(G6)*
L ⟶ *id = L | num*

G6 = ({ id , = , num } , { A , L } , A , Φ)

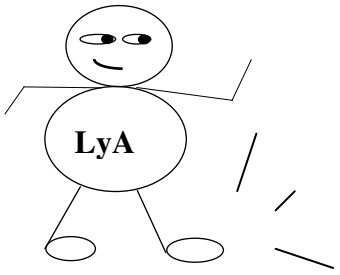
El árbol de parse para la sentencia *a = b = c = 0* es el siguiente :



0
1
2
3

id = L
4
num

La asignación del nivel 3 es la primera en efectuarse. Enseguida se realizan las asignaciones del nivel 2 y 1 en ese orden. *Se respeta la ejecución de derecha a izquierda.*



Operadores asociativos a la izquierda ....
Reglas con recursividad a la **izquierda.**
Operadores asociativos a la derecha ....
Reglas con recursividad a la **derecha.**

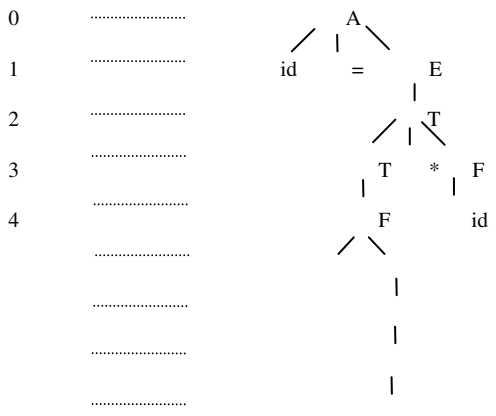
Ejemplo 2.14 *La gramática G4 no puede derivar las sentencias :*

(a) $x = - 5 * y$
(b) $x = x * (y + 2)$

La sentencia (*a)* involucra al operador *menos* en su modalidad de *monario* (generalmente es binario). La multiplicación se realiza sólo hasta que se haya asociado el *menos* al número 5. Lo anterior quiere decir que el operador monario *menos* tiene mayor precedencia que la multiplicación (operador *). La modificación a la gramatica *G4* debe efectuarse en el nivel siguiente a donde se encuentran las producciones para el producto (*) y la division (/), es decir, en las producciones del no terminal *F*. A continuación se muestra la modificación a *G4* que permite reconocer las sentencias tipo *(a).*

$A \rightarrow id = E$
$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow id \mid num \mid - E$

El árbol de parse para la sentencia $x = - 5 * y$, indica que primero se efectúa la asociación del signo menos al numero 5 para enseguida dar paso a la multiplicación por *y*.



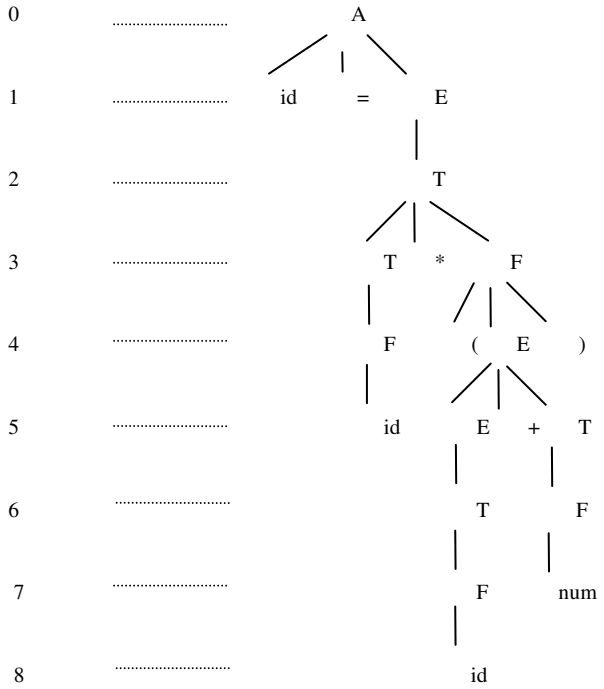
Para efectuar el producto en el nivel 3, es necesario saber el valor de *F* en el nivel 4, y *F* se sabe sólo hasta que se haya realizado la asociación del signo **-** a la expresión *E* del nivel 5. La expresión *E* del nivel 5 es el token *num.*

