SSL 1

Desde sus Usuarios

Jorge D. Muchnik
Ana M. Díaz Bott
2012, 2ª Edición

El libro “Sintaxis y Semántica de los Lenguajes” está formado por tres volúmenes:

Vol. 1 – Desde sus Usuarios (programadores y otros)

Vol.2 – Desde el Compilador

Vol.3 – Algoritmos

Este libro cubre todos los objetivos y los contenidos de la asignatura con el mismo nombre.

Por orden alfabético, los profesores de esta cátedra son: Adamoli, Adriana – Barca, Ricardo – Bruno, Oscar – Díaz Bott, Ana María – Ferrari, Marta – Ortega, Silvina – Sola, José Maria.

Jorge D. Muchnik y Ana M. Díaz Bott
febrero 2012, 2ª Edición

ÍNDICE

1 DEFINICIONES BÁSICAS E INTRODUCCIÓN A LENGUAJES FORMALES 7
1.1 CARACTERES Y ALFABETOS 7
1.2 CADENAS 8
1.2.1 LONGITUD DE UNA CADENA 8
1.2.2 CADENA VACÍA 9
1.2.3 UNA SIMPLIFICACIÓN: LA POTENCIACIÓN DE UN SÍMBOLO 9
1.2.4 CONCATENACIÓN DE DOS CADENAS 9
1.2.5 POTENCIACIÓN DE UNA CADENA 10
1.3 LENGUAJES NATURALES Y LENGUAJES FORMALES 11
1.3.1 PALABRA 13
1.3.2 PROPIEDADES DE LAS PALABRAS 14
1.3.3 CARDINALIDAD DE UN LENGUAJE FORMAL 15
1.3.4 SUBLENGUAJES 15
1.4 LENGUAJES FORMALES INFINITOS 16
1.4.1 LENGUAJE UNIVERSAL SOBRE UN ALFABETO 16
1.4.2 LENGUAJES FORMALES INFINITOS MÁS COMPLEJOS 17
1.5 IMPLEMENTACIÓN EN C 18

2 GRAMÁTICAS FORMALES Y JERARQUÍA DE CHOMSKY 19
2.1 GRAMÁTICA FORMAL 19
2.1.1 DEFINICIÓN FORMAL DE UNA GRAMÁTICA FORMAL 20
2.2 LA JERARQUÍA DE CHOMSKY 22
2.2.1 GRAMÁTICA REGULAR (GR) 22
2.2.1.1 GRAMÁTICA REGULAR QUE GENERA UN LENGUAJE REGULAR INFINITO . . 23
2.2.1.2 GRAMÁTICA QUASI-REGULAR (GQR) 24
2.2.2 GRAMÁTICA INDEPENDIENTE DEL CONTEXTO (GIC) 25
2.2.2.1 GIC QUE GENERA UN LENGUAJE INFINITO 26
2.2.3 GRAMÁTICA IRRESTRICTA 27
2.2.4 GRAMÁTICA SENSIBLE AL CONTEXTO (GSC) 27
2.3 EL PROCESO DE DERIVACIÓN 27
2.4 INTRODUCCIÓN A LAS GQRs, LAS GICs Y LOS LENGUAJES DE PROGRAMACIÓN 30

3 SINTAXIS Y BNF 33
3.1 INTRODUCCIÓN A LA SINTAXIS 33
3.2 IDENTIFICADORES Y SU SINTAXIS 34
3.3 LAS EXPRESIONES Y LA SINTAXIS 36
3.3.1 LA EVALUACIÓN DE UNA EXPRESIÓN, PRECEDENCIA Y ASOCIATIVIDAD . . 38
3.4 BNF Y ALGOL 42
3.4.1 DOS EJEMPLOS DE BNF EN ALGOL 43
3.5 BNF Y EL LENGUAJE PASCAL 44
3.5.1 LA SINTAXIS DEL LENGUAJE PASCAL, SEGÚN WIRTH 44
3.5.2 EXPRESIONES EN PASCAL 47
3.5.3 SENTENCIAS CON CONDICIONES BOOLEANAS 49

3.6 BNF Y EL ANSI C 50
3.6.1 UN PROGRAMA EN ANSI C 51
3.6.2 SINTAXIS DE UN SUBCONJUNTO DE ANSI C 53
3.6.2.1 LAS CATEGORÍAS LÉXICAS 53
3.6.2.1.1 LAS PALABRAS RESERVADAS 55
3.6.2.1.2 LOS IDENTIFICADORES 55
3.6.2.1.3 LAS CONSTANTES 55
3.6.2.1.4 LOS OPERADORES Y LOS CARACTERES DE PUNTUACIÓN 57
3.6.2.2 LAS CATEGORÍAS GRAMATICALES 58
3.6.2.2.1 LAS EXPRESIONES 58
3.6.2.2.2 DECLARACIONES Y DEFINICIONES 62
3.6.2.2.3 SENTENCIAS 64
3.7 OTROS USOS DE BNF 66

4 SEMÁNTICA 67
4.1 DEFINICIÓN DE UN LENGUAJE DE PROGRAMACIÓN 67
4.2 EL ANSI C Y SU SEMÁNTICA 67
4.2.1 IDENTIFICADOR 68
4.2.2 SEMÁNTICA DE LAS CONSTANTES 68
4.2.2.1 CONSTANTES ENTERAS 68
4.2.2.2 CONSTANTES REALES 69
4.2.2.3 CONSTANTES CARÁCTER 70
4.2.2.4 LITERAL CADENA 70
4.2.3 SEMÁNTICA DE LOS OPERADORES 71
4.2.4 SEMÁNTICA DE LOS CARACTERES DE PUNTUACIÓN 71
4.2.5 SEMÁNTICA DE LAS EXPRESIONES 71
4.2.6 SEMÁNTICA DE LAS DECLARACIONES Y DEFINICIONES 73
4.2.7 SEMÁNTICA DE LAS SENTENCIAS 73

5 LENGUAJES REGULARES Y EXPRESIONES REGULARES 75
5.1 DETERMINACIÓN DE LENGUAJES REGULARES 75
5.2 LAS EXPRESIONES REGULARES 76
5.2.1 EXPRESIONES REGULARES PARA LENGUAJES REGULARES FINITOS 77
5.2.1.1 EL OPERADOR POTENCIA 79
5.2.2 EXPRESIONES REGULARES PARA LENGUAJES REGULARES INFINITOS . . . 80
5.2.2.1 EL OPERADOR “CLAUSURA POSITIVA” 81
5.2.2.2 LA EXPRESIÓN REGULAR UNIVERSAL Y SU APLICACIÓN 83
5.3 DEFINICIÓN FORMAL DE LAS EXPRESIONES REGULARES 85
5.4 OPERACIONES SOBRE LENGUAJES REGULARES Y LAS EXPRESIONES REGULARES CORRESPONDIENTES 86
5.4.1 GENERALIDADES 86
5.4.2 LA UNIÓN DE LENGUAJES REGULARES 86
5.4.3 LA CONCATENACIÓN DE LENGUAJES REGULARES 87
5.4.4 LA CLAUSURA DE KLEENE DE UN LENGUAJE REGULAR 87
5.4.5 LA CLAUSURA POSITIVA DE UN LENGUAJE REGULAR 88

5.4.6 EL COMPLEMENTO DE UN LENGUAJE REGULAR 88
5.4.7 LA INTERSECCIÓN DE DOS LENGUAJES REGULARES 89
5.5 EXPRESIONES REGULARES Y LENGUAJES DE PROGRAMACIÓN 89
5.6 EXPRESIONES REGULARES EXTENDIDAS 92
5.6.1 UN METALENGUAJE PARA EXPRESIONES REGULARES 92
5.6.1.1 EL METALENGUAJE, SUS METACARACTERES Y SUS OPERADORES 92
5.6.1.2 EJERCICIOS PARA ESCRIBIR META EXPRESIONES REGULARES 94
5.6.1.3 UNA APLICACIÓN: LEX Y EJEMPLOS 95

6 BIBLIOGRAFÍA 99

7 EJERCICIOS RESUELTOS 101
- Capítulo 1 101
- Capítulo 2 104
- Capítulo 3 108
- Capítulo 4 123
- Capítulo 5 127

1 DEFINICIONES BÁSICAS E INTRODUCCIÓN A LENGUAJES FORMALES

El Volumen 1 de este libro trata la Sintaxis y la Semántica de los Lenguajes de Programación (LPs) desde el punto de vista de los que necesitan conocerla, sus usuarios, mientras que el Volumen 2 la analizará desde el punto de vista del compilador.

NOTA MUY IMPORTANTE

PARA UNA BUENA COMPRENSIÓN DE TODOS LOS TEMAS,
SIEMPRE RESUELVA LOS EJERCICIOS EN LA MEDIDA QUE VAN APARECIENDO.
NO LOS DEJE COMO ÚLTIMA TAREA DE CADA CAPÍTULO.

La SINTAXIS de un LENGUAJE DE PROGRAMACIÓN describe las combinaciones de símbolos que forman un programa sintácticamente correcto. Como esta SINTAXIS está basada en los LENGUAJES FORMALES, debemos comenzar con este concepto.

Los LENGUAJES FORMALES están formados por PALABRAS, las palabras son CADENAS y las cadenas están constituidas por CARACTERES de un ALFABETO. Esta frase involucra cinco términos (*lenguaje formal*, *palabra*, *cadena*, *carácter* y *alfabeto*) que son fundamentales en este campo. A continuación los analizamos detenidamente.

1.1 CARACTERES y ALFABETOS

Un CARÁCTER (también llamado SÍMBOLO) es el elemento constructivo básico. Es la entidad fundamental, indivisible, a partir de la cual se forman los alfabetos.

Un ALFABETO es un conjunto finito de caracteres. Se lo identifica, habitualmente, con la letra griega Σ (sigma), y con sus caracteres se construyen las cadenas de caracteres de un Lenguaje Formal.

Ejemplo 1
La letra a es un carácter o símbolo que forma parte del alfabeto español, del alfabeto inglés, etc.

Ejemplo 2
Los caracteres >, = y + son elementos del alfabeto de los operadores de muchos Lenguajes de Programación.

Ejemplo 3
El alfabeto $\Sigma = \{0, 1\}$ proporciona los caracteres utilizados en la construcción de los números binarios.

Un carácter de un alfabeto también puede ser “múltiple”.

Ejemplo 4
El alfabeto $\Sigma = \{ab, cde\}$ está integrado por dos caracteres: el carácter ab y el carácter cde.

* Ejercicio 1 *
Escriba el alfabeto que se requiere para construir el conjunto de los números enteros con signo en base 10.

1.2 CADENAS

Una CADENA, forma abreviada de la frase *cadena de caracteres*, es una secuencia finita de caracteres tomados de cierto alfabeto y colocados uno a continuación de otro. Es decir: una *cadena* se construye CONCATENANDO caracteres de un alfabeto dado.

Como sinónimo de *cadena* se usa, a veces, el término inglés *string*.

Ejemplo 5
abac (se lee "a-b-a-c") es una cadena formada con caracteres del alfabeto {a, b, c}.

Ejemplo 6
101110 ("uno-cero-uno-uno-uno-cero") es una cadena construida con caracteres del alfabeto {0, 1}.

Ejemplo 7
a es una cadena formada por un solo símbolo de cualquier alfabeto que contenga el carácter a.

➔ Para los conocedores del Lenguaje de Programación C: recuerden la diferencia que existe entre 'a' y "a". Es la misma diferencia que existe entre un carácter de un alfabeto y una cadena compuesta por ese único carácter.

* Ejercicio 2 *
Dado el alfabeto {0, 1, 2}, construya dos cadenas en la que cada uno de estos caracteres aparezca una sola vez.

* Ejercicio 3 *
Dado el alfabeto {ab, cde}, construya una cadena que tenga cuatro caracteres

1.2.1 LONGITUD DE UNA CADENA

La LONGITUD de una cadena S (se representa |S|) es la cantidad de caracteres que la componen.

Ejemplo 8
La longitud de la cadena abac es: |abac| = 4.

Ejemplo 9
La longitud de la cadena b es: |b| = 1.

1.2.2 CADENA VACÍA

La CADENA VACÍA, que se simboliza habitualmente con la letra griega ε (épsilon), es la cadena que no tiene caracteres. En otras palabras: la *cadena vacía* es la cadena de longitud 0 ($|\varepsilon| = 0$).

Nota 1
Este símbolo ε no forma parte de ningún alfabeto.

Nota 2
Algunos autores utilizan la letra griega λ (lambda) para representar a la cadena vacía.

1.2.3 UNA SIMPLIFICACIÓN: LA POTENCIACIÓN DE UN SÍMBOLO

En muchas ocasiones, una cadena puede contener un carácter que se repite un número determinado de veces.

Ejemplo 10
La cadena aaaabbbbbbb, construida sobre el alfabeto {a, b}, está formada por cuatro aes concatenadas con siete bes.

La POTENCIACIÓN de un carácter simplifica la escritura de cadenas como la del ejemplo anterior, de la siguiente manera:

Sea c un carácter cualquiera; entonces, c^n representa la concatenación del carácter c, consigo mismo, n−1 veces. En otras palabras, c^n representa la repetición del carácter c, n veces. Por lo tanto: $c^1 = c$, $c^2 = cc$, $c^3 = ccc$, etc.

Nota 3
Para un carácter, no existe la "potencia" 0.

Ejemplo 11
La cadena del Ejemplo 10 (aaaabbbbbbb) se puede escribir, más sintéticamente, así: a^4b^7.

Como se puede apreciar, esta simplificación también produce una mejor y más rápida lectura –y, por lo tanto, una mejor comprensión– de la cadena en cuestión.

* Ejercicio 4 *
Simplificando con el uso de la "potenciación", escriba la cadena que tiene 1300 aes seguidas de 846 bes seguidas de 257 aes.

1.2.4 CONCATENACIÓN DE DOS CADENAS

La operación de CONCATENACIÓN aplicada a cadenas (se escribe $S_1 \bullet S_2$ o, simplemente, S_1S_2) produce una nueva cadena formada por los caracteres de la primera cadena seguidos inmediatamente por los caracteres de la segunda cadena.

Ejemplo 12
Sean las cadenas $S_1 = aab$ y $S_2 = ba$; entonces, $S_1S_2 = aabba$.

Nota 4
La concatenación se escribe, normalmente, sin utilizar el operador correspondiente (•), como en el ejemplo anterior. Simplemente, se coloca una cadena a continuación de la otra.

Nota 5
La concatenación se extiende, fácilmente, a más de dos cadenas.

Ejemplo 13
$S_1 \bullet S_2 \bullet S_3 = S_1S_2S_3$.

* Ejercicio 5 *
Sean las cadenas $S_1 = aab$ y $S_2 = ba$. Obtenga la cadena $S_1S_2S_1S_2$.

➔ La concatenación NO ES CONMUTATIVA (excepto en casos muy especiales).

Ejemplo 14
La cadena aa concatenada con la cadena bb produce la cadena $aabb$, mientras que la cadena bb concatenada con la cadena aa produce la cadena $bbaa$. Estas dos cadenas son diferentes; en consecuencia, la concatenación no es siempre conmutativa.

A continuación, veamos algunos casos especiales en los que la concatenación sí es conmutativa.

CASO 1: Si ambas cadenas son idénticas, entonces la concatenación es conmutativa.

Ejemplo 15
Si la primera cadena es ab y la segunda cadena también es ab, entonces: $ab \bullet ab$ (en cualquier orden) es $abab$.

CASO 2: Cuando ambas cadenas están compuestas por una repetición del mismo carácter, entonces la concatenación es conmutativa.

Ejemplo 16
Sea $S_1 = aa$ y sea $S_2 = aaa$. Entonces: $S_1S_2 = a^2a^3 = a^{2+3} = a^5$ y $S_2S_1 = a^3a^2 = a^{3+2} = a^5$. En consecuencia, $S_1S_2 = S_2S_1 = a^5$.

CASO 3:
➔ La cadena vacía (ε) es la IDENTIDAD para la concatenación. Esto es: para cualquier cadena S, $S \bullet \varepsilon = \varepsilon \bullet S = S$.

1.2.5 POTENCIACIÓN DE UNA CADENA

El supraíndice utilizado en la sección 1.2.3 para representar la repetición de un carácter se extiende fácilmente a una cadena, de esta manera: si S es una cadena, entonces S^n (con $n \geq 1$ y entero)

representa la cadena que resulta de CONCATENAR la cadena S, consigo misma, n-1 veces; es decir: la cadena final resulta de repetir la cadena original n veces. Por lo tanto:

$$S^1 = S,\ S^2 = SS,\ S^3 = SSS,\ \text{etc.}$$

➔ Esta definición se completa con la siguiente afirmación: S^0 es **ε** (la cadena vacía), para cualquier cadena S.

Ejemplo 17
Sea S = ab; entonces: $S^3 = SSS = (ab)^3 = ababab$.

Ejemplo 18
Nótese la diferencia que existe entre la cadena $(ab)^3$ y la cadena ab^3: $(ab)^3$ representa la cadena ababab, mientras que ab^3 es la cadena abbb.

* Ejercicio 6 *
¿Por qué el supraíndice 0 no es aplicable a caracteres pero sí a cadenas?

* Ejercicio 7 *
Demuestre que $(abc)^0 = (1234)^0$.

➔ Un caso particular: $\varepsilon^n = \varepsilon$ para cualquier valor entero de $n \geq 0$.

* Ejercicio 8 *
Demuestre que las cadenas $(ab^3)^3$ y $((ab)^3)^3$ son diferentes.

1.3 LENGUAJES NATURALES Y LENGUAJES FORMALES

Se denomina LENGUAJE NATURAL a todo lenguaje hablado y/o escrito que es utilizado por los seres humanos para comunicarse.

Ejemplo 19
El español, el inglés y el chino son los tres lenguajes naturales empleados por más personas en el mundo.

Los Lenguajes Naturales tienen cuatro características fundamentales:

1º EVOLUCIONAN con el paso del tiempo, incorporando nuevos términos y nuevas reglas gramaticales para mejorar y actualizar la comunicación;

> *Ejemplo 20*
> Las palabras pánfilo y zopenco no son utilizadas actualmente, aunque figuran en los diccionarios.
>
> * Ejercicio 9 *
> Escriban sinónimos de las palabras del Ejemplo 20, que se utilicen actualmente.

2º Sus REGLAS GRAMATICALES surgen después del desarrollo del lenguaje, para poder explicar su estructura, es decir: su sintaxis;

3º El SIGNIFICADO (o sea, la semántica) de cada palabra, de cada oración y de cada frase de un Lenguaje Natural es, en general, más importante que su composición sintáctica.

Ejemplo 21
"Donya Innes ah vaniado a su perro con javom pra labar la roppa." Esta oración tiene muchos errores, pero seguro que todos nosotros comprendemos su significado. En consecuencia, comprobamos que, en los Lenguajes Naturales, la semántica es más importante que la sintaxis.

Ejemplo 22
En general, los mensajes de texto tienen muchos errores sintácticos. A pesar de ello, las personas que se comunican a través de ellos comprenden su semántica, es decir: su significado.

4º Los Lenguajes Naturales son intrínsecamente AMBIGUOS, por lo siguiente: en todo proceso de comunicación existen un EMISOR y un RECEPTOR. El emisor piensa lo que quiere transmitir, pero no siempre lo pensado es lo que realmente comunica. El receptor percibe lo comunicado por el emisor y, de lo percibido, hay una comprensión que no necesariamente coincide con lo comunicado por el emisor, y menos aun, con lo que éste pensó comunicar. En este proceso interviene mucho el Lenguaje Natural.

Ejemplo 23
"Hay una llama en la cima de la montaña". La palabra llama se puede referir a un animal o a la existencia de fuego. Su interpretación dependerá, fundamentalmente, de la forma en la que la persona lo diga (tono de la voz) y de sus gestos.

Por otro lado, un **LENGUAJE FORMAL** es un conjunto de cadenas formadas con caracteres de un alfabeto dado, y tiene dos características fundamentales:

1) las cadenas que constituyen un Lenguaje Formal no tienen una semántica asociada, solo tienen una sintaxis dada por la ubicación de los caracteres dentro de cada cadena;
2) un Lenguaje Formal nunca es ambiguo.

Ejemplo 24
El Lenguaje Formal {Argentina, Holanda, Brasil} no está formado por los nombres de tres países (semántica). Solo consiste de tres cadenas, cada una de las cuales tiene los caracteres que están escritos y ubicados tal como se muestra (sintaxis).

* Ejercicio 10 *
Escriba el alfabeto mínimo a partir del cual se construye el Lenguaje Formal del ejemplo anterior.

Contrariamente a lo que ocurre con los Lenguajes Naturales, los LENGUAJES FORMALES están definidos por reglas gramaticales PREESTABLECIDAS y se deben ajustar rigurosamente a ellas. En consecuencia, un Lenguaje Formal *nunca puede evolucionar*.

Ejemplo 25
Sea $\Sigma = \{a\}$, un alfabeto con un solo carácter. Los que siguen son algunos Lenguajes Formales que se pueden construir sobre este alfabeto: $L_1 = \{a\}$; $L_2 = \{aa, aaa\}$; $L_3 = \{\varepsilon, a, a^{18}\}$.

Ninguno de estos lenguajes puede "evolucionar", es decir: no se puede modificar mediante el agregado de nuevas cadenas o quitando cadenas ya existentes en el lenguaje. Si así se hiciera, obtendríamos otros lenguajes.

➔ Observe cómo, en el Ejemplo 25, la letra a es utilizada en dos contextos diferentes, representando dos entidades distintas. Por un lado, a es un carácter de un alfabeto, mientras que, por otro lado, a es una cadena de un Lenguaje Formal. En consecuencia, el *contexto* en el que se encuentra un carácter, debe determinar, sin ambigüedades, el significado único de ese símbolo, ya sea como miembro de un alfabeto o bien como componente de un Lenguaje Formal.

* Ejercicio 11*
Escriba un Lenguaje Formal con cuatro cadenas de longitud cinco sobre el alfabeto {c, p}.

1.3.1 PALABRA

Una *cadena* es vacía o bien está compuesta por una sucesión de uno o más caracteres que pertenecen a un alfabeto dado, como hemos visto ya en muchos ejemplos.

Además, una *cadena* puede ser un miembro de un Lenguaje Formal, pero también puede no serlo. Por ello, incorporamos una nueva definición: "si una *cadena* pertenece a un determinado Lenguaje Formal, decimos que la cadena es una PALABRA de ese lenguaje". La pertenencia de una cadena a un Lenguaje Formal le da la propiedad de ser PALABRA de ese lenguaje en particular.

Ejemplo 26
Refiriéndonos al Ejemplo 25, observamos que el lenguaje L_1 tiene una sola palabra, a, mientras que el lenguaje L_3 tiene tres palabras: ε, a, a^{18}. La cadena aa no es una palabra de ninguno de estos dos lenguajes, pero sí lo es del lenguaje L_2.

Ejemplo 27
Sea el siguiente Lenguaje Formal descripto por EXTENSIÓN: L = {101, 1001, 10001, 100001}. Utilizando el operador "supraíndice", este lenguaje puede ser descripto por COMPRENSIÓN, en forma más compacta, así: $L = \{10^n1 / 1 \leq n \leq 4\}$. Entonces, la cadena 1001 ("uno-cero-cero-uno") es una palabra del lenguaje L, mientras que 1100 ("uno-uno-cero-cero") es una cadena construida con caracteres del mismo alfabeto pero no es una palabra de L.

Nota 6
Un Lenguaje Formal puede ser descripto por extensión, por comprensión, mediante una frase en un Lenguaje Natural (castellano, en nuestro caso) o mediante otras formas especiales que veremos más adelante.

Ejemplo 28
Sea el lenguaje L del Ejemplo anterior. Este Lenguaje Formal puede ser descripto mediante una frase en castellano (Lenguaje Natural), de la siguiente manera: "L es un lenguaje de cuatro palabras, cada una de las cuales comienza con el carácter 1, termina con otro carácter 1, y en medio tiene una secuencia de uno a cuatro 0s".

Nota 7
Si al describir un Lenguaje Formal no se indica explícitamente sobre qué alfabeto está construido, se considera que el alfabeto está formado por los caracteres que forman las palabras del lenguaje. Por caso: en el Ejemplo 27, el lenguaje L está construido sobre el alfabeto {0, 1}.

➔ Si analizamos las tres formas de describir un Lenguaje Formal (por comprensión, por extensión o mediante una frase en Lenguaje Natural), seguramente consideraremos que son más simples las dos que figuran en el Ejemplo 27. Sin embargo, hay muchos casos en los que la dificultad para describir un Lenguaje Formal con "simbología matemática" es evidente. En estos casos, debemos utilizar una frase en Lenguaje Natural, *evitando ambigüedades*, u otras herramientas que veremos más adelante.

Ejemplo 29
El lenguaje $L = \{a^{2i}b^i / 0 \leq i \leq 2\}$ está formado por tres palabras: si $i = 0$, $a^{2*0}b^0 = a^0b^0$ (ausencia de aes y ausencia de bes) representa la palabra vacía (ε); si $i = 1$, la palabra es $a^{2*1}b^1 = aab$; y si $i = 2$, la palabra es $a^{2*2}b^2 = aaaabb$. En este caso, obviamente es más sencilla la descripción por extensión que por comprensión.

* Ejercicio 12 *
Describa por comprensión al siguiente lenguaje:
L = {ε, b, bb, bbb, bbbb, bbbbb, bbbbbb, bbbbbbb, bbbbbbbb}.

* Ejercicio 13 *
Describa mediante una frase al lenguaje del ejercicio anterior.

Ejemplo 30
Sea el Lenguaje Formal sobre el alfabeto {a, b, c} formado por todas las palabras de longitud 30 que comienzan con a, terminan con b y, en medio, tienen exactamente tres ces. Una palabra de este lenguaje es: $a^{20}cbbcacb^4$.

* Ejercicio 14 *
Intente describir por comprensión o por extensión el Lenguaje Formal del Ejemplo 30.

1.3.2 PROPIEDADES DE LAS PALABRAS

Dado que una PALABRA es una cadena que pertenece a un determinado Lenguaje Formal, todos los conceptos sobre *cadena* explicados anteriormente se aplican también a las palabras. Por lo tanto, hablaremos de: longitud de una palabra, palabra vacía, concatenación de dos o más palabras, potenciación de una palabra, con el mismo significado ya visto.

Ejemplo 31
Sea el lenguaje $L = \{(abc)^n / 0 \leq n \leq 3\} = \{\varepsilon, abc, abcabc, abcabcabc\}$. Entonces:
– La longitud de la palabra abc es $|abc| = 3$.
– La palabra vacía ε es un miembro de este lenguaje.
– La concatenación de las palabras abc y abcabc produce otra palabra de este lenguaje: abcabcabc. En cambio, la concatenación de la palabra abcabc consigo misma produce la cadena abcabcabcabc, que no es una palabra de este lenguaje.
– La potencia $(abc)^2$ es una palabra del lenguaje.

➔ La concatenación de dos palabras produce una *cadena* que no siempre es una *palabra* del lenguaje, como se observa en el ejemplo anterior y en el ejemplo que sigue.

Ejemplo 32
Sea el lenguaje $L = \{a^{2n+1} / 0 \leq n \leq 200\}$. Este Lenguaje Formal está formado por 201 palabras, cada una de las cuales tiene un número impar de letras a. Si $n = 0$, la palabra es a; ésta es la palabra de menor longitud de L. Si $n = 200$, la palabra es a^{401}; ésta es la palabra de mayor longitud de L.

Si se concatenan dos palabras cualesquiera de L, el resultado será una cadena con una cantidad par de letras a, ya que la suma de dos números impares produce un número par. En consecuencia, la concatenación de dos palabras cualesquiera de este lenguaje produce una cadena que nunca es una palabra de L.

* Ejercicio 15 *
Describa mediante una frase al lenguaje del Ejemplo 32.

* Ejercicio 16 *
Sea el lenguaje $L = \{a^{2n} / 0 \leq n \leq 200\}$. Escriba, por comprensión, el Lenguaje Formal que se obtiene realizando todas las concatenaciones de dos palabras cualesquiera del lenguaje dado.

1.3.3 CARDINALIDAD DE UN LENGUAJE FORMAL

La cardinalidad de un Lenguaje Formal es la cantidad de palabras que lo componen.

Ejemplo 33
$L = \{a, ab, aab\}$ es un lenguaje de cardinalidad 3 sobre el alfabeto {a, b}.

Ejemplo 34
El lenguaje $L = \{\varepsilon\}$ es un lenguaje muy especial, pues está formado solo por la palabra vacía. Su cardinalidad es 1, ya que contiene una palabra.

1.3.4 SUBLENGUAJES

Dado que un Lenguaje Formal es un conjunto, un SUBLENGUAJE es un subconjunto de un Lenguaje Formal dado.

Ejemplo 35
Sea $L_1 = \{a, ab, aab\}$. Entonces, $L_2 = \{ab, aab\}$ es un sublenguaje de L_1, mientras que $L_3 = \{\ \}$ es el sublenguaje vacío de L_1.

➔ El sublenguaje vacío, habitualmente representado con el símbolo ø, es un sublenguaje de cualquier Lenguaje Formal.

➔ No se debe confundir el sublenguaje vacío, que tiene cardinalidad 0, con el lenguaje que solo contiene la palabra vacía, que tiene cardinalidad 1.

1.4 LENGUAJES FORMALES INFINITOS

Todos los lenguajes ejemplificados hasta el momento han sido LENGUAJES FORMALES FINITOS, es decir: lenguajes con un número finito de palabras. Sin embargo, los Lenguajes Formales también pueden ser INFINITOS, lo que significa que estos lenguajes pueden tener una cantidad infinita de palabras, aunque cada una de ellas debe ser de longitud finita; no existen las palabras de longitud infinita.

Nota 8
Las propiedades de las palabras que han sido ejemplificadas anteriormente se aplican también a los lenguajes infinitos.

Ejemplo 36
$L = \{a^n / n \geq 1\}$ es un Lenguaje Formal infinito ya que no existe un límite superior para el supraíndice n. Cada palabra de este lenguaje está formada por una secuencia de una o más $a$es. Por ello, la concatenación de dos palabras cualesquiera de este lenguaje producirá siempre otra palabra del lenguaje L. Por esta propiedad, se dice que este lenguaje L tiene una particularidad especial: es cerrado bajo la concatenación.

Ejemplo 37
El lenguaje $L = \{ab^n / n \geq 0\}$ es otro Lenguaje Formal infinito. Este lenguaje está formado por la palabra a y todas aquellas palabras que comienzan con una a, seguida de una secuencia de una o más $b$es. Este lenguaje no es cerrado bajo la concatenación ya que, por caso: si consideramos sus dos palabras de menor longitud – a y ab – y las concatenamos, obtenemos la cadena aab, que no es una palabra de este lenguaje.

* Ejercicio 17 *
Sea el lenguaje $L = \{ab^na / n \geq 1\}$.
a) Escriba las tres palabras de menor longitud.
b) Describa este lenguaje mediante una frase en castellano.

1.4.1 LENGUAJE UNIVERSAL SOBRE UN ALFABETO

Dado un alfabeto Σ, el LENGUAJE UNIVERSAL sobre este alfabeto es un Lenguaje Formal infinito que contiene todas las palabras que se pueden formar con los caracteres del alfabeto Σ, más la palabra vacía. Se lo representa con la notación Σ^*, que se lee "sigma clausura" o "sigma estrella".

➔ Importante: acabamos de introducir un nuevo operador que representamos con el supraíndice *. Este es un operador unario (opera sobre un solo operando), como lo es la potenciación en matemáticas. Se lo denomina "clausura de Kleene" (se pronuncia *klaini*) o "estrella de Kleene", y será muy utilizado en los próximos capítulos.

Una propiedad fundamental del Lenguaje Universal es que es cerrado bajo concatenación.

Ejemplo 38
Si $\Sigma = \{a, b\}$, entonces el Lenguaje Universal para este alfabeto está formado por la palabra vacía, todas las palabras que solo tienen aes, todas las palabras que solo tienen bes, y todas las palabras formadas por aes y bes concatenadas en cualquier orden:

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, ..., aabaabbbab, \ldots\},$$

Afirmación: La concatenación de palabras de Σ^* siempre produce una palabra de este Lenguaje Universal sobre el alfabeto {a, b}.

* Ejercicio 18 *
¿Puede encontrar un caso en que la afirmación anterior sea falsa? Justifique su respuesta.

➔ Cualquier lenguaje L sobre el alfabeto Σ es un sublenguaje de Σ^*. Por lo tanto, existen infinitos Lenguajes Formales sobre un alfabeto dado.

1.4.2 LENGUAJES FORMALES INFINITOS MÁS COMPLEJOS

Todos los ejemplos vistos hasta ahora se refieren al tipo de Lenguaje Formal llamado LENGUAJE REGULAR. Pero existen otros tipos de Lenguajes Formales infinitos que ejemplificamos a continuación.

Ejemplo 39
Los que siguen son ejemplos de tres diferentes tipos de Lenguajes Formales infinitos, ninguno de los cuales es un Lenguaje Regular:

$L_1 = \{a^n b^n / n \geq 1\}$. Las tres palabras de menor longitud de este lenguaje son: ab, aabb y aaabbb.

$L_2 = \{a^n b^n c^n / n \geq 1\}$. Las tres palabras de menor longitud de este lenguaje son: abc, aabbcc y aaabbbccc.

$L_3 = \{a^n / n$ es una potencia positiva de $2\}$. Las tres palabras de menor longitud de este lenguaje son: aa, aaaa y aaaaaaaa.

En el próximo capítulo veremos porqué hay diferentes tipos de Lenguajes Formales y porqué los lenguajes del Ejemplo 39 no son LENGUAJES REGULARES.

Ciertos Lenguajes Formales están relacionados con la sintaxis de los Lenguajes de Programación.
Un Lenguaje de Programación (LP) tiene palabras reservadas, nombres creados por el programador, constantes enteras y reales, caracteres de puntuación, operadores aritméticos, operadores lógicos, declaraciones, expresiones, sentencias, etc.
Todos estos elementos constituyen diferentes Lenguajes Formales, algunos infinitos y otros finitos.

* Ejercicio 19 *
Para un LP que conozca, escriba dos ejemplos de cada una de las entidades (palabras reservadas, expresiones, etc.) mencionadas en el recuadro anterior. Indique si corresponden, según su criterio, a un Lenguaje Formal finito o a un Lenguaje Formal infinito. Explique su decisión.

1.5 IMPLEMENTACIÓN EN C

Investigue y construya, en LENGUAJE C, la función que realiza cada operación solicitada:

* Ejercicio 20 *
(a) Calcula la longitud de una cadena;
(b) Determina si una cadena dada es vacía.
(c) Concatena dos cadenas.

