SSL 2

Desde el Compilador

Jorge D. Muchnik, 2010

Para el ISBN

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.

Agradezco a los profesores que integran la cátedra “Sintaxis y Semántica de los Lenguajes”, que siempre han apoyado mi gestión y han colaborado de distintas maneras para que este libro sea una realidad.

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, Titular
febrero 2010

ÍNDICE

1 INTRODUCCIÓN A AUTÓMATAS FINITOS Y APLICACIONES 7
1.1 ¿QUÉ ES UN AUTÓMATA FINITO? 8
1.2 AUTÓMATAS FINITOS QUE RECONOCEN LENGUAJES REGULARES FINITOS . . . 10
1.3 AUTÓMATAS FINITOS QUE RECONOCEN LENGUAJES REGULARES INFINITOS . . 12
1.4 AUTÓMATAS FINITOS DETERMINÍSTICOS 16
1.4.1 DEFINICIÓN FORMAL DE UN AFD 17
1.4.2 AFD COMPLETO 19
1.5 UNA APLICACIÓN: VALIDACIÓN DE CADENAS 20
1.6 UNA EXTENSIÓN DE LA APLICACIÓN: UNA SECUENCIA DE CADENAS 22

2 AUTÓMATAS FINITOS CON PILA 27
2.1 DEFINICIÓN FORMAL DE UN AFP 27
2.2 AUTÓMATA FINITO CON PILA DETERMINÍSTICO (AFPD) 29
2.2.1 LA TABLA DE MOVIMIENTOS (TM) 30
2.3 AFPDs Y EXPRESIONES ARITMÉTICAS 32

3 INTRODUCCIÓN AL PROCESO DE COMPILACIÓN 33
3.1 CONCEPTOS BÁSICOS 33
3.1.1 EL ANÁLISIS DEL PROGRAMA FUENTE 33
3.2 UN COMPILADOR SIMPLE 36
3.2.1 DEFINICIÓN INFORMAL DEL LENGUAJE DE PROGRAMACIÓN MICRO 37
3.2.2 SINTAXIS DE MICRO – DEFINICIÓN PRECISA CON UNA GIC (BNF) . . . 38
3.2.3 LA ESTRUCTURA DE UN COMPILADOR Y EL LENGUAJE MICRO 39
3.2.4 UN ANALIZADOR LÉXICO PARA MICRO 40
3.2.4.1 EL AFD PARA IMPLEMENTAR EL SCANNER 42
3.2.4.2 CONCLUSIÓN 44
3.2.5 UN PARSER PARA MICRO 45
3.2.6 LA ETAPA DE TRADUCCIÓN, LAS RUTINAS SEMÁNTICAS Y LA AMPLIACIÓN DE LOS PAS 50
3.2.6.1 INFORMACIÓN SEMÁNTICA 51
3.2.6.2 LA GRAMÁTICA SINTÁCTICA DE MICRO CON LOS SÍMBOLOS DE ACCIÓN . 51
3.2.6.3 ALGUNAS RUTINAS SEMÁNTICAS 53
3.2.6.4 PROCEDIMIENTO DE ANÁLISIS SINTÁCTICO (PAS) CON SEMÁNTICA INCORPORADA 54
3.2.6.5 UN EJEMPLO DE ANÁLISIS SINTÁCTICO DESCENDENTE Y TRADUCCIÓN . 55

4 ANÁLISIS LÉXICO, ANÁLISIS SINTÁCTICO Y ANÁLISIS SEMÁNTICO 57
4.1 INTRODUCCIÓN A LA TABLA DE SÍMBOLOS 57
4.2 EL ANÁLISIS LÉXICO 58
4.2.1 RECUPERACIÓN DE ERRORES 61
4.3 EL ANÁLISIS SINTÁCTICO 62
4.3.1 TIPOS DE ANÁLISIS SINTÁCTICOS Y DE GICs 63

4.3.1.1 GRAMÁTICAS LL Y LR 65
4.3.2 GRAMÁTICAS LL(1) Y APLICACIONES 66
4.3.3 OBTENCIÓN DE GRAMÁTICAS LL(1) 68
4.3.3.1 FACTORIZACIÓN A IZQUIERDA 68
4.3.3.2 ELIMINACIÓN DE LA RECURSIVIDAD A IZQUIERDA 69
4.3.3.3 SÍMBOLOS DE PREANÁLISIS Y EL CONJUNTO *PRIMERO* 70
4.3.3.3.1 OBTENCIÓN DEL CONJUNTO *PRIMERO* 71
4.3.3.3.2 OBTENCIÓN DEL CONJUNTO *SIGUIENTE* 71
4.3.3.4 LA FUNCIÓN *PREDICE* Y EL ASDR 73
4.3.4 USANDO UN AFP PARA IMPLEMENTAR UN PARSER PREDICTIVO 73
4.3.5 OTRA VISIÓN: EL LENGUAJE DE LOS PARÉNTESIS ANIDADOS, UN AFP Y EL PARSER DIRIGIDO POR UNA TABLA 76
4.3.6 EL PROBLEMA DEL IF-THEN-ELSE 77
4.3.7 ANÁLISIS SINTÁCTICO ASCENDENTE 78
4.3.7.1 CÓMO FUNCIONA EL ANÁLISIS SINTÁCTICO ASCENDENTE 78
4.3.7.2 RECURSIVIDAD EN EL ANÁLISIS SINTÁCTICO ASCENDENTE 80
4.3.7.3 OTRA VISIÓN: LA IMPLEMENTACIÓN DE UN PARSER ASCENDENTE COMO UN AFP 80
4.3.7.4 EL USO DE *yacc* 81
4.4 ANÁLISIS SEMÁNTICO 83
4.4.1 EN ANSI C: DERIVABLE VS SINTÁCTICAMENTE CORRECTO 84

5 BIBLIOGRAFÍA 85

6 EJERCICIOS RESUELTOS 87

1 INTRODUCCIÓN A AUTÓMATAS FINITOS Y APLICACIONES

Como hemos visto en el Volumen 1, los Lenguajes Regulares se representan en forma precisa por medio de Expresiones Regulares; aunque también pueden ser descriptos mediante frases en un Lenguaje Natural (siempre que sea sin ambigüedades) o expresados como conjuntos.

También hemos visto que los Lenguajes Formales son generados por las llamadas Gramáticas Formales, que se clasifican en diferentes tipos según la Jerarquía de Chomsky.

Además, para completar este panorama, existen diferentes mecanismos abstractos, llamados Autómatas, que tienen la capacidad de RECONOCER a estos Lenguajes Formales.

Resumiendo: una Gramática Formal GENERA un determinado Lenguaje Formal y un Autómata RECONOCE ese Lenguaje Formal. Pero, ¿qué significa "reconocer un Lenguaje Formal"?

➔ Un Autómata RECONOCE a un determinado Lenguaje Formal si tiene la capacidad de reconocer cada palabra del lenguaje y de rechazar toda cadena que no es una palabra de ese lenguaje.

* Ejercicio 1 *
(a) ¿Existirá un Autómata que reconozca {ab, abc}? Justifique su respuesta.
(b) Si existe, ¿ese Autómata reconoce la cadena abb? Justifique su respuesta.

Al surgir estos mecanismos abstractos llamados Autómatas, completamos el cuadro de la Jerarquía de Chomsky presentado en el Volumen 1, capítulo 2, de esta manera:

G. Formal –> genera:	**L. Formal –> reconocido por:**	**Autómata**
GR	LR	Autómata Finito
GIC	LIC	Autómata Finito con Pila
GSC	LSC	Máquina de Turing
G. Irrestricta	L. Irrestricto	Máquina de Turing

En la tabla anterior se distinguen tres tipos de Autómatas:

– Los Autómatas Finitos, que solo reconocen a los Lenguajes Regulares;
– Los Autómatas Finitos con Pila (con un *stack*), que reconocen a los Lenguajes Independientes del Contexto;
– Las Máquinas de Turing, que reconocen a los Lenguajes Sensibles al Contexto y a los Lenguajes Irrestrictos.

* Ejercicio 2 *
(a) Investigue e informe si un Autómata Finito con Pila puede reconocer a un Lenguaje Regular.
(b) Investigue e informe si una Máquina de Turing puede reconocer a un Lenguaje Regular.

En este capítulo veremos una importante introducción a los Autómatas Finitos y su aplicación. Estos autómatas tienen diversas aplicaciones en el reconocimiento de textos. Una de estas aplicaciones, y muy útil, es la construcción del módulo del compilador llamado Analizador Léxico, que veremos

en este volumen. Por ello, el presente capítulo es importante por sí mismo y, además, es una buena introducción para una mejor comprensión de una aplicación a nivel de compiladores.

En el próximo capítulo estudiaremos los Autómatas Finitos con Pila, que también tienen aplicación en el diseño de un compilador, tema que nos ocupa en este volumen.

En el Volumen 3 analizaremos operaciones fundamentales en las que intervienen Autómatas Finitos, y, finalmente, veremos una introducción a la Máquina de Turing para completar los autómatas que aparecen en la Jerarquía de Chomsky.

1.1 ¿QUÉ ES UN AUTÓMATA FINITO?

Un Autómata Finito (AF) es una herramienta abstracta que se utiliza para reconocer un determinado LR. En otras palabras: un AF es un modelo matemático de un sistema que recibe una cadena constituida por caracteres de cierto alfabeto Σ y tiene la capacidad de determinar si esa cadena pertenece al LR que el AF reconoce.

➔ RECONOCER un LR significa aceptar cada cadena que es una palabra del LR y rechazar cada cadena que no pertenece al lenguaje.

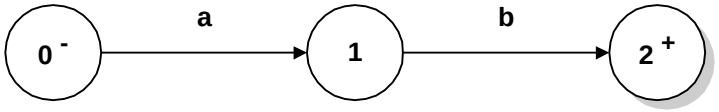
Para realizar esta tarea de reconocimiento, el AF recorre, uno por uno, los caracteres que constituyen la cadena con la que ha sido "alimentado". Cada carácter procesado produce un cambio de estado en el AF. Si al terminar de analizar todos los caracteres de la cadena, el AF se encuentra en un estado especial llamado ESTADO FINAL o ESTADO DE ACEPTACIÓN, entonces se afirma que el AF ha reconocido a la cadena y que, por lo tanto, ésta es una palabra del lenguaje; caso contrario, la cadena es rechazada porque no pertenece al LR tratado. Un AF puede tener varios estados finales.

Existe otro estado especial en todo AF: el estado en el que se encuentra antes de comenzar su actividad. A este estado se lo denomina ESTADO INICIAL y tiene la característica de ser único, es decir: cada AF tiene un único estado inicial.

Todo AF tiene asociado un digrafo, llamado DIAGRAMA DE TRANSICIONES (DT), que permite visualizar, con mayor claridad, el funcionamiento del AF. En este diagrama, los nodos del digrafo representan los diferentes estados del AF, y los arcos, que representan las transiciones entre los estados, informan cada cambio de estado que ocurre en el AF según el carácter que "leyó" de la cadena en estudio. Por ello, las TRANSICIONES estarán etiquetadas con símbolos del alfabeto.

Cada estado del AF tiene un nombre, habitualmente representado por un número entero no negativo o una letra, aunque puede ser cualquier otro. El estado inicial se distingue porque se le agrega un supraíndice "menos" (–) a su nombre, y cada estado final se diferenciará de los otros estados mediante el agregado de un supraíndice "más" (+).

Ejemplo 1
Sea el LR finito representado por la ER ab; este lenguaje contiene solo la palabra ab. El AF que reconoce este lenguaje tiene asociado el siguiente Diagrama de Transiciones:



Este AF está formado por tres estados y dos transiciones entre los estados. Los estados son: 0 (que es el estado inicial), 1 (que es un estado intermedio) y 2 (que es el estado final o estado de aceptación). Analicemos distintas situaciones por las que puede atravesar este AF.

Primer caso: ¿La cadena ab pertenece al LR reconocido por este AF?
El AF comienza su actividad en el estado inicial (el estado 0) y es "alimentado" con una cadena, aquella que debe reconocer o rechazar. Supongamos que deseamos que el AF determine si la cadena ab pertenece al lenguaje que es reconocido por este AF. Entonces, el AF lee el primer carácter (a) y desde el estado 0 transita al estado 1, es decir: cambia de estado. Ahora el AF se encuentra en el estado llamado 1 y lee el siguiente carácter de la cadena analizada (b). De acuerdo al DT, el AF cambia al estado 2. El AF no encuentra más caracteres en la cadena analizada y termina su actividad estando en un estado de aceptación (estado final). Por lo tanto, se afirma que el AF ha detectado que la cadena ab es una palabra del LR que este autómata reconoce.

> La ACTIVIDAD desarrollada por el AF al analizar la cadena ab puede ser descripta de la siguiente manera: 0^- => a => 1 => b => 2^+ => RECONOCE.

Segundo caso: ¿La cadena a es reconocida?
Como ya hemos visto, este AF tiene una transición desde su estado inicial (estado 0), por el carácter a, al estado 1. En ese momento, el autómata se encuentra en el estado 1 y no tiene más caracteres para leer. Por lo tanto, la actividad del AF termina en el estado 1 que, como se aprecia, no es un estado final. En consecuencia, se afirma que este AF rechaza a la cadena a porque ésta no forma parte del LR reconocido por el AF.

> Descripción de la ACTIVIDAD del autómata: 0^- => a => 1 => RECHAZA.

Tercer caso: ¿La cadena abab es reconocida por este AF?
El autómata comienza en el estado 0, lee la primera a y transita al estado 1. Luego, lee la primera b y cambia al estado 2. Continúa la actividad del AF y lee el tercer carácter de la cadena, es decir, la segunda a. Pero desde el estado 2 no existen transiciones, por lo que el AF no puede seguir "trabajando". En consecuencia, la cadena abab no es reconocida por este AF, a pesar de encontrarse, *momentáneamente*, en un estado final. La cadena tiene más caracteres que todavía no fueron tratados por el AF y éste no sabe cómo procesarlos.

> Descripción de la ACTIVIDAD del AF: 0^- => a => 1 => b => 2 => a => ?? => RECHAZA.

Cuarto caso: Otra situación de rechazo se produce si la cadena comienza con b como, por caso, la cadena baa. El AF inicia su actividad en el estado 0, el estado inicial, y no encuentra transición alguna por el símbolo b. Por lo tanto, el autómata no puede recorrer la cadena con la que se lo ha alimentado y, consecuentemente, no la reconoce.

> Descripción de la ACTIVIDAD del AF: 0^- => b => ?? => RECHAZA.

➔ Los cuatro casos presentados en el ejemplo anterior muestran las diferentes situaciones por las que puede atravesar un AF. Ante todo, un AF puede tener éxito o puede fracasar: tiene éxito si reconoce a la cadena "dato" como palabra del LR que el AF reconoce, y fracasa en caso contrario. Para que el AF tenga éxito, existe una única posibilidad: leer todos los caracteres que forman la cadena y finalizar su actividad en un estado final. Si fracasa, en cambio, puede ser por tres motivos:

(a) lee todos los caracteres de la cadena pero finaliza su actividad en un estado no final, como se aprecia en el segundo caso analizado; o (b) no puede leer todos los caracteres de la cadena, como ocurre en los casos tercero y cuarto; o (c) llega al estado final pero la cadena tiene más caracteres que el AF no puede procesar, como en el tercer caso.

➔ Cada AF reconoce un ÚNICO LR. En cambio, para un LR pueden existir muchos AFs que lo reconozcan, como veremos más adelante.

* Ejercicio 3 *
Sea el alfabeto {a, b} y sea el LR {a, aba}.
(a) Dibuje el DT de un AF que reconoce este LR.
(b) Describa los diferentes casos posibles y sus actividades, como se ha hecho en el Ejemplo 1.

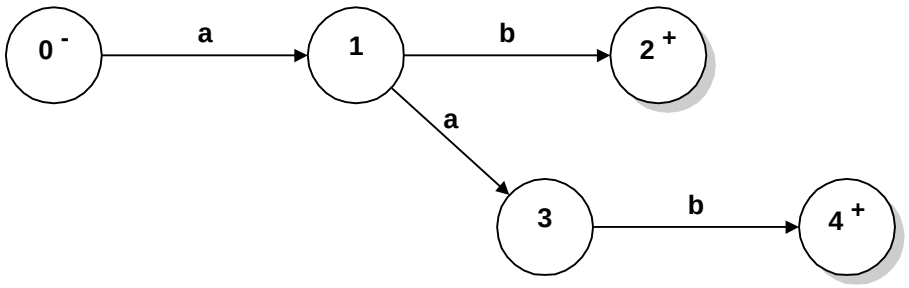
1.2 AUTÓMATAS FINITOS QUE RECONOCEN LENGUAJES REGULARES FINITOS

Los LRs finitos se representan mediante ERs que utilizan tres operadores: concatenación, unión y potencia, aunque éste último solo es un simplificador de la aplicación reiterada del operador concatenación.

En el ejemplo anterior hemos visto cómo se representa una concatenación en el DTs de un AF: es, simplemente, una transición concatenada con otra transición. Analice el diagrama dibujado y verifique que la concatenación de los caracteres a y b que forman la palabra ab se transforma en dos transiciones consecutivas: la primera etiquetada con el carácter a y la segunda etiquetada con el carácter b.

El DT para el operador unión es un poco más complicado porque requiere que, desde un estado, partan dos (o más) transiciones a, normalmente, diferentes estados.

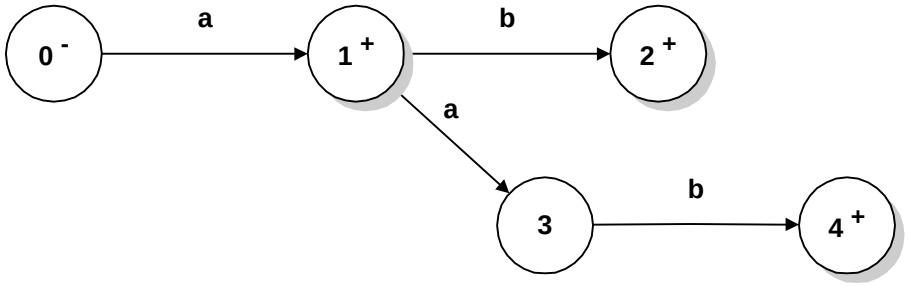
Ejemplo 2
Sea la ER ab+aab, la cual representa un LR con dos palabras. Dado que ambas palabras comienzan con el carácter a, esta expresión también puede escribirse así: a(b+ab). Si utilizamos esta segunda expresión, el DT del AF que reconoce a este LR lo podemos dibujar de la siguiente manera:



La primera a, que es común a ambas palabras, "mueve" al AF desde el estado inicial al estado intermedio 1. Aquí aparece la bifurcación que implementa al operador *unión*: un camino conduce al estado final 2, para reconocer a la palabra ab, y el otro camino conduce al estado 3 y luego al estado final 4, para reconocer a la palabra aab.

➔ Todo AF es construido sobre un ALFABETO determinado, el cual coincide, por ahora, con el del LR que reconoce. La cadena que el AF debe procesar sólo puede contener caracteres de este alfabeto. Los caracteres que etiquetan las transiciones en el DT de un AF son los que constituyen el alfabeto en cuestión.

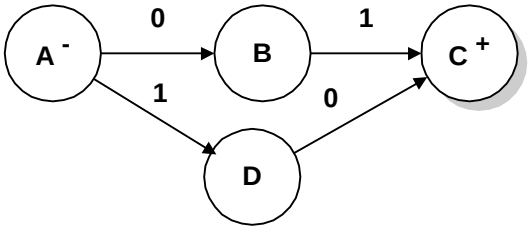
Ejemplo 3
Sea la ER a+ab+aab. Esta ER se ha formado agregándole la palabra a a la ER del ejemplo anterior. Dada la similitud que existe entre ambas ERs, es de suponer que también serán similares los DTs de los AFs que reconocen a los LRs representados. Efectivamente, así es. El nuevo AF, sobre el alfabeto {a, b}, tiene el siguiente DT:



La única, pero muy importante, diferencia que existe entre este AF y el del ejemplo anterior es que, ahora, el estado 1 debe ser un estado final para que se pueda reconocer a la palabra a. Observe que la denominación *estado final* no significa, necesariamente, "último estado del AF". Su significado es más amplio: estado de aceptación, también llamado estado de reconocimiento, como ocurre, en este caso, con el estado 1. [Posiblemente, ya lo encontró en la resolución del Ejercicio 3.]

No es obligatorio que un AF tenga un estado final para cada palabra del LR reconocido, como se ve en el siguiente caso:

Ejemplo 4
Se construye el siguiente AF sobre el alfabeto {0,1} que reconoce un LR que tiene dos palabras. El AF tiene un solo estado final:



* Ejercicio 4 *
¿Qué LR reconoce este AF? Escriba una ER que lo represente.

Como conclusión de esta sección sobre AFs que reconocen Lenguajes Regulares finitos, observe que una palabra de longitud 1 es reconocida por un autómata con una transición y dos estados, una palabra de longitud 2 es aceptada por un autómata con dos transiciones y tres estados, etc.

1.3 AUTÓMATAS FINITOS QUE RECONOCEN LENGUAJES REGULARES INFINITOS

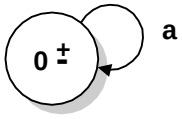
En la sección anterior hemos visto cómo se implementan, en el diseño de un AF, los operadores de concatenación y de unión. Las ERs que representan LRs infinitos requieren la utilización del operador estrella de Kleene, también llamado simplemente operador estrella, el cual completa la trilogía de operadores básicos para la construcción de estas expresiones.

➔ El término *finito* que se agrega a la palabra *autómata* no significa que estas herramientas solo pueden reconocer lenguajes finitos, sino que el número de estados de un autómata es finito.

En un AF, la implementación del operador estrella aplicado a un solo carácter consiste de un CICLO (o bucle) sobre un solo estado.

➔ Cada transición tiene un estado de partida y un estado de llegada. En un CICLO, el estado de partida coincide con el estado de llegada.

Ejemplo 5
La ER a*, que representa al lenguaje $\{a^n / n \geq 0\}$, es reconocida por el siguiente AF:



Este AF, a pesar de tener un solo estado, es mucho más poderoso que los autómatas con varios estados que se han construido en los ejemplos anteriores, ya que reconoce un LR infinito.

El LR que acepta este AF incluye a la palabra vacía, palabra que no tiene caracteres. Esto significa que, sin leer carácter alguno, el AF debe "colocarse" en un estado final que reconozca a esta palabra tan particular. Esto se puede lograr si el estado inicial también es estado final, circunstancia que se representa mediante el símbolo ± (también se puede escribir –+) que superindica al estado 0.

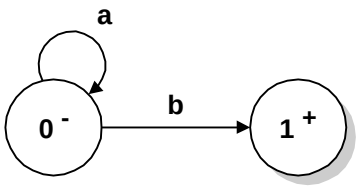
Descripción de la ACTIVIDAD de este AF:
(1) cadena vacía: $0^- \Rightarrow 0^+ \Rightarrow$ RECONOCE (reconoce a la palabra vacía)
(2) cadena aa: $0^- \Rightarrow a \Rightarrow 0 \Rightarrow a \Rightarrow 0^+ \Rightarrow$ RECONOCE (reconoce a la palabra aa)
Dado que el alfabeto es {a}, este AF reconocerá cualquier cadena con la que sea alimentado. En otras palabras: el AF diseñado reconoce el LR Universal sobre el alfabeto {a}.

➔ Sugerencias para diseñar un DT:

(1) Separe siempre el análisis de las palabras de longitud mínima (palabras particulares), del análisis de varias palabras que representan situaciones generales;

(2) Analice también cadenas que no son palabras del lenguaje para verificar que no las reconoce.