5	-	E
6		T
7		F
8		num

Para la sentencia *(b) x = x* (y+2)* se aplica el mismo razonamiento. Las expresiones encerradas entre paréntesis deben evaluarse antes que las demás, es decir, tienen más alta precedencia que las operaciones suma, resta, multiplicación y división. Por lo anterior, agregamos la modificación en las producciones para *F*.

A ⟶ id = E
E ⟶ E + T | E - T | T
T ⟶ T * F | T / F | F ..... *(G7)*
F ⟶ id | num | - E | (E)

El árbol de parse para las sentencias del ejemplo (b) es :



Orden de ejecución :

1o. Suma en el nivel 5
2o. Producto en el nivel 3
3o. Asignación nivel 1

Notación de Backus Naur (BNF).

Esta notación establece las siguientes reglas para representar gramáticas :

1. El símbolo *::=* significa *" es definido por "* y se utiliza en lugar del símbolo $\longrightarrow$.

2. Los símbolos no terminales -variables sintácticas- se denotan delimitándolos con los símbolos < y > . Por ejemplo el símbolo no teminal *E* se denota como < *E* >. Los símbolos terminales no son delimitados por ningún símbolo.

3. Las reglas recursivas que nos permiten expresar la repetición de cierto patrón en las cadenas de un lenguaje se escriben en forma muy diferente. La repetición en la notación *BNF* es expresada utilizando los símbolos *{* y *}*. Por ejemplo *{x}* significa la repetición de cero o más ocurrencias de x. $\{x\}_0^n$ indica la repetición de cero hasta *n ocurrencias de x* . Cuando se omite el límite inferior en las repeticiones, el valor por default es 0. Por ejemplo $\{ x \}^n$ denota las cero hasta *n* ocurrencias de *x.*

4. El uso de los símbolos *[* y *]* indican cero o una ocurrencia. *[x]* denota cero o una ocurrencia de *x. [x]* es equivalente a $\{ x \}^1$.

5. El uso de los paréntesis *(* y *)* denota la agrupación de alternativas .

<A> ::= <B> <C> | <B> <D>

se agrupan en :

<A> ::= <B> (<C> | <D>)

6. Una cadena que represente a un símbolo terminal -token- puede escribirse entre apóstrofes.

Por ejemplo :

<R> ::= read | readln | read ‘(‘ <P> ‘)’ | readln ‘(‘ <P> ‘)’
<P> ::= <P> , id | id

Los tokens *(* y *)* se encerraron entre apóstrofes, ya que puede existir confusión debido a que también son usados para indicar agrupación de alternativas. La *coma* y el *id* son tokens asi como el *read* y *readln.*

Ejemplo 2.15 *Convierte a la notación BNF la gramática del ejemplo 2.12.*

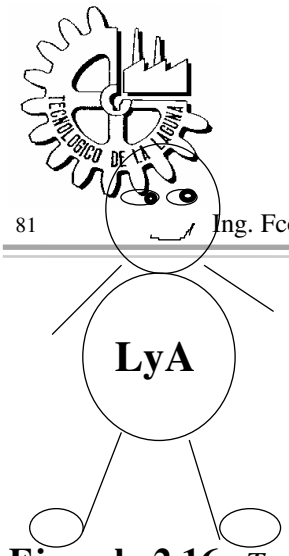
D ⟶ *var R*
R ⟶ *R I : T ;* | *I : T ;*
I ⟶ *I , id* | *id*
T ⟶ *char* | *integer* | *real*