* Ejercicio 21 *
Construya un programa de testeo para cada función del ejercicio anterior.

2 GRAMÁTICAS FORMALES Y JERARQUÍA DE CHOMSKY

Un LENGUAJE FORMAL, sea finito o infinito, es un conjunto de palabras con una connotación sintáctica, sin semántica. Existen ciertas estructuras que tienen la habilidad de GENERAR todas las palabras que forman un Lenguaje Formal. Estas estructuras se denominan GRAMÁTICAS FORMALES.

2.1 GRAMÁTICA FORMAL

Una Gramática Formal es, básicamente, un conjunto de PRODUCCIONES, es decir: *reglas de reescritura* que se aplican para obtener las palabras del Lenguaje Formal que la Gramática Formal en cuestión genera. Una Gramática Formal genera un determinado Lenguaje Formal, único.

Ejemplo 1
Con una simbología simple, que será aclarada posteriormente, podemos representar las producciones de una Gramática Formal que genera el lenguaje L = {ab, ac}. Estas producciones se pueden escribir así: S –> ab y S –> ac.

Para construir las PRODUCCIONES de una Gramática Formal se requieren tres tipos de símbolos:
- los símbolos "productores", como es S en el Ejemplo 1;
- los símbolos que forman las palabras del lenguaje generado, como son los caracteres a, b y c en el Ejemplo 1; y
- ciertos símbolos especiales que llamaremos *metasímbolos*, como –> en el Ejemplo 1.

Ejemplo 2
Sea el lenguaje L = {a}, formado por una sola palabra. Este lenguaje es generado por una Gramática Formal con una única producción: S –> a (se lee "S produce a" o "S es reemplazada por a"). La "flecha" será llamada OPERADOR DE PRODUCCIÓN.

Si bien en estos dos primeros ejemplos se observa que cada producción genera una palabra del lenguaje, esto no siempre es así.

Ejemplo 3
El Lenguaje Formal L = {aa, ab} puede ser generado por una Gramática Formal con dos producciones: S –> aa y S –> ab. Pero este Lenguaje Formal también puede ser generado por la Gramática Formal con producciones: S –> aT, T –> a y T –> b.

Si utilizamos el segundo conjunto de producciones, ¿cómo obtenemos las dos palabras del lenguaje? Comenzamos con la producción S –> aT ("S produce a seguido de T" o "S produce a concatenado con T") para obtener el carácter a con el que comienzan ambas palabras de este Lenguaje Formal. Luego, si aplicamos la producción T –> a obtenemos la palabra aa y si aplicamos T –> b obtenemos la palabra ab.

➔ Toda producción está formada por tres partes: el lado izquierdo, el lado derecho, y el operador de producción, que significa que el lado izquierdo de la producción "produce" –o "es reemplazado por" o "equivale a"– el lado derecho de la misma.

Ejemplo 4
En la producción S −> aT, el *lado izquierdo* está constituido por el símbolo S, mientras que su *lado derecho* está formado por la concatenación del carácter a con el símbolo T. El operador de producción indica que el símbolo S es reemplazado por la secuencia aT.

➔ Una Gramática Formal puede tener, entre sus producciones, una muy particular que se denomina PRODUCCIÓN-ÉPSILON y que se escribe, por ejemplo: S −> ε. Esta notación significa que S es reemplazada por "nada" o que genera la palabra vacía. Veamos los dos casos.

Ejemplo 5
El Lenguaje Formal L = {aa, ε}, que contiene a la palabra vacía, puede ser generado por una Gramática Formal con dos producciones: S −> aa y S −> ε. En este caso, la producción-épsilon genera la palabra vacía.

Ejemplo 6
Veamos, ahora, el segundo caso. El Lenguaje Formal L = {aa, aab} puede ser generado por una Gramática Formal con las siguientes producciones: S −> aaT, T −> ε y T −> b.

* Ejercicio 1 *
a) Describa el orden en que se aplican estas producciones para generar las palabras del lenguaje del Ejemplo 6.
b) ¿La producción T −> ε genera la palabra vacía del lenguaje?

2.1.1 DEFINICIÓN FORMAL DE UNA GRAMÁTICA FORMAL

Toda Gramática Formal es una 4-upla (V_N, V_T, P, S), donde:

– V_N es el vocabulario de noterminales; es un conjunto finito de "productores".

– V_T es el vocabulario de terminales, caracteres del alfabeto sobre el cual se construyen las palabras del Lenguaje Formal que es generado por la gramática descripta; también es un conjunto finito.

– P es el conjunto finito de producciones, y

– S $\in$ V_N es un noterminal especial llamado axioma. Es el noterminal a partir del cual siempre deben comenzar a aplicarse las producciones que generan las palabras de un determinado Lenguaje Formal.

➔ Cuando decimos "una Gramática Formal genera un Lenguaje Formal" significa que esa gramática es capaz de generar todas las palabras del Lenguaje Formal, pero que, a su vez, no genera cadenas que están fuera del lenguaje.

Ejemplo 7
Retomamos el Ejemplo 3, que describe las producciones de una gramática que genera el lenguaje {aa, ab}. La definición formal de esta gramática es la 4-upla:

G = ({S, T}, {a, b}, {S −> aT, T −> a, T −> b}, S).

Dado que el noterminal S es el axioma, la generación de cualquier palabra del lenguaje mencionado comienza con una producción que tenga al noterminal S en su lado izquierdo; en este caso, hay una única producción con esta característica: S −> aT. Esta única producción indica que el símbolo inicial S es reemplazado, obligatoriamente, por la secuencia aT, por lo que toda palabra generada

por esta gramática debe comenzar con el carácter a. Todavía no se ha obtenido una palabra del lenguaje, porque T es un noterminal y sabemos que una palabra debe estar formada solo por caracteres o debe ser la palabra vacía. En consecuencia, debe haber una o más producciones que tengan al noterminal T en su lado izquierdo, y se debe reemplazar a T por su lado derecho. En este caso, el noterminal T tiene dos producciones que representan dos opciones: la producción T −> a significa que el noterminal T es reemplazado por el carácter a, mientras que la producción T −> b representa que el noterminal T es reemplazado por el carácter b. Estas dos producciones para el noterminal T generan dos procesos diferentes:

(1) la aplicación de las producciones S −> aT y T −> a genera la palabra aa;
(2) la aplicación de las producciones S −> aT y T −> b genera la palabra ab.

Ejemplo 8
Realizamos una pequeña variante sobre el Ejemplo 7. Supongamos una Gramática Formal con la siguiente definición:

G = ({S, T}, {a}, {S −> aT, T −> a, T −> ε}, S).

* Ejercicio 2 *
¿Qué Lenguaje Formal genera la gramática del Ejemplo 8? Justifique.

Nota 1
Al describir Gramáticas Formales para aplicaciones teóricas, como las que estamos viendo en este capítulo, es muy común utilizar las siguientes convenciones:
– se denomina S al axioma,
– todo noterminal es representado mediante una letra mayúscula,
– el vocabulario de terminales está formado por letras minúsculas y dígitos.

Nota 2
Agregamos otro conjunto que, si bien no figura en la definición formal de una Gramática Formal, será de gran utilidad en aplicaciones que veremos más adelante. Nos referimos al conjunto de METASÍMBOLOS, que no son terminales ni son noterminales, pero que son símbolos que ayudan a representar las producciones de una Gramática Formal. Uno de ellos ya lo hemos visto: −>, el operador de producción.

Ejemplo 9
Sea la definición formal del Ejemplo 7: G = ({S, T}, {a, b}, {S −> aT, T −> a, T −> b}, S). Es habitual describir esta Gramática Formal mediante la escritura de sus producciones únicamente. De ser así, por convención se entiende que el noterminal que aparece en el lado izquierdo de la primera producción es el axioma y, normalmente, se lo llama S. Por otro lado, si para un noterminal dado existe más de una producción, se utiliza el *metasímbolo* "barra vertical" para indicar "ó", es decir, describir dos o más producciones con el mismo noterminal en su lado izquierdo. Entonces, las producciones de esta gramática se escriben, generalmente, de la siguiente manera:

S −> aT
T −> a | b

Esta Gramática Formal está formada por tres producciones. Los metasímbolos −> y | colaboran en la escritura de estas producciones en forma más compacta.

* Ejercicio 3 *
Sea la Gramática Formal con producciones:

S –> aT | bQ
T –> a | b
Q –> a | ε

a) ¿La gramática con estas seis producciones genera la cadena bab? Justifique.
b) ¿Cuál es el Lenguaje Formal generado por esta gramática? Justifique.

2.2 LA JERARQUÍA DE CHOMSKY

En 1956 y 1959, el lingüista norteamericano Noam Chomsky publicó dos trabajos sobre los Lenguajes Naturales que, aplicados al área de los Lenguajes Formales, produjeron lo que se conoce como "Jerarquía de Chomsky".

Esta Jerarquía de Chomsky establece una clasificación de cuatro tipos de Gramática Formales que, a su vez, generan cuatro tipos diferentes de Lenguajes Formales.

Las Gramáticas Formales se clasifican según las restricciones que se imponen a sus producciones, y la Jerarquía de Chomsky establece estos cuatro niveles:

- Gramáticas Regulares o Gramáticas Tipo 3
- Gramáticas Independientes del Contexto o Gramáticas Tipo 2
- Gramáticas Sensibles al Contexto o Gramáticas Tipo 1
- Gramáticas Irrestrictas o Gramáticas Tipo 0

Por otro lado, estos cuatro tipos de gramáticas generan los siguientes tipos de Lenguajes Formales:

Gramática Formal	**Lenguaje Formal**
Gramática Regular	Lenguaje Regular
Gramática Independiente del Contexto	Lenguaje Independiente del Contexto
Gramática Sensible al Contexto	Lenguaje Sensible al Contexto
Gramática Irrestricta	Lenguaje Irrestricto

A continuación, analizamos las diferencias que existen entre los cuatro tipos de Gramáticas Formales.

2.2.1 GRAMÁTICA REGULAR (GR)

Sus producciones tienen las siguientes restricciones:
- el lado izquierdo debe tener un solo noterminal,
- el lado derecho debe estar formado por:
  a) un solo terminal, o
  b) un terminal seguido por un noterminal,
  c) o "épsilon".

Ejemplo 10
Sea la gramática G = ({S, X}, {a, b}, {S −> aX, X −> b}, S).
Las producciones de esta gramática cumplen con las restricciones mencionadas arriba. Analícelas. Por lo tanto, esta Gramática Formal es una GR.

También es válida una GR en la que se invierte el orden en el lado derecho de aquellas producciones que tienen dos símbolos. Por lo tanto, una segunda definición para las GRs sería:

Una Gramática Formal es una GR si sus producciones tienen las siguientes restricciones:
- el lado izquierdo debe tener un solo noterminal,
- el lado derecho debe estar formado por:
  a) un solo terminal, o
  b) un noterminal seguido de un terminal, o
  c) "épsilon".

Ejemplo 11
La Gramática Formal ({S, X}, {a, b}, {S −> Xa, X −> b}, S) es una GR porque cumple con las restricciones de esta segunda definición.

En general: sean v y v' noterminales y sea t un terminal. Entonces, las producciones de una GR pueden tener estos formatos:

1) v −> t, 2) v −> tv' y 3) v −> ε o 1) v −> t, 2) v −> v't y 3) v −> ε

Sin embargo, debemos tener cuidado con "la mezcla" de ambas definiciones porque la Gramática Formal resultante no será una GR.

Ejemplo 12
Sea G = ({S, X}, {a, b}, {S −> Xa, S −> bX, X −> b}, S). Esta Gramática Formal no es una GR porque el noterminal S produce "noterminal terminal" y también produce "terminal noterminal".

* Ejercicio 4 *
a) Escriba las producciones de una GR que genere el Lenguaje Regular $L = \{a^n b / 1 \leq n \leq 3\}$.
b) Escriba la definición formal de la GR desarrollada en el punto anterior.

* Ejercicio 5 *
a) Escriba las producciones de una GR que genere el Lenguaje Regular $L = \{a^n b^n / 0 \leq n \leq 2\}$.
b) Escriba la definición formal de la GR desarrollada en el punto anterior.

2.2.1.1 GRAMÁTICA REGULAR QUE GENERA UN LENGUAJE REGULAR INFINITO

Las GRs que hemos visto anteriormente generan Lenguajes Regulares (LRs) finitos. Pero las GRs, aun teniendo un número finito de producciones, pueden generar LRs infinitos. Para ello existen las llamadas PRODUCCIONES RECURSIVAS: producciones en las que el noterminal que figura en el lado izquierdo también figura en el lado derecho.

Ejemplo 13
La producción T −> aT es una producción recursiva.

Ejemplo 14
Sea el Lenguaje Formal infinito $L = \{a^n b / n \geq 1\}$. Construyamos las producciones de una GR que genere este lenguaje:

1 S –> aS
2 S –> aT
3 T –> b

Habitualmente, estas producciones se escriben en forma más compacta de la siguiente manera:

S –> aS | aT
T –> b

Análisis: La producción (1) genera tantas aes como se necesiten por ser una producción recursiva, aunque no es obligatorio que esta producción siempre sea elegida ya que el axioma S tiene dos producciones. La producción (2) termina la recursividad y produce una última a, o bien produce la única a si la producción (1) no es utilizada. La producción (3) finaliza la generación de una palabra, reemplazando el noterminal T que está en el lado derecho de la producción (2) por el carácter b con el que debe terminar toda palabra del LR que genera esta gramática.

* Ejercicio 6 *
Indique cuál es la mínima palabra del LR del Ejemplo 14 y muestre cómo la genera.

* Ejercicio 7 *
Escriba la sucesión de producciones que aplicaría para generar la palabra aaab del LR definido en el Ejemplo 14.

2.2.1.2 GRAMÁTICA QUASI–REGULAR (GQR)

Estas gramáticas están muy vinculadas a la sintaxis de los Lenguajes de Programación, como veremos en el próximo capítulo. Son gramáticas similares a una GR, pero donde un conjunto de terminales es reemplazado por un noterminal en una o varias producciones. Una GQR abrevia la escritura de una GR y siempre puede ser re-escrita como una GR.

Ejemplo 15
Sea la gramática con las siguientes producciones:

S –> N | NS
N –> a | b | c

Esta gramática es una GQR, porque las dos producciones del noterminal S no corresponden a la definición de una GR pero sí responden a la definición dada en el párrafo anterior. El noterminal N genera los terminales a, b y c; por lo tanto, N representa al conjunto {a, b, c}.

Ejemplo 16
Re-escribamos las producciones de la GQR del ejemplo anterior como producciones de una GR. Para ello, debemos reemplazar al noterminal N por cada uno de los terminales que representa. Resultan, entonces, las siguientes producciones:

S –> a | b | c | aS | bS | cS

Como se observa, la GQR tiene cinco producciones, mientras que la GR equivalente tiene seis producciones.

➔ Dos Gramáticas Formales son EQUIVALENTES si generan el MISMO Lenguaje Formal.

Ejemplo 17
Las gramáticas de los ejemplos 15 y 16 son equivalentes.

* Ejercicio 8 *
a) Escriba una GR que genere cualquier secuencia de uno o más dígitos decimales.
b) Escriba una GQR que genere cualquier secuencia de uno o más dígitos decimales.
c) Compare la cantidad de producciones de ambas gramáticas.

* Ejercicio 9 *
Escriba las producciones de una GQR que genere un LR infinito cuyas palabras son secuencias de tres o más dígitos octales (en base 8).

* Ejercicio 10 *
Transforme las producciones de la GQR obtenida en el Ejercicio 9 en producciones de una GR equivalente.

2.2.2 GRAMÁTICA INDEPENDIENTE DEL CONTEXTO (GIC)

A diferencia de las GRs, las GICs no tienen restricciones con respecto a la forma del lado derecho de sus producciones, aunque sí se requiere que el lado izquierdo de cada producción siga siendo un único noterminal.

> El origen de la formalización de las GICs se encuentra en Chomsky [1956]: "Three models for the description of language". [*Hopcroft y Ullman, "Introduction to Automata Theory, Languages and Computation", 1979, Addison-Wesley*]

Ejemplo 18
Supongamos que a una GR le agregamos la producción S –> ba. Esta nueva producción posee dos terminales en su lado derecho y, como ya hemos visto, este no es un formato válido para una GR. La nueva Gramática Formal es una GIC.

En general: sean v y v' noterminales y sea t un terminal. Entonces las producciones de una GIC corresponden a este formato general:

v –> (v' + t)*, donde v y v' pueden representar el mismo noterminal

La expresión (v' + t)* (recuerden el operador "estrella" del capítulo anterior) representa ε y cualquier secuencia de noterminales y/o terminales.

Ejemplo 19
Teniendo en cuenta la convención mencionada antes (letra mayúscula para noterminales, y letras minúsculas o dígitos para terminales), he aquí una serie de producciones correctas para las GICs:

A −> ε	producción-épsilon
B −> a	lado derecho con un terminal
C −> ab2c5	lado derecho con cinco terminales
D −> ZXZ	lado derecho con tres noterminales
E −> aEZbRgh	lado derecho con cuatro terminales y tres noterminales

* Ejercicio 11 *
a) ¿Una GR es siempre una GIC? Justifique.
b) ¿Una GIC es siempre una GR? Justifique.

2.2.2.1 GIC QUE GENERA UN LENGUAJE INFINITO

Las GICs pueden generar Lenguajes Independientes del Contexto (LICs) infinitos utilizando PRODUCCIONES RECURSIVAS.

Ejemplo 20
La GIC con producciones:

S −> aSb | a

genera un LIC infinito; la primera producción es recursiva y la segunda producción debe utilizarse para terminar la recursividad.

* Ejercicio 12 *
a) ¿Cuál es la mínima palabra generada por la GIC del ejemplo anterior? Justifique.
b) ¿Cuál es la palabra que le sigue en longitud? Justifique.

* Ejercicio 13 *
Describa, por comprensión, el LIC generado por la GIC del Ejemplo 20.

* Ejercicio 14 *
Sea el lenguaje $L_9 = \{a^n b^{n+1} / n \geq 0\}$. Escriba las producciones de una GIC que genere L_9.

* Ejercicio 15 *
Sea el lenguaje $L_{10} = \{a^{2n} b^{n+1} a^r / n \geq 1, r \geq 0\}$. Escriba la definición formal de una GIC que genere el lenguaje L_{10}.

➔ Por las restricciones establecidas sobre los dos tipos de Gramáticas Formales vistas hasta ahora, la GR y la GIC, es fácil notar que toda GR también es una GIC; la inversa no es cierta.

* Ejercicio 16 *
Demuestre que una GQR es un caso particular de una GIC.

La frase “independiente del contexto” refleja que, como el lado izquierdo de cada producción solo puede contener un único noterminal, toda producción de una GIC puede aplicarse sin importar el contexto donde se encuentre dicho noterminal.

Estas GICs son de gran utilidad en la representación de la sintaxis
de los Lenguajes de Programación; ampliaremos más adelante.

2.2.3 GRAMÁTICA IRRESTRICTA

Estas son las Gramáticas Formales más amplias. Sus producciones tienen la forma general: $\alpha \rightarrow \beta$, donde tanto α como β pueden ser secuencias de noterminales y/o terminales, con $\alpha \neq \varepsilon$.

Ejemplo 21 [de Hopcroft & Ullman, 1979]
La que sigue es una Gramática Irrestricta que genera el lenguaje

$$\{a^i / i \text{ es una potencia positiva de } 2\}:$$

$$G = (\{S, A, B, C, D, E\}, \{a\}, \{S \rightarrow ACaB, Ca \rightarrow aaC, CB \rightarrow DB, CB \rightarrow E,$$
$$aD \rightarrow Da, AD \rightarrow AC, aE \rightarrow Ea, AE \rightarrow \varepsilon\}, S)$$

* Ejercicio 17 *
Compruebe que la gramática escrita en el Ejemplo 21 puede generar la palabra aaaa. Indique, paso a paso, cada producción que aplica y explique porqué utiliza la producción elegida.

2.2.4 GRAMÁTICA SENSIBLE AL CONTEXTO (GSC)

Una GSC es una Gramática Irrestricta, con la siguiente restricción en las longitudes: $|\alpha| \leq |\beta|$.

Ejemplo 22
La Gramática Formal del Ejemplo 21 no es una GSC porque tiene dos producciones que violan la restricción impuesta sobre las longitudes: $CB \rightarrow E$ y $AE \rightarrow \varepsilon$.

Cada tipo de Gramática Formal genera un tipo de Lenguaje Formal cuyo nombre deriva del nombre de la correspondiente gramática, como ya hemos visto anteriormente. En este libro nos interesarán, fundamentalmente: las GICs, las GQRs, los Lenguajes Regulares (LRs) y los Lenguajes Independientes del Contexto (LICs).

2.3 EL PROCESO DE DERIVACIÓN

La derivación es el proceso que permite obtener cada una de las palabras de un Lenguaje Formal a partir del axioma de una Gramática Formal que lo genera, y aplicando, sucesivamente, las producciones convenientes de esa gramática. El mismo proceso también permite descubrir si una cadena dada no es una palabra del lenguaje.

Existen diferentes formas de representar una derivación:

1) en forma horizontal, utilizando el símbolo **=>** para relacionar un paso de la derivación con el paso siguiente;

2) en forma vertical, reemplazando, en cada paso, un noterminal por su lado derecho para producir una nueva línea de esta derivación;

3) en forma de árbol, donde el axioma de la gramática es la raíz del árbol.

En esta sección solo utilizaremos la derivación vertical, aplicándola a GQRs y a GICs.

Ampliamos, entonces, lo indicado para la DERIVACIÓN VERTICAL. Esta derivación producirá una tabla con una sola columna y, eventualmente, una segunda columna con comentarios. En la primera línea estará siempre el axioma de la gramática utilizada. Las líneas sucesivas se obtendrán reemplazando un solo noterminal, de lo logrado hasta el momento, por el lado derecho de la producción más conveniente.

Ejemplo 23
Sea la GIC con producciones S −> aSb y S −> ab. Esta GIC genera el lenguaje $\{a^n b^n / n \geq 1\}$. Una de las palabras de este LIC es aaabbb. Verifiquemos esta situación mediante una derivación vertical:

Derivación Vertical	**Comentario**
S	aplicamos la primera producción y obtenemos:
aSb	aplicamos, nuevamente, la primera producción y obtenemos:
aaSbb	aplicamos la segunda producción y obtenemos:
aaabbb	

Ejemplo 24
En cambio, la cadena aaabb no es una palabra del LIC generado en el Ejemplo 23 y, por lo tanto, no la podremos derivar. Comprobemos esta situación:

Derivación Vertical	**Comentario**
S	aplicamos la primera producción y obtenemos:
aSb	aplicamos, nuevamente, la primera producción y obtenemos:
aaSbb	no tenemos forma de obtener la tercera a sola, ¿por qué?
??	

Por lo tanto, no podemos derivar aaabb y, en consecuencia, ésta no es una palabra del LIC generado por la GIC dada.

* Ejercicio 18 *
(a) Dada la GIC del Ejemplo 23, determine, aplicando una derivación vertical, si la cadena aaabbbb es una palabra del LIC generado por esta GIC.
(b) Investigue y dibuje la misma derivación, pero en forma de árbol.

➔ Antes de proseguir con este tema, introducimos una definición que facilitará la descripción de lo que sigue. Denominaremos CADENA DE DERIVACIÓN a la cadena de terminales y noterminales que se forma en cada paso de la derivación vertical, antes de obtener la palabra (si existe).

Ejemplo 25
En la derivaciones de los ejemplo 23 y 24, las dos cadenas de derivación son: aSb y aaSbb.

En el Ejemplo 23, la GIC tiene un solo noterminal en el lado derecho de una producción. En el próximo capítulo veremos gramáticas que tienen más de un noterminal en el lado derecho de una o más producciones. Por ello, definiremos dos tipos de derivaciones verticales:

➔ DERIVACIÓN VERTICAL A IZQUIERDA: en cada paso de la derivación se reemplaza el noterminal que se encuentra primero (de izquierda a derecha) en la cadena de derivación.

➔ DERIVACIÓN VERTICAL A DERECHA: en cada paso de la derivación se reemplaza el noterminal que se encuentra primero (de derecha a izquierda) en la cadena de derivación.

Ejemplo 26
Sea la GIC con producciones:

1 S –> ST
2 S –> ab
3 T –> aaT
4 T –> b

Una derivación vertical a izquierda produciría la siguiente tabla para obtener la palabra abaabaab:

Derivación Vertical a Izquierda	**Comentario**
S	Se aplica la producción (1)
ST	Se aplica la producción (1)
STT	Se aplica la producción (2)
abTT	Se aplica la producción (3)
abaaTT	Se aplica la producción (4)
abaabT	Se aplica la producción (3)
abaabaaT	Se aplica la producción (4)
abaabaab	

Una derivación vertical a derecha produciría la siguiente tabla para obtener la misma palabra:

Derivación Vertical a Derecha	**Comentario**
S	Se aplica la producción (1)
ST	Se aplica la producción (3)
SaaT	Se aplica la producción (4)
Saab	Se aplica la producción (1)
STaab	Se aplica la producción (3)
SaaTaab	Se aplica la producción (4)
Saabaab	Se aplica la producción (2)
abaabaab	

➔ Si una palabra se puede obtener mediante una derivación vertical a izquierda, entonces también se puede obtener mediante una derivación vertical a derecha.

Si una cadena de derivación tiene más de un noterminal, la próxima cadena de derivación se obtiene reemplazando un ÚNICO noterminal en la cadena de derivación anterior, tal como se observa en ambas tablas del Ejemplo 26. Nunca se pueden reemplazar dos o más noterminales en un solo paso.

* Ejercicio 19 *
Sea el LIC infinito $L = \{a^n bc^n b / n \geq 1\}$. Escriba la Definición Formal de una GIC que genere este Lenguaje Formal.

* Ejercicio 20 *
Dada la GIC construida en el ejercicio anterior, utilice derivación vertical a izquierda para determinar si las siguientes cadenas son o no palabras del LIC generado:

a) aaabcccb
b) aabbccb
c) aaabcccbb
d) aaccb

2.4 INTRODUCCIÓN A LAS GQRs, LAS GICs Y LOS LENGUAJES DE PROGRAMACIÓN

Tanto las GQRs como las GICs son de gran utilidad en la representación de la sintaxis de los Lenguajes de Programación. Si bien este es un tema que desarrollaremos en el próximo capítulo, he aquí unos ejemplos para introducirlo:

Ejemplo 27
Supongamos que un Lenguaje de Programación tiene nombres de variables que deben comenzar con una letra minúscula entre a y d, que puede estar seguida por letras en este mismo intervalo y por dígitos entre 2 y 6. El lenguaje de estos nombres es un LR infinito que puede ser generado por una GQR con estas producciones:

S –> L | SL | SD
L –> a | b | c | d
D –> 2 | 3 | 4 | 5 | 6

* Ejercicio 21 *
Escriba una GR equivalente a la GQR del ejemplo anterior. ¿Cuál es más sencilla para ser leída? Justifique su respuesta.

* Ejercicio 22 *
Dada la GQR del Ejemplo 27, determine si las siguientes cadenas son nombres válidos:

ab23
2a3b

Verifíquelo aplicando: a) derivación a izquierda y b) derivación a derecha.

Ejemplo 28

Supongamos un Lenguaje de Programación en el que sus expresiones aritméticas están formadas por los números enteros 2 y 6, el operador de suma y siempre terminan con un "punto y coma". Algunas expresiones aritméticas de este Lenguaje de Programación son:

6;
2+2+6;

Vamos a construir una GIC que genere la totalidad de estas expresiones aritméticas. Para ello, debemos definir el vocabulario de noterminales, el vocabulario de terminales, el axioma y el conjunto de producciones. Supongamos que el vocabulario de noterminales está formado por: S (el axioma), E (de expresión) y T (de término). Los terminales son 2, 6, + y ; (punto y coma). Las producciones de la GIC que genera el lenguaje de estas expresiones aritméticas son:

S −> E;
E −> T | E+T
T −> 2 | 6

* Ejercicio 23 *

a) Determine, aplicando derivación a izquierda, si 6; es una expresión correcta.
b) Determine, aplicando derivación a izquierda, si 6+6+6+; es una expresión correcta.
c) Determine, aplicando derivación a derecha, si 6+6+6+; es una expresión correcta.
d) Determine, aplicando derivación a izquierda, si 6+6+6+2; es una expresión correcta.

3 SINTAXIS Y BNF

Un Lenguaje de Programación (LP) es una notación utilizada para describir algoritmos y estructuras de datos que resuelven problemas computacionales.

En este capítulo nos ocuparemos de la SINTAXIS de los Lenguajes de Programación y cómo se describe la misma. Desde el punto de vista sintáctico, hay una relación muy importante entre los LPs y los Lenguajes Formales ya que un LP está formado, básicamente, por un conjunto de LRs y un conjunto de LICs, como veremos en este capítulo.

La descripción precisa de esta sintaxis se realiza, normalmente, en base a notaciones que tienen ciertas propiedades similares a las de las Gramáticas Formales que estamos utilizando. A estas notaciones las llamaremos, genéricamente, BNF.

3.1 INTRODUCCIÓN A LA SINTAXIS

Como ya se ha dicho, sintácticamente un LP está compuesto por un conjunto de LRs y otro conjunto de LICs. Las CATEGORÍAS LÉXICAS o TOKENS (palabra en inglés muy utilizada) −como son los identificadores, las palabras reservadas, los números enteros, los números reales, los caracteres, las cadenas constantes, los operadores, y los caracteres de puntuación− constituyen diferentes LRs. Algunos de estos lenguajes son finitos, como los operadores, y otros son infinitos, como sucede con los identificadores y los números (tanto enteros como reales).

Por otro lado, las expresiones y las sentencias de un LP son, en general, LICs. Los llamaremos CATEGORÍAS SINTÁCTICAS. Como tales, estos lenguajes no pueden ser generados por GRs ni por GQRs –como sí ocurre con los LRs– sino que requieren ser generados por GICs.

Es importante recordar que todos estos tipos de gramáticas se caracterizan porque sus producciones tienen un único noterminal en el lado izquierdo.

➔ En esta área de estudio no tenemos en cuenta las restricciones físicas que imponen las computadoras (cantidad de bytes para almacenar un número real, cantidad máxima de caracteres en una línea del monitor, etc.). Por eso podemos hablar de lenguajes infinitos cuyos elementos tienen longitudes indeterminadas aunque finitas, como ocurre con los Lenguajes Formales.

➔ Una Gramática Formal no solo GENERA un Lenguaje Formal, sino que también se la puede utilizar para DESCRIBIR la sintaxis del lenguaje que genera. Este concepto está muy relacionado con el desarrollo de este capítulo.

La sintaxis de un LP debe describirse con precisión, utilizando una notación sin ambigüedades. La gramática cumple con este requisito, pero tiene muy pocos metasímbolos; esto dificulta, muchas veces, la escritura y posterior comprensión de la sintaxis definida.

Por ello, la notación que se utiliza para describir la sintaxis de un LP es la BNF, que en sus comienzos era muy similar a “producciones de una GIC o de una GQR” y que luego se fue extendiendo con nuevos metasímbolos.

> Hopcroft y Ullman [1979] dicen: “El origen de la formalización de las GICs se encuentra en Chomsky [*1956, "Three models for the description of language"*]. La notación relacionada llamada BNF fue usada para describir el lenguaje ALGOL en Backus [*1959, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference"*] y Naur [*1960, "Report on the algorithmic language ALGOL 60"*]. La relación entre GIC y BNF fue percibida en Ginsburg y Rice [*1962, "Two families of languages related to ALGOL"*].”

Comenzaremos con dos secciones introductorias en las que trabajaremos con Gramáticas Formales y luego pasaremos a ver la utilización de BNF en la descripción de la sintaxis de un Lenguaje de Programación. De esta manera, percibiremos claramente la evolución de la notación BNF.

3.2 IDENTIFICADORES Y SU SINTAXIS

El elemento más utilizado en todo LP es el IDENTIFICADOR: una secuencia de uno o más caracteres que nombra diferentes entidades de un LP (variables, funciones, procedimientos, constantes, tipos, etc.). Los “identificadores” constituyen un LR infinito. Habitualmente, las PALABRAS RESERVADAS, que también están compuestas por una secuencia de caracteres, forman un LR finito, independiente de los identificadores.

Ejemplo 1
Como escribe David Watt en su libro *Programming Language Syntax and Semantics* [1991]: "La siguiente es la especificación informal de la sintaxis de identificadores, parafraseada de un manual de un viejo lenguaje de programación: Un identificador es una secuencia de letras mayúsculas, posiblemente con la inclusión de guiones bajos en medio." A partir de esta especificación en lenguaje natural, es evidente que PI, CANTIDAD y C_TOTAL son identificadores correctos. También es muy claro que PI34, 78AA y CANTIDAD___ no son identificadores válidos.

* Ejercicio 1 *
Verifique que la afirmación escrita en el último párrafo del Ejemplo 1 es correcta para cada una de las cadenas descriptas.

Pero, si nos atenemos a la definición dada en el Ejemplo 1, pueden surgir preguntas como las siguientes:

(1) ¿Es una única letra, como X, un identificador válido? (la especificación dice "una secuencia de letras");

(2) ¿La frase "guiones bajos en medio" significa que puede haber varios consecutivos (la frase está en plural) o que deben estar separados?

* Ejercicio 2 *
En base a la especificación dada en el Ejemplo 1, ¿puede responder con seguridad a las preguntas formuladas en el párrafo anterior?

* Ejercicio 3 *
¿Se le ocurre alguna otra ambigüedad en la especificación informal que estamos analizando?

Estas imprecisiones o ambigüedades son casi inevitables en una especificación informal de una sintaxis, como la que se realiza, habitualmente, utilizando un lenguaje natural.

Ahora bien; supongamos que, para que resulte más claro, usamos una GIC para describir estos identificadores, pero con una pequeña modificación respecto de lo visto en el Capítulo 2: los noterminales serán representados por nombres significativos (por ejemplo, Nombre), en lugar de una sola letra mayúscula. Como siempre, y por convención, el axioma, que es lo que estamos definiendo, aparece en la primera producción.

Entonces, una definición precisa de los identificadores del LP mencionado en el Ejemplo 1 podría ser la siguiente:

Ejemplo 2

```
Identificador -> Letra |
                 Identificador Letra |
                 Identificador GuiónBajo Letra
GuiónBajo -> _
Letra -> A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
         P | Q | R | S | T | U | V | W | X | Y | Z
```

Análisis: Recordemos que toda producción tiene un lado izquierdo, el metasímbolo "operador de producción" y un lado derecho. Recordemos, también, que en la escritura de las producciones solo existen noterminales, terminales y metasímbolos. En este caso, los noterminales son palabras o frases significativas (sin espacios), los únicos metasímbolos son los caracteres `->` y `|`, y los terminales son los restantes caracteres.