Ejemplo 6
Supongamos el LR representado por la ER a*b; las tres palabras de menor longitud son: b, ab y aab. Para dibujar el DT del AF que reconoce a este LR, debemos tener en cuenta que el autómata debe funcionar para cualquier cadena que comience con un número indeterminado de aes (inclusive ninguna) y que termine con una b. Esto se representa así:



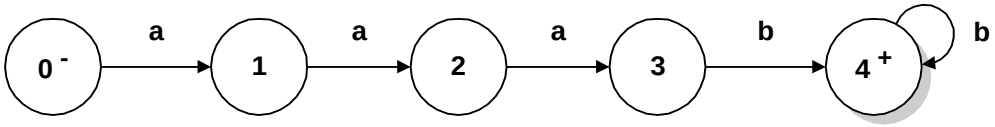
Visualicemos la ACTIVIDAD de este AF en el reconocimiento de ciertas palabras del LR y en el rechazo de algunas cadenas que no pertenecen al lenguaje:

(1) Palabra b: 0^- => b => 1+ => RECONOCE
(2) Palabra aaab: 0^- => a => 0 => a => 0 => a => 0 => b => 1+ => RECONOCE
(3) Cadena aaba: 0^- => a => 0 => a => 0 => b => 1 => a => ?? => RECHAZA
(4) Cadena aaa: 0^- => a => 0 => a => 0 => a => 0 => RECHAZA (el estado 0 no es final)
(5) Cadena baa: 0^- => b => 1 => a => ?? => RECHAZA

* Ejercicio 5 *
Diseñe un AF que reconozca la ER a*bb*a.
(a) Dibuje el DT, y
(b) Muestre las actividades para diferentes casos, como en el último ejemplo.

El otro operador que se utiliza en la construcción de ERs que representan LRs infinitos es el operador clausura positiva, representado con un supraíndice +, que es una simplificación de una operación conjunta entre una concatenación y una clausura de Kleene, como aa*, cuyas respectivas implementaciones ya conocemos.

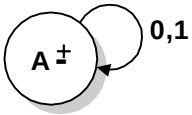
Ejemplo 7
Sea la ER a^3b^+, forma abreviada de la ER aaabb*. En base a esta última expresión, el AF que reconoce al LR representado por esta ER tiene el siguiente DT:



* Ejercicio 6 *
Describa las diferentes actividades de este AF.

Esta sección finaliza con una introducción a la implementación de AFs que reconozcan ERs más complejas. En primer lugar, se presenta la construcción de un AF que reconoce una Expresión Regular Universal (ERU).

Ejemplo 8
Sea la ERU (0+1)*. El AF más simple que reconoce a esta expresión es:

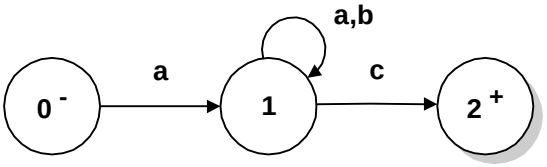


La notación 0,1 – que significa "0 o 1" – etiqueta la única transición dibujada en este AF, un ciclo, pero es una simplificación de la existencia de dos transiciones: una etiquetada con el carácter 0, y la otra etiquetada con el carácter 1. Observemos el siguiente análisis de la actividad del autómata:

(1) palabra vacía: A⁻ => A+ => RECONOCE
(2) palabra 0: A⁻ => 0 => A+ => RECONOCE
(3) palabra 111: A⁻ => 1 => A => 1 => A => 1 => A+ => RECONOCE
(4) palabra 01101: A⁻ => 0 => A => 1 => A => 1 => A => 0 => A => 1 => A+ => RECONOCE

➔ Si un AF reconoce a un Lenguaje Universal sobre cierto alfabeto (como el del ejemplo anterior), ninguna cadena podrá ser rechazada.

Ejemplo 9
Sea la ER a(a+b)*c, en la que la ERU sobre {a,b} es uno de sus factores de esta expresión. El AF más simple que reconoce a este LR tiene el siguiente DT:



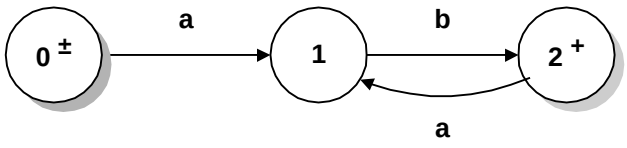
Este DT muestra, en primer lugar, que cualquier palabra reconocida por este autómata debe comenzar con el carácter a (transición del estado 0 al estado 1) y que debe terminar con el carácter c (transición del estado 1 al estado 2). En segundo lugar, se observa que hay un ciclo (bucle) en el estado 1 etiquetado con los caracteres a y b.

Descripción de la ACTIVIDAD del AF:
(1) palabra ac (es la palabra de mínima longitud): 0⁻ => a => 1 => c => 2+ => RECONOCE
(2) palabra abaac: 0⁻ => a => 1 => b => 1 => a => 1 => a => 1 => c => 2+ => RECONOCE
(3) cadena abb: 0⁻ => a => 1 => b => 1 => b => 1 => RECHAZA (no es un estado final)
(4) cadena vacía: 0⁻ => 0 => RECHAZA (no es un estado final)
(5) cadena acc: 0⁻ => a => 1 => c => 2 => c => ?? => RECHAZA

* Ejercicio 7 *
Sea el LR representado por la ER (a+b+c+d)*.
(a) Dibuje el DT de un AF que reconozca este LR;
(b) Describa la actividad del AF diseñado.

Otra situación interesante se presenta cuando el operador clausura de Kleene actúa sobre una concatenación, como en el siguiente caso:

Ejemplo 10
Sea la ER (ab)*. Esta expresión representa al LR que contiene la palabra vacía y toda otra palabra que se forma con secuencias del par de caracteres ab. En consecuencia, el AF que implemente a esta ER debe reconocer tanto a la palabra vacía (caso particular) como al caso general citado. Un DT válido es el siguiente:

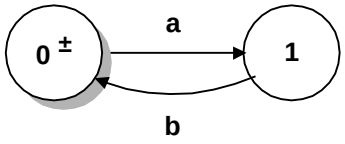


Observe el *ciclo* formado por las transiciones que relacionan a los estados 1 y 2, para reconocer los siguientes pares ab (desde el segundo par en adelante).

Descripción de la ACTIVIDAD del AF:
(1) palabra vacía: 0^- => 0+ => RECONOCE
(2) palabra ab: 0^- => a => 1 => b => 2+ => RECONOCE
(3) palabra abab: 0^- => a => 1 => b => 2 => a => 1 => b => 2+ => RECONOCE
(4) cadena aba: 0^- => a => 1 => b => 2 => a => 1 => RECHAZA (no es estado final)
(5) cadena abb: 0^- => a => 1 => b => 2 => b => ?? => RECHAZA

* Ejercicio 8 *
Dado el AF del ejemplo anterior, describa su actividad para la cadena abababa.

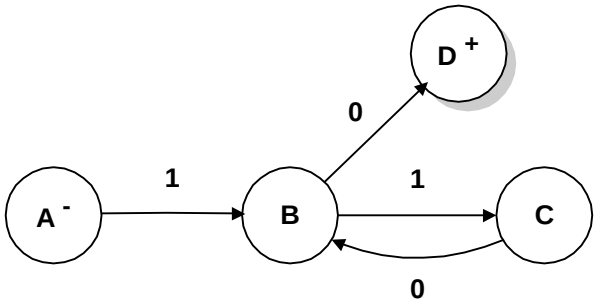
Ejemplo 11
Otro AF que reconoce a la misma ER (ab)* tiene el siguiente DT:



* Ejercicio 9 *
Describa la actividad de este AF para las mismas palabras y cadenas del Ejemplo 10. (vacía, ab, abab, aba, abb

Finalmente, se presenta un ejemplo de un AF que reconoce a una ER en la que uno de sus factores es una clausura de la forma analizada en los dos ejemplos anteriores:

Ejemplo 12
Sea la ER 1(10)*0. Un AF que la reconoce tiene el siguiente DT:



Descripción de la actividad del autómata:
(1) palabra 10 (palabra de longitud mínima): A^- => 1 => B => 0 => D+ => RECONOCE
(2) palabra 1100: A^- => 1 => B => 1 => C => 0 => B => 0 => D+ => RECONOCE
(3) cadena 110: A^- => 1 => B => 1 => C => 0 => B => RECHAZA (no es un estado final)
(4) cadena 01: A^- => 0 => ?? => RECHAZA

* Ejercicio 10 *
(a) Dibuje el DT de un AF que reconoce el lenguaje $a(ab)^+$.
(b) Describa la actividad del AF para diferentes palabras y cadenas.

Esta ha sido solo una introducción al diseño de AFs. Se puede agregar que, además, se trata de una introducción informal, ya que, como veremos en el Volumen 3, existen algoritmos que nos permitirán implementar formalmente a los AFs que reconocen determinadas ERs, por más complicadas que estas expresiones sean. Por ejemplo, el diseño de un AF que reconozca al LR representado por la ER a*+ab no es tan sencillo como parece.

* Ejercicio 11 *
(a) Dibuje un DT de un AF que reconozca al LR representado por la ER a*+ab.
(b) Describa la actividad del AF para varias palabras y cadenas.

➔ Un Autómata Finito reconoce a un Lenguaje Regular cuando acepta cada palabra del lenguaje y rechaza toda cadena que no es una palabra del lenguaje.

1.4 AUTÓMATAS FINITOS DETERMÍNISTICOS

Como ya se ha dicho, un AF es un mecanismo que RECONOCE a un determinado LR. "Reconocer a un lenguaje" significa *aceptar* cada cadena que pertenece al lenguaje – es decir, cada cadena que es una palabra – y *rechazar* toda cadena que no pertenece al lenguaje.

Todos los DTs dibujados hasta el momento pertenecen al grupo de AFs denominado AUTÓMATAS FINITOS DETERMINÍSTICOS (AFDs), cuya característica funcional es que para cualquier estado en que se encuentre el autómata en un momento dado, la lectura de un carácter determina, SIN AMBIGÜEDADES, cuál será el estado de llegada en la próxima transición.

Existe otro grupo de AFs, el de los AUTÓMATAS FINITOS NO DETERMINÍSTICOS (AFNs), que no cumple con lo indicado en el párrafo anterior y que será tratado en el Volumen 3.

Se ha especificado que un AF es determinístico cuando, dado un carácter que es "leído" por el autómata, éste transita desde un estado de partida a un estado de llegada preciso, es decir: el estado de llegada está unívocamente determinado por el estado de partida y el carácter leído por el autómata. En otras palabras: cada estado del AF tiene cero o una transición por cada carácter del alfabeto reconocido por el autómata.

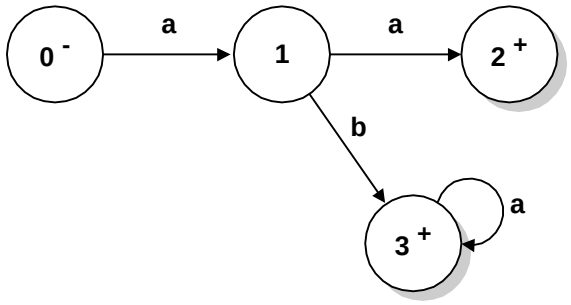
Un AFD es una colección de tres conjuntos, como ya hemos apreciado en las secciones anteriores:

(1) Un conjunto finito de estados, uno de los cuales se designa como el estado inicial (único) y otros (uno o más) son designados como estados finales o estados de aceptación;

(2) Un alfabeto de caracteres con los que se forman las cadenas que serán procesadas por el AFD;

(3) Un conjunto finito de transiciones que indican, para cada estado y para cada carácter del alfabeto, a qué estado debe "moverse" el AFD.

Ejemplo 13
A continuación se dibuja el DT de un AFD formado por:
(a) Conjunto de estados: 0, el estado inicial; 1, un estado intermedio; 2 y 3, estados finales.
(b) Alfabeto: los caracteres a y b.
(c) Conjunto de transiciones: i) desde el estado 0 al estado 1 por el carácter a, ii) desde el estado 1 al estado 2 por el carácter a y al estado 3 por el carácter b, y iii) en el estado 3, un ciclo por el carácter a. Esta descripción se representa más claramente de esta manera:
i) 0 => a = 1; ii) 1 => a => 2 y 1 => b => 3; iii) 3 => a => 3.



1.4.1 DEFINICIÓN FORMAL DE UN AFD

Formalmente, un AFD es una 5-upla (Q, Σ, T, q_0, F), donde:
– Q es un conjunto finito de estados,
– Σ es el alfabeto de caracteres reconocidos por el autómata,
– $q_0 \in Q$ es el estado inicial (único, no es un conjunto),
– $F \subseteq Q$ es el conjunto de estados finales, y
– T: Q x Σ -> Q es la función de transiciones; es decir: T (q, x) = z significa que z es el estado al cual transita el autómata desde el estado q, al leer el carácter x, transición que también podemos simbolizar así: q => x => z. En esta última notación se ve más claramente cuál es el estado de partida y cuál es el estado de llegada en cada transición.

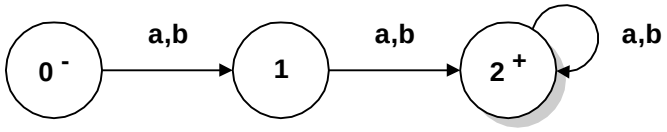
➔ Generalmente, la función de transiciones es representada por medio de una TABLA DE TRANSICIONES (TT), tal como se muestra en el próximo ejemplo.

Ejemplo 14
El AFD del Ejemplo 13 se define formalmente de la siguiente manera:
M = (Q, Σ, T, q_0, F), donde:
Q = {0,1, 2, 3};
Σ = {a, b};
q_0 = 0;
F = {2,3};
T = {0 => a => 1, 1 => a => 2, 1 => b => 3, 3 => a => 3}.
Si la función de transiciones T es representada mediante una TT, obtenemos el siguiente resultado:

TT	a	b
0-	1	-
1	2	3
2+	-	-
3+	3	-

Esta tabla tiene una fila por cada estado del autómata y una columna por cada símbolo del alfabeto. Se lee así: desde el estado 0 (inicial), por a transita al estado 1 y por b no hay transición; desde el estado 1, por a transita al estado 2 y por b transita al estado 3; el estado 2 (final) no tiene transiciones; y desde el estado 3 (final), por a tiene un ciclo y por b no hay transición, como se puede corroborar observando en el DT dibujado en el Ejemplo 13.

Ejemplo 15
Supongamos el lenguaje "Todas las palabras de aes y/o bes que tienen por lo menos dos letras". Las que siguen son las cuatro palabras de menor longitud de este lenguaje: aa, bb, ab, ba. El diagrama de transiciones de un AFD que reconoce este LR infinito es:



En este DT, todas las transiciones tienen la particularidad de estar etiquetadas con a,b. En realidad, cada etiqueta a,b significa dos transiciones diferentes: una por el carácter a y otra por el carácter b, ambas con el mismo estado de partida y el mismo estado de llegada. El uso de la etiqueta a,b simplifica el dibujo.

La definición formal de este AFD es $M = (Q, \Sigma, T, q_0, F)$, donde:
$Q = \{0, 1, 2\}$; $\Sigma = \{a, b\}$; $q_0 = 0$; $F = \{2\}$; T representada por la siguiente tabla de transiciones:

TT	a	b
0-	1	1
1	2	2
2+	2	2

Una diferencia inmediata que se observa al comparar las TTs de los Ejemplos 14 y 15, es que la TT del Ejemplo 14 tiene "huecos" – cuatro, para ser más preciso – que indican ausencia de transiciones, mientras que la tabla del Ejemplo 15 está COMPLETA.

* Ejercicio 12 *
Escriba la definición formal un AFD que reconozca el LR "Todas las palabras sobre el alfabeto {a,b,c} que tienen por lo menos tres letras". Represente la función de transiciones mediante una TT.

1.4.2 AFD COMPLETO

➔ Un AFD es completo si cada estado tiene exactamente una transición por cada carácter del alfabeto.

Otra definición equivalente es:

➔ Un AFD es completo cuando su tabla de transiciones no tiene "huecos"; si los tiene, el AFD es incompleto.

La noción de AFD completo es muy importante en las aplicaciones computacionales de los AFDs, porque, como veremos más adelante, un AFD se implementa mediante una matriz que representa a la TT y esta matriz, obviamente, no puede tener elementos sin información. En consecuencia, si el

AFD constituye una herramienta que será implementada y no es completo, la primera acción a realizar es completarlo.

COMPLETAR un AFD significa eliminar los "huecos" de su TT. Para ello, debemos llevar a cabo los siguientes pasos:

(1) Se agrega un nuevo estado, al que denominamos ESTADO DE RECHAZO o ESTADO DE NO ACEPTACIÓN;

(2) Se reemplaza cada "hueco" que hay en la TT por una transición a este nuevo estado; y

(3) Se incorpora una nueva entrada en la TT para el estado de rechazo, en la que se representarán ciclos para todos los caracteres del alfabeto; con estos ciclos se informa que, una vez que el AF se sitúa en el estado de rechazo, de él "no puede salir".

Ejemplo 16
Se desea completar el AFD del Ejemplo 14: $M = (Q, \Sigma, T, q_0, F)$, donde: $Q = \{0,1,2,3\}$; $\Sigma = \{a,b\}$; $q_0 = 0$; $F = \{2,3\}$; y T representada por la TT:

TT	a	b
0-	1	-
1	2	3
2+	-	-
3+	3	-

Para completar el AFD, agregamos un estado de rechazo (supongamos que este estado se llama 4). Entonces, el nuevo AFD es: $M_2 = (Q, \Sigma, T, q_0, F)$, con $Q = \{0, 1, 2, 3, 4\}$; $\Sigma = \{a, b\}$; $q_0 = 0$; $F = \{2, 3\}$; y T representada por la TT:

TT	a	b
0-	1	4
1	2	3
2+	4	4
3+	3	4
4	4	4

Observe que la lectura de un carácter erróneo provoca que el AFD transite al estado de rechazo (el estado 4), del cual nunca podrá salir porque, como se distingue en la última fila, este estado tiene un ciclo a sí mismo por cualquier carácter del alfabeto.

Se ha obtenido, entonces, un nuevo AFD, con un nuevo conjunto de estados Q y una nueva función de transiciones T, que reconoce el mismo lenguaje que es aceptado por el Autómata Finito del cual se partió. Este hecho conduce a la siguiente definición:

➔ Dos AFDs son EQUIVALENTES si reconocen el mismo Lenguaje Regular.

Nota 1
Si bien desde el punto de vista teórico los AFDs no están obligados a ser completos, desde el punto de vista práctico es necesario trabajar con AFDs completos. Por lo tanto, de aquí en más adoptaremos el siguiente criterio: Cuando construimos un AFD y dibujamos su DT, no agregamos el estado de rechazo para que el dibujo quede más "limpio" y así sea más sencilla su interpretación; pero cuando este DT es representado mediante la TT, esta TT debe ser completa, porque ciertas operaciones que veremos en el Volumen 3 y, en general, el ámbito computacional lo exigen.

* Ejercicio 13 *
Sea el AFD definido formalmente de la siguiente manera: $M = (Q, \Sigma, T, q_0, F)$, donde:

$Q = \{0,1, 2, 3\}$;
$\Sigma = \{a,b,c\}$;
$q_0 = 0$;
$F = \{2,3\}$;
$T = \{0 => a => 1, 1 => c => 2, 1 => b => 3, 3 => a => 3\}$.

(a) Describa, formalmente, el AFD equivalente con la TT completa.
(b) Escriba una ER que represente al LR reconocido por este AFD.
(c) Describa las actividades de este AFD para todos los casos posibles.

1.5 UNA APLICACIÓN: VALIDACIÓN DE CADENAS

Los AFs reconocen LRs. En consecuencia, toda aplicación que requiera el RECONOCIMIENTO de LRs podrá resolverse mediante la construcción y el procesamiento de AFs que acepten a estos lenguajes.

Una aplicación en la que resulta muy útil contar con un AF se puede definir así:

"determinar si una cadena dada corresponde a un patrón preestablecido
que corresponde a un lenguaje infinito"

En otros términos, esto significa verificar si una cadena es una palabra de un LR infinito dado.

Ejemplo 17
Consideremos el LR: "Todos los números enteros en base 10 que pueden estar precedidos por un signo". Algunas palabras de este lenguaje son: 6, 1234, –47822, +9876543210, 000. Algunas cadenas que no pertenecen a este lenguaje son: 123A, *66, 14.23. La verificación de que una cadena dada representa una palabra del lenguaje enunciado ("un número entero en base 10"), es un problema que se puede resolver con un AF.

Para resolver el problema planteado debemos *simular*, mediante un algoritmo adecuado, el *funcionamiento* de un AF que reconoce este LR. Entonces, se comienza construyendo un AFD completo que reconoce al LR dado.

Posteriormente, se aplica el siguiente algoritmo que recorre todos los caracteres de la cadena, para luego determinar si es o no una palabra del LR:

Algoritmo 1

```
- Carácter a analizar: primer carácter de la cadena
- Estado actual del autómata: estado inicial
- Mientras haya caracteres en la cadena, repetir:
  (1) Determinar el nuevo estado actual (estado de llegada de la
      transición)
  (2) Actualizar el carácter a analizar
- Si el último estado actual es un estado final, entonces la cadena
  procesada es una palabra del lenguaje; caso contrario, la cadena no
  pertenece al lenguaje.
```

* Ejercicio 14 *
(a) Dibuje el DT e implemente la TT que resuelve el problema del Ejemplo 17, considerando que el alfabeto original son todos los caracteres ASCII. Para simplificar, supondremos que el alfabeto del AFD está formado por tres símbolos: S, que representa a cualquiera de los dos signos; D, que representa a cualquiera de los diez dígitos decimales; y O, que representa a cualquier otro carácter.
(b) Recorra el Algoritmo 1 para la cadena –1234;
(c) Recorra el Algoritmo 1 para la cadena 12*34.
(d) Implemente el Algoritmo 1 mediante una función en ANSI C, suponiendo que ya existe una función auxiliar tal que, dado un carácter, retorna el número de columna que le corresponde en la TT del AFD construido.
(e) Implemente, en ANSI C, la función auxiliar utilizada en el ítem anterior.

Existe otro algoritmo que termina el proceso, antes de recorrer la cadena completa, si detecta que el estado actual es el estado de rechazo:

Algoritmo 2

```
- Carácter a analizar: primer carácter de la cadena
- Estado actual del autómata: estado inicial
- Mientras haya caracteres en la cadena y el estado actual no sea el de
  rechazo, repetir:
  (1) Determinar el nuevo estado actual (estado de llegada de la
      transición)
  (2) Actualizar el carácter a analizar
- Si el último estado actual es un estado final, entonces la cadena
  procesada es una palabra del lenguaje; caso contrario, la cadena no
  pertenece al lenguaje.
```

* Ejercicio 15 *
a) Resuelva el problema del Ejemplo 17 utilizando el Algoritmo 2.
b) Implemente este algoritmo mediante una función en ANSI C.

Un Análisis Elemental del Algoritmo que Conviene Utilizar

(1) En el Algoritmo 2, la condición del "mientras" es más compleja que la condición del Algoritmo 1; esta mayor complejidad implica más tiempo de procesamiento en ciertas condiciones.

(2) Por ello, si la gran mayoría de las cadenas a analizar son palabras del LR, conviene usar el Algoritmo 1. Si se da la situación inversa, es recomendable usar el Algoritmo 2.

* Ejercicio 16 *
a) ¿Qué ocurre si las cadenas son muy largas y hay una probabilidad de error muy alta en los primeros caracteres de la cadena? Investigue e informe.
b) ¿Y qué ocurre si las cadenas son muy largas y hay una gran probabilidad de error en los últimos caracteres de la cadena? Investigue e informe.

Para concluir esta sección, incorporamos un concepto que será clave en muchas de las situaciones que se presenten en este y en próximos capítulos: el centinela. Cuando existe una secuencia de caracteres que debemos procesar desde el primero hasta el último carácter, necesitamos conocer cuándo termina esa secuencia. Existen dos formas de hacerlo: a) conocemos la longitud de la secuencia, o b) la secuencia termina con un *centinela*.

Entonces, un centinela es un carácter o un grupo de caracteres que indica la terminación de una secuencia de caracteres, pero no forma parte de esta secuencia.

Ejemplo 18

a) Un archivo de texto es una secuencia de caracteres que termina con un carácter o grupo de caracteres conocido como "*end of file*" (eof). El *eof* no pertenece al contenido del archivo a procesar y actúa como centinela.
b) En ANSI C, la cadena `"abc"` se implementa, internamente con estos tres caracteres más el carácter nulo; esto es: `abc\0`. El carácter nulo actúa como centinela.

1.6 UNA EXTENSIÓN DE LA APLICACIÓN: UNA SECUENCIA DE CADENAS

Finalizamos este capítulo con una aplicación que extiende la que fue analizada en la sección anterior: la entrada ya no es una sola cadena sino que es un texto. Esta aplicación prepara el terreno para el estudio del Análisis Léxico en un compilador, tema que veremos a partir del capítulo 3.

Llamamos texto a una secuencia de caracteres, formada, generalmente, por varias cadenas, que termina con un centinela. El *texto* puede encontrarse en un soporte interno (un vector) o en un soporte externo (un archivo de texto). En este último caso, también lo llamaremos flujo de entrada.