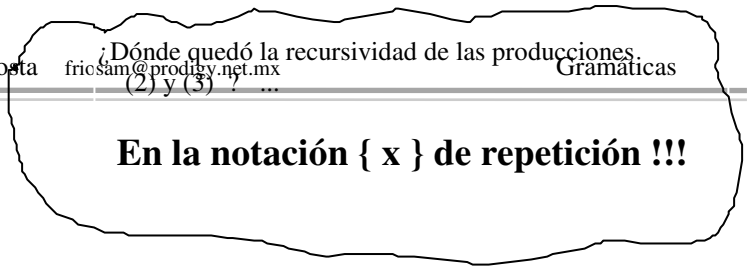
Aplicamos los estatutos 1 y 2 :

(1) <Declaración> ::= var <Renglón>
(2) <Renglón> ::= <Renglón> <ListaIdentificador> : <Tipo> ; |
<ListaIdentificador> : <Tipo> ;
(3) <ListaIdentificador> ::= <ListaIdentificador> , id | id
(4) <Tipo> ::= char | integer | real

Notemos que los tokens no los hemos denotado entre apóstrofes, ya que no es necesario debido a que no puede haber ningún tipo de confusión.
La gramática puede quedar escrita de la anterior manera en la notación BNF. Sin embargo podemos aplicar a las reglas recursivas, la notación BNF para la repetición (estatuto 3) .
La recursividad se presenta en la definición de < Renglón > y < Lista Identificador >.
La gramática en notación BNF después de aplicar el estatuto 3 es :

(1) <Declaración> ::= var <Renglón>
(2) <Renglón> ::= <ListaIdentificador> : <Tipo> ;
{ <ListaIdentificador> : <Tipo> ; }
(3) <ListaIdentificador> ::= id { , id }
(4) <Tipo> ::= char | integer | real





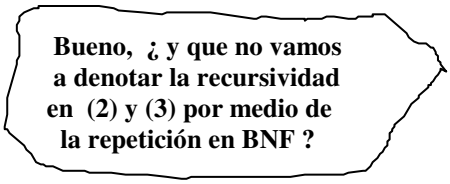
Ejemplo 2.16 *Tomemos la gramática G7 para la asignación, citada en el ejemplo 2.14.*

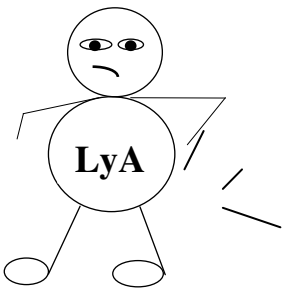
A ⟶ *id = E*
E ⟶ *E + T | E - T | T*
T ⟶ *T * F | T / F | F* ..... *(G7)*
F ⟶ *id | num | - E | (E)*

Hagamos la conversión a notación BNF, aplicando primeramente los estatutos 1 y 2 :

(1) <Asignación> ::= id = <Expresión>
(2) <Expresión> ::= <Expresión> + <Término> |
<Expresión> - <Término> |
<Término>
(3) <Término> ::= <Término> * <Factor> |
<Término> / <Factor> |
<Factor>
(4) <Factor> ::= id | num | - <Expresión> | ‘ (‘ <Expresión> ‘) ‘

El estatuto 6 es necesario aplicarlo a la producción para <*Factor*> (Producción 4), ya que el uso de paréntesis sin apóstrofes en la notación BNF tiene un significado especial -agrupación de alternativas-.

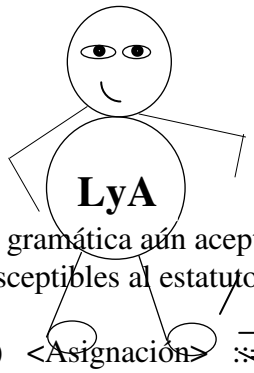




Claro que sí, el estatuto 3 es legalmente funcional en (2) y (3). La notación BNF de la gramática se transforma a :

(1) <Asignación> ::= id = <Expresión>
(2) <Expresión ::= <Término> { + <Término> } | <Término> { - <Término> }
(3) <Término> ::= <Factor> { * <Factor> } | <Factor> { / <Factor> }

(4) <Factor> ::= id | num | - <Expresión> | ‘ (‘ <Expresión> ‘) ‘



Vaya !!, es interesante cómo la recursividad ya no es tan obvia usando la notación BNF.