> ➔ Es muy importante recordar que todo noterminal debe aparecer en el lado izquierdo de, al menos, una producción; por otro lado, un terminal jamás puede aparecer en el lado izquierdo de una producción. Esta ubicación en el contexto de las producciones también facilita saber cuál es un noterminal y cuál es un terminal, aunque ambos tengan el mismo nombre.

Se nota fácilmente que Identificador debe ser el axioma porque es lo que estamos definiendo. Observe que la primera producción nos dice que un identificador puede ser solo una letra (por ejemplo, X), mientras que las otras dos producciones son recursivas (porque Identificador aparece en ambos lados de la producción) y, por lo tanto, permiten generar identificadores de cualquier longitud. Por supuesto, la primera producción debe ser utilizada para terminar el proceso recursivo. Además, se observa que la última producción no permite que haya dos guiones bajos consecutivos.

En otras palabras: el lenguaje de los "identificadores" descripto en el Ejemplo 1 está perfectamente definido mediante esta GIC. Esto es: si una cadena es un identificador de este LP, la podremos derivar a partir del axioma Identificador; caso contrario, no la podremos derivar.

Nota 1

OBSERVACIÓN IMPORTANTE. Esta GIC produce un Identificador de derecha a izquierda por el formato que tienen las producciones recursivas. Este formato se denomina recursiva a izquierda porque el noterminal del lado izquierdo aparece, también, como primer símbolo del lado derecho.

* Ejercicio 4 *

Construya una tabla de derivación para generar el identificador R_X_A.

* Ejercicio 5 *

Verifique, mediante derivación, que A__B (hay dos guiones bajos consecutivos) no es un identificador válido para el LP que estamos analizando.

* Ejercicio 6 *
Convierta la GIC del Ejemplo 2 en una GQR y compare ésta con la GIC utilizada..

El Identificador que hemos definido en el Ejemplo 2 mediante una GIC recursiva a izquierda también puede ser definido mediante una GIC recursiva a derecha, aunque sus producciones resultarán más complejas:

Ejemplo 3

```
Identificador -> Letra |
                 Letra Resto
Resto -> Letra Resto |
         GuiónBajo Letra Resto |
         ε
GuiónBajo -> _
Letra -> A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
         P | Q | R | S | T | U | V | W | X | Y | Z
```

* Ejercicio 7 *
Construya una tabla de derivación para generar el identificador R_X_A a partir de la gramática del Ejemplo 3.

* Ejercicio 8 *
Verifique mediante derivación, utilizando la GIC del Ejemplo 3, que A__B (hay dos guiones bajos consecutivos) no es un identificador válido para el LP que estamos analizando.

3.3 LAS EXPRESIONES Y LA SINTAXIS

Uno de los puntos más importantes en la sintaxis de un LP está representado por las EXPRESIONES que se pueden escribir en ese LP. Una GIC para expresiones no solo debe producir todas las expresiones válidas, sino que también debe preocuparse por la precedencia (o prioridad) y la asociatividad (evaluación de izquierda a derecha o de derecha a izquierda para operadores con igual precedencia) de cada uno de los operadores existentes en ese LP.

Para comenzar, desarrollaremos una GIC que genera las expresiones aritméticas con multiplicación y suma como únicos operadores. Esta GIC también reconoce el uso de paréntesis, empleado especialmente para modificar las prioridades, como, por ejemplo, (2+34)*5.

Los operandos, para simplificar la situación, son solo números enteros sin signo; además, asumiremos que ya están definidas las producciones para el noterminal Número.

Entonces, en esta GIC los terminales son los caracteres + (suma), * (multiplicación) y los paréntesis; el noterminal Número representará cualquier número entero no negativo que actuará como operando.

Por lo tanto, una GIC para las expresiones aritméticas con estas restricciones se describe en el siguiente ejemplo:

Ejemplo 4

```
Expresión -> Término |
             Expresión + Término
Término -> Factor |
           Término * Factor
Factor -> Número |
          ( Expresión )
```

* Ejercicio 9 *

Complete la GIC del Ejemplo 4, desarrollando las producciones para el noterminal Número.

➔ Representación de las Prioridades de los Operadores en una GIC: Cuánto más cerca del axioma, menor es la prioridad de un operador. Vea lo que sucede con la suma, la multiplicación y los paréntesis (si éstos fueran un operador) en la GIC diseñada en el Ejemplo 4. Esta GIC respeta la prioridad y la asociatividad de los operadores que intervienen.

Veamos un ejemplo de derivación con esta GIC para analizar esta situación:

Ejemplo 5

Sea la expresión 1+2*(3+4)+5. Leyéndola de izquierda a derecha, observamos que primero debemos resolver la subexpresión que está entre paréntesis; luego podremos realizar la multiplicación y, a continuación, seguir adelante con la evaluación. Apliquemos el proceso de derivación a izquierda.

Recordemos las producciones de la GIC:

```
1 Expresión -> Término |
2              Expresión + Término
3 Término -> Factor |
4            Término * Factor
5 Factor -> Número |
6           ( Expresión )
7 Número -> 1 | 2 | 3 | 4 | 5    (agregado para completar el proceso de derivación)
```

Entonces, el proceso de derivación será:

```
Expresión
Expresión + Término
Expresión + Término + Término
Término + Término + Término
Factor + Término + Término
Número + Término + Término
1 + Término + Término
1 + Término * Factor + Término
1 + Factor * Factor + Término
1 + Número * Factor + Término
1 + 2 * Factor + Término
1 + 2 * ( Expresión ) + Término
1 + 2 * ( Expresión + Término ) + Término
1 + 2 * ( Término + Término ) + Término
1 + 2 * ( Factor + Término ) + Término
1 + 2 * ( Número + Término ) + Término
1 + 2 * ( 3 + Término ) + Término
1 + 2 * ( 3 + Factor ) + Término
1 + 2 * ( 3 + Número ) + Término
```

```
1 + 2 * ( 3 + 4 ) + Término
1 + 2 * ( 3 + 4 ) + Factor
1 + 2 * ( 3 + 4 ) + Número
1 + 2 * ( 3 + 4 ) + 5
```

➔ Recordatorio: la secuencia de terminales y noterminales que se obtiene en cada paso de una derivación se denomina CADENA DE DERIVACIÓN.

Ejemplo 6
En el proceso de derivación anterior, `1+2*(Número+Término)+Término` es una de las cadenas de derivación.

* Ejercicio 10 *
¿Cuál es la menor expresión que se puede derivar desde la GIC definida en el Ejemplo 4 y re-escrita en el Ejemplo 5? Escríbala y justifique su respuesta.

* Ejercicio 11 *
¿Es ((2)) una expresión válida? Demuestre que sí o que no por derivación. Para cada cadena de derivación obtenida, indique la producción que fue aplicada.

* Ejercicio 12 *
Intente derivar 1+2++3.

3.3.1 LA EVALUACIÓN DE UNA EXPRESIÓN, PRECEDENCIA Y ASOCIATIVIDAD

La EVALUACIÓN de una expresión se realiza como un proceso inverso al de la derivación, hasta llegar al axioma. En otras palabras: debemos hacer una REDUCCIÓN de la tabla generada por la derivación. Es aquí donde se ve si la precedencia o prioridad de los operadores está correctamente determinada por la gramática diseñada.

Esta reducción debe hacerse en el orden inverso a la derivación; es decir: el último paso de la derivación será el primero de la reducción, y así sucesivamente. Para ello, debe tenerse en cuenta la producción aplicada en cada paso de la derivación. En el próximo ejemplo re-escribiremos la derivación realizada en el Ejemplo 5, pero indicando la producción aplicada en cada paso. Luego, haremos la evaluación de la expresión mediante el proceso de reducción.

Ejemplo 7
Repetimos la derivación para la expresión 1+2*(3+4)+5. Aplicamos el proceso visto en el Ejemplo 5 a partir de la GIC con producciones

```
1 Expresión -> Término |
2              Expresión + Término
3 Término -> Factor |
4            Término * Factor
5 Factor -> Número |
6           ( Expresión )
7 Número -> 1 | 2 | 3 | 4 | 5
```

y construimos la siguiente TABLA DE DERIVACIÓN:

PRODUCCIÓN APLICADA	CADENA DE DERIVACIÓN OBTENIDA
(axioma)	Expresión
2	Expresión + Término
2	Expresión + Término + Término
1	Término + Término + Término
3	Factor + Término + Término
5	Número + Término + Término
7	1 + Término + Término
4	1 + Término * Factor + Término
3	1 + Factor * Factor + Término
5	1 + Número * Factor + Término
7	1 + 2 * Factor + Término
6	1 + 2 * (Expresión) + Término
2	1 + 2 * (Expresión + Término) + Término
1	1 + 2 * (Término + Término) + Término
3	1 + 2 * (Factor + Término) + Término
5	1 + 2 * (Número + Término) + Término
7	1 + 2 * (3 + Término) + Término
3	1 + 2 * (3 + Factor) + Término
5	1 + 2 * (3 + Número) + Término
7	1 + 2 * (3 + 4) + Término
3	1 + 2 * (3 + 4) + Factor
5	1 + 2 * (3 + 4) + Número
7	1 + 2 * (3 + 4) + 5

Ahora, evaluaremos esta expresión. Para ello, como ya adelantamos, aplicaremos el proceso de reducción, que es el proceso inverso al de derivación: partimos de la secuencia de terminales que forman la expresión y finalizamos cuando llegamos al axioma de la GIC.

Construiremos una tabla que llamaremos TABLA DE EVALUACIÓN. Como verán en la tabla, junto a cada noterminal se agrega el número (terminal) del cual proviene, para tenerlo presente al realizar la correspondiente evaluación. La Tabla de Evaluación es la siguiente:

CADENA DE DERIVACIÓN A REDUCIR	PRODUCCIÓN A APLICAR	OPERACIÓN
1 + 2 * (3 + 4) + 5	7	
1 + 2 * (3 + 4) + Número5	5	
1 + 2 * (3 + 4) + Factor5	3	
1 + 2 * (3 + 4) + Término5	7	
1 + 2 * (3 + Número4) + Término5	5	
1 + 2 * (3 + Factor4) + Término5	3	
1 + 2 * (3 + Término4) + Término5	7	
1 + 2 * (Número3 + Término4) + Término5	5	
1 + 2 * (Factor3 + Término4) + Término5	3	
1 + 2 * (Término3 + Término4) + Término5	1	
1 + 2 * (**Expresión3 + Término4**) + Término5	2	3 + 4 = 7
1 + 2 * (Expresión7) + Término5	6	
1 + 2 * Factor7 + Término5	7	
1 + Número2 * Factor7 + Término5	5	
1 + Factor2 * Factor7 + Término5	3	
1 + **Término2 * Factor7** + Término5	4	2 * 7 = 14
1 + Término14 + Término5	7	
Número1 + Término14 + Término5	5	

Factor1 + Término14 + Término5	3	
Término1 + Término14 + Término5	1	
Expresión1 + Término14 + Término5	2	1 + 14 = 15 **(*)**
Expresión15 + Término5	2	15 + 5 = 20
Expresión20	(axioma)	Resultado Final

(*) Aquí se ve un ejemplo muy claro de "qué es la ASOCIATIVIDAD". El operando 14 se puede sumar al operando 1 (a su izquierda) o al operando 5 (a su derecha). Como la suma es asociativa a izquierda, el 14 se suma al 1 y no al 5.

En general: cuando hay tres operandos vinculados con operadores de igual precedencia (como en este caso), el operando central operará primero con el operando de la izquierda si la operación es asociativa a izquierda (como sucede aquí); pero si la operación es asociativa a derecha, el operando central operará primero con el operando de la derecha. Un ejemplo de esta última situación lo veremos más adelante, cuando analicemos el BNF del lenguaje ANSI C.

Además, en la Tabla de Evaluación se ve claramente cómo la GIC determina la PRECEDENCIA de los operadores: la expresión entre paréntesis es evaluada primero, luego evalúa la multiplicación y, finalmente, las sumas.

Nota 2
Para construir una Tabla de Evaluación es indispensable partir de la correspondiente Tabla de Derivación y proceder en orden inverso (reducción).

* Ejercicio 13 *
Siguiendo el modelo presentado en el Ejemplo 7, construya una Tabla de Evaluación que muestre la evaluación de la expresión: (1 + 2) * (3 + 4)

Ejemplo 8
Para terminar de comprender totalmente la GIC para expresiones utilizada en los ejemplos y ejercicios anteriores, analicemos el siguiente caso. Supongamos que consideramos que no es necesario que haya tantos niveles de noterminales (expresión, término, factor, número) para una GIC que "solo" genera expresiones aritméticas con los operadores de suma y de multiplicación. Diseñamos, entonces, una GIC más simple que solo tiene dos noterminales: E (por expresión) y N (por número). Esta GIC tiene las siguientes producciones:

```
1 E -> E + E |
2      E * E |
3      (E) |
4      N
5 N -> 1 | 2 | 3 | 4 | 5
```

* Ejercicio 14 *
Describa formalmente la GIC del Ejemplo 8.

Ejemplo 9
Replicaremos el Ejemplo 7, pero ahora utilizando la nueva GIC cuyas producciones figuran en el Ejemplo 8.

Sea, nuevamente, la expresión 1+2*(3+4)+5. Aplicamos el proceso de derivación a izquierda y construimos la siguiente Tabla de Derivación:

PRODUCCIÓN APLICADA	CADENA DE DERIVACIÓN OBTENIDA
(axioma)	E
1	E + E
2	E * E + E
1	E + E * E + E
4	N + E * E + E
5	1 + E * E + E
4	1 + N * E + E
5	1 + 2 * E + E
3	1 + 2 * (E) + E
1	1 + 2 * (E + E) + E
4	1 + 2 * (N + E) + E
5	1 + 2 * (3 + E) + E
4	1 + 2 * (3 + N) + E
5	1 + 2 * (3 + 4) + E
4	1 + 2 * (3 + 4) + N
5	1 + 2 * (3 + 4) + 5

Como se ve, no solo hemos podido derivar la expresión dada sino que, además, lo hemos realizado en menor cantidad de pasos que utilizando la GIC del Ejemplo 7. ¡Parecería que esta GIC es mejor que la anterior!

Siguiendo el ejemplo anterior, evaluaremos esta expresión aplicando el proceso de reducción. Nuevamente, junto al noterminal colocaremos el número (terminal) que le corresponde, con esta notación: `noterminal.terminal`.

Ejemplo 10
Obtenemos, entonces, la siguiente Tabla de Evaluación:

CADENA DE DERIVACIÓN A REDUCIR	PRODUCCIÓN A APLICAR	OPERACIÓN
1 + 2 * (3 + 4) + 5	5	
1 + 2 * (3 + 4) + N.5	4	
1 + 2 * (3 + 4) + E.5	5	
1 + 2 * (3 + N.4) + E.5	4	
1 + 2 * (3 + E.4) + E.5	5	
1 + 2 * (N.3 + E.4) + E.5	4	
1 + 2 * (**E.3** + **E.4**) + E.5	1	3 + 4 = 7
1 + 2 * (E.7) + E.5	3	
1 + 2 * E.7 + E.5	5	
1 + N.2 * E.7 + E.5	4	
1 + E.2 * E.7 + E.5	5	
N.1 + E.2 * E.7 + E.5	4	
E.1 + **E.2** * E.7 + E.5	1	1 + 2 = 3
E.3 * **E.7** + E.5	2	3 * 7 = 21
E.21 + **E.5**	1	21 + 5 = 26
E.26	(axioma)	Resultado Final

Como se observa, el resultado final es incorrecto, y esto se debe a que la segunda gramática está mal diseñada porque no resuelve correctamente el problema de precedencia de los operadores.

* Ejercicio 15 *
Utilizando la GIC cuyas producciones están en el Ejemplo 8, obtenga otra Tabla de Derivación para la expresión del Ejemplo 9. Luego, construya la Tabla de Evaluación correspondiente. ¿Cuáles son sus conclusiones?

* Ejercicio 16 *
¿Puede hacer lo mismo que hizo en el ejercicio anterior, pero con la GIC del Ejemplo 7? Justifique su respuesta.

3.4 BNF Y ALGOL

La sigla BNF corresponde a la frase "Backus Normal Form" (Forma Normal de Backus) o también a la frase "Backus-Naur Form" (Forma de Backus y Naur). Con el breve Manual de Referencia del lenguaje ALGOL (*ALGOrithmic Language*) se publicó, por primera vez en 1960, una descripción formal de la sintaxis de un LP. Esta descripción, similar a las producciones de las GICs, se llamaría luego BNF.

La notación BNF consiste en un conjunto de REGLAS que definen, con precisión, la sintaxis de los componentes y de las estructuras de un LP dado, como hemos hecho en secciones anteriores, pero utilizando GICs. Pero, como veremos en el resto de este capítulo, BNF ha evolucionado incorporando nuevos metasímbolos; en esto radica una diferencia fundamental entre BNF y una Gramática Formal.

Volviendo al origen de BNF, la diferencia entre la notación utilizada para describir las producciones de una GIC y la usada para representar las reglas BNF era mínima: el metasímbolo "operador de producción" comenzó a representarse como ::= (al que podríamos llamar "operador ES") y los nombres de los noterminales fueron encerrados entre corchetes angulares.

Ejemplo 11
Algunos elementos básicos del lenguaje ALGOL pueden ser enumerados de esta forma:

```
<símbolo básico> ::= <letra> | <dígito> | <valor lógico> | <delimitador>
```

Estas cuatro reglas BNF se pueden leer de esta forma: "un símbolo básico es una letra o un dígito o un valor lógico o un delimitador".

Cada regla BNF se forma con elementos provenientes de tres conjuntos disjuntos:

– ***metavariables*** o ***noterminales***, que son palabras o frases encerradas entre corchetes angulares (ejemplo: <símbolo básico>, <identificador>).

– ***terminales***, que son los caracteres del alfabeto o palabras del lenguaje sobre los cuales se construye el LP descripto. Se los llama terminales porque no existen reglas de producción para ellos (ejemplo: una palabra reservada es un terminal).

– ***metasímbolos***, que son caracteres o grupos de caracteres que ayudan a representar estas reglas (ejemplo: <>, ::=, |).

En toda regla BNF, encontramos, al igual que en toda producción de una GIC, tres componentes:

(1) el lado izquierdo, formado por un solo noterminal

(2) el lado derecho, formado por terminales y noterminales o, eventualmente, vacío

(3) un metasímbolo, el operador es, que vincula el lado izquierdo con el lado derecho.

Ejemplo 12

`<símbolo básico> ::= <letra>` que se lee: "un Símbolo Básico es una Letra".

* Ejercicio 17 *

Si el lado derecho de una regla BNF es vacío, ¿qué significa? Justifique su respuesta.

Los tres metasímbolos que aparecen en las reglas del Ejemplo 11 y en la BNF general del lenguaje ALGOL son:

(1) **<** y **>** (para encerrar una palabra o frase que es el nombre de un noterminal);

(2) **::=** (que se lee "es" o "corresponde a"), y

(3) **|** (que se lee "ó").

Estos son los únicos metasímbolos que formaban parte de la definición original de BNF.

3.4.1 DOS EJEMPLOS DE BNF EN ALGOL

En ALGOL 60, un identificador debía comenzar con una letra y, si tenía más caracteres, éstos podían ser letras y dígitos decimales. La categoría léxica "Identificador en ALGOL 60" constituye un LR infinito.

Ejemplo 13

Esta es la definición BNF de un identificador en ALGOL 60:

```
<identificador> ::= <letra> |
                    <identificador> <letra> |
                    <identificador> <dígito>
<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
            p | q | r | s | t | u | v | w | x | y | z | A | B | C | D |
            E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
            T | U | V | W | X | Y | Z
<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Análisis:

– El conjunto de tres reglas para <identificador>, con dos reglas recursivas a izquierda, equivale a una GQR.

– El conjunto de reglas para <letra> se lee: "*letra* es cualquier letra minúscula o mayúscula del alfabeto inglés".

– El conjunto de reglas para <dígito> se lee: "*dígito* es cualquier dígito decimal (base 10)".

Nota 3

<letra> y <dígito> definen dos LRs finitos, mientras que <identificador> define un LR infinito.

En cuanto al LR infinito de los números enteros, veamos cómo se define en ALGOL.

Ejemplo 14

```
<número entero> ::= <entero sin signo> |
                    + <entero sin signo> |
                    - <entero sin signo>
<entero sin signo> ::= <dígito> | <entero sin signo> <dígito>
<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Análisis:
– Las tres reglas de <número entero> definen al número entero sin signo y con signo.
– El noterminal <entero sin signo>, por medio de una regla recursiva a izquierda y otra de terminación de la recursividad, define una sucesión de uno o más dígitos decimales. En consecuencia, un *número entero* es: (a) una sucesión de dígitos, o (b) una sucesión de dígitos precedida por el signo positivo, o (c) una sucesión de dígitos precedida por el signo negativo.

* Ejercicio 18 *
Derive el número entero `-123456`. Si desea, puede abreviar el nombre de los noterminales explicando qué significa cada abreviatura utilizada.

A continuación veremos cómo fue descripto el lenguaje Pascal por su creador, Niklaus Wirth, y la primera ampliación del conjunto de metasímbolos en BNF.

3.5 BNF Y EL LENGUAJE PASCAL

Independientemente de la notación que utilicemos y del LP descripto, las BNFs están formadas por:

1) un conjunto de NOTERMINALES
2) un conjunto de TERMINALES
3) un conjunto de METASÍMBOLOS
4) un conjunto de PRODUCCIONES o REGLAS

Seguidamente, veremos una extensión que realizó Niklaus Wirth a la notación BNF utilizada para describir la sintaxis del ALGOL. Por ello, a partir de esta extensión y de otras que analizaremos luego, la sigla BNF fue cambiada por la de EBNF (BNF extendida). No obstante, muchos autores siguen utilizando la sigla BNF también para las BNFs extendidas; nosotros haremos lo mismo.

3.5.1 LA SINTAXIS DEL LENGUAJE PASCAL, SEGÚN WIRTH

En el libro "Pascal – User Manual and Report" (1974), escrito por Niklaus Wirth –creador de este lenguaje– y Kathleen Jensen, se describen con precisión la sintaxis y la semántica del lenguaje Pascal. En esta sección nos ocuparemos de su descripción sintáctica.

En la Introducción del libro mencionado, Wirth dice: "Una formulación de la sintaxis es la tradicional BNF, en la que los *constructos sintácticos* [categorías léxicas y categorías sintácticas]

son denotados por palabras en inglés [en castellano, en nuestro caso] encerradas entre los corchetes angulares **<** y **>**."

Por otro lado, al conjunto de metasímbolos ya existente en ALGOL, Wirth agrega un metasímbolo muy importante y lo describe así: "Las llaves **{** y **}** denotan la posible repetición de los símbolos que encierra, cero o más veces." Es una forma más compacta de describir una secuencia de elementos, como veremos en próximos ejemplos.

Este metasímbolo es la primera extensión que se realizó sobre la BNF utilizada en ALGOL: a los tres metasímbolos ya existentes en ALGOL, Wirth agregó el metasímbolo indicado por un par de llaves para representar lo que conocemos como el operador *clausura de Kleene*, es decir: cero o más veces lo que está encerrado entre las llaves.

A partir de ésta y de otras extensiones posteriores es que algunos autores utilizan la sigla EBNF para referirse a una BNF extendida. Pero, como ya se mencionó, nosotros seguiremos utilizando la sigla BNF, independientemente de las extensiones que se agreguen y de los caracteres utilizados para representar a los metasímbolos.

Veamos, a continuación, algunos ejemplos de la BNF que utiliza Wirth para describir la sintaxis del lenguaje Pascal en su Manual de Referencia original.

Ejemplo 15
La definición de identificador en el Pascal original y con esta BNF extendida es:

```
<identificador> ::= <letra> { <letra o dígito> }
<letra o dígito> ::= <letra> | <dígito>
<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
            p | q | r | s | t | u | v | w | x | y | z | A | B | C | D |
            E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
            T | U | V | W | X | Y | Z
<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

* Ejercicio 19 *
(a) Interprete, mediante una frase, la regla <identificador>.
(b) Convierta la regla <identificador> a la notación utilizada en ALGOL.
(c) ¿Cuál notación le parece más conveniente?

Ejemplo 16
La definición de la constante real sin signo en el Pascal original y con esta BNF extendida es la siguiente:

```
<real sin signo> ::=
           <entero sin signo> . <entero sin signo> |
           <entero sin signo> . <entero sin signo> E <factor de escala> |
           <entero sin signo> E <factor de escala>
<entero sin signo> ::= <dígito> { <dígito> }
<factor de escala> ::= <entero sin signo> | <signo> <entero sin signo>
<signo> ::= + | -
<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Análisis:
Como se puede apreciar, estas producciones definen con precisión tanto a las constantes reales en punto fijo como a las constantes reales en punto flotante; para éstas últimas utiliza la letra E. Como se puede apreciar también, el Pascal original tiene una única forma de escribir las constantes reales en punto fijo y tiene dos formas de escribir las constantes reales en punto flotante.

* Ejercicio 20 *
(a) Escriba el conjunto de terminales utilizados en la BNF del Ejemplo 16.
(b) Escriba el conjunto de metasímbolos utilizados en la BNF del Ejemplo 16.
(c) Escriba ejemplos de todos los casos de <real sin signo> descriptos por la BNF del Ejemplo 16.

Ejemplo 17
El constructo <programa Pascal> es definido por Wirth de esta manera:

```
<programa Pascal> ::= <encabezamiento> <bloque> .
<encabezamiento> ::= program <identificador> ( <identificador-archivo>
                     {, <identificador-archivo>} ) ;
<identificador-archivo> ::= <identificador>
<bloque> ::= <sección definición de constantes>
             <sección definición de tipos>
             <sección declaración de variables>
             <sección declaración de procedimientos y funciones>
             <sección sentencias>
<sección definición de constantes> ::=
          <vacío> |
          const <definición de constante> { ; <definición de constante> }
<definición de constante> ::= <identificador> = <constante>
<vacío> ::=
<sección declaración de variables> ::=
          <vacío> |
          var <declaración de variable> { ; <declaración de variable> }
<declaración de variable> ::=
          <identificador> { , <identificador> } : <tipo>
<sección sentencias> ::= <sentencia compuesta>
<sentencia compuesta> ::= begin <sentencia> { ; <sentencia> } end
<sentencia> ::= <sentencia simple> | <sentencia estructurada>
<sentencia simple> ::= <sentencia de asignación> |
                       <sentencia de procedimiento> |
                       <sentencia vacía>
<sentencia de asignación> ::= <variable> := <expresión>
<sentencia de procedimiento> ::= <identificador de procedimiento> | . . .
<sentencia vacía> ::= <vacío>
<sentencia estructurada> ::= . . .
```

* Ejercicio 21 *
Preguntas referidas a la BNF descripta en el Ejemplo 17:
(a) ¿Cómo se representan las palabras reservadas en esta BNF?
(b) ¿Cuántos elementos forman el conjunto de terminales y cuáles son?
(c) ¿Qué significa <vacío>?
(d) ¿Cuántas reglas tiene el noterminal <bloque>?

(continúa en la página siguiente)

(e) De acuerdo a esta BNF, ¿existe en Pascal la sentencia compuesta vacía de acciones (sentencias)? Justifique su respuesta.
(f) ¿Cuántas variables se pueden declarar para un <tipo> determinado?

* Ejercicio 22 *
Escriba dos ejemplos de <sentencia compuesta>.

3.5.2 EXPRESIONES EN PASCAL

A continuación definiremos un subconjunto de las expresiones en Pascal. Analizaremos las prioridades de los operadores en este LP y también veremos cómo la BNF no se ocupa de los tipos de los datos.

```
<expresión> ::= <expresiónSimple> |
                <expresiónSimple> <operadorRelacional> <expresiónSimple>
<operadorRelacional> ::= = | <> | < | <= | > | >=
<expresiónSimple> ::= <término> |
                      <signo> <término> |
                      <expresiónSimple> <operadorAditivo> <término>
<operadorAditivo> ::= + | - | or
<término> ::= <factor> | <término> <operadorMultiplicativo> <factor>
<operadorMultiplicativo> ::= * | / | and
<factor> ::= <variable> | <número> | ( <expresión> ) | not <factor>
. . .
```

Precedencia de los operadores
En el libro de Wirth se lee: "El operador not (aplicado a un operando Booleano) tiene la máxima prioridad. Le siguen los operadores multiplicativos, luego los operadores aditivos y los de mínima prioridad son los operadores relacionales."

Como ya hemos dicho, observe que, en la BNF, los operadores van apareciendo en orden inverso a sus prioridades: los operadores relacionales son los que están más cerca del axioma (expresión, en este caso); a continuación aparecen los operadores aditivos, luego los operadores multiplicativos y, finalmente, el operador not (el de máxima prioridad).

Una curiosidad a destacar es que operadores aritméticos y cierto operador lógico aparecen en el mismo nivel de prioridades, como se observa en dos de las reglas BNF.

* Ejercicio 23 *
Escriba los operadores aritméticos y el operador lógico que aparecen en el mismo nivel.

Otra curiosidad del Lenguaje de Programación Pascal surge en su comparación con el lenguaje matemático. Si en matemáticas escribimos la expresión lógica a > 24 y b < 5, no necesitamos utilizar paréntesis porque los operadores relacionales tienen mayor prioridad que los operadores lógicos o Booleanos. En cambio, vemos que en Pascal los operadores relacionales tienen la mínima prioridad; por lo tanto, los paréntesis serán imprescindibles.

Veamos cómo sería el proceso de derivación a izquierda para esa misma expresión lógica en Pascal; se abrevian los nombres de los noterminales para mayor simplicidad:

Ejemplo 18

```
<E>
<ES>
<T>
<T> <OM> <F>
<F> <OM> <F>
(<E>) <OM> <F>
(<ES> <OR> <ES>) <OM> <F>
(<T> <OR> <ES>) <OM> <F>
(<F> <OR> <ES>) <OM> <F>
(<V> <OR> <ES>) <OM> <F>
(A <OR> <ES>) <OM> <F>
(A > <ES>) <OM> <F>
(A > <T>) <OM> <F>
(A > <F>) <OM> <F>
(A > <N>) <OM> <F>
(A > 24) <OM> <F>
(A > 24) and <F>
(A > 24) and (<E>)
(A > 24) and (<ES> <OR> <ES>)
(A > 24) and (<T> <OR> <ES>)
(A > 24) and (<F> <OR> <ES>)
(A > 24) and (<V> <OR> <ES>)
(A > 24) and (B <OR> <ES>)
(A > 24) and (B < <ES>)
(A > 24) and (B < <T>)
(A > 24) and (B < <F>)
(A > 24) and (B < <N>)
(A > 24) and (B < 5)
```

* Ejercicio 24 *
Encuentre otro proceso de derivación a izquierda para la misma expresión lógica.

Ejemplo 19
Analicemos otra derivación que utiliza solo el operador or:

```
<E>
<ES>
<ES> <OA> <T>
<T> <OA> <T>
<F> <OA> <T>
<V> <OA> <T>
A <OA> <T>
A or <T>
A or <F>
A or <V>
A or B
```

La expresión obtenida es, supuestamente, “sintácticamente correcta” porque la pudimos derivar de la BNF oficial del lenguaje Pascal. Sin embargo, esta expresión solo tiene sentido si ambas variables son Booleanas.

Ejemplo 20
Dos situaciones peores que la anterior:

```
. . .
A or <F>
A or <N>
A or 12
```

y

```
<E>
<ES>
<T>
<F>
not <F>
not 436
```

Ambas expresiones obtenidas son, supuestamente, "sintácticamente correctas" porque las pudimos derivar de la BNF oficial del lenguaje Pascal.

* Ejercicio 25 *
Las dos derivaciones del Ejemplo 20, ¿producen expresiones "sintácticamente correctas" en Pascal? Justifique su respuesta. ¿Qué podría hacer para mejorar esta situación?

3.5.3 SENTENCIAS CON CONDICIONES BOOLEANAS

Las descripciones en BNF no siempre son exactas desde el punto de vista del programador. Ya lo hemos visto en los ejemplos 19 y 20. Estas "fallas" del BNF requieren el agregado de una RESTRICCIÓN en Lenguaje Natural que brinde máxima precisión a la situación descripta.

Veamos qué sucede con la descripción de sentencias como while e if. En el Manual de Referencia original de Pascal, Wirth las describe de esta manera:

```
<sentencia if> ::= if <expresión> then <sentencia> |
                   if <expresión> then <sentencia> else <sentencia>

<sentencia while> ::= while <expresión> do <sentencia>
```

Como observamos, tanto para la sentencia if como para la sentencia while la condición está representada mediante <expresión>, aunque sabemos que no puede ser una expresión cualquiera..

Ejemplo 21
De acuerdo a las definiciones escritas en BNF, una sentencia como while 2+3 do a := 5 sería una sentencia sintácticamente correcta ya que 2+3 es una <expresión> válida. Sin embargo, sabemos que la condición de un while en Pascal debe ser una expresión BOOLEANA.

➔ Conclusión importante: En algunos casos, no solo se debe conocer la definición en BNF del constructo, sino también las RESTRICCIONES que aparecen escritas en lenguaje natural. En el caso del libro de Wirth, algunas de estas definiciones en BNF tienen aclaraciones en las secciones del libro donde se trata el correspondiente constructo. En el caso de la <sentencia while>, por ejemplo, luego de su definición en BNF se aclara que <expresión> debe ser de tipo Boolean; esta restricción es tan importante como la misma definición BNF y, por ello, no pueden ser utilizadas separadamente.

* Ejercicio 26 *
Modifique la escritura de la BNF para if y para while de tal forma que no requieran una aclaración en lenguaje natural.

3.6 BNF Y EL ANSI C

El Lenguaje de Programación C fue desarrollado entre 1969 y 1972 por Dennis Ritchie, con el objetivo principal de implementar el sistema operativo UNIX en un lenguaje de alto nivel.

A mediados de la década de los 80, el ANSI (*American National Standards Institute*) comenzó a desarrollar una estandarización de este LP. En diciembre de 1989, esta estandarización fue aprobada y se publicó en 1990 como "American National Standard for Information Systems – Programming Language C"; lo llamaremos *Manual de Referencia Oficial del ANSI C* [MROC].

A partir de ese momento, este LP es conocido como ANSI C, aunque en la actualidad ya se utilizan C y ANSI C como sinónimos.

En esta sección se describe y se analiza la sintaxis de un subconjunto del ANSI C, de acuerdo a su definición oficial. Tengamos en cuenta, entonces, cómo es el BNF que utiliza el MROC:

1º Los noterminales (categorías léxicas y sintácticas) están escritos en itálica, eliminándose los metasímbolos < y >.