Supongamos que existe un texto formado por cadenas separadas entre sí con un carácter numeral (#); obviamente, este carácter # actúa como centinela para cada cadena con excepción de la última cadena del texto.

* Ejercicio 17 *
Investigue e informe cuál será el centinela de la última cadena del texto.

Sea el LR definido en el Ejemplo 17: "Todos los números enteros en base 10 que pueden estar precedidos por un signo". Debemos reconocer cuáles de las cadenas de este texto son palabras del LR descripto. Por ejemplo, supongamos que el texto está formado por estos caracteres:

```
-123#4A67#01234567#ZZZZZZ#86-0,2;4/#+1234123#b444
```

Entonces, el algoritmo que construyamos debería reconocer tres palabras en este texto.

* Ejercicio 18 *
Informe: ¿Cuáles son esas tres palabras? ¿Por qué decimos "palabras" y no "números"?

Diseñaremos un AFD que reconozca las palabras y lo implementaremos en un pseudo-código, basándonos en lo ya aplicado para definir el Algoritmo 1. Además, debemos considerar que existen cinco clases de caracteres que necesitamos distinguir, por el significado que le corresponde a cada clase:

(1) el conjunto formado por los dos signos;
(2) el conjunto formado por los diez dígitos decimales;
(3) el conjunto formado solo por el #, que actúa como centinela de una cadena y también como separador de dos cadenas consecutivas del texto;
(4) el conjunto formado por los restantes caracteres de la tabla ASCII (lo denominamos otro); y
(5) el centinela que indica el fin del texto (la llamaremos fdt).

RESOLUCIÓN

Debemos diseñar un AFD completo que resuelva este problema. Esto significa que, para cada estado, habrá una transición para todos los caracteres de cada una de las clases descriptas en el párrafo anterior; en otras palabras, de cada estado saldrán cinco transiciones, una por cada clase. La única excepción es el estado de llegada para las transiciones por fdt porque, evidentemente, el texto no tiene más caracteres a tratar. Describamos las transiciones del AFD que construimos:

TRANSICIÓN	COMENTARIO
0^- => signo => 1	Primer carácter de la cadena es un signo; es un posible número entero
1 => dígito => 2	Primer dígito de un posible número entero
2 => dígito => 2	Más dígitos de un posible número entero
2 => # => 3^+	Detectó el centinela de una cadena; ES un número entero y hay más cadenas para analizar
2 => fdt => 4^+	Detectó el centinela del texto; ES un número entero y es la última cadena del texto
0^- => dígito => 2	Primer dígito de un posible número entero
0^- => # => 0	Cadena vacía
0^- => fdt => 6	Fin del texto
0^- => otro => 5	No es un número entero
1 => # => 0	Para analizar próxima cadena, si existe
1 => signo, otro => 5	No es un número entero
1 => fdt => 6	Fin del texto
2 => signo, otro => 5	No es un número entero
3 => signo => 1	Primer carácter de la cadena es un signo; es un posible número entero
3 => dígito => 2	Primer dígito de un posible número entero
3 => # => 0	Para analizar próxima cadena, si existe
3 => otro => 5	No es un número entero
3 => fdt => 6	Fin del texto
5 => signo, dígito, otro => 5	No es un número entero
5 => # => 0^-	Para analizar próxima cadena, si existe
5 => fdt => 6	Fin del texto

* Ejercicio 19 *
Desde el estado 2 salen dos transiciones que llegan a dos estados finales diferentes. ¿No podríamos, como ocurre con otras transiciones, etiquetarlas con #,fdt y que lleguen a un único estado final? Justifique su respuesta.

* Ejercicio 20 *
Dibuje el DT del AFD que ha sido descripto, mediante sus transiciones, en la tabla anterior.

A continuación, construimos la Tabla de Transiciones que implementa el AFD descripto:

Estado	Signo	Dígito	#	Otro	fdt
0^-	1	2	0	5	6
1	5	2	0	5	6
2	5	2	3	5	4
3^+	1	2	0	5	6
4^+					
5	5	5	0	5	6
6 (fdt)					

ANÁLISIS:

(1) Los estados 0 (inicial) y 3 (final) tienen las mismas transiciones, como se observa si se comparan las filas respectivas. Esto ocurre porque, en ambos estados, se comienza el análisis de una cadena del texto.

(2) Analicemos las columnas. La columna correspondiente a la clase signo muestra que la lectura de uno de ellos conduce al estado 1 únicamente si es el primer carácter de una cadena del texto; en los restantes casos, el estado de llegada es el estado 5. Este último estado juega el papel de "estado de rechazo o de no reconocimiento".

(3) La columna de la clase dígito muestra que mientras el carácter leído sea correcto para detectar un número entero, el estado de llegada siempre será el 2; solo será el estado 5 cuando el dígito leído esté en medio de otros caracteres y, por lo tanto, nunca pueda formar parte de un número.

(4) El carácter # es el centinela de cada cadena – excepto la última – y, además, actúa como separador de cadenas del texto. Por lo tanto, si al leerlo se detectó un número entero, realiza una transición al estado final 3; en cualquier otro caso, vuelve al estado 0 para iniciar el tratamiento de una nueva cadena.

(5) La columna de la clase otro muestra lo que ocurre cuando se lee un carácter no válido para un número entero.

(6) El centinela fdt tiene un tratamiento especial, porque señala que no hay más caracteres en el texto. Pero se debe separar el caso en que la última cadena del texto es una constante entera de los restantes casos. Y esta diferenciación aparece claramente expresada por los estados 4 y 6.

A continuación, veamos un algoritmo en pseudo-código que implementa una solución a este problema.

Algoritmo 3

```
- Intenta leer primer carácter del texto (porque el texto puede estar
  vacío)
- Mientras no sea fdt, repetir:
  (1) Estado actual del autómata: estado inicial
  (2) Mientras no sea un estado final y no sea el estado FDT, repetir:
      (2.1) Determinar el nuevo estado actual
      (2.2) Actualizar el carácter a analizar
  (3) Si el estado es final, la cadena procesada es una constante entera;
      caso contrario, la cadena no pertenece al lenguaje.
```

* Ejercicio 21 *
a) Sea una función ANSI C cuyo prototipo es int Columna (int); Esta función recibe un carácter y produce el número de columna que le corresponde en la matriz que implementa la TT de la aplicación que estamos desarrollando. Construya esta función.
b) Con la ayuda de la función auxiliar Columna y del Algoritmo 3, escriba un programa en ANSI C que implemente la aplicación desarrollada.

* Ejercicio 22 *
Sea un texto formado por cadenas separadas con un #. Sea el LR de todas las palabras sobre el alfabeto ASCII cuyo segundo carácter es una A (mayúscula o minúscula). Diseñe e implemente un AFD que cuente cuántas palabras de este LR hay en el texto dado. Por ejemplo: si la cadena es `132bx#4azz,de—m#Azz#zAz#aaaaaa#012345` el AFD debe determinar que hay 3 palabras.

Hipótesis de Trabajo:
a) El texto a analizar termina con un centinela que denominamos fdt (fin de texto).
b) En la resolución, describimos en un pseudo-código la o las acciones que se lleven a cabo en un estado determinado. Ejemplo: en el estado inicial se debe inicializar el contador de palabras.
c) El alfabeto para este AFD está formado por cuatro elementos: i) A, que corresponde tanto a la 'A' como a la 'a'; ii) #, que separa cadenas; iii) fdt, que representa el fin del texto; iv) cq, que representa cualquier otro carácter.

* Ejercicio 23 *
a) Escriba, en ANSI C, un programa que implemente la solución del ejercicio anterior y retorne la cantidad de palabras encontradas. Considere que el texto se halla en el flujo de entrada stdin.
b) Investigue e informe qué es stdin.

2 AUTÓMATAS FINITOS CON PILA

Los AUTÓMATAS FINITOS CON PILA (AFPs), también conocidos como *autómatas "push-down"*, son más poderosos que los Autómatas Finitos porque, además de reconocer a los Lenguajes Regulares, tienen la capacidad de reconocer a los LICs, como son, por ejemplo, las expresiones aritméticas y las sentencias de un Lenguaje de Programación.

Antes de continuar, recordemos: Un LIC es un Lenguaje Formal que es generado por una GIC. La GIC tiene la característica que toda producción es del tipo:

noterminal –> (terminal + noterminal)*

Ejemplo 1
Sea el lenguaje $L1 = \{a^n b^n / n \geq 1\}$. Este lenguaje es un LIC, no es un LR.
L1 que puede ser generado por una GIC con estas producciones: S –> aSb | ab.

* Ejercicio 1 *
a) Investigue e informe porqué el lenguaje del Ejemplo 1 no es un LR.
b) Demuestre que las producciones dadas en el Ejemplo 1 generan a L1.

Volviendo al AFP, decíamos que éste tipo de autómata es más poderoso que el Autómata Finito. Esta potencialidad extra del AFP se debe a que, además de tener estados y transiciones entre los estados, como los AFs, el AFP tiene una memoria en forma de PILA (*stack*) que permite almacenar, retirar y consultar cierta información que será útil para reconocer a los LICs. En otras palabras: un AF solo tiene control sobre los caracteres que forman la cadena a analizar, mientras que un AFP controla los caracteres a analizar y la pila.

* Ejercicio 2 *
Informe cómo hace un AF para controlar los caracteres de la secuencia de entrada.

Hopcroft y Ullman [1979] afirman que el AFP es un dispositivo no-determinístico y que la versión determinística solo acepta un subconjunto de la totalidad de LICs. Pero, afortunadamente, este subconjunto incluye la sintaxis de la mayoría de los Lenguajes de Programación.

* Ejercicio 3 *
Investigue e informe qué es un AFP no-determinístico.

2.1 DEFINICIÓN FORMAL DE UN AFP

Un AFP está constituido por:

a) un flujo de entrada, infinito en una dirección, que contiene la secuencia de caracteres que debe analizar, similar a un AF;

b) un control finito formado por estados y transiciones etiquetadas, similar a un AF; y

c) una pila abstracta, que establece la gran diferencia con los AFs. Esta pila se representa como una secuencia de símbolos o caracteres tomados de cierto alfabeto, diferente al alfabeto sobre el que se construye un LIC reconocido por un AFP. En la pila, el primer carácter de la secuencia es el que se encuentra en el tope de la pila (esto es solo una convención).

Definamos formalmente un AFP como la 7-upla M = (E, A, A', T, e0, p0, F), donde:
- E es un conjunto finito de estados
- A es el alfabeto de entrada, cuyos caracteres se utilizan para formar la cadena a analizar
- A' es el alfabeto de la pila
- e0 es el estado inicial
- p0 es el símbolo inicial de la pila, el que indica que la pila no tiene símbolos
- F es el conjunto de estados finales
- T es la función T: E x (A ∪ {ε}) x A' -> conjuntos finitos de E x A'*

Ejemplo 2
Analicemos qué significa la notación de la función T, con un caso particular:

T(4,a,Z) = { (4,RPZ), (5,ε) }

Esta notación se lee así: si el AFP se encuentra en el estado 4, en el tope de la pila tiene el símbolo Z y lee el carácter a del flujo de entrada, entonces se queda en el estado 4 y agrega RP a la pila, ó (aquí está el no-determinismo del AFP) se mueve al estado 5 y quita el símbolo Z que está en el tope de la pila.

Una aclaración importante. La forma de procesar la pila es la siguiente: 1) el AFP realiza un *pop* del símbolo que está en el tope de la pila; 2) el AFP realiza un *push* de los símbolos indicados, siendo el primero el que quedará en el tope. En la transición de este ejemplo, el AFP hace un *pop* del símbolo Z e, inmediatamente después, un *push* de Z, luego P y , finalmente, R, que es el nuevo símbolo del tope de la pila. En el otro caso, ε significa que no agrega símbolos a la pila, solo hace el *pop* del símbolo que está en el tope.

* Ejercicio 4 *
Investigue e informe: sea la transición T(4,ε,Z) = { (5,RP) }. Describa qué significa.

Antes de pasar a analizar los AFP deterministas, es importante conocer este concepto:

un AFP puede reconocer un LIC de dos maneras:
1) por estado final, como en los AFs;
2) por pila vacía

* Ejercicio 5 *
Investigue e informe qué significa lo escrito en el último recuadro.

* Ejercicio 6 *
Investigue e informe si existe alguna forma de diagramar un AFP en forma similar al Diagrama de Transiciones de un Autómata Finito.

2.2 AUTÓMATA FINITO CON PILA DETERMINÍSTICO (AFPD)

En esta sección definiremos y trabajaremos con un AFPD. Si incluimos el flujo de entrada en la definición, entonces un AFPD está formado por una colección de estos ocho elementos:

- Un alfabeto de entrada A. Son los símbolos con los que se forman las palabras del LIC a reconocer. Hay un símbolo especial al que llamaremos fdc (fin de cadena), que solo puede aparecer al final de la cadena en estudio para indicar su terminación.
- Un flujo de entrada infinito en una dirección, que contiene la cadena a analizar.
- Un alfabeto A' de símbolos de la pila. Hay un símbolo especial, al que llamaremos, $, que indica "pila lógicamente vacía".
- Una pila (*stack*) infinita en una dirección. Inicialmente la pila está vacía, lo que se indica mediante el símbolo especial $ en el tope de la pila.
- Un conjunto finito y no vacío de estados E.
- Un estado inicial (único).
- Un conjunto de estados finales o de aceptación.
- Una función de transiciones entre los estados que definimos de la siguiente forma:

$$T: E \times (A \cup \{\varepsilon\}) \times A' \rightarrow E \times A'^*$$

Primera forma de movimiento. Si $a \in A$, entonces $T(e, a, R) = (e', \alpha)$ indica: si el AFPD se encuentra en el estado e, lee el símbolo de entrada a y tiene el símbolo R en el tope de la pila, entonces pasa al estado e' y reemplaza, en la pila, el símbolo R por la secuencia de símbolos α; además, adelanta una posición en el flujo de entrada.

Segunda forma de movimiento. $T(e, \varepsilon, R) = (e', \alpha)$ indica: si el AFPD se encuentra en el estado e, y tiene el símbolo R en el tope de la pila, entonces pasa al estado e' y reemplaza en la pila el símbolo R por la secuencia de símbolos α; además, no adelanta ninguna posición en el flujo de entrada.

➔ Para asegurar que el AFP sea determinístico, solo uno de los dos tipos de movimientos se puede dar para cualquier par (e,R), donde e es un estado y R es el símbolo que se encuentra en el tope de la pila.

* Ejercicio 7 *
Describa las diferencias existentes entre las dos definiciones formales planteadas, la primera en la sección 2.1 y la segunda en esta sección. ¿Significa que alguna de las dos está equivocada? Justifique su respuesta.

Ejemplo 3
Sea el lenguaje L1 del Ejemplo 1: $L1 = \{a^n b^n / n \geq 1\}$. Construyamos un AFPD que lo reconozca.
– El alfabeto de entrada A está formado por los caracteres {a, b}.
– El alfabeto de la pila A' tiene los símbolos: $, para indicar que la pila está vacía, y R, que usaremos para indicar que el AFPD ha leído una a de la cadena que está analizando.

Necesitamos obtener un AFPD que cuente la cantidad de aes con las que comienza la cadena que se está analizando, y que luego pueda "descontar" de esa cantidad por cada b que se lea. Si al final la pila queda vacía, el AFPD reconocerá a la cadena como palabra del lenguaje L1.

Teniendo en cuenta que tanto en un AFP como en un AFPD solo se describen las transiciones correctas, representamos los movimientos del AFPD que resuelve este problema de esta manera:

e0,$ => a => e1,R$ (comienza en el estado inicial y la pila está vacía; si lee una a se mueve al estado e1 y agrega una R a la pila.)
e1,R => b => e2,ε (se halla en el estado e1 y en el tope de la pila hay un símbolo R; si lee una b se mueve al estado e2 y quita la R del tope de la pila.)
. . . y así sucesivamente. En la próxima sección veremos la forma habitual de representar los AFPs y los AFPDs: una Tabla de Movimientos.

* Ejercicio 8 *
¿Cómo se denomina, en la jerga de las pilas, la operación representada con ε en el ejemplo anterior?

2.2.1 LA TABLA DE MOVIMIENTOS (TM)

La TM en un AFP o en un AFPD es similar a la Tabla de Transiciones en un Autómata Finito. La diferencia radica en que en la TM no podemos hablar solo de "estado"; debemos hablar del par "estado y símbolo en el tope de la pila".

Ejemplo 4
Construyamos la TM de un AFPD que reconoce al lenguaje $L1 = \{a^n b^n / n \geq 1\}$:

TM	a	b	fdc
e0,$	e1,R$		
e1,R	e1,RR	e2,ε	
e2,R		e2,ε	
e2,$			e3,$

Como se podrá ver, el conjunto de estados de este AFPD es {e0, e1, e2, e3}, donde e0 es el estado inicial y e3 es el único estado final.

* Ejercicio 9 *
Defina formalmente el AFPD del Ejemplo 4.

* Ejercicio 10 *
a) Describa los movimientos que realiza ese AFPD para reconocer la cadena aaabbb.
b) Describa el trabajo del AFPD para no reconocer la cadena aabbb.
c) Describa el trabajo del AFPD para no reconocer la cadena aaabb.
d) Describa el trabajo del AFPD para no reconocer la cadena aabba.
e) Explique porqué no reconoce a la palabra vacía.

* Ejercicio 11 *
a) Teniendo en cuenta los algoritmos del capítulo anterior, construya en ANSI C una función que implemente la solución descripta en el Ejemplo 4.
b) Escribe un programa en ANSI C que testée la función anterior.

La sección 2.1 termina con un recuadro que dice: "Un AFP puede reconocer un LIC de dos maneras: por estado final, como en los AFs, y por pila vacía". Esto también vale para los AFPDs. Es más:

> Siempre que se puede construir un AFPD que reconozca por estado final,
> se podrá construir un AFPD que reconozca por pila vacía, y viceversa.

En el Ejemplo 4 hemos construido la TM de un AFPD que reconoce al lenguaje $\{a^n\ b^n / n \geq 1\}$ por estado final. En el próximo ejemplo construiremos la TM de un AFPD equivalente que reconoce el mismo lenguaje por pila vacía.

Ejemplo 5

La TM del AFPD que reconoce por pila vacía es la siguiente:

TM	a	b	fdc
e0,$	e1,R$		
e1,R	e1,RR	e2,ε	
e2,R		e2,ε	
e2,$			e2,ε

Como se podrá ver, el conjunto de estados de este AFPD es {e0, e1, e2}, donde e0 es el estado inicial y el conjunto de estados finales es vacío.

* Ejercicio 12 *
Defina formalmente el AFPD del Ejemplo 5.

* Ejercicio 13 *
a) Describa los movimientos que realiza ese AFPD para reconocer la cadena aaabbb.
b) Describa el trabajo del AFPD para no reconocer la cadena aabbb.
c) Describa el trabajo del AFPD para no reconocer la cadena aaabb.
d) Describa el trabajo del AFPD para no reconocer la cadena aabba.

* Ejercicio 14 *
a) Teniendo en cuenta los algoritmos del capítulo anterior, construya en ANSI C una función que implemente la solución descripta en el Ejemplo 5.
b) Escribe un programa en ANSI C que testée la función anterior.

* Ejercicio 15 *
Defina formalmente un AFPD que reconozca el lenguaje $L2 = \{a^n\ b^{n+1} / n \geq 1\}$.

* Ejercicio 16 *
Defina formalmente un AFPD que reconozca el lenguaje $L3 = \{a^{n+1}\ b^n / n \geq 1\}$.

* Ejercicio 17 *
Defina formalmente un AFPD que reconozca el lenguaje $L4 = \{a^n\ b^{2n}\ a^t / n \geq 1, t \geq 0\}$.

* Ejercicio 18 *
Sea el lenguaje L5 de todas las palabras sobre el alfabeto {a, b} en las que la cantidad de aes es igual a la cantidad de bes, como: ab, abba, bababaab, etc. Defina formalmente un AFPD que lo reconozca; tenga en cuenta que las palabras pueden comenzar con a o con b.

2.3 AFPDs Y EXPRESIONES ARITMÉTICAS

Las expresiones aritméticas que utilicen paréntesis para determinar un orden de evaluación, constituyen un LIC existente en la inmensa mayoría de los Lenguaje de Programación. Por lo tanto, se podrá construir un AFPD que las reconozca, es decir: que determine si una secuencia de caracteres, que constituye una supuesta expresión aritmética, es sintácticamente correcta o no.

Ejemplo 6
Sea el LIC de todas las expresiones aritméticas cuyo único operando es 4, los operadores son + (suma) y * (producto), y puede haber paréntesis. Algunos casos correctos de estas expresiones son:

4
4 * (4+4)
4 + 4 + 4 * 4 + ((4))

Construiremos un AFPD que reconozca a este LIC. Aquí debemos tener muy en cuenta los paréntesis, porque siempre deben estar balanceados; para ello, utilizaremos la pila. El alfabeto de la pila será: {R, $}. Los estados serán {e0, e1, e2, e3}, siendo e3 el único estado final.

La Tabla de Movimientos que define a este AFPD será:

TM	4	+,*	(	)	fdc
e0,$	e1,$		e0,R$		
e0,R	e1,R		e0,RR		
e1,$	e1,$	e0,$			e3,$
e1,R	e1,R	e0,R		e2,ε	
e2,$		e0,$			e3,$
e2,R		e0,R		e2,ε	
e3,$					

* Ejercicio 19 *
Defina formalmente el AFPD construido en el ejemplo anterior.

* Ejercicio 20 *
Escriba la secuencia de movimientos del AFPD para cada una de las expresiones siguientes e indique si la acepta o no:
a) ((4))
b) 4 + 4
c) 4 + 4 * (4 + (4))
d) 4 + ((4)))

* Ejercicio 21 *
Investigue e informe porqué un Autómata Finito no puede reconocer el lenguaje de las expresiones aritméticas definido en el Ejemplo 6.

3 INTRODUCCIÓN AL PROCESO DE COMPILACIÓN

En este volumen nos ocuparemos del estudio del proceso que lleva a cabo un compilador, uno de los varios tipos de traductores existentes, y su relación con la sintaxis y la semántica de los Lenguajes de Programación. Además, consideraremos que el programa a traducir siempre se encuentra en un flujo de entrada que está implementado a través de un archivo de texto, como ocurre en Pascal con los archivos con extensión `.pas` o en ANSI C con los archivos con extensión `.c`; esto es:

El programa que se debe compilar es una secuencia de caracteres
que termina con un centinela.

* Ejercicio 1 *
Investigue e informe qué es un traductor en esta disciplina.

* Ejercicio 2 *
Describa cuál sería el centinela en el caso planteado en la frase anterior *

3.1 CONCEPTOS BÁSICOS

Un compilador es un complejo programa que lee un programa escrito en un lenguaje fuente, habitualmente un lenguaje de alto nivel – como Pascal, C, C++, Java y otros miles –, y lo traduce a un programa equivalente en un lenguaje objeto, normalmente más cercano al lenguaje de máquina. Al programa original se lo llama programa fuente y al programa obtenido se lo denomina programa objeto.

* Ejercicio 3 *
Informe qué significa que dos programas sean equivalentes en el contexto citado.

El proceso de compilación está formado por dos partes:

1) el ANÁLISIS, que, a partir del *programa fuente*, crea una representación intermedia del mismo; y

2) la SÍNTESIS, que, a partir de esta representación intermedia, construye el *programa objeto*.

3.1.1 EL ANÁLISIS DEL PROGRAMA FUENTE

En la compilación, el análisis está formado por tres fases:

a) el Análisis Léxico,

b) el Análisis Sintáctico, y

c) el Análisis Semántico.

a) ANÁLISIS LÉXICO

El Análisis Léxico detecta los diferentes elementos básicos que constituyen un programa fuente, como: identificadores, palabras reservadas, constantes, operadores y caracteres de puntuación. Por lo tanto, el Análisis Léxico solo se ocupa de los Lenguajes Regulares que forman parte del Lenguaje de Programación.

Por ello, esta fase de análisis tiene como función detectar palabras de estos Lenguajes Regulares; en la jerga de la compilación, estas palabras se denominan LEXEMAS y los LRs a los que pertenecen estos lexemas se denominan CATEGORÍAS LÉXICAS o TOKENS (palabra inglesa muy utilizada en este campo).

En otras palabras: la *entrada* para el Análisis Léxico son caracteres y la *salida* son tokens.