La gramática aún acepta otra forma en su notación BNF. Las producciones (2) y (3) son susceptibles al estatuto 5, es decir, a la agrupación de alternativas.

(1) <Asignación> ::= id = <Expresión>
(2) <Expresión> ::= <Término> ({ + <Término> } | { - <Término>})
(3) <Término> ::= <Factor> ({ * <Factor> } | { / <Factor> })
(4) <Factor> ::= id | num | - <Expresión> | ‘ (‘ <Expresión> ‘) ‘

Las producciones (2) y (3) siguen aceptando la agrupación de alternativas. La agrupación se muestra enseguida.

(1) <Asignación> ::= id = <Expresión>
(2) <Expresión> ::= <Término> { (+ | -) <Término> }
(3) <Término> ::= <Factor> { (* | /) <Factor> }
(4) <Factor> ::= id | num | - <Expresión> | ‘ (‘ <Expresión> ‘) ‘

Ejemplo 2.17 *Obtener la notación BNF para la gramática escrita en el ejemplo 2.13 (instrucción de lectura scanf), cuyo conjunto de producciones Φ se lista enseguida.*

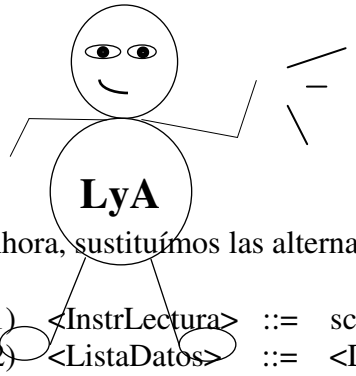
(1) S ⇒ scanf (CteLit , V)
(2) V ⇒ V , I | I
(3) I ⇒ id | &id | idK | &idK
(4) K ⇒ K [L] | [L]
(5) L ⇒ id | num

La gramática se modifica en su notación de la siguiente forma :

Aplicamos los estatutos 1 y 2.

(1) <InstrLectura> ::= scanf ‘(‘ CteLit , <ListaDatos> ‘)‘
(2) <ListaDatos> ::= <ListaDatos> , <Dato> | <Dato>
(3) <Dato> ::= id | &id | id <Dimensión> | &id <Dimensión>
(4) <Dimensión> ::= <Dimensión> ‘[‘ <Indice> ‘]‘ | ‘[‘ <Indice> ‘]‘

(5) <Indice> ::= id | num

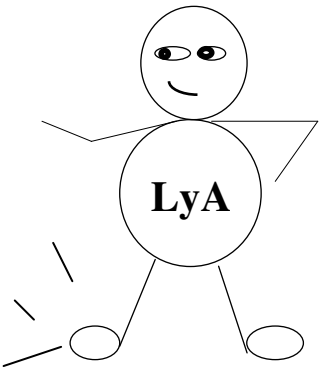


Los símbolos *(,) , [,]* se han delimitado con apóstrofes. El estatuto 6 ha sido empleado pues podrían confundirse estos símbolos con la agrupación de alternativas y la notación del estatuto 4.

Ahora, sustituímos las alternativas recursivas por la notación de repetición en BNF.

(1) <InstrLectura> ::= scanf '(' CteLit , <ListaDatos> ')'
(2) <ListaDatos> ::= <Dato> { , <Dato> }
(3) <Dato> ::= id | &id | id <Dimensión> | &id <Dimensión>
(4) <Dimensión> ::= { '[' <Indice> ']' } 1
(5) <Indice> ::= id | num

El estatuto 3 ha sido utilizado en las producciones (2) y (4). La notación en <Dimensión> indica una o más repeticiones !!!



Veamos ahora las producciones que cumplen las condiciones para el empleo del estatuto 5.

(3) <Dato> ::= id | &id | id <Dimensión> | &id <Dimensión>
* ! * !