2º El "operador es" está representado por : ("dos puntos"). Como dice el MROC, "un : después de un noterminal introduce su definición."

3º No existe el metasímbolo | ("ó"), sino que cada lado derecho de un determinado noterminal se escribe en una línea separada.

4º En algunos casos se usa el metasímbolo uno de para representar varios lados derechos que son caracteres para un noterminal; es cuando un noterminal representa un conjunto de caracteres.

5º Los terminales se escriben en negritas.

6º Un símbolo opcional está indicado con un subíndice $_{op}$.

7º El lado derecho de una regla recursiva se representa como en el BNF original; no utiliza el metasímbolo incorporado por Wirth para indicar una secuencia de "0 o más veces".

Ejemplo 22
Un token o categoría léxica se define se esta manera, con cada lado derecho en una línea separada:

token: *palabraReservada*
identificador
constante
literalCadena
operador
carácterPuntuación

Con el solo fin de escribir estas definiciones en forma más compacta, en algunos casos ampliaremos el uso del metasímbolo uno de, empleándolo así:

Ejemplo 23
La definición del Ejemplo 22 la podemos escribir en forma más compacta de esta manera:

token: uno de *palabraReservada identificador constante literalCadena operador carácterPuntuación*

* Ejercicio 27 *
¿El Ejemplo 23 define LRs o LICs? Si son LRs, indique cuáles son finitos y cuáles son infinitos. Si son LICs, justifique su afirmación.

Ejemplo 24
Con respecto al uso de un símbolo opcional, el MROC, presenta el siguiente ejemplo:

{ *expresión*$_{op}$ **}**

que indica una expresión opcional encerrada entre llaves; esto es:

{ *expresión*$_{op}$ **}** equivale a: **{** *expresión* **}** ó **{ }**

A continuación veremos un ejemplo de un programa simple – pero completo – en ANSI C, que será utilizado luego como referencia. Las líneas del programa serán numeradas al solo efecto de facilitar su identificación cuando nos referimos a ellas en las diversas explicaciones; esta numeración nunca forma parte de un programa en ANSI C.

3.6.1 UN PROGRAMA EN ANSI C

Ejemplo 25

```
 1  /* Programa en ANSI C – J.Muchnik – feb'2010 */
 2  #include <stdio.h>
 3  int Mayor (int, int);
 4  int main (void) { /* comienza main */
 5     int n1, n2, max;
 6     printf ("Ingrese dos números enteros: ");
 7     scanf ("%d %d", &n1, &n2);
 8     max = Mayor (n1, n2);
 9     printf ("El mayor entre %d y %d es %d\n", n1, n2, max);
10     return 0;
11  } /* fin main */
12  int Mayor (int a, int b) { /* comienza Mayor */
13     if (a > b)
14        return a;
15     else
16        return b;
17  } /* fin Mayor */
```

Análisis:
Este programa en ANSI C está compuesto por:
Linea 1: un comentario (no forma parte del lenguaje ANSI C; es analizado por el preprocesador)
Linea 2: una directiva al preprocesador (no forma parte del lenguaje ANSI C)
Linea 3: el prototipo de la función Mayor (es la declaración de esta función)
Lineas 4 a 11: la definición de la función main, función principal de todo programa ANSI C
Lineas 12 a 17: la definición de la función Mayor, que es utilizada por la función main.

* Ejercicio 28 *
(a) Investigue e informe qué es el PREPROCESADOR.
(b) Investigue e informe cada uno de los constructos que componen el programa del Ejemplo 25.
(c) Investigue e informe qué hace el programa C del ejemplo anterior.

A continuación definiremos la sintaxis de este simple programa C, en BNF. Recordemos que los comentarios y las directivas al preprocesador no pertenecen al ANSI C, como se indicó en el Análisis del Ejemplo 25; por ello, en el próximo ejemplo aparece el noterminal *noC* para representar los contenidos de las líneas 1 y 2.

Ejemplo 26

programaC: *noC prototipo main función*
prototipo: *tipoFunción nombreFunción* **(** *tiposParámetros* **) ;**
tiposParámetros: *tipoUnParámetro*
tiposParametros **,** *tipoUnParámetro*
main: *encabezamiento cuerpo*
encabezamiento: **int** main **(void)**
cuerpo: **{** *declaración sentencias* **}**
declaración: *tipoVariables variasVariables ;*
variasVariables: variable
variasVariables variable
sentencias: *sentencia*
sentencias sentencia
. . .

* Ejercicio 29 *
Analice cada noterminal definido en el Ejemplo 26 y escriba a qué líneas del programa C del Ejemplo 25 corresponde.

* Ejercicio 30 *
Sea un programa C cuya función main es la siguiente:

```
int main (int argc, char *argv[]) {
   char vec [10+1];
   int i;
   if (argc > 10) {
      printf ("No se puede\n");
   else
      for (i=0; i < argc; i++)
         vec[i] = argv[i][0];
      vec[i] = '\0';
      printf ("La cadena creada es %s\n", vec);
   return 0;
}
```

Investigue e informe el significado de cada componente de esta función y el de la función en su totalidad. Escriba una BNF que describa la estructura general de esta función main.

3.6.2 SINTAXIS DE UN SUBCONJUNTO DE ANSI C

En el MROC, cada una de las secciones está dividida en tres partes:

1) "Sintaxis", con una BNF que define la sintaxis del constructo analizado;
2) "Restricciones", en la que figuran ciertas restricciones existentes sobre el constructo definido en "Sintaxis" y que la BNF no expresa totalmente (como ya hemos visto en el caso de Pascal);
3) "Semántica", donde se explica el significado de lo definido en "Sintaxis", lo veremos en el capítulo 4.

En esta sección analizaremos la primera parte, "Sintaxis", y, en algunas ocasiones, haremos referencia a la segunda parte, "Restricciones", para completar la comprensión de la sintaxis.

La tercera parte, "Semántica", será analizada en el próximo capítulo.

3.6.2.1 LAS CATEGORÍAS LÉXICAS

Esta sección se refiere a los TOKENS o CATEGORÍAS LÉXICAS del ANSI C, que ya han sido enumerados en el Ejemplo 22: las palabras reservadas, los identificadores, las constantes, las cadenas literales, los operadores y los caracteres de puntuación. Estos tokens son LRs.

Los elementos que componen un LR se denominan palabras en el área de los Lenguajes Formales. Aquí los llamaremos LEXEMAS, término utilizado en el área de los compiladores que cubriremos en el Volumen 2.

Nota 4
Siendo rigurosos, un "lexema" puede ser tanto una palabra de un LR (token), como puede ser una cadena que no pertenece a ningún LR. El último caso, si aparece, será debidamente aclarado.

A continuación analizaremos el programa del Ejemplo 25 y construiremos una tabla en la que figuren algunos de los lexemas de esta codificación en ANSI C y a qué CATEGORÍA LÉXICA o TOKEN pertenecen. En otras palabras, trabajaremos con los LRs que componen la sintaxis del ANSI C.

Ejemplo 27
Como los comentarios y las directas al preprocesador no pertenecen al lenguaje ANSI C, los eliminamos en el proceso de detección de lexemas:

```
 3  int Mayor (int, int);
 4  int main (void) {
 5     int n1, n2, max;
 6     printf ("Ingrese dos numeros enteros: ");
 7     scanf ("%d %d", &n1, &n2);
 8     max = Mayor (n1, n2);
 9     printf ("El mayor entre %d y %d es %d\n", n1, n2, max);
10     return 0;
11  }

/* continúa */
```

```
12  int Mayor (int a, int b) {
13     if (a > b)
14        return a;
15     else
16        return b;
17  }
```

NRO.LÍNEA	LEXEMA	TOKEN
3	int	*palabraReservada*
3	Mayor	*identificador*
3	(	*carácterPuntuación*
3	;	*carácterPuntuación*
4	main	*identificador*
4	void	*palabraReservada*
4	{	*carácterPuntuación*
5	n1	*identificador*
6	printf	*identificador*
6	(	*operador*
6	"Ingrese dos numeros enteros: "	*literalCadena*
7	&	*operador*
8	=	*operador*
10	return	*palabraReservada*
10	0	*constante*
13	if	*palabraReservada*
..	..	..
..	..	..

* Ejercicio 31 *
Complete la tabla del Ejemplo 25 e indique cuántos lexemas tiene en total el programa.

* Ejercicio 32 *
Sea la siguiente función en ANSI C:

```
double XX (double a, int b) {
   while (a > b) b++;
   return b;
}
```

Construya una tabla, como la del ejemplo anterior, con <u>todos</u> los lexemas que hay en esta función y la CATEGORÍA LÉXICA a la que pertenece cada uno.

Habíamos dicho que los tokens del ANSI C son: *palabraReservada*, *identificador*, *constante*, *literalCadena*, *operador* y *carácterPuntuación*.

Algunos de estos tokens representan LRs finitos y otros representan LRs infinitos; pero todos representan LRs. Veamos cada uno de ellos, teniendo en cuenta que <u>las BNFs para los tokens siempre son precisas y completas</u>.

3.6.2.1.1 LAS PALABRAS RESERVADAS

El ANSI C posee 32 palabras reservadas que constituyen los elementos de un LR finito: el token palabraReservada. Mediante BNF haremos una lista de algunas de ellas:

palabraReservada: una de **char do double else float for if int long return sizeof struct typedef void while**

* Ejercicio 33 *
Investigue cada una de esas palabras reservadas; defínala y muestre ejemplos de su uso.

* Ejercicio 34 *
Investigue e informe si main es una palabra reservada. Si no lo es, ¿la puede utilizar como nombre de una variable? Justifique su afirmación.

3.6.2.1.2 LOS IDENTIFICADORES

Otro de los tokens es identificador, que constituye un LR infinito. Siguiendo la línea histórica de ALGOL y de PASCAL, los identificadores en C tampoco pueden comenzar con un dígito. ¿Por qué? Este LR infinito es definido en BNF de la siguiente manera:

identificador: noDígito
identificador noDígito
identificador dígito
noDígito: uno de **_ a b c d e f g h i j k l m n o p q r s t u v w x y z**
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
dígito: uno de **0 1 2 3 4 5 6 7 8 9**

Nota 5
Observe que el carácter "guión bajo" forma parte del conjunto definido por noDígito. Por lo tanto, en ANSI C, un identificador puede comenzar con letra o con "guión bajo".

* Ejercicio 35 *
(a) Investigue e informe si _ (un guión bajo) y __ (dos guiones bajos) son identificadores válidos en ANSI C.
(b) Investigue e informe si los identificadores `ab3` y `AB3` representan el mismo objeto. Justifique su respuesta.

3.6.2.1.3 LAS CONSTANTES

Continuamos con la descripción de los tokens. Las constantes son definidas en el MROC de la siguiente manera:

constante: una de *constanteReal constanteEntera constanteEnumeración constanteCarácter*

Ahora veremos una BNF parcial de las constantes: nos ocuparemos de las constantes numéricas, tanto enteras como reales. Ambas representan diferentes LRs infinitos. Las constantes numéricas se definen "sin signo" porque, en ANSI C, el signo es un operador. Entonces,

constanteNumérica: una de *constanteEntera constanteReal*

Comenzamos con la definición de la constanteEntera teniendo en cuenta que, en ANSI C, existen tres tipos de constantes enteras:

(a) las decimales, o sea, en base 10;
(b) las octales (en base 8); y
(c) las hexadecimales (en base 16).

En consecuencia, La BNF para las constantes enteras en ANSI C es la siguiente:

constanteEntera: *constanteDecimal sufijoEntero*$_{op}$
constanteOctal sufijoEntero$_{op}$
constanteHexadecimal sufijoEntero$_{op}$
constanteDecimal: *dígitoNoCero*
constanteDecimal dígito
constanteOctal: **0**
constanteOctal dígitoOctal
constanteHexadecimal: *ceroX dígitoHexa*
constanteHexadecimal dígitoHexa
ceroX: uno de **0x 0X**
dígitoNoCero: uno de **1 2 3 4 5 6 7 8 9**
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
dígitoOctal: uno de **0 1 2 3 4 5 6 7**
dígitoHexa: uno de **0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F**
<sufijoEntero>: . . .

* Ejercicio 36 *
(a) Investigue e informe qué significa el noterminal sufijoEntero.
(b) Sea la constante 425. ¿Es una constante octal correcta? Justifique su respuesta.

* Ejercicio 37 *
(a) De acuerdo al BNF del MROC, ¿cuál es la mínima constanteDecimal?
(b) ¿Cómo se representa el número entero decimal 0 en ANSI C?

* Ejercicio 38 *
(a) Para cada una de las siguientes constantes enteras, escriba a cuál de los tres tipos pertenece:
00 123U 0x4A2F2 0654 99LU 02L
(b) Derive verticalmente por izquierda la constante 0Xa4b8.

Continuamos con la BNF de las constantes. Ahora corresponde definir la sintaxis de las constantes reales, que se pueden representar en punto fijo (como 2.14) o en punto flotante (como 3E18, que significa $3x10^{18}$). La BNF para las constantes reales en ANSI C es la siguiente:

constanteReal: *constanteFraccionaria parteExponente*$_{op}$ *sufijoReal*$_{op}$
secuenciaDígitos parteExponente sufijoReal$_{op}$
constanteFraccionaria: *secuenciaDígitos*$_{op}$ **.** *secuenciaDígitos*
secuenciaDígitos **.**
parteExponente: *operadorE signo*$_{op}$ *secuenciaDígitos*
operadorE: uno de **e E**
signo: uno de **+ -**

(continúa en la página siguiente)

secuenciaDígitos: dígito
secuenciaDígitos dígito
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
sufijoReal: . . .

* Ejercicio 39 *
(a) Investigue e informe qué significa el noterminal sufijoReal.
(b) Sea la constante 425. ¿Es una constante real? Justifique su respuesta.

* Ejercicio 40 *
(a) De acuerdo al BNF del MROC, ¿cuál es la mínima constanteReal? Escríbala en todos los formatos posibles.
(b) Construya una tabla con todos los formatos de constantes reales existentes en ANSI C y un ejemplo para cada uno de ellos.

* Ejercicio 41 *
Derive verticalmente por izquierda la constante real 4.6E–23

* Ejercicio 42 *
Una constanteCarácter se representa así: 'a'; es decir, el carácter a, en este caso, se escribe entre DELIMITADORES apóstrofos. Investigue e informe cómo se representa la constanteCarácter apóstrofo (').

* Ejercicio 43 *
Investigue e informe qué es y cómo se representa una constanteEnumeración.

* Ejercicio 44 *
El token literalCadena corresponde a todas las cadenas de caracteres que se pueden representar en ANSI C, como "abc"; como se observa, los DELIMITADORES son comillas.
(a) Investigue e informe cómo se representa el literalCadena "".
(b) Investigue e informe cómo se representa el literalCadena formado por estos tres caracteres: "A".
(c) Investigue e informe si un literalCadena puede ser vacío y, en tal caso, cómo se representa. Justifique su respuesta.
(d) Investigue e informe sobre la diferencia existente entre "234" y 234.
(e) Escriba una BNF que defina el subconjunto de cadenas formadas solo por dos o más dígitos, más la cadena vacía.

3.6.2.1.4 LOS OPERADORES Y LOS CARACTERES DE PUNTUACIÓN

Veamos, a continuación, las BNF para algunos operadores y para algunos caracteres de puntuación del ANSI C; ambos son LRs finitos:

operador: uno de **++ * + & ! sizeof / % < <= == != && || ?: = +=**
carácterPuntuación: uno de **( ) { } , ; [ ]**

* Ejercicio 45 *
(a) Investigue e informe sobre cada uno de los operadores mencionados arriba. Tenga en cuenta que, en algunos casos, un símbolo puede representar más de un operador.
(b) Brinde diferentes ejemplos de aplicación de esos operadores.

* Ejercicio 46 *
Investigue e informe sobre cada uno de los caracteres de puntuación mencionados arriba.

* Ejercicio 47 *
Existen caracteres en ANSI C que son operadores y también son caracteres de puntuación. Esto es lo que ocurre con los paréntesis, los corchetes y con la "coma". Investigue e informe, con ejemplos, cuándo estos caracteres representan caracteresPuntuación y cuándo representan operadores.

3.6.2.2 LAS CATEGORÍAS GRAMATICALES

En esta sección analizaremos la sintaxis de tres constructos más complejos:
(a) las expresiones,
(b) las declaraciones; y
(c) las sentencias.

3.6.2.2.1 LAS EXPRESIONES

Las expresiones constituyen un importantísimo LIC en cualquier Lenguaje de Programación. En forma genérica, podemos decir que, sintácticamente, una EXPRESIÓN es una secuencia de operandos y operadores más el posible uso de paréntesis. Téngase en cuenta que siempre una constante, una variable o la invocación a una función son casos particulares y mínimos de una expresión.

Muchas veces, no es suficiente con definir una BNF para las expresiones, sino que, además, debe haber restricciones que agregan importantes características que la GIC o la BNF no pueden representar adecuadamente. Esto ya lo hemos visto con las expresiones Booleanas de Pascal y también lo veremos, ahora, con las expresiones en ANSI C.

Otro propiedad importante a tener en cuenta es que en ANSI C no existen las constantes booleanas, como TRUE y FALSE en Pascal, y tampoco existe el tipo Boolean. La resolución a este problema es muy simple.

* Ejercicio 48 *
(a) Investigue e informe cómo el ANSI C representa el valor verdadero y el valor falso.
(b) Investigue e informe cuál es el resultado de una expresión booleana en ANSI C. Escriba diversos ejemplos.

A continuación, analizaremos un subconjunto reducido de las expresiones del ANSI C. Se debe tener en cuenta que el ANSI C tiene más de 40 operadores distribuidos en 15 niveles de precedencia. Un análisis de la BNF total para expresiones en ANSI C traería más complicaciones que beneficios; por ello, trabajaremos sobre un subconjunto de 33 reglas que representarán todas las características importantes de la BNF de expresiones en ANSI C. En estas producciones encontraremos operadores

binarios (operan sobre dos operandos), operadores unarios (operan sobre un solo operando) y el único operador ternario (opera sobre tres operandos).

Recordemos que una adecuada BNF de expresiones debe representar correctamente tanto la precedencia (o prioridad) como la asociatividad de los operadores. En el caso de ANSI C, veremos que hay operadores asociativos a izquierda y operadores asociativos a derecha, diferencia que es determinada por el formato de la producción: recursiva a izquierda para el primer caso y recursiva a derecha para el segundo caso.

Introducimos dos definiciones que brinda el MROC y que serán de gran utilidad:

– Objeto: es una región de la memoria de la computadora, compuesta por una secuencia de uno o más bytes consecutivos, que tiene la capacidad de contener la representación de valores. Por ejemplo: una variable es un objeto; una expresión no es un objeto.

– Lvalue (traducido, habitualmente, como ValorL; la "L" proviene de "left", palabra inglesa que significa "izquierda" o "izquierdo"): es una expresión que designa un objeto. Su nombre proviene de la operación de asignación, donde la expresión que se encuentra a la izquierda del operador = debe ser un ValorL. Ejemplos de ValorL: variable a, elemento de un vector v[3], expresión *(p+2); en cambio, una constante y una expresión aritmética no son ValoresL.

* Ejercicio 49 *
Investigue e informe qué significa la expresión *(p+2).

Presentamos, entonces, el siguiente subconjunto de la BNF de expresiones en ANSI C con 17 operadores distribuidos en 11 niveles de precedencia. Analizaremos la BNF en la medida que se va desarrollando.

1 *expresión: expAsignación*
2 *expAsignación: expCondicional*
3 *expUnaria operAsignación expAsignación*
4 *operAsignación:* uno de **= +=**

La producción 1 presenta al axioma, expresión, en una forma simplificada; no consideramos la otra producción para el axioma, con el operador de mínima prioridad (el operador "coma"), para simplificar este análisis.

Las producciones 2 y 3 desarrollan expresión de Asignación, una BNF recursiva a derecha, como se ve en la producción 3, que nos informa que operAsignación (el operador de asignación) es asociativo a derecha y que, además, tiene una prioridad muy baja porque está muy cerca del axioma. La realidad indica que solo el operador "coma" tiene menor prioridad que el operador de asignación.

Por último, la producción 4 muestra dos de los operadores de Asignación existentes: la asignación simple (=) y uno (+=), a modo de representante de los 10 operadores de asignación compuestos que tiene el ANSI C.

5 *expCondicional: expOr*
6 *expOr* **?** *expresión* **:** *expCondicional*

Aquí aparece el único operador ternario, conocido como operador condicional. La producción 6 es recursiva a derecha, por lo que se trata de un operador asociativo a derecha. Si el valor de expOr (expresión "or") es distinto de 0 (esto es: verdadero), entonces el resultado de esta operación será el

valor de expresión; en cambio, si el valor de expOr es 0 (esto es: falso), entonces el resultado de esta operación será el valor de expCondicional.

```
 7 expOr: expAnd
 8        expOr || expAnd
```

La producción 8 define el operador booleano "or" (||) que, como se ve, es asociativo a izquierda.

```
 9 expAnd: expIgualdad
10         expAnd && expIgualdad
```

La producción 10 introduce el operador booleano "and" (&&), que también es asociativo a izquierda.

```
11 expIgualdad: expRelacional
12              expIgualdad == expRelacional
```

La producción 12 incorpora el operador de "igualdad", asociativo a izquierda. En el mismo nivel se encuentra también el operador de "desigualdad" (**!=**).

```
13 expRelacional: expAditiva
14                expRelacional >= expAditiva
```

La producción 14 introduce el operador relacional >=. En el mismo nivel se hallan los operadores **>**, **<** y **<=**, que tienen igual precedencia y son asociativos a izquierda. En el caso de todos los operadores relacionales, si la expresión evaluada es verdadera, el valor de verdad producido será 1, y si es falso, será 0.

```
15 expAditiva: expMultiplicativa
16             expAditiva + expMultiplicativa
```

La producción 16 presenta al operador de suma; también en este nivel de precedencia se encuentra el operador de resta y ambos son asociativos a izquierda, como lo muestra la producción recursiva a izquierda.

```
17 expMultiplicativa: expUnaria
18                    expMultiplicativa * expUnaria
```

La producción 18 presenta al operador de multiplicación; también en este nivel de precedencia hay producciones smilares para los operadores **/** (división) y **%** (operador módulo). La producción es recursiva a izquierda y, por lo tanto, estos tres operadores son asociativos a izquierda.

```
19 expUnaria: expPostfijo
20            ++ expUnaria
21            operUnario expUnaria
22            sizeof (nombreTipo)
23 operUnario: uno de & (dirección) * (puntero) – (signo) ! ("not")
```

Las producciones 20 a 22 introducen varios operadores unarios, aunque no son todos los que ofrece el ANSI C. En la producción 20 se introduce el operador de preincremento, que debe aplicarse a un ValorL modificable. La producción 21 introduce los operadores unarios detallados en la producción 23. Observe que las producciones 20 y 21 son recursivas a derecha. La producción 22 introduce un operador inusual que determina una cantidad de bytes; aquí su operando es un Tipo pero también puede ser una variable.

* Ejercicio 50 *
Investigue e informe, con ejemplos, qué es un ValorL modificable.

```
24 expPostfijo: expPrimaria
25              expPostfijo [ expresión ]
26              expPostfijo ( listaArgumentos_op )
27 listaArgumentos: expAsignación
28                  listaArgumentos , expAsignación
```

En este segmento de producciones incorporamos dos operadores. En la linea 25 se introduce el operador representado por el par de corchetes, utilizado para acceder a un elemento de un arreglo de cualquier dimensión. En la línea 26 se introduce el operador representado por un par de paréntesis, utilizado en la invocación a una función. Como se ve, la lista de Argumentos es opcional, pero nunca lo son los paréntesis.

```
29 expPrimaria: identificador
30              constante
31              literalCadena
32              ( expresión )
```

Las líneas 29 y 30 muestran los elementos básicos que intervienen en la escritura de una expresión; ambos están definidos anteriormente, entre los tokens. La línea 32 presenta una recursividad indirecta que nos lleva al axioma.

```
33 nombreTipo: uno de char int double
```

Finamente, la regla 33 presenta algunos tipos de datos primitivos del ANSI C.

A continuación habrá una serie de Ejemplos y de Ejercicios sobre las Expresiones en ANSI C.

Ejemplo 28
Sea la siguiente derivación:

```
expPostfijo
expPostfijo [expresión]
expPostfijo [expresión][expresión]
. . .
```

* Ejercicio 51 *
Continúe la derivación del ejemplo anterior para obtener la expresión: `matriz12[2+3][4*6]`.

* Ejercicio 52 *
Según la BNF de las Expresiones, ¿cuántas dimensiones puede tener un arreglo?

Ejemplo 29
El operador preincremento **++** debe tener un operando ValorL modificable. Por ejemplo, ++a (siendo a una variable entera) es correcto, pero ++12 es incorrecto.

* Ejercicio 53 *
(a) Investigue e informe el significado del operador preincremento **++** y cómo actúa.
(b) Investigue e informe el operador **&** (dirección) y cómo actúa.
(c) Investigue e informe el operador unario `*` (puntero) y cómo actúa.
(d) Investigue e informe el operador unario `!` (“not”) y cómo actúa.
(e) Investigue e informe el operador unario **sizeof** y cómo actúa.

Si nos atenemos estrictamente a la BNF de las expresiones y no tenemos en cuenta ciertas restricciones que se expresan en lenguaje natural, podemos llegar a conclusiones absurdas.

Ejemplo 30
A nadie se le ocurre pensar que la expresión de asignación `1 = 2` es sintácticamente correcta. Sin embargo . . .

* Ejercicio 54 *
Derive verticalmente la expresión del Ejemplo 30.

Conclusión:

En algunos casos, la BNF describe situaciones que son sintácticamente incorrectas. Esto se debe a que la BNF también es utilizada para la construcción del compilador y ciertas detecciones las hace este programa para mayor eficiencia, como veremos en el Volumen 2.

En consecuencia, es indispensable leer las restricciones que figuran en los manuales, acompañando a las descripciones en BNF. Y si no son claras o suficientes, uno mismo debe agregarlas.

Por caso, cuando el ANSI C describe los operadores de asignación, agrega una restricción muy poderosa que dice: "Un operador de asignación debe tener un ValorL modificable como su operando izquierdo." Obviamente, una constante no es un ValorL modificable y, por lo tanto, la expresión de asignación del Ejemplo 30 es incorrecta (aunque se pueda derivar).

* Ejercicio 55 *
Construya una Tabla de Derivación para la expresión `a = b = c = 2`.

* Ejercicio 56 *
(a) Según la BNF de expresiones, ¿es `2 <= 5 <= 4` sintácticamente correcta? Si responde que no, justifique su respuesta.
(b) Si responde que sí, escriba la Tabla de Derivación para esa expresión y luego escriba la Tabla de Evaluación.

* Ejercicio 57 *
(a) Según la BNF de expresiones, ¿es `2 && 3 || 4 && 5 || 6` sintácticamente correcta? Si responde que no, justifique su respuesta.
(b) Si responde que sí, escriba la Tabla de Derivación para esa expresión y luego escriba la Tabla de Evaluación.

* Ejercicio 58 *
Compare la precedencia, la asociatividad y la cantidad de operadores del lenguaje Pascal con los de ANSI C.

3.6.2.2.2 DECLARACIONES Y DEFINICIONES

Una definición es como una declaración, pero provoca una reserva de memoria para el objeto o la función definidos.

Ejemplo 31
Recordemos la parte de ANSI C del programa del Ejemplo 25:

```
 3  int Mayor (int, int);
 4  int main (void) { /* comienza main */
 5     int n1, n2, max;
    . . .
11  } /* fin main */
12  int Mayor (int a, int b) { /* comienza Mayor */
    . . .
17  } /* fin Mayor */
```

En este programa, el prototipo de la linea 3 es una declaración de la función Mayor, mientras que en las líneas 12 a 17 se define esta función. Asimismo, en la línea 5 se definen tres variables.

Ejemplo 32
Otra situación se presenta con la declaración y la definición de una estructura en ANSI C:

```
. . .
typedef struct {
           int a;
           char vec[20+1];
           int mat[4][18];
        } REGISTRO;
. . .
int main (void) {
   REGISTRO x;
   . . .
}
```

Primero se declara REGISTRO como una estructura con tres campos. Luego, en la función main, se define una variable (un objeto) x de tipo REGISTRO.

* Ejercicio 59 *
Investigue e informe, acompañado de ejemplos:
(a) qué es typedef, y
(b) qué es struct.

A continuación, veamos un subconjunto elemental de la BNF para las declaraciones y definiciones de variable simples:

declaVarSimples: tipoDato listaVarSimples ;
tipoDato: uno de **`int double char`**
listaVarSimples: unaVarSimple
listaVarSimples , unaVarSimple
unaVarSimple: variable inicial$_{op}$
variable: identificador
inicial: **=** *constante*
identificador: . . .
constante: . . .

Ejemplo 33

```
int a, b = 10, c, d = 4;
```

* Ejercicio 60 *
Derive la definición del Ejemplo 33 partiendo de la BNF dada.

* Ejercicio 61 *
Escriba una BNF que represente la declaración de un typedef struct, como se mostró en el Ejemplo 32, pero que solo contenga variables simples.

Debemos diferenciar dos conceptos:
(1) DERIVABLE DE UNA BNF DADA, y
(2) SINTÁCTICAMENTE CORRECTO.

* Ejercicio 62 *
(a) Sea el siguiente fragmento de una función:

```
void XX (void) {
   int a;
   double a;
   . . .
}
```

Investigue los dos conceptos incorporados e indique cuál se aplica en este caso. Justifique su respuesta.
(b) ¿A cuál de los dos conceptos responde la expresión 12 = 23+6?
(c) ¿Y la expresión 4/3?

3.6.2.2.3 SENTENCIAS

Como se sabe, las sentencias especifican las acciones que llevará a cabo la computadora en tiempo de ejecución. Seguidamente, describiremos y analizaremos un subconjunto de las sentencias en ANSI C.

sentencia: una de *sentCompuesta sentExpresión sentSelección sentIteración sentSalto*

El axioma, sentencia, presenta cinco tipos de sentencias que analizaremos a continuación.

sentCompuesta: **{** *listaDeclaraciones*$_{op}$ *listaSentencias*$_{op}$ **}**
listaDeclaraciones: declaración
listaDeclaraciones declaración
listaSentencias: sentencia
listaSentencias sentencia

La sentencia compuesta (o bloque) queda definida por las llaves y lo que ellas encierran. Note que las declaraciones (y/o definiciones), si existen, van al comienzo de una sentencia compuesta.

* Ejercicio 63 *
(a) Una sentencia compuesta, ¿puede ser vacía? Justifique su respuesta.
(b) ¿Qué es **{** `/* Hola */` **}** ?

sentExpresión: expresión$_{op}$ *;*

Una sentencia expresión es una expresión que termina con un "punto y coma".

* Ejercicio 64 *
Investigue e informe qué es la sentencia nula.

* Ejercicio 65 *
Investigue e informe si existe la "sentencia expresión" en Pascal.

* Ejercicio 66 *
Sea la función ANSI C:
```
void XX (void) { 14; ; }
```
(a) ¿Es derivable de la BNF?;
(b) ¿Es sintácticamente correcta?
(c) ¿Cuántas sentencias componen su cuerpo?

* Ejercicio 67 *
Investigue e informe sobre la siguiente sentencia compuesta:
```
{ 22; int a; a = 14; }
```

sentSelección: **`if (`** *expresión* **`)`** *sentencia*
`if (` *expresión* **`)`** *sentencia* **`else`** *sentencia*
`switch (` *expresión* **`)`** *sentencia*

* Ejercicio 68*
Investigue e informe sobre la sentencia switch. Escriba ejemplos de su uso.

* Ejercicio 69 *
Compare la sintaxis de la sentencia if de ANSI C con la sintaxis de la sentencia if de Pascal.

sentIteración: **`while (`** *expresión* **`)`** *sentencia*
`do` *sentencia* **`while (`** *expresión* **`) ;`**
`for (` *expresión*$_{op}$ **`;`** *expresión*$_{op}$ **`;`** *expresión*$_{op}$ **`)`** *sentencia*

* Ejercicio 70 *
Investigue e informe sobre la sentencia do while. Escriba ejemplos de su uso.

* Ejercicio 71 *
(a) Investigue e informe sobre la sentencia for. Escriba ejemplos de su uso.
(b) ¿Qué significa `for(;;);` ?

sentSalto: **`return`** *expresión*$_{op}$ **`;`**

* Ejercicio 72 *
Investigue e informe sobre la sentencia return y escriba varios ejemplos de su uso.

* Ejercicio 73 *
(b) ¿Qué ocurre si una sentencia compuesta solo tiene declaraciones? Informe.

* Ejercicio 74 *
Sea la sentencia: for (; –4;) ;
(a) ¿Es derivable de la BNF de ANSI C? Justifique su respuesta.

(b) ¿Es sintácticamente correcta? Justifique su respuesta.
(c) Si lo es, describa qué hace en ejecución.

* Ejercicio 75 *
¿Qué acción realiza la sentencia do 22; while (3); ?

* Ejercicio 76 *
Sea la sentencia return a = 8;
(a) ¿Es derivable de la BNF de ANSI C? Justifique su respuesta.
(b) ¿Es sintácticamente correcta?
(c) Si lo es, ¿qué acción lleva a cabo?

3.7 OTROS USOS DE BNF

Si bien el metalenguaje BNF fue diseñado para describir la sintaxis de los LPs, también puede ser aplicado a otras áreas.

Ejemplo 34
Necesitamos describir un listado de los alumnos de la UTN FRBA en los que consten los siguientes datos: (a) nombre y apellido; (b) legajo; (c) código de carrera. Podemos hacerlo de la siguiente manera:

```
listadoUTN: alumno
            listadoUTN alumno
alumno: nombreApellido legajo códigoCarrera
nombreApellido: apellido , nombre
. . .
```

* Ejercicio 77 *
(a) Sea un archivo de texto que contiene secuencias de uno o más dígitos decimales, cada una terminada con un #. Defina su contenido en una BNF completa.
(b) ¿Cómo cambia la BNF si la secuencia de dígitos puede ser vacía?

* Ejercicio 78 *
Se diseña un LP con las palabras reservadas en castellano. Este LP tiene una única sentencia de selección que condensa los dos tipos de sentencias que aparecen habitualmente. Esta sentencia de selección comienza con la palabra reservada SI y termina con la palabra reservada FIN. Entre estos delimitadores puede haber un número indeterminado de constructos, cada uno de los cuales comienza con una expresión booleana seguida de **:** seguida de una secuencia compuesta encerrada entre "llaves". Antes de FIN, siempre habrá un constructo que comienza con la palabra reservada OTRO seguida de **:** seguida de una secuencia compuesta encerrada entre "llaves".
(a) Escriba las hipótesis que considere necesarias para definir estrictamente esta sentencia de selección, (b) Escriba varios ejemplos de esta sentencia; y c) Escriba la sintaxis en BNF de esta sentencia de selección.