Nota 1
Los espacios que separan a los lexemas son ignorados durante el Análisis Léxico.

* Ejercicio 4 *
Sea, en ANSI C, el siguiente fragmento de programa:

```
...
int WHILE;
while   (WHILE > (32))     -2.46;
...
```

Diseñe una tabla con dos columnas: lexema y token. Complete la tabla con todos los lexemas hallados en el Análisis Léxico y las categorías a las que pertenecen.

* Ejercicio 5 *
Sea, en ANSI C, la siguiente función:

```
void XX (void) {
   int a;
   double a;
   if (a > a)  return 12;
}
```

Diseñe una tabla con dos columnas: lexema y token. Complete la tabla con todos los lexemas hallados en el Análisis Léxico de esta función.

* Ejercicio 6 *
Describa todos los errores que encuentra en la función del Ejercicio 5. Para ello, construya una tabla que indique: Nº de Línea – Descripción del Error – Tipo de Error (Léxico, Sintáctico, Otro)

b) ANÁLISIS SINTÁCTICO

El Análisis Sintáctico trabaja con los tokens detectados durante el Análisis Léxico, es decir: la *entrada* para el Análisis Sintáctico son esos tokens.

Esta etapa de análisis sí conoce la sintaxis del Lenguaje de Programación y, en consecuencia, tendrá la capacidad de determinar si las construcciones que componen el programa son sintácticamente correctas. Sin embargo, no podrá determinar si el programa, en su totalidad, es sintácticamente correcto.

Ejemplo 1
Durante el Análisis Sintáctico, la GIC que describe la sintaxis del lenguaje – por su propia independencia del contexto – no permite detectar si una variable fue declarada antes de ser utilizada,

si la misma variable fue declarada varias veces y con diferentes tipos, como en el Ejercicio 5, y muchas otras situaciones erróneas.

> *Nota 2*
> Debemos diferenciar, claramente, los términos "Análisis Léxico" vs "Analizador Léxico" y "Análisis Sintáctico" vs "Analizador Sintáctico". Por ello, y para evitar confusiones, a partir de ahora siempre llamaremos Scanner (palabra inglesa) al "Analizador Léxico" y Parser (palabra inglesa) al "Analizador Sintáctico".

Dada una definición sintáctica formal, como la que brinda una GIC, el Parser recibe tokens del Scanner y los agrupa en unidades, tal como está especificado por las producciones de la GIC utilizada. Mientras se realiza el Análisis Sintáctico, el Parser verifica la corrección de la sintaxis y, si encuentra un error sintáctico, el Parser emite el diagnóstico correspondiente.

En la medida que las estructuras semánticas son reconocidas, el Parser llama a las correspondientes rutinas semánticas que realizarán el Análisis Semántico.

* Ejercicio 7 *
Investigue e informe: ¿qué son los terminales de la GIC utilizada para realizar el Análisis Sintáctico, caracteres, lexemas o tokens?

c) ANÁLISIS SEMÁNTICO

El Análisis Semántico realiza diferentes tareas, completando lo que hizo el Análisis Sintáctico. Una de estas importantes tareas es la "verificación de tipos", para que cada operador trabaje sobre operandos permitidos según la especificación del Lenguaje de Programación.

* Ejercicio 8 *
Volviendo a la función del Ejercicio 5:

```
void XX (void) {
   int a;
   double a;
   if (a > a)  return 12;
}
```

Informe cuáles son errores que serán detectados durante el Análisis Semántico.

Las rutinas semánticas llevan a cabo dos funciones:

1) Chequean la *semántica estática* de cada construcción; es decir, verifican que la construcción analizada sea legal y que tenga un significado. Verifican que las variables involucradas estén definidas, que los tipos sean correctos, etc.

2) Si la construcción es semánticamente correcta, las rutinas semánticas también hacen la *traducción*; es decir, generan el código para una Máquina Virtual que, a través de sus instrucciones, implementa correctamente la construcción analizada.

Ejemplo 2
Sea una producción `E->E+T`. Cuando esta producción es reconocida por el Parser, éste llama a las rutinas semánticas asociadas para chequear la semántica estática (compatibilidad de tipos) y luego

realizar las operaciones de traducción, generando las instrucciones para la Máquina Virtual que permitan realizar la suma.

Una definición completa de un LP debe incluir las especificaciones de su Sintaxis y de su Semántica. La Sintaxis se divide, normalmente, en componentes Independientes del Contexto y componentes Sensibles al Contexto.

Las GICs sirven para especificar la Sintaxis Independiente del Contexto. Sin embargo, no todo el programa puede ser descripto por una GIC: por caso, la compatibilidad de tipos y las reglas de alcance de las variables son Sensibles al Contexto.

Ejemplo 3
En ANSI C, la sentencia `a = b + c;` es ilegal si cualquiera de las variables no estuviera declarada; esta es una restricción Sensible al Contexto.

Resumiendo: Debido a estas limitaciones de las GICs, las restricciones Sensibles al Contexto son manejadas como situaciones semánticas. En consecuencia, el componente semántico de un LP se divide, habitualmente, en dos clases: *Semántica Estática*, que es la que nos interesa en este momento, y *Semántica en Tiempo de Ejecución*.

La Semántica Estática de un LP define las restricciones Sensibles al Contexto que deben cumplirse para que el programa sea considerado correcto. Algunas reglas típicas de Semántica Estática son:

1) Todos los identificadores deben estar declarados;

2) Los operadores y los operandos deben tener tipos compatibles;

3) Las rutinas (procedimientos y funciones) deben ser llamados con el número apropiado de argumentos.

➔ Ninguna de estas restricciones puede ser expresada mediante una GIC.

* Ejercicio 9 *
a) Investigue e informe qué es el ALCANCE de una variable.
b) Se han mencionado tres reglas típicas de Semántica Estática. Escriba un ejemplo de cada una de ellas.

3.2 UN COMPILADOR SIMPLE

Esta sección describirá un proceso formado por cuatro etapas:

1º la definición de un Lenguaje de Programación muy simple;
2º la escritura de un programa fuente en este lenguaje;
3º el diseño de un compilador para realizar la traducción;
4º la obtención de un programa objeto equivalente.

> *Nota 3*
> Esta sección está basada en el capítulo 2 del libro de Charles Fischer, "*Crafting a Compiler with C*" (1991). Sobre el desarrollo de Fischer, se han realizado varias mejoras; de esta forma, se aplicará lo que hemos visto en el Volumen 1 y en el capítulo 1 de este volumen.

3.2.1 DESCRIPCIÓN INFORMAL DEL LENGUAJE DE PROGRAMACIÓN *MICRO*

Fischer presenta un LP que denomina Micro. Es un lenguaje muy simple que está diseñado, específicamente, para poseer un LP concreto sobre el que se pueda analizar la construcción de un compilador básico.

Informalmente, Fischer lo define de esta manera:

- El único tipo de dato es entero.
- Todos los identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres.
- Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos.
- Las constantes son secuencias de dígitos (números enteros).
- Hay dos tipos de sentencias:

  Asignación

  ID := Expresión;

  Expresión es infija y se construye con identificadores, constantes y los operadores **+** y **–**; los paréntesis están permitidos.

  Entrada/Salida

  leer (lista de IDs);

  escribir (lista de Expresiones);
- Cada sentencia termina con un "punto y coma" (**;**). El cuerpo de un programa está delimitado por inicio y fin.
- **inicio, fin, leer** y **escribir** son palabras reservadas y deben escribirse en minúscula.

* Ejercicio 10 *
a) Investigue e informe qué significa "Expresión es infija".
b) Informe qué significa ID.
c) Informe qué son fin, Fin y finn.
d) Informe qué significa: "Todos los identificadores son declarados implícitamente".

Ejemplo 4
El siguiente es un programa fuente en Micro:

```
inicio
   leer (a,b);
   cc := a + (b-2);
   escribir (cc, a+4);
fin
```

* Ejercicio 11 *
Diseñe una tabla con dos columnas: lexema y token. Suponga que hay seis tipos de tokens: palabraReservada, identificador, constante, operador, asignación y carácterPuntuación. Realice el Análisis Léxico del programa del Ejemplo 4 y complete la tabla.

3.2.2 SINTAXIS DE MICRO – DEFINICIÓN PRECISA CON UNA GIC (BNF)

En esta sección, agregamos una definición más precisa de Micro. Describimos las producciones de una Gramática Léxica que define los posibles lexemas que se pueden encontrar en un programa Micro, y las producciones de una Gramática Sintáctica que permite precisar las construcciones válidas en Micro.

> Si bien no se informa cuáles son las reglas de escritura de las producciones, con lo estudiado en el Volumen 1 el lector / la lectora no debe tener inconvenientes para comprenderlas. Si los tiene, debe repasar el Volumen 1.

Gramática Léxica

```
<token> -> uno de <identificador> <constante> <palabraReservada>
                  <operadorAditivo> <asignación> <carácterPuntuación>
<identificador> -> <letra> {<letra o dígito>}
<constante> -> <dígito> {dígito>}
<letra o dígito> -> uno de <letra> <dígito>
<letra> -> una de a-z A-Z
<dígito> -> uno de 0-9
<palabraReservada> -> una de inicio fin leer escribir
<operadorAditivo> -> uno de + -
<asignación> -> :=
<carácterPuntuación> -> uno de ( ) , ;
```

* Ejercicio 12 *

a) Informe el conjunto de noterminales, el conjunto de terminales y el conjunto de metasímbolos de la Gramática Léxica cuyas producciones están dadas.

b) Investigue e informe qué ocurre si el programa fuente contiene un carácter que no es un terminal ni puede formar parte de un terminal de la Gramática Léxica descripta.

Gramática Sintáctica

```
<programa> -> inicio <listaSentencias> fin
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> <identificador> := <expresión> ; |
               leer ( <listaIdentificadores> ) ; |
               escribir ( <listaExpresiones> ) ;
<listaIdentificadores> -> <identificador> {, <identificador>}
<listaExpresiones> -> <expresión> {, <expresión>}
<expresión> -> <primaria> {<operadorAditivo> <primaria>}
<primaria> -> <identificador> | <constante> |
              ( <expresión> )
```

* Ejercicio 13 *

a) Informe el conjunto de noterminales, el conjunto de terminales y el conjunto de metasímbolos de la Gramática Sintáctica cuyas producciones están dadas.

b) En base a las dos gramáticas descriptas, escriba un programa en Micro que sea mínimo y correcto.

c) En base a las dos gramáticas descriptas, escriba un programa en Micro que sea correcto y que utilice todos los elementos definidos. Debe ser diferente al del Ejemplo 4.

d) Informe si **:=** es un operador. Justifique su respuesta.

Para mayor claridad de lectura, en la Gramática Sintáctica encontramos algunos tokens que permanecen con sus caracteres originales – las palabras reservadas, la asignación, los paréntesis y otros – tal como aparecen en un programa fuente en Micro. Sin embargo, al construir el compilador, cada uno de estos tokens tendrá su propio nombre, como se ve en la siguiente tabla:

En el Programa Fuente	Nombre del Token
Inicio	INICIO
Fin	FIN
Leer	LEER
Escribir	ESCRIBIR
:=	ASIGNACIÓN
(	PARENIZQUIERDO
)	PARENDERECHO
,	COMA
;	PUNTOYCOMA
+	SUMA
-	RESTA

3.2.3 LA ESTRUCTURA DE UN COMPILADOR Y EL LENGUAJE MICRO

– El ANÁLISIS LÉXICO es realizado por un módulo llamado Scanner. Este analizador lee, uno a uno, los caracteres que forman un programa fuente, y produce una secuencia de representaciones de tokens. Es muy importante tener en cuenta que el Scanner es una rutina que produce y retorna la representación del correspondiente token, uno por vez, en la medida que es invocada por el Parser.

Existen dos formas principales de implementar un Scanner:

a) A través de la utilización de un programa auxiliar tipo *lex*, en el que los datos son tokens representados mediante Expresiones Regulares, como ya hemos visto;
b) Mediante la construcción de una rutina basada en el diseño de un apropiado AFD; este método es el que utilizaremos en Micro.

– El ANÁLISIS SINTÁCTICO es realizado por un módulo llamado Parser. Este analizador procesa los tokens que le entrega el Scanner hasta que reconoce una construcción sintáctica que requiere un procesamiento semántico. Entonces, invoca directamente a la rutina semántica que corresponde. Algunas de estas rutinas semánticas utilizan, en sus procesamientos, la información de la representación de un token, como veremos más adelante. Nótese que no existe un módulo independiente llamado Analizador Semántico, sino que el ANÁLISIS SEMÁNTICO se va realizando, en la medida que el Parser lo requiere, a través de las rutinas semánticas.

Las rutinas semánticas realizan el Análisis Semántico y también producen una salida en un lenguaje de bajo nivel para una Máquina Virtual, tal como veremos en la sección 3.2.6; esto último forma parte de la etapa de Síntesis del compilador.

Existen dos formas fundamentales de Análisis Sintáctico: el Análisis Sintáctico Descendente (conocido como *top-down*), que permite ser construido por un programador, y el Análisis Sintáctico Ascendente (conocido como *bottom-up*), que requiere el auxilio de un programa especializado tipo *yacc*. Para Micro, construiremos un Parser basado en el Análisis Sintáctico Descendente y en el próximo capítulo explicaremos más sobre ambos métodos y sus diferencias.

– En cuanto al ANÁLISIS SEMÁNTICO, del cual ya hemos hablado anteriormente, es la tercera fase de análisis que existe en un compilador. Por un lado, el Análisis Semántico está inmerso dentro del Análisis Sintáctico, y, por otro lado, es el comienzo de la etapa de Síntesis del compilador. Lo veremos, con más precisión, al desarrollar el compilador Micro.

– La TABLA DE SÍMBOLOS (TS) es una estructura de datos compleja que es utilizada para el almacenamiento de todos los identificadores del programa a compilar. Dependiendo del diseño del compilador, a veces también es utilizada para que contenga las palabras reservadas del LP, los literales-constante y las constantes numéricas que existen en el programa, pero éstas últimas en forma de secuencia de caracteres.

Cada elemento de la TS está formada por una cadena y sus atributos. En el caso de Micro, la TS contendrá las palabras reservadas y los identificadores; cada entrada tendrá, como único atributo, un código que indique si la cadena representa una "palabra reservada" o "un identificador".

En general, la TS es muy utilizada durante toda la compilación y, específicamente, en la etapa de Análisis por las rutinas semánticas.

* Ejercicio 14 *
Investigue e informe:
a) qué es una máquina virtual;
b) qué significa "en tiempo de compilación";
c) qué significa "en tiempo de ejecución".
d) qué programa se compila;
e) qué programa se ejecuta;
f) ¿son equivalentes el programa que se compila y el programa que se ejecuta?

3.2.4 UN ANALIZADOR LÉXICO PARA MICRO

Los lexemas que constituyen un programa fuente en este LP Micro son: identificadores, constantes, las cuatro palabras reservadas ya mencionadas, paréntesis izquierdo y derecho, el "punto y coma" (para terminar cada sentencia), la "coma" (para formar una lista), el símbolo de asignación (**:=**), y los operadores de suma (**+**) y de resta (**–**).

* Ejercicio 15 *
Pregunta: ¿Las gramáticas dadas generan la información que figura en el párrafo anterior? Justifique su respuesta.

➔ Atención con la siguiente propiedad: cada lexema está formado por la mayor secuencia posible de caracteres para el token correspondiente, y esto requiere, a veces, que se lea el primer carácter que no pertenece a ese lexema, es decir: el carácter que actúa como centinela.

> *Ejemplo 5*
> En la expresión abc+4, el identificador abc es detectado completamente por el Scanner cuando lee el carácter +, que es el primer carácter no válido (un centinela) para el token identificador.
>
> * Ejercicio 16 *
> Escriba 4 caracteres que puedan actuar como centinelas para un identificador en Micro.

Este carácter extra que se lee debe ser retornado al flujo de entrada porque será el primer carácter (o único carácter) del siguiente lexema. Para ello, en ANSI C se utiliza la función ungetc.

* Ejercicio 17 *
Investigue e informe sobre la función ungetc: cómo es su prototipo, qué significa cada parámetro y el valor que retorna, y qué tarea realiza con exactitud.

➔ Si, en cualquier momento de la compilación, el primer carácter del flujo de entrada que lee el Scanner para detectar el próximo lexema no es un carácter válido para ningún lexema en el LP Micro, entonces se detectó un error léxico. En este caso, se despliega un mensaje de error y se repara el error para continuar con el proceso. La reparación más sencilla consiste en ignorar este carácter espúreo y reiniciar el Análisis Léxico a partir del carácter siguiente.

Ejemplo 6

```
        _Abc := 3;
```

El supuesto identificador comienza con un carácter espúreo para Micro, el "guión bajo". Esto produce un error léxico.

* Ejercicio 18 *
Informe, formalmente, porqué el "guión bajo" es un carácter espúreo en Micro.

* Ejercicio 19 *
Si el Scanner de Micro debe procesar el texto `abc*ab`. ¿Qué produce? Descríbalo con precisión.

Otra tarea del Scanner es ignorar los espacios – ya sea espacios en blanco, tabulados o marcas de fin de línea – que actúen como separadores de lexemas dentro del texto del programa fuente.

* Ejercicio 20 *
Investigue e informe:
a) qué sucede si el programa fuente comienza con muchos espacios;
b) qué sucede si, en ANSI C, un literal-cadena tiene espacios, como `"H  O \n  L\tA  ";`
c) qué sucede si el programa ANSI C tiene un carácter constante que es un espacio, como `' '` o `'\n'`.

➔ El Scanner para Micro estará basado en la construcción de un AFD que reconoce los LRs que forman los tokens de Micro.

En cuanto al reconocimiento de las palabras reservadas, el primer problema es que son identificadores. Existen varias soluciones; veamos dos de ellas.

Una solución es que cada palabra reservada sea reconocida, por el AFD diseñado, mediante un estado final propio para cada una de ellas, independiente del estado final para el reconocimiento de los identificadores generales.

Otra solución es que las palabras reservadas sean colocadas, previamente, en las primeras entradas de la Tabla de Símbolos y con el atributo "reservada"; es decir, la TS ya contiene las palabras reservadas antes de comenzar el Análisis Léxico.

En la segunda solución, que es la que usaremos, el AFD estará diseñado con un único estado final para reconocer a todos los identificadores, incluyendo las palabras reservadas. Entonces, cuando un identificador es reconocido por el Scanner, lo busca en la TS. Si lo encuentra y tiene el atributo "reservada", sabe que es una palabra reservada y cuál es; caso contrario, y si el identificador detectado todavía no se encuentra almacenado en la TS, lo agrega con el atributo "id".

* Ejercicio 21 *
Sea el siguiente programa Micro:

```
inicio a := 23; escribir (a, b+2); fin
```

a) Describa el contenido que queda en la Tabla de Símbolos una vez que el Análisis Léxico haya finalizado.
b) ¿Cómo hace el Scanner de Micro para reconocer que el identificador inicio es una Palabra Reservada?

Por último, el Parser debe saber cuándo el Scanner le pasó el último token detectado en el flujo de entrada. Para ello, se crea un token que llamaremos fdt (fin del texto). En consecuencia, cuando el Scanner llegue al final del flujo que contiene al programa fuente, retornará el token fdt.

El Scanner será implementado como una función sin parámetros que retorna valores de tipo TOKEN. Por ello, definimos los siguientes tipos de tokens en ANSI C:

```
typedef enum {
   INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO,
   PARENDERECHO, PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA, FDT
} TOKEN;
```

Suponemos que el prototipo de la función que implementa al Scanner es:

```
TOKEN Scanner (void);
```

Nota 4
El compilador Micro es muy simple. No obstante, la función `Scanner` no puede retornar solo un valor por enumeración y pretender que el compilador pueda seguir desarrollando el trabajo necesario con su Parser y con las rutinas de análisis semántico. Si la función `Scanner` retorna `ID`, los otros analizadores necesitan saber a qué identificador se refiere. Lo mismo ocurre cuando `Scanner` retorna `CONSTANTE`: los otros analizadores necesitan conocer su valor. Por ello, supondremos que existen variables globales que contendrán esta información y que podrán ser accedidas por los otros analizadores.

* Ejercicio 22 *
Investigue e informe:

a) Qué es typedef;
b) Qué es enum;
c) Qué diferencia hay entre el prototipo de una función y la definición de la misma función.

3.2.4.1 EL AFD PARA IMPLEMENTAR EL SCANNER

Dado que el Scanner debe reconocer palabras de diferentes Lenguajes Regulares, es lógico suponer que el diseño de un AFD adecuado sea una buena solución.

Tal como expresamos antes, construiremos un AFD con un único estado final para todos los identificadores, incluyendo las palabras reservadas; de esta manera, el AFD será más compacto. Además, este AFD deberá realizar ciertas acciones en varios estados, como veremos a continuación.

El AFD será representado mediante su Tabla de Transiciones. En la TT hay una fila por cada estado del AFD y una columna por cada carácter o conjunto de caracteres que tienen el mismo estado de llegada para cualquier transición. ¿Qué ocurre en el caso de Micro?

1º Todas las letras son equivalentes en cuanto a su funcionalidad, por lo que llamaremos L a la columna correspondiente. Estos caracteres se detectan en ANSI C por medio de la función isalpha;

2º Todos los dígitos, que se detectan en ANSI C con isdigit, cumplen la misma función. Llamaremos D a la columna correspondiente;

3º Los caracteres +, –, (,), "punto y coma", :, = y "coma" tienen, cada uno, una función particular; por lo tanto, habrá una columna para cada uno de ellos;

4º El centinela de "fin de texto", al que llamamos fdt, tiene una función especial y muy importante, ya que le indica al Parser que se terminaron los tokens. Por lo tanto, deberá tener su propia columna en la TT del AFD;

5º El AFD ignorará los espacios. En consecuencia habrá una columna, a la que llamaremos sp, para los tres caracteres especiales que ya analizamos anteriormente. Estos caracteres son reconocidos en ANSI C con la función isspace.

6º Finalmente, habrá una columna llamada otro para cualquier otro carácter no mencionado en los puntos anteriores. Estos caracteres no pueden intervenir en ningún lexema válido en Micro, por lo que su detección producirá un error léxico.

* Ejercicio 23 *
Investigue e informe sobre las funciones: isalpha, isdigit e isspace. Para cada una de ellas describa su prototipo, y la función que cumple cada parámetro y el valor que retorna. Escriba ejemplos de sus usos.

Con esta información previa, la Tabla de Transiciones de un AFD que resuelve este problema es la siguiente:

TT	L	D	+	-	(	)	,	;	:	=	fdt	sp	otro
0-	1	3	5	6	7	8	9	10	11	14	13	0	
1	1	1	2	2	2	2	2	2	2	2	2		2
2+	99	99	99	99	99	99	99	99	99	99	99		99
3	4	3	4	4	4	4	4	4	4	4	4		4
4+													
5+													
6+													
7+													
8+													
9+													
10+													
11										12			
12+													
13+													
14													

* Ejercicio 24 *
a) Obviamente, esta TT no está completa. Complétela.
b) ¿Qué puede representar el valor 99 que aparece en la TT?
c) Declare e inicialice, en ANSI C, la matriz tt que implementa la Tabla de Transiciones construida y completada en el punto a).

Como dijimos anteriormente, ciertos estados provocarán que el Scanner lleve a cabo determinadas acciones. Analicemos esta situación.

1º En los estados 1 y 3, el Scanner debe almacenar el carácter leído en el buffer, un vector de caracteres externo a la función en el que se guardarán los caracteres que forman los presuntos identificadores (incluidas las palabras reservadas) y los dígitos que forman las presuntas constantes.

2º En el estado 2, el Scanner debe realizar varias acciones: a) ungetc; b) determinar si es una palabra reservada o no; c) si es un identificador, debe verificar que su longitud no supera los 32 caracteres; d) si es un identificador correcto, debe almacenarlo en la Tabla de Símbolos.

3º En el estado 4, el Scanner debe realizar dos acciones: a) ungetc; b) almacenar la secuencia de dígitos en la Tabla de Símbolos.

4º En el estado 13, el Scanner debe retornar el código que indique la detección del fdt para que el Parser conozca que se han terminado los lexemas del programa fuente.

5º En el estado 14, el AFD ha detectado un Error Léxico y, entonces, debe llevar a cabo las acciones que correspondan.

6º En el estado 7, debe retornar el código que corresponde al carácter **(**.

. . .

* Ejercicio 25 *
a) Verifique que las acciones descriptas son correctas.
b) Complete las acciones que deben añadirse a este AFD.