Se observa que las alternativas marcadas con asterisco (*) coinciden en el prefijo *id*, y las alternativas marcadas con el símbolo de admiración (!) tienen *&id* como prefijo común. Agrupemos estas alternativas ayudándonos de los estatutos 5 y 4 .

<Dato> ::= id [<Dimensión>] | &id [<Dimensión>]

<Dato> ::= (id | &id) [<Dimensión>] ⟵

Se aplica el estatuto 4 : [x] 0 o una ocurrencia

↑

Aún puede aplicarse otra simplificación en el primer término : (id | &id) ⟶ <Dato> ::= [&] id [<Dimensión>]

Por último escribimos la gramática resultante en notación BNF.

<InstLectura> ::= scanf '(' CteLit , <ListaDatos> ')'
<ListaDatos> ::= <Dato> { , <Dato> }
<Dato> ::= [&] id [<Dimensión>]
<Dimensión> ::= { '[' <Indice> ']' } $_1$
<Indice> ::= id | num

2.5 EJERCICIOS PROPUESTOS.

1. *Dadas las siguientes gramáticas, obtener sus componentes, Vt, Vn, S y Φ :*

a) S → (L) | a
L → L , S | S

b) S → Aa | bAc | Bc | bBa
A → d
B → d

c) R → readln | readln (P)
P → P , Q | Q
Q → id | id [S]
S → S , T | T
T → id | num

2. *Encuentra el lenguaje generado por las siguientes gramáticas :*

(a) S → aSb | ab

(b) S → aSd | aAd
A → bAc | bc

(c) S → AB
A → aAb | ab
B → cBd | cd

(d) S → 0S1 | 01

(e) S → aSBC
S → abC
bB → bb
bC → bc
CB → BC
cC → cc

(f) A → A0B0 | A1B1
A0 → 1
A1 → 0
0B0 → 1
1B1 → 0

(g) K → K , T | T
T → num

(h) S → (S) S | ∈

(i) S → ccc
S → Abccc
A → Ab
A → aBa
B → aBa
B → AC
C → Cb
C → b

3. *Modificar la gramática del ejemplo 2.12 para que en adición especifique tipos subrango y tipos string [ ].*

```
var
   i , j        :   integer;
   subrango  :   1 .. MAX;
   limite       :   MIN .. 40;
   sNombre  :   string [40];
```

4. *Escribe la gramática para especificar el encabezado de :*

a) Una función en C. Los tipos son void, int, char, float y apuntadores a ellos.
b) Una función en Pascal. Los tipos aceptados son integer, char, real. El pasaje de parámetros por valor y por referencia.
c) Un procedure en Pascal. Los tipos y pasaje de parámetros igual que el inciso b).

5. *Encuentra la derivación y el árbol de parse para las sentencias :*

```
readln ( i , j , x [ i ,j ] )
readln ( x [ 3 ] , y [ i,j ] , z )
```

utilizando la gramática del ejercicio 1 c).

6. *Escribe la gramática para la instrucción puts en lenguaje C. Recuerda que puts escribe una cadena en la pantalla.*

```
puts ( "HOLA" );                    puts ( sCad [ 1 ] [ 5 ] );
puts ( sCad [ i ]);                 puts ( sLinea);
```

7. *Escribe la gramática para el printf en C. Los especificadores de formato no los toma en cuenta, es decir, el token CteLit (Constante literal o cadena) ya los considera. En la lista de datos a escribir aceptar sólo identificadores simples, apuntadores y arreglos.*

```
printf ( " HOLA \ n " );
printf ( " % d % f " , i , *x );
printf ( " % f " , y [ 0 ] [ i ] );
```

8. *El encabezado del FOR en Pascal.*

9. *Convierte a notación B.N.F. las gramáticas obtenidas en los ejercicios 4, 5, 6, 7 y 8.*

10. *Escribe la gramática para la declaración de tipos definidos por el usuario en Pascal. Incluye los arreglos de n dimensiones en notación* ***array [1 .. 10 , 1 .. max].***