4 SEMÁNTICA

4.1 DEFINICIÓN DE UN LENGUAJE DE PROGRAMACIÓN

La definición de un LP es un documento muy importante que debe cumplir varias funciones:

(1) Debe servir como un manual de referencia para los programadores en ese LP, por lo que debe describir con precisión, y sin ambigüedades, la sintaxis del LP;

(2) Debe especificar la semántica y las restricciones de cada constructo del LP;

(3) Debe ser el punto de partida para quienes desarrollen compiladores para ese LP.

Como ya hemos visto en el capítulo anterior, la sintaxis de un LP se define con precisión mediante las reglas que se representan en BNF. En realidad, la experiencia adquirida en el Capítulo 3 nos indica que:

(1) Las BNFs que definen a los tokens, es decir, a los LRs que forman parte del LP, son absolutamente precisas;

(2) Las BNFs que definen a las expresiones pueden "fallar" en algunos casos, como ya hemos analizado tanto para el lenguaje Pascal como para el ANSI C. Para remediar estas situaciones especiales se requiere leer las restricciones en lenguaje natural que acompañan a las BNFs.

Ejemplo 1
En el capítulo anterior hemos visto que, a partir de la BNF de Expresiones en ANSI C, podíamos derivar la expresión `1 = 2`. Sin embargo, sabemos que no podemos realizar una asignación a una constante; por ello, se agrega la restricción: El operador de asignación requiere que el operando izquierdo sea un ValorL.

La especificación de la semántica de un LP es más compleja. Si bien existen varios métodos formales para describir la semántica de un LP, muchas definiciones de LPs siguen utilizando el lenguaje natural para realizar esta tarea. En la próxima sección, veremos cómo el MROC describe la semántica en lenguaje natural (traducida al castellano).

4.2 EL ANSI C Y SU SEMÁNTICA

Ante todo, tengamos en cuenta dos definiciones que provee el MROC:

OBJETO: Es una región del almacenamiento de datos (memoria) cuyo contenido puede representar valores durante la ejecución del programa; por ejemplo: una variable.

TIPO: Brinda el significado de un valor almacenado en un objeto o retornado por una función; por ejemplo, al declarar una variable se especifica cuál es su tipo y, en consecuencia, qué valores puede contener.

Ejemplo 2
– La definición `double a;` indica que la variable `a` es de tipo `double`.
– El prototipo (declaración de una función) `int X (void);` señala que la función `X` retorna un valor de tipo `int`.

Siguiendo con el concepto de "tipo", el MROC agrega que el tipo char con signo y sin signo, los tipos enteros (short int, int, long int) con signo y sin signo, y los tipos reales (float, double, long double) forman el conjunto de *tipos básicos*. Por otro lado, el tipo void equivale a un conjunto vacío de valores.

* Ejercicio 1 *
Investigue e informe:
(a) qué es un *tipo arreglo* y qué describe;
(b) qué es un *tipo puntero*.
Brinde diversos ejemplos de ambos casos.

Otra definición muy importante, que ya hemos visto en el capítulo 3, es la del término ValorL: es una expresión que designa a un objeto; por ejemplo: un identificador que es el nombre de una variable simple, un elemento de un arreglo, etc.

* Ejercicio 2 *
¿Por qué un elemento de un arreglo es una expresión más compleja que una simple variable?

A continuación veremos algunas semánticas descriptas en el MROC. Evitaremos analizar aquellas semánticas que son obvias, como, por ejemplo: *en los dígitos que forman una constante entera, el primero de la izquierda es el más significativo*.

4.2.1 IDENTIFICADOR

Un identificador denota un objeto, una función y otras entidades.

* Ejercicio 3 *
Investigue e informe qué otras entidades pueden ser denotadas por identificadores.

4.2.2 SEMÁNTICA DE LAS CONSTANTES

Cada constante tiene un tipo, determinado por su forma y su valor.

4.2.2.1 CONSTANTES ENTERAS

Dada una lista de tipos predeterminada, el tipo de una constante entera es el primero en el que su valor puede ser representado.

Ejemplo 3
Para una constante entera decimal sin sufijo (vea la BNF en el capítulo 3), la lista predeterminada de tipos es: int, long int, unsigned long int.

* Ejercicio 4 *
Investigue e informe qué es cada uno de estos tipos.

Ejemplo 4
Supongamos que una constante de tipo int se representa en dos bytes. Esto significa que su rango de valores estará entre –32768 y +32767. Por lo tanto, la constante 14 es de tipo int; en cambio, la constante –32769, que no puede ser representada en los dos bytes del tipo int, es una constante de tipo long int.

* Ejercicio 5 *
Para facilitar el análisis en este ejercicio, supongamos que los valores de tipo int se representan en un byte y que los valores de tipo long int se representan en dos bytes. Entonces:

(a) Investigue el rango de valores enteros que se puede representar en un byte, con signo.

(b) Escriba el tipo de cada una de las siguientes constantes enteras: 0, +222, –128, 30765, –1400, 68432, 1234567, –1234567.

Nota 1
Si bien la constante entera 0 está definida como octal, su valor coincide con el valor 0 en base 10. Por lo tanto, semánticamente afirmamos que la constante entera 0 es tanto octal como decimal.

4.2.2.2 CONSTANTES REALES

Para analizar esta semántica con más cuidado, primero reiteremos la BNF completa de las constantes reales:

constanteReal: constanteFraccionaria parteExponente$_{op}$ *sufijoReal*$_{op}$
secuenciaDígitos parteExponente sufijoReal$_{op}$
constanteFraccionaria: secuenciaDígitos$_{op}$ **.** *secuenciaDígitos*
secuenciaDígitos **.**
parteExponente: operadorE signo$_{op}$ *secuenciaDígitos*
operadorE: uno de **e E**
signo: uno de **+ –**
secuenciaDígitos: dígito
secuenciaDígitos dígito
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
sufijoReal: uno de **f F l L**

Las constantes reales pueden ser:
(a) constantes reales en Punto Fijo; y
(b) constantes reales en Punto Flotante.
Las últimas están constituidas por el par (mantisa, exponente).

Ejemplo 5
Constantes reales en Punto Fijo: 4.32 236. .08
Constantes reales en Punto Flotante: 4.32E6 28E–4

En general, una constante real tiene una *parte significativa* que puede estar seguida de una *parte exponente*, y de un *sufijo real* que especifica su tipo.

La *parte significativa* es interpretada como un número racional en base 10; la *parte exponente* es interpretada como un entero en base 10. La representación de un número real no siempre es exacta, sino la representación más cercana a la exacta. ¿Por qué?

Ejemplo 6
El número real 0.1, tan simple como se lo ve, no tiene una representación exacta.

* Ejercicio 6 *
Investigue e informe porqué el número 0.1 no tiene una representación exacta.

Una constante real sin sufijo es de tipo double. Si el sufijo es **f** o **F**, entonces la constante real es de tipo float; y si el sufijo es **l** o **L**, entonces su tipo es long double. La diferencia entre estos tres tipos de constantes reales radica en el rango y en la precisión: las constantes float son las que tienen menor rango y menor precisión; las constantes long double son las que tienen mayor rango y mayor precisión.

* Ejercicio 7 *
Investigue e informe porqué se producen estas diferencias en el rango y en la precisión.

4.2.2.3 CONSTANTES CARÁCTER

Una constante carácter se delimitada por apóstrofos, como 'a', y su valor es de tipo int. Una constante carácter es, en realidad, la representación numérica de ese carácter en una tabla de caracteres como la Tabla ASCII.

Existen ciertas constantes carácter que requieren un formato de escritura especial, que comienza con una "barra invertida". He aquí algunos casos:

Ejemplo 7
\' corresponde al carácter "apóstrofo". Como éste es el delimitador, se lo diferencia de esta forma.
\\ corresponde al carácter "barra invertida".
\n corresponde al carácter conocido como "nueva línea" ("new line"). Hace visible un carácter no imprimible.
\0 representa el carácter nulo y es utilizado para indicar la terminación de una cadena de caracteres.

* Ejercicio 8 *
(a) ¿Es válida la expresión 4 + 'b' – 'a'? Justifique su respuesta. Si lo es, ¿cuál es su valor?
(b) ¿Es válida la expresión 4 + '\0'? Justifique su respuesta. Si lo es, ¿cuál es su valor?

4.2.2.4 LITERAL CADENA

Un literal cadena es una secuencia de cero o más caracteres delimitada por comillas ("). Como las comillas son delimitadores, debe escribírselas de otra manera cuando se las utiliza como carácter de una cadena, como ya hemos visto en el capítulo 3. Si no lo recuerda, repáselo.

Ejemplo 8
"" representa la cadena vacía.
" " representa la cadena cuyo único carácter es un espacio.
"\"Hola\"" representa la cadena "Hola".

* Ejercicio 9 *
Escriba las longitudes de las tres cadenas del Ejemplo 8.

4.2.3 SEMÁNTICA DE LOS OPERADORES

En su aplicación más común, un operador especifica la realización de una operación que producirá un valor. Un operando es una entidad sobre la que actúa un operador.

* Ejercicio 10 *
(a) ¿Un operador produce un solo valor o puede producir un conjunto de valores. Justifique su respuesta.
(b) Describa operadores, operandos y los valores que producen.

4.2.4 SEMÁNTICA DE LOS CARACTERES DE PUNTUACIÓN

Un carácter de puntuación es un símbolo que tiene un significado semántico determinado, no especifica una operación y no produce un valor.

Dependiendo del contexto, el mismo símbolo puede representar un carácter de puntuación y un operador (o parte de un operador).

* Ejercicio 11 *
Investigue e informe, con ejemplos, sobre cuatro símbolos que representen caracteres de puntuación y operadores.

4.2.5 SEMÁNTICA DE LAS EXPRESIONES

En ANSI C, una expresión puede tener varias aplicaciones; la más común es que especifique el cálculo de un valor. Analizaremos ahora la semántica de un subconjunto muy pequeño de expresiones posibles.

expresiónPrimaria: uno de *identificador constante*

Semántica: Un identificador es una expresión primaria si ha sido declarado para designar un objeto, en cuyo caso es un ValorL. Una constante es una expresión primaria; su tipo depende de su forma y de su valor, como ya se ha visto.

Semántica de un objeto arreglo. En ANSI C, un objeto arreglo puede tener muchas sintaxis, pero la más común, y también la más empleada, es la que consiste en un identificador y expresiones entre corchetes.

Ejemplo 9
Sea int vec[10];
(a) vec, el nombre del arreglo, es una dirección constante;
(b) vec[3] equivale a *(vec+3).

Ejemplo 10
Sea el objeto arreglo definido por la declaración:

```
int matriz [3][5];
```

Esta declaración indica que matriz es un arreglo bi-dimensional que contiene elementos de tipo int, y que se compone de 3 filas y de 5 columnas. Todo arreglo comienza en el elemento con subíndice 0; por lo tanto, en este caso el primer elemento es `matriz[0][0]`.

Semántica de las expresiones Booleanas. Para que una expresión sea Booleana, al evaluarla su resultado debe representar Verdadero o Falso (que en ANSI C se codifican como 1 o 0). Por ello, la expresión debe tener, como mínimo, un operador de relación o un operador Booleano (lógico), o ambos.

Ejemplo 11
Las siguientes son expresiones Booleanas derivables de la BNF del ANSI C (¡verifíquelo!):
a > 3 a > 3 && b < a+1 2 < 5 < 4 2 && 3 !(4+18*236)

* Ejercicio 12 *
Investigue e informe si la expresión 2 < 5 < 4 es semánticamente correcta.

* Ejercicio 13 *
Suponiendo que la variable a contiene el valor 6 y que la variable b contiene el valor 2, indique el resultado para cada uno de los cinco casos del Ejemplo 11. Justifique sus respuestas.

Semántica de la expresión de asignación. El operador de asignación tiene asociatividad a derecha. El valor del operando derecho es convertido al tipo del operando izquierdo, que debe ser un ValorL. Además, este será el tipo de la expresión de asignación.

Ejemplo 12
Sea int a, b, c; Entonces:
a = b = c = 4 es sintácticamente correcta y es equivalente a (a = (b = (c = 4))). Su efecto es asignar el valor 4 a las tres variables.

* Ejercicio 14 *
Sean las declaraciones del ejemplo anterior.
(a) ¿Por qué c = b = a = 4 tiene el mismo efecto que la asignación del ejemplo anterior?
(b) ¿Por qué a = b+2 = c = 4 es incorrecta?

* Ejercicio 15 *
Explique las diferencias que hay entre las sentencias de asignación en Pascal y las expresiones de asignación en ANSI C.

4.2.6 SEMÁNTICA DE LAS DECLARACIONES Y DEFINICIONES

Las declaraciones y las definiciones básicas especifican la interpretación de un conjunto de identificadores.

* Ejercicio 16 *
Investigue la siguiente declaración:

```
typedef struct {
   int a;
   char b[4][7];
} XX;
```

Describa su semántica.

* Ejercicio 17 *
Investigue la siguiente definición:

```
int a = 10;
```

Describa su semántica.

* Ejercicio 18 *
Investigue la siguiente definición:

```
int mat [2][3] = {{1,2,3},
                  {4,5,6}};
```

Describa su semántica.

4.2.7 SEMÁNTICA DE LAS SENTENCIAS

Una sentencia especifica la realización de una acción.

Semántica de la Sentencia Compuesta. Una sentencia compuesta, también llamada bloque, permite que un grupo de sentencias sea tratado como una unidad. El bloque puede tener su propio conjunto de declaraciones.

* Ejercicio 19 *
Investigue sobre las sentencias compuestas y describa la semántica de cierto bloque con declaraciones y sentencias.

sentenciaExpresión: expresión$_{op}$ *;*

* Ejercicio 20 *
(a) Escriba un ejemplo útil de sentenciaExpresión y describa su semántica.
(b) ¿Cuál es la diferencia entre “expresión de asignación” y “sentencia de asignación” en ANSI C?
(c) Investigue e informe sobre la utilización de la sentencia nula. Escriba varios ejemplos.

Formato de la Condición en las Sentencias de Selección e Iterativas. Dado que en ANSI C no existe el tipo Booleano, cualquier expresión –no necesariamente una expresión Booleana– puede utilizarse como condición de las sentencias mencionadas. Si el valor de la expresión es distinto de cero, la condición es verdadera; si el valor es cero, la condición es falsa.

* Ejercicio 21 *
Investigue y escriba ejemplos de lo descripto en el párrafo anterior, para las sentencias if, while, for. Explique la semántica de cada ejemplo escrito.

* Ejercicio 22 *
Investigue e informe cuál es la diferencia semántica entre el prototipo de una función y la definición de esa misma función.

* Ejercicio 23 *
Explique la semántica del constructo cuyo BNF ha desarrollado en el Ejercicio 78 del capítulo 3.

* Ejercicio 24 *
Defina un LP para un campo de aplicaciones particular, con la característica de que este LP esté formado por un grupo de comandos que son LRs. Describa su sintaxis en BNF, su semántica en lenguaje natural y presente varios ejemplos de su uso.

5 LENGUAJES REGULARES Y EXPRESIONES REGULARES

Un LENGUAJE FORMAL es un conjunto de palabras que solo tienen una sintaxis, no tienen semántica. Los Lenguajes Formales más simples son los LENGUAJES REGULARES.

➔ Los Lenguajes Regulares tienen gran importancia en el diseño de los LENGUAJES DE PROGRAMACIÓN, ya que los componentes básicos de un LP, los "tokens" – como los identificadores, las palabras reservadas, las constantes, los literales cadena, los operadores y los caracteres de puntuación – constituyen diferentes LENGUAJES REGULARES.

En el capítulo 2 hemos visto la Jerarquía de Chomsky y sus diferentes tipos de gramáticas. Una de ellas, la Gramática Regular, solo tiene la capacidad de GENERAR Lenguajes Regulares. Entonces:

➔ Un lenguaje es Regular si puede ser generado por una Gramática Regular (GR).

A continuación veremos algunas particularidades de los Lenguajes Regulares, para luego dedicarnos a las EXPRESIONES REGULARES.

5.1 DETERMINACIÓN DE LENGUAJES REGULARES

Si un Lenguaje Formal es FINITO, entonces SIEMPRE es un LENGUAJE REGULAR. A partir de este momento, utilizaremos la sigla LR para referirnos a un Lenguaje Regular.

Ejemplo 1
$L_1 = \{a^n b^n / 1 \leq n \leq 3\}$ es un lenguaje finito constituido por tres palabras: ab, aabb, aaabbb. Este lenguaje es un LR porque puede ser generado por una GR con las siguientes producciones:

```
S -> aT
T -> b | aQ
Q -> bR | aM
R -> b
M -> bP
P -> bR
```

* Ejercicio 1 *
(a) Verifique, por derivación vertical, que la GR del Ejemplo 1 es correcta.
(b) Escriba la definición formal de otra GR que genere el LR del Ejemplo 1.

Por otro lado, y como hemos visto en la Jerarquía de Chomsky, hay lenguajes infinitos que son LRs y otros que no lo son.

Ejemplo 2
El lenguaje "Todas las palabras formadas por una o más aes" es un LR infinito. Este lenguaje, que se puede describir por comprensión como $L_2 = \{a^n / n \geq 1\}$, es un LR porque podemos diseñar una GR que lo genere, con las siguientes producciones:

```
S -> aS | a
```

* Ejercicio 2 *
(a) Verifique, por derivación vertical, que la GR del Ejemplo 2 es correcta.
(b) Escriba la definición formal de una GR que genere el LR $L_3 = \{a^n / n \geq 0\}$.

Ejemplo 3
Sea el lenguaje infinito $L_4 = \{a^n b^t / n \geq 1, t \geq 1\}$. Este lenguaje es un LR porque puede ser generado por una GR con estas producciones:

```
S -> aS | aT
T -> bT | b
```

* Ejercicio 3 *
(a) Verifique, por derivación vertical, que la GR del Ejemplo 3 es correcta.
(b) Escriba la definición formal de una GR que genere el LR $L_5 = \{a^n b^t / n \geq 0, t \geq 0\}$.

Ejemplo 4
La GR es un caso particular de la GIC. Por lo tanto, los LRs también pueden ser generado por GICs. Sea el lenguaje $L_6 = \{(ab)^n (cde)^t a^z / n \geq 1, t \geq 1, z \geq 0\}$. Este lenguaje infinito es un LR que puede ser generado por una GIC con las siguientes producciones:

```
S -> abS | abT
T -> cdeT | cdeR
R -> aR | ε
```

* Ejercicio 4 *
(a) Verifique, por derivación vertical, que la GIC del Ejemplo 4 genera el LR dado en el mismo ejemplo.
(b) Escriba la definición formal de una GR que genere el LR del Ejemplo 4.
(c) Verifique, por derivación vertical, que la GR que diseñó es correcta.

Ejemplo 5
El lenguaje "Todos los números binarios que comienzan con una cantidad impar de 1s que es seguida por un 0 como último carácter" es un LR infinito. Por comprensión, este LR se puede escribir de la siguiente manera: $L_7 = \{1^{2n+1}0 / n \geq 0\}$. Las primeras tres palabras de este lenguaje son: 10, 1110, 111110.

* Ejercicio 5 *
(a) Escriba la definición formal de una GR que genere el LR del Ejemplo 5.
(b) Verifique, por derivación vertical, que la GR que diseñó es correcta.

En la próxima sección desarrollaremos un concepto muy importante, vinculado con los LRs: las EXPRESIONES REGULARES. El conocimiento de este tipo de expresiones es esencial para comprender mejor qué Lenguajes Formales infinitos son Regulares y cuáles no lo son. Además, tienen aplicaciones prácticas que también presentaremos.

5.2 LAS EXPRESIONES REGULARES

La forma más precisa de REPRESENTAR a los LRs es mediante las llamadas EXPRESIONES REGULARES. Utilizaremos la sigla ER para referirnos a una Expresión Regular.

Una ER es una expresión que describe un conjunto de cadenas: todas aquellas que son palabras del LR que la ER representa.

Las ERs se construyen utilizando los caracteres del alfabeto sobre el cual se define el LR, el símbolo ε (que representa la ausencia de cualquier carácter), ciertos operadores especiales que veremos a continuación, y, eventualmente, haciendo uso de paréntesis para modificar las prioridades de estos operadores, tal como ocurre también con las expresiones aritméticas.

5.2.1 EXPRESIONES REGULARES PARA LENGUAJES REGULARES FINITOS

Una ER para un LR finito se obtiene mediante la utilización de los caracteres del alfabeto, el símbolo ε, y los operadores: concatenación (·), que ordena la ubicación de ciertos caracteres, y unión (+), también llamado operador "ó" u operador "más", que representa una elección entre caracteres o grupos de caracteres. El operador "concatenación" tiene mayor prioridad que el operador "unión".

Ejemplo 6
La ER a representa al LR que contiene solamente la palabra a, es decir: L = {a}.

➔ Observe que, ahora, un carácter puede representar tres elementos distintos: a) un símbolo de un alfabeto; b) una palabra de longitud 1; o c) una ER. Esta ambigüedad es resuelta por el contexto en el cual se encuentra el carácter en cuestión en una situación determinada.

Ejemplo 7
La ER ε representa al LR que solo contiene la palabra vacía: L = {ε}.

Ejemplo 8
La ER a·b (habitualmente se escribe ab y se lee "a concatenado con b" o, simplemente, "ab") representa el LR de una sola palabra, L = {ab}.

➔ En general, la concatenación no es conmutativa, contrariamente a lo que ocurre con la multiplicación en las expresiones aritméticas. Por ello, en el área de ls ERs se puede hablar de "concatenación a izquierda" y de "concatenación a derecha".

Ejemplo 9
Por lo enfatizado anteriormente, las ERs ab y ba representan dos lenguajes distintos.

* Ejercicio 6 *
Describa dos casos en los que la concatenación sí puede ser conmutativa.

Ejemplo 10
La ER a+ba, que se lee "a unión ba" o bien "a ó ba" o bien "a más ba", describe la posibilidad de elegir entre dos palabras: la palabra a o la palabra ba. En consecuencia, esta ER está asociada a la unión de dos LRs: {a} ∪ {ba} o, más simplemente, al LR con dos palabras: {a, ba}.

➔ La operación de "unión" sí es conmutativa. Por consiguiente, las ERs a+ba y ba+a representan el mismo LR.

Nota 1
El operador "unión" (+) también se puede indicar con el símbolo | ("barra vertical"). En consecuencia: a+ba y a|ba son dos formas de representar la misma ER.

Ejemplo 11
La ER ab+aa+ε, que también se puede escribir ab|aa|ε, representa al LR que contiene tres palabras: la palabra vacía, la palabra ab y la palabra aa; es decir: L = {ε, ab, aa}.

* Ejercicio 7 *
Escriba una ER que represente al LR del Ejemplo 1: $L_1 = \{a^n b^n / 1 \leq n \leq 3\}$.

➔ Como se ha dicho anteriormente, el operador "concatenación" tiene *mayor prioridad* que el operador "unión", como ocurre con la multiplicación y la suma en el ámbito de las expresiones aritméticas. Por lo tanto, en caso de tener que modificar estas prioridades se deben utilizar paréntesis.

Ejemplo 12
La ER aa+b es diferente a la ER a(a+b). La primera ER representa al LR {aa, b}, mientras que la segunda ER, a(a+b) = aa+ab, representa al LR {aa, ab}.

➔ La *factorización* en el ámbito de las ERs tiene alguna diferencia con la que se puede hacer entre las expresiones aritméticas. Ello se debe a que, como ya se informó, la "concatenación" no es conmutativa. En consecuencia, debemos hablar de "factorización a izquierda" y de "factorización a derecha".

Ejemplo 13
La ER abb+aab tiene el primer carácter a y el último carácter b que son comunes a ambas palabras. Por lo tanto: se puede "factorizar a izquierda", obteniendo la ER a(bb+ab), o se puede "factorizar a derecha", obteniendo la ER (ab+aa)b, o se puede "factorizar a izquierda y a derecha", obteniendo, así, la ER a(b+a)b.

* Ejercicio 8 *
Describa, mediante una frase sin ambigüedades, el LR representado por la ER a(b+a)b.

Ejemplo 14
La ER ba+ab no se puede factorizar (ni "a izquierda" ni "a derecha").

* Ejercicio 9 *
Sea la ER abbaacc+abbbac+abbacccacc. Reescriba esta ER, factorizándola con la mayor cantidad de caracteres comunes a los tres términos tanto a la izquierda como a la derecha.

➔ Definición: Dos ERs son EQUIVALENTES si representan el mismo LR.

Ejemplo 15
Las ERs a+b y b+a son equivalentes porque ambas representan al LR {a, b}. En cambio, las ERs a(a+b) y (a+b)a no son equivalentes porque la "concatenación" no es conmutativa y, por ende, representan a diferentes LRs: la primera ER representa al LR {aa, ab}, mientras que la segunda ER representa al LR {aa, ba}.

Ejemplo 16
El lenguaje "Todas las palabras con dos caracteres sobre el alfabeto {a, b}" puede ser representado por las siguientes ERs equivalentes: aa+ab+ba+bb = a(a+b)+b(a+b) = (a+b)(a+b), entre otras.

* Ejercicio 10 *
Verifique que la afirmación del Ejemplo 16 es correcta.

* Ejercicio 11 *
Determine si las siguientes ERs son equivalentes: aa+ab+ba, a(a+b)+ba y ab+ba+aa. Justifique su respuesta.

5.2.1.1 EL OPERADOR POTENCIA

El operador potencia se utiliza para simplificar la escritura, la lectura y la comprensión de ciertas ERs, como veremos en los próximos ejemplos.

Ejemplo 17
La ER $\varepsilon+(a+b)^3$ representa al LR: "La palabra vacía y todas las palabras de tres caracteres sobre el alfabeto {a, b}", es decir:

$$(a+b)^3 = (a+b)(a+b)(a+b) = aaa+aab+aba+abb+baa+bab+bba+bbb.$$

Nota 2
Si bien el operador POTENCIA no es un operador "oficial" para la construcción de ERs, sí aparecerá en las extensiones de las ERs, como veremos más adelante.

Ahora disponemos de tres operadores para crear ERs que representen a LRs finitos:
– El operador + (ó), que permite una elección y tiene la menor prioridad de estos tres operadores;
– El operador de concatenación, con una prioridad intermedia;
– El operador potencia, de máxima prioridad; como sucede, habitualmente, con los operadores unarios (es decir: los que operan sobre un único operando).

Ejemplo 18
El lenguaje "Todas las palabras de longitud 29 sobre el alfabeto {a, b} que comienzan con 25 aes" se puede representar, fácilmente, utilizando el operador "potencia" en la siguiente ER: $a^{25}(a+b)^4$.

* Ejercicio 12 *
Escriba una ER que represente al LR: "Todas las palabras sobre el alfabeto {a, b} que tienen longitud 100 y terminan con 28 bes". Utilice el operador "potencia".

* Ejercicio 13 *
Escriba una ER que represente al LR: "Todas las palabras sobre el alfabeto {a, b, c} que comienzan con 16 aes, terminan con 16 aes y tienen una longitud total de 1200 caracteres, o comienzan con 342 bes y terminan con 100 repeticiones de cualquiera de los caracteres del alfabeto (pero siempre uno solo de ellos).

Observación: No todos los LRs finitos se representan mejor con ERs. Preste atención al siguiente ejemplo:

Ejemplo 19
Sea el lenguaje $L = \{a^n / 1 \leq n \leq 1000\}$. Este es un LR finito y, por lo tanto, puede ser representado por una ER. Dado que el lenguaje L tiene 1000 palabras, la ER que lo representa deberá tener 1000 términos, es decir: $a + aa + aaa + \ldots + a^{999} + a^{1000}$, pero "*puntos suspensivos*" no es un operador ni una codificación válidos. Deberíamos "inventar" un nuevo operador "potencia múltiple", como $a^{1..1000}$. En consecuencia, como este operador aun no existe, para este LR es mejor su representación por comprensión.

* Ejercicio 14 *
Escriba una ER que represente al LR: "La palabra vacía y todas las palabras sobre {a, b, c} de longitud 45, que comienzan con aa o con bb, y que terminan con aa o con bb".

5.2.2 EXPRESIONES REGULARES PARA LENGUAJES REGULARES INFINITOS

En esta sección analizamos a las ERs más útiles: las que representan a los LRs infinitos. La característica fundamental de estos lenguajes es que tienen, al menos, un carácter o un grupo de caracteres que se repite un número indeterminado de veces, como ya hemos visto en los ejemplos 2, 3 y 4 de este capítulo (¡revéalos!).

Para representar estas repeticiones "indeterminadas", incorporamos el operador unario que fuera presentado en el primer capítulo y que se denomina "clausura de Kleene" o "estrella de Kleene", o, más simplemente, "operador estrella". Este operador se representa con un asterisco (*) colocado como supraíndice de un carácter o de una expresión, y tiene mayor prioridad que los operadores "concatenación" y "unión".

➔ El operador clausura de Kleene (se pronuncia *klaini*) o "estrella de Kleene" representa a la palabra vacía y a todas las palabras que se forman con la repetición (concatenación) de su operando, un número INDETERMINADO de veces.

Ejemplo 20
La ER a*, que corresponde a la "expresión infinita" ε+a+aa+aaa+aaaa+…, representa el LR infinito $\{a^n / n \geq 0\}$, que podemos describir mediante la frase: "La palabra vacía y todas las palabras formadas solo por aes". En consecuencia, la ER a* tiene la capacidad de representar a todas estas palabras.

* Ejercicio 15 *
Compare las ERs a^{1000} y a*. Informe sus conclusiones.

Ejemplo 21
Consideremos la ER (ab)*. Esta ER corresponde a la "expresión infinita":

ε+ab+abab+ababab+abababab+…

Nota 3
La frase "expresión infinita" y su desarrollo se utilizan solo para mostrar a qué corresponde una ER que representa un LR infinito. No se pueden usar en otro contexto. En general, si e es una subER, entonces $e^* = e^0+e^1+e^2+e^3+\ldots = \varepsilon+e+ee+eee+\ldots$

* Ejercicio 16 *
a) Escriba las tres palabras de menor longitud del LR representado por la ER (aba)*.
b) Utilizando el operador "potencia", escriba la palabra de longitud 300 del LR representado por la ER (aba)*.
c) ¿La cadena ababa pertenece a este LR? Justifique su respuesta.

* Ejercicio 17 *
¿La cadena $a^{24}b^{24}a^{24}$ es una palabra del LR representado por la ER (aba)*? Justifique su respuesta.

Ejemplo 22
La ER aa* corresponde a la "expresión infinita" a+aa+aaa+aaaa+…, y representa el LR infinito $\{a^n / n \geq 1\}$, que podemos describir mediante la frase: "Todas las palabras formadas solo por aes". Note la diferencia sutil que hay entre las ERs a* y aa*: una sola palabra de diferencia. ¿Cuál es?

* Ejercicio 18 *
De los dos LRs infinitos mencionados en el Ejemplo 22, ¿cuál tiene mayor cantidad de palabras? Justifique su respuesta.

Las ERs fueron introducidas por Stephen Kleene en 1956.
Los operadores "clausura de Kleene", concatenación y unión son los tres operadores ORIGINALES descriptos por Kleene.
Son los operadores necesarios y suficientes para construir cualquier ER.
Los llamaremos "operadores básicos de las ERs".

A estos tres operadores básicos agregaremos otros operadores que facilitan la escritura y la comprensión de ERs más complejas, como el operador "potencia", que ya conocemos, el operador que veremos a continuación, y otros que incorporaremos más adelante.

5.2.2.1 EL OPERADOR "CLAUSURA POSITIVA"

Introducimos, entonces, un nuevo operador "no básico", llamado clausura positiva, que tiene la misma prioridad que el operador "clausura de Kleene"; el operador "clausura positiva" se representa mediante un símbolo + ubicado como supraíndice, así: a^+.

El operador "clausura positiva" se utiliza para simplificar la escritura de ERs como la del Ejemplo 22; la ER aa* se puede escribir a^+, por lo que $a^+ = aa^* = a^*a$.

Ejemplo 23
La ER $(a+b)^+$ representa el Lenguaje Universal sobre el alfabeto {a, b} "menos" la palabra vacía. Esta ER representa a la "expresión infinita":

$$(a+b) + (a+b)(a+b) + (a+b)(a+b)(a+b) + \ldots$$

que equivale a:

$$a+b+aa+ab+ba+bb+aaa+aab+aba+abb+baa+bab+bba+bbb+\ldots$$

* Ejercicio 19 *
Describa, mediante una frase, el LR representado por la ER $(a+b)^+$.

La diferencia existente entre el operador "clausura positiva" y el operador "clausura de Kleene" es, justamente, la que se señala en los Ejemplos 20 y 22: mientras que el operador "clausura de Kleene" garantiza la existencia de la palabra vacía, el operador "clausura positiva" la elimina, excepto que la palabra vacía pertenezca al lenguaje que es "clausurado positivamente", como veremos en el próximo ejemplo.

Ejemplo 24
Si la ER a "clausurar positivamente" fuera $\varepsilon+a$, entonces: $(\varepsilon+a)^+ = (\varepsilon+a)^* = a^*$, porque la palabra vacía pertenece al lenguaje original.

* Ejercicio 20 *
Justifique la afirmación del Ejemplo 24.

Ejemplo 25
Las siguientes ERs son equivalentes: $\varepsilon = \varepsilon^* = \varepsilon^+$. ¿Está de acuerdo?

Ejemplo 26
Volviendo a las ERs a^* y a^+, se pueden comprobar las siguientes equivalencias:
(1) $a^*a^* = a^*$;
(2) $a^+ = aa^* = a^*a = aa^*a^* = a^+a^* = a^*a^+ = a^*a^*aa^*$.

* Ejercicio 21 *
Justifique las afirmaciones del Ejemplo 26.

* Ejercicio 22 *
Sea la ER a^+ba^+. Describa, por comprensión, el LR representado por esta ER.

Ejemplo 27
La ER $(ab)^*$ parece equivalente a la ER a^*b^*, pero no lo es: el operador "estrella" no es distributivo con respecto a la concatenación y tampoco lo es con respecto a la unión.

* Ejercicio 23 *
Escriba las únicas palabras comunes a los LRs representados por las ERs $(ab)^*$ y a^*b^*.

* Ejercicio 24 *
Sea el LR representado por la ER a^*+b^3.
(a) Escriba las cinco palabras de menor longitud;
(b) Describa este LR mediante una frase en castellano.