* Ejercicio 26 *
a) Si el AFD reconociera a cada una de las cuatro palabras reservadas en forma independiente, ¿cuántos estados y cuántas columnas deberíamos agregarle a la TT diseñada?
b) Informe qué modificaciones habría en las acciones a llevar a cabo en ciertos estados del AFD.

3.2.4.2 CONCLUSIÓN

El Parser invoca al Scanner cada vez que necesita un token. El Scanner analiza el flujo de entrada a partir de la posición que corresponde, detecta el próximo lexema en el Programa Fuente y retorna el token correspondiente, o tal vez encuentra un error léxico y le informa al Parser de esta situación anómala.

En la construcción del Scanner juega un papel muy importante el AFD diseñado para reconocer los lexemas. Además de este AFD, existen almacenamientos y funciones auxiliares que completan la implementación del Scanner en ANSI C; estos son:

– La función ungetc de ANSI C.

– El vector de caracteres externo que llamamos buffer, utilizado para almacenar los caracteres de los identificadores y los dígitos de las constantes en la medida que son reconocidos por el AFD.

– La función con prototipo void AgregarCaracter(int); que añade un carácter al buffer.

– La función con prototipo TOKEN EsReservada(void); que, dado el identificador reconocido por el AFD y almacenado en el buffer, retorna el token que le corresponde (`ID` o el código de alguna de las cuatro palabras reservadas).

– La función con prototipo void LimpiarBuffer(void); que "vacía" el buffer para el próximo uso.

– La función feof de ANSI C, para detectar el centinela del texto, el fdt, y así saber que todos los caracteres del Programa Fuente fueron procesados.

– La función fgetc de ANSI C, para leer cada carácter del flujo de entrada.

– Las funciones ANSI C isspace, isalpha, isalnum e isdigit.

* Ejercicio 27 *
a) Investigue e informe sobre las funciones ANSI C: feof, fgetc e isalnum.
b) Investigue e informe porqué la función AgregarCaracter tiene un parámetro int si trabaja con caracteres.

3.2.5 UN PARSER PARA MICRO

Recordemos que el Parser trabaja con tokens y es guiado por la GIC que describe la sintaxis del LP, Micro en este caso. Estos tokens le son provistos uno a uno por el Scanner, cada vez que el Parser lo invoca.

Antes de desarrollar el Parser, recordemos la Gramática Sintáctica que hemos definido en la sección 3.2.2:

```
<programa> -> inicio <listaSentencias> fin
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> <identficador> := <expresión>; |
               leer ( <listaIdentificadores> ); |
               escribir ( <listaExpresiones> );
<listaIdentificadores> -> <identificador> {, <identificador>}
<listaExpresiones> -> <expresión> {, <expresión>}
<expresión> -> <primaria> {<operadorAditivo> <primaria>}
<primaria> -> <identificador> | <constante> |
              ( <expresión> )
```

Extenderemos y modificaremos esta GIC para que refleje exactamente cómo será utilizada por el Parser. En primer lugar, agregaremos una producción global que defina la totalidad del programa que es analizado, recordando que el programa fuente es un texto que termina con un centinela. En segundo lugar, utilizaremos los nombres que les dimos a los tokens en la declaración que presentamos en la sección 3.2.2 y que repetimos a continuación:

```
typedef enum {
   INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO,
   PARENDERECHO, PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA, FDT
} TOKEN;
```

En consecuencia la GIC actualizada que utilizará el Parser será:

```
<objetivo> -> <programa> FDT    (Esta es la producción global que se agrega)
<programa> -> INICIO <listaSentencias> FIN
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> ID ASIGNACIÓN <expresión> PUNTOYCOMA |
     LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA |
     ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA
<listaIdentificadores> -> ID {COMA ID}
<listaExpresiones> -> <expresión> {COMA <expresión>}
<expresión> -> <primaria> {<operadorAditivo> <primaria>}
<primaria> -> ID | CONSTANTE |
              PARENIZQUIERDO <expresión> PARENDERECHO
<operadorAditivo> -> uno de SUMA RESTA
```

> En general, no es común que se realice esta transformación. En esta oportunidad la hacemos para afianzar este concepto: los terminales de la GIC utilizada en el Análisis Sintáctico son los tokens que retorna el Scanner y no los caracteres o las secuencias de caracteres – según corresponda – que se encuentran en el Programa Fuente.

Utilizaremos una técnica de Análisis Sintáctico muy conocida, llamada ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO (ASDR). Su nombre se debe a que utiliza rutinas, que pueden ser recursivas, cuya ejecución va "construyendo" un árbol de análisis sintáctico (AAS) para la secuencia de entrada – formada por tokens – que debe reconocer. Estos árboles no son, generalmente, estructuras de datos físicas sino que son implícitos.

> Un Árbol de Análisis Sintáctico parte del axioma de una GIC y representa la derivación de una construcción (declaración, sentencia, expresión, bloque y hasta el programa completo). Un AAS tiene las siguientes propiedades:
>
> 1) La raíz del AAS está etiquetada con el axioma de la GIC.
> 2) Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.
> 3) Cada nodo interior está etiquetado con un noterminal.
> 4) Si `A` es un noterminal y `X1`, `X2`,...,`Xm` son sus hijos de izquierda a derecha, entonces existe la producción `A->X1...Xm`, donde cada `Xi` puede ser un noterminal o un terminal.

El ASDR es una de las técnicas más simples de las utilizadas en la construcción de un compilador, pero no se puede utilizar con cualquier GIC, como veremos en el próximo capítulo.

La idea básica del ASDR es que cada noterminal de la Gramática Sintáctica tiene asociado una rutina de Análisis Sintáctico que puede reconocer cualquier secuencia de tokens generada por ese noterminal; esta rutina se implementa como un procedimiento (lo llamaremos PAS, Procedimiento de Análisis Sintáctico). En ANSI C, los PAS se construyen como funciones void.

Como dijimos, cada PAS implementa un noterminal de la Gramática Sintáctica. La estructura de cada PAS sigue fielmente el desarrollo del lado derecho de la producción que implementa, es decir: dentro de un PAS, tanto los noterminales como los terminales del lado derecho de la producción deben ser procesados y en el orden en que aparecen. Esto se realiza de la siguiente manera:

1) Si se debe procesar un noterminal <A>, invocamos al PAS correspondiente, que, por convención, lo llamaremos con el mismo nombre, A. Esta llamada puede ser recursiva y de ahí el nombre de este tipo de Análisis Sintáctico.
2) Para procesar un terminal t, invocaremos al procedimiento llamado Match con argumento t.

Este procedimiento Match con argumento t, es decir, Match(t), invoca al Scanner para obtener el próximo token del flujo de tokens de entrada. Si el token obtenido por el Scanner es t, es decir, coincide con el argumento con el cual se invoca a Match, entonces todo es correcto porque hubo concordancia (eso significa *match*); si es así, el token es guardado en una variable global llamada tokenActual.

En cambio, si el token obtenido por el Scanner no coincide con el argumento t, entonces se ha producido un Error Sintáctico; se debe emitir un mensaje de error y tener en cuenta esta situación porque el proceso de compilación ya no puede ser correcto. Además, se debe realizar algún tipo de "reparación" del error para poder continuar con el Análisis Sintáctico.

> * Ejercicio 28 *
> a) Suponiendo que ya existe una función auxiliar llamada RepararErrorSintactico, implemente el procedimiento Match en ANSI C.
> b) Sea el programa Micro inicio a := 23; escribir (a, b+2); fin. Construya el árbol de análisis sintáctico para este programa.

Para ver cómo trabajan estos PAS (procedimientos de Análisis Sintáctico), desarrollaremos algunos de ellos. En todos los casos, veremos cómo estos procedimientos reflejan las definiciones dadas por las producciones de la Gramática Sintáctica actualizada y descripta en la página anterior.

El Parser comienza su actividad cuando se invoca al PAS que corresponde al noterminal <objetivo>, cuya producción hemos agregado en la GIC actualizada:

```
void Objetivo (void) {
/* <objetivo> -> <programa> FDT */
   Programa();
   Match(FDT);
}
```

Esto significa: para determinar que el Análisis Sintáctico de un programa en Micro ha finalizado, debemos hallar la correspondencia con la secuencia de tokens generada por <programa> y seguida de FDT, el token-centinela que agregamos para indicar la terminación del programa fuente.

Siguiendo el orden de las producciones de la GIC, para el noterminal <programa> tenemos un PAS que se puede construir fácilmente a partir de su producción:

```
<programa> -> INICIO <listaSentencias> FIN
```

* Ejercicio 29 *
Construya el PAS Programa en ANSI C.

Para el noterminal <listaSentencias>, el PAS es un poco más complicado porque hay una secuencia opcional de sentencias; es decir, <listaSentencias> es una o más sentencias, como se ve en la definición de este noterminal: `<listaSentencias> -> <sentencia> {<sentencia>}`

Además, existen tres tipos de sentencias, como se aprecia en la definición de <sentencia>:

```
<sentencia> -> ID ASIGNACIÓN <expresión> PUNTOYCOMA |
     LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA |
     ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA
```

Estos tres tipos de sentencias tienen una importante particularidad: cada uno comienza con un token diferente (`ID`, `LEER` y `ESCRIBIR`). Por lo tanto, conociendo el primer token, ya sabemos cuál es el tipo de sentencia que corresponde. Este enfoque no funciona para todas las GICs pero sí lo hace para la gramática LL(1) como la que estamos utilizando y que analizaremos más extensamente en el próximo capítulo.

Sea ProximoToken una función que retorna el próximo token a ser correspondido. Como se ve en las producciones de <sentencia>, el próximo token debería ser `ID`, `LEER` o `ESCRIBIR`, cada uno de los cuales es el primer token en una de las tres posibles sentencias, como se indicó anteriormente. Entonces, si el token retornado por ProximoToken es uno de los tres mencionados, el Parser tratará de reconocer una sentencia de la secuencia opcional definida en el lado derecho de <listaSentencias>. Caso contrario, concluiremos que la lista completa de sentencias fue procesada.

Esto se ve más claramente en el desarrollo del siguiente PAS para el noterminal <listaSentencias>:

```
void ListaSentencias (void) {
/* <listaSentencias> -> <sentencia> {<sentencia>} */
   Sentencia(); /* la primera de la lista de sentencias */
   while (1) { /* un ciclo indefinido */
      switch (ProximoToken()) {
         case ID: case LEER: case ESCRIBIR: /* detectó token correcto */
            Sentencia(); /* procesa la secuencia opcional */
            break;
         default:
            return;
      } /* fin switch */
   }
}
```

* Ejercicio 30 *
Describa, en castellano, la tarea que realiza el PAS ListaSentencias.

Al definir el PAS correspondiente al noterminal sentencia nos encontramos con el problema que hay varias producciones que tienen al noterminal sentencia en su lado izquierdo. En consecuencia, debemos decidir cuál de las producciones deberá ser procesada. En el caso de Micro, esta situación se simplifica porque cada producción para el noterminal sentencia comienza con un terminal diferente y, por lo tanto, con solo conocer cuál es el próximo terminal ya sabemos que producción aplicar. Esta es una propiedad muy ventajosa de las gramáticas LL(1).

El PAS para el noterminal sentencia puede ser construido de esta manera:

```
void Sentencia(void) {
   TOKEN tok = ProximoToken();
   switch (tok) {
      case ID: /* <sentencia> -> ID := <expresion>; */
         Match(ID); Match(Asignación>); Expresión(); Match(PUNTOYCOMA);
         break;
      case LEER: /* <sentencia> -> LEER ( <listaIdentificadores> ); */
         Match(LEER); Match(PARENIZQUIERDO); ListaIdentificadores();
         Match(PARENDERECHO); Match(PUNTOYCOMA);
         break;
      case ESCRIBIR: /* <sentencia> -> ESCRIBIR (<listaExpresiones>); */
         Match(ESCRIBIR); Match(PARENIZQUIERDO); ListaExpresiones();
         Match(PARENDERECHO); Match(PUNTOYCOMA);
         break;
      default:
         ErrorSintactico(tok); break;
   }
}
```

Desarrollemos otro PAS, con una dificultad diferente a los anteriores, aunque siempre reflejando, en su construcción, la estructura de la correspondiente producción de la GIC:

```
void Expresion (void) {
/* <expresion> -> <primaria> {<operadorAditivo> <primaria>} */
   TOKEN t;
   Primaria();
   for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken())
   {
     OperadorAditivo();  Primaria();
   }
}
```

* Ejercicio 31 *
Describa, en castellano, la tarea que realiza el PAS Expresion.

Por último, veamos otro PAS en el que aparece un llamado al procedimiento ErrorSintactico:

```
void OperadorAditivo (void) {
/* <operadorAditivo> -> uno de SUMA RESTA */
   TOKEN t = ProximoToken();
   if (t == SUMA || t == RESTA)
      Match(t);
   else
      ErrorSintactico(t);
}
```

* Ejercicio 32 *
Describa, en castellano, la tarea que realiza el PAS OperadorAditivo.

* Ejercicio 33 *
Analice los PAS ListaSentencias, Sentencia, Expresion y OperadorAditivo. Explique las diferencias entre sus implementaciones.

De esta manera, hemos comenzado con la construcción del Parser. Hasta el momento, se trata, básicamente, de un conjunto de procedimientos que denominamos PAS y que implementan las definiciones dadas por las producciones de la Gramática Sintáctica de Micro. Por supuesto, se ha visto que también hay un grupo de funciones auxiliares que contribuyen en la construcción del Parser.

* Ejercicio 34 *
¿Cuáles son las funciones auxiliares mencionadas hasta ahora?

* Ejercicio 35 *
Sea el programa Micro inicio escribir(4); fin
a) Escriba los lexemas que encuentra el Scanner;
b) Escriba la secuencia de llamadas a los PAS para realizar el Análisis Sintáctico de este programa.

3.2.6 LA ETAPA DE TRADUCCIÓN, LAS RUTINAS SEMÁNTICAS Y LA AMPLIACIÓN DE LOS PAS

La etapa de traducción de un programa en Micro, que pertenece a lo que se denomina Síntesis dentro de la compilación, consistirá en la generación de código para una máquina virtual (MV). Supongamos que esta MV tiene instrucciones con el siguiente formato:

OP A,B,C

donde OP es el código de operación, A y B designan a los operandos, y C especifica la dirección donde se almacena el resultado de la operación. Los operandos A y B pueden ser nombres de variables o constantes enteras. Para algunos códigos de operación, A o B o C pueden no ser usadas. El formato de la salida será una cadena de caracteres.

Ejemplo 7
He aquí una instrucción de esta máquina virtual:

```
Declara A,Entera
```

Esta instrucción, que declara a la variable A como entera, tiene `Declara` como código de operación y dos operandos; no utiliza el tercer operando.

Gran parte de la traducción es realizada por rutinas semánticas llamadas por el Parser. A la Gramática Sintáctica conocida se le pueden agregar símbolos de acción para especificar cuándo se debe realizar un procesamiento semántico.

Los símbolos de acción, indicados como #nombre, se pueden ubicar en cualquier lugar del lado derecho de una producción. A cada símbolo de acción le corresponde una rutina semántica implementada como una función o un procedimiento.

Ejemplo 8
A un símbolo de acción llamado #sumar le corresponderá la rutina semántica Sumar.

Cuando se crea un PAS, las invocaciones a las rutinas semánticas son insertadas en las posiciones designadas por los símbolos de acción.

Los símbolos de acción no forman parte de la sintaxis especificada por una GIC, sino que le agregan "comentarios" a esta GIC para indicar cuándo se necesitan ejecutar las acciones semánticas correspondientes.

Ejemplo 9
La producción para <operadorAditivo> con el agregado del símbolo de acción que le corrsponde, sería:

```
<operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op
```

3.2.6.1 INFORMACIÓN SEMÁNTICA

Al diseñar las rutinas semánticas, es muy importante la especificación de los datos sobre los cuales operan y la información que producen.

El enfoque utilizado en este caso es asociar un registro semántico a cada símbolo gramatical, tanto noterminal (por ejemplo, `<expresión>`) como terminal (por ejemplo, `ID`). Como caso especial, habrá ciertos símbolos con registros semánticos nulos, sin datos.

> *Ejemplo 10*
> El símbolo `PUNTOYCOMA` no requiere un registro semántico.

Como ejemplo representativo de la situación explicada en el párrafo anterior, consideremos la siguiente producción, ampliada con el agregado del correspondiente símbolo de acción:

```
<expresión> -> <primaria> + <primaria> #sumar
```

Por cada aparición del noterminal `<primaria>` en el lado derecho de esta producción, se generará un registro semántico. Cada registro semántico contendrá datos sobre cada operando, como, por ejemplo, dónde está almacenado y cuál es su valor.

Cuando la función Sumar es invocada como resultado de la aparición del símbolo de acción `#sumar`, se le deben pasar estos registros semánticos como argumentos. Como resultado, Sumar producirá un nuevo registro semántico correspondiente al noterminal <expresión> con la información necesaria.

Ejemplo 11
En el caso de Micro, tenemos solo estos dos registros semánticos:

1. REG_OPERACION, que solo contendrá el valor del token SUMA o RESTA.
2. REG_EXPRESION, que contendrá el tipo de expresión y el valor que le corresponde; este valor puede ser una cadena (para el caso de un identificador) o un número entero (para el caso de una constante). La cadena será almacenada en un vector de caracteres previamente definido y tendrá una longitud máxima de 32 caracteres, límite que se informa en la sección 3.2.1 con respecto a los identificadores.

3.2.6.2 LA GRAMÁTICA SINTÁCTICA DE MICRO CON LOS SÍMBOLOS DE ACCIÓN

En primer lugar, agregamos una nueva producción:

```
<identificador> -> ID #procesar_id
```

Esta producción es muy útil porque `ID` aparece en varios contextos diferentes de la GIC, y necesitamos llamar a la función Procesar_ID inmediatamente después que el Parser haya encontrado el correspondiente `ID`, para acceder a los caracteres que se encuentran en el buffer y así construir el registro semántico apropiado.

Además, se utilizarán algunas funciones auxiliares en la construcción del compilador:

1. Generar: una función que recibe cuatro argumentos que son cadenas, que corresponden al código de operación y a los tres operandos de cada instrucción de la MV; esta función producirá la correspondiente instrucción en el flujo de salida.
2. Extraer: una función tal que dado un registro semántico, retorna la cadena que contiene. Esta cadena puede ser un identificador, un código de operación, representar una constante antes de ser convertida a número entero, etc.

Dado que la Tabla de Símbolos de Micro es muy simple porque el LP es muy sencillo, las rutinas que se necesitan para operar con ella son solo dos:

1. Buscar: una función tal que dada una cadena que representa a un identificador, determina si ya se encuentra en la Tabla de Símbolos.
2. Colocar: almacena una cadena en la Tabla de Símbolos.

Una rutina auxiliar utilizada por varias rutinas semánticas es Chequear. He aquí su implementación en ANSI C:

```
void Chequear (string s) {
   if (! Buscar(s)) { /* ¿la cadena está en la Tabla de Símbolos? No: */
      Colocar(s);     /* almacenarla, es el nombre de una variable */
      Generar("Declara", s, "Entera", ""); /* genera la instrucción */
   }
}
```

* Ejercicio 36 *
El parámetro de `Chequear` dice: `string s`. Defina en ANSI C el tipo `string`.

He aquí tres funciones auxiliares en ANSI C que son necesarias para definir las rutinas semánticas que corresponden a los símbolos de acción de Micro:

```
void Comenzar (void); /* inicializaciones semánticas */

void Terminar (void) {
/* genera la instrucción para terminar la ejecución del programa */
   Generar("Detiene", "", "", "");
}

void Asignar (REG_EXPRESION izquierda, REG_EXPRESION derecha) {
/* genera la instrucción para la asignación */
   Generar("Almacena", Extraer(derecha), izquierda.nombre, "");
}
```

La Gramática Sintáctica para Micro, ampliada con los símbolos de acción correspondientes, será:

```
<objetivo> -> <programa> FDT #terminar
<programa> -> #comenzar inicio <listaSentencias> fin
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> <identificador> := <expresión> #asignar ; |
               read ( <listaIdentificadores> ); |
               write ( <listaExpresiones> );
<listaIdentificadores> -> <identificador> #leer_id
                                          {, <identificador> #leer_id}
<listaExpresiones> -> <expresión> #escribir_exp
                                         {, <expresión> #escribir_exp}
<expresión> -> <primaria> {<operadorAditivo> <primaria> #gen_infijo}
<primaria> -> <identificador> |
              CONSTANTE #procesar_cte |
              ( <expresión> )
<operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op
<identificador> -> ID #procesar_id
```

3.2.6.3 ALGUNAS RUTINAS SEMÁNTICAS

Veamos, ahora, algunas de las rutinas semánticas utilizadas en el compilador de Micro.

```
void Leer (REG_EXPRESION in) {
/* genera la instrucción para leer */
   Generar("Read", in.nombre, "Entera", "");
}
```

```
REG_EXPRESION ProcesarCte (void) {
/* convierte cadena que representa número a número entero y construye un
   registro semántico */
   REG_EXPRESION t;
   t.clase = CONSTANTE;
   sscanf (buffer, "%d", &t.valor);
   return t;
}
```

* Ejercicio 37 *
Investigue e informe cómo se define el registro semántico de tipo `REG_EXPRESION` en ANSI C.

* Ejercicio 38 *
Investigue e informe sobre la función sscanf de ANSI C.

Otras rutinas semánticas interesantes son:

```
void Escribir (REG_EXPRESION out) {
   Generar("Write", Extraer(out), "Entera", "");
}
```

```
REG_EXPRESION ProcesarId (void) {
/* Declara ID y construye el correspondiente registro semántico */
   REG_EXPRESION t;
   Chequear(buffer);  t.clase = ID;
   strcpy(t.nombre, buffer);
   return t;
}
```

* Ejercicio 39 *
a) Investigue e informe sobre la función strcpy de ANSI C.
b) ¿Qué diferencia encuentra entre sscanf y strcpy?

3.2.6.4 PROCEDIMIENTO DE ANÁLISIS SINTÁCTICO (PAS) CON SEMÁNTICA INCORPORADA

Recordemos uno de los PAS que hemos descripto oportunamente:

```
void Expresion (void) {
/* <expresion> -> <primaria> {<operadorAditivo> <primaria>} */
   token t;
   Primaria();
   for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken())
   {
     OperadorAditivo();  Primaria();
   }
}
```

Si ahora le agregamos el procesamiento semántico, vemos cómo se modifica este procedimiento y cuál es su contenido definitivo:

```
void Expresion (REG_EXPRESION *resultado) {
/* <expresión> -> <primaria> {<operadorAditivo> <primaria> #gen_infijo}*/
   REG_EXPRESION operandoIzq, operandoDer;
   REG_OPERACION op;
   token t;
   Primaria(&operandoIzq);
   for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken())
   {
      OperadorAditivo(&op);  Primaria(&operandoDer);
      operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
   }
   *resultado = operandoIzq;
}
```

> * Ejercicio 40 *
> Investigue e informe el significado del asterisco en el parámetro y en la última sentencia de este procedimiento.

En el procedimiento Expresion se invoca a la función GenInfijo; esta función tiene la siguiente estructura general y realiza la siguiente actividad:

```
REG_EXPRESION GenInfijo (REG_EXPRESION e1, REG_OPERACION op,
                         REG_EXPRESION e2) {
/* Genera la instrucción para una operación infija y construye un
   registro semántico con el resultado */
. . .
}
```

* Ejercicio 41 *
Describa con precisión qué hace la nueva función Expresion.

3.2.6.5 UN EJEMPLO DE ANÁLISIS SINTÁCTICO DESCENDENTE Y TRADUCCIÓN

Como un ejemplo final, consideremos el proceso de compilación del siguiente programa en Micro:

```
inicio A := BB – 34 + A; fin
```

Recordemos que a todo programa fuente, el compilador le agrega el token fdt (fin de texto) para detectar el fin del flujo de entrada a compilar.

A continuación figura una tabla en la que se describen 35 pasos con las acciones tomadas por el Parser en la medida que procesa este programa. Es conveniente que lo verifique, revisando las diferentes funciones desarrolladas previamente.

Paso	Acción del Parser	Programa sin Procesar	Instruc. Generada para MV
1	Llamar Objetivo()	inicio A:=BB–34+A; fin FDT	
2	Llamar Programa()	inicio A:=BB–34+A; fin FDT	
3	Acción Semántica Comenzar()	inicio A:=BB–34+A; fin FDT	
4	Match(INICIO)	inicio A:=BB–34+A; fin FDT	
5	Llamar ListaSentencias()	A:=BB–34+A; fin FDT	
6	Llamar Sentencia()	A:=BB–34+A; fin FDT	
7	Llamar Identificador()	A:=BB–34+A; fin FDT	
8	Match(ID)	A:=BB–34+A; fin FDT	
9	Acción Semántica ProcesarId()	:=BB–34+A; fin FDT	Declara A,Entera
10	Match(ASIGNA)	:=BB–34+A; fin FDT	
11	Llamar Expresion()	BB–34+A; fin FDT	
12	Llamar Primaria()	BB–34+A; fin FDT	
13	Llamar Identificador()	BB–34+A; fin FDT	
14	Match(ID)	BB–34+A; fin FDT	
15	Acción Semántica ProcesarId()	–34+A; fin FDT	Declara BB,Entera
16	Llamar OperadorAditivo()	–34+A; fin FDT	
17	Match(RESTA)	–34+A; fin FDT	
18	Acción Semántica ProcesarOp()	34+A; fin FDT	
19	Llamar Primaria()	34+A; fin FDT	
20	Match(CONSTANTE)	34+A; fin FDT	
21	Acción Semántica ProcesarCte()	+A; fin FDT	
22	Acción Semántica GenInfijo()	+A; fin FDT	Declara Temp&1,Entera Resta BB,34,Temp&1
23	Llamar OperadorAditivo()	+A; fin FDT	
24	Match(SUMA)	+A; fin FDT	
25	Acción Semántica ProcesarOp()	A; fin FDT	
26	Llamar Primaria()	A; fin FDT	
27	Llamar Identificador()	A; fin FDT	

28	Match(ID)	A; fin FDT	
29	Acción Semántica ProcesarId()	; fin FDT	
30	Acción Semántica GenInfijo()	; fin FDT	Declara Temp&2,Entera Suma Temp&1,A,Temp&2
31	Acción Semántica Asignar()	; fin FDT	Almacena Temp&2,A
32	Match(PUNTOYCOMA)	; fin FDT	
33	Match(FIN)	fin FDT	
34	Match(FDT)	FDT	
35	Acción Semántica Terminar()		Detiene

Se concluye que la compilación terminó correctamente y que para el programa Micro:

inicio A := BB – 34 + A; **fin**

generó las siguientes instrucciones para la MV:

```
Declara A,Entera
Declara BB,Entera
Declara Temp&1,Entera
Resta BB,34,Temp&1
Declara Temp&2,Entera
Suma Temp&1,A,Temp&2
Almacena Temp&2,A
Detiene
```

4 ANÁLISIS LÉXICO, ANÁLISIS SINTÁCTICO Y ANÁLISIS SEMÁNTICO

Luego que el capítulo anterior mostrara una introducción al diseño de un compilador y algunos detalles de cómo se construye uno para un LP muy simple, nos ocuparemos ahora de los tres ingredientes que forman la etapa de compilación conocida como ANÁLISIS: el Análisis Léxico, el Análisis Sintáctico y el Análisis Semántico. También veremos cómo la llamada Tabla de Símbolos interactúa con todos ellos y, además, ciertos conceptos importantes sobre la misma.

4.1 INTRODUCCIÓN A LA TABLA DE SÍMBOLOS

Dado que la Tabla de Símbolos interactúa con las tres fases de análisis, es conveniente que, primero, veamos algunos conceptos fundamentales de la misma.

La Tabla de Símbolos, también conocida como Diccionario, es un conjunto de estructuras de datos que se utiliza, como mínimo, para contener todos los identificadores del programa fuente que se está compilando, junto con los atributos que posee cada identificador.

En el caso que estos identificadores sean nombres de variables, sus atributos son: tipo y ámbito (la parte del programa donde tiene validez). En el caso que los identificadores sean nombres de rutinas (funciones, procedimientos), algunos atributos son: cantidad de parámetros y tipo de cada uno, métodos de transferencia, tipo del valor que retorna. Estos atributos se codifican mediante números enteros o valores por enumeración.

Muchos diseñadores de compiladores también usan la Tabla de Símbolos para almacenar en ella todas las palabras reservadas del Lenguaje de Programación; cada una de ellas tendrá el atributo que indica que es una palabra reservada.

Ejemplo 1
Sea la siguiente función en ANSI C:

```
int XX (double a) {
   char s[12];
   double b;
   b = a + 1.2;
   if (b > 4.61) return 1;
   else return 0;
}
```

Esta función posee diferentes identificadores: palabras reservadas, nombre de una función y nombres de variables.

El identificador `XX` debe poseer atributos que indiquen que es una función, que retorna un valor de tipo int, que tiene un parámetro y que ese parámetro es de tipo double. Podríamos representarlo de esta manera: (XX, *funcion, int, 1, double*). Nótese que, en esta representación, XX es una secuencia de caracteres porque es el identificador, pero los restantes elementos son los atributos de XX y se representan mediante números enteros o valores por enumeración.

* Ejercicio 1 *
Escriba cómo representaría a los restantes identificadores de la función del Ejemplo 1.

4.2 EL ANÁLISIS LÉXICO

El Análisis Léxico, realizado por el Scanner, es el proceso que consiste en recorrer el flujo de caracteres que forman el Programa Fuente, detectar los lexemas que componen este programa, y traducir la secuencia de estos lexemas en una secuencia de tokens cuya representación es más útil para el resto del compilador.

```
Flujo de Caracteres -> Secuencia de Lexemas -> Secuencia de Tokens
```

* Ejercicio 2 *
Defina dos tokens que puedan estar formados por más de 50 lexemas cada uno.

* Ejercicio 3 *
¿Verdadero o Falso?

a) Un lexema es una secuencia de uno o más caracteres.
b) El espacio entre dos lexemas es un lexema.
c) Un token es un valor.
d) Un token puede ser representado mediante un número entero.

En cada caso, justifique su respuesta.

Debemos tener en cuenta que el Análisis Léxico es un proceso que se realiza sobre palabras (los lexemas) de Lenguajes Regulares (los tokens). Cuando estos LRs son infinitos, sus lexemas son obtenidos recién cuando se detecta un carácter espúreo llamado centinela. En cambio, si los LRs son finitos, pueden requerir o no un centinela para obtener el lexema correspondiente.

Ejemplo 2
Sea la siguiente función en ANSI C:

```
int Abcd (void) {
   int a, b = 4;
   a = b + 18;
   return a;
}
```

1º caso: El nombre de la función, Abcd, es un identificador. Como pertenece a un LR infinito, requiere un centinela para saber cuándo termina; en este caso, el centinela es el carácter *espacio*.

2º caso: El carácter {, que pertenece a un LR (token) finito, no requiere un centinela porque, en ANSI C, no hay lexemas que comiencen con { y que luego puedan tener más caracteres.

3º caso: Pero, ¿qué ocurre con el operador +? Este lexema también pertenece a un LR finito, el lenguaje de los operadores. Este LR contiene otros dos operadores que comienzan con +: el operador incremento (++) y el operador de asignación combinado con la suma (+=). En consecuencia, cuando el Scanner encuentra el carácter + no puede afirmar que encontró un lexema; necesita conocer la existencia de un centinela para confirmarlo.

* Ejercicio 4 *
Investigue e informe si los siguientes lexemas en ANSI C requieren un centinela para ser detectados:

a) el identificador abc
b) el carácter ,
c) el operador >
d) la constante 2.14

e) la palabra reservada for
f) el literal-cadena "abc"
g) el operador !
h) el operador ++

Nota 1
Recuerde que cada vez que el Scanner requiere conocer un centinela para detectar un lexema, luego debe retornarlo al flujo de entrada porque ese carácter será el primero a tratar en la búsqueda del próximo lexema.

* Ejercicio 5 *
¿Qué token identificador tiene más lexemas que representen objetos diferentes, el de Pascal o el de ANSI C?

* Ejercicio 6 *
Supongamos que existe un LP que solo tiene tres operadores: /, // y //+. Explique cómo hace el Scanner para detectar cada uno de estos lexemas.

Una de las primeras decisiones que debe tomar el diseñador de un compilador es la elección del conjunto de tokens, es decir: el conjunto de Lenguajes Regulares diferentes que será reconocido durante el Análisis Léxico.

* Ejercicio 7 *
¿Por qué es importante la afirmación anterior?

Como resultado de la actividad del Scanner, cada token se representa mediante un número entero o un valor de un tipo enumerado, y así será reconocido por el resto del compilador.

Pero no es suficiente con obtener el token al cual pertenece cierto lexema. Las fases siguientes del compilador también requieren conocer el lexema. No nos olvidemos de la segunda etapa del compilador – la Síntesis – y del programa objeto que se obtiene, como hemos visto en el capítulo anterior, aunque éstos no sean temas para profundizar en este libro.

Ejemplo 3
Sean los operadores de comparación >, >=, < y <=. Supongamos que el diseñador del compilador decide que estos cuatro lexemas pertenecen al token "Operador de Comparación". Entonces, además de conocerse el token, el lexema debe ser utilizado para distinguir cada operador de comparación entre los cuatro que componen este token; y esto es fundamental para generar el programa objeto.

En otras palabra: si bien la distinción entre estos cuatro operadores de comparación no afecta la actividad del Análisis Sintáctico, el Scanner deberá retornar el par ordenado (operadorComparación, cuálEs) para que en el Análisis Semántico y en la etapa de Síntesis se conozca la información exacta, y así se puedan generar las instrucciones correspondientes.

En muchos casos, los lexemas son almacenados en la Tabla de Símbolos. En un LP simple, que solo tenga variables globales y declaraciones, es común que el Scanner almacene un identificador ni bien lo detecta, si todavía no se encuentra en la TS. Además de identificadores, a menudo también se almacenan las palabras reservadas, como se ha visto en el capítulo 3, y los literales-cadena. De ser así, cada lexema se representará mediante un índice que indica la posición

que ocupa en la TS. Entonces, el Scanner retornará el par ordenado (palabraReservada, índice) o (identificador, índice) o (literal-cadena, índice), según corresponda.

En el caso de las constantes numéricas, la situación es más compleja y se presentan, normalmente, dos soluciones:

1º Hay Scanners que guardan la secuencia de caracteres que forma cada constante numérica, sea entera o real, en la TS. En este caso, el Scanner retorna (tipoConstanteNumérica, índice);

2º Hay otro diseño de Scanners que convierte la secuencia de caracteres que forma la constante al valor numérico que corresponde, sin guardarlo en la TS; en este caso, el Scanner retorna el par ordenado (tipoConstanteNumérica, valor).

* Ejercicio 8 *
a) Analice la explicación sobre el par ordenado que retorna el Scanner.
b) ¿Cómo codificaría palabraReservada, identificador y tipoConstanteNumérica?

* Ejercicio 9 *
Si usted diseña un Scanner, ¿qué retornaría al detectar una constante carácter?

Analicemos, ahora, que sucede con Lenguajes de Programación con estructuras de bloque, como Pascal, ANSI C, C++, Java y muchos otros. El Scanner no puede almacenar ni buscar un identificador en la Tabla de Símbolos porque el identificador puede pertenecer a distintos bloques, con diferentes atributos, y el Scanner no tiene la capacidad de distinguir los diferentes bloques.

Ejemplo 4
Sea el siguiente fragmento de un programa en ANSI C:

```
. . .
{ /* bloque 1 */
  int a;
  . . .
  { /* bloque 2 */
    char a[20];
    . . .
  }
  . . .
}
{ /* bloque 3 */
  struct {int x;} a;
  . . .
}
. . .
```

Como se observa, en este ejemplo hay tres bloques; además, el bloque 2 está anidado dentro del bloque 1. En total, hay tres variables con el mismo nombre y, por lo que se ve en las diferentes declaraciones, con atributos muy distintos. Si el Scanner fuera el encargado de almacenar estos identificadores en la TS, encontraría que el segundo identificador "ya está almacenado" (¿?) porque el Scanner no conoce la sintaxis del Lenguaje de Programación y, por lo tanto, no sabe qué es un bloque.

La solución es que el Scanner retorne el identificador detectado y que una rutina semántica resuelva el papel que juega ese identificador, según el bloque en el que está declarado. Por ello, normalmente el Scanner utiliza un espacio de cadenas, que es un gran vector de caracteres, para

almacenar los identificadores y retorna un puntero a este espacio. Posteriormente, en la TS se almacena dónde comienza el identificador en este gran espacio y su longitud.

* Ejercicio 10 *
Investigue e informe si el bloque en el que está declarado debería ser otro atributo para un identificador.

* Ejercicio 11 *
Investigue e informe: ¿Se puede construir un compilador que realice el Análisis Léxico de todo el programa fuente, almacene el resultado de esta operación en un archivo de texto y luego continúe el proceso de compilación?
– Si responde que NO, justifique su respuesta.
– Si responde que SI, describa cómo sería el diseño del archivo de texto de salida y ejemplifique este archivo a partir del texto de una función escrita en ANSI C que tenga, al menos, dos bloques.

Hay una situación interesante que produce un cambio de atributo. Supongamos la siguiente declaración en ANSI C:

```
typedef int entero;
```

Cuando el Scanner analiza esta secuencia de caracteres, detecta cuatro lexemas: la palabra reservada typedef, la palabra reservada int, el identificador entero y el carácter de puntuación ; (“punto y coma”). En el caso del identificador entero, el Scanner retornará un puntero al espacio de caracteres y el atributo `ID`. Pero la declaración typedef crea una nueva palabra clave: entero. Después que el Parser procesa esta sentencia, él mismo, conjuntamente con la rutina semántica correspondiente, se encarga de colocar el atributo correcto, del identificador entero, en la TS: un código que represente “nombre de tipo”.

* Ejercicio 12 *
Sea la declaración en ANSI C: `typedef char strings[200][32];`
a) ¿Qué lexemas detecta el Scanner?
b) ¿Qué información tendrá el identificador strings en la TS después que el Parser procese esta declaración?

Nota 2
Las técnicas usadas en el Análisis Léxico son útiles en muchas otras aplicaciones de programación, no solo en la construcción de compiladores. Un Scanner es un reconocedor de patrones, y este reconocimiento puede ser aplicado en programas como editores y manejadores de bases de datos bibliográficos, por ejemplo.

4.2.1 RECUPERACIÓN DE ERRORES

Ciertos caracteres son ilegales en ANSI C porque no pertenecen a los alfabetos de ninguno de los LRs que forman este Lenguaje de Programación. Esto es lo que ocurre con caracteres como @ (arroba) y ` (apóstrofo invertido), excepto en dos circunstancias especiales.

* Ejercicio 13 *
¿En qué dos circunstancias estos caracteres no son ilegales en ANSI C?

El Scanner se puede recuperar de estos errores de varias maneras; la más simple es descartar el carácter inválido, imprimir un mensaje de error adecuado, pasar un código de error al Parser y continuar el Análisis Léxico a partir del siguiente carácter. Obviamente, el resultado final de todo el proceso ya no será una compilación correcta.

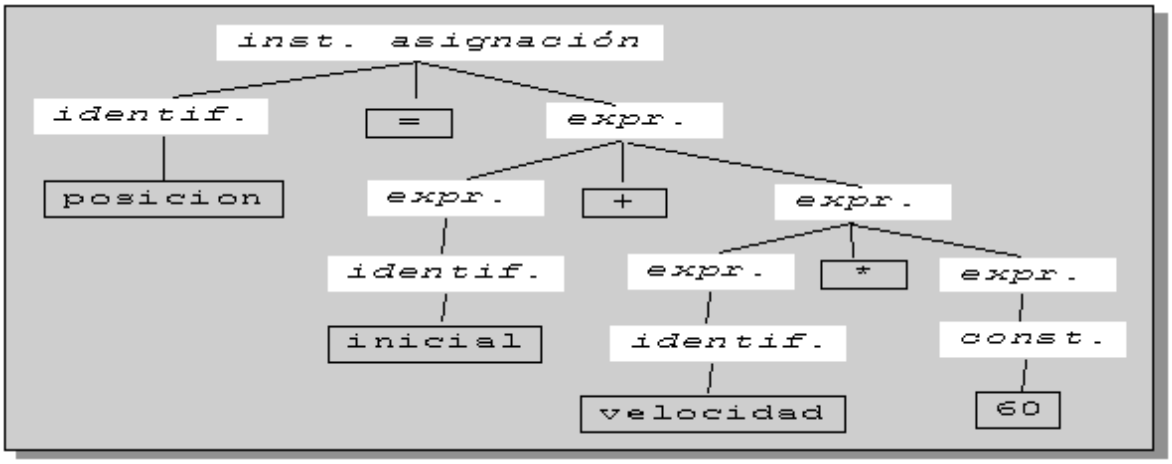
4.3 EL ANÁLISIS SINTÁCTICO

El Análisis Sintáctico es llevado a cabo por el Parser. Éste verifica si los tokens que recibe del Scanner forman secuencias o construcciones válidas, según la Gramática Sintáctica del LP correspondiente.

El Parser está formado por un grupo de rutinas que convierte el flujo de tokens en un árbol de análisis sintáctico, generalmente implícito. Este árbol representa, de un modo jerárquico, a la secuencia analizada, donde los tokens que forman la construcción son las hojas del árbol.

Ejemplo 5
Veamos un AAS que figura en un sitio no confiable de internet y hagamos un ejercicio sobre este árbol. Reemplace "instrucción" por "sentencia", que es el término utilizado, habitualmente, en los lenguajes de alto nivel (el término *instrucción* se emplea más en lenguajes de bajo nivel).

El árbol muestra otra forma de derivación; a diferencia de lo que ocurre en la derivación vertical, aquí no sabemos en qué orden se han aplicado todas las producciones:



* Ejercicio 14 *
a) Los nodos del árbol representan noterminales y terminales de la GIC. Realice una lista de los noterminales y otra de los terminales que figuran en este árbol.
b) La raíz del árbol, ¿a qué corresponde en la GIC?
c) Suponiendo que el LP es ANSI C, ¿cuál es el error que se detecta en este árbol?
d) Corrigiendo ese error, ¿cuál es la construcción que representa este árbol?
e) En base a la información provista por este árbol, construya una derivación por izquierda que obtenga la secuencia de terminales representada en el árbol.

4.3.1 TIPOS DE ANÁLISIS SINTÁCTICOS Y DE GICs

Al derivar una secuencia de tokens, si existe más de un noterminal en una cadena de derivación debemos elegir cuál es el próximo noterminal que se va a expandir, es decir: cuál será reemplazado por su lado derecho de la producción. Por ello se utilizan dos tipos de derivaciones que determinan con precisión cuál será el noterminal a tratar: la derivación a izquierda y la derivación a derecha, ambas vistas ya en el Volumen 1.

Repasando lo visto en el Volumen 1 y en esta etapa de la compilación, una DERIVACIÓN es una forma de mostrar cómo una construcción (declaración, expresión, sentencia, el programa completo) del flujo de tokens se puede obtener a partir de una Gramática Sintáctica dada. Este proceso puede ser:

1. DERIVACIÓN POR IZQUIERDA (o *Derivación a Izquierda*): ocurre cuando siempre se reemplaza el primer noterminal que se encuentre en una cadena de derivación leída de izquierda a derecha;

2. DERIVACIÓN POR DERECHA (o *Derivación a Derecha*): ocurre cuando siempre reemplaza el último noterminal de la cadena de derivación leída de izquierda a derecha.

Es importante destacar la relación que existe entre el tipo de Análisis Sintáctico y el tipo de Derivación.

El Análisis Sintáctico Descendente (*top-down*, en inglés) produce una Derivación por Izquierda, que comienza en el noterminal llamado axioma y finaliza con los terminales que forman la construcción analizada. En otras palabras, "despliega el árbol de análisis sintáctico" hasta llegar a sus hojas.

La Derivación por Derecha se utiliza en el Análisis Sintáctico Ascendente (*bottom-up*, en inglés), pero de una manera especial. En principio, tengamos en cuenta que la Derivación, sea por izquierda o por derecha, siempre comienza en el axioma de la GIC y finaliza con una secuencia de terminales.

Un Parser Ascendente utiliza una Derivación a Derecha, pero en orden inverso, esto es: la última producción aplicada en la Derivación a Derecha, es la primera produción que es "descubierta", mientras que la primera producción utilizada, la que involucra al axioma, es la última producción en ser "descubierta". En otras palabras, "reduce el árbol de análisis sintáctico" hasta llegar al axioma. En conclusión: la secuencia de producciones reconocida en el Análisis Sintáctico Ascendente es exactamente la inversa a la secuencia de producciones que forma la Derivación a Derecha. (Este proceso ya se ha visto en el Volumen 1, página 38, al evaluar una expresión).

Ejemplo 6
Partimos de una GIC muy simple para mostrar los dos tipos de derivaciones y de análisis sintácticos. Sea la GIC con producciones:

```
1 S -> aST
2 S -> b
3 T -> cT
4 T -> d
```

Supongamos que debemos analizar si la construcción **aabcdd** es sintácticamente correcta.

1º Derivación a Izquierda – Análisis Sintáctico Descendente (comienza en el axioma)

Cadena de Derivación Obtenida	Próxima Producción a Aplicar
S (axioma)	S -> aST (1)
aST	S -> aST (1)
aaSTT	S -> b (2)
aabTT	T -> cT (3)
aabcTT	T -> d (4)
aabcdT	T -> d (4)
aabcdd	correcta

Nota 3

Si no recuerda qué es una cadena de derivación, repase su definición en el Volumen 1, página 28, última frase.

2º Derivación a Derecha

Cadena de Derivación Obtenida	Próxima Producción a Aplicar
S (axioma)	S -> aST (1)
aST	T -> d (4)
aSd	S -> aST (1)
aaSTd	T -> cT (3)
aaScTd	T -> d (4)
aaScdd	S -> b (2)
aabcdd	

3º Análisis Sintáctico Ascendente (a partir de la tabla anterior, en orden inverso)

Cadena de Derivación	Próxima Producción a Aplicar
aabcdd	S -> b (2)
aaScdd	T -> d (4)
aaScTd	T -> cT (3)
aaSTd	S -> aST (1)
aSd	T -> d (4)
aST	S -> aST (1)
S	correcta

* Ejercicio 15 *

a) Dada la misma GIC del Ejemplo anterior, aplique los dos tipos de Análisis Sintácticos para verificar si la construcción **aaabddccd** es correcta.

b) En el Análisis Sintáctico Ascendente, y con lo visto en el ejemplo, explique qué ocurre si se detecta un Error Sintáctico.

Si bien las GICs son muy utilizadas para definir la sintaxis de un LP, hay reglas sintácticas que no pueden ser expresadas empleando solo GICs. Por ejemplo, la regla que indica que toda variable debe ser declarada antes de ser usada no puede ser expresada en una GIC. En la práctica los detalles sintácticos que no pueden ser representados en una GIC son considerados parte de la semántica estática y son chequeados por rutinas semánticas.

ERRORES EN LAS GICs. Una GIC es un mecanismo utilizado para definir una sintaxis. Una GIC puede tener errores que deben evitarse antes de construir un compilador.

Uno de los errores más importantes surge cuando una GIC permite que un programa tenga una construcción con dos o más árboles sintácticos diferentes, que es equivalente a decir dos o más derivaciones por izquierda (por derecha) diferentes.

Consideremos la GIC con dos producciones:

```
E -> E + E
E -> num
```

Si derivamos la expresión `num+num+num` obtenemos dos derivaciones por izquierda diferentes, como vemos a continuación:

Derivación 1

```
E
E + E
num + E
num + E + E
num + num + E
num + num + num
```

Derivación 2

```
E
E + E
E + E + E
num + E + E
num num + E
num + num + num
```

Una que GIC que permite dos derivaciones a izquierda para obtener la misma cadena de terminales se denomina ambigua. Estas gramáticas deben ser evitadas.

* Ejercicio 16 *
a) Muestre esta ambigüedad con una derivación a derecha;
b) Muestre la misma ambigüedad con árboles sintácticos.

4.3.1.1 GRAMÁTICAS LL Y LR

Como hemos visto, los conceptos de derivación por izquierda y derivación por derecha se aplican directamente a los dos tipos de Análisis Sintácticos más comunes (Descendente y Ascendente). La derivación es un proceso muy cercano al que utiliza el Parser para analizar una construcción dada.

Un Análisis Sintáctico LL consiste en analizar el flujo de tokens de izquierda a derecha – eso representa la primera L, de *left* – por medio de una derivación por izquierda (esa es la segunda L, de *left*). El ASDR que utilizamos en el capítulo 3 es un ejemplo de LL.