Ejemplo 28
La ER $b^*ab^*ab^*$ representa al lenguaje "Todas las palabras sobre el alfabeto $\{a, b\}$ con exactamente dos $a$es". Algunas palabras de este lenguaje son: aa, $babba$, $bbaabbb$, aba.

* Ejercicio 25 *
Justifique la afirmación del Ejemplo 28.

Ejemplo 29
Consideremos ahora la ER $(b^*ab^*ab^*)^*$, construida con la aplicación del operador "estrella" a la ER del ejemplo anterior. Dado que el operador "estrella" representa también a la palabra vacía, el

lenguaje correspondiente a esta nueva ER contiene la palabra ε. Por otro lado, el operador "estrella" representa la concatenación del operando consigo mismo un número indeterminado de veces; entonces, todas las palabras representadas por la expresión

(b*ab*ab*)(b*ab*ab*) = b*ab*ab*ab*ab*

también pertenecen a este LR, y estas son: "todas las palabras con exactamente cuatro aes". De la misma forma,

(b*ab*ab*)(b*ab*ab*)(b*ab*ab*) = b*ab*ab*ab*ab*ab*ab*,

que representa "todas las palabras con exactamente seis aes", también pertenece a este lenguaje. Y así sucesivamente.

* Ejercicio 26 *
Defina, mediante una frase, el LR representado por la ER del Ejemplo 29.

Ejemplo 30
La ER (1+0)1***** representa el LR "Todos los números binarios que comienzan con un 1 o con un 0, el cual puede estar seguido por una secuencia de 1s". Los cuatro números binarios de menor longitud de este lenguaje son: 1, 0, 11 y 01.

* Ejercicio 27 *
Construya la ER que representa al LR: "Todas las palabras que comienzan con una secuencia de dos o más 1s seguida de un 0 como último carácter o que comienzan con una secuencia de dos o más 0s seguida de un 1 como último carácter".

5.2.2.2 LA EXPRESIÓN REGULAR UNIVERSAL Y SU APLICACIÓN

Denominamos EXPRESIÓN REGULAR UNIVERSAL (ERU) a la ER que representa al Lenguaje Universal sobre un alfabeto dado (¿Recuerda qué es el Lenguaje Universal? Revise el capítulo 1). En consecuencia, la ERU representa al LR que contiene la palabra vacía y todas las palabras que se pueden formar con caracteres del alfabeto dado.

Ejemplo 31
- Si el alfabeto es {a, b}, la ERU es (a+b)*. Algunas palabras son: ε, a, bbbabaa.
- Si el alfabeto es {0, 1}, la ERU es (0+1)*. Algunas palabras son: 000110101, 00000.
- Si el alfabeto es {a, b, c}, la ERU es (a+b+c)*. Algunas palabras son: cbaabbcca, bbbbbab.

* Ejercicio 28 *
Si el alfabeto fuera "todos los dígitos decimales", ¿cuál sería la ERU sobre ese alfabeto?

Las ERUs tienen una aplicación muy importante en la construcción de ERs que representan a un gran número de LRs infinitos, como apreciaremos en los ejemplos que siguen.

Ejemplo 32
El LR descripto mediante la frase "Todas las palabras sobre el alfabeto {a, b} que comienzan con a", se representa fácilmente mediante la siguiente ER: a(a+b)* (note que la palabra a pertenece a este lenguaje).

Si el LR fuera "Todas las palabras que comienzan con a, pero sobre el alfabeto {a, b, c}", la ER que lo representa sería: a(a+b+c)*.

* Ejercicio 29 *
Sea el LR: "Todas las palabras sobre el alfabeto {a, b} que comienzan con a y tienen longitud mayor o igual que 2". Escriba una ER que lo represente.

Ejemplo 33
Sea el alfabeto {a, b} y sea el LR: "Todas las palabras que comienzan con una a y terminan con otra a". Este LR se puede representar fácilmente mediante la ER a(a+b)*a.

* Ejercicio 30 *
Sea el alfabeto {a, b, c} y sea el LR: "Todas las palabras que comienzan con una a y terminan con una b, o que comienzan con una b y terminan con una a". Escriba una ER que represente a este LR.

Ejemplo 34
Dado el alfabeto {a, b}, el LR "Todas las palabras que terminan con aa o con bb" se representa mediante la ER (a+b)*(aa+bb). Note que las palabras aa y bb pertenecen a este lenguaje.

Nota 4
En el Ejemplo 34 consideramos que "terminar" no significa que debe existir, al menos, un carácter antes de aa o bb. Es decir: asumimos que las palabras aa y bb son las palabras de menor longitud de ese LR.

* Ejercicio 31 *
Dado el alfabeto {a, b, c}, obtenga una ER que represente al LR "Todas las palabras que terminan con aa o con bb y cuya longitud es mayor que 2".

* Ejercicio 32 *
Dado el alfabeto {a, b, c}, sea el LR "Todas las palabras de longitud mayor o igual que seis, que terminan con aa o con bb". Escriba una ER que lo represente.

* Ejercicio 33 *
Dado el alfabeto {a, b, c} y el LR "Todas las palabras que contienen como mínimo dos aes", obtenga una ER que lo represente.

* Ejercicio 34 *
Sea la ER (0+1+2)*11(0+1+2)*. Describa, mediante una frase, el LR representado por esta ER.

Definición
Un Lenguaje Formal es un LENGUAJE REGULAR si existe una EXPRESIÓN REGULAR que lo represente.

➔ A cada ER le corresponde un <u>único</u> LR (aunque se lo puede describir de diferentes maneras). Sin embargo, un LR puede ser representado por varias ERs. Afirmamos entonces: dos ERs son EQUIVALENTES si representan al mismo LR.

Ejemplo 35
En el Ejemplo 34 hemos visto la ER (a+b)*(aa+bb). Dado que el operador "ó" o "unión" es conmutativo, esta ER es equivalente a esta otra ER: (b+a)*(bb+aa).

* Ejercicio 35 *
(a) Sea la ER $(a+b)^3 + b^*$. Escriba dos ERs que sean equivalentes a la dada.
(b) Sea la ER $(ab)^3(ab)^*$. Escriba dos ERs que sean equivalentes a la dada.

5.3 DEFINICIÓN FORMAL DE LAS EXPRESIONES REGULARES

La ER original se construye utilizando solo los operadores básicos *unión* (+), *concatenación* (·) y *clausura de Kleene* (*), y se define formalmente de la siguiente manera recursiva:

(1) ø es una ER que representa al LR vacío (sin palabras, de cardinalidad 0).

(2) ε es una ER que representa al LR que solo contiene la palabra vacía: {ε}.

(3) Todo carácter x de un alfabeto corresponde a una ER x que representa a un LR que solo tiene una palabra formada por ese carácter x. *Ejemplo*: Si Σ = {a, b}, entonces a es una ER que representa al LR {a} y b es una ER que representa al LR {b}.

(4) Una cadena s es una ER s que representa a un LR que solo contiene la palabra s. *Ejemplo*: Dada la cadena abc, la ER abc representa al LR {abc}.

A partir de las cuatro reglas básicas citadas hasta ahora, toda ER más compleja se construye de esta manera:

(5) Si R_1 y R_2 son ERs, entonces R_1+R_2 es una ER. *Ejemplo*: Sean $R_1 = a$ y $R_2 = \varepsilon$; entonces $a+\varepsilon$ es una ER.

(6) Si R_1 y R_2 son ERs, entonces $R_1 \cdot R_2$ (o, simplemente, R_1R_2) es una ER. *Ejemplo*: Sean $R_1 = a$ y $R_2 = b$; entonces ab es una ER.

(7) Si R_1 es una ER, entonces R_1^* es una ER. *Ejemplo*: Sea $R_1 = a$; entonces a^* es una ER.

(8) Si R_1 es una ER, entonces (R_1) es una ER. *Ejemplo*: Sea $R_1 = a$, entonces (a) es una ER.

Esta es la definición formal oficial de las ERs. Podemos ampliar esta definición, agregando los dos operadores que hemos descripto oportunamente y que simplifican sensiblemente la escritura de algunas ERs. Ellos son: el operador *clausura positiva* y el operador *potencia*. En consecuencia, agregamos dos nuevas reglas a la definición anterior:

(9) Si R_1 es una ER, entonces R_1^+ es una ER. *Ejemplo*: Sea $R_1 = a$, entonces a^+ es una ER.

(10) Si R_1 es una ER, entonces R_1^n (con $n \geq 0$ y entero) es una ER. *Ejemplo*: si $R_1 = a$, entonces a^{216} es una ER.

Con respecto a la precedencia de los operadores, ya se ha informado que:

(1) Los tres operadores unarios – "clausura de Kleene", "clausura positiva" y "potencia" – tienen prioridad máxima;

(2) El operador "concatenación" tiene prioridad media; y

(3) El operador "unión" tiene prioridad mínima.

* Ejercicio 36 *
En base a la Definición Formal anterior, escriba la secuencia de pasos para llegar a la conclusión que $(a+b)^*aa+b$ es una ER correcta.

5.4 OPERACIONES SOBRE LENGUAJES REGULARES Y LAS EXPRESIONES REGULARES CORRESPONDIENTES

Recordemos esta definición: "Un Lenguaje Formal es un LR si puede ser representado mediante una ER". En esta sección, analizaremos diversas propiedades que tienen los LRs y las relacionaremos con las ERs.

5.4.1 GENERALIDADES

Teniendo en cuenta que los Lenguajes Formales son conjuntos, las operaciones entre conjuntos – como la unión, la intersección y el complemento – se aplican también a los LRs. Además, hay otras operaciones propias de los Lenguajes Formales – como la concatenación y las clausuras – que, lógicamente, se aplican también a los LRs.

Por ello, afirmamos: los LRs son cerrados bajo la unión, la concatenación, las dos clausuras (la de Kleene y la positiva), el complemento con respecto al Lenguaje Universal y la intersección. Esto es: (1) la *unión* de dos LRs es un LR; (2) la *concatenación* de dos LRs es un LR; (3) la *clausura de Kleene* de un LR es un LR; (4) la *clausura positiva* de un LR es un LR; (5) el *complemento* de un LR con respecto al Lenguaje Universal es un LR; (6) la *intersección* de dos LRs es un LR.

5.4.2 LA UNIÓN DE LENGUAJES REGULARES

Sean L_1 y L_2 dos LRs. Entonces, la unión de L_1 con L_2, que se escribe $L_1 \cup L_2$, es un LR que contiene todas las palabras que pertenecen a cualquiera de los dos LRs.

Si L_1 es representado por una ER R_1 y L_2 es representado por cierta expresión R_2, la unión $L_1 \cup L_2$ es representada por la ER R_1+R_2.

Ejemplo 36
Si L_1 es representado por la ER a^*b y L_2 es representado por la ER $ba+b^*$,
$L_1 \cup L_2$ es representado por la ER $(a^*b)+(ba+b^*) = a^*b+ba+b^*$.

* Ejercicio 37 *
a) Dados dos LRs representados, respectivamente, por las ERs $(a+b)^*a^*$ y $a^{24}b$, escriba dos ERs que representen a la unión de los dos LRs.
b) Escriba cinco palabras del LR "unión".

5.4.3 LA CONCATENACIÓN DE LENGUAJES REGULARES

La concatenación de dos LRs, $L_1 \cdot L_2$ (o, simplemente, L_1L_2), es un LR en el que cada palabra está formada por la concatenación de una palabra de L_1 con una palabra de L_2. Por ende, la cardinalidad del LR concatenación es el producto de las cardinalidades de los LRs de partida.

Ejemplo 37
Para ver más fácilmente esta operación, trabajemos, inicialmente, sobre LRs finitos.
Sea L_1 = {ab, c} y sea L_2 = {aa, acc, ad}. Entonces:

L_1L_2 = {abaa, abacc, abad, caa, cacc, cad} y su cardinalidad es 6.

Si L_1 está representado por una ER R_1 y L_2 está representado por una ER R_2, entonces la concatenación L_1L_2 es representada por la ER R_1R_2.

Ejemplo 38
Si L_1 es representado por la ER a*b y L_2 es representado por la ER a+b*, el LR concatenación L_1L_2 es representado por la ER a*b(a+b*). Concatenando una palabra de a*b, como aab, con una palabra de a+b*, como bb, se obtiene aabbb, que es una palabra del LR concatenación.

* Ejercicio 38 *
Sean los LRs representados por las ERs a*bc y b(a+b)*.
a) Escriba la ER que representa a la concatenación de estos dos LRs.
b) Escriba las cinco palabras de menor longitud del LR concatenación.

5.4.4 LA CLAUSURA DE KLEENE DE UN LENGUAJE REGULAR

Si L es un LR, su clausura de Kleene, L*, es un LR infinito (salvo una excepción) formado por:
(1) la palabra vacía,
(2) las palabras de L, y
(3) todas aquellas palabras que se obtienen concatenando palabras de L, un número arbitrario de veces.
La excepción se produce si el LR está formado solo por la palabra vacía, en cuyo caso L* estará formado solo por la palabra vacía.

Ejemplo 39
Sea L = {ab, ba}. Entonces, L* está formado por las palabras ε, ab, ba, abab, abba, baba, ababab, ababba, abbaba, bababa, y muchísimas más.
Si L es representado por la ER R, L* es representado por R*.

Ejemplo 40
Si L es representado por la ER a*b, L* es representado por la ER (a*b)*.
Las palabras ε, aab, aabab, babaabb pertenecen al lenguaje L*.

* Ejercicio 39 *
Escriba otras cuatro palabras del L* del ejemplo anterior.

* Ejercicio 40 *
Sea el LR L representado por la ER ab^+.
a) Escriba la ER de L*.
b) Escriba las cinco palabras de menor longitud de L*.

5.4.5 LA CLAUSURA POSITIVA DE UN LENGUAJE REGULAR

Si L es un LR, su clausura positiva, L^+, es un LR formado por las palabras de L y todas aquellas palabras que se obtienen concatenando palabras de L, un número arbitrario de veces.

Si L es representado por R, L^+ es representado por R^+.

Ejemplo 41
Si L es representado por la ER a*b, L^+ es representado por la ER $(a^*b)^+$. Las palabras aab, aabab, babaabb pertenecen al lenguaje L^+.

* Ejercicio 41 *
a) Sea $L = \{a^n bc^t / n, t \geq 1\}$. Escriba la ER que representa a L^+.
b) Escriba las cinco palabras de menor longitud de L^+.

➔ La clausura positiva de un LR contiene a la palabra vacía solo si ésta pertenece al LR original.

5.4.6 EL COMPLEMENTO DE UN LENGUAJE REGULAR

El complemento de un LR con respecto al Lenguaje Universal, que se escribe L^c, es un LR que está formado por todas aquellas palabras que no pertenecen al LR original.

Ejemplo 42
Sea el alfabeto {a,b}. Si L está representado por la ER $(a+b)^+$, o sea, la ERU "menos" la palabra vacía, entonces L^c solo contiene a la palabra vacía.

Ejemplo 43
Si L es a(a+b)*, es decir, el LR formado por todas las palabras que comienzan con a, entonces L^c es $b(a+b)^* + \varepsilon$.

Nota 5
No existen operadores oficiales para describir el complemento de una ER, aunque en algunos casos se utiliza el operador not. En el Volumen 3 de este libro se estudia cómo se puede obtener el complemento de una ER a través de ciertos algoritmos.

* Ejercicio 42 *
Sea el LR representado por la ER b(a+b+c)*. Escriba las cinco palabras de menor longitud de su lenguaje complemento.

5.4.7 LA INTERSECCIÓN DE DOS LENGUAJES REGULARES

Dado que los LR son conjuntos, se pueden aplicar sobre ellos las operaciones propias de los conjuntos. La intersección es una de ellas. Lo más importante es que la aplicación de estas operaciones, muy útiles para trabajar con lenguajes difíciles de definir de otra manera, produce lenguajes que siguen siendo LRs.

➔ La intersección de dos LRs es un LR constituido por todas aquellas palabras que pertenecen a los dos lenguajes dados.

Ejemplo 44
Sea el alfabeto {a, b}. Si L_1 es a(a+b)* ["todas las palabras que comienzan con a"] y L_2 es (a+b)*b ["todas las palabras que terminan con b"], entonces $L_1 \cap L_2$ es a(a+b)*b ["todas las palabras que comienzan con a y terminan con b"].

* Ejercicio 43 *
Sea el alfabeto {a, b}. Sea L_1 representado por la ER aa(a+b)*b y sea L_2 representado por la ER (a+b)*b. Escriba la ER que representa al lenguaje $L_1 \cap L_2$. Justifique su solución.

Nota 6
No existen operadores para describir la intersección de dos ERs. En el Volumen 3 de este libro se estudia cómo se puede obtener la intersección de dos ERs mediante la aplicación de ciertos métodos.

5.5 EXPRESIONES REGULARES Y LENGUAJES DE PROGRAMACIÓN

Los componentes léxicos de un LP – los identificadores, las palabras reservadas, las constantes, los literales cadena, los operadores, los símbolos de puntuación – constituyen diferentes LRs. Como tales, los podremos representar mediante ERs.

Ejemplo 45
Sea un LP_1 en el que IDENTIFICADOR es una secuencia de letras minúsculas y dígitos que comienza con una letra minúscula. *Supongamos* que la letra L se utiliza para representar a cualquiera de las letras minúsculas del alfabeto inglés y que la letra D representa a cualquiera de los dígitos decimales. Entonces, el LR infinito IDENTIFICADOR puede representarse mediante la ER L(L+D)*.

➔ Si queremos hacer la representación completa, *sin suposiciones*, deberíamos construir la llamada DEFINICIÓN REGULAR. En ésta, se le da un nombre a cada ER auxiliar, hasta obtener la ER principal en función de los nombres dados a las ERs auxiliares.

Ejemplo 46
Una Definición Regular para el LR IDENTIFICADOR del Ejemplo 45 podría escibirse así:

Letra = a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z
Dígito = 0+1+2+3+4+5+6+7+8+9
Identificador = Letra (Letra + Dígito)*

* Ejercicio 44 *

Sea un LP_2 en el que IDENTIFICADOR es una posible secuencia de letras minúsculas y dígitos que comienza con una letra mayúscula. Escriba una Definición Regular que represente a este LR.

En ANSI C, el LR IDENTIFICADOR difiere del definido en el Ejemplo 45: además de letras (minúsculas y mayúsculas) y dígitos, un identificador puede tener "guiones bajos" (_). Para el LR IDENTIFICADOR en ANSI C, este carácter, al que simbolizaremos como G, tiene las mismas propiedades que las letras.

* Ejercicio 45 *
En base a lo visto en el Ejemplo 46 y a lo que hemos descripto en el párrafo anterior, represente el LR IDENTIFICADOR EN ANSI C mediante una Definición Regular, utilizando las ERs auxiliares que llamaremos L, D y G.

Ejemplo 47
Sea un LP_2 en el que CONSTANTE REAL es una secuencia de dígitos (por lo menos uno), seguida de un punto, seguido, no obligatoriamente, de una secuencia de dígitos. Entonces, el LR infinito CONSTANTE REAL puede representarse mediante la ER $D^+.D^*$

* Ejercicio 46 *
Escriba tres palabras que muestren diferentes situaciones del LR CONSTANTE REAL del ejemplo anterior.

* Ejercicio 47 *
Sea un LP_3 en el que CONSTANTE REAL es una secuencia de dígitos (por lo menos uno) seguida de un punto seguido, no obligatoriamente, de una secuencia de dígitos, o bien es una secuencia de dígitos no obligatoria seguida de un punto seguido, obligatoriamente, de una secuencia de dígitos. Escriba una Definición Regular que represente a este LR.

* Ejercicio 48 *
Escriba ejemplos de todas las palabras que considere necesarias para mostrar los diferentes casos representados por la Definición Regular del ejercicio anterior.

* Ejercicio 49 *
Escriba una Definición Regular que represente el LR infinito de los NÚMEROS ENTEROS en base 10, con y sin signo.

Ejemplo 48
Sea un LP_4 que solo tiene tres PALABRAS RESERVADAS: if, else y while. Entonces, el LR finito de las Palabras Reservadas puede representarse mediante la ER if+else+while.

Finalizamos esta sección con la representación, mediante Definiciones Regulares, de las constantes numéricas completas en ANSI C. Como vimos en el capítulo 3, las constantes numéricas, enteras y reales, representan dos LRs infinitos. Estas constantes numéricas son "sin signo" porque, en ANSI C, el signo es un operador. Entonces,

constanteNumérica: uno de *constanteEntera constanteReal*

Comenzamos recordando la definición, en BNF, de las constantes enteras. En ANSI C existen tres tipos de constantes enteras: (1) las decimales, o sea, en base 10; (2) las octales (en base 8); y (3) las hexadecimales (en base 16). La BNF para las constantes enteras en ANSI C es la siguiente:

constanteEntera: *constanteDecimal sufijoEntero*$_{op}$
constanteOctal sufijoEntero$_{op}$
constanteHexadecimal sufijoEntero$_{op}$
constanteDecimal: *dígitoNoCero*
constanteDecimal dígito
constanteOctal: **0**
constanteOctal dígitoOctal
constanteHexadecimal: *ceroX dígitoHexa*
constanteHexadecimal dígitoHexa
ceroX: uno de **0x 0X**
dígitoNoCero: uno de **1 2 3 4 5 6 7 8 9**
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
dígitoOctal: uno de **0 1 2 3 4 5 6 7**
dígitoHexa: uno de **0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F**
sufijoEntero: *sufijoUnsigned sufijoLong*$_{op}$
sufijoLong sufijoUnsigned$_{op}$
sufijoUnsigned: uno de **u U**
sufijoLong: uno de **l L**

* Ejercicio 50 *
En base a la anterior definición en BNF, construya una Definición Regular que representa a las Constantes Enteras en ANSI C.

Continuamos con la BNF de las constantes. Ahora desarrollamos la sintaxis de las constantes reales, que se pueden representar en punto fijo (como 2.14) o en punto flotante (como 3E18, que significa $3x10^{18}$). La BNF para las constantes reales en ANSI C es la siguiente:

constanteReal: *constanteFraccionaria parteExponente*$_{op}$ *sufijoReal*$_{op}$
secuenciaDígitos parteExponente sufijoReal$_{op}$
constanteFraccionaria: *secuenciaDígitos*$_{op}$ **.** *secuenciaDígitos*
secuenciaDígitos **.**
parteExponente: *operadorE signo*$_{op}$ *secuenciaDígitos*
operadorE: uno de **e E**
signo: uno de **+ -**
secuenciaDígitos: *dígito*
secuenciaDígitos dígito
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
sufijoReal: uno de **f F l L**

* Ejercicio 51 *
En base a la anterior definición en BNF, construya una Definición Regular que representa a las Constantes Reales en ANSI C.

5.6 EXPRESIONES REGULARES EXTENDIDAS

Las ERs también se emplean para representar datos que serán procesados por diversas herramientas de software.

Lex es una herramienta diseñada para ayudar a construir un Analizador Léxico, un módulo fundamental que tiene todo compilador y del cual nos ocuparemos en el Volumen 2. Pero *Lex* también resulta útil para otras aplicaciones que no son tan complejas; por ejemplo, aquellas que requieren algún tipo de búsqueda de patrones en un texto de entrada. En todos los casos, *Lex* utiliza ERs para describir los patrones que debe encontrar y procesar. Más adelante veremos ejemplos completos de su utilización.

5.6.1 UN METALENGUAJE PARA EXPRESIONES REGULARES

Toda herramienta de software que necesite representar ERs para su posterior procesamiento, considera a estas expresiones como un lenguaje: el Lenguaje de las Expresiones Regulares.

Un metalenguaje es un lenguaje que se usa para describir otro lenguaje. El lenguaje que queremos describir es el de las ERs, y el metalenguaje que utilizaremos para describirlo tiene ciertos símbolos – además de los operandos y de los operadores – que agregamos para representar, en forma inequívoca, a las ERs. A estos símbolos los llamamos metacaracteres.

En definitiva, el Lenguaje de las ERs está formado por:

(1) operandos, que son los caracteres del alfabeto que intervienen en la construcción de las ERs;

(2) operadores, como clausura de Kleene, concatenación, unión, y otros que serán incorporados a continuación;

(3) ciertos caracteres especiales, llamados metacaracteres, que colaboran en la descripción, sin ambigüedades, de cada ER;

(4) paréntesis para agrupar, como en las expresiones aritméticas.

Los metalenguajes para describir a las ERs están bastante estandarizados. En la siguiente sección haremos una introducción a un subconjunto de un metalenguaje. Comprender y aprender a trabajar con este metalenguaje en particular servirá, luego, para comprender y poder trabajar con cualquier otro metalenguaje para ERs.

5.6.1.1 EL METALENGUAJE, SUS METACARACTERES Y SUS OPERADORES

A continuación se presenta un subconjunto del metalenguaje utilizado por diversas herramientas de software, para describir a las ERs. Debemos tener en cuenta dos características importantes:

(1) No pueden existir subíndices ni supraíndices; todo el texto debe escribirse en la línea (como cuando escribimos un programa en algún LP).

(2) Los metacaracteres y los operadores deben representar las operaciones básicas para la escritura de ERs (unión, concatenación y clausura de Kleene) más otras operaciones que resultan útiles porque simplifican la escritura de las ERs; por lo tanto, habrá nuevos operadores.

La siguiente tabla nos muestra los metacaracteres y operadores que utilizaremos para construir las ERs con este metalenguaje. Para diferenciarlas de las ERs que hemos escrito hasta el momento, a las ERs que escribiremos con este metalenguaje las llamaremos metaERs.

Metacaracteres Operadores	Explicación y Ejemplo
. (punto)	Se corresponde con cualquier carácter, excepto el "nueva línea" (\n). *Ejemplo*: a.a representa cualquier cadena de tres caracteres en la que el primer carácter y el tercer carácter son a.
| (barra vertical)	Es el operador *unión* de ERs. *Ejemplo*: ab|b representa la ER ab+b.
[ ] (corchetes)	Describe un conjunto de caracteres. Simplifica el uso del operador "unión". *Ejemplo*: [abx] representa la ER a+b+x.
[–]	Describe una clase de caracteres en uno o más intervalos. *Ejemplo 1*: [a–d] representa la ER a+b+c+d. *Ejemplo 2*: [0-9a-z] representa cualquier dígito decimal o cualquier letra minúscula (del alfabeto inglés).
{ } (llaves)	Es el operador *potencia*, una repetición *determinada* del patrón que lo precede como operando. *Ejemplo 1*: a{3} representa la ER aaa. *Ejemplo 2*: (ab){4} representa la ER abababab.
{,}	Es el operador *potencia* extendido a un intervalo. *Ejemplo*: a{1,3} representa la ER a+aa+aaa.
?	Es un operador que indica cero o una ocurrencia de la ER que lo precede. *Ejemplo*: a? representa la ER a+ε.
*	Es el operador *clausura de Kleene*: cero o más ocurrencias de la ER que lo precede. *Ejemplo*: a* representa la ER a*.
+	Es el operador *clausura positiva*: una o más ocurrencias de la ER que lo precede. *Ejemplo*: a+ representa la ER a^+.
( ) (paréntesis)	Se utilizan para agrupar una ER. *Ejemplo*: ((ab)? | b)+ representa la ER $(ab+\varepsilon+b)^+$.

A esta tabla de metacaracteres/operadores le agregamos la utilización del carácter \ (barra invertida), cuyo uso es muy conocido en ANSI C. Aquí lo utilizaremos tanto para representar caracteres "no imprimibles", como el \n (nueva línea), como así también para "convertir" y usar, como caracteres, a ciertos metacaracteres definidos en la tabla precedente. *Ejemplo*: el metacarácter + es un operador; para emplear el "más" como carácter lo debemos escribir precedido de una barra invertida, así: \+.

Ejemplo 49
La metaER [0–9]*\.[0–9]+ utiliza el "punto" como carácter (no como operador) para representar ciertos números reales en punto fijo.

* Ejercicio 52 *
(a) Escriba tres palabras, que muestren diferentes situaciones, del LR representado por la metaER del Ejemplo 49.
(b) Escriba una Definición Regular que sea equivalente a la metaER del Ejemplo 49.
(c) Compare la Definición Regular construida con la metaER del Ejemplo 49. Informe sus observaciones.

Ejemplo 50
Sea la metaER (\+|\–)?[0–9]+

* Ejercicio 53 *
(a) Escriba tres palabras, que muestren diferentes situaciones, del LR representado por la metaER del Ejemplo 50.
(b) Escriba una Definición Regular que sea equivalente a la metaER del Ejemplo 50.
(c) Compare la Definición Regular construida con la metaER del Ejemplo 50. Informe sus conclusiones.

Nota 7
Observe que, en el metalenguaje, no existe un operador explícito para la concatenación. La aparición de un carácter o de una clase de caracteres inmediatamente a continuación de otro representa la concatenación de estos caracteres o clases de caracteres, como en el siguiente ejemplo:

Ejemplo 51
(a) La metaER [ab][cd] equivale a (a+b)(c+d) = ac + ad + bc + bd.
(b) La metaER [ab]c equivale a (a+b)c = ac + bc.

Nota 8
En este metalenguaje, los espacios en blanco son opcionales, pero no pueden existir en una concatenación.

Nota 9
Los operadores unarios – clausura de Kleene, clausura positiva, potencia, potencia extendida y ? – tienen máxima prioridad.

Analizando la Tabla de Metacaracteres/Operadores descripta anteriormente, se verifica algo muy importante: algunos de los operadores actúan sobre caracteres, como [ ], mientras que otros operan sobre metaERs, como el operador +.

* Ejercicio 54 *
Para cada operador de la Tabla descripta, indique si actúa sobre caracteres o sobre metaERs.

5.6.1.2 EJERCICIOS PARA ESCRIBIR META EXPRESIONES REGULARES

* Ejercicio 55 *
Escriba una metaER que sea equivalente a la ER $\varepsilon+(a+b)^{34}$.

* Ejercicio 56 *
(a) Escriba una metaER que represente al LR: "Todas las palabras sobre el alfabeto {a, b, c, d} que tienen longitud 100 y terminan con 28 bes".

(b) Escriba una metaER que represente al LR: "La palabra vacía y todas las palabras sobre {a, b, c} cuya longitud está entre 234 y 543, y que, además, comienzan con aa o con bb".

* Ejercicio 57 *
Escriba una metaER que sea equivalente a la ER $(a+b)^{+}(aa+ba)+a^{*}+b^{3}$.

* Ejercicio 58 *
Escriba una metaER que represente al LR: "Todas las palabras sobre las letras minúsculas del alfabeto inglés que comienzan con a y tienen longitud mayor o igual que 2".

* Ejercicio 59 *
Dado el alfabeto de los dígitos decimales, obtenga una metaER que represente al LR "Todas las palabras que terminan con 22 o con 47 y tienen longitud mayor o igual que 3".

* Ejercicio 60 *
Escriba una metaER para cada uno de los siguientes lenguajes:
(a) identificadores en ANSI C;
(b) constantes enteras en ANSI C;
(c) constantes reales en ANSI C;
(d) expresiones en ANSI C.

5.6.1.3 UNA APLICACIÓN: LEX Y EJEMPLOS

(Agradecemos la colaboración de la Prof. Adriana Adamoli, del Prof. José María Sola, y de la Srta. Andrea Alegretti, ayudante de nuestra cátedra)

El *Lex* es una herramienta que permite detectar y procesar palabras de uno o varios LRs. Para detectar esas palabras utiliza ERs escritas en un metalenguaje, y para procesarlas utiliza el Lenguaje de Programación ANSI C. En el próximo ejemplo veremos cómo lo hace.

Ejemplo 52
El siguiente es un programa *Lex* que detecta e imprime todos los números enteros decimales que hay en un archivo de texto. Si no se especifica lo contrario, *Lex* lee desde el archivo estándar de entrada (que habitualmente es el teclado).

El programa *Lex* es el siguiente:

```
%{
/* Detecta e imprime los números enteros */
#include <stdio.h>
%}
%%
[0-9]+   { printf("%s\n", yytext); }
.|\n    ;
%%
int main(void) {
   yylex();
   return 0;
}
```

Descripción

– El programa *Lex* está compuesto por tres secciones.

– La PRIMERA SECCIÓN, llamada "sección de definiciones", introduce codificación (en ANSI C) que será copiada en el programa final. Esta sección siempre está delimitada por %{ y %}, como se ve en el programa del ejemplo.

– En este caso, la "sección de definiciones" tiene una primera línea con un comentario y una segunda línea #include <stdio.h>, porque luego se usará la función printf.

– La SEGUNDA SECCIÓN, llamada "sección de las reglas", está delimitada por %% al comienzo y al final. Cada regla está formada por dos partes: un "patrón", descripto mediante una metaER, y una "acción", descripta mediante un bloque en ANSI C. Ambas partes están separadas por uno o más espacios.

– En este ejemplo, la "sección de las reglas" está formada por dos líneas: (1) la primera línea de esta sección detecta las "palabras" formadas por uno o más dígitos (equivale a dígito^+) y las imprime (el arreglo yytext contiene el texto que concuerda con el "patrón"); (2) la segunda línea, que es por defecto, nos informa que, con cualquier otro carácter o con "nueva línea" ("fin de línea"), no haga nada (el punto y coma representa una sentencia vacía en ANSI C).

– ¿Cómo detecta estas palabras del LR formadas por secuencias de dígitos? Simplemente, recorriendo la secuencia de caracteres que hay en el texto de entrada y detectando un dígito; a partir de este carácter, todos los dígitos sucesivos irán formando una palabra del LR, hasta que encuentre un carácter que no es un dígito.

– La TERCERA SECCIÓN, llamada "sección de subrutinas del usuario", puede contener cualquier codificación correcta en ANSI C.

– En este ejemplo está constituida solo por la función main, función principal de todo programa en ANSI C, que llama a la función yylex. Esta función yylex contiene, básicamente, la codificación en ANSI C de la "sección de las reglas" que ha sido construida.

Ejemplo 53
Veamos, a continuación, un caso un poco más complejo, en el que también se distinguen las tres secciones descriptas arriba. Se trata de un programa *Lex* que reconoce los números enteros y las palabras formadas solo por letras (minúsculas y mayúsculas); además, cuenta cuántos elementos hay de cada una de estas dos clases.

```
%{
/* Reconoce números enteros y palabras alfabéticas; además,
   cuenta las ocurrencias de ambos */
#include <stdio.h>
int nros = 0, pals = 0;
%}
%%
[0-9]+      { nros++; }
[a-zA-Z]+   { pals++; }
.|\n        ;
%%
```

(continúa en la página siguiente)

```
int main(void) {
   yylex();
   printf("Se reconocieron:\n");
   printf("%d Numeros y\n", nros);
   printf("%d Palabras.\n\n", pals);
   return 0;
}
```

* Ejercicio 61 *
Sea el siguiente texto de entrada:

```
Hoy hay 234,65,728   ab32de  ABCZZ--ABCZZ-------87654329900,0000000,,,0
```

Muestre lo que imprime el programa *Lex* del Ejemplo 53.