Una gramática LL es una GIC que puede ser utilizada en un Análisis Sintáctico LL. No todas las gramáticas son LL, como veremos más adelante; pero, afortunadamente, la mayoría de los LP pueden describir sus respectivas sintaxis por medio de las gramáticas LL.

Una gramática LL especial es la llamada LL(1). Esta gramática puede ser utilizada por un Parser LL con un solo símbolo de preanálisis. Esto es: si un noterminal tiene varias producciones, el Parser puede decidir cuál lado derecho debe aplicar con solo conocer el próximo token del flujo de entrada.

Ejemplo 7
La GIC con producciones:

```
R -> aR | b
```

permite construir un Parser LL(1) – también conocido como Parser Predictivo – porque con solo conocer el próximo token (aquí representado por un carácter) de la secuencia analizada – es el llamado símbolo de preanálisis – determina, inequívocamente, si aplica una producción o la otra.

* Ejercicio 17 *
Analice si lo visto en el Ejemplo anterior se puede aplicar también a la GIC con producciones:

```
R -> aR | a
```

Justifique su respuesta.

Otro tipo de análisis sintáctico es el Análisis Sintáctico LR, que consiste en analizar el flujo de tokens de izquierda a derecha (esa es la L, de "left"), pero por medio de una derivación por derecha (esa es la R, de "right"), aunque utilizada a la inversa: la última producción *expandida* en la derivación será la primera en ser *reducida*. De todas las gramáticas LR, la que más nos interesa en este caso es la gramática LR(1), porque solo requiere conocer el próximo token (el símbolo de preanálisis) para hacer un Análisis Sintáctico correcto, como en el caso de la LL(1).

En general, un Parser LL(1) puede ser construido por un programador a partir de una GIC adecuada. En cambio, un Parser LR(1) requiere la utilización de un programa especial tipo *yacc*.

4.3.2 GRAMÁTICAS LL(1) Y APLICACIONES

En el Volumen 1 hemos visto que la sintaxis de un Lenguaje de Programación se define mediante una GIC. Completamos esta frase agregando que la GIC no puede definir aquello que sea sensible al contexto, como ya hemos visto.

> * Ejercicio 18 *
> Describa dos circunstancias sensibles al contexto en Pascal y otras dos en ANSI C que no pueden representarse en las respectivas GICs.

Supongamos que un LP tiene una construcción que consiste en una lista de expresiones aritméticas simples – con operadores de suma y producto, sin paréntesis –, separadas con "punto y coma" (;). La siguiente sería una GIC que genera este LIC:

GIC 1

```
1  listaExpresiones: expresión |
2                    listaExpresiones ; expresión
3  expresión: término |
4             expresión + término
5  término: factor |
6           término * factor
7  factor: num |
8          ( expresión )
```

* Ejercicio 19 *
a) Escriba tres listas con los elementos utilizados en la GIC anterior: una con los noterminales, otra con los terminales y una tercera con los metasímbolos.
b) Escriba dos ejemplos diferentes en los que se utilicen todos los terminales que hay en esta GIC 1.

Como se observa, la GIC descripta es recursiva a izquierda (vea las producciones 2, 4 y 6). Por un lado, las producciones recursivas son muy importantes porque permiten generar y describir un LIC infinito con un número finito de producciones. Además, desde el punto de vista de la documentación, esta GIC es muy útil para comprender todas las posibles listas de expresiones correctas que se pueden generar.

Sin embargo, la GIC 1 tiene un problema muy importante para aplicarla al Análisis Sintáctico Descendente de estas listas de expresiones. Muchos Parsers, como el ASDR que hemos visto en el capítulo anterior, no pueden manejar las producciones recursivas a izquierda porque los procedimientos que implementan esas producciones recursivas entrarían en un ciclo infinito.

Ejemplo 8
Volvamos a la GIC 1. Si el Parser tiene que tratar *factor*, definido en las producciones 7 y 8, no habrá inconvenientes. Sabe cómo elegir el lado derecho de la producción con solo conocer el primer símbolo o token:

a) si éste es un **num**, el Parser aplicará la producción 7 y reemplazará *factor* por **num**;
b) y si es un **(** aplicará la producción 8.

No sucede lo mismo con las producciones 5 y 6:

```
5 término: factor |
6          término * factor
```

En el caso de la producción 6, su lado derecho comienza con *término* y, por lo tanto, el procedimiento entraría en un ciclo indefinido.

En definitiva, el Parser de una GIC LL(1) debe ser capaz de elegir entre los distintos lados derechos de un determinado noterminal con solo conocer el próximo token. Y esto no lo puede hacer con las producciones 5 y 6 de la GIC 1, como veremos a continuación.

Ejemplo 9
Comparemos las siguientes derivaciones que se obtienen utilizando las producciones 5, 6 y 7:

```
. . .
término
factor
num
```

y

```
. . .
término
término * factor
factor * factor
num * factor
. . .
```

En la primera derivación utilizamos las producciones 5 y 7, mientras que en la segunda derivación empleamos las producciones 5, 6 y 7. Pero en ambos casos obtenemos cadenas de derivación que comienzan con el mismo terminal (num) y este hecho no es correcto en el método de Análisis Sintáctico que estamos viendo.

Esta situación que ocurre cuando el Parser no puede decidir qué producción aplicar se llama CONFLICTO y una de las tareas más difíciles en el diseño de un compilador es la creación de una GIC que no tenga conflictos.

Varias técnicas poducen GICs equivalentes y sin conflictos. Si bien las GICs resultantes no son útiles para una documentación clara del Lenguaje de Programación generado, sí lo son para el Análisis Sintáctico.

Repasemos: LL(1) es una GIC que trabaja con un Parser que lee la entrada de izquierda a derecha y que realiza una derivación por izquierda. LL(1) es muy útil porque solo necesita conocer un símbolo de preanálisis (un token) para realizar la derivación por izquierda. Por ello, a continuación veremos cómo transformar una GIC, que genera y describe la sintaxis de un LP, para que esa GIC tenga la propiedad de ser LL(1).

4.3.3 OBTENCIÓN DE GRAMÁTICAS LL(1)

Para ser útiles y eficientes, los Análisis Sintácticos Descendentes deben basarse en GICs que sean LL(1). El problema principal es que estas gramáticas no pueden ser recursivas a izquierda, aunque hemos visto, en el Volumen 1, que esta propiedad es muy útil para la descripción de la sintaxis de los Lenguajes de Programación.

Además, como hemos visto anteriormente, la GIC LL(1) tampoco puede tener un noterminal con dos o más producciones cuyos lados derechos comiencen con el mismo terminal.

Estas restricciones en las producciones de una GIC permitieron desarrollar, con cierta facilidad, los PAS (Procedimientos de Análisis Sintáctico) que hemos visto en el capítulo 3 de este volumen.

Veamos, entonces, algunas operaciones que nos permiten transformar la GIC empleada en la documentación de la sintaxis de un LP (como en el Volumen 1) en una GIC equivalente pero apta para ser utilizada por un Parser LL(1) o Parser Predictivo.

Para ello, y entre otras operaciones, deberemos construir el conjunto PRIMERO para cada noterminal de la GIC. Por el momento, lo introducimos informalmente de la siguiente manera: Si el noterminal *expresión* tiene varias producciones, entonces PRIMERO(*expresión*) sería el conjunto formado por todos los terminales (o tokens) que pueden comenzar una expresión. En el caso de ANSI C y otros LPs, este conjunto incluye números, el signo menos, el paréntesis que abre y otros tokens; pero nunca puede incluir al paréntesis que cierra.

* Ejercicio 20 *
Investigue e informe qué significa que un noterminal de una GIC tenga dos o más producciones con un PREFIJO común.

4.3.3.1 FACTORIZACIÓN A IZQUIERDA (cuando hay un prefijo común)

Sea

```
<sentencia if> -> if ( <condición> ) <sentencia> else <sentencia> |
                  if ( <condición> ) <sentencia>
```

Este par de producciones es muy útil para definir la sintaxis de la sentencia if en ANSI C, pero no es apto para una GIC LL(1) porque ambos lados izquierdos comienzan con el mismo token – más aun, con el mismo prefijo – y, por lo tanto, el Parser no sabrá cuál de las dos producciones seleccionar.

La solución es modificar las producciones de este noterminal de tal forma que el prefijo común quede aislado en una sola producción. En general, sea:

```
A -> αβ1 |
     . . .
     αβn
```

donde α es una secuencia de uno o más símbolos (el prefijo) y cada βi es una secuencia de cero o más símbolos, diferente para cada producción. Entonces, factorizando a izquierda, obtenemos estas producciones equivalentes:

```
A -> αB
B -> β1 |
     ...
     βn
```

Si aplicamos este algoritmo a la <sentencia if>, entonces las producciones:

```
<sentencia if> -> if ( <condición> ) <sentencia> else <sentencia> |
                  if ( <condición> ) <sentencia>
```

se transforman en las producciones equivalentes:

```
<sentencia if> -> if ( <condición> ) <sentencia> <opción else>
<opción else> -> else <sentencia> | ε
```

* Ejercicio 21 *
En la <sentencia if> original indique qué es α y qué es cada βi.

* Ejercicio 22 *
Resuelva, detalladamente, el problema de prefijos comunes que presenta la siguiente GIC, con producciones:

```
S -> aTbRba | aTbc
T -> a | Ta
R -> c | cRb
```

4.3.3.2 ELIMINACIÓN DE LA RECURSIVIDAD A IZQUIERDA

Como ya se ha informado al comenzar la sección 4.3.3, las gramáticas LL(1) no pueden ser recursivas a izquierda. Sin embargo, y como ya hemos visto en el Volumen 1, la recursividad a izquierda es necesaria para lograr la asociatividad a izquierda de un operador. Existe una forma de eliminar la recursividad a izquierda en una GIC, obteniendo una GIC equivalente (con la misma propiedad). La estrategia básica consiste en observar qué hacen las gramáticas recursivas a izquierda:

Sea el par de producciones:

```
X -> Xα | β
```

donde X, obviamente, es un noterminal, y α y β son secuencias de terminales y noterminales. Este par de producciones genera la secuencia βα*, escrita con los operadores de las Expresiones Regulares para que sea más compacta y más legible.

Como se ve a continuación, podemos eliminar la recursividad a izquierda para las producciones del noterminal X la siguiente manera:

X -> βZ	toda secuencia comienza con β
Z -> αZ | ε	seguida de cero o más α

Conclusión: hemos eliminado la recursividad a izquierda mediante una simple transformación a "recursiva a derecha".

* Ejercicio 23 *
Una expresión infija básica puede considerarse como una lista de operandos separados por operadores, excepto un caso especial; su definición más formal sería:

(1) un solo operando, o
(2) un operando seguido por uno o más pares operador/operando.

Esta estructura se puede definir mediante la siguiente GIC recursiva a izquierda:

```
<listaExp> -> <operando> | <listaExp> <operador> <operando>
```

Elimine la recursividad a izquierda.

* Ejercicio 24 *
Sea la GIC 1:

```
1  listaExpresiones: expresión |
2                    listaExpresiones ; expresión
3  expresión: término |
4             expresión + término
5  término: factor |
6           término * factor
7  factor: num |
8          ( expresión )
```

Conviértala en una GIC 2 que no sea recursiva a izquierda.

4.3.3.3 SÍMBOLOS DE PREANÁLISIS Y EL CONJUNTO PRIMERO

Para construir un Parser que realice un Análisis Sintáctico Descendente LL(1) es necesario que, para cada noterminal de la GIC, el Parser pueda determinar la producción a aplicar con solo conocer cuál es el símbolo de preanálisis, que es, simplemente, el próximo token del flujo de tokens a procesar. Aclaremos esta situación.

Sea la producción A -> X1...Xm, en la que cada Xi puede ser un terminal o un noterminal. Se necesita conocer el conjunto de todos los posibles símbolos de preanálisis que indican que esta producción será elegida. Este conjunto está formado, claramente, por aquellos terminales que pueden ser producidos por X1...Xm.

Como un símbolo de preanálisis siempre es un único terminal, se necesita el primer símbolo (de la izquierda) que puede ser producido por X1...Xm. Llamamos Primero(X1...Xm) al conjunto de "primer símbolo" que pueden ser producidos por X1...Xm.

4.3.3.3.1 OBTENCIÓN DEL CONJUNTO *PRIMERO*

Supongamos que existen dos o más producciones para un determinado noterminal. Para distinguir entre estas producciones y saber, entonces, cómo se *expande* ese noterminal, se debe examinar el conjunto Primero del lado derecho de cada una de las producciones. Este conjunto se define así:

Sea la producción `A->X1...Xm`; entonces Primero`(X1...Xm)` es el conjunto de terminales que pueden iniciar cualquier cadena de derivación que se obtenga a partir de `X1...Xm`. Si `X1...Xm` puede derivar en ε, entonces el conjunto Primero también contiene a ε. Una ayuda para comprender mejor esta definición la da la siguiente descripción del algoritmo para obtener Primero:

1º Si el primer símbolo, `X1`, es un terminal, entonces `Primero(X1...Xm) = {X1}`.

2º Si `X1` es un noterminal, entonces se calculan los conjuntos `Primero` para cada lado derecho de las producciones que tenga `X1`.

3º Si `X1` puede generar ε, entonces `X1` puede ser eliminada y, en consecuencia, `Primero(X1...Xm)` depende de `X2`.

5º Si `X2` es un terminal, entonces es incluido en `Primero(X1...Xm)`.

6º Si `X2` es un noterminal, entonces se calculan los conjuntos `Primero` para cada lado derecho de las producciones que tenga `X2`.

7º En forma similar, si tanto `X1` como `X2` pueden producir ε, consideramos `X3`, luego `X4`, etc.

Ejemplo 10
Calculemos el conjunto Primero para cada noterminal de la GIC con producciones:

```
        S -> aSe | B
        B -> bBe | C
        C -> cCe | d
```

```
Primero(S) = {a,b,c,d}
Primero(B) = {b,c,d}
Primero(C) = {c,d}
```

* Ejercicio 25 *
Explique cómo se obtuvieron los conjuntos del ejemplo anterior.

* Ejercicio 26 *
Obtenga los conjuntos Primero para los noterminales de la GIC con producciones:

```
        S -> ABc
        A -> a | ε
        B -> b | ε
```

4.3.3.3.2 OBTENCIÓN DEL CONJUNTO *SIGUIENTE*

Supongamos que tratamos de obtener los símbolos de preanálisis que sugieren que se debe aplicar la producción `A->X1...Xm`. ¿Qué ocurre si cada `Xi` puede producir ε?

En ese caso, el símbolo de preanálisis para `A` está determinado por aquellos terminales que siguen inmediatamente al noterminal `A` en las cadenas de derivación que contengan al noterminal `A`. Este

conjunto de terminales se denomina Siguiente(A) y es útil porque define el contexto derecho para el noterminal A.

Para obtener el conjunto Siguiente(A), inspeccionamos la GIC buscando todas las ocurrencias de A. En cada producción se pueden dar estas situaciones:

– El noterminal A puede estar seguido por el terminal x (...Ax...), en cuyo caso x pertenece a Siguiente(A).

– El noterminal A puede estar seguido por el noterminal B (...AB...), en cuyo caso Siguiente(A) incluye a Primero(B).

– El noterminal A puede ser el último símbolo del lado derecho en cierta producción de un noterminal T (T -> Y1...YmA), en cuyo caso Siguiente(A) incluye a Siguiente(T).

Ejemplo 11
Calculemos el conjunto Siguiente para cada noterminal de la GIC con producciones:

```
        S -> aSe | B
        B -> bBe | C
        C -> cCe | d
Siguiente(S) = {e}
Siguiente(B) = {e}
Siguiente(C) = {e}
```

Ejemplo 12
Supongamos la siguiente GIC recursiva a izquierda para expresiones aritméticas:

```
        E -> E + T | E -> E - T | E -> T
        T -> T * F | T -> T / F | T -> F
        F -> ( E ) | F -> num
```

Con las operaciones vistas anteriormente, la convertimos en la LL(1):

```
        E1 -> T1 E2
        E2 -> + T1 E2 | - T1 E2 | ε
        T1 -> F1 T2
        T2 -> * F1 T2 | / F1 T2 | ε
        F1 -> ( E1 )  | num
```

Calculemos el conjunto Primero para el noterminal E1 de esta GIC:

```
      Primero(E1) = Primero(T1 E2)
         Primero(T1 E2) = Primero(T1)
         Primero(T1) = Primero(F1 T2)
         Primero(F1 T2) = Primero(F1)
         Primero(F1) = {(,num} => Primero(E1) = {(,num}
```

* Ejercicio 27 *
a) Verifique que lo calculado en el ejemplo anterior es correcto.
b) ¿Por qué una expresión, según la GIC del ejemplo anterior, no puede comenzar con un operador?
c) ¿Y por qué no puede comenzar con un identificador?

4.3.3.4 LA FUNCIÓN PREDICE Y EL ANÁLISIS SINTÁCTICO PREDICTIVO

En el capítulo 3, hemos visto cómo se construyen algunos PAS (Procedimientos de Análisis Sintáctico) utilizando el ASDR, que también se denomina Análisis Sintáctico Predictivo. El principal problema para construir los PAS es decidir qué producción utilizar para buscar la concordancia con el símbolo de preanálisis.

Esta decisión se puede formalizar mediante la definición de una función de Predicción que examine el símbolo de preanálisis para deducir qué producción debe ser utilizada para *expandir* cada noterminal (expandir, en este caso, significa elegir el lado derecho que corresponde al noterminal).

Sea una producción `A->X1...Xm`. Se necesita calcular el conjunto de tokens de preanálisis que pueden indicar que ésta es la producción a ser elegida para la concordancia buscada. Este conjunto estará formado por aquellos terminales que pueden ser producidos por `X1...Xm`.

Se define el conjunto de tokens de preanálisis que causarán la selección, es decir, la predicción de la producción `A->X1...Xm`. Llamaremos a este conjunto Predice y lo definimos así:

```
Predice(A->X1...Xm) =
si ε en Primero(X1...Xm) entonces es
(Primero(X1...Xm) – {ε}) U Siguiente(A)
caso contrario es Primero(X1...Xm)
```

* Ejercicio 28 *
Obtenga la función Predice de la producción `E1 -> T1 E2` del Ejemplo 12.

4.3.4 USANDO UNA PILA PARA IMPLEMENTAR UN PARSER PREDICTIVO

Un Análisis Sintáctico Predictivo, como el realizado en el capítulo 3, puede ser implementado usando una GIC, una pila y el siguiente algoritmo, en el que la pila es inicializada con el axioma de la GIC.

(1) Si la pila está vacía y el flujo de tokens de entrada ha llegado al fdt (*fin de tokens*) el Análisis Sintáctico se ha completado.

(2) Si el símbolo que está en el tope de la pila es un noterminal, reemplácelo por su lado derecho pero invirtiendo el orden de los símbolos. Por ejemplo, si la producción es `A->BCD`, y `A` es el noterminal que está en el tope de la pila, aplique la operación *pop* para eliminar `A` y luego aplique la operación *push* a los símbolos que forman el lado derecho de la producción pero en el orden inverso: primero el símbolo D, luego C y finalmente el símbolo `B`, que queda en el tope de la pila. Si se trata de una producción-ε, solo se elimina (*pop*) el símbolo que está en tope de la pila, sin reemplazarlo por nada. Volver a (1).

(3) Si el símbolo en el tope de la pila es un terminal, debe coincidir con el símbolo de preanálisis (token): a) Si no lo es, ha ocurrido un Error Sintáctico; b) si lo es, elimine (*pop*) ese terminal y avance al próximo token del flujo de entrada. Volver a (1).

Un Parser que puede implementarse tan sencillamente se lo conoce como Parser Predictivo, porque puede predecir cuál será el próximo token y, si la gramática es adecuada, puede expandir inmediatamente el noterminal que está en tope de la pila y así seguir avanzando con este proceso. Para poder construir un Parser Predictivo, la GIC debe ser LL(1).

Ejemplo 13
Supongamos una GIC en la que el noterminal `E` tiene las siguientes producciones:

```
1 E: aEb |
2    cd
```

Si `E` está en el tope de la pila, el Parser debe determinar cuál de las dos producciones debe aplicarse en el proceso de derivación. Esta decisión es tomada en función del símbolo de preanálisis: si éste es **`a`** lo reemplazará en la pila por `bEa`, si es **`c`** aplicará la producción 2, y si es otro símbolo habrá un error sintáctico.

Ejemplo 14
En el Ejemplo anterior nos referimos a un noterminal `E` con producciones:

```
1 E: aEb |
2    cd
```

Supongamos que le agregamos la siguiente producción:

```
3 E: cfE
```

Obsérvese que ahora las producciones 2 y 3 tendrían concordancia con el mismo símbolo de preanálisis ya que ambas comienzan con el mismo terminal. Si queremos analizar sintácticamente la cadena cfacdb, no lo podemos hacer con un método basado en un solo símbolo de preanálisis. Esto significa que la GIC en cuestión ya no es LL(1).

* Ejercicio 29 *
¿Cuántos símbolos de preanálisis hacen falta para resolver el problema planteado en el ejemplo anterior?

Ejemplo 15
Sea una GIC LL(1) que define una lista de expresiones aritméticas (con suma, producto y paréntesis), cada una de las cuales termina con un "punto y coma". La lista completa termina con un fdt (*fin de tokens*) y puede ser vacía. Las producciones de esta GIC son:

```
 1 LE: fdt |
 2     E ; LE
 3 E: T E'
 4 E': + T E' |
 5     ε
 6 T: F T'
 7 T': * F T'|
 8     ε
 9 F: n |
10    ( E )
```

Veamos como un Parser Predictivo procesa la expresión **`1+2;`**

	PILA	ENTRADA	COMENTARIO
1	LE	1+2; fdt	Aplicar producción 2
2	-	1+2; fdt	Pop el noterminal *LE*
3	LE;E	1+2; fdt	Aplicar producción 3
4	LE;	1+2; fdt	Pop el noterminal *E*
5	LE;E'T	1+2; fdt	Aplicar producción 6
6	LE;E'	1+2; fdt	Pop el noterminal *T*
7	LE;E'T'F	1+2; fdt	Aplicar producción 9
8	LE;E'T'	1+2; fdt	Pop el noterminal *F*
9	LE;E'T'**n**	1+2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
10	LE;E'T'	+2; fdt	Aplicar producción 8
11	LE;E'	+2; fdt	Pop el noterminal *T'*
12	LE;E'	+2; fdt	Aplicar producción 4
13	LE;	+2; fdt	Pop el noterminal *E'*
14	LE;E'T**+**	+2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
15	LE;E'T	2; fdt	Aplicar producción 6
16	LE;E'	2; fdt	Pop el noterminal *T*
17	LE;E'T'F	2; fdt	Aplicar producción 9
18	LE;E'T'	2; fdt	Pop el noterminal *F*
19	LE;E'T'**n**	2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
20	LE;E'T'	; fdt	Aplicar producción 8
21	LE;E'	; fdt	Pop el noterminal *T'*
22	LE;E'	; fdt	Aplicar producción 5
23	LE;	; fdt	Pop el noterminal *E'*
24	LE;	; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
25	LE	fdt	Aplicar producción 1
26	fdt	fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
27	-	fdt	Fin del Análisis Sintáctico

Ejemplo 16

Veamos, ahora, qué sucede cuando existe un Error Sintáctico. Supongamos que el flujo de entrada es **1+2*;** que, como se observa, es una lista de expresiones incorrecta. El proceso de Análisis Sintáctico realizado en el ejemplo anterior nos resultará útil porque los pasos 1 a 19 serán idénticos, ya que sirven para procesar la entrada **1+2**. En consecuencia, esos pasos los respetaremos y veremos como continúa:

	PILA	ENTRADA	COMENTARIO
1	LE	1+2*; fdt	Aplicar producción 2
...	. . .	. . .	. . .
19	LE;E'T'**n**	2*; fdt	El terminal en el tope de la pila coincide con el símbolo de preanálisis; eliminarlo y avanzar
20	LE;E'T'	*; fdt	Aplicar producción 7
21	LE;E'	*; fdt	Pop el noterminal *T'*
22	LE;E'T'F*	*; fdt	El terminal en el tope de la pila coincide con el símbolo de preanálisis; eliminarlo y avanzar
23	LE;E'T'F	; fdt	El noterminal *F* tiene dos producciones, ninguna de las cuales comienza con un terminal que coincida con el símbolo de preanálisis => **Error Sintáctico**.

* Ejercicio 30 *
En base a la GIC definida en el Ejemplo 15 y a las tablas construidas, analice sintácticamente el flujo de lexemas **`22*35;16;`** convertidos a tokens.

4.3.5 OTRA VISIÓN: EL LENGUAJE DE LOS PARÉNTESIS ANIDADOS, UN AFP Y EL PARSER DIRIGIDO POR UNA TABLA

Las expresiones con operadores infijos, como tienen la mayoría de los Lenguajes de Programación, tienen cierta complejidad para ser descriptas sintácticamente mediante una GIC. Esto se debe a que la GIC también debe representar la precedencia y asociatividad de cada operador y, además, debe mostrar el correcto uso de los paréntesis.

Ejemplo 17
`(4*(2+((3))))` es una expresión sintácticamente correcta.

Consideremos, ahora, un lenguaje formado únicamente por paréntesis que pueden estar anidados, como se ve en el ejemplo anterior. Estos paréntesis anidados constituyen un LIC que puede ser reconocido por un AFP.

Un Parser Dirigido por una Tabla (*table-driven parser*) está basado en un AFP. Este Parser es dirigido por una máquina de estados (como los estados y transiciones de un Autómata Finito) y utiliza una pila para mantener un registro del progreso de esta máquina de estados.

Supongamos, entonces, un LIC cuyas palabras son paréntesis anidados, como `(()(()))`. Una GIC que genera este LIC puede tener las siguientes producciones:

```
listaParéntesis: plista | plista listaParéntesis
plista: ( listaParéntesis ) | ε
```

* Ejercicio 31 *
a) Escriba una lista de noterminales y otra lista de terminales de esta GIC.
b) Derive por izquierda la palabra de mínima longitud.
c) Derive por izquierda la palabra `(()(()))`.

Construiremos un AFP que reconozca el LIC de los paréntesis anidados. Supongamos las conocidas operaciones para pilas *push* y *pop*; supongamos, también, que existe el símbolo fdt para indicar el fin de tokens a analizar. Estas son las características del AFP, algunas de las cuales lo diferencian del AFP tradicional visto en el capítulo 2 de este volumen:

1. La pila está integrada al autómata;
2. En la pila se guardan números enteros que representan a los nombres de los estados;
3. En el tope de la pila estará el número del estado actual;
4. La tabla de estados tiene tantas filas como estados halla y tantas columnas como caracteres o símbolos tenga el alfabeto. En este caso, los símbolos son solo), (y fdt;
5. El contenido de la Tabla de Estados no mostrará “el próximo estado” sino la acción que se debe llevar a cabo para cada par (estado en el tope de la pila, símbolo de entrada);

6. Hay cuatro tipos de acciones:
   a) aceptada: la secuencia de caracteres analizada es una palabra del lenguaje;
   b) error: hay un error sintáctico en la cadena analizada;
   c) push N: para insertar el estado N en la pila. N es el nuevo estado actual;
   d) pop: para quitar el estado en el tope de la pila; habrá un nuevo estado actual.
7. Con cada operación *push* o *pop* se avanza un carácter en la sentencia de entrada.

Esta es la Tabla de Estados del AFP que reconoce el lenguaje de los paréntesis anidados:

Estado	(	)	fdt
0	push 1	error	aceptado
1	push 1	pop	error

Usemos este AFP para realizar el Análisis Sintáctico de `((()()))`. Nótese cómo la pila es utilizada para contar:

Tabla de Análisis Sintáctico 1

PILA (tope a la derecha)	ENTRADA A ANALIZAR	PRÓXIMA ACCIÓN
0	((()())) fdt	Push 1 y avanza
01	(()())) fdt	Push 1 y avanza
011	)()())) fdt	Pop y avanza
01	()())) fdt	Push 1 y avanza
011	())) fdt	Push 1 y avanza
0111	))) fdt	Pop y avanza
011	)) fdt	Pop y avanza
01	) fdt	Pop y avanza
0	fdt	Acepta

Analicemos otra cadena: `()(())(()`:

Tabla de Análisis Sintáctico 2

PILA (tope a la derecha)	ENTRADA A ANALIZAR	PRÓXIMA ACCIÓN
0	()(())(() fdt	Push 1 y avanza
01	)(())(() fdt	Pop y avanza
0	(())(() fdt	Push 1 y avanza
01	())(() fdt	Push 1 y avanza
011	))(() fdt	Pop y avanza
01	)(() fdt	Pop y avanza
0	(() fdt	Push 1 y avanza
01	() fdt	Push 1 y avanza
011	) fdt	Pop y avanza
01	fdt	Error

* Ejercicio 32 *
a) Construya la Tabla de Análisis Sintáctico para la cadena `()()`.
b) Construya la Tabla de Análisis Sintáctico para la cadena `(()()`.

4.3.6 EL PROBLEMA DEL IF-THEN-ELSE: EL "ELSE COLGANTE"

Casi todas las construcciones de un LP pueden especificarse con una GIC LL(1). Sin embargo, hay una importante excepción: la construcción if-then-else, que nació con el Algol 60 y se propagó a los lenguajes derivados de él como Pascal, ANSI C, y luego C++, Java y otros.

El problema es que como la cláusula else es opcional (vea la descripción gramatical en la sección 4.3.3.1), puede haber más partes then que partes else y, por lo tanto, la correspondencia entre ambas partes no es única. Por ello, la sentencia if-then-else tiene el mismo comportamiento que el lenguaje $L = \{a^n b^m / n \geq m, n > 0, m \geq 0\}$, donde a representa la parte then y b representa la parte else.

En muchos libros sobre diseño de compiladores se puede ver cómo analizan detalladamente este problema. Nosotros solo vamos a dedicarnos a buscar una solución. Una técnica utilizada para manejar el problema del "else colgante" con un Parser LL(1) consiste en emplear una gramática ambigua junto a cierta regla para unificar las predicciones.

Consideremos la GIC con producciones:

```
S -> if S E | otro
E -> else S | ε
```

Resolvemos la ambigüedad de esta GIC con una regla auxiliar: todo else se asocia al if más cercano. Esto es: al predecir un noterminal E, si else es el símbolo de preanálisis entonces concuerda inmediatamente.

Como cierre de este tema, es interesante considerar que, como dice Fischer (1991), el "else colgante" no es, en realidad, un problema gramatical ni un problema de Análisis Sintáctico; es un problema de diseño de un LP. Si toda sentencia if terminase con un end if o con algún símbolo equivalente, el problema desaparece. En ese caso, podríamos utilizar una GIC con las siguientes producciones:

```
S -> if S E | otro
E -> else S end if | end if
```

Esta GIC sí es LL(1). Y vemos como el diseño de un LP influye en el proceso de compilación.

* Ejercicio 33 *
Demuestre que la GIC para if-else-end if es LL(1).

4.3.7 ANÁLISIS SINTÁCTICO ASCENDENTE (*BOTTOM-UP*)

En este texto nos interesa el Análisis Sintáctico Ascendente realizado a partir de una GIC LR(1), donde L significa que el flujo de tokens es leído de izquierda a derecha y R significa que se utiliza una derivación a derecha (*right*). Toda gramática LL(1) es también LR(1), pero la inversa no es cierta.

Para construir un Parser a partir de una gramática LR se requiere la existencia de una herramienta tipo *yacc* que ayude a construir las tablas utilizadas por el Parser.

4.3.7.1 CÓMO FUNCIONA EL ANÁLISIS SINTÁCTICO ASCENDENTE (ASA)

El ASA es realizado por un Parser Ascendente, que es, básicamente, un AFP: requiere un conjunto de estados para guiarlo y una pila para recordar el estado actual. El Parser construye el árbol sintáctico en forma ascendente (desde las hojas hacia la raíz). Por ahora, atendamos solo la pila.

El ASA trabaja así:

1) Si los elementos que están en la parte superior de la pila – el tope y los que siguen – forman el lado derecho de una producción, se quitan (*pop*) estos elementos y se reemplazan por el noterminal del lado izquierdo (*push*). Esto se conoce como la operación de REDUCCIÓN. Si se trata de la producción nula, no hay *pop*. Entonces, REDUCCIÓN es el proceso de reemplazar el lado derecho de una producción en la pila con el correspondiente noterminal del lado izquierdo.

2) Caso contrario, *push* el símbolo de entrada actual (un token) en la pila y avance en el flujo de tokens de entrada (lo llamaremos input). Esta operación se conoce como DESPLAZAMIENTO porque desplaza el siguiente token desde el flujo de entrada a la pila. En definitiva, un DESPLAZAMIENTO es el proceso de mover un token desde el input a la pila – *push* el token actual y avanza el *input* hasta el próximo token.

3) Si la operación previa consistió en una REDUCCIÓN que resultó con el noterminal objetivo – el axioma – en el tope de la pila, el *input* es ACEPTADO y el Análisis Sintáctico Ascendente se ha completado. De lo contrario, volver al paso (1).

Ejemplo 18
Sea la siguiente GIC para generar expresiones simples:

```
0 S -> E              S es el noterminal llamado objetivo
1 E -> E + T
2 E -> T
3 T -> T * F
4 T -> F
5 F -> ( E )
6 F -> num
```

Una GIC, para poder realizar el ASA, debe tener un noterminal "objetivo" con una sola producción. Por eso, E no podría ser el noterminal "objetivo" de esta GIC y, en consecuencia, necesitamos agregar un nuevo noterminal que actúe como axioma. Note que esta gramática no podría ser utilizada para un Análisis Sintáctico Descendente porque es recursiva a izquierda, pero esta propiedad no impide realizar un ASA.

Ilustremos el proceso de ASA con la expresión 1*(2+3) como *input*.

PILA (tope a la derecha)	INPUT (flujo de tokens)	PRÓXIMO PASO
vacía	1 * (2 + 3)	Desplazar
num	* (2 + 3)	Reducir con producción 6
F	* (2 + 3)	Reducir con producción 4
T	* (2 + 3)	No se puede reducir con producción 2 porque después no existe E***** => Desplazar
T*****	(2 + 3)	Desplazar
T***(**	2 + 3)	Desplazar
T***(num**	+ 3)	Reducir con producción 6
T***(**F	+ 3)	Reducir con producción 4
T***(**T	+ 3)	Reducir con producción 2
T***(**E	+ 3)	Desplazar
T***(**E**+**	3)	Desplazar
T***(**E**+num**	)	Reducir con producción 6
T***(**E**+**F	)	Reducir con producción 4
T***(**E**+**T	)	Reducir con producción 1
T*(E	)	Desplazar
T***(**E**)**		Reducir con producción 5

T*****F		Reducir con producción 3
T		Reducir con producción 2
E		Reducir con producción 0 porque no hay input
S		FIN CORRECTO

Ejemplo 19
Ahora procesaremos un input incorrecto: 1*(2+3)). Como se nota, el input es igual al anterior pero con el agregado de otro paréntesis que cierra al final => es sintácticamente incorrecto. El proceso y la tabla diseñada serán iguales hasta la penúltima entrada. Pero comencemos, para mayor claridad, en la antepenúltima entrada:

PILA (tope a la derecha)	INPUT (flujo de tokens)	PRÓXIMO PASO
T	)	Reducir con producción 2
E	)	No se puede reducir E al objetivo porque todavía hay input => Desplazar
E**)**		La GIC no tiene producción con E**)** en su lado derecho y no hay más tokens => no se puede reducir ni desplazar => FIN INCORRECTO

* Ejercicio 34 *
a) Derive verticalmente a derecha la expresión 1*(2+3).
b) A partir de esta derivación, construya el proceso inverso (reducción) y compárelo con la tabla construida en el Ejemplo 18.

4.3.7.2 RECURSIVIDAD EN EL ANÁLISIS SINTÁCTICO ASCENDENTE

Hemos visto que, para realizar el Análisis Sintáctico Descendente, la gramática no puede ser recursiva a izquierda. Esta situación se invierte al realizar un Análisis Sintáctico Ascendente: la recursividad a izquierda siempre debe utilizarse para aquellas listas en las que la asociatividad no es un tema importante o debe ser a izquierda. Por otro lado, la recursividad a derecha debe ser utilizada solo cuando se requiere la asociatividad a derecha.

4.3.7.3 OTRA VISIÓN: LA IMPLEMENTACIÓN DE UN PARSER ASCENDENTE COMO UN AFP

En la implementación de un Parser Ascendente como un AFP es fundamental que se encuentre el lado derecho de una producción en la pila para que se pueda aplicar la operación de reducción. El parser puede diferir la reducción, aunque el lado derecho se encuentre en la parte superior de la pila, si las reglas de asociatividad y de precedencia construidas dentro de la gramática requieren más tokens para que esta reducción sea posible. Sin embargo, en la mayoría de las situaciones una reducción ocurre ni bien el lado derecho de una producción aparece en la pila.

Para automatizar un Parser Ascendente utilizamos un AFP. Los estados finales (o de aceptación) del AFP "gatillan" la operación de reducción.

Ejemplo 20
Supongamos la siguiente GIC de expresiones:

```
0 S -> E
1 E -> E + T
2 E -> T
3 T -> num
```

En la pila se almacenan los números de estados del autómata; el número de estado que está en el tope indica el estado actual del autómata. La dirección de una transición del autómata representa *push*; esto es: el número del estado de llegada es almacenado en la pila, por lo que la pila sirve para registrar las diferentes transiciones que se llevaron a cabo para llegar al estado actual. Al mismo tiempo, con sucesivas operaciones *pop* podemos recorrer esas transiciones "hacia atrás", como veremos a continuación.

Para la GIC presentada, estos serían los estados, las transiciones y lo que ocurre con la pila del AFP:

```
Estado inicial: 0
Estados finales: 1, 2, 3 y 5
```

– Al comenzar, el AFP se halla en el estado 0 y, en consecuencia, 0 estará en el tope de la pila.

– El desplazamiento de **num** se indica mediante la transición `0->num->1+`; esto es: si lee un token **num**, coloca un 1 en la pila (*push*) para representar esta transición.

– La reducción de **num** a T (ver producción 3) se representa mediante una operación *pop* en la pila para quitar el 1 y volver al estado 0 más un *push* del estado 3 para indicar la transición `0->T->3+`.

– Importante: (1) La única forma de "colocar" un terminal en la pila (a través del número de estado que corresponda) es mediante la operación *push* que ocurre en el desplazamiento. (2) La única forma de "colocar" un noterminal en la pila es mediante la operación *push* del estado que corresponda, durante una reducción.

Para esta GIC, los estados y transiciones del AFP serán:

```
0 -> num -> 1+
0 -> E -> 2+
0 -> T -> 3+
2 -> + -> 4
4 -> num -> 1+
4 -> T -> 5+
```

* Ejercicio 35 *
Sea fdt el indicador del final del flujo de tokens. Realice el ASA de 11+22+33 fdt, teniendo en cuenta la información dada en esta sección.
Construya una tabla con 3 columnas: PILA – INPUT – PRÓXIMO PASO.

4.3.7.4 EL USO DE yacc

Finalizamos la sección de Análisis Sintáctico con un programa que construye un Parser Ascendente utilizando la herramienta de programación llamada *yacc*. Agradezco al Prof. José María Sola, quien me proporcionó este ejemplo.

Ejemplo 21

```
%{
/* Ejemplo de Parser generado automáticamente.
 * JMS 20090722
 * El LIC analizado por el Parser es generado por la GIC:
 *    S -> aTc
 *    T -> aTc | b
 */
#include <stdio.h>
int yylex(void);
void yyerror(const char *);
%}
%%
S
     : 'a' T 'c'
     ;
T
     : 'a' T 'c'
     | 'b'
     ;
%%
int main(void){
     switch( yyparse() ){
          case 0:
            puts("Pertenece al LIC"); return 0;
          case 1:
            puts("No pertenece al LIC"); return 1;
          case 2:
            puts("Memoria insuficiente"); return 2;
     }
}
/* yylex es el Scanner. Retorna el siguiente símbolo (token).
 * Si no hay más símbolos, retorna 0.
 */
int yylex(void){
     int c = getchar();
     if(c == EOF)
          return 0;
     return c;
}
/* Informa la ocurrencia de un error. */
void yyerror(const char *s){
   puts(s);
   return;
}
```

* Ejercicio 36 *

Siguiendo el modelo que brinda el programa del ejemplo anterior, utilice *yacc* para construir un Parser Ascendente que analice, sintácticamente, el LIC generado por estas producciones:

```
S -> aaRbb | ab
R -> c
```

4.4 ANÁLISIS SEMÁNTICO

El Análisis Semántico en un compilador no se refiere a la interpretación semántica – como sinónimo de *significado* – de una declaración, de una expresión, de una sentencia o de una función de algún Lenguaje de Programación.

Desde el punto de vista de etapa de Análisis del compilador, el Análisis Semántico consiste, básicamente, en la realización de tareas de verificación y de conversión que no puede realizar el Análisis Sintáctico.

Los Lenguajes de Programación a los que nos referimos en este libro tienen una sintaxis representada mediante gramáticas INDEPENDIENTE DEL CONTEXTO. Justamente, el nombre de la gramática lo clarifica: significa que no tiene en cuenta lo precedente ni lo posterior, no tiene en cuenta el contexto en el que se define la sintaxis de cualquier construcción del LP.

Entonces, el Análisis Semántico debe verificar que se cumplan todas las reglas sensibles al contexto como, por ejemplo, que una variable haya sido declarada antes de su utilización. Para ello, el Análisis Semántico utiliza mucho la Tabla de Símbolos.

Otra manera de describir la frase del párrafo anterior sería: Hay reglas sintácticas que no se pueden expresar por medio de GICs. Por ejemplo, la regla que obliga a que – en LPs como Pascal, ANSI C, C++ y Java – toda variable sea declarada antes de ser utilizada, no puede ser representada por una GIC.

Ejemplo 22
Sea una producción de una GIC que define así la sintaxis de una asignación en algún LP:

```
Asignación: identificadorVariable = constanteEntera;
```

Esta producción no puede saber si la variable ha sido declarada previamente o no. En la práctica, los detalles sintácticos que no pueden ser representados por la GIC son resueltos por las rutinas semánticas.

Otras dificultades semánticas que aparecen en el proceso de compilación y que deben ser resueltas durante el Análisis Semántico son:

a) La GIC no puede informarnos si una variable usada en una expresión es del tipo correcto.

b) La GIC no puede reconocer declaraciones múltiples en un mismo bloque (como int x, double x;).
c) Si una expresión aritmética es sintácticamente correcta pero los operandos son de diferentes tipos numéricos, debe haber un proceso de conversión (si el LP lo permite) para que las instrucciones del lenguaje objeto puedan operar.

* Ejercicio 37 *
Sea la expresión 24 + 4.16
a) ¿Qué ocurre durante el Análisis Léxico?
b) ¿Qué ocurre durante el Análisis Sintáctico?
c) ¿Qué ocurre durante el Análisis Semántico?

Una GIC define un *lenguaje* compuesto por un conjunto de secuencias de tokens. En general, toda secuencia de tokens derivable desde una GIC se considera sintácticamente válida para el compilador, aunque no lo sea para el programador. Cuando la Semántica Estática es chequeada

por las rutinas semánticas en un programa sintácticamente válido, se pueden descubrir errores semánticos. Por ejemplo, en Pascal, la sentencia A := 'X' + true; no tiene errores sintácticos pero sí tiene un error semántico porque el operador + no está definido para sumar un carácter a un valor booleano.

4.4.1 EN ANSI C: DERIVABLE VS SINTÁCTICAMENTE CORRECTO

En el Volumen 1, página 64, se introducen estos dos conceptos: a) construcción DERIVABLE de una GIC; b) construcción SINTÁCTICAMENTE CORRECTA. Si bien este tema es aplicable a muchos LPs, nos ocuparemos específicamente del ANSI C.

En muchos casos, una construcción cumple con ambos conceptos, pero en otras ocasiones no. Si es DERIVABLE, significa que la GIC lo genera; pero si es DERIVABLE y no es SINTÁCTICAMENTE CORRECTA, significa que la GIC lo genera pero en la fase de Análisis Semántico se detecta un error insalvable.

Ejemplo 23
Para ejemplificar esta última situación, partamos de un subconjunto muy simplificado de la GIC que genera expresiones en ANSI C:

1 *expAsignación: expUnaria* = *expCondicional*
2 *expCondicional: expUnaria*
3 *expUnaria: expPrimaria*
4 *expPrimaria: identificador* | *constante*

Según esta GIC, se puede DERIVAR la expresión 2 = 4, aunque sintácticamente no sea correcta. Obviamente, no le podemos asignar un valor a una constante EN NINGÚN LENGUAJE DE PROGRAMACIÓN CONOCIDO. Esta inconsistencia sintáctica se determina, entonces, durante el Análisis Semántico.

* Ejercicio 38 *
Sea el siguiente bloque en ANSI C: `{ int a; 2 = a; double b; b = 4++; }`

a) Describa el resultado del Análisis Léxico;
b) Describa el resultado del Análisis Sintáctico;
c) Describa el resultado del Análisis Semántico.

* Ejercicio 39 *
Sea el siguiente bloque en ANSI C: `{ int a,b; double c; a = b+c; }`

a) Describa el resultado del Análisis Léxico;
b) Describa el resultado del Análisis Sintáctico;
c) Describa el resultado del Análisis Semántico.

5 BIBLIOGRAFÍA

Alfonseca Cubero, Enrique y otros (2007): "*Teoría de Autómatas y Lenguajes Formales*; McGraw-Hill/Interamericana, España.

Aho, Alfred V. y otros (1990): "*Compiladores – Principios, Técnicas y Herramientas*"; Addison-Wesley Iberoamericana, EE. UU.

ANSI (1990): "*American National Standard for Information Systems – Programming Language C*"; American National Standards Institute, Nueva York, EE. UU.

Cohen, Daniel (1986): "*Introduction to Computer Theory*"; John Wiley & Sons, EE. UU.

Fischer, Charles N. y otro (1991): "*Crafting a Compiler with C*"; Benjamin/Cummings, EE. UU.

Hollub, Allen I. (1990):"*Compiler Design in C*"; Prentice-Hall, EE. UU.

Hopcroft, John E. y otro (1979): "*Introduction to Automata Theory, Languages, and Computation*"; Addison-Wesley, 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.

6 EJERCICIOS RESUELTOS

CAPÍTULO 1

* Ejercicio 1 *
(a) SI - (b) NO

* Ejercicio 2 *
(a) SI - (b) SI

* Ejercicio 3 *
palabra ab: 0- => a => 1 => b => 2+ RECONOCE
cadena abab: 0- => a => 1 => b => 2 => a => ?? RECHAZA (no puede continuar)
cadena vacía: 0- => ε => ?? RECHAZA (el estado inicial no es final)

* Ejercicio 5 *

T	a	b
0-	0	1
1	2	1
2+	-	-

* Ejercicio 7 *

T	a	b	c	d
0-+	0	0	0	0

* Ejercicio 12 *
DF = ({0, 1, 2, 3}, {a, b,c}, TT, 0, {3}) donde la TT es:

TT	a	b	c
0-	1	1	1
1	2	2	2
2	3	3	3
3+	3	3	3

CAPÍTULO 2

* Ejercicio 4 *
El ε significa que el AFP, estando en el estado 4 y no leyendo ningún carácter, hace un *pop* del símbolo Z y pasa al estado 5 haciendo un *push*, primero de P y luego de R, quedando este último símbolo como nuevo tope de la pila.

* Ejercicio 6 *
SÍ.

* Ejercicio 8 *
Elimina el símbolo que está en el tope de la pila.

* Ejercicio 18 *

T	a	b	fdc
e0,$	e1,R$	e1,Z$	
e1,R	e1,RR	e1,ε	
e1,Z	e1,ε	e1,ZZ	
e1,$	e1,R$	e1,Z$	e2,$

CAPÍTULO 3

* Ejercicio 4 *

Lexema	Token
int	Palabra reservada
WHILE	Identificador
;	Carácter de puntuación
while	Palabra reservada
(	Carácter de puntuación
WHILE	Identificador
>	Operador
(	Carácter de puntuación
32	Constante
)	Carácter de puntuación
)	Carácter de puntuación
-	Operador
2.46	Constante
;	Carácter de puntuación

* Ejercicio 29 *

```
void Programa (void) {
/* <programa> -> INICIO <listaSentencias> FIN */
   Match(INICIO); /* recuerde que Match invoca al Scanner */
   ListaSentencias();
   Match(FIN);
}
```

CAPÍTULO 4

* Ejercicio 26 *

```
Primero(S) = {a,b,c}
Primero(A) = {a,ε}
Primero(B) = {b,ε}
```