* Ejercicio 62 *
Basándose en los dos ejemplos anteriores, escriba un programa *Lex* tal que, dado un texto de entrada, cuente la cantidad de palabras formadas por dígitos decimales y letras mayúsculas del alfabeto inglés, que comienzan con un dígito y terminan con una letra.

6 BIBLIOGRAFÍA

ANSI (1990): "*American National Standard for Information Systems – Programming Language C*"; American National Standards Institute, Nueva York, EE. UU.

Backus, J.W. y otros (1963): "*Revised Report on the Algorithmic Language ALGOL 60*"; Communications of the A.C.M., EE. UU.

Cohen, Daniel (1986): "*Introduction to Computer Theory*"; John Wiley & Sons, EE. UU.

Hopcroft, John E. y otro (1979): "*Introduction to Automata Theory, Languages, and Computation*"; Addison-Wesley, EE. UU.

Levine, John R. y otros (1992): "*lex & yacc*"; O'Reilly & Associates; EE.UU.

Muchnik, Jorge D. (2005): "*Autómatas Finitos y Expresiones Regulares*"; Editorial CEIT, Cdad. de Bs. As.

Watson, Des (1989): "*High-Level Languages and Their Compilers*"; Addison-Wesley, EE. UU.

Wirth, Niklaus y otro (1984): "*Pascal: Manual del Usuario e Informe*"; El Ateneo, Cdad. de Bs. As.

7 EJERCICIOS RESUELTOS

CAP. 1: DEFINICIONES BÁSICAS E INTRODUCCIÓN A LENGUAJES FORMALES

* Ejercicio 1 * (pág.8)
Σ = {0, 1, 2, 3, 4, 5, 6, 7, 9, +, –}

* Ejercicio 2 * (pág.8)
012 210

* Ejercicio 3 * (pág.8)
abababcde

* Ejercicio 4 * (pág.9)
$a^{1300}b^{846}a^{257}$

* Ejercicio 5 * (pág.10)
aabbaaabba

*Ejercicio 6 * (pág.11)
Porque c^0 significaría la ausencia de carácter y eso no tiene sentido.

* Ejercicio 7 * (pág.11)
Ambos representan la cadena vacía porque S^0 es la cadena vacía para cualquier cadena S.

* Ejercicio 8 * (pág.11)
$(ab^3)^3 = (abbb)^3 = abbbabbbabbb$ y $((ab)^3)^3 = (ababab)^3 = abababababababababab$

* Ejercicio 10 * (pág.12)
El alfabeto mínimo es = {A, r, g, e, n, t, i, a, H, o, l, d, B, s}

* Ejercicio 11* (pág.13)
Por ejemplo: L = {ccccc, ppppp, cpcpc, ppccp}

* Ejercicio 12 * (pág.14)
$L = \{b^n / 0 \leq n \leq 8\}$

* Ejercicio 13 * (pág.14)
"El lenguaje de todas las palabras sobre el alfabeto {b} que están formadas por la concatenación del carácter b consigo mismo, entre una y ocho veces e incluye a la palabra vacía".

* Ejercicio 14 * (pág.14)
Por extensión se podría describir aunque sería bastante tedioso. Por comprensión no se puede porque no tenemos los operadores adecuados para hacer esta descripción.

* Ejercicio 15 * (pág.15)
"El lenguaje de todas las primeras 201 palabras sobre el alfabeto {a} formado por la concatenación de la letra a consigo misma y donde cada una de ellas tiene un número impar de letras a".

* Ejercicio 16 * (pág.15)
$L = \{a^{2n} / 0 \leq n \leq 400\}$

* Ejercicio 17 * (pág.16)
a) aba, abba, abbba.
b) “El lenguaje de todas las palabras sobre el alfabeto {a,b} que comienza con una única a seguida de una o varias bes y terminan con exactamente una a”.
b’) “El lenguaje sobre el alfabeto {a, b} donde todas las palabras tienen exactamente dos aes (una como primer carácter de la palabra y otra como último carácter de la palabra) y en el medio tienen una o más bes.”

* Ejercicio 18 * (pág.17)
NO porque el Lenguaje Universal es cerrado bajo concatenación.

*Ejercicio 19 * (pág.18)
palabras reservadas : L Finito
nombres creados por el programador (Identificadores): L Infinito
constantes enteras y reales: L Infinito
caracteres de puntuación: L Finito
operadores aritméticos: L Finito
operadores lógicos: L Finito
declaraciones: L Infinito (no regular)
expresiones: L Infinito (no regular)
sentencias: L Infinito (no regular)

* Ejercicio 20 * (pág.18)
(a)

```
unsigned int LongitudCadena (char *s) {
  unsigned int i;
  for(i=0; s[i]!='\0'; i++);
 return i;
 }
```

(b)
Solución 1:

```
int EsCadenaVacia (char *s) {
   if (s[0] == '\0')
      return 1;
   else
      return 0;
}
```

Solución 2:

```
int EsCadenaVacia (char s[]) {
   return s[0]=='\0';
}
```

(c)

```
void ConcatenaDosCadenas(char* s1, const char* s2) {
  unsigned int i,j;
  for (i=0; s1[i]!='\0'; i++);
  for (j=0; s2[j]!='\0'; i++, j++)
     s1[i] = s2[j];
  s1[i]='\0';
}
```

* Ejercicio 21 * (pág.18)
Sugerencia: utilice cadenas constantes.

(a)
```
#include<stdio.h>
unsigned int LongitudCadena (char*);
int main (void)  {
  char cad1[] = "longitud 11";
  char cad2[] = "";
  char cad3[] = " ";
  printf("La longitud de cad1 es: %u\n", LongitudCadena (cad1));
  printf("La longitud de cad2 es: %u\n", LongitudCadena (cad2));
  printf("La longitud de cad3 es: %u\n", LongitudCadena (cad3));
  return 0;
} /* fin-main */

/* Desarrollo funcion LongitudCadena */
unsigned int LongitudCadena (char *s)  {
   unsigned int i;
   for(i=0; s[i]!='\0'; i++);
   return i;
} /* fin-LongitudCadena */
```

(b)
```
#include<stdio.h>
int EsCadenaVacia (char[]);
int main (void)  {
  char cad1[] = "no vacia";
  char cad2[] = "";
  if (EsCadenaVacia(cad1)) printf("La cadena 1 es vacia\n");
  else printf("La cadena 1 no es vacia %s\n", cad1);
  if (EsCadenaVacia(cad2)) printf("La cadena 2 es vacia\n");
  else printf("La cadena 2 no es vacia %s\n", cad2);
return 0;
} /* fin-main*/

/* Desarrollo funcion EsCadenaVacia */
int EsCadenaVacia(char s[])  {
  return s[0]=='\0';
} /* fin-EsCadenaVacia */
```

(c)
```
#include<stdio.h>
void ConcatenaDosCadenas (char*, const char*);
int main (void)  {
  char cad1[27+1] = "Primera parte ";
  char cad2[] = "Segunda parte";
  ConcatenaDosCadenas (cad1, cad2);
  printf ("La cadena concatenada es: %s\n", cad1);
  return 0;
} /* fin-main */
```

(continúa en la página siguiente)

```
/* Desarrollo funcion ConcatenaDosCadenas */
void ConcatenaDosCadenas(char* s1, const char* s2) {
      unsigned int i,j;
      for (i=0; s1[i]!='\0'; i++);
      for (j=0; s2[j]!='\0'; i++, j++)
         s1[i] = s2[j];
      s1[i]='\0';
    }
```

CAP. 2: GRAMÁTICAS FORMALES Y JERARQUÍA DE CHOMSKY

* Ejercicio 1 * (pág.20)
a) S −> aaT, T −> ε => aa
S −> aaT, T −> b => aab
b) No

* Ejercicio 2 * (pág.21)
El lenguaje formal L = {a, aa}
S -> aT, T -> ε => a
S -> aT, T -> a => aa

* Ejercicio 3 * (pág.22)
a) No, porque no hay una producción que permita obtener la última b. S −> bQ, Q −> a, ?
b) {aa, ab, ba, b}

* Ejercicio 4 * (pág.23)
a) S −> aT | aQ
Q−> aT | aR
T −> b
R −> aT
b) G = ({S, Q, T, R}, {a, b}, { S −> aT | aQ, Q−> aT | aR, T −> b, R -> aT}, S)

* Ejercicio 5 * (pág.23)
a) S −> aR | aQ | ε
Q−> aT
T −> bR
R −> b
b) G = ({S, Q, T, R}, {a, b}, {S-> aR, S -> aQ, S -> ε, Q -> aT, T -> bR, R -> b}, S)

* Ejercicio 6 * (pág.24)
ab S −> aT, T −> b

* Ejercicio 7 * (pág.24)
1º) S −> aS, 2º) S -> aS, 3º) S -> aT y 4º) T -> b y se genera la palabra: **aaab**

* Ejercicio 8 * (pág.25)
a) GR = ({S}, {0,1,2,3,4,5,6,7,8,9}, {S −> 0S, S −>1S, S −> 2S, S −> 3S, S −> 4S, S −> 5S,
S −> 6S, S −> 7S, S −> 8S, S −> 9S, S −> 0, S −>1, S −> 2, S −> 3, S −> 4, S −> 5,
S −> 6, S −> 7, S −> 8, S −> 9}, S)

b) GQR = ({S,N}, {0,1,2,3,4,5,6,7,8,9}, {S −> N, S −> NS, N −> 0, N −> 1, N −> 2, N −> 3, N −> 4,
N −> 5, N −> 6, N −> 7, N −> 8, N −> 9}, S)
c) GR 20 producciones y la GQR 12 producciones

*Ejercicio 9 * (pág.25)
S −> AN
A −> BN
B −> N | BN
N −> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

* Ejercicio 10 * (pág.25)
S −> 0A, S−> 1A, S −> 2A, S −> 3A, S −> 4A, S −> 5A, S −> 6A, S −> 7A,
A −> 0B, A−> 1B, A −> 2B, A −> 3B, A −> 4B, A −> 5B, A −> 6B, A −> 7B,
B −> 0B, B−> 1B, B −> 2B, B −> 3B, B −> 4B, B −> 5B, B −> 6B, B −> 7B,
B −> 0, B−> 1, B −> 2, B −> 3, B −> 4, B −> 5, B −> 6, B −> 7

* Ejercicio 11 * (pág.26)
a) Sí, porque una GR siempre es un caso especial de una GIC.
b) No, porque las producciones de una GR son un subconjunto de las producciones de una GIC.
Por ejemplo: S −> abc es una producción válida para una GIC pero no lo es para una GR.

* Ejercicio 12 * (pág.26)
a) **a** aplicando la producción S −> a entonces se genera la palabra **a**
b) **aab** aplicando las producciones: 1º) S −> aSb, 2º) S −> a entonces se genera la palabra **aab**

* Ejercicio 13 * (pág.26)
$L = \{a^{n+1}b^n / n \geq 0\}$

* Ejercicio 14 * (pág.26)
S −> aSb | b

* Ejercicio 15 * (pág.26)
S -> aaTbQ
T -> aaTb | b
Q -> aQ | ε

* Ejercicio 16 * (pág.26)
Si toda GQR puede ser re-escrita mediante una GR y, a su vez, toda GR es un subconjunto de las producciones de una GIC entonces, una GQR es un caso particular de una GIC.

* Ejercicio 17 * (pág.27)
S
ACaB
AaaCB (aplicada Ca −> aaC)
AaaDB (“ CB −> DB)
AaDaB (“ aD −> Da)
ADaaB (“ aD −> Da)
ACaaB (“ AD −> AC)
AaaCaB (“ Ca −> aaC)
AaaaaCB (“ Ca −> aaC)
AaaaaE (“ CB −> E)

AaaaEa (“ aE −> Ea)
AaaEaa (“ aE −> Ea)
AaEaaa (“ aE −> Ea)
AEaaaa(“ aE −> Ea)
εaaaa (“ AE −> ε)
aaaa

* Ejercicio 18 * (pág.28)
S
aSb
aaSbb
aaaSbbb
aaaabbbb No es una palabra del LIC
No hay manera de producir una b más sin una a

* Ejercicio 19 * (pág.30)
G= ({S, T}, {a, b, c}, {S −> Tb, T −> aTc, T −> abc}, S)

* Ejercicio 20 * (pág.30)
a) aaabcccb
S
Tb
aTcb
aaTccb
aaaTcccb
aaabcccb Es palabra del lenguaje
b) aabbccb
S
Tb
aTcb
aaTccb
¿? No hay manera de producir dos bes entre la a y la c, por lo tanto, no es palabra del lenguaje
c) aaabcccbb
S
Tb
aTcb
aaTccb
aaaTcccb
¿? No hay manera de producir una segunda b después de la última c, por lo tanto, no es una palabra del lenguaje
d) aaccb
S
Tb
aTcb
aaTccb
¿? No hay manera de no producir la b entre la a y la c, por lo tanto, no es palabra del lenguaje

* Ejercicio 21 * (pág.30)
GR = ({S, T}, {a,b,c,d,2,3,4,5,6}, {S -> a | b | c | d | aT | bT | cT | dT, T -> aT | bT | cT | dT | 2T | 3T | 4T | 5T | 6T | 2 | 3 | 4 | 5 | 6}, S)
La GQR es más sencilla para ser leída

* Ejercicio 22 * (pág.30)
a) ab23 (Derivación a izquierda)
S
SD
SDD
SLDD
LLDD
aLDD
abDD
ab2D
ab23 es una palabra válida
a') 2a3b (Derivación a izquierda)
S
SD
SDD
SLDD
?? no se puede seguir derivando porque S no produce D, por lo tanto, no es válida
b) ab23 (Derivación a derecha)
S
SD
S3
SD3
S23
SL
Sb23
Lb23
ab23 es una palabra válida
b') 2a3b (Derivación a derecha)
S
SL
Sb
SDb
S3b
SL3b
Sa3b
?? no se puede seguir derivando porque S no produce D, por lo tanto, no es válida

* Ejercicio 23 * (pág.31)
a) S
E;
T;
6; Correcta
b) S
E;
E + T; ¿? No hay manera de producir el terminal + en el extremo derecho.
Por lo tanto, no es correcta

c) S
E;
¿? No hay manera de producir el terminal + en el extremo derecho.
Por lo tanto, no es correcta

d) S
E;
E + T;
E + T + T;
E + T + T + T;
T + T + T + T;
6 + T + T + T;
6 + 6 + T + T;
6 + 6 + 6 + T;
6 + 6 + 6 + 2; Correcta

CAP. 3: SINTAXIS Y BNF

* Ejercicio 2 * (pág.34)
No se puede por la ambigüedad de la descripción de los Identificadores mediante una frase en Lenguaje Natural.

* Ejercicio 3 * (pág.34)
Hay otras. Investigue.

* Ejercicio 4 * (pág.35)
(Abreviamos los nombres de los noterminales: Id, Let y GB)
Id
Id GB Let
Id GB Let GB Let
Let GB Let GB Let
R GB Let GB Let
R_ Let GB Let
R_X GB Let
R_X_ Let
R_X_A

* Ejercicio 5 * (pág.35)
(Abreviamos los nombres de los noterminales: Id, Let y GB)
Id
Id GB Let
¿? No hay manera de producir dos guiones bajos consecutivos porque no hay una producción que sea: Identificador -> Identificador GuiónBajo

* Ejercicio 6 * (pág.36)
(Abreviamos los nombres de los noterminales: Id, Let, Res y GB)
Id -> Let | Id Let | Id Res
Res -> GB Let
GB -> _
Let -> A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
P | Q | R | S | T | U | V | W | X | Y | Z
Tiene una producción más

* Ejercicio 7 * (pág.36)
En base al Ejemplo 3 de la página 36
(Abreviamos los nombres de los noterminales: Id, Let, Res y GB)

Derivación Vertical a Derecha	**Comentario: Se aplica la producción**
Id	Id −> Let Res
Let Res	Res −> GB Let Res
Let GB Let Res	Res −> GB Let Res
Let GB Let GB Let Res	Res −> ε
Let GB Let GB Let	Let −> A
Let GB Let GB A	GB −> _
Let GB Let _A	Let −> X
Let GB X_A	GB −> _
Let _X_A	Let −> R
R_X_A	

* Ejercicio 8 * (pág.36)
(Abreviamos los nombres de los noterminales: Id, Let, Res y GB)
Id
Let Res
A Res
A GB Let Res
A_ Let Res
¿? No hay manera de producir dos guiones seguidos.

* Ejercicio 9 * (pág.37)

```
Expresión -> Término |
             Expresión + Término
Término -> Factor |
           Término * Factor
Factor -> Número |
          ( Expresión )
Número -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

* Ejercicio 10 * (pág.38)
Por ejemplo:
Expresión
Término
Factor
Número
4

* Ejercicio 11 * (pág.38)
(Abreviamos los nombres de los noterminales: Exp, Tér, Fac y Num)

PRODUCCIÓN APLICADA	CADENA DE DERIVACIÓN OBTENIDA
(axioma)	Exp
1	Tér
3	Fac
6	(Exp)
1	(Tér)
3	(Fac)
6	((Exp))
1	((Tér))
3	((Fac))
5	((Num))
7	((2))

* Ejercicio 12 * (pág.38)
Expresión
Expresión + Término
Expresión + Término + Término
¿? No hay manera de producir dos ++ consecutivos

* Ejercicio 13 * (pág.40)

CADENA DE DERIVACIÓN A REDUCIR	PRODUCCIÓN A APLICAR	OPERACIÓN
(1 + 2) * (3 + 4)	7	
(1 + 2) * (3 + Número4)	5	
(1 + 2) * (3 + Factor4)	3	
(1 + 2) * (3 + Término4)	7	
(1 + 2) * (Número3 + Término4)	5	
(1 + 2) * (Factor3 + Término4)	3	
(1 + 2) * (Término3 + Término4)	1	
(1 + 2) * (Expresión3 + Término4)	2	3 + 4 = 7
(1 + 2) * (Expresión7)	6	
(1 + 2) * Factor7	7	
(1 + Número2) * Factor7	5	
(1 + Factor2) * Factor7	3	
(1 + Término2) * factor7	7	
(Número1 + Número2) * Factor7	5	
(Factor1 + Término2) * Factor7	3	
(Término1 + Término2) * Factor7	1	
(Expresión1 + Término2) * Factor7	2	1 + 2 = 3
(Expresión3) * Factor7	6	
Factor3 * Factor7	3	
Término3 * Factor7	4	3 * 7 = 21
Término21	1	
Expresión	(axioma)	Resultado Final

* Ejercicio 14 * (pág.40)
G = ({E, N}, {+, *, (,), 1, 2, 3, 4, 5}, {E −> E + E, E −> E * E, E −> (E), E −> N, N −> 1|2|3|4|5}, E)

* Ejercicio 16 * (pág.42)
SÍ.

* Ejercicio 17 * (pág.43)
Significa la producción vacía, es decir, que no produce nada.

* Ejercicio 18 * (pág.44)
(Abreviamos los nombres de los noterminales: <número entero> <NE>, <entero sin signo> <ESS>, <dígito> <D>)
<NE>
- <ESS>
- <ESS> <D>
- <ESS> <D> <D>
- <ESS> <D> <D> <D>
- <ESS> <D> <D> <D> <D>
- <ESS> <D> <D> <D> <D> <D>
- <D> <D> <D> <D> <D> <D>
- 1 <D> <D> <D> <D> <D>

-12 <D> <D> <D> <D>
-123 <D> <D> <D>
-1234 <D> <D>
-12345 <D>
-123456

* Ejercicio 19 * (pág.45)
(a) <identificador>: “Un identificador debe comenzar obligatoriamente con una letra y puede o no estar seguida de una secuencia de una o más letras o dígitos (en cualquier orden y cantidad).”
(b) <identificador> ::= <letra> |
<identificador> <letra> |
<identificador> <dígito>
<letra> y <dígito> quedan iguales

* Ejercicio 20 * (pág.46)
(a) {**.**, **E**, **+**, **-**, **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**}
(b) {**<>**, **::=**, **|**, **{}** }
(c) 70.70, 70.70E7, 70.70E+7, 70.70E-7, 7000E7, 7000E+7, 7000E-7

* Ejercicio 21 * (págs.46)
(a) subrayadas
(b) 14
. program **( ) { } ; ,** const **=** var **:** begin end
(c) Significa sentencia vacía
(d) 5
(e) SÍ.
<sentencia compuesta>
begin <sentencia> end
begin <sentencia simple> end
begin <sentencia vacía> end
begin <vacío> end
begin end
(f) Infinitas

* Ejercicio 22 * (pág.47)
(1) begin
a := 3
end;
(2) begin
readln (a);
readln (b);
writeln (a+b)
end;

* Ejercicio 23 * (pág.47)
+ – or
* / and

* Ejercicio 24 * (pág.48)
No se puede.

* Ejercicio 25 * (pág.49)
No. Son derivables pero no son sintácticamente correctas.

* Ejercicio 26 * (pág.50)
No se puede porque la Derivación a izquierda debe ser única (lo mismo ocurre a derecha).

* Ejercicio 27 * (pág.51)
LRs: *identificador*, *constantes numéricas* y *literalCadena* son infinitos; los restantes son finitos.

* Ejercicio 28 * (pág.52)
(a) El Preprocesador es un programa que se ejecuta antes del proceso de compilación. Algunas de sus tareas es quitar los comentarios reemplazando cada uno por un blanco, ejecutar las directivas que son para el procesador como los #include y los #define entre otras tareas.
(c) Pide el ingreso de dos valores enteros, calcula cuál de ellos es el mayor y luego despliega por pantalla el mensaje indicado con los valores ingresados y el valor máximo, quedando el cursor en la línea siguiente.

* Ejercicio 29 * (pág.52)
programaC líneas 1 a 17
noC líneas 1 y 2
prototipo línea 3
main líneas 4 a 11
función líneas 12 a 17
etc.

* Ejercicio 30 * (pág.52)
main: *encabezamiento cuerpo*
encabezamiento: **int main (***tipoParámetrosmain***)**
tipoParámetrosmain: **int** *nombreVariable* **, char **** *nombreVariable*
cuerpo: **{** *declaración sentencias* **}**
declaración: *tipoVariables variasVariables*
variasVariables: *variable*
variasVariables variable
sentencias: *sentencia*
sentencias sentencia
nombreVariable: *noDígito*
nombreVariable noDígito
nombreVariable dígito
noDígito: uno de **_ a b c d e f g h i j k l m n o p q r s t u v w x y z**
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
. . . . . .

* Ejercicio 31 * (pág.54)
85

* Ejercicio 32 * (pág.54)

LEXEMA	TOKEN
double	palabraReservada
XX	identificador
(	carácterPuntuación
double	palabraReservada
a	identificador
,	carácterPuntuación
Int	palabraReservada
b	identificador

)	carácterPuntuación
{	carácterPuntuación
while	palabraReservada
(	carácterPuntuación
a	identificador
>	operador
b	identificador
)	carácterPuntuación
b	identificador
++	operador
;	carácterPuntuación
return	palabraReservada
b	identificador
;	carácterPuntuación
}	carácterPuntuación

* Ejercicio 34 * (pág.55)
SÍ se puede usar porque es una palabra clave, no es una palabra reservada, por lo tanto, puede ser el nombre de un identificador en cualquier otro contexto que no sea dentro de la función main.

* Ejercicio 35 * (pág.55)
(a) SÍ. De acuerdo a la BNF se pueden derivar.
(b) NO porque ANSI C hace diferencia entre las letras minúsculas y las mayúsculas.

* Ejercicio 36 * (pág.56)
(a) Sufijo entero puede ser: l, L, u, U o la combinación de l o L con u o U. Indica que la constante entera es long, unsigned, unsigned long, respectivamente.
(b) NO. Porque de acuerdo a la BNF toda constante Octal tiene que comenzar con 0.

* Ejercicio 37 * (pág.56)
(a) un dígito no cero (como cadena) o 1 (como valor)
(b) se representa con cero (si bien el cero es octal, coincide con la constante decimal cero)

* Ejercicio 38 * (pág.56)
(a) octal, decimal, hexadecimal, octal, decimal, octal
(b) (Abreviamos los nombres de los noterminales: constanteHexadecimal cH, dígitoHexadecimal dH)
cH
cH dH
cH dH dH
cH dH dH dH
0X dH dH dH dH
0Xa dH dH dH
0Xa4 dH dH
0Xa4b dH
0Xa4b8

* Ejercicio 39 * (pág.57)
(a) El sufijoReal determina si una constante real es float (f, F) o long double (l, L). La falta de un sufijo indica que la constante es double.
(b) SÍ porque es derivable
(Abreviando los noterminales: *constanteReal* CR, *constanteFraccionaria* CF, *secuenciaDígitos* SD, *dígito* d)

CR
CF
SD.
SD d.
SD d d.
d d d.
4 d d.
42 d.
425.

* Ejercicio 40 * (pág.57)
(a) **0. .0 0E0 0e0 0E+0 0e+0**
(b) (Abreviando los noterminales: SD secuenciaDígitos, OE operadorE, S signo)

FORMATO REAL	EJEMPLO
SD.	13.
SD.f	13.f
SD.F	13.F
SD.l	13.l
SD.L	13.L
.SD	.42
.SDf	.42f
.SDF	.42F
.SDl	.42l
.SDL	.42L
SD.OE SD	23.e2
SD.OE SD	23.E2
SD.OE SDf	23.e2f
SD.OE SDF	23.E2F
SD.OE SDl	23.e2l
SD.OE SDL	23.E2L
SD.OE S SD	50.e+6
SD.OE S SD	50.E+6
SD.OE S SD	50.e-3
SD.OE S SD	50.E-3
SD.OE S SDf	43.e+2f
SD.OE S SDf	43.E-2f
SD.OE S SDF	43.E+2F
SD.OE S SDF	43.e-2F
SD.OE S SDl	31.e+3l
SD.OE S SDL	31.E+3L
SD.OE S SDl	31.e-3l
SD.OE S SDL	31.e-3L

.SD OE SD	.120E2
.SD OE SD	.120e2
.SD OE SDf	.79E2f
. SD OE SDF	.79e2F
SD OE SD	205E8
SD OE SD	205e8
SD OE SDf	205e8f
SD OE SDL	205E8F
SD OE S SD	1000e+4
SD OE S SD	1000E-4
SD OE S SDf	36E+5f
SD OE S SDL	36E-5L
Faltan combinaciones	. . . . .

* Ejercicio 41 * (pág.57)
(Abreviando los noterminales: CR *constanteReal*, CF *constanteFraccionaria*, PE *parteExponenete*, SD *secuenciaDígitos*, S *signo*, OE *operadorE*)
CR
CF PE
SD.SD PE
D.SD PE
4.SD PE
4.D PE
4.6 PE
4.6 OE S SD
4.6E S SD
4.6E- SD
4.6E- SD D
4.6E- D D
4.6E-2 D
4.6E-23

* Ejercicio 42 * (pág.57)
‘\’’

* Ejercicio 43 * (pág.57)
Un ejemplo:

```
typedef enum {ROJO, AMARILLO, AZUL, VERDE, NARANJA, MARRON, BLANCO,
                NEGRO} COLORES;
```

* Ejercicio 44 * (pág. 57)
(a) “\””
(b) “AAA”
(c) SÍ. “”
(d) “234” es un literalCadena formada por la secuencia de los dígitos ‘2’, ‘3’ y ‘4’, mientras que 234 es una constante entera decimal
(e) literalCadena: “dígito secuenciaDígitos” |
“”

secuenciaDígitos: dígito |
secuenciaDígitos dígito

* Ejercicio 45 * (pág.58)
(a) **++** operador incremento, es un operador unario; incrementa en uno el valor de la variable a la que se le aplica este operador
& operador dirección, aplicado a una variable cualquiera se obtiene la dirección de comienzo de esa variable. (Ejemplo: Sea **int *p, a=10;** entonces **p = &a;** Ahora la variable puntero **p** tiene la dirección de comienzo de la variable **a**)
el carácter ***** puede ser el operador producto (Ejemplo: **a * 7**) o el operador indirección (este operador se aplica a una variable puntero y permite hacer referencia al contenido de lo apuntado por la variable puntero. (Ejemplo: Sea **int *p, a, b=10;** entonces **p = &b; a = *p;** Ahora la variable **a** contiene el valor 10)
+ operador aditivo
! operador negación lógica, es un operador unario
sizeof operador que se puede aplicar a un tipo de dato (encerrando el tipo de dato entre paréntesis) o a una variable; devuelve la cantidad de bytes que corresponden al tipo de dato o a la variable
/ operador división entera o real
% operador módulo, devuelve el resto de la división entera o
carácterPuntuación (Ejemplo: Sea **int a=20;** entonces **printf("%d ", a);**)
< <= operadores relacionales
== != operadores igualdad
&& || operadores lógicos
?: operador trinario (es una forma simplificada de escribir un if-else)
= operador asignación
+= operador que le suma al contenido de la variable que se encuentra a la izquierda del operador asignación, el resultado de la expresión que se encuentra del lado derecho de la asignación (Ejemplo: **a += 5** significa **a = a +5**)
(b) Ejemplos sencillos de su uso

* Ejercicio 46 * (pág.58)
Las llaves **{ }** indican el comienzo y el fin de un bloque
El punto y coma **;** es el carácter de puntuación que transforma una expresión en una sentencia
La coma es carácter de puntuación cuando se utiliza en la declaración o definición de variables
Los paréntesis **( )** son caracteres de puntuación en cualquier caso que no sea la invocación de una función
Los corchetes **[ ]** son caracteres de putuación cuando se utilizan en la declaración o defínción de un arreglo

* Ejercicio 47 * (pág.58)
Los paréntesis **()** son OPERADORES solamente en la invocación a una función (Ejemplo: printf**(**"Hola"**)**; y son caracteres de puntuación en cualquier otro caso. Ejemplos: while **(**a < 30**)** …
ó int Suma **(**int, int**)**;
La coma **,** es OPERADOR cuando es utilizada en una lista de expresiones (concatenando una de otra); ejemplo: **(**a*3, b+4, c++**)**, en cualquier otro caso, es carácter de puntuación.
Los corchetes son OPERADORES únicamente cuando se hace referencia a un elemento de un arreglo: vec**[**3**]** = 10; mat**[**10**][**20**]** 0 0;

* Ejercicio 48 * (pág.58)
(a) Verdadero: cualquier valor diferente de cero (negativo o positivo), Falso: cero
(b) Si el resultado de la evaluación de una expresión booleana es Verdadero, produce automáticamente el valor 1, si es Falso produce el valor 0.

* Ejercicio 49 * (pág.59)
Sea int *p, vec[10]; si p = vec; entonces, *(p+2) es equivalente a vec[2] .

* Ejercicio 50 * (pág.60)
Una variable es un valorL modificable. Por ejemplo, **int a, vec[10];** la variable **a** de tipo int es un valorL modificable y el arreglo unidimensional **vec**, con elementos de tipo int, cada elemento de vec es un valorL modificable. En cambio, **const int b=20;** no es un valorL modificable o una expresión aritmética tampoco es un valorL.

* Ejercicio 51 * (pág.61)
expPostfijo
expPostfijo [expresión]
expPostfijo [expresión] [expresión]
expPrimaria [expresión] [expresión]
identificador [expresión] [expresión]
identificador dígito [expresión] [expresión]
identificador dígito dígito [expresión] [expresión]
identificador noDígito dígito dígito [expresión] [expresión]
identificador noDígito noDígito dígito dígito [expresión] [expresión]
identificador noDígito noDígito noDígito dígito dígito [expresión] [expresión]
identificador noDígito noDígito noDígito noDígito dígito dígito [expresión] [expresión]
identificador nodígito noDígito noDígito noDígito noDígito dígito dígito [expresión] [expresión]
noDígito nodígito noDígito noDígito noDígito noDígito dígito dígito [expresión] [expresión]
m nodígito noDígito noDígito noDígito noDígito dígito dígito [expresión] [expresión]
ma noDígito noDígito noDígito noDígito dígito dígito [expresión] [expresión]
mat noDígito noDígito noDígito dígito dígito [expresión] [expresión]
matr noDígito noDígito dígito dígito [expresión] [expresión]
matri noDígito dígito dígito [expresión] [expresión]
matriz dígito dígito [expresión] [expresión]
matri1 dígito [expresión] [expresión]
matriz12 [expresión] [expresión]
matriz12 [expAsignación] [expresión]
matriz12 [expCondicional] [expresión]
matriz12 [expOr] [expresión]
matriz12 [expAnd] [expresión]
matriz12 [expIgualdad] [expresión]
matriz12 [expRelacional] [expresión]
matriz12 [expAditiva] [expresión]
matriz12 [expAditiva +expMultiplicativa] [expresión]
matriz12 [expMultiplicativa + expMultiplicativa] [expresión]
matriz12 [expUnaria + expMultiplicativa] [expresión]
matriz12 [expPostfijo + expMultiplicativa] [expresión]
matriz12 [expPrimaria expMultiplicativa] [expresión]
matriz12 [constante + expMultiplicativa] [expresión]
matriz12 [2 + expMultiplicativa] [expresión]
matri12 [2+ expUnaria] [expresión]
matriz12 [2+ expPostfijo] [expresión]
matriz12 [2+ expPrimaria] [expresión]
matriz12 [2+ constante] [expresión]
matriz12 [2+3] [expresión]
matriz12 [2+3] [expAsignación]
matriz12 [2+3] [expCondicional]
matriz12 [2+3] [expOr]
matriz12 [2+3] [expAnd]
matriz12 [2+3] [expIgualdad]
matriz12 [2+3] [expRelacional]

matriz12 [2+3] [expAditiva]
matriz12 [2+3] [expMultiplicativa]
matriz12 [2+3] [expMultiplicativa * expUnaria]
matriz12 [2+3] [expUnaria * expUnaria]
matriz12 [2+3] [expPostfijo * expUnaria]
matriz12 [2+3] [expPrimaria * expUnaria]
matriz12 [2+3] [constante * expUnaria]
matriz12 [2+3] [4* expUnaria]
matriz12 [2+3] [4* expPostfijo]
matriz12 [2+3] [4* expPrimaria]
matriz12 [2+3] [4* constante]
matriz12 [2+3] [4*6]

* Ejercicio 52 * (pág. 61)
Infinitas.

* Ejercicio 53 * (pág.61)
(a) el operador preincremento es un operador unario que incrementa el valor de una variable en uno. Por ejemplo: Sea int a=0; entonces ++a; la variable a pasa a tener el valor 1. Es una manera simplificada de escribir: a = a + 1; Puede aplicarse solamente sobre variables que sean valorL modificable.
(b) El operador dirección es un operador unario que se puede aplicar a cualquier variable para obtener la dirección de comienzo de esa variable. Por ejemplo: Sea **int *p, a=3;** entonces, **p=&a;** luego de esta asignación la variable puntero **p** contiene la dirección de comienzo de la variable **a**, por lo tanto, apunta a la variable **a**.
(c) El operador unario ***** (indirección) se puede aplicar a variables puntero y permite acceder al contenido de la variable apuntada por la variable puntero. Es decir, si una variable puntero **p** tiene la dirección de una variable **a**, entonces ***p** hace referencia al contenido de la variable **a**. Continuemos con el ejemplo del punto anterior: si agregamos la sentencia ***p = 7;** entonces, ahora la variable **a** contiene el valor entero 7.
(d) El operador unario **!** (negación lógica) niega el valor de la expresión a la cual se le aplica este oeprador. Ejemplo: Sea int a=3, b=0; Entonces, if (!(a<b) es Verdadero porque a<b es Falso y no falso es Verdadero. **!a** es Falso y **!b** es Verdadero.
(e) Este operador retorna la cantidad de bytes que ocupa una variable o un tipo de dato. Ejemplos: Sea **double a;** entonces, **sizeof a** retornará **8**. **sizeof(float)** retornará **4**
Tener en cuenta que cuando se aplica este operador sobre una variable se puede o no encerrar entre paréntesis. Los paréntesis son obligatorios cuando se aplica sobre un tipo de dato.

* Ejercicio 54 * (pág.62)
Expresión
expAsignación
expUnaria operAsignación expAsignación
expPostfijo operasignación expAsignación
expPrimaria operAsignación expAsignación
constante operAsignación expAsignación
1 operAsignación expAsignación
1 = expAsignación
1 = expCondicional
1 = expOr
1 = expAnd
1 = expIgualdad
1 = expRelacional
1 = expAditiva

1 = expMultiplicativa
1 = expUnaria
1 = expPostfijo
1 = expPrimaria
1 = constante
1 = 2

* Ejercicio 55 * (pág.62)
(Como el ejemplo 9 de la pág.41)

* Ejercicio 56 * (pág.62)
NO es sintácticamente correcta, es DERIVABLE. Desde el punto de vista del programador no tiene sentido.

* Ejercicio 57 * (pàg.62)
NO es sintácticamente correcta, es DERIVABLE. Desde el punto de vista del programador no tiene sentido.

* Ejercicio 58 * (pág.62)
El lenguaje Pascal tiene alrededor de 20 operadores y ANSI C tiene 45 operadores.
En Pascal, el and tiene la misma prioridad que la multiplicación; mientras que en ANSI C, el **&&** tiene menor prioridad que la multiplicación; etc.

* Ejercicio 59 (pág.63)
(a) **typedef** permite declarar un nuevo tipo. Ejemplo**: typedef int vector[20]; vector** es un tipo, por lo tanto, si en alguna función se define una variable **vector vec;** entonces **vec** es de tipo vector.
(b) **struct** permite definir una estructura formada por campos. Ejemplo: **struct {int a; float b; char c[10];} estructura1, estructura2;** Se define a las variables estructura1 y estructura2 de tipo struct, en donde cada una de ellas tendrá tres campos: el primero entero (int) llamado a, el segundo real (float) y el tercero un arreglo unidimensional de 10 elementos de tipo entero (char).
Combinación de ambos tipos:
typedef struct {int a; float b; char v[100];} REGISTRO; En este caso se está declarando a REGISTRO como el nombre de un tipo que es una estructura formada por tres campos: el primero entero (int) llamado a, el segundo real (float) llamado b y el tercero un arreglo unidimensional llamado v cuyos elementos serán de tipo entero (char). Si luego en cualquiera de las funciones del programa se define, por ejemplo, **REGISTRO reg;** reg es una variable de tipo REGISTRO que tendrá los tres campos antes mencionados.

* Ejercicio 60 * (pág.64)
declaVarSimples
tipoDato listaVarSimples **;**
int listaVarSimples **;**
int listaVarSimples **,** unaVarSimple **;**
int listaVarSimples **,** unaVarSimple **,** unaVarSimple **;**
int listaVarSimples **,** unaVarSimple **,** unaVarSimple **,** unaVarSimple **;**
int unaVarSimple **,** unaVarSimple **,** unaVarSimple **,** unaVarSimple **;**
int variable **,** unaVarSimple **,** unaVarSimple **,** unaVarSimple **;**
int identificador **,** unaVarSimple **,** unaVarSimple **,** unaVarSimple **;**
int a, unaVarSimple **,** unaVarSimple **,** unaVarSimple **;**
int a, variable inicial **,** unaVarSimple **,** unaVarSimple **;**
int a, b inicial **,** unaVarSimple **,** unaVarSimple **;**
int a, b = constante **,** unaVarSimple **,** unaVarSimple **;**

int a, b = 10, unaVarSimple **,** unaVarSimple **;**
int a, b = 10, variable **,** unaVarSimple **;**
int a, b = 10, identificador **,** unaVarSimple **;**
int a, b = 10, c, unaVarSimple **;**
int a, b = 10, c, variable inicial **;**
int a, b = 10, c, identificador inicial**;**
int a, b = 10, c, d inicial **;**
int a, b = 10, c, d = constante **;**
int a, b = 10, c, d = 4;

* Ejercicio 61 * (pág.64)
declaraTipo: *tipo estructura* {*listaCampos*} *nombreTipo* **;**
tipo: **typedef**
estructura: **struct**
listaCampos: *unCampo*
listaCampos **;** *unCampo*
unCampo: *tipoDato listaVarSimples* **;**
tipoDato: *uno de* **int char double**
listaVarSimples: *unaVarSimple*
listaVarSimples **,** *unaVarSimple*
unaVarSimple: *variable*
variable: *identificador*
nombreTipo: *identificador*
identificador: *noDígito*
identificador noDígito
identificador dígito
noDígito: *uno de* **_ a b c d e f g h i j k l m n o p q r s t u v w x y z**
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
dígito: *uno de* **0 1 2 3 4 5 6 7 8 9**

* Ejercicio 62 * (pág.64)
(a) Es Derivable, pero no es Sintácticamente correcta porque no se puede definir dos variables con el mismo nombre: int a y double a
(b) Derivable pero no Sintácticamente correcta porque 12 no es un valorL modificable
(c) Es Derivable y Sintácticamente correcta

* Ejercicio 63 * (pág.64)
(a) SÍ porque en la BNF:
sentCompuesta: **{***listaDeclaraciones*$_{op}$ *listaSentencias*$_{op}$**}** esto implica que puede escribirse **{}**
(b) Es una sentencia compuesta vacía porque lo encerrado entre las llaves es un comentario y cada comentario es reemplazado por un blanco por el Preprocesador.

* Ejercicio 64 * (pág.65)
Es lo mismo que sentencia vacía. En la BNF: *sentExpresión*: *expresión*$_{op}$ **;** como la expresión es opcional puede escribirse solamente **;** (punto y coma) que es una sentencia nula o vacía

* Ejercicio 65 * (pág.65)
No forma parte del lenguaje Pascal

* Ejercicio 66 * (pág.65)
(a) Sí ; (b) Sí ; (c) dos: **14;** y **;** (sentencia vacía)

* Ejercicio 67 * (pág.65)
Es Sintácticamente Incorrecta porque en ANSI C solamente se pueden escribir declaraciones al comienzo de un bloque (de una sentencia compuesta).

* Ejercicio 68 * (pág.65)
switch es la sentencia de selección múltiple. La expresión debe dar como resultado de su evaluación siempre un valor entero.
Dar ejemplos donde se use el break y otros casos en los que no se use. Marcar las diferencias. Lo mismo con default.

* Ejercicio 69 * (pág.65)
En Pascal la expresión tiene que ser booleana y los paréntesis son opcionales. En ANSI C puede ser cualquier expresión y los paréntesis son obligatorios. Además, en Pascal es if-then-else y en ANSI C es if-else.
En Pascal:

```
<sentencia if> ::= if <expresión> then <sentencia> |
                   if <expresión> then <sentencia> else <sentencia>
```

En ANSI C:

```
sentSelección:  if ( expresión ) sentencia
                if ( expresión ) sentencia else sentencia
```

* Ejercicio 70 (pág.65)
Hacer mientras la condición del while sea verdadera; esta sentencia de iteración se ejecuta por lo menos una vez antes de ser evaluada su condición.
Ejemplos:

```
(a)   do                  (b)   do
        a+ = 3;                   printf("Ingrese un valor entero: "
      while (a< 90);              scanf("%s", valor);
                                  while (!Entero(valor));
```

En el ejemplo (b) mientras el contenido de la variable *valor* no sean dígitos seguirá iterando. Es una manera de poder validar el ingreso por teclado, en este caso, de un valor entero.

* Ejercicio 71 * (pág.65)
(a) la sentencia iterativa *for* es una forma abreviada de escribir un *while*. La primera expresión corresponde a la inicialización de una o varias variables y se ejecuta una única vez al entrar en el ciclo; la segunda expresión corresponde a la condición para seguir o no iterando (mientras sea verdadera sigue en el ciclo); y la tercera expresión corresponde a la expresión que será evaluada después de ser ejecutada la última sentencia del cuerpo del for y será comparado su valor con la condición que figura como segunda expresión.
Ejemplos:
(1) Sea: int i;
for (i=0; i<100; i++) printf ("%d\n", i);
Desplegará por pantalla los 100 primeros números enteros del 0 al 99, a razón de uno por línea
Lo mismo pero utlizando la sentencia iterativa while:
Sea:
int i = 0;
while (i<100) { printf("%d\n", i); i++; }

(2) Sea: int i, j, vec[100];
for (i=0, j=100; i<100; i++, j--) vec[i] = j;
Inicializa el vector vec con los valores enteros del 100 al 1
Lo mismo pero utlizando la sentencia iterativa while:

Sea.

```
int i, j, vec[100];
i = j = 0;
while (i<100)  vec[i++] = j++;
```

(b) Es un ciclo infinito (sintácticamente correcto)

* Ejercicio 72 * (pág.65)

Es una sentencia de salto: **return** *expresión*$_{op}$**;**

Permite que se evalúe la expresión, si existe, y retorna el valor de la misma. Permite también interrumpir la ejecución de una función en cualquier lugar de la misma.

* Ejercicio 73 * (pág.65)

No se ejecutará ninguna sentencia. Es sintácticamente correcto.

* Ejercicio 74 * (pág.65)

(a) SÍ.

sentIteración
for (; *expresión* **;)** *sentencia*
for (; *expAsignación* **;)** *sentencia*
for (; *expCondicional* **;)** *sentencia*
for (; *expOr* **;)** *sentencia*
for (; *expAnd* **;)** *sentencia*
for (; *expIgualdad* **;)** *sentencia*
for (; *expRelacional* **;)** *sentencia*
for (; *expAditiva* **;)** *sentencia*
for (; *expMultiplicativa* **;)** *sentencia*
for (; *expUnaria* **;)** *sentencia*
for (; operUnario *expUnaria* **;)** *sentencia*
for (; - *expUnaria* **;)** *sentencia*
for (; - *expPostfijo* **;)** *sentencia*
for (; - *expPrimaria* **;)** *sentencia*
for (; - *expUnaria* **;)** *sentencia*
for (; - *constante* **;)** *sentencia*
. . . . .
for (; -*4* **;)** *sentencia*
for (; -*4* **;)** *sentExpresión*
for (; -*4* **;) ;**

(b) SÍ.

(c) Es un ciclo infinito porque su segunda expresión siempre será verdadera.

* Ejercicio 75 * (pág.66)

Es un ciclo infinito que no hace nada.

* Ejercicio 76 * (pág.66)

(a) SÍ.

sentSalto
return *expresión* **;**
return *expresión* **;**
return *expUnaria operAsignación expasignación* **;**
return *expPostfijo operAsignación expAsignación* **;**
return *expPrimaria operAssignación expAsignación* **;**
return *identificador operAsignación expAsignación* **;**
. . . .

return a *operAsignación expAsignación* **;**
return a = *expAsignación* **;**
return a = *expCondicional* **;**
return a = *expOr* **;**
return a = *expAnd* **;**
return a = *expIgualdad* **;**
return a = *expRelacional* **;**
return a = *expAditiva* **;**
return a = *expMultiplicativa* **;**
return a = *expUnaria* **;**
return a = *expPostfijo* **;**
return a = *expPrimaria* **;**
return a = *constante* **;**
. . . . .
r**eturn a = 8;**

(b) Sí, es derivable y sintácticamente correcta.
(c) Le asigna el valor 8 a la variable a y retorna ese valor.

* Ejercicio 77 * (pág.66)
(a) Archivo: Secuencia | Archivo Secuencia
Secuencia: Dígitos FIN
Dígitos: Dígito | Dígitos Dígito
FIN: #
Dígito: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
(b) Archivo: Secuencia | Archivo Secuencia
Secuencia: Dígitos FIN | FIN
Dígitos: Dígito | Dígitos Dígito
FIN: #
Dígito: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

* Ejercicio 78 * (pág.66)
(a) Suponemos que secuenciaCompuesta y expresiónBooleana son terminales para no tener que definirlas.
(c) sentenciaSelección: **SI** Constructos **FIN**
Constructos: Constructo1 Constructo2
Constructo1: expresiónBooleana **: {** secuenciaCompuesta **}**
Constructo1 expresiónBooleana **: {** secuenciaCompuesta **}**
Constructo2: **OTRO : {** secuenciaCompuesta **}**

CAP. 4: SEMÁNTICA

* Ejercicio 1 * (pág.68)
(a) un tipo arreglo es una lista de elementos de un tipo determinado que se reconocen a través de un subíndice
(b) un tipo puntero es un objeto que contiene una dirección de memoria.

* Ejercicio 2 * (pág.68)
Porque de alguna manera se tiene que hacer referencia a un determinado elemento del arreglo. Para ello se utiliza un subíndice. Por ejemplo: vec[3]. Y cuantas más dimensiones tenga el arreglo más compleja será. Por ejemplo: mat [2][4]

* Ejercicio 3 * (pág.68)
Las constantes, las estructuras, los nombres de los archivos.

* Ejercicio 4 * (pág.68)
Todos son tipos enteros pero el rango es diferente: int y long int son con signo y unsigned long int es sin signo.

* Ejercicio 5 * (pág.69)
(a) **char** tiene un rango de -128 a 127 o 0 a 255 porque en ANSI C no está definido, depende de la implementación. Lo que sí es seguro es que si se pone **signed char** su rango será de -128 a 127 y si se pone **unsigned char** su rango será de 0 a 255.
(b) int, unsigned int, int, int, int, int (16 bites) o long int (32 bites), long int, long int

* Ejercicio 6 * (pág.70)
Porque la representación interna de 0.1 no es exacta porque internamente, en bits, es una representación periódica.

* Ejercicio 7 * (pág.70)
Porque las constantes float se representan en 4 bytes (con 7 dígitos de precisión);
las constantes double se representan en 8 bytes (con 15 dígitos de precisión); y
las constantes long double se representan en 10 bytes (con 19 dígitos de precisión)

* Ejercicio 8 * (pág.70)
(a) Sí, su valor es 5
(b) Sí, su valor es 4

* Ejercicio 9 * (pág.71)
0, 1, 6

* Ejercicio 10 * (pág.71)
(a) Un solo valor porque cuando se evalúa la expresión el valor resultado es único.
(b) Sean int a=5, b=4; double c=3.2; entonces:
- a + b a y b son operandos de tipo int, el operador es + y el valor que produce es 9 (tipo int)
- a + c a es operando de tipo int y c es operando de tipo double, el operador es + y el valor que produce es 8.2 (tipo double)
- c / a – b a y b son operandos de tipo int y c es operando de tipo double, los operadores son **/** y **-** y el valor que produce es 3.2 (tipo double)

* Ejercicio 11 * (pág.71)
Los paréntesis, los corchetes, la coma y el símbolo = son tanto operadores como caracteres de puntuación.
Los paréntesis **()** son OPERADORES solamente en la invocación a una función (Ejemplo: printf(“Hola”)); y son caracteres de puntuación en cualquier otro caso (Ejemplos: while (a < 30) … ó int Suma (int, int);).
Los corchetes **[]** son OPERADORES cuando se hace referencia a un determinado elemento de un arreglo (Ejemplos: vec[3] > c ó mat[0][0] = 2;); en cualquier otro caso, son caracteres de puntuación (Ejemplos: int vec[100]; int mat [4][5];).
La coma **,** es OPERADOR cuando es utilizada en una lista de expresiones (concatenando una con otra), en cualquier otro caso es carácter de puntuación.
El símbolo **=** es OPERADOR cuando es operador de asignación, ejemplos: a = b +c ó a = 100; y es carácter de puntuación en cualquier otro caso, ejemplo: int a = 0;

* Ejercicio 12 * (pág.72)
Es derivable pero no es semánticamente correcta (desde la visión del programador) porque:
2 < 5 es 1 (verdadero) y 1 < 4 es 1 (verdadero) y esto debería ser falso porque 5 no es menor que 4.
En realidad, para el programador, tiene sentido si se escribe: 2 < 5 && 5 < 4

* Ejercicio 13 * (pág.72)

Caso	Resultado
a > 3	1 (verdadero)
a > 3 && b < a + 1	a>3 es 1, a+1 es 7, 2<7 es 1 y 1 && 1 es 1 (verdadero)
2 < 5 < 4	2 < 5 es 1, 1 < 4 es 1 (verdadero)
2 && 3	1 (verdadero)
!(4 + 18 * 236)	0 (falso)

* Ejercicio 14 * (pág.72)
(a) Porque las tres variables terminan siendo inicializadas con el valor 4.
(b) Porque b+2 no es un valorL modificable.

* Ejercicio 15 * (pág.72)
La sentencia de asignación en Pascal produce una acción; mientras que en ANSI C la asignación es una expresión y como tal, produce un valor.
Ejemplos:
En Pascal, **a := 3** ya es una sentencia. La acción es asignarle el valor 3 a la variable a
En ANSI C, **a = 3** es una expresión de asignación. Su valor es 3

* Ejercicio 16 * (pág.73)
Se declara el nombre de un tipo llamado **XX** que es una estructura de dos campos: el primero es un int llamado **a** y el segundo es un arreglo bidimensional de 4 filas por 7 columnas llamdo **b**, en donde cada elemento es de tipo char.

* Ejercicio 17 (pág.73)
Se define a una variable llamada **a** de tipo int con el valor inicial 10.

* Ejercicio 18 * (pág.73)
Se define un arreglo bidimensional llamado **mat** de 2 filas por 3 columnas, en donde cada elemento es de tipo int, conteniendo, en la primera fila, los valores 1, 2 y 3 y, en la segunda fila, los valores 4, 5 y 6.

* Ejercicio 19 * (pág.73)

```
{ int a = 0;
  if (a+3 > 10)  printf(" 1\n");
  else printf (" 0\n"); }
```

Despliega por pantalla el valor 0 y el cursor queda al comienzo de la siguiente línea.

* Ejercicio 20 * (pág.73)
(a) **a = 3 * 9;**
La variable entera **a** contiene el valor entero 27.
(b) La sentencia de asignación es una expresión de asignación seguida de un punto y coma.
Ejemplos: **a = 3;** es una sentencia de asignación; en cambio, **a = 3** es una expresión de asignación

* Ejercicio 21 * (pág.74)
Sean: int a = 0, b = 3, t, c = -3;
(1) if (a + b) t = 1;
else t = 0;
la variable **t** contiene el valor 1

(2) if (a) t = 1;
else t = 0;
La variable **t** contiene el valor 0
(3) while (-3);
es un ciclo infinito
(4) while (a); no hace nada
(5) for (; b > c; b--) a++;
la variable **a** contiene el valor 6
(6) for (; b+c;); no hace nada

* Ejercicio 22 * (pág.74)
El prototipo de una función es una declaración donde se describe cuál es el nombre de la función, el tipo de dato del valor que retorna y, si tiene parámetros, el tipo de datos de cada uno de ellos. La definición de una función describe con precisión las acciones que lleva a cabo la misma.

* Ejercicio 23 * (pág.74)
Si la SentenciaSelección tiene varios constructos, por cada expresiónBooleana evaluada, si esta es verdadera, se ejecutarán la o las acciones que estén indicadas; y si la expresiónBooleana es falsa, existirá, por lo menos, una secuenciaCompuesta y se ejecutarán la o las acciones indicadas en ella.

* Ejercicio 24 * (pág.74)
BNF:
ABRIRGL *identificador tipoAcción*
CERRAR *identificador*
CREAR *identificador valorMax valorMin*
LISTAR *identificador*
identificador: noDígito
identificador noDígito
identificador dígito
noDígito: uno de **_ a b c d e f g h i j k l m n o p q r s t u v w x y z**
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
dígito: uno de **0 1 2 3 4 5 6 7 8 9**
tipoAcción: uno de **L l G g**
valorMax: *enteroDecimal*
valorMin: *enteroDecimal*
enteroDecimal: *dígito*
enteroDecimal dígito

CAP. 5: LENGUAJES REGULARES Y EXPRESIONES REGULARES

* Ejercicio 1 * (pág.75)

(a)

ab	aabb	aaabbb
S	S	S
aT	aT	aT
ab	aaQ	aaQ
	aabR	aaaM
	aabb	aaabP
		aaabbR
		aaabbb

(b) GR = ({S, R, T, Q, N, W, Z}, {a, b}, {S −> aR, R −> b, R −> aT, T −> bQ, T −> aN, Q −> b, N −> bW, W −> bZ, Z −> b}, S)

* Ejercicio 2 * (pág.76)

(a)

a	aaa
S	S
a	aS
	aaS
	aaa

Como cada aplicación de la producción recursiva agrega una **a**, queda verificado que esa GR genera cualquier palabra formada por aes.

(b) GR = ({S}, {a}, {S -> aS, S -> ε}, S)

* Ejercicio 3 * (pág76)

(a)

ab	abb	aab	aaabb
S	S	S	S
aT	aT	aS	aS
ab	abT	aaT	aaS
	abb	**aab**	aaaT
			aaabT
			aaabb

(b) GR = ({S, R, T}, {a, b}, {S −> aT, S −> bR, S −> ε, T −> aT, T −> bR, T −> ε, R −> bR, R −> ε}, S)

* Ejercicio 4 * (pág.76)

(a)

abcde	abcdecdeaaa	ababcdea
S	S	S
abT	abT	abS
abcdeR	abcdeT	ababT
abcde	abcdecdeR	ababcdeR
	abcdecdeaR	ababcdeaR
	abcdecdeaR	**ababcdea**
	abcdecdeaaR	
	abcdecdeaaaR	
	abcdecdeaaa	

(b) DF = ({S, Q, D}, {a, b, c, d, e}, {S −> abS, S −> abQ, Q −> cdeQ, Q −> cdeD, D −> aD, D −> ε}, S}

(c)

abcde	abcdecdeaaa	ababcdea
S	S	S
abQ	abQ	abS
abcdeD	abcdeQ	ababQ
abcde	abcdecdeD	ababcdeD
	abcdecdeaD	ababcdeaD
	abcdecdeaD	**ababcdea**
	abcdecdeaaD	
	abcdecdeaaaD	
	abcdecdeaaa	

* Ejercicio 5 * (pág.76)
(a) GR = ({S, A, B, C},{0, 1}, {S -> 1A | 1B, A -> 0, B -> 1C, C -> 1B | 1A}, S)
(b)

10	1110	111110
S	S	S
1A	1B	1B
10	11C	11C
	111A	111B
	1110	1111C
		11111A
		111110

* Ejercicio 6 * (pág.77)
Sea ER1 aa y ER2 aaa entonces, ER1ER2 = ER2ER1 = aaaaa
Sea ER3 b y ER4 ε entonces, ER3ER4 = ER4ER3 = b

* Ejercicio 7 * (pág.78)
ab+aabb+aaabbb

* Ejercicio 8 * (pág.78)
“El LR sobre el alfabeto {a, b} formado por las palabras que comienzan con a, terminan con b y tienen longitud tres.”

* Ejercicio 9 * (pág.78)
abb(aac+ba+acccac)c

* Ejercicio 10 * (pág.79)
Sí, es correcta.
Si aplicamos la distribución en la ER a(a+b)+b(a+b) nos queda: aa+ab+ba+bb.
Si realizamos lo mismo en la otra ER (a+b)(a+b) nos queda: aa+ab+ba+bb.
Por lo tanto, queda demostrado que las tres ERs son equivalentes.

* Ejercicio 11 * (pág.79)
Sí, son equivalentes porque distribuyendo en a(a+b)+ba nos queda aa+ab+ba que es equivalente a la primera ER. Como la unión es conmutativa la ER aa+ab+ba es iequivalente a la ER aa+ba+ab

* Ejercicio 12 * (pág.79)
$(a+b)^{72}b^{28}$

*Ejercicio 13 * (pág.79)
ER $a^{16}(a+b+c)^{1168}a^{16} + b^{342}(a^{100}+b^{100}+c^{100})$

* Ejercicio 14 * (pág.80)
ER ε + (aa+bb)(a+b+c)^{41}(aa+bb)

* Ejercicio 15 * (pág.80)
La primera ER describe a un LR finito que está incluído dentro del segundo LR que es infinito.

* Ejercicio 16 * (pág.81)
a) ε, aba, abaaba
b) (aba)^{100}
c) No. Porque son todas las palabras que resultan de la concatenación de **aba** consigo mismo; son palabras cuya longitud son múltiplos de tres, como por ejemplo: (aba)^2, (aba)^3, (aba)^4, etc. Por lo tanto, la cadena ababa no es un palabra de este LR.

* Ejercicio 17 * (pág.81)
No. Porque a^{24}b^{24}a^{24} es equivalente a la cadena:
aaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaa
que no es una palabra del LR porque la ER (aba)*, corresponde al LR Infinito L = {(aba)^n / n ≥ 0} cuyas palabras resultan de la concatenación de **aba** consigo mismo; por ejemplo: abaaba, abaabaaba, etc.

* Ejercicio 18 * (pág.81)
El primero porque incluye a la palabra vacía.

* Ejercicio 19 * (pág.81)
“Todas las palabras sobre el alfabeto {a, b} que tienen una longitud mayor o igual a uno .”

* Ejercicio 20 * (pág.82)
La ER **(ε+a)$^+$** está representada por la “expresión infinita”:
(ε+a) + (ε+a)(ε+a) + (ε+a) (ε+a)(ε+a) + . . .
que equivale a:
ε+a+εε+εa+aε+aa + . . .
pero, como la cadena vacía **(ε)** es la IDENTIDAD para la concatenación de cualquier cadena, entonces nos queda:
ε+a+aa+ . . .
y esto corresponde a la ER **a*** que ya contiene a la palabra vacía.
Por lo tanto, es igual a **(ε+a)***

* Ejercicio 21 * (pág.82)
(1) La ER a*a* representa la concatenación de a* con a*; por lo tanto, a* está representada por la “expresión infinita”:
ε+a+aa+ . . . y la segunda a* está representada por la misma “expresión infinita”:
ε+a+aa+ . . . , entonces, (**ε+a+aa+ . . .**)(**ε+a+aa+ . . .**) equivale a:
εε+εa+εaa+. . .+aε+aa+aaa+. . .
pero, como la cadena vacía **(ε)** es la IDENTIDAD para la concatenación de cualquier cadena, entonces nos queda:
ε+a+aa+aaa+. . .
y esto corresponde a la ER **a***
Por lo tanto, **a*a* = a***

* Ejercicio 22 * (pág.82)
L = {a^{n}ba^t / n ≥ 1, t ≥ 1}

* Ejercicio 23 * (pág.82)
ε, ab

* Ejercicio 24 * (pág.82)
(a) ε, a, aa, aaa, bbb
(b) "Todas las palabras que contienen solamente una o más aes o solamente tres bes e incluye a la palabra vacía".

* Ejercicio 25 * (pág.82)
Por comprensión: $L = \{b^n ab^t ab^r / n \geq 0, t \geq 0 \text{ y } r \geq 0\}$;

* Ejercicio 26 * (pág.83)
"Todas las palabras sobre el ∑ {a,b} que incluye a la palabra vacía y tienen en total un número par de aes, consecutivas o no ".

* Ejercicio 27 * (pág.83)
ER 111*0 + 000*1

* Ejercicio 28 * (pág.83)
(0+1+2+3+4+5+6+7+8+9)*

* Ejercicio 29 * (pág.84)
a(a+b)(a+b)*

* Ejercicio 30 * (pág.84)
ER a(a+b*c)*b + b(a+b*c)*a

* Ejercicio 31 * (pág.84)
ER $(a+b+c)^+aa + (a+b+c)^+bb$ ER $(a+b+c)^+ (aa+bb)$

* Ejercicio 32 * (pág.84)
ER $(a+b+c)^4(a+b+c)^*(aa+bb)$

* Ejercicio 33 * (pág.84)
ER (a+b+c)* a (a+b+c)* a (a+b+c)*

* Ejercicio 34 * (pág.84)
Dado el alfabeto {0,1,2} y el LR “Todas las palabras que contienen 11 y son de longitud mayor o igual que dos”.

* Ejercicio 35 * (pág.85)
(a) ER (a+b)(a+b)(a+b) + b* ER $ε + b^+ + (a+b)^3$
(b) ER abababab(ab)* ER $(ab)^3(ab+ε)^+$

* Ejercicio 36 * (pág.86)
Una solución:
1º por (3), a es una ER
2º por (3), b es una ER
3º por (5), a+b es una ER
4º por (8), (a+b) es una ER
5º por (7), (a+b)* es una ER
6º por (3), a es una ER
7º por (6), (a+b)*a es una ER

8º por (3) a es una ER
9º por (6), (a+b)*aa es una ER
10º por (3), b es una ER
11º por (5), (a+b)*aa+b es una ER

* Ejercicio 37 * (pág.86)
(a) ER $a^{24}b + (a+b)^*a^*$ ER $(\varepsilon+a+b)^+a^* + a^{24}b$
(b) ε, a, bb, aaaaaaaaaaaaaaaaaaaaaaab, babaab

* Ejercicio 38 * (pág.87)
(a) a*bcb(a+b)*

* Ejercicio 39 * (pág.87)
aababb, bbbbbbb, bbbbbbabbbbb, bababab

* Ejercicio 40 * (pág.88)
a) ER $(ab^+)^*$
b) ε, ab, abb, abab, abbb

* Ejercicio 41 * (pág.88)
La ER de L es a^+bc^+
Entonces la ER de L^+ es $(a^+bc^+)^+$

* Ejercicio 42 * (pág.88)
El lenguaje complemento es: ε + (a+c)(a+b+c)
ε, a, c, aa, ab, ac, ca, cb, cc

* Ejercicio 43 * (pág.89)
ER aa(a+b)*b

* Ejercicio 44 * (pág.90)
Inicial = A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S+T+U+V+W+X+Y+Z
Otro = a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z+0+1+2+3+4+5+6+7+8+9
Identificador = Inicial Otro*

* Ejercicio 45 * (pág.90)
G = _
L = a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z
D = 0+1+2+3+4+5+6+7+8+9
Identificador = (L+G)(L+D+G)*

* Ejercicio 46 * (pág.90)
34. 70.90 0.0

* Ejercicio 47 * (pág.90)
D = 0+1+2+3+4+5+6+7+8+9
P = .
constanteReal = $(D^+PD^*) + (D^*PD^+)$

* Ejercicio 48 * (pág.90)
29. .89 307.45 0. .0 0.0

* Ejercicio 49 * (pág.90)

D = 0+1+2+3+4+5+6+7+8+9
S = + | -
numeroEntero = (S+ ε)D^{+}

* Ejercicio 50 * (pág.91)

DC = 0
DNC = 1+2+3+4+5+6+7+8+9
CHx = 0x+0X
DO = 0+1+2+3+4+5+6+7
DHx = 0+1+2+3+4+5+6+7+8+9+a+b+c+d+e+f+A+B+C+D+E+F
SU = u+U
SL = l+L
constanteEntera = (DNC D* (SU SL + SL SU + SL + SU + ε)) +
(DC DO* (SU SL + SL SU + SL + SU + ε)) +
(CHx DHx^{+} (SU SL + SL SU + SL + SU + ε))

* Ejercicio 51 * (pág.91)

D = = 0+1+2+3+4+5+6+7+8+9
SD = D +
SD D
S = + | -
SR = f+F+l+L
operE = e | E
parteExponente = operE (S+ε) SD^{+}
constanteFracción = SD^{*} . SD^{+} +
SD^{+} .
constanteReal = (CF (PE+ε) (SR+ε)) +
(SD PE SR+ε))

* Ejercicio 52 * (pág.94)

(b) Dígito = 0+1+2+3+4+5+6+7+8+9
Punto = .
Real = Dígito* Punto $Dígito^{+}$

* Ejercicio 53 * (pág.94)

(b) S = + | -
D = 0+1+2+3+4+5+6+7+8+9
E50 = (S+ε)D^{+}

* Ejercicio 54 * (pág.94)

Metacaracteres Operadores	Actúa sobre:
. (punto)	caracteres
| (barra vertical)	metaER
[ ] (corchetes)	caracteres
[–]	caracteres
{ } (llaves)	caracteres
{,}	caracteres
?	metaER

*	metaER
+	metaER
( ) (paréntesis)	metaER

* Ejercicio 55 * (pág.95)
((a|b){34})?

* Ejercicio 56 * (pág.95)
(a) [abcd]{72}b{28}
(b) ((aa|bb)[a-c]{232,541})?

* Ejercicio 57 * (pág.95)

[ab]+(aa|ba)|a*|b{3}

* Ejercicio 58 * (pág.95)

a[a-z][a-z]*

* Ejercicio 59 * (pág.95)
[0-9]+(22|47)

* Ejercicio 60 * (pág.95)

(a) [a-z_A-Z][a-zA-Z_0-9]*

(b) [1-9][0-9]*(u|U|l|L|ul|lu|UL|LU)?
(d) No es un Lenguaje Regular

* Ejercicio 61 * (pág.97)
Se reconocieron:
7 Numeros y
6 Palabras.

* Ejercicio 62 * (pág.97)

```
%{
/* Cuenta la cantidad de palabras formadas por digitos decimales
   Y letras mayúsculas del alfabeto ingles, que comienzan con un
   digito y terminan con una letra */
#include <stdio.h>
int cuenta = 0;
%}
%%
[0-9][0-9A-Z]+[A-Z]      { cuenta++; }
.|\n        ;
%%
int main(void) {
   yylex();
   printf("Se reconocieron:\n");
   printf("%d Palabras\n\n", cuenta);
   return 0;
}
```